# SAT/SMT by example

Dennis Yurichev <dennis@yurichev.com>

July 15, 2018

# Contents



[1]Satisfiability modulo theories
[2]Boolean satisfiability problem
[3]Linear congruential generator

[4] Rivest–Shamir–Adleman cryptosystem
[5] If-Then-Else

[6] Also Known As
[7] Printed circuit board

[8] Cyclic redundancy check
[9] Greatest Common Divisor
[10] Least Common Multiple

# 1 Introduction

## 1.1 Disclaimer

This collection is a non-academic reading for "end-users", i.e., programmers, etc.

The author of these lines is no expert in SAT/SMT, by any means. This is not a book, rather a student's notes. Take it with grain of salt...

## 1.2 Latest versions

Latest version is always available at http://yurichev.com/writings/SAT_SMT_by_example.pdf.

Russian version has been dropped – it's too hard for me to maintain two versions. Sorry.

New parts are appearing here from time to time, see: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/ChangeLog.

## 1.3 Illustrator wanted

... who can draw in the following manner: 1, 2, 3, 4.

... for this book, which is open-source and free, and unlikely to be published. However, someone maybe interested, in a self-advertisement sense...

Please contact me: dennis@yurichev.com.

## 1.4 The source code

Some people find it inconvenient to copy&paste source code from this PDF. Everything is available on GitHub: https://github.com/DennisYurichev/SAT_SMT_by_example.

## 1.5 Thanks

Armin Biere[11] has patiently answered to my endless boring questions.

Leonardo Mendonça de Moura[12], Nikolaj Bjørner[13] and Mate Soos[14] have also helped.

Masahiro Sakai[15] has helped with numberlink puzzle: 7.11.

Alex "clayrat" Gryzlov and @mztropics on twitter found couple of bugs.

Xenia Galinskaya – for carefully measured periods of forced distraction from the work.

## 1.6 Praise

"This is quite instructive for students. I will point my students to this!" (Armin Biere).

"An excellent source of well-worked through and motivating examples of using Z3's python interface." [16] (Nikolaj Bjorner, one of Z3's authors).

"Impressive collection of fun examples!" (Pascal Fontaine[17], one of veriT solver's authors.)

## 1.7 Introduction

SAT/SMT solvers can be viewed as solvers of huge systems of equations. The difference is that SMT solvers takes systems in arbitrary format, while SAT solvers are limited to boolean equations in CNF[18] form.

A lot of real world problems can be represented as problems of solving system of equations.

## 1.8 Is it a hype? Yet another fad?

Some people say, this is just another hype. No, SAT is old enough and fundamental to CS[19]. The reason of increased interest to it is that computers gets faster over the last couple decades, so there are attempts to solve old problems using SAT/SMT, which were inaccessible in past.

[11] http://fmv.jku.at/biere/
[12] https://www.microsoft.com/en-us/research/people/leonardo/
[13] https://www.microsoft.com/en-us/research/people/nbjorner/
[14] https://www.msoos.org/
[15] https://twitter.com/masahiro_sakai
[16] https://github.com/Z3Prover/z3/wiki
[17] https://members.loria.fr/PFontaine/
[18] Conjunctive normal form
[19] Computer science

# 2 Basics

## 2.1 One-hot encoding

Throughout this book, we'll often use so-called "one-hot encoding". In short, this is:

| Decimal | One-hot |
|---|---|
| 0 | 00000001 |
| 1 | 00000010 |
| 2 | 00000100 |
| 3 | 00001000 |
| 4 | 00010000 |
| 5 | 00100000 |
| 6 | 01000000 |
| 7 | 10000000 |

Or in reversed form:

| Decimal | One-hot |
|---|---|
| 0 | 10000000 |
| 1 | 01000000 |
| 2 | 00100000 |
| 3 | 00010000 |
| 4 | 00001000 |
| 5 | 00000100 |
| 6 | 00000010 |
| 7 | 00000001 |

It has several advantages and disadvantages as well. See also: https://en.wikipedia.org/wiki/One-hot.
It's worth noting that one-hot encoding is also called "unitary code" in Russian literature.

## 2.2 SMT-solvers

### 2.2.1 School-level system of equations

This is school-level system of equations copypasted from Wikipedia [20]:

$$\begin{aligned} 3x + 2y - z &= 1 \\ 2x - 2y + 4z &= -2 \\ -x + \tfrac{1}{2}y - z &= 0 \end{aligned}$$

Will it be possible to solve it using Z3? Here it is:

```
#!/usr/bin/python
from z3 import *

x = Real('x')
y = Real('y')
z = Real('z')
s = Solver()
```

[20] https://en.wikipedia.org/wiki/System_of_linear_equations

```
s.add(3*x + 2*y - z == 1)
s.add(2*x - 2*y + 4*z == -2)
s.add(-x + 0.5*y - z == 0)
print s.check()
print s.model()
```

We see this after run:

```
sat
[z = -2, y = -2, x = 1]
```

If we change any equation in some way so it will have no solution, s.check() will return "unsat".

I've used "Real" *sort* (some kind of data type in SMT-solvers) because the last expression equals to $\frac{1}{2}$, which is, of course, a real number. For the integer system of equations, "Int" *sort* would work fine.

Python (and other high-level PL[21]s like C#) interface is highly popular, because it's practical, but in fact, there is a standard language for SMT-solvers called SMT-LIB [22].

Our example rewritten to it looks like this:

```
(declare-const x Real)
(declare-const y Real)
(declare-const z Real)
(assert (=(-(+(* 3 x) (* 2 y)) z) 1))
(assert (=(+(-(* 2 x) (* 2 y)) (* 4 z)) -2))
(assert (=(-(+ (- 0 x) (* 0.5 y)) z) 0))
(check-sat)
(get-model)
```

This language is very close to LISP, but is somewhat hard to read for untrained eyes.

Now we run it:

```
% z3 -smt2 example.smt
sat
(model
  (define-fun z () Real
    (- 2.0))
  (define-fun y () Real
    (- 2.0))
  (define-fun x () Real
    1.0)
)
```

So when you look back to my Python code, you may feel that these 3 expressions could be executed. This is not true: Z3Py API offers overloaded operators, so expressions are constructed and passed into the guts of Z3 without any execution [23]. I would call it "embedded DSL[24]".

Same thing for Z3 C++ API, you may find there "operator+" declarations and many more [25].

Z3 API[26]s for Java, ML and .NET are also exist [27].

Z3Py tutorial: https://github.com/ericpony/z3py-tutorial.

Z3 tutorial which uses SMT-LIB language: http://rise4fun.com/Z3/tutorial/guide.

### 2.2.2 Another school-level system of equations

I've found this somewhere at Facebook:

[21] Programming Language
[22] http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-06-28.pdf
[23] https://github.com/Z3Prover/z3/blob/6e852762baf568af2aad1e35019fdf41189e4e12/src/api/python/z3.py
[24] Domain-specific language
[25] https://github.com/Z3Prover/z3/blob/6e852762baf568af2aad1e35019fdf41189e4e12/src/api/c%2B%2B/z3%2B%2B.h
[26] Application programming interface
[27] https://github.com/Z3Prover/z3/tree/6e852762baf568af2aad1e35019fdf41189e4e12/src/api

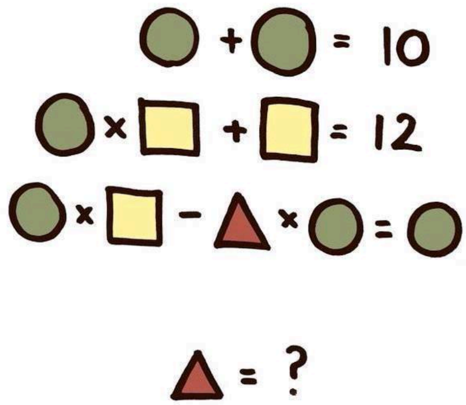



Figure 1: System of equations

It's that easy to solve it in Z3:

```
#!/usr/bin/python
from z3 import *

circle, square, triangle = Ints('circle square triangle')
s = Solver()
s.add(circle+circle==10)
s.add(circle*square+square==12)
s.add(circle*square-triangle*circle==circle)
print s.check()
print s.model()
```

```
sat
[triangle = 1, square = 2, circle = 5]
```

#### 2.2.3 Connection between SAT and SMT solvers

SMT-solvers are frontends to SAT solvers, i.e., they translating input SMT expressions into CNF and feed SAT-solver with it. Translation process is sometimes called "bit blasting". Some SMT-solvers uses external SAT-solver: STP uses MiniSAT or CryptoMiniSAT as backend. Some other SMT-solvers (like Z3) has their own SAT solver.

#### 2.2.4 List of SMT-solvers

- Yices[28], created by Bruno Dutertre et al.
- Z3[29], created by Leonardo de Moura, Nikolaj Bjorner, Christoph M. Wintersteiger.

  Many examples here uses Python 3.x API for Z3. Here is how to install it in Ubuntu:

  ```
          sudo apt-get install python3-pip
          sudo pip3 install z3-solver
  ```

- STP[30], used in KLEE.
- CVC3/CVC4[31].

[28] http://yices.csl.sri.com/
[29] https://github.com/Z3Prover/z3
[30] https://github.com/stp/stp
[31] http://cvc4.stanford.edu/

- Boolector[32], created by Aina Niemetz, Mathias Preiner and Armin Biere.
- Alt-Ergo[33], used in Frama-C.
- MathSAT[34]. Created by Alberto Griggio, Alessandro Cimatti and Roberto Sebastiani.
- veriT[35]. Created by David Déharbe, Pascal Fontaine, Haniel Barbosa. Lacks bitvectors.
- toysolver[36] by Masahiro Sakai, written in Haskell.
- MK85[37]. Created by Dennis Yurichev, as a toy-level bit-blaster.
- dReal: "An SMT Solver for Nonlinear Theories of the Reals" [38].

## 2.3 SAT-solvers

SMT vs. SAT is like high level PL vs. assembly language. The latter can be much more efficient, but it's hard to program in it.

SAT is abbreviation of "Boolean satisfiability problem". The problem is to find such a set of variables, which, if plugged into boolean expression, will result in "true".

### 2.3.1 CNF form

CNF[39] is a *normal form*.

Any boolean expression can be converted to *normal form* and CNF is one of them. The CNF expression is a bunch of clauses (sub-expressions) constisting of terms (variables), ORs and NOTs, all of which are then glueled together with AND into a full expression. There is a way to memorize it: CNF is "AND of ORs" (or "product of sums") and DNF[40] is "OR of ANDs" (or "sum of products").

Example is: $(\neg A \vee B) \wedge (C \vee \neg D)$.

$\vee$ stands for OR (logical disjunction[41]), "+" sign is also sometimes used for OR.

$\wedge$ stands for AND (logical conjunction[42]). It is easy to memorize: $\wedge$ looks like "A" letter. "$\cdot$" is also sometimes used for AND.

$\neg$ is negation (NOT).

### 2.3.2 Example: 2-bit adder

SAT-solver is merely a solver of huge boolean equations in CNF form. It just gives the answer, if there is a set of input values which can satisfy CNF expression, and what input values must be.

Here is a 2-bit adder for example:

The adder in its simplest form: it has no carry-in and carry-out, and it has 3 XOR gates and one AND gate. Let's try to figure out, which sets of input values will force adder to set both two output bits? By doing quick memory calculation, we can see that there are 4 ways to do so: $0+3=3$, $1+2=3$, $2+1=3$, $3+0=3$. Here is also truth table, with these rows highlighted:

[32] http://fmv.jku.at/boolector/
[33] https://alt-ergo.ocamlpro.com/
[34] http://mathsat.fbk.eu/
[35] http://www.verit-solver.org/
[36] https://github.com/msakai/toysolver
[37] https://github.com/DennisYurichev/MK85
[38] http://dreal.cs.cmu.edu, https://github.com/dreal
[39] https://en.wikipedia.org/wiki/Conjunctive_normal_form
[40] Disjunctive normal form
[41] https://en.wikipedia.org/wiki/Logical_disjunction
[42] https://en.wikipedia.org/wiki/Logical_conjunction

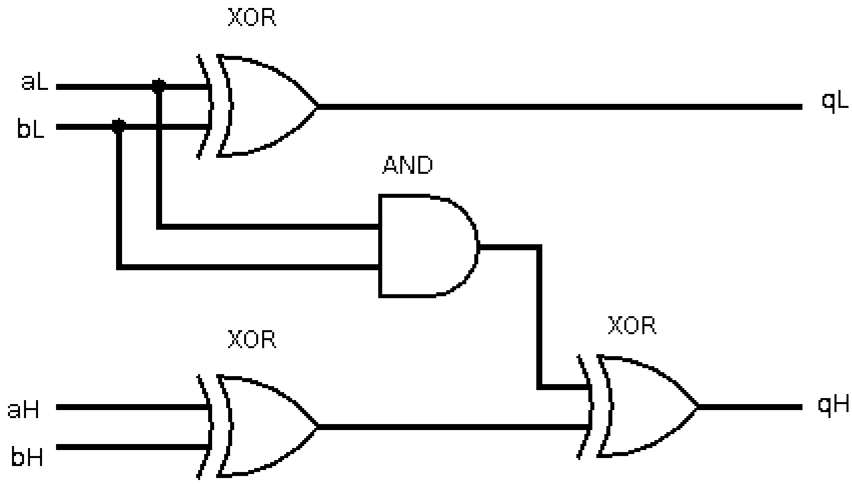



Figure 2: 2-bit adder circuit

| | aH | aL | bH | bL | qH | qL |
|---|---|---|---|---|---|---|
| 3+3 = 6 ≡ 2 (mod 4) | 1 | 1 | 1 | 1 | 1 | 0 |
| 3+2 = 5 ≡ 1 (mod 4) | 1 | 1 | 1 | 0 | 0 | 1 |
| 3+1 = 4 ≡ 0 (mod 4) | 1 | 1 | 0 | 1 | 0 | 0 |
| 3+0 = 3 ≡ 3 (mod 4) | 1 | 1 | 0 | 0 | 1 | 1 |
| 2+3 = 5 ≡ 1 (mod 4) | 1 | 0 | 1 | 1 | 0 | 1 |
| 2+2 = 4 ≡ 0 (mod 4) | 1 | 0 | 1 | 0 | 0 | 0 |
| 2+1 = 3 ≡ 3 (mod 4) | 1 | 0 | 0 | 1 | 1 | 1 |
| 2+0 = 2 ≡ 2 (mod 4) | 1 | 0 | 0 | 0 | 1 | 0 |
| 1+3 = 4 ≡ 0 (mod 4) | 0 | 1 | 1 | 1 | 0 | 0 |
| 1+2 = 3 ≡ 3 (mod 4) | 0 | 1 | 1 | 0 | 1 | 1 |
| 1+1 = 2 ≡ 2 (mod 4) | 0 | 1 | 0 | 1 | 1 | 0 |
| 1+0 = 1 ≡ 1 (mod 4) | 0 | 1 | 0 | 0 | 0 | 1 |
| 0+3 = 3 ≡ 3 (mod 4) | 0 | 0 | 1 | 1 | 1 | 1 |
| 0+2 = 2 ≡ 2 (mod 4) | 0 | 0 | 1 | 0 | 1 | 0 |
| 0+1 = 1 ≡ 1 (mod 4) | 0 | 0 | 0 | 1 | 0 | 1 |
| 0+0 = 0 ≡ 0 (mod 4) | 0 | 0 | 0 | 0 | 0 | 0 |

Let's find, what SAT-solver can say about it?
First, we should represent our 2-bit adder as CNF expression.
Using Wolfram Mathematica, we can express 1-bit expression for both adder outputs:

```
In[]:=AdderQ0[aL_,bL_]=Xor[aL,bL]
Out[]:=aL ⊻ bL
```

```
In[]:=AdderQ1[aL_,aH_,bL_,bH_]=Xor[And[aL,bL],Xor[aH,bH]]
Out[]:=aH ⊻ bH ⊻ (aL && bL)
```

We need such expression, where both parts will generate 1's. Let's use Wolfram Mathematica find all instances of such expression (I glueled both parts with And):

```
In[]:=Boole[SatisfiabilityInstances[And[AdderQ0[aL,bL],AdderQ1[aL,aH,bL,bH]],{aL,aH,bL,bH},4]]
Out[]:={1,1,0,0},{1,0,0,1},{0,1,1,0},{0,0,1,1}
```

Yes, indeed, Mathematica says, there are 4 inputs which will lead to the result we need. So, Mathematica can also be used as SAT solver.

Nevertheless, let's proceed to CNF form. Using Mathematica again, let's convert our expression to CNF form:

```
In[]:=cnf=BooleanConvert[And[AdderQ0[aL,bL],AdderQ1[aL,aH,bL,bH]],``CNF'']
Out[]:=(!aH ∥ !bH) && (aH ∥ bH) && (!aL ∥ !bL) && (aL ∥ bL)
```

Looks more complex. The reason of such verbosity is that CNF form doesn't allow XOR operations.

**MiniSat** For starters, we can try MiniSat[43]. The standard way to encode CNF expression for MiniSat is to enumerate all OR parts at each line. Also, MiniSat doesn't support variable names, just numbers. Let's enumerate our variables: 1 will be aH, 2 – aL, 3 – bH, 4 – bL.

Here is what I've got when I converted Mathematica expression to the MiniSat input file:

```
p cnf 4 4
-1 -3 0
1 3 0
-2 -4 0
2 4 0
```

Two 4's at the first lines are number of variables and number of clauses respectively. There are 4 lines then, each for each OR clause. Minus before variable number meaning that the variable is negated. Absence of minus – not negated. Zero at the end is just terminating zero, meaning end of the clause.

In other words, each line is OR-clause with optional negations, and the task of MiniSat is to find such set of input, which can satisfy all lines in the input file.

That file I named as *adder.cnf* and now let's try MiniSat:

```
% minisat -verb=0 adder.cnf results.txt
SATISFIABLE
```

The results are in *results.txt* file:

```
SAT
-1 -2 3 4 0
```

This means, if the first two variables (aH and aL) will be *false*, and the last two variables (bH and bL) will be set to *true*, the whole CNF expression is satisfiable. Seems to be true: if bH and bL are the only inputs set to *true*, both resulting bits are also has *true* states.

Now how to get other instances? SAT-solvers, like SMT solvers, produce only one solution (or *instance*).

MiniSat uses PRNG[44] and its initial seed can be set explicitely. I tried different values, but result is still the same. Nevertheless, CryptoMiniSat in this case was able to show all possible 4 instances, in chaotic order, though. So this is not very robust way.

Perhaps, the only known way is to negate solution clause and add it to the input expression. We've got `-1 -2 3 4`, now we can negate all values in it (just toggle minuses: `1 2 -3 -4`) and add it to the end of the input file:

```
p cnf 4 5
```

[43] http://minisat.se/MiniSat.html

[44] Pseudorandom number generator

```
-1 -3 0
1 3 0
-2 -4 0
2 4 0
1 2 -3 -4
```

Now we've got another result:

```
SAT
1 2 -3 -4 0
```

This means, aH and aL must be both *true* and bH and bL must be *false*, to satisfy the input expression. Let's negate this clause and add it again:

```
p cnf 4 6
-1 -3 0
1 3 0
-2 -4 0
2 4 0
1 2 -3 -4
-1 -2 3 4 0
```

The result is:

```
SAT
-1 2 3 -4 0
```

aH=false, aL=true, bH=true, bL=false. This is also correct, according to our truth table.

Let's add it again:

```
p cnf 4 7
-1 -3 0
1 3 0
-2 -4 0
2 4 0
1 2 -3 -4
-1 -2 3 4 0
1 -2 -3 4 0
```

```
SAT
1 -2 -3 4 0
```

*aH=true, aL=false, bH=false, bL=true.* This is also correct.

This is fourth result. There are shouldn't be more. What if to add it?

```
p cnf 4 8
-1 -3 0
1 3 0
-2 -4 0
2 4 0
1 2 -3 -4
-1 -2 3 4 0
1 -2 -3 4 0
-1 2 3 -4 0
```

Now MiniSat just says "UNSATISFIABLE" without any additional information in the resulting file.

Our example is tiny, but MiniSat can work with huge CNF expressions.

**CryptoMiniSat** XOR operation is absent in CNF form, but crucial in cryptographical algorithms. Simplest possible way to represent single XOR operation in CNF form is: $(\neg x \vee \neg y) \wedge (x \vee y)$ – not that small expression, though, many XOR operations in single expression can be optimized better.

One significant difference between MiniSat and CryptoMiniSat is that the latter supports clauses with XOR operations instead of ORs, because CryptoMiniSat has aim to analyze crypto algorithms[45]. XOR clauses are handled by CryptoMiniSat in a special way without translating to OR clauses.

You need just to prepend a clause with "x" in CNF file and OR clause is then treated as XOR clause by CryptoMiniSat. As of 2-bit adder, this smallest possible XOR-CNF expression can be used to find all inputs where both output adder bits are set:

$(aH \oplus bH) \wedge (aL \oplus bL)$

This is `.cnf` file for CryptoMiniSat:

```
p cnf 4 2
x1 3 0
x2 4 0
```

Now I run CryptoMiniSat with various random values to initialize its PRNG …

```
% cryptominisat4 --verb 0 --random 0 XOR_adder.cnf
s SATISFIABLE
v 1 2 -3 -4 0
% cryptominisat4 --verb 0 --random 1 XOR_adder.cnf
s SATISFIABLE
v -1 -2 3 4 0
% cryptominisat4 --verb 0 --random 2 XOR_adder.cnf
s SATISFIABLE
v 1 -2 -3 4 0
% cryptominisat4 --verb 0 --random 3 XOR_adder.cnf
s SATISFIABLE
v 1 2 -3 -4 0
% cryptominisat4 --verb 0 --random 4 XOR_adder.cnf
s SATISFIABLE
v -1 2 3 -4 0
% cryptominisat4 --verb 0 --random 5 XOR_adder.cnf
s SATISFIABLE
v -1 2 3 -4 0
% cryptominisat4 --verb 0 --random 6 XOR_adder.cnf
s SATISFIABLE
v -1 -2 3 4 0
% cryptominisat4 --verb 0 --random 7 XOR_adder.cnf
s SATISFIABLE
v 1 -2 -3 4 0
% cryptominisat4 --verb 0 --random 8 XOR_adder.cnf
s SATISFIABLE
v 1 2 -3 -4 0
% cryptominisat4 --verb 0 --random 9 XOR_adder.cnf
s SATISFIABLE
v 1 2 -3 -4 0
```

Nevertheless, all 4 possible solutions are:

```
v -1 -2 3 4 0
v -1 2 3 -4 0
v 1 -2 -3 4 0
v 1 2 -3 -4 0
```

…the same as reported by MiniSat.

[45] http://www.msoos.org/xor-clauses/

### 2.3.3 Picosat

At least Picosat can enumerate all possible solutions without crutches I just shown:

```
% picosat --all adder.cnf
s SATISFIABLE
v -1 -2 3 4 0
s SATISFIABLE
v -1 2 3 -4 0
s SATISFIABLE
v 1 2 -3 -4 0
s SATISFIABLE
v 1 -2 -3 4 0
s SOLUTIONS 4
```

### 2.3.4 MaxSAT

MaxSAT problem is a problem where as many clauses should be satisfied, as possible, but maybe not all.

(Usual) clauses which *must* be satisfied, called *hard clauses*. Clauses which *should* be satisfied, called *soft clauses*.

MaxSAT solver tries to satisfy all *hard clauses* and as much *soft clauses*, as possible.

*.wcnf files are used, the format is almost the same as in DIMACS files, like:

```
p wcnf 207 796 208
208 1 0
208 2 0
208 3 0
208 4 0

...

1 -152 0
1 -153 0
1 -154 0
1 155 0
1 -156 0
1 -157 0
```

Each clause is written as in DIMACS file, but the first number if weight. MaxSAT solver tries to maximize clauses with bigger weights first.

If the weight has *top weight*, the clause is *hard clause* and must always be satisfied. *Top weight* is set in header. In our case, it's 208.

Some well-known MaxSAT solvers are Open-WBO[46], etc.

### 2.3.5 List of SAT-solvers

- MiniSat[47] by Niklas Een and Niklas Sörensson, serving as a base for some others.
- PicoSat, PrecoSat, Lingeling, CaDiCaL[48]. All created by Armin Biere. Plingeling supports multithreading.
- CryptoMiniSat. Created by Mate Soos for cryptographical problems exploration. Supports XOR clauses, multithreading. Has Python API.

MaxSAT solvers:

- Open-WBO[49], by Ruben Martins, Vasco Manquinho, Inês Lynce.

[46] http://sat.inesc-id.pt/open-wbo
[47] http://minisat.se/
[48] https://github.com/arminbiere/cadical
[49] http://sat.inesc-id.pt/open-wbo/

# 3 Equations

## 3.1 Solving XKCD 287

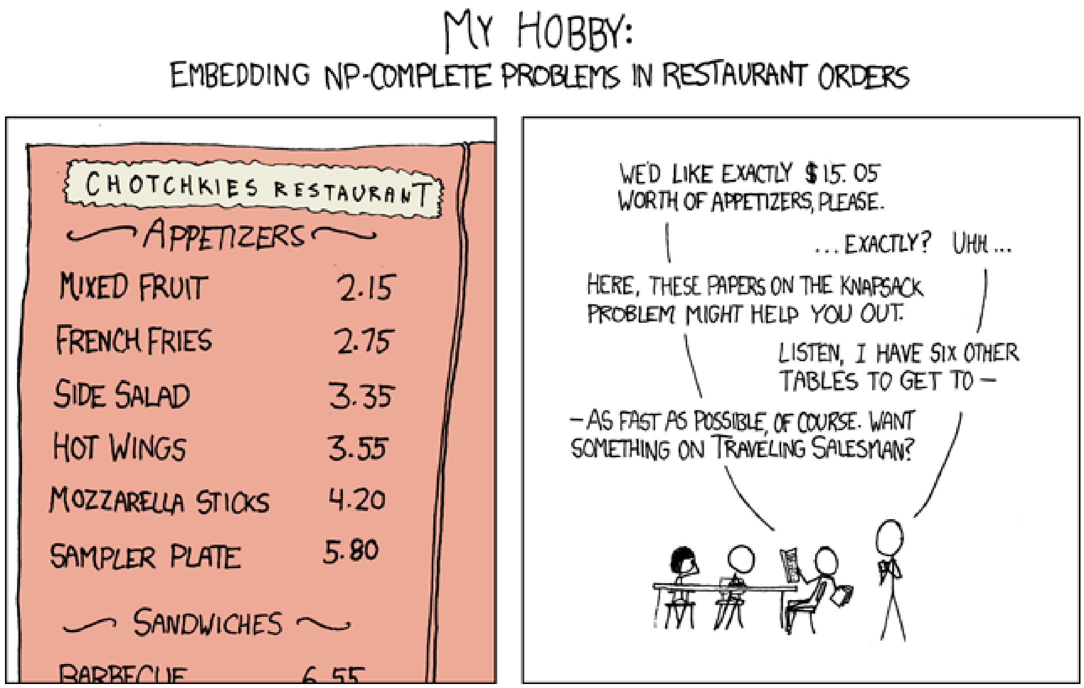



Figure 3: xkcd #287

( https://www.xkcd.com/287/ )

The problem is to solve the following equation: $2.15a + 2.75b + 3.35c + 3.55d + 4.20e + 5.80f == 15.05$, where a..f are integers. So this is a linear diophantine equation.

```
from MK85 import *

s=MK85()
a=s.BitVec("a", 16)
b=s.BitVec("b", 16)
c=s.BitVec("c", 16)
d=s.BitVec("d", 16)
e=s.BitVec("e", 16)
f=s.BitVec("f", 16)

s.add(a<=10)
s.add(b<=10)
s.add(c<=10)
s.add(d<=10)
s.add(e<=10)
s.add(f<=10)

s.add(a*215 + b*275 + c*335 + d*355 + e*420 + f*580 == 1505)

while s.check():
    m=s.model()
    print m
    # block current solution and solve again:
```

```
    s.add(expr.Not(And(a==m["a"], b==m["b"], c==m["c"], d==m["d"], e==m["e"], f==m["f"])
        ))
```

( The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/xkcd287/xkcd287_MK85.py )

There are just 2 solutions:

```
{'a': 7, 'c': 0, 'b': 0, 'e': 0, 'd': 0, 'f': 0}
{'a': 1, 'c': 0, 'b': 0, 'e': 0, 'd': 2, 'f': 1}
```

Wolfram Mathematica can solve the equation as well:

```
In[]:= FindInstance[2.15 a + 2.75 b + 3.35 c + 3.55 d + 4.20 e + 5.80 f == 15.05 &&
        a >= 0 && b >= 0 && c >= 0 && d >= 0  && e >= 0 && f >= 0,
        {a, b, c, d, e, f}, Integers, 1000]

Out[]= {{a -> 1, b -> 0, c -> 0, d -> 2, e -> 0, f -> 1},
        {a -> 7, b -> 0, c -> 0, d -> 0, e -> 0, f -> 0}}
```

1000 means "find at most 1000 solutions", but only 2 are found. See also: http://reference.wolfram.com/language/ref/FindInstance.html.

Other ways to solve it: https://stackoverflow.com/questions/141779/solving-the-np-complete-problem-in-xkcd, http://www.explainxkcd.com/wiki/index.php/287:_NP-Complete.

The solution using Z3: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/xkcd287/xkcd287_Z3.py )

## 3.2 XKCD 287 in SMT-LIB 2.x format

```
; tested using MK85
; would work for Z3 if you uncomment "check-sat" and "get-model" and comment "get-all
   models"

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun a () (_ BitVec 16))
(declare-fun b () (_ BitVec 16))
(declare-fun c () (_ BitVec 16))
(declare-fun d () (_ BitVec 16))
(declare-fun e () (_ BitVec 16))
(declare-fun f () (_ BitVec 16))

(assert (bvult a #x0010))
(assert (bvult b #x0010))
(assert (bvult c #x0010))
(assert (bvult d #x0010))
(assert (bvult e #x0010))
(assert (bvult f #x0010))

(assert
        (=
                (bvadd
                        (bvmul (_ bv215 16) a)
                        (bvmul (_ bv275 16) b)
                        (bvmul (_ bv335 16) c)
                        (bvmul (_ bv355 16) d)
                        (bvmul (_ bv420 16) e)
```

```
                    (bvmul (_ bv580 16) f)
                )
                (_ bv1505 16)
        )
)

;(check-sat)
;(get-model)
(get-all-models)

; correct answer:

;(model
;        (define-fun a () (_ BitVec 16) (_ bv7 16)) ; 0x7
;        (define-fun b () (_ BitVec 16) (_ bv0 16)) ; 0x0
;        (define-fun c () (_ BitVec 16) (_ bv0 16)) ; 0x0
;        (define-fun d () (_ BitVec 16) (_ bv0 16)) ; 0x0
;        (define-fun e () (_ BitVec 16) (_ bv0 16)) ; 0x0
;        (define-fun f () (_ BitVec 16) (_ bv0 16)) ; 0x0
;)
;(model
;        (define-fun a () (_ BitVec 16) (_ bv1 16)) ; 0x1
;        (define-fun b () (_ BitVec 16) (_ bv0 16)) ; 0x0
;        (define-fun c () (_ BitVec 16) (_ bv0 16)) ; 0x0
;        (define-fun d () (_ BitVec 16) (_ bv2 16)) ; 0x2
;        (define-fun e () (_ BitVec 16) (_ bv0 16)) ; 0x0
;        (define-fun f () (_ BitVec 16) (_ bv1 16)) ; 0x1
;)
;Model count: 2
```

### 3.3 Art of problem solving

http://artofproblemsolving.com/wiki/index.php?title=2017_AMC_12A_Problems/Problem_2:

```
The sum of two nonzero real numbers is 4 times their product.
What is the sum of the reciprocals of the two numbers?
```

We're going to solve this over real numbers:

```
from z3 import *

x, y = Reals('x y')

s=Solver()

s.add(x>0)
s.add(y>0)

s.add(x+y == 4*x*y)

print s.check()
m=s.model()
print "the model:"
print m
print "the answer:", m.evaluate (1/x + 1/y)
```

Instead of pulling values from the model and then compute the final result on Python's side, we can evaluate an expression ($\frac{1}{x} + \frac{1}{y}$) *inside* the model we've got:

```
sat
the model:
[x = 1, y = 1/3]
the answer: 4
```

## 3.4 Yet another explanation of modulo inverse using SMT-solvers

### 3.4.1 Introduction

Let's imagine, we work on 4-bit CPU, it has 4-bit registers, each can hold a value in 0..15 range.

Now we want to divide by 3 using multiplication. Let's find modulo inverse of 3 using Wolfram Mathematica:

```
In[]:= PowerMod[3, -1, 16]
Out[]= 11
```

This is in fact solution of a $3m = 16k + 1$ equation ($16 = 2^4$):

```
In[]:= FindInstance[3 m == 16 k + 1, {m, k}, Integers]
Out[]= {{m -> 11, k -> 2}}
```

The "magic number" for division by 3 is 11. Multiply by 11 instead of dividing by 3 and you'll get a result (quotient).

This works, let's divide 6 by 3. We can now do this by multiplying 6 by 11, this is 66=0x42, but on 4-bit register, only 0x2 will be left in register ($0x42 \equiv 2 \mod 2^4$). Yes, 2 is correct answer, 6/3=2.

Let's divide 3, 6 and 9 by 3, by multiplying by 11 (m).

```
          |123456789abcdef0|12 ... f0|123456789abcdef0|12 ... f0|123456789abcdef0|12 ... f0|123456789abcdef0|
    m=11  |***********     |   ...   |                |   ...   |                |   ...   |                |
3/3 3m=33 |****************|** ... **|*               |   ...   |                |   ...   |                |
6/3 6m=66 |****************|** ... **|****************|** ... **|**              |   ...   |                |
9/3 9m=99 |****************|** ... **|****************|** ... **|****************|** ... **|***             |
```

A "protruding" asterisk(s) (*) in the last non-empty chunk is what will be left in 4-bit register. This is 1 in case of 33, 2 if 66, 3 if 99.

In fact, this "protrusion" is defined by 1 in the equation we've solved. Let's replace 1 with 2:

```
In[]:= FindInstance[3 m == 16 k + 2, {m, k}, Integers]
Out[]= {{m -> 6, k -> 1}}
```

Now the new "magic number" is 6. Let's divide 3 by 3. 3*6=18=0x12, 2 will be left in 4-bit register. This is incorrect, we have 2 instead of 1. 2 asterisks are "protruding". Let's divide 6 by 3. 6*6=36=0x24, 4 will be left in the register. This is also incorrect, we now have 4 "protruding" asterisks instead of correct 2.

Replace 1 in the equation by 0, and nothing will "protrude".

Now the problem: this only works for dividends in 3x form, i.e., which can be divided by 3 with no remainder. Try to divide 4 by 3, 4*11=44=0x2c, 12 will be left in register, this is incorrect. The correct quotient is 1.

We can also notice that the 4-bit register is "overflown" during multiplication twice as much as in "incorrect" result in low 4 bits.

Here is what we can do: use only high 4 bits and drop low 4 bits. 4*11=0x2c and 2 is high 4 bits. Divide 2 by 2, this is 1.

Let's "divide" 8 by 3. 8*11=88=0x58. 5/2=2, this is correct answer again.

Now this is the formula we can use on our 4-bit CPU to divide numbers by 3: "x*3 » 4 / 2" or "x*3 » 5". This is the same as almost all modern compilers do instead of integer division, but they do this for 32-bit and 64-bit registers.

### 3.4.2 Back to SMT-solver

By which constant we must multiply a random number, so that the result would be as if we divided them by 3?

```
from z3 import *

m=BitVec('m', 32)

```

```
s=Solver()

# wouldn't work for 10, etc
divisor=3

# random constant, must be divisible by divisor:
const=(0x1234567*divisor)

s.add(const*m == const/divisor)

print s.check()
print "%x" % s.model()[m].as_long()
```

The magic number is:

```
sat
aaaaaaab
```

Indeed, this is modulo inverse of 3 modulo $2^{32}$: https://www.wolframalpha.com/input/?i=PowerMod%5B3,-1,2%5E32%5D.

Let's check using my calculator:

```
[3] 123456*0xaaaaaaab
[3] (unsigned) 353492988371136 0x141800000a0c0 0
    b1010000011000000000000000000000001010000011000000
[4] 123456/3
[4] (unsigned) 41152 0xa0c0 0b1010000011000000
```

The problem is simple enough to be solved using MK85:

```
; find modulo inverse
; checked with Z3 and MK85
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun m () (_ BitVec 16))
(declare-fun a () (_ BitVec 16))
(declare-fun b () (_ BitVec 16))

(assert (= a (bvudiv #x1236 #x0003)))
(assert (= b (bvmul #x1236 m)))

(assert (= a b))

; without this constraint, two results would be generated (with MSB=1 and MSB=0),
; but we need only one indeed, MSB of m has no effect of multiplication here
; and SMT-solver offers two solutions
(assert (= (bvand m #x8000) #x0000))

(check-sat)
(get-model)
;(get-all-models)
```

```
sat
(model
        (define-fun m () (_ BitVec 16) (_ bv10923 16)) ; 0x2aab
        (define-fun a () (_ BitVec 16) (_ bv1554 16)) ; 0x612
        (define-fun b () (_ BitVec 16) (_ bv1554 16)) ; 0x612
)
```

However, it wouldn't work for 10, because there are no modulo inverse of 10 modulo $2^{32}$, SMT solver would give "unsat".

### 3.5 School-level equation

Let's revisit school-level system of equations from (2.2.2).

We will force KLEE to find a path, where all the constraints are satisfied:

```
int main()
{
        int circle, square, triangle;

        klee_make_symbolic(&circle, sizeof circle, "circle");
        klee_make_symbolic(&square, sizeof square, "square");
        klee_make_symbolic(&triangle, sizeof triangle, "triangle");

        if (circle+circle!=10) return 0;
        if (circle*square+square!=12) return 0;
        if (circle*square-triangle*circle!=circle) return 0;

        // all constraints should be satisfied at this point
        // force KLEE to produce .err file:
        klee_assert(0);
};
```

```
% clang -emit-llvm -c -g klee_eq.c
...

% klee klee_eq.bc
KLEE: output directory is "/home/klee/klee-out-93"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_eq.c:18: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 32
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

Let's find out, where `klee_assert()` has been triggered:

```
% ls klee-last | grep err
test000001.external.err

% ktest-tool --write-ints klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['klee_eq.bc']
num objects: 3
object    0: name: b'circle'
object    0: size: 4
object    0: data: 5
object    1: name: b'square'
object    1: size: 4
object    1: data: 2
object    2: name: b'triangle'
object    2: size: 4
object    2: data: 1
```

This is indeed correct solution to the system of equations.

KLEE has *intrinsic* `klee_assume()` which tells KLEE to cut path if some constraint is not satisfied. So we can rewrite our example in such cleaner way:

```
int main()
{
        int circle, square, triangle;

        klee_make_symbolic(&circle, sizeof circle, "circle");
        klee_make_symbolic(&square, sizeof square, "square");
        klee_make_symbolic(&triangle, sizeof triangle, "triangle");

        klee_assume (circle+circle==10);
        klee_assume (circle*square+square==12);
        klee_assume (circle*square-triangle*circle==circle);

        // all constraints should be satisfied at this point
        // force KLEE to produce .err file:
        klee_assert(0);
};
```

## 3.6 Cracking Minesweeper with SMT solver

For those who are not very good at playing Minesweeper (like me), it's possible to predict bombs' placement without touching debugger.

Here is a clicked somewhere and I see revealed empty cells and cells with known number of "neighbours":


What we have here, actually? Hidden cells, empty cells (where bombs are not present), and empty cells with numbers, which shows how many bombs are placed nearby.

### 3.6.1 The method

Here is what we can do: we will try to place a bomb to all possible hidden cells and ask Z3 SMT solver, if it can disprove the very fact that the bomb can be placed there.

Take a look at this fragment. "?" mark is for hidden cell, "." is for empty cell, number is a number of neighbours.

| | C1 | C2 | C3 |
|---|---|---|---|
| R1 | ? | ? | ? |
| R2 | ? | 3 | . |
| R3 | ? | 1 | . |

So there are 5 hidden cells. We will check each hidden cell by placing a bomb there. Let's first pick top/left cell:

| | C1 | C2 | C3 |
|---|---|---|---|
| R1 | * | ? | ? |
| R2 | ? | 3 | . |
| R3 | ? | 1 | . |

Then we will try to solve the following system of equations (*RrCc* is cell of row *r* and column *c*):

- R1C2+R2C1+R2C2=1 (because we placed bomb at R1C1)
- R2C1+R2C2+R3C1=1 (because we have "1" at R3C2)
- R1C1+R1C2+R1C3+R2C1+R2C2+R2C3+R3C1+R3C2+R3C3=3 (because we have "3" at R2C2)
- R1C2+R1C3+R2C2+R2C3+R3C2+R3C3=0 (because we have "." at R2C3)
- R2C2+R2C3+R3C2+R3C3=0 (because we have "." at R3C3)

As it turns out, this system of equations is satisfiable, so there could be a bomb at this cell. But this information is not interesting to us, since we want to find cells we can freely click on. And we will try another one. And if the equation will be unsatisfiable, that would imply that a bomb cannot be there and we can click on it.

#### 3.6.2 The code

```
#!/usr/bin/python

known=[
"01?10001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"????1001?",
"????3101?",
"?????211?",
"?????????"]

from z3 import *
import sys

WIDTH=len(known[0])
HEIGHT=len(known)

print "WIDTH=", WIDTH, "HEIGHT=", HEIGHT

def chk_bomb(row, col):

    s=Solver()

    cells=[[Int('cell_r=%d_c=%d' % (r,c)) for c in range(WIDTH+2)] for r in range(HEIGHT
        +2)]

    # make border
    for c in range(WIDTH+2):
        s.add(cells[0][c]==0)
        s.add(cells[HEIGHT+1][c]==0)
    for r in range(HEIGHT+2):
```

```
        s.add(cells[r][0]==0)
        s.add(cells[r][WIDTH+1]==0)

    for r in range(1,HEIGHT+1):
        for c in range(1,WIDTH+1):

            t=known[r-1][c-1]
            if t in "012345678":
                s.add(cells[r][c]==0)
                # we need empty border so the following expression would be able to work
                    for all possible cases:
                s.add(cells[r-1][c-1] + cells[r-1][c] + cells[r-1][c+1] + cells[r][c-1]
                   + cells[r][c+1] + cells[r+1][c-1] + cells[r+1][c] + cells[r+1][c
                   +1]==int(t))

    # place bomb:
    s.add(cells[row][col]==1)

    result=str(s.check())
    if result=="unsat":
        print "row=%d col=%d, unsat!" % (row, col)

# enumerate all hidden cells:
for r in range(1,HEIGHT+1):
    for c in range(1,WIDTH+1):
        if known[r-1][c-1]=="?":
            chk_bomb(r, c)
```

The code is almost self-explanatory. We need border for the same reason, why Conway's "Game of Life" implementations also has border (to make calculation function simpler). Whenever we know that the cell is free of bomb, we put zero there. Whenever we know number of neighbours, we add a constraint, again, just like in "Game of Life": number of neighbours must be equal to the number we have seen in the Minesweeper. Then we place bomb somewhere and check.

Let's run:

```
row=1 col=3, unsat!
row=6 col=2, unsat!
row=6 col=3, unsat!
row=7 col=4, unsat!
row=7 col=9, unsat!
row=8 col=9, unsat!
```

These are cells where I can click safely, so I did:

Now we have more information, so we update input:

```
known=[
"01110001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"?11?1001?",
"???331011",
"?????2110",
"???????10"]
```

I run it again:

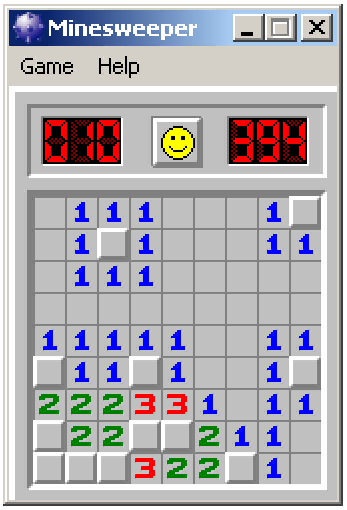
```
row=7 col=1, unsat!
row=7 col=2, unsat!
row=7 col=3, unsat!
row=8 col=3, unsat!
row=9 col=5, unsat!
row=9 col=6, unsat!
```

I click on these cells again:

I update it again:

```
known=[
"01110001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"?11?1001?",
"222331011",
"??2??2110",
"????22?10"]
```

```
row=8 col=2, unsat!
row=9 col=4, unsat!
```




This is last update:

```
known=[
"01110001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"?11?1001?",
"222331011",
"?22??2110",
"???322?10"]
```

...last result:

```
row=9 col=1, unsat!
row=9 col=2, unsat!
```

Voila!

Some discussion on HN: https://news.ycombinator.com/item?id=13797375.

## 3.7 Cracking Minesweeper with SAT solver

### 3.7.1 Simple *population count* function

First of all, somehow we need to count neighbour bombs. The counting function is similar to *population count* function.

We can try to make CNF expression using Wolfram Mathematica. This will be a function, returning *True* if any of 2 bits of 8 inputs bits are *True* and others are *False*. First, we make truth table of such function:

```
In[]:= tbl2 =
 Table[PadLeft[IntegerDigits[i, 2], 8] ->
   If[Equal[DigitCount[i, 2][[1]], 2], 1, 0], {i, 0, 255}]

Out[]= {{0, 0, 0, 0, 0, 0, 0, 0} -> 0, {0, 0, 0, 0, 0, 0, 0, 1} -> 0,
{0, 0, 0, 0, 0, 0, 1, 0} -> 0, {0, 0, 0, 0, 0, 0, 1, 1} -> 1,
{0, 0, 0, 0, 0, 1, 0, 0} -> 0, {0, 0, 0, 0, 0, 1, 0, 1} -> 1,
{0, 0, 0, 0, 0, 1, 1, 0} -> 1, {0, 0, 0, 0, 0, 1, 1, 1} -> 0,
{0, 0, 0, 0, 1, 0, 0, 0} -> 0, {0, 0, 0, 0, 1, 0, 0, 1} -> 1,
{0, 0, 0, 0, 1, 0, 1, 0} -> 1, {0, 0, 0, 0, 1, 0, 1, 1} -> 0,
...
{1, 1, 1, 1, 1, 0, 1, 0} -> 0, {1, 1, 1, 1, 1, 0, 1, 1} -> 0,
{1, 1, 1, 1, 1, 1, 0, 0} -> 0, {1, 1, 1, 1, 1, 1, 0, 1} -> 0,
{1, 1, 1, 1, 1, 1, 1, 0} -> 0, {1, 1, 1, 1, 1, 1, 1, 1} -> 0}
```

Now we can make CNF expression using this truth table:

```
In[]:= BooleanConvert[
 BooleanFunction[tbl2, {a, b, c, d, e, f, g, h}], "CNF"]

Out[]= (! a || ! b || ! c) && (! a || ! b || ! d) && (! a || !
    b || ! e) && (! a || ! b || ! f) && (! a || ! b || ! g) && (!
    a || ! b || ! h) && (! a || ! c || ! d) && (! a || ! c || !
    e) && (! a || ! c || ! f) && (! a || ! c || ! g) && (! a || !
    c || ! h) && (! a || ! d || ! e) && (! a || ! d || ! f) && (!
    a || ! d || ! g) && (! a || ! d || ! h) && (! a || ! e || !
    f) && (! a || ! e || ! g) && (! a || ! e || ! h) && (! a || !
    f || ! g) && (! a || ! f || ! h) && (! a || ! g || ! h) && (a ||
   b || c || d || e || f || g) && (a || b || c || d || e || f ||
   h) && (a || b || c || d || e || g || h) && (a || b || c || d || f ||
    g || h) && (a || b || c || e || f || g || h) && (a || b || d ||
```

```
  e || f || g || h) && (a || c || d || e || f || g ||
  h) && (! b || ! c || ! d) && (! b || ! c || ! e) && (! b || !
   c || ! f) && (! b || ! c || ! g) && (! b || ! c || ! h) && (!
   b || ! d || ! e) && (! b || ! d || ! f) && (! b || ! d || !
   g) && (! b || ! d || ! h) && (! b || ! e || ! f) && (! b || !
   e || ! g) && (! b || ! e || ! h) && (! b || ! f || ! g) && (!
   b || ! f || ! h) && (! b || ! g || ! h) && (b || c || d || e ||
  f || g ||
  h) && (! c || ! d || ! e) && (! c || ! d || ! f) && (! c || !
   d || ! g) && (! c || ! d || ! h) && (! c || ! e || ! f) && (!
   c || ! e || ! g) && (! c || ! e || ! h) && (! c || ! f || !
   g) && (! c || ! f || ! h) && (! c || ! g || ! h) && (! d || !
   e || ! f) && (! d || ! e || ! g) && (! d || ! e || ! h) && (!
   d || ! f || ! g) && (! d || ! f || ! h) && (! d || ! g || !
   h) && (! e || ! f || ! g) && (! e || ! f || ! h) && (! e || !
   g || ! h) && (! f || ! g || ! h)
```

The syntax is similar to C/C++. Let's check it.

I wrote a Python function to convert Mathematica's output into CNF file which can be feeded to SAT solver:

```
#!/usr/bin/python

import subprocess

def mathematica_to_CNF (s, a):
    s=s.replace("a", a[0]).replace("b", a[1]).replace("c", a[2]).replace("d", a[3])
    s=s.replace("e", a[4]).replace("f", a[5]).replace("g", a[6]).replace("h", a[7])
    s=s.replace("!", "-").replace("||", " ").replace("(", "").replace(")", "")
    s=s.split ("&&")
    return s

def POPCNT2 (a):
    s="(!a||!b||!c)&&(!a||!b||!d)&&(!a||!b||!e)&&(!a||!b||!f)&&(!a||!b||!g)&&(!a||!b||!h
       )&&(!a||!c||!d)&&" \
      "(!a||!c||!e)&&(!a||!c||!f)&&(!a||!c||!g)&&(!a||!c||!h)&&(!a||!d||!e)&&(!a||!d||!f
        )&&(!a||!d||!g)&&" \
      "(!a||!d||!h)&&(!a||!e||!f)&&(!a||!e||!g)&&(!a||!e||!h)&&(!a||!f||!g)&&(!a||!f||!h
        )&&(!a||!g||!h)&&" \
      "(a||b||c||d||e||f||g)&&(a||b||c||d||e||f||h)&&(a||b||c||d||e||g||h)&&(a||b||c||d
        ||f||g||h)&&" \
      "(a||b||c||e||f||g||h)&&(a||b||d||e||f||g||h)&&(a||c||d||e||f||g||h)&&(!b||!c||!d)
        &&(!b||!c||!e)&&" \
      "(!b||!c||!f)&&(!b||!c||!g)&&(!b||!c||!h)&&(!b||!d||!e)&&(!b||!d||!f)&&(!b||!d||!g
        )&&(!b||!d||!h)&&" \
      "(!b||!e||!f)&&(!b||!e||!g)&&(!b||!e||!h)&&(!b||!f||!g)&&(!b||!f||!h)&&(!b||!g||!h
        )&&(b||c||d||e||f||g||h)&&" \
      "(!c||!d||!e)&&(!c||!d||!f)&&(!c||!d||!g)&&(!c||!d||!h)&&(!c||!e||!f)&&(!c||!e||!g
        )&&(!c||!e||!h)&&" \
      "(!c||!f||!g)&&(!c||!f||!h)&&(!c||!g||!h)&&(!d||!e||!f)&&(!d||!e||!g)&&(!d||!e||!h
        )&&(!d||!f||!g)&&" \
      "(!d||!f||!h)&&(!d||!g||!h)&&(!e||!f||!g)&&(!e||!f||!h)&&(!e||!g||!h)&&(!f||!g||!h
        )"
    return mathematica_to_CNF(s, a)

clauses=POPCNT2(["1","2","3","4","5","6","7","8"])

f=open("tmp.cnf", "w")
```

```
f.write ("p cnf 8 "+str(len(clauses))+"\n")
for c in clauses:
    f.write(c+" 0\n")
f.close()
```

It replaces a/b/c/... variables to the variable names passed (1/2/3...), reworks syntax, etc. Here is a result:

```
p cnf 8 64
-1 -2 -3 0
-1 -2 -4 0
-1 -2 -5 0
-1 -2 -6 0
-1 -2 -7 0
-1 -2 -8 0
-1 -3 -4 0
-1 -3 -5 0
-1 -3 -6 0
-1 -3 -7 0
-1 -3 -8 0
-1 -4 -5 0
-1 -4 -6 0
-1 -4 -7 0
-1 -4 -8 0
-1 -5 -6 0
-1 -5 -7 0
-1 -5 -8 0
-1 -6 -7 0
-1 -6 -8 0
-1 -7 -8 0
1 2 3 4 5 6 7 0
1 2 3 4 5 6 8 0
1 2 3 4 5 7 8 0
1 2 3 4 6 7 8 0
1 2 3 5 6 7 8 0
1 2 4 5 6 7 8 0
1 3 4 5 6 7 8 0
-2 -3 -4 0
-2 -3 -5 0
-2 -3 -6 0
-2 -3 -7 0
-2 -3 -8 0
-2 -4 -5 0
-2 -4 -6 0
-2 -4 -7 0
-2 -4 -8 0
-2 -5 -6 0
-2 -5 -7 0
-2 -5 -8 0
-2 -6 -7 0
-2 -6 -8 0
-2 -7 -8 0
2 3 4 5 6 7 8 0
-3 -4 -5 0
-3 -4 -6 0
-3 -4 -7 0
-3 -4 -8 0
-3 -5 -6 0
```

```
-3 -5 -7 0
-3 -5 -8 0
-3 -6 -7 0
-3 -6 -8 0
-3 -7 -8 0
-4 -5 -6 0
-4 -5 -7 0
-4 -5 -8 0
-4 -6 -7 0
-4 -6 -8 0
-4 -7 -8 0
-5 -6 -7 0
-5 -6 -8 0
-5 -7 -8 0
-6 -7 -8 0
```

I can run it:

```
% minisat -verb=0 tst1.cnf results.txt
SATISFIABLE

% cat results.txt
SAT
1 -2 -3 -4 -5 -6 -7 8 0
```

The variable name in results lacking minus sign is *True*. Variable name with minus sign is *False*. We see there are just two variables are *True*: 1 and 8. This is indeed correct: MiniSat solver found a condition, for which our function returns *True*. Zero at the end is just a terminal symbol which means nothing.

We can ask MiniSat for another solution, by adding current solution to the input CNF file, but with all variables negated:

```
...
-5 -6 -8 0
-5 -7 -8 0
-6 -7 -8 0
-1 2 3 4 5 6 7 -8 0
```

In plain English language, this means "give me ANY solution which can satisfy all clauses, but also not equal to the last clause we've just added".

MiniSat, indeed, found another solution, again, with only 2 variables equal to *True*:

```
% minisat -verb=0 tst2.cnf results.txt
SATISFIABLE

% cat results.txt
SAT
1 2 -3 -4 -5 -6 -7 -8 0
```

By the way, *population count* function for 8 neighbours (POPCNT8) in CNF form is simplest:

```
a&&b&&c&&d&&e&&f&&g&&h
```

Indeed: it's true if all 8 input bits are *True*.

The function for 0 neighbours (POPCNT0) is also simple:

```
!a&&!b&&!c&&!d&&!e&&!f&&!g&&!h
```

It means, it will return *True*, if all input variables are *False*.

By the way, POPCNT1 function is also simple:

```
(!a||!b)&&(!a||!c)&&(!a||!d)&&(!a||!e)&&(!a||!f)&&(!a||!g)&&(!a||!h)&&(a||b||c||d||e||f
    ||g||h)&&
(!b||!c)&&(!b||!d)&&(!b||!e)&&(!b||!f)&&(!b||!g)&&(!b||!h)&&(!c||!d)&&(!c||!e)&&(!c||!f)
    &&(!c||!g)&&
(!c||!h)&&(!d||!e)&&(!d||!f)&&(!d||!g)&&(!d||!h)&&(!e||!f)&&(!e||!g)&&(!e||!h)&&(!f||!g)
    &&(!f||!h)&&(!g||!h)
```

There is just enumeration of all possible pairs of 8 variables (a/b, a/c, a/d, etc), which implies: no two bits must be present simultaneously in each possible pair. And there is another clause: "(a||b||c||d||e||f||g||h)", which implies: at least one bit must be present among 8 variables.

And yes, you can ask Mathematica for finding CNF expressions for any other truth table.

### 3.7.2 Minesweeper

Now we can use Mathematica to generate all *population count* functions for 0..8 neighbours.

For $9 \cdot 9$ Minesweeper matrix including invisible border, there will be $11 \cdot 11 = 121$ variables, mapped to Minesweeper matrix like this:

```
 1    2   3   4   5   6   7   8   9  10  11
12   13  14  15  16  17  18  19  20  21  22
23   24  25  26  27  28  29  30  31  32  33
34   35  36  37  38  39  40  41  42  43  44

...

100 101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120 121
```

Then we write a Python script which stacks all *population count* functions: each function for each known number of neighbours (digit on Minesweeper field). Each POPCNTx() function takes list of variable numbers and outputs list of clauses to be added to the final CNF file.

As of empty cells, we also add them as clauses, but with minus sign, which means, the variable must be *False*. Whenever we try to place bomb, we add its variable as clause without minus sign, this means the variable must be *True*.

Then we execute external minisat process. The only thing we need from it is exit code. If an input CNF is `UNSAT`, it returns 20:

We use here the information from the previous solving of Minesweeper: 3.6.

```
#!/usr/bin/python

import subprocess

WIDTH=9
HEIGHT=9
VARS_TOTAL=(WIDTH+2)*(HEIGHT+2)

known=[
"01?10001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"????1001?",
"????3101?",
"?????211?",
"?????????"]

def mathematica_to_CNF (s, a):
```

```
    s=s.replace("a", a[0]).replace("b", a[1]).replace("c", a[2]).replace("d", a[3])
    s=s.replace("e", a[4]).replace("f", a[5]).replace("g", a[6]).replace("h", a[7])
    s=s.replace("!", "-").replace("||", " ").replace("(", "").replace(")", "")
    s=s.split ("&&")
    return s

def POPCNT0 (a):
    s="!a&&!b&&!c&&!d&&!e&&!f&&!g&&!h"
    return mathematica_to_CNF(s, a)

def POPCNT1 (a):
    s="(!a||!b)&&(!a||!c)&&(!a||!d)&&(!a||!e)&&(!a||!f)&&(!a||!g)&&(!a||!h)&&(a||b||c||d||e||f||g||h)&&" \
      "(!b||!c)&&(!b||!d)&&(!b||!e)&&(!b||!f)&&(!b||!g)&&(!b||!h)&&(!c||!d)&&(!c||!e)&&(!c||!f)&&(!c||!g)&&" \
      "(!c||!h)&&(!d||!e)&&(!d||!f)&&(!d||!g)&&(!d||!h)&&(!e||!f)&&(!e||!g)&&(!e||!h)&&(!f||!g)&&(!f||!h)&&(!g||!h)"
    return mathematica_to_CNF(s, a)

def POPCNT2 (a):
    s="(!a||!b||!c)&&(!a||!b||!d)&&(!a||!b||!e)&&(!a||!b||!f)&&(!a||!b||!g)&&(!a||!b||!h)&&(!a||!c||!d)&&" \
      "(!a||!c||!e)&&(!a||!c||!f)&&(!a||!c||!g)&&(!a||!c||!h)&&(!a||!d||!e)&&(!a||!d||!f)&&(!a||!d||!g)&&" \
      "(!a||!d||!h)&&(!a||!e||!f)&&(!a||!e||!g)&&(!a||!e||!h)&&(!a||!f||!g)&&(!a||!f||!h)&&(!a||!g||!h)&&" \
      "(a||b||c||d||e||f||g)&&(a||b||c||d||e||f||h)&&(a||b||c||d||e||g||h)&&(a||b||c||d||f||g||h)&&" \
      "(a||b||c||e||f||g||h)&&(a||b||d||e||f||g||h)&&(a||c||d||e||f||g||h)&&(!b||!c||!d)&&(!b||!c||!e)&&" \
      "(!b||!c||!f)&&(!b||!c||!g)&&(!b||!c||!h)&&(!b||!d||!e)&&(!b||!d||!f)&&(!b||!d||!g)&&(!b||!d||!h)&&" \
      "(!b||!e||!f)&&(!b||!e||!g)&&(!b||!e||!h)&&(!b||!f||!g)&&(!b||!f||!h)&&(!b||!g||!h)&&(b||c||d||e||f||g||h)&&" \
      "(!c||!d||!e)&&(!c||!d||!f)&&(!c||!d||!g)&&(!c||!d||!h)&&(!c||!e||!f)&&(!c||!e||!g)&&(!c||!e||!h)&&" \
      "(!c||!f||!g)&&(!c||!f||!h)&&(!c||!g||!h)&&(!d||!e||!f)&&(!d||!e||!g)&&(!d||!e||!h)&&(!d||!f||!g)&&" \
      "(!d||!f||!h)&&(!d||!g||!h)&&(!e||!f||!g)&&(!e||!f||!h)&&(!e||!g||!h)&&(!f||!g||!h)"
    return mathematica_to_CNF(s, a)

def POPCNT3 (a):
    s="(!a||!b||!c||!d)&&(!a||!b||!c||!e)&&(!a||!b||!c||!f)&&(!a||!b||!c||!g)&&(!a||!b||!c||!h)&&" \
      "(!a||!b||!d||!e)&&(!a||!b||!d||!f)&&(!a||!b||!d||!g)&&(!a||!b||!d||!h)&&(!a||!b||!e||!f)&&" \
      "(!a||!b||!e||!g)&&(!a||!b||!e||!h)&&(!a||!b||!f||!g)&&(!a||!b||!f||!h)&&(!a||!b||!g||!h)&&" \
      "(!a||!c||!d||!e)&&(!a||!c||!d||!f)&&(!a||!c||!d||!g)&&(!a||!c||!d||!h)&&(!a||!c||!e||!f)&&" \
      "(!a||!c||!e||!g)&&(!a||!c||!e||!h)&&(!a||!c||!f||!g)&&(!a||!c||!f||!h)&&(!a||!c||!g||!h)&&" \
      "(!a||!d||!e||!f)&&(!a||!d||!e||!g)&&(!a||!d||!e||!h)&&(!a||!d||!f||!g)&&(!a||!d||!f||!h)&&" \
      "(!a||!d||!g||!h)&&(!a||!e||!f||!g)&&(!a||!e||!f||!h)&&(!a||!e||!g||!h)&&(!a||!f
```

```
        ||!g||!h)&&" \
      "(a||b||c||d||e||f)&&(a||b||c||d||e||g)&&(a||b||c||d||e||h)&&(a||b||c||d||f||g)&&(
        a||b||c||d||f||h)&&" \
      "(a||b||c||d||g||h)&&(a||b||c||e||f||g)&&(a||b||c||e||f||h)&&(a||b||c||e||g||h)&&(
        a||b||c||f||g||h)&&" \
      "(a||b||d||e||f||g)&&(a||b||d||e||f||h)&&(a||b||d||e||g||h)&&(a||b||d||f||g||h)&&(
        a||b||e||f||g||h)&&" \
      "(a||c||d||e||f||g)&&(a||c||d||e||f||h)&&(a||c||d||e||g||h)&&(a||c||d||f||g||h)&&(
        a||c||e||f||g||h)&&" \
      "(a||d||e||f||g||h)&&(!b||!c||!d||!e)&&(!b||!c||!d||!f)&&(!b||!c||!d||!g)&&(!b||!c
        ||!d||!h)&&" \
      "(!b||!c||!e||!f)&&(!b||!c||!e||!g)&&(!b||!c||!e||!h)&&(!b||!c||!f||!g)&&(!b||!c
        ||!f||!h)&&" \
      "(!b||!c||!g||!h)&&(!b||!d||!e||!f)&&(!b||!d||!e||!g)&&(!b||!d||!e||!h)&&(!b||!d
        ||!f||!g)&&" \
      "(!b||!d||!f||!h)&&(!b||!d||!g||!h)&&(!b||!e||!f||!g)&&(!b||!e||!f||!h)&&(!b||!e
        ||!g||!h)&&" \
      "(!b||!f||!g||!h)&&(b||c||d||e||f||g)&&(b||c||d||e||f||h)&&(b||c||d||e||g||h)&&(b
        ||c||d||f||g||h)&&" \
      "(b||c||e||f||g||h)&&(b||d||e||f||g||h)&&(!c||!d||!e||!f)&&(!c||!d||!e||!g)&&(!c
        ||!d||!e||!h)&&" \
      "(!c||!d||!f||!g)&&(!c||!d||!f||!h)&&(!c||!d||!g||!h)&&(!c||!e||!f||!g)&&(!c||!e
        ||!f||!h)&&" \
      "(!c||!e||!g||!h)&&(!c||!f||!g||!h)&&(c||d||e||f||g||h)&&(!d||!e||!f||!g)&&(!d||!e
        ||!f||!h)&&" \
      "(!d||!e||!g||!h)&&(!d||!f||!g||!h)&&(!e||!f||!g||!h)"
    return mathematica_to_CNF(s, a)

def POPCNT4 (a):
    s="(!a||!b||!c||!d||!e)&&(!a||!b||!c||!d||!f)&&(!a||!b||!c||!d||!g)&&(!a||!b||!c||!d
      ||!h)&&" \
      "(!a||!b||!c||!e||!f)&&(!a||!b||!c||!e||!g)&&(!a||!b||!c||!e||!h)&&(!a||!b||!c||!f
        ||!g)&&" \
      "(!a||!b||!c||!f||!h)&&(!a||!b||!c||!g||!h)&&(!a||!b||!d||!e||!f)&&(!a||!b||!d||!e
        ||!g)&&" \
      "(!a||!b||!d||!e||!h)&&(!a||!b||!d||!f||!g)&&(!a||!b||!d||!f||!h)&&(!a||!b||!d||!g
        ||!h)&&" \
      "(!a||!b||!e||!f||!g)&&(!a||!b||!e||!f||!h)&&(!a||!b||!e||!g||!h)&&(!a||!b||!f||!g
        ||!h)&&" \
      "(!a||!c||!d||!e||!f)&&(!a||!c||!d||!e||!g)&&(!a||!c||!d||!e||!h)&&(!a||!c||!d||!f
        ||!g)&&" \
      "(!a||!c||!d||!f||!h)&&(!a||!c||!d||!g||!h)&&(!a||!c||!e||!f||!g)&&(!a||!c||!e||!f
        ||!h)&&" \
      "(!a||!c||!e||!g||!h)&&(!a||!c||!f||!g||!h)&&(!a||!d||!e||!f||!g)&&(!a||!d||!e||!f
        ||!h)&&" \
      "(!a||!d||!e||!g||!h)&&(!a||!d||!f||!g||!h)&&(!a||!e||!f||!g||!h)&&(a||b||c||d||e)
        &&(a||b||c||d||f)&&" \
      "(a||b||c||d||g)&&(a||b||c||d||h)&&(a||b||c||e||f)&&(a||b||c||e||g)&&(a||b||c||e||
        h)&&(a||b||c||f||g)&&" \
      "(a||b||c||f||h)&&(a||b||c||g||h)&&(a||b||d||e||f)&&(a||b||d||e||g)&&(a||b||d||e||
        h)&&(a||b||d||f||g)&&" \
      "(a||b||d||f||h)&&(a||b||d||g||h)&&(a||b||e||f||g)&&(a||b||e||f||h)&&(a||b||e||g||
        h)&&(a||b||f||g||h)&&" \
      "(a||c||d||e||f)&&(a||c||d||e||g)&&(a||c||d||e||h)&&(a||c||d||f||g)&&(a||c||d||f||
        h)&&(a||c||d||g||h)&&" \
      "(a||c||e||f||g)&&(a||c||e||f||h)&&(a||c||e||g||h)&&(a||c||f||g||h)&&(a||d||e||f||
```

```
        g)&&(a||d||e||f||h)&&" \
      "(a||d||e||g||h)&&(a||d||f||g||h)&&(a||e||f||g||h)&&(!b||!c||!d||!e||!f)&&(!b||!c
        ||!d||!e||!g)&&" \
      "(!b||!c||!d||!e||!h)&&(!b||!c||!d||!f||!g)&&(!b||!c||!d||!f||!h)&&(!b||!c||!d||!g
        ||!h)&&" \
      "(!b||!c||!e||!f||!g)&&(!b||!c||!e||!f||!h)&&(!b||!c||!e||!g||!h)&&(!b||!c||!f||!g
        ||!h)&&" \
      "(!b||!d||!e||!f||!g)&&(!b||!d||!e||!f||!h)&&(!b||!d||!e||!g||!h)&&(!b||!d||!f||!g
        ||!h)&&" \
      "(!b||!e||!f||!g||!h)&&(b||c||d||e||f)&&(b||c||d||e||g)&&(b||c||d||e||h)&&(b||c||d
        ||f||g)&&" \
      "(b||c||d||f||h)&&(b||c||d||g||h)&&(b||c||e||f||g)&&(b||c||e||f||h)&&(b||c||e||g||
        h)&&" \
      "(b||c||f||g||h)&&(b||d||e||f||g)&&(b||d||e||f||h)&&(b||d||e||g||h)&&(b||d||f||g||
        h)&&" \
      "(b||e||f||g||h)&&(!c||!d||!e||!f||!g)&&(!c||!d||!e||!f||!h)&&(!c||!d||!e||!g||!h)
        &&" \
      "(!c||!d||!f||!g||!h)&&(!c||!e||!f||!g||!h)&&(c||d||e||f||g)&&(c||d||e||f||h)&&(c
        ||d||e||g||h)&&" \
      "(c||d||f||g||h)&&(c||e||f||g||h)&&(!d||!e||!f||!g||!h)&&(d||e||f||g||h)"
    return mathematica_to_CNF(s, a)

def POPCNT5 (a):
    s="(!a||!b||!c||!d||!e||!f)&&(!a||!b||!c||!d||!e||!g)&&(!a||!b||!c||!d||!e||!h)&&" \
      "(!a||!b||!c||!d||!f||!g)&&(!a||!b||!c||!d||!f||!h)&&(!a||!b||!c||!d||!g||!h)&&" \
      "(!a||!b||!c||!e||!f||!g)&&(!a||!b||!c||!e||!f||!h)&&(!a||!b||!c||!e||!g||!h)&&" \
      "(!a||!b||!c||!f||!g||!h)&&(!a||!b||!d||!e||!f||!g)&&(!a||!b||!d||!e||!f||!h)&&" \
      "(!a||!b||!d||!e||!g||!h)&&(!a||!b||!d||!f||!g||!h)&&(!a||!b||!e||!f||!g||!h)&&" \
      "(!a||!c||!d||!e||!f||!g)&&(!a||!c||!d||!e||!f||!h)&&(!a||!c||!d||!e||!g||!h)&&" \
      "(!a||!c||!d||!f||!g||!h)&&(!a||!c||!e||!f||!g||!h)&&(!a||!d||!e||!f||!g||!h)&&" \
      "(a||b||c||d)&&(a||b||c||e)&&(a||b||c||f)&&(a||b||c||g)&&(a||b||c||h)&&(a||b||d||e
        )&&" \
      "(a||b||d||f)&&(a||b||d||g)&&(a||b||d||h)&&(a||b||e||f)&&(a||b||e||g)&&(a||b||e||h
        )&&" \
      "(a||b||f||g)&&(a||b||f||h)&&(a||b||g||h)&&(a||c||d||e)&&(a||c||d||f)&&(a||c||d||g
        )&&" \
      "(a||c||d||h)&&(a||c||e||f)&&(a||c||e||g)&&(a||c||e||h)&&(a||c||f||g)&&(a||c||f||h
        )&&" \
      "(a||c||g||h)&&(a||d||e||f)&&(a||d||e||g)&&(a||d||e||h)&&(a||d||f||g)&&(a||d||f||h
        )&&" \
      "(a||d||g||h)&&(a||e||f||g)&&(a||e||f||h)&&(a||e||g||h)&&(a||f||g||h)&&(!b||!c||!d
        ||!e||!f||!g)&&" \
      "(!b||!c||!d||!e||!f||!h)&&(!b||!c||!d||!e||!g||!h)&&(!b||!c||!d||!f||!g||!h)&&" \
      "(!b||!c||!e||!f||!g||!h)&&(!b||!d||!e||!f||!g||!h)&&(b||c||d||e)&&(b||c||d||f)&&"
         \
      "(b||c||d||g)&&(b||c||d||h)&&(b||c||e||f)&&(b||c||e||g)&&(b||c||e||h)&&(b||c||f||g
        )&&" \
      "(b||c||f||h)&&(b||c||g||h)&&(b||d||e||f)&&(b||d||e||g)&&(b||d||e||h)&&(b||d||f||g
        )&&" \
      "(b||d||f||h)&&(b||d||g||h)&&(b||e||f||g)&&(b||e||f||h)&&(b||e||g||h)&&(b||f||g||h
        )&&" \
      "(!c||!d||!e||!f||!g||!h)&&(c||d||e||f)&&(c||d||e||g)&&(c||d||e||h)&&(c||d||f||g)
        &&" \
      "(c||d||f||h)&&(c||d||g||h)&&(c||e||f||g)&&(c||e||f||h)&&(c||e||g||h)&&(c||f||g||h
        )&&" \
      "(d||e||f||g)&&(d||e||f||h)&&(d||e||g||h)&&(d||f||g||h)&&(e||f||g||h)"
```

```
    return mathematica_to_CNF(s, a)

def POPCNT6 (a):
    s="(!a||!b||!c||!d||!e||!f||!g)&&(!a||!b||!c||!d||!e||!f||!h)&&(!a||!b||!c||!d||!e||!g||!h)&&" \
      "(!a||!b||!c||!d||!f||!g||!h)&&(!a||!b||!c||!e||!f||!g||!h)&&(!a||!b||!d||!e||!f||!g||!h)&&" \
      "(!a||!c||!d||!e||!f||!g||!h)&&(a||b||c)&&(a||b||d)&&(a||b||e)&&(a||b||f)&&(a||b||g)&&(a||b||h)&&" \
      "(a||c||d)&&(a||c||e)&&(a||c||f)&&(a||c||g)&&(a||c||h)&&(a||d||e)&&(a||d||f)&&(a||d||g)&&" \
      "(a||d||h)&&(a||e||f)&&(a||e||g)&&(a||e||h)&&(a||f||g)&&(a||f||h)&&(a||g||h)&&" \
      "(!b||!c||!d||!e||!f||!g||!h)&&(b||c||d)&&(b||c||e)&&(b||c||f)&&(b||c||g)&&(b||c||h)&&(b||d||e)&&" \
      "(b||d||f)&&(b||d||g)&&(b||d||h)&&(b||e||f)&&(b||e||g)&&(b||e||h)&&(b||f||g)&&(b||f||h)&&(b||g||h)&&" \
      "(c||d||e)&&(c||d||f)&&(c||d||g)&&(c||d||h)&&(c||e||f)&&(c||e||g)&&(c||e||h)&&(c||f||g)&&(c||f||h)&&" \
      "(c||g||h)&&(d||e||f)&&(d||e||g)&&(d||e||h)&&(d||f||g)&&(d||f||h)&&(d||g||h)&&" \
      "(e||f||g)&&(e||f||h)&&(e||g||h)&&(f||g||h)"
    return mathematica_to_CNF(s, a)

def POPCNT7 (a):
    s="(!a||!b||!c||!d||!e||!f||!g||!h)&&(a||b)&&(a||c)&&(a||d)&&(a||e)&&(a||f)&&(a||g)&&(a||h)&&(b||c)&&" \
      "(b||d)&&(b||e)&&(b||f)&&(b||g)&&(b||h)&&(c||d)&&(c||e)&&(c||f)&&(c||g)&&(c||h)&&(d||e)&&(d||f)&&(d||g)&&" \
      "(d||h)&&(e||f)&&(e||g)&&(e||h)&&(f||g)&&(f||h)&&(g||h)"
    return mathematica_to_CNF(s, a)

def POPCNT8 (a):
    s="a&&b&&c&&d&&e&&f&&g&&h"
    return mathematica_to_CNF(s, a)

POPCNT_functions=[POPCNT0, POPCNT1, POPCNT2, POPCNT3, POPCNT4, POPCNT5, POPCNT6, POPCNT7, POPCNT8]

def coords_to_var (row, col):
    # we always use SAT variables as strings, anyway.
    # the 1st variables is 1, not 0
    return str(row*(WIDTH+2)+col+1)

def chk_bomb(row, col):
    clauses=[]

    # make empty border
    # all variables are negated (because they must be False)
    for c in range(WIDTH+2):
        clauses.append ("-"+coords_to_var(0,c))
        clauses.append ("-"+coords_to_var(HEIGHT+1,c))
    for r in range(HEIGHT+2):
        clauses.append ("-"+coords_to_var(r,0))
        clauses.append ("-"+coords_to_var(r,WIDTH+1))

    for r in range(1,HEIGHT+1):
        for c in range(1,WIDTH+1):
```

```
            t=known[r-1][c-1]
            if t in "012345678":
                # cell at r, c is empty (False):
                clauses.append ("-"+coords_to_var(r,c))
                # we need an empty border so the following expression would work for all
                    possible cells:
                neighbours=[coords_to_var(r-1, c-1), coords_to_var(r-1, c),
                    coords_to_var(r-1, c+1), coords_to_var(r, c-1),
                        coords_to_var(r, c+1), coords_to_var(r+1, c-1), coords_to_var(r
                            +1, c), coords_to_var(r+1, c+1)]
                clauses=clauses+POPCNT_functions[int(t)](neighbours)

    # place a bomb
    clauses.append (coords_to_var(row,col))

    f=open("tmp.cnf", "w")
    f.write ("p cnf "+str(VARS_TOTAL)+" "+str(len(clauses))+"\n")
    for c in clauses:
        f.write(c+" 0\n")
    f.close()

    child = subprocess.Popen(["minisat", "tmp.cnf"], stdout=subprocess.PIPE)
    child.wait()
    # 10 is SAT, 20 is UNSAT
    if child.returncode==20:
        print "row=%d, col=%d, unsat!" % (row, col)

for r in range(1,HEIGHT+1):
    for c in range(1,WIDTH+1):
        if known[r-1][c-1]=="?":
            chk_bomb(r, c)
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/minesweeper_SAT/minesweeper_SAT.py )

The output CNF file can be large, up to ≈ 2000 clauses, or more, here is an example: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/minesweeper_SAT/sample.cnf.

Anyway, it works just like my previous Z3Py script:

```
row=1, col=3, unsat!
row=6, col=2, unsat!
row=6, col=3, unsat!
row=7, col=4, unsat!
row=7, col=9, unsat!
row=8, col=9, unsat!
```

…but it runs way faster, even considering overhead of executing external program. Perhaps, Z3Py version could be optimized much better?

The files, including Wolfram Mathematica notebook: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/equations/minesweeper_SAT.

## 3.8 Cracking LCG with Z3

There are well-known weaknesses of LCG [50], but let's see, if it would be possible to crack it straightforwardly, without any special knowledge. We will define all relations between LCG states in terms of Z3. Here is a test progam:

[50] http://en.wikipedia.org/wiki/Linear_congruential_generator#Advantages_and_disadvantages_of_LCGs, http://www.reteam.org/papers/e59.pdf, http://stackoverflow.com/questions/8569113/why-1103515245-is-used-in-rand/8574774#8574774

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main()
{
        int i;

        srand(time(NULL));

        for (i=0; i<10; i++)
                printf ("%d\n", rand()%100);
};
```

It is printing 10 pseudorandom numbers in 0..99 range:

```
37
29
74
95
98
40
23
58
61
17
```

Let's say we are observing only 8 of these numbers (from 29 to 61) and we need to predict next one (17) and/or previous one (37).

The program is compiled using MSVC 2013 (I choose it because its LCG is simpler than that in Glib):

```
.text:0040112E rand             proc near
.text:0040112E                  call    __getptd
.text:00401133                  imul    ecx, [eax+0x14], 214013
.text:0040113A                  add     ecx, 2531011
.text:00401140                  mov     [eax+14h], ecx
.text:00401143                  shr     ecx, 16
.text:00401146                  and     ecx, 7FFFh
.text:0040114C                  mov     eax, ecx
.text:0040114E                  retn
.text:0040114E rand             endp
```

Let's define LCG in Z3Py:

```
#!/usr/bin/python
from z3 import *

output_prev = BitVec('output_prev', 32)
state1 = BitVec('state1', 32)
state2 = BitVec('state2', 32)
state3 = BitVec('state3', 32)
state4 = BitVec('state4', 32)
state5 = BitVec('state5', 32)
state6 = BitVec('state6', 32)
state7 = BitVec('state7', 32)
state8 = BitVec('state8', 32)
state9 = BitVec('state9', 32)
state10 = BitVec('state10', 32)
```

```
output_next = BitVec('output_next', 32)

s = Solver()

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)
s.add(state4 == state3*214013+2531011)
s.add(state5 == state4*214013+2531011)
s.add(state6 == state5*214013+2531011)
s.add(state7 == state6*214013+2531011)
s.add(state8 == state7*214013+2531011)
s.add(state9 == state8*214013+2531011)
s.add(state10 == state9*214013+2531011)

s.add(output_prev==URem((state1>>16)&0x7FFF,100))
s.add(URem((state2>>16)&0x7FFF,100)==29)
s.add(URem((state3>>16)&0x7FFF,100)==74)
s.add(URem((state4>>16)&0x7FFF,100)==95)
s.add(URem((state5>>16)&0x7FFF,100)==98)
s.add(URem((state6>>16)&0x7FFF,100)==40)
s.add(URem((state7>>16)&0x7FFF,100)==23)
s.add(URem((state8>>16)&0x7FFF,100)==58)
s.add(URem((state9>>16)&0x7FFF,100)==61)
s.add(output_next==URem((state10>>16)&0x7FFF,100))

print(s.check())
print(s.model())
```

*URem* states for *unsigned remainder*. It works for some time and gave us correct result!

```
sat
[state3 = 2276903645,
 state4 = 1467740716,
 state5 = 3163191359,
 state7 = 4108542129,
 state8 = 2839445680,
 state2 = 998088354,
 state6 = 4214551046,
 state1 = 1791599627,
 state9 = 548002995,
 output_next = 17,
 output_prev = 37,
 state10 = 1390515370]
```

I added $\approx 10$ states to be sure result will be correct. It may be not in case of smaller set of information.

That is the reason why LCG is not suitable for any security-related task. This is why cryptographically secure pseudorandom number generators exist: they are designed to be protected against such simple attack. Well, at least if NSA[51] don't get involved [52].

Security tokens like "RSA SecurID" can be viewed just as CPRNG[53] with a secret seed. It shows new pseudorandom number each minute, and the server can predict it, because it knows the seed. Imagine if such token would implement LCG—it would be much easier to break!

[51] National Security Agency
[52] https://en.wikipedia.org/wiki/Dual_EC_DRBG
[53] Cryptographically Secure Pseudorandom Number Generator

## 3.9 Can rand() generate 10 consecutive zeroes?

I've always been wondering, if it's possible or not. As of simplest linear congruential generator from MSVC's rand(), I could get a state at which rand() will output 8 zeroes modulo 10:

```
#!/usr/bin/python
from z3 import *

state1 = BitVec('state1', 32)
state2 = BitVec('state2', 32)
state3 = BitVec('state3', 32)
state4 = BitVec('state4', 32)
state5 = BitVec('state5', 32)
state6 = BitVec('state6', 32)
state7 = BitVec('state7', 32)
state8 = BitVec('state8', 32)
state9 = BitVec('state9', 32)

s = Solver()

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)
s.add(state4 == state3*214013+2531011)
s.add(state5 == state4*214013+2531011)
s.add(state6 == state5*214013+2531011)
s.add(state7 == state6*214013+2531011)
s.add(state8 == state7*214013+2531011)
s.add(state9 == state8*214013+2531011)

s.add(URem((state2>>16)&0x7FFF,10)==0)
s.add(URem((state3>>16)&0x7FFF,10)==0)
s.add(URem((state4>>16)&0x7FFF,10)==0)
s.add(URem((state5>>16)&0x7FFF,10)==0)
s.add(URem((state6>>16)&0x7FFF,10)==0)
s.add(URem((state7>>16)&0x7FFF,10)==0)
s.add(URem((state8>>16)&0x7FFF,10)==0)
s.add(URem((state9>>16)&0x7FFF,10)==0)

print(s.check())
print(s.model())
```

```
sat
[state3 = 1181667981,
 state4 = 342792988,
 state5 = 4116856175,
 state7 = 1741999969,
 state8 = 3185636512,
 state2 = 1478548498,
 state6 = 4036911734,
 state1 = 286227003,
 state9 = 1700675811]
```

This is a case if, in some video game, you'll find a code:

```
for (int i=0; i<8; i++)
    printf ("%d\n", rand() % 10);
```

... and at some point, this piece of code can generate 8 zeroes in row, if the state will be 286227003 (decimal).

Just checked this piece of code in MSVC 2015:

```
// MSVC 2015 x86

#include <stdio.h>

int main()
{
        srand(286227003);

        for (int i=0; i<8; i++)
                printf ("%d\n", rand() % 10);
};
```

Yes, its output is 8 zeroes!
What about other modulos?
I can get 4 consecutive zeroes modulo 100:

```
#!/usr/bin/python
from z3 import *

state1 = BitVec('state1', 32)
state2 = BitVec('state2', 32)
state3 = BitVec('state3', 32)
state4 = BitVec('state4', 32)
state5 = BitVec('state5', 32)

s = Solver()

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)
s.add(state4 == state3*214013+2531011)
s.add(state5 == state4*214013+2531011)

s.add(URem((state2>>16)&0x7FFF,100)==0)
s.add(URem((state3>>16)&0x7FFF,100)==0)
s.add(URem((state4>>16)&0x7FFF,100)==0)
s.add(URem((state5>>16)&0x7FFF,100)==0)

print(s.check())
print(s.model())
```

```
sat
[state3 = 635704497,
 state4 = 1644979376,
 state2 = 1055176198,
 state1 = 3865742399,
 state5 = 1389375667]
```

However, 4 consecutive zeroes modulo 100 is impossible (given these constants at least), this gives "unsat": https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/LCG/LCG100_v1.py.
... and 3 consecutive zeroes modulo 1000:

```
#!/usr/bin/python
from z3 import *

state1 = BitVec('state1', 32)
state2 = BitVec('state2', 32)
```

```
state3 = BitVec('state3', 32)
state4 = BitVec('state4', 32)

s = Solver()

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)
s.add(state4 == state3*214013+2531011)

s.add(URem((state2>>16)&0x7FFF,1000)==0)
s.add(URem((state3>>16)&0x7FFF,1000)==0)
s.add(URem((state4>>16)&0x7FFF,1000)==0)

print(s.check())
print(s.model())
```

```
sat
[state3 = 1179663182,
 state2 = 720934183,
 state1 = 4090229556,
 state4 = 786474201]
```

What if we could use rand()'s output without division? Which is in 0..0x7fff range (i.e., 15 bits)? As it can be checked quickly, 2 zeroes at output is possible:

```
#!/usr/bin/python
from z3 import *

state1 = BitVec('state1', 32)
state2 = BitVec('state2', 32)
state3 = BitVec('state3', 32)

s = Solver()

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)

s.add((state2>>16)&0x7FFF==0)
s.add((state3>>16)&0x7FFF==0)

print(s.check())
print(s.model())
```

```
sat
[state2 = 20057, state1 = 3385131726, state3 = 22456]
```

### 3.9.1 UNIX time and srand(time(NULL))

Given the fact that it's highly popular to initialize LCG PRNG with UNIX time (i.e., `srand(time(NULL))`), you can probably calculate a moment in time so that LCG PRNG will be initialized as you want to.

For example, can we get a moment in time from now (5-Dec-2017) till 12-Dec-2017 (that is one week from now), when, if initialized by UNIX time, rand() will output as many similar numbers (modulo 10), as possible?

```
#!/usr/bin/python
from z3 import *

state1 = BitVec('state1', 32)
```

```
state2 = BitVec('state2', 32)
state3 = BitVec('state3', 32)
state4 = BitVec('state4', 32)
state5 = BitVec('state5', 32)
state6 = BitVec('state6', 32)
state7 = BitVec('state7', 32)

s = Solver()

s.add(state1>=1512499124) # Tue Dec  5 20:38:44 EET 2017
s.add(state1<=1513036800) # Tue Dec 12 02:00:00 EET 2017

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)
s.add(state4 == state3*214013+2531011)
s.add(state5 == state4*214013+2531011)
s.add(state6 == state5*214013+2531011)
s.add(state7 == state6*214013+2531011)

c = BitVec('c', 32)

s.add(URem((state2>>16)&0x7FFF,10)==c)
s.add(URem((state3>>16)&0x7FFF,10)==c)
s.add(URem((state4>>16)&0x7FFF,10)==c)
s.add(URem((state5>>16)&0x7FFF,10)==c)
s.add(URem((state6>>16)&0x7FFF,10)==c)
s.add(URem((state7>>16)&0x7FFF,10)==c)

print(s.check())
print(s.model())
```

Yes:

```
sat
[state3 = 2234253076,
 state4 = 497021319,
 state5 = 4160988718,
 c = 3,
 state2 = 333151205,
 state6 = 46785593,
 state1 = 1512500810,
 state7 = 1158878744]
```

If `srand(time(NULL))` will be executed at `Tue Dec 5 21:06:50 EET 2017` (this precise second, UNIX time=1512500810), a next 6 `rand() % 10` lines will output six numbers of 3 in a row. Don't know if it useful or not, but you've got the idea.

### 3.9.2 etc:

The files: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/equations/LCG.

Further work: check glibc's `rand()`, Mersenne Twister, etc. Simple 32-bit LCG as described can be checked using simple brute-force, I think.

## 3.10 Integer factorization using Z3 SMT solver

Integer factorization is method of breaking a composite (non-prime number) into prime factors. Like 12345 = 3*4*823.

Though for small numbers, this task can be accomplished by Z3:

```
#!/usr/bin/env python
```

```
import random
from z3 import *
from operator import mul

def factor(n):
    print "factoring",n

    in1,in2,out=Ints('in1 in2 out')

    s=Solver()
    s.add(out==n)
    s.add(in1*in2==out)
    # inputs cannot be negative and must be non-1:
    s.add(in1>1)
    s.add(in2>1)

    if s.check()==unsat:
        print n,"is prime (unsat)"
        return [n]
    if s.check()==unknown:
        print n,"is probably prime (unknown)"
        return [n]

    m=s.model()
    # get inputs of multiplier:
    in1_n=m[in1].as_long()
    in2_n=m[in2].as_long()

    print "factors of", n, "are", in1_n, "and", in2_n
    # factor factors recursively:
    rt=sorted(factor (in1_n) + factor (in2_n))
    # self-test:
    assert reduce(mul, rt, 1)==n
    return rt

# infinite test:
def test():
    while True:
        print factor (random.randrange(1000000000))

#test()

print factor(1234567890)
```

( The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/factor_SMT/factor_z3.py )

When factoring 1234567890 recursively:

```
% time python z.py
factoring 1234567890
factors of 1234567890 are 342270 and 3607
factoring 342270
factors of 342270 are 2 and 171135
factoring 2
2 is prime (unsat)
factoring 171135
```

```
factors of 171135 are 3803 and 45
factoring 3803
3803 is prime (unsat)
factoring 45
factors of 45 are 3 and 15
factoring 3
3 is prime (unsat)
factoring 15
factors of 15 are 5 and 3
factoring 5
5 is prime (unsat)
factoring 3
3 is prime (unsat)
factoring 3607
3607 is prime (unsat)
[2, 3, 3, 5, 3607, 3803]
python z.py  19.30s user 0.02s system 99% cpu 19.443 total
```

So, 1234567890 = 2*3*3*5*3607*3803.

One important note: there is no primality test, no lookup tables, etc. Prime number is a number for which "x*y=prime" (where x>1 and y>1) diophantine equation (which allows only integers in solution) has no solutions. It can be solved for real numbers, though.

Z3 is not yet good enough for non-linear integer arithmetic and sometimes returns "unknown" instead of "unsat", but, as Leonardo de Moura (one of Z3's author) commented about this:

```
...Z3 will solve the problem as a real problem. If no real solution is found, we know
   there is no integer solution.
If a solution is found, Z3 will check if the solution is really assigning integer values
    to integer variables.
If that is not the case, it will return unknown to indicate it failed to solve the
   problem.
```

( https://stackoverflow.com/questions/13898175/how-does-z3-handle-non-linear-integer-arithmetic )

Probably, this is the case: we getting "unknown" in the case when a number cannot be factored, i.e., it's prime.

It's also very slow. Wolfram Mathematica can factor number around $2^{80}$ in a matter of seconds. Still, I've written this for demonstration.

The problem of breaking RSA is a problem of factorization of very large numbers, up to $2^{4096}$. It's currently not possible to do this in practice.

## 3.11 Integer factorization using SAT solver

See also: integer factorization using Z3 SMT solver (3.10).

We are going to simulate electronic circuit of binary multiplier in SAT and then ask solver, what multiplier's inputs must be so the output will be a desired number? If this situation is impossible, the desired number is prime.

First we should build multiplier out of adders.

### 3.11.1 Binary adder in SAT

Simple binary adder usually constists of full-adders and one half-adder. These are basic elements of adders.

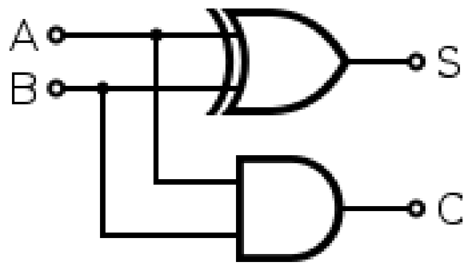



Figure 4: Half-adder

( The image has been taken from Wikipedia. )
Full-adder:

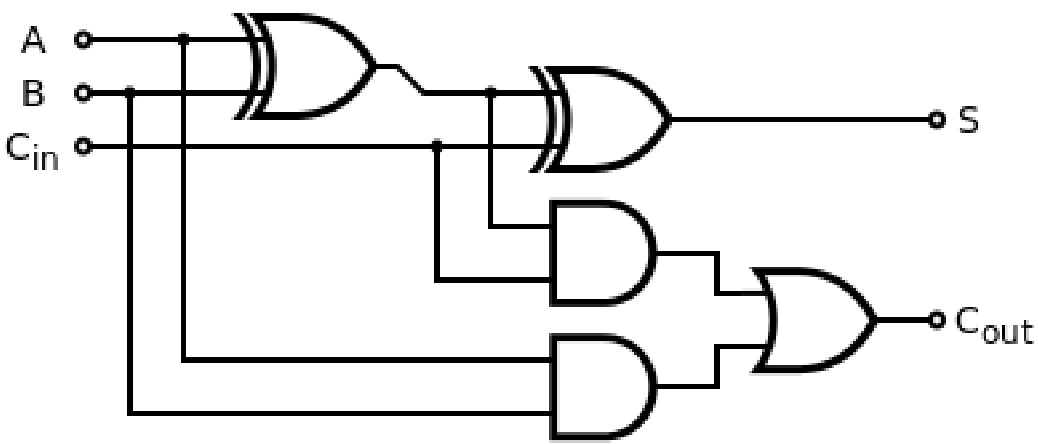



Figure 5: Full-adder

( The image has been taken from Wikipedia. )
Here is a 4-bit ripple-carry adder:

```
          X3  Y3                  X2  Y2                  X1  Y1                  X0  Y0
          |   |                   |   |                   |   |                   |   |
          v   v                   v   v                   v   v                   v   v
         +-----+                 +-----+                 +-----+                 +-----+
Cout  <- | FA  | <- carry  <-    | FA  | <- carry  <-    | FA  | <- carry  <-    | HA  |
         +-----+                 +-----+                 +-----+                 +-----+
            ^                       ^                       ^                       ^
            |                       |                       |                       |
           S3                      S2                      S1                      S0
```

It can be used for most tasks.
Here is a 4-bit ripple-carry adder with carry-in:

```
          X3  Y3                  X2  Y2                  X1  Y1                  X0  Y0
          |   |                   |   |                   |   |                   |   |
          v   v                   v   v                   v   v                   v   v
         +-----+                 +-----+                 +-----+                 +-----+
Cout  <- | FA  | <- carry  <-    | FA  | <- carry  <-    | FA  | <- carry  <-    | FA  | <- Cin
         +-----+                 +-----+                 +-----+                 +-----+
            ^                       ^                       ^                       ^
            |                       |                       |                       |
           S3                      S2                      S1                      S0
```

What carries are? 4-bit adder can sum up two numbers up to 0b1111 (15). 15+15=30 and this is 0b11110, i.e., 5 bits. Lowest 4 bits is a sum. 5th most significant bit is not a part of sum, but is a carry bit.

If you sum two numbers on x86 CPU, CF flag is a carry bit connected to ALU[54]. It is set if a resulting sum is bigger than it can be fit into result.

Now you can also need carry-in. Again, x86 CPU has ADC instruction, it takes CF flag state. It can be said, CF flag is connected to adder's carry-in input. Hence, combining two ADD and ADC instructions you can sum up 128 bits on 64-bit CPU.

By the way, this is a good explanation of "carry-ripple". The very first full-adder's result is depending on the carry-out of the previous full-adder. Hence, adders cannot work in parallel. This is a problem of simplest possible adder, other adders can solve this.

To represent full-adders in CNF form, we can use Wolfram Mathematica. I've taken truth table for full-adder from Wikipedia:

```
Inputs   |  Outputs
--------+----------
A B Cin |  Cout Sum
0 0 0   |  0    0
0 0 1   |  0    1
0 1 0   |  0    1
0 1 1   |  1    0
1 0 0   |  0    1
1 0 1   |  1    0
1 1 0   |  1    0
1 1 1   |  1    1
```

In Mathematica, I'm setting "->1" if row is correct and "->0" if not correct.

```
In[59]:= FaTbl = {{0, 0, 0, 0, 0} -> 1, {0, 0, 0, 0, 1} ->
   0, {0, 0, 0, 1, 0} -> 0, {0, 0, 0, 1, 1} -> 0, {0, 0, 1, 0, 0} ->
   0, {0, 0, 1, 0, 1} -> 1, {0, 0, 1, 1, 0} -> 0, {0, 0, 1, 1, 1} ->
   0, {0, 1, 0, 0, 0} -> 0, {0, 1, 0, 0, 1} -> 1, {0, 1, 0, 1, 0} ->
   0, {0, 1, 0, 1, 1} -> 0, {0, 1, 1, 0, 0} -> 0, {0, 1, 1, 0, 1} ->
   0, {0, 1, 1, 1, 0} -> 1, {0, 1, 1, 1, 1} -> 0, {1, 0, 0, 0, 0} ->
   0, {1, 0, 0, 0, 1} -> 1, {1, 0, 0, 1, 0} -> 0, {1, 0, 0, 1, 1} ->
   0, {1, 0, 1, 0, 0} -> 0, {1, 0, 1, 0, 1} -> 0, {1, 0, 1, 1, 0} ->
   1, {1, 0, 1, 1, 1} -> 0, {1, 1, 0, 0, 0} -> 0, {1, 1, 0, 0, 1} ->
   0, {1, 1, 0, 1, 0} -> 1, {1, 1, 0, 1, 1} -> 0, {1, 1, 1, 0, 0} ->
   0, {1, 1, 1, 0, 1} -> 0, {1, 1, 1, 1, 0} -> 0, {1, 1, 1, 1, 1} -> 1}

...

In[60]:= BooleanConvert[
 BooleanFunction[FaTbl, {a, b, cin, cout, s}], "CNF"]

Out[60]= (! a || ! b || ! cin || s) && (! a || ! b ||
   cout) && (! a || ! cin || cout) && (! a || cout || s) && (a || b ||
    cin || ! s) && (a || b || ! cout) && (a ||
   cin || ! cout) && (a || ! cout || ! s) && (! b || ! cin ||
   cout) && (! b || cout || s) && (b ||
   cin || ! cout) && (b || ! cout || ! s) && (! cin || cout ||
   s) && (cin || ! cout || ! s)
```

These clauses can be used as full-adder.

Here is it:

```
    # full-adder, as found by Mathematica using truth table:
    def FA (self, a,b,cin):
```

[54]Arithmetic logic unit

```
        s=self.create_var()
        cout=self.create_var()

        self.add_clause([self.neg(a), self.neg(b), self.neg(cin), s])
        self.add_clause([self.neg(a), self.neg(b), cout])
        self.add_clause([self.neg(a), self.neg(cin), cout])
        self.add_clause([self.neg(a), cout, s])
        self.add_clause([a, b, cin, self.neg(s)])
        self.add_clause([a, b, self.neg(cout)])
        self.add_clause([a, cin, self.neg(cout)])
        self.add_clause([a, self.neg(cout), self.neg(s)])
        self.add_clause([self.neg(b), self.neg(cin), cout])
        self.add_clause([self.neg(b), cout, s])
        self.add_clause([b, cin, self.neg(cout)])
        self.add_clause([b, self.neg(cout), self.neg(s)])
        self.add_clause([self.neg(cin), cout, s])
        self.add_clause([cin, self.neg(cout), self.neg(s)])

        return s, cout
```

And the adder:

```
    # bit order: [MSB..LSB]
    # n-bit adder:
    def adder(self, X,Y):
        assert len(X)==len(Y)
        # first full-adder could be half-adder
        # start with lowest bits:
        inputs=my_utils.rvr(list(zip(X,Y)))
        carry=self.const_false
        sums=[]
        for pair in inputs:
            # "carry" variable is replaced at each iteration.
            # so it is used in the each FA() call from the previous FA() call.
            s, carry = self.FA(pair[0], pair[1], carry)
            sums.append(s)
        return my_utils.rvr(sums), carry
```

### 3.11.2 Binary multiplier in SAT

Remember school-level long division? This multiplier works in a same way, but for binary digits.

Here is example of multiplying 0b1101 (X) by 0b0111 (Y):

```
         LSB
          |
          v
       1101 <- X
       -------
LSB 0|    0000
    1|   1101
    1|  1101
    1| 1101
    ^
    |
    Y
```

If bit from Y is zero, a row is zero. If bit from Y is non-zero, a row is equal to X, but shifted each time. Then you just sum up all rows (which are called "partial products".)

This is 4-bit binary multiplier. It takes 4-bit inputs and produces 8-bit output:

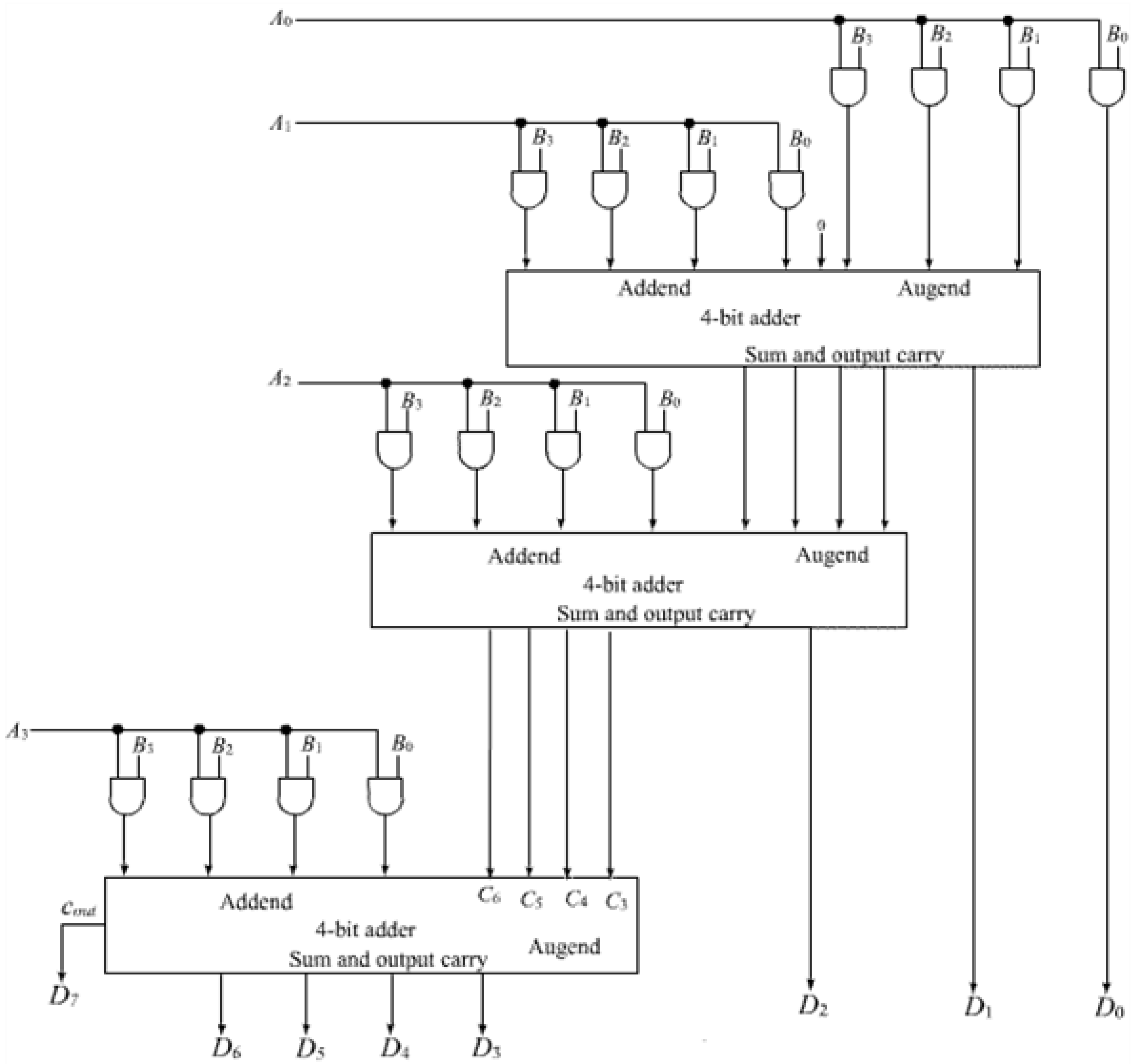



Figure 6: 4-bit binary multiplier

( The image has been taken from http://www.chegg.com/homework-help/binary-multiplier-multiplies-two-unsigned-fo
)

I would build separate block, "multiply by one bit" as a latch for each partial product:

```
    def AND_Tseitin(self, v1, v2, out):
        self.add_clause([self.neg(v1), self.neg(v2), out])
        self.add_clause([v1, self.neg(out)])
        self.add_clause([v2, self.neg(out)])

    def AND(self,v1, v2):
        out=self.create_var()
        self.AND_Tseitin(v1, v2, out)
        return out

...

    # bit is 0 or 1.
```

```
    # i.e., if it's 0, output is 0 (all bits)
    # if it's 1, output=input
    def mult_by_bit(self, X, bit):
        return [self.AND(i, bit) for i in X]

    # bit order: [MSB..LSB]
    # build multiplier using adders and mult_by_bit blocks:
    def multiplier(self, X, Y):
        assert len(X)==len(Y)
        out=[]
        #initial:
        prev=[self.const_false]*len(X)
        # first adder can be skipped, but I left thing "as is" to make it simpler
        for Y_bit in my_utils.rvr(Y):
            s, carry = self.adder(self.mult_by_bit(X, Y_bit), prev)
            out.append(s[-1])
            prev=[carry] + s[:-1]

        return prev + my_utils.rvr(out)
```

AND gate is constructed here using Tseitin transformations. This is quite popular way of encoding gates in CNF form, by adding additional variable: https://en.wikipedia.org/wiki/Tseytin_transformation. In fact, full-adder can be constructed without Mathematica, using logic gates, and encoded by Tseitin transformation.

### 3.11.3 Glueing all together

```
#!/usr/bin/env python3

import itertools, subprocess, os, math, random
from operator import mul
import my_utils, SAT_lib
import functools

def factor(n):
    print ("factoring %d" % n)

    # size of inputs.
    # in other words, how many bits we have to allocate to store 'n'?
    input_bits=int(math.ceil(math.log(n,2)))
    print ("input_bits=%d" % input_bits)

    s=SAT_lib.SAT_lib(False)

    factor1,factor2=s.alloc_BV(input_bits),s.alloc_BV(input_bits)
    product=s.multiplier(factor1,factor2)

    # at least one bit in each input must be set, except lowest.
    # hence we restrict inputs to be greater than 1
    s.fix(s.OR_list(factor1[:-1]), True)
    s.fix(s.OR_list(factor2[:-1]), True)

    # output has a size twice as bigger as each input:
    s.fix_BV(product, SAT_lib.n_to_BV(n,input_bits*2))

    if s.solve()==False:
        print ("%d is prime (unsat)" % n)
        return [n]
```

```
    # get inputs of multiplier:
    factor1_n=SAT_lib.BV_to_number(s.get_BV_from_solution(factor1))
    factor2_n=SAT_lib.BV_to_number(s.get_BV_from_solution(factor2))

    print ("factors of %d are %d and %d" % (n, factor1_n, factor2_n))
    # factor factors recursively:
    rt=sorted(factor (factor1_n) + factor (factor2_n))
    assert functools.reduce(mul, rt, 1)==n
    return rt

# infinite test:
def test():
    while True:
        print (factor (random.randrange(100000000000)))

#test()

print (factor(1234567890))
```

I just connect our number to output of multiplier and ask SAT solver to find inputs. If it's UNSAT, this is prime number. Then we factor factors recursively.

Also, we want block input factors of 1, because obviously, we do not interesting in the fact that n*1=n. I'm using wide OR gates for this.

Output:

```
 % python factor_SAT.py
factoring 1234567890
input_bits=31
factors of 1234567890 are 2 and 617283945
factoring 2
input_bits=1
2 is prime (unsat)
factoring 617283945
input_bits=30
factors of 617283945 are 3 and 205761315
factoring 3
input_bits=2
3 is prime (unsat)
factoring 205761315
input_bits=28
factors of 205761315 are 3 and 68587105
factoring 3
input_bits=2
3 is prime (unsat)
factoring 68587105
input_bits=27
factors of 68587105 are 5 and 13717421
factoring 5
input_bits=3
5 is prime (unsat)
factoring 13717421
input_bits=24
factors of 13717421 are 3607 and 3803
factoring 3607
input_bits=12
3607 is prime (unsat)
```

```
factoring 3803
input_bits=12
3803 is prime (unsat)
[2, 3, 3, 5, 3607, 3803]
```

So, $1234567890 = 2 \cdot 3 \cdot 3 \cdot 5 \cdot 3607 \cdot 3803$.

It works way faster than by Z3 solution, but still slow. It can factor numbers up to maybe ~$2^{40}$, while Wolfram Mathematica can factor ~$2^{80}$ easily.

The full source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/factor_SAT/factor_SAT.py.

### 3.11.4 Division using multiplier

Hard to believe, but why we couldn't define one of factors and ask SAT solver to find another factor? Then it will divide numbers! But, unfortunately, this is somewhat impractical, since it will work only if remainder is zero:

```
#!/usr/bin/env python3

import itertools, subprocess, os, math, random
from operator import mul
import my_utils, SAT_lib

def div(dividend,divisor):

    # size of inputs.
    # in other words, how many bits we have to allocate to store 'n'?
    input_bits=int(math.ceil(math.log(dividend,2)))
    print ("input_bits=%d" % input_bits)

    s=SAT_lib.SAT_lib(False)

    factor1,factor2=s.alloc_BV(input_bits),s.alloc_BV(input_bits)
    product=s.multiplier(factor1,factor2)

    # connect divisor to one of multiplier's input:
    s.fix_BV(factor1, SAT_lib.n_to_BV(divisor,input_bits))
    # output has a size twice as bigger as each input.
    # connect dividend to multiplier's output:
    s.fix_BV(product, SAT_lib.n_to_BV(dividend,input_bits*2))

    if s.solve()==False:
        print ("remainder!=0 (unsat)")
        return None

    # get 2nd input of multiplier, which is quotient:
    return SAT_lib.BV_to_number(s.get_BV_from_solution(factor2))

print (div (1234567890123456789012345678 9*12345, 12345))
```

The full source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/factor_SAT/div.py.

It works very fast, but still, slower than conventional ways.

### 3.11.5 Breaking RSA

It's not a problem to build multiplier with 4096 bit inputs and 8192 output, but it will not work in practice. Still, you can break toy-level demonstrational RSA problems with key less than $2^{40}$ or something like that (or larger, using Wolfram Mathematica).

### 3.11.6 Further reading

1, 2, 3.

## 3.12 Recalculating micro-spreadsheet using Z3Py

There is a nice exercise[55]: write a program to recalculate micro-spreadsheet, like this one:

```
1       0       B0+B2           A0*B0*C0
123     10      12              11
667     A0+B1   (C1*A0)*122     A3+C2
```

The result must be:

```
1       0       135     82041
123     10      12      11
667     11      1342    83383
```

As it turns out, though overkill, this can be solved using MK85 with little effort:

```
#!/usr/bin/python

from MK85 import *
import sys, re

# MS Excel or LibreOffice style.
# first top-left cell is A0, not A1
def coord_to_name(R, C):
    return "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[R]+str(C)

# open file and parse it as list of lists:
f=open(sys.argv[1],"r")
# filter(None, ...) to remove empty sublists:
ar=filter(None, [item.rstrip().split() for item in f.readlines()])
f.close()

WIDTH=len(ar[0])
HEIGHT=len(ar)

s=MK85()

# cells{} is a dictionary with keys like "A0", "B9", etc:
cells={}
for R in range(HEIGHT):
    for C in range(WIDTH):
        name=coord_to_name(R, C)
        cells[name]=s.BitVec(name, 32)

cur_R=0
cur_C=0

for row in ar:
    for c in row:
        # string like "A0+B2" becomes "cells["A0"]+cells["B2"]":
        c=re.sub(r'([A-Z]{1}[0-9]+)', r'cells["\1"]', c)
        st="cells[\"%s\"]==%s" % (coord_to_name(cur_R, cur_C), c)
        # evaluate string. Z3Py expression is constructed at this step:
```

55 The blog post in Russian: http://thesz.livejournal.com/280784.html

```
        e=eval(st)
        # add constraint:
        s.add (e)
        cur_C=cur_C+1
    cur_R=cur_R+1
    cur_C=0

if s.check():
    m=s.model()
    for r in range(HEIGHT):
        for c in range(WIDTH):
            sys.stdout.write (str(m[coord_to_name(r, c)])+"\t")
        sys.stdout.write ("\n")
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/spreadsheet/spreadsheet_MK85.py )

All we do is just creating pack of variables for each cell, named A0, B1, etc, of integer type. All of them are stored in *cells[]* dictionary. Key is a string. Then we parse all the strings from cells, and add to list of constraints *A0=123* (in case of number in cell) or *A0=B1+C2* (in case of expression in cell). There is a slight preparation: string like *A0+B2* becomes *cells["A0"]+cells["B2"]*.

Then the string is evaluated using Python *eval()* method, which is highly dangerous [56]: imagine if end-user could add a string to cell other than expression? Nevertheless, it serves our purposes well, because this is a simplest way to pass a string with expression into MK85.

### 3.12.1 Z3

The source code almost the same:

```
#!/usr/bin/python

from z3 import *
import sys, re

# MS Excel or LibreOffice style.
# first top-left cell is A0, not A1
def coord_to_name(R, C):
    return "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[R]+str(C)

# open file and parse it as list of lists:
f=open(sys.argv[1],"r")
# filter(None, ...) to remove empty sublists:
ar=filter(None, [item.rstrip().split() for item in f.readlines()])
f.close()

WIDTH=len(ar[0])
HEIGHT=len(ar)

# cells{} is a dictionary with keys like "A0", "B9", etc:
cells={}
for R in range(HEIGHT):
    for C in range(WIDTH):
        name=coord_to_name(R, C)
        cells[name]=Int(name)

s=Solver()
```

[56] http://stackoverflow.com/questions/1832940/is-using-eval-in-python-a-bad-practice

```
cur_R=0
cur_C=0

for row in ar:
    for c in row:
        # string like "A0+B2" becomes "cells["A0"]+cells["B2"]":
        c=re.sub(r'([A-Z]{1}[0-9]+)', r'cells["\1"]', c)
        st="cells[\"%s\"]==%s" % (coord_to_name(cur_R, cur_C), c)
        # evaluate string. Z3Py expression is constructed at this step:
        e=eval(st)
        # add constraint:
        s.add (e)
        cur_C=cur_C+1
    cur_R=cur_R+1
    cur_C=0

result=str(s.check())
print result
if result=="sat":
    m=s.model()
    for r in range(HEIGHT):
        for c in range(WIDTH):
            sys.stdout.write (str(m[cells[coord_to_name(r, c)]])+"\t")
        sys.stdout.write ("\n")
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/spreadsheet/spreadsheet_Z3_1.py )

### 3.12.2 Unsat core

Now the problem: what if there is circular dependency? Like:

```
1       0       B0+B2           A0*B0
123     10      12              11
C1+123  C0*123  A0*122          A3+C2
```

Two first cells of the last row (C0 and C1) are linked to each other. Our program will just tells "unsat", meaning, it couldn't satisfy all constraints together. We can't use this as error message reported to end-user, because it's highly unfriendly.

However, we can fetch *unsat core*, i.e., list of variables which Z3 finds conflicting.

```
...
s=Solver()
s.set(unsat_core=True)
...
        # add constraint:
        s.assert_and_track(e, coord_to_name(cur_R, cur_C))
...
if result=="sat":
...
else:
    print s.unsat_core()
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/spreadsheet/spreadsheet_Z3_2.py )

We must explicitly turn on unsat core support and use *assert_and_track()* instead of *add()* method, because this feature slows down the whole process, and is turned off by default. That works:

```
 % python 2.py test_circular
```

```
unsat
[C0, C1]
```

Perhaps, these variables could be removed from the 2D array, marked as *unresolved* and the whole spreadsheet could be recalculated again.

### 3.12.3 Stress test

How to generate large random spreadsheet? What we can do. First, create random DAG[57], like this one:

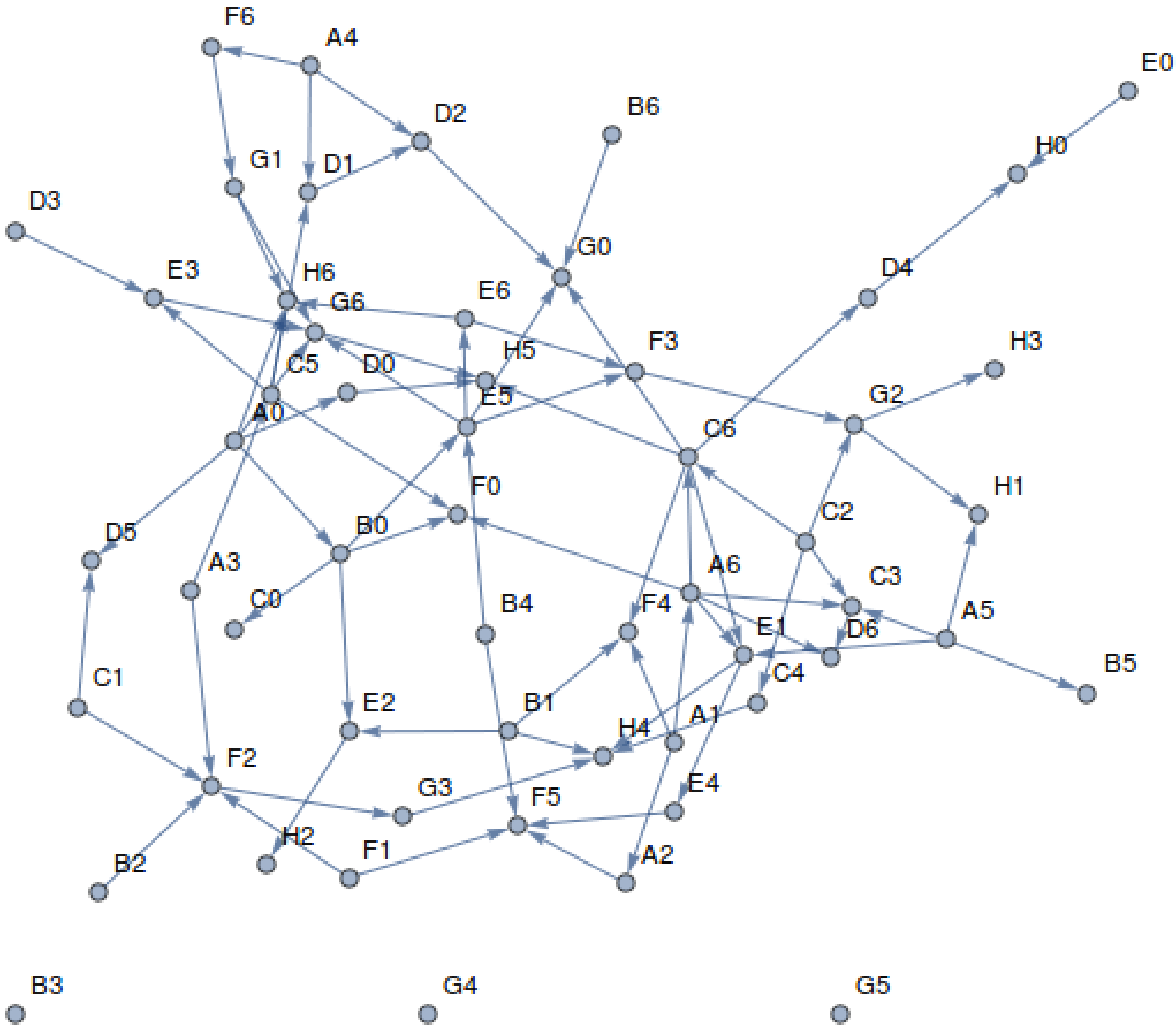



Figure 7: Random DAG

Arrows will represent information flow. So a vertex (node) which has no incoming arrows to it (indegree=0), can be set to a random number. Then we use topological sort to find dependencies between vertices. Then we assign spreadsheet cell names to each vertex. Then we generate random expression with random operations/numbers/cells to each cell, with the use of information from topological sorted graph.

[57] Directed acyclic graph

```
(* Utility functions *)
In[1]:= findSublistBeforeElementByValue[lst_,element_]:=lst[[ 1;;Position[lst, element
   ][[1]][[1]]-1]]

(* Input in ϖ1.. range. 1->A0, 2->A1, etc *)
In[2]:= vertexToName[x_,width_]:=StringJoin[FromCharacterCode[ToCharacterCode["A"][[1]]+
   Floor[(x-1)/width]],ToString[Mod[(x-1),width]]]

In[3]:= randomNumberAsString[]:=ToString[RandomInteger[{1,1000}]]

In[4]:= interleaveListWithRandomNumbersAsStrings[lst_]:=Riffle[lst,Table[
   randomNumberAsString[],Length[lst]-1]]

(* We omit division operation because micro-spreadsheet evaluator can't handle division
   by zero *)
In[5]:= interleaveListWithRandomOperationsAsStrings[lst_]:=Riffle[lst,Table[RandomChoice
   [{"+","-","*"}],Length[lst]-1]]

In[6]:= randomNonNumberExpression[g_,vertex_]:=StringJoin[
   interleaveListWithRandomOperationsAsStrings[interleaveListWithRandomNumbersAsStrings
   [Map[vertexToName[#,WIDTH]&,pickRandomNonDependentVertices[g,vertex]]]]]

In[7]:= pickRandomNonDependentVertices[g_,vertex_]:=DeleteDuplicates[RandomChoice[
   findSublistBeforeElementByValue[TopologicalSort[g],vertex],RandomInteger[{1,5}]]]

In[8]:= assignNumberOrExpr[g_,vertex_]:=If[VertexInDegree[g,vertex]==0,
   randomNumberAsString[],randomNonNumberExpression[g,vertex]]

(* Main part *)
(* Create random graph *)
In[21]:= WIDTH=7;HEIGHT=8;TOTAL=WIDTH*HEIGHT
Out[21]= 56

In[24]:= g=DirectedGraph[RandomGraph[BernoulliGraphDistribution[TOTAL,0.05]],"Acyclic"];

...

(* Generate random expressions and numbers *)
In[26]:= expressions=Map[assignNumberOrExpr[g,#]&,VertexList[g]];

(* Make 2D table of it *)
In[27]:= t=Partition[expressions,WIDTH];

(* Export as tab-separated values *)
In[28]:= Export["/home/dennis/1.txt",t,"TSV"]
Out[28]= /home/dennis/1.txt

In[29]:= Grid[t,Frame->All,Alignment->Left]
```

Here is an output from *Grid[]*:

| | | | | | | |
|---|---|---|---|---|---|---|
| 846 | 499 | A3*913-H4 | ... | ... | ... | ... |
| B4*860+D2 | 999 | 59 | ... | ... | ... | ... |
| G6*379-C3-436-C4-289+H6 | 972 | 804 | ... | ... | ... | ... |
| F2 | E0 | B6-731-D3+791+B4*92+C1 | ... | ... | ... | ... |
| 519 | G1*402+D1*107*G3-458*A1 | D3 | ... | ... | ... | ... |
| F5-531+B5-222*E4 | 9 | B5+106*B6+600-B1 | ... | ... | ... | ... |
| C3-956*A5 | G4*408-D3*290*B6-899*G5+400+F1 | B2-701+H6 | ... | ... | ... | .. |
| B4-792*H4*407+F6-425-E1 | D2 | D3 | ... | ... | ... | ... |

Using this script, I can generate random spreadsheet of $26 \cdot 500 = 13000$ cells, which seems to be processed by Z3 in couple of seconds.

### 3.12.4 The files

The files, including Mathematica notebook: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/equations/spreadsheet.

## 3.13 Discrete tomography

How computed tomography (CT scan) actually works? A human body is bombarded by X-rays in various angles by X-ray tube in rotating torus. X-ray detectors are also located in torus, and all the information is recorded.

Here is we can simulate simple tomograph. An "i" character is rotating and will be "enlighten" at 4 angles. Let's imagine, character is bombarded by X-ray tube at left. All asterisks in each row is then summed and sum is "received" by X-ray detector at the right.

```
WIDTH= 11 HEIGHT= 11
angle =(/4)*0
    **      2
    **      2
            0
   ***      3
    **      2
    **      2
    **      2
    **      2
    **      2
   ****     4
            0
[2, 2, 0, 3, 2, 2, 2, 2, 2, 4, 0] ,
angle =(/4)*1
            0
            0
  *         1
 **         2
    *       1
    **      2
     **     2
     ****   4
       *    1
      *     1
            0
[0, 0, 1, 2, 1, 2, 2, 4, 1, 1, 0] ,
angle =(/4)*2
            0
            0
            0
            0
```

```
          *  1
** *******  9
** *******  9
   *     *  2
            0
            0
            0
[0, 0, 0, 0, 1, 9, 9, 2, 0, 0, 0] ,
angle =(/4)*3
            0
            0
       *    1
       **   2
      ** *  3
     ***    3
    **      2
            0
  **        2
   *        1
            0
[0, 0, 1, 2, 3, 3, 2, 0, 2, 1, 0] ,
```

( The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/tomo/gen.py )

All we got from our toy-level tomograph is 4 vectors, these are sums of all asterisks in rows for 4 angles:

```
[2, 2, 0, 3, 2, 2, 2, 2, 2, 4, 0] ,
[0, 0, 1, 2, 1, 2, 2, 4, 1, 1, 0] ,
[0, 0, 0, 0, 1, 9, 9, 2, 0, 0, 0] ,
[0, 0, 1, 2, 3, 3, 2, 0, 2, 1, 0] ,
```

How do we recover initial image? We are going to represent 11*11 matrix, where sum of each row must be equal to some value we already know. Then we rotate matrix, and do this again.

For the first matrix, the system of equations looks like that (we put there a values from the first vector):

```
C1,1 + C1,2 + C1,3 + ... + C1,11 =      2
C2,1 + C2,2 + C2,3 + ... + C2,11 =      2

...

C10,1 + C10,2 + C10,3 + ... + C10,11 =  4
C11,1 + C11,2 + C11,3 + ... + C11,11 =  0
```

We also build similar systems of equations for each angle.

The “rotate” function has been taken from the generation program, because, due to Python’s dynamic typization nature, it’s not important for the function to what operate on: strings, characters, or Z3 variable instances, so it works very well for all of them.

```
#-*- coding: utf-8 -*-

import math, sys
from z3 import *

# https://en.wikipedia.org/wiki/Rotation_matrix
def rotate(pic, angle):
    WIDTH=len(pic[0])
    HEIGHT=len(pic)
    #print WIDTH, HEIGHT
    assert WIDTH==HEIGHT
```

```
    ofs=WIDTH/2

    out = [[0 for x in range(WIDTH)] for y in range(HEIGHT)]

    for x in range(-ofs,ofs):
        for y in range(-ofs,ofs):
            newX = int(round(math.cos(angle)*x - math.sin(angle)*y,3))+ofs
            newY = int(round(math.sin(angle)*x + math.cos(angle)*y,3))+ofs
            # clip at boundaries, hence min(..., HEIGHT-1)
            out[min(newX,HEIGHT-1)][min(newY,WIDTH-1)]=pic[x+ofs][y+ofs]
    return out

vectors=[
[2, 2, 0, 3, 2, 2, 2, 2, 2, 4, 0] ,
[0, 0, 1, 2, 1, 2, 2, 4, 1, 1, 0] ,
[0, 0, 0, 0, 1, 9, 9, 2, 0, 0, 0] ,
[0, 0, 1, 2, 3, 3, 2, 0, 2, 1, 0]]

WIDTH = HEIGHT = len(vectors[0])

s=Solver()
cells=[[Int('cell_r=%d_c=%d' % (r,c)) for c in range(WIDTH)] for r in range(HEIGHT)]

# monochrome picture, only 0's or 1's:
for c in range(WIDTH):
    for r in range(HEIGHT):
        s.add(Or(cells[r][c]==0, cells[r][c]==1))

def all_zeroes_in_vector(vec):
    for v in vec:
        if v!=0:
            return False
    return True

ANGLES=len(vectors)
for a in range(ANGLES):
    angle=a*(math.pi/ANGLES)
    rows=rotate(cells, angle)
    r=0
    for row in rows:
        # skip empty rows:
        if all_zeroes_in_vector(row)==False:
            # sum of row must be equal to the corresponding element of vector:
            s.add(Sum(*row)==vectors[a][r])
        r=r+1

print s.check()
m=s.model()
for r in range(HEIGHT):
    for c in range(WIDTH):
        if str(m[cells[r][c]])=="1":
            sys.stdout.write("*")
        else:
            sys.stdout.write(" ")
    print ""
```

( The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/equations/tomo/

solve.py )

That works:

```
% python solve.py
sat
    **
    **

   ***
    **
    **
    **
    **
    **
   ****
```

In other words, all SMT-solver does here is solving a system of equations.

So, 4 angles are enough. What if we could use only 3 angles?

```
WIDTH= 11 HEIGHT= 11
angle =(/3)*0
    **      2
    **      2
            0
   ***      3
    **      2
    **      2
    **      2
    **      2
    **      2
   ****     4
            0
[2, 2, 0, 3, 2, 2, 2, 2, 2, 4, 0] ,
angle =(/3)*1
            0
            0
            0
 **         2
 **         2
   ***      3
     ****   4
       **   2
       *    1
            0
            0
[0, 0, 0, 2, 2, 3, 4, 2, 1, 0, 0] ,
angle =(/3)*2
            0
            0
            0
       **   2
       **   2
     *****  5
    **      2
 **         2
  *         1
            0
            0
```

```
[0, 0, 0, 2, 2, 5, 2, 2, 1, 0, 0] ,
```

No, it's not enough:

```
% time python solve3.py
sat
 *  *
    **

     * **
   **
   *  *
    **
     *   *
*   *
   ****
```

However, the result is correct, but only 3 vectors allows too many possible "initial images", and Z3 SMT-solver finds first.

Further reading: https://en.wikipedia.org/wiki/Discrete_tomography, https://en.wikipedia.org/wiki/2-satisfiabi Discrete_tomography.

## 3.14 Cribbage

I've found this problem in the Ronald L. Graham, Donald E. Knuth, Oren Patashnik – "Concrete Mathematics" book:

> Cribbage players have long been aware that $15 = 7 + 8 = 4 + 5 + 6 = 1 + 2 + 3 + 4 + 5$ . Find the number of ways to represent 1050 as a sum of consecutive positive integers. (The trivial representation '1050' by itself counts as one way; thus there are four, not three, ways to represent 15 as a sum of consecutive positive integers. Incidentally, a knowledge of cribbage rules is of no use in this problem.)

My solution:

```
#!/usr/bin/env python

from z3 import *

def attempt(terms, N):
    #print "terms = %d" % terms

    cells=[Int('%d' % i) for i in range(terms)]

    s=Solver()

    for i in range(terms-1):
        s.add(cells[i]+1 == cells[i+1])

    s.add(Sum(cells)==N)

    s.add(cells[0]>0)

    if s.check()==sat:
        m=s.model()
        print "(%d terms) %d + ... + %d == %d" % (terms, m[cells[0]].as_long(), m[cells[
            terms-1]].as_long(), N)
```

```
#N=15
N=1050

for i in range(2,N):
    attempt(i, N)
```

The result:

```
(3 terms) 349 + ... + 351 == 1050
(4 terms) 261 + ... + 264 == 1050
(5 terms) 208 + ... + 212 == 1050
(7 terms) 147 + ... + 153 == 1050
(12 terms) 82 + ... + 93 == 1050
(15 terms) 63 + ... + 77 == 1050
(20 terms) 43 + ... + 62 == 1050
(21 terms) 40 + ... + 60 == 1050
(25 terms) 30 + ... + 54 == 1050
(28 terms) 24 + ... + 51 == 1050
(35 terms) 13 + ... + 47 == 1050
```

### 3.15 Solving Problem Euler 31: "Coin sums"

> In England the currency is made up of pound, £, and pence, p, and there are eight coins in general circulation:
>
> 1p, 2p, 5p, 10p, 20p, 50p, £1 (100p) and £2 (200p). It is possible to make £2 in the following way:
>
> 1£1 + 150p + 220p + 15p + 12p + 31p How many different ways can £2 be made using any number of coins?

( Problem Euler 31 — Coin sums )

Using Z3 for solving this is overkill, and also slow, but nevertheless, it works, showing all possible solutions as well. The piece of code for blocking already found solution and search for next, and thus, counting all solutions, was taken from Stack Overflow answer [58]. This is also called "model counting". Constraints like "a>=0" must be present, because Z3 solver will find solutions with negative numbers.

```
#!/usr/bin/python

from z3 import *

a,b,c,d,e,f,g,h = Ints('a b c d e f g h')
s = Solver()
s.add(1*a + 2*b + 5*c + 10*d + 20*e + 50*f + 100*g + 200*h == 200,
   a>=0, b>=0, c>=0, d>=0, e>=0, f>=0, g>=0, h>=0)
result=[]

while True:
    if s.check() == sat:
        m = s.model()
        print m
        result.append(m)
        # Create a new constraint the blocks the current model
        block = []
```

[58] http://stackoverflow.com/questions/11867611/z3py-checking-all-solutions-for-equation, another question: http://stackoverflow.com/questions/13395391/z3-finding-all-satisfying-models

```
        for d in m:
            # create a constant from declaration
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print len(result)
        break
```

Works very slow, and this is what it produces:

```
[h = 0, g = 0, f = 0, e = 0, d = 0, c = 0, b = 0, a = 200]
[f = 1, b = 5, a = 0, d = 1, g = 1, h = 0, c = 2, e = 1]
[f = 0, b = 1, a = 153, d = 0, g = 0, h = 0, c = 1, e = 2]
...
[f = 0, b = 31, a = 33, d = 2, g = 0, h = 0, c = 17, e = 0]
[f = 0, b = 30, a = 35, d = 2, g = 0, h = 0, c = 17, e = 0]
[f = 0, b = 5, a = 50, d = 2, g = 0, h = 0, c = 24, e = 0]
```

73682 results in total.

### 3.16 Exercise 15 from TAOCP "7.1.3 Bitwise tricks and techniques"

Page 53 from the fasc1a.ps, or: http://www.cs.utsa.edu/~wagner/knuth/fasc1a.pdf

▶ **15.** [*M26*] J. H. Quick noticed that $((x+2)\oplus 3)-2=((x-2)\oplus 3)+2$ for all $x$. Find all constants $a$ and $b$ such that $((x+a)\oplus b)-a=((x-a)\oplus b)+a$ is an identity.

Figure 8: Page 53

Soltuion:

```
from z3 import *

s=Solver()

a, b=BitVecs('a b', 4)
x, y=BitVecs('x y', 4)

s.add(ForAll(x, ForAll(y,  ((x+a)^b)-a == ((x-a)^b)+a  )))

# enumerate all possible solutions:
results=[]
while True:
    if s.check() == sat:
        m = s.model()
        print m

        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "results total=", len(results)
        break
```

For 4-bit bitvectors:

```
...

[b = 7, a = 0]
[b = 6, a = 8]
[b = 7, a = 8]
[b = 6, a = 12]
[b = 7, a = 12]
[b = 12, a = 0]
[b = 13, a = 0]
[b = 12, a = 8]
[b = 13, a = 8]
[b = 12, a = 4]
[b = 13, a = 4]
[b = 12, a = 12]
[b = 13, a = 12]
[b = 14, a = 0]
[b = 15, a = 0]
[b = 14, a = 4]
[b = 15, a = 4]
[b = 14, a = 8]
[b = 15, a = 8]
[b = 14, a = 12]
[b = 15, a = 12]
results total= 128
```

## 3.17 De Bruijn sequences; leading/trailing zero bits counting

### 3.17.1 Introduction

Let's imagine there is a very simplified code lock accepting 2 digits, but it has no "enter" key, it just checks 2 last entered digits. Our task is to brute force each 2-digit combination. Naïve method is to try 00, 01, 02 ... 99. That require 2*100=200 key pressings. Will it be possible to reduce number of key pressings during brute-force? It is indeed so, with the help of De Bruijn sequences. We can generate them for the code lock, using Wolfram Mathematica:

```
In[]:= DeBruijnSequence[{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, 2]
Out[]= {6, 8, 6, 5, 4, 3, 2, 1, 7, 8, 7, 1, 1, 0, 9, 0, 8, 0, 6, 6, \
0, 5, 5, 0, 4, 4, 0, 3, 3, 0, 2, 7, 2, 2, 0, 7, 7, 9, 8, 8, 9, 9, 7, \
0, 0, 1, 9, 1, 8, 1, 6, 1, 5, 1, 4, 1, 3, 7, 3, 1, 2, 9, 2, 8, 2, 6, \
2, 5, 2, 4, 7, 4, 2, 3, 9, 3, 8, 3, 6, 3, 5, 7, 5, 3, 4, 9, 4, 8, 4, \
6, 7, 6, 4, 5, 9, 5, 8, 5, 6, 9}
```

The result has exactly 100 digits, which is 2 times less than our initial idea can offer. By scanning visually this 100-digits array, you'll find any number in 00..99 range. All numbers are overlapped with each other: second half of each number is also first half of the next number, etc.

Here is another. We need a sequence of binary bits with all 3-bit numbers in it:

```
In[]:= DeBruijnSequence[{0, 1}, 3]
Out[]= {1, 0, 1, 0, 0, 0, 1, 1}
```

Sequence length is just 8 bits, but it has all binary numbers in 000..111 range. You may visually spot 000 in the middle of sequence. 111 is also present: two first bits of it at the end of sequence and the last bit is in the beginning. This is so because De Bruijn sequences are cyclic.

There is also visual demonstration: http://demonstrations.wolfram.com/DeBruijnSequences/.

### 3.17.2 Trailing zero bits counting

In the Wikipedia article about De Bruijn sequences we can find:

> The symbols of a De Bruijn sequence written around a circular object (such as a wheel of a robot) can be used to identify its angle by examining the n consecutive symbols facing a fixed point.

Indeed: if we know De Bruijn sequence and we observe only part of it (any part), we can deduce exact position of this part within sequence.

Let's see, how this feature can be used.

Let's say, there is a need to detect position of input bit within 32-bit word. For 0x1, the algorithm should report 1. 2 for 0x2. 3 for 0x4. And 31 for 0x80000000.

The result is in 0..31 range, so the result can be stored in 5 bits.

We can construct binary De Bruijn sequence for all 5-bit numbers:

```
In[]:= tmp = DeBruijnSequence[{0, 1}, 5]
Out[]= {1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1,
     0, 0, 0, 0, 0}

In[]:= BaseForm[FromDigits[tmp, 2], 16]
Out[]:= e6bec520
```

Let's also recall that division some number by $2^n$ number is the same thing as shifting it by $n$ bits. So if you divide 0xe6bec520 by 1, the result is not shifted, it is still the same. If if divide 0xe6bec520 by 4 ($2^2$), the result is shifted by 2 bits. We then take result and isolate lowest 5 bits. This result is unique number for each input. Let's shift 0xe6bec520 by all possible count number, and we'll get all possible last 5-bit values:

```
In[]:= Table[BitAnd[BitShiftRight[FromDigits[tmp, 2], i], 31], {i, 0, 31}]
Out[]= {0, 16, 8, 4, 18, 9, 20, 10, 5, 2, 17, 24, 12, 22, 27, 29, \
30, 31, 15, 23, 11, 21, 26, 13, 6, 19, 25, 28, 14, 7, 3, 1}
```

The table has no duplicates:

```
In[]:= DuplicateFreeQ[%]
Out[]= True
```

Using this table, it's easy to build a *magic* table. OK, now working C example:

```
#include <stdint.h>
#include <stdio.h>

int magic_tbl[32];

// returns single bit position counting from LSB
// not working for i==0
int bitpos (uint32_t i)
{
        return magic_tbl[(0xe6bec520/i) & 0x1F];
};

int main()
{
        // construct magic table
        // may be omitted in production code
        for (int i=0; i<32; i++)
                magic_tbl[(0xe6bec520/(1<<i)) & 0x1F]=i;
```

```
        // test
        for (int i=0; i<32; i++)
        {
                printf ("input=0x%x, result=%d\n", 1<<i, bitpos (1<<i));
        };
};
```

Here we feed our bitpos() function with numbers in 0..0x80000000 range and we got:

```
input=0x1, result=0
input=0x2, result=1
input=0x4, result=2
input=0x8, result=3
input=0x10, result=4
input=0x20, result=5
input=0x40, result=6
input=0x80, result=7
input=0x100, result=8
input=0x200, result=9
input=0x400, result=10
input=0x800, result=11
input=0x1000, result=12
input=0x2000, result=13
input=0x4000, result=14
input=0x8000, result=15
input=0x10000, result=16
input=0x20000, result=17
input=0x40000, result=18
input=0x80000, result=19
input=0x100000, result=20
input=0x200000, result=21
input=0x400000, result=22
input=0x800000, result=23
input=0x1000000, result=24
input=0x2000000, result=25
input=0x4000000, result=26
input=0x8000000, result=27
input=0x10000000, result=28
input=0x20000000, result=29
input=0x40000000, result=30
input=0x80000000, result=31
```

The bitpos() function actually counts trailing zero bits, but it works only for input values where only one bit is set. To make it more practical, we need to devise a method to drop all leading bits except of the last one. This method is very simple and well-known:

```
input & (-input)
```

This bit twiddling hack can solve the job. Feeding 0x11 to it, it will return 0x1. Feeding 0xFFFF0000, it will return 0x10000. In other words, it leaves lowest significant bit of the value, dropping all others.

It works because negated value in two's complement environment is the value with all bits flipped but also 1 added (because there is a zero in the middle of ring). For example, let's take 0xF0. -0xF0 is 0x10 or 0xFFFFFF10. ANDing 0xF0 and 0xFFFFFF10 will produce 0x10.

Let's modify our algorithm to support true trailing zero bits count:

```
#include <stdint.h>
#include <stdio.h>

int magic_tbl[32];
```

```
// not working for i==0
int tzcnt (uint32_t i)
{
        uint32_t a=i & (-i);
        return magic_tbl[(0xe6bec520/a) & 0x1F];
};

int main()
{
        // construct magic table
        // may be omitted in production code
        for (int i=0; i<32; i++)
                magic_tbl[(0xe6bec520/(1<<i)) & 0x1F]=i;

        // test:
        printf ("%d\n", tzcnt (0xFFFF0000));
        printf ("%d\n", tzcnt (0xFFFF0010));
};
```

It works!

```
16
4
```

But it has one drawback: it uses division, which is slow. Can we just multiplicate De Bruijn sequence by the value with the bit isolated instead of dividing sequence? Yes, indeed. Let's check in Mathematica:

```
In[]:= BaseForm[16^^e6bec520*16^^80000000, 16]
Out[]:= 0x735f629000000000
```

The result is just too big to fit in 32-bit register, but can be used. MUL/IMUL instruction 32-bit x86 CPUs stores 64-bit result into two 32-bit registers pair, yes. But let's suppose we would like to make portable code which will work on any 32-bit architecture. First, let's again take a look on De Bruijn sequence Mathematica first produced:

```
In[]:= tmp = DeBruijnSequence[{0, 1}, 5]
Out[]= {1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, \
0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0}
```

There is exactly 5 bits at the end which can be dropped. The "magic" constant will be much smaller:

```
In[]:= BaseForm[BitShiftRight[FromDigits[tmp, 2], 5], 16]
Out[]:=0x735f629
```

The "magic" constant is now "divided by 32 (or 1»5)". This mean that the result of multiplication of some value with one isolated bit by new magic number will also be smaller, so the bits we need will be stored at the high 5 bits of the result.

De Bruijn sequence is not broken after 5 lowest bits dropped, because these zero bits are "relocated" to the start of the sequence. Sequence is cyclic after all.

```
#include <stdint.h>
#include <stdio.h>

int magic_tbl[32];

// not working for i==0
int tzcnt (uint32_t i)
{
        uint32_t a=i & (-i);
```

```
        // 5 bits we need are stored in 31..27 bits of product, shift and isolate them
           after multiplication:
        return magic_tbl[((0x735f629*a)>>27) & 0x1F];
};

int main()
{
        // construct magic table
        // may be omitted in production code
        for (int i=0; i<32; i++)
                magic_tbl[(0x735f629<<i >>27) & 0x1F]=i;

        // test:
        printf ("%d\n", tzcnt (0xFFFF0000));
        printf ("%d\n", tzcnt (0xFFFF0010));
};
```

### 3.17.3 Leading zero bits counting

This is almost the same task, but most significant bit must be isolated instead of lowest. This is typical algorithm for 32-bit integer values:

```
x |= x >> 1;
x |= x >> 2;
x |= x >> 4;
x |= x >> 8;
x |= x >> 16;
```

For example, 0x100 becomes 0x1ff, 0x1000 becomes 0x1fff, 0x20000 becomes 0x3ffff, 0x12340000 becomes 0x1fffffff. It works because all 1 bits are gradually propagated towards the lowest bit in 32-bit number, while zero bits at the left of most significant 1 bit are not touched.

It's possible to add 1 to resulting number, so it will becomes 0x2000 or 0x20000000, but in fact, since multiplication by magic number is used, these numbers are very close to each other, so there are no error.

This example I used in my reverse engineering exercise from 15-Aug-2015: https://yurichev.com/blog/2015-aug-18/.

```
int v[64]=
        { -1,31, 8,30, -1, 7,-1,-1, 29,-1,26, 6, -1,-1, 2,-1,
          -1,28,-1,-1, -1,19,25,-1, 5,-1,17,-1, 23,14, 1,-1,
           9,-1,-1,-1, 27,-1, 3,-1, -1,-1,20,-1, 18,24,15,10,
          -1,-1, 4,-1, 21,-1,16,11, -1,22,-1,12, 13,-1, 0,-1 };

int LZCNT(uint32_t x)
{
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    x *= 0x4badf0d;
    return v[x >> 26];
}
```

This piece of code I took from here. It is slightly different: the table is twice bigger, and the function returns -1 if input value is zero. The magic number I found using just brute-force, so the readers will not be able to google it, for the sake of exercise. (By the way, I've got 12,665,720 magic numbers which can serve this purpose. This is about 0.294

The code is tricky after all, and the moral of the exercise is that practicing reverse engineer sometimes may just observe input/outputs to understand code's behaviour instead of diving into it.

### 3.17.4 Performance

The algorithms considered are probably fastest known, they has no conditional jumps, which is very good for CPUs starting at RISCs. Newer CPUs has LZCNT and TZCNT instructions, even 80386 had BSF/BSR instructions which can be used for this: https://en.wikipedia.org/wiki/Find_first_set. Nevertheless, these algorithms can be still used on cheaper RISC CPUs without specialized instructions.

### 3.17.5 Applications

Number of leading zero bits is binary logarithm of value. My article about logarithms including binary: https://yurichev.com/writings/log_intro.pdf.

These algorithms are also extensively used in chess engines programming, where each piece is represented as 64-bit bitmask (chess board has 64 squares): http://chessprogramming.wikispaces.com/BitScan.

There are more: https://en.wikipedia.org/wiki/Find_first_set#Applications.

### 3.17.6 Generation of De Bruijn sequences

De Bruijn graph is a graph where all values are represented as vertices (or nodes) and each edge (or link) connects two nodes which can be "overlapped". Then we need to visit each edge only once, this is called *eulerian path*. It is like the famous *task of seven bridges of Königsberg*: traveller must visit each bridge only once.

There are also simpler algorithms exist: https://en.wikipedia.org/wiki/De_Bruijn_sequence#Algorithm.

### 3.17.7 Other articles

At least these are worth reading: http://supertech.csail.mit.edu/papers/debruijn.pdf, http://alexandria.tue.nl/repository/books/252901.pdf, Wikipedia Article about De Bruijn sequences.

https://chessprogramming.wikispaces.com/De+Bruijn+sequence, https://chessprogramming.wikispaces.com/De+Bruijn+Sequence+Generator.

## 3.18 Generating de Bruijn sequences using Z3

The following piece of quite esoteric code calculates number of leading zero bits [59]:

```
int v[64]=
        { -1,31, 8,30, -1, 7,-1,-1, 29,-1,26, 6, -1,-1, 2,-1,
          -1,28,-1,-1, -1,19,25,-1, 5,-1,17,-1, 23,14, 1,-1,
           9,-1,-1,-1, 27,-1, 3,-1, -1,-1,20,-1, 18,24,15,10,
          -1,-1, 4,-1, 21,-1,16,11, -1,22,-1,12, 13,-1, 0,-1 };

int LZCNT(uint32_t x)
{
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    x *= 0x4badf0d;
    return v[x >> 26];
}
```

(This is usually done using simpler algorithm, but it will contain conditional jumps, which is bad for CPUs starting at RISC. There are no conditional jumps in this algorithm.)

The magic number used here is called *de Bruijn sequence*, and I once used bruteforce to find it (one of the results was *0x4badf0d*, which is used here). But what if we need magic number for 64-bit values? Bruteforce is not an option here.

If you already read about these sequences in my blog or in other sources, you can see that the 32-bit magic number is a number consisting of 5-bit overlapping chunks, and all chunks must be unique, i.e., must not be repeating.

For 64-bit magic number, these are 6-bit overlapping chunks.

[59] https://en.wikipedia.org/wiki/Find_first_set

To find the magic number, one can find a Hamiltonian path of a de Bruijn graph. But I've found that Z3 is also can do this, though, overkill, but this is more illustrative.

```
#!/usr/bin/python
from z3 import *

out = BitVec('out', 64)

tmp=[]
for i in range(64):
    tmp.append((out>>i)&0x3F)

s=Solver()

# all overlapping 6-bit chunks must be distinct:
s.add(Distinct(*tmp))
# MSB must be zero:
s.add((out&0x8000000000000000)==0)

print s.check()

result=s.model()[out].as_long()
print "0x%x" % result

# print overlapping 6-bit chunks:
for i in range(64):
    t=(result>>i)&0x3F
    print " "*(63-i) + format(t, 'b').zfill(6)
```

We just enumerate all overlapping 6-bit chunks and tell Z3 that they must be unique (see `Distinct`). Output:

```
sat
0x79c52dd0991abf60
                                                               100000
                                                              110000
                                                             011000
                                                            101100
                                                           110110
                                                          111011
                                                         111101
                                                        111110
                                                       111111
                                                      011111
                                                     101111
                                                    010111
                                                   101011
                                                  010101
                                                 101010
                                                110101
                                               011010
                                              001101
                                             000110
                                            100011
                                           010001
                                          001000
                                         100100
                                        110010
                                       011001
```

```
                                      001100
                                     100110
                                    010011
                                   001001
                                  000100
                                 000010
                                100001
                               010000
                              101000
                             110100
                            111010
                           011101
                          101110
                         110111
                        011011
                       101101
                      010110
                     001011
                    100101
                   010010
                  101001
                 010100
                001010
               000101
              100010
             110001
            111000
           011100
          001110
         100111
        110011
       111001
      111100
     011110
    001111
   000111
  000011
 000001
000000
```

Overlapping chunks are clearly visible. So the magic number is *0x79c52dd0991abf60*. Let's check:

```
#include <stdint.h>
#include <stdio.h>
#include <assert.h>

#define MAGIC 0x79c52dd0991abf60

int magic_tbl[64];

// returns single bit position counting from LSB
// not works for i==0
int bitpos (uint64_t i)
{
        return magic_tbl[(MAGIC/i) & 0x3F];
};
```

```
// count trailing zeroes
// not works for i==0
int tzcnt (uint64_t i)
{
        uint64_t a=i & (-i);
        return magic_tbl[(MAGIC/a) & 0x3F];
};

int main()
{
        // construct magic table
        // may be omitted in production code
        for (int i=0; i<64; i++)
                magic_tbl[(MAGIC/(1ULL<<i)) & 0x3F]=i;

        // test
        for (int i=0; i<64; i++)
        {
                printf ("input=0x%llx, result=%d\n", 1ULL<<i, bitpos (1ULL<<i));
                assert(bitpos(1ULL<<i)==i);
        };
        assert(tzcnt (0xFFFF0000)==16);
        assert(tzcnt (0xFFFF0010)==4);
};
```

That works!

## 3.19 Solving the $x^y = 19487171$ equation

Find x,y for $x^y = 19487171$. The correct result x=11, y=7. It's like http://reference.wolfram.com/language/ref/Surd.html.

The non-standard function `bvmul_no_overflow` is used here. it behaves like bvmul, but high part is forced to be zero. This is not like most programming languages and CPUs do multiplication (the result there is modulo $2^n$, where $n$ is width of CPU register). However, thus it's simpler for me to write this all without adding additional `zero_extend` function.

```
; tested with MK85

(set-logic QF_BV)
(declare-fun x () (_ BitVec 32))
(declare-fun y () (_ BitVec 4))
(declare-fun out () (_ BitVec 32))

; like switch() or if() tree:
(assert (= out
        (ite (= y #x2) (bvmul_no_overflow x x)
        (ite (= y #x3) (bvmul_no_overflow x x x)
        (ite (= y #x4) (bvmul_no_overflow x x x x)
        (ite (= y #x5) (bvmul_no_overflow x x x x x)
        (ite (= y #x6) (bvmul_no_overflow x x x x x x)
        (ite (= y #x7) (bvmul_no_overflow x x x x x x x)
        (_ bv0 32))))))))

(assert (= out (_ bv19487171 32)))

(check-sat)
(get-model)
```

Listing 1: The solution

```
(model
        (define-fun x () (_ BitVec 32) (_ bv11 32)) ; 0xb
        (define-fun y () (_ BitVec 4) (_ bv7 4)) ; 0x7
        (define-fun out () (_ BitVec 32) (_ bv19487171 32)) ; 0x12959c3
)
```

# 4 Proofs

SAT/SMT solvers can't prove *correctness* of something, or if the model behaves as the author wanted.

However, it can prove equivalence of two expressions or models.

## 4.1 Using Z3 theorem prover to prove equivalence of some weird alternative to XOR operation

(The test was first published in my blog at April 2015: http://blog.yurichev.com/node/86).

There is a "A Hacker's Assistant" program[60] (*Aha!*) written by Henry Warren, who is also the author of the great "Hacker's Delight" book.

The *Aha!* program is essentially *superoptimizer*[61], which blindly brute-force a list of some generic RISC CPU instructions to achieve shortest possible (and jumpless or branch-free) CPU code sequence for desired operation. For example, *Aha!* can find jumpless version of abs() function easily.

Compiler developers use superoptimization to find shortest possible (and/or jumpless) code, but I tried to do otherwise—to find longest code for some primitive operation. I tried *Aha!* to find equivalent of basic XOR operation without usage of the actual XOR instruction, and the most bizarre example *Aha!* gave is:

```
Found a 4-operation program:
   add   r1,ry,rx
   and   r2,ry,rx
   mul   r3,r2,-2
   add   r4,r3,r1
   Expr: (((y & x)*-2) + (y + x))
```

And it's hard to say, why/where we can use it, maybe for obfuscation, I'm not sure. I would call this *suboptimization* (as opposed to *superoptimization*). Or maybe *superdeoptimization*.

But my another question was also, is it possible to prove that this is correct formula at all? The *Aha!* checking some intput/output values against XOR operation, but of course, not all the possible values. It is 32-bit code, so it may take very long time to try all possible 32-bit inputs to test it.

We can try Z3 theorem prover for the job. It's called *prover*, after all.

So I wrote this:

```
#!/usr/bin/python
from z3 import *

x = BitVec('x', 32)
y = BitVec('y', 32)
output = BitVec('output', 32)
s = Solver()
s.add(x^y==output)
s.add(((y & x)*0xFFFFFFFE) + (y + x)!=output)
print s.check()
```

In plain English language, this means "are there any case for $x$ and $y$ where $x \oplus y$ doesn't equals to $((y\&x)*-2)+(y+x)$?" ...and Z3 prints "unsat", meaning, it can't find any counterexample to the equation. So this *Aha!* result is proved to be working just like XOR operation.

[60] http://www.hackersdelight.org/

[61] http://en.wikipedia.org/wiki/Superoptimization

Oh, I also tried to extend the formula to 64 bit:

```
#!/usr/bin/python
from z3 import *

x = BitVec('x', 64)
y = BitVec('y', 64)
output = BitVec('output', 64)
s = Solver()
s.add(x^y==output)
s.add(((y & x)*0xFFFFFFFE) + (y + x)!=output)
print s.check()
```

Nope, now it says "sat", meaning, Z3 found at least one counterexample. Oops, it's because I forgot to extend -2 number to 64-bit value:

```
#!/usr/bin/python
from z3 import *

x = BitVec('x', 64)
y = BitVec('y', 64)
output = BitVec('output', 64)
s = Solver()
s.add(x^y==output)
s.add(((y & x)*0xFFFFFFFFFFFFFFFE) + (y + x)!=output)
print s.check()
```

Now it says "unsat", so the formula given by *Aha!* works for 64-bit code as well.

### 4.1.1 In SMT-LIB form

Now we can rephrase our expression to more suitable form: $(x + y - ((x \& y) << 1))$. It also works well in Z3Py:

```
#!/usr/bin/python
from z3 import *

x = BitVec('x', 64)
y = BitVec('y', 64)
output = BitVec('output', 64)
s = Solver()
s.add(x^y==output)
s.add((x + y - ((x & y)<<1)) != output)
print s.check()
```

Here is how to define it in SMT-LIB way:

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
        (not
                (=
                        (bvsub
                                (bvadd x y)
                                (bvshl (bvand x y) (_ bv1 64)))
                        (bvxor x y)
                )
        )
)
(check-sat)
```

### 4.1.2 Using universal quantifier

Z3 supports universal quantifier **exists**, which is true if at least one set of variables satistfied underlying condition:

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
        (exists ((x (_ BitVec 64)) (y (_ BitVec 64)))
                (not (=
                        (bvsub
                                (bvadd x y)
                                (bvshl (bvand x y) (_ bv1 64))
                        )
                        (bvxor x y)
                ))
        )
)
(check-sat)
```

It returns "unsat", meaning, Z3 couldn't find any counterexample of the equation, i.e., it's not exist.

This is also known as ∃ in mathematical logic lingo.

Z3 also supports universal quantifier `forall`, which is true if the equation is true for all possible values. So we can rewrite our SMT-LIB example as:

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
        (forall ((x (_ BitVec 64)) (y (_ BitVec 64)))
                (=
                        (bvsub
                                (bvadd x y)
                                (bvshl (bvand x y) (_ bv1 64))
                        )
                        (bvxor x y)
                )
        )
)
(check-sat)
```

It returns "sat", meaning, the equation is correct for all possible 64-bit `x` and `y` values, like them all were checked. Mathematically speaking: $\forall n \in \mathbb{N}\ (x \oplus y = (x + y - ((x \& y) << 1)))$ [62]

### 4.1.3 How the expression works

First of all, binary addition can be viewed as binary XORing with carrying (2.3.2). Here is an example: let's add 2 (10b) and 2 (10b). XORing these two values resulting 0, but there is a carry generated during addition of two second bits. That carry bit is propagated further and settles at the place of the 3rd bit: 100b. 4 (100b) is hence a final result of addition.

If the carry bits are not generated during addition, the addition operation is merely XORing. For example, let's add 1 (1b) and 2 (10b). $1 + 2$ equals to 3, but $1 \oplus 2$ is also 3.

If the addition is XORing plus carry generation and application, we should eliminate effect of carrying somehow here. The first part of the expression $(x + y)$ is addition, the second $((x \& y) << 1)$ is just calculation of every carry bit which was used during addition. If to subtract carry bits from the result of addition, the only XOR effect is left then.

It's hard to say how Z3 proves this: maybe it just simplifies the equation down to single XOR using simple boolean algebra rewriting rules?

[62] ∀ means *equation must be true for all possible values*, which are choosen from natural numbers ($\mathbb{N}$).

## 4.2 Proving bizarre XOR alternative using SAT solver

Now let's try to prove it using SAT.

We would build an electric circuit for $x \oplus y = -2 * (x \& y) + (x + y)$ like that:

```
x --- +---+
      |AND| --- +---+
y --- +---+     |MUL| --- +---+
           -2 +---+       |ADD| --- +---+
                      x - +---+     |ADD| --> out1 --- +--------+
                                y - +---+              |        |       +----+
                                                       |  XOR   | --> | OR | --> must be 0
x --- +---+                                            |        |       +----+
      |XOR| ---------------------------------------> out2 --- +--------+
y --- +---+
```

So it has two parts: generic XOR block and a block which must be equivalent to XOR. Then we compare its outputs using XOR and OR. If outputs of these parts are always equal to each other for all possible x and y, output of the whole block must be 0.

I do otherwise, I'm trying to find such an input pair, for which output will be 1:

```
def chk1():
    input_bits=8

    s=SAT_lib.SAT_lib(False)

    x,y=s.alloc_BV(input_bits),s.alloc_BV(input_bits)
    step1=s.BV_AND(x,y)
    minus_2=[s.const_true]*(input_bits-1)+[s.const_false]
    product=s.multiplier(step1,minus_2)[input_bits:]
    result1=s.adder(s.adder(product, x)[0], y)[0]

    result2=s.BV_XOR(x,y)

    s.fix(s.OR(s.BV_XOR(result1, result2)), True)

    if s.solve()==False:
        print ("unsat")
        return

    print ("sat")
    print ("x=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(x)))
    print ("y=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(y)))
    print ("step1=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(step1)))
    print ("product=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(product)))
    print ("result1=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(result1)))
    print ("result2=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(result2)))
    print ("minus_2=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(minus_2)))
```

The full source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/proofs/XOR_SAT/XOR_SAT.py.

SAT solver returns "unsat", meaning, it couldn't find such a pair. In other words, it couldn't find a counterexample. So the circuit always outputs 0, for all possible inputs, meaning, outputs of two parts are always the same.

Modify the circuit, and the program will find such a state, and print it.

That circuit also called "miter". According to Google translate, one meaning of the word is:

```
a joint made between two pieces of wood or other material at an angle of 90°, such that
   the line of junction bisects this angle.
```

It's also slow, because multiplier block is used: so we use small 8-bit x's and y's.

But the whole thing can be rewritten: $x \oplus y = x + y - (x \& y) << 1$. And subtraction is addition, but with one negated operand. So, $x \oplus y = (-(x \& y)) << 1 + (x + y)$ or $x \oplus y = (x \& y) * 2 - (x + y)$.

```
x --- +---+      +---+      +---+
      |AND| ---  |NEG| ---  |<<1| -+
y --- +---+      +---+      +---+  +- +---+
                                      |ADD| --- +---+
                             x -  +---+         |ADD| --> out1 --- +-------+
                                          y - +---+                |       |      +-----+
                                                                   |  XOR  | ---> | OR  | ---> must be 0
x --- +---+                                                        |       |      +-----+
      |XOR| -------------------------------------------------> out2 --- +-------+
y --- +---+
```

NEG is negation block, in two's complement system. It just inverts all bits and adds 1:

```
    def NEG(self, x):
        # invert all bits
        tmp=self.BV_NOT(x)
        # add 1
        one=self.alloc_BV(len(tmp))
        self.fix_BV(one,n_to_BV(1, len(tmp)))
        return self.adder(tmp, one)[0]
```

Shift by one bit does nothing except rewiring.

That works way faster, and can prove correctness for 64-bit x's and y's, or for even bigger input values:

```
def chk2():
    input_bits=64

    s=SAT_lib.SAT_lib(False)

    x,y=s.alloc_BV(input_bits),s.alloc_BV(input_bits)
    step1=s.BV_AND(x,y)
    step2=s.shift_left_1(s.NEG(step1))

    result1=s.adder(s.adder(step2, x)[0], y)[0]

    result2=s.BV_XOR(x,y)

    s.fix(s.OR(s.BV_XOR(result1, result2)), True)

    if s.solve()==False:
        print ("unsat")
        return

    print ("sat")
    print ("x=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(x)))
    print ("y=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(y)))
    print ("step1=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(step1)))
    print ("step2=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(step2)))
    print ("result1=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(result1)))
    print ("result2=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(result2)))
```

The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/proofs/XOR_SAT/XOR_SAT.py.

## 4.3 Dietz's formula

One of the impressive examples of *Aha!* work is finding of Dietz's formula[63], which is the code of computing average number of two numbers without overflow (which is important if you want to find average number of numbers like 0xFFFFFF00 and so on, using 32-bit registers).

Taking this in input:

```
int userfun(int x, int y) {      // To find Dietz's formula for
                                 // the floor-average of two
                                 // unsigned integers.
   return ((unsigned long long)x + (unsigned long long)y) >> 1;
}
```

...the *Aha!* gives this:

```
Found a 4-operation program:
   and   r1,ry,rx
   xor   r2,ry,rx
   shrs  r3,r2,1
   add   r4,r3,r1
   Expr: (((y ^ x) >>s 1) + (y & x))
```

And it works correctly[64]. But how to prove it?

We will place Dietz's formula on the left side of equation and $x + y/2$ (or $x + y >> 1$) on the right side:

$$\forall n \in 0..2^{64} - 1.(x\&y) + (x \oplus y) >> 1 = x + y >> 1$$

One important thing is that we can't operate on 64-bit values on right side, because result will overflow. So we will zero extend inputs on right side by 1 bit (in other words, we will just 1 zero bit before each value). The result of Dietz's formula will also be extended by 1 bit. Hence, both sides of the equation will have a width of 65 bits:

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
        (forall ((x (_ BitVec 64)) (y (_ BitVec 64)))
                (=
                        ((_ zero_extend 1)
                                (bvadd
                                        (bvand x y)
                                        (bvlshr (bvxor x y) (_ bv1 64))
                                )
                        )
                        (bvlshr
                                (bvadd ((_ zero_extend 1) x) ((_ zero_extend 1) y))
                                (_ bv1 65)
                        )
                )
        )
)
(check-sat)
```

Z3 says "sat".

65 bits are enough, because the result of addition of two biggest 64-bit values has width of 65 bits:
`0xFF...FF + 0xFF...FF = 0x1FF...FE`.

As in previous example about XOR equivalent, `(not (= ... ))` and `exists` can also be used here instead of `forall`.

[63] http://aggregate.org/MAGIC/#Average%20of%20Integers

[64] For those who interesting how it works, its mechanics is closely related to the weird XOR alternative we just saw. That's why I placed these two pieces of text one after another.

## 4.4 XOR swapping algorithm

This is well-known XOR swap algorithm (which don't use additional variable). How it works?

```
1  #!/usr/bin/env python
2
3  from z3 import *
4
5  init_X, init_Y=BitVecs('init_X init_Y', 32)
6
7  X, Y=init_X, init_Y
8
9  X=X^Y
10 Y=Y^X
11 X=X^Y
12
13 print "X=", X
14 print "Y=", Y
15
16 s=Solver()
17
18 s.add(init_X^init_Y != X^Y)
19 print s.check()
```

Now we see a final states of X/Y variables:

```
X= init_X ^ init_Y ^ init_Y ^ init_X ^ init_Y
Y= init_Y ^ init_X ^ init_Y
unsat
```

Z3 gave "unsat", meaning, it can't find any counterexample to the last equation (line 18). Hence, the equation is correct and so is the whole algorithm.

### 4.4.1 In SMT-LIB form

```
; tested with Z3 and MK85

; prove that XOR swap algorithm is correct.
; https://en.wikipedia.org/wiki/XOR_swap_algorithm

; initial: X1, Y1
;X2 := X1 XOR Y1
;Y3 := Y1 XOR X2
;X4 := X2 XOR Y3
; prove X1=Y3 and Y1=X4 for all

; must be unsat, of course

; needless to say that other SMT solvers may use simplification to prove this, MK85 can't do it,
; it "proves" on SAT level, by absence of counterexample to the expressions.

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun x1 () (_ BitVec 32))
(declare-fun y1 () (_ BitVec 32))
(declare-fun x2 () (_ BitVec 32))
```

```
(declare-fun y3 () (_ BitVec 32))
(declare-fun x4 () (_ BitVec 32))

(assert (= x2 (bvxor x1 y1)))
(assert (= y3 (bvxor y1 x2)))
(assert (= x4 (bvxor x2 y3)))

(assert (not (and (= x4 y1) (= y3 x1))))

(check-sat)
```

```
; tested with Z3 and MK85

; prove that XOR swap algorithm (using addition/subtraction) is correct.
; https://en.wikipedia.org/wiki/XOR_swap_algorithm

; initial: X1, Y1
;X2 := X1 ADD Y1
;Y3 := X2 SUB Y1
;X4 := X2 SUB Y3
; prove X1=Y3 and Y1=X4 for all

; must be unsat, of course

; needless to say that other SMT solvers may use simplification to prove this, MK85 can'
   t do it,
; it "proves" on SAT level, by absence of counterexample to the expressions.

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun x1 () (_ BitVec 32))
(declare-fun y1 () (_ BitVec 32))
(declare-fun x2 () (_ BitVec 32))
(declare-fun y3 () (_ BitVec 32))
(declare-fun x4 () (_ BitVec 32))

(assert (= x2 (bvadd x1 y1)))
(assert (= y3 (bvsub x2 y1)))
(assert (= x4 (bvsub x2 y3)))

(assert (not (and (= x4 y1) (= y3 x1))))

(check-sat)
```

### 4.5 Simplifying long and messy expressions using Mathematica and Z3

…which can be results of Hex-Rays and/or manual rewriting.

I've added to my RE4B book about Wolfram Mathematica capabilities to minimize expressions [65].

Today I stumbled upon this Hex-Rays output:

```
if ( ( x != 7 || y!=0 ) && (x < 6 || x > 7) )
{
        ...
```

[65] https://github.com/DennisYurichev/RE-for-beginners/blob/cd85356051937e87f90967cc272248084808223b/other/hexrays_EN.tex#L412, https://beginners.re/

```
};
```

Both Mathematica and Z3 (using "simplify" command) can't make it shorter, but I've got gut feeling, that there is something redundant.

Let's take a look at the right part of the expression. If $x$ must be less than 6 *OR* greater than 7, then it can hold any value except 6 *AND* 7, right? So I can rewrite this manually:

```
if ( ( x != 7 || y!=0 ) && x != 6 && x != 7) )
{
        ...
};
```

And this is what Mathematica can simplify:

```
In[]:= BooleanMinimize[(x != 7 || y != 0) && (x != 6 && x != 7)]
Out[]:= x != 6 && x != 7
```

$y$ gets reduced.

But am I really right? And why Mathematica and Z3 didn't simplify this at first place?

I can use Z3 to prove that these expressions are equal to each other:

```
#!/usr/bin/env python

from z3 import *

x=Int('x')
y=Int('y')

s=Solver()

exp1=And(Or(x!=7, y!=0), Or(x<6, x>7))
exp2=And(x!=6, x!=7)

s.add(exp1!=exp2)

print simplify(exp1) # no luck

print s.check()
print s.model()
```

Z3 can't find counterexample, so it says "unsat", meaning, these expressions are equivalent to each other. So I've rewritten this expression in my code, tests has been passed, etc.

Yes, using both Mathematica and Z3 is overkill, and this is basic boolean algebra, but after ~10 hours of sitting at a computer you can make really dumb mistakes, and additional proof your piece of code is correct is never unwanted.

## 4.6 Proving sorting network correctness

Sorting networks are highly popular in electronics, GPGPU and even in SAT encodings: https://en.wikipedia.org/wiki/Sorting_network.

Especially bitonic sorters, which are also sorting networks: https://en.wikipedia.org/wiki/Bitonic_sorter.

Its popularity is probably related to the fact they can be parallelized easily.

They are relatively easy to construct, but, finding a smallest possible is a challenge.

There is a smallest network (only 25 comparators) for 9-channel sorting network:

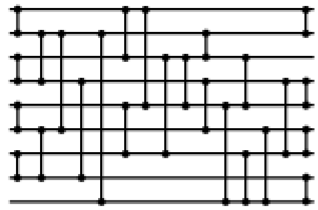


Figure 9: Smallest possible

This is combinational circuit, each connection is a comparator+swapper, it swaps if one of input values is bigger and passes output to the next level.

I copypasted it from the article: Michael Codish, Luı́s Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp – "Twenty-Five Comparators is Optimal when Sorting Nine Inputs (and Twenty-Nine for Ten)".

Another article about it: Ian Parberry – A Computer Assisted Optimal Depth Lower Bound for Nine-Input Sorting Networks.

I don't know (yet) how they proved it, but it's interesting, that it's extremely easy to prove its correctness using Z3 SMT solver. We just construct network out of comparators/swappers and asking Z3 to find counterexample, for which the output of the network will not be sorted. And it can't, meaning, output's state is always sorted, no matter what values are plugged into inputs.

```
from z3 import *

a, b, c, d, e, f, g, h, i=Ints('a b c d e f g h i')

def Z3_min (a, b):
    return If(a<b, a, b)

def Z3_max (a, b):
    return If(a>b, a, b)

def comparator (a, b):
    return (Z3_min(a, b), Z3_max(a, b))

def line(lst, params):
    rt=lst
    start=0
    while start+1 < len(params):
        try:
            first=params.index("+", start)
        except ValueError:
            # no more "+" in parameter string
            return rt
        second=params.index("+", first+1)
        rt[first], rt[second]=comparator(lst[first], lst[second])
        start=second+1
    # parameter string ended
    return rt

l=[i, h, g, f, e, d, c, b, a]
l=line(l, " ++++++++")
l=line(l, " + + + + ")
l=line(l, "   +   + ")
l=line(l, " +   +   ")
l=line(l, "+      + ")
```

```
l=line(l, "  + + + +")
l=line(l, "    +   +")
l=line(l, "  +   +  ")
l=line(l, "    + +  ")
l=line(l, "   + +++ ")
l=line(l, "+   +    ")
l=line(l, "+ + + +  ")
l=line(l, "+  +     ")
l=line(l, "  +  +   ")
l=line(l, "++++++ ++")

# construct expression like And(..., k[2]>=k[1], k[1]>=k[0])
expr=[(l[k+1]>=l[k]) for k in range(len(l)-1)]

# True if everything works correctly:
correct=And(*expr)

s=Solver()

# we want to find inputs for which correct==False:
s.add(Not(correct))
print s.check() # must be unsat
```

( The full source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/proofs/sorting_network/test9.py. )

There is also smaller 4-channel network I copypasted from Wikipedia:

```
...

l=line(l, " + +")
l=line(l, "+ + ")
l=line(l, "++++")
l=line(l, " ++ ")

...
```

( The full source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/proofs/sorting_network/test4.py. )

It also proved to be correct, but it's interesting, what Z3Py expression we've got at each of 4 outputs:

```
If(If(a < c, a, c) < If(b < d, b, d),
   If(a < c, a, c),
   If(b < d, b, d))

If(If(If(a < c, a, c) > If(b < d, b, d),
      If(a < c, a, c),
      If(b < d, b, d)) <
   If(If(a > c, a, c) < If(b > d, b, d),
      If(a > c, a, c),
      If(b > d, b, d)),
   If(If(a < c, a, c) > If(b < d, b, d),
      If(a < c, a, c),
      If(b < d, b, d)),
   If(If(a > c, a, c) < If(b > d, b, d),
      If(a > c, a, c),
      If(b > d, b, d)))

If(If(If(a < c, a, c) > If(b < d, b, d),
```

```
      If(a < c, a, c),
      If(b < d, b, d)) >
   If(If(a > c, a, c) < If(b > d, b, d),
      If(a > c, a, c),
      If(b > d, b, d)),
   If(If(a < c, a, c) > If(b < d, b, d),
      If(a < c, a, c),
      If(b < d, b, d)),
   If(If(a > c, a, c) < If(b > d, b, d),
      If(a > c, a, c),
      If(b > d, b, d)))

If(If(a > c, a, c) > If(b > d, b, d),
   If(a > c, a, c),
   If(b > d, b, d))
```

The first and the last are shorter than the 2nd and the 3rd, they are just $min(min(min(a,b),c),d)$ and $max(max(max(a,b),c),d)$.

## 4.7 ITE example

From Daniel Kroening and Ofer Strichman - "Decision Procedures, An Algorithmic Point of View", 2ed:

```
Problem 2.3 (modeling: program equivalence). Show that the two if-then-else expressions
   below are equivalent:

!(a || b) ? h : !(a == b) ? f : g

!(!a || !b) ? g : (!a && !b) ? h : f

You can assume that the variables have only one bit.
```

```
; tested with MK85 and Z3

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun a () Bool)
(declare-fun b () Bool)
(declare-fun f () Bool)
(declare-fun g () Bool)
(declare-fun h () Bool)

(declare-fun out1 () Bool)
(declare-fun out2 () Bool)

(assert (=(ite (not (or a b)) h (ite (not (= a b)) f g)) out1))
(assert (=(ite (not (or (not a) (not b))) g (ite (and (not a) (not b)) h f)) out2))

; find counterexample:
(assert (distinct out1 out2))

; must be unsat (no counterexample):
(check-sat)
```

### 4.8 Branchless abs()

Prove that branchless abs() function from the Henry Warren 2ed, "2-4 Absolute Value Function" is correct:

```
; tested with Z3 and MK85

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun x () (_ BitVec 32))

; result1=abs(x), my version:
(declare-fun result1 () (_ BitVec 32))

(assert (= result1 (ite (bvslt x #x00000000) (bvneg x) x)))

; from Henry Warren book.
; y = x>>s 31
; result2=(x xor y) - y
(declare-fun y () (_ BitVec 32))
(declare-fun result2 () (_ BitVec 32))

(assert (= y (bvashr x (_ bv31 32))))
(assert (= result2 (bvsub (bvxor x y) y)))

(assert (distinct result1 result2))

; must be unsat:
(check-sat)
```

### 4.9 Proving branchless min/max functions are correct

... from https://graphics.stanford.edu/~seander/bithacks.html#IntegerMinOrMax.

Which are, $min(x,y) = y \oplus ((x \oplus y) \wedge (-(x < y)))$

And $max(x,y) = x \oplus ((x \oplus y) \wedge (-(x < y)))$

```
; tested with MK85 and Z3

; unsigned version
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun x () (_ BitVec 32))
(declare-fun y () (_ BitVec 32))

(declare-fun min1 () (_ BitVec 32))
(declare-fun max1 () (_ BitVec 32))

; this is our min/max functions, "reference" ones:
(assert (= min1 (ite (bvule x y) x y)))
(assert (= max1 (ite (bvuge x y) x y)))

(declare-fun min2 () (_ BitVec 32))
(declare-fun max2 () (_ BitVec 32))

; functions we will "compare" against:
```

```
; y ^ ((x ^ y) & -(x < y)); // min(x, y)
(assert (= min2
        (bvxor
                y
                (bvand
                        (bvxor x y)
                        (bvneg (ite (bvult x y) #x00000001 #x00000000))
                )
        )
))

; x ^ ((x ^ y) & -(x < y)); // max(x, y)
(assert (= max2
        (bvxor
                x
                (bvand
                        (bvxor x y)
                        (bvneg (ite (bvult x y) #x00000001 #x00000000))
                )
        )
))

; find any set of variables for which min1!=min2 or max1!=max2
(assert (or
        (not (= min1 min2))
        (not (= max1 max2))
))

; must be unsat (no counterexample)
(check-sat)
```

```
; tested with MK85 and Z3

; signed version
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun x () (_ BitVec 32))
(declare-fun y () (_ BitVec 32))

(declare-fun min1 () (_ BitVec 32))
(declare-fun max1 () (_ BitVec 32))

; this is our min/max functions, "reference" ones:
(assert (= min1 (ite (bvsle x y) x y)))
(assert (= max1 (ite (bvsge x y) x y)))

(declare-fun min2 () (_ BitVec 32))
(declare-fun max2 () (_ BitVec 32))

; functions we will "compare" against:

; y ^ ((x ^ y) & -(x < y)); // min(x, y)
(assert (= min2
        (bvxor
                y
```

```
                (bvand
                        (bvxor x y)
                        (bvneg (ite (bvslt x y) #x00000001 #x00000000))
                )
        )
))

; x ^ ((x ^ y) & -(x < y)); // max(x, y)
(assert (= max2
        (bvxor
                x
                (bvand
                        (bvxor x y)
                        (bvneg (ite (bvslt x y) #x00000001 #x00000000))
                )
        )
))

; find any set of variables for which min1!=min2 or max1!=max2
(assert (or
        (not (= min1 min2))
        (not (= max1 max2))
))

; must be unsat (no counterexample)
(check-sat)
```

### 4.10 Proving "Determine if a word has a zero byte" bit twiddling hack

... which is:

```
#define haszero(v) (((v) - 0x01010101UL) & ~(v) & 0x80808080UL)
```

( https://graphics.stanford.edu/~seander/bithacks.html#ZeroInWord )

The expression returns zero if there are no zero bytes in 32-bit word, or non-zero, if at least one is present. Here we prove that it's correct for all possible 32-bit words.

```
; checked with Z3 and MK85

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun v () (_ BitVec 32))
(declare-fun out () (_ BitVec 32))

; (((v) - 0x01010101UL) & ~(v) & 0x80808080UL)
(assert (= out (bvand (bvsub v #x01010101) (bvnot v) #x80808080)))

(declare-fun HasZeroByte () Bool)

(assert (= HasZeroByte
        (or
                (= (bvand v #xff000000) #x00000000)
                (= (bvand v #x00ff0000) #x00000000)
                (= (bvand v #x0000ff00) #x00000000)
                (= (bvand v #x000000ff) #x00000000)
        )
```

```
        )
)

; at least one zero byte must be present
(assert HasZeroByte)

; out==0
(assert (= out #x00000000))

; must be unsat (no counterexample)
(check-sat)
```

### 4.11 Arithmetical shift bit twiddling hack

Prove that `((x+0x80000000) >>u n) - (0x80000000 >>u n)` works like arithmetical shift (`bvashr` function in SMT-LIB or `SAR` x86 instruction).

See: Henry Warren 2ed: "2-7 Shift Right Signed from Unsigned".

Also, check if I implemented signed shift right correctly in MK85: https://github.com/DennisYurichev/MK85/blob/834305a9851ec7976946247d42bb13d052aba005/MK85.cc#L1195.

In other words, we prove equivalence of the expression above and my implementation.

```
; tested with MK85 and Z3

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun x () (_ BitVec 32))
(declare-fun n () (_ BitVec 32))
(declare-fun result1 () (_ BitVec 32))
(declare-fun result2 () (_ BitVec 32))

(assert (bvule n (_ bv31 32)))

(assert (= result1 (bvashr x n)))

; ((x+0x80000000) >>u n) - (0x80000000 >>u n)
(assert (= result2
        (bvsub
                (bvlshr (bvadd x #x80000000) n)
                (bvlshr #x80000000 n)
        )
))

(assert (distinct result1 result2))

; must be unsat:
(check-sat)
```

# 5 Regular expressions

## 5.1 KLEE

I've always wanted to generate possible strings for given regular expression. This is not so hard if to dive into regular expression matcher theory and details, but can we force RE matcher to do this?

I took lightest RE engine I've found: https://github.com/cesanta/slre, and wrote this:

```
int main(void)
{
        char s[6];
        klee_make_symbolic(s, sizeof s, "s");
        s[5]=0;
        if (slre_match("^\\d[a-c]+(x|y|z)", s, 5, NULL, 0, 0)==5)
                klee_assert(0);
}
```

So I wanted a string consisting of digit, “a” or “b” or “c” (at least one character) and “x” or “y” or “z” (one character). The whole string must have size of 5 characters.

```
% klee --libc=uclibc slre.bc
...
KLEE: ERROR: /home/klee/slre.c:445: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location
...

% ls klee-last | grep err
test000014.external.err

% ktest-tool --write-ints klee-last/test000014.ktest
ktest file : 'klee-last/test000014.ktest'
args       : ['slre.bc']
num objects: 1
object    0: name: b's'
object    0: size: 6
object    0: data: b'5aaax\xff'
```

This is indeed correct string. “\xff” is at the place where terminal zero byte should be, but RE engine we use ignores the last zero byte, because it has buffer lenght as a passed parameter. Hence, KLEE doesn’t *reconstruct* final byte.

Can we get more? Now we add additional constraint:

```
int main(void)
{
        char s[6];
        klee_make_symbolic(s, sizeof s, "s");
        s[5]=0;
        if (slre_match("^\\d[a-c]+(x|y|z)", s, 5, NULL, 0, 0)==5 &&
                        strcmp(s, "5aaax")!=0)
                klee_assert(0);
}
```

```
% ktest-tool --write-ints klee-last/test000014.ktest
ktest file : 'klee-last/test000014.ktest'
args       : ['slre.bc']
num objects: 1
object    0: name: b's'
object    0: size: 6
object    0: data: b'7aaax\xff'
```

Let’s say, out of whim, we don’t like “a” at the 2nd position (starting at 0th):

```
int main(void)
{
        char s[6];
        klee_make_symbolic(s, sizeof s, "s");
        s[5]=0;
```

```
        if (slre_match("^\\d[a-c]+(x|y|z)", s, 5, NULL, 0, 0)==5 &&
                        strcmp(s, "5aaax")!=0 &&
                        s[2]!='a')
                klee_assert(0);
}
```

KLEE found a way to satisfy our new constraint:

```
% ktest-tool --write-ints klee-last/test000014.ktest
ktest file : 'klee-last/test000014.ktest'
args       : ['slre.bc']
num objects: 1
object    0: name: b's'
object    0: size: 6
object    0: data: b'7abax\xff'
```

Let's also define constraint KLEE cannot satisfy:

```
int main(void)
{
        char s[6];
        klee_make_symbolic(s, sizeof s, "s");
        s[5]=0;
        if (slre_match("^\\d[a-c]+(x|y|z)", s, 5, NULL, 0, 0)==5 &&
                        strcmp(s, "5aaax")!=0 &&
                        s[2]!='a' &&
                        s[2]!='b' &&
                        s[2]!='c')
                klee_assert(0);
}
```

It cannot indeed, and KLEE finished without reporting about `klee_assert()` triggering.

## 5.2 Enumerating all possible inputs for a specific regular expression

Regular expression if first converted to FSM[66] before matching. Hence, many RE[67] libraries has two functions: "compile" and "execute" (when you match many strings against single RE, no need to recompile it to FSM each time).

And I've found this website, which can visualize FSM for a regular expression. http://hokein.github.io/Automata.js/. This is fun!

This FSM (DFA[68]) is for the expression `(dark|light)?(red|blue|green)(ish)?`

[66] Finite State Machine
[67] Regular Expression
[68] Deterministic finite automaton

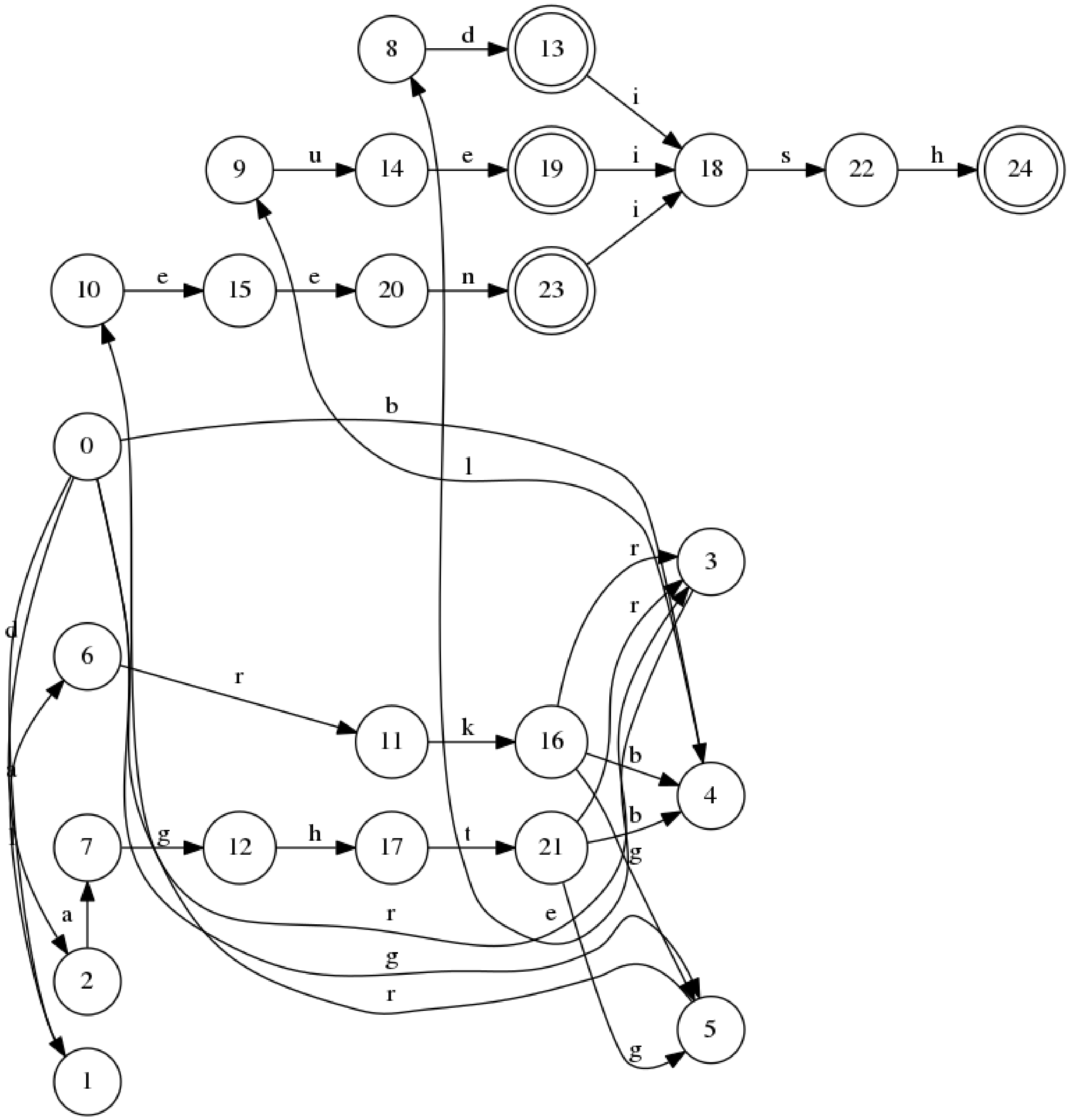



Figure 10:

Another version: https://raw.githubusercontent.com/DennisYurichev/SAT_SMT_by_example/master/regexp/SMT/FSM.png.

Accepting states are in double circles, these are the states where matching process stops.

How can we generate an input string which regular expression would match? In other words, which inputs FSM would accept? This task is surprisingly simple for SMT-solver.

We just define a transition function. For each pair (state, input) it defines new state.

FSM has been visualized by the website mentioned above, and I used this information to write "transition()" function.

Then we chain transition functions... then we try a chain for all lengths in range of 2..14.

```
#!/usr/bin/env python
from MK85 import *

BIT_WIDTH=16
```

```
INVALID_STATE=999
ACCEPTING_STATES=[13, 19, 23, 24]

# st - state
# i - input character
def transition (s, st, i):
    # this is like switch()
    return s.If(And(st==0, i==ord('r')), s.BitVecConst(3, BIT_WIDTH),
    s.If(And(st==0, i==ord('b')), s.BitVecConst(4, BIT_WIDTH),
    s.If(And(st==0, i==ord('g')), s.BitVecConst(5, BIT_WIDTH),
    s.If(And(st==0, i==ord('d')), s.BitVecConst(1, BIT_WIDTH),
    s.If(And(st==0, i==ord('l')), s.BitVecConst(2, BIT_WIDTH),
    s.If(And(st==1, i==ord('a')), s.BitVecConst(6, BIT_WIDTH),
    s.If(And(st==2, i==ord('i')), s.BitVecConst(7, BIT_WIDTH),
    s.If(And(st==3, i==ord('e')), s.BitVecConst(8, BIT_WIDTH),
    s.If(And(st==4, i==ord('l')), s.BitVecConst(9, BIT_WIDTH),
    s.If(And(st==5, i==ord('r')), s.BitVecConst(10, BIT_WIDTH),
    s.If(And(st==6, i==ord('r')), s.BitVecConst(11, BIT_WIDTH),
    s.If(And(st==7, i==ord('g')), s.BitVecConst(12, BIT_WIDTH),
    s.If(And(st==8, i==ord('d')), s.BitVecConst(13, BIT_WIDTH),
    s.If(And(st==9, i==ord('u')), s.BitVecConst(14, BIT_WIDTH),
    s.If(And(st==10, i==ord('e')), s.BitVecConst(15, BIT_WIDTH),
    s.If(And(st==11, i==ord('k')), s.BitVecConst(16, BIT_WIDTH),
    s.If(And(st==12, i==ord('h')), s.BitVecConst(17, BIT_WIDTH),
    s.If(And(st==13, i==ord('i')), s.BitVecConst(18, BIT_WIDTH),
    s.If(And(st==14, i==ord('e')), s.BitVecConst(19, BIT_WIDTH),
    s.If(And(st==15, i==ord('e')), s.BitVecConst(20, BIT_WIDTH),
    s.If(And(st==16, i==ord('r')), s.BitVecConst(3, BIT_WIDTH),
    s.If(And(st==16, i==ord('b')), s.BitVecConst(4, BIT_WIDTH),
    s.If(And(st==16, i==ord('g')), s.BitVecConst(5, BIT_WIDTH),
    s.If(And(st==17, i==ord('t')), s.BitVecConst(21, BIT_WIDTH),
    s.If(And(st==18, i==ord('s')), s.BitVecConst(22, BIT_WIDTH),
    s.If(And(st==19, i==ord('i')), s.BitVecConst(18, BIT_WIDTH),
    s.If(And(st==20, i==ord('n')), s.BitVecConst(23, BIT_WIDTH),
    s.If(And(st==21, i==ord('r')), s.BitVecConst(3, BIT_WIDTH),
    s.If(And(st==21, i==ord('b')), s.BitVecConst(4, BIT_WIDTH),
    s.If(And(st==21, i==ord('g')), s.BitVecConst(5, BIT_WIDTH),
    s.If(And(st==22, i==ord('h')), s.BitVecConst(24, BIT_WIDTH),
    s.If(And(st==23, i==ord('i')), s.BitVecConst(18, BIT_WIDTH),
        s.BitVecConst(INVALID_STATE, 16))))))))))))))))))))))))))))))))))

def print_model(m, length, inputs):
    #print "length=", length
    tmp=""
    for i in range(length-1):
        tmp=tmp+chr(m["inputs_%d" % i])
    print tmp

def make_FSM(length):
    s=MK85(verbose=0)
    states=[s.BitVec('states_%d' % i,BIT_WIDTH) for i in range(length)]
    inputs=[s.BitVec('inputs_%d' % i,BIT_WIDTH) for i in range(length-1)]

    # initial state:
    s.add(states[0]==0)
```

```
    # the last state must be equal to one of the acceping states
    s.add(Or(*[states[length-1]==i for i in ACCEPTING_STATES]))

    # all states are in limits...
    for i in range(length):
        s.add(And(states[i]>=0, states[i]<=24))
        # redundant, though. however, we are not interesting in non-matched inputs,
           right?
        s.add(states[i]!=INVALID_STATE)

    # "insert" transition() functions between subsequent states
    for i in range(length-1):
        s.add(states[i+1] == transition(s, states[i], inputs[i]))

    # enumerate results:
    results=[]
    while s.check():
        m=s.model()
        #print m
        print_model(m, length, inputs)
        # add the current solution negated:
        tmp=[]
        for pair in m:
            tmp.append(s.var_by_name(pair) == m[pair])
        s.add(expr.Not(And(*tmp)))

for l in range(2,15):
    make_FSM(l)
```

Results:

```
red
blue
green
redish
darkred
blueish
darkblue
greenish
lightred
lightblue
darkgreen
lightgreen
darkredish
darkblueish
lightredish
darkgreenish
lightblueish
lightgreenish
```

As simple as this.

The source code for Z3Py: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/regexp/SMT/re_Z3.py.

It can be said, what we did is enumeration of all paths between two vertices of a digraph (representing FSM).

Also, the "transition()" function itself can act as a RE matcher, with no relevance to SMT solver(s). Just feed input characters to it and track state. Whenever you hit one of accepting states, return "match", whenever you hit `INVALID_STATE`, return "no match".

# 6 Gray code

## 6.1 Balanced Gray code and Z3 SMT solver

Suppose, you are making a rotary encoder. This is a device that can signal its angle in some form, like:

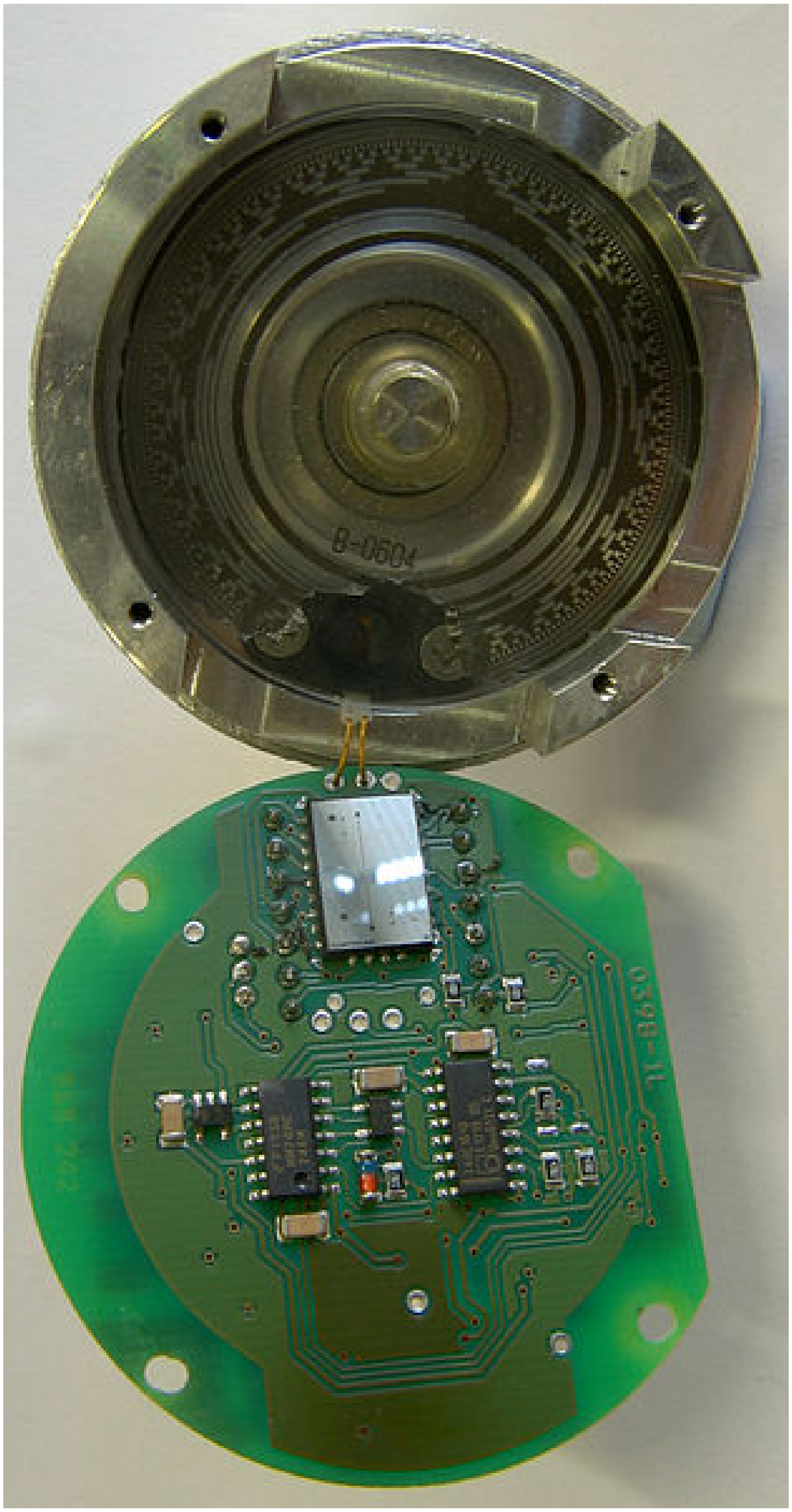



Figure 11: Rotary encoder

( The image has been taken from Wikipedia: `https://en.wikipedia.org/wiki/Gray_code` )
Click on bigger image.
This is a rotary (shaft) encoder: `https://en.wikipedia.org/wiki/Rotary_encoder`.

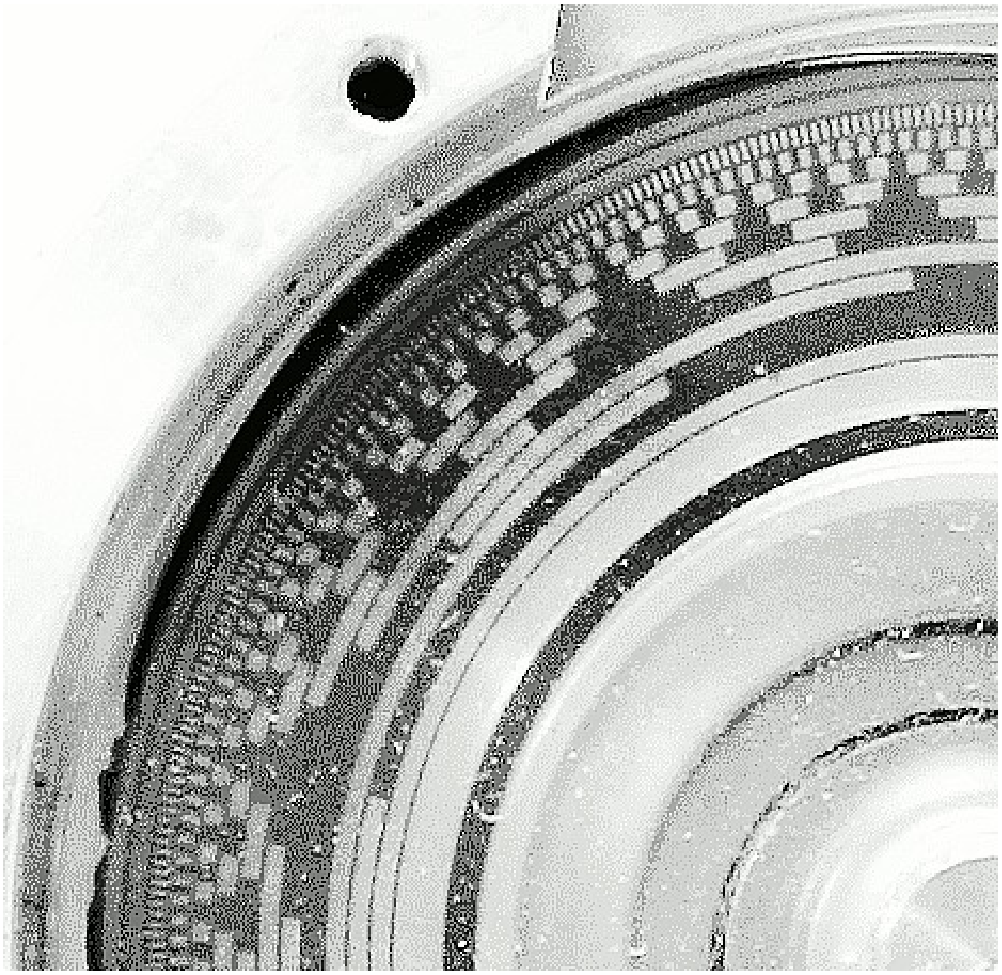


Figure 12: Cropped and photoshopped version

( Source: http://homepages.dordt.edu/~ddeboer//S10/304/c_at_d/304S10_RC_TRK.HTM )

Click on bigger one.

There are pins and tracks on rotating wheel. How would you do this? Easiest way is to use binary code. But it has a problem: when a wheel is rotating, in a moment of transition from one state to another, several bits may be changed, hence, undesirable state may be present for a short period of time. This is bad. To deal with it, Gray code was invented: only 1 bit is changed during rotation. Like:

```
Decimal Binary  Gray
0       0000    0000
1       0001    0001
2       0010    0011
3       0011    0010
4       0100    0110
5       0101    0111
6       0110    0101
7       0111    0100
8       1000    1100
9       1001    1101
10      1010    1111
11      1011    1110
12      1100    1010
13      1101    1011
14      1110    1001
15      1111    1000
```

Now the second problem. Look at the picture again. It has a lot of bit changes on the outer circles. And this is electromechanical device. Surely, you may want to make tracks as long as possible, to reduce wearing of both tracks and pins. This is a first problem. The second: wearing should be even across all tracks (this is balanced Gray code).

How we can find a table for all states using Z3:

```
#!/usr/bin/env python3

from z3 import *

BITS=5

# how many times a run of bits for each bit can be changed (max).
# it can be 4 for 4-bit Gray code or 8 for 5-bit code.
# 12 for 6-bit code (maybe even less)
CHANGES_MAX=8

ROWS=2**BITS
MASK=ROWS-1 # 0x1f for 5 bits, 0xf for 4 bits, etc

def bool_to_int (b):
    if b==True:
        return 1
    return 0

s=Solver()

# add a constraint: Hamming distance between two bitvectors must be 1
# i.e., two bitvectors can differ in only one bit.
# for 4 bits it works like that:
#    s.add(Or(
#        And(a3!=b3,a2==b2,a1==b1,a0==b0),
#        And(a3==b3,a2!=b2,a1==b1,a0==b0),
#        And(a3==b3,a2==b2,a1!=b1,a0==b0),
#        And(a3==b3,a2==b2,a1==b1,a0!=b0)))
def hamming1(l1, l2):
    assert len(l1)==len(l2)
    r=[]
    for i in range(len(l1)):
        t=[]
        for j in range(len(l1)):
            if i==j:
                t.append(l1[j]!=l2[j])
            else:
                t.append(l1[j]==l2[j])
        r.append(And(t))
    s.add(Or(r))

# add a constraint: bitvectors must be different.
# for 4 bits works like this:
#    s.add(Or(a3!=b3, a2!=b2, a1!=b1, a0!=b0))
def not_eq(l1, l2):
    assert len(l1)==len(l2)
    t=[l1[i]!=l2[i] for i in range(len(l1))]
    s.add(Or(t))

code=[[Bool('code_%d_%d' % (r,c)) for c in range(BITS)] for r in range(ROWS)]
```

```
ch=[[Bool('ch_%d_%d' % (r,c)) for c in range(BITS)] for r in range(ROWS)]

# each rows must be different from a previous one and a next one by 1 bit:
for i in range(ROWS):
    # get bits of the current row:
    lst1=[code[i][bit] for bit in range(BITS)]
    # get bits of the next row.
    # important: if the current row is the last one, (last+1)&MASK==0, so we overlap
        here:
    lst2=[code[(i+1)&MASK][bit] for bit in range(BITS)]
    hamming1(lst1, lst2)

# no row must be equal to any another row:
for i in range(ROWS):
    for j in range(ROWS):
        if i==j:
            continue
        lst1=[code[i][bit] for bit in range(BITS)]
        lst2=[code[j][bit] for bit in range(BITS)]
        not_eq(lst1, lst2)

# 1 in ch[] table means that run of 1's has been changed to run of 0's, or back.
# "run" change detected using simple XOR:
for i in range(ROWS):
    for bit in range(BITS):
        # row overlapping works here as well:
        s.add(ch[i][bit]==Xor(code[i][bit],code[(i+1)&MASK][bit]))

# only CHANGES_MAX of 1 bits is allowed in ch[] table for each bit:
for bit in range(BITS):
    t=[ch[i][bit] for i in range(ROWS)]
    # this is a dirty hack.
    # AtMost() takes arguments like:
    # AtMost(v1, v2, v3, v4, 2) <- this means, only 2 booleans (or less) from the list
        can be True.
    # but we need to pass a list here.
    # so a CHANGES_MAX number is appended to a list and a new list is then passed as
        arguments list:
    s.add(AtMost(*(t+[CHANGES_MAX])))

result=s.check()
if result==unsat:
    exit(0)
m=s.model()

# get the model.

print ("code table:")

for i in range(ROWS):
    t=""
    for bit in range(BITS):
        # comma at the end means "no newline":
        t=t+str(bool_to_int(is_true(m[code[i][BITS-1-bit]])))+" "
    print (t)
```

```
print ("ch table:")

stat={}

for i in range(ROWS):
    t=""
    for bit in range(BITS):
        x=is_true(m[ch[i][BITS-1-bit]])
        if x:
            # increment if bit is present in dict, set 1 if not present
            stat[bit]=stat.get(bit, 0)+1
        # comma at the end means "no newline":
        t=t+str(bool_to_int(x))+" "
    print (t)

print ("stat (bit number: number of changes): ", stat)
```

( The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/gray_code/SMT/gray.py )

For 4 bits, 4 changes is enough:

```
code table:
0 1 0 1
0 0 0 1
0 0 1 1
0 0 1 0
1 0 1 0
1 0 1 1
1 1 1 1
1 1 0 1
1 0 0 1
1 0 0 0
0 0 0 0
0 1 0 0
1 1 0 0
1 1 1 0
0 1 1 0
0 1 1 1
ch table:
0 1 0 0
0 0 1 0
0 0 0 1
1 0 0 0
0 0 0 1
0 1 0 0
0 0 1 0
0 1 0 0
0 0 0 1
1 0 0 0
0 1 0 0
1 0 0 0
0 0 1 0
1 0 0 0
0 0 0 1
0 0 1 0
stat (bit number: count of changes):  {0: 4, 1: 4, 2: 4, 3: 4}
```

8 changes for 5 bits: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/gray_code/SMT/5.txt. 12 for 6 bits (or maybe even less): https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/gray_code/SMT/6.txt.

### 6.1.1 Duke Nukem 3D from 1990s

Figure 13: Duke Nukem 3D

Another application of Gray code:

```
with.inspiring@flair.and.erudition (Mike Naylor) wrote:

>In Duke Nukem, you often come upon panels that have four buttons in a row,
>all in their "off" position. Each time you "push" a button, it toggles from
>one state to the other. The object is to find the unique combination that
>unlocks something in the game.

>My question is: What is the most efficient order in which to push the
>buttons so that every combination is tested with no wasted effort?

A Gray Code. :-)

(Oh, you wanted to know what one would be?  How about:
0000
0001
0011
```

```
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1000
1001
1011

Or, if you prefer, with buttons A,B,C,D: D,C,D,B,D,C,D,A,D,C,D,B,C,D,C
It isn't the "canonical" Gray code (or if it is, it is by Divine
Providence), but it works.

Douglas Limmer -- lim...@math.orst.edu
"No wonder these mathematical wizards were nuts - went off the beam -
he'd be pure squirrel-food if he had half that stuff in _his_ skull!"
E. E. "Doc" Smith, _Second Stage Lensmen_
```

( https://groups.google.com/forum/#!topic/rec.puzzles/Dh2H-pGJcbI )

Obviously, using our solution, you can minimize all movements in this ancient videogame, for 4 switches, that would be 4*4=16 switches. With our solution (balanced Gray code), wearing would be even across all 4 switches.

## 6.2 Gray code in MaxSAT

This is remake of gray code generator for Z3 (6.1).

Here is also *ch[]* table, but we add soft clauses for it here. The goal is to make as many *False*'s in *ch[]* table, as possible.

```
#!/usr/bin/env python3

import subprocess, os, itertools
import my_utils, SAT_lib

BITS=5

# how many times a run of bits for each bit can be changed (max).
# it can be 4 for 4-bit Gray code or 8 for 5-bit code.
# 12 for 6-bit code (maybe even less)

ROWS=2**BITS
MASK=ROWS-1 # 0x1f for 5 bits, 0xf for 4 bits, etc

def do_all():
    s=SAT_lib.SAT_lib(maxsat=True)

    code=[s.alloc_BV(BITS) for r in range(ROWS)]
    ch=[s.alloc_BV(BITS) for r in range(ROWS)]

    # each rows must be different from a previous one and a next one by 1 bit:
    for i in range(ROWS):
        # get bits of the current row:
        lst1=[code[i][bit] for bit in range(BITS)]
        # get bits of the next row.
```

```
        # important: if the current row is the last one, (last+1)&MASK==0, so we overlap
            here:
        lst2=[code[(i+1)&MASK][bit] for bit in range(BITS)]
        s.hamming1(lst1, lst2)

    # no row must be equal to any another row:
    for i in range(ROWS):
        for j in range(ROWS):
            if i==j:
                continue
            lst1=[code[i][bit] for bit in range(BITS)]
            lst2=[code[j][bit] for bit in range(BITS)]
            s.fix_BV_NEQ(lst1, lst2)

    # 1 in ch[] table means that run of 1's has been changed to run of 0's, or back.
    # "run" change detected using simple XOR:
    for i in range(ROWS):
        for bit in range(BITS):
            # row overlapping works here as well.
            # we add here "soft" constraint with weight=1:
            s.fix_soft(s.EQ(ch[i][bit], s.XOR(code[i][bit],code[(i+1)&MASK][bit])),
                False, weight=1)

    if s.solve()==False:
        print ("unsat")
        exit(0)

    print ("code table:")

    for i in range(ROWS):
        tmp=""
        for bit in range(BITS):
            t=s.get_var_from_solution(code[i][BITS-1-bit])
            if t:
                tmp=tmp+"*"
            else:
                tmp=tmp+" "
        print (tmp)

    # get statistics:
    stat={}

    for i in range(ROWS):
        for bit in range(BITS):
            x=s.get_var_from_solution(ch[i][BITS-1-bit])
            if x==0:
                # increment if bit is present in dict, set 1 if not present
                stat[bit]=stat.get(bit, 0)+1

    print ("stat (bit number: number of changes): ")
    print (stat)

do_all()
```

So it does, for 5-bit Gray code:

```
code table:
```

```
 ****
*****
* ***
  ***
  **
 ***
 **
 *
**
**  *
*** *
***
****
* **
*  *
*
*   *
* * *
* *
  *
  * *
    *

   *
   **
*  **
** **
** *
 * *
 * **
 *  *
 ** *
stat (bit number: number of changes):
{0: 6, 1: 4, 2: 6, 3: 6, 4: 10}
```

# 7 Recreational mathematics and puzzles

## 7.1 Sudoku

Sudoku puzzle is a 9*9 grid with some cells filled with values, some are empty:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 5 | 3 | | | | | |
| 8 | | | | | | | 2 | |
| | 7 | | | 1 | | 5 | | |
| 4 | | | | | 5 | 3 | | |
| | 1 | | | 7 | | | | 6 |
| | | 3 | 2 | | | | 8 | |
| | 6 | | 5 | | | | | 9 |
| | | 4 | | | | | 3 | |
| | | | | | 9 | 7 | | |

Unsolved Sudoku

Numbers of each row must be unique, i.e., it must contain all 9 numbers in range of 1..9 without repetition. Same story for each column and also for each 3*3 square.

This puzzle is good candidate to try SMT solver on, because it's essentially an unsolved system of equations.

### 7.1.1 Simple sudoku in SMT

**The first idea** The only thing we must decide is that how to determine in one expression, if the input 9 variables has all 9 unique numbers? They are not ordered or sorted, after all.

From the school-level arithmetics, we can devise this idea:

$$\underbrace{10^{i_1} + 10^{i_2} + \cdots + 10^{i_9}}_{9} = 1111111110 \tag{1}$$

Take each input variable, calculate $10^i$ and sum them all. If all input values are unique, each will be settled at its own place. Even more than that: there will be no holes, i.e., no skipped values. So, in case of Sudoku, 1111111110 number will be final result, indicating that all 9 input values are unique, in range of 1..9.

Exponentiation is heavy operation, can we use binary operations? Yes, just replace 10 with 2:

$$\underbrace{2^{i_1} + 2^{i_2} + \cdots + 2^{i_9}}_{9} = 1111111110_2 \tag{2}$$

The effect is just the same, but the final value is in base 2 instead of 10.

Now a working example:

```
#!/usr/bin/env python

import sys
from z3 import *

"""
coordinates:
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
```

```
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
"""

s=Solver()

# using Python list comprehension, construct array of arrays of BitVec instances:
cells=[[BitVec('cell%d%d' % (r, c), 16) for c in range(9)] for r in range(9)]

# http://www.norvig.com/sudoku.html
# http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
puzzle="
    ..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97.."

# process text line:
current_column=0
current_row=0
for i in puzzle:
        if i!='.':
                s.add(cells[current_row][current_column]==BitVecVal(int(i),16))
        current_column=current_column+1
        if current_column==9:
                current_column=0
                current_row=current_row+1

one=BitVecVal(1,16)
mask=BitVecVal(0b1111111110,16)

# for all 9 rows
for r in range(9):
        s.add(((one<<cells[r][0]) +
                (one<<cells[r][1]) +
                (one<<cells[r][2]) +
                (one<<cells[r][3]) +
                (one<<cells[r][4]) +
                (one<<cells[r][5]) +
                (one<<cells[r][6]) +
                (one<<cells[r][7]) +
                (one<<cells[r][8]))==mask)

# for all 9 columns
for c in range(9):
        s.add(((one<<cells[0][c]) +
                (one<<cells[1][c]) +
                (one<<cells[2][c]) +
                (one<<cells[3][c]) +
                (one<<cells[4][c]) +
                (one<<cells[5][c]) +
                (one<<cells[6][c]) +
                (one<<cells[7][c]) +
                (one<<cells[8][c]))==mask)

# enumerate all 9 squares
for r in range(0, 9, 3):
        for c in range(0, 9, 3):
```

```
                # add constraints for each 3*3 square:
                        s.add((one<<cells[r+0][c+0]) +
                                (one<<cells[r+0][c+1]) +
                                (one<<cells[r+0][c+2]) +
                                (one<<cells[r+1][c+0]) +
                                (one<<cells[r+1][c+1]) +
                                (one<<cells[r+1][c+2]) +
                                (one<<cells[r+2][c+0]) +
                                (one<<cells[r+2][c+1]) +
                                (one<<cells[r+2][c+2])==mask)

print s.check()
#print s.model()
m=s.model()

for r in range(9):
        for c in range(9):
                sys.stdout.write (str(m[cells[r][c]])+" ")
        print ""
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/sudoku/1/sudoku_plus_Z3.py )

```
% time python sudoku_plus_Z3.py
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4

real    0m11.717s
user    0m10.896s
sys     0m0.068s
```

Even more, we can replace summing operation to logical OR:

$$\underbrace{2^{i_1} \vee 2^{i_2} \vee \cdots \vee 2^{i_9}}_{9} = 1111111110_2 \tag{3}$$

```
#!/usr/bin/env python

import sys
from z3 import *

"""
coordinates:
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
```

```
-----------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
-----------------------------
"""

s=Solver()

# using Python list comprehension, construct array of arrays of BitVec instances:
cells=[[BitVec('cell%d%d' % (r, c), 16) for c in range(9)] for r in range(9)]

# http://www.norvig.com/sudoku.html
# http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
puzzle="
    ..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97.."

# process text line:
current_column=0
current_row=0
for i in puzzle:
        if i!='.':
                s.add(cells[current_row][current_column]==BitVecVal(int(i),16))
        current_column=current_column+1
        if current_column==9:
                current_column=0
                current_row=current_row+1

one=BitVecVal(1,16)
mask=BitVecVal(0b1111111110,16)

# for all 9 rows
for r in range(9):
        s.add(((one<<cells[r][0]) |
                (one<<cells[r][1]) |
                (one<<cells[r][2]) |
                (one<<cells[r][3]) |
                (one<<cells[r][4]) |
                (one<<cells[r][5]) |
                (one<<cells[r][6]) |
                (one<<cells[r][7]) |
                (one<<cells[r][8]))==mask)

# for all 9 columns
for c in range(9):
        s.add(((one<<cells[0][c]) |
                (one<<cells[1][c]) |
                (one<<cells[2][c]) |
                (one<<cells[3][c]) |
                (one<<cells[4][c]) |
                (one<<cells[5][c]) |
                (one<<cells[6][c]) |
                (one<<cells[7][c]) |
                (one<<cells[8][c]))==mask)

# enumerate all 9 squares
```

```
for r in range(0, 9, 3):
        for c in range(0, 9, 3):
                # add constraints for each 3*3 square:
                        s.add(one<<cells[r+0][c+0] |
                                one<<cells[r+0][c+1] |
                                one<<cells[r+0][c+2] |
                                one<<cells[r+1][c+0] |
                                one<<cells[r+1][c+1] |
                                one<<cells[r+1][c+2] |
                                one<<cells[r+2][c+0] |
                                one<<cells[r+2][c+1] |
                                one<<cells[r+2][c+2]==mask)

print s.check()
#print s.model()
m=s.model()

for r in range(9):
        for c in range(9):
                sys.stdout.write (str(m[cells[r][c]])+" ")
        print ""
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/sudoku/1/sudoku_or_Z3.py )

Now it works much faster. Z3 handles OR operation over bit vectors better than addition?

```
% time python sudoku_or_Z3.py
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4

real    0m1.429s
user    0m1.393s
sys     0m0.036s
```

The puzzle I used as example is dubbed as one of the hardest known [69] (well, for humans). It took ≈ 1.4 seconds on my Intel Core i3-3110M 2.4GHz notebook to solve it.

**The second idea** My first approach is far from effective, I did what first came to my mind and worked. Another approach is to use `distinct` command from SMT-LIB, which tells Z3 that some variables must be distinct (or unique). This command is also available in Z3 Python interface.

I've rewritten my first Sudoku solver, now it operates over *Int sort*, it has `distinct` commands instead of bit operations, and now also other constaint added: each cell value must be in 1..9 range, because, otherwise, Z3 will offer (although correct) solution with too big and/or negative numbers.

```
#!/usr/bin/env python

import sys
from z3 import *

"""
```

[69] http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294

```
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
"""

s=Solver()

# using Python list comprehension, construct array of arrays of BitVec instances:
cells=[[Int('cell%d%d' % (r, c)) for c in range(9)] for r in range(9)]

# http://www.norvig.com/sudoku.html
# http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
puzzle="
    ..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97.."

# process text line:
current_column=0
current_row=0
for i in puzzle:
        if i!='.':
                s.add(cells[current_row][current_column]==int(i))
        current_column=current_column+1
        if current_column==9:
                current_column=0
                current_row=current_row+1

# this is important, because otherwise, Z3 will report correct solutions with too big
    and/or negative numbers in cells
for r in range(9):
        for c in range(9):
                s.add(cells[r][c]>=1)
                s.add(cells[r][c]<=9)

# for all 9 rows
for r in range(9):
        s.add(Distinct(cells[r][0],
                cells[r][1],
                cells[r][2],
                cells[r][3],
                cells[r][4],
                cells[r][5],
                cells[r][6],
                cells[r][7],
                cells[r][8]))

# for all 9 columns
```

```
for c in range(9):
        s.add(Distinct(cells[0][c],
                cells[1][c],
                cells[2][c],
                cells[3][c],
                cells[4][c],
                cells[5][c],
                cells[6][c],
                cells[7][c],
                cells[8][c]))

# enumerate all 9 squares
for r in range(0, 9, 3):
        for c in range(0, 9, 3):
                # add constraints for each 3*3 square:
                        s.add(Distinct(cells[r+0][c+0],
                                cells[r+0][c+1],
                                cells[r+0][c+2],
                                cells[r+1][c+0],
                                cells[r+1][c+1],
                                cells[r+1][c+2],
                                cells[r+2][c+0],
                                cells[r+2][c+1],
                                cells[r+2][c+2]))

print s.check()
#print s.model()
m=s.model()

for r in range(9):
        for c in range(9):
                sys.stdout.write (str(m[cells[r][c]])+" ")
        print ""
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/sudoku/1/sudoku2_Z3.py )

```
% time python sudoku2_Z3.py
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4

real    0m0.382s
user    0m0.346s
sys     0m0.036s
```

That's much faster.

**Conclusion** SMT-solvers are so helpful, is that our Sudoku solver has nothing else, we have just defined relationships between variables (cells).

**Homework** As it seems, true Sudoku puzzle is the one which has only one solution. The piece of code I've included here shows only the first one. Using the method described earlier (3.15, also called "model counting"), try to find more solutions, or prove that the solution you have just found is the only one possible.

**Further reading** http://www.norvig.com/sudoku.html

**Sudoku as a SAT problem** It's also possible to represend Sudoku puzzle as a huge CNF equation and use SAT-solver to find solution, but it's just trickier.

Some articles about it: *Building a Sudoku Solver with SAT*[70], Tjark Weber, *A SAT-based Sudoku Solver*[71], Ines Lynce, Joel Ouaknine, *Sudoku as a SAT Problem*[72], Gihwon Kwon, Himanshu Jain, *Optimized CNF Encoding for Sudoku Puzzles*[73].

SMT-solver can also use SAT-solver in its core, so it does all mundane translating work. As a "compiler", it may not do this in the most efficient way, though.

### 7.1.2 Greater Than Sudoku

I've found this on http://www.killersudokuonline.com:

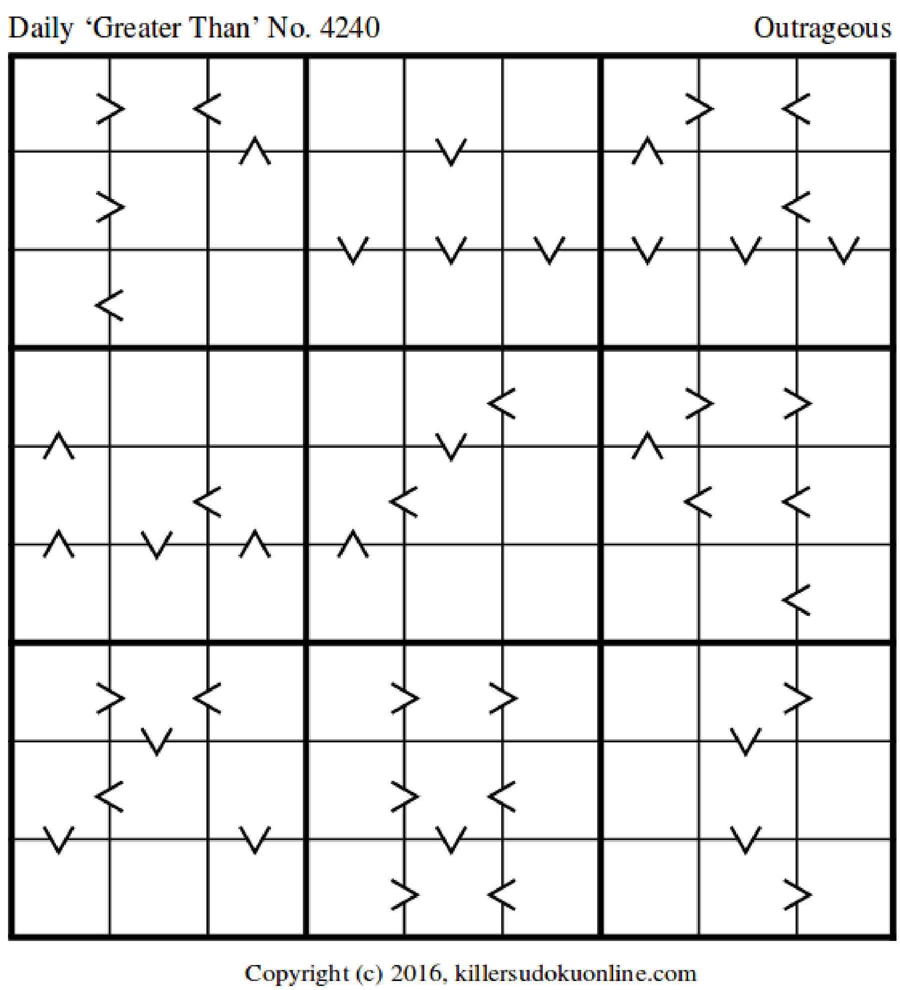



Figure 14:

[70] https://dspace.mit.edu/bitstream/handle/1721.1/106923/6-005-fall-2011/contents/assignments/MIT6_005F11_ps4.pdf
[71] https://www.lri.fr/~conchon/mpri/weber.pdf
[72] http://sat.inesc-id.pt/~ines/publications/aimath06.pdf
[73] http://www.cs.cmu.edu/~hjain/papers/sudoku-as-SAT.pdf

It can be solved easily with Z3. I've took the same piece of code I used for the usual Sudoku: ().

... and added this:

```
...

"""
Subsquares:

------------------------------
         |          |
 1,1     | 1,2      | 1,3
         |          |
------------------------------
         |          |
 2,1     | 2,2      | 2,3
         |          |
------------------------------
         |          |
 3,1     | 3,2      | 3,3
         |          |
------------------------------
"""

# from http://www.killersudokuonline.com/puzzles/2017/puzzle-GD4hzi164344.pdf

# subsquare 1,1:
s.add(cells[0][0]>cells[0][1])
s.add(cells[1][0]>cells[1][1])
s.add(cells[2][0]<cells[2][1])

s.add(cells[0][1]<cells[0][2])
s.add(cells[0][2]<cells[1][2])

# subsquare 1,2:
s.add(cells[0][4]>cells[1][4])
s.add(cells[1][3]>cells[2][3])
s.add(cells[1][4]>cells[2][4])
s.add(cells[1][5]>cells[2][5])

# subsquare 1,3:
s.add(cells[0][6]>cells[0][7])
s.add(cells[0][7]<cells[0][8])
s.add(cells[0][6]<cells[1][6])
s.add(cells[1][7]<cells[1][8])
s.add(cells[1][6]>cells[2][6])
s.add(cells[1][7]>cells[2][7])
s.add(cells[1][8]>cells[2][8])

# subsquare 2,1:
s.add(cells[3][0]<cells[4][0])
s.add(cells[4][0]<cells[5][0])
s.add(cells[4][1]<cells[4][2])
s.add(cells[4][0]<cells[5][0])
s.add(cells[4][1]>cells[5][1])
s.add(cells[4][2]<cells[5][2])

# subsquare 2,2:
```

```
s.add(cells[3][4]>cells[4][4])
s.add(cells[3][4]<cells[3][5])
s.add(cells[4][3]<cells[4][4])
s.add(cells[4][3]<cells[5][3])

# subsquare 2,3:
s.add(cells[3][6]>cells[3][7])
s.add(cells[3][7]>cells[3][8])
s.add(cells[3][6]>cells[4][6])
s.add(cells[4][6]<cells[4][7])
s.add(cells[4][7]<cells[4][8])
s.add(cells[5][7]<cells[5][8])

# subsquare 3,1:
s.add(cells[6][0]>cells[6][1])
s.add(cells[6][1]<cells[6][2])
s.add(cells[6][1]>cells[7][1])
s.add(cells[7][0]<cells[7][1])
s.add(cells[7][0]>cells[8][0])
s.add(cells[7][2]>cells[8][2])

# subsquare 3,2:
s.add(cells[6][3]>cells[6][4])
s.add(cells[6][4]>cells[6][5])
s.add(cells[7][3]>cells[7][4])
s.add(cells[7][4]<cells[7][5])
s.add(cells[8][3]>cells[8][4])
s.add(cells[8][4]<cells[8][5])
s.add(cells[7][4]>cells[8][4])

# subsquare 3,3:
s.add(cells[6][7]>cells[6][8])
s.add(cells[6][7]>cells[7][7])
s.add(cells[7][7]>cells[8][7])
s.add(cells[8][7]>cells[8][8])

...
```

( The full file: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/sudoku/GT/sudoku_GT.py )

The puzzle marked as "Outrageous" (for humans?), however it took ≈ 30 seconds on my old Intel Xeon E3-1220 3.10GHz to solve it:

```
7 3 4 6 9 2 5 1 8
2 1 5 8 3 7 9 4 6
6 8 9 5 1 4 7 2 3
1 7 3 2 8 9 6 5 4
5 4 6 1 7 3 2 8 9
9 2 8 4 5 6 1 3 7
8 6 7 3 2 1 4 9 5
4 5 2 9 6 8 3 7 1
3 9 1 7 4 5 8 6 2
```

### 7.1.3 Solving Killer Sudoku

I've found this on https://krazydad.com/killersudoku/sfiles/KD_Killer_ST16_8_v52.pdf:

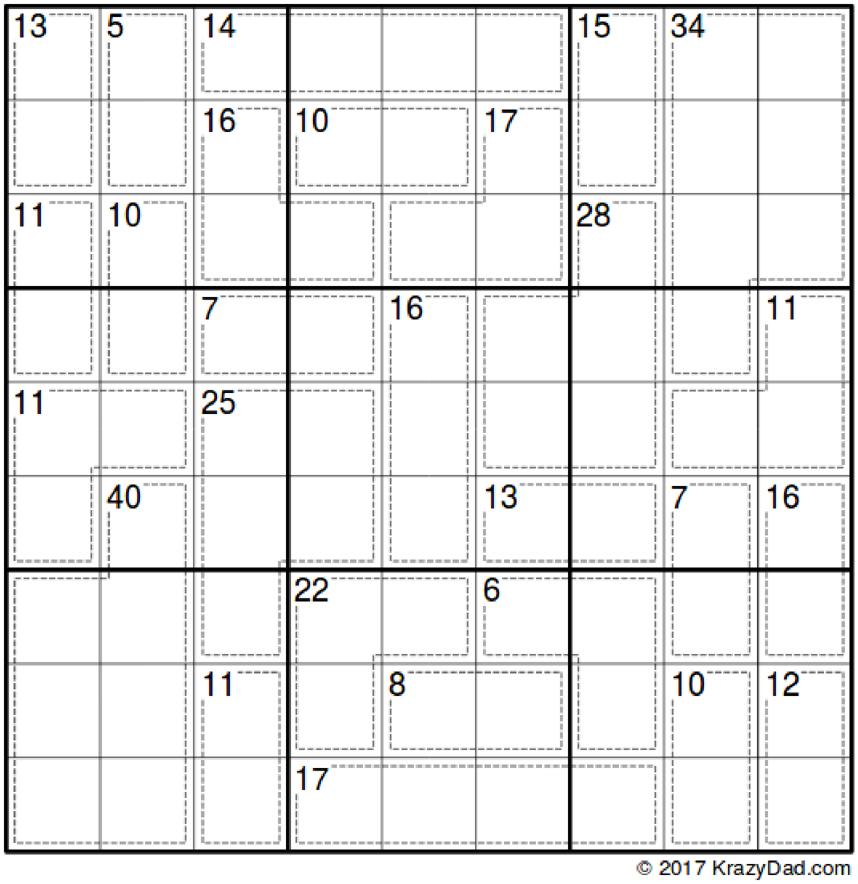



Figure 15:

There are “cages”, each cage must have distinct digits, and its sum must be equal to the number written there in a manner of crossword. See also: https://en.wikipedia.org/wiki/Killer_sudoku.

This is also piece of cake for Z3. I’ve took the same piece of code I used for usual Sudoku ( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/sudoku/1/sudoku2_Z3.py ).

```
...

cage=[cells[0][0], cells[1][0]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==13)

cage=[cells[0][1], cells[1][1]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==5)

cage=[cells[0][2], cells[0][3], cells[0][4], cells[0][5]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==14)

cage=[cells[0][6], cells[1][6]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==15)

cage=[cells[0][7], cells[0][8], cells[1][7], cells[1][8], cells[2][7], cells[2][8],
   cells[3][7]]
```

```
s.add(Distinct(*cage))
s.add(Sum(*cage)==34)

cage=[cells[1][2], cells[2][2], cells[2][3]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==16)

cage=[cells[1][3], cells[1][4]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==10)

cage=[cells[1][5], cells[2][4], cells[2][5]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==17)

cage=[cells[2][6], cells[3][5], cells[3][6], cells[4][5], cells[4][6]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==28)

cage=[cells[3][2], cells[3][3]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==7)

cage=[cells[3][4], cells[4][4], cells[5][4]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==16)

cage=[cells[3][8], cells[4][7], cells[4][8]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==11)

cage=[cells[4][0], cells[4][1], cells[5][0]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==11)

cage=[cells[4][2], cells[4][3], cells[5][2], cells[5][3], cells[6][2]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==25)

cage=[cells[5][1], cells[6][0], cells[6][1], cells[7][0], cells[7][1], cells[8][0],
    cells[8][1]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==40)

cage=[cells[5][5], cells[5][6]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==13)

cage=[cells[5][7], cells[6][7]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==7)

cage=[cells[5][8], cells[6][8]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==16)
```

```
cage=[cells[6][3], cells[6][4], cells[7][3]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==22)

cage=[cells[6][5], cells[6][6], cells[7][6]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==6)

cage=[cells[7][2], cells[8][2]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==11)

cage=[cells[7][4], cells[7][5]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==8)

cage=[cells[7][7], cells[8][7]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==10)

cage=[cells[7][8], cells[8][8]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==12)

cage=[cells[8][3], cells[8][4], cells[8][5], cells[8][6]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==17)

...
```

( The full file: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/sudoku/killer/killer_sudoku.py )

The puzzle marked as "Super-Tough Killer Sudoku Puzzle" (again, for humans?), however it took ≈ 30 seconds on my old Intel Xeon E3-1220 3.10GHz to solve it:

```
5 3 4 7 1 2 8 9 6
8 2 1 4 6 9 7 5 3
9 6 7 8 3 5 4 2 1
2 4 6 1 9 7 3 8 5
7 1 9 3 5 8 6 4 2
3 8 5 6 2 4 9 1 7
4 7 2 5 8 3 1 6 9
6 5 8 9 7 1 2 3 4
1 9 3 2 4 6 5 7 8
```

#### 7.1.4 KLEE

I've also rewritten Sudoku example (7.1.1) for KLEE:

```
#include <stdint.h>

/*
coordinates:
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
```

```
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
*/

uint8_t cells[9][9];

// http://www.norvig.com/sudoku.html
// http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
char *puzzle="..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97..";

int main()
{
        klee_make_symbolic(cells, sizeof cells, "cells");

        // process text line:
        for (int row=0; row<9; row++)
                for (int column=0; column<9; column++)
                {
                        char c=puzzle[row*9 + column];
                        if (c!='.')
                        {
                                if (cells[row][column]!=c-'0') return 0;
                        }
                        else
                        {
                                // limit cells values to 1..9:
                                if (cells[row][column]<1) return 0;
                                if (cells[row][column]>9) return 0;
                        };
                };

        // for all 9 rows
        for (int row=0; row<9; row++)
        {

                if (((1<<cells[row][0]) |
                        (1<<cells[row][1]) |
                        (1<<cells[row][2]) |
                        (1<<cells[row][3]) |
                        (1<<cells[row][4]) |
                        (1<<cells[row][5]) |
                        (1<<cells[row][6]) |
                        (1<<cells[row][7]) |
                        (1<<cells[row][8]))!=0x3FE ) return 0; // 11 1111 1110
        };

        // for all 9 columns
```

```
        for (int c=0; c<9; c++)
        {
                if (((1<<cells[0][c]) |
                        (1<<cells[1][c]) |
                        (1<<cells[2][c]) |
                        (1<<cells[3][c]) |
                        (1<<cells[4][c]) |
                        (1<<cells[5][c]) |
                        (1<<cells[6][c]) |
                        (1<<cells[7][c]) |
                        (1<<cells[8][c]))!=0x3FE ) return 0; // 11 1111 1110
        };

        // enumerate all 9 squares
        for (int r=0; r<9; r+=3)
                for (int c=0; c<9; c+=3)
                {
                        // add constraints for each 3*3 square:
                        if ((1<<cells[r+0][c+0] |
                                1<<cells[r+0][c+1] |
                                1<<cells[r+0][c+2] |
                                1<<cells[r+1][c+0] |
                                1<<cells[r+1][c+1] |
                                1<<cells[r+1][c+2] |
                                1<<cells[r+2][c+0] |
                                1<<cells[r+2][c+1] |
                                1<<cells[r+2][c+2])!=0x3FE ) return 0; // 11 1111 1110
                };

        // at this point, all constraints must be satisfied
        klee_assert(0);
};
```

Let's run it:

```
% clang -emit-llvm -c -g klee_sudoku_or1.c
...

\$ time klee klee_sudoku_or1.bc
KLEE: output directory is "/home/klee/klee-out-98"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_sudoku_or1.c:93: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 7512
KLEE: done: completed paths = 161
KLEE: done: generated tests = 161

real    3m44.111s
user    3m43.319s
sys     0m0.951s
```

Now this is really slower (on my Intel Core i3-3110M 2.4GHz notebook) in comparison to Z3Py solution (7.1.1). But the answer is correct:

```
% ls klee-last | grep err
test000161.external.err
```

```
% ktest-tool --write-ints klee-last/test000161.ktest
ktest file : 'klee-last/test000161.ktest'
args       : ['klee_sudoku_or1.bc']
num objects: 1
object    0: name: b'cells'
object    0: size: 81
object    0: data: b'\x01\x04\x05\x03\x02\x07\x06\t\x08\x08\x03\t\x06\x05\x04\x01\x02\
    x07\x06\x07\x02\t\x01\x08\x05\x04\x03\x04\t\x06\x01\x08\x05\x03\x07\x02\x02\x01\x08\
    x04\x07\x03\t\x05\x06\x07\x05\x03\x02\t\x06\x04\x08\x01\x03\x06\x07\x05\x04\x02\x08\
    x01\t\t\x08\x04\x07\x06\x01\x02\x03\x05\x05\x02\x01\x08\x03\t\x07\x06\x04'
```

Character `\t` has code of 9 in C/C++, and KLEE prints byte array as a C/C++ string, so it shows some values in such way. We can just keep in mind that there is 9 at the each place where we see `\t`. The solution, while not properly formatted, correct indeed.

By the way, at lines 42 and 43 you may see how we tell to KLEE that all array elements must be within some limits. If we comment these lines out, we've got this:

```
% time klee klee_sudoku_or1.bc
KLEE: output directory is "/home/klee/klee-out-100"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: ERROR: /home/klee/klee_sudoku_or1.c:51: overshift error
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_sudoku_or1.c:51: overshift error
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_sudoku_or1.c:51: overshift error
KLEE: NOTE: now ignoring this error at this location
...
```

KLEE warns us that shift value at line 51 is too big. Indeed, KLEE may try all byte values up to 255 (0xFF), which are pointless to use there, and may be a symptom of error or bug, so KLEE warns about it.

Now let's use `klee_assume()` again:

```
#include <stdint.h>

/*
coordinates:
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
*/

uint8_t cells[9][9];

// http://www.norvig.com/sudoku.html
// http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
char *puzzle
    ="..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97..";
```

```
int main()
{
        klee_make_symbolic(cells, sizeof cells, "cells");

        // process text line:
        for (int row=0; row<9; row++)
                for (int column=0; column<9; column++)
                {
                        char c=puzzle[row*9 + column];
                        if (c!='.')
                                klee_assume (cells[row][column]==c-'0');
                        else
                        {
                                klee_assume (cells[row][column]>=1);
                                klee_assume (cells[row][column]<=9);
                        };
                };


        // for all 9 rows
        for (int row=0; row<9; row++)
        {

                klee_assume (((1<<cells[row][0]) |
                                (1<<cells[row][1]) |
                                (1<<cells[row][2]) |
                                (1<<cells[row][3]) |
                                (1<<cells[row][4]) |
                                (1<<cells[row][5]) |
                                (1<<cells[row][6]) |
                                (1<<cells[row][7]) |
                                (1<<cells[row][8]))==0x3FE ); // 11 1111 1110

        };

        // for all 9 columns
        for (int c=0; c<9; c++)
        {
                klee_assume (((1<<cells[0][c]) |
                                (1<<cells[1][c]) |
                                (1<<cells[2][c]) |
                                (1<<cells[3][c]) |
                                (1<<cells[4][c]) |
                                (1<<cells[5][c]) |
                                (1<<cells[6][c]) |
                                (1<<cells[7][c]) |
                                (1<<cells[8][c]))==0x3FE ); // 11 1111 1110
        };


        // enumerate all 9 squares
        for (int r=0; r<9; r+=3)
                for (int c=0; c<9; c+=3)
                {
```

```
                        // add constraints for each 3*3 square:
                        klee_assume ((1<<cells[r+0][c+0] |
                                        1<<cells[r+0][c+1] |
                                        1<<cells[r+0][c+2] |
                                        1<<cells[r+1][c+0] |
                                        1<<cells[r+1][c+1] |
                                        1<<cells[r+1][c+2] |
                                        1<<cells[r+2][c+0] |
                                        1<<cells[r+2][c+1] |
                                        1<<cells[r+2][c+2])==0x3FE ); // 11 1111 1110
                };

        // at this point, all constraints must be satisfied
        klee_assert(0);
};
```

```
% time klee klee_sudoku_or2.bc
KLEE: output directory is "/home/klee/klee-out-99"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_sudoku_or2.c:93: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 7119
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1

real    0m35.312s
user    0m34.945s
sys     0m0.318s
```

That works much faster: perhaps KLEE indeed handle this *intrinsic* in a special way. And, as we see, the only one path has been found (one we actually interesting in it) instead of 161.

It's still much slower than Z3Py solution, though.

### 7.1.5 Sudoku in SAT

One might think that we can encode each 1..9 number in binary form: 5 bits or variables would be enough. But there is even simpler way: allocate 9 bits, where only one bit will be *True*. The number 1 can be encoded as [1, 0, 0, 0, 0, 0, 0, 0, 0], the number 3 as [0, 0, 1, 0, 0, 0, 0, 0, 0], etc. Seems uneconomical? Yes, but other operations would be simpler.

First of all, we'll reuse important `POPCNT1` function I've described earlier: 7.7.1.

The second important operation we need to invent is making 9 numbers unique. If each number is encoded as 9-bits vector, 9 numbers can form a matrix, like:

```
0 0 0 0 0 0 1 0 0 <- 1st number
0 0 0 0 0 1 0 0 0 <- 2nd number
0 1 0 0 0 0 0 0 0 <- ...
0 0 1 0 0 0 0 0 0 <- ...
0 0 0 0 0 0 0 0 1 <- ...
0 0 0 0 1 0 0 0 0 <- ...
0 0 0 0 0 0 0 1 0 <- ...
1 0 0 0 0 0 0 0 0 <- ...
0 0 0 1 0 0 0 0 0 <- 9th number
```

Now we will use a `POPCNT1` function to make each row in the matrix to contain only one *True* bit, that will preserve consistency in encoding, since no vector can contain more than 1 *True* bit, or no *True* bits at all. Then we will use a `POPCNT1` function again to make all columns in the matrix to have only one single *True* bit. That will make all rows in matrix unique, in other words, all 9 encoded numbers will always be unique.

After applying POPCNT1 function 9+9=18 times we'll have 9 unique numbers in 1..9 range.

Using that operation we can make each row of Sudoku puzzle unique, each column unique and also each $3 \cdot 3 = 9$ box.

```
#-*- coding: utf-8 -*-

#!/usr/bin/env python

import itertools, subprocess, os

# global variables:
clauses=[]
vector_names={}
last_var=1

BITS_PER_VECTOR=9

def read_lines_from_file (fname):
    f=open(fname)
    new_ar=[item.rstrip() for item in f.readlines()]
    f.close()
    return new_ar

def run_minisat (CNF_fname):
    child = subprocess.Popen(["minisat", CNF_fname, "results.txt"], stdout=subprocess.
        PIPE)
    child.wait()
    # 10 is SAT, 20 is UNSAT
    if child.returncode==20:
        os.remove ("results.txt")
        return None

    if child.returncode!=10:
        print "(minisat) unknown retcode: ", child.returncode
        exit(0)

    solution=read_lines_from_file("results.txt")[1].split(" ")
    os.remove ("results.txt")

    return solution

def write_CNF(fname, clauses, VARS_TOTAL):
    f=open(fname, "w")
    f.write ("p cnf "+str(VARS_TOTAL)+" "+str(len(clauses))+"\n")
    [f.write(" ".join(c)+" 0\n") for c in clauses]
    f.close()

def neg(v):
    return "-"+v

def add_popcnt1(vars):
    global clauses
    # enumerate all possible pairs
    # no pair can have both True's
    # so add "~var OR ~var2"
    for pair in itertools.combinations(vars, r=2):
        clauses.append([neg(pair[0]), neg(pair[1])])
    # at least one var must be present:
```

```
    clauses.append(vars)

def make_distinct_bits_in_vector(vec_name):
    global vector_names
    global last_var

    add_popcnt1([vector_names[(vec_name,i)] for i in range(BITS_PER_VECTOR)])

def make_distinct_vectors(vectors):
    # take each bit from all vectors, call add_popcnt1()
    for i in range(BITS_PER_VECTOR):
        add_popcnt1([vector_names[(vec,i)] for vec in vectors])

def cvt_vector_to_number(vec_name, solution):
    for i in range(BITS_PER_VECTOR):
        if vector_names[(vec_name,i)] in solution:
            # variable present in solution as non-negated (without a "-" prefix)
            return i+1
    raise AssertionError

def alloc_var():
    global last_var
    last_var=last_var+1
    return str(last_var-1)

def alloc_vector(l, name):
    global last_var
    global vector_names
    rt=[]
    for i in range(l):
        v=alloc_var()
        vector_names[(name,i)]=v
        rt.append(v)
    return rt

def add_constant(var,b):
    global clauses
    if b==True or b==1:
        clauses.append([var])
    else:
        clauses.append([neg(var)])

# vec is a list of True/False/0/1
def add_constant_vector(vec_name, vec):
    global vector_names
    for i in range(BITS_PER_VECTOR):
        add_constant (vector_names[(vec_name, i)], vec[i])

# 1 -> [1, 0, 0, 0, 0, 0, 0, 0, 0]
# 3 -> [0, 0, 1, 0, 0, 0, 0, 0, 0]
def number_to_vector(n):
    rt=[0]*(n-1)
    rt.append(1)
    rt=rt+[0]*(BITS_PER_VECTOR-len(rt))
    return rt
```

```
"""
coordinates we're using here:

+--------+--------+--------+
|11 12 13|14 15 16|17 18 19|
|21 22 23|24 25 26|27 28 29|
|31 32 33|34 35 36|37 38 39|
+--------+--------+--------+
|41 42 43|44 45 46|47 48 49|
|51 52 53|54 55 56|57 58 59|
|61 62 63|64 65 66|67 68 69|
+--------+--------+--------+
|71 72 73|74 75 76|77 78 79|
|81 82 83|84 85 86|87 88 89|
|91 92 93|94 95 96|97 98 99|
+--------+--------+--------+
"""
def make_vec_name(row, col):
    return "cell"+str(row)+str(col)

def puzzle_to_clauses (puzzle):
    # process text line:
    current_column=1
    current_row=1
    for i in puzzle:
        if i!='.':
            add_constant_vector(make_vec_name(current_row, current_column),
                number_to_vector(int(i)))
        current_column=current_column+1
        if current_column==10:
            current_column=1
            current_row=current_row+1

def print_solution(solution):
    for row in range(1,9+1):
        # print row:
        print " ".join([str(cvt_vector_to_number(make_vec_name(row, col), solution)) for
             col in range(1,9+1)])

def main():
    # allocate 9*9*9=729 variables:
    for row in range(1, 9+1):
        for col in range(1, 9+1):
            alloc_vector(9, make_vec_name(row, col))
            make_distinct_bits_in_vector(make_vec_name(row, col))

    # variables in each row are unique:
    for row in range(1, 9+1):
        make_distinct_vectors([make_vec_name(row, col) for col in range(1, 9+1)])

    # variables in each column are unique:
    for col in range(1, 9+1):
        make_distinct_vectors([make_vec_name(row, col) for row in range(1, 9+1)])

    # variables in each 3*3 box are unique:
    for row in range(1, 9+1, 3):
```

```
        for col in range(1, 9+1, 3):
            tmp=[]
            tmp.append(make_vec_name(row+0, col+0))
            tmp.append(make_vec_name(row+0, col+1))
            tmp.append(make_vec_name(row+0, col+2))
            tmp.append(make_vec_name(row+1, col+0))
            tmp.append(make_vec_name(row+1, col+1))
            tmp.append(make_vec_name(row+1, col+2))
            tmp.append(make_vec_name(row+2, col+0))
            tmp.append(make_vec_name(row+2, col+1))
            tmp.append(make_vec_name(row+2, col+2))
            make_distinct_vectors(tmp)

    # http://www.norvig.com/sudoku.html
    # http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
    puzzle_to_clauses("
       ..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97..
       ")

    print "len(clauses)=",len(clauses)
    write_CNF("1.cnf", clauses, last_var-1)
    solution=run_minisat("1.cnf")
    #os.remove("1.cnf")
    if solution==None:
        print "unsat!"
        exit(0)

    print_solution(solution)

main()
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/sudoku/SAT/sudoku_SAT.py )

The `make_distinct_bits_in_vector()` function preserves consistency of encoding.
The `make_distinct_vectors()` function makes 9 numbers unique.
The `cvt_vector_to_number()` decodes vector to number.
The `number_to_vector()` encodes number to vector.
The `main()` function has all necessary calls to make rows/columns/$3 \cdot 3$ boxes unique.

That works:

```
% python sudoku_SAT.py
len(clauses)= 12195
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4
```

Same solution as earlier: 7.1.1.

Picosat tells this SAT instance has only one solution. Indeed, as they say, true Sudoku puzzle can have only one solution.

**Getting rid of one POPCNT1 function call** To make 9 unique 1..9 numbers we can use `POPCNT1` function to make each row in matrix be unique and use *OR* boolean operation for all columns. That will have merely the same effect: all

rows has to be unique to make each column to be evaluated to *True* if all variables in column are OR'ed. (I will do this in the next example: 7.5.)

That will make 3447 clauses instead of 12195, but somehow, SAT solvers works slower. No idea why.

## 7.2 Zebra puzzle (AKA Einstein puzzle)

## 7.3 SMT

Zebra puzzle is a popular puzzle, defined as follows:

> 1.There are five houses.
> 2.The Englishman lives in the red house.
> 3.The Spaniard owns the dog.
> 4.Coffee is drunk in the green house.
> 5.The Ukrainian drinks tea.
> 6.The green house is immediately to the right of the ivory house.
> 7.The Old Gold smoker owns snails.
> 8.Kools are smoked in the yellow house.
> 9.Milk is drunk in the middle house.
> 10.The Norwegian lives in the first house.
> 11.The man who smokes Chesterfields lives in the house next to the man with the fox.
> 12.Kools are smoked in the house next to the house where the horse is kept.
> 13.The Lucky Strike smoker drinks orange juice.
> 14.The Japanese smokes Parliaments.
> 15.The Norwegian lives next to the blue house.
>
> Now, who drinks water? Who owns the zebra?
>
> In the interest of clarity, it must be added that each of the five houses is painted a different color, and their inhabitants are of different national extractions, own different pets, drink different beverages and smoke different brands of American cigarets [sic]. One other thing: in statement 6, right means your right.

( https://en.wikipedia.org/wiki/Zebra_Puzzle )

It's a very good example of CSP[74].

We would encode each entity as integer variable, representing number of house.

Then, to define that Englishman lives in red house, we will add this constraint: `Englishman == Red`, meaning that number of a house where Englishmen resides and which is painted in red is the same.

To define that Norwegian lives next to the blue house, we don't realy know, if it is at left side of blue house or at right side, but we know that house numbers are different by just 1. So we will define this constraint: `Norwegian==Blue-1 OR Norwegian==Blue+1`.

We will also need to limit all house numbers, so they will be in range of 1..5.

We will also use `Distinct` to show that all various entities of the same type are all has different house numbers.

```
#!/usr/bin/env python
from z3 import *

Yellow, Blue, Red, Ivory, Green=Ints('Yellow Blue Red Ivory Green')
Norwegian, Ukrainian, Englishman, Spaniard, Japanese=Ints('Norwegian Ukrainian
    Englishman Spaniard Japanese')
Water, Tea, Milk, OrangeJuice, Coffee=Ints('Water Tea Milk OrangeJuice Coffee')
```

[74] Constraint satisfaction problem

```
Kools, Chesterfield, OldGold, LuckyStrike, Parliament=Ints('Kools Chesterfield OldGold
    LuckyStrike Parliament')
Fox, Horse, Snails, Dog, Zebra=Ints('Fox Horse Snails Dog Zebra')

s = Solver()

# colors are distinct for all 5 houses:
s.add(Distinct(Yellow, Blue, Red, Ivory, Green))

# all nationalities are living in different houses:
s.add(Distinct(Norwegian, Ukrainian, Englishman, Spaniard, Japanese))

# so are beverages:
s.add(Distinct(Water, Tea, Milk, OrangeJuice, Coffee))

# so are cigarettes:
s.add(Distinct(Kools, Chesterfield, OldGold, LuckyStrike, Parliament))

# so are pets:
s.add(Distinct(Fox, Horse, Snails, Dog, Zebra))

# limits.
# adding two constraints at once (separated by comma) is the same
# as adding one And() constraint with two subconstraints
s.add(Yellow>=1, Yellow<=5)
s.add(Blue>=1, Blue<=5)
s.add(Red>=1, Red<=5)
s.add(Ivory>=1, Ivory<=5)
s.add(Green>=1, Green<=5)

s.add(Norwegian>=1, Norwegian<=5)
s.add(Ukrainian>=1, Ukrainian<=5)
s.add(Englishman>=1, Englishman<=5)
s.add(Spaniard>=1, Spaniard<=5)
s.add(Japanese>=1, Japanese<=5)

s.add(Water>=1, Water<=5)
s.add(Tea>=1, Tea<=5)
s.add(Milk>=1, Milk<=5)
s.add(OrangeJuice>=1, OrangeJuice<=5)
s.add(Coffee>=1, Coffee<=5)

s.add(Kools>=1, Kools<=5)
s.add(Chesterfield>=1, Chesterfield<=5)
s.add(OldGold>=1, OldGold<=5)
s.add(LuckyStrike>=1, LuckyStrike<=5)
s.add(Parliament>=1, Parliament<=5)

s.add(Fox>=1, Fox<=5)
s.add(Horse>=1, Horse<=5)
s.add(Snails>=1, Snails<=5)
s.add(Dog>=1, Dog<=5)
s.add(Zebra>=1, Zebra<=5)

# main constraints of the puzzle:
```

```
# 2.The Englishman lives in the red house.
s.add(Englishman==Red)

# 3.The Spaniard owns the dog.
s.add(Spaniard==Dog)

# 4.Coffee is drunk in the green house.
s.add(Coffee==Green)

# 5.The Ukrainian drinks tea.
s.add(Ukrainian==Tea)

# 6.The green house is immediately to the right of the ivory house.
s.add(Green==Ivory+1)

# 7.The Old Gold smoker owns snails.
s.add(OldGold==Snails)

# 8.Kools are smoked in the yellow house.
s.add(Kools==Yellow)

# 9.Milk is drunk in the middle house.
s.add(Milk==3) # i.e., 3rd house

# 10.The Norwegian lives in the first house.
s.add(Norwegian==1)

# 11.The man who smokes Chesterfields lives in the house next to the man with the fox.
s.add(Or(Chesterfield==Fox+1, Chesterfield==Fox-1)) # left or right

# 12.Kools are smoked in the house next to the house where the horse is kept.
s.add(Or(Kools==Horse+1, Kools==Horse-1)) # left or right

# 13.The Lucky Strike smoker drinks orange juice.
s.add(LuckyStrike==OrangeJuice)

# 14.The Japanese smokes Parliaments.
s.add(Japanese==Parliament)

# 15.The Norwegian lives next to the blue house.
s.add(Or(Norwegian==Blue+1, Norwegian==Blue-1)) # left or right

r=s.check()
print r
if r==unsat:
    exit(0)
m=s.model()
print(m)
```

When we run it, we got correct result:

```
sat
[Snails = 3,
 Blue = 2,
 Ivory = 4,
 OrangeJuice = 4,
 Parliament = 5,
```

```
Yellow = 1,
Fox = 1,
Zebra = 5,
Horse = 2,
Dog = 4,
Tea = 2,
Water = 1,
Chesterfield = 2,
Red = 3,
Japanese = 5,
LuckyStrike = 4,
Norwegian = 1,
Milk = 3,
Kools = 1,
OldGold = 3,
Ukrainian = 2,
Coffee = 5,
Green = 5,
Spaniard = 4,
Englishman = 3]
```

### 7.4 KLEE

We just define all variables and add constraints:

```
int main()
{
        int Yellow, Blue, Red, Ivory, Green;
        int Norwegian, Ukrainian, Englishman, Spaniard, Japanese;
        int Water, Tea, Milk, OrangeJuice, Coffee;
        int Kools, Chesterfield, OldGold, LuckyStrike, Parliament;
        int Fox, Horse, Snails, Dog, Zebra;

        klee_make_symbolic(&Yellow, sizeof(int), "Yellow");
        klee_make_symbolic(&Blue, sizeof(int), "Blue");
        klee_make_symbolic(&Red, sizeof(int), "Red");
        klee_make_symbolic(&Ivory, sizeof(int), "Ivory");
        klee_make_symbolic(&Green, sizeof(int), "Green");

        klee_make_symbolic(&Norwegian, sizeof(int), "Norwegian");
        klee_make_symbolic(&Ukrainian, sizeof(int), "Ukrainian");
        klee_make_symbolic(&Englishman, sizeof(int), "Englishman");
        klee_make_symbolic(&Spaniard, sizeof(int), "Spaniard");
        klee_make_symbolic(&Japanese, sizeof(int), "Japanese");

        klee_make_symbolic(&Water, sizeof(int), "Water");
        klee_make_symbolic(&Tea, sizeof(int), "Tea");
        klee_make_symbolic(&Milk, sizeof(int), "Milk");
        klee_make_symbolic(&OrangeJuice, sizeof(int), "OrangeJuice");
        klee_make_symbolic(&Coffee, sizeof(int), "Coffee");

        klee_make_symbolic(&Kools, sizeof(int), "Kools");
        klee_make_symbolic(&Chesterfield, sizeof(int), "Chesterfield");
        klee_make_symbolic(&OldGold, sizeof(int), "OldGold");
        klee_make_symbolic(&LuckyStrike, sizeof(int), "LuckyStrike");
        klee_make_symbolic(&Parliament, sizeof(int), "Parliament");
```

```
klee_make_symbolic(&Fox, sizeof(int), "Fox");
klee_make_symbolic(&Horse, sizeof(int), "Horse");
klee_make_symbolic(&Snails, sizeof(int), "Snails");
klee_make_symbolic(&Dog, sizeof(int), "Dog");
klee_make_symbolic(&Zebra, sizeof(int), "Zebra");

// limits.
if (Yellow<1 || Yellow>5) return 0;
if (Blue<1 || Blue>5) return 0;
if (Red<1 || Red>5) return 0;
if (Ivory<1 || Ivory>5) return 0;
if (Green<1 || Green>5) return 0;

if (Norwegian<1 || Norwegian>5) return 0;
if (Ukrainian<1 || Ukrainian>5) return 0;
if (Englishman<1 || Englishman>5) return 0;
if (Spaniard<1 || Spaniard>5) return 0;
if (Japanese<1 || Japanese>5) return 0;

if (Water<1 || Water>5) return 0;
if (Tea<1 || Tea>5) return 0;
if (Milk<1 || Milk>5) return 0;
if (OrangeJuice<1 || OrangeJuice>5) return 0;
if (Coffee<1 || Coffee>5) return 0;

if (Kools<1 || Kools>5) return 0;
if (Chesterfield<1 || Chesterfield>5) return 0;
if (OldGold<1 || OldGold>5) return 0;
if (LuckyStrike<1 || LuckyStrike>5) return 0;
if (Parliament<1 || Parliament>5) return 0;

if (Fox<1 || Fox>5) return 0;
if (Horse<1 || Horse>5) return 0;
if (Snails<1 || Snails>5) return 0;
if (Dog<1 || Dog>5) return 0;
if (Zebra<1 || Zebra>5) return 0;

// colors are distinct for all 5 houses:
if (((1<<Yellow) | (1<<Blue) | (1<<Red) | (1<<Ivory) | (1<<Green))!=0x3E) return
    0; // 111110

// all nationalities are living in different houses:
if (((1<<Norwegian) | (1<<Ukrainian) | (1<<Englishman) | (1<<Spaniard) | (1<<
   Japanese))!=0x3E) return 0; // 111110

// so are beverages:
if (((1<<Water) | (1<<Tea) | (1<<Milk) | (1<<OrangeJuice) | (1<<Coffee))!=0x3E)
   return 0; // 111110

// so are cigarettes:
if (((1<<Kools) | (1<<Chesterfield) | (1<<OldGold) | (1<<LuckyStrike) | (1<<
   Parliament))!=0x3E) return 0; // 111110

// so are pets:
if (((1<<Fox) | (1<<Horse) | (1<<Snails) | (1<<Dog) | (1<<Zebra))!=0x3E) return
```

```
            0; // 111110

        // main constraints of the puzzle:

        // 2.The Englishman lives in the red house.
        if (Englishman!=Red) return 0;

        // 3.The Spaniard owns the dog.
        if (Spaniard!=Dog) return 0;

        // 4.Coffee is drunk in the green house.
        if (Coffee!=Green) return 0;

        // 5.The Ukrainian drinks tea.
        if (Ukrainian!=Tea) return 0;

        // 6.The green house is immediately to the right of the ivory house.
        if (Green!=Ivory+1) return 0;

        // 7.The Old Gold smoker owns snails.
        if (OldGold!=Snails) return 0;

        // 8.Kools are smoked in the yellow house.
        if (Kools!=Yellow) return 0;

        // 9.Milk is drunk in the middle house.
        if (Milk!=3) return 0; // i.e., 3rd house

        // 10.The Norwegian lives in the first house.
        if (Norwegian!=1) return 0;

        // 11.The man who smokes Chesterfields lives in the house next to the man with
            the fox.
        if (Chesterfield!=Fox+1 && Chesterfield!=Fox-1) return 0; // left or right

        // 12.Kools are smoked in the house next to the house where the horse is kept.
        if (Kools!=Horse+1 && Kools!=Horse-1) return 0; // left or right

        // 13.The Lucky Strike smoker drinks orange juice.
        if (LuckyStrike!=OrangeJuice) return 0;

        // 14.The Japanese smokes Parliaments.
        if (Japanese!=Parliament) return 0;

        // 15.The Norwegian lives next to the blue house.
        if (Norwegian!=Blue+1 && Norwegian!=Blue-1) return 0; // left or right

        // all constraints are satisfied at this point
        // force KLEE to produce .err file:
        klee_assert(0);

        return 0;
};
```

I force KLEE to find distinct values for colors, nationalities, cigarettes, etc, in the same way as I did for Sudoku earlier (7.1.1).

Let's run it:

```
% clang -emit-llvm -c -g klee_zebra1.c
...

% klee klee_zebra1.bc
KLEE: output directory is "/home/klee/klee-out-97"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_zebra1.c:130: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 761
KLEE: done: completed paths = 55
KLEE: done: generated tests = 55
```

It works for ≈ 7 seconds on my Intel Core i3-3110M 2.4GHz notebook. Let's find out path, where `klee_assert()` has been executed:

```
% ls klee-last | grep err
test000051.external.err

% ktest-tool --write-ints klee-last/test000051.ktest | less

ktest file : 'klee-last/test000051.ktest'
args       : ['klee_zebra1.bc']
num objects: 25
object    0: name: b'Yellow'
object    0: size: 4
object    0: data: 1
object    1: name: b'Blue'
object    1: size: 4
object    1: data: 2
object    2: name: b'Red'
object    2: size: 4
object    2: data: 3
object    3: name: b'Ivory'
object    3: size: 4
object    3: data: 4

...

object   21: name: b'Horse'
object   21: size: 4
object   21: data: 2
object   22: name: b'Snails'
object   22: size: 4
object   22: data: 3
object   23: name: b'Dog'
object   23: size: 4
object   23: data: 4
object   24: name: b'Zebra'
object   24: size: 4
object   24: data: 5
```

This is indeed correct solution.

`klee_assume()` also can be used this time:

```
int main()
```

```
{
        int Yellow, Blue, Red, Ivory, Green;
        int Norwegian, Ukrainian, Englishman, Spaniard, Japanese;
        int Water, Tea, Milk, OrangeJuice, Coffee;
        int Kools, Chesterfield, OldGold, LuckyStrike, Parliament;
        int Fox, Horse, Snails, Dog, Zebra;

        klee_make_symbolic(&Yellow, sizeof(int), "Yellow");
        klee_make_symbolic(&Blue, sizeof(int), "Blue");
        klee_make_symbolic(&Red, sizeof(int), "Red");
        klee_make_symbolic(&Ivory, sizeof(int), "Ivory");
        klee_make_symbolic(&Green, sizeof(int), "Green");

        klee_make_symbolic(&Norwegian, sizeof(int), "Norwegian");
        klee_make_symbolic(&Ukrainian, sizeof(int), "Ukrainian");
        klee_make_symbolic(&Englishman, sizeof(int), "Englishman");
        klee_make_symbolic(&Spaniard, sizeof(int), "Spaniard");
        klee_make_symbolic(&Japanese, sizeof(int), "Japanese");

        klee_make_symbolic(&Water, sizeof(int), "Water");
        klee_make_symbolic(&Tea, sizeof(int), "Tea");
        klee_make_symbolic(&Milk, sizeof(int), "Milk");
        klee_make_symbolic(&OrangeJuice, sizeof(int), "OrangeJuice");
        klee_make_symbolic(&Coffee, sizeof(int), "Coffee");

        klee_make_symbolic(&Kools, sizeof(int), "Kools");
        klee_make_symbolic(&Chesterfield, sizeof(int), "Chesterfield");
        klee_make_symbolic(&OldGold, sizeof(int), "OldGold");
        klee_make_symbolic(&LuckyStrike, sizeof(int), "LuckyStrike");
        klee_make_symbolic(&Parliament, sizeof(int), "Parliament");

        klee_make_symbolic(&Fox, sizeof(int), "Fox");
        klee_make_symbolic(&Horse, sizeof(int), "Horse");
        klee_make_symbolic(&Snails, sizeof(int), "Snails");
        klee_make_symbolic(&Dog, sizeof(int), "Dog");
        klee_make_symbolic(&Zebra, sizeof(int), "Zebra");

        // limits.
        klee_assume (Yellow>=1 && Yellow<=5);
        klee_assume (Blue>=1 && Blue<=5);
        klee_assume (Red>=1 && Red<=5);
        klee_assume (Ivory>=1 && Ivory<=5);
        klee_assume (Green>=1 && Green<=5);

        klee_assume (Norwegian>=1 && Norwegian<=5);
        klee_assume (Ukrainian>=1 && Ukrainian<=5);
        klee_assume (Englishman>=1 && Englishman<=5);
        klee_assume (Spaniard>=1 && Spaniard<=5);
        klee_assume (Japanese>=1 && Japanese<=5);

        klee_assume (Water>=1 && Water<=5);
        klee_assume (Tea>=1 && Tea<=5);
        klee_assume (Milk>=1 && Milk<=5);
        klee_assume (OrangeJuice>=1 && OrangeJuice<=5);
        klee_assume (Coffee>=1 && Coffee<=5);
```

```
klee_assume (Kools>=1 && Kools<=5);
klee_assume (Chesterfield>=1 && Chesterfield<=5);
klee_assume (OldGold>=1 && OldGold<=5);
klee_assume (LuckyStrike>=1 && LuckyStrike<=5);
klee_assume (Parliament>=1 && Parliament<=5);

klee_assume (Fox>=1 && Fox<=5);
klee_assume (Horse>=1 && Horse<=5);
klee_assume (Snails>=1 && Snails<=5);
klee_assume (Dog>=1 && Dog<=5);
klee_assume (Zebra>=1 && Zebra<=5);

// colors are distinct for all 5 houses:
klee_assume (((1<<Yellow) | (1<<Blue) | (1<<Red) | (1<<Ivory) | (1<<Green))==0
   x3E); // 111110

// all nationalities are living in different houses:
klee_assume (((1<<Norwegian) | (1<<Ukrainian) | (1<<Englishman) | (1<<Spaniard)
   | (1<<Japanese))==0x3E); // 111110

// so are beverages:
klee_assume (((1<<Water) | (1<<Tea) | (1<<Milk) | (1<<OrangeJuice) | (1<<Coffee)
   )==0x3E); // 111110

// so are cigarettes:
klee_assume (((1<<Kools) | (1<<Chesterfield) | (1<<OldGold) | (1<<LuckyStrike) |
    (1<<Parliament))==0x3E); // 111110

// so are pets:
klee_assume (((1<<Fox) | (1<<Horse) | (1<<Snails) | (1<<Dog) | (1<<Zebra))==0x3E
   ); // 111110

// main constraints of the puzzle:

// 2.The Englishman lives in the red house.
klee_assume (Englishman==Red);

// 3.The Spaniard owns the dog.
klee_assume (Spaniard==Dog);

// 4.Coffee is drunk in the green house.
klee_assume (Coffee==Green);

// 5.The Ukrainian drinks tea.
klee_assume (Ukrainian==Tea);

// 6.The green house is immediately to the right of the ivory house.
klee_assume (Green==Ivory+1);

// 7.The Old Gold smoker owns snails.
klee_assume (OldGold==Snails);

// 8.Kools are smoked in the yellow house.
klee_assume (Kools==Yellow);

// 9.Milk is drunk in the middle house.
```

```
        klee_assume (Milk==3); // i.e., 3rd house

        // 10.The Norwegian lives in the first house.
        klee_assume (Norwegian==1);

        // 11.The man who smokes Chesterfields lives in the house next to the man with
           the fox.
        klee_assume (Chesterfield==Fox+1 || Chesterfield==Fox-1); // left or right

        // 12.Kools are smoked in the house next to the house where the horse is kept.
        klee_assume (Kools==Horse+1 || Kools==Horse-1); // left or right

        // 13.The Lucky Strike smoker drinks orange juice.
        klee_assume (LuckyStrike==OrangeJuice);

        // 14.The Japanese smokes Parliaments.
        klee_assume (Japanese==Parliament);

        // 15.The Norwegian lives next to the blue house.
        klee_assume (Norwegian==Blue+1 || Norwegian==Blue-1); // left or right

        // all constraints are satisfied at this point
        // force KLEE to produce .err file:
        klee_assert(0);
};
```

…and this version works slightly faster (≈ 5 seconds), maybe because KLEE is aware of this *intrinsic* and handles it in a special way?

## 7.5 Zebra puzzle as a SAT problem

I would define each variable as vector of 5 variables, as I did before in Sudoku solver: 7.1.5.

I also use POPCNT1 function, but unlike previous example, I used Wolfram Mathematica to generate it in CNF form:

```
In[]:= tbl1=Table[PadLeft[IntegerDigits[i,2],5] ->If[Equal[DigitCount[i
   ,2][[1]],1],1,0],{i,0,63}]
Out[]= {{0,0,0,0,0}->0,
{0,0,0,0,1}->1,
{0,0,0,1,0}->1,
{0,0,0,1,1}->0,
{0,0,1,0,0}->1,
{0,0,1,0,1}->0,

...

{1,1,1,1,0}->0,
{1,1,1,1,1}->0}

In[]:= BooleanConvert[BooleanFunction[tbl1,{a,b,c,d,e}],"CNF"]
Out[]= (!a||!b)&&(!a||!c)&&(!a||!d)&&(!a||!e)&&(a||b||c||d||e)&&(!b||!c)&&(!b||!d)&&(!b
   ||!e)&&(!c||!d)&&(!c||!e)&&(!d||!e)
```

Also, as I suggested before (7.1.5), I used *OR* operation as the second step.

```
def mathematica_to_CNF (s, d):
    for k in d.keys():
        s=s.replace(k, d[k])
    s=s.replace("!", "-").replace("||", " ").replace("(", "").replace(")", "")
```

```
    s=s.split ("&&")
    return s

def add_popcnt1(v1, v2, v3, v4, v5):
    global clauses
    s="(!a||!b)&&" \
      "(!a||!c)&&" \
      "(!a||!d)&&" \
      "(!a||!e)&&" \
      "(!b||!c)&&" \
      "(!b||!d)&&" \
      "(!b||!e)&&" \
      "(!c||!d)&&" \
      "(!c||!e)&&" \
      "(!d||!e)&&" \
      "(a||b||c||d||e)"

    clauses=clauses+mathematica_to_CNF(s, {"a":v1, "b":v2, "c":v3, "d":v4, "e":v5})

...

# k=tuple: ("high-level" variable name, number of bit (0..4))
# v=variable number in CNF
vars={}
vars_last=1

...

def alloc_distinct_variables(names):
    global vars
    global vars_last
    for name in names:
        for i in range(5):
            vars[(name,i)]=str(vars_last)
            vars_last=vars_last+1

        add_popcnt1(vars[(name,0)], vars[(name,1)], vars[(name,2)], vars[(name,3)], vars
           [(name,4)])

    # make them distinct:
    for i in range(5):
        clauses.append(vars[(names[0],i)] + " " + vars[(names[1],i)] + " " + vars[(names
           [2],i)] + " " + vars[(names[3],i)] + " " + vars[(names[4],i)])

...

alloc_distinct_variables(["Yellow", "Blue", "Red", "Ivory", "Green"])
alloc_distinct_variables(["Norwegian", "Ukrainian", "Englishman", "Spaniard", "Japanese
   "])
alloc_distinct_variables(["Water", "Tea", "Milk", "OrangeJuice", "Coffee"])
alloc_distinct_variables(["Kools", "Chesterfield", "OldGold", "LuckyStrike", "Parliament
   "])
alloc_distinct_variables(["Fox", "Horse", "Snails", "Dog", "Zebra"])

...
```

Now we have 5 boolean variables for each *high-level* variable, and each group of variables will always have distinct

values.

Now let's reread puzzle description: "2.The Englishman lives in the red house.". That's easy. In my Z3 and KLEE examples I just wrote "Englishman==Red". Same story here: we just add a clauses showing that 5 boolean variables for "Englishman" must be equal to 5 booleans for "Red".

On a lowest CNF level, if we want to say that two variables must be equal to each other, we add two clauses:

$(var1 \vee \neg var2) \wedge (\neg var1 \vee var2)$

That means, both *var1* and *var2* values must be *False* or *True*, but they cannot be different.

```
def add_eq_clauses(var1, var2):
    global clauses
    clauses.append(var1 + " -" + var2)
    clauses.append("-"+var1 + " " + var2)

def add_eq (n1, n2):
    for i in range(5):
        add_eq_clauses(vars[(n1,i)], vars[(n2, i)])

...

# 2.The Englishman lives in the red house.
add_eq("Englishman","Red")

# 3.The Spaniard owns the dog.
add_eq("Spaniard","Dog")

# 4.Coffee is drunk in the green house.
add_eq("Coffee","Green")

...
```

Now the next conditions: "9.Milk is drunk in the middle house." (i.e., 3rd house), "10.The Norwegian lives in the first house." We can just assign boolean values directly:

```
# n=1..5
def add_eq_var_n (name, n):
    global clauses
    global vars
    for i in range(5):
        if i==n-1:
            clauses.append(vars[(name,i)]) # always True
        else:
            clauses.append("-"+vars[(name,i)]) # always False

...

# 9.Milk is drunk in the middle house.
add_eq_var_n("Milk",3) # i.e., 3rd house

# 10.The Norwegian lives in the first house.
add_eq_var_n("Norwegian",1)
```

For "Milk" we will have "0 0 1 0 0" value, for "Norwegian": "1 0 0 0 0".

What to do with this? "6.The green house is immediately to the right of the ivory house." I can construct the following condition:

```
    Ivory      Green
AND(1 0 0 0 0  0 1 0 0 0)
.. OR ..
```

```
AND(0 1 0 0 0  0 0 1 0 0)
.. OR ..
AND(0 0 1 0 0  0 0 0 1 0)
.. OR ..
AND(0 0 0 1 0  0 0 0 0 1)
```

There is no “0 0 0 0 1” for “Ivory”, because it cannot be the last one. Now I can convert these conditions to CNF using Wolfram Mathematica:

```
In[]:= BooleanConvert[(a1&& !b1&&!c1&&!d1&&!e1&&!a2&& b2&&!c2&&!d2&&!e2) ||
(!a1&& b1&&!c1&&!d1&&!e1&&!a2&& !b2&&c2&&!d2&&!e2) ||
(!a1&& !b1&&c1&&!d1&&!e1&&!a2&& !b2&&!c2&&d2&&!e2) ||
(!a1&& !b1&&!c1&&d1&&!e1&&!a2&& !b2&&!c2&&!d2&&e2) ,"CNF"]

Out[]= (!a1||!b1)&&(!a1||!c1)&&(!a1||!d1)&&(a1||b1||c1||d1)&&!a2&&(!b1||!b2)&&(!b1||!c1)
    &&
(!b1||!d1)&&(b1||b2||c1||d1)&&(!b2||!c1)&&(!b2||!c2)&&(!b2||!d1)&&(!b2||!d2)&&(!b2||!e2)
    &&
(b2||c1||c2||d1)&&(b2||c2||d1||d2)&&(b2||c2||d2||e2)&&(!c1||!c2)&&(!c1||!d1)&&(!c2||!d1)
    &&
(!c2||!d2)&&(!c2||!e2)&&(!d1||!d2)&&(!d2||!e2)&&!e1
```

And here is a piece of my Python code:

```
def add_right (n1, n2):
    global clauses
    s="(!a1||!b1)&&(!a1||!c1)&&(!a1||!d1)&&(a1||b1||c1||d1)&&!a2&&(!b1||!b2)&&(!b1||!c1)
        &&(!b1||!d1)&&" \
      "(b1||b2||c1||d1)&&(!b2||!c1)&&(!b2||!c2)&&(!b2||!d1)&&(!b2||!d2)&&(!b2||!e2)&&(b2
         ||c1||c2||d1)&&" \
      "(b2||c2||d1||d2)&&(b2||c2||d2||e2)&&(!c1||!c2)&&(!c1||!d1)&&(!c2||!d1)&&(!c2||!d2
         )&&(!c2||!e2)&&" \
      "(!d1||!d2)&&(!d2||!e2)&&!e1"

    clauses=clauses+mathematica_to_CNF(s, {
        "a1": vars[(n1,0)], "b1": vars[(n1,1)], "c1": vars[(n1,2)], "d1": vars[(n1,3)],
           "e1": vars[(n1,4)],
        "a2": vars[(n2,0)], "b2": vars[(n2,1)], "c2": vars[(n2,2)], "d2": vars[(n2,3)],
           "e2": vars[(n2,4)]})

...

# 6.The green house is immediately to the right of the ivory house.
add_right("Ivory", "Green")
```

What we will do with that? “11.The man who smokes Chesterfields lives in the house next to the man with the fox.” “12.Kools are smoked in the house next to the house where the horse is kept.”

We don’t know side, left or right, but we know that they are differ in one. Here is a clauses I would add:

```
    Chesterfield  Fox
AND(0 0 0 0 1     0 0 0 1 0)
.. OR ..
AND(0 0 0 1 0     0 0 0 0 1)
AND(0 0 0 1 0     0 0 1 0 0)
.. OR ..
AND(0 0 1 0 0     0 1 0 0 0)
AND(0 0 1 0 0     0 0 0 1 0)
.. OR ..
AND(0 1 0 0 0     1 0 0 0 0)
```

```
AND(0 1 0 0 0     0 0 1 0 0)
.. OR ..
AND(1 0 0 0 0     0 1 0 0 0)
```

I can convert this into CNF using Mathematica again:

```
In[]:= BooleanConvert[(a1&& !b1&&!c1&&!d1&&!e1&&!a2&& b2&&!c2&&!d2&&!e2) ||

(!a1&& b1&&!c1&&!d1&&!e1&&a2&& !b2&&!c2&&!d2&&!e2) ||
(!a1&& b1&&!c1&&!d1&&!e1&&!a2&& !b2&&c2&&!d2&&!e2) ||

(!a1&& !b1&&c1&&!d1&&!e1&&!a2&& b2&&!c2&&!d2&&!e2) ||
(!a1&& !b1&&c1&&!d1&&!e1&&!a2&& !b2&&!c2&&d2&&!e2) ||

(!a1&& !b1&&!c1&&d1&&!e1&&!a2&& !b2&&c2&&!d2&&!e2) ||
(!a1&& !b1&&!c1&&d1&&!e1&&!a2&& !b2&&!c2&&!d2&&e2) ||

(!a1&& !b1&&!c1&&!d1&&e1&&!a2&& !b2&&!c2&&d2&&!e2) ,"CNF"]

Out[]= (!a1||!b1)&&(!a1||!c1)&&(!a1||!d1)&&(!a1||!e1)&&(a1||b1||c1||d1||e1)&&(!a2||b1)
    &&(!a2||!b2)&&
(!a2||!c2)&&(!a2||!d2)&&(!a2||!e2)&&(a2||b2||c1||c2||d1||e1)&&(a2||b2||c2||d1||d2)&&(a2
    ||b2||c2||d2||e2)&&
(!b1||!b2)&&(!b1||!c1)&&(!b1||!d1)&&(!b1||!e1)&&(b1||b2||c1||d1||e1)&&(!b2||!c2)&&(!b2
    ||!d1)&&(!b2||!d2)&&
(!b2||!e1)&&(!b2||!e2)&&(!c1||!c2)&&(!c1||!d1)&&(!c1||!e1)&&(!c2||!d2)&&(!c2||!e1)&&(!c2
    ||!e2)&&
(!d1||!d2)&&(!d1||!e1)&&(!d2||!e2)
```

And here is my code:

```
def add_right_or_left (n1, n2):
    global clauses
    s="(!a1||!b1)&&(!a1||!c1)&&(!a1||!d1)&&(!a1||!e1)&&(a1||b1||c1||d1||e1)&&(!a2||b1)
        &&" \
      "(!a2||!b2)&&(!a2||!c2)&&(!a2||!d2)&&(!a2||!e2)&&(a2||b2||c1||c2||d1||e1)&&(a2||b2
          ||c2||d1||d2)&&" \
       "(a2||b2||c2||d2||e2)&&(!b1||!b2)&&(!b1||!c1)&&(!b1||!d1)&&(!b1||!e1)&&(b1||b2||
          c1||d1||e1)&&" \
       "(!b2||!c2)&&(!b2||!d1)&&(!b2||!d2)&&(!b2||!e1)&&(!b2||!e2)&&(!c1||!c2)&&(!c1||!
          d1)&&(!c1||!e1)&&" \
       "(!c2||!d2)&&(!c2||!e1)&&(!c2||!e2)&&(!d1||!d2)&&(!d1||!e1)&&(!d2||!e2)"

    clauses=clauses+mathematica_to_CNF(s, {
        "a1": vars[(n1,0)], "b1": vars[(n1,1)], "c1": vars[(n1,2)], "d1": vars[(n1,3)],
            "e1": vars[(n1,4)],
        "a2": vars[(n2,0)], "b2": vars[(n2,1)], "c2": vars[(n2,2)], "d2": vars[(n2,3)],
            "e2": vars[(n2,4)]})

...

# 11.The man who smokes Chesterfields lives in the house next to the man with the fox.
add_right_or_left("Chesterfield","Fox") # left or right

# 12.Kools are smoked in the house next to the house where the horse is kept.
add_right_or_left("Kools","Horse") # left or right
```

This is it! The full source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/zebra/SAT/zebra_SAT.py.

Resulting CNF instance has 125 boolean variables and 511 clauses: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/zebra/SAT/1.cnf. It is a piece of cake for any SAT solver. Even my toy-level SAT-solver (22.1) can solve it in ~1 second on my ancient Intel Atom netbook.

And of course, there is only one possible solution, what is acknowledged by Picosat.

```
% python zebra_SAT.py
Yellow 1
Blue 2
Red 3
Ivory 4
Green 5
Norwegian 1
Ukrainian 2
Englishman 3
Spaniard 4
Japanese 5
Water 1
Tea 2
Milk 3
OrangeJuice 4
Coffee 5
Kools 1
Chesterfield 2
OldGold 3
LuckyStrike 4
Parliament 5
Fox 1
Horse 2
Snails 3
Dog 4
Zebra 5
```

## 7.6 Solving pipe puzzle using Z3 SMT-solver

"Pipe puzzle" is a popular puzzle (just google "pipe puzzle" and look at images).

This is how shuffled puzzle looks like:

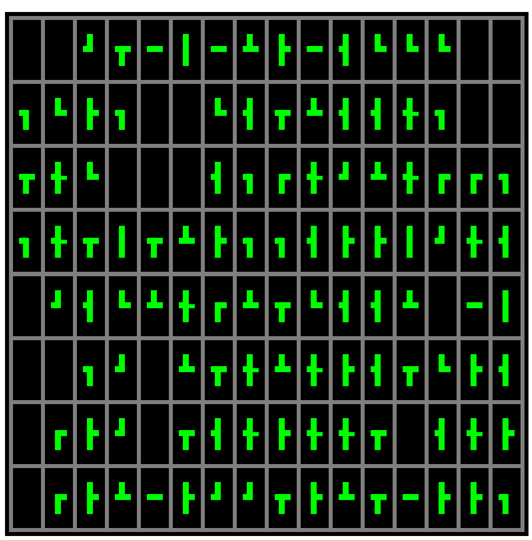


Figure 16: Shuffled puzzle

...and solved:

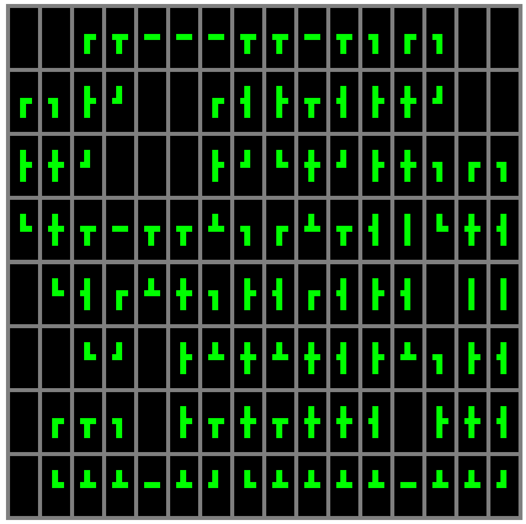
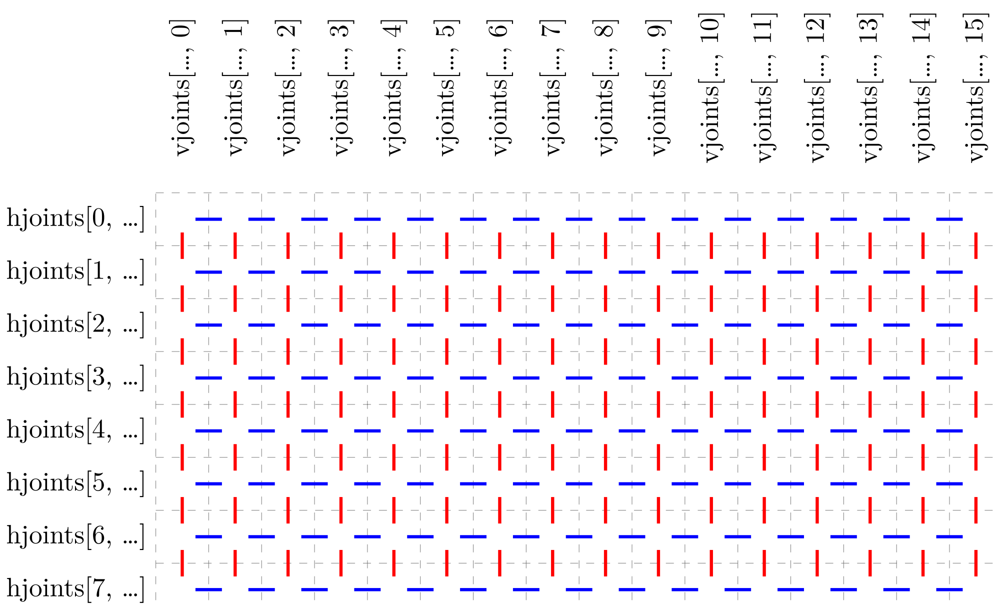


Figure 17: Solved puzzle

Let's try to find a way to solve it.

### 7.6.1 Generation

First, we need to generate it. Here is my quick idea on it. Take 8*16 array of cells. Each cell may contain some type of block. There are joints between cells:




Blue lines are horizontal joints, red lines are vertical joints. We just set each joint to 0/false (absent) or 1/true (present), randomly.

Once set, it's now easy to find type for each cell. There are:

| joints | our internal name | angle | symbol |
|---|---|---|---|
| 0 | type 0 | 0° | (space) |
| 2 | type 2a | 0° | ┃ |
| 2 | type 2a | 90° | ╸ |

| | | | |
|---|---|---|---|
| 2 | type 2b | 0° | ┏ |
| 2 | type 2b | 90° | ┓ |
| 2 | type 2b | 180° | ┛ |
| 2 | type 2b | 270° | ┗ |
| 3 | type 3 | 0° | ┣ |
| 3 | type 3 | 90° | ┳ |
| 3 | type 3 | 180° | ┫ |
| 3 | type 3 | 270° | ┻ |
| 4 | type 4 | 0° | ╋ |

*Dangling* joints can be preset at a first stage (i.e., cell with only one joint), but they are removed recursively, these cells are transforming into empty cells. Hence, at the end, all cells has at least two joints, and the whole plumbing system has no connections with outer world—I hope this would make things clearer.

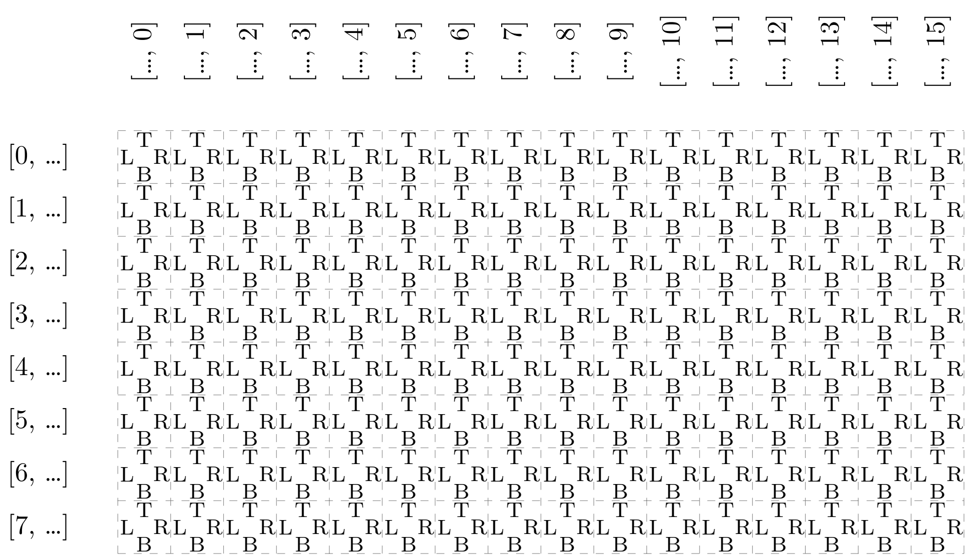
The C source code of generator is here: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/puzzles/pipe/generator. All horizontal joints are stored in the global array *hjoints[]* and vertical in *vjoints[]*.

The C program generates ANSI-colored output like it has been showed above (7.6, 7.6) plus an array of types, with no angle information about each cell:

```
[
["0", "0", "2b", "3", "2a", "2a", "2a", "3", "3", "2a", "3", "2b", "2b", "2b", "0",
   "0"],
["2b", "2b", "3", "2b", "0", "0", "2b", "3", "3", "3", "3", "3", "4", "2b", "0", "0"],
["3", "4", "2b", "0", "0", "0", "3", "2b", "2b", "4", "2b", "3", "4", "2b", "2b", "2b"],
["2b", "4", "3", "2a", "3", "3", "3", "2b", "2b", "3", "3", "3", "2a", "2b", "4", "3"],
["0", "2b", "3", "2b", "3", "4", "2b", "3", "3", "2b", "3", "3", "3", "0", "2a", "2a"],
["0", "0", "2b", "2b", "0", "3", "3", "4", "3", "4", "3", "3", "3", "2b", "3", "3"],
["0", "2b", "3", "2b", "0", "3", "3", "4", "3", "4", "4", "3", "0", "3", "4", "3"],
["0", "2b", "3", "3", "2a", "3", "2b", "2b", "3", "3", "3", "3", "2a", "3", "3", "2b"],
]
```

### 7.6.2 Solving

First of all, we would think about 8*16 array of cells, where each has four bits: "T" (top), "B" (bottom), "L" (left), "R" (right). Each bit represents half of joint.




Now we define arrays of each of four half-joints + angle information:

```
HEIGHT=8
WIDTH=16

# if T/B/R/L is Bool instead of Int, Z3 solver will work faster
T=[[Bool('cell_%d_%d_top' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
B=[[Bool('cell_%d_%d_bottom' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
R=[[Bool('cell_%d_%d_right' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
L=[[Bool('cell_%d_%d_left' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
A=[[Int('cell_%d_%d_angle' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
```

We know that if each of half-joints is present, corresponding half-joint must be also present, and vice versa. We define this using these constraints:

```
# shorthand variables for True and False:
t=True
f=False

# "top" of each cell must be equal to "bottom" of the cell above
# "bottom" of each cell must be equal to "top" of the cell below
# "left" of each cell must be equal to "right" of the cell at left
# "right" of each cell must be equal to "left" of the cell at right
for r in range(HEIGHT):
    for c in range(WIDTH):
        if r!=0:
            s.add(T[r][c]==B[r-1][c])
        if r!=HEIGHT-1:
            s.add(B[r][c]==T[r+1][c])
        if c!=0:
            s.add(L[r][c]==R[r][c-1])
        if c!=WIDTH-1:
            s.add(R[r][c]==L[r][c+1])

# "left" of each cell of first column shouldn't have any connection
# so is "right" of each cell of the last column
for r in range(HEIGHT):
    s.add(L[r][0]==f)
    s.add(R[r][WIDTH-1]==f)

# "top" of each cell of the first row shouldn't have any connection
# so is "bottom" of each cell of the last row
for c in range(WIDTH):
    s.add(T[0][c]==f)
    s.add(B[HEIGHT-1][c]==f)
```

Now we'll enumerate all cells in the initial array (7.6.1). First two cells are empty there. And the third one has type "2b". This is "┏" and it can be oriented in 4 possible ways. And if it has angle 0°, bottom and right half-joints are present, others are absent. If it has angle 90°, it looks like "┓", and bottom and left half-joints are present, others are absent.

In plain English: "if cell of this type has angle 0°, these half-joints must be present **OR** if it has angle 90°, these half-joints must be present, **OR**, etc, etc."

Likewise, we define all these rules for all types and all possible angles:

```
for r in range(HEIGHT):
    for c in range(WIDTH):
        ty=cells_type[r][c]

        if ty=="0":
            s.add(A[r][c]==f)
```

```
            s.add(T[r][c]==f, B[r][c]==f, L[r][c]==f, R[r][c]==f)

        if ty=="2a":
            s.add(Or(And(A[r][c]==0, L[r][c]==f, R[r][c]==f, T[r][c]==t, B[r][c]==t),
                # ┃
                    And(A[r][c]==90, L[r][c]==t, R[r][c]==t, T[r][c]==f, B[r][c]==f)))
                    # ━

        if ty=="2b":
            s.add(Or(And(A[r][c]==0, L[r][c]==f, R[r][c]==t, T[r][c]==f, B[r][c]==t),
                # ┏
                    And(A[r][c]==90, L[r][c]==t, R[r][c]==f, T[r][c]==f, B[r][c]==t),
                    # ┓
                    And(A[r][c]==180, L[r][c]==t, R[r][c]==f, T[r][c]==t, B[r][c]==f),
                    # ┛
                    And(A[r][c]==270, L[r][c]==f, R[r][c]==t, T[r][c]==t, B[r][c]==f)))
                    # ┗

        if ty=="3":
            s.add(Or(And(A[r][c]==0, L[r][c]==f, R[r][c]==t, T[r][c]==t, B[r][c]==t),
                # ┣
                    And(A[r][c]==90, L[r][c]==t, R[r][c]==t, T[r][c]==f, B[r][c]==t),
                    # ┳
                    And(A[r][c]==180, L[r][c]==t, R[r][c]==f, T[r][c]==t, B[r][c]==t),
                    # ┫
                    And(A[r][c]==270, L[r][c]==t, R[r][c]==t, T[r][c]==t, B[r][c]==f)))
                    # ┻

        if ty=="4":
            s.add(A[r][c]==0)
            s.add(T[r][c]==t, B[r][c]==t, L[r][c]==t, R[r][c]==t) # ╋
```

Full source code is here: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/pipe/solver/solve_pipe_puzzle1.py.

It produces this result (prints angle for each cell and (pseudo)graphical representation):

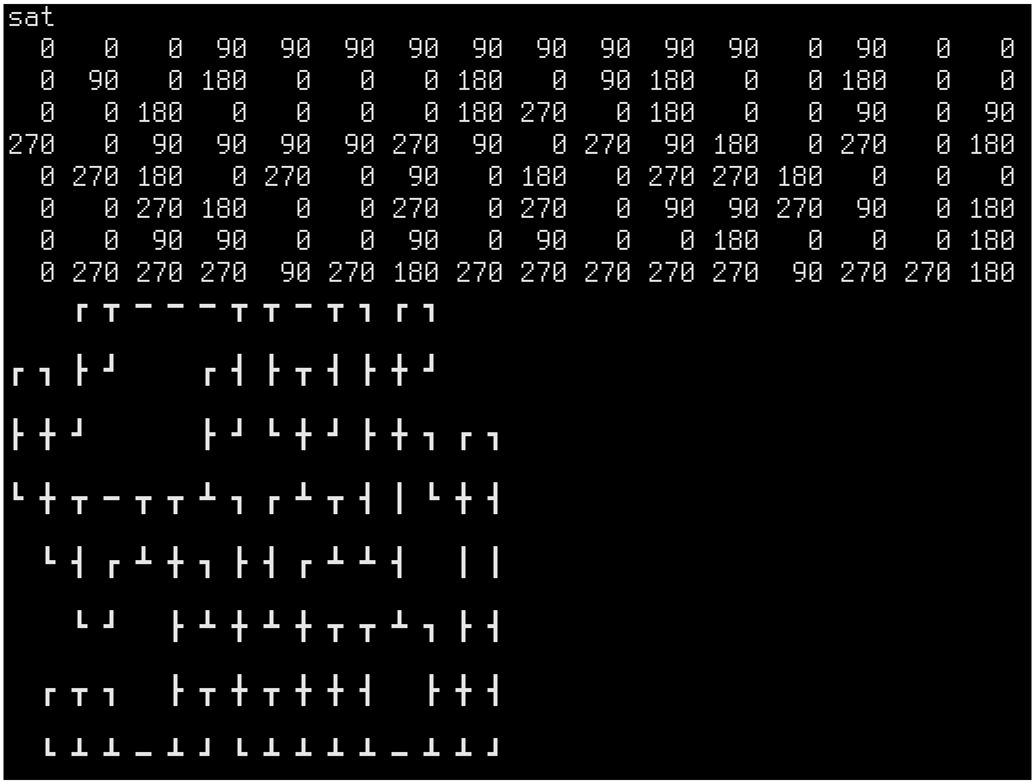



Figure 18: Solver script output

It worked ≈ 4 seconds on my old and slow Intel Atom N455 1.66GHz. Is it fast? I don't know, but again, what is really cool, we do not know about any mathematical background of all this, we just defined cells, (half-)joints and defined relations between them.

Now the next question is, how many solutions are possible? Using method described earlier (3.15), I've altered solver script [75] and solver said two solutions are possible.

Let's compare these two solutions using gvimdiff:

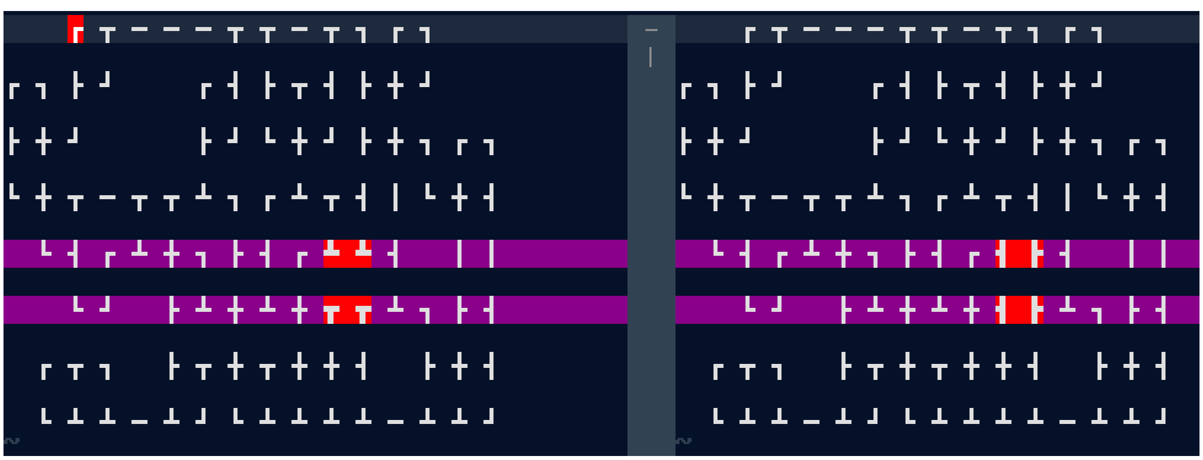


Figure 19: gvimdiff output (pardon my red cursor at left pane at left-top corner)

4 cells in the middle can be orientated differently. Perhaps, other puzzles may produce different results.

P.S. *Half-joint* is defined as boolean type. But in fact, the first version of the solver has been written using integer type for half-joints, and 0 was used for False and 1 for True. I did it so because I wanted to make source code tidier and

[75] https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/pipe/solver/solve_pipe_puzzle2.py

narrower without using long words like "False" and "True". And it worked, but slower. Perhaps, Z3 handles boolean data types faster? Better? Anyway, I writing this to note that integer type can also be used instead of boolean, if needed.

## 7.7 Eight queens problem (SAT)

Eight queens is a very popular problem and often used for measuring performance of SAT solvers. The problem is to place 8 queens on chess board so they will not attack each other. For example:

```
| | | |*| | | | |
| | | | | | |*| |
| | | | |*| | | |
| |*| | | | | | |
| | | | | |*| | |
|*| | | | | | | |
| | |*| | | | | |
| | | | | | | |*|
```

Let's try to figure out how to solve it.

### 7.7.1 make_one_hot

One important function we will (often) use is `make_one_hot`. This is a function which returns *True* if one single of inputs is *True* and others are *False*. It will return *False* otherwise.

In my other examples, I've used Wolfram Mathematica to generate CNF clauses for it, for example: 3.7. What expression Mathematica offers as `make_one_hot` function with 8 inputs?

```
(!a||!b)&&(!a||!c)&&(!a||!d)&&(!a||!e)&&(!a||!f)&&(!a||!g)&&(!a||!h)&&(a||b||c||d||e||f
    ||g||h)&&
(!b||!c)&&(!b||!d)&&(!b||!e)&&(!b||!f)&&(!b||!g)&&(!b||!h)&&(!c||!d)&&(!c||!e)&&(!c||!f)
    &&(!c||!g)&&
(!c||!h)&&(!d||!e)&&(!d||!f)&&(!d||!g)&&(!d||!h)&&(!e||!f)&&(!e||!g)&&(!e||!h)&&(!f||!g)
    &&(!f||!h)&&(!g||!h)
```

We can clearly see that the expression constisting of all possible variable pairs (negated) plus enumeration of all variables (non-negated). In plain English terms, this means: "no pair can be equal to two *True*'s *AND* at least one *True* must be present among all variables".

This is how it works: if two variables will be *True*, negated they will be both *False*, and this clause will not be evaluated to *True*, which is our ultimate goal. If one of variables is *True*, both negated will produce one *True* and one *False* (fine). If both variables are False, both negated will produce two *True's* (again, fine).

Here is how I can generate clauses for the function using *itertools* module from Python, which provides many important functions from combinatorics:

```
    # naive/pairwise encoding
    def AtMost1(self, lst):
        for pair in itertools.combinations(lst, r=2):
            self.add_clause([self.neg(pair[0]), self.neg(pair[1])])

    # make one-hot (AKA unitary) variable
    def make_one_hot(self, lst):
        self.AtMost1(lst)
        self.OR_always(lst)
```

`AtMost1()` function enumerates all possible pairs using *itertools* function *combinations()*.

`make_one_hot()` function does the same, but also adds a final clause, which forces at least one *True* variable to be present.

What clauses will be generated for 5 variables (1..5)?

```
p cnf 5 11
-2 -5 0
```

```
-2 -4 0
-4 -5 0
-2 -3 0
-1 -4 0
-1 -5 0
-1 -2 0
-1 -3 0
-3 -4 0
-3 -5 0
1 2 3 4 5 0
```

Yes, these are all possible pairs of 1..5 numbers + all 5 numbers.

We can get all solutions using Picosat:

```
% picosat --all popcnt1.cnf

s SATISFIABLE
v -1 -2 -3 -4 5 0
s SATISFIABLE
v -1 -2 -3 4 -5 0
s SATISFIABLE
v -1 -2 3 -4 -5 0
s SATISFIABLE
v -1 2 -3 -4 -5 0
s SATISFIABLE
v 1 -2 -3 -4 -5 0
s SOLUTIONS 5
```

5 possible solutions indeed.

### 7.7.2 Eight queens

Now let's get back to eight queens.

We can assign 64 variables to $8 \cdot 8 = 64$ cells. Cell occuppied with queen will be *True*, vacant cell will be *False*.

The problem of placing non-attacking (each other) queens on chess board (of any size) can be stated in plain English like this:

- one single queen must be present at each row;
- one single queen must be present at each column;
- zero or one queen must be present at each diagonal (empty diagonals can be present in valid solution).

These rules can be translated like that:

- make_one_hot(each row)==*True*
- make_one_hot(each column)==*True*
- AtMost1(each diagonal)==*True*

Now all we need is to enumerate rows, columns and diagonals and gather all clauses:

```
#!/usr/bin/env python3

#-*- coding: utf-8 -*-

import itertools, subprocess, os, my_utils, SAT_lib

SIZE=8
SKIP_SYMMETRIES=True
```

```
#SKIP_SYMMETRIES=False

def row_col_to_var(row, col):
    global first_var
    return str(row*SIZE+col+first_var)

def gen_diagonal(s, start_row, start_col, increment, limit):
    col=start_col
    tmp=[]
    for row in range(start_row, SIZE):
        tmp.append(row_col_to_var(row, col))
        col=col+increment
        if col==limit:
            break
    # ignore diagonals consisting of 1 cell:
    if len(tmp)>1:
        # we can't use POPCNT1() here, since some diagonals are empty in valid solutions
            .
        s.AtMost1(tmp)

def add_2D_array_as_negated_constraint(s, a):
    negated_solution=[]
    for row in range(SIZE):
        for col in range(SIZE):
            negated_solution.append(s.neg_if(a[row][col], row_col_to_var(row, col)))
    s.add_clause(negated_solution)

def main():
    global first_var

    s=SAT_lib.SAT_lib(False)

    _vars=s.alloc_BV(SIZE**2)
    first_var=int(_vars[0])

    # enumerate all rows:
    for row in range(SIZE):
        s.make_one_hot([row_col_to_var(row, col) for col in range(SIZE)])

    # enumerate all columns:
    # make_one_hot() could be used here as well:
    for col in range(SIZE):
        s.AtMost1([row_col_to_var(row, col) for row in range(SIZE)])

    # enumerate all diagonals:
    for row in range(SIZE):
        for col in range(SIZE):
            gen_diagonal(s, row, col, 1, SIZE) # from L to R
            gen_diagonal(s, row, col, -1, -1)  # from R to L

    # find all solutions:
    sol_n=1
    while True:
        if s.solve()==False:
            print ("unsat!")
            print ("solutions total=", sol_n-1)
```

```
            exit(0)

        # print solution:
        print ("solution number", sol_n, ":")

        # get solution and make 2D array of bools:
        solution_as_2D_bool_array=[]
        for row in range(SIZE):
            solution_as_2D_bool_array.append ([s.get_var_from_solution(row_col_to_var(
                row, col)) for col in range(SIZE)])

        # print 2D array:
        for row in range(SIZE):
            tmp=[(["  ", "*"][solution_as_2D_bool_array[row][col]]+"|") for col in range(
                SIZE)]
            print ("|"+"".join(tmp))

        # add 2D array as negated constraint:
        add_2D_array_as_negated_constraint(s, solution_as_2D_bool_array)

        # if we skip symmetries, rotate/reflect soluion and add them as negated
           constraints:
        if SKIP_SYMMETRIES:
            for a in range(4):
                tmp=my_utils.rotate_rect_array(solution_as_2D_bool_array, a)
                add_2D_array_as_negated_constraint(s, tmp)

                tmp=my_utils.reflect_horizontally(my_utils.rotate_rect_array(
                    solution_as_2D_bool_array, a))
                add_2D_array_as_negated_constraint(s, tmp)

        sol_n=sol_n+1

main()
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/8queens/8queens.py )

Perhaps, `gen_diagonal()` function is not very aesthetically appealing: it enumerates also subdiagonals of already enumerated longer diagonals. To prevent presence of duplicate clauses, *clauses* global variable is not a list, rather set, which allows only unique data to be present there.

Also, I've used `AtMost1` for each column, this will help to produce slightly lower number of clauses. Each column will have a queen anyway, this is implied from the first rule (`make_one_hot` for each row).

After running, we got CNF file with 64 variables and 736 clauses (https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/8queens/8queens.cnf). Here is one solution:

```
% python 8queens.py
len(clauses)= 736
| | | |*| | | | |
| | | | | | |*| |
| | | | |*| | | |
| |*| | | | | | |
| | | | | |*| | |
|*| | | | | | | |
| | |*| | | | | |
| | | | | | | |*|
```

How many possible solutions are there? Picosat tells 92, which is indeed correct number of solutions (https://oeis.org/A000170).

Performance of Picosat is not impressive, probably because it has to output all the solutions. It took 34 seconds on my ancient Intel Atom 1.66GHz netbook to enumerate all solutions for $11 \cdot 11$ chess board (2680 solutions), which is way slower than my straigt brute-force program: https://yurichev.com/blog/8queens/. Nevertheless, it's lighting fast (as other SAT solvers) in finding first solution.

The SAT instance is also small enough to be easily solved by my simplest possible backtracking SAT solver: 22.1.

### 7.7.3 Counting all solutions

We get a solution, negate it and add as a new constraint. In plain English language this sounds "find a solution, which is also can't be equal to the recently found/added". We add them consequently and the process is slowing—just because a problem (*instance*) is growing and SAT solver experience hard times in find yet another solution.

### 7.7.4 Skipping symmetrical solutions

We can also add rotated and reflected (horizontally) solution, so to skip symmetrical solutions. By doing so, we're getting 12 solutions for 8*8 board, 46 for 9*9 board, etc. This is https://oeis.org/A002562.

## 7.8 Solving pocket Rubik's cube (2*2*2) using Z3

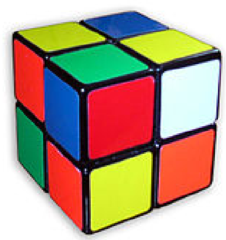


Figure 20: Pocket cube

( The image has been taken from Wikipedia. )

Solving Rubik's cube is not a problem, finding shortest solution is.

### 7.8.1 Intro

First, a bit of terminology. There are 6 colors we have: white, green, blue, orange, red, yellow. We also have 6 sides: front, up, down, left, right, back.

This is how we will name all facelets:

```
        U1 U2
        U3 U4

       -------
L1 L2 | F1 F2 | R1 R2 | B1 B2
L3 L4 | F3 F4 | R3 R4 | B3 B4
       -------

        D1 D2
        D3 D4
```

Colors on a solved cube are:

```
    G G
    G G
    ---
R R|W W|O O|Y Y
R R|W W|O O|Y Y
```

```
    ---
    B B
    B B
```

There are 6 possible turns: front, left, right, back, up, down. But each turn can be clockwise, counterclockwise and half-turn (equal to two CW or two CCW). Each CW is equal to 3 CCW and vice versa. Hence, there are 6*3=18 possible turns.

It is known, that 11 turns (including half-turns) are enough to solve any pocket cube (God's algorithm). This means, graph has a diameter of 11. For 3*3*3 cube one need 20 turns (http://www.cube20.org/). See also: https://en.wikipedia.org/wiki/Rubik%27s_Cube_group.

### 7.8.2 Z3

There are 6 sides and 4 facelets on each, hence, 6*4=24 variables we need to define a state.

Then we define how state is transformed after each possible turn:

```
FACE_F, FACE_U, FACE_D, FACE_R, FACE_L, FACE_B = 0,1,2,3,4,5

def rotate_FCW(s):
    return [
        [ s[FACE_F][2], s[FACE_F][0], s[FACE_F][3], s[FACE_F][1] ],   # for F
        [ s[FACE_U][0], s[FACE_U][1], s[FACE_L][3], s[FACE_L][1] ],   # for U
        [ s[FACE_R][2], s[FACE_R][0], s[FACE_D][2], s[FACE_D][3] ],   # for D
        [ s[FACE_U][2], s[FACE_R][1], s[FACE_U][3], s[FACE_R][3] ],   # for R
        [ s[FACE_L][0], s[FACE_D][0], s[FACE_L][2], s[FACE_D][1] ],   # for L
        [ s[FACE_B][0], s[FACE_B][1], s[FACE_B][2], s[FACE_B][3] ] ]  # for B

def rotate_FH(s):
    return [
        [ s[FACE_F][3], s[FACE_F][2], s[FACE_F][1], s[FACE_F][0] ],
        [ s[FACE_U][0], s[FACE_U][1], s[FACE_D][1], s[FACE_D][0] ],
        [ s[FACE_U][3], s[FACE_U][2], s[FACE_D][2], s[FACE_D][3] ],
        [ s[FACE_L][3], s[FACE_R][1], s[FACE_L][1], s[FACE_R][3] ],
        [ s[FACE_L][0], s[FACE_R][2], s[FACE_L][2], s[FACE_R][0] ],
        [ s[FACE_B][0], s[FACE_B][1], s[FACE_B][2], s[FACE_B][3] ] ]

...
```

Then we define a function, which takes turn number and transforms a state:

```
# op is turn number
def rotate(turn, state, face, facelet):
    return If(op==0,  rotate_FCW (state)[face][facelet],
           If(op==1,  rotate_FCCW(state)[face][facelet],
           If(op==2,  rotate_UCW (state)[face][facelet],
           If(op==3,  rotate_UCCW(state)[face][facelet],
           If(op==4,  rotate_DCW (state)[face][facelet],

...

           If(op==17, rotate_BH  (state)[face][facelet],
                      0))))))))))))))))))
```

Now set "solved" state, initial state and connect everything:

```
move_names=["FCW", "FCCW", "UCW", "UCCW", "DCW", "DCCW", "RCW", "RCCW", "LCW", "LCCW", "
   BCW", "BCCW", "FH", "UH", "DH", "RH", "LH", "BH"]

def colors_to_array_of_ints(s):
```

```
    return [{"W":0, "G":1, "B":2, "O":3, "R":4, "Y":5}[c] for c in s]

def set_current_state (d):
    F=colors_to_array_of_ints(d["FACE_F"])
    U=colors_to_array_of_ints(d["FACE_U"])
    D=colors_to_array_of_ints(d["FACE_D"])
    R=colors_to_array_of_ints(d["FACE_R"])
    L=colors_to_array_of_ints(d["FACE_L"])
    B=colors_to_array_of_ints(d["FACE_B"])
    return F,U,D,R,L,B # return tuple

# 4
init_F, init_U, init_D, init_R, init_L, init_B=set_current_state({"FACE_F":"RYOG", "
   FACE_U":"YRGO", "FACE_D":"WRBO", "FACE_R":"GYWB", "FACE_L":"BYWG", "FACE_B":"BOWR"})

...

for TURNS in range(1,12): # 1..11
    print "turns=", TURNS

    s=Solver()

    state=[[[Int('state%d_%d_%d' % (n, side, i)) for i in range(FACELETS)] for side in
       range(FACES)] for n in range(TURNS+1)]
    op=[Int('op%d' % n) for n in range(TURNS+1)]

    for i in range(FACELETS):
        s.add(state[0][FACE_F][i]==init_F[i])
        s.add(state[0][FACE_U][i]==init_U[i])
        s.add(state[0][FACE_D][i]==init_D[i])
        s.add(state[0][FACE_R][i]==init_R[i])
        s.add(state[0][FACE_L][i]==init_L[i])
        s.add(state[0][FACE_B][i]==init_B[i])

    # solved state
    for face in range(FACES):
        for facelet in range(FACELETS):
            s.add(state[TURNS][face][facelet]==face)

    # turns:
    for turn in range(TURNS):
        for face in range(FACES):
            for facelet in range(FACELETS):
                s.add(state[turn+1][face][facelet]==rotate(op[turn], state[turn], face,
                   facelet))

    if s.check()==sat:
        print "sat"
        m=s.model()
        for turn in range(TURNS):
            print move_names[int(str(m[op[turn]]))]
        exit(0)
```

( The full source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/rubik2/failed_SMT/rubik2_z3.py )

That works:

```
turns= 1
turns= 2
turns= 3
turns= 4
sat
RCW
UCW
DCW
RCW
```

...but very slow. It takes up to 1 hours to find a path of 8 turns, which is not enough, we need 11.

Nevetheless, I decided to include Z3 solver as a demonstration.

## 7.9 Pocket Rubik's Cube (2*2*2) and SAT solver

.

I had success with my SAT-based solver, which can find an 11-turn path for a matter of 10-20 minutes on my old Intel Xeon E3-1220 3.10GHz.

First, we will encode each color as 3-bit bit vector. Then we can build electronic circuit, which will take initial state of cube and output final state. It can have switches for each turn on each state.

```
                  +-----+     +-----+      +-----+
initial state -> | blk | -> | blk | ... | blk | -> final state
                  +-----+     +-----+      +-----+
                     ^           ^            ^
                     |           |            |
                  turn 1      turn 2     last turn
```

You set all turns and the device "calculates" final state.

Each "blk" can be constisted of 24 multiplexers (MUX), each for each facelet. Each MUX is controlled by 5-bit command (turn number). Depending on command, MUX takes 3-bit color from a facelet from a previous state.

Here is a table: the first column is a "destination" facelet, then a list of "source" facelets for each turn:

```
    #      dst,  FCW   FH   FCCW UCW   UH   UCCW DCW   DH   DCCW RCW   RH   RCCW LCW   LH
        LCCW BCW   BH   BCCW
    add_r("F1",["F3","F4","F2","R1","B1","L1","F1","F1","F1","F1","F1","F1","U1","B4","
        D1","F1","F1","F1"])
    add_r("F2",["F1","F3","F4","R2","B2","L2","F2","F2","F2","D2","B3","U2","F2","F2","
        F2","F2","F2","F2"])
    add_r("F3",["F4","F2","F1","F3","F3","F3","L3","B3","R3","F3","F3","F3","U3","B2","
        D3","F3","F3","F3"])
    add_r("F4",["F2","F1","F3","F4","F4","F4","L4","B4","R4","D4","B1","U4","F4","F4","
        F4","F4","F4","F4"])
    add_r("U1",["U1","U1","U1","U3","U4","U2","U1","U1","U1","U1","U1","U1","B4","D1","
        F1","R2","D4","L3"])
    add_r("U2",["U2","U2","U2","U1","U3","U4","U2","U2","U2","F2","D2","B3","U2","U2","
        U2","R4","D3","L1"])
    add_r("U3",["L4","D2","R1","U4","U2","U1","U3","U3","U3","U3","U3","U3","B2","D3","
        F3","U3","U3","U3"])
    add_r("U4",["L2","D1","R3","U2","U1","U3","U4","U4","U4","F4","D4","B1","U4","U4","
        U4","U4","U4","U4"])
    add_r("D1",["R3","U4","L2","D1","D1","D1","D3","D4","D2","D1","D1","D1","F1","U1","
        B4","D1","D1","D1"])
    add_r("D2",["R1","U3","L4","D2","D2","D2","D1","D3","D4","B3","U2","F2","D2","D2","
        D2","D2","D2","D2"])
    add_r("D3",["D3","D3","D3","D3","D3","D3","D4","D2","D1","D3","D3","D3","F3","U3","
        B2","L1","U2","R4"])
```

```
    add_r("D4",["D4","D4","D4","D4","D4","D4","D2","D1","D3","B1","U4","F4","D4","D4","
        D4","L3","U1","R2"])
    add_r("R1",["U3","L4","D2","B1","L1","F1","R1","R1","R1","R3","R4","R2","R1","R1","
        R1","R1","R1","R1"])
    add_r("R2",["R2","R2","R2","B2","L2","F2","R2","R2","R2","R1","R3","R4","R2","R2","
        R2","D4","L3","U1"])
    add_r("R3",["U4","L2","D1","R3","R3","R3","F3","L3","B3","R4","R2","R1","R3","R3","
        R3","R3","R3","R3"])
    add_r("R4",["R4","R4","R4","R4","R4","R4","F4","L4","B4","R2","R1","R3","R4","R4","
        R4","D3","L1","U2"])
    add_r("L1",["L1","L1","L1","F1","R1","B1","L1","L1","L1","L1","L1","L1","L3","L4","
        L2","U2","R4","D3"])
    add_r("L2",["D1","R3","U4","F2","R2","B2","L2","L2","L2","L2","L2","L2","L1","L3","
        L4","L2","L2","L2"])
    add_r("L3",["L3","L3","L3","L3","L3","L3","B3","R3","F3","L3","L3","L3","L4","L2","
        L1","U1","R2","D4"])
    add_r("L4",["D2","R1","U3","L4","L4","L4","B4","R4","F4","L4","L4","L4","L2","L1","
        L3","L4","L4","L4"])
    add_r("B1",["B1","B1","B1","L1","F1","R1","B1","B1","B1","U4","F4","D4","B1","B1","
        B1","B3","B4","B2"])
    add_r("B2",["B2","B2","B2","L2","F2","R2","B2","B2","B2","B2","B2","B2","D3","F3","
        U3","B1","B3","B4"])
    add_r("B3",["B3","B3","B3","B3","B3","B3","R3","F3","L3","U2","F2","D2","B3","B3","
        B3","B4","B2","B1"])
    add_r("B4",["B4","B4","B4","B4","B4","B4","R4","F4","L4","B4","B4","B4","D1","F1","
        U1","B2","B1","B3"])
```

Each MUX has 32 inputs, each has width of 3 bits: colors from "source" facelets. It has 3-bit output (color for "destination" facelet). It has 5-bit selector, for 18 turns. Other selector values (32-18=14 values) are not used at all.

The whole problem is to build a circuit and then ask SAT solver to set "switches" to such a state, when input and output are determined (by us).

Now the problem is to represent MUX in CNF terms.

From EE[76] courses we can remember about a simple if-then-else (ITE) gate, it takes 3 inputs ("selector", "true" and "false") and it has 1 output. Depending on "selector" input it "connects" output with "true" or "false" input. Using tree of ITE gates we first can build 32-to-1 MUX, then wide 32*3-to-3 MUX.

I once have written small utility to search for shortest possible CNF formula for a specific function, in a brute-force manner (`https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/rubik2/SAT/XOR_CNF_bf.c`). It was inspired by "aha! hacker assistant" by Henry Warren. So here is a function:

```
bool func(bool v[VARIABLES])
{

        // ITE:

        bool tmp;
        if (v[0]==0)
                tmp=v[1];
        else
                tmp=v[2];

        return tmp==v[3];
}
```

A shortest CNF for it:

```
try_all_CNFs_of_len(1)
```

[76] Electrical engineering

```
try_all_CNFs_of_len(2)
try_all_CNFs_of_len(3)
try_all_CNFs_of_len(4)
found a CNF:
p cnf 4 4
-1 3 -4 0
1 2 -4 0
-1 -3 4 0
1 -2 4 0
```

1st variable is "select", 2nd is "false", 3rd is "true", 4th is "output". "output" is an additional variable, added just like in Tseitin transformations.

Hence, CNF formula is:

```
(!select OR true OR !output) AND (select OR false OR !output) AND (!select OR !true OR
    output) AND (select OR !false OR output)
```

It assures that the "output" will always be equal to one of inputs depending on "select".

Now we can build a tree of ITE gates:

```
def create_ITE(s,f,t):
    x=create_var()

    clauses.append([neg(s),t,neg(x)])
    clauses.append([s,f,neg(x)])
    clauses.append([neg(s),neg(t),x])
    clauses.append([s,neg(f),x])

    return x

# ins=16 bits
# sel=4 bits
def create_MUX(ins, sel):
    t0=create_ITE(sel[0],ins[0],ins[1])
    t1=create_ITE(sel[0],ins[2],ins[3])
    t2=create_ITE(sel[0],ins[4],ins[5])
    t3=create_ITE(sel[0],ins[6],ins[7])
    t4=create_ITE(sel[0],ins[8],ins[9])
    t5=create_ITE(sel[0],ins[10],ins[11])
    t6=create_ITE(sel[0],ins[12],ins[13])
    t7=create_ITE(sel[0],ins[14],ins[15])

    y0=create_ITE(sel[1],t0,t1)
    y1=create_ITE(sel[1],t2,t3)
    y2=create_ITE(sel[1],t4,t5)
    y3=create_ITE(sel[1],t6,t7)

    z0=create_ITE(sel[2],y0,y1)
    z1=create_ITE(sel[2],y2,y3)

    return create_ITE(sel[3], z0, z1)
```

This is my old MUX I wrote for 16 inputs and 4-bit selector, but you've got the idea: this is 4-tier tree. It has 15 ITE gates or 15*4=60 clauses.

Now the question, is it possible to make it smaller? I've tried to use Mathematica. First I've built truth table for 4-bit selector:

```
...
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 0 1 1    0    0
```

```
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 0 1 1     1     1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 0 0     0     0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 0 0     1     1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 0 1     0     0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 0 1     1     1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 0     0     0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 0     1     1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1     0     0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1     1     1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 0 0     0     0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 0 0     1     1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 0 1     0     0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 0 1     1     1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 1 0     0     0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 1 0     1     1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 1 1     0     0
...
```

First 4 bits is selector, then 16 bits of input. Then the possible output and the bit, indicating, if the output bit equals to one of the inputs.

Then I load this table to Mathematica and make CNF expression out of truth table:

```
arr=Import["/home/dennis/P/Rubik/TT","Table"]
{{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1},{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0},{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1},  2097135  ,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0},{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,0},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1},{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0},{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}}



TT=(Most[#]->Last[#])&/@arr
{{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}->1,{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1}->0,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0}->0,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1}->1,{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0}->1,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1}->0,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0}->0,{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1}->1,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0}->1,  2097135  ,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0}->0,{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1}->1,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0}->0,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1}->1,{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0}->0,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1}->1,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0}->0,{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}->1}



BooleanConvert[BooleanFunction[TT,{s3,s2,s1,s0, i15,i14,i13,i12,i11,i10,i9,i8,i7,i6,i5,i4,i3, i2,
    i1, i0,x}],"CNF"]
(!i0||s0||s1||s2||s3||x)&&(i0||s0||s1||s2||s3||!x)&&(!i1||!s0||s1||s2||s3||x)&&(i1||!s0||s1||s2||
    s3||!x)&&(!i10||s0||!s1||s2||!s3||x)&&
(i10||s0||!s1||s2||!s3||!x)&&(!i11||!s0||!s1||s2||!s3||x)&&(i11||!s0||!s1||s2||!s3||!x)&&(!i12||
    s0||s1||!s2||!s3||x)&&(i12||s0||s1||!s2||!s3||!x)&&
(!i13||!s0||s1||!s2||!s3||x)&&(i13||!s0||s1||!s2||!s3||!x)&&(!i14||s0||!s1||!s2||!s3||x)&&(i14||
    s0||!s1||!s2||!s3||!x)&&
(!i15||!s0||!s1||!s2||!s3||x)&&(i15||!s0||!s1||!s2||!s3||!x)&&(!i2||s0||!s1||s2||s3||x)&&(i2||s0
    ||!s1||s2||s3||!x)&&(!i3||!s0||!s1||s2||s3||x)&&
```

```
(i3||!s0||!s1||s2||s3||!x)&&(!i4||s0||s1||!s2||s3||x)&&(i4||s0||s1||!s2||s3||!x)&&(!i5||!s0||s1
    ||!s2||s3||x)&&(i5||!s0||s1||!s2||s3||!x)&&
(!i6||s0||!s1||!s2||s3||x)&&(i6||s0||!s1||!s2||s3||!x)&&(!i7||!s0||!s1||!s2||s3||x)&&(i7||!s0||!
    s1||!s2||s3||!x)&&(!i8||s0||s1||s2||!s3||x)&&
(i8||s0||s1||s2||!s3||!x)&&(!i9||!s0||s1||s2||!s3||x)&&(i9||!s0||s1||s2||!s3||!x)
```

Look closer to CNF expression:

```
(!i0|   |s0||s1||s2|| s3|| x)&&
( i0||  s0|| s1||s2|| s3||!x)&&
(!i1||! s0|| s1||s2|| s3|| x)&&
( i1||! s0|| s1||s2|| s3||!x)&&
(!i10|| s0||!s1||s2||!s3|| x)&&
( i10|| s0||!s1||s2||!s3||!x)&&
(!i11||!s0||!s1||s2||!s3|| x)&&
( i11||!s0||!s1||s2||!s3||!x)&&

...

(!i8|| s0||s1||s2||!s3|| x)&&
( i8|| s0||s1||s2||!s3||!x)&&
(!i9||!s0||s1||s2||!s3|| x)&&
( i9||!s0||s1||s2||!s3||!x)
```

It has simple structure and there are 32 clauses, against 60 in my previous attempt. Will it work faster? No, as my experience shows, it doesn't speed up anything. Anyway, I used the latter idea to make MUX.

The following function makes pack of MUXes for each state, based on what I've got from Mathematica:

```
    def create_MUX(self, ins, sels):
        assert 2**len(sels)==len(ins)
        x=self.create_var()
        for sel in range(len(ins)): # 32 for 5-bit selector
            tmp=[self.neg_if((sel>>i)&1==1, sels[i]) for i in range(len(sels))] # 5 for
               5-bit selector

            self.add_clause([self.neg(ins[sel])] + tmp + [x])
            self.add_clause([ins[sel]] + tmp + [self.neg(x)])
        return x

    def create_wide_MUX (self, ins, sels):
        out=[]
        for i in range(len(ins[0])):
            inputs=[x[i] for x in ins]
            out.append(self.create_MUX(inputs, sels))
        return out
```

Now the function that glues all together:

```
# src=list of 18 facelets
def add_r(dst, src):
    global facelets, selectors, s
    t=s.create_var()
    s.fix(t,True)
    for state in range(STATES-1):
        src_vectors=[]
        for tmp in src:
            src_vectors.append(facelets[state][tmp])

        # padding: there are 18 used MUX inputs, so add 32-18=14 for padding
```

```
        for i in range(32-18):
            src_vectors.append([t,t,t])

        dst_vector=s.create_wide_MUX (src_vectors, selectors[state])
        s.fix_BV_EQ(dst_vector,facelets[state+1][dst])

...

    #      dst,  FCW   FH    FCCW  UCW   UH    UCCW  DCW   DH    DCCW  RCW   RH    RCCW  LCW   LH
        LCCW  BCW   BH    BCCW
    add_r("F1",["F3","F4","F2","R1","B1","L1","F1","F1","F1","F1","F1","F1","U1","B4","
        D1","F1","F1","F1"])
    add_r("F2",["F1","F3","F4","R2","B2","L2","F2","F2","F2","D2","B3","U2","F2","F2","
        F2","F2","F2","F2"])
    add_r("F3",["F4","F2","F1","F3","F3","F3","L3","B3","R3","F3","F3","F3","U3","B2","
        D3","F3","F3","F3"])
    add_r("F4",["F2","F1","F3","F4","F4","F4","L4","B4","R4","D4","B1","U4","F4","F4","
        F4","F4","F4","F4"])
    add_r("U1",["U1","U1","U1","U3","U4","U2","U1","U1","U1","U1","U1","U1","B4","D1","
        F1","R2","D4","L3"])
...
```

Now the full source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/rubik2/SAT/solver.py. I tried to make it as concise as possible. It requires minisat to be installed.

And it works up to 11 turns, starting at 11, then decreasing number of turns. Here is an output for a state which can be solved with 4 turns:

```
set_state(0, {"F":"RYOG", "U":"YRGO", "D":"WRBO", "R":"GYWB", "L":"BYWG", "B":"BOWR"})
```

```
TURNS= 11
sat!
RCW
DH
BCW
UCW
UH
DCCW
FH
BH
BH
FH
UH

TURNS= 10
sat!
RCCW
UCCW
UCW
RCW
RCW
UH
FCW
UCCW
DCCW
DH

TURNS= 9
```

```
sat!
BCCW
LH
RH
BCW
RCW
RCCW
DCW
FH
LCW

TURNS= 8
sat!
RCW
UH
BCW
UCCW
LCW
LH
RCW
FCW

TURNS= 7
sat!
DCCW
DH
UH
UCW
BCW
UH
RCW

TURNS= 6
sat!
RCW
UCCW
DCCW
LCW
UH
DH

TURNS= 5
sat!
LCW
FCW
BCW
RH
LCCW

TURNS= 4
sat!
RCW
DCW
UCW
RCW
```

```
TURNS= 3
unsat!
```

Even on my relic Intel Atom 1.5GHz netbook it takes just 20s.

You can find redundant turns in 11-turn solution, like double UH turns. Of course, two UH turns returns the cube to the previous state. So these "annihilating turns" are added if the final solution can be shorter. Why the solver added it? There is no "no operation" turn. And the solver is forced to fit into 11 turns. Hence, it do what it can to produce correct solution.

Now a hard example:

```
set_state(0, {"F":"RORW", "U":"BRBB", "D":"GOOR", "R":"WYGY", "L":"OWYW", "B":"BYGG"})
```

```
TURNS= 11
sat!
UCW
BCW
LCCW
BH
DCW
RH
DCCW
FCW
UH
LCW
RCW

TURNS= 10
sat!
RH
BCCW
LCCW
RH
BCCW
LH
BCW
DH
BCW
LCCW

...
```

( $\approx$ 5 minutes on my old Intel Xeon E3-1220 3.10GHz. )

I couldn't find a pure "11-turn state" which is "unsat" for 10-turn, it seems, these are rare. According to wikipedia, there are just 0.072% of these states. Like "20-turn states" for 3*3*3 cube, which are also rare.

### 7.9.1 Several solutions

According to picosat (–all option to get all possible solutions), the 4-turn example we just saw has 2 solutions. Indeed, two consequent turns UCW and DCW can be interchanged, they do not conflict with each other.

### 7.9.2 Other (failed) ideas

Pocket cube (2*2*2) has no central facelets, so to solve it, you don't need to stick each color to each face. Rather, you can define a constraint so that a colors on each face must be equal to each other. Somehow, this slows down drastically my both Z3 and SAT-based solvers.

Also, to prevent "annihilating" turns, we can set a constraint so that each state cannot be equal to any of previous states, i.e., states cannot repeat. This also slows down my both solvers.

#### 7.9.3 3*3*3 cube

3*3*3 cube requires much more turns (20), so I couldn't solve it with my methods. I have success to solve maybe 10 or 11 turns. But some people do all 20 turns: Jingchao Chen.

However, you can use 3*3*3 cube to play, because it can act as 2*2*2 cube: just use corners and ignore edge and center cubes. Here is mine I used, you can see that corners are correctly solved:

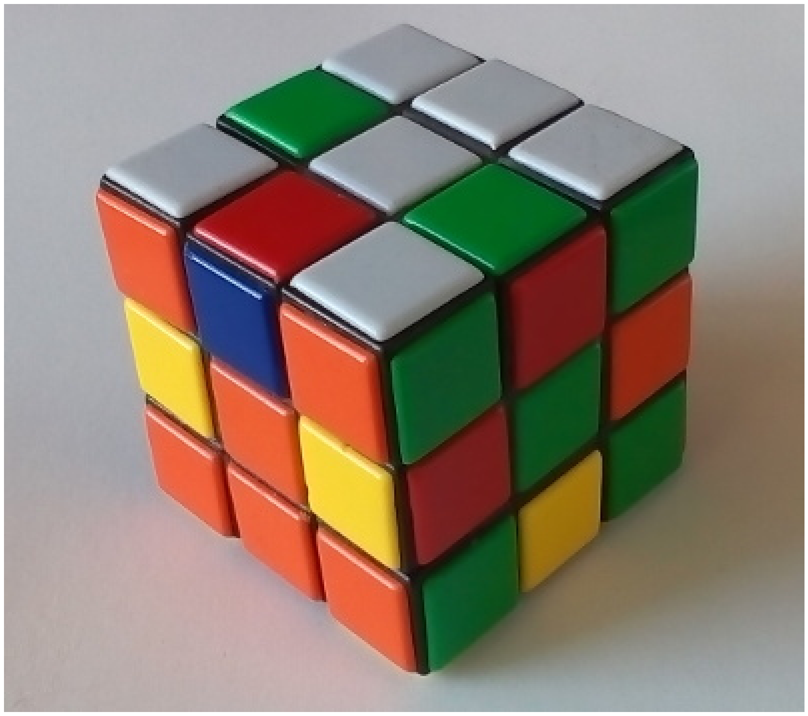


Figure 21: Solved Rubik's cube as a pocket cube

#### 7.9.4 Some discussion

https://news.ycombinator.com/item?id=15214439,
https://www.reddit.com/r/compsci/comments/6zb34i/solving_pocket_rubiks_cube_222_using_z3_and_sat/,
https://www.reddit.com/r/Cubers/comments/6ze3ua/theory_dennis_yurichev_solving_pocket_rubiks_cube/.

## 7.10 Rubik's cube (3*3*3) and Z3 SMT-solver

As I wrote before, I couldn't solve even 2*2*2 pocket cube with Z3 (7.8), but I wanted to play with it further, and found that it's still possible to solve one face instead of all 6.

I tried to model color of each facelet using integer sort (type), but now I can use bool: white facelet is True and all other non-white is False. I can encode state of Rubik's cube like that: "W" for white facelet, "." for other.

Now the process of solving is a matter of minutes on a decent computer, or faster.

```
#!/usr/bin/env python

from z3 import *

FACES=6
FACELETS=9

def rotate_FCW(s):
        return [
                [ s[0][6], s[0][3], s[0][0], s[0][7], s[0][4], s[0][1], s[0][8], s
                    [0][5], s[0][2], ],  # new F
```

```
                [ s[1][0], s[1][1], s[1][2], s[1][3], s[1][4], s[1][5], s[4][8], s
                    [4][5], s[4][2], ],  # new U
                [ s[3][6], s[3][3], s[3][0], s[2][3], s[2][4], s[2][5], s[2][6], s
                    [2][7], s[2][8], ],  # new D
                [ s[1][6], s[3][1], s[3][2], s[1][7], s[3][4], s[3][5], s[1][8], s
                    [3][7], s[3][8], ],  # new R
                [ s[4][0], s[4][1], s[2][0], s[4][3], s[4][4], s[2][1], s[4][6], s
                    [4][7], s[2][2], ],  # new L
                [ s[5][0], s[5][1], s[5][2], s[5][3], s[5][4], s[5][5], s[5][6], s
                    [5][7], s[5][8], ] ] # new B

def rotate_FH(s):
        return [
                [ s[0][8], s[0][7], s[0][6], s[0][5], s[0][4], s[0][3], s[0][2], s
                    [0][1], s[0][0], ],
                [ s[1][0], s[1][1], s[1][2], s[1][3], s[1][4], s[1][5], s[2][2], s
                    [2][1], s[2][0], ],
                [ s[1][8], s[1][7], s[1][6], s[2][3], s[2][4], s[2][5], s[2][6], s
                    [2][7], s[2][8], ],
                [ s[4][8], s[3][1], s[3][2], s[4][5], s[3][4], s[3][5], s[4][2], s
                    [3][7], s[3][8], ],
                [ s[4][0], s[4][1], s[3][6], s[4][3], s[4][4], s[3][3], s[4][6], s
                    [4][7], s[3][0], ],
                [ s[5][0], s[5][1], s[5][2], s[5][3], s[5][4], s[5][5], s[5][6], s
                    [5][7], s[5][8], ] ]


...


def rotate(op, st, side, j):
        return If(op==0, rotate_FCW(st)[side][j],
                If(op==1, rotate_FCCW(st)[side][j],
                If(op==2, rotate_UCW(st)[side][j],
                If(op==3, rotate_UCCW(st)[side][j],
                If(op==4, rotate_DCW(st)[side][j],
                If(op==5, rotate_DCCW(st)[side][j],
                If(op==6, rotate_RCW(st)[side][j],
                If(op==7, rotate_RCCW(st)[side][j],
                If(op==8, rotate_LCW(st)[side][j],
                If(op==9, rotate_LCCW(st)[side][j],
                If(op==10, rotate_BCW(st)[side][j],
                If(op==11, rotate_BCCW(st)[side][j],
                If(op==12, rotate_FH(st)[side][j],
                If(op==13, rotate_UH(st)[side][j],
                If(op==14, rotate_DH(st)[side][j],
                If(op==15, rotate_RH(st)[side][j],
                If(op==16, rotate_LH(st)[side][j],
                If(op==17, rotate_BH(st)[side][j],
                        rotate_BH(st)[side][j], # default
                        ))))))))))))))))))

move_names=["FCW", "FCCW", "UCW", "UCCW", "DCW", "DCCW", "RCW", "RCCW", "LCW", "LCCW", "
   BCW", "BCCW", "FH", "UH", "DH", "RH", "LH", "BH"]

def colors_to_array_of_ints(s):
```

```
    rt=[]
    for c in s:
        if c=='W':
            rt.append(True)
        else:
            rt.append(False)
    return rt

def set_current_state (d):
    F=colors_to_array_of_ints(d["F"])
    U=colors_to_array_of_ints(d["U"])
    D=colors_to_array_of_ints(d["D"])
    R=colors_to_array_of_ints(d["R"])
    L=colors_to_array_of_ints(d["L"])
    B=colors_to_array_of_ints(d["B"])
    return F,U,D,R,L,B # return tuple

init_F, init_U, init_D, init_R, init_L, init_B=set_current_state({"F":"....W..W.", "U
    ":"...W...W.", "D":".......W.", "R":"..W...W..", "L":"......W..", "B":"..W......"})

for STEPS in range(1, 20):
        print "trying %d steps" % STEPS

        s=Solver()
        state=[[[Bool('state%d_%d_%d' % (n, side, i)) for i in range(FACELETS)] for side
            in range(FACES)] for n in range(STEPS+1)]

        op=[Int('op%d' % n) for n in range(STEPS+1)]

        # initial state
        for i in range(FACELETS):
                s.add(state[0][0][i]==init_F[i])
                s.add(state[0][1][i]==init_U[i])
                s.add(state[0][2][i]==init_D[i])
                s.add(state[0][3][i]==init_R[i])
                s.add(state[0][4][i]==init_L[i])
                s.add(state[0][5][i]==init_B[i])

        # "must be" state for one (front/white) face
        for j in range(FACELETS):
                s.add(state[STEPS][0][j]==True)

        for n in range(STEPS):
                for side in range(FACES):
                        for j in range(FACELETS):
                                s.add(state[n+1][side][j]==rotate(op[n], state[n], side,
                                    j))

        if s.check()==sat:
                print "sat"
                m=s.model()
                for n in range(STEPS):
                        print move_names[int(str(m[op[n]]))]
                exit(0)
```

( The full source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/rubik3/one_face_SMT/rubik3_z3.py )

```
trying 1 steps
trying 2 steps
trying 3 steps
trying 4 steps
trying 5 steps
trying 6 steps
trying 7 steps
trying 8 steps
sat
LCW
UCCW
BCW
LCW
RCCW
BCCW
UCW
LCCW
```

Now the fun statistics. Using random walk I collected 928 states and then I tried to solve one (white/front) face for each state.

```
      1 turns= 4
      5 turns= 5
     57 turns= 6
    307 turns= 7
    501 turns= 8
     56 turns= 9
      1 turns= 10
```

It seems that majority of all states can be solved for 7-8 half turns (half-turn is one of 18 turns we used here). But there is at least one state which must be solved with 10 half turns. Maybe 10 is a "god's number" for one face, like 20 for all 6 faces?

## 7.11 Numberlink

### 7.11.1 Numberlink (AKA Flow Free) puzzle (Z3Py)

You probably saw Flow Free puzzle:

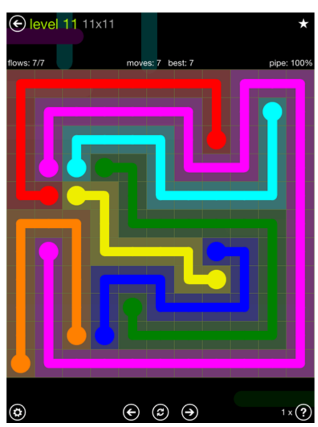



Figure 22:

I'll stick to Numberlink version of the puzzle. This is the example puzzle from Wikipedia:

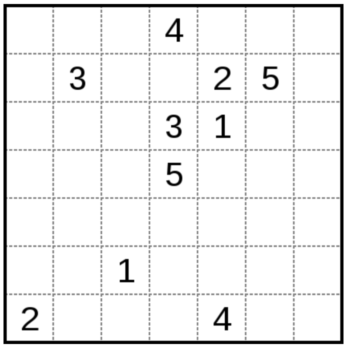



Figure 23:

This is solved:

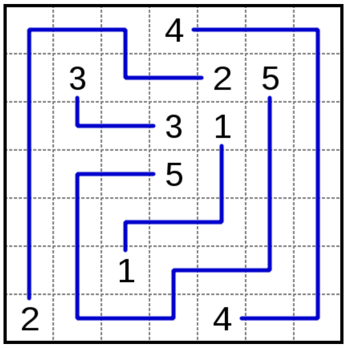



Figure 24:

See also: https://en.wikipedia.org/wiki/Numberlink, https://en.wikipedia.org/wiki/Flow_Free.
The code:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from z3 import *

puzzle=["   4   ",
" 3  25 ",
"   31  ",
"   5   ",
"       ",
"  1    ",
"2   4  "]

width=len(puzzle[0])
height=len(puzzle)

# number for each cell:
cells=[[Int('cell_r%d_c%d' % (r,c)) for c in range(width)] for r in range(height)]

# connections between cells. L means the cell has connection with cell at left, etc:
L=[[Bool('L_r%d_c%d' % (r,c)) for c in range(width)] for r in range(height)]
R=[[Bool('R_r%d_c%d' % (r,c)) for c in range(width)] for r in range(height)]
U=[[Bool('U_r%d_c%d' % (r,c)) for c in range(width)] for r in range(height)]
D=[[Bool('D_r%d_c%d' % (r,c)) for c in range(width)] for r in range(height)]
```

```
s=Solver()

# U for a cell must be equal to D of the cell above, etc:
for r in range(height):
    for c in range(width):
        if r!=0:
            s.add(U[r][c]==D[r-1][c])
        if r!=height-1:
            s.add(D[r][c]==U[r+1][c])
        if c!=0:
            s.add(L[r][c]==R[r][c-1])
        if c!=width-1:
            s.add(R[r][c]==L[r][c+1])

# yes, I know, we have 4 bools for each cell at this point, and we can half this number,
# but anyway, for the sake of simplicity, this could be better.

for r in range(height):
    for c in range(width):
        t=puzzle[r][c]
        if t==' ':
            # puzzle has space, so degree=2, IOW, this cell must have 2 connections, no
                more, no less.
            # enumerate all possible L/R/U/D booleans. two of them must be True, others
                are False.
            t=[]
            t.append(And(L[r][c], R[r][c], Not(U[r][c]), Not(D[r][c])))
            t.append(And(L[r][c], Not(R[r][c]), U[r][c], Not(D[r][c])))
            t.append(And(L[r][c], Not(R[r][c]), Not(U[r][c]), D[r][c]))
            t.append(And(Not(L[r][c]), R[r][c], U[r][c], Not(D[r][c])))
            t.append(And(Not(L[r][c]), R[r][c], Not(U[r][c]), D[r][c]))
            t.append(And(Not(L[r][c]), Not(R[r][c]), U[r][c], D[r][c]))
            s.add(Or(*t))
        else:
            # puzzle has number, add it to cells[][] as a constraint:
            s.add(cells[r][c]==int(t))
            # cell has degree=1, IOW, this cell must have 1 connection, no more, no less
            # enumerate all possible ways:
            t=[]
            t.append(And(L[r][c], Not(R[r][c]), Not(U[r][c]), Not(D[r][c])))
            t.append(And(Not(L[r][c]), R[r][c], Not(U[r][c]), Not(D[r][c])))
            t.append(And(Not(L[r][c]), Not(R[r][c]), U[r][c], Not(D[r][c])))
            t.append(And(Not(L[r][c]), Not(R[r][c]), Not(U[r][c]), D[r][c]))
            s.add(Or(*t))

        # if L[][]==True, cell's number must be equal to the number of cell at left, etc
            :
        if c!=0:
            s.add(If(L[r][c], cells[r][c]==cells[r][c-1], True))
        if c!=width-1:
            s.add(If(R[r][c], cells[r][c]==cells[r][c+1], True))
        if r!=0:
            s.add(If(U[r][c], cells[r][c]==cells[r-1][c], True))
        if r!=height-1:
            s.add(If(D[r][c], cells[r][c]==cells[r+1][c], True))
```

```
# L/R/U/D at borders sometimes must be always False:
for r in range(height):
    s.add(L[r][0]==False)
    s.add(R[r][width-1]==False)

for c in range(width):
    s.add(U[0][c]==False)
    s.add(D[height-1][c]==False)

# print solution:

print s.check()
m=s.model()

print ""

for r in range(height):
    for c in range(width):
        print m[cells[r][c]],
    print ""

print ""

for r in range(height):
    for c in range(width):
        t=""
        t=t+("L" if str(m[L[r][c]])=="True" else " ")
        t=t+("R" if str(m[R[r][c]])=="True" else " ")
        t=t+("U" if str(m[U[r][c]])=="True" else " ")
        t=t+("D" if str(m[D[r][c]])=="True" else " ")
        print t+"|",
    print ""

print ""

for r in range(height):
    row=""
    for c in range(width):
        t=puzzle[r][c]
        if t==' ':
            tl=(True if str(m[L[r][c]])=="True" else False)
            tr=(True if str(m[R[r][c]])=="True" else False)
            tu=(True if str(m[U[r][c]])=="True" else False)
            td=(True if str(m[D[r][c]])=="True" else False)

            if tu and td:
                row=row+" "
            if tr and td:
                row=row+" "
            if tr and tu:
                row=row+" "
            if tl and td:
                row=row+" "
            if tl and tu:
                row=row+" "
            if tl and tr:
```

```
                row=row+" "
        else:
            row=row+t
    print row
```

The solution:

```
sat

2 2 2 4 4 4 4
2 3 2 2 2 5 4
2 3 3 3 1 5 4
2 5 5 5 1 5 4
2 5 1 1 1 5 4
2 5 1 5 5 5 4
2 5 5 5 4 4 4

 R D| LR  | L  D|  R  | LR  | LR  | L  D|
  UD|    D|  RU | LR  | L   |    D|   UD|
  UD|  RU | LR  | L   |    D|   UD|   UD|
  UD|  R D| LR  | L   |   UD|   UD|   UD|
  UD|   UD|  R D| LR  | L U |   UD|   UD|
  UD|   UD|   U |  R D| LR  | L U |   UD|
  U |  RU | LR  | L U |  R  | LR  | L U |
```

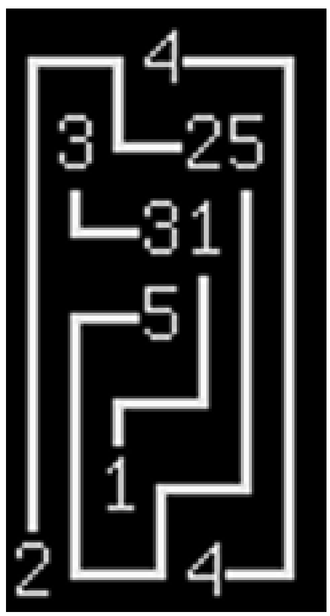



Figure 25: The solution

#### 7.11.2 Numberlink (AKA Flow Free) puzzle as a MaxSAT problem + toy PCB router

Let's revisit my solution for Numberlink (AKA Flow Free) puzzle written for Z3Py.

What if holes in the puzzle exist? Can we make all paths as short as possible?

I've rewritten the puzzle solver using my own SAT library and now I use Open-WBO MaxSAT solver, see the source code, which is almost the same: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/numberlink/MaxSAT/numberlink_WBO.py.

But now we "maximize" number of empty cells:

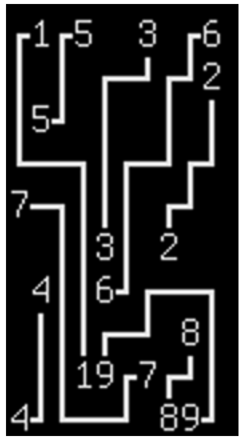



Figure 26:

This is a solution with shortest possible paths. Others are possible, but their sum wouldn't be shorter. This is like toy PCB routing.

What if we comment the `s.fix_soft_always_true(cell_is_empty[r][c], 1)` line and set `maxsat=True`?

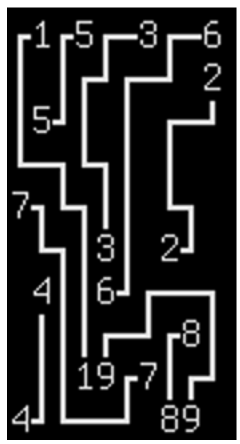



Figure 27:

Lines 2 and 3 "roaming" chaotically, but the solution is correct, under given constraints.

The files: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/puzzles/numberlink/MaxSAT.

### 7.12 1959 AHSME Problems, Problem 6

From http://artofproblemsolving.com/wiki/index.php?title=1959_AHSME_Problems:

> With the use of three different weights, namely 1 lb., 3 lb., and 9 lb., how many objects of different weights can be weighed, if the objects is to be weighed and the given weights may be placed in either pan of the scale? 15, 13, 11, 9, 7

This is fun!

```
from z3 import *

# 0 - weight absent, 1 - on left pan, 2 - on right pan:
w1, w3, w9, obj = Ints('w1 w3 w9 obj')
```

```
obj_w = Int('obj_w')

s=Solver()

s.add(And(w1>=0, w1<=2))
s.add(And(w3>=0, w3<=2))
s.add(And(w9>=0, w9<=2))

# object is always on left or right pan:
s.add(And(obj>=1, obj<=2))

# object must weight something:
s.add(obj_w>0)

left, right = Ints('left right')

# left pan is a sum of weights/object, if they are present on pan:
s.add(left  == If(w1==1, 1, 0) + If(w3==1, 3, 0) + If(w9==1, 9, 0) + If(obj==1, obj_w,
   0))
# same for right pan:
s.add(right == If(w1==2, 1, 0) + If(w3==2, 3, 0) + If(w9==2, 9, 0) + If(obj==2, obj_w,
   0))

# both pans must weight something:
s.add(left>0)
s.add(right>0)

# pans must have equal weights:
s.add(left==right)

# get all results:
results=[]
while True:
    if s.check() == sat:
        m = s.model()
        #print m
        print "left: ",
        print ("w1" if m[w1].as_long()==1 else "  "),
        print ("w3" if m[w3].as_long()==1 else "  "),
        print ("w9" if m[w9].as_long()==1 else "  "),
        print (("obj_w=%2d" % m[obj_w].as_long()) if m[obj].as_long()==1 else "         "
            ),

        print "    | right: ",
        print ("w1" if m[w1].as_long()==2 else "  "),
        print ("w3" if m[w3].as_long()==2 else "  "),
        print ("w9" if m[w9].as_long()==2 else "  "),
        print (("obj_w=%2d" % m[obj_w].as_long()) if m[obj].as_long()==2 else "         "
            ),
        print ""

        results.append(m)
        block = []
        for d in m:
            # skip internal variables, do not add them to blocking constraint:
```

```
            if str(d).startswith ("z3name"):
                continue
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "total results", len(results)
        break
```

The output:

```
left:     w3                 | right:  w1         obj_w= 2
left:     w3    obj_w= 7     | right:  w1    w9
left:     w3 w9              | right:  w1         obj_w=11
left:     w3    obj_w= 6     | right:        w9
left:  w1 w3    obj_w= 5     | right:        w9
left:     w3 w9              | right:           obj_w=12
left:  w1 w3 w9              | right:           obj_w=13
left:  w1 w3                 | right:           obj_w= 4
left:     w3                 | right:           obj_w= 3
left:           obj_w= 4     | right:  w1 w3
left:           obj_w=13     | right:  w1 w3 w9
left:           obj_w= 3     | right:     w3
left:  w1       obj_w= 2     | right:     w3
left:  w1       obj_w=11     | right:     w3 w9
left:           obj_w=12     | right:     w3 w9
left:  w1                    | right:           obj_w= 1
left:        w9              | right:  w1         obj_w= 8
left:        w9              | right:           obj_w= 9
left:  w1    w9              | right:           obj_w=10
left:  w1    w9              | right:     w3    obj_w= 7
left:        w9              | right:  w1 w3    obj_w= 5
left:        w9              | right:     w3    obj_w= 6
left:           obj_w= 9     | right:        w9
left:           obj_w=10     | right:  w1    w9
left:           obj_w= 1     | right:  w1
left:  w1       obj_w= 8     | right:        w9
total results 26
```

There are 13 distinct `obj_w` values. So this is the answer.

## 7.13 Two parks

The problem from the “Discrete Structures, Logic and Computability” book by James L. Hein, 4th ed.

> Suppose a survey revealed that 70% of the population visited an amusement park and 80% visited a national park. At least what percentage of the population visited both?

The problem is supposed to be solved using finite sets counting... But I’m slothful student and would try Z3.

```
from z3 import *

# each element is 0/1, reflecting 10% of park1/park2 levels of attendance...
p1=[Int('park1_%d' % i) for i in range(10)]
p2=[Int('park2_%d' % i) for i in range(10)]
```

```
# 1 if visited both, 0 otherwise:
b=[Int('both_%d' % i) for i in range(10)]

s=Optimize()
# sum of p1[] must be 7 (or 70%)
s.add(Sum(p1)==7)
# sum of p2[] must be 8 (or 80%)
s.add(Sum(p2)==8)

for i in range(10):
    # all in limits:
    s.add(And(p1[i]>=0, p1[i]<=1))
    s.add(And(p2[i]>=0, p2[i]<=1))
    # if both p1[] and p2[] has 1, b[] would have 1 as well:
    s.add(b[i]==If(And(p1[i]==1, p2[i]==1), 1, 0))

both=Int('both')
s.add(Sum(b)==both)

s.minimize(both)
#s.maximize(both)
assert s.check()==sat
m=s.model()

print "park1 : "+"".join([("*" if m[p1[i]].as_long()==1 else ".") for i in range(10)])
print "park2 : "+"".join([("*" if m[p2[i]].as_long()==1 else ".") for i in range(10)])
print "both  : "+"".join([("*" if m[b[i]].as_long()==1 else ".") for i in range(10)])+"
   (%d)" % m[both].as_long()
```

This is fun!

If minimize:

```
park1 : ***...****
park2 : *..*******
both  : *.....**** (5)
```

If maximize:

```
park1 : ..*******.
park2 : ..********
both  : ..*******. (7)
```

In other words, “stars” are allocated in such a way, so that the sum of “stars” in `b[]` would be minimal/maximal.

Observing this, we can deduce the general formula:

Maximal both = min(park1, park2)

What about minimal both? We can see that “stars” from one park1 must “shift out” or “hide in” to what corresponding empty space of park2. So, minimal both = park2 - (100% - park1)

SMT solver is overkill for the job, but perfect for illustration and helping in better understanding.

#### 7.13.1 Variations of the problem from the same book

> Suppose that 100 senators voted on three separate senate bills as follows: 70 percent of the senators voted for the first bill, 65 percent voted for the second bill, and 60 percent voted for the third bill. At least what percentage of the senators voted for all three bills?

Suppose that 25 people attended a conference with three sessions, where 15 people attended the first session, 18 the second session, and 12 the third session. At least how many people attended all three sessions?

### 7.14 Alphametics

According to Donald Knuth, the term “Alphametics” was coined by J. A. H. Hunter. This is a puzzle: what decimal digits in 0..9 range must be assigned to each letter, so the following equation will be true?

```
  SEND
+ MORE
 -----
 MONEY
```

So is easy for Z3:

```
from z3 import *

# SEND+MORE=MONEY

D, E, M, N, O, R, S, Y = Ints('D, E, M, N, O, R, S, Y')

s=Solver()

s.add(Distinct(D, E, M, N, O, R, S, Y))
s.add(And(D>=0, D<=9))
s.add(And(E>=0, E<=9))
s.add(And(M>=0, M<=9))
s.add(And(N>=0, N<=9))
s.add(And(O>=0, O<=9))
s.add(And(R>=0, R<=9))
s.add(And(S>=0, S<=9))
s.add(And(Y>=0, Y<=9))

s.add(1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E == 10000*M+1000*O+100*N+10*E+Y)

print s.check()
print s.model()
```

Output:

```
sat
[E, = 5,
 S, = 9,
 M, = 1,
 N, = 6,
 D, = 7,
 R, = 8,
 O, = 0,
 Y = 2]
```

Another one, also from TAOCP[77] volume IV (http://www-cs-faculty.stanford.edu/~uno/fasc2b.ps.gz):

```
from z3 import *
```

[77] The Art Of Computer Programming

```
# VIOLIN+VIOLIN+VIOLA = TRIO+SONATA

A, I, L, N, O, R, S, T, V = Ints('A, I, L, N, O, R, S, T, V')

s=Solver()

s.add(Distinct(A, I, L, N, O, R, S, T, V))
s.add(And(A>=0, A<=9))
s.add(And(I>=0, I<=9))
s.add(And(L>=0, L<=9))
s.add(And(N>=0, N<=9))
s.add(And(O>=0, O<=9))
s.add(And(R>=0, R<=9))
s.add(And(S>=0, S<=9))
s.add(And(T>=0, T<=9))
s.add(And(V>=0, V<=9))

VIOLIN, VIOLA, SONATA, TRIO = Ints('VIOLIN, VIOLA, SONATA, TRIO')

s.add(VIOLIN==100000*V+10000*I+1000*O+100*L+10*I+N)
s.add(VIOLA==10000*V+1000*I+100*O+10*L+A)
s.add(SONATA==100000*S+10000*O+1000*N+100*A+10*T+A)
s.add(TRIO==1000*T+100*R+10*I+O)

s.add(VIOLIN+VIOLIN+VIOLA == TRIO+SONATA)

print s.check()
print s.model()
```

```
sat
[L, = 6,
 S, = 7,
 N, = 2,
 T, = 1,
 I, = 5,
 V = 3,
 A, = 8,
 R, = 9,
 O, = 4,
 TRIO = 1954,
 SONATA, = 742818,
 VIOLA, = 35468,
 VIOLIN, = 354652]
```

This puzzle I've found in pySMT examples[78]:

```
from z3 import *

# H+E+L+L+O = W+O+R+L+D = 25

H, E, L, O, W, R, D = Ints ('H E L O W R D')

s=Solver()

s.add(Distinct(H, E, L, O, W, R, D))
```

[78] http://pysmt.readthedocs.io/en/latest/getting_started.html

```
s.add(And(H>=1, H<=9))
s.add(And(E>=1, E<=9))
s.add(And(L>=1, L<=9))
s.add(And(O>=1, O<=9))
s.add(And(W>=1, W<=9))
s.add(And(R>=1, R<=9))
s.add(And(D>=1, D<=9))

s.add(H+E+L+L+O == 25)
s.add(W+O+R+L+D == 25)

print s.check()
print s.model()
```

```
sat
[D = 5, R = 4, O = 3, E = 8, L = 6, W = 7, H = 2]
```

This is an exercise Q209 from the [Companion to the Papers of Donald Knuth][79].

```
 KNIFE
  FORK
 SPOON
  SOUP
------
SUPPER
```

I've added a helper function (list_to_expr()) to make things simpler:

```
from z3 import *

# KNIFE+FORK+SPOON+SOUP = SUPPER

E, F, I, K, N, O, P, R, S, U = Ints('E F I K N O P R S U')

s=Solver()

s.add(Distinct(E, F, I, K, N, O, P, R, S, U))
s.add(And(E>=0, E<=9))
s.add(And(F>=0, F<=9))
s.add(And(I>=0, I<=9))
s.add(And(K>=0, K<=9))
s.add(And(N>=0, N<=9))
s.add(And(O>=0, O<=9))
s.add(And(P>=0, P<=9))
s.add(And(R>=0, R<=9))
s.add(And(S>=0, S<=9))
s.add(And(U>=0, U<=9))

#s.add(S!=0)

KNIFE, FORK, SPOON, SOUP, SUPPER = Ints('KNIFE FORK SPOON SOUP SUPPER')

# construct expression in form like:
# 10000000*L+1000000*U+100000*N+10000*C+1000*H+100*E+10*O+N
def list_to_expr(lst):
    coeff=1
```

[79] http://www-cs-faculty.stanford.edu/~knuth/cp.html

```
    _sum=0
    for var in lst[::-1]:
        _sum=_sum+var*coeff
        coeff=coeff*10
    return _sum

s.add(KNIFE==list_to_expr([K,N,I,F,E]))
s.add(FORK==list_to_expr([F,O,R,K]))
s.add(SPOON==list_to_expr([S,P,O,O,N]))
s.add(SOUP==list_to_expr([S,O,U,P]))
s.add(SUPPER==list_to_expr([S,U,P,P,E,R]))

s.add(KNIFE+FORK+SPOON+SOUP == SUPPER)

print s.check()
print s.model()
```

```
sat
[K = 7,
 N = 4,
 R = 9,
 I = 1,
 E = 6,
 S = 0,
 O = 3,
 F = 5,
 U = 8,
 P = 2,
 SUPPER = 82269,
 SOUP = 382,
 SPOON = 2334,
 FORK = 5397,
 KNIFE = 74156]
```

S is zero, so SUPPER value is starting with leading (removed) zero. Let's say, we don't like it. Add this to resolve it:

```
s.add(S!=0)
```

```
sat
[K = 8,
 N = 4,
 R = 3,
 I = 7,
 E = 6,
 S = 1,
 O = 9,
 F = 2,
 U = 0,
 P = 5,
 SUPPER = 105563,
 SOUP = 1905,
 SPOON = 15994,
 FORK = 2938,
 KNIFE = 84726]
```

**Devising your own puzzle** Here is a problem: you can only use 10 letters, but how to select them among words? We can try to offload this task to Z3:

```
from z3 import *

def char_to_idx(c):
    return ord(c)-ord('A')

def idx_to_char(i):
    return chr(ord('A')+i)

# construct expression in form like:
# 10000000*L+1000000*U+100000*N+10000*C+1000*H+100*E+10*O+N
def list_to_expr(lst):
    coeff=1
    _sum=0
    for var in lst[::-1]:
        _sum=_sum+var*coeff
        coeff=coeff*10
    return _sum

# this table has 10 items, it reflects character for each number:
digits=[Int('digit_%d' % i) for i in range(10)]

# this is "reverse" table, it has value for each letter:
letters=[Int('letter_%d' % i) for i in range(26)]

s=Solver()

# all items in digits[] table must be distinct, because no two letters can share same
    number:
s.add(Distinct(digits))

# all numbers are in 0..25 range, because each number in this table defines character:
for i in range(10):
    s.add(And(digits[i]>=0,digits[i]<26))

# define "reverse" table.
# letters[i] is 0..9, depending on which digits[] item contains this letter:
for i in range(26):
    s.add(letters[i] ==
        If(digits[0]==i,0,
        If(digits[1]==i,1,
        If(digits[2]==i,2,
        If(digits[3]==i,3,
        If(digits[4]==i,4,
        If(digits[5]==i,5,
        If(digits[6]==i,6,
        If(digits[7]==i,7,
        If(digits[8]==i,8,
        If(digits[9]==i,9,99999999)))))))))))

# the last word is "sum" all the rest are "addends":

words=['APPLE', 'BANANA', 'BREAD', 'CAKE', 'CAVIAR', 'CHEESE', 'CHIPS', 'COFFEE', 'EGGS'
    , 'FISH', 'HONEY', 'JAM', 'JELLY', 'JUICE', 'MILK', 'OATMEAL', 'ORANGE', 'PANCAKE',
    'PIZZA', 'STEAK', 'TEA', 'TOMATO', 'WAFFLE', 'LUNCH']
```

```
words_total=len(words)

word=[Int('word_%d' % i) for i in range(words_total)]
word_used=[Bool('word_used_%d' % i) for i in range(words_total)]

# last word is always used:
s.add(word_used[words_total-1]==True)

#s.add(word_used[words.index('CAKE')])
s.add(word_used[words.index('EGGS')])

for i in range(words_total):
    # get list of letters for the word:
    lst=[letters[char_to_idx(c)] for c in words[i]]
    # construct expression for letters. it must be equal to the value of the word:
    s.add(word[i]==list_to_expr(lst))
    # if word_used, word's value must be less than 99999999, i.e., all letters are used
       in the word:
    s.add(If(word_used[i], word[i], 0) < 99999999)

# if word_used, add value of word to the whole expression
expr=[If(word_used[i], word[i], 0) for i in range(words_total-1)]
# sum up all items in expression. sum must be equal to the value of the last word:
s.add(sum(expr)==word[-1])

print s.check()
m=s.model()
#print m

for i in range(words_total):
    # if word_used, print it:
    if str(m[word_used[i]])=="True" or i+1==words_total:
        print words[i]

for i in range(26):
    # it letter is used, print it:
    if m[letters[i]].as_long()!=99999999:
        print idx_to_char(i), m[letters[i]]
```

This is the first generated puzzle:

```
sat
EGGS
JELLY
LUNCH
C 5
E 6
G 3
H 7
J 0
L 1
N 4
S 8
U 2
Y 9
```

What if we want “CAKE” to be present among “addends”?
Add this:

```
s.add(word_used[words.index('CAKE')])
```

```
sat
CAKE
TEA
LUNCH
A 8
C 3
E 1
H 9
J 6
K 2
L 0
N 5
T 7
U 4
```

Add this:

```
s.add(word_used[words.index('EGGS')])
```

Now it can find pair to EGGS:

```
sat
EGGS
HONEY
LUNCH
C 6
E 7
G 9
H 4
L 5
N 8
O 2
S 3
U 0
Y 1
```

**The files** https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/puzzles/alphametics.

## 7.15 2015 AIME II Problems/Problem 12

At http://artofproblemsolving.com/wiki/index.php?title=2015_AIME_II_Problems/Problem_12:

> There are $2^{10} = 1024$ possible 10-letter strings in which each letter is either an A or a B. Find the number of such strings that do not have more than 3 adjacent letters that are identical.

We just find all 10-bit numbers, which don’t have 4-bit runs of zeros or ones:

```
from z3 import *

a = BitVec('a', 10)
```

```
s=Solver()

for i in range(10-4+1):
    s.add(((a>>i)&15)!=0)
    s.add(((a>>i)&15)!=15)

results=[]
while True:
    if s.check() == sat:
        m = s.model()
        print "0x%x" % m[a].as_long()

        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "total results", len(results)
        break
```

It's 548.

## 7.16 Fred puzzle

Found this:

```
Three fellows accused of stealing CDs make the following statements:

(1) Ed: "Fred did it, and Ted is innocent".
(2) Fred: "If Ed is guilty, then so is Ted".
(3) Ted: "'Im innocent, but at least one of the others is guilty".

If the innocent told the truth and the guilty lied, who is guilty? (Remember that false
    statements imply anything).

I think Ed and Ted are innocent and Fred is guilty. Is it in contradiction with
    statement 2.

What do you say?
```

( https://math.stackexchange.com/questions/15199/implication-of-three-statements )

And how to convert this into logic statements:

```
Let us write the following propositions:

Fg means Fred is guilty, and Fi means Fred is innocent, Tg and Ti for Ted and Eg and Ei
    for Ed.

1. Ed says: Fg ∧ Ti
2. Fred says: Eg → Tg
3. Ted says: Ti ∧ (Fg ∨ Eg)

We know that the guilty is lying and the innocent tells the truth.

...
```

This is how I can implement it using Z3Py:

```
#!/usr/bin/env python3

from z3 import *

fg, fi, tg, ti, eg, ei = Bools('fg fi tg ti eg ei')

s=Solver()

s.add(fg!=fi)
s.add(tg!=ti)
s.add(eg!=ei)

s.add(ei==And(fg, ti))

s.add(fi==Implies(eg, tg))
#s.add(fi==Or(Not(eg), tg)) # Or(-x, y) is the same as Implies

s.add(ti==And(ti, Or(fg, eg)))

print (s.check())
print (s.model())
```

The result:

```
sat
[fg = False,
 ti = False,
 tg = True,
 eg = True,
 ei = False,
 fi = True]
```

(Fred is innocent, others are guilty.)

(`Implies` can be replaced with `Or(Not(x), y)`.)

Now in SMT-LIB v2 form:

```
; tested with Z3 and MK85:

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun fg () Bool)
(declare-fun fi () Bool)
(declare-fun tg () Bool)
(declare-fun ti () Bool)
(declare-fun eg () Bool)
(declare-fun ei () Bool)

(assert (not (= fg fi)))
(assert (not (= tg ti)))
(assert (not (= eg ei)))

(assert (= ei (and fg ti)))

; Or(-x, y) is the same as Implies
(assert (= fi (or (not eg) tg)))
```

```
(assert (= ti (and ti (or fg eg))))

(check-sat)
(get-model)
```

Again, it's small enought to be solved by MK85:

```
$ MK85 --dump-internal-variables fred.smt2
sat
(model
        (define-fun always_false () Bool false) ; var_no=1
        (define-fun always_true () Bool true) ; var_no=2
        (define-fun fg () Bool false) ; var_no=3
        (define-fun fi () Bool true) ; var_no=4
        (define-fun tg () Bool true) ; var_no=5
        (define-fun ti () Bool false) ; var_no=6
        (define-fun eg () Bool true) ; var_no=7
        (define-fun ei () Bool false) ; var_no=8
        (define-fun internal!1 () Bool true) ; var_no=9
        (define-fun internal!2 () Bool false) ; var_no=10
        (define-fun internal!3 () Bool true) ; var_no=11
        (define-fun internal!4 () Bool true) ; var_no=12
        (define-fun internal!5 () Bool false) ; var_no=13
        (define-fun internal!6 () Bool true) ; var_no=14
        (define-fun internal!7 () Bool true) ; var_no=15
        (define-fun internal!8 () Bool false) ; var_no=16
        (define-fun internal!9 () Bool true) ; var_no=17
        (define-fun internal!10 () Bool false) ; var_no=18
        (define-fun internal!11 () Bool false) ; var_no=19
        (define-fun internal!12 () Bool true) ; var_no=20
        (define-fun internal!13 () Bool false) ; var_no=21
        (define-fun internal!14 () Bool true) ; var_no=22
        (define-fun internal!15 () Bool false) ; var_no=23
        (define-fun internal!16 () Bool true) ; var_no=24
        (define-fun internal!17 () Bool true) ; var_no=25
        (define-fun internal!18 () Bool false) ; var_no=26
        (define-fun internal!19 () Bool false) ; var_no=27
        (define-fun internal!20 () Bool true) ; var_no=28
)
```

What is in the CNF file generated by MK85?

```
p cnf 28 64
-1 0
2 0
c generate_EQ id1=fg, id2=fi, var1=3, var2=4
c generate_XOR id1=fg id2=fi var1=3 var2=4 out id=internal!1 out var=9
-3 -4 -9 0
3 4 -9 0
3 -4 9 0
-3 4 9 0
c generate_NOT id=internal!1 var=9, out id=internal!2 out var=10
-10 -9 0
10 9 0
c generate_NOT id=internal!2 var=10, out id=internal!3 out var=11
-11 -10 0
11 10 0
```

```
c create_assert() id=internal!3 var=11
11 0
c generate_EQ id1=tg, id2=ti, var1=5, var2=6
c generate_XOR id1=tg id2=ti var1=5 var2=6 out id=internal!4 out var=12
-5 -6 -12 0
5 6 -12 0
5 -6 12 0
-5 6 12 0
c generate_NOT id=internal!4 var=12, out id=internal!5 out var=13
-13 -12 0
13 12 0
c generate_NOT id=internal!5 var=13, out id=internal!6 out var=14
-14 -13 0
14 13 0
c create_assert() id=internal!6 var=14
14 0
c generate_EQ id1=eg, id2=ei, var1=7, var2=8
c generate_XOR id1=eg id2=ei var1=7 var2=8 out id=internal!7 out var=15
-7 -8 -15 0
7 8 -15 0
7 -8 15 0
-7 8 15 0
c generate_NOT id=internal!7 var=15, out id=internal!8 out var=16
-16 -15 0
16 15 0
c generate_NOT id=internal!8 var=16, out id=internal!9 out var=17
-17 -16 0
17 16 0
c create_assert() id=internal!9 var=17
17 0
c generate_AND id1=fg id2=ti var1=3 var2=6 out id=internal!10 out var=18
-3 -6 18 0
3 -18 0
6 -18 0
c generate_EQ id1=ei, id2=internal!10, var1=8, var2=18
c generate_XOR id1=ei id2=internal!10 var1=8 var2=18 out id=internal!11 out var=19
-8 -18 -19 0
8 18 -19 0
8 -18 19 0
-8 18 19 0
c generate_NOT id=internal!11 var=19, out id=internal!12 out var=20
-20 -19 0
20 19 0
c create_assert() id=internal!12 var=20
20 0
c generate_NOT id=eg var=7, out id=internal!13 out var=21
-21 -7 0
21 7 0
c generate_OR id1=internal!13 id2=tg var1=21 var2=5 out id=internal!14 out var=22
21 5 -22 0
-21 22 0
-5 22 0
c generate_EQ id1=fi, id2=internal!14, var1=4, var2=22
c generate_XOR id1=fi id2=internal!14 var1=4 var2=22 out id=internal!15 out var=23
-4 -22 -23 0
4 22 -23 0
```

```
4 -22 23 0
-4 22 23 0
c generate_NOT id=internal!15 var=23, out id=internal!16 out var=24
-24 -23 0
24 23 0
c create_assert() id=internal!16 var=24
24 0
c generate_OR id1=fg id2=eg var1=3 var2=7 out id=internal!17 out var=25
3 7 -25 0
-3 25 0
-7 25 0
c generate_AND id1=ti id2=internal!17 var1=6 var2=25 out id=internal!18 out var=26
-6 -25 26 0
6 -26 0
25 -26 0
c generate_EQ id1=ti, id2=internal!18, var1=6, var2=26
c generate_XOR id1=ti id2=internal!18 var1=6 var2=26 out id=internal!19 out var=27
-6 -26 -27 0
6 26 -27 0
6 -26 27 0
-6 26 27 0
c generate_NOT id=internal!19 var=27, out id=internal!20 out var=28
-28 -27 0
28 27 0
c create_assert() id=internal!20 var=28
28 0
```

Let's filter out comments:

```
c generate_EQ id1=fg, id2=fi, var1=3, var2=4
c generate_XOR id1=fg id2=fi var1=3 var2=4 out id=internal!1 out var=9
c generate_NOT id=internal!1 var=9, out id=internal!2 out var=10
c generate_NOT id=internal!2 var=10, out id=internal!3 out var=11
c create_assert() id=internal!3 var=11
c generate_EQ id1=tg, id2=ti, var1=5, var2=6
c generate_XOR id1=tg id2=ti var1=5 var2=6 out id=internal!4 out var=12
c generate_NOT id=internal!4 var=12, out id=internal!5 out var=13
c generate_NOT id=internal!5 var=13, out id=internal!6 out var=14
c create_assert() id=internal!6 var=14
c generate_EQ id1=eg, id2=ei, var1=7, var2=8
c generate_XOR id1=eg id2=ei var1=7 var2=8 out id=internal!7 out var=15
c generate_NOT id=internal!7 var=15, out id=internal!8 out var=16
c generate_NOT id=internal!8 var=16, out id=internal!9 out var=17
c create_assert() id=internal!9 var=17
c generate_AND id1=fg id2=ti var1=3 var2=6 out id=internal!10 out var=18
c generate_EQ id1=ei, id2=internal!10, var1=8, var2=18
c generate_XOR id1=ei id2=internal!10 var1=8 var2=18 out id=internal!11 out var=19
c generate_NOT id=internal!11 var=19, out id=internal!12 out var=20
c create_assert() id=internal!12 var=20
c generate_NOT id=eg var=7, out id=internal!13 out var=21
c generate_OR id1=internal!13 id2=tg var1=21 var2=5 out id=internal!14 out var=22
c generate_EQ id1=fi, id2=internal!14, var1=4, var2=22
c generate_XOR id1=fi id2=internal!14 var1=4 var2=22 out id=internal!15 out var=23
c generate_NOT id=internal!15 var=23, out id=internal!16 out var=24
c create_assert() id=internal!16 var=24
c generate_OR id1=fg id2=eg var1=3 var2=7 out id=internal!17 out var=25
c generate_AND id1=ti id2=internal!17 var1=6 var2=25 out id=internal!18 out var=26
```

```
c generate_EQ id1=ti, id2=internal!18, var1=6, var2=26
c generate_XOR id1=ti id2=internal!18 var1=6 var2=26 out id=internal!19 out var=27
c generate_NOT id=internal!19 var=27, out id=internal!20 out var=28
c create_assert() id=internal!20 var=28
```

Again, this instance is small enough to be solved by small backtracking SAT-solver:

```
$ python SAT_backtrack.py tmp.cnf
SAT
-1 2 -3 4 5 -6 7 -8 9 -10 11 12 -13 14 15 -16 17 -18 -19 20 -21 22 -23 24 25 -26 -27 28
   0
UNSAT
solutions= 1
```

## 7.17 Multiple choice logic puzzle

The source of this puzzle is probably Ross Honsberger's "More Mathematical Morsels (Dolciani Mathematical Expositions)" book:

```
A certain question has the following possible answers.

    a. All of the below
    b. None of the below
    c. All of the above
    d. One of the above
    e. None of the above
    f. None of the above

Which answer is correct?
```

Cited from: https://www.johndcook.com/blog/2015/07/06/multiple-choice/.

This problem can be easily represented as a system of boolean equations. Let's try to solve it using Z3. Each bool represents if the specific sentence is true.

```
#!/usr/bin/env python

from z3 import *

a, b, c, d, e, f = Bools('a b c d e f')

s=Solver()

s.add(a==And(b,c,d,e,f))
s.add(b==And(Not(c),Not(d),Not(e),Not(f)))
s.add(c==And(a,b))
s.add(d==Or(And(a,Not(b),Not(c)), And(Not(a),b,Not(c)), And(Not(a),Not(b),c)))
s.add(e==And(Not(a),Not(b),Not(c),Not(d)))
s.add(f==And(Not(a),Not(b),Not(c),Not(d), Not(e)))

print s.check()
print s.model()
```

The answer:

```
sat
[f = False,
 b = False,
 a = False,
```

```
 c = False,
 d = False,
 e = True]
```

I can also rewrite this in SMT-LIB v2 form:

```
; tested with Z3 and MK85

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun a () Bool)
(declare-fun b () Bool)
(declare-fun c () Bool)
(declare-fun d () Bool)
(declare-fun e () Bool)
(declare-fun f () Bool)

(assert (= a (and b c d e f)))
(assert (= b (and (not c) (not d) (not e) (not f))))
(assert (= c (and a b)))
(assert (= d (or
        (and a (not b) (not c))
        (and (not a) b (not c))
        (and (not a) (not b) c)
        )))
(assert (= e (and (not a) (not b) (not c) (not d))))
(assert (= f (and (not a) (not b) (not c) (not d) (not e))))

(check-sat)
(get-model)
```

The problem is easy enough to be solved using MK85:

```
sat
(model
        (define-fun a () Bool false)
        (define-fun b () Bool false)
        (define-fun c () Bool false)
        (define-fun d () Bool false)
        (define-fun e () Bool true)
        (define-fun f () Bool false)
)
```

Now let's have fun and see how my toy SMT solver tackles this example. What internal variables it creates?

```
$ ./MK85 --dump-internal-variables mc.smt2

sat
(model
        (define-fun always_false () Bool false) ; var_no=1
        (define-fun always_true () Bool true) ; var_no=2
        (define-fun a () Bool false) ; var_no=3
        (define-fun b () Bool false) ; var_no=4
        (define-fun c () Bool false) ; var_no=5
        (define-fun d () Bool false) ; var_no=6
        (define-fun e () Bool true) ; var_no=7
        (define-fun f () Bool false) ; var_no=8
        (define-fun internal!1 () Bool false) ; var_no=9
```

```
        (define-fun internal!2 () Bool false) ; var_no=10
        (define-fun internal!3 () Bool false) ; var_no=11
        (define-fun internal!4 () Bool false) ; var_no=12
        (define-fun internal!5 () Bool false) ; var_no=13
        (define-fun internal!6 () Bool true) ; var_no=14
        (define-fun internal!7 () Bool true) ; var_no=15
        (define-fun internal!8 () Bool true) ; var_no=16
        (define-fun internal!9 () Bool true) ; var_no=17
        (define-fun internal!10 () Bool false) ; var_no=18
        (define-fun internal!11 () Bool false) ; var_no=19
        (define-fun internal!12 () Bool true) ; var_no=20
        (define-fun internal!13 () Bool false) ; var_no=21
        (define-fun internal!14 () Bool false) ; var_no=22
        (define-fun internal!15 () Bool true) ; var_no=23
        (define-fun internal!16 () Bool false) ; var_no=24
        (define-fun internal!17 () Bool false) ; var_no=25
        (define-fun internal!18 () Bool true) ; var_no=26
        (define-fun internal!19 () Bool true) ; var_no=27
        (define-fun internal!20 () Bool false) ; var_no=28
        (define-fun internal!21 () Bool true) ; var_no=29
        (define-fun internal!22 () Bool false) ; var_no=30
        (define-fun internal!23 () Bool true) ; var_no=31
        (define-fun internal!24 () Bool false) ; var_no=32
        (define-fun internal!25 () Bool true) ; var_no=33
        (define-fun internal!26 () Bool false) ; var_no=34
        (define-fun internal!27 () Bool false) ; var_no=35
        (define-fun internal!28 () Bool true) ; var_no=36
        (define-fun internal!29 () Bool true) ; var_no=37
        (define-fun internal!30 () Bool true) ; var_no=38
        (define-fun internal!31 () Bool false) ; var_no=39
        (define-fun internal!32 () Bool false) ; var_no=40
        (define-fun internal!33 () Bool false) ; var_no=41
        (define-fun internal!34 () Bool true) ; var_no=42
        (define-fun internal!35 () Bool true) ; var_no=43
        (define-fun internal!36 () Bool true) ; var_no=44
        (define-fun internal!37 () Bool true) ; var_no=45
        (define-fun internal!38 () Bool true) ; var_no=46
        (define-fun internal!39 () Bool true) ; var_no=47
        (define-fun internal!40 () Bool true) ; var_no=48
        (define-fun internal!41 () Bool true) ; var_no=49
        (define-fun internal!42 () Bool false) ; var_no=50
        (define-fun internal!43 () Bool true) ; var_no=51
        (define-fun internal!44 () Bool true) ; var_no=52
        (define-fun internal!45 () Bool true) ; var_no=53
        (define-fun internal!46 () Bool true) ; var_no=54
        (define-fun internal!47 () Bool true) ; var_no=55
        (define-fun internal!48 () Bool true) ; var_no=56
        (define-fun internal!49 () Bool true) ; var_no=57
        (define-fun internal!50 () Bool true) ; var_no=58
        (define-fun internal!51 () Bool false) ; var_no=59
        (define-fun internal!52 () Bool false) ; var_no=60
        (define-fun internal!53 () Bool false) ; var_no=61
        (define-fun internal!54 () Bool true) ; var_no=62
)
```

What is in resulting CNF file to be fed into the external picosat SAT-solver?

```
p cnf 62 151
-1 0
2 0
c generate_AND id1=b id2=c var1=4 var2=5 out id=internal!1 out var=9
-4 -5 9 0
4 -9 0
5 -9 0
c generate_AND id1=internal!1 id2=d var1=9 var2=6 out id=internal!2 out var=10
-9 -6 10 0
9 -10 0
6 -10 0
c generate_AND id1=internal!2 id2=e var1=10 var2=7 out id=internal!3 out var=11
-10 -7 11 0
10 -11 0
7 -11 0
c generate_AND id1=internal!3 id2=f var1=11 var2=8 out id=internal!4 out var=12
-11 -8 12 0
11 -12 0
8 -12 0
c generate_EQ id1=a, id2=internal!4, var1=3, var2=12
c generate_XOR id1=a id2=internal!4 var1=3 var2=12 out id=internal!5 out var=13
-3 -12 -13 0
3 12 -13 0
3 -12 13 0
-3 12 13 0
c generate_NOT id=internal!5 var=13, out id=internal!6 out var=14
-14 -13 0
14 13 0
c create_assert() id=internal!6 var=14
14 0
c generate_NOT id=c var=5, out id=internal!7 out var=15
-15 -5 0
15 5 0
c generate_NOT id=d var=6, out id=internal!8 out var=16
-16 -6 0
16 6 0
c generate_AND id1=internal!7 id2=internal!8 var1=15 var2=16 out id=internal!9 out var
   =17
-15 -16 17 0
15 -17 0
16 -17 0
c generate_NOT id=e var=7, out id=internal!10 out var=18
-18 -7 0
18 7 0
c generate_AND id1=internal!9 id2=internal!10 var1=17 var2=18 out id=internal!11 out var
   =19
-17 -18 19 0
17 -19 0
18 -19 0
c generate_NOT id=f var=8, out id=internal!12 out var=20
-20 -8 0
20 8 0
c generate_AND id1=internal!11 id2=internal!12 var1=19 var2=20 out id=internal!13 out
   var=21
-19 -20 21 0
19 -21 0
```

```
20 -21 0
c generate_EQ id1=b, id2=internal!13, var1=4, var2=21
c generate_XOR id1=b id2=internal!13 var1=4 var2=21 out id=internal!14 out var=22
-4 -21 -22 0
4 21 -22 0
4 -21 22 0
-4 21 22 0
c generate_NOT id=internal!14 var=22, out id=internal!15 out var=23
-23 -22 0
23 22 0
c create_assert() id=internal!15 var=23
23 0
c generate_AND id1=a id2=b var1=3 var2=4 out id=internal!16 out var=24
-3 -4 24 0
3 -24 0
4 -24 0
c generate_EQ id1=c, id2=internal!16, var1=5, var2=24
c generate_XOR id1=c id2=internal!16 var1=5 var2=24 out id=internal!17 out var=25
-5 -24 -25 0
5 24 -25 0
5 -24 25 0
-5 24 25 0
c generate_NOT id=internal!17 var=25, out id=internal!18 out var=26
-26 -25 0
26 25 0
c create_assert() id=internal!18 var=26
26 0
c generate_NOT id=b var=4, out id=internal!19 out var=27
-27 -4 0
27 4 0
c generate_AND id1=a id2=internal!19 var1=3 var2=27 out id=internal!20 out var=28
-3 -27 28 0
3 -28 0
27 -28 0
c generate_NOT id=c var=5, out id=internal!21 out var=29
-29 -5 0
29 5 0
c generate_AND id1=internal!20 id2=internal!21 var1=28 var2=29 out id=internal!22 out
   var=30
-28 -29 30 0
28 -30 0
29 -30 0
c generate_NOT id=a var=3, out id=internal!23 out var=31
-31 -3 0
31 3 0
c generate_AND id1=internal!23 id2=b var1=31 var2=4 out id=internal!24 out var=32
-31 -4 32 0
31 -32 0
4 -32 0
c generate_NOT id=c var=5, out id=internal!25 out var=33
-33 -5 0
33 5 0
c generate_AND id1=internal!24 id2=internal!25 var1=32 var2=33 out id=internal!26 out
   var=34
-32 -33 34 0
32 -34 0
```

```
33 -34 0
c generate_OR id1=internal!22 id2=internal!26 var1=30 var2=34 out id=internal!27 out var
   =35
30 34 -35 0
-30 35 0
-34 35 0
c generate_NOT id=a var=3, out id=internal!28 out var=36
-36 -3 0
36 3 0
c generate_NOT id=b var=4, out id=internal!29 out var=37
-37 -4 0
37 4 0
c generate_AND id1=internal!28 id2=internal!29 var1=36 var2=37 out id=internal!30 out
   var=38
-36 -37 38 0
36 -38 0
37 -38 0
c generate_AND id1=internal!30 id2=c var1=38 var2=5 out id=internal!31 out var=39
-38 -5 39 0
38 -39 0
5 -39 0
c generate_OR id1=internal!27 id2=internal!31 var1=35 var2=39 out id=internal!32 out var
   =40
35 39 -40 0
-35 40 0
-39 40 0
c generate_EQ id1=d, id2=internal!32, var1=6, var2=40
c generate_XOR id1=d id2=internal!32 var1=6 var2=40 out id=internal!33 out var=41
-6 -40 -41 0
6 40 -41 0
6 -40 41 0
-6 40 41 0
c generate_NOT id=internal!33 var=41, out id=internal!34 out var=42
-42 -41 0
42 41 0
c create_assert() id=internal!34 var=42
42 0
c generate_NOT id=a var=3, out id=internal!35 out var=43
-43 -3 0
43 3 0
c generate_NOT id=b var=4, out id=internal!36 out var=44
-44 -4 0
44 4 0
c generate_AND id1=internal!35 id2=internal!36 var1=43 var2=44 out id=internal!37 out
   var=45
-43 -44 45 0
43 -45 0
44 -45 0
c generate_NOT id=c var=5, out id=internal!38 out var=46
-46 -5 0
46 5 0
c generate_AND id1=internal!37 id2=internal!38 var1=45 var2=46 out id=internal!39 out
   var=47
-45 -46 47 0
45 -47 0
46 -47 0
```

```
c generate_NOT id=d var=6, out id=internal!40 out var=48
-48 -6 0
48 6 0
c generate_AND id1=internal!39 id2=internal!40 var1=47 var2=48 out id=internal!41 out
    var=49
-47 -48 49 0
47 -49 0
48 -49 0
c generate_EQ id1=e, id2=internal!41, var1=7, var2=49
c generate_XOR id1=e id2=internal!41 var1=7 var2=49 out id=internal!42 out var=50
-7 -49 -50 0
7 49 -50 0
7 -49 50 0
-7 49 50 0
c generate_NOT id=internal!42 var=50, out id=internal!43 out var=51
-51 -50 0
51 50 0
c create_assert() id=internal!43 var=51
51 0
c generate_NOT id=a var=3, out id=internal!44 out var=52
-52 -3 0
52 3 0
c generate_NOT id=b var=4, out id=internal!45 out var=53
-53 -4 0
53 4 0
c generate_AND id1=internal!44 id2=internal!45 var1=52 var2=53 out id=internal!46 out
    var=54
-52 -53 54 0
52 -54 0
53 -54 0
c generate_NOT id=c var=5, out id=internal!47 out var=55
-55 -5 0
55 5 0
c generate_AND id1=internal!46 id2=internal!47 var1=54 var2=55 out id=internal!48 out
    var=56
-54 -55 56 0
54 -56 0
55 -56 0
c generate_NOT id=d var=6, out id=internal!49 out var=57
-57 -6 0
57 6 0
c generate_AND id1=internal!48 id2=internal!49 var1=56 var2=57 out id=internal!50 out
    var=58
-56 -57 58 0
56 -58 0
57 -58 0
c generate_NOT id=e var=7, out id=internal!51 out var=59
-59 -7 0
59 7 0
c generate_AND id1=internal!50 id2=internal!51 var1=58 var2=59 out id=internal!52 out
    var=60
-58 -59 60 0
58 -60 0
59 -60 0
c generate_EQ id1=f, id2=internal!52, var1=8, var2=60
c generate_XOR id1=f id2=internal!52 var1=8 var2=60 out id=internal!53 out var=61
```

```
-8 -60 -61 0
8 60 -61 0
8 -60 61 0
-8 60 61 0
c generate_NOT id=internal!53 var=61, out id=internal!54 out var=62
-62 -61 0
62 61 0
c create_assert() id=internal!54 var=62
62 0
```

Here are comments (starting with “c ” prefix), and my SMT-solver indicate, how each low-level logical gate is added, its inputs (variable IDs and numbers) and outputs.

Let’s filter comments:

```
$ cat tmp.cnf | grep "^c "

c generate_AND id1=b id2=c var1=4 var2=5 out id=internal!1 out var=9
c generate_AND id1=internal!1 id2=d var1=9 var2=6 out id=internal!2 out var=10
c generate_AND id1=internal!2 id2=e var1=10 var2=7 out id=internal!3 out var=11
c generate_AND id1=internal!3 id2=f var1=11 var2=8 out id=internal!4 out var=12
c generate_EQ id1=a, id2=internal!4, var1=3, var2=12
c generate_XOR id1=a id2=internal!4 var1=3 var2=12 out id=internal!5 out var=13
c generate_NOT id=internal!5 var=13, out id=internal!6 out var=14
c create_assert() id=internal!6 var=14
c generate_NOT id=c var=5, out id=internal!7 out var=15
c generate_NOT id=d var=6, out id=internal!8 out var=16
c generate_AND id1=internal!7 id2=internal!8 var1=15 var2=16 out id=internal!9 out var
   =17
c generate_NOT id=e var=7, out id=internal!10 out var=18
c generate_AND id1=internal!9 id2=internal!10 var1=17 var2=18 out id=internal!11 out var
   =19
c generate_NOT id=f var=8, out id=internal!12 out var=20
c generate_AND id1=internal!11 id2=internal!12 var1=19 var2=20 out id=internal!13 out
   var=21
c generate_EQ id1=b, id2=internal!13, var1=4, var2=21
c generate_XOR id1=b id2=internal!13 var1=4 var2=21 out id=internal!14 out var=22
c generate_NOT id=internal!14 var=22, out id=internal!15 out var=23
c create_assert() id=internal!15 var=23
c generate_AND id1=a id2=b var1=3 var2=4 out id=internal!16 out var=24
c generate_EQ id1=c, id2=internal!16, var1=5, var2=24
c generate_XOR id1=c id2=internal!16 var1=5 var2=24 out id=internal!17 out var=25
c generate_NOT id=internal!17 var=25, out id=internal!18 out var=26
c create_assert() id=internal!18 var=26
c generate_NOT id=b var=4, out id=internal!19 out var=27
c generate_AND id1=a id2=internal!19 var1=3 var2=27 out id=internal!20 out var=28
c generate_NOT id=c var=5, out id=internal!21 out var=29
c generate_AND id1=internal!20 id2=internal!21 var1=28 var2=29 out id=internal!22 out
   var=30
c generate_NOT id=a var=3, out id=internal!23 out var=31
c generate_AND id1=internal!23 id2=b var1=31 var2=4 out id=internal!24 out var=32
c generate_NOT id=c var=5, out id=internal!25 out var=33
c generate_AND id1=internal!24 id2=internal!25 var1=32 var2=33 out id=internal!26 out
   var=34
c generate_OR id1=internal!22 id2=internal!26 var1=30 var2=34 out id=internal!27 out var
   =35
c generate_NOT id=a var=3, out id=internal!28 out var=36
c generate_NOT id=b var=4, out id=internal!29 out var=37
```

```
c generate_AND id1=internal!28 id2=internal!29 var1=36 var2=37 out id=internal!30 out
    var=38
c generate_AND id1=internal!30 id2=c var1=38 var2=5 out id=internal!31 out var=39
c generate_OR id1=internal!27 id2=internal!31 var1=35 var2=39 out id=internal!32 out var
    =40
c generate_EQ id1=d, id2=internal!32, var1=6, var2=40
c generate_XOR id1=d id2=internal!32 var1=6 var2=40 out id=internal!33 out var=41
c generate_NOT id=internal!33 var=41, out id=internal!34 out var=42
c create_assert() id=internal!34 var=42
c generate_NOT id=a var=3, out id=internal!35 out var=43
c generate_NOT id=b var=4, out id=internal!36 out var=44
c generate_AND id1=internal!35 id2=internal!36 var1=43 var2=44 out id=internal!37 out
    var=45
c generate_NOT id=c var=5, out id=internal!38 out var=46
c generate_AND id1=internal!37 id2=internal!38 var1=45 var2=46 out id=internal!39 out
    var=47
c generate_NOT id=d var=6, out id=internal!40 out var=48
c generate_AND id1=internal!39 id2=internal!40 var1=47 var2=48 out id=internal!41 out
    var=49
c generate_EQ id1=e, id2=internal!41, var1=7, var2=49
c generate_XOR id1=e id2=internal!41 var1=7 var2=49 out id=internal!42 out var=50
c generate_NOT id=internal!42 var=50, out id=internal!43 out var=51
c create_assert() id=internal!43 var=51
c generate_NOT id=a var=3, out id=internal!44 out var=52
c generate_NOT id=b var=4, out id=internal!45 out var=53
c generate_AND id1=internal!44 id2=internal!45 var1=52 var2=53 out id=internal!46 out
    var=54
c generate_NOT id=c var=5, out id=internal!47 out var=55
c generate_AND id1=internal!46 id2=internal!47 var1=54 var2=55 out id=internal!48 out
    var=56
c generate_NOT id=d var=6, out id=internal!49 out var=57
c generate_AND id1=internal!48 id2=internal!49 var1=56 var2=57 out id=internal!50 out
    var=58
c generate_NOT id=e var=7, out id=internal!51 out var=59
c generate_AND id1=internal!50 id2=internal!51 var1=58 var2=59 out id=internal!52 out
    var=60
c generate_EQ id1=f, id2=internal!52, var1=8, var2=60
c generate_XOR id1=f id2=internal!52 var1=8 var2=60 out id=internal!53 out var=61
c generate_NOT id=internal!53 var=61, out id=internal!54 out var=62
c create_assert() id=internal!54 var=62
```

Now you can juxtapose list of internal variables and comments in CNF file. For example, equality gate is generated as NOT(XOR(a,b)).

create_assert() function fixes a bool variable to be always True.

Other (internal) variables are added by SMT solver as “joints” to connect logic gates with each other.

Hence, my SMT solver constructing a digital circuit based on the input SMT file. Logic gates are then converted into CNF form using Tseitin transformations. The task of SAT solver is then to find such an assignment, for which CNF expression would be true. In other words, its task is to find such inputs/outputs for which this “virtual” digital circuit would work without contradictions.

The SAT instance is also small enough to be solved using my simplest backtracking SAT solver written:

```
$ ./SAT_backtrack.py tmp.cnf

SAT
-1 2 -3 -4 -5 -6 7 -8 -9 -10 -11 -12 -13 14 15 16 17 -18 -19 20 -21 -22 23 -24 -25 26 27
     -28 29 -30 31 -32 33 -34 -35 36 37 38 -39 -40 -41 42 43 44 45 46 47 48 49 -50 51 52
     53 54 55 56 57 58 -59 -60 -61 62 0
```

```
UNSAT
solutions= 1
```

You can juxtapose variables from solver's result and variable numbers from MK85 listing. Therefore, MK85 + my small SAT solver is standalone program under 3000 SLOC, which still can solve such (simple enough) system of boolean equations, without external aid like minisat/picosat.

***

Among comments at the John D. Cook's blog, there is also a solution by Aaron Meurer, using SymPy, which also has SAT-solver inside:

```
7 July 2015 at 01:34

Decided to run this through 'SymPys SAT solver.

In [1]: var`(a b c d e 'f)
Out[1]: (a, b, c, d, e, f)

In [2]: facts = [
Equivalent(a, (b & c & d & e & f)),
Equivalent(b, (~c & ~d & ~e & ~f)),
Equivalent(c, a & b),
Equivalent(d, ((a & ~b & ~c) | (~a & b & ~c) | (~a & ~b & c))),
Equivalent(e, (~a & ~b & ~c & ~d)),
Equivalent(f, (~a & ~b & ~c & ~d & ~e)),
]

In [3]: list(satisfiable(And(*facts), all_models=True))
Out[3]: [{e: True, c: False, b: False, a: False, f: False, d: False}]

So it seems e is the only answer, assuming I got the facts correct. And it is important
   to use Equivalent (a bidirectional implication) rather than just Implies. If you
   only use -> (which I guess would mean that an answer not being chosen 'doesnt
   necessarily mean that it 'isnt true), then `'none, b, and f are also ""solutions.

Also, if I replace the d fact with Equivalent(d, a | b | c), the result is the same. So
   it seems that the interpretation of ""one both in terms of choice d and in terms of
   how many choices there are is irrelevant.

Thanks for the fun problem. I hope others took the time to solve this in their head
   before reading the comments.
```

### 7.18 Coin flipping problem

Found this on reddit:

```
Coin flipping problem (KOI '06) (self.algorithms)

Hi. I'm struggling on this problem for weeks.

There is no official solution for this Korean Olympiad in Informatics problem.

The input is N × N coin boards, (0 < N < 33) 'H' means the coin's head is facing upward,
    and 'T' means the tail is facing upward.

The problem is minimizing T-coins with some operations:

    Flip all the coins in the same row.
```

```
    Flip all the coins in the same column.

Naïve O(N*2N) algorithm makes TLE. There are some heuristic approaches (Simulated
    Annealing) and branch-and-cut algorithm which reduces running time to about 200ms,
    but I have no idea to solve this problem in poly-time, or reduce this problem to any
     famous NP-complete problem.

Would you give some idea for me?
```

( https://www.reddit.com/r/algorithms/comments/7aq9if/coin_flipping_problem_koi_06/ )
My solution for Z3:

```
from z3 import *
import math, random

SIZE=8

# initial state:
total=0
initial=[]
print "initial (random):"
for i in range(SIZE):
    t=random.randint(0,255)
    total=total+bin(t).count("1")
    initial.append(t)
    print "{0:b}".format(t).zfill(SIZE)
print "total=", total
print ""

# population count AKA hamming weight.
# it counts bits:
# FIXME, works only for SIZE=8
def popcnt(a):
    rt=0
    rt=rt+If(a&0x01!=0, 1, 0)
    rt=rt+If(a&0x02!=0, 1, 0)
    rt=rt+If(a&0x04!=0, 1, 0)
    rt=rt+If(a&0x08!=0, 1, 0)
    rt=rt+If(a&0x10!=0, 1, 0)
    rt=rt+If(a&0x20!=0, 1, 0)
    rt=rt+If(a&0x40!=0, 1, 0)
    rt=rt+If(a&0x80!=0, 1, 0)
    return rt

def do_all(STEPS):

    STATES=STEPS+1

    states=[[BitVec('state_%d_row_%d' % (s, r), SIZE) for r in range(SIZE)] for s in
        range(STATES)]

    # False - horizontal, True - vertical:
    H_or_V=[Bool('H_or_V_%d' % i) for i in range(STEPS)]

    # this is "shared" variable, can hold both and or column:
    row_col=[BitVec('row_col_%d' % i, SIZE) for i in range(STEPS)]
```

```
s=Optimize()

for i in range(SIZE):
    s.add(states[0][i]==BitVecVal(initial[i], 8))

# row_col can only have 0b1, 0b01, etc, in 2^n form:
for st in range(STEPS):
    s.add(Or(*[row_col[st]==(2**i) for i in range(SIZE)]))

# this is the most essential part
for st in range(STEPS):
    # if H_or_V[]==False, all rows are flipped with a value in 2^n form, i.e., a
        column is flipped
    # column is set by row_col[]:
    t=[]
    for r in range(SIZE):
        t.append(states[st][r] ^ If(H_or_V[st], row_col[st], BitVecVal(0, SIZE)))

    # if H_or_V[]==True, a row is flipped (by XORing by 0xff), and row is choosen by
         row_cal[]'s value.
    # otherwise, row is XORed by 0, i.e., idle operation.
    for r in range(SIZE):
        s.add(states[st+1][r] == t[r] ^ If(And(Not(H_or_V[st]), row_col[st]==2**r),
            BitVecVal(0xff, SIZE), BitVecVal(0, SIZE)))

# find such a solution, for which population count of all rows as small as possible,
# i.e., there are as many zero bits, as possible:
# FIXME, works only for SIZE=8
s.minimize(popcnt(states[STEPS][0])+
    popcnt(states[STEPS][1])+
    popcnt(states[STEPS][2])+
    popcnt(states[STEPS][3])+
    popcnt(states[STEPS][4])+
    popcnt(states[STEPS][5])+
    popcnt(states[STEPS][6])+
    popcnt(states[STEPS][7]))

if s.check()==unsat:
    return
m=s.model()

# print solution:
for st in range(STATES):
    if st<STEPS:
        if str(m[H_or_V[st]])=="True":
            print "vertical, ",
        else:
            print "horizontal, ",
        print int(math.log(m[row_col[st]].as_long(), 2))
    if st==STEPS-1:
        total=0
        for r in range(SIZE):
            v=m[states[st][r]].as_long()
            total=total+bin(v).count("1")
            print "{0:b}".format(v).zfill(SIZE)
        print "total=", total
```

```
            print ""

for s in range(1, 4+1):
    print "trying %d steps" % s
    #set_option(timeout=3)
    do_all(s)
```

Results:

```
initial (random):
11011011
11100001
00111111
01101100
00011010
00101010
11010011
01011101
total= 36

trying 1 steps
vertical,  3
11011011
11100001
00111111
01101100
00011010
00101010
11010011
01011101
total= 36

trying 2 steps
horizontal,  2
vertical,  6
11011011
11100001
11000000
01101100
00011010
00101010
11010011
01011101
total= 32

trying 3 steps
horizontal,  7
horizontal,  2
horizontal,  0
11011011
11100001
11000000
01101100
00011010
00101010
11010011
10100010
```

```
total= 30

trying 4 steps
vertical,  4
horizontal,  1
vertical,  1
vertical,  3
11001001
00001100
00101101
01111110
00001000
00111000
11000001
01001111
total= 28
```

And also MaxSAT solution (Open-WBO MaxSAT solver):

```
#!/usr/bin/env python

import math, SAT_lib, random

SIZE=8

# generate random 8*8 input:
initial=[[random.randint(0,1) for r in range(SIZE)] for c in range(SIZE)]

print "initial (random):"
s=0
for r in range(SIZE):
    for c in range(SIZE):
        print initial[r][c],
        s=s+initial[r][c]
    print ""
print "total=",s
print ""

"""
# test:
initial=[
    [0,0,0,0,0,1,0,0],
    [0,0,0,0,0,1,0,0],
    [0,0,0,0,0,1,0,0],
    [0,0,0,0,0,1,0,0],
    [1,1,1,1,1,1,1,1],
    [0,0,0,0,0,1,0,0],
    [0,0,0,0,0,1,0,0],
    [0,0,0,0,0,1,0,0]]
"""

def do_all(STEPS):

    # this is MaxSAT problem:
    s=SAT_lib.SAT_lib(True)

    STATES=STEPS+1
```

```
# bool variables for all 64 cells for all states:
states=[[[s.create_var() for r in range(SIZE)] for c in range(SIZE)] for st in range
    (STATES)]

# for all states:
# False - horizontal, True - vertical:
H_or_V=[s.create_var() for i in range(STEPS)]
# this is "shared" variable, can hold both and or column:
row_col=[[s.create_var() for r in range(SIZE)] for i in range(STEPS)]

# set variables on state #0:
for r in range(SIZE):
    for c in range(SIZE):
        s.fix(states[0][r][c], initial[r][c])

# each row_col bitvector can only hold one single bit:
for st in range(STEPS):
    s.make_one_hot(row_col[st])

# now this is the essence of the problem.
# each bit is flipped if H_or_V[]==False AND row_col[] is equal to its row
# ... OR H_or_V[]==True AND row_col[] is equal to its column
for st in range(STEPS):
    for r in range(SIZE):
        for c in range(SIZE):
            cond1=s.AND(s.NOT(H_or_V[st]), row_col[st][r])
            cond2=s.AND(H_or_V[st], row_col[st][c])
            bit=s.OR_list([cond1,cond2])

            s.fix(s.EQ(states[st+1][r][c], s.XOR(states[st][r][c], bit)), True)

# soft constraint: unlike hard constraints, they may be satisfied, or may be not,
# but MaxSAT solver's task is to find a solution, where maximum of soft clauses will
     be satisfied:
for r in range(SIZE):
    for c in range(SIZE):
        s.fix_soft(states[STEPS][r][c], False, 1)
        # set to True if you like to find solution with the most cells set to 1:
        #s.fix_soft(states[STEPS][r][c], True, 1)

assert s.solve()==True

# get solution:
total=0
for r in range(SIZE):
    for c in range(SIZE):
        t=s.get_var_from_solution(states[STEPS][r][c])
        print t,
        total=total+t
    print ""
print "total=", total

for st in range(STEPS):
    if s.get_var_from_solution(H_or_V[st]):
        print "vertical, ",
```

```
        else:
            print "horizontal, ",
        print int(math.log(SAT_lib.BV_to_number(s.get_BV_from_solution(row_col[st])), 2)
            )

for s in range(1,5+1):
    print "trying %d steps" % s
    do_all(s)
    print ""
```

Results:

```
initial (random):
1 0 0 1 0 1 0 1
1 1 0 1 0 1 1 1
0 1 1 1 1 0 1 1
0 1 1 1 0 0 0 0
0 0 1 0 1 1 1 0
1 1 0 0 1 0 0 1
0 1 0 0 1 1 1 1
1 1 1 1 1 0 1 1
total= 39

trying 1 steps
1 0 0 1 0 1 0 1
1 1 0 1 0 1 1 1
0 1 1 1 1 0 1 1
0 1 1 1 0 0 0 0
0 0 1 0 1 1 1 0
1 1 0 0 1 0 0 1
0 1 0 0 1 1 1 1
0 0 0 0 0 1 0 0
total= 33
horizontal,  0

trying 2 steps
1 0 0 1 0 1 0 1
1 1 0 1 0 1 1 1
1 0 0 0 0 1 0 0
0 1 1 1 0 0 0 0
0 0 1 0 1 1 1 0
1 1 0 0 1 0 0 1
0 1 0 0 1 1 1 1
0 0 0 0 0 1 0 0
total= 29
horizontal,  5
horizontal,  0

trying 3 steps
1 0 0 1 0 0 0 1
1 1 0 1 0 0 1 1
1 0 0 0 0 0 0 0
0 1 1 1 0 1 0 0
0 0 1 0 1 0 1 0
1 1 0 0 1 1 0 1
0 1 0 0 1 0 1 1
0 0 0 0 0 0 0 0
```

```
total= 25
horizontal,  0
horizontal,  5
vertical,  2

trying 4 steps
1 1 0 0 0 1 0 0
1 0 0 0 0 1 1 0
0 0 1 0 1 0 1 0
0 0 1 0 0 0 0 1
1 0 0 0 0 0 0 0
1 0 0 1 1 0 0 0
0 0 0 1 1 1 1 0
1 0 1 0 1 0 1 0
total= 23
horizontal,  3
vertical,  6
vertical,  0
vertical,  4

trying 5 steps
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 1 1
0 0 0 1 1 0 0 0
0 1 0 0 0 1 1 0
1 0 1 0 0 0 0 1
0 0 1 0 0 1 1 1
1 0 0 1 0 0 1 1
total= 21
horizontal,  6
horizontal,  7
vertical,  3
vertical,  5
vertical,  6
```

4-5 row/column flips can be achieved using this on 8*8 board (like chessboard) in reasonable time (couple of minutes).

## 7.19 Lucky tickets

It's quite popular meme in ex-USSR countries: "lucky" ticket is a ticket, which 6-digit number has the following property: sum of the first 3 digits equals to the sum of the last 3 digits.

How many "lucky" tickets exists?

```
; tested with MK85

(declare-fun n1 () (_ BitVec 8))
(declare-fun n2 () (_ BitVec 8))
(declare-fun n3 () (_ BitVec 8))
(declare-fun n4 () (_ BitVec 8))
(declare-fun n5 () (_ BitVec 8))
(declare-fun n6 () (_ BitVec 8))

(assert (and (bvuge n1 #x00) (bvule n1 #x09)))
(assert (and (bvuge n2 #x00) (bvule n2 #x09)))
(assert (and (bvuge n3 #x00) (bvule n3 #x09)))
(assert (and (bvuge n4 #x00) (bvule n4 #x09)))
```

```
(assert (and (bvuge n5 #x00) (bvule n5 #x09)))
(assert (and (bvuge n6 #x00) (bvule n6 #x09)))

(declare-fun sum1 () (_ BitVec 8))
(assert (= sum1 (bvadd n1 n2 n3)))

(declare-fun sum2 () (_ BitVec 8))
(assert (= sum2 (bvadd n4 n5 n6)))

(assert (= sum1 sum2))

;(check-sat)
;(get-model)
;(get-all-models)
(count-models)

; correct answer:
; Model count: 55252
```

## 7.20 Art of problem solving

http://artofproblemsolving.com/wiki/index.php?title=Mock_AIME_2_2006-2007_Problems/Problem_8:

> The positive integers $x_1, x_2, ..., x_7$ satisfy $x_6 = 144$ and $x_{n+3} = x_{n+2}(x_{n+1} + x_n)$ for $n = 1, 2, 3, 4$. Find the last three digits of $x_7$.

This is it:

```
from z3 import *

s=Solver()

x1, x2, x3, x4, x5, x6, x7=Ints('x1 x2 x3 x4 x5 x6 x7')

s.add(x1>=0)
s.add(x2>=0)
s.add(x3>=0)
s.add(x4>=0)
s.add(x5>=0)
s.add(x6>=0)
s.add(x7>=0)

s.add(x6==144)

s.add(x4==x3*(x2+x1))
s.add(x5==x4*(x3+x2))
s.add(x6==x5*(x4+x3))
s.add(x7==x6*(x5+x4))

# get all results:

results=[]
while True:
    if s.check() == sat:
```

```
        m = s.model()
        print m

        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "total results", len(results)
        break
```

Two solutions possible, but in both x7 is ending by 456:

```
[x2 = 1,
 x3 = 1,
 x1 = 7,
 x4 = 8,
 x5 = 16,
 x7 = 3456,
 x6 = 144]
[x3 = 2,
 x2 = 1,
 x1 = 2,
 x6 = 144,
 x4 = 6,
 x5 = 18,
 x7 = 3456]
total results 2
```

### 7.21 2012 AIME I Problems/Problem 1

http://artofproblemsolving.com/wiki/index.php?title=2012_AIME_I_Problems/Problem_1

Find the number of positive integers with three not necessarily distinct digits, $abc$, with $a \neq 0$ and $c \neq 0$ such that both $abc$ and $cba$ are multiples of 4.

```
from z3 import *

a, b, c = Ints('a b c')

s=Solver()

s.add(a>0)
s.add(b>=0)
s.add(c>0)

s.add(a<=9)
s.add(b<=9)
s.add(c<=9)

s.add((a*100 + b*10 + c) % 4 == 0)
```

```
s.add((c*100 + b*10 + a) % 4 == 0)

results=[]
while True:
    if s.check() == sat:
        m = s.model()
        print m

        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "total results", len(results)
        break
```

Let's see:

```
[c = 4, b = 0, a = 4]
[b = 1, c = 2, a = 2]
[b = 6, c = 4, a = 4]
[b = 4, c = 4, a = 4]
[b = 2, c = 4, a = 4]
[b = 4, c = 4, a = 8]
[b = 8, c = 4, a = 4]
[b = 6, c = 4, a = 8]
[b = 8, c = 4, a = 8]
[b = 0, c = 4, a = 8]
[b = 2, c = 4, a = 8]
[b = 8, c = 8, a = 8]
[b = 9, c = 6, a = 6]
[b = 2, c = 8, a = 8]
[b = 2, c = 8, a = 4]
[b = 4, c = 8, a = 4]
[b = 4, c = 8, a = 8]
[b = 8, c = 8, a = 4]
[b = 6, c = 8, a = 4]
[b = 6, c = 8, a = 8]
[b = 0, c = 8, a = 4]
[b = 0, c = 8, a = 8]
[b = 7, c = 6, a = 6]
[b = 7, c = 2, a = 6]
[b = 7, c = 6, a = 2]
[b = 7, c = 2, a = 2]
[b = 9, c = 2, a = 6]
[b = 9, c = 2, a = 2]
[b = 9, c = 6, a = 2]
[b = 5, c = 6, a = 2]
[b = 1, c = 6, a = 2]
[b = 3, c = 6, a = 2]
[b = 5, c = 2, a = 2]
[b = 3, c = 2, a = 2]
[b = 5, c = 6, a = 6]
[b = 5, c = 2, a = 6]
[b = 3, c = 2, a = 6]
```

```
[b = 1, c = 2, a = 6]
[b = 1, c = 6, a = 6]
[b = 3, c = 6, a = 6]
total results 40
```

My toy-level SMT-solver MK85 can enumerate all solutions as well:

```
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun a () (_ BitVec 8))
(declare-fun b () (_ BitVec 8))
(declare-fun c () (_ BitVec 8))

(assert (bvugt a #x00))
(assert (bvuge b #x00))
(assert (bvugt c #x00))

(assert (bvule a #x09))
(assert (bvule b #x09))
(assert (bvule c #x09))

; slower:
;(assert (= (bvurem (bvadd (bvmul a (_ bv100 8)) (bvmul b (_ bv00 8)) c) #x04) #x00))
;(assert (= (bvurem (bvadd (bvmul c (_ bv100 8)) (bvmul b (_ bv00 8)) a) #x04) #x00))

; faster:
(assert (= (bvand (bvadd (bvmul a (_ bv100 8)) (bvmul b (_ bv00 8)) c) #x03) #x00))
(assert (= (bvand (bvadd (bvmul c (_ bv100 8)) (bvmul b (_ bv00 8)) a) #x03) #x00))

;(check-sat)
;(get-model)
(get-all-models)
;(count-models)
```

Faster version doesn't find remainder, it just isolates two last bits.

## 7.22 Recreational math, calculator's keypad and divisibility

I've once read about this puzzle. Imagine calculator's keypad:

```
789
456
123
```

If you form any rectangle or square out of keys, like:

```
 7 8 9
+---+
|4 5|6
|1 2|3
+---+
```

The number is 4521. Or 2145, or 5214. All these numbers are divisible by 11, 111 and 111. One explanation: https://files.eric.ed.gov/fulltext/EJ891796.pdf.

However, I could try to prove that all these numbers are indeed divisible.

```


from z3 import *
```

```
"""
We will keep track on numbers using row/col representation:

 |0 1 2 <-col
-|- - -
0|7 8 9
1|4 5 6
2|1 2 3
^
|
row

"""

# map coordinates to number on keypad:
def coords_to_num (r, c):
    return If(And(r==0, c==0), 7,
    If(And(r==0, c==1), 8,
    If(And(r==0, c==2), 9,
    If(And(r==1, c==0), 4,
    If(And(r==1, c==1), 5,
    If(And(r==1, c==2), 6,
    If(And(r==2, c==0), 1,
    If(And(r==2, c==1), 2,
    If(And(r==2, c==2), 3, 9999)))))))))

s=Solver()

# coordinates of upper left corner:
from_r, from_c = Ints('from_r from_c')
# coordinates of bottom right corner:
to_r, to_c = Ints('to_r to_c')

# all coordinates are in [0..2]:
s.add(And(from_r>=0, from_r<=2, from_c>=0, from_c<=2))
s.add(And(to_r>=0, to_r<=2, to_c>=0, to_c<=2))

# bottom-right corner is always under left-upper corner, or equal to it, or to the right
    of it:
s.add(to_r>=from_r)
s.add(to_c>=from_c)

# numbers on keypads for all 4 corners:
LT, RT, BL, BR = Ints('LT RT BL BR')

# ... which are:
s.add(LT==coords_to_num(from_r, from_c))
s.add(RT==coords_to_num(from_r, to_c))
s.add(BL==coords_to_num(to_r, from_c))
s.add(BR==coords_to_num(to_r, to_c))

# 4 possible 4-digit numbers formed by passing by 4 corners:
n1, n2, n3, n4 = Ints('n1 n2 n3 n4')

s.add(n1==LT*1000 + RT*100 + BR*10 + BL)
```

```
s.add(n2==RT*1000 + BR*100 + BL*10 + LT)
s.add(n3==BR*1000 + BL*100 + LT*10 + RT)
s.add(n4==BL*1000 + LT*100 + RT*10 + BR)

# what we're going to do?
prove=False
enumerate_rectangles=True

assert prove != enumerate_rectangles

if prove:
    # prove by finding counterexample.
    # find any variable state for which remainder will be non-zero:
    s.add(And((n1%11) != 0), (n1%111) != 0, (n1%1111) != 0)
    s.add(And((n2%11) != 0), (n2%111) != 0, (n2%1111) != 0)
    s.add(And((n3%11) != 0), (n3%111) != 0, (n3%1111) != 0)
    s.add(And((n4%11) != 0), (n4%111) != 0, (n4%1111) != 0)

    # this is impossible, we're getting unsat here, because no counterexample exist:
    print s.check()

# ... or ...

if enumerate_rectangles:
    # enumerate all possible solutions:
    results=[]
    while True:
        if s.check() == sat:
            m = s.model()
            #print_model(m)
            print m
            print m[n1]
            print m[n2]
            print m[n3]
            print m[n4]
            results.append(m)
            block = []
            for d in m:
                c=d()
                block.append(c != m[d])
            s.add(Or(block))
        else:
            print "results total=", len(results)
            break
```

Enumeration. only 36 rectangles exist on 3*3 keypad:

```
[n1 = 7821,
 BL = 1,
 n2 = 8217,
 to_r = 2,
 LT = 7,
 RT = 8,
 BR = 2,
 n4 = 1782,
 from_r = 0,
 n3 = 2178,
```

```
 from_c = 0,
 to_c = 1]
7821
8217
2178
1782
[n1 = 7931,
 BL = 1,
 n2 = 9317,
 to_r = 2,
 LT = 7,
 RT = 9,
 BR = 3,
 n4 = 1793,
 from_r = 0,
 n3 = 3179,
 from_c = 0,
 to_c = 2]
7931
9317
3179
1793

...

[n1 = 5522,
 BL = 2,
 n2 = 5225,
 to_r = 2,
 LT = 5,
 RT = 5,
 BR = 2,
 n4 = 2552,
 from_r = 1,
 n3 = 2255,
 from_c = 1,
 to_c = 1]
5522
5225
2255
2552
results total= 36
```

### 7.23 Android lock screen (9 dots) has exactly 140240 possible ways to (un)lock it

How would you count?

```
from z3 import *

"""
1 2 3
4 5 6
7 8 9
"""

# where the next dot can be if the current dot is at $a$
```

```
# next dot can only be a neighbour
# here we define starlike connections between dots (as in Android lock screen)
# this is like switch() or multiplexer

# lines like these are also counted:
# * . .
# . . *
# . * .
def next_dot(a, b):
    return If(a==1, Or(b==2, b==4, b==5, b==6, b==8),
        If(a==2, Or(b==1, b==3, b==4, b==5, b==6, b==7, b==9),
        If(a==3, Or(b==2, b==5, b==6, b==4, b==8),
        If(a==4, Or(b==1, b==2, b==5, b==7, b==8, b==3, b==9),
        If(a==5, Or(b==1, b==2, b==3, b==4, b==6, b==7, b==8, b==9),
        If(a==6, Or(b==2, b==3, b==5, b==8, b==9, b==1, b==7),
        If(a==7, Or(b==4, b==5, b==8, b==2, b==6),
        If(a==8, Or(b==4, b==5, b==6, b==7, b==9, b==1, b==3),
        If(a==9, Or(b==5, b==6, b==8, b==4, b==2),
            False))))))))) # default

# if only non-diagonal lines between dots are allowed:
"""
def next_dot(a, b):
    return If(a==1, Or(b==2, b==4),
        If(a==2, Or(b==1, b==3, b==5),
        If(a==3, Or(b==2, b==6),
        If(a==4, Or(b==1, b==5, b==7),
        If(a==5, Or(b==2, b==4, b==6, b==8),
        If(a==6, Or(b==3, b==5, b==9),
        If(a==7, Or(b==4, b==8),
        If(a==8, Or(b==5, b==7, b==9),
        If(a==9, Or(b==6, b==8),
            False))))))))) # default
"""

# old version, hasn't counted lines like
# * . .
# . . *
# . * .

"""
def next_dot(a, b):
    return If(a==1, Or(b==2, b==4, b==5),
        If(a==2, Or(b==1, b==3, b==4, b==5, b==6),
        If(a==3, Or(b==2, b==5, b==6),
        If(a==4, Or(b==1, b==2, b==5, b==7, b==8),
        If(a==5, Or(b==1, b==2, b==3, b==4, b==6, b==7, b==8, b==9),
        If(a==6, Or(b==2, b==3, b==5, b==8, b==9),
        If(a==7, Or(b==4, b==5, b==8),
        If(a==8, Or(b==4, b==5, b==6, b==7, b==9),
        If(a==9, Or(b==5, b==6, b==8),
            False))))))))) # default
"""

def paths_for_length (LENGTH):
    s=Solver()
```

```
    path=[Int('path_%d' % i) for i in range(LENGTH)]

    # all elements of path must be distinct
    s.add(Distinct(path))

    # all elements in [1..9] range:
    for i in range(LENGTH):
        s.add(And(path[i]>=1, path[i]<=9))

    # next element of path is defined by next_dot() function, unless it's the last one:
    for i in range(LENGTH-1):
        s.add(next_dot(path[i], path[i+1]))

    results=[]

    # enumerate all possible solutions:
    while True:
        if s.check() == sat:
            m = s.model()
            tmp=[]
            for i in range(LENGTH):
                tmp.append(m[path[i]].as_long())
            #print m
            print "path", tmp
            # print visual representation:
            for k in [[1,2,3],[4,5,6],[7,8,9]]:
                for j in k:
                    if j in tmp:
                        print tmp.index(j)+1,
                    else:
                        print ".",
                print ""
            print ""
            results.append(m)
            block = []
            for d in m:
                c=d()
                block.append(c != m[d])
            s.add(Or(block))
        else:
            print "length=", LENGTH, "results total=", len(results)
            return len(results)

total=0
for l in range(2,10):
    total=total+paths_for_length(l)

print "total=", total
```

Sample paths of 7 elements:

```
...

path [7, 5, 1, 4, 8, 6, 3]
3 . 7
4 2 6
```

```
1 5 .

path [9, 5, 7, 4, 8, 6, 3]
. . 7
4 2 6
3 5 1

path [9, 5, 1, 4, 8, 6, 3]
3 . 7
4 2 6
. 5 1

...
```

Each element of "path" is number of dot, like on phone's keypad:

```
1 2 3
4 5 6
7 8 9
```

Numbers on $3 \cdot 3$ box represent a sequence: which dot is the 1st, 2nd, etc...
Of 9:

```
...

path [7, 8, 9, 5, 4, 1, 2, 6, 3]
6 7 9
5 4 8
1 2 3

path [1, 4, 7, 5, 2, 3, 6, 9, 8]
1 5 6
2 4 7
3 9 8

path [9, 6, 8, 7, 4, 1, 5, 2, 3]
6 8 9
5 7 2
4 3 1

...
```

All possible paths: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/puzzles/Android/all.bz2.
Statistics:

```
length= 2 results total= 56
length= 3 results total= 304
length= 4 results total= 1400
length= 5 results total= 5328
length= 6 results total= 16032
length= 7 results total= 35328
length= 8 results total= 49536
length= 9 results total= 32256
total= 140240
```

What if only non-diagonal lines would be allowed (which isn't a case of a real Android lock screen)?

```
length= 2 results total= 24
```

```
length= 3 results total= 44
length= 4 results total= 80
length= 5 results total= 104
length= 6 results total= 128
length= 7 results total= 112
length= 8 results total= 112
length= 9 results total= 40
total= 644
```

Also, at first, when I published this note, lines like these weren't counted (but allowable on Andoird lock screen, as it was pointed out by @mztropics):

```
* . .
. . *
. * .
```

And the [incorrect] statistics was like this:

```
length= 2 results total= 40
length= 3 results total= 160
length= 4 results total= 496
length= 5 results total= 1208
length= 6 results total= 2240
length= 7 results total= 2984
length= 8 results total= 2384
length= 9 results total= 784
total= 10296
```

Now you can see, how drastically number of all possibilities can change, when you add $\approx 2$ more branches at each element of path.

## 7.24 Crossword generator

We assign an integer to each character in crossword, it reflects ASCII code of it.

Then we enumerate all possible horizontal/vertical "sticks" longer than 1 and assign words to them.

For example, there is a horizontal stick of length 3. And we have such 3-letter words in our dictionary: "the", "she", "xor".

We add the following constraint:

```
Or(
        And(chars[X][Y]=='t', chars[X][Y+1]=='h', chars[X][Y+2]=='e'),
        And(chars[X][Y]=='s', chars[X][Y+1]=='h', chars[X][Y+2]=='e'),
        And(chars[X][Y]=='x', chars[X][Y+1]=='o', chars[X][Y+2]=='r'))
```

One of these words would be choosen automatically.

Index of each word is also considered, because duplicates are not allowed.

Sample pattern:

```
**** **********
 * * *  * * * *
***************
 * * *  * * * *
********* *****
 * * * * * *  *
****** ********
   * * * * *
******** ******
*  * * * * * *
***** *********
```

```
* * * *  * * *
***************
* * * *  * * *
********** ****
```

Sample result:

```
spur stimulated
 r e c  i a h e
congratulations
 m u t  a i s c
violation niece
 s a e p e n  n
rector penitent
   i i o c e
accounts herald
s  n g e a r o
press edinburgh
e x e n  t p i
characteristics
t c l r  n e a
satisfying dull

horizontal:
((0, 0), (0, 3)) spur
((0, 5), (0, 14)) stimulated
((2, 0), (2, 14)) congratulations
((4, 0), (4, 8)) violation
((4, 10), (4, 14)) niece
((6, 0), (6, 5)) rector
((6, 7), (6, 14)) penitent
((8, 0), (8, 7)) accounts
((8, 9), (8, 14)) herald
((10, 0), (10, 4)) press
((10, 6), (10, 14)) edinburgh
((12, 0), (12, 14)) characteristics
((14, 0), (14, 9)) satisfying
((14, 11), (14, 14)) dull
vertical:
((8, 0), (14, 0)) aspects
((0, 1), (6, 1)) promise
((10, 2), (14, 2)) exact
((0, 3), (10, 3)) regulations
((10, 4), (14, 4)) seals
((0, 5), (9, 5)) scattering
((10, 6), (14, 6)) entry
((4, 7), (10, 7)) opposed
((0, 8), (4, 8)) milan
((5, 9), (14, 9)) enchanting
((0, 10), (4, 10)) latin
((4, 11), (14, 11)) interrupted
((0, 12), (4, 12)) those
((8, 13), (14, 13)) logical
((0, 14), (6, 14)) descent
```

Unsat is possible if the dictionary is too small or have no words of length present in pattern.

The source code:

```
#!/usr/bin/env python

from z3 import *
import sys

"""
# https://commons.wikimedia.org/wiki/File:Khachbar-1.jpg
pattern=[
"*****",
"* * *",
"* ***",
"*** *",
"* ***",
"* * *",
"*****"]
"""

"""
# https://commons.wikimedia.org/wiki/File:Khachbar-4.jpg
pattern=[
"*******",
"* * * *",
"*******",
"* * * *",
"*******",
"* * * *",
"*******"]
"""

# https://commons.wikimedia.org/wiki/File:British_crossword.svg
pattern=[
"**** **********",
" * * *  * * * *",
"***************",
" * * *  * * * *",
"********* *****",
" * * * * * *  *",
"****** ********",
"   * * * * *   ",
"******** ******",
"*  * * * * * * ",
"***** *********",
"* * * *  * * * ",
"***************",
"* * * *  * * * ",
"********** ****"]

HEIGHT=len(pattern)
WIDTH=len(pattern[0])

# scan pattern[] and find all "sticks" longer than 1 and collect its coordinates:

horizontal=[]

in_the_middle=False
for r in range(HEIGHT):
```

```
    for c in range(WIDTH):
        if pattern[r][c]=='*' and in_the_middle==False:
            in_the_middle=True
            start=(r,c)
        elif pattern[r][c]==' ' and in_the_middle==True:
            if c-start[1]>1:
                horizontal.append ((start, (r, c-1)))
            in_the_middle=False
    if in_the_middle:
        if c-start[1]>1:
            horizontal.append ((start, (r, c)))
        in_the_middle=False

vertical=[]

in_the_middle=False
for c in range(WIDTH):
    for r in range(HEIGHT):
        if pattern[r][c]=='*' and in_the_middle==False:
            in_the_middle=True
            start=(r,c)
        elif pattern[r][c]==' ' and in_the_middle==True:
            if r-start[0]>1:
                vertical.append ((start, (r-1, c)))
            in_the_middle=False
    if in_the_middle:
        if r-start[0]>1:
            vertical.append ((start, (r, c)))
        in_the_middle=False

# for the first simple pattern, we will have such coordinates of "sticks":
# horizontal=[((0, 0), (0, 4)), ((2, 2), (2, 4)), ((3, 0), (3, 2)), ((4, 2), (4, 4)),
   ((6, 0), (6, 4))]
# vertical=[((0, 0), (6, 0)), ((0, 2), (6, 2)), ((0, 4), (6, 4))]

# the list in this file is assumed to not have duplicates, otherwise duplicates can be
   present in the final resulting crossword:
with open("words.txt") as f:
    content = f.readlines()
words = [x.strip() for x in content]

# FIXME: slow, called too often
def find_words_len(l):
    rt=[]
    i=0
    for word in words:
        if len(word)==l:
            rt.append ((i, word))
        i=i+1
    return rt

# 2D array of ASCII codes of all characters:
chars=[[Int('chars_%d_%d' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
# indices of horizontal words:
horizontal_idx=[Int('horizontal_idx_%d' % i) for i in range(len(horizontal))]
# indices of vertical words:
```

```
vertical_idx=[Int('vertical_idx_%d' % i) for i in range(len(vertical))]

s=Solver()

# this function takes coordinates, word length and word itself
# for "hello", it returns array like:
# [chars[0][0]==ord('h'), chars[0][1]==ord('e'), chars[0][2]==ord('l'), chars[0][3]==ord
    ('l'), chars[0][4]==ord('o')]
def form_H_expr(r, c, l, w):
    return [chars[r][c+i]==ord(w[i]) for i in range(l)]

# now we find all horizonal "sticks", we find all possible words of corresponding length
    ...
for i in range(len(horizontal)):
    h=horizontal[i]
    _from=h[0]
    _to=h[1]
    l=_to[1]-_from[1]+1
    list_of_ANDs=[]
    for idx, word in find_words_len(l):
        # at this point, we form expression like:
        # And(chars[0][0]==ord('h'), chars[0][1]==ord('e'), chars[0][2]==ord('l'), chars
            [0][3]==ord('l'), chars[0][4]==ord('o'), horizontal_idx[]==...)
        list_of_ANDs.append(And(form_H_expr(_from[0], _from[1], l, word)+[horizontal_idx
            [i]==idx]))
    # at this point, we form expression like:
    # Or(And(chars...==word1), And(chars...==word2), And(chars...==word3))
    s.add(Or(*list_of_ANDs))

# same for vertical "sticks":
def form_V_expr(r, c, l, w):
    return [chars[r+i][c]==ord(w[i]) for i in range(l)]

for i in range(len(vertical)):
    v=vertical[i]
    _from=v[0]
    _to=v[1]
    l=_to[0]-_from[0]+1
    list_of_ANDs=[]
    for idx, word in find_words_len(l):
        list_of_ANDs.append(And(form_V_expr(_from[0], _from[1], l, word)+[vertical_idx[i
            ]==idx]))
    s.add(Or(*list_of_ANDs))

# we collected indices of horizonal/vertical words to make sure they will not be
    duplicated on resulting crossword:
s.add(Distinct(*(horizontal_idx+vertical_idx)))

def print_model (m):
    print ""
    for r in range(HEIGHT):
        for c in range(WIDTH):
            if pattern[r][c]=='*':
                sys.stdout.write(chr(m[chars[r][c]].as_long()))
            else:
                sys.stdout.write(' ')
```

```
        print ""
    print ""

    print "horizontal:"
    for i in range(len(horizontal)):
        print horizontal[i], words[m[horizontal_idx[i]].as_long()]

    print "vertical:"
    for i in range(len(vertical)):
        print vertical[i], words[m[vertical_idx[i]].as_long()]

N=10
# get 10 results:
results=[]
for i in range(N):
    if s.check() == sat:
        m = s.model()
        print_model(m)
        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "total results", len(results)
        break
```

The files, including my dictionary: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/puzzles/cross.

# 8 Graph coloring

It's possible to color all countries on any political map (or planar graph) using only 4 colors. This is example from Wolfram Mathematica's website (using FindInstance):

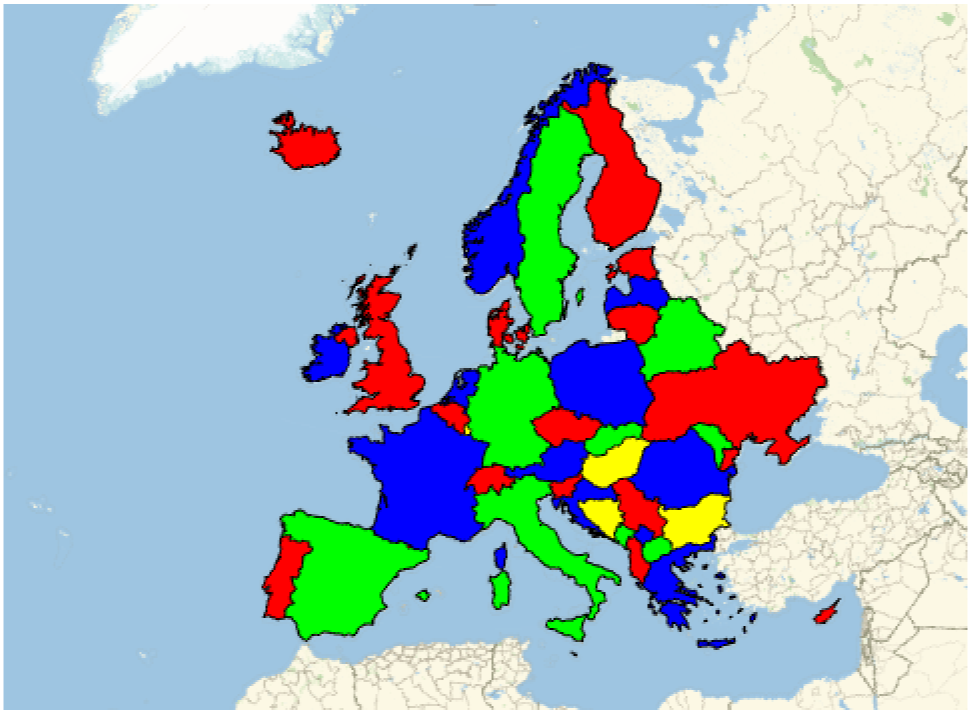


Figure 28: Colored map

( https://www.wolfram.com/mathematica/new-in-10/entity-based-geocomputation/find-a-four-coloring-of-a-map-html )

Any map or vertices on planar graph can be colored using at most 4 colors. This is quite interesting story behind this. This is a first serious proof finished using automated theorem prover (Coq): https://en.wikipedia.org/wiki/Four_color_theorem.

An example where I use graph coloring: 21.6.

## 8.1 Using graph coloring in scheduling

I've found this problem in the "Discrete Structures, Logic and Computability" book by James L. Hein:

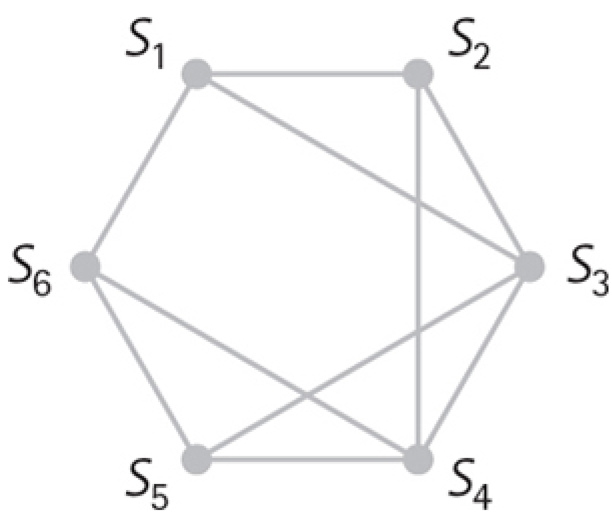



Figure 29: Graph

Suppose some people form committees to do various tasks. The problem is to schedule the committee meetings in as few time slots as possible. To simplify the discussion, we'll represent each person with a number. For example, let S = 1, 2, 3, 4, 5, 6, 7 represent a set of seven people, and suppose they have formed six three-person committees as follows:

S1 = 1, 2, 3, S2 = 2, 3, 4, S3 = 3, 4, 5, S4 = 4, 5, 6, S5 = 5, 6, 7, S6 = 1, 6, 7.

We can model the problem with the graph pictured in Figure 1.4.4, where the committee names are the vertices and an edge connects two vertices if a person belongs to both committees represented by the vertices. If each committee meets for one hour, what is the smallest number of hours needed for the committees to do their work? From the graph, it follows that an edge between two committees means that they have at least one member in common. Thus, they cannot meet at the same time. No edge between committees means that they can meet at the same time. For example, committees S1 and S4 can meet the first hour. Then committees S2 and S5 can meet the second hour. Finally, committees S3 and S6 can meet the third hour. Can you see why three hours is the smallest number of hours that the six committees can meet?

And this is solution:

```
#!/usr/bin/env python

import itertools
from z3 import *

# 7 peoples, 6 committees

S={}
S[1]=set([1, 2, 3])
S[2]=set([2, 3, 4])
S[3]=set([3, 4, 5])
S[4]=set([4, 5, 6])
S[5]=set([5, 6, 7])
S[6]=set([1, 6, 7])

committees=len(S)

s=Solver()

Color_or_Hour=[Int('Color_or_hour_%d' % i) for i in range(committees)]

# enumerate all possible pairs of committees:
for pair in itertools.combinations(S, r=2):
    # if intersection of two sets has *something* (i.e., if two committees has at least
        one person in common):
    if len(S[pair[0]] & S[pair[1]])>0:
        # ... add an edge between vertices (or persons) -- these colors (or hours) must
            differ:
        s.add(Color_or_Hour[pair[0]-1] != Color_or_Hour[pair[1]-1])

# limit all colors (or hours) in 0..2 range (3 colors/hours):
for i in range(committees):
    s.add(And(Color_or_Hour[i]>=0, Color_or_Hour[i]<=2))

assert s.check()==sat
m=s.model()
#print m
```

```
schedule={}

for i in range(committees):
    hour=m[Color_or_Hour[i]].as_long()
    if hour not in schedule:
        schedule[hour]=[]
    schedule[hour].append(i+1)

for t in schedule:
    print "hour:", t, "committees:", schedule[t]
```

The result:

```
hour: 0 committees: [1, 4]
hour: 1 committees: [2, 5]
hour: 2 committees: [3, 6]
```

If you increase total number of hours to 4, the result is somewhat sparser:

```
hour: 0 committees: [3]
hour: 1 committees: [1, 4]
hour: 2 committees: [2, 5]
hour: 3 committees: [6]
```

## 8.2 Another example

What if we want to divide our community/company/university by groups. There are 16 persons and, which must be divided by 4 groups, 4 persons in each. However, several persons hate each other, maybe, for personal reasons. Can we group all them so the "haters" would be separated?

```
from z3 import *

# 16 peoples, 4 groups

PERSONS=16
GROUPS=4

s=Solver()

person=[Int('person_%d' % i) for i in range(PERSONS)]

# each person must belong to some group in 0..GROUPS range:
for i in range(PERSONS):
    s.add(And(person[i]>=0, person[i]<GROUPS))

# each pair of persons can't be in the same group, because they hate each other.
# IOW, we add an edge between vertices.
s.add(person[0] != person[7])
s.add(person[0] != person[8])
s.add(person[0] != person[9])
s.add(person[2] != person[9])
s.add(person[9] != person[14])
s.add(person[11] != person[15])
s.add(person[11] != person[1])
s.add(person[11] != person[2])
s.add(person[11] != person[9])
s.add(person[10] != person[1])
```

```
persons_in_group=[Int('persons_in_group_%d' % i) for i in range(GROUPS)]

def count_persons_in_group(g):
    """
    Form expression like:

    If(person_0 == g, 1, 0) +
    If(person_1 == g, 1, 0) +
    If(person_2 == g, 1, 0) +
    ...
    If(person_15 == g, 1, 0)

    """
    return Sum(*[If(person[i]==g, 1, 0) for i in range(PERSONS)])

# each group must have 4 persons:
for g in range(GROUPS):
    s.add(count_persons_in_group(g)==4)

assert s.check()==sat
m=s.model()

groups={}
for i in range(PERSONS):
    g=m[person[i]].as_long()
    if g not in groups:
        groups[g]=[]
    groups[g].append(i)

for g in groups:
    print "group %d, persons:" % g, groups[g]
```

The result:

```
group 0, persons: [1, 2, 5, 8]
group 1, persons: [4, 7, 9, 12]
group 2, persons: [0, 3, 11, 13]
group 3, persons: [6, 10, 14, 15]
```

# 9 Knapsack problems

## 9.1 Popsicles

Found this problem at http://artofproblemsolving.com/wiki/index.php?title=2017_AMC_12A_Problems/Problem_1:

> Pablo buys popsicles for his friends. The store sells single popsicles for \$1 each, 3-popsicle boxes for \$2, and 5-popsicle boxes for \$3. What is the greatest number of popsicles that Pablo can buy with \$8?

This is optimization problem, and the solution using z3:

```
from z3 import *
```

```
box1pop, box3pop, box5pop = Ints('box1pop box3pop box5pop')
pop_total = Int('pop_total')
cost_total = Int('cost_total')

s=Optimize()

s.add(pop_total == box1pop*1 + box3pop*3 + box5pop*5)
s.add(cost_total == box1pop*1 + box3pop*2 + box5pop*3)

s.add(cost_total==8)

s.add(box1pop>=0)
s.add(box3pop>=0)
s.add(box5pop>=0)

s.maximize(pop_total)

print s.check()
print s.model()
```

The result:

```
sat
[box3pop = 1,
 box5pop = 2,
 cost_total = 8,
 pop_total = 13,
 box1pop = 0]
```

### 9.1.1 SMT-LIB 2.x

```
; tested with MK85 and Z3

(declare-fun box1pop () (_ BitVec 16))
(declare-fun box3pop () (_ BitVec 16))
(declare-fun box5pop () (_ BitVec 16))
(declare-fun pop_total () (_ BitVec 16))
(declare-fun cost_total () (_ BitVec 16))

(assert (=
        ((_ zero_extend 16) pop_total)
        (bvadd
                ((_ zero_extend 16) box1pop)
                (bvmul ((_ zero_extend 16) box3pop) #x00000003)
                (bvmul ((_ zero_extend 16) box5pop) #x00000005)
        )))

(assert (=
        ((_ zero_extend 16) cost_total)
        (bvadd
                ((_ zero_extend 16) box1pop)
                (bvmul ((_ zero_extend 16) box3pop) #x00000002)
                (bvmul ((_ zero_extend 16) box5pop) #x00000003)
        )))

(assert (= cost_total #x0008))
```

```
(maximize pop_total)

(check-sat)
(get-model)

; correct solution:

;(model
;         (define-fun box1pop () (_ BitVec 16) (_ bv0 16)) ; 0x0
;         (define-fun box3pop () (_ BitVec 16) (_ bv1 16)) ; 0x1
;         (define-fun box5pop () (_ BitVec 16) (_ bv2 16)) ; 0x2
;         (define-fun pop_total () (_ BitVec 16) (_ bv13 16)) ; 0xd
;         (define-fun cost_total () (_ BitVec 16) (_ bv8 16)) ; 0x8
;)
```

# 10 Social Golfer Problem

"Twenty golfers wish to play in foursomes for 5 days. Is it possible for each golfer to play no more than once with any other golfer?" ( http://mathworld.wolfram.com/SocialGolferProblem.html )

## 10.1 Kirkman's Schoolgirl Problem (Z3Py)

> Fifteen young ladies in a school walk out three abreast for seven days in succession: it is required to arrange them daily so that no two shall walk twice abreast.

( https://en.wikipedia.org/wiki/Kirkman%27s_schoolgirl_problem )

This is naive and straightforward solution:

```
from z3 import *
import itertools

PERSONS, DAYS, GROUPS = 15, 7, 5
#PERSONS, DAYS, GROUPS = 20, 5, 5

# each element - group for each person and each day:
tbl=[[Int('%d_%d' % (person, day)) for day in range(DAYS)] for person in range(PERSONS)]

s=Solver()

for person in range(PERSONS):
    for day in range(DAYS):
        s.add(And(tbl[person][day]>=0, tbl[person][day] < GROUPS))

# one in pair must be equal, all others must differ:
def only_one_in_pair_can_be_equal(l1, l2):
    assert len(l1)==len(l2)
    expr=[]
    for pair_eq in range(len(l1)):
        tmp=[]
        for i in range(len(l1)):
            if pair_eq==i:
                tmp.append(l1[i]==l2[i])
```

```
            else:
                tmp.append(l1[i]!=l2[i])
        expr.append(And(*tmp))

# at this point, expression like this constructed:
# Or(
#       And(l1[0]==l2[0], l1[1]!=l2[1], l1[2]!=l2[2])
#       And(l1[0]!=l2[0], l1[1]==l2[1], l1[2]!=l2[2])
#       And(l1[0]!=l2[0], l1[1]!=l2[1], l1[2]==l2[2])
#   )

    s.add(Or(*expr))

# enumerate all possible pairs.
for pair in itertools.combinations(range(PERSONS), r=2):
    only_one_in_pair_can_be_equal (tbl[pair[0]], tbl[pair[1]])

print s.check()
m=s.model()

print "group for each person:"
print "person:"+"".join([chr(ord('A')+i)+" " for i in range(PERSONS)])
for day in range(DAYS):
    print "day=%d:" % day,
    for person in range(PERSONS):
        print m[tbl[person][day]].as_long(),
    print ""

def persons_in_group(day, group):
    rt=""
    for person in range(PERSONS):
        if m[tbl[person][day]].as_long()==group:
            rt=rt+chr(ord('A')+person)
    return rt

print ""
print "persons grouped:"
for day in range(DAYS):
    print "day=%d:" % day,
    for group in range(GROUPS):
        print persons_in_group(day, group)+" ",
    print ""
```

```
sat
group for each person:
person:A B C D E F G H I J K L M N O
day=0: 2 2 3 1 0 0 3 4 0 1 1 3 4 4 2
day=1: 2 1 2 4 3 1 4 4 2 1 0 3 0 3 0
day=2: 4 3 1 0 4 0 4 2 2 1 2 3 3 0 1
day=3: 4 3 1 4 2 1 3 1 0 2 3 4 2 0 0
day=4: 3 0 0 1 1 2 4 3 4 3 2 2 4 0 1
day=5: 2 4 1 1 4 3 3 4 0 0 2 0 1 2 3
day=6: 0 2 4 2 4 0 1 3 2 1 4 3 0 1 3

persons grouped:
day=0: EFI  DJK  ABO  CGL  HMN
```

```
day=1: KMO  BFJ  ACI  ELN  DGH
day=2: DFN  CJO  HIK  BLM  AEG
day=3: INO  CFH  EJM  BGK  ADL
day=4: BCN  DEO  FKL  AHJ  GIM
day=5: IJL  CDM  AKN  FGO  BEH
day=6: AFM  GJN  BDI  HLO  CEK
```

It took ≈ 48 seconds on my old Intel Xeon E3-1220 3.10GHz.

I've also tried to represent each number (group in which schoolgirl/golfer is) as a single bit:

```
from z3 import *
import itertools, math

PERSONS, DAYS, GROUPS = 15, 7, 5 # OK
#PERSONS, DAYS, GROUPS = 20, 5, 5 # no answer
#PERSONS, DAYS, GROUPS = 21, 10, 7 # no answer

# each element - group for each person and each day:
tbl=[[BitVec('%d_%d' % (person, day), GROUPS) for day in range(DAYS)] for person in
    range(PERSONS)]

s=Solver()

for person in range(PERSONS):
    for day in range(DAYS):
        s.add(Or(*[tbl[person][day]==(2**i) for i in range(GROUPS)]))

# enumerate all variables
# we add Or(pair1!=0, pair2!=0) constraint, so two non-zero variables couldn't be
    present,
# but both zero variables in pair is OK, one non-zero and one zero variable is also OK:
def only_one_must_be_zero(lst):
    for pair in itertools.combinations(lst, r=2):
        s.add(Or(pair[0]!=0, pair[1]!=0))
    # at least one variable must be zero:
    s.add(Or(*[l==0 for l in lst]))

# get two arrays of variables XORed. one element of this new array must be zero:
def only_one_in_pair_can_be_equal(l1, l2):
    assert len(l1)==len(l2)
    only_one_must_be_zero([l1[i]^l2[i] for i in range(len(l1))])

# enumerate all possible pairs:
for pair in itertools.combinations(range(PERSONS), r=2):
    only_one_in_pair_can_be_equal (tbl[pair[0]], tbl[pair[1]])

print s.check()
m=s.model()

print "group for each person:"
print "person:"+"".join([chr(ord('A')+i)+" " for i in range(PERSONS)])
for day in range(DAYS):
    print "day=%d:" % day,
    for person in range(PERSONS):
        print int(math.log(m[tbl[person][day]].as_long(),2)),
    print ""
```

```
def persons_in_group(day, group):
    rt=""
    for person in range(PERSONS):
        if int(math.log(m[tbl[person][day]].as_long(),2))==group:
            rt=rt+chr(ord('A')+person)
    return rt

print ""
print "persons grouped:"
for day in range(DAYS):
    print "day=%d:" % day,
    for group in range(GROUPS):
        print persons_in_group(day, group)+" ",
    print ""
```

This is way faster, ≈ 1.5 seconds on the same CPU.

Unfortunately, sample SGP[80] are out of reach. Yet? http://www.mathpuzzle.com/MAA/54-Golf%20Tournaments/mathgames_08_14_07.html.

The files: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/SGP/Z3.

## 10.2 School teams scheduling (SAT)

I've found this in the [Dennis E. Shasha – "Puzzles for Programmers and Pros"] book:

> Scheduling Tradition
>
> There are 12 school teams, unimaginatively named A, B, C, D, E, F, G, H, I, J, K, and L. They must play one another on 11 consecutive days on six fields. Every team must play every other team exactly once. Each team plays one game per day.
>
> Warm-Up Suppose there were four teams A, B, C, D and each team has to play every other in three days on two fields. How can you do it?
>
> Solution to Warm-Up
>
> We'll represent the solution in two columns corresponding to the two playing fields. Thus, in the first day, A plays B on field 1 and C plays D on field 2. AB CD AC DB AD BC
>
> Not only does the real problem involve 12 teams instead of merely four, but there are certain constraints due to traditional team rivalries: A must play B on day 1, G on day 3, and H on day 6. F must play I on day 2 and J on day 5. K must play H on day 9 and E on day 11. L must play E on day 8 and B on day 9. H must play I on day 10 and L on day 11. There are no constraints on C or D because these are new teams.
>
> 1. Can you form an 11-day schedule for these teams that satisfies the constraints?
>
> It may seem difficult, but look again at the warm-up. Look in particular at the non-A columns. They are related to one another. If you understand how, you can solve the harder problem.

This is like Kirkman's Schoolgirl Problem I have solved using Z3 before, but this time I've rewritten it as a SAT problem. Also, I added additional constraints relating to "team rivalries".

```
#!/usr/bin/env python3

import SAT_lib
import itertools, math

PERSONS, DAYS, GROUPS = 12, 11, 6

s=SAT_lib.SAT_lib(False)
```

[80] Social Golfer Problem

```
# each element - group for each person and each day:
tbl=[[s.alloc_BV(GROUPS) for day in range(DAYS)] for person in range(PERSONS)]

def chr_to_n (c):
    return ord(c)-ord('A')

# A must play B on day 1, G on day 3, and H on day 6.
# A/B, day 1:
s.fix_BV_EQ(tbl[chr_to_n('A')][0], tbl[chr_to_n('B')][0])
# A/G, day 3:
s.fix_BV_EQ(tbl[chr_to_n('A')][2], tbl[chr_to_n('G')][2])
# A/H, day 5:
s.fix_BV_EQ(tbl[chr_to_n('A')][4], tbl[chr_to_n('H')][4])

# F must play I on day 2 and J on day 5.
s.fix_BV_EQ(tbl[chr_to_n('F')][1], tbl[chr_to_n('I')][1])
s.fix_BV_EQ(tbl[chr_to_n('F')][4], tbl[chr_to_n('J')][4])

# K must play H on day 9 and E on day 11.
s.fix_BV_EQ(tbl[chr_to_n('K')][8], tbl[chr_to_n('H')][8])
s.fix_BV_EQ(tbl[chr_to_n('K')][10], tbl[chr_to_n('E')][10])

# L must play E on day 8 and B on day 9.
s.fix_BV_EQ(tbl[chr_to_n('L')][7], tbl[chr_to_n('E')][7])
s.fix_BV_EQ(tbl[chr_to_n('L')][8], tbl[chr_to_n('B')][8])

# H must play I on day 10 and L on day 11.
s.fix_BV_EQ(tbl[chr_to_n('H')][9], tbl[chr_to_n('I')][9])
s.fix_BV_EQ(tbl[chr_to_n('H')][10], tbl[chr_to_n('L')][10])

for person in range(PERSONS):
    for day in range(DAYS):
        s.make_one_hot(tbl[person][day])

# enumerate all variables
# we add Or(pair1!=0, pair2!=0) constraint, so two non-zero variables couldn't be
   present,
# but both zero variables in pair is OK, one non-zero and one zero variable is also OK:
def only_one_must_be_zero(lst):
    for pair in itertools.combinations(lst, r=2):
        s.OR_always([s.BV_not_zero(pair[0]), s.BV_not_zero(pair[1])])
    # at least one variable must be zero:
    s.OR_always([s.BV_zero(l) for l in lst])

# get two arrays of variables XORed. one element of this new array must be zero:
def only_one_in_pair_can_be_equal(l1, l2):
    assert len(l1)==len(l2)
    only_one_must_be_zero([s.BV_XOR(l1[i], l2[i]) for i in range(len(l1))])

# enumerate all possible pairs:
for pair in itertools.combinations(range(PERSONS), r=2):
    only_one_in_pair_can_be_equal (tbl[pair[0]], tbl[pair[1]])

assert s.solve()
```

```
print ("group for each person:")
print ("person: "+"".join([chr(ord('A')+i)+" " for i in range(PERSONS)]))
for day in range(DAYS):
    t=("day=%2d: " % day)
    for person in range(PERSONS):
        t=t+str(int(math.log(s.get_val_from_solution(tbl[person][day]),2)))+" "
    print (t)

def persons_in_group(day, group):
    rt=""
    for person in range(PERSONS):
        if int(math.log(s.get_val_from_solution(tbl[person][day]),2))==group:
            rt=rt+chr(ord('A')+person)
    return rt

print ("")
print ("persons grouped:")
for day in range(DAYS):
    t=("day=%2d: " % day)
    for group in range(GROUPS):
        t=t+persons_in_group(day, group)+" "
    print (t)
```

The solution:

```
group for each person:
person: A B C D E F G H I J K L
day= 0: 4 4 1 3 5 0 2 0 5 3 2 1
day= 1: 5 0 3 3 2 4 1 2 4 1 0 5
day= 2: 4 5 1 0 1 2 4 5 3 3 2 0
day= 3: 3 5 4 1 1 4 2 2 0 5 3 0
day= 4: 3 0 4 5 0 2 5 3 4 2 1 1
day= 5: 3 5 4 5 3 0 0 4 1 2 1 2
day= 6: 5 3 5 4 1 0 1 4 3 2 2 0
day= 7: 5 2 0 1 3 1 2 4 5 4 0 3
day= 8: 4 2 5 4 1 1 3 0 3 5 0 2
day= 9: 4 2 2 1 5 4 0 3 3 5 1 0
day=10: 2 5 0 4 3 5 0 1 4 2 3 1

persons grouped:
day= 0: FH  CL  GK  DJ  AB  EI
day= 1: BK  GJ  EH  CD  FI  AL
day= 2: DL  CE  FK  IJ  AG  BH
day= 3: IL  DE  GH  AK  CF  BJ
day= 4: BE  KL  FJ  AH  CI  DG
day= 5: FG  IK  JL  AE  CH  BD
day= 6: FL  EG  JK  BI  DH  AC
day= 7: CK  DF  BG  EL  HJ  AI
day= 8: HK  EF  BL  GI  AD  CJ
day= 9: GL  DK  BC  HI  AF  EJ
day=10: CG  HL  AJ  EK  DI  BF
```

("Person" and "team" terms are interchangeable in my code.)

Thanks to parallel lingeling SAT solver[81] I've used this time, it takes couple of minutes on a decent 4-core CPU.

The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/SGP/SAT.

[81] http://fmv.jku.at/lingeling/

## 10.3 Latin squares

Magic/Latin square is a square filled with numbers/letters, which are all distinct in each row and column. Sudoku is 9*9 magic square with additional constraints (for each 3*3 subsquare).

### 10.3.1 Magic/Latin square of Knut Vik design (Z3Py)

"Knut Vik design" is a square, where all (broken) diagonals has distinct numbers.

This is diagonal of 5*5 square:

```
. . . . *
. . . * .
. . * . .
. * . . .
* . . . .
```

These are broken diagonals:

```
. . * . .
. . . * .
. . . . *
* . . . .
. * . . .
```

```
* . . . .
. . . . *
. . . * .
. . * . .
. * . . .
```

I could only find 5*5 and 7*7 squares using Z3, couldn't find 11*11 square, however, it's possible to prove there are no 6*6 and 4*4 squares (such squares doesn't exist if size is divisible by 2 or 3).

```
from z3 import *

#SIZE=4 # unsat
#SIZE=5 # OK
#SIZE=6 # unsat
SIZE=7 # OK

a=[[Int('%d_%d' % (r,c)) for c in range(SIZE)] for r in range(SIZE)]

s=Solver()

# all numbers must be in 1..SIZE limits
for r in range(SIZE):
    for c in range(SIZE):
        s.add(And(a[r][c]>=1, a[r][c]<=SIZE))

# all numbers in all rows must be distinct:
for r in range(SIZE):
    # expression like s.add(Distinct(a[r][0], a[r][1], ..., a[r][last])) is formed here:
    s.add(Distinct(*[a[r][c] for c in range(SIZE)]))

# ... in all columns as well:
for c in range(SIZE):
    s.add(Distinct(*[a[r][c] for r in range(SIZE)]))

# all (broken) diagonals must also be distinct:
```

```
for r in range(SIZE):
    s.add(Distinct(*[a[(r+r2) % SIZE][r2 % SIZE] for r2 in range(SIZE)]))
    # this line of code is the same as previous, but the column is "flipped"
       horizontally (SIZE-1-column):
    s.add(Distinct(*[a[(r+r2) % SIZE][SIZE-1-(r2 % SIZE)] for r2 in range(SIZE)]))

print s.check()
m=s.model()

for r in range(SIZE):
    for c in range(SIZE):
        print m[a[r][c]].as_long(),
    print ""
```

5*5 Knut Vik square:

```
3 4 5 1 2
5 1 2 3 4
2 3 4 5 1
4 5 1 2 3
1 2 3 4 5
```

7*7:

```
4 7 6 5 1 2 3
6 5 1 2 3 4 7
1 2 3 4 7 6 5
3 4 7 6 5 1 2
7 6 5 1 2 3 4
5 1 2 3 4 7 6
2 3 4 7 6 5 1
```

This is a good example of NP-problem: you can check the result visually, but it takes several seconds for computer to find it.

We can also use different encoding: each number can be represented by one bit. 0b0001 for 1, 0b0010 for 2, 0b1000 for 4, etc. Then a "Distinct" operator can be replaced by OR operation and comparison against mask with all bits present.

```
import math, operator
from z3 import *

#SIZE=4 # unsat
SIZE=5 # OK
#SIZE=6 # unsat
#SIZE=7 # OK
#SIZE=11 # no answer

a=[[BitVec('%d_%d' % (r,c), SIZE) for c in range(SIZE)] for r in range(SIZE)]

# 0b11111 for SIZE=5:
mask=2**SIZE-1

s=Solver()

# all numbers must have form 2^n, like 1, 2, 4, 8, 16, etc.
# we add contraint like Or(a[r][c]==2, a[r][c]==4, ..., a[r][c]==32, ...)
for r in range(SIZE):
    for c in range(SIZE):
        s.add(Or(*[a[r][c]==(2**i) for i in range(SIZE)]))

```

```
# all numbers in all rows must be distinct:
for r in range(SIZE):
    # expression like s,add(a[r][0] | a[r][1] | ... | a[r][last] == mask) is formed here
        :
    s.add(reduce(operator.ior, [a[r][c] for c in range(SIZE)])==mask)

# ... in all columns as well:
for c in range(SIZE):
    # expression like s,add(a[0][c] | a[1][c] | ... | a[last][c] == mask) is formed here
        :
    s.add(reduce(operator.ior, [a[r][c] for r in range(SIZE)])==mask)

# for all (broken) diagonals:
for r in range(SIZE):
    s.add(reduce(operator.ior, [a[(r+r2) % SIZE][r2 % SIZE] for r2 in range(SIZE)])==
        mask)
    # this line of code is the same as previous, but the column is "flipped"
        horizontally (SIZE-1-column):
    s.add(reduce(operator.ior, [a[(r+r2) % SIZE][SIZE-1-(r2 % SIZE)] for r2 in range(
        SIZE)])==mask)

print s.check()
m=s.model()

for r in range(SIZE):
    for c in range(SIZE):
        print int(math.log(m[a[r][c]].as_long(),2)),
    print ""
```

That works twice as faster (however, numbers are in 0..SIZE-1 range instead of 1..SIZE, but you've got the idea).

Besides recreational mathematics, Knut Vik squares like these are very important in design of experiments.

Further reading:

- A. Hedayat and W, T. Federer - On the Nonexistence of Knut Vik Designs for all Even Orders (1973)
- A. Hedayat - A Complete Solution to the Existence and Nonexistence of Knut Vik Designs and Orthogonal Knut Vik Designs (1975)

# 11 Cyclic redundancy check

## 11.1 Yet another explanation of CRC

### 11.1.1 What is wrong with checksum?

If you just sum up values of several bytes, two bit flips (increment one bit and decrement another bit) can give the same checksum. No good.

### 11.1.2 Division by prime

You can represent a file of buffer as a (big) number, then to divide it by prime. The remainder is then very sensitive to bit flips. For example, a prime 0x10015 (65557).

Wolfram Mathematica:

```
In[]:= divisor=16^^10015
Out[]= 65557
```

```
In[]:= BaseForm[Mod[16^^abcdef1234567890, divisor],16]
Out[]= d8c1

In[]:= BaseForm[Mod[16^^abcdef0234567890, divisor],16]
Out[]= bd31

In[]:= BaseForm[Mod[16^^bbcdef1234567890, divisor],16]
Out[]= 382b

In[]:= BaseForm[Mod[16^^abcdee1234567890, divisor],16]
Out[]= 1fd6

In[]:= BaseForm[Mod[16^^abcdef0234567891, divisor],16]
Out[]= bd32
```

This is what is called "avalanche effect" in cryptography: one bit flip of input can affect many bits of output. Go figure out which bits must be also flipped to preserve specific remainder.

You can build such a divisor in hardware, but it would require at least one adder or subtractor, you will have a carry-ripple problem in simple case, or you would have to create more complicated circuit.

#### 11.1.3 (Binary) long divison

Binary long division is in fact simpler then the paper-n-pencil algorithm taught in schools.

The algorithm is:

- 1) Allocate some "tmp" variable and copy dividend to it.
- 2) Pad divisor by zero bits at left so that MSB of divisor is at the place of MSB of the value in tmp.
- 3) If the divisor is larger than tmp or equal, subtract divider from tmp and add 1 bit to the quotient. If the divisor is smaller than tmp, add 0 bit to the quotient.
- 4) Shift divisor right. If the divisor is 0, stop. Remainder is in tmp.
- 5) Goto 3

The following piece of code I've copypasted from somewhere:

```
unsigned int divide(unsigned int dividend, unsigned int divisor)
{
        unsigned int tmp = dividend;
        unsigned int denom = divisor;
        unsigned int current = 1;
        unsigned int answer = 0;

        if (denom > tmp)
                return 0;

        if (denom == tmp)
                return 1;

        // align divisor:
        while (denom <= tmp)
        {
                denom = denom << 1;
                current = current << 1;
        }

        denom = denom >> 1;
```

```
	current = current >> 1;

	while (current!=0)
	{
		printf ("current=%d, denom=%d\n", current, denom);
		if (tmp >= denom)
		{
			tmp -= denom;
			answer |= current;
		}
		current = current >> 1;
		denom = denom >> 1;
	}
	printf ("tmp/remainder=%d\n", tmp); // remainder!
	return answer;
}
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/CRC/explanation/div.c )

Let's divide 1234567 by 813 and find remainder:

```
current=1024, denom=832512
current=512, denom=416256
current=256, denom=208128
current=128, denom=104064
current=64, denom=52032
current=32, denom=26016
current=16, denom=13008
current=8, denom=6504
current=4, denom=3252
current=2, denom=1626
current=1, denom=813
tmp/remainder=433
1518
```

#### 11.1.4 (Binary) long division, version 2

Now let's say, you only need to compute a remainder, and throw away a quotient. Also, maybe you work on some kind BigInt values and you've got a function like `get_next_bit()` and that's it.

What we can do: tmp value will be shifted at each iteration, while divisor is not:

```
uint8_t *buf;
int buf_pos;
int buf_bit_pos;

int get_bit()
{
	if (buf_pos==-1)
		return -1; // end

	int rt=(buf[buf_pos] >> buf_bit_pos) & 1;
	if (buf_bit_pos==0)
	{
		buf_pos--;
		buf_bit_pos=7;
	}
	else
		buf_bit_pos--;
```

```
        return rt;
};

uint32_t remainder_arith(uint32_t dividend, uint32_t divisor)
{
        buf=(uint8_t*)&dividend;
        buf_pos=3;
        buf_bit_pos=7;

        uint32_t tmp=0;

        for(;;)
        {
                int bit=get_bit();
                if (bit==-1)
                {
                        printf ("exit. remainder=%d\n", tmp);
                        return tmp;
                };

                tmp=tmp<<1;
                tmp=tmp|bit;

                if (tmp>=divisor)
                {
                        printf ("%d greater or equal to %d\n", tmp, divisor);
                        tmp=tmp-divisor;
                        printf ("new tmp=%d\n", tmp);
                }
                else
                        printf ("tmp=%d, can't subtract\n", tmp);
        };
}
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/CRC/explanation/div_both.c )

Let's divide 1234567 by 813 and find remainder:

```
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=1, can't subtract
tmp=2, can't subtract
tmp=4, can't subtract
tmp=9, can't subtract
tmp=18, can't subtract
tmp=37, can't subtract
tmp=75, can't subtract
tmp=150, can't subtract
tmp=301, can't subtract
```

```
tmp=602, can't subtract
1205 greater or equal to 813
new tmp=392
tmp=785, can't subtract
1570 greater or equal to 813
new tmp=757
1515 greater or equal to 813
new tmp=702
1404 greater or equal to 813
new tmp=591
1182 greater or equal to 813
new tmp=369
tmp=738, can't subtract
1476 greater or equal to 813
new tmp=663
1327 greater or equal to 813
new tmp=514
1029 greater or equal to 813
new tmp=216
tmp=433, can't subtract
exit. remainder=433
```

### 11.1.5 Shortest possible introduction into GF(2)

There is a difference between digit and number. Digit is a symbol, number is a group of digits. 0 can be both digit and number.

Binary digits are 0 and 1, but a binary number can be any.

There are just two numbers in Galois Field (2): 0 and 1. No other numbers.

What practical would you do with just two numbers? Not much, but you can pack GF(2) numbers into some kind of structure or tuple or even array. Such structures are represented using polynomials. For example, CRC32 polynomial you can find in source code is 0x04C11DB7. Each bit represent a number in GF(2), not a digit. The 0x04C11DB7 polynomial is written as:

$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

Wherever $x^n$ is present, that means, you have a bit at position $n$. Just $x$ means, bit present at LSB. There is, however, bit at $x^{32}$, so the CRC32 polynomial has the size of 33 bits, bit the MSB is always 1 and is omitted in all algorithms.

It's important to say that unlike in algebra, GF(2) polynomials are never evaluated here. $x$ is symbol is present mereley as a convention. People represent GF(2) "structures" as polynomials to emphasize the fact that "numbers" are isolated from each other.

Now, subtraction and addition are the same operations in GF(2) and actually works as XOR. This is present in many tutorials, so I'll omit this here.

Also, by convention, whenever you compare two numbers in GF(2), you only compare two most significant bits, and ignore the rest.

### 11.1.6 CRC32

Now we can take the binary division algorithm and change it a little:

```
uint32_t remainder_GF2(uint32_t dividend, uint32_t divisor)
{
        // necessary bit shuffling/negation to make it compatible with other CRC32
           implementations.
        // N.B.: input data is not an array, but a 32-bit integer, hence we need to swap
            endiannes.
        uint32_t dividend_negated_swapped = ~swap_endianness32(bitrev32(dividend));
        buf=(uint8_t*)&dividend_negated_swapped;
        buf_pos=3;
        buf_bit_pos=7;
```

```
        uint32_t tmp=0;

        // process 32 bits from the input + 32 zero bits:
        for(int i=0; i<32+32; i++)
        {
                int bit=get_bit();
                int shifted_bit=tmp>>31;

                // fetch next bit:
                tmp=tmp<<1;
                if (bit==-1)
                {
                        // no more bits, but continue, we fetch 32 more zero bits.
                        // shift left operation set leftmost bit to zero.
                }
                else
                {
                        // append next bit at right:
                        tmp=tmp|bit;
                };

                // at this point, tmp variable/value has 33 bits: shifted_bit + tmp
                // now take the most significant bit (33th) and test it:
                // 33th bit of polynomial (not present in "divisor" variable is always 1
                // so we have to only check shifted_bit value
                if (shifted_bit)
                {
                        // use only 32 bits of polynomial, ingore 33th bit, which is
                            always 1:
                        tmp=tmp^divisor;
                };
        };
        // bit shuffling/negation for compatibility once again:
        return ~bitrev32(tmp);
}
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/CRC/explanation/div_both.c )

And voila, this is the function which computes CRC32 for the input 32-bit value.

There are only 3 significant changes:

- 1) XOR instead of minus.
- 2) Only MSB is checked during comparison. But the MSB of all CRC polynomials is always 1, so we only need to check MSB (33th bit) of the tmp variable.
- 3) There are 32+32=64 iterations instead of 32. As you can see, only MSB of tmp affects the whole behaviour of the algorithm. So when tmp variable is filled by 32 bits which never affected anything so far, we need to ”blow out” all these bits through 33th bit of tmp variable to get correct remainder (or CRC32 sum).

All the rest algorithms you can find on the Internet are optimized version, which may be harder to understand. No algorithms used in practice “blows” anything “out” due to optimization. Many practical algorithms are either bytewise (process input stream by bytes, not by bits) or table-based.

My goal was to write two functions, as similar to each other as possible, to demonstrate the difference.

So the CRC value is in fact remainder of division of input date by CRC polynomial in GF(2) environment. As simple as that.

### 11.1.7 Rationale

Why would anyone use such an unusual mathematics? The answer is: many GF(2) operations can be done using bit shifts and XOR, which are very cheap operations.

Electronic circuit for CRC generator is extremely simple, it consists of only shift register and XOR gates. This one is for CRC16:

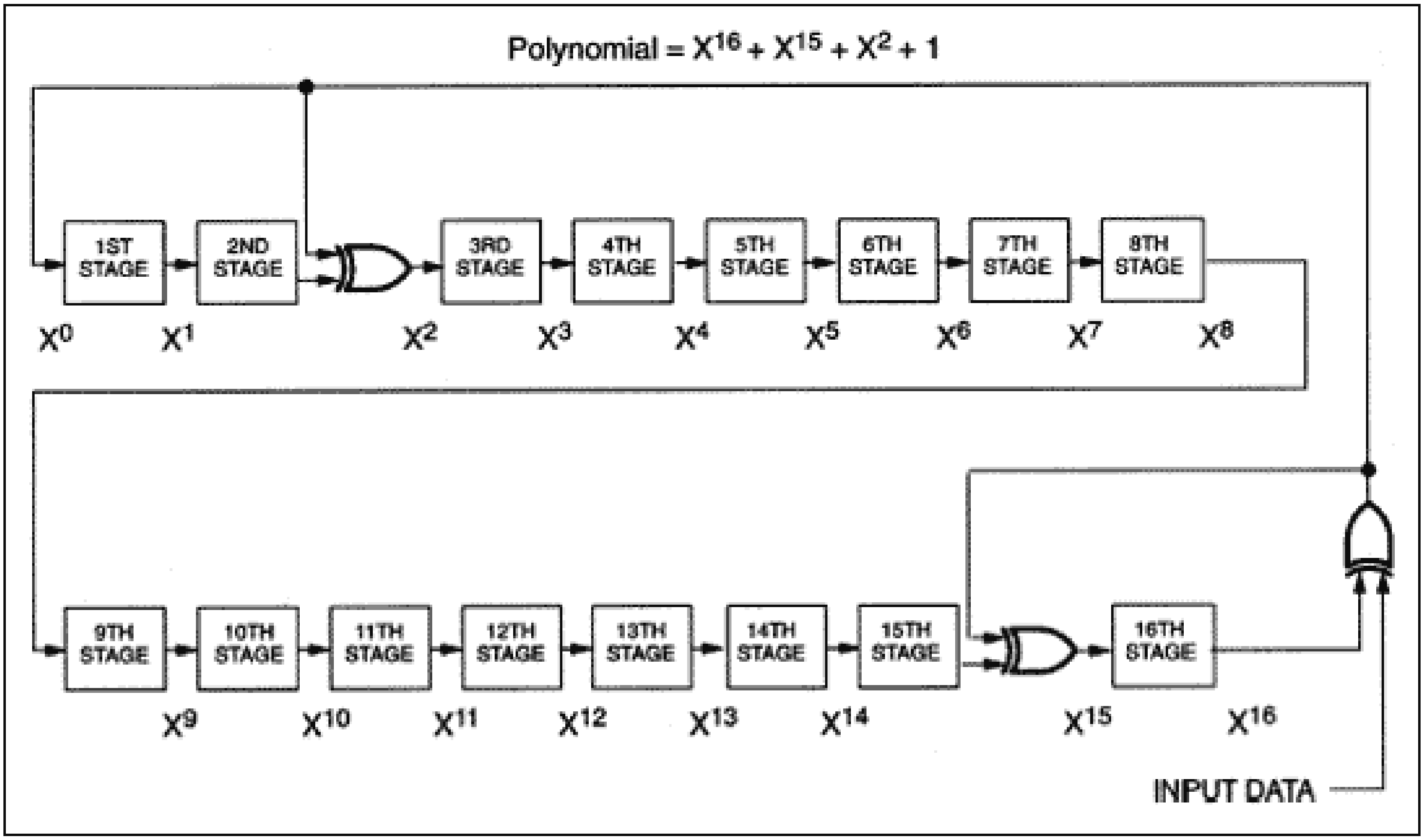



Figure 30:

( The source of image: https://olimex.wordpress.com/2014/01/10/weekend-programming-challenge-week-39-crc-16/ )

Only 3 XOR gates are present aside of shift register.

The following page has animation: https://en.wikipedia.org/wiki/Computation_of_cyclic_redundancy_checks.

It can be implemented maybe even using vacuum tubes.

And the task is not to compute remainder according to rules of arithmetics, but rather to detect errors.

Compare this to a division circuit with at least one binary adder/subtractor, which will have carry-ripple problem. On the other hand, addition over GF(2) has no carries, hence, this problem absent.

### 11.1.8 Further reading

These documents I've found interesting/helpful:

- http://www.ross.net/crc/download/crc_v3.txt
- https://www.kernel.org/doc/Documentation/crc32.txt
- http://web.archive.org/web/20161220015646/http://www.hackersdelight.org/crc.pdf

## 11.2 Factorize GF(2)/CRC polynomials

GF(2)/CRC polynomials, like usual numbers, can also be factored, because a polynomial can be a product of two other polynomial (or not).

Some people say that good CRC polynomial should be irreducible (i.e., cannot be factored), some other say that this is not a requirement. I've checked several CRC-16 and CRC-32 polynomials from the Wikipedia article.

The multiplier is constructed in the same manner, as I did it earlier for integer factorization using SAT. Factors are not prime integers, but prime polynomials.

Another important thing to notice is that replacing XOR with addition will make this script factor integers, because addition in GF(2) is XOR.

Also, can be used for tests, online GF(2) polynomials factorization: `http://www.ee.unb.ca/cgi-bin/tervo/factor.pl?binary=101`.

```
import operator
from z3 import *

INPUT_SIZE=32
OUTPUT_SIZE=INPUT_SIZE*2

a=BitVec('a', INPUT_SIZE)
b=BitVec('b', INPUT_SIZE)

"""
rows with dots are partial products:

     aaaa
b    ....
b   ....
b  ....
b ....

"""

# partial products
p=[BitVec('p_%d' % i, OUTPUT_SIZE) for i in range(INPUT_SIZE)]

s=Solver()

for i in range(INPUT_SIZE):
    # if there is a bit in b[], assign shifted a[] padded with zeroes at left/right
    # if there is no bit in b[], let p[] be zero

    # Concat() is for glueling together bitvectors (of different widths)
    # BitVecVal() is constant of specific width

    if i==0:
        s.add(p[i] == If((b>>i)&1==1, Concat(BitVecVal(0, OUTPUT_SIZE-i-INPUT_SIZE), a),
             0))
    else:
        s.add(p[i] == If((b>>i)&1==1, Concat(BitVecVal(0, OUTPUT_SIZE-i-INPUT_SIZE), a,
            BitVecVal(0, i)), 0))

# tests

# from http://mathworld.wolfram.com/IrreduciblePolynomial.html
#poly=7 # irreducible
#poly=5 # reducible
```

```
# from Colbourn, Dinitz - Handbook of Combinatorial Designs (2ed, 2007), p.809:
#poly=0b10000001001 # irreducible
#poly=0b10000001111 # irreducible

# MSB is always 1 in CRC polynomials, and it's omitted
# but we add it here (leading 1 bit):
poly=0x18005 # CRC-16-IBM, reducible
#poly=0x11021 # CRC-16-CCITT, reducible
#poly=0x1C867 # CRC-16-CDMA2000, irreducible
#poly=0x104c11db7 # CRC-32, irreducible
#poly=0x11EDC6F41 # CRC-32C (Castagnoli), CRC32 x86 instruction, reducible
#poly=0x1741B8CD7 # CRC-32K (Koopman {1,3,28}), reducible
#poly=0x132583499 # CRC-32K2 (Koopman {1,1,30}), reducible
#poly=0x1814141AB # CRC-32Q, reducible

# form expression like s.add(p[0] ^ p[1] ^ ... ^ p[OUTPUT_SIZE-1] == poly)
# replace operator.xor to operator.add to factorize numbers:
s.add(reduce (operator.xor, p)==poly)

# we are not interesting in outputs like these:
s.add(a!=1)
s.add(b!=1)

if s.check()==unsat:
    print "unsat"
    exit(0)

m=s.model()
print "sat, a=0x%x, b=0x%x" % (m[a].as_long(), m[b].as_long())
```

## 11.3 Getting CRC polynomial and other CRC generator parameters

Sometimes CRC implementations are incompatible with each other: polynomial and other parameters can be different. Aside of polynomial, initial state can be either 0 or -1, final value can be inverted or not, endianness of the final value can be changed or not. Trying all these parameters by hand to match with someone's else implementation can be a real pain. Also, you can bruteforce 32-bit polynomial, but 64-bit polynomials is too much.

Deducing all these parameters is surprisingly simple using Z3, just get two values for 01 byte and 02, or any other bytes.

```
#!/usr/bin/env python

from z3 import *
import struct

# knobs:

# CRC-16 on https://www.lammertbies.nl/comm/info/crc-calculation.html
#width=16
#samples=["\x01", "\x02"]
#must_be=[0xC0C1, 0xC181]
#sample_len=1

# CRC-16 (Modbus) on https://www.lammertbies.nl/comm/info/crc-calculation.html
#width=16
#samples=["\x01", "\x02"]
```

```
#must_be=[0x807E, 0x813E]
#sample_len=1

# CRC-16-CCITT, Kermit on https://www.lammertbies.nl/comm/info/crc-calculation.html
#width=16
#samples=["\x01", "\x02"]
#must_be=[0x8911, 0x1223]
#sample_len=1

width=32
samples=["\x01", "\x02"]
must_be=[0xA505DF1B, 0x3C0C8EA1]
sample_len=1

# crc64_1.c:
#width=64
#samples=["\x01", "\x02"]
#must_be=[0x28d250b0f0900abe, 0x6c9fd98969f81a9d]
#sample_len=1

# crc64_2.c (redis):
#width=64
#samples=["\x01", "\x02"]
#must_be=[0x7ad870c830358979, 0xf5b0e190606b12f2]
#sample_len=1

# crc64_3.c:
#width=64
#samples=["\x01", "\x02"]
#must_be=[0xb32e4cbe03a75f6f, 0xf4843657a840a05b]
#sample_len=1

# http://www.unit-conversion.info/texttools/crc/
#width=32
#samples=["0","1"]
#must_be=[0xf4dbdf21, 0x83dcefb7]
#sample_len=1

# recipe-259177-1.py, CRC-64-ISO
#width=64
#samples=["\x01", "\x02"]
#must_be=[0x01B0000000000000, 0x0360000000000000]
#sample_len=1

# recipe-259177-1.py, CRC-64-ISO
#width=64
#samples=["\x01"]
#must_be=[0x01B0000000000000]
#sample_len=1

#width=32
#samples=["12","ab"]
#must_be=[0x4F5344CD, 0x9E83486D]
#sample_len=2

def swap_endianness_16(val):
```

```
    return struct.unpack("<H", struct.pack(">H", val))[0]

def swap_endianness_32(val):
    return struct.unpack("<I", struct.pack(">I", val))[0]

def swap_endianness_64(val):
    return struct.unpack("<Q", struct.pack(">Q", val))[0]

def swap_endianness(width, val):
    if width==64:
        return swap_endianness_64(val)
    if width==32:
        return swap_endianness_32(val)
    if width==16:
        return swap_endianness_16(val)
    raise AssertionError

mask=2**width-1
poly=BitVec('poly', width)

# states[sample][0][8] is an initial state
# ...
# states[sample][i][0] is a state where it was already XORed with input bit
# states[sample][i][1] ... where the 1th shift/XOR operation has been done
# states[sample][i][8] ... where the 8th shift/XOR operation has been done
# ...
# states[sample][sample_len][8] - final state

states=[[[BitVec('state_%d_%d_%d' % (sample, i, bit), width) for bit in range(8+1)] for
   i in range(sample_len+1)] for sample in range(len(samples))]
s=Solver()

def invert(val):
    return ~val & mask

for sample in range(len(samples)):
    # initial state can be either zero or -1:
    s.add(Or(states[sample][0][8]==mask, states[sample][0][8]==0))

    # implement basic CRC algorithm
    for i in range(sample_len):
        s.add(states[sample][i+1][0] == states[sample][i][8] ^ ord(samples[sample][i]))

        for bit in range(8):
            # LShR() is logical shift, while >> is arithmetical shift, we use the first:
            s.add(states[sample][i+1][bit+1] == LShR(states[sample][i+1][bit],1) ^ If(
               states[sample][i+1][bit]&1==1, poly, 0))

    # final state must be equial to one of these:
    s.add(Or(
        states[sample][sample_len][8]==must_be[sample],
        states[sample][sample_len][8]==invert(must_be[sample]),
        states[sample][sample_len][8]==swap_endianness(width, must_be[sample]),
        states[sample][sample_len][8]==invert(swap_endianness(width, must_be[sample]))))

# get all possible results:
```

```
results=[]
while True:
    if s.check() == sat:
        m = s.model()
        # what final state was?
        if m[states[0][sample_len][8]].as_long()==must_be[0]:
            outparams="XORout=0"
        elif invert(m[states[0][sample_len][8]].as_long())==must_be[0]:
            outparams="XORout=-1"
        elif m[states[0][sample_len][8]].as_long()==swap_endianness(width, must_be[0]):
            outparams="XORout==0, ReflectOut=true"
        elif invert(m[states[0][sample_len][8]].as_long())==swap_endianness(width,
           must_be[0]):
            outparams="XORout=-1, ReflectOut=true"
        else:
            raise AssertionError

        print "poly=0x%x, init=0x%x, %s" % (m[poly].as_long(), m[states[0][0][8]].
           as_long(), outparams)

        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "total results", len(results)
        break
```

This is for CRC-16:

```
poly=0xa001, init=0x0, XORout=0
```

Sometimes, we have no enough information, but still can get something. This is for CRC-16-CCITT:

```
poly=0xb30f, init=0x0, XORout=-1
poly=0x7c07, init=0x0, XORout==0, ReflectOut=true
poly=0x8408, init=0x0, XORout==0, ReflectOut=true
```

One of these results is correct.

We can get something even if we have only one result for one input byte:

```
# recipe-259177-1.py, CRC-64-ISO
width=64
samples=["\x01"]
must_be=[0x01B0000000000000]
sample_len=1
```

```
poly=0x1fb12, init=0x0, XORout==0, ReflectOut=true
poly=0x1d24924924924924, init=0xffffffffffffffff, XORout=0
poly=0x86a9466cbb890d53, init=0x0, XORout=-1, ReflectOut=true
poly=0x580080, init=0x0, XORout==0, ReflectOut=true
poly=0xce9ce, init=0x0, XORout==0, ReflectOut=true
poly=0x53ffffffffffffff, init=0xffffffffffffffff, XORout=0
poly=0xd800000000000000, init=0x0, XORout=0
poly=0x38ad6, init=0x0, XORout==0, ReflectOut=true
poly=0x131e56e82623cae, init=0xffffffffffffffff, XORout==0, ReflectOut=true
```

```
poly=0x3ffffffffd3ffbf, init=0xffffffffffffffff, XORout==0, ReflectOut=true
poly=0x461861861861861, init=0xffffffffffffffff, XORout=0
total results 11
```

The files: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/CRC/cracker.

The shortcoming: longer samples slows down everything significantly. I had luck with samples up to 4 bytes, but no larger.

Further reading I've found interesting/helpful:

- http://www.cosc.canterbury.ac.nz/greg.ewing/essays/CRC-Reverse-Engineering.html
- http://reveng.sourceforge.net/crc-catalogue/1-15.htm
- http://reveng.sourceforge.net/crc-catalogue/16.htm
- http://reveng.sourceforge.net/crc-catalogue/17plus.htm

## 11.4 Finding (good) CRC polynomial

Finding good CRC polynomial is tricky, and my results can't compete with other tested popular CRC polynomial. Nevertheless, it was fun to use Z3 to find them.

I just generate 32 random samples, all has size between 1 and 32 bytes. Then I flip 1..3 random bits and I add a constraint: CRC hash of the sample and hash of the modified sample (with 1..3 bits flipped) must differ.

```
#!/usr/bin/env python

from z3 import *
import copy, random

width=32

poly=BitVec('poly', width)

s=Solver()

no_call=0

def CRC(_input, poly):
    # make each variable name unique
    # no_call (number of call) increments at each call to CRC() function
    global no_call
    states=[[BitVec('state_%d_%d_%d' % (no_call, i, bit), width) for bit in range(8+1)]
        for i in range(len(_input)+1)]
    no_call=no_call+1
    # initial state is always 0:
    s.add(states[0][8]==0)

    for i in range(len(_input)):
        s.add(states[i+1][0] == states[i][8] ^ _input[i])

        for bit in range(8):
            s.add(states[i+1][bit+1] == LShR(states[i+1][bit],1) ^ If(states[i+1][bit
                ]&1==1, poly, 0))

    return states[len(_input)][8]

# generate 32 random samples:
for i in range(32):
    print "pair",i
```

```
    # each sample has random size 1..32
    buf1=bytearray(os.urandom(random.randrange(32)+1))
    buf2=copy.deepcopy(buf1)
    # flip 1, 2 or 3 random bits in second sample:
    for bits in range(1,random.randrange(3)+2):
        # get random position and bit to flip:
        pos=random.randrange(0, len(buf2))
        to_flip=1<<random.randrange(8)
        print "  pos=", pos, "bit=",to_flip
        # flip random bit at random position:
        buf2[pos]=buf2[pos]^to_flip

    # original sample and sample with 1..3 random bits flipped.
    # their hashes must be different:
    s.add(CRC(buf1, poly)!=CRC(buf2, poly))

# get all possible results:
results=[]
while True:
    if s.check() == sat:
        m = s.model()
        print "poly=0x%x" % (m[poly].as_long())
        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "total results", len(results)
        break
```

Several polynomials for CRC8:

```
poly=0xf9
poly=0x50
poly=0x90
...
```

... for CRC16:

```
poly=0xf7af
poly=0x368
poly=0x268
poly=0x228
...
```

... for CRC32:

```
poly=0x1683a5ab
poly=0x78553eda
poly=0x7a153eda
poly=0x7b353eda
...
```

... for CRC64:

```
poly=0x8000000000000006
poly=0x926b19b536a62f10
```

```
poly=0x4a7bb0a7da78a370
poly=0xbbc781e7e83dabf0
...
```

Problem: at least this one. CRC must be able to detect errors in very long buffers, up to $2^{32}$ for CRC32. We can't feed that huge buffers to SMT solver. I had success only with samples up to $\approx 32$ bytes.

## 11.5 CRC (Cyclic redundancy check)

### 11.5.1 Buffer alteration case #1

Sometimes, you need to alter a piece of data which is *protected* by some kind of checksum or CRC, and you can't change checksum or CRC value, but can alter piece of data so that checksum will remain the same.

Let's pretend, we've got a piece of data with "Hello, world!" string at the beginning and "and goodbye" string at the end. We can alter 14 characters at the middle, but for some reason, they must be in *a..z* limits, but we can put any characters there. CRC64 of the whole block must be `0x12345678abcdef12`.

Let's see[82]:

```
#include <string.h>
#include <stdint.h>

uint64_t crc64(uint64_t crc, unsigned char *buf, int len)
{
        int k;

        crc = ~crc;
        while (len--)
        {
                crc ^= *buf++;
                for (k = 0; k < 8; k++)
                        crc = crc & 1 ? (crc >> 1) ^ 0x42f0e1eba9ea3693 : crc >> 1;
        }
        return crc;
}

int main()
{
#define HEAD_STR "Hello, world!.. "
#define HEAD_SIZE strlen(HEAD_STR)
#define TAIL_STR " ... and goodbye"
#define TAIL_SIZE strlen(TAIL_STR)
#define MID_SIZE 14 // work
#define BUF_SIZE HEAD_SIZE+TAIL_SIZE+MID_SIZE

        char buf[BUF_SIZE];

        klee_make_symbolic(buf, sizeof buf, "buf");

        klee_assume (memcmp (buf, HEAD_STR, HEAD_SIZE)==0);

        for (int i=0; i<MID_SIZE; i++)
                klee_assume (buf[HEAD_SIZE+i]>='a' && buf[HEAD_SIZE+i]<='z');

        klee_assume (memcmp (buf+HEAD_SIZE+MID_SIZE, TAIL_STR, TAIL_SIZE)==0);

        klee_assume (crc64 (0, buf, BUF_SIZE)==0x12345678abcdef12);
```

[82]There are several slightly different CRC64 implementations, the one I use here can also be different from popular ones.

```
        klee_assert(0);

        return 0;
}
```

Since our code uses memcmp() standard C/C++ function, we need to add `--libc=uclibc` switch, so KLEE will use its own uClibc implementation.

```
% clang -emit-llvm -c -g klee_CRC64.c

% time klee --libc=uclibc klee_CRC64.bc
```

It takes about 1 minute (on my Intel Core i3-3110M 2.4GHz notebook) and we getting this:

```
...
real    0m52.643s
user    0m51.232s
sys     0m0.239s
...
% ls klee-last | grep err
test000001.user.err
test000002.user.err
test000003.user.err
test000004.external.err

% ktest-tool --write-ints klee-last/test000004.ktest
ktest file : 'klee-last/test000004.ktest'
args       : ['klee_CRC64.bc']
num objects: 1
object    0: name: b'buf'
object    0: size: 46
object    0: data: b'Hello, world!.. qqlicayzceamyw ... and goodbye'
```

Maybe it's slow, but definitely faster than bruteforce. Indeed, $log_2 26^{14} \approx 65.8$ which is close to 64 bits. In other words, one need $\approx$ 14 latin characters to encode 64 bits. And KLEE + SMT solver needs 64 bits at some place it can alter to make final CRC64 value equal to what we defined.

I tried to reduce length of the *middle block* to 13 characters: no luck for KLEE then, it has no space enough.

### 11.5.2 Buffer alteration case #2

I went sadistic: what if the buffer must contain the CRC64 value which, after calculation of CRC64, will result in the same value? Fascinatedly, KLEE can solve this. The buffer will have the following format:

```
Hello, world! <8 bytes (64-bit value)> and goodbye <6 more bytes>
```

```
int main()
{
#define HEAD_STR "Hello, world!.. "
#define HEAD_SIZE strlen(HEAD_STR)
#define TAIL_STR " ... and goodbye"
#define TAIL_SIZE strlen(TAIL_STR)
// 8 bytes for 64-bit value:
#define MID_SIZE 8
#define BUF_SIZE HEAD_SIZE+TAIL_SIZE+MID_SIZE+6

        char buf[BUF_SIZE];

        klee_make_symbolic(buf, sizeof buf, "buf");
```

```
        klee_assume (memcmp (buf, HEAD_STR, HEAD_SIZE)==0);

        klee_assume (memcmp (buf+HEAD_SIZE+MID_SIZE, TAIL_STR, TAIL_SIZE)==0);

        uint64_t mid_value=*(uint64_t*)(buf+HEAD_SIZE);
        klee_assume (crc64 (0, buf, BUF_SIZE)==mid_value);

        klee_assert(0);

        return 0;
}
```

It works:

```
% time klee --libc=uclibc klee_CRC64.bc
...
real    5m17.081s
user    5m17.014s
sys     0m0.319s

% ls klee-last | grep err
test000001.user.err
test000002.user.err
test000003.external.err

% ktest-tool --write-ints klee-last/test000003.ktest
ktest file : 'klee-last/test000003.ktest'
args       : ['klee_CRC64.bc']
num objects: 1
object    0: name: b'buf'
object    0: size: 46
object    0: data: b'Hello, world!.. T+]\xb9A\x08\x0fq ... and goodbye\xb6\x8f\x9c\xd8\
   xc5\x00'
```

8 bytes between two strings is 64-bit value which equals to CRC64 of this whole block. Again, it's faster than brute-force way to find it. If to decrease last spare 6-byte buffer to 4 bytes or less, KLEE works so long so I've stopped it.

### 11.5.3 Recovering input data for given CRC32 value of it

I've always wanted to do so, but everyone knows this is impossible for input buffers larger than 4 bytes. As my experiments show, it's still possible for tiny input buffers of data, which is constrained in some way.

The CRC32 value of 6-byte "SILVER" string is known: `0xDFA3DFDD`. KLEE can find this 6-byte string, if it knows that each byte of input buffer is in $A..Z$ limits:

```
#include <stdint.h>
#include <stdbool.h>

uint32_t crc32(uint32_t crc, unsigned char *buf, int len)
{
    int k;

    crc = ~crc;
    while (len--)
    {
        crc ^= *buf++;
        for (k = 0; k < 8; k++)
            crc = crc & 1 ? (crc >> 1) ^ 0xedb88320 : crc >> 1;
```

```
14     }
15     return ~crc;
16 }
17 
18 #define SIZE 6
19 
20 bool find_string(char str[SIZE])
21 {
22         int i=0;
23         for (i=0; i<SIZE; i++)
24                 if (str[i]<'A' || str[i]>'Z')
25                         return false;
26 
27         if (crc32(0, &str[0], SIZE)!=0xDFA3DFDD)
28                 return false;
29 
30         // OK, input str is valid
31         klee_assert(0); // force KLEE to produce .err file
32         return true;
33 };
34 
35 int main()
36 {
37         uint8_t str[SIZE];
38 
39         klee_make_symbolic(str, sizeof str, "str");
40 
41         find_string(str);
42 
43         return 0;
44 }
```

```
% clang -emit-llvm -c -g klee_SILVER.c
...

% klee klee_SILVER.bc
...

% ls klee-last | grep err
test000013.external.err

% ktest-tool --write-ints klee-last/test000013.ktest
ktest file : 'klee-last/test000013.ktest'
args       : ['klee_SILVER.bc']
num objects: 1
object    0: name: b'str'
object    0: size: 6
object    0: data: b'SILVER'
```

Still, it's no magic: if to remove condition at lines 23..25 (i.e., if to relax constraints), KLEE will produce some other string, which will be still correct for the CRC32 value given.

It works, because 6 Latin characters in $A..Z$ limits contain $\approx 28.2$ bits: $log_2 26^6 \approx 28.2$, which is even smaller value than 32. In other words, the final CRC32 value holds enough bits to recover $\approx 28.2$ bits of input.

The input buffer can be even bigger, if each byte of it will be in even tighter constraints (decimal digits, binary digits, etc).

### 11.5.4 In comparison with other hashing algorithms

Things are that easy for some other hashing algorithms like *Fletcher checksum*, but not for cryptographically secure ones (like MD5, SHA1, etc), they are protected from such simple cryptoanalysis. See also: 18.

# 12 MaxSMT

## 12.1 Making smallest possible test suite using Z3

I once worked on rewriting large piece of code into pure C, and there were a tests, several thousands. Testing process was painfully slow, so I thought if the test suite can be minimized somehow.

What we can do is to run each test and get code coverage (information about which lines of code was executed and which are not). Then the task is to make such test suite, where coverage is maximum, and number of tests is minimal.

In fact, this is *set cover problem* (also known as *hitting set problem*). While simpler algorithms exist (see Wikipedia[83]), it is also possible to solve with SMT-solver.

First, I took LZSS[84] compression/decompression code [85] for the example, from Apple sources. Such routines are not easy to test. Here is my version of it: `https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/MaxSMT/set_cover/compression.c`. I've added random generation of input data to be compressed. Random generation is dependent of some kind of input seed. Standard `srand()`/`rand()` are not recommended to be used, but for such simple task as ours, it's OK. I'll generate[86] 1000 tests with 0..999 seeds, that would produce random data to be compressed/decompressed/checked.

After the compression/decompression routine has finished its work, GNU gcov utility is executed, which produces result like this:

```
...
     3395:  189:        for (i = 1; i < F; i++) {
     3395:  190:            if ((cmp = key[i] - sp->text_buf[p + i]) != 0)
     2565:  191:                break;
        -:  192:        }
     2565:  193:        if (i > sp->match_length) {
     1291:  194:            sp->match_position = p;
     1291:  195:            if ((sp->match_length = i) >= F)
    #####:  196:                break;
        -:  197:        }
     2565:  198:    }
    #####:  199:    sp->parent[r] = sp->parent[p];
    #####:  200:    sp->lchild[r] = sp->lchild[p];
    #####:  201:    sp->rchild[r] = sp->rchild[p];
    #####:  202:    sp->parent[sp->lchild[p]] = r;
    #####:  203:    sp->parent[sp->rchild[p]] = r;
    #####:  204:    if (sp->rchild[sp->parent[p]] == p)
    #####:  205:        sp->rchild[sp->parent[p]] = r;
...
```

A leftmost number is an execution count for each line. `#####` means the line of code hasn't been executed at all. The second column is a line number.

Now the Z3Py script, which will parse all these 1000 gcov results and produce minimal *hitting set*:

```
#!/usr/bin/env python

import re, sys
from z3 import *

TOTAL_TESTS=1000
```

[83] `https://en.wikipedia.org/wiki/Set_cover_problem`
[84] Lempel–Ziv–Storer–Szymanski
[85] `https://github.com/opensource-apple/kext_tools/blob/master/compression.c`
[86] `https://github.com/DennisYurichev/yurichev.com/blob/master/blog/set_cover/gen_gcov_tests.sh`

```
# read gcov result and return list of lines executed:
def process_file (fname):
    lines=[]
    f=open(fname,"r")

    while True:
        l=f.readline().rstrip()
        m = re.search('^ *([0-9]+):  ([0-9]+):.*$', l)
        if m!=None:
            lines.append(int(m.group(2)))
        if len(l)==0:
            break

    f.close()
    return lines

# k=test number; v=list of lines executed
stat={}
for test in range(TOTAL_TESTS):
    stat[test]=process_file("compression.c.gcov."+str(test))

# that will be a list of all lines in all tests:
all_lines=set()
# k=line, v=list of tests, which trigger that line:
tests_for_line={}

for test in stat:
    all_lines|=set(stat[test])
    for line in stat[test]:
        tests_for_line[line]=tests_for_line.get(line, []) + [test]

# int variable for each test:
tests=[Int('test_%d' % (t)) for t in range(TOTAL_TESTS)]

# this is optimization problem, so Optimize() instead of Solver():
opt = Optimize()

# each test variable is either 0 (absent) or 1 (present):
for t in tests:
    opt.add(Or(t==0, t==1))

# we know which tests can trigger each line
# so we enumerate all tests when preparing expression for each line
# we form expression like "test_1==1 OR test_2==1 OR ..." for each line:
for line in list(all_lines):
    expressions=[tests[s]==1 for s in tests_for_line[line]]
    # expression is a list which unfolds as list of arguments into Z3's Or() function (
       see asterisk)
    # that results in big expression like "Or(test_1==1, test_2==1, ...)"
    # the expression is then added as a constraint:
    opt.add(Or(*expressions))

# we need to find a such solution, where minimal number of all "test_X" variables will
   have 1
# "*tests" unfolds to a list of arguments: [test_1, test_2, test_3,...]
```

```
# "Sum(*tests)" unfolds to the following expression: "Sum(test_1, test_2, ...)"
# the sum of all "test_X" variables should be as minimal as possible:
h=opt.minimize(Sum(*tests))

print (opt.check())
m=opt.model()

# print all variables set to 1:
for t in tests:
    if m[t].as_long()==1:
        print (t)
```

And what it produces (~19s on my old Intel Quad-Core Xeon E3-1220 3.10GHz):

```
% time python set_cover.py
sat
test_7
test_48
test_134
python set_cover.py  18.95s user 0.03s system 99% cpu 18.988 total
```

We need just these 3 tests to execute (almost) all lines in the code: looks impressive, given the fact, that it would be notoriously hard to pick these tests by hand! The result can be checked easily, again, using gcov utility.

This is sometimes also called MaxSAT/MaxSMT — the problem is to find solution, but the solution where some variable/expression is maximal as possible, or minimal as possible.

Also, the code gives incorrect results on Z3 4.4.1, but working correctly on Z3 4.5.0 (so please upgrade). This is relatively fresh feature in Z3, so probably it was not stable in previous versions?

The files: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/MaxSMT/set_cover.

Further reading: https://en.wikipedia.org/wiki/Set_cover_problem, https://en.wikipedia.org/wiki/Maximum_satisfiability_problem, https://en.wikipedia.org/wiki/Optimization_problem.

## 12.2 GCD and LCM

### 12.2.1 Explanation of the GCD

What is Greatest common divisor (GCD)?

Let's suppose, you want to cut a rectangle by squares. What is maximal square could be?

For a 14*8 rectangle, this is 2*2 square:

```
**************
**************
**************
**************
**************
**************
**************
**************
```

->

```
** ** ** ** ** ** **
** ** ** ** ** ** **

** ** ** ** ** ** **
** ** ** ** ** ** **

** ** ** ** ** ** **
** ** ** ** ** ** **

```

```
** ** ** ** ** ** **
** ** ** ** ** ** **
```

What for 14*7 rectangle? It's 7*7 square:

```
**************
**************
**************
**************
**************
**************
**************
```

->

```
******* *******
******* *******
******* *******
******* *******
******* *******
******* *******
******* *******
```

14*9 rectangle? 1, i.e., smallest possible.

```
**************
**************
**************
**************
**************
**************
**************
**************
**************
**************
```

Also, GCD can be used to determine biggest "granule" for problem like 3.1, and this is 0.05 or 5 for that example.

To compute GCD, one of the oldest algorithms is used: Euclidean algorithm. But, I can demonstrate how to make things much less efficient, but more spectacular.

To find GCD of 14 and 8, we are going to solve this system of equations:

```
x*GCD=14
y*GCD=8
```

Then we drop $x$ and $y$, we don't need them. This system can be solved using paper and pencil, but GCD must be as big as possible. Here we can use Z3 in MaxSMT mode:

```
#!/usr/bin/env python

from z3 import *

opt = Optimize()

x,y,GCD=Ints('x y GCD')

opt.add(x*GCD==14)
opt.add(y*GCD==8)

h=opt.maximize(GCD)

```

```
print (opt.check())
print (opt.model())
```

That works:

```
sat
[y = 4, x = 7, GCD = 2]
```

What if we need to find GCD for 3 numbers? Maybe we are going to fill a space with biggest possible cubes?

```
#!/usr/bin/env python

from z3 import *

opt = Optimize()

x,y,z,GCD=Ints('x y z GCD')

opt.add(x*GCD==300)
opt.add(y*GCD==333)
opt.add(z*GCD==900)

h=opt.maximize(GCD)

print (opt.check())
print (opt.model())
```

This is 3:

```
sat
[z = 300, y = 111, x = 100, GCD = 3]
```

In SMT-LIB form:

```
; checked with Z3 and MK85

; must be 21
; see also: https://www.wolframalpha.com/input/?i=GCD[861,3969,840]

(declare-fun x () (_ BitVec 16))
(declare-fun y () (_ BitVec 16))
(declare-fun z () (_ BitVec 16))
(declare-fun GCD () (_ BitVec 16))

(assert (= (bvmul ((_ zero_extend 16) x) ((_ zero_extend 16) GCD)) (_ bv861 32)))
(assert (= (bvmul ((_ zero_extend 16) y) ((_ zero_extend 16) GCD)) (_ bv3969 32)))
(assert (= (bvmul ((_ zero_extend 16) z) ((_ zero_extend 16) GCD)) (_ bv840 32)))

(maximize GCD)

(check-sat)
(get-model)

; correct result:
;(model
;        (define-fun x () (_ BitVec 16) (_ bv41 16)) ; 0x29
;        (define-fun y () (_ BitVec 16) (_ bv189 16)) ; 0xbd
;        (define-fun z () (_ BitVec 16) (_ bv40 16)) ; 0x28
;        (define-fun GCD () (_ BitVec 16) (_ bv21 16)) ; 0x15
;)
```

### 12.2.2 Couple of words about GCD

GCD of coprimes is 1.

***

GCD is also a common set of factors of several numbers. This we can see in Mathematica:

```
In[]:= FactorInteger[300]
Out[]= {{2, 2}, {3, 1}, {5, 2}}

In[]:= FactorInteger[333]
Out[]= {{3, 2}, {37, 1}}

In[]:= GCD[300, 333]
Out[]= 3
```

I.e., $300 = 2^2 \cdot 3 \cdot 5^2$ and $333 = 3^2 \cdot 37$ and $GCD = 3$, which is smallest factor.

Or:

```
In[]:= FactorInteger[11*13*17]
Out[]= {{11, 1}, {13, 1}, {17, 1}}

In[]:= FactorInteger[7*11*13*17]
Out[]= {{7, 1}, {11, 1}, {13, 1}, {17, 1}}

In[]:= GCD[11*13*17, 7*11*13*17]
Out[]= 2431

In[]:= 11*13*17
Out[]= 2431
```

(Common factors are 11, 13 and 17, so $GCD = 11 \cdot 13 \cdot 17 = 2431$.)

### 12.2.3 Oculus VR Flicks and GCD

I've found this:

```
A flick (frame-tick) is a very small unit of time. It is 1/705600000 of a second,
    exactly.

1 flick = 1/705600000 second

This unit of time is the smallest time unit which is LARGER than a nanosecond,
and can in integer quantities exactly represent a single frame duration for
24hz, 25hz, 30hz, 48hz, 50hz, 60hz, 90hz, 100hz, 120hz, and also 1/1000 divisions of
    each.
This makes it suitable for use via std::chrono::duration and std::ratio
for doing timing work against the system high resolution clock, which is in nanoseconds,
but doesn't get slightly out of sync when doing common frame rates.

In order to accomodate media playback, we also support some common audio sample rates as
     well.
This list is not exhaustive, but covers the majority of digital audio formats.
They are 8kHz, 16kHz, 22.05kHz, 24kHz, 32kHz, 44.1kHz, 48kHz, 88.2kHz, 96kHz, and 192kHz
    .
While humans can't hear higher than 48kHz, the higher sample rates are used
for working audio files which might later be resampled or retimed.

The NTSC variations (~29.97, etc) are actually defined as 24 * 1000/1001 and
```

```
30 * 1000/1001, which are impossible to represent exactly in a way where 1 second is
   exact,
so we don't bother - they'll be inexact in any circumstance.

Details

    1/24 fps frame: 29400000 flicks
    1/25 fps frame: 28224000 flicks
    1/30 fps frame: 23520000 flicks
    1/48 fps frame: 14700000 flicks
    1/50 fps frame: 14112000 flicks
    1/60 fps frame: 11760000 flicks
    1/90 fps frame: 7840000 flicks
    1/100 fps frame: 7056000 flicks
    1/120 fps frame: 5880000 flicks
    1/8000 fps frame: 88200 flicks
    1/16000 fps frame: 44100 flicks
    1/22050 fps frame: 32000 flicks
    1/24000 fps frame: 29400 flicks
    1/32000 fps frame: 22050 flicks
    1/44100 fps frame: 16000 flicks
    1/48000 fps frame: 14700 flicks
    1/88200 fps frame: 8000 flicks
    1/96000 fps frame: 7350 flicks
    1/192000 fps frame: 3675 flicks
```

( https://github.com/OculusVR/Flicks )

Where the number came from? Let's enumerate all possible time intervals they want to use and find GCD using Mathematica:

```
In[]:= GCD[1/24, 1/24000, 1/25, 1/25000, 1/30, 1/30000, 1/48, 1/50,
 1/50000, 1/60, 1/60000, 1/90, 1/90000, 1/100, 1/100000, 1/120,
 1/120000, 1/8000, 1/16000, 1/22050, 1/24000, 1/32000, 1/44100,
 1/48000, 1/88200, 1/96000, 1/192000]

Out[]= 1/705600000
```

Rationale: you may want to play a video with $\frac{1}{50}$ fps and, simultaneously, play audio with 96kHz sampling rate. Given that, you can change video frame after each 14112000 flicks and change one audio sample after each 7350 flicks. Use any other video fps and any audio sampling rate and you will have all time periods as integer numbers. No ratios any more.

On contrary, one nanosecond wouldn't fit: try to represent 1/30 second in nanoseconds, this is (still) ratio: 33333.33333... nanoseconds.

### 12.2.4 Explanation of the Least Common Multiple

Many people use LCM in school. Sum up $\frac{1}{4}$ and $\frac{1}{6}$. To find an answer mentally, you ought to find Lowest Common Denominator, which can be 4*6=24. Now you can sum up $\frac{6}{24} + \frac{4}{24} = \frac{10}{24}$.

But the lowest denominator is also a LCM. LCM of 4 and 6 is 12: $\frac{3}{12} + \frac{2}{12} = \frac{5}{12}$.

To find LCM of 4 and 6, we are going to solve the following diophantine (i.e., allowing only integer solutions) system of equations:

$4x = 6y = LCM$

... where LCM>0 and as small, as possible.

```
#!/usr/bin/env python

from z3 import *

opt = Optimize()
```

```
x,y,LCM=Ints('x y LCM')

opt.add(x*4==LCM)
opt.add(y*6==LCM)
opt.add(LCM>0)

h=opt.minimize(LCM)

print (opt.check())
print (opt.model())
```

The (correct) answer:

```
sat
[y = 2, x = 3, LCM = 12]
```

### 12.2.5 File copying routine

In GNU coreutils, we can find that LCM is used to find optimal buffer size, if buffer sizes in input and ouput files are differ. For example, input file has buffer of 4096 bytes, and output is 6144. Well, these sizes are somewhat suspicious. I made up this example. Nevertheless, LCM of 4096 and 6144 is 12288. This is a buffer size you can allocate, so that you will minimize number of read/write operations during copying.

https://github.com/coreutils/coreutils/blob/4cb3f4faa435820dc99c36b30ce93c7d01501f65/src/copy.c#L1246. https://github.com/coreutils/coreutils/blob/master/gl/lib/buffer-lcm.c.

## 12.3 Assignment problem

I've found this at http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf and took screenshot:

**Example 1:** You work as a sales manager for a toy manufacturer, and you currently have three salespeople on the road meeting buyers. Your salespeople are in Austin, TX; Boston, MA; and Chicago, IL. You want them to fly to three other cities: Denver, CO; Edmonton, Alberta; and Fargo, ND. The table below shows the cost of airplane tickets in dollars between these cities.

| From \ To | Denver | Edmonton | Fargo |
|---|---|---|---|
| Austin | 250 | 400 | 350 |
| Boston | 400 | 600 | 350 |
| Chicago | 200 | 400 | 250 |

Where should you send each of your salespeople in order to minimize airfare?

As in my previous examples, Z3 and SMT-solver may be overkill for the task. Simpler algorithm exists for this task (*Hungarian algorithm/method*) [87].

But again, I use it to demonstrate the problem + as SMT-solvers demonstration.

Here is what I do:

```
#!/usr/bin/env python

from z3 import *

# this is optimization problem:
s=Optimize()

choice=[Int('choice_%d' % i) for i in range(3)]
row_value=[Int('row_value_%d' % i) for i in range(3)]

for i in range(3):
    s.add(And(choice[i]>=0, choice[i]<=2))

s.add(Distinct(choice))
s.add(row_value[0]==
        If(choice[0]==0, 250,
        If(choice[0]==1, 400,
        If(choice[0]==2, 350, -1))))

s.add(row_value[1]==
        If(choice[1]==0, 400,
```

[87] See also: https://en.wikipedia.org/wiki/Hungarian_algorithm.

```
        If(choice[1]==1, 600,
        If(choice[1]==2, 350, -1))))

s.add(row_value[2]==
        If(choice[2]==0, 200,
        If(choice[2]==1, 400,
        If(choice[2]==2, 250, -1))))

final_sum=Int('final_sum')

# final_sum equals to sum of all row_value[] values:
s.add(final_sum==Sum(*row_value))

# find such a (distinct) choice[]'s, so that the final_sum would be minimal:
s.minimize(final_sum)

print s.check()
print s.model()
```

In plain English this means *choose such columns, so that their sum would be as small as possible.*

Result is seems to be correct:

```
sat
[choice_0 = 1,
 choice_1 = 2,
 choice_2 = 0,
 z3name!12 = 0,
 z3name!7 = 1,
 z3name!10 = 2,
 z3name!8 = 0,
 z3name!11 = 0,
 z3name!9 = 0,
 final_sum = 950,
 row_value_2 = 200,
 row_value_1 = 350,
 row_value_0 = 400]
```

Again, as it is in the corresponding PDF presentation:

After checking all six possible assignments we can determine that the optimal one is the following.

$$\begin{bmatrix} 250 & \boxed{400} & 350 \\ 400 & 600 & \boxed{350} \\ \boxed{200} & 400 & 250 \end{bmatrix}$$

The total cost of this assignment is $\$400 + \$350 + \$200 = \$950$.

Thus your salespeople should travel from Austin to Edmonton, Boston to Fargo, and Chicago to Denver.

(However, I've no idea what "z3name" variables mean, perhaps, some internal variables?)

The problem can also be stated in SMT-LIB 2.0 format, and solved using MK85:

```
; checked with Z3 and MK85

(declare-fun choice1 () (_ BitVec 2))
(declare-fun choice2 () (_ BitVec 2))
(declare-fun choice3 () (_ BitVec 2))

(declare-fun row_value1 () (_ BitVec 16))
(declare-fun row_value2 () (_ BitVec 16))
(declare-fun row_value3 () (_ BitVec 16))

(declare-fun final_sum () (_ BitVec 16))

(assert (bvule choice1 (_ bv3 2)))
(assert (bvule choice2 (_ bv3 2)))
(assert (bvule choice3 (_ bv3 2)))

(assert (distinct choice1 choice2 choice3))

(assert (= row_value1
        (ite (= choice1 (_ bv0 2)) (_ bv250 16)
        (ite (= choice1 (_ bv1 2)) (_ bv400 16)
        (ite (= choice1 (_ bv2 2)) (_ bv250 16)
        (_ bv999 16))))))
```

```
(assert (= row_value2
        (ite (= choice2 (_ bv0 2)) (_ bv400 16)
        (ite (= choice2 (_ bv1 2)) (_ bv600 16)
        (ite (= choice2 (_ bv2 2)) (_ bv350 16)
        (_ bv999 16))))))

(assert (= row_value3
        (ite (= choice3 (_ bv0 2)) (_ bv200 16)
        (ite (= choice3 (_ bv1 2)) (_ bv400 16)
        (ite (= choice3 (_ bv2 2)) (_ bv250 16)
        (_ bv999 16))))))

(assert (= final_sum (bvadd row_value1 row_value2 row_value3)))

(minimize final_sum)

(check-sat)
(get-model)
```

Listing 2: The result

```
sat
(model
        (define-fun choice1 () (_ BitVec 2) (_ bv1 2)) ; 0x1
        (define-fun choice2 () (_ BitVec 2) (_ bv2 2)) ; 0x2
        (define-fun choice3 () (_ BitVec 2) (_ bv0 2)) ; 0x0
        (define-fun row_value1 () (_ BitVec 16) (_ bv400 16)) ; 0x190
        (define-fun row_value2 () (_ BitVec 16) (_ bv350 16)) ; 0x15e
        (define-fun row_value3 () (_ BitVec 16) (_ bv200 16)) ; 0xc8
        (define-fun final_sum () (_ BitVec 16) (_ bv950 16)) ; 0x3b6
)
```

## 12.4 Finding function minimum

```
; 1959 AHSME Problems, Problem 8

; 8th problem at http://artofproblemsolving.com/wiki/index.php?title=1959_AHSME_Problems

; "The value of $x^2-6x+13$ can never be less than:
; 4, 4.5, 5, 7, 13"
;
; must be x=3, result=4, see also: https://www.wolframalpha.com/input/?i=minimize+x*x-6x
   %2B13+over+[-100,100]

(declare-fun x () (_ BitVec 16))
(declare-fun result () (_ BitVec 16))

(assert (=
        result
        (bvadd
                (bvsub
                        (bvmul_no_overflow x x)
                        (bvmul_no_overflow x #x0006))
                #x000d))
)
(minimize result)
```

```
(check-sat)
(get-model)

; correct result:
;(model
;        (define-fun x () (_ BitVec 16) (_ bv3 16)) ; 0x3
;        (define-fun result () (_ BitVec 16) (_ bv4 16)) ; 0x4
;)
```

## 12.5 Travelling salesman problem

This is it:

```
from z3 import *

# the city matrix has been copypasted from https://developers.google.com/optimization/
    routing/tsp/tsp
matrix=[
  [   0, 2451,  713, 1018, 1631, 1374, 2408,  213, 2571,  875, 1420, 2145, 1972], # New
      York
  [2451,    0, 1745, 1524,  831, 1240,  959, 2596,  403, 1589, 1374,  357,  579], # Los
      Angeles
  [ 713, 1745,    0,  355,  920,  803, 1737,  851, 1858,  262,  940, 1453, 1260], #
      Chicago
  [1018, 1524,  355,    0,  700,  862, 1395, 1123, 1584,  466, 1056, 1280,  987], #
      Minneapolis
  [1631,  831,  920,  700,    0,  663, 1021, 1769,  949,  796,  879,  586,  371], #
      Denver
  [1374, 1240,  803,  862,  663,    0, 1681, 1551, 1765,  547,  225,  887,  999], #
      Dallas
  [2408,  959, 1737, 1395, 1021, 1681,    0, 2493,  678, 1724, 1891, 1114,  701], #
      Seattle
  [ 213, 2596,  851, 1123, 1769, 1551, 2493,    0, 2699, 1038, 1605, 2300, 2099], #
      Boston
  [2571,  403, 1858, 1584,  949, 1765,  678, 2699,    0, 1744, 1645,  653,  600], # San
      Francisco
  [ 875, 1589,  262,  466,  796,  547, 1724, 1038, 1744,    0,  679, 1272, 1162], # St.
      Louis
  [1420, 1374,  940, 1056,  879,  225, 1891, 1605, 1645,  679,    0, 1017, 1200], #
      Houston
  [2145,  357, 1453, 1280,  586,  887, 1114, 2300,  653, 1272, 1017,    0,  504], #
      Phoenix
  [1972,  579, 1260,  987,  371,  999,  701, 2099,  600, 1162,  1200,  504,    0]] # Salt
       Lake City

# 13 cities:
city_names = ["New York", "Los Angeles", "Chicago",
              "Minneapolis", "Denver", "Dallas", "Seattle", "Boston", "San Francisco",
              "St. Louis", "Houston", "Phoenix", "Salt Lake City"]

# unfortunately, we use only first 6 cities:
cities_t=6
#cities_t=len(city_names)

"""
```

```
this function generates expression like:

If(And(route_5 == 5, route_0 == 5),
   0,
   If(And(route_5 == 5, route_0 == 4),
      663,
      If(And(route_5 == 5, route_0 == 3),
         862,
         If(And(route_5 == 5, route_0 == 2),
            803,
            If(And(route_5 == 5, route_0 == 1),
               1240,
...

so it's like switch()
"""

def distance_fn(c1, c2):
    t=9999999 # default value
    for i in range(cities_t):
        for j in range(cities_t):
            t=If(And(c1==i, c2==j), matrix[i][j], t)
    #print t
    return t

s=Optimize()

# which city is visited on each step of route?
route=[Int('route_%d' % i) for i in range(cities_t)]
# what distance between the current city and the next in route[]?
distance=[Int('distance_%d' % i) for i in range(cities_t)]

# all ints in route[] must be in [0..cities_t) range:
for r in route:
    s.add(r>=0, r<cities_t)

# no city may be visited twice:
s.add(Distinct(route))

for i in range(cities_t):
    s.add(distance[i]==distance_fn(route[i], route[(i+1) % cities_t]))

distance_total=Int('distance_total')

s.add(distance_total==Sum(distance))

# minimize total distance:
#s.minimize(distance_total)
# or maximize:
s.maximize(distance_total)

print s.check()
m=s.model()
#print m
for i in range(cities_t):
    print "%s (%d mi to the next city) ->" % (city_names[m[route[i]].as_long()], m[
```

```
        distance[i]].as_long())

print "distance_total=", m[distance_total], "mi"
```

The result:

```
sat
Dallas (1240 mi to the next city) ->
Los Angeles (831 mi to the next city) ->
Denver (700 mi to the next city) ->
Minneapolis (355 mi to the next city) ->
Chicago (713 mi to the next city) ->
New York (1374 mi to the next city) ->
distance_total= 5213 mi
```

Map I generated with Wolfram Mathematica:

Figure 31:

Maximizing:

```
sat
Dallas (862 mi to the next city) ->
Minneapolis (700 mi to the next city) ->
Denver (1631 mi to the next city) ->
New York (2451 mi to the next city) ->
Los Angeles (1745 mi to the next city) ->
Chicago (803 mi to the next city) ->
distance_total= 8192 mi
```

The map:

Figure 32:

I could only process 6 cities, and it takes starting at several seconds up to 1 minute on my venerable Intel Quad-Core Xeon E3-1220 3.10GHz. Perhaps, this is not a right tool for the job, well-known TSP algorithms way faster.

Even bruteforce enumeration is way faster ($6! = 720$ paths).

However, it still can serve as demonstration.

# 13 Program synthesis

Program synthesis is a process of automatic program generation, in accordance with some specific goals.

## 13.1 Synthesis of simple program using Z3 SMT-solver

Sometimes, multiplication operation can be replaced with a several operations of shifting/addition/subtraction. Compilers do so, because pack of instructions can be executed faster.

For example, multiplication by 19 is replaced by GCC 5.4 with pair of instructions: `lea edx, [eax+eax*8]` and `lea eax, [eax+edx*2]`. This is sometimes also called “superoptimization”.

Let’s see if we can find a shortest possible instructions pack for some specified multiplier.

As I’ve already wrote once, SMT-solver can be seen as a solver of huge systems of equations. The task is to construct such system of equations, which, when solved, could produce a short program. I will use electronics analogy here, it can make things a little simpler.

First of all, what our program will be consting of? There will be 3 operations allowed: ADD/SUB/SHL. Only registers allowed as operands, except for the second operand of SHL (which could be in 1..31 range). Each register will be assigned only once (as in SSA[88]).

And there will be some “magic block”, which takes all previous register states, it also takes operation type, operands and produces a value of next register’s state.

```
                op ------------+
                op1_reg -----+ |
                op2_reg ---+ | |
                           | | |
                           v v v
                     +---------------+
                     |               |
        registers -> |               | -> new register's state
                     |               |
                     +---------------+
```

Now let’s take a look on our schematics on top level:

[88] Static single assignment form

```
        0 -> blk -> blk -> blk .. -> blk -> 0

        1 -> blk -> blk -> blk .. -> blk -> multiplier
```

Each block takes previous state of registers and produces new states. There are two chains. First chain takes 0 as state of R0 at the very beginning, and the chain is supposed to produce 0 at the end (since zero multiplied by any value is still zero). The second chain takes 1 and must produce multiplier as the state of very last register (since 1 multiplied by multiplier must equal to multiplier).

Each block is "controlled" by operation type, operands, etc. For each column, there is each own set.

Now you can view these two chains as two equations. The ultimate goal is to find such state of all operation types and operands, so the first chain will equal to 0, and the second to multiplier.

Let's also take a look into "magic block" inside:

```
                        op1_reg         op
                           |            v
                           v          +-----+
        registers ---> selector1 --> | ADD |
                  +                  | SUB | ---> result
                  |                  | SHL |
                  +-> selector2 --> +-----+
                          ^             ^
                          |             |
                       op2_reg       op2_imm
```

Each selector can be viewed as a simple multipositional switch. If operation is SHL, a value in range of 1..31 is used as second operand.

So you can imagine this electric circuit and your goal is to turn all switches in such a state, so two chains will have 0 and multiplier on output. This sounds like logic puzzle in some way. Now we will try to use Z3 to solve this puzzle.

First, we define all varibles:

```
R=[[BitVec('S_s%d_c%d' % (s, c), 32) for s in range(MAX_STEPS)] for c in range (CHAINS)]
op=[Int('op_s%d' % s) for s in range(MAX_STEPS)]
op1_reg=[Int('op1_reg_s%d' % s) for s in range(MAX_STEPS)]
op2_reg=[Int('op2_reg_s%d' % s) for s in range(MAX_STEPS)]
op2_imm=[BitVec('op2_imm_s%d' % s, 32) for s in range(MAX_STEPS)]
```

R[][] is registers state for each chain and each step.
On contrary, `op/op1_reg/op2_reg/op2_imm` variables are defined for each step, but for both chains, since both chains at each column has the same operation/operands.

Now we must limit count of operations, and also, register's number for each step must not be bigger than step number, in other words, instruction at each step is allowed to access only registers which were already set before:

```
for s in range(1, STEPS):
    # for each step
    sl.add(And(op[s]>=0, op[s]<=2))
    sl.add(And(op1_reg[s]>=0, op1_reg[s]<s))
    sl.add(And(op2_reg[s]>=0, op2_reg[s]<s))
    sl.add(And(op2_imm[s]>=1, op2_imm[s]<=31))
```

Fix register of first step for both chains:

```
for c in range(CHAINS):
    # for each chain:
    sl.add(R[c][0]==chain_inputs[c])
    sl.add(R[c][STEPS-1]==chain_inputs[c]*multiplier)
```

Now let's add "magic blocks":

```
for s in range(1, STEPS):
    sl.add(R[c][s]==simulate_op(R,c, op[s], op1_reg[s], op2_reg[s], op2_imm[s]))
```

Now how “magic block” is defined?

```
def selector(R, c, s):
    # for all MAX_STEPS:
    return If(s==0, R[c][0],
            If(s==1, R[c][1],
            If(s==2, R[c][2],
            If(s==3, R[c][3],
            If(s==4, R[c][4],
            If(s==5, R[c][5],
            If(s==6, R[c][6],
            If(s==7, R[c][7],
            If(s==8, R[c][8],
            If(s==9, R[c][9],
                0)))))))))) # default

def simulate_op(R, c, op, op1_reg, op2_reg, op2_imm):
    op1_val=selector(R,c,op1_reg)
    return If(op==0, op1_val + selector(R, c, op2_reg),
           If(op==1, op1_val - selector(R, c, op2_reg),
           If(op==2, op1_val << op2_imm,
               0))) # default
```

This is very important to understand: if the operation is ADD/SUB, `op2_imm`'s value is just ignored. Otherwise, if operation is SHL, value of `op2_reg` is ignored. Just like in case of digital circuit.

The code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/pgm_synth/mult/mult.py.

Now let's see how it works:

```
% ./mult.py 12
multiplier= 12
attempt, STEPS= 2
unsat
attempt, STEPS= 3
unsat
attempt, STEPS= 4
sat!
r1=SHL r0, 2
r2=SHL r1, 1
r3=ADD r1, r2
tests are OK
```

The first step is always a step containing 0/1, or, r0. So when our solver reporting about 4 steps, this means 3 instructions.

Something harder:

```
% ./mult.py 123
multiplier= 123
attempt, STEPS= 2
unsat
attempt, STEPS= 3
unsat
attempt, STEPS= 4
unsat
attempt, STEPS= 5
sat!
r1=SHL r0, 2
r2=SHL r1, 5
r3=SUB r2, r1
```

```
r4=SUB r3, r0
tests are OK
```

Now the code multiplying by 1234:

```
r1=SHL r0, 6
r2=ADD r0, r1
r3=ADD r2, r1
r4=SHL r2, 4
r5=ADD r2, r3
r6=ADD r5, r4
```

Looks great, but it took $\approx$ 23 seconds to find it on my Intel Xeon CPU E31220 @ 3.10GHz. I agree, this is far from practical usage. Also, I'm not quite sure that this piece of code will work faster than a single multiplication instruction. But anyway, it's a good demonstration of SMT solvers capabilities.

The code multiplying by 12345 ($\approx$ 150 seconds):

```
r1=SHL r0, 5
r2=SHL r0, 3
r3=SUB r2, r1
r4=SUB r1, r3
r5=SHL r3, 9
r6=SUB r4, r5
r7=ADD r0, r6
```

Multiplication by 123456 ($\approx$ 8 minutes!):

```
r1=SHL r0, 9
r2=SHL r0, 13
r3=SHL r0, 2
r4=SUB r1, r2
r5=SUB r3, r4
r6=SHL r5, 4
r7=ADD r1, r6
```

#### 13.1.1 Few notes

I've removed SHR instruction support, simply because the code multiplying by a constant makes no use of it. Even more: it's not a problem to add support of constants as second operand for all instructions, but again, you wouldn't find a piece of code which does this job and uses some additional constants. Or maybe I wrong?

Of course, for another job you'll need to add support of constants and other operations. But at the same time, it will work slower and slower. So I had to keep ISA[89] of this toy CPU[90] as compact as possible.

#### 13.1.2 The code

https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/pgm_synth/mult.

### 13.2 Rockey dongle: finding unknown algorithm using only input/output pairs

(This text was first published in August 2012 in my blog: http://blog.yurichev.com/node/71.)

Some smartcards can execute Java or .NET code - that's the way to hide your sensitive algorithm into chip that is very hard to break (decapsulate). For example, one may encrypt/decrypt data files by hidden crypto algorithm rendering software piracy of such software close to impossible — an encrypted date file created on software with connected smartcard would be impossible to decrypt on cracked version of the same software. (This leads to many nuisances, though.)

That's what called *black box*.

[89] Instruction Set Architecture
[90] Central processing unit

Some software protection dongles offers this functionality too. One example is Rockey 4[91].

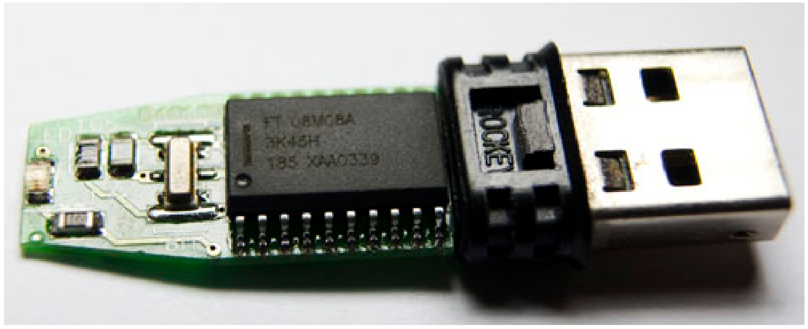


Figure 33: Rockey 4 dongle

This is a small dongle connected via USB. Is contain some user-defined memory but also memory for user algorithms.

The virtual (toy) CPU for these algorithms is very simple: it offer only 8 16-bit registers (however, only 4 can be set and read) and 8 operations (addition, subtractation, cyclic left shifting, multiplication, OR, XOR, AND, negation).

Second instruction argument can be a constant (from 0 to 63) instead of register.

Each algorithm is described by string like
`A=A+B, B=C*13, D=D^A, C=B*55, C=C&A, D=D|A, A=A*9, A=A&B`.

There are no memory, stack, conditional/unconditional jumps, etc.

Each algorithm, obviously, can't have side effects, so they are actually *pure functions* and their results can be *memoized*.

By the way, as it has been mentioned in Rockey 4 manual, first and last instruction cannot have constants. Maybe that's because these fields used for some internal data: each algorithm start and end should be marked somehow internally anyway.

Would it be possible to reveal hidden impossible-to-read algorithm only by recording input/output dongle traffic? Common sense tell us "no". But we can try anyway.

Since, my goal wasn't to break into some Rockey-protected software, I was interesting only in limits (which algorithms could we find), so I make some things simpler: we will work with only 4 16-bit registers, and there will be only 6 operations (add, subtract, multiply, OR, XOR, AND).

Let's first calculate, how much information will be used in brute-force case.

There are 384 of all possible instructions in `reg=reg,op,reg` format for 4 registers and 6 operations, and also 6144 instructions in `reg=reg,op,constant` format. Remember that constant limited to 63 as maximal value? That help us for a little.

So, there are 6528 of all possible instructions. This mean, there are $\approx 1.1 \cdot 10^{19}$ 5-instruction algorithms. That's too much.

How can we express each instruction as system of equations? While remembering some school mathematics, I wrote this:

```
Function one\_step()=

# Each Bx is integer, but may be only 0 or 1.

# only one of B1..B4 and B5..B9 can be set
reg1=B1*A + B2*B + B3*C + B4*D
reg_or_constant2=B5*A + B6*B + B7*C + B8*D + B9*constant
reg1 should not be equal to reg_or_constant2

# Only one of B10..B15 can be set
result=result+B10*(reg1*reg2)
result=result+B11*(reg1^reg2)
result=result+B12*(reg1+reg2)
result=result+B13*(reg1-reg2)
result=result+B14*(reg1|reg2)
```

[91] http://www.rockey.nl/en/rockey.html

```
result=result+B15*(reg1&reg2)

B16 - true if register isn't updated in this part
B17 - true if register is updated in this part
(B16 cannot be equal to B17)
A=B16*A + B17*result
B=B18*A + B19*result
C=B20*A + B21*result
D=B22*A + B23*result
```

That's how we can express each instruction in algorithm.

5-instructions algorithm can be expressed like this:
`one_step (one_step (one_step (one_step (one_step (input_registers))))).`

Let's also add five known input/output pairs and we'll get system of equations like this:

```
one_step (one_step (one_step (one_step (one_step (input_1)))))==output_1
one_step (one_step (one_step (one_step (one_step (input_2)))))==output_2
one_step (one_step (one_step (one_step (one_step (input_3)))))==output_3
one_step (one_step (one_step (one_step (one_step (input_4)))))==output_4
.. etc
```

So the question now is to find $5 \cdot 23$ boolean values satisfying known input/output pairs.

I wrote small utility to probe Rockey 4 algorithm with random numbers, it produce results in form:

```
RY_CALCULATE1: (input) p1=30760 p2=18484 p3=41200 p4=61741 (output) p1=49244 p2=11312 p3
    =27587 p4=12657
RY_CALCULATE1: (input) p1=51139 p2=7852 p3=53038 p4=49378 (output) p1=58991 p2=34134 p3
    =40662 p4=9869
RY_CALCULATE1: (input) p1=60086 p2=52001 p3=13352 p4=45313 (output) p1=46551 p2=42504 p3
    =61472 p4=1238
RY_CALCULATE1: (input) p1=48318 p2=6531 p3=51997 p4=30907 (output) p1=54849 p2=20601 p3
    =31271 p4=44794
```

p1/p2/p3/p4 are just another names for A/B/C/D registers.

Now let's start with Z3. We will need to express Rockey 4 toy CPU in Z3Py (Z3 Python API) terms.

It can be said, my Python script is divided into two parts:

- constraint definitions (like, *output_1 should be n for input_1=m*, *constant cannot be greater than 63*, etc);
- functions constructing system of equations.

This piece of code define some kind of *structure* consisting of 4 named 16-bit variables, each represent register in our toy CPU.

```
Registers_State=Datatype ('Registers_State')
Registers_State.declare('cons', ('A', BitVecSort(16)), ('B', BitVecSort(16)), ('C',
    BitVecSort(16)), ('D', BitVecSort(16)))
Registers_State=Registers_State.create()
```

These enumerations define two new types (or *sorts* in Z3's terminology):

```
Operation, (OP_MULT, OP_MINUS, OP_PLUS, OP_XOR, OP_OR, OP_AND) = EnumSort('Operation',
    ('OP_MULT', 'OP_MINUS', 'OP_PLUS', 'OP_XOR', 'OP_OR', 'OP_AND'))

Register, (A, B, C, D) = EnumSort('Register', ('A', 'B', 'C', 'D'))
```

This part is very important, it defines all variables in our system of equations. `op_step` is type of operation in instruction. `reg_or_constant` is selector between register and constant in second argument — *False* if it's a register and *True* if it's a constant. `reg_step` is a destination register of this instruction. `reg1_step` and `reg2_step` are just registers at arg1 and arg2. `constant_step` is constant (in case it's used in instruction instead of arg2).

```
op_step=[Const('op_step%s' % i, Operation) for i in range(STEPS)]
reg_or_constant_step=[Bool('reg_or_constant_step%s' % i) for i in range(STEPS)]
reg_step=[Const('reg_step%s' % i, Register) for i in range(STEPS)]
reg1_step=[Const('reg1_step%s' % i, Register) for i in range(STEPS)]
reg2_step=[Const('reg2_step%s' % i, Register) for i in range(STEPS)]
constant_step = [BitVec('constant_step%s' % i, 16) for i in range(STEPS)]
```

Adding constraints is very simple. Remember, I wrote that each constant cannot be larger than 63?

```
# according to Rockey 4 dongle manual, arg2 in first and last instructions cannot be a
   constant
s.add (reg_or_constant_step[0]==False)
s.add (reg_or_constant_step[STEPS-1]==False)

...

for x in range(STEPS):
   s.add (constant_step[x]>=0, constant_step[x]<=63)
```

Known input/output values are added as constraints too.

Now let's see how to construct our system of equations:

```
# Register, Registers_State -> int
def register_selector (register, input_registers):
    return If(register==A, Registers_State.A(input_registers),
           If(register==B, Registers_State.B(input_registers),
           If(register==C, Registers_State.C(input_registers),
           If(register==D, Registers_State.D(input_registers),
                           0)))) # default
```

This function returning corresponding register value from *structure*. Needless to say, the code above is not executed. `If()` is Z3Py function. The code only declares the function, which will be used in another. Expression declaration resembling LISP PL in some way.

Here is another function where `register_selector()` is used:

```
# Bool, Register, Registers_State, int -> int
def register_or_constant_selector (register_or_constant, register, input_registers,
   constant):
    return If(register_or_constant==False, register_selector(register, input_registers),
        constant)
```

The code here is never executed too. It only constructs one small piece of very big expression. But for the sake of simplicity, one can think all these functions will be called during bruteforce search, many times, at fastest possible speed.

```
# Operation, Bool, Register, Register, Int, Registers_State -> int
def one_op (op, register_or_constant, reg1, reg2, constant, input_registers):
    arg1=register_selector(reg1, input_registers)
    arg2=register_or_constant_selector (register_or_constant, reg2, input_registers,
       constant)
    return If(op==OP_MULT,   arg1*arg2,
           If(op==OP_MINUS,  arg1-arg2,
           If(op==OP_PLUS,   arg1+arg2,
           If(op==OP_XOR,    arg1^arg2,
           If(op==OP_OR,     arg1|arg2,
           If(op==OP_AND,    arg1&arg2,
                          0)))))) # default
```

Here is the expression describing each instruction. `new_val` will be assigned to destination register, while all other registers' values are copied from input registers' state:

```
# Bool, Register, Operation, Register, Register, Int, Registers_State -> Registers_State
def one_step (register_or_constant, register_assigned_in_this_step, op, reg1, reg2,
    constant, input_registers):
    new_val=one_op(op, register_or_constant, reg1, reg2, constant, input_registers)
    return If (register_assigned_in_this_step==A, Registers_State.cons (new_val,
                                                                        Registers_State.
                                                                            B(
                                                                            input_registers
                                                                            ),
                                                                        Registers_State.
                                                                            C(
                                                                            input_registers
                                                                            ),
                                                                        Registers_State.
                                                                            D(
                                                                            input_registers
                                                                            )),
            If (register_assigned_in_this_step==B, Registers_State.cons (Registers_State.
                A(input_registers),
                                                                        new_val,
                                                                        Registers_State.
                                                                            C(
                                                                            input_registers
                                                                            ),
                                                                        Registers_State.
                                                                            D(
                                                                            input_registers
                                                                            )),
            If (register_assigned_in_this_step==C, Registers_State.cons (Registers_State.
                A(input_registers),
                                                                        Registers_State.
                                                                            B(
                                                                            input_registers
                                                                            ),
                                                                        new_val,
                                                                        Registers_State.
                                                                            D(
                                                                            input_registers
                                                                            )),
            If (register_assigned_in_this_step==D, Registers_State.cons (Registers_State.
                A(input_registers),
                                                                        Registers_State.
                                                                            B(
                                                                            input_registers
                                                                            ),
                                                                        Registers_State.
                                                                            C(
                                                                            input_registers
                                                                            ),
                                                                        new_val),
                                                    Registers_State.cons(0,0,0,0))))) #
                                                        default
```

This is the last function describing a whole $n$-step program:

```
def program(input_registers, STEPS):
    cur_input=input_registers
```

```
    for x in range(STEPS):
        cur_input=one_step (reg_or_constant_step[x], reg_step[x], op_step[x], reg1_step[
            x], reg2_step[x], constant_step[x], cur_input)
    return cur_input
```

Again, for the sake of simplicity, it can be said, now Z3 will try each possible registers/operations/constants against this expression to find such combination which satisfy all input/output pairs. Sounds absurdic, but this is close to reality. SAT/SMT-solvers indeed tries them all. But the trick is to prune search tree as early as possible, so it will work for some reasonable time. And this is hardest problem for solvers.

Now let's start with very simple 3-step algorithm: `B=A^D, C=D*D, D=A*C`. Please note: register `A` left unchanged. I programmed Rockey 4 dongle with the algorithm, and recorded algorithm outputs are:

```
RY_CALCULATE1: (input) p1=8803 p2=59946 p3=36002 p4=44743 (output) p1=8803 p2=36004 p3
    =7857 p4=24691
RY_CALCULATE1: (input) p1=5814 p2=55512 p3=52155 p4=55813 (output) p1=5814 p2=52403 p3
    =33817 p4=4038
RY_CALCULATE1: (input) p1=25206 p2=2097 p3=55906 p4=22705 (output) p1=25206 p2=15047 p3
    =10849 p4=43702
RY_CALCULATE1: (input) p1=10044 p2=14647 p3=27923 p4=7325 (output) p1=10044 p2=15265 p3
    =47177 p4=20508
RY_CALCULATE1: (input) p1=15267 p2=2690 p3=47355 p4=56073 (output) p1=15267 p2=57514 p3
    =26193 p4=53395
```

It took about one second and only 5 pairs above to find algorithm (on my quad-core Xeon E3-1220 3.1GHz, however, Z3 solver working in single-thread mode):

```
B = A ^ D
C = D * D
D = C * A
```

Note the last instruction: `C` and `A` registers are swapped comparing to version I wrote by hand. But of course, this instruction is working in the same way, because multiplication is commutative operation.

Now if I try to find 4-step program satisfying to these values, my script will offer this:

```
B = A ^ D
C = D * D
D = A * C
A = A | A
```

…and that's really fun, because the last instruction do nothing with value in register `A`, it's like NOP[92]—but still, algorithm is correct for all values given.

Here is another 5-step algorithm: `B=B^D, C=A*22, A=B*19, A=A&42, D=B&C` and values:

```
RY_CALCULATE1: (input) p1=61876 p2=28737 p3=28636 p4=50362 (output) p1=32 p2=46331 p3
    =50552 p4=33912
RY_CALCULATE1: (input) p1=46843 p2=43355 p3=39078 p4=24552 (output) p1=8 p2=63155 p3
    =47506 p4=45202
RY_CALCULATE1: (input) p1=22425 p2=51432 p3=40836 p4=14260 (output) p1=0 p2=65372 p3
    =34598 p4=34564
RY_CALCULATE1: (input) p1=44214 p2=45766 p3=19778 p4=59924 (output) p1=2 p2=22738 p3
    =55204 p4=20608
RY_CALCULATE1: (input) p1=27348 p2=49060 p3=31736 p4=59576 (output) p1=0 p2=22300 p3
    =11832 p4=1560
```

It took 37 seconds and we've got:

```
B = D ^ B
C = A * 22
```

[92]No Operation

```
A = B * 19
A = A & 42
D = C & B
```

A=A&42 was correctly deduced (look at these five p1's at output (assigned to output A register): 32,8,0,2,0)

6-step algorithm A=A+B, B=C*13, D=D^A, C=C&A, D=D|B, A=A&B and values:

```
RY_CALCULATE1: (input) p1=4110 p2=35411 p3=54308 p4=47077 (output) p1=32832 p2=50644 p3
    =36896 p4=60884
RY_CALCULATE1: (input) p1=12038 p2=7312 p3=39626 p4=47017 (output) p1=18434 p2=56386 p3
    =2690 p4=64639
RY_CALCULATE1: (input) p1=48763 p2=27663 p3=12485 p4=20563 (output) p1=10752 p2=31233 p3
    =8320 p4=31449
RY_CALCULATE1: (input) p1=33174 p2=38937 p3=54005 p4=38871 (output) p1=4129 p2=46705 p3
    =4261 p4=48761
RY_CALCULATE1: (input) p1=46587 p2=36275 p3=6090 p4=63976 (output) p1=258 p2=13634 p3
    =906 p4=48966
```

90 seconds and we've got:

```
A = A + B
B = C * 13
D = D ^ A
D = B | D
C = C & A
A = B & A
```

But that was simple, however. Some 6-step algorithms are not possible to find, for example: A=A^B, A=A*9, A=A^C, A=A*19, A=A^D, A=A&B. Solver was working too long (up to several hours), so I didn't even know is it possible to find it anyway.

#### 13.2.1 Conclusion

This is in fact an exercise in program synthesis.

Some short algorithms for tiny CPUs are really possible to find using so small set set of data. Of course it's still not possible to reveal some complex algorithm, but this method definitely should not be ignored.

#### 13.2.2 The files

Rockey 4 dongle programmer and reader, Rockey 4 manual, Z3Py script for finding algorithms, input/output pairs: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/pgm_synth/rockey.

#### 13.2.3 Further work

Perhaps, constructing LISP-like S-expression can be better than a program for toy-level CPU.

It's also possible to start with smaller constants and then proceed to bigger. This is somewhat similar to increasing password length in password brute-force cracking.

#### 13.2.4 Exercise

https://challenges.re/25/.

### 13.3 TAOCP 7.1.3 Exercise 198, UTF-8 encoding and program synthesis by sketching

Found this exercise in TAOCP 7.1.3 (Bitwise Tricks and Techniques):

▶ **198.** [*21*] Unicode characters are often represented as strings of bytes using a scheme called UTF-8, which is the encoding of exercise 196 restricted to integers in the range $0 \le x < 2^{20}+2^{16}$. Notice that UTF-8 efficiently preserves the standard ASCII character set (the codepoints with $x < 2^7$), and that it is quite different from UTF-16.

Let $\alpha_1$ be the first byte of a UTF-8 string $\alpha(x)$. Show that there are reasonably small integer constants $a$, $b$, and $c$ such that only four bitwise operations

$$(a \gg ((\alpha_1 \gg b)\ \&\ c))\ \&\ 3$$

suffice to determine the number $l-1$ of bytes between $\alpha_1$ and the end of $\alpha(x)$.

Figure 34: Exercise from TAOCP book

This is like program synthesis by sketching: you give a sketch with several "holes" missing and ask some automated software to fill the "holes". In our case, $a$, $b$ and $c$ are "holes".

Let's find them using Z3:

```
from z3 import *

a, b, c=BitVecs('a b c', 22)

s=Solver()

def bytes_in_UTF8_seq(x):
    if (x>>7)==0:
        return 1
    if (x>>5)==0b110:
        return 2
    if (x>>4)==0b1110:
        return 3
    if (x>>3)==0b11110:
        return 4
    # invalid 1st byte
    return None

for x in range(256):
    t=bytes_in_UTF8_seq(x)
    if t!=None:
        s.add(((a >> ((x>>b) & c)) & 3) == (t-1))

# enumerate all solutions:
results=[]
while s.check() == sat:
    m = s.model()

    print "a,b,c = 0x%x 0x%x 0x%x" % (m[a].as_long(), m[b].as_long(), m[c].as_long())

    results.append(m)
    block = []
    for d in m:
        t=d()
        block.append(t != m[d])
    s.add(Or(block))
```

```
print "results total=", len(results)
```

I tried various bit widths for a, b and c and found that 22 bits are enough. I've lots of results like:

```
...

a,b,c = 0x250100 0x3 0x381416
a,b,c = 0x258100 0x3 0x381416
a,b,c = 0x258900 0x3 0x381416
a,b,c = 0x250900 0x3 0x381416
a,b,c = 0x251100 0x3 0x381416
a,b,c = 0x259100 0x3 0x381416
a,b,c = 0x259100 0x3 0x389416
a,b,c = 0x251100 0x3 0x389416
a,b,c = 0x251100 0x3 0x189416
a,b,c = 0x259100 0x3 0x189416
a,b,c = 0x259100 0x3 0x189016

...
```

It seems that several least significant bits of a and c are not used. After little experimentation, I've come to this:

```
...

# make a,c more aesthetically appealing:
s.add((a&0xffff)==0)
s.add((c&0xffff00)==0)

...
```

And the results:

```
a,b,c = 0x250000 0x3 0x36
a,b,c = 0x250000 0x3 0x16
a,b,c = 0x250000 0x3 0x96
a,b,c = 0x250000 0x3 0xd6
a,b,c = 0x250000 0x3 0xf6
a,b,c = 0x250000 0x3 0x76
a,b,c = 0x250000 0x3 0xb6
a,b,c = 0x250000 0x3 0x56
results total= 8
```

Pick any.

But how it works? Its operation is very similar to the bitwise trick related to leading/trailing zero bits counting based on De Bruijn sequences. Read more about it: https://yurichev.com/blog/de_bruijn/, 3.18.

## 13.4 TAOCP 7.1.3 Exercise 203, MMIX MOR instruction and program synthesis by sketching

Found this exercise in TAOCP 7.1.3 (Bitwise Tricks and Techniques):

**203.** [*22*] Suppose we want to convert a tetrabyte $x = (x_7 \ldots x_1 x_0)_{16}$ to the octabyte $y = (y_7 \ldots y_1 y_0)_{256}$, where $y_j$ is the ASCII code for the hexadecimal digit $x_j$. For example, if $x = {}^{\#}$`1234abcd`, $y$ should represent the 8-character string `"1234abcd"`. What clever choices of five constants `a`, `b`, `c`, `d`, and `e` will make the following MMIX instructions do the job?

```
MOR t,x,a;  SLU s,t,4;  XOR t,s,t;  AND t,t,b;
ADD t,t,c;  MOR s,d,t;  ADD t,t,e;  ADD y,t,s.
```

Figure 35: Screenshot from TAOCP book

What is MOR instruction in MMIX?

- `MOR $X,$Y,$Z|Z` 'multiple or'.

Suppose the 64 bits of register Y are indexed as

$$y_{00}y_{01}\ldots y_{07}y_{10}y_{11}\ldots y_{17}\ldots y_{70}y_{71}\ldots y_{77};$$

in other words, $y_{ij}$ is the $j$th bit of the $i$th byte, if we number the bits and bytes from 0 to 7 in big-endian fashion from left to right. Let the bits of the other operand, \$Z or Z, be indexed similarly:

$$z_{00}z_{01}\ldots z_{07}z_{10}z_{11}\ldots z_{17}\ldots z_{70}z_{71}\ldots z_{77}.$$

The `MOR` operation replaces each bit $x_{ij}$ of register X by the bit

$$y_{0j}z_{i0} \vee y_{1j}z_{i1} \vee \cdots \vee y_{7j}z_{i7}.$$

Thus, for example, if register Z contains the constant #`0102040810204080`, `MOR` reverses the order of the bytes in register Y, converting between little-endian and big-endian addressing. (The $i$th byte of \$X depends on the bytes of \$Y as specified by the $i$th byte of \$Z or Z. If we regard 64-bit words as $8 \times 8$ Boolean matrices, with one byte per column, this operation computes the Boolean product \$X = \$Y \$Z or \$X = \$Y Z. Alternatively, if we regard 64-bit words as $8 \times 8$ matrices with one byte per *row*, `MOR` computes the Boolean product \$X = \$Z \$Y or \$X = Z \$Y with operands in the opposite order. The immediate form `MOR $X,$Y,Z` always sets the leading seven bytes of register X to zero; the other byte is set to the bitwise or of whatever bytes of register Y are specified by the immediate operand Z.)

Exercise: Explain how to compute a mask `m` that is #`ff` in byte positions where `a` exceeds `b`, #`00` in all other bytes. Answer: `BDIF x,a,b; MOR m,minusone,x;` here `minusone` is a register consisting of all 1s. (Moreover, if we `AND` this result with #`8040201008040201`, then `MOR` with Z = 255, we get a one-byte encoding of `m`.)

Figure 36: Screenshot from the MMIX book

( http://mmix.cs.hm.edu/doc/mmix-doc.pdf )

Let's try to solve. We create two functions. First has MOR instructions simulation + the program from TAOCP. The second is a naive implementation. Then we add "forall" quantifier: for all inputs, both functions must produce the same result. **But**, we don't know a/b/c/d/e/f and ask Z3 SMT-solver to

```
from z3 import *

s=Solver()

a, b, c, d, e=BitVecs('a b c d e', 64)
```

```
def simulate_MOR(y,z):
    """
    set each bit of 64-bit result, as:

    $x_i_j = y_0_j z_i_0 \vee y_1_j z_i_1 \vee \cdots \vee y_7_j z_i_7$
    https://latexbase.com/d/bf2243f8-5d0b-4231-8891-66fb47d846f0

    IOW:

    x<byte><bit> = (y<0><bit> AND z<byte><0>) OR (y<1><bit> AND z<byte><1>) OR ... OR (y
        <7><bit> AND z<byte><7>)
    """

    def get_ij(x, i, j):
        return (x>>(i*8+j))&1

    rt=0
    for byte in range(8):
        for bit in range(8):
            t=0
            for i in range(8):
                t|=get_ij(y, i, bit) & get_ij(z, byte, i)

            pos=byte*8+bit
            rt|=t<<pos

    return rt

def simulate_pgm(x):
    t=simulate_MOR(x,a)
    s=t<<4
    t=s^t
    t=t&b
    t=t+c
    s=simulate_MOR(d,t)
    t=t+e
    y=t+s
    return y

def nibble_to_ASCII(x):
    return If(And(x>=0, x<=9), 0x30+x, 0x61+x-10)

def method2(x):
    rt=0
    for i in range(8):
        rt|=nibble_to_ASCII((x >> i*4)&0xf) << i*8
    return rt

#"""
# new version.
# for all possible 32-bit x's, find such a/b/c/d/e, so that these two parts would be
    equal to each other
# zero extend x to 64-bit value in both cases
x=BitVec('x', 32)
s.add(ForAll([x], simulate_pgm(ZeroExt(32, x))==method2(ZeroExt(32, x))))
```

```
#"""

"""
# previous version:
for i in range(5):
    x=random.getrandbits(32)
    t="%08x" % x
    y=int(''.join("%02X" % ord(c) for c in t), 16)
    print "%x %x" % (x, y)

    s.add(simulate_pgm(x)==y)
"""

# enumerate all solutions:
results=[]
while s.check() == sat:
    m = s.model()

    print "a,b,c,d,e = %x %x %x %x %x" % (m[a].as_long(), m[b].as_long(), m[c].as_long()
        , m[d].as_long(), m[e].as_long())

    results.append(m)
    block = []
    for d1 in m:
        t=d1()
        block.append(t != m[d1])
    s.add(Or(block))

print "results total=", len(results)
```

Very slow, it takes several hours on my venerable Intel Quad-Core Xeon E3-1220 3.10GHz but found at least one solution:

```
a,b,c,d,e = 8000400020001 f0f0f0f0f0f0f0f 56d656d616969616 411a00000000 bf3fbf3fff7f8000
```

...which is correct (I've wrote bruteforce checker, here: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/pgm_synth/TAOCP_713_203/check.c.

D.Knuth's TAOCP also has answers:

> **203.** a = #0008000400020001, b = #0f0f0f0f0f0f0f0f, c = #0606060606060606, d = #0000002700000000, e = #2a2a2a2a2a2a2a2a. (The ASCII code for 0 is 6 + #2a; the ASCII code for a is 6 + #2a + 10 + #27.)

Figure 37: Screenshot from TAOCP book

...which are different, but also correct.

What if a==0x0008000400020001 always? I'm adding a new constraint:

```
s.add(a==0x0008000400020001)
```

We've getting many results (much faster, and also correct):

```
...

a,b,c,d,e = 8000400020001 7f0fcf0fcf0f7f0f 1616d6d656561656 8680522903020000
    eeeda9aa2e2eee2f
a,b,c,d,e = 8000400020001 7f0fcf0fcf0f6f0f 1616d6d656561656 8680522903020000
    eeeda9aa2e2eee2f
```

```
a,b,c,d,e = 8000400020001 7f0fcf0fdf0f6f0f 1616d6d656561656 8680522903020000
    eeeda9aa2e2eee2f
a,b,c,d,e = 8000400020001 5f0fcf0fdf0f6f0f 1616d6d656561656 8680522903020000
    eeeda9aa2e2eee2f

...
```

The files: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/pgm_synth/TAOCP_713_203.

# 14 Logic circuits synthesis

## 14.1 Simple logic synthesis using Z3 and Apollo Guidance Computer

What a smallest possible logic circuit is possible for a given truth table?

Let's try to define truth tables for inputs and outputs and find smallest circuit. This program is almost the same as I mentioned earlier: 13.1, but reworked slightly:

```
#!/usr/bin/env python
from z3 import *
import sys

I_AND, I_OR, I_XOR, I_NOT, I_NOR3 = 0,1,2,3,4

# 1-bit NOT
"""
INPUTS=[0b10]
OUTPUTS=[0b01]
BITS=2
add_always_false=False
add_always_true=True
avail=[I_XOR]
#avail=[I_NOR3]
"""

# 2-input AND
"""
INPUTS=[0b1100, 0b1010]
OUTPUTS=[0b1000]
BITS=4
add_always_false=False
add_always_true=False
avail=[I_OR, I_NOT]
#avail=[I_NOR3]
"""

# 2-input XOR
#"""
INPUTS=[0b1100, 0b1010]
OUTPUTS=[0b0110]
BITS=4
add_always_false=False
#add_always_false=True
add_always_true=False
#add_always_true=True
#avail=[I_NOR3]
#avail=[I_AND, I_NOT]
avail=[I_OR, I_NOT]
```

```
#"""

# parity (or popcnt1)
""" TT for parity
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 0
"""
"""
INPUTS=[0b11110000, 0b11001100, 0b10101010]
OUTPUTS=[0b00010110]
BITS=8
add_always_false=False
add_always_true=False
#add_always_false=True
#add_always_true=True
avail=[I_AND, I_XOR, I_OR, I_NOT]
#avail=[I_XOR, I_OR, I_NOT]
#avail=[I_AND, I_OR, I_NOT]
"""

# full-adder
"""
INPUTS=[0b11110000, 0b11001100, 0b10101010]
OUTPUTS=[0b11101000, 0b10010110] # carry-out, sum
BITS=8
add_always_false=False
add_always_true=False
avail=[I_AND, I_OR, I_XOR, I_NOT]
#avail=[I_AND, I_OR, I_NOT]
#avail=[I_NOR3]
"""

# popcnt
""" TT for popcnt

in  HL
000 00
001 01
010 01
011 10
100 01
101 10
110 10
111 11
"""
"""
INPUTS=[0b11110000, 0b11001100, 0b10101010]
OUTPUTS=[0b11101000, 0b10010110] # high, low
BITS=8
add_always_false=False
```

```
add_always_true=False
#avail=[I_AND, I_OR, I_XOR, I_NOT]
avail=[I_NOR3]
"""

# majority for 3 bits
""" TT for majority (true if 2 or 3 bits are True)
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
"""
"""
INPUTS=[0b11110000, 0b11001100, 0b10101010]
OUTPUTS=[0b11101000]
BITS=8
add_always_false=False
add_always_true=False
avail=[I_AND, I_OR, I_XOR, I_NOT]
"""

# 2-bit adder:
"""
INPUTS=[0b1111111100000000, 0b1111000011110000, 0b1100110011001100, 0b1010101010101010]
OUTPUTS=[0b1001001101101100, 0b0101101001011010] # high, low
BITS=16
add_always_false=True
add_always_true=True
#add_always_false=False
#add_always_true=False
avail=[I_AND, I_OR, I_XOR, I_NOT]
#avail=[I_NOR3]
"""

"""
#7-segment display
INPUTS=[0b1111111100000000, 0b1111000011110000, 0b1100110011001100, 0b1010101010101010]
# "g" segment, like here: http://www.nutsvolts.com/uploads/wygwam/
   NV_0501_Marston_Figure02.jpg
OUTPUTS=[0b1110111101111100] # g
BITS=16
add_always_false=False
add_always_true=False
avail=[I_AND, I_OR, I_XOR, I_NOT]
#avail=[I_NOR3]
"""

MAX_STEPS=20

# if additional always-false or always-true must be present:
if add_always_false:
    INPUTS.append(0)
```

```
if add_always_true:
    INPUTS.append(2**BITS-1)

# this called during self-testing:
def eval_ins(R, s, m, STEPS, op, op1_reg, op2_reg, op3_reg):
    op_n=m[op[s]].as_long()
    op1_reg_tmp=m[op1_reg[s]].as_long()
    op1_val=R[op1_reg_tmp]
    op2_reg_tmp=m[op2_reg[s]].as_long()
    op3_reg_tmp=m[op3_reg[s]].as_long()
    if op_n in [I_AND, I_OR, I_XOR, I_NOR3]:
        op2_val=R[op2_reg_tmp]
    if op_n==I_AND:
        return op1_val&op2_val

    elif op_n==I_OR:
        return op1_val|op2_val

    elif op_n==I_XOR:
        return op1_val^op2_val

    elif op_n==I_NOT:
        return ~op1_val

    elif op_n==I_NOR3:
        op3_val=R[op3_reg_tmp]
        return ~(op1_val|op2_val|op3_val)

    else:
        raise AssertionError

# evaluate program we've got. for self-testing.
def eval_pgm(m, STEPS, op, op1_reg, op2_reg, op3_reg):
    R=[None]*STEPS
    for i in range(len(INPUTS)):
        R[i]=INPUTS[i]

    for s in range(len(INPUTS),STEPS):
        R[s]=eval_ins(R, s, m, STEPS, op, op1_reg, op2_reg, op3_reg)

    return R

# get all states, for self-testing:
def selftest(m, STEPS, op, op1_reg, op2_reg, op3_reg):
    l=eval_pgm(m, STEPS, op, op1_reg, op2_reg, op3_reg)
    print "simulate:"
    for i in range(len(l)):
        print "r%d=" % i, format(l[i] & 2**BITS-1, '0'+str(BITS)+'b')

"""
selector() functions generates expression like:

If(op1_reg_s5 == 0,
   S_s0,
   If(op1_reg_s5 == 1,
      S_s1,
```

```
      If(op1_reg_s5 == 2,
         S_s2,
         If(op1_reg_s5 == 3,
            S_s3,
            If(op1_reg_s5 == 4,
               S_s4,
               If(op1_reg_s5 == 5,
                  S_s5,
                  If(op1_reg_s5 == 6,
                     S_s6,
                     If(op1_reg_s5 == 7,
                        S_s7,
                        If(op1_reg_s5 == 8,
                           S_s8,
                           If(op1_reg_s5 == 9,
                              S_s9,
                              If(op1_reg_s5 == 10,
                                 S_s10,
                                 If(op1_reg_s5 == 11,
                                    S_s11,
                                    0))))))))))))

this is like multiplexer or switch()
"""
def selector(R, s):
    t=0 # default value
    for i in range(MAX_STEPS):
        t=If(s==(MAX_STEPS-i-1), R[MAX_STEPS-i-1], t)
    return t

def simulate_op(R, op, op1_reg, op2_reg, op3_reg):
    op1_val=selector(R, op1_reg)
    return If(op==I_AND, op1_val & selector(R, op2_reg),
           If(op==I_OR, op1_val | selector(R, op2_reg),
           If(op==I_XOR, op1_val ^ selector(R, op2_reg),
           If(op==I_NOT, ~op1_val,
           If(op==I_NOR3, ~(op1_val | selector(R, op2_reg) | selector (R, op3_reg)),
               0))))) # default

op_to_str_tbl=["AND", "OR", "XOR", "NOT", "NOR3"]

def print_model(m, R, STEPS, op, op1_reg, op2_reg, op3_reg):
    print "%d instructions" % (STEPS-len(INPUTS))
    for s in range(STEPS):
        if s<len(INPUTS):
            t="r%d=input" % s
        else:
            op_n=m[op[s]].as_long()
            op_s=op_to_str_tbl[op_n]
            op1_reg_n=m[op1_reg[s]].as_long()
            op2_reg_n=m[op2_reg[s]].as_long()
            op3_reg_n=m[op3_reg[s]].as_long()
            if op_n in [I_AND, I_OR, I_XOR]:
                t="r%d=%s r%d, r%d" % (s, op_s, op1_reg_n, op2_reg_n)
            elif op_n==I_NOT:
                t="r%d=%s r%d" % (s, op_s, op1_reg_n)
```

```
            else: # else NOR3
                t="r%d=%s r%d, r%d, r%d" % (s, op_s, op1_reg_n, op2_reg_n, op3_reg_n)

        tt=format(m[R[s]].as_long(), '0'+str(BITS)+'b')
        print t+" "*(25-len(t))+tt

def attempt(STEPS):
    print "attempt, STEPS=", STEPS
    sl=Solver()

    # state of each register:
    R=[BitVec(('S_s%d' % s), BITS) for s in range(MAX_STEPS)]
    # operation type and operands for each register:
    op=[Int('op_s%d' % s) for s in range(MAX_STEPS)]
    op1_reg=[Int('op1_reg_s%d' % s) for s in range(MAX_STEPS)]
    op2_reg=[Int('op2_reg_s%d' % s) for s in range(MAX_STEPS)]
    op3_reg=[Int('op3_reg_s%d' % s) for s in range(MAX_STEPS)]

    for s in range(len(INPUTS), STEPS):
        # for each step.
        # expression like Or(op[s]==0, op[s]==1, ...) is formed here. values are taken
           from avail[]
        sl.add(Or(*[op[s]==j for j in avail]))
        # each operand can refer to one of registers BEFORE the current instruction:
        sl.add(And(op1_reg[s]>=0, op1_reg[s]<s))
        sl.add(And(op2_reg[s]>=0, op2_reg[s]<s))
        sl.add(And(op3_reg[s]>=0, op3_reg[s]<s))

    # fill inputs:
    for i in range(len(INPUTS)):
        sl.add(R[i]==INPUTS[i])
    # fill outputs, "must be's"
    for o in range(len(OUTPUTS)):
        sl.add(R[STEPS-(o+1)]==list(reversed(OUTPUTS))[o])

    # connect each register to "simulator":
    for s in range(len(INPUTS), STEPS):
        sl.add(R[s]==simulate_op(R, op[s], op1_reg[s], op2_reg[s], op3_reg[s]))

    tmp=sl.check()
    if tmp==sat:
        print "sat!"
        m=sl.model()
        #print m
        print_model(m, R, STEPS, op, op1_reg, op2_reg, op3_reg)
        selftest(m, STEPS, op, op1_reg, op2_reg, op3_reg)
        exit(0)
    else:
        print tmp

for s in range(len(INPUTS)+len(OUTPUTS), MAX_STEPS):
    attempt(s)
```

I could generate only small circuits maybe up to $\approx 10$ gates, but this is interesting nonetheless.

Also, I've always wondering how you can do something usable for Apollo Guidance Computer, which had only one single gate: NOR3? See also its schematics: http://klabs.org/history/ech/agc_schematics/. The answer is De Morgan's laws, but this is not obvious.

INPUTS[] has all possible bit combinations for all inputs, or all possible truth tables. OUTPUTS[] has truth table for each output. All the rest is processed in bitsliced manner. Given that, the resulting program will work on 4/8/16-bit CPU and will generate defined OUTPUTS for defined INPUTS. Or, this program can be treated just like a logic circuit.

### 14.1.1 AND gate

How to build 2-input AND gate using only OR's and NOT's?

```
INPUTS=[0b1100, 0b1010]
OUTPUTS=[0b1000]
BITS=4
avail=[I_OR, I_NOT]

...

r0=input                  1100
r1=input                  1010
r2=NOT r1                 0101
r3=NOT r0                 0011
r4=OR r3, r2              0111
r5=NOT r4                 1000
```

This is indeed like stated in De Morgan's laws: $x \land y$ is equivalent to $\neg(\neg x \lor \neg y)$. Can be used for obfuscation?

Now using only NOR3 gate?

```
avail=[I_NOR3]

...

r0=input                  1100
r1=input                  1010
r2=NOR3 r1, r1, r1        0101
r3=NOR3 r2, r0, r0        0010
r4=NOR3 r3, r2, r2        1000
```

### 14.1.2 XOR gate

How to build 2-input XOR using only OR's and NOT's?

```
INPUTS=[0b1100, 0b1010]
OUTPUTS=[0b0110]
BITS=4
avail=[I_OR, I_NOT]

...

7 instructions
r0=input                  1100
r1=input                  1010
r2=OR r1, r0              1110
r3=NOT r2                 0001
r4=OR r0, r3              1101
r5=NOT r4                 0010
r6=OR r1, r3              1011
r7=NOT r6                 0100
r8=OR r5, r7              0110
```

... using only AND's and NOT's?

```
avail=[I_AND, I_NOT]
...
r0=input                    1100
r1=input                    1010
r2=NOT r1                   0101
r3=AND r1, r0               1000
r4=NOT r3                   0111
r5=NOT r0                   0011
r6=AND r2, r5               0001
r7=NOT r6                   1110
r8=AND r4, r7               0110
```

... using only NOR3 gates?

```
avail=[I_NOR3]
...
r0=input                    1100
r1=input                    1010
r2=NOR3 r1, r1, r1          0101
r3=NOR3 r0, r0, r1          0001
r4=NOR3 r0, r2, r3          0010
r5=NOR3 r2, r4, r2          1000
r6=NOR3 r3, r5, r3          0110
```

### 14.1.3 Full-adder

According to Wikipedia, full-adder can be constructed using two XOR gates, two AND gates and one OR gate. But I had no idea 3 XORs and 2 ANDs can be used instead:

```
INPUTS=[0b11110000, 0b11001100, 0b10101010]
OUTPUTS=[0b11101000, 0b10010110] # carry-out, sum
BITS=8
avail=[I_AND, I_OR, I_XOR, I_NOT]

...

5 instructions
r0=input                    11110000
r1=input                    11001100
r2=input                    10101010
r3=XOR r2, r1               01100110
r4=AND r0, r3               01100000
r5=AND r2, r1               10001000
r6=XOR r4, r5               11101000
r7=XOR r0, r3               10010110
```

... using only NOR3 gates:

```
avail=[I_NOR3]
...
8 instructions
r0=input                    11110000
r1=input                    11001100
r2=input                    10101010
r3=NOR3 r0, r0, r1          00000011
r4=NOR3 r2, r3, r1          00010000
r5=NOR3 r3, r2, r0          00000100
```

```
r6=NOR3 r3, r0, r5        00001000
r7=NOR3 r5, r2, r4        01000001
r8=NOR3 r3, r4, r1        00100000
r9=NOR3 r3, r5, r4        11101000
r10=NOR3 r8, r7, r6       10010110
```

### 14.1.4 POPCNT

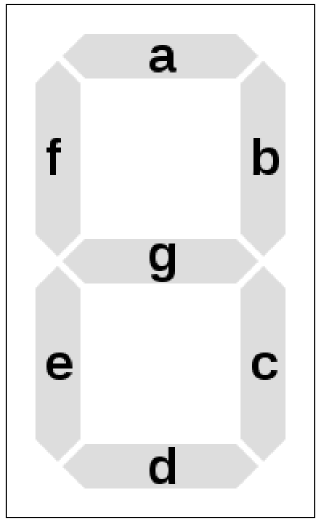
Smallest circuit to count bits in 3-bit input, producing 2-bit output:

```
INPUTS=[0b11110000, 0b11001100, 0b10101010]
OUTPUTS=[0b11101000, 0b10010110] # high, low
BITS=8
avail=[I_AND, I_OR, I_XOR, I_NOT]
...
5 instructions
r0=input                  11110000
r1=input                  11001100
r2=input                  10101010
r3=XOR r2, r1             01100110
r4=AND r0, r3             01100000
r5=AND r2, r1             10001000
r6=XOR r4, r5             11101000
r7=XOR r0, r3             10010110
```

... using only NOR3 gates:

```
avail=[I_NOR3]
...
8 instructions
r0=input                  11110000
r1=input                  11001100
r2=input                  10101010
r3=NOR3 r0, r0, r1        00000011
r4=NOR3 r2, r3, r1        00010000
r5=NOR3 r3, r2, r0        00000100
r6=NOR3 r3, r0, r5        00001000
r7=NOR3 r5, r2, r4        01000001
r8=NOR3 r3, r4, r1        00100000
r9=NOR3 r3, r5, r4        11101000
r10=NOR3 r8, r7, r6       10010110
```

### 14.1.5 Circuit for a central "g" segment of 7-segment display




(The image taken from Wikipedia.)

I couldn't find a circuit for the all 7 segments, but found for one, a central one ("g"). Yes, encoders like these are usually implemented using a ROM. But I always been wondering, how to do this using only gates.

The truth table for "g" segment I've used from this table:

| Segments (✓= ON) | | | | | | | Display | Segments (✓= ON) | | | | | | | Display |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | f | g | | a | b | c | d | e | f | g | |
| ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | 0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 8 |
| | ✓ | ✓ | | | | | 1 | ✓ | ✓ | ✓ | | | ✓ | ✓ | 9 |
| ✓ | ✓ | | ✓ | ✓ | | ✓ | 2 | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | A |
| ✓ | ✓ | ✓ | ✓ | | | ✓ | 3 | | | ✓ | ✓ | ✓ | ✓ | ✓ | b |
| | ✓ | ✓ | | | ✓ | ✓ | 4 | ✓ | | | ✓ | ✓ | ✓ | | C |
| ✓ | | ✓ | ✓ | | ✓ | ✓ | 5 | | ✓ | ✓ | ✓ | ✓ | | ✓ | d |
| ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | 6 | ✓ | | | ✓ | ✓ | ✓ | ✓ | E |
| ✓ | ✓ | ✓ | | | | | 7 | ✓ | | | | ✓ | ✓ | ✓ | F |

(The image taken from http://www.nutsvolts.com.)

```
INPUTS=[0b1111111100000000, 0b1111000011110000, 0b1100110011001100, 0b1010101010101010]
```

```
# "g" segment, like here: http://www.nutsvolts.com/uploads/wygwam/
    NV_0501_Marston_Figure02.jpg
OUTPUTS=[0b1110111101111100] # g
BITS=16
avail=[I_AND, I_OR, I_XOR, I_NOT]
...
5 instructions
r0=input                 1111111100000000
r1=input                 1111000011110000
r2=input                 1100110011001100
r3=input                 1010101010101010
r4=AND r3, r1            1010000010100000
r5=XOR r4, r0            0101111110100000
r6=XOR r4, r2            0110110001101100
r7=XOR r5, r1            1010111101010000
r8=OR r7, r6             1110111101111100
```

Using only NOR3 gates:

```
avail=[I_NOR3]
...
8 instructions
r0=input                 1111111100000000
r1=input                 1111000011110000
r2=input                 1100110011001100
r3=input                 1010101010101010
r4=NOR3 r1, r1, r1       0000111100001111
r5=NOR3 r3, r3, r4       0101000001010000
r6=NOR3 r2, r0, r0       0000000000110011
r7=NOR3 r6, r6, r5       1010111110001100
r8=NOR3 r4, r0, r7       0000000001110000
r9=NOR3 r8, r7, r2       0001000000000011
r10=NOR3 r0, r8, r4      0000000010000000
r11=NOR3 r9, r10, r10    1110111101111100
```

## 14.2 TAOCP 7.1.1 exercises 4 and 5

Found this exercise in TAOCP section 7.1.1 (Boolean basics):

**Table 1**

THE SIXTEEN LOGICAL OPERATIONS ON TWO VARIABLES

| Truth table | Notation(s) | Operator symbol $\circ$ | Name(s) |
|---|---|---|---|
| 0000 | $0$ | $\bot$ | Contradiction; falsehood; antilogy; constant 0 |
| 0001 | $xy,\ \ x \wedge y,\ \ x \,\&\, y$ | $\wedge$ | Conjunction; and |
| 0010 | $x \wedge \bar{y},\ \ x \not\supset y,\ \ [x > y],\ \ x \dot{-} y$ | $\overline{\supset}$ | Nonimplication; difference; but not |
| 0011 | $x$ | $\llcorner$ | Left projection |
| 0100 | $\bar{x} \wedge y,\ \ x \not\subset y,\ \ [x < y],\ \ y \dot{-} x$ | $\overline{\subset}$ | Converse nonimplication; not ... but |
| 0101 | $y$ | $\mathsf{R}$ | Right projection |
| 0110 | $x \oplus y,\ \ x \not\equiv y,\ \ x\hat{\ }y$ | $\oplus$ | Exclusive disjunction; nonequivalence; "xor" |
| 0111 | $x \vee y,\ \ x \mid y$ | $\vee$ | (Inclusive) disjunction; or; and/or |
| 1000 | $\bar{x} \wedge \bar{y},\ \ \overline{x \vee y},\ \ x \mathbin{\overline{\vee}} y,\ \ x \downarrow y$ | $\overline{\vee}$ | Nondisjunction; joint denial; neither ... nor |
| 1001 | $x \equiv y,\ \ x \leftrightarrow y,\ \ x \Leftrightarrow y$ | $\equiv$ | Equivalence; if and only if; "iff" |
| 1010 | $\bar{y},\ \ \neg y,\ \ !y,\ \ \sim y$ | $\bar{\mathsf{R}}$ | Right complementation |
| 1011 | $x \vee \bar{y},\ x \subset y,\ x \Leftarrow y,\ [x \geq y],\ x^y$ | $\subset$ | Converse implication; if |
| 1100 | $\bar{x},\ \ \neg x,\ \ !x,\ \ \sim x$ | $\overline{\llcorner}$ | Left complementation |
| 1101 | $\bar{x} \vee y,\ x \supset y,\ x \Rightarrow y,\ [x \leq y],\ y^x$ | $\supset$ | Implication; only if; if ... then |
| 1110 | $\bar{x} \vee \bar{y},\ \ \overline{x \wedge y},\ \ x \mathbin{\overline{\wedge}} y,\ \ x \mid y$ | $\overline{\wedge}$ | Nonconjunction; not both ... and; "nand" |
| 1111 | $1$ | $\top$ | Affirmation; validity; tautology; constant 1 |

Figure 38: Page 3

**4.** [*24*] (H. M. Sheffer.) The purpose of this exercise is to show that all of the operations in Table 1 can be expressed in terms of NAND. (a) For each of the 16 operators $\circ$ in that table, find a formula equivalent to $x \circ y$ that uses only $\overline{\wedge}$ as an operator. Your formula should be as short as possible. For example, the answer for operation $\llcorner$ is simply "$x$", but the answer for $\overline{\llcorner}$ is "$x \mathbin{\overline{\wedge}} x$". Do not use the constants 0 or 1 in your formulas. (b) Similarly, find 16 short formulas when constants *are* allowed. For example, $x \mathbin{\overline{\llcorner}} y$ can now be expressed also as "$x \mathbin{\overline{\wedge}} 1$".

**5.** [*24*] Consider exercise 4 with $\overline{\subset}$ as the basic operation instead of $\overline{\wedge}$.

Figure 39: Page 34

I'm, not clever enough to solve this manually, but I could try using logic synthesis, as I did before. As they say, "machines should work; people should think".

The modified Z3Py script:

```
#!/usr/bin/env python
from z3 import *
import sys

I_AND, I_OR, I_XOR, I_NOT, I_NOR3, I_NAND, I_ANDN = 0,1,2,3,4,5,6
```

```
# 2-input function
BITS=4
add_always_false=False
add_always_true=False
#add_always_false=True
#add_always_true=True
avail=[I_NAND]
#avail=[I_ANDN]

MAX_STEPS=10

# My representation of TT is: [MSB..LSB].
# Knuth's representation in the section 7.1.1 (Boolean basics) of TAOCP is different: [
    LSB..MSB].
# so I'll reverse bits before printing TT:
def rvr_4_bits(i):
    return ((i>>0)&1)<<3 | ((i>>1)&1)<<2 | ((i>>2)&1)<<1 | ((i>>3)&1)<<0

def find_NAND_only_for_TT (OUTPUTS):
    INPUTS=[0b1100, 0b1010]
    # if additional always-false or always-true must be present:
    if add_always_false:
        INPUTS.append(0)
    if add_always_true:
        INPUTS.append(2**BITS-1)

    # this called during self-testing:
    def eval_ins(R, s, m, STEPS, op, op1_reg, op2_reg, op3_reg):
        op_n=m[op[s]].as_long()
        op1_reg_tmp=m[op1_reg[s]].as_long()
        op1_val=R[op1_reg_tmp]
        op2_reg_tmp=m[op2_reg[s]].as_long()
        op3_reg_tmp=m[op3_reg[s]].as_long()
        if op_n in [I_AND, I_OR, I_XOR, I_NOR3, I_NAND, I_ANDN]:
            op2_val=R[op2_reg_tmp]
        if op_n==I_AND:
            return op1_val&op2_val

        elif op_n==I_OR:
            return op1_val|op2_val

        elif op_n==I_XOR:
            return op1_val^op2_val

        elif op_n==I_NOT:
            return ~op1_val

        elif op_n==I_NOR3:
            op3_val=R[op3_reg_tmp]
            return ~(op1_val|op2_val|op3_val)
        elif op_n==I_NAND:
            return ~(op1_val&op2_val)
        elif op_n==I_ANDN:
            return (~op1_val)&op2_val
```

```
    else:
        raise AssertionError

# evaluate program we've got. for self-testing.
def eval_pgm(m, STEPS, op, op1_reg, op2_reg, op3_reg):
    R=[None]*STEPS
    for i in range(len(INPUTS)):
        R[i]=INPUTS[i]

    for s in range(len(INPUTS),STEPS):
        R[s]=eval_ins(R, s, m, STEPS, op, op1_reg, op2_reg, op3_reg)

    return R

# get all states, for self-testing:
def selftest(m, STEPS, op, op1_reg, op2_reg, op3_reg):
    l=eval_pgm(m, STEPS, op, op1_reg, op2_reg, op3_reg)
    print "simulate:"
    for i in range(len(l)):
        print "r%d=" % i, format(l[i] & 2**BITS-1, '0'+str(BITS)+'b')

"""
selector() function generates expression like:

If(op1_reg_s5 == 0,
   S_s0,
   If(op1_reg_s5 == 1,
      S_s1,
      If(op1_reg_s5 == 2,
         S_s2,
         If(op1_reg_s5 == 3,
            S_s3,
            If(op1_reg_s5 == 4,
               S_s4,
               If(op1_reg_s5 == 5,
                  S_s5,
                  If(op1_reg_s5 == 6,
                     S_s6,
                     If(op1_reg_s5 == 7,
                        S_s7,
                        If(op1_reg_s5 == 8,
                           S_s8,
                           If(op1_reg_s5 == 9,
                              S_s9,
                              If(op1_reg_s5 == 10,
                                 S_s10,
                                 If(op1_reg_s5 == 11,
                                    S_s11,
                                0))))))))))))

this is like multiplexer or switch()
"""
def selector(R, s):
    t=0 # default value
    for i in range(MAX_STEPS):
        t=If(s==(MAX_STEPS-i-1), R[MAX_STEPS-i-1], t)
```

```
    return t

def simulate_op(R, op, op1_reg, op2_reg, op3_reg):
    op1_val=selector(R, op1_reg)
    return If(op==I_AND, op1_val & selector(R, op2_reg),
           If(op==I_OR, op1_val | selector(R, op2_reg),
           If(op==I_XOR, op1_val ^ selector(R, op2_reg),
           If(op==I_NOT, ~op1_val,
           If(op==I_NOR3, ~(op1_val | selector(R, op2_reg) | selector (R, op3_reg)),
           If(op==I_NAND, ~(op1_val & selector(R, op2_reg)),
           If(op==I_ANDN, (~op1_val) & selector(R, op2_reg),
               0))))))) # default

op_to_str_tbl=["AND", "OR", "XOR", "NOT", "NOR3", "NAND", "ANDN"]

def print_model(m, R, STEPS, op, op1_reg, op2_reg, op3_reg):
    print ("%d instructions for OUTPUTS TT (Knuth's representation)=" % (STEPS-len(
       INPUTS))), format(rvr_4_bits(OUTPUTS[0]) & 2**BITS-1, '0'+str(BITS)+'b')
    for s in range(STEPS):
        if s<len(INPUTS):
            t="r%d=input" % s
        else:
            op_n=m[op[s]].as_long()
            op_s=op_to_str_tbl[op_n]
            op1_reg_n=m[op1_reg[s]].as_long()
            op2_reg_n=m[op2_reg[s]].as_long()
            op3_reg_n=m[op3_reg[s]].as_long()
            if op_n in [I_AND, I_OR, I_XOR, I_NAND, I_ANDN]:
                t="r%d=%s r%d, r%d" % (s, op_s, op1_reg_n, op2_reg_n)
            elif op_n==I_NOT:
                t="r%d=%s r%d" % (s, op_s, op1_reg_n)
            else: # else NOR3
                t="r%d=%s r%d, r%d, r%d" % (s, op_s, op1_reg_n, op2_reg_n, op3_reg_n
                   )

        tt=format(m[R[s]].as_long(), '0'+str(BITS)+'b')
        print t+" "*(25-len(t))+tt

def attempt(STEPS):
    print "attempt, STEPS=", STEPS
    sl=Solver()

    # state of each register:
    R=[BitVec(('S_s%d' % s), BITS) for s in range(MAX_STEPS)]
    # operation type and operands for each register:
    op=[Int('op_s%d' % s) for s in range(MAX_STEPS)]
    op1_reg=[Int('op1_reg_s%d' % s) for s in range(MAX_STEPS)]
    op2_reg=[Int('op2_reg_s%d' % s) for s in range(MAX_STEPS)]
    op3_reg=[Int('op3_reg_s%d' % s) for s in range(MAX_STEPS)]

    for s in range(len(INPUTS), STEPS):
        # for each step.
        # expression like Or(op[s]==0, op[s]==1, ...) is formed here. values are
           taken from avail[]
        sl.add(Or(*[op[s]==j for j in avail]))
        # each operand can refer to one of registers BEFORE the current instruction:
```

```
            sl.add(And(op1_reg[s]>=0, op1_reg[s]<s))
            sl.add(And(op2_reg[s]>=0, op2_reg[s]<s))
            sl.add(And(op3_reg[s]>=0, op3_reg[s]<s))

        # fill inputs:
        for i in range(len(INPUTS)):
            sl.add(R[i]==INPUTS[i])
        # fill outputs, "must be's"
        for o in range(len(OUTPUTS)):
            sl.add(R[STEPS-(o+1)]==list(reversed(OUTPUTS))[o])

        # connect each register to "simulator":
        for s in range(len(INPUTS), STEPS):
            sl.add(R[s]==simulate_op(R, op[s], op1_reg[s], op2_reg[s], op3_reg[s]))

        tmp=sl.check()
        if tmp==sat:
            print "sat!"
            m=sl.model()
            #print m
            print_model(m, R, STEPS, op, op1_reg, op2_reg, op3_reg)
            selftest(m, STEPS, op, op1_reg, op2_reg, op3_reg)
            return True
        else:
            print tmp
        return False

    for s in range(len(INPUTS)+len(OUTPUTS), MAX_STEPS):
        if attempt(s):
            return

for i in range(16):
    print "getting circuit for TT=", format(i & 2**BITS-1, '0'+str(BITS)+'b')
    find_NAND_only_for_TT ([i])
```

My solution for NAND:

```
3 instructions for OUTPUTS TT (Knuth's representation)= 0000
r0=input                 1100
r1=input                 1010
r2=NAND r1, r1           0101
r3=NAND r1, r2           1111
r4=NAND r3, r3           0000

4 instructions for OUTPUTS TT (Knuth's representation)= 1000
r0=input                 1100
r1=input                 1010
r2=NAND r0, r0           0011
r3=NAND r1, r1           0101
r4=NAND r2, r3           1110
r5=NAND r4, r4           0001

3 instructions for OUTPUTS TT (Knuth's representation)= 0100
r0=input                 1100
r1=input                 1010
r2=NAND r1, r0           0111
r3=NAND r1, r2           1101
```

```
r4=NAND r3, r3           0010

1 instructions for OUTPUTS TT (Knuth's representation)= 1100
r0=input                 1100
r1=input                 1010
r2=NAND r0, r0           0011

3 instructions for OUTPUTS TT (Knuth's representation)= 0010
r0=input                 1100
r1=input                 1010
r2=NAND r1, r1           0101
r3=NAND r0, r2           1011
r4=NAND r3, r3           0100

1 instructions for OUTPUTS TT (Knuth's representation)= 1010
r0=input                 1100
r1=input                 1010
r2=NAND r1, r1           0101

4 instructions for OUTPUTS TT (Knuth's representation)= 0110
r0=input                 1100
r1=input                 1010
r2=NAND r0, r1           0111
r3=NAND r2, r1           1101
r4=NAND r2, r0           1011
r5=NAND r3, r4           0110

1 instructions for OUTPUTS TT (Knuth's representation)= 1110
r0=input                 1100
r1=input                 1010
r2=NAND r0, r1           0111

2 instructions for OUTPUTS TT (Knuth's representation)= 0001
r0=input                 1100
r1=input                 1010
r2=NAND r0, r1           0111
r3=NAND r2, r2           1000

5 instructions for OUTPUTS TT (Knuth's representation)= 1001
r0=input                 1100
r1=input                 1010
r2=NAND r1, r1           0101
r3=NAND r2, r0           1011
r4=NAND r3, r0           0111
r5=NAND r2, r3           1110
r6=NAND r5, r4           1001

2 instructions for OUTPUTS TT (Knuth's representation)= 0101
r0=input                 1100
r1=input                 1010
r2=NAND r1, r1           0101
r3=NAND r2, r2           1010

2 instructions for OUTPUTS TT (Knuth's representation)= 1101
r0=input                 1100
r1=input                 1010
```

```
r2=NAND r1, r1            0101
r3=NAND r2, r0            1011

2 instructions for OUTPUTS TT (Knuth's representation)= 0011
r0=input                  1100
r1=input                  1010
r2=NAND r0, r0            0011
r3=NAND r2, r2            1100

2 instructions for OUTPUTS TT (Knuth's representation)= 1011
r0=input                  1100
r1=input                  1010
r2=NAND r0, r1            0111
r3=NAND r1, r2            1101

3 instructions for OUTPUTS TT (Knuth's representation)= 0111
r0=input                  1100
r1=input                  1010
r2=NAND r0, r0            0011
r3=NAND r1, r1            0101
r4=NAND r3, r2            1110

2 instructions for OUTPUTS TT (Knuth's representation)= 1111
r0=input                  1100
r1=input                  1010
r2=NAND r1, r1            0101
r3=NAND r2, r1            1111
```

My solution for NAND with 0/1 constants:

```
1 instructions for OUTPUTS TT (Knuth's representation)= 0000
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=NAND r3, r3            0000

4 instructions for OUTPUTS TT (Knuth's representation)= 1000
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=NAND r1, r1            0101
r5=NAND r0, r0            0011
r6=NAND r4, r5            1110
r7=NAND r6, r6            0001

3 instructions for OUTPUTS TT (Knuth's representation)= 0100
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=NAND r0, r0            0011
r5=NAND r4, r1            1101
r6=NAND r5, r3            0010

1 instructions for OUTPUTS TT (Knuth's representation)= 1100
```

```
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r0, r3           0011

3 instructions for OUTPUTS TT (Knuth's representation)= 0010
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r0, r1           0111
r5=NAND r4, r0           1011
r6=NAND r5, r3           0100

1 instructions for OUTPUTS TT (Knuth's representation)= 1010
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r1, r3           0101

4 instructions for OUTPUTS TT (Knuth's representation)= 0110
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r1, r0           0111
r5=NAND r4, r0           1011
r6=NAND r1, r4           1101
r7=NAND r6, r5           0110

1 instructions for OUTPUTS TT (Knuth's representation)= 1110
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r1, r0           0111

2 instructions for OUTPUTS TT (Knuth's representation)= 0001
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r1, r0           0111
r5=NAND r4, r4           1000

5 instructions for OUTPUTS TT (Knuth's representation)= 1001
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r1, r1           0101
r5=NAND r4, r0           1011
r6=NAND r5, r4           1110
r7=NAND r0, r1           0111
```

```
r8=NAND r7, r6           1001

2 instructions for OUTPUTS TT (Knuth's representation)= 0101
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r1, r1           0101
r5=NAND r4, r4           1010

2 instructions for OUTPUTS TT (Knuth's representation)= 1101
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r0, r1           0111
r5=NAND r0, r4           1011

2 instructions for OUTPUTS TT (Knuth's representation)= 0011
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r0, r0           0011
r5=NAND r4, r4           1100

2 instructions for OUTPUTS TT (Knuth's representation)= 1011
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r1, r0           0111
r5=NAND r1, r4           1101

3 instructions for OUTPUTS TT (Knuth's representation)= 0111
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r1, r1           0101
r5=NAND r0, r0           0011
r6=NAND r5, r4           1110

1 instructions for OUTPUTS TT (Knuth's representation)= 1111
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r2, r2           1111
```

My solution for ANDN:

```
1 instructions for OUTPUTS TT (Knuth's representation)= 0000
r0=input                 1100
r1=input                 1010
r2=ANDN r1, r1           0000
```

```
1 instructions for OUTPUTS TT (Knuth's representation)= 0100
r0=input                  1100
r1=input                  1010
r2=ANDN r0, r1            0010

1 instructions for OUTPUTS TT (Knuth's representation)= 0010
r0=input                  1100
r1=input                  1010
r2=ANDN r1, r0            0100

2 instructions for OUTPUTS TT (Knuth's representation)= 0001
r0=input                  1100
r1=input                  1010
r2=ANDN r0, r1            0010
r3=ANDN r2, r1            1000

2 instructions for OUTPUTS TT (Knuth's representation)= 0101
r0=input                  1100
r1=input                  1010
r2=ANDN r1, r1            0000
r3=ANDN r2, r1            1010

2 instructions for OUTPUTS TT (Knuth's representation)= 0011
r0=input                  1100
r1=input                  1010
r2=ANDN r0, r1            0010
r3=ANDN r2, r0            1100
```

My solution for ANDN with 0/1 constants:

```
1 instructions for OUTPUTS TT (Knuth's representation)= 0000
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=ANDN r3, r2            0000

2 instructions for OUTPUTS TT (Knuth's representation)= 1000
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=ANDN r0, r3            0011
r5=ANDN r1, r4            0001

1 instructions for OUTPUTS TT (Knuth's representation)= 0100
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=ANDN r0, r1            0010

1 instructions for OUTPUTS TT (Knuth's representation)= 1100
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
```

```
r4=ANDN r0, r3           0011

1 instructions for OUTPUTS TT (Knuth's representation)= 0010
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r1, r0           0100

1 instructions for OUTPUTS TT (Knuth's representation)= 1010
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r1, r3           0101

5 instructions for OUTPUTS TT (Knuth's representation)= 0110
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r0, r3           0011
r5=ANDN r1, r4           0001
r6=ANDN r4, r1           1000
r7=ANDN r6, r3           0111
r8=ANDN r5, r7           0110

3 instructions for OUTPUTS TT (Knuth's representation)= 1110
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r1, r0           0100
r5=ANDN r4, r0           1000
r6=ANDN r5, r3           0111

2 instructions for OUTPUTS TT (Knuth's representation)= 0001
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r0, r1           0010
r5=ANDN r4, r1           1000

4 instructions for OUTPUTS TT (Knuth's representation)= 1001
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r1, r0           0100
r5=ANDN r0, r1           0010
r6=ANDN r4, r3           1011
r7=ANDN r5, r6           1001

1 instructions for OUTPUTS TT (Knuth's representation)= 0101
r0=input                 1100
```

```
r1=input                1010
r2=input                0000
r3=input                1111
r4=ANDN r2, r1          1010

2 instructions for OUTPUTS TT (Knuth's representation)= 1101
r0=input                1100
r1=input                1010
r2=input                0000
r3=input                1111
r4=ANDN r1, r0          0100
r5=ANDN r4, r3          1011

1 instructions for OUTPUTS TT (Knuth's representation)= 0011
r0=input                1100
r1=input                1010
r2=input                0000
r3=input                1111
r4=ANDN r2, r0          1100

2 instructions for OUTPUTS TT (Knuth's representation)= 1011
r0=input                1100
r1=input                1010
r2=input                0000
r3=input                1111
r4=ANDN r0, r1          0010
r5=ANDN r4, r3          1101

3 instructions for OUTPUTS TT (Knuth's representation)= 0111
r0=input                1100
r1=input                1010
r2=input                0000
r3=input                1111
r4=ANDN r0, r3          0011
r5=ANDN r1, r4          0001
r6=ANDN r5, r3          1110

1 instructions for OUTPUTS TT (Knuth's representation)= 1111
r0=input                1100
r1=input                1010
r2=input                0000
r3=input                1111
r4=ANDN r2, r3          1111
```

Correct answers from TAOCP:

**4.** [*Trans. Amer. Math. Soc.* **14** (1913), 481–488.] (a) Start with the truth tables for $\mathsf{L}$ and $\mathsf{R}$; then compute truth table $\alpha \mathbin{\overline{\wedge}} \beta$ bitwise from each known pair of truth tables $\alpha$ and $\beta$, generating the results in order of the length of each formula and writing down a shortest formula that leads to each new 4-bit table:

$\perp$: $(x \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} x)) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} x))$
$\wedge$: $(x \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y)$
$\overline{\supset}$: $(x \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y)) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y))$
$\mathsf{L}$: $x$
$\overline{\subset}$: $(y \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} x)) \mathbin{\overline{\wedge}} (y \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} x))$
$\mathsf{R}$: $y$
$\oplus$: $(y \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} x)) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y))$
$\vee$: $(y \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} x)$

$\overline{\vee}$: $(x \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} x)) \mathbin{\overline{\wedge}} ((y \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} x))$
$\equiv$: $(x \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} ((y \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} x))$
$\overline{\mathsf{R}}$: $y \mathbin{\overline{\wedge}} y$
$\subset$: $y \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} x)$
$\overline{\mathsf{L}}$: $x \mathbin{\overline{\wedge}} x$
$\supset$: $x \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y)$
$\overline{\wedge}$: $x \mathbin{\overline{\wedge}} y$
$\top$: $x \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} x)$

(b) In this case we start with four tables $\perp$, $\top$, $\mathsf{L}$, $\mathsf{R}$, and we prefer formulas with fewer occurrences of variables whenever there's a choice between formulas of a given length:

$\perp$: $0$
$\wedge$: $(x \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} 1$
$\overline{\supset}$: $((y \mathbin{\overline{\wedge}} 1) \mathbin{\overline{\wedge}} x) \mathbin{\overline{\wedge}} 1$
$\mathsf{L}$: $x$
$\overline{\subset}$: $(y \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} 1)) \mathbin{\overline{\wedge}} 1$
$\mathsf{R}$: $y$
$\oplus$: $(y \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} 1)) \mathbin{\overline{\wedge}} ((y \mathbin{\overline{\wedge}} 1) \mathbin{\overline{\wedge}} x)$
$\vee$: $(y \mathbin{\overline{\wedge}} 1) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} 1)$

$\overline{\vee}$: $1 \mathbin{\overline{\wedge}} ((y \mathbin{\overline{\wedge}} 1) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} 1))$
$\equiv$: $(x \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} ((y \mathbin{\overline{\wedge}} 1) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} 1))$
$\overline{\mathsf{R}}$: $y \mathbin{\overline{\wedge}} 1$
$\subset$: $y \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} 1)$
$\overline{\mathsf{L}}$: $x \mathbin{\overline{\wedge}} 1$
$\supset$: $(y \mathbin{\overline{\wedge}} 1) \mathbin{\overline{\wedge}} x$
$\overline{\wedge}$: $x \mathbin{\overline{\wedge}} y$
$\top$: $1$

**5.** (a) $\perp$: $x \mathbin{\overline{\subset}} x$; $\wedge$: $(x \mathbin{\overline{\subset}} y) \mathbin{\overline{\subset}} y$; $\overline{\supset}$: $y \mathbin{\overline{\subset}} x$; $\mathsf{L}$: $x$; $\overline{\subset}$: $x \mathbin{\overline{\subset}} y$; $\mathsf{R}$: $y$; the other 10 cannot be expressed. (b) With constants, however, all 16 are possible:

$\perp$: $0$
$\wedge$: $(y \mathbin{\overline{\subset}} 1) \mathbin{\overline{\subset}} x$
$\overline{\supset}$: $y \mathbin{\overline{\subset}} x$
$\mathsf{L}$: $x$
$\overline{\subset}$: $x \mathbin{\overline{\subset}} y$
$\mathsf{R}$: $y$
$\oplus$: $((y \mathbin{\overline{\subset}} x) \mathbin{\overline{\subset}} ((x \mathbin{\overline{\subset}} y) \mathbin{\overline{\subset}} 1)) \mathbin{\overline{\subset}} 1$
$\vee$: $(y \mathbin{\overline{\subset}} (x \mathbin{\overline{\subset}} 1)) \mathbin{\overline{\subset}} 1$

$\overline{\vee}$: $y \mathbin{\overline{\subset}} (x \mathbin{\overline{\subset}} 1)$
$\equiv$: $(y \mathbin{\overline{\subset}} x) \mathbin{\overline{\subset}} ((x \mathbin{\overline{\subset}} y) \mathbin{\overline{\subset}} 1)$
$\overline{\mathsf{R}}$: $y \mathbin{\overline{\subset}} 1$
$\subset$: $(x \mathbin{\overline{\subset}} y) \mathbin{\overline{\subset}} 1$
$\overline{\mathsf{L}}$: $x \mathbin{\overline{\subset}} 1$
$\supset$: $(y \mathbin{\overline{\subset}} x) \mathbin{\overline{\subset}} 1$
$\overline{\wedge}$: $((y \mathbin{\overline{\subset}} 1) \mathbin{\overline{\subset}} x) \mathbin{\overline{\subset}} 1$
$\top$: $1$

[B. A. Bernstein, *University of California Publications in Mathematics* **1** (1914), 87–96.]

Figure 40: Page 51

My solutions are slightly different: I haven't "pass through" instruction, so sometimes a value is copied from the input to the output using NAND/ANDN. Also, my versions are sometimes different, but correct and has the same length.

# 15 Toy decompiler

## 15.1 Introduction

A modern-day compiler is a product of hundreds of developer/year. At the same time, toy compiler can be an exercise for a student for a week (or even weekend).

Likewise, commercial decompiler like Hex-Rays can be extremely complex, while toy decompiler like this one, can be easy to understand and remake.

The following decompiler written in Python, supports only short basic blocks, with no jumps. Memory is also not supported.

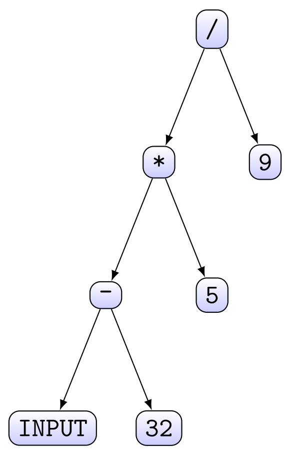
## 15.2 Data structure

Our toy decompiler will use just one single data structure, representing expression tree.

Many programming textbooks has an example of conversion from Fahrenheit temperature to Celsius, using the following formula:

$$celsius = (fahrenheit - 32) \cdot \frac{5}{9}$$

This expression can be represented as a tree:




How to store it in memory? We see here 3 types of nodes: 1) numbers (or values); 2) arithmetical operations; 3) symbols (like "INPUT").

Many developers with OOP[93] in their mind will create some kind of class. Other developer maybe will use "variant type".

I'll use simplest possible way of representing this structure: a Python tuple. First element of tuple can be a string: either "EXPR_OP" for operation, "EXPR_SYMBOL" for symbol or "EXPR_VALUE" for value. In case of symbol or value, it follows the string. In case of operation, the string followed by another tuples.

Node type and operation type are stored as plain strings—to make debugging output easier to read.

There are *constructors* in our code, in OOP sense:

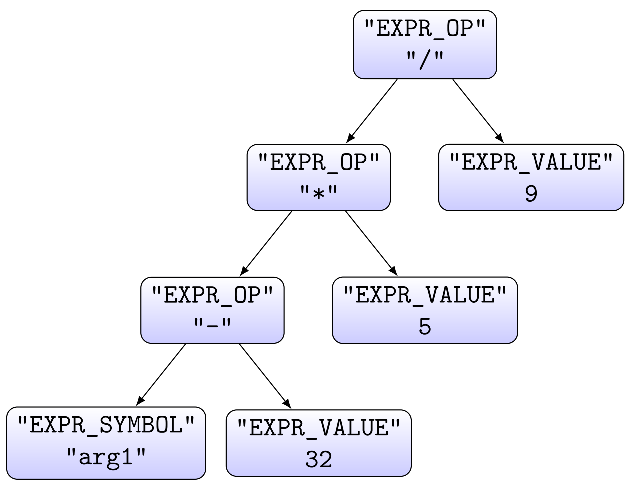
```
def create_val_expr (val):
    return ("EXPR_VALUE", val)

def create_symbol_expr (val):
    return ("EXPR_SYMBOL", val)

def create_binary_expr (op, op1, op2):
    return ("EXPR_OP", op, op1, op2)
```

There are also *accessors*:

[93] Object-oriented programming

```
def get_expr_type(e):
    return e[0]

def get_symbol (e):
    assert get_expr_type(e)=="EXPR_SYMBOL"
    return e[1]

def get_val (e):
    assert get_expr_type(e)=="EXPR_VALUE"
    return e[1]

def is_expr_op(e):
    return get_expr_type(e)=="EXPR_OP"

def get_op (e):
    assert is_expr_op(e)
    return e[1]

def get_op1 (e):
    assert is_expr_op(e)
    return e[2]

def get_op2 (e):
    assert is_expr_op(e)
    return e[3]
```

The temperature conversion expression we just saw will be represented as:




...or as Python expression:

```
('EXPR_OP', '/',
        ('EXPR_OP', '*',
        ('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 32)),
        ('EXPR_VALUE', 5)),
('EXPR_VALUE', 9))
```

In fact, this is AST[94] in its simplest form. ASTs are used heavily in compilers.

## 15.3 Simple examples

Let's start with simplest example:

[94]Abstract syntax tree

```
        mov     rax, rdi
        imul    rax, rsi
```

At start, these symbols are assigned to registers: RAX=initial_RAX, RBX=initial_RBX, RDI=arg1, RSI=arg2, RDX=arg3, RCX=arg4.

When we handle MOV instruction, we just copy expression from RDI to RAX. When we handle IMUL instruction, we create a new expression, adding together expressions from RAX and RSI and putting result into RAX again.

I can feed this to decompiler and we will see how register's state is changed through processing:

```
python td.py --show-registers --python-expr tests/mul.s

...

line=[mov       rax, rdi]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_SYMBOL', 'arg1')

line=[imul      rax, rsi]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_SYMBOL', 'arg2'))

...

result=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_SYMBOL', 'arg2'))
```

IMUL instruction is mapped to "*" string, and then new expression is constructed in `handle_binary_op()`, which puts result into RAX.

In this output, the data structures are dumped using Python `str()` function, which does mostly the same, as `print()`.

Output is bulky, and we can turn off Python expressions output, and see how this internal data structure can be rendered neatly using our internal `expr_to_string()` function:

```
python td.py --show-registers tests/mul.s

...

line=[mov       rax, rdi]
rcx=arg4
rsi=arg2
rbx=initial_RBX
rdx=arg3
rdi=arg1
rax=arg1

line=[imul      rax, rsi]
rcx=arg4
rsi=arg2
rbx=initial_RBX
rdx=arg3
rdi=arg1
```

```
rax=(arg1 * arg2)

...

result=(arg1 * arg2)
```

Slightly advanced example:

```
        imul    rdi, rsi
        lea     rax, [rdi+rdx]
```

LEA instruction is treated just as ADD.

```
python td.py --show-registers --python-expr tests/mul_add.s

...

line=[imul      rdi, rsi]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_SYMBOL', 'arg2'))
rax=('EXPR_SYMBOL', 'initial_RAX')

line=[lea       rax, [rdi+rdx]]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_SYMBOL', 'arg2'))
rax=('EXPR_OP', '+', ('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_SYMBOL', 'arg2')),
    ('EXPR_SYMBOL', 'arg3'))

...

result=('EXPR_OP', '+', ('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_SYMBOL', 'arg2
   ')), ('EXPR_SYMBOL', 'arg3'))
```

And again, let's see this expression dumped neatly:

```
python td.py --show-registers tests/mul_add.s

...

result=((arg1 * arg2) + arg3)
```

Now another example, where we use 2 input arguments:

```
        imul    rdi, rdi, 1234
        imul    rsi, rsi, 5678
        lea     rax, [rdi+rsi]
```

```
python td.py --show-registers --python-expr tests/mul_add3.s

...

line=[imul      rdi, rdi, 1234]
rcx=('EXPR_SYMBOL', 'arg4')
```

```
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 1234))
rax=('EXPR_SYMBOL', 'initial_RAX')

line=[imul      rsi, rsi, 5678]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg2'), ('EXPR_VALUE', 5678))
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 1234))
rax=('EXPR_SYMBOL', 'initial_RAX')

line=[lea       rax, [rdi+rsi]]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg2'), ('EXPR_VALUE', 5678))
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 1234))
rax=('EXPR_OP', '+', ('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 1234)), ('
   EXPR_OP', '*', ('EXPR_SYMBOL', 'arg2'), ('EXPR_VALUE', 5678)))

...

result=('EXPR_OP', '+', ('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 1234)),
     ('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg2'), ('EXPR_VALUE', 5678)))
```

...and now neat output:

```
python td.py --show-registers tests/mul_add3.s

...

result=((arg1 * 1234) + (arg2 * 5678))
```

Now conversion program:

```
        mov     rax, rdi
        sub     rax, 32
        imul    rax, 5
        mov     rbx, 9
        idiv    rbx
```

You can see, how register's state is changed over execution (or parsing).
Raw:

```
python td.py --show-registers --python-expr tests/fahr_to_celsius.s

...

line=[mov       rax, rdi]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_SYMBOL', 'arg1')
```

```
line=[sub       rax, 32]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 32))

line=[imul      rax, 5]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_OP', '*', ('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 32)), ('
   EXPR_VALUE', 5))

line=[mov       rbx, 9]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_VALUE', 9)
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_OP', '*', ('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 32)), ('
   EXPR_VALUE', 5))

line=[idiv      rbx]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_VALUE', 9)
rdx=('EXPR_OP', '%', ('EXPR_OP', '*', ('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('
   EXPR_VALUE', 32)), ('EXPR_VALUE', 5)), ('EXPR_VALUE', 9))
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_OP', '/', ('EXPR_OP', '*', ('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('
   EXPR_VALUE', 32)), ('EXPR_VALUE', 5)), ('EXPR_VALUE', 9))

...

result=('EXPR_OP', '/', ('EXPR_OP', '*', ('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('
   EXPR_VALUE', 32)), ('EXPR_VALUE', 5)), ('EXPR_VALUE', 9))
```

Neat:

```
python td.py --show-registers tests/fahr_to_celsius.s

...

line=[mov       rax, rdi]
rcx=arg4
rsi=arg2
rbx=initial_RBX
rdx=arg3
rdi=arg1
rax=arg1

line=[sub       rax, 32]
rcx=arg4
```

```
rsi=arg2
rbx=initial_RBX
rdx=arg3
rdi=arg1
rax=(arg1 - 32)

line=[imul      rax, 5]
rcx=arg4
rsi=arg2
rbx=initial_RBX
rdx=arg3
rdi=arg1
rax=((arg1 - 32) * 5)

line=[mov       rbx, 9]
rcx=arg4
rsi=arg2
rbx=9
rdx=arg3
rdi=arg1
rax=((arg1 - 32) * 5)

line=[idiv      rbx]
rcx=arg4
rsi=arg2
rbx=9
rdx=(((arg1 - 32) * 5) % 9)
rdi=arg1
rax=(((arg1 - 32) * 5) / 9)

...

result=(((arg1 - 32) * 5) / 9)
```

It is interesting to note that IDIV instruction also calculates reminder of division, and it is placed into RDX register. It's not used, but is available for use.

This is how quotient and remainder are stored in registers:

```
def handle_unary_DIV_IDIV (registers, op1):
    op1_expr=register_or_number_in_string_to_expr (registers, op1)
    current_RAX=registers["rax"]
    registers["rax"]=create_binary_expr ("/", current_RAX, op1_expr)
    registers["rdx"]=create_binary_expr ("%", current_RAX, op1_expr)
```

Now this is `align2grain()` function[95]:

```
        ; uint64_t align2grain (uint64_t i, uint64_t grain)
        ;    return ((i + grain-1) & ~(grain-1));

        ; rdi=i
        ; rsi=grain

        sub     rsi, 1
        add     rdi, rsi
        not     rsi
        and     rdi, rsi
```

[95]Taken from https://docs.oracle.com/javase/specs/jvms/se6/html/Compiling.doc.html

```
        mov     rax, rdi
```

```
...

line=[sub       rsi, 1]
rcx=arg4
rsi=(arg2 - 1)
rbx=initial_RBX
rdx=arg3
rdi=arg1
rax=initial_RAX

line=[add       rdi, rsi]
rcx=arg4
rsi=(arg2 - 1)
rbx=initial_RBX
rdx=arg3
rdi=(arg1 + (arg2 - 1))
rax=initial_RAX

line=[not       rsi]
rcx=arg4
rsi=(~(arg2 - 1))
rbx=initial_RBX
rdx=arg3
rdi=(arg1 + (arg2 - 1))
rax=initial_RAX

line=[and       rdi, rsi]
rcx=arg4
rsi=(~(arg2 - 1))
rbx=initial_RBX
rdx=arg3
rdi=((arg1 + (arg2 - 1)) & (~(arg2 - 1)))
rax=initial_RAX

line=[mov       rax, rdi]
rcx=arg4
rsi=(~(arg2 - 1))
rbx=initial_RBX
rdx=arg3
rdi=((arg1 + (arg2 - 1)) & (~(arg2 - 1)))
rax=((arg1 + (arg2 - 1)) & (~(arg2 - 1)))

...

result=((arg1 + (arg2 - 1)) & (~(arg2 - 1)))
```

## 15.4 Dealing with compiler optimizations

The following piece of code …

```
        mov     rax, rdi
        add     rax, rax
```

…will be transormed into *(arg1 + arg1)* expression. It can be reduced to *(arg1 * 2)*. Our toy decompiler can identify patterns like such and rewrite them.

```
# X+X -> X*2
def reduce_ADD1 (expr):
    if is_expr_op(expr) and get_op (expr)=="+" and get_op1 (expr)==get_op2 (expr):
        return dbg_print_reduced_expr ("reduce_ADD1", expr, create_binary_expr ("*",
            get_op1 (expr), create_val_expr (2)))

    return expr # no match
```

This function will just test, if the current node has *EXPR_OP* type, operation is "+" and both children are equal to each other. By the way, since our data structure is just tuple of tuples, Python can compare them using plain "==" operation. If the testing is finished successfully, current node is then replaced with a new expression: we take one of children, we construct a node of *EXPR_VALUE* type with "2" number in it, and then we construct a node of *EXPR_OP* type with "*".

`dbg_print_reduced_expr()` serving solely debugging purposes—it just prints the old and the new (reduced) expressions.

Decompiler is then traverse expression tree recursively in *deep-first search* fashion.

```
def reduce_step (e):
    if is_expr_op (e)==False:
        return e # expr isn't EXPR_OP, nothing to reduce (we don't reduce EXPR_SYMBOL
            and EXPR_VAL)

    if is_unary_op(get_op(e)):
        # recreate expr with reduced operand:
        return reducers(create_unary_expr (get_op(e), reduce_step (get_op1 (e))))
    else:
        # recreate expr with both reduced operands:
        return reducers(create_binary_expr (get_op(e), reduce_step (get_op1 (e)),
            reduce_step (get_op2 (e))))

...


# same as "return ...(reduce_MUL1 (reduce_ADD1 (reduce_ADD2 (... expr))))"
reducers=compose([
        ...
    reduce_ADD1, ...
    ...])

def reduce (e):
    print "going to reduce " + expr_to_string (e)
    new_expr=reduce_step(e)
    if new_expr==e:
        return new_expr # we are done here, expression can't be reduced further
    else:
        return reduce(new_expr) # reduced expr has been changed, so try to reduce it
            again
```

Reduction functions called again and again, as long, as expression changes.

Now we run it:

```
python td.py tests/add1.s

...

going to reduce (arg1 + arg1)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
```

```
going to reduce (arg1 * 2)
result=(arg1 * 2)
```

So far so good, now what if we would try this piece of code?

```
        mov     rax, rdi
        add     rax, rax
        add     rax, rax
        add     rax, rax
```

```
python td.py tests/add2.s

...

working out tests/add2.s
going to reduce (((arg1 + arg1) + (arg1 + arg1)) + ((arg1 + arg1) + (arg1 + arg1)))
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() ((arg1 * 2) + (arg1 * 2)) -> ((arg1 * 2) * 2)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() ((arg1 * 2) + (arg1 * 2)) -> ((arg1 * 2) * 2)
reduction in reduce_ADD1() (((arg1 * 2) * 2) + ((arg1 * 2) * 2)) -> (((arg1 * 2) * 2) *
   2)
going to reduce (((arg1 * 2) * 2) * 2)
result=(((arg1 * 2) * 2) * 2)
```

This is correct, but too verbose.

We would like to rewrite *(X*n)*m* expression to *X*(n*m)*, where $n$ and $m$ are numbers. We can do this by adding another function like `reduce_ADD1()`, but there is much better option: we can make matcher for tree. You can think about it as regular expression matcher, but over trees.

```
def bind_expr (key):
    return ("EXPR_WILDCARD", key)

def bind_value (key):
    return ("EXPR_WILDCARD_VALUE", key)

def match_EXPR_WILDCARD (expr, pattern):
    return {pattern[1] : expr} # return {key : expr}

def match_EXPR_WILDCARD_VALUE (expr, pattern):
    if get_expr_type (expr)!="EXPR_VALUE":
        return None
    return {pattern[1] : get_val(expr)} # return {key : expr}

def is_commutative (op):
    return op in ["+", "*", "&", "|", "^"]

def match_two_ops (op1_expr, op1_pattern, op2_expr, op2_pattern):
    m1=match (op1_expr, op1_pattern)
    m2=match (op2_expr, op2_pattern)
    if m1==None or m2==None:
        return None # one of match for operands returned False, so we do the same
    # join two dicts from both operands:
    rt={}
    rt.update(m1)
    rt.update(m2)
```

```
    return rt

def match_EXPR_OP (expr, pattern):
    if get_expr_type(expr)!=get_expr_type(pattern): # be sure, both EXPR_OP.
        return None
    if get_op (expr)!=get_op (pattern): # be sure, ops type are the same.
        return None

    if (is_unary_op(get_op(expr))):
        # match unary expression.
        return match (get_op1 (expr), get_op1 (pattern))
    else:
        # match binary expression.

        # first try match operands as is.
        m=match_two_ops (get_op1 (expr), get_op1 (pattern), get_op2 (expr), get_op2 (
           pattern))
        if m!=None:
            return m
        # if matching unsuccessful, AND operation is commutative, try also swapped
           operands.
        if is_commutative (get_op (expr))==False:
            return None
        return match_two_ops (get_op1 (expr), get_op2 (pattern), get_op2 (expr), get_op1
            (pattern))

# returns dict in case of success, or None
def match (expr, pattern):
    t=get_expr_type(pattern)
    if t=="EXPR_WILDCARD":
        return match_EXPR_WILDCARD (expr, pattern)
    elif t=="EXPR_WILDCARD_VALUE":
        return match_EXPR_WILDCARD_VALUE (expr, pattern)
    elif t=="EXPR_SYMBOL":
        if expr==pattern:
            return {}
        else:
            return None
    elif t=="EXPR_VALUE":
        if expr==pattern:
            return {}
        else:
            return None
    elif t=="EXPR_OP":
        return match_EXPR_OP (expr, pattern)
    else:
        raise AssertionError
```

Now how we will use it:

```
# (X*A)*B -> X*(A*B)
def reduce_MUL1 (expr):
    m=match (expr, create_binary_expr ("*", (create_binary_expr ("*", bind_expr ("X"),
       bind_value ("A"))), bind_value ("B")))
    if m==None:
        return expr # no match
```

```
    return dbg_print_reduced_expr ("reduce_MUL1", expr, create_binary_expr ("*",
        m["X"], # new op1
        create_val_expr (m["A"] * m["B"]))) # new op2
```

We take input expression, and we also construct pattern to be matched. Matcher works recursively over both expressions synchronously. Pattern is also expression, but can use two additional node types: *EXPR_WILDCARD* and *EXPR_WILDCARD_VALUE*. These nodes are supplied with keys (stored as strings). When matcher encounters *EXPR_WILDCARD* in pattern, it just stashes current expression and will return it. If matcher encounters *EXPR_WILDCARD_VA* it does the same, but only in case the current node has *EXPR_VALUE* type.

`bind_expr()` and `bind_value()` are functions which create nodes with the types we have seen.

All this means, `reduce_MUL1()` function will search for the expression in form *(X*A)*B*, where $A$ and $B$ are numbers. In other cases, matcher will return input expression untouched, so these reducing function can be chained.

Now when `reduce_MUL1()` encounters (sub)expression we are interesting in, it will return dictionary with keys and expressions. Let's add `print m` call somewhere before return and rerun:

```
python td.py tests/add2.s

...

going to reduce (((arg1 + arg1) + (arg1 + arg1)) + ((arg1 + arg1) + (arg1 + arg1)))
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() ((arg1 * 2) + (arg1 * 2)) -> ((arg1 * 2) * 2)
{'A': 2, 'X': ('EXPR_SYMBOL', 'arg1'), 'B': 2}
reduction in reduce_MUL1() ((arg1 * 2) * 2) -> (arg1 * 4)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() ((arg1 * 2) + (arg1 * 2)) -> ((arg1 * 2) * 2)
{'A': 2, 'X': ('EXPR_SYMBOL', 'arg1'), 'B': 2}
reduction in reduce_MUL1() ((arg1 * 2) * 2) -> (arg1 * 4)
reduction in reduce_ADD1() ((arg1 * 4) + (arg1 * 4)) -> ((arg1 * 4) * 2)
{'A': 4, 'X': ('EXPR_SYMBOL', 'arg1'), 'B': 2}
reduction in reduce_MUL1() ((arg1 * 4) * 2) -> (arg1 * 8)
going to reduce (arg1 * 8)
...
result=(arg1 * 8)
```

The dictionary has keys we supplied plus expressions matcher found. We then can use them to create new expression and return it. Numbers are just summed while forming second operand to "*" opeartion.

Now a real-world optimization technique—optimizing GCC replaced multiplication by 31 by shifting and subtraction operations:

```
        mov     rax, rdi
        sal     rax, 5
        sub     rax, rdi
```

Without reduction functions, our decompiler will translate this into *((arg1 « 5) - arg1)*. We can replace shifting left by multiplication:

```
# X<<n -> X*(2^n)
def reduce_SHL1 (expr):
    m=match (expr, create_binary_expr ("<<", bind_expr ("X"), bind_value ("Y")))
    if m==None:
        return expr # no match

    return dbg_print_reduced_expr ("reduce_SHL1", expr, create_binary_expr ("*", m["X"],
         create_val_expr (1<<m["Y"])))
```

Now we getting *((arg1 * 32) - arg1)*. We can add another reduction function:

```
# (X*n)-X -> X*(n-1)
def reduce_SUB3 (expr):
    m=match (expr, create_binary_expr ("-",
        create_binary_expr ("*", bind_expr("X1"), bind_value ("N")),
        bind_expr("X2")))

    if m!=None and match (m["X1"], m["X2"])!=None:
        return dbg_print_reduced_expr ("reduce_SUB3", expr, create_binary_expr ("*", m["
           X1"], create_val_expr (m["N"]-1)))
    else:
        return expr # no match
```

Matcher will return two X's, and we must be assured that they are equal. In fact, in previous versions of this toy decompiler, I did comparison with plain "==", and it worked. But we can reuse `match()` function for the same purpose, because it will process commutative operations better. For example, if X1 is "Q+1" and X2 is "1+Q", expressions are equal, but plain "==" will not work. On the other side, `match()` function, when encounter "+" operation (or another commutative operation), and it fails with comparison, it will also try swapped operand and will try to compare again.

However, to understand it easier, for a moment, you can imagine there is "==" instead of the second `match()`.

Anyway, here is what we've got:

```
working out tests/mul31_GCC.s
going to reduce ((arg1 << 5) - arg1)
reduction in reduce_SHL1() (arg1 << 5) -> (arg1 * 32)
reduction in reduce_SUB3() ((arg1 * 32) - arg1) -> (arg1 * 31)
going to reduce (arg1 * 31)
...
result=(arg1 * 31)
```

Another optimization technique is often seen in ARM thumb code: AND-ing a value with a value like 0xFFFFFFF0, is implemented using shifts:

```
        mov rax, rdi
        shr rax, 4
        shl rax, 4
```

This code is quite common in ARM thumb code, because it's a headache to encode 32-bit constants using couple of 16-bit thumb instructions, while single 16-bit instruction can shift by 4 bits left or right.

Also, the expression *(x»4)«4* can be jokingly called as "twitching operator": I've heard the "--i++" expression was called like this in Russian-speaking social networks, it was some kind of meme ("operator podergivaniya").

Anyway, these reduction functions will be used:

```
# X>>n -> X / (2^n)
...
def reduce_SHR2 (expr):
    m=match(expr, create_binary_expr(">>", bind_expr("X"), bind_value("Y")))
    if m==None or m["Y"]>=64:
        return expr # no match

    return dbg_print_reduced_expr ("reduce_SHR2", expr, create_binary_expr ("/", m["X"],
        create_val_expr (1<<m["Y"])))

...

# X<<n -> X*(2^n)
def reduce_SHL1 (expr):
    m=match (expr, create_binary_expr ("<<", bind_expr ("X"), bind_value ("Y")))
    if m==None:
        return expr # no match
```

```
    return dbg_print_reduced_expr ("reduce_SHL1", expr, create_binary_expr ("*", m["X"],
        create_val_expr (1<<m["Y"])))

...

# FIXME: slow
# returns True if n=2^x or popcnt(n)=1
def is_2n(n):
    return bin(n).count("1")==1


# AND operation using DIV/MUL or SHL/SHR
# (X / (2^n)) * (2^n) -> X&(~((2^n)-1))
def reduce_MUL2 (expr):
    m=match(expr, create_binary_expr ("*", create_binary_expr ("/", bind_expr("X"),
       bind_value("N1")), bind_value("N2")))
    if m==None or m["N1"]!=m["N2"] or is_2n(m["N1"])==False: # short-circuit expression
        return expr # no match

    return dbg_print_reduced_expr("reduce_MUL2", expr, create_binary_expr ("&", m["X"],
        create_val_expr(~(m["N1"]-1)&0xffffffffffffffff)))
```

Now the result:

```
working out tests/AND_by_shifts2.s
going to reduce ((arg1 >> 4) << 4)
reduction in reduce_SHR2() (arg1 >> 4) -> (arg1 / 16)
reduction in reduce_SHL1() ((arg1 / 16) << 4) -> ((arg1 / 16) * 16)
reduction in reduce_MUL2() ((arg1 / 16) * 16) -> (arg1 & 0xfffffffffffffff0)
going to reduce (arg1 & 0xfffffffffffffff0)
...
result=(arg1 & 0xfffffffffffffff0)
```

### 15.4.1 Division using multiplication

Division is often replaced by multiplication for performance reasons.

From school-level arithmetics, we can remember that division by 3 can be replaced by multiplication by $\frac{1}{3}$. In fact, sometimes compilers do so for floating-point arithmetics, for example, FDIV instruction in x86 code can be replaced by FMUL. At least MSVC 6.0 will replace division by 3 by multiplication by $\frac{1}{3}$ and sometimes it's hard to be sure, what operation was in original source code.

But when we operate over integer values and CPU registers, we can't use fractions. However, we can rework fraction:

$$result = \frac{x}{3} = x \cdot \frac{1}{3} = x \cdot \frac{1 \cdot MagicNumber}{3 \cdot MagicNumber}$$

Given the fact that division by $2^n$ is very fast, we now should find that $MagicNumber$, for which the following equation will be true: $2^n = 3 \cdot MagicNumber$.

This code performing division by 10:

```
        mov     rax, rdi
        movabs  rdx, 0cccccccccccccccdh
        mul     rdx
        shr     rdx, 3
        mov     rax, rdx
```

Division by $2^{64}$ is somewhat hidden: lower 64-bit of product in RAX is not used (dropped), only higher 64-bit of product (in RDX) is used and then shifted by additional 3 bits.

RDX register is set during processing of MUL/IMUL like this:

```
def handle_unary_MUL_IMUL (registers, op1):
    op1_expr=register_or_number_in_string_to_expr (registers, op1)
    result=create_binary_expr ("*", registers["rax"], op1_expr)
    registers["rax"]=result
    registers["rdx"]=create_binary_expr (">>", result, create_val_expr(64))
```

In other words, the assembly code we have just seen multiplicates by $\frac{0cccccccccccccccdh}{2^{64+3}}$, or divides by $\frac{2^{64+3}}{0cccccccccccccccdh}$. To find divisor we just have to divide numerator by denominator.

```
# n = magic number
# m = shifting coefficient
# return = 1 / (n / 2^m) = 2^m / n
def get_divisor (n, m):
    return (2**float(m))/float(n)

# (X*n)>>m, where m>=64 -> X/...
def reduce_div_by_MUL (expr):
    m=match (expr, create_binary_expr(">>", create_binary_expr ("*", bind_expr("X"),
        bind_value("N")), bind_value("M")))
    if m==None:
        return expr # no match

    divisor=get_divisor(m["N"], m["M"])
    return dbg_print_reduced_expr ("reduce_div_by_MUL", expr, create_binary_expr ("/", m
        ["X"], create_val_expr (int(divisor))))
```

This works, but we have a problem: this rule takes *(arg1 * 0xcccccccccccccccd) » 64* expression first and finds divisor to be equal to 1.25. This is correct: result is shifted by 3 bits after (or divided by 8), and $1.25 \cdot 8 = 10$. But our toy decompiler doesn't support real numbers.

We can solve this problem in the following way: if divisor has fractional part, we postpone reducing, with a hope, that two subsequent right shift operations will be reduced into single one:

```
# (X*n)>>m, where m>=64 -> X/...
def reduce_div_by_MUL (expr):
    m=match (expr, create_binary_expr(">>", create_binary_expr ("*", bind_expr("X"),
        bind_value("N")), bind_value("M")))
    if m==None:
        return expr # no match

    divisor=get_divisor(m["N"], m["M"])
    if math.floor(divisor)==divisor:
        return dbg_print_reduced_expr ("reduce_div_by_MUL", expr, create_binary_expr
            ("/", m["X"], create_val_expr (int(divisor))))
    else:
        print "reduce_div_by_MUL(): postponing reduction, because divisor=", divisor
        return expr
```

That works:

```
working out tests/div_by_mult10_unsigned.s
going to reduce (((arg1 * 0xcccccccccccccccd) >> 64) >> 3)
reduce_div_by_MUL(): postponing reduction, because divisor= 1.25
reduction in reduce_SHR1() (((arg1 * 0xcccccccccccccccd) >> 64) >> 3) -> ((arg1 * 0
   xcccccccccccccccd) >> 67)
going to reduce ((arg1 * 0xcccccccccccccccd) >> 67)
reduction in reduce_div_by_MUL() ((arg1 * 0xcccccccccccccccd) >> 67) -> (arg1 / 10)
going to reduce (arg1 / 10)
```

```
result=(arg1 / 10)
```

I don't know if this is best solution. In early version of this decompiler, it processed input expression in two passes: first pass for everything except division by multiplication, and the second pass for the latter. I don't know which way is better. Or maybe we could support real numbers in expressions?

Couple of words about better understanding division by multiplication. Many people miss "hidden" division by $2^{32}$ or $2^{64}$, when lower 32-bit part (or 64-bit part) of product is not used (or just dropped). Also, there is misconception that modulo inverse is used here. This is close, but not the same thing. Extended Euclidean algorithm is usually used to find *magic coefficient*, but in fact, this algorithm is rather used to solve the equation. You can solve it using any other method. Also, needless to mention, the equation is unsolvable for some divisors, because this is diophantine equation (i.e., equation allowing result to be only integer), since we work on integer CPU registers, after all.

## 15.5 Obfuscation/deobfuscation

Despite simplicity of our decompiler, we can see how to deobfuscate (or optimize) using several simple tricks.

For example, this piece of code does nothing:

```
        mov rax, rdi
        xor rax, 12345678h
        xor rax, 0deadbeefh
        xor rax, 12345678h
        xor rax, 0deadbeefh
```

We would need these rules to tame it:

```
# (X^n)^m -> X^(n^m)
def reduce_XOR4 (expr):
    m=match(expr,
        create_binary_expr("^",
            create_binary_expr ("^", bind_expr("X"), bind_value("N")),
                bind_value("M")))

    if m!=None:
        return dbg_print_reduced_expr ("reduce_XOR4", expr, create_binary_expr ("^", m["
           X"],
            create_val_expr (m["N"]^m["M"])))
    else:
        return expr # no match

...

# X op 0 -> X, where op is ADD, OR, XOR, SUB
def reduce_op_0 (expr):
    # try each:
    for op in ["+", "|", "^", "-"]:
        m=match(expr, create_binary_expr(op, bind_expr("X"), create_val_expr (0)))
        if m!=None:
            return dbg_print_reduced_expr ("reduce_op_0", expr, m["X"])

    # default:
    return expr # no match
```

```
working out tests/t9_obf.s
going to reduce ((((arg1 ^ 0x12345678) ^ 0xdeadbeef) ^ 0x12345678) ^ 0xdeadbeef)
reduction in reduce_XOR4() ((arg1 ^ 0x12345678) ^ 0xdeadbeef) -> (arg1 ^ 0xcc99e897)
reduction in reduce_XOR4() ((arg1 ^ 0xcc99e897) ^ 0x12345678) -> (arg1 ^ 0xdeadbeef)
reduction in reduce_XOR4() ((arg1 ^ 0xdeadbeef) ^ 0xdeadbeef) -> (arg1 ^ 0x0)
going to reduce (arg1 ^ 0x0)
```

```
reduction in reduce_op_0() (arg1 ^ 0x0) -> arg1
going to reduce arg1
result=arg1
```

This piece of code can be deobfuscated (or optimized) as well:

```
; toggle last bit

        mov rax, rdi
        mov rbx, rax
        mov rcx, rbx
        mov rsi, rcx
        xor rsi, 12345678h
        xor rsi, 12345679h
        mov rax, rsi
```

```
working out tests/t7_obf.s
going to reduce ((arg1 ^ 0x12345678) ^ 0x12345679)
reduction in reduce_XOR4() ((arg1 ^ 0x12345678) ^ 0x12345679) -> (arg1 ^ 1)
going to reduce (arg1 ^ 1)
result=(arg1 ^ 1)
```

I also used *aha!*[96] superoptimizer to find weird piece of code which does nothing.

*Aha!* is so called superoptimizer, it tries various piece of codes in brute-force manner, in attempt to find shortest possible alternative for some mathematical operation. While sane compiler developers use superoptimizers for this task, I tried it in opposite way, to find oddest pieces of code for some simple operations, including NOP operation. In past, I've used it to find weird alternative to XOR operation (4.1).

So here is what *aha!* can find for NOP:

```
; do nothing (as found by aha)

        mov rax, rdi
        and rax, rax
        or rax, rax
```

```
# X & X -> X
def reduce_AND3 (expr):
    m=match (expr, create_binary_expr ("&", bind_expr ("X1"), bind_expr ("X2")))
    if m!=None and match (m["X1"], m["X2"])!=None:
        return dbg_print_reduced_expr("reduce_AND3", expr, m["X1"])
    else:
        return expr # no match

...

# X | X -> X
def reduce_OR1 (expr):
    m=match (expr, create_binary_expr ("|", bind_expr ("X1"), bind_expr ("X2")))
    if m!=None and match (m["X1"], m["X2"])!=None:
        return dbg_print_reduced_expr("reduce_OR1", expr, m["X1"])
    else:
        return expr # no match
```

```
working out tests/t11_obf.s
going to reduce ((arg1 & arg1) | (arg1 & arg1))
```

[96] http://www.hackersdelight.org/aha/aha.pdf

```
reduction in reduce_AND3() (arg1 & arg1) -> arg1
reduction in reduce_AND3() (arg1 & arg1) -> arg1
reduction in reduce_OR1() (arg1 | arg1) -> arg1
going to reduce arg1
result=arg1
```

This is weirder:

```
; do nothing (as found by aha)

;Found a 5-operation program:
;   neg   r1,rx
;   neg   r2,rx
;   neg   r3,r1
;   or    r4,rx,2
;   and   r5,r4,r3
;   Expr: ((x | 2) & -(-(x)))

        mov rax, rdi
        neg rax
        neg rax
        or rdi, 2
        and rax, rdi
```

Rules added (I used "NEG" string to represent sign change and to be different from subtraction operation, which is just minus ("-")):

```
# (op(op X)) -> X, where both ops are NEG or NOT
def reduce_double_NEG_or_NOT (expr):
    # try each:
    for op in ["NEG", "~"]:
        m=match (expr, create_unary_expr (op, create_unary_expr (op, bind_expr("X"))))
        if m!=None:
            return dbg_print_reduced_expr ("reduce_double_NEG_or_NOT", expr, m["X"])

    # default:
    return expr # no match

...

# X & (X | ...) -> X
def reduce_AND2 (expr):
    m=match (expr, create_binary_expr ("&", create_binary_expr ("|", bind_expr ("X1"),
       bind_expr ("REST")), bind_expr ("X2")))
    if m!=None and match (m["X1"], m["X2"])!=None:
        return dbg_print_reduced_expr("reduce_AND2", expr, m["X1"])
    else:
        return expr # no match
```

```
going to reduce ((-(-arg1)) & (arg1 | 2))
reduction in reduce_double_NEG_or_NOT() (-(-arg1)) -> arg1
reduction in reduce_AND2() (arg1 & (arg1 | 2)) -> arg1
going to reduce arg1
result=arg1
```

I also forced *aha!* to find piece of code which adds 2 with no addition/subtraction operations allowed:

```
; arg1+2, without add/sub allowed, as found by aha:
```

```
;Found a 4-operation program:
;   not   r1,rx
;   neg   r2,r1
;   not   r3,r2
;   neg   r4,r3
;   Expr: -(~(-(~(x))))

        mov     rax, rdi
        not     rax
        neg     rax
        not     rax
        neg     rax
```

Rule:

```
# (- (~X)) -> X+1
def reduce_NEG_NOT (expr):
    m=match (expr, create_unary_expr ("NEG", create_unary_expr ("~", bind_expr("X"))))
    if m==None:
        return expr # no match

    return dbg_print_reduced_expr ("reduce_NEG_NOT", expr, create_binary_expr ("+", m["X
       "],create_val_expr(1)))
```

```
working out tests/add_by_not_neg.s
going to reduce (-(~(-(~arg1))))
reduction in reduce_NEG_NOT() (-(~arg1)) -> (arg1 + 1)
reduction in reduce_NEG_NOT() (-(~(arg1 + 1))) -> ((arg1 + 1) + 1)
reduction in reduce_ADD3() ((arg1 + 1) + 1) -> (arg1 + 2)
going to reduce (arg1 + 2)
result=(arg1 + 2)
```

This is artifact of two's complement system of signed numbers representation. Same can be done for subtraction (just swap NEG and NOT operations).

Now let's add some fake luggage to Fahrenheit-to-Celsius example:

```
        ; celsius = 5 * (fahr-32) / 9
        ; fake luggage:
        mov     rbx, 12345h
        mov     rax, rdi
        sub     rax, 32
        ; fake luggage:
        add     rbx, rax
        imul    rax, 5
        mov     rbx, 9
        idiv    rbx
        ; fake luggage:
        sub     rdx, rax
```

It's not a problem for our decompiler, because the noise is left in RDX register, and not used at all:

```
working out tests/fahr_to_celsius_obf1.s
line=[mov       rbx, 12345h]
rcx=arg4
rsi=arg2
rbx=0x12345
rdx=arg3
```

```
rdi=arg1
rax=initial_RAX

line=[mov       rax, rdi]
rcx=arg4
rsi=arg2
rbx=0x12345
rdx=arg3
rdi=arg1
rax=arg1

line=[sub       rax, 32]
rcx=arg4
rsi=arg2
rbx=0x12345
rdx=arg3
rdi=arg1
rax=(arg1 - 32)

line=[add       rbx, rax]
rcx=arg4
rsi=arg2
rbx=(0x12345 + (arg1 - 32))
rdx=arg3
rdi=arg1
rax=(arg1 - 32)

line=[imul      rax, 5]
rcx=arg4
rsi=arg2
rbx=(0x12345 + (arg1 - 32))
rdx=arg3
rdi=arg1
rax=((arg1 - 32) * 5)

line=[mov       rbx, 9]
rcx=arg4
rsi=arg2
rbx=9
rdx=arg3
rdi=arg1
rax=((arg1 - 32) * 5)

line=[idiv      rbx]
rcx=arg4
rsi=arg2
rbx=9
rdx=(((arg1 - 32) * 5) % 9)
rdi=arg1
rax=(((arg1 - 32) * 5) / 9)

line=[sub       rdx, rax]
rcx=arg4
rsi=arg2
rbx=9
rdx=((((arg1 - 32) * 5) % 9) - (((arg1 - 32) * 5) / 9))
```

```
rdi=arg1
rax=(((arg1 - 32) * 5) / 9)

going to reduce (((arg1 - 32) * 5) / 9)
result=(((arg1 - 32) * 5) / 9)
```

We can try to pretend we affect the result with the noise:

```
        ; celsius = 5 * (fahr-32) / 9
        ; fake luggage:
        mov     rbx, 12345h
        mov     rax, rdi
        sub     rax, 32
        ; fake luggage:
        add     rbx, rax
        imul    rax, 5
        mov     rbx, 9
        idiv    rbx
        ; fake luggage:
        sub     rdx, rax
        mov     rcx, rax
        ; OR result with garbage (result of fake luggage):
        or      rcx, rdx
        ; the following instruction shouldn't affect result:
        and     rax, rcx
```

...but in fact, it's all reduced by `reduce_AND2()` function we already saw (15.5):

```
working out tests/fahr_to_celsius_obf2.s
going to reduce ((((arg1 - 32) * 5) / 9) & ((((arg1 - 32) * 5) / 9) | ((((arg1 - 32) *
    5) % 9) - (((arg1 - 32) * 5) / 9))))
reduction in reduce_AND2() ((((arg1 - 32) * 5) / 9) & ((((arg1 - 32) * 5) / 9) | ((((
    arg1 - 32) * 5) % 9) - (((arg1 - 32) * 5)
/ 9)))) -> (((arg1 - 32) * 5) / 9)
going to reduce (((arg1 - 32) * 5) / 9)
result=(((arg1 - 32) * 5) / 9)
```

We can see that deobfuscation is in fact the same thing as optimization used in compilers. We can try this function in GCC:

```
int f(int a)
{
        return -(~a);
};
```

Optimizing GCC 5.4 (x86) generates this:

```
f:
        mov     eax, DWORD PTR [esp+4]
        add     eax, 1
        ret
```

GCC has its own rewriting rules, some of which are, probably, close to what we use here.

## 15.6 Tests

Despite simplicity of the decompiler, it's still error-prone. We need to be sure that original expression and reduced one are equivalent to each other.

### 15.6.1 Evaluating expressions

First of all, we would just evaluate (or *run*, or *execute*) expression with random values as arguments, and then compare results.

Evaluator do arithmetical operations when possible, recursively. When any symbol is encountered, its value (randomly generated before) is taken from a table.

```
un_ops={"NEG":operator.neg,
        "~":operator.invert}

bin_ops={">>":operator.rshift,
        "<<":(lambda x, c: x<<(c&0x3f)), # operator.lshift should be here, but it doesn'
           t handle too big counts
        "&":operator.and_,
        "|":operator.or_,
        "^":operator.xor,
        "+":operator.add,
        "-":operator.sub,
        "*":operator.mul,
        "/":operator.div,
        "%":operator.mod}

def eval_expr(e, symbols):
    t=get_expr_type (e)
    if t=="EXPR_SYMBOL":
        return symbols[get_symbol(e)]
    elif t=="EXPR_VALUE":
        return get_val (e)
    elif t=="EXPR_OP":
        if is_unary_op (get_op (e)):
            return un_ops[get_op(e)](eval_expr(get_op1(e), symbols))
        else:
            return bin_ops[get_op(e)](eval_expr(get_op1(e), symbols), eval_expr(get_op2(
               e), symbols))
    else:
        raise AssertionError

def do_selftest(old, new):
    for n in range(100):
        symbols={"arg1":random.getrandbits(64),
                "arg2":random.getrandbits(64),
                "arg3":random.getrandbits(64),
                "arg4":random.getrandbits(64)}
        old_result=eval_expr (old, symbols)&0xffffffffffffffff # signed->unsigned
        new_result=eval_expr (new, symbols)&0xffffffffffffffff # signed->unsigned
        if old_result!=new_result:
            print "self-test failed"
            print "initial expression: "+expr_to_string(old)
            print "reduced expression: "+expr_to_string(new)
            print "initial expression result: ", old_result
            print "reduced expression result: ", new_result
            exit(0)
```

In fact, this is very close to what LISP *EVAL* function does, or even LISP interpreter. However, not all symbols are set. If the expression is using initial values from RAX or RBX (to which symbols "initial_RAX" and "initial_RBX" are assigned, decompiler will stop with exception, because no random values assigned to these registers, and these symbols are absent in *symbols* dictionary.

Using this test, I've suddenly found a bug here (despite simplicity of all these reduction rules). Well, no-one protected

from eye strain. Nevertheless, the test has a serious problem: some bugs can be revealed only if one of arguments is 0, or 1, or $-1$. Maybe there are even more special cases exists.

Mentioned above *aha!* superoptimizer tries at least these values as arguments while testing: 1, 0, -1, 0x80000000, 0x7FFFFFFF, 0x80000001, 0x7FFFFFFE, 0x01234567, 0x89ABCDEF, -2, 2, -3, 3, -64, 64, -5, -31415.

Still, you cannot be sure.

### 15.6.2 Using Z3 SMT-solver for testing

So here we will try Z3 SMT-solver. SMT-solver can *prove* that two expressions are equivalent to each other.

For example, with the help of *aha!*, I've found another weird piece of code, which does nothing:

```
; do nothing (obfuscation)

;Found a 5-operation program:
;   neg   r1,rx
;   neg   r2,r1
;   sub   r3,r1,3
;   sub   r4,r3,r1
;   sub   r5,r4,r3
;   Expr: (((-(x) - 3) - -(x)) - (-(x) - 3))

        mov rax, rdi
        neg rax
        mov rbx, rax
        ; rbx=-x
        mov rcx, rbx
        sub rcx, 3
        ; rcx=-x-3
        mov rax, rcx
        sub rax, rbx
        ; rax=(-(x) - 3) - -(x)
        sub rax, rcx
```

Using toy decompiler, I've found that this piece is reduced to *arg1* expression:

```
working out tests/t5_obf.s
going to reduce ((((-arg1) - 3) - (-arg1)) - ((-arg1) - 3))
reduction in reduce_SUB2() ((-arg1) - 3) -> (-(arg1 + 3))
reduction in reduce_SUB5() ((-(arg1 + 3)) - (-arg1)) -> ((-(arg1 + 3)) + arg1)
reduction in reduce_SUB2() ((-arg1) - 3) -> (-(arg1 + 3))
reduction in reduce_ADD_SUB() (((-(arg1 + 3)) + arg1) - (-(arg1 + 3))) -> arg1
going to reduce arg1
result=arg1
```

But is it correct? I've added a function which can output expression(s) to SMT-LIB-format, it's as simple as a function which converts expression to string.

And this is SMT-LIB-file for Z3:

```
(assert
    (forall ((arg1 (_ BitVec 64)) (arg2 (_ BitVec 64)) (arg3 (_ BitVec 64)) (arg4 (_
       BitVec 64)))
        (=
            (bvsub (bvsub (bvsub (bvneg arg1) #x0000000000000003) (bvneg arg1)) (bvsub (
               bvneg arg1) #x0000000000000003))
            arg1
        )
    )
)
(check-sat)
```

In plain English terms, what we asking it to be sure, that *forall* four 64-bit arguments, two expressions are equivalent (second is just *arg1*).

The syntax maybe hard to understand, but in fact, this is very close to LISP, and arithmetical operations are named "bvsub", "bvadd", etc, because "bv" stands for *bit vector*.

While running, Z3 shows "sat", meaning "satisfiable". In other words, Z3 couldn't find counterexample for this expression.

In fact, I can rewrite this expression in the following form: *expr1 != expr2*, and we would ask Z3 to find at least one set of input arguments, for which expressions are not equal to each other:

```
(declare-const arg1 (_ BitVec 64))
(declare-const arg2 (_ BitVec 64))
(declare-const arg3 (_ BitVec 64))
(declare-const arg4 (_ BitVec 64))

(assert
    (not
        (=
            (bvsub (bvsub (bvsub (bvneg arg1) #x0000000000000003) (bvneg arg1)) (bvsub (
               bvneg arg1) #x0000000000000003))
            arg1
        )
    )
)
(check-sat)
```

Z3 says "unsat", meaning, it couldn't find any such counterexample. In other words, for all possible input arguments, results of these two expressions are always equal to each other.

Nevertheless, Z3 is not omnipotent. It fails to prove equivalence of the code which performs division by multiplication. First of all, I extended it so boths results will have size of 128 bit instead of 64:

```
(declare-const x (_ BitVec 64))
    (assert
        (forall ((x (_ BitVec 64)))
            (=
                ((_ zero_extend 64) (bvudiv x (_ bv17 64)))
                (bvlshr (bvmul ((_ zero_extend 64) x) #x0000000000000000f0f0f0f0f0f0f0f1
                   ) (_ bv68 128))
            )
        )
    )
(check-sat)
(get-model)
```

(*bv17* is just 64-bit number 17, etc. "bv" stands for "bit vector", as opposed to integer value.)

Z3 works too long without any answer, and I had to interrupt it.

As Z3 developers mentioned, such expressions are hard for Z3 so far: `https://github.com/Z3Prover/z3/issues/514`.

Still, division by multiplication can be tested using previously described brute-force check.

## 15.7 My other implementations of toy decompiler

When I made attempt to write it in C++, of course, node in expression was represented using class. There is also implementation in pure C[97], node is represented using structure.

Matchers in both C++ and C versions doesn't return any dictionary, but instead, `bind_value()` functions takes pointer to a variable which will contain value after successful matching. `bind_expr()` takes pointer to a pointer, which will points to the part of expression, again, in case of success. I took this idea from LLVM.

Here are two pieces of code from LLVM source code with couple of reducing rules:

[97] `https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/toy_decompiler/files/C`

```
// (X >> A) << A -> X
  Value *X;
  if (match(Op0, m_Exact(m_Shr(m_Value(X), m_Specific(Op1)))))
    return X;
```

( lib/Analysis/InstructionSimplify.cpp )

```
// (A | B) | C  and  A | (B | C)                  -> bswap if possible.
  // (A >> B) | (C << D)  and  (A << B) | (B >> C)  -> bswap if possible.
  if (match(Op0, m_Or(m_Value(), m_Value())) ||
      match(Op1, m_Or(m_Value(), m_Value())) ||
      (match(Op0, m_LogicalShift(m_Value(), m_Value())) &&
       match(Op1, m_LogicalShift(m_Value(), m_Value())))) {
    if (Instruction *BSwap = MatchBSwap(I))
      return BSwap;
```

( lib/Transforms/InstCombine/InstCombineAndOrXor.cpp )

As you can see, my matcher tries to mimic LLVM. What I call *reduction* is called *folding* in LLVM. Both terms are popular.

I have also a blog post about LLVM obfuscator, in which LLVM matcher is mentioned: https://yurichev.com/blog/llvm/.

Python version of toy decompiler uses strings in place where enumerate data type is used in C version (like *OP_AND*, *OP_MUL*, etc) and symbols used in Racket version[98] (like *'OP_DIV*, etc). This may be seen as inefficient, nevertheless, thanks to strings interning, only address of strings are compared in Python version, not strings as a whole. So strings in Python can be seen as possible replacement for LISP symbols.

### 15.7.1 Even simpler toy decompiler

Knowledge of LISP makes you understand all these things naturally, without significant effort. But when I had no knowledge of it, but still tried to make a simple toy decompiler, I made it using usual text strings which holded expressions for each registers (and even memory).

So when MOV instruction copies value from one register to another, we just copy string. When arithmetical instruction occurred, we do string concatenation:

```
std::string registers[TOTAL];

...

// all 3 arguments are strings
switch (ins, op1, op2)
{
    ...
    case ADD:    registers[op1]="(" + registers[op1] + " + " + registers[op2] + ")";
                 break;
    ...
    case MUL:    registers[op1]="(" + registers[op1] + " / " + registers[op2] + ")";
                 break;
    ...
}
```

Now you'll have long expressions for each register, represented as strings. For reducing them, you can use plain simple regular expression matcher.

For example, for the rule `(X*n)+(X*m) -> X*(n+m)`, you can match (sub)string using the following regular expression: `((.*)*(.*))+((.*)*(.*))` [99]. If the string is matched, you're getting 4 groups (or substrings). You then just compare 1st and 3rd using string comparison function, then you check if the 2nd and 4th are numbers, you convert them to numbers,

[98] Racket is Scheme (which is, in turn, LISP dialect) dialect. https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/toy_decompiler/files/Racket

[99] This regular expression string hasn't been properly escaped, for the reason of easier readability and understanding.

sum them and you make new string, consisting of 1st group and sum, like this: `(" + X + "*" + (int(n) + int(m)) + ")`.

It was naïve, clumsy, it was source of great embarrassment, but it worked correctly.

## 15.8 Difference between toy decompiler and commercial-grade one

Perhaps, someone, who currently reading this text, may rush into extending my source code. As an exercise, I would say, that the first step could be support of partial registers: i.e., AL, AX, EAX. This is tricky, but doable.

Another task may be support of FPU[100] x86 instructions (FPU stack modeling isn't a big deal).

The gap between toy decompiler and a commercial decompiler like Hex-Rays is still enormous. Several tricky problems must be solved, at least these:

- C data types: arrays, structures, pointers, etc. This problem is virtually non-existent for JVM[101] (Java, etc) and .NET decompilers, because type information is present in binary files.
- Basic blocks, C/C++ statements. Mike Van Emmerik in his thesis [102] shows how this can be tackled using SSA forms (which are also used heavily in compilers).
- Memory support, including local stack. Keep in mind pointer aliasing problem. Again, decompilers of JVM and .NET files are simpler here.

## 15.9 Further reading

There are several interesting open-source attempts to build decompiler. Both source code and theses are interesting study.

- *decomp* by Jim Reuter[103].
- *DCC* by Cristina Cifuentes[104].

  It is interesting that this decompiler supports only one type (*int*). Maybe this is a reason why DCC decompiler produces source code with *.B* extension? Read more about B typeless language (C predecessor): `https://yurichev.com/blog/typeless/`.
- *Boomerang* by Mike Van Emmerik, Trent Waddington et al[105].

As I've said, LISP knowledge can help to understand this all much easier. Here is well-known micro-interpreter of LISP by Peter Norvig, also written in Python: `https://web.archive.org/web/20161116133448/http://www.norvig.com/lispy.html`, `https://web.archive.org/web/20160305172301/http://norvig.com/lispy2.html`.

## 15.10 The files

Python version and tests: `https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/toy_decompiler/files`.

There are also C and Racket versions, but outdated.

Keep in mind—this decompiler is still at toy level, and it was tested only on tiny test files supplied.

[100] Floating-point unit
[101] Java Virtual Machine
[102] `https://yurichev.com/mirrors/vanEmmerik_ssa.pdf`
[103] `http://www.program-transformation.org/Transform/DecompReadMe`, `http://www.program-transformation.org/Transform/DecompDecompiler`
[104] `http://www.program-transformation.org/Transform/DccDecompiler`, thesis: `https://yurichev.com/mirrors/DCC_decompilation_thesis.pdf`
[105] `http://boomerang.sourceforge.net/`, `http://www.program-transformation.org/Transform/MikeVanEmmerik`, thesis: `https://yurichev.com/mirrors/vanEmmerik_ssa.pdf`

# 16 Symbolic execution

## 16.1 Symbolic computation

Let's first start with symbolic computation[106].

Some numbers can only be represented in binary system approximately, like $\frac{1}{3}$ and $\pi$. If we calculate $\frac{1}{3} \cdot 3$ step-by-step, we may have loss of significance. We also know that $sin(\frac{\pi}{2}) = 1$, but calculating this expression in usual way, we can also have some noise in result. Arbitrary-precision arithmetic[107] is not a solution, because these numbers cannot be stored in memory as a binary number of finite length.

How we could tackle this problem? Humans reduce such expressions using paper and pencil without any calculations. We can mimic human behaviour programmatically if we will store expression as tree and symbols like $\pi$ will be converted into number at the very last step(s).

This is what Wolfram Mathematica[108] does. Let's start it and try this:

```
In[]:= x + 2*8
Out[]= 16 + x
```

Since Mathematica has no clue what $x$ is, it's left *as is*, but $2 \cdot 8$ can be reduced easily, both by Mathematica and by humans, so that is what has done. In some point of time in future, Mathematica's user may assign some number to $x$ and then, Mathematica will reduce the expression even further.

Mathematica does this because it parses the expression and finds some known patterns. This is also called *term rewriting*[109]. In plain English language it may sounds like this: "if there is a + operator between two known numbers, replace this subexpression by a computed number which is sum of these two numbers, if possible". Just like humans do.

Mathematica also has rules like "replace $sin(\pi)$ by 0" and "replace $sin(\frac{\pi}{2})$ by 1", but as you can see, $\pi$ must be preserved as some kind of symbol instead of a number.

So Mathematica left $x$ as unknown value. This is, in fact, common mistake by Mathematica's users: a small typo in an input expression may lead to a huge irreducible expression with the typo left.

Another example: Mathematica left this deliberately while computing binary logarithm:

```
In[]:= Log[2, 36]
Out[]= Log[36]/Log[2]
```

Because it has a hope that at some point in future, this expression will become a subexpression in another expression and it will be reduced nicely at the very end. But if we really need a numerical answer, we can force Mathematica to calculate it:

```
In[]:= Log[2, 36] // N
Out[]= 5.16993
```

Sometimes unresolved values are desirable:

```
In[]:= Union[{a, b, a, c}, {d, a, e, b}, {c, a}]
Out[]= {a, b, c, d, e}
```

Characters in the expression are just unresolved symbols[110] with no connections to numbers or other expressions, so Mathematica left them *as is*.

Another real world example is symbolic integration[111], i.e., finding formula for integral by rewriting initial expression using some predefined rules. Mathematica also does it:

```
In[]:= Integrate[1/(x^5), x]
Out[]= -(1/(4 x^4))
```

[106] https://en.wikipedia.org/wiki/Symbolic_computation
[107] https://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic
[108] Another well-known symbolic computation system are Maxima and SymPy
[109] https://en.wikipedia.org/wiki/Rewriting
[110] *Symbol* like in LISP
[111] https://en.wikipedia.org/wiki/Symbolic_integration

Benefits of symbolic computation are obvious: it is not prone to loss of significance[112] and round-off errors[113], but drawbacks are also obvious: you need to store expression in (possible huge) tree and process it many times. Term rewriting is also slow. All these things are extremely clumsy in comparison to a fast FPU.

"Symbolic computation" is opposed to "numerical computation", the last one is just processing numbers step-by-step, using calculator, CPU or FPU.

Some task can be solved better by the first method, some others – by the second one.

### 16.1.1 Rational data type

Some LISP implementations can store a number as a ratio/fraction [114], i.e., placing two numbers in a cell (which, in this case, is called *atom* in LISP lingo). For example, you divide 1 by 3, and the interpreter, by understanding that $\frac{1}{3}$ is an irreducible fraction[115], creates a cell with 1 and 3 numbers. Some time after, you may multiply this cell by 6, and the multiplication function inside LISP interpreter may return much better result (2 without *noise*).

Printing function in interpreter can also print something like `1 / 3` instead of floating point number.

This is sometimes called "fractional arithmetic" [see Donald E. Knuth, *The Art of Computing Programming*, 3rd ed., (1997), 4.5.1, page 330].

This is not symbolic computation in any way, but this is slightly better than storing ratios/fractions as just floating point numbers.

Drawbacks are clearly visible: you need more memory to store ratio instead of a number; and all arithmetic functions are more complex and slower, because they must handle both numbers and ratios.

Perhaps, because of drawbacks, some programming languages offers separate (*rational*) data type, as language feature, or supported by a library [116]: Haskell, OCaml, Perl, Ruby, Python (*fractions*), Smalltalk, Java, Clojure, C/C++[117].

## 16.2 Symbolic execution

### 16.2.1 Swapping two values using XOR

There is a well-known (but counterintuitive) algorithm for swapping two values in two variables using XOR operation without use of any additional memory/register:

```
X=X^Y
Y=Y^X
X=X^Y
```

How it works? It would be better to construct an expression at each step of execution.

```
#!/usr/bin/env python
class Expr:
    def __init__(self,s):
        self.s=s

    def __str__(self):
        return self.s

    def __xor__(self, other):
        return Expr("(" + self.s + "^" + other.s + ")")

def XOR_swap(X, Y):
    X=X^Y
    Y=Y^X
    X=X^Y
    return X, Y
```

[112]https://en.wikipedia.org/wiki/Loss_of_significance
[113]https://en.wikipedia.org/wiki/Round-off_error
[114]https://en.wikipedia.org/wiki/Rational_data_type
[115]https://en.wikipedia.org/wiki/Irreducible_fraction
[116]More detailed list: https://en.wikipedia.org/wiki/Rational_data_type
[117]By GNU Multiple Precision Arithmetic Library

```
new_X, new_Y=XOR_swap(Expr("X"), Expr("Y"))
print "new_X", new_X
print "new_Y", new_Y
```

It works, because Python is dynamicaly typed PL, so the function doesn't care what to operate on, numerical values, or on objects of Expr() class.

Here is result:

```
new_X ((X^Y)^(Y^(X^Y)))
new_Y (Y^(X^Y))
```

You can remove double variables in your mind (since XORing by a value twice will result in nothing). At new_X we can drop two X-es and two Y-es, and single Y will left. At new_Y we can drop two Y-es, and single X will left.

#### 16.2.2 Change endianness

What does this code do?

```
mov     eax, ecx
mov     edx, ecx
shl     edx, 16
and     eax, 0000ff00H
or      eax, edx
mov     edx, ecx
and     edx, 00ff0000H
shr     ecx, 16
or      edx, ecx
shl     eax, 8
shr     edx, 8
or      eax, edx
```

In fact, many reverse engineers play shell game a lot, keeping track of what is stored where, at each point of time.

Figure 41: Hieronymus Bosch – The Conjurer

Again, we can build equivalent function which can take both numerical variables and Expr() objects. We also extend Expr() class to support many arithmetical and boolean operations. Also, Expr() methods would take both Expr() objects on input and integer values.

```
#!/usr/bin/env python
class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __xor__(self, other):
        return Expr("(" + self.s + "^" + self.convert_to_Expr_if_int(other).s + ")")

    def __and__(self, other):
        return Expr("(" + self.s + "&" + self.convert_to_Expr_if_int(other).s + ")")
```

```
    def __or__(self, other):
        return Expr("(" + self.s + "|" + self.convert_to_Expr_if_int(other).s + ")")

    def __lshift__(self, other):
        return Expr("(" + self.s + "<<" + self.convert_to_Expr_if_int(other).s + ")")

    def __rshift__(self, other):
        return Expr("(" + self.s + ">>" + self.convert_to_Expr_if_int(other).s + ")")

# change endianness
ecx=Expr("initial_ECX") # 1st argument
eax=ecx             #  mov     eax, ecx
edx=ecx             #  mov     edx, ecx
edx=edx<<16         #  shl     edx, 16
eax=eax&0xff00      #  and     eax, 0000ff00H
eax=eax|edx         #  or      eax, edx
edx=ecx             #  mov     edx, ecx
edx=edx&0x00ff0000  #  and     edx, 00ff0000H
ecx=ecx>>16         #  shr     ecx, 16
edx=edx|ecx         #  or      edx, ecx
eax=eax<<8          #  shl     eax, 8
edx=edx>>8          #  shr     edx, 8
eax=eax|edx         #  or      eax, edx

print eax
```

I run it:

```
((((initial_ECX&65280)|(initial_ECX<<16))<<8)|(((initial_ECX&16711680)|(initial_ECX>>16)
    )>>8))
```

Now this is something more readable, however, a bit LISPy at first sight. In fact, this is a function which change endianness in 32-bit word.

By the way, my Toy Decompiler can do this job as well, but operates on AST instead of plain strings: 15.

### 16.2.3 Fast Fourier transform

I've found one of the smallest possible FFT implementations on reddit:

```
#!/usr/bin/env python
from cmath import exp,pi

def FFT(X):
    n = len(X)
    w = exp(-2*pi*1j/n)
    if n > 1:
        X = FFT(X[::2]) + FFT(X[1::2])
        for k in xrange(n/2):
            xk = X[k]
            X[k] = xk + w**k*X[k+n/2]
            X[k+n/2] = xk - w**k*X[k+n/2]
    return X

print FFT([1,2,3,4,5,6,7,8])
```

Just interesting, what value has each element on output?

```
#!/usr/bin/env python
```

```
from cmath import exp,pi

class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __add__(self, other):
        return Expr("(" + self.s + "+" + self.convert_to_Expr_if_int(other).s + ")")

    def __sub__(self, other):
        return Expr("(" + self.s + "-" + self.convert_to_Expr_if_int(other).s + ")")

    def __mul__(self, other):
        return Expr("(" + self.s + "*" + self.convert_to_Expr_if_int(other).s + ")")

    def __pow__(self, other):
        return Expr("(" + self.s + "**" + self.convert_to_Expr_if_int(other).s + ")")

def FFT(X):
    n = len(X)
    # cast complex value to string, and then to Expr
    w = Expr(str(exp(-2*pi*1j/n)))
    if n > 1:
        X = FFT(X[::2]) + FFT(X[1::2])
        for k in xrange(n/2):
            xk = X[k]
            X[k] = xk + w**k*X[k+n/2]
            X[k+n/2] = xk - w**k*X[k+n/2]
    return X

input=[Expr("input_%d" % i) for i in range(8)]
output=FFT(input)
for i in range(len(output)):
    print i, ":", output[i]
```

FFT() function left almost intact, the only thing I added: complex value is converted into string and then Expr() object is constructed.

```
0 : (((input_0+(((-1-1.22464679915e-16j)**0)*input_4))+(((6.12323399574e-17-1j)**0)*(
    input_2+(((-1-1.22464679915e-16j)**0)*input_6))))+(((0.707106781187-0.707106781187j)
    **0)*((input_1+(((-1-1.22464679915e-16j)**0)*input_5))+(((6.12323399574e-17-1j)**0)
    *(input_3+(((-1-1.22464679915e-16j)**0)*input_7))))))
1 : (((input_0-(((-1-1.22464679915e-16j)**0)*input_4))+(((6.12323399574e-17-1j)**1)*(
    input_2-(((-1-1.22464679915e-16j)**0)*input_6))))+(((0.707106781187-0.707106781187j)
    **1)*((input_1-(((-1-1.22464679915e-16j)**0)*input_5))+(((6.12323399574e-17-1j)**1)
    *(input_3-(((-1-1.22464679915e-16j)**0)*input_7))))))
2 : (((input_0+(((-1-1.22464679915e-16j)**0)*input_4))-(((6.12323399574e-17-1j)**0)*(
```

```
    input_2+(((-1-1.22464679915e-16j)**0)*input_6))))+(((0.707106781187-0.707106781187j)
    **2)*((input_1+(((-1-1.22464679915e-16j)**0)*input_5))-(((6.12323399574e-17-1j)**0)
    *(input_3+(((-1-1.22464679915e-16j)**0)*input_7))))))
3 : (((input_0-(((-1-1.22464679915e-16j)**0)*input_4))-(((6.12323399574e-17-1j)**1)*(
    input_2-(((-1-1.22464679915e-16j)**0)*input_6))))+(((0.707106781187-0.707106781187j)
    **3)*((input_1-(((-1-1.22464679915e-16j)**0)*input_5))-(((6.12323399574e-17-1j)**1)
    *(input_3-(((-1-1.22464679915e-16j)**0)*input_7))))))
4 : (((input_0+(((-1-1.22464679915e-16j)**0)*input_4))+(((6.12323399574e-17-1j)**0)*(
    input_2+(((-1-1.22464679915e-16j)**0)*input_6))))-(((0.707106781187-0.707106781187j)
    **0)*((input_1+(((-1-1.22464679915e-16j)**0)*input_5))+(((6.12323399574e-17-1j)**0)
    *(input_3+(((-1-1.22464679915e-16j)**0)*input_7))))))
5 : (((input_0-(((-1-1.22464679915e-16j)**0)*input_4))+(((6.12323399574e-17-1j)**1)*(
    input_2-(((-1-1.22464679915e-16j)**0)*input_6))))-(((0.707106781187-0.707106781187j)
    **1)*((input_1-(((-1-1.22464679915e-16j)**0)*input_5))+(((6.12323399574e-17-1j)**1)
    *(input_3-(((-1-1.22464679915e-16j)**0)*input_7))))))
6 : (((input_0+(((-1-1.22464679915e-16j)**0)*input_4))-(((6.12323399574e-17-1j)**0)*(
    input_2+(((-1-1.22464679915e-16j)**0)*input_6))))-(((0.707106781187-0.707106781187j)
    **2)*((input_1+(((-1-1.22464679915e-16j)**0)*input_5))-(((6.12323399574e-17-1j)**0)
    *(input_3+(((-1-1.22464679915e-16j)**0)*input_7))))))
7 : (((input_0-(((-1-1.22464679915e-16j)**0)*input_4))-(((6.12323399574e-17-1j)**1)*(
    input_2-(((-1-1.22464679915e-16j)**0)*input_6))))-(((0.707106781187-0.707106781187j)
    **3)*((input_1-(((-1-1.22464679915e-16j)**0)*input_5))-(((6.12323399574e-17-1j)**1)
    *(input_3-(((-1-1.22464679915e-16j)**0)*input_7))))))
```

We can see subexpressions in form like $x^0$ and $x^1$. We can eliminate them, since $x^0 = 1$ and $x^1 = x$. Also, we can reduce subexpressions like $x \cdot 1$ to just $x$.

```
    def __mul__(self, other):
        op1=self.s
        op2=self.convert_to_Expr_if_int(other).s

        if op1=="1":
            return Expr(op2)
        if op2=="1":
            return Expr(op1)

        return Expr("(" + op1 + "*" + op2 + ")")

    def __pow__(self, other):
        op2=self.convert_to_Expr_if_int(other).s
        if op2=="0":
            return Expr("1")
        if op2=="1":
            return Expr(self.s)

        return Expr("(" + self.s + "**" + op2 + ")")
```

```
0 : (((input_0+input_4)+(input_2+input_6))+((input_1+input_5)+(input_3+input_7)))
1 : (((input_0-input_4)+((6.12323399574e-17-1j)*(input_2-input_6)))
    +((0.707106781187-0.707106781187j)*((input_1-input_5)+((6.12323399574e-17-1j)*(
    input_3-input_7)))))
2 : (((input_0+input_4)-(input_2+input_6))+(((0.707106781187-0.707106781187j)**2)*((
    input_1+input_5)-(input_3+input_7))))
3 : (((input_0-input_4)-((6.12323399574e-17-1j)*(input_2-input_6)))
    +(((0.707106781187-0.707106781187j)**3)*((input_1-input_5)-((6.12323399574e-17-1j)*(
    input_3-input_7)))))
4 : (((input_0+input_4)+(input_2+input_6))-((input_1+input_5)+(input_3+input_7)))
```

```
5 : (((input_0-input_4)+((6.12323399574e-17-1j)*(input_2-input_6)))
    -((0.707106781187-0.707106781187j)*((input_1-input_5)+((6.12323399574e-17-1j)*(
    input_3-input_7)))))
6 : (((input_0+input_4)-(input_2+input_6))-(((0.707106781187-0.707106781187j)**2)*((
    input_1+input_5)-(input_3+input_7))))
7 : (((input_0-input_4)-((6.12323399574e-17-1j)*(input_2-input_6)))
    -(((0.707106781187-0.707106781187j)**3)*((input_1-input_5)-((6.12323399574e-17-1j)*(
    input_3-input_7)))))
```

### 16.2.4 Cyclic redundancy check

I've always been wondering, which input bit affects which bit in the final CRC32 value.

From the CRC theory (good and concise introduction: http://web.archive.org/web/20161220015646/http://www.hackersdelight.org/crc.pdf ) we know that CRC is shifting register with taps.

We will track each bit rather than byte or word, which is highly inefficient, but serves our purpose better:

```
#!/usr/bin/env python
import sys

class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __xor__(self, other):
        return Expr("(" + self.s + "^" + self.convert_to_Expr_if_int(other).s + ")")

BYTES=1

def crc32(buf):
    #state=[Expr("init_%d" % i) for i in range(32)]
    state=[Expr("1") for i in range(32)]
    for byte in buf:
        for n in range(8):
            bit=byte[n]
            to_taps=bit^state[31]
            state[31]=state[30]
            state[30]=state[29]
            state[29]=state[28]
            state[28]=state[27]
            state[27]=state[26]
            state[26]=state[25]^to_taps
            state[25]=state[24]
            state[24]=state[23]
            state[23]=state[22]^to_taps
            state[22]=state[21]^to_taps
            state[21]=state[20]
```

```
            state[20]=state[19]
            state[19]=state[18]
            state[18]=state[17]
            state[17]=state[16]
            state[16]=state[15]^to_taps
            state[15]=state[14]
            state[14]=state[13]
            state[13]=state[12]
            state[12]=state[11]^to_taps
            state[11]=state[10]^to_taps
            state[10]=state[9]^to_taps
            state[9]=state[8]
            state[8]=state[7]^to_taps
            state[7]=state[6]^to_taps
            state[6]=state[5]
            state[5]=state[4]^to_taps
            state[4]=state[3]^to_taps
            state[3]=state[2]
            state[2]=state[1]^to_taps
            state[1]=state[0]^to_taps
            state[0]=to_taps

    for i in range(32):
        print "state %d=%s" % (i, state[31-i])

buf=[[Expr("in_%d_%d" % (byte, bit)) for bit in range(8)] for byte in range(BYTES)]
crc32(buf)
```

Here are expressions for each CRC32 bit for 1-byte buffer:

```
state 0=(1^(in_0_2^1))
state 1=((1^(in_0_0^1))^(in_0_3^1))
state 2=(((1^(in_0_0^1))^(in_0_1^1))^(in_0_4^1))
state 3=(((1^(in_0_1^1))^(in_0_2^1))^(in_0_5^1))
state 4=(((1^(in_0_2^1))^(in_0_3^1))^(in_0_6^(1^(in_0_0^1))))
state 5=(((1^(in_0_3^1))^(in_0_4^1))^(in_0_7^(1^(in_0_1^1))))
state 6=((1^(in_0_4^1))^(in_0_5^1))
state 7=((1^(in_0_5^1))^(in_0_6^(1^(in_0_0^1))))
state 8=(((1^(in_0_0^1))^(in_0_6^(1^(in_0_0^1))))^(in_0_7^(1^(in_0_1^1))))
state 9=((1^(in_0_1^1))^(in_0_7^(1^(in_0_1^1))))
state 10=(1^(in_0_2^1))
state 11=(1^(in_0_3^1))
state 12=((1^(in_0_0^1))^(in_0_4^1))
state 13=(((1^(in_0_0^1))^(in_0_1^1))^(in_0_5^1))
state 14=((((1^(in_0_0^1))^(in_0_1^1))^(in_0_2^1))^(in_0_6^(1^(in_0_0^1))))
state 15=((((1^(in_0_1^1))^(in_0_2^1))^(in_0_3^1))^(in_0_7^(1^(in_0_1^1))))
state 16=((((1^(in_0_0^1))^(in_0_2^1))^(in_0_3^1))^(in_0_4^1))
state 17=(((((1^(in_0_0^1))^(in_0_1^1))^(in_0_3^1))^(in_0_4^1))^(in_0_5^1))
state 18=(((((1^(in_0_1^1))^(in_0_2^1))^(in_0_4^1))^(in_0_5^1))^(in_0_6^(1^(in_0_0^1))))
state 19=((((((1^(in_0_0^1))^(in_0_2^1))^(in_0_3^1))^(in_0_5^1))^(in_0_6^(1^(in_0_0^1)))
   )^(in_0_7^(1^(in_0_1^1))))
state 20=((((((1^(in_0_0^1))^(in_0_1^1))^(in_0_3^1))^(in_0_4^1))^(in_0_6^(1^(in_0_0^1)))
   )^(in_0_7^(1^(in_0_1^1))))
state 21=(((((1^(in_0_1^1))^(in_0_2^1))^(in_0_4^1))^(in_0_5^1))^(in_0_7^(1^(in_0_1^1))))
state 22=(((((1^(in_0_0^1))^(in_0_2^1))^(in_0_3^1))^(in_0_5^1))^(in_0_6^(1^(in_0_0^1))))
state 23=((((((1^(in_0_0^1))^(in_0_1^1))^(in_0_3^1))^(in_0_4^1))^(in_0_6^(1^(in_0_0^1)))
   )^(in_0_7^(1^(in_0_1^1))))
```

```
state 24=((((((in_0_0^1)^(in_0_1^1))^(in_0_2^1))^(in_0_4^1))^(in_0_5^1))^(in_0_7^(1^(
    in_0_1^1))))
state 25=(((((in_0_1^1)^(in_0_2^1))^(in_0_3^1))^(in_0_5^1))^(in_0_6^(1^(in_0_0^1))))
state 26=(((((in_0_2^1)^(in_0_3^1))^(in_0_4^1))^(in_0_6^(1^(in_0_0^1))))^(in_0_7^(1^(
    in_0_1^1))))
state 27=((((in_0_3^1)^(in_0_4^1))^(in_0_5^1))^(in_0_7^(1^(in_0_1^1))))
state 28=(((in_0_4^1)^(in_0_5^1))^(in_0_6^(1^(in_0_0^1))))
state 29=(((in_0_5^1)^(in_0_6^(1^(in_0_0^1))))^(in_0_7^(1^(in_0_1^1))))
state 30=((in_0_6^(1^(in_0_0^1)))^(in_0_7^(1^(in_0_1^1))))
state 31=(in_0_7^(1^(in_0_1^1)))
```

For larger buffer, expressions gets increasing exponentially. This is 0th bit of the final state for 4-byte buffer:

```
state 0=((((((((((((((in_0_0^1)^(in_0_1^1))^(in_0_2^1))^(in_0_4^1))^(in_0_5^1))^(in_0_7
    ^(1^(in_0_1^1))))^
(in_1_0^(1^(in_0_2^1))))^(in_1_2^(((1^(in_0_0^1))^(in_0_1^1))^(in_0_4^1))))^(in_1_3
    ^(((1^(in_0_1^1))^
(in_0_2^1))^(in_0_5^1))))^(in_1_4^(((1^(in_0_2^1))^(in_0_3^1))^(in_0_6^(1^(in_0_0^1)))))
    )^(in_2_0^((((1^
(in_0_0^1))^(in_0_6^(1^(in_0_0^1))))^(in_0_7^(1^(in_0_1^1))))^(in_1_2^(((1^(in_0_0^1))^(
    in_0_1^1))^(in_0_4^
1))))))^(in_2_6^(((((((1^(in_0_0^1))^(in_0_1^1))^(in_0_2^1))^(in_0_6^(1^(in_0_0^1))))^(
    in_1_4^(((1^(in_0_2^1))^
(in_0_3^1))^(in_0_6^(1^(in_0_0^1))))))^(in_1_5^(((1^(in_0_3^1))^(in_0_4^1))^(in_0_7^(1^(
    in_0_1^1))))))^
(in_2_0^((((1^(in_0_0^1))^(in_0_6^(1^(in_0_0^1))))^(in_0_7^(1^(in_0_1^1))))^(in_1_2
    ^(((1^(in_0_0^1))^(in_0_1^1))^
(in_0_4^1))))))))^(in_2_7^(((((((1^(in_0_1^1))^(in_0_2^1))^(in_0_3^1))^(in_0_7^(1^(
    in_0_1^1))))^(in_1_5^(((1^
(in_0_3^1))^(in_0_4^1))^(in_0_7^(1^(in_0_1^1))))))^(in_1_6^(((1^(in_0_4^1))^(in_0_5^1))
    ^(in_1_0^(1^(in_0_2^
1))))))^(in_2_1^((((1^(in_0_1^1))^(in_0_7^(1^(in_0_1^1))))^(in_1_0^(1^(in_0_2^1))))^(
    in_1_3^(((1^(in_0_1^1))^
(in_0_2^1))^(in_0_5^1))))))))^(in_3_2^(((((((((1^(in_0_1^1))^(in_0_2^1))^(in_0_4^1))^(
    in_0_5^1))^(in_0_6^(1^
(in_0_0^1))))^(in_1_2^(((1^(in_0_0^1))^(in_0_1^1))^(in_0_4^1))))^(in_2_0^((((1^(in_0_0
    ^1))^(in_0_6^(1^(in_0_0^
1))))^(in_0_7^(1^(in_0_1^1))))^(in_1_2^(((1^(in_0_0^1))^(in_0_1^1))^(in_0_4^1))))))^(
    in_2_1^((((1^(in_0_1^1))^
(in_0_7^(1^(in_0_1^1))))^(in_1_0^(1^(in_0_2^1))))^(in_1_3^(((1^(in_0_1^1))^(in_0_2^1))^(
    in_0_5^1))))))^(in_2_4^
(((((1^(in_0_0^1))^(in_0_4^1))^(in_1_2^(((1^(in_0_0^1))^(in_0_1^1))^(in_0_4^1))))^(
    in_1_3^(((1^(in_0_1^1))^
(in_0_2^1))^(in_0_5^1))))^(in_1_6^(((1^(in_0_4^1))^(in_0_5^1))^(in_1_0^(1^(in_0_2^1)))))
    )))))
```

Expression for the 0th bit of the final state for 8-byte buffer has length of $\approx 350KiB$, which is, of course, can be reduced significantly (because this expression is basically XOR tree), but you can feel the weight of it.

Now we can process this expressions somehow to get a smaller picture on what is affecting what. Let's say, if we can find "in_2_3" substring in expression, this means that 3rd bit of 2nd byte of input affects this expression. But even more than that: since this is XOR tree (i.e., expression consisting only of XOR operations), if some input variable is occurring twice, it's *annihilated*, since $x \oplus x = 0$. More than that: if a vairable occurred even number of times (2, 4, 8, etc), it's annihilated, but left if it's occurred odd number of times (1, 3, 5, etc).

```
    for i in range(32):
        #print "state %d=%s" % (i, state[31-i])
        sys.stdout.write ("state %02d: " % i)
        for byte in range(BYTES):
```

```
            for bit in range(8):
                s="in_%d_%d" % (byte, bit)
                if str(state[31-i]).count(s) & 1:
                    sys.stdout.write ("*")
                else:
                    sys.stdout.write (" ")
        sys.stdout.write ("\n")
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/symbolic/4_CRC/2.py )

Now this how each bit of 1-byte input buffer affects each bit of the final CRC32 state:

```
state 00:   *
state 01: *  *
state 02: **  *
state 03:  **  *
state 04: * **  *
state 05:  * **  *
state 06:     **
state 07: *    **
state 08:  *    **
state 09:        *
state 10:   *
state 11:    *
state 12: *   *
state 13: **   *
state 14:  **   *
state 15:   **   *
state 16: * ***
state 17: ** ***
state 18: *** ***
state 19:  *** ***
state 20:    ** **
state 21:   * ** *
state 22:   ** **
state 23:    ** **
state 24: * * ** *
state 25: **** **
state 26: ***** **
state 27:  * *** *
state 28: *   ***
state 29: **   ***
state 30: **    **
state 31:  *     *
```

This is 8*8=64 bits of 8-byte input buffer:

```
state 00:  * ** *  ***  * ** **      *  * ***** ***       *   * **  *
state 01: * * ** *  ***  * ** **      *  * ***** ***       *   * **  *
state 02: ** * ** *  ***  * ** **      *  * ***** ***       *   * **  *
state 03: *** * ** *  ***  * ** **      *  * ***** ***       *   * **  *
state 04: **** * ** *  ***  * ** **      *  * ***** ***       *   * **  *
state 05:  **** * ** *  ***  * ** **      *  * ***** ***       *   * **  *
state 06:  **  ***   ** **   *  ** ***  * * **     ** *** *   *  *    **
state 07: * **  ***   ** **   *  ** ***  * * **     ** *** *   *  *    **
state 08:  * **  ***   ** **   *  ** ***  * * **     ** *** *   *  *    **
state 09:  *** ** *  *   ** *** *  *****  * * ** **   ** * * ** *        *
state 10:  **    *  *** *      * *  * **  * * ** * *   **   *  **   *
state 11:   **    *  *** *      * *  * **  * * ** * *   **   *  **   *
```

```
state 12:    **    *  *** *      * *  * **  * * ** * *   **   *  **   *
state 13:     **    *  *** *      * *  * **  * * ** * *   **   *  **   *
state 14:      **    *  *** *      * *  * **  * * ** * *   **   *  **   *
state 15:       **    *  *** *      * *  * **  * * ** * *   **   *  **   *
state 16:  * ** ****** **   **         **  *  *  *  ** * **  *  *** ***
state 17: * * ** ****** **   **         **  *  *  *  ** * **  *  *** ***
state 18:  * * ** ****** **   **         **  *  *  *  ** * **  *  *** ***
state 19: * * * ** ****** **   **         **  *  *  *  ** * **  *  *** ***
state 20:     ******  ** ** *** **   *  * *  *****   *  **** *  *    ** **
state 21: ** *** **  * *       * **  ** *** ** *      *  * **   *   * ** *
state 22:   ** *  * ***   ** ** * ** *****  *    **    *    *** *   ** **
state 23: *  ** *  * ***   ** ** * ** *****  *    **    *    *** *   ** **
state 24:    * *** * ***  *** *** * *  * *  **  *****    **    * ** * ** *
state 25:  * *   *** ***  * * **** *       **   *  ***     *  *  ***** **
state 26: * * *   *** ***  * * **** *       **   *  ***     *  *  ***** **
state 27: *   ***      * *****  ****    * ***   **   ***  *  **  * * *** *
state 28:    *** * ***      *    *****  ***   * *     *** **   ****   ***
state 29:     *** * ***      *    *****  ***   * *     *** **   ****   ***
state 30: ** *** *  * *** ** *     ** ***    **  *      **  *** * **    **
state 31: * ** *  ***  * ** **      *  * ***** ***       *   * **  *     *
```

### 16.2.5 Linear congruential generator

This is popular PRNG from OpenWatcom CRT[118] library: https://github.com/open-watcom/open-watcom-v2/blob/d468b609ba6ca61eeddad80dd2485e3256fc5261/bld/clib/math/c/rand.c.

What expression it generates on each step?

```
#!/usr/bin/env python
class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __xor__(self, other):
        return Expr("(" + self.s + "^" + self.convert_to_Expr_if_int(other).s + ")")

    def __mul__(self, other):
        return Expr("(" + self.s + "*" + self.convert_to_Expr_if_int(other).s + ")")

    def __add__(self, other):
        return Expr("(" + self.s + "+" + self.convert_to_Expr_if_int(other).s + ")")

    def __and__(self, other):
        return Expr("(" + self.s + "&" + self.convert_to_Expr_if_int(other).s + ")")

    def __rshift__(self, other):
```

[118] C runtime library

```
        return Expr("(" + self.s + ">>" + self.convert_to_Expr_if_int(other).s + ")")

seed=Expr("initial_seed")

def rand():
    global seed
    seed=seed*1103515245+12345
    return (seed>>16) & 0x7fff

for i in range(10):
    print i, ":", rand()
```

```
0 : ((((initial_seed*1103515245)+12345)>>16)&32767)
1 : ((((((initial_seed*1103515245)+12345)*1103515245)+12345)>>16)&32767)
2 : ((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)>>16)
   &32767)
3 : ((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
   *1103515245)+12345)>>16)&32767)
4 : ((((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
   *1103515245)+12345)*1103515245)+12345)>>16)&32767)
5 : ((((((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
   *1103515245)+12345)*1103515245)+12345)*1103515245)+12345)>>16)&32767)
6 : ((((((((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)
   +12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
   >>16)&32767)
7 : ((((((((((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)
   +12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
   *1103515245)+12345)>>16)&32767)
8 : ((((((((((((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)
   +12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
   *1103515245)+12345)*1103515245)+12345)>>16)&32767)
9 : ((((((((((((((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)
   +12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
   *1103515245)+12345)*1103515245)+12345)*1103515245)+12345)>>16)&32767)
```

Now if we once got several values from this PRNG, like 4583, 16304, 14440, 32315, 28670, 12568..., how would we recover the initial seed? The problem in fact is solving a system of equations:

```
((((initial_seed*1103515245)+12345)>>16)&32767)==4583
((((((initial_seed*1103515245)+12345)*1103515245)+12345)>>16)&32767)==16304
((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)>>16)
   &32767)==14440
((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
   *1103515245)+12345)>>16)&32767)==32315
```

As it turns out, Z3 can solve this system correctly using only two equations:

```
#!/usr/bin/env python
from z3 import *

s=Solver()

x=BitVec("x",32)

a=1103515245
c=12345
s.add((((x*a)+c)>>16)&32767==4583)
s.add((((((x*a)+c)*a)+c)>>16)&32767==16304)
```

```
#s.add((((((((x*a)+c)*a)+c)*a)+c)>>16)&32767==14440)
#s.add((((((((((x*a)+c)*a)+c)*a)+c)*a)+c)>>16)&32767==32315)

s.check()
print s.model()
```

```
[x = 11223344]
```

(Though, it takes ≈ 20 seconds on my ancient Intel Atom netbook.)

### 16.2.6 Path constraint

How to get weekday from UNIX timestamp?

```
#!/usr/bin/env python

input=...
SECS_DAY=24*60*60
dayno = input / SECS_DAY
wday = (dayno + 4) % 7
if wday==5:
    print "Thanks God, it's Friday!"
```

Let's say, we should find a way to run the block with print() call in it. What input value should be?

First, let's build expression of *wday* variable:

```
#!/usr/bin/env python
class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __div__(self, other):
        return Expr("(" + self.s + "/" + self.convert_to_Expr_if_int(other).s + ")")

    def __mod__(self, other):
        return Expr("(" + self.s + "%" + self.convert_to_Expr_if_int(other).s + ")")

    def __add__(self, other):
        return Expr("(" + self.s + "+" + self.convert_to_Expr_if_int(other).s + ")")

input=Expr("input")
SECS_DAY=24*60*60
dayno = input / SECS_DAY
wday = (dayno + 4) % 7
print wday
if wday==5:
    print "Thanks God, it's Friday!"
```

```
(((input/86400)+4)%7)
```

In order to execute the block, we should solve this equation: $((\frac{input}{86400}+4) \equiv 5 \mod 7$.

So far, this is easy task for Z3:

```
#!/usr/bin/env python
from z3 import *

s=Solver()

x=Int("x")

s.add(((x/86400)+4)%7==5)

s.check()
print s.model()
```

```
[x = 86438]
```

This is indeed correct UNIX timestamp for Friday:

```
% date --date='@86438'
Fri Jan  2 03:00:38 MSK 1970
```

Though the date back in year 1970, but it's still correct!

This is also called "path constraint", i.e., what constraint must be satisified to execute specific block? Several tools has "path" in their names, like "pathgrind", Symbolic PathFinder, CodeSurfer Path Inspector, etc.

Like the shell game, this task is also often encounters in practice. You can see that something dangerous can be executed inside some basic block and you're trying to deduce, what input values can cause execution of it. It may be buffer overflow, etc. Such input values are sometimes also called "inputs of death".

Many crackmes are solved in this way, all you need is find a path into block which prints "key is correct" or something like that.

We can extend this tiny example:

```
input=...
SECS_DAY=24*60*60
dayno = input / SECS_DAY
wday = (dayno + 4) % 7
print wday
if wday==5:
    print "Thanks God, it's Friday!"
else:
    print "Got to wait a little"
```

Now we have two blocks: for the first we should solve this equation: $((\frac{input}{86400}+4) \equiv 5 \mod 7$. But for the second we should solve inverted equation: $((\frac{input}{86400}+4) \not\equiv 5 \mod 7$. By solving these equations, we will find two paths into both blocks.

KLEE (or similar tool) tries to find path to each [basic] block and produces "ideal" unit test. Hence, KLEE can find a path into the block which crashes everything, or reporting about correctness of the input key/license, etc. Surprisingly, KLEE can find backdoors in the very same manner.

KLEE is also called "KLEE Symbolic Virtual Machine" – by that its creators mean that the KLEE is VM[119] which executes a code symbolically rather than numerically (like usual CPU).

Let's extend our tiny example again. We would like to find Friday 13th. To make things simpler, we can limit ourselves to year 1970. Let's get all 12 13th days of year 1970:

[119]Virtual Machine

```
 % date +"%j" --date="13 Jan 1970"
013
 % date +"%j" --date="13 Feb 1970"
044
 % date +"%j" --date="13 Mar 1970"
072
 % date +"%j" --date="13 Apr 1970"
103
 % date +"%j" --date="13 May 1970"
133
 % date +"%j" --date="13 Jun 1970"
164
 % date +"%j" --date="13 Jul 1970"
194
 % date +"%j" --date="13 Aug 1970"
225
 % date +"%j" --date="13 Sep 1970"
256
 % date +"%j" --date="13 Oct 1970"
286
 % date +"%j" --date="13 Nov 1970"
317
 % date +"%j" --date="13 Dec 1970"
347
```

The script checking if the current date is Friday 13th:

```
input=...
SECS_DAY=24*60*60
dayno = input / SECS_DAY
wday = (dayno + 4) % 7
print wday
if wday==5:
    print "Thanks God, it's Friday!"

    if dayno in [13,44,72,103,133,164,194,225,256,286,317,347]:
        print "Friday 13th"
```

To get the second "print" executed, we must satisfy two constraints:

```
#!/usr/bin/env python
from z3 import *

s=Solver()

x=Int("x")

dayno=Int("dayno")

s.add(dayno==x/86400)

# 1st constraint:
s.add((dayno+4)%7==5) # must be Friday

# 2nd constraint:
s.add(Or(dayno==13-1,dayno==44-1,dayno==72-1,dayno==103-1,dayno==133-1,dayno==164-1,
    dayno==194-1,dayno==225-1,dayno==256-1,dayno==286-1,dayno==317-1,dayno==347-1))
```

```
s.check()
print s.model()
```

Easy task for Z3 as well:

```
 % python Z3_solve2.py
[dayno = 316, x = 27302400]
 % date --date='@27302400'
Fri Nov 13 03:00:00 MSK 1970
```

This is an UNIX date for which both constructs are satisfied: 13th November 1970, Friday.

### 16.2.7 Division by zero

If division by zero is unwrapped by sanitizing check, and exception isn't caught, it can crash process.

Let's calculate simple expression $\frac{x}{2y+4z-12}$. We can add a warning into `__div__` method:

```
#!/usr/bin/env python
class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __mul__(self, other):
        return Expr("(" + self.s + "*" + self.convert_to_Expr_if_int(other).s + ")")

    def __div__(self, other):
        op2=self.convert_to_Expr_if_int(other).s
        print "warning: division by zero if "+op2+"==0"
        return Expr("(" + self.s + "/" + op2 + ")")

    def __add__(self, other):
        return Expr("(" + self.s + "+" + self.convert_to_Expr_if_int(other).s + ")")

    def __sub__(self, other):
        return Expr("(" + self.s + "-" + self.convert_to_Expr_if_int(other).s + ")")

"""
     x
------------
2y + 4z - 12

"""

def f(x, y, z):
    return x/(y*2 + z*4 - 12)

print f(Expr("x"), Expr("y"), Expr("z"))
```

...so it will report about dangerous states and conditions:

```
warning: division by zero if (((y*2)+(z*4))-12)==0
(x/(((y*2)+(z*4))-12))
```

This equation is easy to solve, let's try Wolfram Mathematica this time:

```
In[]:= FindInstance[{(y*2 + z*4) - 12 == 0}, {y, z}, Integers]
Out[]= {{y -> 0, z -> 3}}
```

These values for $y$ and $z$ can also be called "inputs of death".

### 16.2.8 Merge sort

How merge sort works? I have copypasted Python code from rosettacode.com almost intact:

```
#!/usr/bin/env python
class Expr:
    def __init__(self,s,i):
        self.s=s
        self.i=i

    def __str__(self):
        # return both symbolic and integer:
        return self.s+" (" + str(self.i)+")"

    def __le__(self, other):
        # compare only integer parts:
        return self.i <= other.i

# copypasted from http://rosettacode.org/wiki/Sorting_algorithms/Merge_sort#Python
def merge(left, right):
    result = []
    left_idx, right_idx = 0, 0
    while left_idx < len(left) and right_idx < len(right):
        # change the direction of this comparison to change the direction of the sort
        if left[left_idx] <= right[right_idx]:
            result.append(left[left_idx])
            left_idx += 1
        else:
            result.append(right[right_idx])
            right_idx += 1

    if left_idx < len(left):
        result.extend(left[left_idx:])
    if right_idx < len(right):
        result.extend(right[right_idx:])
    return result

def tabs (t):
    return "\t"*t

def merge_sort(m, indent=0):
    print tabs(indent)+"merge_sort() begin. input:"
    for i in m:
        print tabs(indent)+str(i)

    if len(m) <= 1:
        print tabs(indent)+"merge_sort() end. returning single element"
```

```
        return m

    middle = len(m) // 2
    left = m[:middle]
    right = m[middle:]

    left = merge_sort(left, indent+1)
    right = merge_sort(right, indent+1)
    rt=list(merge(left, right))
    print tabs(indent)+"merge_sort() end. returning:"
    for i in rt:
        print tabs(indent)+str(i)
    return rt

# input buffer has both symbolic and numerical values:
input=[Expr("input1",22), Expr("input2",7), Expr("input3",2), Expr("input4",1), Expr("input5",8), Expr("input6",4)]
merge_sort(input)
```

But here is a function which compares elements. Obviously, it wouldn't work correctly without it.

So we can track both expression for each element and numerical value. Both will be printed finally. But whenever values are to be compared, only numerical parts will be used.

Result:

```
merge_sort() begin. input:
input1 (22)
input2 (7)
input3 (2)
input4 (1)
input5 (8)
input6 (4)
        merge_sort() begin. input:
        input1 (22)
        input2 (7)
        input3 (2)
                merge_sort() begin. input:
                input1 (22)
                merge_sort() end. returning single element
                merge_sort() begin. input:
                input2 (7)
                input3 (2)
                        merge_sort() begin. input:
                        input2 (7)
                        merge_sort() end. returning single element
                        merge_sort() begin. input:
                        input3 (2)
                        merge_sort() end. returning single element
                merge_sort() end. returning:
                input3 (2)
                input2 (7)
        merge_sort() end. returning:
        input3 (2)
        input2 (7)
        input1 (22)
        merge_sort() begin. input:
        input4 (1)
        input5 (8)
```

```
        input6 (4)
                merge_sort() begin. input:
                input4 (1)
                merge_sort() end. returning single element
                merge_sort() begin. input:
                input5 (8)
                input6 (4)
                        merge_sort() begin. input:
                        input5 (8)
                        merge_sort() end. returning single element
                        merge_sort() begin. input:
                        input6 (4)
                        merge_sort() end. returning single element
                merge_sort() end. returning:
                input6 (4)
                input5 (8)
        merge_sort() end. returning:
        input4 (1)
        input6 (4)
        input5 (8)
merge_sort() end. returning:
input4 (1)
input3 (2)
input6 (4)
input2 (7)
input5 (8)
input1 (22)
```

### 16.2.9 Extending Expr class

This is somewhat senseless, nevertheless, it's easy task to extend my Expr class to support AST instead of plain strings. It's also possible to add folding steps (like I demonstrated in Toy Decompiler: 15). Maybe someone will want to do this as an exercise. By the way, the toy decompiler can be used as simple symbolic engine as well, just feed all the instructions to it and it will track contents of each register.

### 16.2.10 Conclusion

For the sake of demonstration, I made things as simple as possible. But reality is always harsh and inconvenient, so all this shouldn't be taken as a silver bullet.

The files used in this part: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/symbolic.

## 16.3 Further reading

Robert W. Floyd — Assigning meaning to programs [120].

James C. King — Symbolic Execution and Program Testing [121]

# 17 KLEE

## 17.1 Installation

KLEE building from source is tricky. Easiest way to use KLEE is to install docker[122] and then to run KLEE docker image[123]. The path where KLEE files residing can look like **/var/lib/docker/aufs/mnt/(lots of hexadecimal digits)/home/klee**.

[120] https://classes.soe.ucsc.edu/cmps290g/Fall09/Papers/AssigningMeanings1967.pdf
[121] https://yurichev.com/mirrors/king76symbolicexecution.pdf
[122] https://docs.docker.com/engine/installation/linux/ubuntulinux/
[123] http://klee.github.io/docker/

## 17.2 Unit test: HTML/CSS color

The most popular ways to represent HTML/CSS color is by English name (e.g., “red”) and by 6-digit hexadecimal number (e.g., “#0077CC”). There is third, less popular way: if each byte in hexadecimal number has two doubling digits, it can be *abbreviated*, thus, “#0077CC” can be written just as “#07C”.

Let’s write a function to convert 3 color components into name (if possible, first priority), 3-digit hexadecimal form (if possible, second priority), or as 6-digit hexadecimal form (as a last resort).

```
#include <string.h>
#include <stdio.h>
#include <stdint.h>

void HTML_color(uint8_t R, uint8_t G, uint8_t B, char* out)
{
        if (R==0xFF && G==0 && B==0)
        {
                strcpy (out, "red");
                return;
        };

        if (R==0x0 && G==0xFF && B==0)
        {
                strcpy (out, "green");
                return;
        };

        if (R==0 && G==0 && B==0xFF)
        {
                strcpy (out, "blue");
                return;
        };

        // abbreviated hexadecimal
        if (R>>4==(R&0xF) && G>>4==(G&0xF) && B>>4==(B&0xF))
        {
                sprintf (out, "#%X%X%X", R&0xF, G&0xF, B&0xF);
                return;
        };

        // last resort
        sprintf (out, "#%02X%02X%02X", R, G, B);
};

int main()
{
        uint8_t R, G, B;
        klee_make_symbolic (&R, sizeof R, "R");
        klee_make_symbolic (&G, sizeof R, "G");
        klee_make_symbolic (&B, sizeof R, "B");

        char tmp[16];

        HTML_color(R, G, B, tmp);
};
```

There are 5 possible paths in function, and let’s see, if KLEE could find them all? It’s indeed so:

```
% clang -emit-llvm -c -g color.c
```

```
% klee color.bc
KLEE: output directory is "/home/klee/klee-out-134"
KLEE: WARNING: undefined reference to function: sprintf
KLEE: WARNING: undefined reference to function: strcpy
KLEE: WARNING ONCE: calling external: strcpy(51867584, 51598960)
KLEE: ERROR: /home/klee/color.c:33: external call with symbolic argument: sprintf
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/color.c:28: external call with symbolic argument: sprintf
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 479
KLEE: done: completed paths = 19
KLEE: done: generated tests = 5
```

We can ignore calls to strcpy() and sprintf(), because we are not really interesting in state of `out` variable.

So there are exactly 5 paths:

```
% ls klee-last
assembly.ll   run.stats           test000003.ktest     test000005.ktest
info          test000001.ktest    test000003.pc        test000005.pc
messages.txt  test000002.ktest    test000004.ktest     warnings.txt
run.istats    test000003.exec.err test000005.exec.err
```

1st set of input variables will result in “red” string:

```
% ktest-tool --write-ints klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['color.bc']
num objects: 3
object    0: name: b'R'
object    0: size: 1
object    0: data: b'\xff'
object    1: name: b'G'
object    1: size: 1
object    1: data: b'\x00'
object    2: name: b'B'
object    2: size: 1
object    2: data: b'\x00'
```

2nd set of input variables will result in “green” string:

```
% ktest-tool --write-ints klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['color.bc']
num objects: 3
object    0: name: b'R'
object    0: size: 1
object    0: data: b'\x00'
object    1: name: b'G'
object    1: size: 1
object    1: data: b'\xff'
object    2: name: b'B'
object    2: size: 1
object    2: data: b'\x00'
```

3rd set of input variables will result in “#010000” string:

```
% ktest-tool --write-ints klee-last/test000003.ktest
```

```
ktest file : 'klee-last/test000003.ktest'
args       : ['color.bc']
num objects: 3
object    0: name: b'R'
object    0: size: 1
object    0: data: b'\x01'
object    1: name: b'G'
object    1: size: 1
object    1: data: b'\x00'
object    2: name: b'B'
object    2: size: 1
object    2: data: b'\x00'
```

4th set of input variables will result in "blue" string:

```
% ktest-tool --write-ints klee-last/test000004.ktest
ktest file : 'klee-last/test000004.ktest'
args       : ['color.bc']
num objects: 3
object    0: name: b'R'
object    0: size: 1
object    0: data: b'\x00'
object    1: name: b'G'
object    1: size: 1
object    1: data: b'\x00'
object    2: name: b'B'
object    2: size: 1
object    2: data: b'\xff'
```

5th set of input variables will result in "#F01" string:

```
% ktest-tool --write-ints klee-last/test000005.ktest
ktest file : 'klee-last/test000005.ktest'
args       : ['color.bc']
num objects: 3
object    0: name: b'R'
object    0: size: 1
object    0: data: b'\xff'
object    1: name: b'G'
object    1: size: 1
object    1: data: b'\x00'
object    2: name: b'B'
object    2: size: 1
object    2: data: b'\x11'
```

These 5 sets of input variables can form a unit test for our function.

## 17.3 Unit test: strcmp() function

The standard `strcmp()` function from C library can return 0, -1 or 1, depending of comparison result.

Here is my own implementation of `strcmp()`:

```
int my_strcmp(const char *s1, const char *s2)
{
	int ret = 0;

	while (1)
	{
		ret = *(unsigned char *) s1 - *(unsigned char *) s2;
```

```
                if (ret!=0)
                        break;
                if ((*s1==0) || (*s2)==0)
                        break;
                s1++;
                s2++;
        };

        if (ret < 0)
        {
                return -1;
        } else if (ret > 0)
        {
                return 1;
        }

        return 0;
}

int main()
{
        char input1[2];
        char input2[2];

        klee_make_symbolic(input1, sizeof input1, "input1");
        klee_make_symbolic(input2, sizeof input2, "input2");

        klee_assume((input1[0]>='a') && (input1[0]<='z'));
        klee_assume((input2[0]>='a') && (input2[0]<='z'));

        klee_assume(input1[1]==0);
        klee_assume(input2[1]==0);

        my_strcmp (input1, input2);
};
```

Let's find out, if KLEE is capable of finding all three paths? I intentionaly made things simpler for KLEE by limiting input arrays to two 2 bytes or to 1 character + terminal zero byte.

```
% clang -emit-llvm -c -g strcmp.c

% klee strcmp.bc
KLEE: output directory is "/home/klee/klee-out-131"
KLEE: ERROR: /home/klee/strcmp.c:35: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/strcmp.c:36: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 137
KLEE: done: completed paths = 5
KLEE: done: generated tests = 5

% ls klee-last
assembly.ll   run.stats           test000002.ktest     test000004.ktest
info          test000001.ktest    test000002.pc        test000005.ktest
messages.txt  test000001.pc       test000002.user.err  warnings.txt
run.istats    test000001.user.err test000003.ktest
```

The first two errors are about `klee_assume()`. These are input values on which `klee_assume()` calls are stuck. We can ignore them, or take a peek out of curiosity:

```
% ktest-tool --write-ints klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['strcmp.bc']
num objects: 2
object    0: name: b'input1'
object    0: size: 2
object    0: data: b'\x00\x00'
object    1: name: b'input2'
object    1: size: 2
object    1: data: b'\x00\x00'

% ktest-tool --write-ints klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['strcmp.bc']
num objects: 2
object    0: name: b'input1'
object    0: size: 2
object    0: data: b'a\xff'
object    1: name: b'input2'
object    1: size: 2
object    1: data: b'\x00\x00'
```

Three rest files are the input values for each path inside of my implementation of `strcmp()`:

```
% ktest-tool --write-ints klee-last/test000003.ktest
ktest file : 'klee-last/test000003.ktest'
args       : ['strcmp.bc']
num objects: 2
object    0: name: b'input1'
object    0: size: 2
object    0: data: b'b\x00'
object    1: name: b'input2'
object    1: size: 2
object    1: data: b'c\x00'

% ktest-tool --write-ints klee-last/test000004.ktest
ktest file : 'klee-last/test000004.ktest'
args       : ['strcmp.bc']
num objects: 2
object    0: name: b'input1'
object    0: size: 2
object    0: data: b'c\x00'
object    1: name: b'input2'
object    1: size: 2
object    1: data: b'a\x00'

% ktest-tool --write-ints klee-last/test000005.ktest
ktest file : 'klee-last/test000005.ktest'
args       : ['strcmp.bc']
num objects: 2
object    0: name: b'input1'
object    0: size: 2
object    0: data: b'a\x00'
object    1: name: b'input2'
object    1: size: 2
```

```
object    1: data: b'a\x00'
```

3rd is about first argument ("b") is lesser than the second ("c"). 4th is opposite ("c" and "a"). 5th is when they are equal ("a" and "a").

Using these 3 test cases, we've got full coverage of our implementation of `strcmp()`.

## 17.4 UNIX date/time

UNIX date/time[124] is a number of seconds that have elapsed since 1-Jan-1970 00:00 UTC. C/C++ gmtime() function is used to decode this value into human-readable date/time.

Here is a piece of code I've copypasted from some ancient version of Minix OS (http://www.cise.ufl.edu/~cop4600/cgi-bin/lxr/http/source.cgi/lib/ansi/gmtime.c) and reworked slightly:

```
#include <stdint.h>
#include <time.h>
#include <assert.h>

/*
 * copypasted and reworked from
 * http://www.cise.ufl.edu/~cop4600/cgi-bin/lxr/http/source.cgi/lib/ansi/loc_time.h
 * http://www.cise.ufl.edu/~cop4600/cgi-bin/lxr/http/source.cgi/lib/ansi/misc.c
 * http://www.cise.ufl.edu/~cop4600/cgi-bin/lxr/http/source.cgi/lib/ansi/gmtime.c
 */

#define YEAR0           1900
#define EPOCH_YR        1970
#define SECS_DAY        (24L * 60L * 60L)
#define YEARSIZE(year)  (LEAPYEAR(year) ? 366 : 365)

const int _ytab[2][12] =
{
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
        { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};

const char *_days[] =
{
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
};

const char *_months[] =
{
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
};

#define LEAPYEAR(year)  (!((year) % 4) && (((year) % 100) || !((year) % 400)))

void decode_UNIX_time(const time_t time)
{
        unsigned int dayclock, dayno;
        int year = EPOCH_YR;

```

[124] https://en.wikipedia.org/wiki/Unix_time

```
	dayclock = (unsigned long)time % SECS_DAY;
	dayno = (unsigned long)time / SECS_DAY;

	int seconds = dayclock % 60;
	int minutes = (dayclock % 3600) / 60;
	int hour = dayclock / 3600;
	int wday = (dayno + 4) % 7;
	while (dayno >= YEARSIZE(year))
	{
		dayno -= YEARSIZE(year);
		year++;
	}

	year = year - YEAR0;

	int month = 0;

	while (dayno >= _ytab[LEAPYEAR(year)][month])
	{
		dayno -= _ytab[LEAPYEAR(year)][month];
		month++;
	}

	char *s;
	switch (month)
	{
		case 0: s="January"; break;
		case 1: s="February"; break;
		case 2: s="March"; break;
		case 3: s="April"; break;
		case 4: s="May"; break;
		case 5: s="June"; break;
		case 6: s="July"; break;
		case 7: s="August"; break;
		case 8: s="September"; break;
		case 9: s="October"; break;
		case 10: s="November"; break;
		case 11: s="December"; break;
		default:
			assert(0);
	};

	printf ("%04d-%s-%02d %02d:%02d:%02d\n", YEAR0+year, s, dayno+1, hour, minutes,
	   seconds);
	printf ("week day: %s\n", _days[wday]);
}

int main()
{
	uint32_t time;

	klee_make_symbolic(&time, sizeof time, "time");

	decode_UNIX_time(time);

	return 0;
```

```
99 }
```

Let's try it:

```
% clang -emit-llvm -c -g klee_time1.c
...

% klee klee_time1.bc
KLEE: output directory is "/home/klee/klee-out-107"
KLEE: WARNING: undefined reference to function: printf
KLEE: ERROR: /home/klee/klee_time1.c:86: external call with symbolic argument: printf
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_time1.c:83: ASSERTION FAIL: 0
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 101579
KLEE: done: completed paths = 1635
KLEE: done: generated tests = 2
```

Wow, assert() at line 83 has been triggered, why? Let's see a value of UNIX time which triggers it:

```
% ls klee-last | grep err
test000001.exec.err
test000002.assert.err

% ktest-tool --write-ints klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['klee_time1.bc']
num objects: 1
object    0: name: b'time'
object    0: size: 4
object    0: data: 978278400
```

Let's decode this value using UNIX date utility:

```
% date -u --date='@978278400'
Sun Dec 31 16:00:00 UTC 2000
```

After my investigation, I've found that `month` variable can hold incorrect value of 12 (while 11 is maximal, for December), because LEAPYEAR() macro should receive year number as 2000, not as 100. So I've introduced a bug during rewritting this function, and KLEE found it!

Just interesting, what would be if I'll replace switch() to array of strings, like it usually happens in concise C/C++ code?

```
        ...

const char *_months[] =
{
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
};

        ...

        while (dayno >= _ytab[LEAPYEAR(year)][month])
        {
                dayno -= _ytab[LEAPYEAR(year)][month];
```

```
                month++;
        }

        char *s=_months[month];

        printf ("%04d-%s-%02d %02d:%02d:%02d\n", YEAR0+year, s, dayno+1, hour, minutes,
            seconds);
        printf ("week day: %s\n", _days[wday]);

        ...
```

KLEE detects attempt to read beyond array boundaries:

```
% klee klee_time2.bc
KLEE: output directory is "/home/klee/klee-out-108"
KLEE: WARNING: undefined reference to function: printf
KLEE: ERROR: /home/klee/klee_time2.c:69: external call with symbolic argument: printf
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_time2.c:67: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 101716
KLEE: done: completed paths = 1635
KLEE: done: generated tests = 2
```

This is the same UNIX time value we've already seen:

```
% ls klee-last | grep err
test000001.exec.err
test000002.ptr.err

% ktest-tool --write-ints klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['klee_time2.bc']
num objects: 1
object    0: name: b'time'
object    0: size: 4
object    0: data: 978278400
```

So, if this piece of code can be triggered on remote computer, with this input value (*input of death*), it's possible to crash the process (with some luck, though).

OK, now I'm fixing a bug by moving year subtracting expression to line 43, and let's find, what UNIX time value corresponds to some fancy date like 2022-February-2?

```
1  #include <stdint.h>
2  #include <time.h>
3  #include <assert.h>
4
5  #define YEAR0           1900
6  #define EPOCH_YR        1970
7  #define SECS_DAY        (24L * 60L * 60L)
8  #define YEARSIZE(year)  (LEAPYEAR(year) ? 366 : 365)
9
10 const int _ytab[2][12] =
11 {
12         { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
13         { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
14 };
```

```
#define LEAPYEAR(year)  (!((year) % 4) && (((year) % 100) || !((year) % 400)))

void decode_UNIX_time(const time_t time)
{
        unsigned int dayclock, dayno;
        int year = EPOCH_YR;

        dayclock = (unsigned long)time % SECS_DAY;
        dayno = (unsigned long)time / SECS_DAY;

        int seconds = dayclock % 60;
        int minutes = (dayclock % 3600) / 60;
        int hour = dayclock / 3600;
        int wday = (dayno + 4) % 7;
        while (dayno >= YEARSIZE(year))
        {
                dayno -= YEARSIZE(year);
                year++;
        }

        int month = 0;

        while (dayno >= _ytab[LEAPYEAR(year)][month])
        {
                dayno -= _ytab[LEAPYEAR(year)][month];
                month++;
        }
        year = year - YEAR0;

        if (YEAR0+year==2022 && month==1 && dayno+1==22)
                klee_assert(0);
}
int main()
{
        uint32_t time;

        klee_make_symbolic(&time, sizeof time, "time");

        decode_UNIX_time(time);

        return 0;
}
```

```
% clang -emit-llvm -c -g klee_time3.c
...

% klee klee_time3.bc
KLEE: output directory is "/home/klee/klee-out-109"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_time3.c:47: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 101087
KLEE: done: completed paths = 1635
```

```
KLEE: done: generated tests = 1635

% ls klee-last | grep err
test000587.external.err

% ktest-tool --write-ints klee-last/test000587.ktest
ktest file : 'klee-last/test000587.ktest'
args       : ['klee_time3.bc']
num objects: 1
object    0: name: b'time'
object    0: size: 4
object    0: data: 1645488640

% date -u --date='@1645488640'
Tue Feb 22 00:10:40 UTC 2022
```

Success, but hours/minutes/seconds are seems random—they are random indeed, because, KLEE satisfied all constraints we've put, nothing else. We didn't ask it to set hours/minutes/seconds to zeroes.

Let's add constraints to hours/minutes/seconds as well:

```
        ...

        if (YEAR0+year==2022 && month==1 && dayno+1==22 && hour==22 && minutes==22 &&
           seconds==22)
                klee_assert(0);

        ...
```

Let's run it and check ...

```
% ktest-tool --write-ints klee-last/test000597.ktest
ktest file : 'klee-last/test000597.ktest'
args       : ['klee_time3.bc']
num objects: 1
object    0: name: b'time'
object    0: size: 4
object    0: data: 1645568542

% date -u --date='@1645568542'
Tue Feb 22 22:22:22 UTC 2022
```

Now that is precise.

Yes, of course, C/C++ libraries has function(s) to encode human-readable date into UNIX time value, but what we've got here is KLEE working *antipode* of decoding function, *inverse function* in a way.

## 17.5 Inverse function for base64 decoder

It's piece of cake for KLEE to reconstruct input base64 string given just base64 decoder code without corresponding encoder code. I've copypasted this piece of code from http://www.opensource.apple.com/source/QuickTimeStreamingServer/QuickTimeStreamingServer-452/CommonUtilitiesLib/base64.c.

We add constraints (lines 84, 85) so that output buffer must have byte values from 0 to 15. We also tell to KLEE that the Base64decode() function must return 16 (i.e., size of output buffer in bytes, line 82).

```
1 #include <string.h>
2 #include <stdint.h>
3 #include <stdbool.h>
4
5 // copypasted from http://www.opensource.apple.com/source/QuickTimeStreamingServer/
    QuickTimeStreamingServer-452/CommonUtilitiesLib/base64.c
```

```

static const unsigned char pr2six[256] =
{
        /* ASCII table */
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 62, 64, 64, 64, 63,
        52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 64, 64, 64, 64, 64, 64,
        64,  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,
        15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 64, 64, 64, 64, 64,
        64, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
        41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64
};

int Base64decode(char *bufplain, const char *bufcoded)
{
        int nbytesdecoded;
        register const unsigned char *bufin;
        register unsigned char *bufout;
        register int nprbytes;

        bufin = (const unsigned char *) bufcoded;
        while (pr2six[*(bufin++)] <= 63);
        nprbytes = (bufin - (const unsigned char *) bufcoded) - 1;
        nbytesdecoded = ((nprbytes + 3) / 4) * 3;

        bufout = (unsigned char *) bufplain;
        bufin = (const unsigned char *) bufcoded;

        while (nprbytes > 4) {
                *(bufout++) =
                        (unsigned char) (pr2six[*bufin] << 2 | pr2six[bufin[1]] >> 4);
                *(bufout++) =
                        (unsigned char) (pr2six[bufin[1]] << 4 | pr2six[bufin[2]] >> 2);
                *(bufout++) =
                        (unsigned char) (pr2six[bufin[2]] << 6 | pr2six[bufin[3]]);
                bufin += 4;
                nprbytes -= 4;
        }

        /* Note: (nprbytes == 1) would be an error, so just ingore that case */
        if (nprbytes > 1) {
                *(bufout++) =
                        (unsigned char) (pr2six[*bufin] << 2 | pr2six[bufin[1]] >> 4);
        }
        if (nprbytes > 2) {
                *(bufout++) =
                        (unsigned char) (pr2six[bufin[1]] << 4 | pr2six[bufin[2]] >> 2);
```

```
        }
        if (nprbytes > 3) {
                *(bufout++) =
                        (unsigned char) (pr2six[bufin[2]] << 6 | pr2six[bufin[3]]);
        }

        *(bufout++) = '\0';
        nbytesdecoded -= (4 - nprbytes) & 3;
        return nbytesdecoded;
}

int main()
{
        char input[32];
        uint8_t output[16+1];

        klee_make_symbolic(input, sizeof input, "input");

        klee_assume(input[31]==0);

        klee_assume (Base64decode(output, input)==16);

        for (int i=0; i<16; i++)
                klee_assume (output[i]==i);

        klee_assert(0);

        return 0;
}
```

```
% clang -emit-llvm -c -g klee_base64.c
...

% klee klee_base64.bc
KLEE: output directory is "/home/klee/klee-out-99"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: ERROR: /home/klee/klee_base64.c:99: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_base64.c:104: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_base64.c:85: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_base64.c:81: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_base64.c:65: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location

...
```

We're interesting in the second error, where `klee_assert()` has been triggered:

```
% ls klee-last | grep err
test000001.user.err
test000002.external.err
test000003.ptr.err
test000004.ptr.err
```

```
test000005.ptr.err

% ktest-tool --write-ints klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['klee_base64.bc']
num objects: 1
object    0: name: b'input'
object    0: size: 32
object    0: data: b'AAECAwQFBgcICQoLDA0OD4\x00\xff\xff\xff\xff\xff\xff\xff\xff\x00'
```

This is indeed a real base64 string, terminated with the zero byte, just as it's requested by C/C++ standards. The final zero byte at 31th byte (starting at zeroth byte) is our deed: so that KLEE would report lesser number of errors.

The base64 string is indeed correct:

```
% echo AAECAwQFBgcICQoLDA0OD4 | base64 -d | hexdump -C
base64: invalid input
00000000  00 01 02 03 04 05 06 07  08 09 0a 0b 0c 0d 0e 0f  |................|
00000010
```

base64 decoder Linux utility I've just run blaming for "invalid input"—it means the input string is not properly padded. Now let's pad it manually, and decoder utility will no complain anymore:

```
% echo AAECAwQFBgcICQoLDA0OD4== | base64 -d | hexdump -C
00000000  00 01 02 03 04 05 06 07  08 09 0a 0b 0c 0d 0e 0f  |................|
00000010
```

The reason our generated base64 string is not padded is because base64 decoders are usually discards padding symbols ("=") at the end. In other words, they are not require them, so is the case of our decoder. Hence, padding symbols are left unnoticed to KLEE.

So we again made *antipode* or *inverse function* of base64 decoder.

## 17.6 LZSS decompressor

I've googled for a very simple LZSS decompressor and landed at this page: http://www.opensource.apple.com/source/boot/boot-132/i386/boot2/lzss.c.

Let's pretend, we're looking at unknown compressing algorithm with no compressor available. Will it be possible to reconstruct a compressed piece of data so that decompressor would generate data we need?

Here is my first experiment:

```
// copypasted from http://www.opensource.apple.com/source/boot/boot-132/i386/boot2/lzss.
    c
//
#include <string.h>
#include <stdint.h>
#include <stdbool.h>

//#define N         4096  /* size of ring buffer - must be power of 2 */
#define N         32  /* size of ring buffer - must be power of 2 */
#define F         18    /* upper limit for match_length */
#define THRESHOLD 2     /* encode string into position and length
                           if match_length is greater than this */
#define NIL       N     /* index for root of binary search trees */

int
decompress_lzss(uint8_t *dst, uint8_t *src, uint32_t srclen)
{
    /* ring buffer of size N, with extra F-1 bytes to aid string comparison */
    uint8_t *dststart = dst;
    uint8_t *srcend = src + srclen;
```

```
    int  i, j, k, r, c;
    unsigned int flags;
    uint8_t text_buf[N + F - 1];

    dst = dststart;
    srcend = src + srclen;
    for (i = 0; i < N - F; i++)
        text_buf[i] = ' ';
    r = N - F;
    flags = 0;
    for ( ; ; ) {
        if (((flags >>= 1) & 0x100) == 0) {
            if (src < srcend) c = *src++; else break;
            flags = c | 0xFF00;  /* uses higher byte cleverly */
        }   /* to count eight */
        if (flags & 1) {
            if (src < srcend) c = *src++; else break;
            *dst++ = c;
            text_buf[r++] = c;
            r &= (N - 1);
        } else {
            if (src < srcend) i = *src++; else break;
            if (src < srcend) j = *src++; else break;
            i |= ((j & 0xF0) << 4);
            j  =  (j & 0x0F) + THRESHOLD;
            for (k = 0; k <= j; k++) {
                c = text_buf[(i + k) & (N - 1)];
                *dst++ = c;
                text_buf[r++] = c;
                r &= (N - 1);
            }
        }
    }

    return dst - dststart;
}

int main()
{
#define COMPRESSED_LEN 15
        uint8_t input[COMPRESSED_LEN];
        uint8_t plain[24];
        uint32_t size=COMPRESSED_LEN;

        klee_make_symbolic(input, sizeof input, "input");

        decompress_lzss(plain, input, size);

        // https://en.wikipedia.org/wiki/
           Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffalo_buffalo
        for (int i=0; i<23; i++)
                klee_assume (plain[i]=="Buffalo buffalo Buffalo"[i]);

        klee_assert(0);

        return 0;
```

```
}
```

What I did is changing size of ring buffer from 4096 to 32, because if bigger, KLEE consumes all RAM[125] it can. But I've found that KLEE can live with that small buffer. I've also decreased `COMPRESSED_LEN` gradually to check, whether KLEE would find compressed piece of data, and it did:

```
% clang -emit-llvm -c -g klee_lzss.c
...

% time klee klee_lzss.bc
KLEE: output directory is "/home/klee/klee-out-7"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: ERROR: /home/klee/klee_lzss.c:122: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_lzss.c:47: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_lzss.c:37: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_lzss.c:124: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 41417919
KLEE: done: completed paths = 437820
KLEE: done: generated tests = 4

real    13m0.215s
user    11m57.517s
sys     1m2.187s

% ls klee-last | grep err
test000001.user.err
test000002.ptr.err
test000003.ptr.err
test000004.external.err

% ktest-tool --write-ints klee-last/test000004.ktest
ktest file : 'klee-last/test000004.ktest'
args       : ['klee_lzss.bc']
num objects: 1
object    0: name: b'input'
object    0: size: 15
object    0: data: b'\xffBuffalo \x01b\x0f\x03\r\x05'
```

KLEE consumed $\approx 1GB$ of RAM and worked for $\approx$ 15 minutes (on my Intel Core i3-3110M 2.4GHz notebook), but here it is, a 15 bytes which, if decompressed by our copypasted algorithm, will result in desired text!

During my experimentation, I've found that KLEE can do even more cooler thing, to find out size of compressed piece of data:

```
int main()
{
        uint8_t input[24];
        uint8_t plain[24];
        uint32_t size;

        klee_make_symbolic(input, sizeof input, "input");
```

[125] Random-access memory

```
        klee_make_symbolic(&size, sizeof size, "size");

        decompress_lzss(plain, input, size);

        for (int i=0; i<23; i++)
                klee_assume (plain[i]=="Buffalo buffalo Buffalo"[i]);

        klee_assert(0);

        return 0;
}
```

...but then KLEE works much slower, consumes much more RAM and I had success only with even smaller pieces of desired text.

So how LZSS works? Without peeking into Wikipedia, we can say that: if LZSS compressor observes some data it already had, it replaces the data with a link to some place in past with size. If it observes something yet unseen, it puts data as is. This is theory. This is indeed what we've got. Desired text is three "Buffalo" words, the first and the last are equivalent, but the second is *almost* equivalent, differing with first by one character.

That's what we see:

```
'\xffBuffalo \x01b\x0f\x03\r\x05'
```

Here is some control byte (0xff), "Buffalo" word is placed *as is*, then another control byte (0x01), then we see beginning of the second word ("b") and more control bytes, perhaps, links to the beginning of the buffer. These are command to decompressor, like, in plain English, "copy data from the buffer we've already done, from that place to that place", etc.

Interesting, is it possible to meddle into this piece of compressed data? Out of whim, can we force KLEE to find a compressed data, where not just "b" character has been placed *as is*, but also the second character of the word, i.e., "bu"?

I've modified main() function by adding `klee_assume()`: now the 11th byte of input (compressed) data (right after "b" byte) must have "u". I has no luck with 15 byte of compressed data, so I increased it to 16 bytes:

```
int main()
{
#define COMPRESSED_LEN 16
        uint8_t input[COMPRESSED_LEN];
        uint8_t plain[24];
        uint32_t size=COMPRESSED_LEN;

        klee_make_symbolic(input, sizeof input, "input");

        klee_assume(input[11]=='u');

        decompress_lzss(plain, input, size);

        for (int i=0; i<23; i++)
                klee_assume (plain[i]=="Buffalo buffalo Buffalo"[i]);

        klee_assert(0);

        return 0;
}
```

...and voilà: KLEE found a compressed piece of data which satisfied our whimsical constraint:

```
% time klee klee_lzss.bc
KLEE: output directory is "/home/klee/klee-out-9"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: ERROR: /home/klee/klee_lzss.c:97: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_lzss.c:47: memory error: out of bound pointer
```

```
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_lzss.c:37: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_lzss.c:99: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 36700587
KLEE: done: completed paths = 369756
KLEE: done: generated tests = 4

real    12m16.983s
user    11m17.492s
sys     0m58.358s

% ktest-tool --write-ints klee-last/test000004.ktest
ktest file : 'klee-last/test000004.ktest'
args       : ['klee_lzss.bc']
num objects: 1
object    0: name: b'input'
object    0: size: 16
object    0: data: b'\xffBuffalo \x13bu\x10\x02\r\x05'
```

So now we find a piece of compressed data where two strings are placed *as is*: “Buffalo” and “bu”.

```
'\xffBuffalo \x13bu\x10\x02\r\x05'
```

Both pieces of compressed data, if feeded into our copypasted function, produce “Buffalo buffalo Buffalo” text string.

Please note, I still have no access to LZSS compressor code, and I didn’t get into LZSS decompressor details yet.

Unfortunately, things are not that cool: KLEE is very slow and I had success only with small pieces of text, and also ring buffer size had to be decreased significantly (original LZSS decompressor with ring buffer of 4096 bytes cannot decompress correctly what we found).

Nevertheless, it’s very impressive, taking into account the fact that we’re not getting into internals of this specific LZSS decompressor. Once more time, we’ve created *antipode* of decompressor, or *inverse function*.

Also, as it seems, KLEE isn’t very good so far with decompression algorithms (but who’s good then?). I’ve also tried various JPEG/PNG/GIF decoders (which, of course, has decompressors), starting with simplest possible, and KLEE had stuck.

## 17.7 strtodx() from RetroBSD

Just found this function in RetroBSD: https://github.com/RetroBSD/retrobsd/blob/master/src/libc/stdlib/strtod.c. It converts a string into floating point number for given radix.

```
#include <stdio.h>

// my own version, only for radix 10:
int isdigitx (char c, int radix)
{
        if (c>='0' && c<='9')
                return 1;
        return 0;
};

/*
 * double strtodx (char *string, char **endPtr, int radix)
 *      This procedure converts a floating-point number from an ASCII
 *      decimal representation to internal double-precision format.
 *
```

```
 * Original sources taken from 386bsd and modified for variable radix
 * by Serge Vakulenko, <vak@kiae.su>.
 *
 * Arguments:
 * string
 *      A decimal ASCII floating-point number, optionally preceded
 *      by white space. Must have form "-I.FE-X", where I is the integer
 *      part of the mantissa, F is the fractional part of the mantissa,
 *      and X is the exponent.  Either of the signs may be "+", "-", or
 *      omitted.  Either I or F may be omitted, or both.  The decimal point
 *      isn't necessary unless F is present. The "E" may actually be an "e",
 *      or "E", "S", "s", "F", "f", "D", "d", "L", "l".
 *      E and X may both be omitted (but not just one).
 *
 * endPtr
 *      If non-NULL, store terminating character's address here.
 *
 * radix
 *      Radix of floating point, one of 2, 8, 10, 16.
 *
 * The return value is the double-precision floating-point
 * representation of the characters in string.  If endPtr isn't
 * NULL, then *endPtr is filled in with the address of the
 * next character after the last one that was part of the
 * floating-point number.
 */
double strtodx (char *string, char **endPtr, int radix)
{
        int sign = 0, expSign = 0, fracSz, fracOff, i;
        double fraction, dblExp, *powTab;
        register char *p;
        register char c;

        /* Exponent read from "EX" field. */
        int exp = 0;

        /* Exponent that derives from the fractional part.  Under normal
         * circumstances, it is the negative of the number of digits in F.
         * However, if I is very long, the last digits of I get dropped
         * (otherwise a long I with a large negative exponent could cause an
         * unnecessary overflow on I alone).  In this case, fracExp is
         * incremented one for each dropped digit. */
        int fracExp = 0;

        /* Number of digits in mantissa. */
        int mantSize;

        /* Number of mantissa digits BEFORE decimal point. */
        int decPt;

        /* Temporarily holds location of exponent in string. */
        char *pExp;

        /* Largest possible base 10 exponent.
         * Any exponent larger than this will already
         * produce underflow or overflow, so there's
```

```
	 * no need to worry about additional digits. */
	static int maxExponent = 307;

	/* Table giving binary powers of 10.
	 * Entry is 10^2^i.  Used to convert decimal
	 * exponents into floating-point numbers. */
	static double powersOf10[] = {
		1e1, 1e2, 1e4, 1e8, 1e16, 1e32, //1e64, 1e128, 1e256,
	};
	static double powersOf2[] = {
		2, 4, 16, 256, 65536, 4.294967296e9, 1.8446744073709551616e19,
		//3.4028236692093846346e38, 1.1579208923731619542e77, 1.3407807929942597099e154,
	};
	static double powersOf8[] = {
		8, 64, 4096, 2.81474976710656e14, 7.9228162514264337593e28,
		//6.2771017353866807638e57, 3.9402006196394479212e115, 1.5525180923007089351e231,
	};
	static double powersOf16[] = {
		16, 256, 65536, 1.8446744073709551616e19,
		//3.4028236692093846346e38, 1.1579208923731619542e77, 1.3407807929942597099e154,
	};

	/*
	 * Strip off leading blanks and check for a sign.
	 */
	p = string;
	while (*p==' ' || *p=='\t')
		++p;
	if (*p == '-') {
		sign = 1;
		++p;
	} else if (*p == '+')
		++p;

	/*
	 * Count the number of digits in the mantissa (including the decimal
	 * point), and also locate the decimal point.
	 */
	decPt = -1;
	for (mantSize=0; ; ++mantSize) {
		c = *p;
		if (!isdigitx (c, radix)) {
			if (c != '.' || decPt >= 0)
				break;
			decPt = mantSize;
		}
		++p;
	}

	/*
	 * Now suck up the digits in the mantissa.  Use two integers to
	 * collect 9 digits each (this is faster than using floating-point).
	 * If the mantissa has more than 18 digits, ignore the extras, since
```

```
	 * they can't affect the value anyway.
	 */
	pExp = p;
	p -= mantSize;
	if (decPt < 0)
		decPt = mantSize;
	else
		--mantSize;		/* One of the digits was the point. */

	switch (radix) {
	default:
	case 10: fracSz = 9;  fracOff = 1000000000; powTab = powersOf10; break;
	case 2:  fracSz = 30; fracOff = 1073741824; powTab = powersOf2;  break;
	case 8:  fracSz = 10; fracOff = 1073741824; powTab = powersOf8;  break;
	case 16: fracSz = 7;  fracOff = 268435456;  powTab = powersOf16; break;
	}
	if (mantSize > 2 * fracSz)
		mantSize = 2 * fracSz;
	fracExp = decPt - mantSize;
	if (mantSize == 0) {
		fraction = 0.0;
		p = string;
		goto done;
	} else {
		int frac1, frac2;

		for (frac1=0; mantSize>fracSz; --mantSize) {
			c = *p++;
			if (c == '.')
				c = *p++;
			frac1 = frac1 * radix + (c - '0');
		}
		for (frac2=0; mantSize>0; --mantSize) {
			c = *p++;
			if (c == '.')
				c = *p++;
			frac2 = frac2 * radix + (c - '0');
		}
		fraction = (double) fracOff * frac1 + frac2;
	}

	/*
	 * Skim off the exponent.
	 */
	p = pExp;
	if (*p=='E' || *p=='e' || *p=='S' || *p=='s' || *p=='F' || *p=='f' ||
	    *p=='D' || *p=='d' || *p=='L' || *p=='l') {
		++p;
		if (*p == '-') {
			expSign = 1;
			++p;
		} else if (*p == '+')
			++p;
		while (isdigitx (*p, radix))
			exp = exp * radix + (*p++ - '0');
	}
```

```
        if (expSign)
                exp = fracExp - exp;
        else
                exp = fracExp + exp;

        /*
         * Generate a floating-point number that represents the exponent.
         * Do this by processing the exponent one bit at a time to combine
         * many powers of 2 of 10. Then combine the exponent with the
         * fraction.
         */
        if (exp < 0) {
                expSign = 1;
                exp = -exp;
        } else
                expSign = 0;
        if (exp > maxExponent)
                exp = maxExponent;
        dblExp = 1.0;
        for (i=0; exp; exp>>=1, ++i)
                if (exp & 01)
                        dblExp *= powTab[i];
        if (expSign)
                fraction /= dblExp;
        else
                fraction *= dblExp;

done:
        if (endPtr)
                *endPtr = p;

        return sign ? -fraction : fraction;
}

#define BUFSIZE 10
int main()
{
        char buf[BUFSIZE];
        klee_make_symbolic (buf, sizeof buf, "buf");
        klee_assume(buf[9]==0);

        strtodx (buf, NULL, 10);
};
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/KLEE/strtodx.c )

Interestinly, KLEE cannot handle floating-point arithmetic, but nevertheless, found something:

```
...

KLEE: ERROR: /home/klee/klee_test.c:202: memory error: out of bound pointer

...

% ktest-tool klee-last/test003483.ktest
ktest file : 'klee-last/test003483.ktest'
args       : ['klee_test.bc']
num objects: 1
```

```
object    0: name: b'buf'
object    0: size: 10
object    0: data: b'-.0E-66\x00\x00\x00'
```

As it seems, string "-.0E-66" makes out of bounds array access (read) at line 202. While further investigation, I've found that `powersOf10[]` array is too short: 6th element (started at 0th) has been accessed. And here we see part of array commented (line 79)! Probably someone's mistake?

## 17.8 Unit testing: simple expression evaluator (calculator)

I has been looking for simple expression evaluator (calculator in other words) which takes expression like "2+2" on input and gives answer. I've found one at http://stackoverflow.com/a/13895198. Unfortunately, it has no bugs, so I've introduced one: a token buffer (`buf[]` at line 31) is smaller than input buffer (`input[]` at line 19).

```
1  // copypasted from http://stackoverflow.com/a/13895198 and reworked
2  
3  // Bare bones scanner and parser for the following LL(1) grammar:
4  // expr -> term { [+-] term }     ; An expression is terms separated by add ops.
5  // term -> factor { [*/] factor } ; A term is factors separated by mul ops.
6  // factor -> unsigned_factor      ; A signed factor is a factor,
7  //         | - unsigned_factor    ;   possibly with leading minus sign
8  // unsigned_factor -> ( expr )    ; An unsigned factor is a parenthesized expression
9  //         | NUMBER               ;   or a number
10 //
11 // The parser returns the floating point value of the expression.
12 
13 #include <string.h>
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <stdint.h>
17 #include <stdbool.h>
18 
19 char input[128];
20 int input_idx=0;
21 
22 char my_getchar()
23 {
24         char rt=input[input_idx];
25         input_idx++;
26         return rt;
27 };
28 
29 // The token buffer. We never check for overflow! Do so in production code.
30 // it's deliberately smaller than input[] so KLEE could find buffer overflow
31 char buf[64];
32 int n = 0;
33 
34 // The current character.
35 int ch;
36 
37 // The look-ahead token.  This is the 1 in LL(1).
38 enum { ADD_OP, MUL_OP, LEFT_PAREN, RIGHT_PAREN, NOT_OP, NUMBER, END_INPUT } look_ahead;
39 
40 // Forward declarations.
41 void init(void);
42 void advance(void);
43 int expr(void);
```

```
void error(char *msg);

// Parse expressions, one per line.
int main(void)
{
	// take input expression from input[]
	//input[0]=0;
	//strcpy (input, "2+2");
	klee_make_symbolic(input, sizeof input, "input");
	input[127]=0;

	init();
	while (1)
	{
		int val = expr();
		printf("%d\n", val);

		if (look_ahead != END_INPUT)
			error("junk after expression");
		advance();  // past end of input mark
	}
	return 0;
}

// Just die on any error.
void error(char *msg)
{
	fprintf(stderr, "Error: %s. Exiting.\n", msg);
	exit(1);
}

// Buffer the current character and read a new one.
void read()
{
	buf[n++] = ch;
	buf[n] = '\0';  // Terminate the string.
	ch = my_getchar();
}

// Ignore the current character.
void ignore()
{
	ch = my_getchar();
}

// Reset the token buffer.
void reset()
{
	n = 0;
	buf[0] = '\0';
}

// The scanner.  A tiny deterministic finite automaton.
int scan()
{
	reset();
```

```
START:
        // first character is digit?
        if (isdigit (ch))
                goto DIGITS;

        switch (ch)
        {
                case ' ': case '\t': case '\r':
                        ignore();
                        goto START;

                case '-': case '+': case '^':
                        read();
                        return ADD_OP;

                case '~':
                        read();
                        return NOT_OP;

                case '*': case '/': case '%':
                        read();
                        return MUL_OP;

                case '(':
                        read();
                        return LEFT_PAREN;

                case ')':
                        read();
                        return RIGHT_PAREN;

                case 0:
                case '\n':
                        ch = ' ';    // delayed ignore()
                        return END_INPUT;

                default:
                        printf ("bad character: 0x%x\n", ch);
                        exit(0);
        }

DIGITS:
        if (isdigit (ch))
        {
                read();
                goto DIGITS;
        }
        else
                return NUMBER;
}

// To advance is just to replace the look-ahead.
void advance()
{
        look_ahead = scan();
}
```

```

// Clear the token buffer and read the first look-ahead.
void init()
{
        reset();
        ignore(); // junk current character
        advance();
}

int get_number(char *buf)
{
        char *endptr;

        int rt=strtoul (buf, &endptr, 10);

        // is the whole buffer has been processed?
        if (strlen(buf)!=endptr-buf)
        {
                fprintf (stderr, "invalid number: %s\n", buf);
                exit(0);
        };
        return rt;
};

int unsigned_factor()
{
        int rtn = 0;
        switch (look_ahead)
        {
                case NUMBER:
                        rtn=get_number(buf);
                        advance();
                        break;

                case LEFT_PAREN:
                        advance();
                        rtn = expr();
                        if (look_ahead != RIGHT_PAREN) error("missing ')'");
                        advance();
                        break;

                default:
                        printf("unexpected token: %d\n", look_ahead);
                        exit(0);
        }
        return rtn;
}

int factor()
{
        int rtn = 0;
        // If there is a leading minus...
        if (look_ahead == ADD_OP && buf[0] == '-')
        {
                advance();
                rtn = -unsigned_factor();
```

```
	}
	else
		rtn = unsigned_factor();

	return rtn;
}

int term()
{
	int rtn = factor();
	while (look_ahead == MUL_OP)
	{
		switch(buf[0])
		{
			case '*':
				advance();
				rtn *= factor();
				break;

			case '/':
				advance();
				rtn /= factor();
				break;
			case '%':
				advance();
				rtn %= factor();
				break;
		}
	}
	return rtn;
}

int expr()
{
	int rtn = term();
	while (look_ahead == ADD_OP)
	{
		switch(buf[0])
		{
			case '+':
				advance();
				rtn += term();
				break;

			case '-':
				advance();
				rtn -= term();
				break;
		}
	}
	return rtn;
}
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/KLEE/calc.c )

KLEE found buffer overflow with little effort (65 zero digits + one tabulation symbol):

```
% ktest-tool --write-ints klee-last/test000468.ktest
```

```
ktest file : 'klee-last/test000468.ktest'
args       : ['calc.bc']
num objects: 1
object    0: name: b'input'
object    0: size: 128
object    0: data: b'0\t0000000000000000000000000000000000000000000000000000000000000000
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff'
```

Hard to say, how tabulation symbol (`\t`) got into input[] array, but KLEE achieved what has been desired: buffer overflown.

KLEE also found two expression strings which leads to division error ("0/0" and "0%0"):

```
% ktest-tool --write-ints klee-last/test000326.ktest
ktest file : 'klee-last/test000326.ktest'
args       : ['calc.bc']
num objects: 1
object    0: name: b'input'
object    0: size: 128
object    0: data: b'0/0\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff'

% ktest-tool --write-ints klee-last/test000557.ktest
ktest file : 'klee-last/test000557.ktest'
args       : ['calc.bc']
num objects: 1
object    0: name: b'input'
object    0: size: 128
object    0: data: b'0%0\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
   \xff\xff\xff\xff'
```

Maybe this is not impressive result, nevertheless, it's yet another reminder that division and remainder operations must be wrapped somehow in production code to avoid possible crash.

## 17.9 More examples

https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/

## 17.10 Exercise

Here is my crackme/keygenme, which may be tricky, but easy to solve using KLEE: http://challenges.re/74/.

# 18 (Amateur) cryptography

## 18.1 *Serious* cryptography

Let's back to the method we previously used (16.2) to construct expressions using running Python function.

We can try to build expression for the output of XXTEA encryption algorithm:

```
#!/usr/bin/env python

class Expr:
    def __init__(self,s):
        self.s=s

    def __str__(self):
        return self.s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __xor__(self, other):
        return Expr("(" + self.s + "^" + self.convert_to_Expr_if_int(other).s + ")")

    def __mul__(self, other):
        return Expr("(" + self.s + "*" + self.convert_to_Expr_if_int(other).s + ")")

    def __add__(self, other):
        return Expr("(" + self.s + "+" + self.convert_to_Expr_if_int(other).s + ")")

    def __and__(self, other):
        return Expr("(" + self.s + "&" + self.convert_to_Expr_if_int(other).s + ")")

    def __lshift__(self, other):
        return Expr("(" + self.s + "<<" + self.convert_to_Expr_if_int(other).s + ")")

    def __rshift__(self, other):
        return Expr("(" + self.s + ">>" + self.convert_to_Expr_if_int(other).s + ")")

    def __getitem__(self, d):
        return Expr("(" + self.s + "[" + d.s + "])")

# reworked from:

# Pure Python (2.x) implementation of the XXTEA cipher
# (c) 2009. Ivan Voras <ivoras@gmail.com>
# Released under the BSD License.

def raw_xxtea(v, n, k):

    def MX():
        return ((z>>5)^(y<<2)) + ((y>>3)^(z<<4))^(sum^y) + (k[(Expr(str(p)) & 3)^e]^z)

    y = v[0]
    sum = Expr("0")
```

```
    DELTA = 0x9e3779b9
    # Encoding only
    z = v[n-1]

    # number of rounds:
    #q = 6 + 52 / n
    q=1

    while q > 0:
        q -= 1
        sum = sum + DELTA
        e = (sum >> 2) & 3
        p = 0
        while p < n - 1:
            y = v[p+1]
            z = v[p] = v[p] + MX()
            p += 1
        y = v[0]
        z = v[n-1] = v[n-1] + MX()
    return 0

v=[Expr("input1"), Expr("input2"), Expr("input3"), Expr("input4")]
k=Expr("key")

raw_xxtea(v, 4, k)

for i in range(4):
    print i, ":", v[i]
#print len(str(v[0]))+len(str(v[1]))+len(str(v[2]))+len(str(v[3]))
```

A key is choosen according to input data, and, obviously, we can't know it during symbolic execution, so we leave expression like `k[...]`.

Now results for just one round, for each of 4 outputs:

```
0 : (input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^
    input2)+
((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))

1 : (input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^
input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3<<2))+((input3>>3)
    ^((input1+
((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key
    [((0&3)^
(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((key[((1&3)
    ^(((0+2654435769)>>2)&
3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^
    input2)+
((key[((0&3)^(((0+2654435769)>>2)&3))])^input4)))))))

2 : (input3+(((((input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)
    ))^
(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3
    <<2))+
((input3>>3)^((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^
input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3
```

```
   )+
((key[((1&3)^(((0+2654435769)>>2)&3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)
   ^(input4<<4)))
^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))))))>>5)^(
   input4<<2))+
((input4>>3)^((input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
   ^(((0+2654435769)^
input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3<<2))+((input3>>3)
   ^((input1+
((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key
   [((0&3)^
(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((key[((1&3)
   ^(((0+2654435769)>>2)&3))])^
(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)
   +((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))))))<<4)))^(((0+2654435769)^input4)+((key[((2&3)
   ^(((0+2654435769)>>2)&
3))])^(input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
   ^(((0+2654435769)^
input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3<<2))+((input3>>3)
   ^((input1+
((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key
   [((0&3)^
(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((key[((1&3)
   ^(((0+2654435769)>>2)&
3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^
   input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4)))))))))))

3 : (input4+(((((input3+(((((input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)
   ^(input4<<4)))^
(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3
   <<2))+((input3>>3)^
((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)
   +((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((key[((1&3)
   ^(((0+2654435769)>>2)&3))])^
(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)
   +((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))))))>>5)^(input4<<2))+((input4>>3)^((input2+(((((
   input1+((((input4>>5)^
(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)
   ^(((0+2654435769)>>2)&3))])^
input4))))>>5)^(input3<<2))+((input3>>3)^((input1+((((input4>>5)^(input2<<2))+((input2
   >>3)^(input4<<4)))^
(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<4)))
   ^(((0+2654435769)^input3)+
((key[((1&3)^(((0+2654435769)>>2)&3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)
   ^(input4<<4)))^
(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))))))<<4)))
   ^(((0+2654435769)^
input4)+((key[((2&3)^(((0+2654435769)>>2)&3))])^(input2+(((((input1+((((input4>>5)^(
   input2<<2))+((input2>>3)^
(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4)))
   )>>5)^(input3<<2))+
((input3>>3)^((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
```

```
    ^(((0+2654435769)^input2)+
((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((key
    [((1&3)^(((0+2654435769)>>
2)&3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)
    ^input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))))))))))>>5)^((input1+((((input4>>5)^(input2<<2))
    +((input2>>3)^(input4<<
4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<2))
    +(((input1+((((input4>>5)^
(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)
    ^(((0+2654435769)>>2)&3))])^
input4))))>>3)^((input3+(((((input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)
    ^(input4<<4)))^(((0+
2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3<<2))
    +((input3>>3)^((input1+
((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key
    [((0&3)^(((0+2654435769)>>
2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((key[((1&3)^(((0+2654435769)>>2)&3))
    ])^(input1+((((input4>>
5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)
    ^(((0+2654435769)>>2)&3))])^
input4))))))))>>5)^(input4<<2))+((input4>>3)^((input2+(((((input1+((((input4>>5)^(input2
    <<2))+((input2>>3)^
(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4)))
    )>>5)^(input3<<2))+
((input3>>3)^((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+
((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((key
    [((1&3)^(((0+2654435769)>>
2)&3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)
    ^input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))))))<<4)))^(((0+2654435769)^input4)+((key[((2&3)
    ^(((0+2654435769)>>2)&3))])^
(input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3<<2))+((input3>>3)^((input1+((((input4
    >>5)^(input2<<2))+
((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3)
    )])^input4))))<<4)))^(((
0+2654435769)^input3)+((key[((1&3)^(((0+2654435769)>>2)&3))])^(input1+((((input4>>5)^(
    input2<<2))+((input2>>3)^(
input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))
    ))))))))<<4)))^(((0+
2654435769)^(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4)))))+((key[((3&3)^(((0+2654435769)>>2)&3))])^(input3
    +(((((input2+(((((input1+
((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key
    [((0&3)^(((0+2654435769)>>
2)&3))])^input4))))>>5)^(input3<<2))+((input3>>3)^((input1+((((input4>>5)^(input2<<2))
    +((input2>>3)^(input4<<
4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<4)))
    ^(((0+2654435769)^input3)+
((key[((1&3)^(((0+2654435769)>>2)&3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)
    ^(input4<<4)))^(((0+
2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))))))>>5)^(input4
```

```
    <<2))+((input4>>3)^((
input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3<<2))+((input3>>3)^((input1+((((input4
    >>5)^(input2<<2))+((
input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))
    ])^input4))))<<4)))^(((0+
2654435769)^input3)+((key[((1&3)^(((0+2654435769)>>2)&3))])^(input1+((((input4>>5)^(
    input2<<2))+((input2>>3)^
(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4)))
    )))))<<4)))^(((0+
2654435769)^input4)+((key[((2&3)^(((0+2654435769)>>2)&3))])^(input2+(((((input1+((((
    input4>>5)^(input2<<2))+
((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3)
    )])^input4))))>>5)^
(input3<<2))+((input3>>3)^((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4))
    )^(((0+2654435769)^
input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3
    )+((key[((1&3)^(((0+
2654435769)>>2)&3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+
((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))))))))))))
```

Somehow, size of expression for each subsequent output is bigger. I hope I haven't been mistaken? And this is just for 1 round. For 2 rounds, size of all 4 expression is $\approx 970KB$. For 3 rounds, this is $\approx 115MB$. For 4 rounds, I have not enough RAM on my computer. Expressions *exploding* exponentially. And there are 19 rounds. You can weigh it.

Perhaps, you can simplify these expressions: there are a lot of excessive parenthesis, but I'm highly pessimistic, cryptoalgorithms constructed in such a way to not have any spare operations.

In order to crack it, you can use these expressions as system of equation and try to solve it using SMT-solver. This is called "algebraic attack".

In other words, theoretically, you can build a system of equation like this: $MD5(x) = 12341234...$, but expressions are so huge so it's impossible to solve them. Yes, cryptographers are fully aware of this and one of the goals of the successful cipher is to make expressions as big as possible, using resonable time and size of algorithm.

Nevertheless, you can find numerous papers about breaking these cryptosystems with reduced number of rounds: when expression isn't *exploded* yet, sometimes it's possible. This cannot be applied in practice, but such an experience has some interesting theoretical uses.

### 18.1.1 Attempts to break "serious" crypto

CryptoMiniSat itself exist to support XOR operation, which is ubiquitous in cryptography.

- Bitcoin mining with SAT solver: `http://jheusser.github.io/2013/02/03/satcoin.html`, `https://github.com/msoos/sha256-sat-bitcoin`.
- Alexander Semenov, attempts to break A5/1, etc. (Russian presentation)
- Vegard Nossum - SAT-based preimage attacks on SHA-1
- Algebraic Attacks on the Crypto-1 Stream Cipher in MiFare Classic and Oyster Cards
- Attacking Bivium Using SAT Solvers
- Extending SAT Solvers to Cryptographic Problems
- Applications of SAT Solvers to Cryptanalysis of Hash Functions
- Algebraic-Differential Cryptanalysis of DES

## 18.2 Amateur cryptography

This is what you can find in serial numbers, license keys, executable file packers, CTF[126], malware, etc. Sometimes even ransomware (but rarely nowadays, in 2017).

Amateur cryptography is often can be broken using SMT solver, or even KLEE.

Amateur cryptography is usually based not on theory, but on visual complexity: if its creator getting results which are seems chaotic enough, often, one stops to improve it further. This is security based not even on obscurity, but on a chaotic mess. This is also sometimes called "The Fallacy of Complex Manipulation" (see RFC4086).

Devising your own cryptoalgorithm is a very tricky thing to do. This can be compared to devising your own PRNG. Even famous Donald Knuth in 1959 constructed one, and it was visually very complex, but, as it turns out in practice, it has very short cycle of length 3178. [See also: The Art of Computer Programming vol.II page 4, (1997).]

The very first problem is that making an algorithm which can generate very long expressions is tricky thing itself. Common error is to use operations like XOR and rotations/permutations, which can't help much. Even worse: some people think that XORing a value several times can be better, like: $(x \oplus 1234) \oplus 5678$. Obviously, these two XOR operations (or more precisely, any number of it) can be reduced to a single one. Same story about applied operations like addition and subtraction—they all also can be reduced to single one.

Real cryptoalgorithms, like IDEA, can use several operations from different groups, like XOR, addition and multiplication. Applying them all in specific order will make resulting expression irreducible.

When I prepared this article, I tried to make an example of such amateur hash function:

```
// copypasted from http://blog.regehr.org/archives/1063
uint32_t rotl32b (uint32_t x, uint32_t n)
{
		assert (n<32);
			if (!n) return x;
				return (x<<n) | (x>>(32-n));
}

uint32_t rotr32b (uint32_t x, uint32_t n)
{
		assert (n<32);
			if (!n) return x;
				return (x>>n) | (x<<(32-n));
}

void megahash (uint32_t buf[4])
{
	for (int i=0; i<4; i++)
	{
		uint32_t t0=buf[0]^0x12345678^buf[1];
		uint32_t t1=buf[1]^0xabcdef01^buf[2];
		uint32_t t2=buf[2]^0x23456789^buf[3];
		uint32_t t3=buf[3]^0x0abcdef0^buf[0];

		buf[0]=rotl32b(t0, 1);
		buf[1]=rotr32b(t1, 2);
		buf[2]=rotl32b(t2, 3);
		buf[3]=rotr32b(t3, 4);
	};
};

int main()
{
	uint32_t buf[4];
	klee_make_symbolic(buf, sizeof buf);
	megahash (buf);
```

[126]Capture the Flag

```
        if (buf[0]==0x18f71ce6          // or whatever
                && buf[1]==0xf37c2fc9
                && buf[2]==0x1cfe96fe
                && buf[3]==0x8c02c75e)
                klee_assert(0);
};
```

KLEE can break it with little effort. Functions of such complexity is common in shareware, which checks license keys, etc.

Here is how we can make its work harder by making rotations dependent of inputs, and this makes number of possible inputs much, much bigger:

```
void megahash (uint32_t buf[4])
{
        for (int i=0; i<16; i++)
        {
                uint32_t t0=buf[0]^0x12345678^buf[1];
                uint32_t t1=buf[1]^0xabcdef01^buf[2];
                uint32_t t2=buf[2]^0x23456789^buf[3];
                uint32_t t3=buf[3]^0x0abcdef0^buf[0];

                buf[0]=rotl32b(t0, t1&0x1F);
                buf[1]=rotr32b(t1, t2&0x1F);
                buf[2]=rotl32b(t2, t3&0x1F);
                buf[3]=rotr32b(t3, t0&0x1F);
        };
};
```

Addition (or modular addition, as cryptographers say) can make thing even harder:

```
void megahash (uint32_t buf[4])
{
        for (int i=0; i<4; i++)
        {
                uint32_t t0=buf[0]^0x12345678^buf[1];
                uint32_t t1=buf[1]^0xabcdef01^buf[2];
                uint32_t t2=buf[2]^0x23456789^buf[3];
                uint32_t t3=buf[3]^0x0abcdef0^buf[0];

                buf[0]=rotl32b(t0, t2&0x1F)+t1;
                buf[1]=rotr32b(t1, t3&0x1F)+t2;
                buf[2]=rotl32b(t2, t1&0x1F)+t3;
                buf[3]=rotr32b(t3, t2&0x1F)+t4;
        };
};
```

As en exercise, you can try to make a block cipher which KLEE wouldn't break. This is quite sobering experience. But even if you can, this is not a panacea, there is an array of other cryptoanalytical methods to break it.

Summary: if you deal with amateur cryptography, you can always give KLEE and SMT solver a try. Even more: sometimes you have only decryption function, and if algorithm is simple enough, KLEE or SMT solver can reverse things back.

One funny thing to mention: if you try to implement amateur cryptoalgorithm in Verilog/VHDL language to run it on FPGA[127], maybe in brute-force way, you can find that EDA[128] tools can optimize many things during synthesis (this is the word they use for "compilation") and can leave cryptoalgorithm much smaller/simpler than it was. Even if you try to implement DES[129] algorithm *in bare metal* with a fixed key, Altera Quartus can optimize first round of it and make it smaller than others.

[127]Field-programmable gate array
[128]Electronic design automation
[129]Data Encryption Standard

### 18.2.1 Bugs

Another prominent feature of amateur cryptography is bugs. Bugs here often left uncaught because output of encrypting function visually looked "good enough" or "obfuscated enough", so a developer stopped to work on it.

This is especially feature of hash functions, because when you work on block cipher, you have to do two functions (encryption/decryption), while hashing function is single.

Weirdest ever amateur encryption algorithm I once saw, encrypted only odd bytes of input block, while even bytes left untouched, so the input plain text has been partially seen in the resulting encrypted block. It was encryption routine used in license key validation. Hard to believe someone did this on purpose. Most likely, it was just an unnoticed bug.

### 18.2.2 XOR ciphers

Simplest possible amateur cryptography is just application of XOR operation using some kind of table. Maybe even simpler. This is a real algorithm I once saw:

```
for (i=0; i<size; i++)
    buf[i]=buf[i]^(31*(i+1));
```

This is not even encryption, rather concealing or hiding.

Some other examples of simple XOR-cipher cryptoanalysis are present in the "Reverse Engineering for Beginners" [130] book.

### 18.2.3 Other features

**Tables** There are often table(s) with pseudorandom data, which is/are used chaotically.

**Checksumming** End-users can have proclivity of changing license codes, serial numbers, etc., with a hope this could affect behaviour of software. So there is often some kind of checksum: starting at simple summing and CRC. This is close to MAC[131] in real cryptography.

**Entropy level** Maybe (much) lower, despite the fact that data looks random.

### 18.2.4 Examples

- A popular FLEXlm license manager was based on a simple amateur cryptoalgorithm (before they switched to ECC[132]), which can be cracked easily.
- Pegasus Mail Password Decoder: `http://phrack.org/issues/52/3.html` - a very typical example.
- You can find a lot of blog posts about breaking CTF-level crypto using Z3, etc. Here is one of them: `http://doar-e.github.io/blog/2015/08/18/keygenning-with-klee/`.
- Another: Automated algebraic cryptanalysis with OpenREIL and Z3. By the way, this solution tracks state of each register at each EIP/RIP, this is almost the same as SSA, which is heavily used in compiers and worth learning.
- Many examples of amateur cryptography I've taken from an old Fravia website: `https://yurichev.com/mirrors/amateur_crypto_examples_from_Fravia/`.

## 18.3 Case study: simple hash function

(This piece of text was initially added to my "Reverse Engineering for Beginners" book (`beginners.re`) at March 2014) [133].

Here is one-way hash function, that converted a 64-bit value to another and we need to try to reverse its flow back.

[130] `http://beginners.re`

[131] Message authentication code

[132] Elliptic curve cryptography

[133] This example was also used by Murphy Berzish in his lecture about SAT and SMT: `http://mirror.csclub.uwaterloo.ca/csclub/mtrberzi-sat-smt-slides.pdf`, `http://mirror.csclub.uwaterloo.ca/csclub/mtrberzi-sat-smt.mp4`

### 18.3.1 Manual decompiling

Here its assembly language listing in IDA:

```
sub_401510      proc near
                ; ECX = input
                mov     rdx, 5D7E0D1F2E0F1F84h
                mov     rax, rcx        ; input
                imul    rax, rdx
                mov     rdx, 388D76AEE8CB1500h
                mov     ecx, eax
                and     ecx, 0Fh
                ror     rax, cl
                xor     rax, rdx
                mov     rdx, 0D2E9EE7E83C4285Bh
                mov     ecx, eax
                and     ecx, 0Fh
                rol     rax, cl
                lea     r8, [rax+rdx]
                mov     rdx, 8888888888888889h
                mov     rax, r8
                mul     rdx
                shr     rdx, 5
                mov     rax, rdx
                lea     rcx, [r8+rdx*4]
                shl     rax, 6
                sub     rcx, rax
                mov     rax, r8
                rol     rax, cl
                ; EAX = output
                retn
sub_401510      endp
```

The example was compiled by GCC, so the first argument is passed in ECX.

If you don't have Hex-Rays, or if you distrust to it, you can try to reverse this code manually. One method is to represent the CPU registers as local C variables and replace each instruction by a one-line equivalent expression, like:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;
        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        rdx=rdx>>5;
        rax=rdx;
        rcx=r8+rdx*4;
```

```
        rax=rax<<6;
        rcx=rcx-rax;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

If you are careful enough, this code can be compiled and will even work in the same way as the original.

Then, we are going to rewrite it gradually, keeping in mind all registers usage. Attention and focus is very important here—any tiny typo may ruin all your work!

Here is the first step:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        // RDX here is a high part of multiplication result
        rdx=rdx>>5;
        // RDX here is division result!
        rax=rdx;

        rcx=r8+rdx*4;
        rax=rax<<6;
        rcx=rcx-rax;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

Next step:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
```

```
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        // RDX here is a high part of multiplication result
        rdx=rdx>>5;
        // RDX here is division result!
        rax=rdx;

        rcx=(r8+rdx*4)-(rax<<6);
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

We can spot the division using multiplication. Indeed, let's calculate the divider in Wolfram Mathematica:

Listing 3: Wolfram Mathematica

```
In[1]:=N[2^(64 + 5)/16^^8888888888888889]
Out[1]:=60.
```

We get this:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rax=rdx=r8/60;

        rcx=(r8+rax*4)-(rax*64);
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

One more step:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        rax=input;
        rax*=0x5D7E0D1F2E0F1F84;
```

```
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=0x388D76AEE8CB1500;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+0xD2E9EE7E83C4285B;

        rcx=r8-(r8/60)*60;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

By simple reducing, we finally see that it's calculating the remainder, not the quotient:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        rax=input;
        rax*=0x5D7E0D1F2E0F1F84;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=0x388D76AEE8CB1500;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+0xD2E9EE7E83C4285B;

        return _lrotl (r8, r8 % 60); // rotate left
};
```

We end up with this fancy formatted source-code:

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <intrin.h>

#define C1 0x5D7E0D1F2E0F1F84
#define C2 0x388D76AEE8CB1500
#define C3 0xD2E9EE7E83C4285B

uint64_t hash(uint64_t v)
{
        v*=C1;
        v=_lrotr(v, v&0xF); // rotate right
        v^=C2;
        v=_lrotl(v, v&0xF); // rotate left
        v+=C3;
        v=_lrotl(v, v % 60); // rotate left
        return v;
};

int main()
{
        printf ("%llu\n", hash(...));
};
```

Since we are not cryptoanalysts we can't find an easy way to generate the input value for some specific output value. The rotate instruction's coefficients look frightening—it's a warranty that the function is not bijective, it is rather surjective, it has collisions, or, speaking more simply, many inputs may be possible for one output.

Brute-force is not solution because values are 64-bit ones, that's beyond reality.

### 18.3.2 Now let's use the Z3

Still, without any special cryptographic knowledge, we may try to break this algorithm using Z3.

Here is the Python source code:

```
1 #!/usr/bin/env python
2
3 from z3 import *
4
5 C1=0x5D7E0D1F2E0F1F84
6 C2=0x388D76AEE8CB1500
7 C3=0xD2E9EE7E83C4285B
8
9 inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
10
11 s = Solver()
12 s.add(i1==inp*C1)
13 s.add(i2==RotateRight (i1, i1 & 0xF))
14 s.add(i3==i2 ^ C2)
15 s.add(i4==RotateLeft(i3, i3 & 0xF))
16 s.add(i5==i4 + C3)
17 s.add(outp==RotateLeft (i5, URem(i5, 60)))
18
19 s.add(outp==10816636949158156260)
20
21 print s.check()
22 m=s.model()
23 print m
24 print (" inp=0x%X" % m[inp].as_long())
25 print ("outp=0x%X" % m[outp].as_long())
```

This is going to be our first solver.

We see the variable definitions on line 7. These are just 64-bit variables. `i1..i6` are intermediate variables, representing the values in the registers between instruction executions.

Then we add the so-called constraints on lines 10..15. The last constraint at 17 is the most important one: we are going to try to find an input value for which our algorithm will produce 10816636949158156260.

*RotateRight, RotateLeft, URem*—are functions from the Z3 Python API, not related to Python language.

Then we run it:

```
...>python.exe 1.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 1364123924608584563,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
 inp=0x12EE577B63E80B73
outp=0x961C69FF0AEFD7E4
```

"sat" mean "satisfiable", i.e., the solver was able to find at least one solution. The solution is printed in the square brackets. The last two lines are the input/output pair in hexadecimal form. Yes, indeed, if we run our function with `0x12EE577B63E80B73` as input, the algorithm will produce the value we were looking for.

But, as we noticed before, the function we work with is not bijective, so there may be other correct input values. The Z3 is not capable of producing more than one result, but let's hack our example slightly, by adding line 19, which implies "look for any other results than this":

```
1 #!/usr/bin/env python
```

```

from z3 import *

C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))

s.add(outp==10816636949158156260)

s.add(inp!=0x12EE577B63E80B73)

print s.check()
m=s.model()
print m
print (" inp=0x%X" % m[inp].as_long())
print ("outp=0x%X" % m[outp].as_long())
```

Indeed, it finds another correct result:

```
...>python.exe 2.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 10587495961463360371,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
 inp=0x92EE577B63E80B73
outp=0x961C69FF0AEFD7E4
```

This can be automated. Each found result can be added as a constraint and then the next result will be searched for. Here is a slightly more sophisticated example:

```
#!/usr/bin/env python

from z3 import *

C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
```

```
15 s.add(i4==RotateLeft(i3, i3 & 0xF))
16 s.add(i5==i4 + C3)
17 s.add(outp==RotateLeft (i5, URem(i5, 60)))
18
19 s.add(outp==10816636949158156260)
20
21 # copypasted from http://stackoverflow.com/questions/11867611/z3py-checking-all-
       solutions-for-equation
22 result=[]
23 while True:
24     if s.check() == sat:
25         m = s.model()
26         print m[inp]
27         result.append(m)
28         # Create a new constraint the blocks the current model
29         block = []
30         for d in m:
31             # d is a declaration
32             if d.arity() > 0:
33                 raise Z3Exception("uninterpreted functions are not supported")
34             # create a constant from declaration
35             c=d()
36             if is_array(c) or c.sort().kind() == Z3_UNINTERPRETED_SORT:
37                 raise Z3Exception("arrays and uninterpreted sorts are not supported")
38             block.append(c != m[d])
39         s.add(Or(block))
40     else:
41             print "results total=",len(result)
42             break
```

We got:

```
1364123924608584563
1234567890
9223372038089343698
4611686019661955794
13835058056516731602
3096040143925676201
12319412180780452009
7707726162353064105
16931098199207839913
1906652839273745429
11130024876128521237
15741710894555909141
6518338857701133333
5975809943035972467
15199181979890748275
10587495961463360371
results total= 16
```

So there are 16 correct input values for `0x92EE577B63E80B73` as a result.

The second is 1234567890—it is indeed the value which was used by me originally while preparing this example.

Let's also try to research our algorithm a bit more. Acting on a sadistic whim, let's find if there are any possible input/output pairs in which the lower 32-bit parts are equal to each other?

Let's remove the *outp* constraint and add another, at line 17:

```
1 #!/usr/bin/env python
2
```

```
from z3 import *

C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))

s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)

print s.check()
m=s.model()
print m
print (" inp=0x%X" % m[inp].as_long())
print ("outp=0x%X" % m[outp].as_long())
```

It is indeed so:

```
sat
[i1 = 14869545517796235860,
 i3 = 8388171335828825253,
 i5 = 6918262285561543945,
 inp = 1370377541658871093,
 outp = 14543180351754208565,
 i4 = 10167065714588685486,
 i2 = 5541032613289652645]
 inp=0x13048F1D12C00535
outp=0xC9D3C17A12C00535
```

Let's be more sadistic and add another constraint: last 16 bits must be `0x1234`:

```
#!/usr/bin/env python

from z3 import *

C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))

s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
```

```
s.add(outp & 0xFFFF == 0x1234)

print s.check()
m=s.model()
print m
print (" inp=0x%X" % m[inp].as_long())
print ("outp=0x%X" % m[outp].as_long())
```

Oh yes, this possible as well:

```
sat
[i1 = 2834222860503985872,
 i3 = 2294680776671411152,
 i5 = 17492621421353821227,
 inp = 461881484695179828,
 outp = 419247225543463476,
 i4 = 2294680776671411152,
 i2 = 2834222860503985872]
 inp=0x668EEC35F961234
outp=0x5D177215F961234
```

Z3 works very fast and it implies that the algorithm is weak, it is not cryptographic at all (like the most of the amateur cryptography).

## 18.4 Cracking simple XOR cipher with Z3

Here is a problem: a text encrypted with simple XOR cipher. Trying all possible keys is not an option.

Relationships between plain text, cipher text and key can be described using simple system of equations. But we can do more: we can ask Z3 to find such a key (array of bytes), so the plain text will have as many lowercase letters (a...z) as possible, but a solution will still satisfy all conditions.

```
import sys, hexdump
from z3 import *

def xor_strings(s,t):
    # https://en.wikipedia.org/wiki/XOR_cipher#Example_implementation
    """xor two strings together"""
    return "".join(chr(ord(a)^ord(b)) for a,b in zip(s,t))

def read_file(fname):
    file=open(fname, mode='rb')
    content=file.read()
    file.close()
    return content

def chunks(l, n):
    """divide input buffer by n-len chunks"""
    n = max(1, n)
    return [l[i:i + n] for i in range(0, len(l), n)]

def print_model(m, KEY_LEN, key):
    # fetch key from model:
    test_key="".join(chr(int(obj_to_string(m[key[i]]))) for i in range(KEY_LEN))
    print "key="
    hexdump.hexdump(test_key)

    # decrypt using the key:
    tmp=chunks(cipher_file, KEY_LEN)
```

```
    plain_attempt="".join(map(lambda x: xor_strings(x, test_key), tmp))
    print "plain="
    hexdump.hexdump(plain_attempt)

def try_len(KEY_LEN, cipher_file):
    cipher_len=len(cipher_file)
    print "len=", KEY_LEN
    s=Optimize()

    # variables for each byte of key:
    key=[BitVec('key_%d' % i, 8) for i in range (KEY_LEN)]
    # variables for each byte of input cipher text:
    cipher=[BitVec('cipher_%d' % i, 8) for i in range (cipher_len)]
    # variables for each byte of input plain text:
    plain=[BitVec('plain_%d' % i, 8) for i in range (cipher_len)]
    # variable for each byte of plain text: 1 if the byte in 'a'...'z' range:
    az_in_plain=[Int('az_in_plain_%d' % i) for i in range (cipher_len)]

    for i in range(cipher_len):
        # assign each byte of cipher text from the input file:
        s.add(cipher[i]==ord(cipher_file[i]))
        # plain text is cipher text XOR-ed with key:
        s.add(plain[i]==cipher[i]^key[i % KEY_LEN])
        # each byte must be in printable range, or CR of LF:
        s.add(Or(And(plain[i]>=0x20, plain[i]<=0x7E),plain[i]==0xA,plain[i]==0xD))
        # 1 if in 'a'...'z' range, 0 otherwise:
        s.add(az_in_plain[i]==If(And(plain[i]>=ord('a'),plain[i]<=ord('z')), 1, 0))

    # find solution, where the sum of all az_in_plain variables is maximum:
    s.maximize(Sum(*az_in_plain))

    if s.check()==unsat:
        return
    m=s.model()
    print_model(m, KEY_LEN, key)

cipher_file=read_file (sys.argv[1])

for i in range(1,20):
    try_len(i, cipher_file)

#try_len(17, cipher_file)
```

( The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/crypto/XOR/1.py )

Let's try it on a small 350-bytes file[134]:

```
% python 1.py cipher1.txt
len= 1
len= 2
len= 3
len= 4
len= 5

...

len= 16
```

[134] https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/crypto/XOR/cipher1.txt

```
len= 17
key=
00000000: 90 A0 21 52 48 84 FB FF  86 83 CF 50 46 12 7A F9  ..!RH......PF.z.
00000010: 36                                                6
plain=
00000000: 4D 72 2E 22 54 63 65 72  6F 6F 63 6D 27 48 6F 6C  Mr."Tceroocm'Hol
00000010: 6A 65 73 2C 22 70 63 6F  20 74 61 73 26 72 73 75  jes,"pco tas&rsu
00000020: 61 6B 6C 79 20 74 62 79  79 20 6F 61 74 63 27 69  akly tbyy oatc'i
00000030: 6E 20 73 68 65 20 6F 68  79 6E 69 6D 67 73 2A 27  n she ohynimgs*'
00000040: 73 61 76 62 0D 0A 75 72  68 65 20 74 6B 6F 73 63  savb..urhe tkosc
00000050: 27 6E 6F 74 27 69 6E 66  70 62 7A 75 65 6D 74 20  'not'infpbzuemt
00000060: 69 64 63 61 73 6E 6F 6E  73 22 70 63 65 6E 23 68  idcasnons"pcen#h
00000070: 65 26 70 61 73 20 72 70  20 61 6E 6B 2B 6E 69 64  e&pas rp ank+nid
00000080: 68 74 2A 27 77 61 73 27  73 65 61 76 62 6F 0D 0A  ht*'was'seavbo..
00000090: 62 74 20 72 6F 65 20 62  75 65 61 6B 64 66 78 74  bt roe bueakdfxt
000000A0: 20 77 61 62 6A 62 2E 20  49 27 73 74 6F 6D 63 2B   wabjb. I'stomc+
000000B0: 75 70 6C 6E 20 72 6F 65  20 68 62 61 72 74 6A 2A  upln roe hbartj*
000000C0: 79 75 67 23 61 6E 62 27  70 69 63 6C 65 64 20 77  yug#anb'picled w
000000D0: 77 2B 74 68 66 0D 0A 75  73 69 63 6B 27 77 68 69  w+thf..usick'whi
000000E0: 61 6F 2B 6F 75 71 20 76  6F 74 69 74 6F 75 20 68  ao+ouq votitou h
000000F0: 61 66 27 67 65 66 77 20  62 63 6F 69 6E 64 27 68  af'gefw bcoind'h
00000100: 69 6D 22 73 63 65 20 6D  69 67 6E 73 20 62 65 61  im"sce migns bea
00000110: 6F 72 65 2C 27 42 74 20  74 61 73 26 66 0D 0A 66  ore,'Bt tas&f..f
00000120: 6E 6E 65 2C 22 73 63 69  63 68 20 70 6F 62 63 65  nne,"scich pobce
00000130: 20 68 66 20 77 6D 68 6F  2C 20 61 75 6C 64 68 75   hf wmho, auldhu
00000140: 73 2D 6F 65 61 64 67 63  27 20 6F 65 20 74 6E 62  s-oeadgc' oe tnb
00000150: 20 73 6F 75 74 20 77 6A  6E 68 68 20 6A 73         sout wjnhh js
len= 18
len= 19
```

This is not readable. But what is interesting, the solution exist only for 17-byte key.

What do we know about English language texts? Digits are rare there, so we can *minimize* them in plain text.

There are so called *digraphs*—a very popular combinations of two letters. The most popular in English are: *th*, *he*, *in* and *er*. We can count them in plain text and *maximize* them:

```
...

    # ... for each byte of plain text: 1 if the byte is digit:
    digits_in_plain=[Int('digits_in_plain_%d' % i) for i in range (cipher_len)]
    # ... for each byte of plain text: 1 if the byte + next byte is "th" characters:
    th_in_plain=[Int('th_in_plain_%d' % i) for i in range (cipher_len-1)]
    # ... etc:
    he_in_plain=[Int('he_in_plain_%d' % i) for i in range (cipher_len-1)]
    in_in_plain=[Int('in_in_plain_%d' % i) for i in range (cipher_len-1)]
    er_in_plain=[Int('er_in_plain_%d' % i) for i in range (cipher_len-1)]

...

    for i in range(cipher_len-1):
        # ... for each byte of plain text: 1 if the byte + next byte is "th" characters:
        s.add(th_in_plain[i]==(If(And(plain[i]==ord('t'),plain[i+1]==ord('h')), 1, 0)))
        # ... etc:
        s.add(he_in_plain[i]==(If(And(plain[i]==ord('h'),plain[i+1]==ord('e')), 1, 0)))
        s.add(in_in_plain[i]==(If(And(plain[i]==ord('i'),plain[i+1]==ord('n')), 1, 0)))
        s.add(er_in_plain[i]==(If(And(plain[i]==ord('e'),plain[i+1]==ord('r')), 1, 0)))

    # find solution, where the sum of all az_in_plain variables is maximum:
```

```
    s.maximize(Sum(*az_in_plain))
    # ... and sum of digits_in_plain is minimum:
    s.minimize(Sum(*digits_in_plain))

    # "maximize" presence of "th", "he", "in" and "er" digraphs:
    s.maximize(Sum(*th_in_plain))
    s.maximize(Sum(*he_in_plain))
    s.maximize(Sum(*in_in_plain))
    s.maximize(Sum(*er_in_plain))

...
```

( The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/crypto/XOR/2.py )

Now this is something familiar:

```
len= 17
key=
00000000: 90 A0 22 50 4F 8F FB FF  85 83 CF 56 41 12 7A FE  .."PO......VA.z.
00000010: 31                                                1
plain=
00000000: 4D 72 2D 20 53 68 65 72  6C 6F 63 6B 20 48 6F 6B  Mr- Sherlock Hok
00000010: 6D 65 73 2F 20 77 68 6F  20 77 61 73 20 75 73 75  mes/ who was usu
00000020: 66 6C 6C 79 23 76 65 72  79 20 6C 61 74 65 20 69  flly#very late i
00000030: 6E 27 74 68 65 23 6D 6F  72 6E 69 6E 67 73 2C 20  n'the#mornings,
00000040: 73 61 71 65 0D 0A 76 70  6F 6E 20 74 68 6F 73 65  saqe..vpon those
00000050: 20 6E 6F 73 20 69 6E 65  72 65 71 75 65 6E 74 20   nos inerequent
00000060: 6F 63 63 61 74 69 6F 6E  70 20 77 68 65 6E 20 68  occationp when h
00000070: 65 20 77 61 73 27 75 70  20 62 6C 6C 20 6E 69 67  e was'up bll nig
00000080: 68 74 2C 20 77 61 74 20  73 65 62 74 65 64 0D 0A  ht, wat sebted..
00000090: 61 74 20 74 68 65 20 65  72 65 61 68 66 61 73 74  at the ereahfast
000000A0: 20 74 61 62 6C 65 2E 20  4E 20 73 74 6C 6F 64 20   table. N stlod
000000B0: 75 70 6F 6E 20 74 68 65  20 6F 65 61 72 77 68 2D  upon the oearwh-
000000C0: 72 75 67 20 61 6E 64 20  70 69 64 6B 65 64 23 75  rug and pidked#u
000000D0: 70 20 74 68 65 0D 0A 73  74 69 63 6C 20 77 68 6A  p the..sticl whj
000000E0: 63 68 20 6F 75 72 20 76  69 73 69 74 68 72 20 68  ch our visithr h
000000F0: 62 64 20 6C 65 66 74 20  62 65 68 69 6E 63 20 68  bd left behinc h
00000100: 69 6E 20 74 68 65 20 6E  69 67 68 74 20 62 62 66  in the night bbf
00000110: 6F 72 66 2E 20 49 74 20  77 61 73 20 61 0D 0A 61  orf. It was a..a
00000120: 69 6E 65 2F 20 74 68 69  63 6B 20 70 69 65 63 65  ine/ thick piece
00000130: 27 6F 66 20 74 6F 6F 64  2C 20 62 75 6C 62 6F 75  'of tood, bulbou
00000140: 73 2A 68 65 61 67 65 64  2C 20 6F 66 20 74 68 65  s*heaged, of the
00000150: 20 73 68 72 74 20 74 68  69 63 68 20 69 73           shrt thich is
```

Several characters are wrong. But we can fix them, adding these conditions:

```
...
    # 3 known characters of plain text:
    s.add(plain[0xf]==ord('l'))
    s.add(plain[0x20]==ord('a'))
    s.add(plain[0x57]==ord('f'))
...
```

( The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/crypto/XOR/3.py )

This key is seems correct:

```
len= 17
key=
00000000: 90 A0 21 50 4F 8F FB FF  85 83 CF 56 41 12 7A F9  ..!PO......VA.z.
00000010: 31                                                1
```

```
plain=
00000000: 4D 72 2E 20 53 68 65 72  6C 6F 63 6B 20 48 6F 6C  Mr. Sherlock Hol
00000010: 6D 65 73 2C 20 77 68 6F  20 77 61 73 20 75 73 75  mes, who was usu
00000020: 61 6C 6C 79 20 76 65 72  79 20 6C 61 74 65 20 69  ally very late i
00000030: 6E 20 74 68 65 20 6D 6F  72 6E 69 6E 67 73 2C 20  n the mornings,
00000040: 73 61 76 65 0D 0A 75 70  6F 6E 20 74 68 6F 73 65  save..upon those
00000050: 20 6E 6F 74 20 69 6E 66  72 65 71 75 65 6E 74 20   not infrequent
00000060: 6F 63 63 61 73 69 6F 6E  73 20 77 68 65 6E 20 68  occasions when h
00000070: 65 20 77 61 73 20 75 70  20 61 6C 6C 20 6E 69 67  e was up all nig
00000080: 68 74 2C 20 77 61 73 20  73 65 61 74 65 64 0D 0A  ht, was seated..
00000090: 61 74 20 74 68 65 20 62  72 65 61 6B 66 61 73 74  at the breakfast
000000A0: 20 74 61 62 6C 65 2E 20  49 20 73 74 6F 6F 64 20   table. I stood
000000B0: 75 70 6F 6E 20 74 68 65  20 68 65 61 72 74 68 2D  upon the hearth-
000000C0: 72 75 67 20 61 6E 64 20  70 69 63 6B 65 64 20 75  rug and picked u
000000D0: 70 20 74 68 65 0D 0A 73  74 69 63 6B 20 77 68 69  p the..stick whi
000000E0: 63 68 20 6F 75 72 20 76  69 73 69 74 6F 72 20 68  ch our visitor h
000000F0: 61 64 20 6C 65 66 74 20  62 65 68 69 6E 64 20 68  ad left behind h
00000100: 69 6D 20 74 68 65 20 6E  69 67 68 74 20 62 65 66  im the night bef
00000110: 6F 72 65 2E 20 49 74 20  77 61 73 20 61 0D 0A 66  ore. It was a..f
00000120: 69 6E 65 2C 20 74 68 69  63 6B 20 70 69 65 63 65  ine, thick piece
00000130: 20 6F 66 20 77 6F 6F 64  2C 20 62 75 6C 62 6F 75   of wood, bulbou
00000140: 73 2D 68 65 61 64 65 64  2C 20 6F 66 20 74 68 65  s-headed, of the
00000150: 20 73 6F 72 74 20 77 68  69 63 68 20 69 73          sort which is
```

So this is correct 17-byte XOR-key.

Needless to say, that the bigger ciphertext for analysis we have, the better. That 350-byte file is in fact the beginning of bigger file I prepared (https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/crypto/XOR/cipher2.txt, 12903 bytes). And a correct key for it can be found for it without additional *heuristics* we used here.

SMT solver is overkill for this. I once solved this problem naively, and it was much faster: https://yurichev.com/blog/XOR_mask_2/. Nevertheless, this is yet another demonstration of yet another optimization problem.

The files: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/crypto/XOR.

# 19 First-Order Logic

## 19.1 Exercise 56 from TAOCP "7.1.1 Boolean Basics", solving it using Z3

Page 41 from fasc0b.ps or http://www.cs.utsa.edu/~wagner/knuth/fasc0b.pdf.

▶ **56.** [*20*] The satisfiability problem for a Boolean function $f(x_1, x_2, \ldots, x_n)$ can be stated formally as the question of whether or not the quantified formula

$$\exists x_1 \, \exists x_2 \ldots \exists x_n \, f(x_1, x_2, \ldots, x_n)$$

is true; here '$\exists x_j \, \alpha$' means, "there exists a Boolean value $x_j$ such that $\alpha$ holds."

A much more general evaluation problem arises when we replace one or more of the existential quantifiers $\exists x_j$ by the universal quantifier $\forall x_j$, where '$\forall x_j \, \alpha$' means, "for all Boolean values $x_j$, $\alpha$ holds."

Which of the eight quantified formulas $\exists x \, \exists y \, \exists z \, f(x,y,z)$, $\exists x \, \exists y \, \forall z \, f(x,y,z)$, $\ldots$, $\forall x \, \forall y \, \forall z \, f(x,y,z)$ are true when $f(x,y,z) = (x \vee y) \wedge (\bar{x} \vee z) \wedge (y \vee \bar{z})$?

Figure 42: Page 41

For exists/forall/forall:

```
(assert
        (exists ((x Bool)) (forall ((y Bool)) (forall ((z Bool))
                (and
```

```
                                (or x y)
                                (or (not x) z)
                                (or y (not z))
                        )))
                )
)
(check-sat)
```

All the rest: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/FOL/TAOCP_7_1_1_exercise_56.

Results:

```
z3 -smt2 KnuthAAA.smt
z3 -smt2 KnuthAAE.smt
z3 -smt2 KnuthAEA.smt
z3 -smt2 KnuthAEE.smt
z3 -smt2 KnuthEAA.smt
z3 -smt2 KnuthEAE.smt
z3 -smt2 KnuthEEA.smt
z3 -smt2 KnuthEEE.smt
```

```
unsat
unsat
unsat
sat
unsat
unsat
sat
sat
```

### 19.2 Exercise 9 from TAOCP "7.1.1 Boolean Basics", solving it using Z3

Page 34 from fasc0b.ps or http://www.cs.utsa.edu/~wagner/knuth/fasc0b.pdf.

**9.** [*16*] True or false? (a) $(x \oplus y) \vee z = (x \vee z) \oplus (y \vee z)$; (b) $(w \oplus x \oplus y) \vee z = (w \vee z) \oplus (x \vee z) \oplus (y \vee z)$; (c) $(x \oplus y) \vee (y \oplus z) = (x \oplus z) \vee (y \oplus z)$.

Figure 43: Page 34

For (a):

```
(assert
        (forall ((x Bool) (y Bool) (z Bool))
                (=
                        (or (xor x y) z)
                        (xor (or x z) (or y z))
                )
        )
)
(check-sat)
```

For (b):

```
(assert
        (forall ((x Bool) (y Bool) (z Bool) (w Bool))
                (=
```

```
                        (or (xor w x y) z)
                        (xor (or w z) (or x z) (or y z))
                )
        )
)
(check-sat)
```

For (c):

```
(assert
        (forall ((x Bool) (y Bool) (z Bool))
                (=
                        (or (xor x y) (xor y z))
                        (or (xor x z) (xor y z))
                )
        )
)
(check-sat)
```

Results:

```
% z3 -smt2 Knuth_a.smt
unsat
% z3 -smt2 Knuth_b.smt
sat
% z3 -smt2 Knuth_c.smt
sat
```

# 20 Cellular automata

## 20.1 Conway's "Game of Life"

### 20.1.1 Reversing back the state of "Game of Life"

How could we reverse back a known state of GoL? This can be solved by brute-force, but this is extremely slow and inefficient.

Let's try to use SAT solver.

First, we need to define a function which will tell, if the new cell will be created/born, preserved/stay or died. Quick refresher: cell is born if it has 3 neighbours, it stays alive if it has 2 or 3 neighbours, it dies in any other case.

This is how I can define a function reflecting state of a new cell in the next state:

```
if center==true:
        return popcnt2(neighbours) || popcnt3(neighbours)
if center==false
        return popcnt3(neighbours)
```

We can get rid of "if" construction:

```
result=(center==true && (popcnt2(neighbours) || popcnt3(neighbours))) || (center==false
    && popcnt3(neighbours))
```

…where "center" is state of central cell, "neighbours" are 8 neighbouring cells, popcnt2 is a function which returns True if it has exactly 2 bits on input, popcnt3 is the same, but for 3 bits (just like these were used in my "Minesweeper" example (3.7)).

Using Wolfram Mathematica, I first create all helper functions and truth table for the function, which returns *true*, if a cell must be present in the next state, or *false* if not:

```
In[1]:= popcount[n_Integer]:=IntegerDigits[n,2] // Total
```

```
In[2]:= popcount2[n_Integer]:=Equal[popcount[n],2]

In[3]:= popcount3[n_Integer]:=Equal[popcount[n],3]

In[4]:= newcell[center_Integer,neighbours_Integer]:=(center==1 && (popcount2[neighbours
    ]|| popcount3[neighbours]))||
(center==0 && popcount3[neighbours])

In[13]:= NewCellIsTrue=Flatten[Table[Join[{center},PadLeft[IntegerDigits[neighbours
    ,2],8]] ->
Boole[newcell[center, neighbours]],{neighbours,0,255},{center,0,1}]]

Out[13]= {{0,0,0,0,0,0,0,0,0}->0,
{1,0,0,0,0,0,0,0,0}->0,
{0,0,0,0,0,0,0,0,1}->0,
{1,0,0,0,0,0,0,0,1}->0,
{0,0,0,0,0,0,0,1,0}->0,
{1,0,0,0,0,0,0,1,0}->0,
{0,0,0,0,0,0,0,1,1}->0,
{1,0,0,0,0,0,0,1,1}->1,

...
```

Now we can create a CNF-expression out of truth table:

```
In[14]:= BooleanConvert[BooleanFunction[NewCellIsTrue,{center,a,b,c,d,e,f,g,h}],"CNF"]
Out[14]= (!a||!b||!c||!d)&&(!a||!b||!c||!e)&&(!a||!b||!c||!f)&&(!a||!b||!c||!g)&&(!a||!b
    ||!c||!h)&&
(!a||!b||!d||!e)&&(!a||!b||!d||!f)&&(!a||!b||!d||!g)&&(!a||!b||!d||!h)&&(!a||!b||!e||!f)
    &&
(!a||!b||!e||!g)&&(!a||!b||!e||!h)&&(!a||!b||!f||!g)&&(!a||!b||!f||!h)&&(!a||!b||!g||!h)
    &&
(!a||!c||!d||!e)&&(!a||!c||!d||!f)&&(!a||!c||!d||!g)&&(!a||!c||!d||!h)&&(!a||!c||!e||!f)
    &&
(!a||!c||!e||!g)&&(!a||!c||!e||!h)&&(!a||!c||!f||!g)&&(!a||!c||!f||!h)&&

...
```

Also, we need a second function, *inverted one*, which will return *true* if the cell must be absent in the next state, or *false* otherwise:

```
In[15]:= NewCellIsFalse=Flatten[Table[Join[{center},PadLeft[IntegerDigits[neighbours
    ,2],8]] ->
Boole[Not[newcell[center, neighbours]]],{neighbours,0,255},{center,0,1}]]
Out[15]= {{0,0,0,0,0,0,0,0,0}->1,
{1,0,0,0,0,0,0,0,0}->1,
{0,0,0,0,0,0,0,0,1}->1,
{1,0,0,0,0,0,0,0,1}->1,
{0,0,0,0,0,0,0,1,0}->1,

...

In[16]:= BooleanConvert[BooleanFunction[NewCellIsFalse,{center,a,b,c,d,e,f,g,h}],"CNF"]
Out[16]= (!a||!b||!c||d||e||f||g||h)&&(!a||!b||c||!d||e||f||g||h)&&(!a||!b||c||d||!e||f
    ||g||h)&&
(!a||!b||c||d||e||!f||g||h)&&(!a||!b||c||d||e||f||!g||h)&&(!a||!b||c||d||e||f||g||!h)&&
(!a||!b||!center||d||e||f||g||h)&&(!a||b||!c||!d||e||f||g||h)&&(!a||b||!c||d||!e||f||g||
    h)&&
```

```
(!a||b||!c||d||e||!f||g||h)&&(!a||b||!c||d||e||f||!g||h)&&(!a||b||!c||d||e||f||g||!h)&&
(!a||b||c||!d||!e||f||g||h)&&(!a||b||c||!d||e||!f||g||h)&&(!a||b||c||!d||e||f||!g||h)&&

...
```

Using the very same way as in my "Minesweeper" example, I can convert CNF expression to list of clauses:

```
def mathematica_to_CNF (s, center, a):
    s=s.replace("center", center)
    s=s.replace("a", a[0]).replace("b", a[1]).replace("c", a[2]).replace("d", a[3])
    s=s.replace("e", a[4]).replace("f", a[5]).replace("g", a[6]).replace("h", a[7])
    s=s.replace("!", "-").replace("||", " ").replace("(", "").replace(")", "")
    s=s.split ("&&")
    return s
```

And again, as in "Minesweeper", there is an invisible border, to make processing simpler. SAT variables are also numbered as in previous example:

```
 1    2   3   4   5   6   7   8   9  10  11
12   13  14  15  16  17  18  19  20  21  22
23   24  25  26  27  28  29  30  31  32  33
34   35  36  37  38  39  40  41  42  43  44

...

100 101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120 121
```

Also, there is a visible border, always fixed to *False*, to make things simpler.

Now the working source code. Whenever we encounter "*" in `final_state[]`, we add clauses generated by `cell_is_true()` function, or `cell_is_false()` if otherwise. When we get a solution, it is negated and added to the list of clauses, so when minisat is executed next time, it will skip solution which was already printed.

```
...

def cell_is_false (center, a):
    s="(!a||!b||!c||d||e||f||g||h)&&(!a||!b||c||!d||e||f||g||h)&&(!a||!b||c||d||!e||f||g
        ||h)&&" \
      "(!a||!b||c||d||e||!f||g||h)&&(!a||!b||c||d||e||f||!g||h)&&(!a||!b||c||d||e||f||g
        ||!h)&&" \
      "(!a||!b||!center||d||e||f||g||h)&&(!a||b||!c||!d||e||f||g||h)&&(!a||b||!c||d||!e
        ||f||g||h)&&" \
      "(!a||b||!c||d||e||!f||g||h)&&(!a||b||!c||d||e||f||!g||h)&&(!a||b||!c||d||e||f||g
        ||!h)&&" \
      "(!a||b||c||!d||!e||f||g||h)&&(!a||b||c||!d||e||!f||g||h)&&(!a||b||c||!d||e||f||!g
        ||h)&&" \
      "(!a||b||c||!d||e||f||g||!h)&&(!a||b||c||d||!e||!f||g||h)&&(!a||b||c||d||!e||f||!g
        ||h)&&" \
      "(!a||b||c||d||!e||f||g||!h)&&(!a||b||c||d||e||!f||!g||h)&&(!a||b||c||d||e||!f||g
        ||!h)&&" \
      "(!a||b||c||d||e||f||!g||!h)&&(!a||!c||!center||d||e||f||g||h)&&(!a||c||!center||!
        d||e||f||g||h)&&" \
      "(!a||c||!center||d||!e||f||g||h)&&(!a||c||!center||d||e||!f||g||h)&&(!a||c||!
        center||d||e||f||!g||h)&&" \
      "(!a||c||!center||d||e||f||g||!h)&&(a||!b||!c||!d||e||f||g||h)&&(a||!b||!c||d||!e
        ||f||g||h)&&" \
      "(a||!b||!c||d||e||!f||g||h)&&(a||!b||!c||d||e||f||!g||h)&&(a||!b||!c||d||e||f||g
        ||!h)&&" \
```

```
        "(a||!b||c||!d||!e||f||g||h)&&(a||!b||c||!d||e||!f||g||h)&&(a||!b||c||!d||e||f||!g||h)&&" \
        "(a||!b||c||!d||e||f||g||!h)&&(a||!b||c||d||!e||!f||g||h)&&(a||!b||c||d||!e||f||!g||h)&&" \
        "(a||!b||c||d||!e||f||g||!h)&&(a||!b||c||d||e||!f||!g||h)&&(a||!b||c||d||e||!f||g||!h)&&" \
        "(a||!b||c||d||e||f||!g||!h)&&(a||b||!c||!d||!e||f||g||h)&&(a||b||!c||!d||e||!f||g||h)&&" \
        "(a||b||!c||!d||e||f||!g||h)&&(a||b||!c||!d||e||f||g||!h)&&(a||b||!c||d||!e||!f||g||h)&&" \
        "(a||b||!c||d||!e||f||!g||h)&&(a||b||!c||d||!e||f||g||!h)&&(a||b||!c||d||e||!f||!g||h)&&" \
        "(a||b||!c||d||e||!f||g||!h)&&(a||b||!c||d||e||f||!g||!h)&&(a||b||c||!d||!e||!f||g||h)&&" \
        "(a||b||c||!d||!e||f||!g||h)&&(a||b||c||!d||!e||f||g||!h)&&(a||b||c||!d||e||!f||!g||h)&&" \
        "(a||b||c||!d||e||!f||g||!h)&&(a||b||c||!d||e||f||!g||!h)&&(a||b||c||d||!e||!f||!g||h)&&" \
        "(a||b||c||d||!e||!f||g||!h)&&(a||b||c||d||!e||f||!g||!h)&&(a||b||c||d||e||!f||!g||!h)&&" \
        "(!b||!c||!center||d||e||f||g||h)&&(!b||c||!center||!d||e||f||g||h)&&(!b||c||!center||d||!e||f||g||h)&&" \
        "(!b||c||!center||d||e||!f||g||h)&&(!b||c||!center||d||e||f||!g||h)&&(!b||c||!center||d||e||f||g||!h)&&" \
        "(b||!c||!center||!d||e||f||g||h)&&(b||!c||!center||d||!e||f||g||h)&&(b||!c||!center||d||e||!f||g||h)&&" \
        "(b||!c||!center||d||e||f||!g||h)&&(b||!c||!center||d||e||f||g||!h)&&(b||c||!center||!d||!e||f||g||h)&&" \
        "(b||c||!center||!d||e||!f||g||h)&&(b||c||!center||!d||e||f||!g||h)&&(b||c||!center||!d||e||f||g||!h)&&" \
        "(b||c||!center||d||!e||!f||g||h)&&(b||c||!center||d||!e||f||!g||h)&&(b||c||!center||d||!e||f||g||!h)&&" \
        "(b||c||!center||d||e||!f||!g||h)&&(b||c||!center||d||e||!f||g||!h)&&(b||c||!center||d||e||f||!g||!h)"

    return mathematica_to_CNF(s, center, a)

def cell_is_true (center, a):
    s="(!a||!b||!c||!d)&&(!a||!b||!c||!e)&&(!a||!b||!c||!f)&&(!a||!b||!c||!g)&&(!a||!b||!c||!h)&&" \
        "(!a||!b||!d||!e)&&(!a||!b||!d||!f)&&(!a||!b||!d||!g)&&(!a||!b||!d||!h)&&(!a||!b||!e||!f)&&" \
        "(!a||!b||!e||!g)&&(!a||!b||!e||!h)&&(!a||!b||!f||!g)&&(!a||!b||!f||!h)&&(!a||!b||!g||!h)&&" \
        "(!a||!c||!d||!e)&&(!a||!c||!d||!f)&&(!a||!c||!d||!g)&&(!a||!c||!d||!h)&&(!a||!c||!e||!f)&&" \
        "(!a||!c||!e||!g)&&(!a||!c||!e||!h)&&(!a||!c||!f||!g)&&(!a||!c||!f||!h)&&(!a||!c||!g||!h)&&" \
        "(!a||!d||!e||!f)&&(!a||!d||!e||!g)&&(!a||!d||!e||!h)&&(!a||!d||!f||!g)&&(!a||!d||!f||!h)&&" \
        "(!a||!d||!g||!h)&&(!a||!e||!f||!g)&&(!a||!e||!f||!h)&&(!a||!e||!g||!h)&&(!a||!f||!g||!h)&&" \
        "(a||b||c||center||d||e||f)&&(a||b||c||center||d||e||g)&&(a||b||c||center||d||e||h)&&" \
        "(a||b||c||center||d||f||g)&&(a||b||c||center||d||f||h)&&(a||b||c||center||d||g||h)&&" \
```

```
        "(a||b||c||center||e||f||g)&&(a||b||c||center||e||f||h)&&(a||b||c||center||e||g||h)&&" \
        "(a||b||c||center||f||g||h)&&(a||b||c||d||e||f||g)&&(a||b||c||d||e||f||h)&&(a||b||c||d||e||g||h)&&" \
        "(a||b||c||d||f||g||h)&&(a||b||c||e||f||g||h)&&(a||b||center||d||e||f||g)&&(a||b||center||d||e||f||h)&&" \
        "(a||b||center||d||e||g||h)&&(a||b||center||d||f||g||h)&&(a||b||center||e||f||g||h)&&" \
        "(a||b||d||e||f||g||h)&&(a||c||center||d||e||f||g)&&(a||c||center||d||e||f||h)&&" \
        "(a||c||center||d||e||g||h)&&(a||c||center||d||f||g||h)&&(a||c||center||e||f||g||h)&&" \
        "(a||c||d||e||f||g||h)&&(a||center||d||e||f||g||h)&&(!b||!c||!d||!e)&&(!b||!c||!d||!f)&&" \
        "(!b||!c||!d||!g)&&(!b||!c||!d||!h)&&(!b||!c||!e||!f)&&(!b||!c||!e||!g)&&(!b||!c||!e||!h)&&" \
        "(!b||!c||!f||!g)&&(!b||!c||!f||!h)&&(!b||!c||!g||!h)&&(!b||!d||!e||!f)&&(!b||!d||!e||!g)&&" \
        "(!b||!d||!e||!h)&&(!b||!d||!f||!g)&&(!b||!d||!f||!h)&&(!b||!d||!g||!h)&&(!b||!e||!f||!g)&&" \
        "(!b||!e||!f||!h)&&(!b||!e||!g||!h)&&(!b||!f||!g||!h)&&(b||c||center||d||e||f||g)&&" \
        "(b||c||center||d||e||f||h)&&(b||c||center||d||e||g||h)&&(b||c||center||d||f||g||h)&&" \
        "(b||c||center||e||f||g||h)&&(b||c||d||e||f||g||h)&&(b||center||d||e||f||g||h)&&" \
        "(!c||!d||!e||!f)&&(!c||!d||!e||!g)&&(!c||!d||!e||!h)&&(!c||!d||!f||!g)&&(!c||!d||!f||!h)&&" \
        "(!c||!d||!g||!h)&&(!c||!e||!f||!g)&&(!c||!e||!f||!h)&&(!c||!e||!g||!h)&&(!c||!f||!g||!h)&&" \
        "(c||center||d||e||f||g||h)&&(!d||!e||!f||!g)&&(!d||!e||!f||!h)&&(!d||!e||!g||!h)&&(!d||!f||!g||!h)&&" \
        "(!e||!f||!g||!h)"

    return mathematica_to_CNF(s, center, a)

...
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/CA/GoL/GoL_SAT_utils.py )

```
#!/usr/bin/python3

import os
from GoL_SAT_utils import *

final_state=[
" * ",
"* *",
" * "]

H=len(final_state) # HEIGHT
W=len(final_state[0]) # WIDTH

print ("HEIGHT=", H, "WIDTH=", W)

VARS_TOTAL=W*H+1
VAR_FALSE=str(VARS_TOTAL)
```

```
def try_again (clauses):
    # rules for the main part of grid
    for r in range(H):
        for c in range(W):
            if final_state[r][c]=="*":
                clauses=clauses+cell_is_true(coords_to_var(r, c, H, W), get_neighbours(r
                    , c, H, W))
            else:
                clauses=clauses+cell_is_false(coords_to_var(r, c, H, W), get_neighbours(
                    r, c, H, W))

    # cells behind visible grid must always be false:
    for c in range(-1, W+1):
        for r in [-1,H]:
            clauses=clauses+cell_is_false(coords_to_var(r, c, H, W), get_neighbours(r, c
                , H, W))
    for c in [-1,W]:
        for r in range(-1, H+1):
            clauses=clauses+cell_is_false(coords_to_var(r, c, H, W), get_neighbours(r, c
                , H, W))

    write_CNF("tmp.cnf", clauses, VARS_TOTAL)

    print ("%d clauses" % len(clauses))

    solution=run_minisat ("tmp.cnf")
    os.remove("tmp.cnf")
    if solution==None:
        print ("unsat!")
        exit(0)

    grid=SAT_solution_to_grid(solution, H, W)

    print_grid(grid)
    write_RLE(grid)

    return grid

clauses=[]
# always false:
clauses.append ("-"+VAR_FALSE)

while True:
    solution=try_again(clauses)
    clauses.append(negate_clause(grid_to_clause(solution, H, W)))
    print ("")
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/CA/GoL/reverse1.py )

Here is the result:

```
HEIGHT= 3 WIDTH= 3
2525 clauses
.*.
*.*
.*.
1.rle written
```

```
2526 clauses
.**
*..
*.*
2.rle written

2527 clauses
**.
..*
*.*
3.rle written

2528 clauses
*.*
*..
.**
4.rle written

2529 clauses
*.*
..*
**.
5.rle written

2530 clauses
*.*
.*.
*.*
6.rle written

2531 clauses
unsat!
```

The first result is the same as initial state. Indeed: this is "still life", i.e., state which will never change, and it is correct solution. The last solution is also valid.

Now the problem: 2nd, 3rd, 4th and 5th solutions are equivalent to each other, they just mirrored or rotated. In fact, this is reflectional[135] (like in mirror) and rotational[136] symmetries. We can solve this easily: we will take each solution, reflect and rotate it and add them negated to the list of clauses, so minisat will skip them during its work:

```
...

while True:
    solution=try_again(clauses)
    clauses.append(negate_clause(grid_to_clause(solution, H, W)))
    clauses.append(negate_clause(grid_to_clause(reflect_vertically(solution), H, W)))
    clauses.append(negate_clause(grid_to_clause(reflect_horizontally(solution), H, W)))
    # is this square?
    if W==H:
        clauses.append(negate_clause(grid_to_clause(rotate_square_array(solution,1), H,
            W)))
        clauses.append(negate_clause(grid_to_clause(rotate_square_array(solution,2), H,
            W)))
```

[135] https://en.wikipedia.org/wiki/Reflection_symmetry
[136] https://en.wikipedia.org/wiki/Rotational_symmetry

```
        clauses.append(negate_clause(grid_to_clause(rotate_square_array(solution,3), H,
            W)))
    print ""

...
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/CA/GoL/reverse2.py )

Functions `reflect_vertically()`, `reflect_horizontally` and `rotate_squarearray()` are simple array manipulation routines.

Now we get just 3 solutions:

```
HEIGHT= 3 WIDTH= 3
2525 clauses
.*.
*.*
.*.
1.rle written

2531 clauses
.**
*..
*.*
2.rle written

2537 clauses
*.*
.*.
*.*
3.rle written

2543 clauses
unsat!
```

This one has only one single ancestor:

```
final_state=[
" * ",
" * ",
" * "]
_PRE_END

_PRE_BEGIN
HEIGHT= 3 WIDTH= 3
2503 clauses
...
***
...
1.rle written

2509 clauses
unsat!
```

This is oscillator, of course.

How many states can lead to such picture?

```
final_state=[
"  *  ",
"     ",
" **  ",
```

```
"  *  ",
"  *  ",
" *** "]
```

28, these are few:

```
HEIGHT= 6 WIDTH= 5
5217 clauses
.*.*.
..*..
.**.*
..*..
..*.*
.**..
1.rle written

5220 clauses
.*.*.
..*..
.**.*
..*..
*.*.*
.**..
2.rle written

5223 clauses
..*.*
..**.
.**..
....*
*.*.*
.**..
3.rle written

5226 clauses
..*.*
..**.
.**..
*...*
..*.*
.**..
4.rle written

...
```

Now the biggest, “space invader”:

```
final_state=[
"             ",
"   *     *   ",
"    *   *    ",
"   *******   ",
"  ** *** **  ",
" *********** ",
" * ******* * ",
" * *     * * ",
"    ** **    ",
"             "]
```

```
HEIGHT= 10 WIDTH= 13
16469 clauses
..*.*.**.....
.....*****...
....**..*....
......*...*..
..**...*.*...
.*..*.*.**..*
*....*....*.*
..*.*..*.....
..*.....*.*..
....**..*.*..
1.rle written

16472 clauses
*.*.*.**.....
.....*****...
....**..*....
......*...*..
..**...*.*...
.*..*.*.**..*
*....*....*.*
..*.*..*.....
..*.....*.*..
....**..*.*..
2.rle written

16475 clauses
..*.*.**.....
*....*****...
....**..*....
......*...*..
..**...*.*...
.*..*.*.**..*
*....*....*.*
..*.*..*.....
..*.....*.*..
....**..*.*..
3.rle written

...
```

I don't know how many possible states can lead to "space invader", perhaps, too many. Had to stop it. And it slows down during execution, because number of clauses is increasing (because of negating solutions addition).

All solutions are also exported to RLE files, which can be opened by Golly[137].

### 20.1.2 Finding "still lives"

"Still life" in terms of GoL is a state which doesn't change at all.

First, using previous definitions, we will define a truth table of function, which will return *true*, if the center cell of the next state is the same as it has been in the previous state, i.e., hasn't been changed:

```
In[17]:= stillife=Flatten[Table[Join[{center},PadLeft[IntegerDigits[neighbours,2],8]]->
Boole[Boole[newcell[center,neighbours]]==center],{neighbours,0,255},{center,0,1}]]
Out[17]= {{0,0,0,0,0,0,0,0,0}->1,
```

[137] http://golly.sourceforge.net/

```
{1,0,0,0,0,0,0,0,0}->0,
{0,0,0,0,0,0,0,0,1}->1,
{1,0,0,0,0,0,0,0,1}->0,

...

In[18]:= BooleanConvert[BooleanFunction[stillife,{center,a,b,c,d,e,f,g,h}],"CNF"]
Out[18]= (!a||!b||!c||!center||!d)&&(!a||!b||!c||!center||!e)&&(!a||!b||!c||!center||!f)
    &&
(!a||!b||!c||!center||!g)&&(!a||!b||!c||!center||!h)&&(!a||!b||!c||center||d||e||f||g||h
    )&&
(!a||!b||c||center||!d||e||f||g||h)&&(!a||!b||c||center||d||!e||f||g||h)&&(!a||!b||c||
    center||d||e||!f||g||h)&&
(!a||!b||c||center||d||e||f||!g||h)&&(!a||!b||c||center||d||e||f||g||!h)&&(!a||!b||!
    center||!d||!e)&&

...
```

```
#!/usr/bin/python3

import os
from GoL_SAT_utils import *
import SL_common

W=3 # WIDTH
H=3 # HEIGHT

VARS_TOTAL=W*H+1
VAR_FALSE=str(VARS_TOTAL)

def try_again (clauses):
    # rules for the main part of grid
    for r in range(H):
        for c in range(W):
            clauses=clauses + SL_common.gen_SL(coords_to_var(r, c, H, W), get_neighbours
                (r, c, H, W))

    # cells behind visible grid must always be false:
    for c in range(-1, W+1):
        for r in [-1,H]:
            clauses=clauses+cell_is_false(coords_to_var(r, c, H, W), get_neighbours(r, c
                , H, W))
    for c in [-1,W]:
        for r in range(-1, H+1):
            clauses=clauses+cell_is_false(coords_to_var(r, c, H, W), get_neighbours(r, c
                , H, W))

    write_CNF("tmp.cnf", clauses, VARS_TOTAL)

    print ("%d clauses" % len(clauses))

    solution=run_minisat ("tmp.cnf")
    os.remove("tmp.cnf")
    if solution==None:
        print ("unsat!")
        exit(0)
```

```
    grid=SAT_solution_to_grid(solution, H, W)
    print_grid(grid)
    write_RLE(grid)

    return grid

clauses=[]
# always false:
clauses.append ("-"+VAR_FALSE)

while True:
    solution=try_again(clauses)
    clauses.append(negate_clause(grid_to_clause(solution, H, W)))
    clauses.append(negate_clause(grid_to_clause(my_utils.reflect_vertically(solution), H
        , W)))
    clauses.append(negate_clause(grid_to_clause(my_utils.reflect_horizontally(solution),
         H, W)))
    # is this square?
    if W==H:
        clauses.append(negate_clause(grid_to_clause(my_utils.rotate_rect_array(solution
            ,1), H, W)))
        clauses.append(negate_clause(grid_to_clause(my_utils.rotate_rect_array(solution
            ,2), H, W)))
        clauses.append(negate_clause(grid_to_clause(my_utils.rotate_rect_array(solution
            ,3), H, W)))
    print ("")
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/CA/GoL/SL1.py )

What we've got for $2 \cdot 2$?

```
1881 clauses
..
..
1.rle written

1887 clauses
**
**
2.rle written

1893 clauses
unsat!
```

Both solutions are correct: empty square will progress into empty square (no cells are born). $2 \cdot 2$ box is also known "still life".

What about $3 \cdot 3$ square?

```
2887 clauses
...
...
...
1.rle written

2893 clauses
.**
.**
...
```

```
2.rle written

2899 clauses
.**
*.*
**.
3.rle written

2905 clauses
.*.
*.*
**.
4.rle written

2911 clauses
.*.
*.*
.*.
5.rle written

2917 clauses
unsat!
```

Here is a problem: we see familiar $2 \cdot 2$ box, but shifted. This is indeed correct solution, but we don't interested in it, because it has been already seen.

What we can do is add another condition. We can force minisat to find solutions with no empty rows and columns. This is easy. These are SAT variables for $5 \cdot 5$ square:

```
1   2  3  4  5
6   7  8  9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

Each clause is "OR" clause, so all we have to do is to add 5 clauses:

```
1 OR 2 OR 3 OR 4 OR 5
6 OR 7 OR 8 OR 9 OR 10

...
```

That means that each row must have at least one *True* value somewhere. We can also do this for each column as well.

```

...

    # each row must contain at least one cell!
    for r in range(H):
        clauses.append(" ".join([coords_to_var(r, c, H, W) for c in range(W)]))

    # each column must contain at least one cell!
    for c in range(W):
        clauses.append(" ".join([coords_to_var(r, c, H, W) for r in range(H)]))

...
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/CA/GoL/SL2.py )

Now we can see that $3 \cdot 3$ square has 3 possible "still lives":

```
2893 clauses
.*.
*.*
**.
1.rle written

2899 clauses
.*.
*.*
.*.
2.rle written

2905 clauses
.**
*.*
**.
3.rle written

2911 clauses
unsat!
```

$4 \cdot 4$ has 7:

```
4169 clauses
..**
...*
***.
*...
1.rle written

4175 clauses
..**
..*.
*.*.
**..
2.rle written

4181 clauses
..**
.*.*
*.*.
**..
3.rle written

4187 clauses
..*.
.*.*
*.*.
**..
4.rle written

4193 clauses
.**.
*..*
*.*.
.*..
5.rle written
```

```
4199 clauses
..*.
.*.*
*.*.
.*..
6.rle written

4205 clauses
.**.
*..*
*..*
.**.
7.rle written

4211 clauses
unsat!
```

When I try large squares, like $20 \cdot 20$, funny things happen. First of all, minisat finds solutions not very pleasing aesthetically, but still correct, like:

```
61033 clauses
....**.**.**.**.**.*
**..*.**.**.**.**.**
*...................
.*..................
**..................
*...................
.*..................
**..................
*...................
.*..................
**..................
*...................
.*..................
**..................
*...................
.*..................
..*.................
...*................
***.................
*...................
1.rle written

...
```

Indeed: all rows and columns has at least one *True* value.

Then minisat begins to add smaller "still lives" into the whole picture:

```
61285 clauses
.**....**...**...**.
.**...*..*.*.*...*..
.......**...*......*
..................**
...**............*..
...*.*...........*..
....*.*........**...
**...*.*...**..*....
```

```
*.*...*....*....*...
.*..........****.*..
................*...
..*...**..******....
.*.*..*..*..........
*..*...*.*..****....
***.***..*.*....*...
....*..***.**..**...
**.*..*.............
.*.**.**..........**
*..*..*..*......*..*
**..**..**......**..
43.rle written
```

In other words, result is a square consisting of smaller "still lives". It then altering these parts slightly, shifting back and forth. Is it cheating? Anyway, it does it in a strict accordance to rules we defined.

But we want *denser* picture. We can add a rule: in all 5-cell chunks there must be at least one *True* cell. To achieve this, we just split the whole square by 5-cell chunks and add clause for each:

```
...

    # make result denser:
    lst=[]
    for r in range(H):
        for c in range(W):
            lst.append(coords_to_var(r, c, H, W))
    # divide them all by chunks and add to clauses:
    CHUNK_LEN=5
    for c in list_partition(lst,len(lst)/CHUNK_LEN):
        tmp=" ".join(c)
        clauses.append(tmp)

...
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/CA/GoL/SL3.py )

This is indeed denser:

```
61113 clauses
..**.**......*.*.*..
...*.*.....***.**.*.
...*..*...*.......*.
....*.*..*.*......**
...**.*.*..*...**.*.
..*...*.***.....*.*.
...*.*.*.......*..*..
****.*..*....*.**...
*....**.*....*.*....
...**..*...**..*....
..*..*....*....*.**.
.*.*.**....****.*..*
..*.*....*.*..*..**.
....*.****..*..*.*..
....**....*.*.**..*.
*.**...****.*..*.**.
**...**.....**.*....
...**..*..**..*.**.*
***.*.*..*.*..*.*.**
```

```
*....*....*....*....
1.rle written

61119 clauses
..**.**......*.*.*..
...*.*.....***.**.*.
...*..*...*.......*.
....*.*..*.*......**
...**.*.*..*...**.*.
..*...*.***.....*.*.
...*.*.*......*..*..
****.*..*....*.**...
*....**.*....*.*....
...**..*...**..*....
..*..*....*....*.**.
.*.*.**....****.*..*
..*.*....*.*..*..**.
....*.****..*..*.*..
....**....*.*.**..*.
*.**...****.*..*.**.
**...**.....**.*....
...**..*.***..*.**.*
***.*..*.*..*.*.*.**
*.......*..**.**....
2.rle written

...
```

Let's try more dense, one mandatory *true* cell per each 4-cell chunk:

```
61133 clauses
.**.**...*....**..**
*.*.*...*.*..*..*..*
*....*...*.*..*.**..
.***.*.....*.**.*...
..*.*.....**...*..*.
*......**..*...*.**.
**.....*...*.**.*...
...**...*...**..*...
**.*..*.*......*...*
.*...**.**..***.****
.*....*.*..*..*.*...
**.***...*.**...*.**
.*.*..****.....*..*.
*....*.....**..**.*.
*.***.*..**.*.....**
.*...*..*......**...
...*.*.**......*.***
..**.*.....**......*
*..*.*.**..*.*..***.
**....*.*...*...*...
1.rle written

61139 clauses
.**.**...*....**..**
*.*.*...*.*..*..*..*
*....*...*.*..*.**..
```

```
.***.*.....*.**.*...
..*.*.....**...*..*.
*......**..*...*.**.
**.....*...*.**.*...
...**...*...**..*...
**.*..*.*......*...*
.*...**.**..***.****
.*....*.*..*..*.*...
**.***...*.**...*.**
.*.*..****.....*..*.
*....*.....**..**.*.
*.***.*..**.*.....**
.*...*..*......**..*
...*.*.**......*.**.
..**.*.....**....*..
*..*.*.**..*.*...*.*
**....*.*...*.....**
2.rle written

...
```

...and even more: one cell per each 3-cell chunk:

```
61166 clauses
**.*..**...**.**....
*.**..*.*...*.*.*.**
....**..*...*...*.*.
.**..*.*.**.*.*.*.*.
..**.*.*...*.**.*.**
*...*.*.**.*....*.*.
**.*..*...*.*.***..*
.*.*.*.***..**...**.
.*.*.*.*..**...*.*..
**.**.*..*...**.*..*
..*...*.**.**.*.*.**
..*.**.*..*.*.*.*...
**.*.*...*..*.*.*...
.*.*...*.**..*..***.
.*..****.*....**...*
..*.*...*..*...*..*.
.**...*.*.**...*.*..
..*..**.*.*...**.**.
..*.*..*..*..*..*..*
.**.**....**..**..**
1.rle written

61172 clauses
**.*..**...**.**....
*.**..*.*...*.*.*.**
....**..*...*...*.*.
.**..*.*.**.*.*.*.*.
..**.*.*...*.**.*.**
*...*.*.**.*....*.*.
**.*..*...*.*.***..*
.*.*.*.***..**...**.
.*.*.*.*..**...*.*..
**.**.*..*...**.*..*
```

```
..*...*.**.**.*.*.**
..*.**.*..*.*.*.*...
**.*.*...*..*.*.*...
.*.*...*.**..*..***.
.*..****.*....**...*
..*.*...*..*...*..*.
.**..**.*.**...*.*..
*..*.*..*.*...**.**.
*..*.*.*..*..*..*..*
.**...*...**..**..**
2.rle written

...
```

This is most dense. Unfortunaly, it's impossible to construct "still life" with one mandatory *true* cell per each 2-cell chunk.

### 20.1.3 The source code

Source code and Wolfram Mathematica notebook: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/CA/GoL.

## 20.2 One-dimensional cellular automata and Z3 SMT-solver

Remember John Conway's Game of Life? It's a two-dimensional CA. This one is one-dimensional.

It was fully described in Stephen Wolfram's book "New Kind of Science". Including, the famous Rule 110: https://en.wikipedia.org/wiki/Rule_110.

Can we find *oscillators* (repeating states) and *gliders* (repeating and shifted states)?

N.B.: state is *wrapped*: the leftmost invisible cell is a rightmost one and vice versa.

The source code:

```
from z3 import *

WIDTH=15

def _try (RULE, STATES, WRAPPED, SHIFTED):

    rules=[]
    for i in range(8):
        if ((RULE>>i)&1)==1:
            rules.append(True)
        else:
            rules.append(False)

    rules=rules[::-1]
    #print "rules=", rules

    def f(a, b, c):
        return If(And(a==True, b==True, c==True), rules[7],
            If(And(a==True, b==True, c==False), rules[6],
            If(And(a==True, b==False, c==True), rules[5],
            If(And(a==True, b==False, c==False), rules[4],
            If(And(a==False, b==True, c==True), rules[3],
            If(And(a==False, b==True, c==False), rules[2],
            If(And(a==False, b==False, c==True), rules[1],
            If(And(a==False, b==False, c==False), rules[0], False))))))))

    S=[[Bool("%d_%d" % (s, i)) for i in range(WIDTH)] for s in range(STATES)]
```

```
s=Solver()

if WRAPPED==False:
    for st in range(STATES):
       s.add(S[st][0]==False)
       s.add(S[st][WIDTH-1]==False)

#s.add(S[0][15]==True)

if WRAPPED==False:
    for st in range(1,STATES):
        for i in range(1, WIDTH-1):
            s.add(S[st][i] == f(S[st-1][i-1], S[st-1][i], S[st-1][i+1]))
else:
    for st in range(1,STATES):
        for i in range(WIDTH):
            s.add(S[st][i] == f(S[st-1][(i-1) % WIDTH], S[st-1][i], S[st-1][(i+1) %
               WIDTH]))

def is_empty(st):
    t=[]
    for i in range(WIDTH):
        t.append(S[st][i]==False)
    return And(*t)

def is_full(st):
    t=[]
    for i in range(WIDTH):
        t.append(S[st][i]==True)
    return And(*t)

def non_equal_states (st1, st2):
    t=[]
    for i in range(WIDTH):
        t.append(S[st1][i] != S[st2][i])
    return Or(*t)

#s.add(non_equal_states(0, 1))

for st in range(STATES):
    s.add(is_empty(st)==False)
    s.add(is_full(st)==False)

# first and last states are equal to each other:
if WRAPPED==False:
    for i in range(1,WIDTH-1):
        if SHIFTED==0:
            s.add(S[0][i]==S[STATES-1][i])
        if SHIFTED==1:
            s.add(S[0][i]==S[STATES-1][i-1])
        if SHIFTED==2:
            s.add(S[0][i]==S[STATES-1][i+1])
else:
    for i in range(WIDTH):
        if SHIFTED==0:
```

```
                s.add(S[0][i]==S[STATES-1][i % WIDTH])
            if SHIFTED==1:
                s.add(S[0][i]==S[STATES-1][(i-1) % WIDTH])
            if SHIFTED==2:
                s.add(S[0][i]==S[STATES-1][(i+1) % WIDTH])

    if s.check()==unsat:
        return
        #print "unsat"
        #exit(0)

    m=s.model()

    print "RULE=%d STATES=%d, WRAPPED=%s, SHIFTED=%d" % (RULE, STATES, str(WRAPPED),
        SHIFTED)
    for st in range(STATES):
        t=""
        for i in range(WIDTH):
            if str(m[S[st][i]])=="False":
                t=t+"."
            else:
                t=t+"*"
        print t

for RULE in range(0, 256):
    for STATES in range(2, 10):
        if True:
        #for WRAPPED in [False, True]:
            WRAPPED=True
            for SHIFTED in [0,1,2]:
                _try (RULE, STATES, WRAPPED, SHIFTED)
```

Some *oscillators* and *gliders* are nice:

```
RULE=26 STATES=7, WRAPPED=True, SHIFTED=1
.***..***..***.
.*.**.*.**.*.**
...**...**...**
*..***..***..**
**.*.**.*.**.*.
**...**...**...
***..***..***..

RULE=29 STATES=3, WRAPPED=True, SHIFTED=1
..*..*..*..*..*
*..*..*..*..*..
.*..*..*..*..*.

RULE=30 STATES=4, WRAPPED=True, SHIFTED=1
**..***..***..*
.**.*.**.*.**.*
****.****.****.
*..***..***..**

RULE=38 STATES=4, WRAPPED=True, SHIFTED=0
*.**.**.**.**.*
**.**.**.**.**.
```

```
.**.**.**.**.**
*.**.**.**.**.*

RULE=40 STATES=5, WRAPPED=True, SHIFTED=1
*...**...**...*
.*....*....*...
.**...**...**..
...*....*....*.
...**...**...**

RULE=41 STATES=5, WRAPPED=True, SHIFTED=1
...*....*....*.
...**...**...**
*....*....*....
**...**...**...
..*....*....*..

RULE=42 STATES=3, WRAPPED=True, SHIFTED=2
*..*..*..*..*..
**.**.**.**.**.
.*..*..*..*..*.

RULE=42 STATES=5, WRAPPED=True, SHIFTED=2
.*.*..*.*..*.*.
.*.**.*.**.*.**
.*..*.*..*.*..*
.**.*.**.*.**.*
..*.*..*.*..*.*

RULE=43 STATES=7, WRAPPED=True, SHIFTED=0
**.**.**.**.**.
.*..*..*..*..*.
.**.**.**.**.**
..*..*..*..*..*
*.**.**.**.**.*
*..*..*..*..*..
**.**.**.**.**.

RULE=44 STATES=3, WRAPPED=True, SHIFTED=1
.**.**.**.**.**
*..*..*..*..*..
**.**.**.**.**.

RULE=44 STATES=5, WRAPPED=True, SHIFTED=1
*...**...**...*
.*....*....*...
.**...**...**..
...*....*....*.
...**...**...**

RULE=45 STATES=3, WRAPPED=True, SHIFTED=1
*..*..*..*..*..
**.**.**.**.**.
..*..*..*..*..*

RULE=60 STATES=4, WRAPPED=True, SHIFTED=1
```

```
.*.*..*.*..*.*.
.****.****.****
**...**...**...
*.*..*.*..*.*..

RULE=60 STATES=5, WRAPPED=True, SHIFTED=2
*..*...****.*.*
.*.**..*...****
****.*.**..*...
*...****.*.**..
**..*...****.*.

RULE=72 STATES=3, WRAPPED=True, SHIFTED=0
**..*.*..*.*...
..**...**...*.*
**..*.*..*.*...

RULE=73 STATES=3, WRAPPED=True, SHIFTED=0
.**..*.*..*.*..
*..**...**...*.
.**..*.*..*.*..

RULE=74 STATES=5, WRAPPED=True, SHIFTED=1
..*....*....*..
.*.*..*.*..*.*.
*...**...**...*
**.*.**.*.**.*.
.*....*....*...

RULE=75 STATES=3, WRAPPED=True, SHIFTED=1
..*..*..*..*..*
**.**.**.**.**.
.*..*..*..*..*.

RULE=76 STATES=3, WRAPPED=True, SHIFTED=0
*.**.**.*.*.*..
.*..*..*.*.*.**
*.**.**.*.*.*..

RULE=78 STATES=3, WRAPPED=True, SHIFTED=1
..**.***..**.**
**.**..***.**.*
.**.***..**.**.

RULE=82 STATES=7, WRAPPED=True, SHIFTED=2
*.**.*.**.*.**.
..**...**...**.
.***..***..***.
**.*.**.*.**.*.
**...**...**...
**..***..***..*
.*.**.*.**.*.**

RULE=90 STATES=4, WRAPPED=True, SHIFTED=0
..*.**..**.*...
.*..******..*..
```

```
*.***....***.*.
..*.**..**.*...

RULE=98 STATES=3, WRAPPED=True, SHIFTED=0
*.*.**.*.**.*.*
*.*..*.*..*.*..
*.*.**.*.**.*.*

RULE=100 STATES=3, WRAPPED=True, SHIFTED=2
*.**.**.**.**.*
.*..*..*..*..*.
**.**.**.**.**.

RULE=101 STATES=3, WRAPPED=True, SHIFTED=2
*..*..*..*..*..
*.**.**.**.**.*
.*..*..*..*..*.

RULE=102 STATES=4, WRAPPED=True, SHIFTED=2
***.****.****.*
..**...**...**.
.*.*..*.*..*.*.
****.****.****.

RULE=102 STATES=6, WRAPPED=True, SHIFTED=0
***.*.**..*...*
..****.*.**..*.
.*...****.*.**.
**..*...****.*.
.*.**..*...****
***.*.**..*...*

RULE=105 STATES=4, WRAPPED=True, SHIFTED=0
.**...**...**..
*..*.*..*.*..*.
****.****.****.
.**...**...**..

RULE=105 STATES=6, WRAPPED=True, SHIFTED=0
.*.*...**..**..
**.**.*..**..*.
......***..***.
.....*.*.**.*.*
*...**.*....*.*
.*.*...**..**..

RULE=105 STATES=7, WRAPPED=True, SHIFTED=0
*....*....*....
**..***..***..*
*.**.*.**.*.**.
*....*....*....
**..***..***..*
*.**.*.**.*.**.
*....*....*....

RULE=106 STATES=3, WRAPPED=True, SHIFTED=1
```

```
***..***..***..
..***..***..***
**..***..***..*

RULE=106 STATES=6, WRAPPED=True, SHIFTED=0
***..*.**.*...*
..****..*.**.*.
.*...****..*.**
.**.*...****..*
..*.**.*...****
***..*.**.*...*

RULE=106 STATES=8, WRAPPED=True, SHIFTED=0
.*.**...**..**.
**..**.*.***.**
.***.*.*...*...
*..*.*.**.***..
****.*..*...***
...*.*****.*...
..**.....*.**..
.*.**...**..**.

RULE=108 STATES=5, WRAPPED=True, SHIFTED=0
*****.***.***.*
.....*...*...*.
....***.***.***
*..*...*...*...
*****.***.***.*

RULE=108 STATES=9, WRAPPED=True, SHIFTED=0
.*..*...*.*....
******.*****...
......*.....*.*
*....***...****
.*..*...*.*....
******.*****...
......*.....*.*
*....***...****
.*..*...*.*....

RULE=109 STATES=3, WRAPPED=True, SHIFTED=0
***.*.**..***..
.*.***..**.*.**
***.*.**..***..

RULE=110 STATES=3, WRAPPED=True, SHIFTED=1
**.**.**..***.*
.**.**.***..**.
*.**.**..***.**

RULE=110 STATES=4, WRAPPED=True, SHIFTED=0
*...**.****...*
**.*.**...**.*.
.****.**.*.****
*...**.****...*
```

```
RULE=110 STATES=7, WRAPPED=True, SHIFTED=0
*.*.**...**...*
****.**.*.**.*.
...**.****.****
*.*.**...**...*
****.**.*.**.*.
...**.****.****
*.*.**...**...*

RULE=120 STATES=8, WRAPPED=True, SHIFTED=0
*.**.*.*..*..**
..*..*.*******.
.*****.*......*
.*.....**....**
.**...**.*..**.
**.*.**..****.*
...*.*.***....*
*.**.*.*..*..**

RULE=122 STATES=7, WRAPPED=True, SHIFTED=0
*****.**.**.*.*
....*.**.**.*.*
*..**.**.**.*.*
*****.**.**.*.*
....*.**.**.*.*
*..**.**.**.*.*
*****.**.**.*.*

RULE=124 STATES=4, WRAPPED=True, SHIFTED=0
.**....*..*****
**.*..*****....
*.*****....*..*
.**....*..*****

RULE=124 STATES=7, WRAPPED=True, SHIFTED=0
..**.*****....*
***.**....*..**
...**.*..*****.
..**.*****....*
***.**....*..**
...**.*..*****.
..**.*****....*

RULE=128 STATES=3, WRAPPED=True, SHIFTED=0
...*...*....***
.*...*...**....
...*...*....***

RULE=129 STATES=3, WRAPPED=True, SHIFTED=0
..*....**....*.
*...**....**...
..*....**....*.

RULE=134 STATES=4, WRAPPED=True, SHIFTED=1
...*....*....*.
**...**...**...
```

```
.*.*..*.*..*.*.
..*....*....*..

RULE=135 STATES=4, WRAPPED=True, SHIFTED=1
..**...**...**.
*..*.*..*.*..*.
....*....*....*
.**...**...**..

RULE=137 STATES=4, WRAPPED=True, SHIFTED=0
*.....*..*..*..
.****..*..*..*.
..**.*..*..*..*
*.....*..*..*..

RULE=148 STATES=4, WRAPPED=True, SHIFTED=2
.*.*..*.*..*.*.
..*....*....*..
*...**...**...*
..*.*..*.*..*.*

RULE=149 STATES=4, WRAPPED=True, SHIFTED=2
*...**...**...*
..*.*..*.*..*.*
...*....*....*.
**...**...**...

RULE=150 STATES=7, WRAPPED=True, SHIFTED=0
**.****.****.**
.***..***..***.
.*.*..*.*..*.*.
..*....*....*..
*...**...**...*
*.*.**.*.**.*.*
**.****.****.**

RULE=154 STATES=7, WRAPPED=True, SHIFTED=1
.*.**.*.**.*.**
...**...**...**
**.****.****.**
.*.*..*.*..*.*.
....*....*....*
***..***..***..
*.**.*.**.*.**.

RULE=166 STATES=7, WRAPPED=True, SHIFTED=1
*.*.**.*.**.*.*
****.****.****.
...**...**...**
.*..*.*..*.*..*
**..***..***..*
.*....*....*...
.*.**.*.**.*.**

RULE=167 STATES=7, WRAPPED=True, SHIFTED=1
*.*..*.*..*.*..
```

```
***..***..***..
.**...**...**..
..*.*..*.*..*.*
..***..***..***
...**...**...**
.*..*.*..*.*..*

RULE=169 STATES=6, WRAPPED=True, SHIFTED=0
..**.*..*.***..
*....**.*..*.**
.***....**.*..*
..*.***....**.*
*.*..*.***....*
..**.*..*.***..

RULE=169 STATES=8, WRAPPED=True, SHIFTED=0
.*.*.***.****..
.*.*..*...**.**
.*.**.***......
.*.....*.******
.*****.*..****.
..***..**..**.*
*..*.*...*....*
.*.*.***.****..

RULE=182 STATES=6, WRAPPED=True, SHIFTED=0
*..*.*****.*..*
*..***...***..*
*..*.*.*.*.*..*
*..*********..*
*..*.......*..*
*..*.*****.*..*

RULE=188 STATES=7, WRAPPED=True, SHIFTED=2
.**.*.**.*.**.*
**.****.****.**
..**...**...**.
*.*.**.*.**.*.*
.****.****.****
**...**...**...
*.**.*.**.*.**.

RULE=193 STATES=4, WRAPPED=True, SHIFTED=0
***.**..****..*
**.....*.**..*.
...****.....*..
***.**..****..*

RULE=195 STATES=5, WRAPPED=True, SHIFTED=2
*.***....*.*..*
*..**.***....*.
..*.*..**.***..
**....*.*..**.*
**.***....*.*..

RULE=195 STATES=6, WRAPPED=True, SHIFTED=0
```

```
*..*.*....***.*
*.*....***.**..
....***.**..*.*
.***.**..*.*...
*.**..*.*....**
*..*.*....***.*

RULE=201 STATES=5, WRAPPED=True, SHIFTED=0
......***...***
******.*.***.*.
.****.....*....
*.**.*****.****
......***...***

RULE=201 STATES=9, WRAPPED=True, SHIFTED=0
..****.....*...
**.**.*****.***
*......***...**
.******.*.***.*
..****.....*...
**.**.*****.***
*......***...**
.******.*.***.*
..****.....*...

RULE=210 STATES=7, WRAPPED=True, SHIFTED=2
*....*....*....
..***..***..***
.**.*.**.*.**.*
.**...**...**..
***.****.****.*
..*.*..*.*..*.*
.*....*....*...

RULE=225 STATES=8, WRAPPED=True, SHIFTED=0
.**.*.......***
....*.******.*.
*****..****..*.
.***..*.**..**.
*.*..**....*...
*.*.*...****.**
..*.*.**.**...*
.**.*.......***
```

SHIFTED=0 means *oscillator*, SHIFTED=1 means *glider* slipping left, SHIFTED=2 — slipping right.

# 21 Everything else

## 21.1 Ménage problem

```
In combinatorial mathematics, the ménage problem or problème des ménages[1] asks for the
    number of different ways in which it is possible to seat a set of male-female
   couples at a dining table so that men and women alternate and nobody sits next to
   his or her partner. This problem was formulated in 1891 by Édouard Lucas and
   independently, a few years earlier, by Peter Guthrie Tait in connection with knot
```

```
theory.[2] For a number of couples equal to 3, 4, 5, ...  the number of seating
arrangements is

 12, 96, 3120, 115200, 5836320, 382072320, 31488549120, ... (sequence A059375 in the
    OEIS).
```

( Wikipedia. )

We can count it using Z3, but also get actual men/women allocations:

```
from z3 import *

COUPLES=3

# a pair each men and women related to:
men=[Int('men_%d' % i) for i in range(COUPLES)]
women=[Int('women_%d' % i) for i in range(COUPLES)]

# men and women are placed around table like this:

# m m m
#  w w w

# i.e., women[0] is placed between men[0] and men[1]
# the last women[COUPLES-1] is between men[COUPLES-1] and men[0] (wrapping)

s=Solver()
s.add(Distinct(men))
s.add(Distinct(women))

[s.add(And(men[i]>=0, men[i]<COUPLES)) for i in range (COUPLES)]
[s.add(And(women[i]>=0, women[i]<COUPLES)) for i in range (COUPLES)]

# a pair, each woman belong to, cannot be the same as men's located at left and right.
# "% COUPLES" is wrapping, so that the last woman is between the last man and the first
    man.
for i in range(COUPLES):
    s.add(And(women[i]!=men[i], women[i]!=men[(i+1) % COUPLES]))

def print_model(m):
    print "  men",
    for i in range(COUPLES):
        print m[men[i]],
    print ""

    print "women ",

    for i in range(COUPLES):
        print m[women[i]],
    print ""
    print ""

results=[]

# enumerate all possible solutions:
while True:
    if s.check() == sat:
        m = s.model()
```

```
        print_model(m)
        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "results total=", len(results)
        print "however, according to https://oeis.org/A059375 :", len(results)*2
        break
```

( https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/other/menage/menage.py )

For 3 couples:

```
  men 0 2 1
women  1 0 2

  men 1 2 0
women  0 1 2

  men 0 1 2
women  2 0 1

  men 2 1 0
women  0 2 1

  men 2 0 1
women  1 2 0

  men 1 0 2
women  2 1 0

results total= 6
however, according to https://oeis.org/A059375 : 12
```

We are getting “half” of results because men and women can be then swapped (their sex swapped (or reassigned)) and you’ve got another 6 results. 6+6=12 in total. This is kind of symmetry.

For 4 couples:

```

...

  men 3 0 2 1
women  1 3 0 2

  men 3 0 1 2
women  2 3 0 1

  men 1 0 2 3
women  3 1 0 2

  men 2 0 1 3
women  3 2 0 1

results total= 48
however, according to https://oeis.org/A059375 : 96
```

For 5 couples:

```
...

  men 0 4 1 2 3
women  1 3 0 4 2

  men 0 3 1 2 4
women  1 4 0 3 2

  men 0 3 1 2 4
women  1 0 4 3 2

  men 4 3 1 0 2
women  0 2 4 1 3

results total= 1560
however, according to https://oeis.org/A059375 : 3120
```

## 21.2 Dependency graphs and topological sorting

Topological sorting is an operation many programmers well familiar with: this is what "make" tool do when it find an order of items to process. Items not dependent of anything can be processed first. The most dependent items at the end.

Dependency graph is a graph and topological sorting is such a "contortion" of the a graph, when you can see an order of items.

For example, let's create a sample graph in Wolfram Mathematica:

```
In[]:= g = Graph[{7 -> 1, 7 -> 0, 5 -> 1, 3 -> 0, 3 -> 4, 1 -> 2, 1 -> 6,
   1 -> 4, 0 -> 6}, VertexLabels -> "Name"]
```

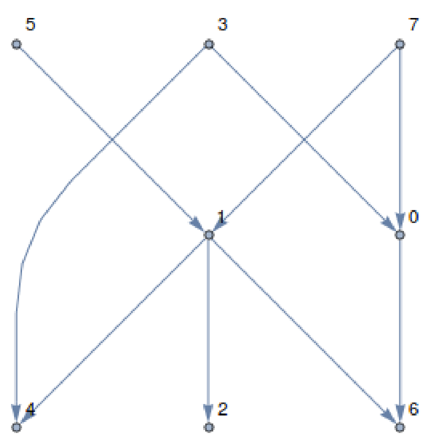



Figure 44:

Each arrow shows that an item is needed by an item arrow pointing to, i.e., if "a -> b", then item "a" must be first processed, because "b" needs it, or "b" depends on "a".

How Mathematica would "sort" the dependency graph?

```
In[]:= TopologicalSort[g]
Out[]= {7, 3, 0, 5, 1, 4, 6, 2}
```

So you're going to process item 7, then 3, 0, and 2 at the very end.

The algorithm in the Wikipedia article is probably used in the "make" and whatever IDE you use for building your code.

Also, many UNIX platforms had separate "tsort" utility: https://en.wikipedia.org/wiki/Tsort.

How would "tsort" sort the graph? I'm making the text file with input data:

```
7 1
7 0
5 1
3 0
3 4
1 2
1 6
1 4
0 6
```

And run tsort:

```
 % tsort tst
3
5
7
0
1
4
6
2
```

Now I'll use Z3 SMT-solver for topological sort, which is overkill, but quite spectacular: all we need to do is to add constraint for each edge (or "connection") in graph, if "a -> b", then "a" must be less then "b", where each variable reflects ordering.

```
from MK85 import *

TOTAL=8

s=MK85()

order=[s.BitVec(("%d" % i), 4) for i in range(TOTAL)]

s.add(s.Distinct(order))

for i in range(TOTAL):
    s.add(And(order[i]>=0, order[i]<TOTAL))

s.add(order[5]<order[1])

s.add(order[3]<order[4])
s.add(order[3]<order[0])

s.add(order[7]<order[0])
s.add(order[7]<order[1])

s.add(order[1]<order[2])
s.add(order[1]<order[4])
s.add(order[1]<order[6])

s.add(order[0]<order[6])
```

```
print s.check()

m=s.model()

order_to_print=[None]*(TOTAL)
for i in range(TOTAL):
    order_to_print[m[str(i)]]=i

print order_to_print
```

Almost the same result, but also correct:

```
True
[5, 7, 1, 3, 0, 4, 6, 2]
```

The solution using Z3: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/other/tsort/tsort_Z3.py.

Yet another demonstration of topological sort: "less than" relation would indicate, who is whose boss, and who is whose "yes man". The resulting order after sorting is then represent how good each position in social hierarchy is.

## 21.3 Package manager and Z3

Here is simplified example. We have libA, libB, libC and libD, available in various versions (and flavors). We're going to install programA and programB, which use these libraries.

```
#!/usr/bin/env python

from z3 import *

s=Optimize()

libA=Int('libA')
# libA's version is 1..5 or 999 (which means library will not be installed):
s.add(Or(And(libA>=1, libA<=5),libA==999))

libB=Int('libB')
# libB's version is 1, 4, 5 or 999:
s.add(Or(libB==1, libB==4, libB==5, libB==999))

libC=Int('libC')
# libC's version is 10, 11, 14 or 999:
s.add(Or(libC==10, libC==11, libC==14, libC==999))

# libC is dependent on libA
# libC v10 is dependent on libA v1..3, but not newer
# libC v11 requires at least libA v3
# libC v14 requires at least libA v5
s.add(If(libC==10, And(libA>=1, libA<=3), True))
s.add(If(libC==11, libA>=3, True))
s.add(If(libC==14, libA>=5, True))

libD=Int('libD')
# libD's version is 1..10
s.add(Or(And(libD>=1, libD<=10),libD==999))

programA=Int('programA')
# programA came as v1 or v2:
s.add(Or(programA==1, programA==2))
```

```
# programA is dependent on libA, libB and libC
# programA v1 requires libA v2 (only this version), libB v4 or v5, libC v10:
s.add(If(programA==1, And(libA==2, Or(libB==4, libB==5), libC==10), True))
# programA v2 requires these libraries: libA v3, libB v5, libC v11
s.add(If(programA==2, And(libA==3, libB==5, libC==11), True))

programB=Int('programB')
# programB came as v7 or v8:
s.add(Or(programB==7, programB==8))

# programB v7 requires libA at least v2 and libC at least v10:
s.add(If(programB==7, And(libA>=2, libC>=10), True))
# programB v8 requires libA at least v6 and libC at least v11:
s.add(If(programB==8, And(libA>=6, libC>=11), True))

s.add(programA==1)
s.add(programB==7) # change this to 8 to make it unsat

# we want latest libraries' versions.
# if the library is not required, its version is "pulled up" to 999,
# and 999 means the library is not needed to be installed
s.maximize(Sum(libA,libB,libC,libD))

print s.check()
print s.model()
```

( The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/other/dep/dependency.py )

The output:

```
sat
[libB = 5,
 libD = 999,
 libC = 10,
 programB = 7,
 programA = 1,
 libA = 2]
```

999 means that there is no need to install libD, it's not required by other packages.

Change version of ProgramB to v8 and it will says "unsat", meaning, there is a conflict: ProgramA requires libA v2, but ProgramB v8 eventually requires newer libA.

Still, there is a work to do: "unsat" message is somewhat useless to end user, some information about conflicting items should be printed.

Here is my another optimization problem example: 12.1.

More about using SAT/SMT solvers in package managers: https://research.swtch.com/version-sat, https://cseweb.ucsd.edu/~lerner/papers/opium.pdf.

Now in the opposite direction: forcing aptitude package manager to solve Sudoku: http://web.archive.org/web/20160326062818/http://algebraicthunk.net/~dburrows/blog/entry/package-management-s

Some readers may ask, how to order libraries/programs/packages to be installed? This is simpler problem, which is often solved by topological sorting. The algorithm reorders graph in such a way so that vertices not depended on anything will be on the top of queue. Next, there will be vertices dependend on vertices from the previous layer. And so on.

*make* UNIX utility does this while constructing order of items to be processed. Even more: older *make* utilities offloaded the job to the external utility (*tsort*). Some older UNIX has it, at least some versions of NetBSD [138].

[138] http://netbsd.gw.com/cgi-bin/man-cgi/man?tsort+1+NetBSD-current

### 21.4 Knight's tour

```
from z3 import *
import pprint, math

SIZE=8
#closed=False
closed=True
# find King's tour instead of Knight's. just for demonstration
#king_tour=True
king_tour=False

def coord_to_idx(r, c):
    if r<0 or c<0:
        return None
    if r>=SIZE or c>=SIZE:
        return None
    return r*SIZE+c

"""
knight's movements:

. x . x . . . .
x . . . x . . .
. . o . . . . .
x . . . x . . .
. x . x . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .

"""
G={}

if king_tour:
# King's tour
    for r in range(SIZE):
        for c in range(SIZE):
            _from=coord_to_idx(r, c)
            _to=[]
            _to.append(coord_to_idx(r-1, c-1))
            _to.append(coord_to_idx(r-1, c+0))
            _to.append(coord_to_idx(r-1, c+1))
            _to.append(coord_to_idx(r-0, c-1))
            _to.append(coord_to_idx(r-0, c+1))
            _to.append(coord_to_idx(r+1, c-1))
            _to.append(coord_to_idx(r+1, c+0))
            _to.append(coord_to_idx(r+1, c+1))
            # remove "None" elements (moves beyond physical board):
            _to=filter(lambda x: x!=None, _to)
            G[_from]=_to
else:
# Knight's tour
    for r in range(SIZE):
        for c in range(SIZE):
            _from=coord_to_idx(r, c)
```

```
            _to=[]
            _to.append(coord_to_idx(r-2, c-1))
            _to.append(coord_to_idx(r-2, c+1))
            _to.append(coord_to_idx(r-1, c-2))
            _to.append(coord_to_idx(r-1, c+2))
            _to.append(coord_to_idx(r+1, c-2))
            _to.append(coord_to_idx(r+1, c+2))
            _to.append(coord_to_idx(r+2, c-1))
            _to.append(coord_to_idx(r+2, c+1))
            # remove "None" elements (moves beyond physical board):
            _to=filter(lambda x: x!=None, _to)
            G[_from]=_to

pp = pprint.PrettyPrinter(indent=4)
pp.pprint(G)

s=Solver()

L=len(G)
# we use one-hot (or unitary) variables, thus we eliminate the need of using adding +
    remainder
# as in https://github.com/Z3Prover/z3/blob/master/examples/python/hamiltonian/
    hamiltonian.py
V=[BitVec('V_%d' % i, L) for i in range(L)]

# on closed tour, we may omit this constraint, SAT/SMT solver got to know this is one-
    hot/unitary variable!
if closed==False:
    # without: faster on closed tours
    for v in range(L):
        or_list=[]
        for i in range(L):
            or_list.append(V[v]==2**i)
        s.add(Or(*or_list))

s.add(Distinct(V))

# first cell:
s.add(V[0]==BitVecVal(1, L))

# can create expression like:
# If(selector=c1, val[0],
# If(selector=c2, val[1],
# If(selector=c3, val[2],
# If(selector=c4, val[3], val[4]))))
def MUX(selector, selectors, vals):
    assert len(selectors)+1 == len(vals)
    l=len(vals)
    t=vals[0]
    for i in range(l-1):
        t=If(selector==selectors[i], vals[i+1], t)
    return t

for i in range(L):
    if closed==False and i==0:
        continue
```

```
    or_list=[]
    for j in G[i]:
        or_list.append(RotateLeft(V[j], 1))
    sel=Int('sel%d' % i)
    # no idea why, but using multiplexer is faster than chain of Or's as in
    # https://github.com/Z3Prover/z3/blob/master/examples/python/hamiltonian/hamiltonian
       .py
    e=MUX(sel, range(len(or_list)-1), or_list)
    """
    at this point e can look like:

    54 If(sel54 == 2,
       RotateLeft(V_60, 1),
       If(sel54 == 1,
          RotateLeft(V_44, 1),
          If(sel54 == 0,
             RotateLeft(V_39, 1),
             RotateLeft(V_37, 1))))

    selector is not used at all
    """
    #print i, e
    s.add(V[i]==e)

if s.check()==unsat:
    print "unsat"
    exit(0)
m=s.model()
#print m

print ""
for r in range(SIZE):
    for c in range(SIZE):
        t=coord_to_idx(r, c)
        print ("%2d" % int(math.log(m[V[t]].as_long(), 2))),
    print ""
```

Can find a closed knight's tour on 8*8 chess board for 150s on Intel Quad-Core Xeon E3-1220 3.10GHz:

```
 0 57 44 41  2 39 12 29
43 46  1 58 11 30 23 38
56 63 42 45 40  3 28 13
47  8 59 10 31 24 37 22
60 55 62 51  4 27 14 25
 7 48  9 32 17 34 21 36
54 61 50  5 52 19 26 15
49  6 53 18 33 16 35 20
```

However, this is WAY slower than C implementation on Rosetta Code: https://rosettacode.org/wiki/Knight%27s_tour#C ... which uses Warnsdorf's rule: https://en.wikipedia.org/wiki/Knight%27s_tour#Warnsdorff.27s_algorithm.

Another program for Z3 for finding Hamiltonian cycle: https://github.com/Z3Prover/z3/blob/master/examples/python/hamiltonian/hamiltonian.py. (Clever trick of using remainder.)

## 21.5 Stable marriage problem

See also in Wikipedia and Rosetta code.

Layman's explanation in Russian: https://lenta.ru/articles/2012/10/15/nobel/.

My solution is much less efficient, because much simpler/better algorithm exists (Gale/Shapley algorithm), but I did it to demonstrate the essence of the problem plus as a yet another SMT-solvers and Z3 demonstration.

See comments:

```
#!/usr/bin/env python

from z3 import *

SIZE=10

# names and preferences has been copypasted from https://rosettacode.org/wiki/
   Stable_marriage_problem

# males:
abe, bob, col, dan, ed, fred, gav, hal, ian, jon = 0,1,2,3,4,5,6,7,8,9
MenStr=["abe", "bob", "col", "dan", "ed", "fred", "gav", "hal", "ian", "jon"]
# females:
abi, bea, cath, dee, eve, fay, gay, hope, ivy, jan = 0,1,2,3,4,5,6,7,8,9
WomenStr=["abi", "bea", "cath", "dee", "eve", "fay", "gay", "hope", "ivy", "jan"]

# men's preferences. better is at left (at first):
ManPrefer={}
ManPrefer[abe]=[abi, eve, cath, ivy, jan, dee, fay, bea, hope, gay]
ManPrefer[bob]=[cath, hope, abi, dee, eve, fay, bea, jan, ivy, gay]
ManPrefer[col]=[hope, eve, abi, dee, bea, fay, ivy, gay, cath, jan]
ManPrefer[dan]=[ivy, fay, dee, gay, hope, eve, jan, bea, cath, abi]
ManPrefer[ed]=[jan, dee, bea, cath, fay, eve, abi, ivy, hope, gay]
ManPrefer[fred]=[bea, abi, dee, gay, eve, ivy, cath, jan, hope, fay]
ManPrefer[gav]=[gay, eve, ivy, bea, cath, abi, dee, hope, jan, fay]
ManPrefer[hal]=[abi, eve, hope, fay, ivy, cath, jan, bea, gay, dee]
ManPrefer[ian]=[hope, cath, dee, gay, bea, abi, fay, ivy, jan, eve]
ManPrefer[jon]=[abi, fay, jan, gay, eve, bea, dee, cath, ivy, hope]

# women's preferences:
WomanPrefer={}
WomanPrefer[abi]=[bob, fred, jon, gav, ian, abe, dan, ed, col, hal]
WomanPrefer[bea]=[bob, abe, col, fred, gav, dan, ian, ed, jon, hal]
WomanPrefer[cath]=[fred, bob, ed, gav, hal, col, ian, abe, dan, jon]
WomanPrefer[dee]=[fred, jon, col, abe, ian, hal, gav, dan, bob, ed]
WomanPrefer[eve]=[jon, hal, fred, dan, abe, gav, col, ed, ian, bob]
WomanPrefer[fay]=[bob, abe, ed, ian, jon, dan, fred, gav, col, hal]
WomanPrefer[gay]=[jon, gav, hal, fred, bob, abe, col, ed, dan, ian]
WomanPrefer[hope]=[gav, jon, bob, abe, ian, dan, hal, ed, col, fred]
WomanPrefer[ivy]=[ian, col, hal, gav, fred, bob, abe, ed, jon, dan]
WomanPrefer[jan]=[ed, hal, gav, abe, bob, jon, col, ian, fred, dan]

s=Solver()

ManChoice=[Int('ManChoice_%d' % i) for  i in range(SIZE)]
WomanChoice=[Int('WomanChoice_%d' % i) for  i in range(SIZE)]

# all values in ManChoice[]/WomanChoice[] are in 0..9 range:
for i in range(SIZE):
    s.add(And(ManChoice[i]>=0, ManChoice[i]<=9))
    s.add(And(WomanChoice[i]>=0, WomanChoice[i]<=9))

s.add(Distinct(ManChoice))
```

```
# "inverted index", make sure all men and women are "connected" to each other, i.e.,
    form pairs.
# FIXME: only work for SIZE=10
for i in range(SIZE):
    s.add(WomanChoice[i]==
        If(ManChoice[0]==i, 0,
        If(ManChoice[1]==i, 1,
        If(ManChoice[2]==i, 2,
        If(ManChoice[3]==i, 3,
        If(ManChoice[4]==i, 4,
        If(ManChoice[5]==i, 5,
        If(ManChoice[6]==i, 6,
        If(ManChoice[7]==i, 7,
        If(ManChoice[8]==i, 8,
        If(ManChoice[9]==i, 9, -1)))))))))))

# this is like ManChoice[] value, but "inverted index". it reflects wife's rating in man
    's own rating system.
# 0 if he married best women, 1 if there is 1 women who he would prefer (if there is a
    chance):
ManChoiceInOwnRating=[Int('ManChoiceInOwnRating_%d' % i) for  i in range(SIZE)]
# same for all women:
WomanChoiceInOwnRating=[Int('WomanChoiceInOwnRating_%d' % i) for  i in range(SIZE)]

# set values in "inverted" indices according to values in ManPrefer[]/WomenPrefer[].
# FIXME: only work for SIZE=10
for m in range(SIZE):
    s.add (ManChoiceInOwnRating[m]==
       If(ManChoice[m]==ManPrefer[m][0],0,
       If(ManChoice[m]==ManPrefer[m][1],1,
       If(ManChoice[m]==ManPrefer[m][2],2,
       If(ManChoice[m]==ManPrefer[m][3],3,
       If(ManChoice[m]==ManPrefer[m][4],4,
       If(ManChoice[m]==ManPrefer[m][5],5,
       If(ManChoice[m]==ManPrefer[m][6],6,
       If(ManChoice[m]==ManPrefer[m][7],7,
       If(ManChoice[m]==ManPrefer[m][8],8,
       If(ManChoice[m]==ManPrefer[m][9],9, -1)))))))))))

for w in range(SIZE):
    s.add (WomanChoiceInOwnRating[w]==
       If(WomanChoice[w]==WomanPrefer[w][0],0,
       If(WomanChoice[w]==WomanPrefer[w][1],1,
       If(WomanChoice[w]==WomanPrefer[w][2],2,
       If(WomanChoice[w]==WomanPrefer[w][3],3,
       If(WomanChoice[w]==WomanPrefer[w][4],4,
       If(WomanChoice[w]==WomanPrefer[w][5],5,
       If(WomanChoice[w]==WomanPrefer[w][6],6,
       If(WomanChoice[w]==WomanPrefer[w][7],7,
       If(WomanChoice[w]==WomanPrefer[w][8],8,
       If(WomanChoice[w]==WomanPrefer[w][9],9, -1)))))))))))

# the last part is the essence of this script:

# this is 2D bool array. "true" if a (married or already connected) man would prefer
```

```
    another women over his wife.
ManWouldPrefer=[[Bool('ManWouldPrefer_%d_%d' % (m, w)) for w in range(SIZE)] for m in
    range(SIZE)]
# same for all women:
WomanWouldPrefer=[[Bool('WomanWouldPrefer_%d_%d' % (w, m)) for m in range(SIZE)] for w
    in range(SIZE)]

# set "true" in ManWouldPrefer[][] table for all women who are better than the wife a
    man currently has.
# all others can be "false"
# if the man married best women, all entries would be "false"
for m in range(SIZE):
    for w in range(SIZE):
        s.add(ManWouldPrefer[m][w] == (ManPrefer[m].index(w) < ManChoiceInOwnRating[m]))

# do the same for WomanWouldPrefer[][]:
for w in range(SIZE):
    for m in range(SIZE):
        s.add(WomanWouldPrefer[w][m] == (WomanPrefer[w].index(m) <
            WomanChoiceInOwnRating[w]))

# this is the most important constraint.
# enumerate all possible man/woman pairs
# no pair can exist with both "true" in "mirrored" entries of ManWouldPrefer[][]/
    WomanWouldPrefer[][].
# we block this by the following constraint: Not(And(x,y)): all x/y values are allowed,
    except if both are set to 1/true:
for m in range(SIZE):
    for w in range(SIZE):
        s.add(Not(And(ManWouldPrefer[m][w], WomanWouldPrefer[w][m])))

print s.check()
mdl=s.model()

print ""

print "ManChoice:"
for m in range(SIZE):
    w=mdl[ManChoice[m]].as_long()
    print MenStr[m], "<->", WomenStr[w]

print ""

print "WomanChoice:"
for w in range(SIZE):
    m=mdl[WomanChoice[w]].as_long()
    print WomenStr[w], "<->", MenStr[m]
```

( The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/other/stable_marriage/stable.py )

Result is seems to be correct:

```
sat

ManChoice:
abe <-> ivy
bob <-> cath
```

```
col <-> dee
dan <-> fay
ed <-> jan
fred <-> bea
gav <-> gay
hal <-> eve
ian <-> hope
jon <-> abi

WomanChoice:
abi <-> jon
bea <-> fred
cath <-> bob
dee <-> col
eve <-> hal
fay <-> dan
gay <-> gav
hope <-> ian
ivy <-> abe
jan <-> ed
```

This is what we did in plain English language. "Connect men and women somehow, we don't care how. But no pair must exist of those who prefer each other (simultaneously) over their current spouses". Gale/Shapley algorithm uses "steps" to "stabilize" marriage. There are no "steps", all pairs are married couples already.

Another important thing to notice: only one solution must exist.

```
...

results=[]

# enumerate all possible solutions:
while True:
    if s.check() == sat:
        m = s.model()
        #print m
        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "results total=", len(results)
        break

...
```

( The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/other/stable_marriage/stable2.py )

That reports only 1 model available, which is correct indeed.

## 21.6 Tiling puzzle and Z3 SMT solver

This is classic problem: given 12 polyomino titles, cover mutilated chessboard with them (it has 60 squares with no central 4 squares).

The problem is covered at least in Donald E. Knuth - Dancing Links, and this Z3 solution has been inspired by it.

Another thing I've added: graph coloring. You see, my script gives correct solutions, but somewhat unpleasant visually. So I used colored pseudographics. There are 12 tiles, it's not a problem to assign 12 colors to them. But there is another

heavily used SAT problem: graph coloring.

Given a graph, assign a color to each vertex/node, so that colors wouldn't be equal in adjacent nodes. The problem can be solved easily in SMT: assign variable to each vertex. If two vertices are connected, add a constraint: *vertex1_color != vertex2_color*. As simple as that. In my case, each polynomio is vertex and if polyomino is adjacent to another polyomino, an edge/link is added between vertices. So I did, and output is now colored.

But this is planar graph (i.e., a graph which is, if represented in two-dimensional space has no intersected edges/links). And here is a famous four color theorem can be used. The solution of tiled polynomios is in fact like planar graph, or, a map, like a world map. Theorem states that any planar graph (or map) can be colored only 4 colors.

This is true, even more, several tilings can be colors with only 3 colors:

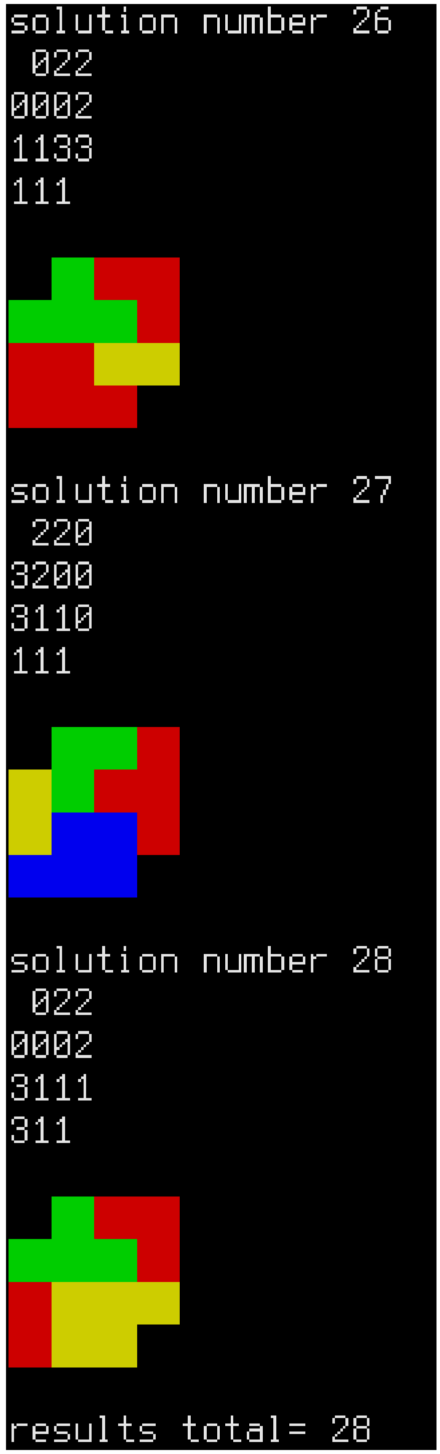
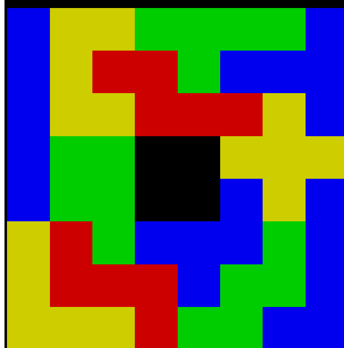



Figure 45:

Now the classic: 12 pentominos and "mutilated" chess board, several solutions:

```
211  999
211  094
7b100084
7bbb0884
777b8844
```



```
solution number 7
44445666
41555696
11bb5999
11b  092
abb  002
aa880072
a8833372
a8337772
```

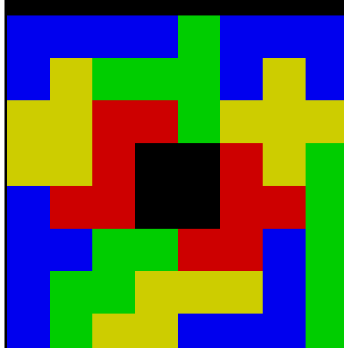


Figure 46:

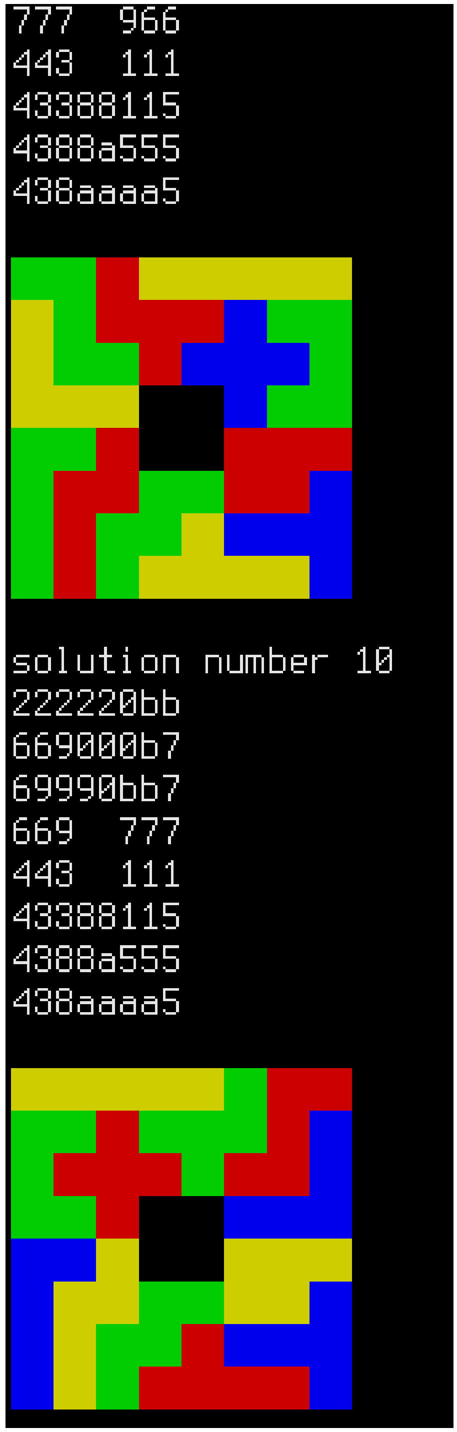



Figure 47:

The source code: https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/other/tiling/tiling.py.

Further reading: https://en.wikipedia.org/wiki/Exact_cover#Pentomino_tiling.

Four-color theorem has an interesting story, it has been finally proved in 2005 by Coq proof assistant: https://en.wikipedia.org/wiki/Four_color_theorem.

## 21.7 Hilbert's 10th problem, Fermat's last theorem and SMT solvers

Hilbert's 10th problem states that you cannot devise an algorithm which can solve any diophantine equation over integers. However, it's important to understand, that this is possible over fixed-size bitvectors.

Fermat's last theorem states that there are no integer solution(s) for $a^n + b^n = c^n$, for $n >= 3$.

Let's prove it for n=3 and for a in 0..255 range:

```
from z3 import *

# for a 8-bit bitvec, to prevent overflow during multiplication/addition, 25 bits must
   be allocated
# because log2(256)*3 = 8*3 = 24
# and a sum of two 24-bit bitvectors can be 25-bit
a,b,c = BitVecs ('a b c', 25)

s=Solver()

# only 8-bit values are allowed in these 3 bitvectors:
s.add((a&0xffff00)==0)
s.add((b&0xffff00)==0)
s.add((c&0xffff00)==0)

# non-zero values:
s.add(a!=0)
s.add(b!=0)
s.add(c!=0)

# a^3 + b^3 = c^3
s.add(a*a*a + b*b*b == c*c*c)

print s.check()
```

Z3 gives "unsat", meaning, it couldn't find any a/b/c. However, this is possible to check even using brute-force search.

If to replace "BitVecs" by "Ints", Z3 would give "unknown":

```
from z3 import *

a,b,c = Ints ('a b c')

s=Solver()

s.add(a!=0)
s.add(b!=0)
s.add(c!=0)

# a^3 + b^3 = c^3
s.add(a*a*a + b*b*b == c*c*c)

print s.check()
```

In short: anything is decidable (you can build an algorithm which can solve equation or not) under fixed-size bitvectors. Given enough computational power, you can solve such equations for big bit-vectors. But this is not possible for integers or bit-vectors of any size.

Another interesting reading about this by Leonardo de Moura: https://stackoverflow.com/questions/13898175/how-does-z3-handle-non-linear-integer-arithmetic.

# 22 Toy-level solvers

... which has been written by me and serves as a demonstration and playground.

## 22.1 Simplest SAT solver in ~120 lines

This is simplest possible backtracking SAT solver written in Python (not a DPLL[139] one). It uses the same backtracking algorithm you can find in many simple Sudoku and 8 queens solvers. It works significantly slower, but due to its extreme simplicity, it can also count solutions. For example, it can count all solutions of 8 queens problem (7.7).

Also, there are 70 solutions for POPCNT4 function [140] (the function is true if any 4 of its input 8 variables are true):

```
SAT
-1 -2 -3 -4 5 6 7 8 0
SAT
-1 -2 -3 4 -5 6 7 8 0
SAT
-1 -2 -3 4 5 -6 7 8 0
SAT
-1 -2 -3 4 5 6 -7 8 0
...

SAT
1 2 3 -4 -5 6 -7 -8 0
SAT
1 2 3 -4 5 -6 -7 -8 0
SAT
1 2 3 4 -5 -6 -7 -8 0
UNSAT
solutions= 70
```

It was also tested on my SAT-based Minesweeper cracker (3.7), and finishes in reasonable time (though, slower than MiniSat by a factor of ~10).

On bigger CNF instances, it gets stuck, though.

The source code:

```
#!/usr/bin/env python

count_solutions=True
#count_solutions=False

import sys

def read_text_file (fname):
    with open(fname) as f:
        content = f.readlines()
    return [x.strip() for x in content]

def read_DIMACS (fname):
    content=read_text_file(fname)

    header=content[0].split(" ")

    assert header[0]=="p" and header[1]=="cnf"
    variables_total, clauses_total = int(header[2]), int(header[3])

    # array idx=number (of line) of clause
```

[139] Davis–Putnam–Logemann–Loveland

[140] https://github.com/DennisYurichev/SAT_SMT_by_example/blob/master/solvers/backtrack/POPCNT4.cnf

```
    # val=list of terms
    # term can be negative signed integer
    clauses=[]
    for c in content[1:]:
        if c.startswith ("c "):
            continue
        clause=[]
        for var_s in c.split(" "):
            var=int(var_s)
            if var!=0:
                clause.append(var)
        clauses.append(clause)

    # this is variables index.
    # for each variable, it has list of clauses, where this variable is used.
    # key=variable
    # val=list of numbers of clause
    variables_idx={}
    for i in range(len(clauses)):
        for term in clauses[i]:
            variables_idx.setdefault(abs(term), []).append(i)

    return clauses, variables_idx

# clause=list of terms. signed integer. -x means negated.
# values=list of values: from 0th: [F,F,F,F,T,F,T,T...]
def eval_clause (terms, values):
    try:
        # we search for at least one True
        for t in terms:
            # variable is non-negated:
            if t>0 and values[t-1]==True:
                return True
            # variable is negated:
            if t<0 and values[(-t)-1]==False:
                return True
        # all terms enumerated at this point
        return False
    except IndexError:
        # values[] has not enough values
        # None means "maybe"
        return None

def chk_vals(clauses, variables_idx, vals):
    # check only clauses which affected by the last (new/changed) value, ignore the rest
    # because since we already got here, all other values are correct, so no need to
       recheck them
    idx_of_last_var=len(vals)
    # variable can be absent in index, because no clause uses it:
    if idx_of_last_var not in variables_idx:
        return True
    # enumerate clauses which has this variable:
    for clause_n in variables_idx[idx_of_last_var]:
        clause=clauses[clause_n]
        # if any clause evaluated to False, stop checking, new value is incorrect:
        if eval_clause (clause, vals)==False:
```

```
            return False
    # all clauses evaluated to True or None ("maybe")
    return True

def print_vals(vals):
    # enumerate all vals[]
    # prepend "-" if vals[i] is False (i.e., negated).
    print "".join([["-",""][vals[i]] + str(i+1) + " " for i in range(len(vals))])+"0"

clauses, variables_idx = read_DIMACS(sys.argv[1])

solutions=0

def backtrack(vals):
    global solutions

    if len(vals)==len(variables_idx):
        # we reached end - all values are correct
        print "SAT"
        print_vals(vals)
        if count_solutions:
            solutions=solutions+1
            # go back, if we need more solutions:
            return
        else:
            exit(10) # as in MiniSat
        return

    for next in [False, True]:
        # add new value:
        new_vals=vals+[next]
        if chk_vals(clauses, variables_idx, new_vals):
            # new value is correct, try add another one:
            backtrack(new_vals)
        else:
            # new value (False) is not correct, now try True (variable flip):
            continue

# try to find all values:
backtrack([])
print "UNSAT"
if count_solutions:
    print "solutions=", solutions
exit(20) # as in MiniSat
```

As you can see, all it does is enumerate all possible solutions, but prunes search tree as early as possible. This is backtracking.

The files: https://github.com/DennisYurichev/SAT_SMT_by_example/tree/master/solvers/backtrack.

Some comments: https://www.reddit.com/r/compsci/comments/6jn3th/simplest_sat_solver_in_120_lines/.

## 22.2 MK85 toy-level SMT-solver

Thanks to PicoSAT SAT solver, its performance on small and simple bitvector examples is comparable to Z3. In many cases, it's used instead of Z3, whenever you see `from MK85 import *` in Python file.

### 22.2.1 Simple adder in SAT/SMT

Let's solve the following equation $a + b = 4 \equiv 2^4$ on the 4-bit CPU (hence, modulo $2^4$):

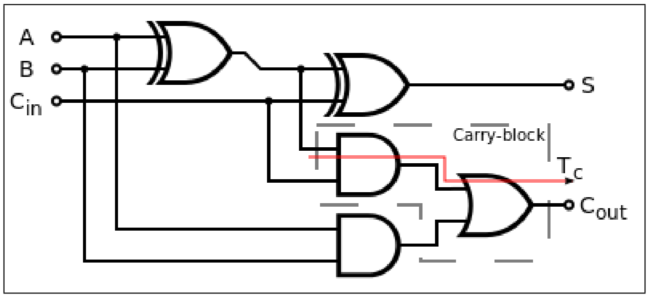
```
(declare-fun a () (_ BitVec 4))
(declare-fun b () (_ BitVec 4))

(assert (= (bvadd a b) #x4))

; find a, b:
(get-all-models)
```

There are 16 possible solutions (easy to check even by hand):

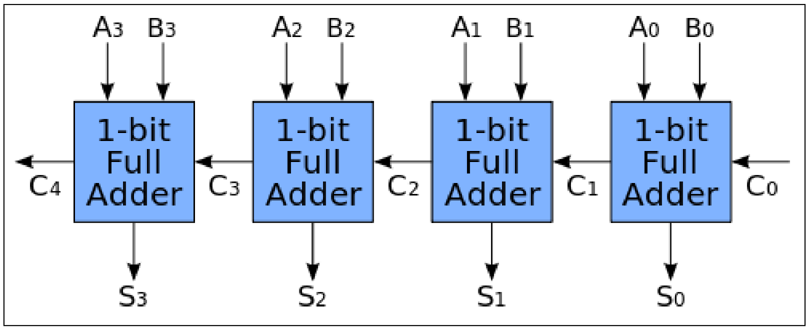
```
(model
        (define-fun a () (_ BitVec 4) (_ bv0 4)) ; 0x0
        (define-fun b () (_ BitVec 4) (_ bv4 4)) ; 0x4
)
(model
        (define-fun a () (_ BitVec 4) (_ bv12 4)) ; 0xc
        (define-fun b () (_ BitVec 4) (_ bv8 4)) ; 0x8
)

...

(model
        (define-fun a () (_ BitVec 4) (_ bv9 4)) ; 0x9
        (define-fun b () (_ BitVec 4) (_ bv11 4)) ; 0xb
)
Model count: 16
```

How I implemented this in my toy-level SMT solver?

First, we need an electronic adder, like it's implemented in digital circuits. This is basic block (full-adder) (image taken from Wikipeda):




This is how full adders gathered together to form a simple 4-bit carry-ripple adder (also from Wikipedia):



More info from wikipedia: https://en.wikipedia.org/wiki/Adder_(electronics).

I'm implementing full-adder like this:

```
void add_Tseitin_XOR(int v1, int v2, int v3)
{
        add_comment ("%s %d=%d^%d", __FUNCTION__, v3, v1, v2);
        add_clause3 (-v1, -v2, -v3);
        add_clause3 (v1, v2, -v3);
        add_clause3 (v1, -v2, v3);
        add_clause3 (-v1, v2, v3);
};

void add_Tseitin_OR2(int v1, int v2, int var_out)
{
        add_comment ("%s %d=%d|%d", __FUNCTION__, var_out, v1, v2);
        add_clause("%d %d -%d", v1, v2, var_out);
        add_clause2(-v1, var_out);
        add_clause2(-v2, var_out);
};

void add_Tseitin_AND(int a, int b, int out)
{
        add_comment ("%s %d=%d&%d", __FUNCTION__, out, a, b);
        add_clause3 (-a, -b, out);
        add_clause2 (a, -out);
        add_clause2 (b, -out);
};

void add_FA(int a, int b, int cin, int s, int cout)
{
        add_comment("%s inputs=%d, %d, cin=%d, s=%d, cout=%d", __FUNCTION__, a, b, cin,
           s, cout);
        // allocate 3 "joint" variables:
        int XOR1_out=next_var_no++;
        int AND1_out=next_var_no++;
        int AND2_out=next_var_no++;
        // add gates and connect them.
        // order doesn't matter, BTW:
        add_Tseitin_XOR(a, b, XOR1_out);
        add_Tseitin_XOR(XOR1_out, cin, s);
        add_Tseitin_AND(XOR1_out, cin, AND1_out);
        add_Tseitin_AND(a, b, AND2_out);
        add_Tseitin_OR2(AND1_out, AND2_out, cout);
};
```

( https://github.com/DennisYurichev/MK85/blob/master/MK85.cc )

|add_Tseitin*()| functions makes logic gates in CNF form: https://en.wikipedia.org/wiki/Tseytin_transformation. Then I connect logic gates to make full-adder.

Then I connect full-adders to create a n-bit adder:

```
void generate_adder(struct variable* a, struct variable* b, struct variable *carry_in,
    // inputs
        struct variable** sum, struct variable** carry_out) // outputs
{


        ...

        *sum=create_internal_variable("adder_sum", TY_BITVEC, a->width);
```

```
        int carry=carry_in->var_no;

        // the first full-adder could be half-adder, but we make things simple here
        for (int i=0; i<a->width; i++)
        {
                *carry_out=create_internal_variable("adder_carry", TY_BOOL, 1);
                add_FA(a->var_no+i, b->var_no+i, carry, (*sum)->var_no+i, (*carry_out)->
                   var_no);
                // newly created carry_out is a carry_in for the next full-adder:
                carry=(*carry_out)->var_no;
        };
};
```

( https://github.com/DennisYurichev/MK85/blob/master/MK85.cc )

Let's take a look on output CNF file:

```
p cnf 40 114
c always false
-1 0
c always true
2 0
c generate_adder
c add_FA inputs=3, 7, cin=1, s=11, cout=15
c add_Tseitin_XOR 16=3^7
-3 -7 -16 0
3 7 -16 0
3 -7 16 0
-3 7 16 0
c add_Tseitin_XOR 11=16^1
-16 -1 -11 0
16 1 -11 0
16 -1 11 0
-16 1 11 0
c add_Tseitin_AND 17=16&1
-16 -1 17 0
16 -17 0
1 -17 0
c add_Tseitin_AND 18=3&7
-3 -7 18 0
3 -18 0
7 -18 0
c add_Tseitin_OR2 15=17|18
17 18 -15 0
-17 15 0
-18 15 0
c add_FA inputs=4, 8, cin=15, s=12, cout=19
c add_Tseitin_XOR 20=4^8
-4 -8 -20 0
4 8 -20 0
4 -8 20 0
-4 8 20 0
c add_Tseitin_XOR 12=20^15
-20 -15 -12 0
20 15 -12 0
20 -15 12 0
-20 15 12 0
```

```
c add_Tseitin_AND 21=20&15
-20 -15 21 0
20 -21 0
15 -21 0
c add_Tseitin_AND 22=4&8
-4 -8 22 0
4 -22 0
8 -22 0
c add_Tseitin_OR2 19=21|22
21 22 -19 0
-21 19 0
-22 19 0
c add_FA inputs=5, 9, cin=19, s=13, cout=23
c add_Tseitin_XOR 24=5^9
-5 -9 -24 0
5 9 -24 0
5 -9 24 0
-5 9 24 0
c add_Tseitin_XOR 13=24^19
-24 -19 -13 0
24 19 -13 0
24 -19 13 0
-24 19 13 0
c add_Tseitin_AND 25=24&19
-24 -19 25 0
24 -25 0
19 -25 0
c add_Tseitin_AND 26=5&9
-5 -9 26 0
5 -26 0
9 -26 0
c add_Tseitin_OR2 23=25|26
25 26 -23 0
-25 23 0
-26 23 0
c add_FA inputs=6, 10, cin=23, s=14, cout=27
c add_Tseitin_XOR 28=6^10
-6 -10 -28 0
6 10 -28 0
6 -10 28 0
-6 10 28 0
c add_Tseitin_XOR 14=28^23
-28 -23 -14 0
28 23 -14 0
28 -23 14 0
-28 23 14 0
c add_Tseitin_AND 29=28&23
-28 -23 29 0
28 -29 0
23 -29 0
c add_Tseitin_AND 30=6&10
-6 -10 30 0
6 -30 0
10 -30 0
c add_Tseitin_OR2 27=29|30
29 30 -27 0
```

```
-29 27 0
-30 27 0
c generate_const(val=4, width=4). var_no=[31..34]
-31 0
-32 0
33 0
-34 0
c generate_EQ for two bitvectors, v1=[11...14], v2=[31...34]
c generate_BVXOR v1=[11...14] v2=[31...34]
c add_Tseitin_XOR 35=11^31
-11 -31 -35 0
11 31 -35 0
11 -31 35 0
-11 31 35 0
c add_Tseitin_XOR 36=12^32
-12 -32 -36 0
12 32 -36 0
12 -32 36 0
-12 32 36 0
c add_Tseitin_XOR 37=13^33
-13 -33 -37 0
13 33 -37 0
13 -33 37 0
-13 33 37 0
c add_Tseitin_XOR 38=14^34
-14 -34 -38 0
14 34 -38 0
14 -34 38 0
-14 34 38 0
c generate_OR_list(var=35, width=4) var out=39
c add_Tseitin_OR_list(var=35, width=4, var_out=39)
35 36 37 38 -39 0
-35 39 0
-36 39 0
-37 39 0
-38 39 0
c generate_NOT id=internal!8 var=39, out id=internal!9 out var=40
-40 -39 0
40 39 0
c create_assert() id=internal!9 var=40
40 0
```

Filter out comments:

```
c always false
c always true
c generate_adder
c add_FA inputs=3, 7, cin=1, s=11, cout=15
c add_Tseitin_XOR 16=3^7
c add_Tseitin_XOR 11=16^1
c add_Tseitin_AND 17=16&1
c add_Tseitin_AND 18=3&7
c add_Tseitin_OR2 15=17|18
c add_FA inputs=4, 8, cin=15, s=12, cout=19
c add_Tseitin_XOR 20=4^8
c add_Tseitin_XOR 12=20^15
c add_Tseitin_AND 21=20&15
```

```
c add_Tseitin_AND 22=4&8
c add_Tseitin_OR2 19=21|22
c add_FA inputs=5, 9, cin=19, s=13, cout=23
c add_Tseitin_XOR 24=5^9
c add_Tseitin_XOR 13=24^19
c add_Tseitin_AND 25=24&19
c add_Tseitin_AND 26=5&9
c add_Tseitin_OR2 23=25|26
c add_FA inputs=6, 10, cin=23, s=14, cout=27
c add_Tseitin_XOR 28=6^10
c add_Tseitin_XOR 14=28^23
c add_Tseitin_AND 29=28&23
c add_Tseitin_AND 30=6&10
c add_Tseitin_OR2 27=29|30
c generate_const(val=4, width=4). var_no=[31..34]
c generate_EQ for two bitvectors, v1=[11...14], v2=[31...34]
c generate_BVXOR v1=[11...14] v2=[31...34]
c add_Tseitin_XOR 35=11^31
c add_Tseitin_XOR 36=12^32
c add_Tseitin_XOR 37=13^33
c add_Tseitin_XOR 38=14^34
c generate_OR_list(var=35, width=4) var out=39
c add_Tseitin_OR_list(var=35, width=4, var_out=39)
c generate_NOT id=internal!8 var=39, out id=internal!9 out var=40
c create_assert() id=internal!9 var=40
```

I make these functions add variable numbers to comments. And you can see how all the signals are routed inside each full-adder.

|generate_EQ()| function makes two bitvectors equal by XOR-ing two bitvectors. Resulting bitvector is then OR-ed, and result must be zero.

Again, this SAT instance is small enough to be handled by my simple SAT backtracking solver:

```
SAT
-1 2 -3 -4 -5 -6 -7 -8 9 -10 -11 -12 13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 24 -25
    -26 -27 -28 -29 -30 -31 -32 33 -34 -35 -36
-37 -38 -39 40 0
SAT
-1 2 -3 -4 -5 6 -7 -8 9 10 -11 -12 13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 24 -25 -26
     27 -28 -29 30 -31 -32 33 -34 -35 -36 -37
-38 -39 40 0

...

SAT
-1 2 3 4 5 -6 7 -8 9 10 -11 -12 13 -14 15 -16 -17 18 19 20 21 -22 23 -24 -25 26 27 28 29
     -30 -31 -32 33 -34 -35 -36 -37 -38 -39 40 0
SAT
-1 2 3 4 5 6 7 -8 9 -10 -11 -12 13 -14 15 -16 -17 18 19 20 21 -22 23 -24 -25 26 27 28 29
     -30 -31 -32 33 -34 -35 -36 -37 -38 -39 40 0
UNSAT
solutions= 16
```

### 22.2.2 Combinatorial optimization

This is minimize/maximize commands in SMT-LIB. See simple example on GCD: 12.2.1.

It was surprisingly easy to add support of it to MK85. First, we take MaxSAT/WBO solver Open-WBO[141]. It supports

[141] http://sat.inesc-id.pt/open-wbo/

both hard and soft clauses. Hard are clauses which are *must* be satisfied. Soft are *should* be satisfied, but they are also weighted. The task of MaxSAT solver is to find such an assignment for variables, so the sum of weights of soft clauses would be *maximized*.

This is GCD example rewritten to SMT-LIB format:

```
; checked with Z3 and MK85

; must be 21
; see also: https://www.wolframalpha.com/input/?i=GCD[861,3969,840]

(declare-fun x () (_ BitVec 16))
(declare-fun y () (_ BitVec 16))
(declare-fun z () (_ BitVec 16))
(declare-fun GCD () (_ BitVec 16))

(assert (= (bvmul ((_ zero_extend 16) x) ((_ zero_extend 16) GCD)) (_ bv861 32)))
(assert (= (bvmul ((_ zero_extend 16) y) ((_ zero_extend 16) GCD)) (_ bv3969 32)))
(assert (= (bvmul ((_ zero_extend 16) z) ((_ zero_extend 16) GCD)) (_ bv840 32)))

(maximize GCD)

(check-sat)
(get-model)

; correct result:
;(model
;        (define-fun x () (_ BitVec 16) (_ bv41 16)) ; 0x29
;        (define-fun y () (_ BitVec 16) (_ bv189 16)) ; 0xbd
;        (define-fun z () (_ BitVec 16) (_ bv40 16)) ; 0x28
;        (define-fun GCD () (_ BitVec 16) (_ bv21 16)) ; 0x15
;)
```

We are going to find such an assignment, for which GCD variable will be as big as possible (that would not break *hard* constraints, of course).

Whenever my MK85 encounters *minimize/maximize* command, the following function is called:

```
void create_min_max (struct expr* e, bool min_max)
{
        ...

        struct SMT_var* v=generate(e);

        // if "minimize", negate input value:
        if (min_max==false)
                v=generate_BVNEG(v);

        assert (v->type==TY_BITVEC);
        add_comment ("%s(min_max=%d) id=%s var=%d", __FUNCTION__, min_max, v->id, v->
           SAT_var);

        // maximize always. if we need to minimize, $v$ is negated at this point:
        for (int i=0; i<v->width; i++)
                add_soft_clause1(/* weight */ 1<<i, v->SAT_var+i);

        ...
};
```

( https://github.com/DennisYurichev/MK85/blob/master/MK85.cc )

Lowest bit of variable to maximize receives weight 1. Second bit receives weight 2. Then 4, 8, 16, etc. Hence, MaxSAT solver, in order to maximize weights of soft clauses, would maximize the binary variable as well!

What is in the WCNF (weighted CNF) file for the GCD example?

```
...

c create_min_max(min_max=1) id=GCD var=51
1 51 0
2 52 0
4 53 0
8 54 0
16 55 0
32 56 0
64 57 0
128 58 0
256 59 0
512 60 0
1024 61 0
2048 62 0
4096 63 0
8192 64 0
16384 65 0
32768 66 0
```

Weights from 1 to 32768 to be assigned to specific bits of GCD variable.

Minimization works just as the same, but the input value is negated.

Now some practical examples MK85 can already solve: Assignment problem: 12.3, Finding minimum of function: 12.4, Minimizing cost9.1.

More optimization examples from my blog, mostly using z3: Making smallest possible test suite using Z3: 12.1, Coin flipping problem: 7.18, Cracking simple XOR cipher with Z3: 18.4.

### 22.2.3 Making (almost) barrel shifter in my toy-level SMT solver

...so the functions `bvshl` and `bvlshr` (logical shift right) would be supported.

We will simulate barrel shifter, a device which can shift a value by several bits in one cycle. This one is one nice illustration: https://i.stack.imgur.com/AefYE.jpg.

See also: https://en.wikipedia.org/wiki/Barrel_shifter

So we have a pack of multiplexers. A tier of them for each bit in *cnt* variable (number of bits to shift).

First, I define functions which do *rewiring* rather than shifting, it's just another name. Part of input is *connected* to output bits, other bits are fixed to zero:

```
        // "cnt" is not a SMT variable!
        struct SMT_var* generate_shift_left(struct SMT_var* X, unsigned int cnt)
        {
                int w=X->width;

                struct SMT_var* rt=create_internal_variable("shifted_left", TY_BITVEC, w
                   );

                fix_BV_to_zero(rt->SAT_var, cnt);

                add_Tseitin_EQ_bitvecs(w-cnt, rt->SAT_var+cnt, X->SAT_var);

                return rt;
        };

        // cnt is not a SMT variable!
        struct SMT_var* generate_shift_right(struct SMT_var* X, unsigned int cnt)
```

```
{
        ... likewise
};
```

It can be said, the |cnt| variable would be set during SAT instance creation, but it cannot be changed during solving.

Now let's create a *real* shifter. Now for 8-bit left shifter, I'm generating the following (long) expression:

```
X=ITE(cnt&1, X<<1, X)
X=ITE((cnt>>1)&1, X<<2, X)
X=ITE((cnt>>2)&1, X<<4, X)
```

I.e., if a specific bit is set in |cnt|, shift $X$ by that number of bits, or do nothing otherwise. `ITE()` is a if-then-else gate, works for bitvectors as well.

Glueing all this together:

```
// direction=false for shift left
// direction=true for shift right
struct SMT_var* generate_shifter (struct SMT_var* X, struct SMT_var* cnt, bool
    direction)
{
        int w=X->width;

        struct SMT_var* in=X;
        struct SMT_var* out;
        struct SMT_var* tmp;

        // bit vector must have width=2^x, i.e., 8, 16, 32, 64, etc
        assert (popcount64c (w)==1);

        int bits_in_selector=mylog2(w);

        for (int i=0; i<bits_in_selector; i++)
        {
                if (direction==false)
                        tmp=generate_shift_left(in, 1<<i);
                else
                        tmp=generate_shift_right(in, 1<<i);

                out=create_internal_variable("tmp", TY_BITVEC, w);

                add_Tseitin_ITE_BV (cnt->SAT_var+i, tmp->SAT_var, in->SAT_var,
                    out->SAT_var, w);

                in=out;
        };

        // if any bit is set in high part of "cnt" variable, result is 0
        // i.e., if a 8-bit bitvector is shifted by cnt>8, give a zero
        struct SMT_var *disable_shifter=create_internal_variable("...", TY_BOOL,
             1);
        add_Tseitin_OR_list(cnt->SAT_var+bits_in_selector, w-bits_in_selector,
            disable_shifter->SAT_var);

        return generate_ITE(disable_shifter, generate_const(0, w), in);
};

struct SMT_var* generate_BVSHL (struct SMT_var* X, struct SMT_var* cnt)
{
```

```
        return generate_shifter (X, cnt, false);
};
```

Now the puzzle. `a>>b` must be equal to |0x12345678|, while several bits in $a$ must be reset, like `(a&0xf1110100)==0`. Find a, b:

```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))

(assert (= (bvand a #xf1110100) #x00000000))

(assert (= (bvshl a b) #x12345678))

(check-sat)
(get-model)
```

The solution:

```
sat
(model
        (define-fun a () (_ BitVec 32) (_ bv38177487 32)) ; 0x2468acf
        (define-fun b () (_ BitVec 32) (_ bv3 32)) ; 0x3
)
```

## 23 Further reading

- Julien Vanegue, Sean Heelan, Rolf Rolles – SMT Solvers for Software Security [142]
- Armin Biere, Marijn Heule, Hans van Maaren, Toby Walsh – Handbook of Satisfiability (2009)
- Donald Knuth – TAOCP 7.2.2.2. Satisfiability [143].
- Rui Reis – Practical Symbolic Execution and SATisfiability Module Theories (SMT) 101 [144].
- Daniel Kroening and Ofer Strichman – Decision Procedures – An Algorithmic Point of View [145].

## 24 Some applications

- Cryptol [146]: a language for cryptoalgorithms specification and prooving it's correctness. Uses Z3.

## 25 Acronyms used

**GCD** Greatest Common Divisor .................... 4

**LCM** Least Common Multiple .................... 4

**CNF** Conjunctive normal form .................... 7

**DNF** Disjunctive normal form .................... 11

[142] https://yurichev.com/mirrors/SMT/woot12.pdf
[143] http://www-cs-faculty.stanford.edu/~knuth/fasc6a.ps.gz
[144] http://deniable.org/reversing/symbolic-execution
[145] http://www.decision-procedures.org
[146] http://cryptol.net, https://github.com/GaloisInc/cryptol

**DSL** Domain-specific language . . . . . 9

**CPRNG** Cryptographically Secure Pseudorandom Number Generator . . . . . 39

**SMT** Satisfiability modulo theories . . . . . 1

**SAT** Boolean satisfiability problem . . . . . 1

**LCG** Linear congruential generator . . . . . 1

**PL** Programming Language . . . . . 9

**OOP** Object-oriented programming . . . . . 304

**SSA** Static single assignment form . . . . . 265

**CPU** Central processing unit . . . . . 268

**FPU** Floating-point unit . . . . . 329

**PRNG** Pseudorandom number generator . . . . . 13

**CRT** C runtime library . . . . . 341

**CRC** Cyclic redundancy check . . . . . 4

**AST** Abstract syntax tree . . . . . 305

**AKA** Also Known As . . . . . 3

**CTF** Capture the Flag . . . . . 383

**ISA** Instruction Set Architecture . . . . . 268

**CSP** Constraint satisfaction problem . . . . . 127

**CS** Computer science . . . . . 7

**DAG** Directed acyclic graph . . . . . 56

**NOP** No Operation . . . . . 273

**JVM** Java Virtual Machine . . . 329

**VM** Virtual Machine . . . 344

**LZSS** Lempel–Ziv–Storer–Szymanski . . . 250

**RAM** Random-access memory . . . 365

**FPGA** Field-programmable gate array . . . 384

**EDA** Electronic design automation . . . 384

**MAC** Message authentication code . . . 385

**ECC** Elliptic curve cryptography . . . 385

**API** Application programming interface . . . 9

**NSA** National Security Agency . . . 39

**DPLL** Davis–Putnam–Logemann–Loveland . . . 447

**SGP** Social Golfer Problem . . . 227

**FSM** Finite State Machine . . . 91

**RE** Regular Expression . . . 91

**RSA** Rivest–Shamir–Adleman cryptosystem . . . 2

**ITE** If-Then-Else . . . 2

**DFA** Deterministic finite automaton . . . 91

**TAOCP** The Art Of Computer Programming . . . 174

**DES** Data Encryption Standard . . . 384

**ALU** Arithmetic logic unit . . . 47

**EE** Electrical engineering . . . 155

**PCB** Printed circuit board . . . 3