Time Complexity of Knuth-Morris-Pratt String Matching Algorithm

Course Project Report for COMP-160, Fall 2010

Mengfei Cao MCAO01@CS.TUFTS.EDU
Dept. Computer Science, Tufts University, Medford, USA

Abstract

This project centers on the evaluation for the time complexity of Knuth-Morris-Pratt(KMP) string matching algorithm. String matching problem is to locate a pattern string within a larger string. The best performance in terms of asymptotic time complexity is currently linear, given by the KMP algorithm. In this algorithm, firstly a prefix for the pattern string is computed and then based on this prefix, only linear time is needed to find the pattern string in the larger string. In details of both steps, the prefix computation and matcher, two loops are needed, where one goes over the whole string and the other loop travels back to some point of the prefix array. I test these two steps with a large number of random input strings, of which the lengths are different, and for evaluation, ranges of the strings are also under control to simulate the various cases. As much as possible, I expect to get the tight analysis of the KMP algorithm's time complexity. During these tests, the numbers of basic operations are recorded and then linear analysis is conducted, giving the coefficients of linear regression and the correlation coefficient. Finally, together with the analysis, it is concluded that the linear time complexity is validated based on the experiments.

1. Introduction

The time complexity of a given algorithm can be obtained from theoretical analysis and computational analysis according to the algorithm's running process. Both methods estimate the time complexity by counting the number of basic operations, which cost some basic unit of time. In terms of the theoretical analysis, the task sometimes is quite simple by just counting the maximum number of operations; however, when some conditional evaluations are added, the strategy of theoretically counting maximum number of operations may be confusing, or far too pessimistic because due to the conditional evaluations, many seemingly possible operations will never co-occur, thus resulting in great decrease in the number of total operations. Yet still, if all of the cases are known, it is obvious for the theoretical counting to obtain the tight analysis of the time complexity. Thus, the computational approximation is useful to help understand the tight analysis of time complexity. In this project, the estimation of KMP algorithm's time complexity is confusing because in both of its two steps, there are two seemingly-going-over-all loops but the time complexity is still linear to the input size. Thus, I generate different random inputs and computationally count the number of basic operations. After gaining all these operations' numbers respect to the input size, I plot the results and do the linear regression. It is so notable of the linear relationship in the experiment part that no polynomial approximation is necessary.

The intuition of testing the time complexity computationally is to count every basic operation respect to different inputs. As for the generation of inputs, two demands should be satisfied:

- The input strings with a given length should cover all possible cases;
- The input strings' lengths should range from the minimum 1 to infinite.

These two demands are the intuitively complete rules for testing an algorithm's asymptotic time complexity. However, the infinite inputs with infinite lengths are impossible; thus, computationally, I replace the first infinite cases rule with sampling randomly, and the second rule with some very large length. It is true that singly based on my experiments it is impossible to reach any conclusion of the algorithm's performance, but the random sampling and some large number of experiments can still provide the estimation of the ground truth performance. Namely, the finite experiments provide the clues of analysis and directions to work on. In this project, after the experiments are presented the discussion with concrete theoretical analysis is given to validate the conclusion that the KMP algorithm's asymptotic time complexity is O(n).

The rest of the report will give the formulation of the KMP algorithm, the details of my strategy to test the two key steps of KMP algorithm, and the experiments together with the discussion.

2. Knuth-Morris-Pratt Algorithm

KMP algorithm was conceived in 1977 by three eminent computer scientists: Dr. James H. Morris, Dr. Vaughan Pratt, and Dr. Donald Ervin Knuth. Instead of directly match the pattern string to the larger string, KMP algorithm firstly compute a prefix for the pattern string and then match the string on the larger string based on the prefix array. Only by one travel of the two strings will it achieve the goal of computing prefix and finding the match, resulting in the best linear time complexity.

Given a pattern string P and a text string T. Then sizes of the two strings are respectively m and n. The KMP algorithm firstly compute a prefix array Pi for P, with size of m. Each element in Pi records the index from which if a mismatch occurs the next match should begin. Thus, it is not necessary to test the whole pattern string over and over whenever there is a mismatch. This is the intuition of the algorithm's matching strategy. As for how to compute the prefix, in the preprocessing of the pattern string, one travel of this string is needed so as to find out the partial match itself, but what is important that the partial match information is reusable for the following travel. Thus no matter for the matcher or the computing prefix, the time complexity is always linear to the size of the input. The time complexity of computing prefix is linear to the size of the pattern string, and that of matcher is linear to the size of the text string.

The following is the pseudocode of KMP algorithm.

```
ComputePrefix(P[0...m-1], m)
1.  Pi[0] = -1;
2.  k = -1;
3.  for q=1:m-1 {
4.      while(k>=0 && P[k+1]!=P[q]){
5.          k = Pi[k];}
6.      if(P[k+1]==P[q]) then k++;
7.      P[q] = k;
8.  }return Pi;
```

```
KMPMatcher(T, n, P, m)
1.  Pi = ComputePrefix(P,m);
2.  q = -1;
3.  for i = 0:n-1 {
4.      while(q>=0 && P[q+1]!=T[i]){
5.          q = Pi[q];}
6.      if(P[q+1]==T[i])    then q++;
7.      if(q == m-1)    then report index i-m+1; q = Pi[q];
8.  }return;
```

In the following section, I will test these two components separately and obtain the estimation of the time complexity.

3. Strategy to Test the Performance

As mentioned before, the time complexity is usually estimated by counting the number of basic operations. In addition, the two infinite rules should be approximately obeyed so as to obtain the tight performance of the algorithm. Thus, I use the randomness to tackle the problem derived from the first infinite rule and use iterations and large number of input strings to tackle the problem out of the second rule.

3.1 Test ComputePrefix()

For the first component, ComputePrefix(), there are two loops for and while. Also, both two loops go over the array P, of which the length is m. Naively, the time complexity may be m times m, resulting in m square. However, the while loop doesn't always go over all the array P due to the condition k>=0 and P[k+1]!=P[q]. In fact, inside the loop, every time the k is updated with the space, that goes across a partial matched sequence instead of single step. In order to test the number of basic operations, I set a count variable inside the while evaluation so that whenever the expression inside the while bracket is evaluated, the count variable will add one.

After the analysis of the function as well as the measure of the time complexity, here what is given is the strategy to estimate the asymptotic time complexity. As elaborated above, two rules should be obeyed so as to obtain the time complexity. Here I generate many strings randomly, of which each character appears in the range a-z randomly. Moreover, for a given length, I repeatedly generate 100 different random strings P_m^j, $j = 1,...,100$ to serve as the input strings. Afterwards, I calculate the average number $Number_m$ of the basic operations according to the 100 iterations. In addition, what's also important is that I generate the strings with different lengths. For example, I set the maximum size of the strings as 1000, and thus for each length in 1-1000, I generate 100 random strings to compute the prefix, at the same time recording the number of basic operations. Finally according to the following formula, I compute the coefficients c_1, c_0 and the correlation coefficient r:

$$Number = c_1 * m + c_0 \tag{1}$$

$$r = \frac{(\sum_{m=1}^{maximumlength} (Number_m - \overline{N}) \bullet (m - \overline{m}))}{\sqrt{\sum_{m=1}^{maximumlength} (Number_m - \overline{N})^2 \bullet \sum_{m=1}^{maximumlength} (m - \overline{m})^2}} \tag{2}$$

where $\overline{N}, \overline{m}$ represent the average number of operations and average length.

The correlation coefficient r is used to measure the linear correlation. If r is equal to 1, the two variables are positively linear correlated; if r is equal to -1, the two variables are negatively linear correlated. In order to evaluate the linearity, two statistics are computed, namely, $R^2 - statistics$ and $F - statistics$, where the more first statistic is close to 1 and the bigger the second statistic is, the more obvious the linearity is. More precisely for the second statistic, $F - statistics$, it corresponds a p-value that represents the probability how much significantly that the linear regression model can **not** describe the data.

Moreover, the partial match will affect the while loop. Thus, the cases where the number of partial matches differs are also considered by setting the range of the characters in the string. For example, when I set each character in the string falls in a~z, namely 26 letters, the random string with length less than 50 is rare to have partial match sequence with length over 3; yet, if the characters are in a~c, namely 3 letters in total, it is quite probable that the longer partial match will take place. Therefore, by setting different ranges of the characters in the string, I observe the correlation between numbers of basic operations and the size of the strings.

3.2 Test KMPMatcher()

As for the KMPMatcher() function, there are two input strings and in my detailed implementation there is the evaluation that the size of string P should always be smaller than that of T. Thus, here what needs to be validated is the linear relationship between basic operations' number and the sum of n, size of T, and m, size of P ($m \leqslant n$). In addition, the basic operation here is defined as the evaluation expression inside the while loop bracket. Similar to above, the naive analysis will lead to the time complexity of n multiplying m. Thus, the more precise analysis is necessary.

Firstly, I consider string P as constant and then again I generate many strings randomly to serve as input, counting the basic operations. Given the length of string T, I generate a certain number of strings with the same length randomly and then average all the iterations. Also, the maximum number of length is set as large so as to approximate the infinite cases(although never enough, yet what matters is the tendency). Then I release the length of P, namely let m vary, and get the average number for the given length of P and the given length of T. For example, I set the length of P, m varying from 1 to 100, and set the length of T, n varying from m to m+1000; for each m and n, I generate a random string P, and generate 100 random strings T to run KMP string matching; after I get the 100 numbers of basic operations, I average them and keep the average as the number of operations given m and n; after all these, I analyze the relationship between the number of operations and m and n using following formula:

$$Number = c_2 * n + c_1 * m + c_0 \quad (3)$$

In order to compare the linearity results, I also conduct the approximation:

$$Number = \hat{c}_1 * (m \bullet n) + \hat{c}_0 \quad (4)$$

And calculate the $R^2 - statistics$.

Based on the input type of this function, I conduct another comparison group experiments with different types of input. Since the partial match will matter, the range of the strings' character is set different so that I could observe the different results under different configurations. For example, when every character in the string falls in the range a-z, thus 26 different kinds of characters in total, it is less likely to have partial match during the matching step, thus less back track during the while loop; when each character is in the range a-c, namely the range is much smaller, it is more likely to have partial match and during the while loop, the step length of each back tracking is also small, thus may resulting in more basic operations. This setting derives from the intuition that in the while loop of the function, if the partial match is more or the repetitive characters are more, it is more seemingly that the index q will track back more slowly. Then I will also compare the results with different settings of ranges.

4. Experiments & Discussion

In this section, first I will give the results and analysis respect to the ComputePrefix() and KMPMatcher(), and then I will give the discussion based on these two experiments and conclude that the linear complexity is solid according to the experimental inspiration and theoretical analysis.

4.1 Performance and Analysis of ComputePrefix()

In this part, the linear approximation (1) is computed together with the correlation coefficient (2); as well, the $R^2 -$ statistics and $F - statistics$ are computed to evaluate the linearity. For the input pattern string, I set the length from 1 to 1000, and for each length, I generate 100 random strings to iterate, averaging them. Also, the ranges of the characters are separately $\{a\},\{a,b\},\{a,b,c\},\{a,b,c,d,e\},\{a\sim z\}$.

Table 1. Linear Regression Results and Evaluation for Compute Prefix

range / pars.	{a}	{a,b}	{a,b,c}	{a,b~e}	{a,b~z}
c_0	0.0000	-0.9724	-0.7866	-0.4199	-0.0732
c_1	1.0000	1.4177	1.3195	1.1983	1.0385
r	1.0000	1.0000	1.0000	1.0000	1.0000
R^2 – *statistics*	1.0000	1.0000	1.0000	1.0000	1.0000
F – *statistics*	5.6750e +33	2.5078e +07	1.0387e +08	1.7146e +08	4.6896e +08
p-value	0.0000	0.0000	0.0000	0.0000	0.0000

In Table 1, the line corresponds to different range and the row corresponds to different parameters. As seen above, there are two phenomena concluded:

- For all ranges, the coefficient c_1 is positive makes sense that with the increase of the string's size, the number of basic operations increase proportionally. Moreover, the linearity is solid based on the three different evaluation standards: correlation coefficient, $R^2 - statistics$ and $F - statistics$. As seen from the chart, both of the correlation coefficients and $R^2 - statistics$ are 1.0000; the $F - statistics$ is also high that the corresponding p-value is 0.0000. This reveals that the linearity is significantly solid based on the experiments.
- For different ranges, the coefficient differs. Especially the ranges with 2 characters {a,b} and that with 3 characters {a,b,c} have the largest coefficient c_1 which indicates that the number of basic operations increases the fastest in these two cases. It is reasonable that since there are fewer types of characters, thus the partial match inside the string is more likely to happen, resulting in more possibility of backtrack in the while loop. As for the case where range is 1, in fact the string is certain, with the format "a...a", and the while loop will never track back but continue to add 1 to k, resulting in the coefficient 1 with 0 standard deviation (Figure 1).

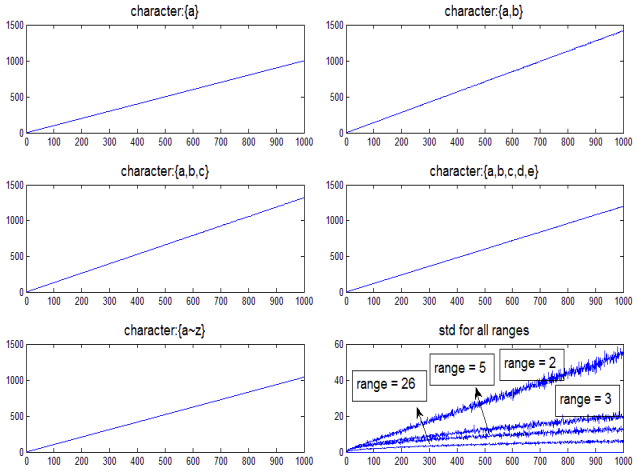

Figure 1. Numbers of Basic Operations and Std with Different Ranges

4.2 Performance and Analysis of KMPMatcher()

As comparison, both linear regression of (3) and (4) are conducted. Due to the limit of the computer's capacity, I downsize the iterations to 50. In details, the maximum length of the pattern string is 50, while the maximum additional length of the text string is 500. Namely, for each length m from 1 to 50, I randomly generate the pattern string P with length m, then given the pattern string m, for each length n from m to m+500, I generate 50 random text strings T with length n. Afterwards, input these strings and count the basic operations, averaging the 50 iterations, and use the 50*500 groups of data to linear approximate.

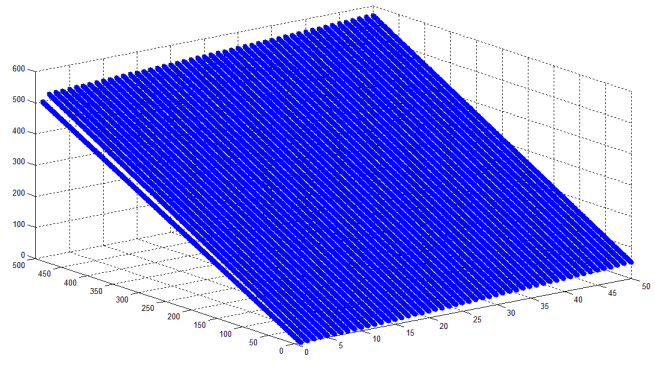

Figure 2. 3D Plot of Linear Regression Model(3) (this is the case where range = 26, namely each character falls in {a~z})

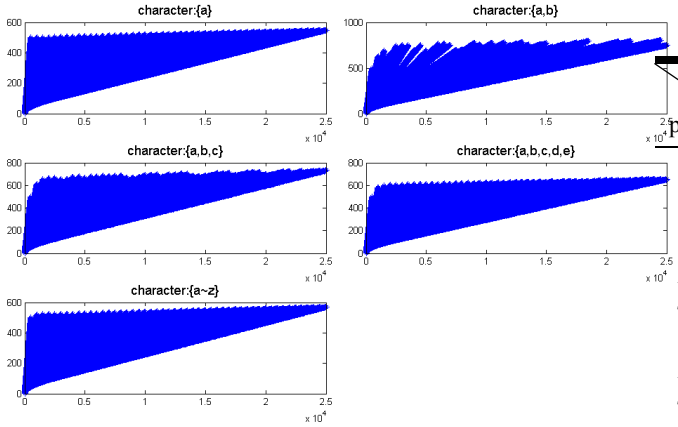

Figure 3. Numbers of Basic Operations with Different Ranges Using Model(4) (the x-axis is the product of m and n)

From the figures above, it seems that obviously the time complexity, namely the number of basic operations, is linear to both m and n, thus resulting in the 3D plane. However, if we plot the time complexity with the product of m and n, the figure above indicates that no solid principle between these two variables are obvious. Then let's look at the numerical results as following:

Table 2. Linear Regression Results and Evaluation for KMP Matcher Using (3)

pars. \ range	{a}	{a,b}	{a,b,c}	{a,b~e}	{a,b~z}
c_0	1.0000	-8.0394	-8.6998	-4.4128	-1.6885
c_1	1.0000	1.6502	1.5915	1.3129	1.0623
c_2	1.0000	1.4010	1.3010	1.1927	1.0377
R^2 – *statistics*	1.0000	0.9867	0.9948	0.9979	0.9999
F – *statistics*	2.2386e +33	9.2494e +07	2.4082e +08	5.8351e +08	1.1400e +08
p-value	0.0000	0.0000	0.0000	0.0000	0.0000

Table 3. Linear Regression Results and Evaluation for KMP Matcher Using (4)

pars. \ range	{a}	{a,b}	{a,b,c}	{a,b~e}	{a,b~z}
c_0	154.6014	214.0045	198.3111	182.5114	160.0493
c_1	0.0188	0.0271	0.0253	0.0227	0.0196
R^2 – *statistics*	0.5226	0.5350	0.5440	0.5329	0.5252
F – *statistics*	2.7365e +04	2.8761e +04	2.9829e +04	2.8525e +04	2.7656e +04
p-value	0.0000	0.0000	0.0000	0.0000	0.0000

Table 2 gives the results of 2-agents linear regression model. The linear coefficient for m and n is around 1 or 2, which still makes sense that with m or n's increasing, the number of basic operations increases proportional to these two variables. Also, the $R^2 - statistics$ is nearly 1 and the $F - statistics$ is high enough that the p-value falls 0, which reveals that this linear model can describe the data significantly. As for the Table 3, which consists of the results for model (4), it can be seen that the linearity is not significantly obvious according to $R^2 - statistics$ because the value is only slightly over 0.5; alghouth it is also true that the $F - statistics$ is high but it can only tell that overall the model can describe the data better than that with high orders of m*n. In conclusion, the model (3):

$$Number = c_2 * n + c_1 * m + c_0 \qquad (5)$$

is statistically significant descriptive based on the experiments. Moreover, the time complexity of the KMP matcher is linear to the size m of the pattern string and to the size n of the text string according to our expeirments.

4.3 Theoretical Analysis and Conclusion

Consider the case if we had no idea what exactly the time complexity of KMP algorithm was, we could be led now towards the linear results by the experiments. However, there are still two loops, one embedded into the other, so how to understand the linear complexity? Since we've got the possibly correct intuitive that the time complexity is linear, we may try to go over the algorithm and think of it again, from another perspective.

For ComputePrefix(), when we go over the pattern string, the index of the while loop in fact at most but never would travel the string twice. Whenever there is a partial mismatch, the while loop would travel back to find one that is the partial match for current subsequence, with the step recorded in the pattern array Pi. Each time it will cross the impossible match subsequence, with the step bigger or equal to 1. In fact, sum up all of these steps, and

it is clear that the length of the sum is no bigger than m. This makes sense that each time when we come across a mismatch we try to track back to find out the correct partial match but at most we can find as long as the current length and once we've track back we can never reach the current length, namely q. From this opinion, it is concluded that the while loop would run at most 2*m. It is also indicated in the experiments that each linear coefficient is no more than 2(the highest in the first experiment is 1.4177). Or think it in another way, each time there is a partial match, the variable k would add one, and thus at most the k could be m; yet each time there is a mismatch, k will in large scale track back until find one partial match (P[k+1]=P[q]) or to the start point (k=0). Thus, the k would at most changes 2*m times. Again, we get the 2*m upper bound for ComputePrefix(). As for the constant coefficients for different cases, it can be seen from the experiments as well as the analysis above.

For KMPMatcher(), similar to above, although there are two loops, the time complexity is still no worse that linear to m and n due to the fast backtrack and slow increment respect to q. When there is a mismatch, q will jump forward several numbers of positions, which depends on the occurrence of current subsequence in the pattern sequence. If we stretch the pattern string to as long as the text string, it is obvious that the scan of text string will at most be the sum of this two strings, namely 2*n. From another perspective similar to the last paragraph, for each i from 1 to n, the index q at most add one at one time but will always decrease much, namely backtracking a lot especially after the q has been added continuously. Thus, the time of q's changing will always be less than 2*n. Therefore, it can also be concluded that the time complexity is bounded by 2*n. This may also be inspired from the experiments, where the linear coefficients are never over 2(the largest pair appears when range = 2, and the values are 1.6502 and 1.4010). As for the constant coefficient and the linear coefficient, it can be referred to the former parts where experiments and analysis are given.

In all, the linear complexity for both component of the KMP algorithm is validated. It also follows the common clues of solving problems: theoretical analysis, put forward a possible model, experiment this model, and finally analyze the solutions solidly based on the supportive experimental results.

Acknowledgements

From this project, I practiced how to computationally estimate the time complexity of a given algorithm. This is useful when I come across an algorithm of which the time complexity is hard to deducted from theoretical counting. Also, this project helps me understand the KMP algorithm. In addition, I reviewed the linear regression analysis through this project. Thank Prof. Souvaine for the chance, and many thanks to her wonderful course. Thank the two teaching assistant, Mr. Winslow and Mr. Majidi, for their help and encouragement.

References

.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms (second edition)*. MIT Press, Cambridge, MA&McGraw Hill.

http://en.wikipedia.org/wiki/Knuth_Morris_Pratt_algorithm. The wikipedia for KMP algirthm.