CS 655: Analyzing Sequences
Homework 5

Part 1: Implementing Knuth-Morris-Pratt and Boyer-Moore

Knuth-Morris-Pratt

The pattern is scanned from left to right. If there is a mismatch at the first position, the pattern is shifted right by one place. If a mismatch occurs beyond the first position, the pattern is shifted to the right by the value of failure function of the position of mismatch. If a match is found, the pattern is shifted right by m-sp[m-1] places (m is the length of the pattern).

Boyer-Moore

Here the pattern is scanned from right to left. The amount of shift in different conditions is as below:

- Mismatch at first comparison (right-most position (pat_len-1)):
  Shift = max (1, bad_char)
- Mismatch at position j ($j \neq$ pat_len-1):
  - If there is a matching suffix (L'[j] > 0):
    Shift = max (bad_char, pat_len - L')
  - If there is no matching suffix
    Shift = max (bad_char, pat_len - l')
- Match found:
  Shift = pat_len – l'[1]

Implementation of Knuth-Morris-Pratt was pretty straightforward. But implementation of Boyer-Moore from the Gusfield book took a little while, the only reason being off-by-one errors.

Part 3: Implementing Optimal Mismatch and Maximal Shift algorithms

1. Optimal Mismatch Algorithm:

The basic idea of this algorithm is to scan characters in the pattern, from the least frequent to most frequent ones. Searching for the least frequent characters can lead to faster mismatch, consequently the whole text is scanned faster. The only caveat is, the frequencies of each character in the alphabet should be known. This requires processing on the text to be searched (haystack).

The preprocessing phase of this algorithm has three parts:

1. Sorting the pattern in increasing order of their frequencies
2. Calculating bad-character shift values
3. Calculating good-suffix shift values

Sorting pattern according to frequency

Consider the text and pattern given below:

Text : gcatcgcagagagtatacagtacg
Pattern : gcagagag

First, the frequencies of all characters in text are calculated.

Character	g	c	a	t
Frequency	7	5	8	4

Sorting by the order of increasing frequencies: t > c > g > a. The characters in the pattern are re-ordered to follow this rule. If a character is present in multiple places in the pattern, the right-most occurrence is considered first, as it can lead to larger shifts while using bad character or good suffix shift functions. If two characters have the same frequency, again the one closest to the end of the pattern is selected. Using this knowledge the above pattern is re-ordered as:

j	0	1	2	3	4	5	6	7
x[j]	g	c	a	g	a	g	a	g
I[j]	1	7	5	3	0	6	4	2
p[I[j]]	c	g	g	g	g	a	a	a

While searching, the pattern is matched in the order given by pat, i.e. first c is matched, then g and so on.

Bad-Character shift function:
The bad-character shift function is a little different from that in Boyer-Moore algorithm. Here, they consider the fact that, the pattern is always shifted right by at least one character. Therefore, the character in the text immediately after the end of the pattern T[k+m] (k is the position in the text and m is the length of the pattern), is always considered while checking for the next match. Thus, the bad character shift is calculated with respect to T[k+m]. The value of the bad-character shift is given by length of the pattern minus the index of the first leftward occurrence of this character from the end of the pattern. If the character is not present in the pattern, then the bad-character shift is given as len_patt+1, i.e. as the next character in text is not present in the pattern, we can move the pattern beyond that point without missing any matches.

For example, in the pattern 'gcagagag', the bad character shift value for character 'a' is 2. Its first leftward appearance from end of pattern is at position 6, and length of pattern is 8. So, bad-character shift is 2 (8-6). Now when we shift the pattern right by 2 places, the 'a' in text will be aligned to 'a' in position 6 of the pattern. Similarly, the bad-character shift values for rest of the characters in the pattern 'gcagagag' are:

Character	a	c	g	t
Bad_char shift	2	7	1	9

The advantages of calculating bad-character shift values this way over Boyer-Moore are:

- It is an absolute shift and is not defined relative to the position of last mismatch.
- As the value is defined based on a character that lies outside the range of present comparison, it does not change with the order in which the pattern is scanned, i.e. in which ever order the pattern is scanned the bad-character shift value for a character remains the same.
- It results in slightly larger shifts than Boyer-Moore as we are considering one character more in advance.

Good Suffix Shift Function:

The order in which the pattern is scanned is represented as I[ ]= {I[0],....,I[m-1]}. I[j] is the position of the character in the original pattern of the j th scan element and p[I[j]] is the character at that position (as shown in table under sorting pattern).

Good-suffix shift function as defined by Sunday [1] is the minimum left shift so that p[I[0]].... p[I[j-1]] match their aligned characters in original pattern, but such that p[I[j]] does not.

This value is calculated in two steps. First the values of minimum left shift so that p[I[0]].... p[I[j-1]] match their aligned characters in original pattern are calculated. Then these values are corrected to add the condition that the current character should not match.

Step 1: Calculate minimum left shift to match p[I[0]].... p[I[j-1]]

Considering the same pattern in previous examples 'gcagagag'. Let GS[] represent the Good-suffix shift values. The order in which the pattern is scanned is

j	0	1	2	3	4	5	6	7
I[j]	1	7	5	3	0	6	4	2
p[I[j]]	c	g	g	g	g	a	a	a

The shift at position j = 0, is set as 1 (GS[0] = 1), as mismatch at first comparison leads to shift of 1. For position j = 1, the second character in the re-ordered pattern, 'g' is matched as shown below:

j	0	1	2	3	4	5	6	7
x[j]	g	c	a	g	a	g	a	g
		c						g

Now we have to calculate the minimum shift to the left so that p[I[0]].... p[I[j-1]] match their characters in original pattern. Here as j=1, its preceding character is only p[I[0]] = c. At this point, we are not concerned if p[I[j]] i.e. g is matching. When we introduce one left shift, c aligns with g in position 0 of the pattern (as shown in table in next page). As these two characters do not match, new pattern is shifted left by one more space. Now, character 'c' oes beyond the beginning of pattern. Therefore, there is no matching ‘c’ to the left of the pattern and we can shift the pattern by 2 places without missing a match. Thus GS[1] = 2.

j	0	1	2	3	4	5	6	7
x[j]	g	c	a	g	a	g	a	g
l_shift =1	c						g	

j	0	1	2	3	4	5	6	7
x[j]	g	c	a	g	a	g	a	g
l_shift =2						g		

Similarly when j = 3, the pattern matched is g at position 5,

j	0	1	2	3	4	5	6	7
x[j]	g	c	a	g	a	g	a	g
l_shift =1		c				g		g

Now, the pattern observed is cgg. We need to calculate the minimum left shift so that ‘cg’ (p[I[0]].... p[I[j-1]]) matches their aligned characters in x[j]

j	0	1	2	3	4	5	6	7
x[j]	g	c	a	g	a	g	a	g
l_shift =1	c				g		g	

j	0	1	2	3	4	5	6	7
x[j]	g	c	a	g	a	g	a	g
l_shift =2				g		g		

After 2 shift a match is found for g, therefore this value is taken to be GS[2]. Similarly the minimum shift for all positions is calculated and is as follows:

j	0	1	2	3	4	5	6	7
GS [j]	1	2	2	2	7	7	7	7

These values are now altered to consider the condition that p[I[j]] should not match.

In order to calculate this, for every position, the pattern is shifted by the minimum shift calculated previously. Then, if p[I[j]] is matching the character in original string the pattern is shifted again till a non match is found. For example for position j = 2, GS[2]=2. When it is shifted by 2 places, p[I[2]] = g is aligned to g in position 3 of the original pattern. Now, the pattern is again shifted left to see if there are any matches. When l_shift = 4, there is again a match. And now the 'g' we are concerned (g at p[I[2]]) is aligned with c. Therefore, there is a mismatch. New GS[2] = 4.

j	0	1	2	3	4	5	6	7
x[j]	g	c	a	g	a	g	a	g
l_shift =2				g		g		

j	0	1	2	3	4	5	6	7
x[j]	g	c	a	g	a	g	a	g
l_shift =4		g		g				

The final good suffix shifts for the pattern are:

j	0	1	2	3	4	5	6	7
GS [j]	1	3	4	2	7	7	7	7

Searching the text:
Once we have calculated the re-ordered pattern, bad character shift and the good suffix shift values, we can search for pattern in the text. We scan the pattern in the order calculated, and when there is a mismatch we shift the pattern by max (bad-char shift, good suffix shift). When there is a match the pattern is shifted by the value of bad-char shift, as it is dependent on the character appearing next in the text.

2. Maximal Shift Algorithm

Maximal shift algorithm is similar to Optimal Mismatch algorithm. The only difference is the order in which the pattern is scanned. In this algorithm, the pattern is scanned from the character that will lead to the maximum shift to the one that gives minimum shift. This will lead to larger shifts at earlier comparisons. The bad-character and good-suffix functions are same as that in Optimal mismatch algorithm.

Sorting pattern according to the length of shift:

First, for each position in the pattern the minimum left shift needed to find a matching character in the pattern is calculated. For example, for 'g' in position 3, there is a matching character 'g' at position 0, when we shift left by 3 places. Once these shifts are calculated, the pattern is re-ordered by placing the characters with maximum shift at the beginning followed by characters leading to smaller shifts.

j	0	1	2	3	4	5	6	7
x[j]	g	c	a	g	a	g	a	g
minShift[j]	1	2	3	3	2	2	2	2
I[j]	3	2	7	6	5	4	1	0
P[I[j]]	g	a	g	a	g	a	c	g

Part 2: Evaluation

Newswire text:

The speed up obtained for different cases is tabulated in table 1 and the time taken by different algorithms for searching different patterns is shown in figure 1.

Table 1: Speedup ratio in different cases for newswire text

Patt Length	Naïve/KMP	Naïve/BM	Naïve/OM	Naïve/MS	BM/OM	BM/MS
5	0.99	0.99	0.98	0.93	1.01	1.02
6	0.89	0.91	1.00	0.95	1.07	1.01
6	0.93	0.97	0.97	1.00	1.00	1.03
8	1.00	1.02	1.09	1.01	1.06	0.98
9	0.93	0.95	1.00	1.02	1.06	1.00
9	1.01	1.04	1.07	1.00	1.05	0.97
14	0.99	1.00	1.11	1.06	1.06	1.00
32	1.02	1.06	1.06	1.00	1.04	1.00

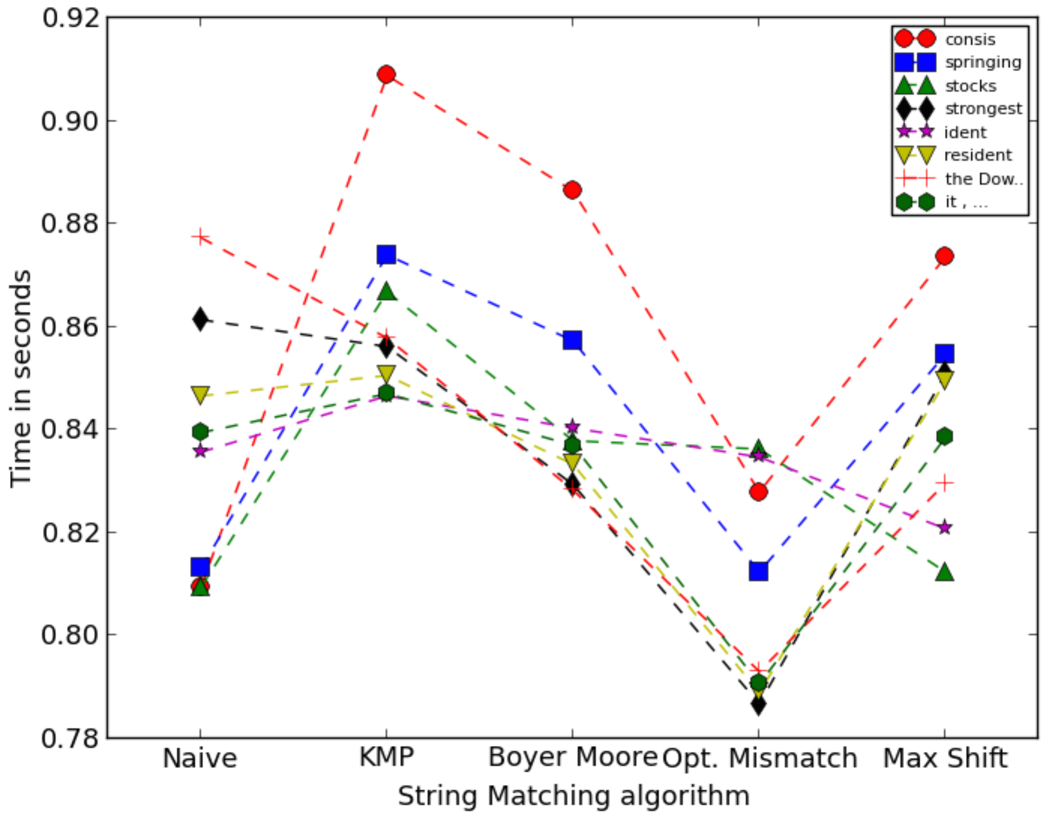

Figure 1. Time taken to search a string by different algorithms in newswire text

In normal newswire text, the length of patterns for searching are comparatively small and the repetitions are less. Knuth-Morris-Pratt takes much longer time in searching small patterns than naïve algorithm. But as the length of pattern increase its performance becomes close to that of naïve or better in a few cases. Boyer-Moore runs faster as the pattern length increases as here the rightmost character is checked first and pattern may be shifted my multiple spaces if there is a mismatch. So longer patterns may lead to longer shifts. Optimal mismatch algorithm almost always outperforms all other algorithms. This can be due to the fact that the alphabet in newswire text is large, and the characters in the pattern are few, so most of the time there would be a mismatch, leading to faster scanning of the document. As optimal shift algorithm is a modified version of Boyer-Moore algorithm, speedup compared to Boyer-Moore algorithm is also considered. It can be seen from table 1 that optimal shift algorithm is always faster than Boyer-Moore in this case. Maximal Shift algorithm also performs better than Naïve and Boyer-Moore in most of the cases. Again this can be due to the order in which the pattern is scanned, larger jumps in initial mismatches.

EMBL Nucleotide Sequence Database:

The time taken by different algorithms for searching patterns in nucleotide sequence database is shown in figure 2. The speed up obtained for different cases is tabulated in table 2.

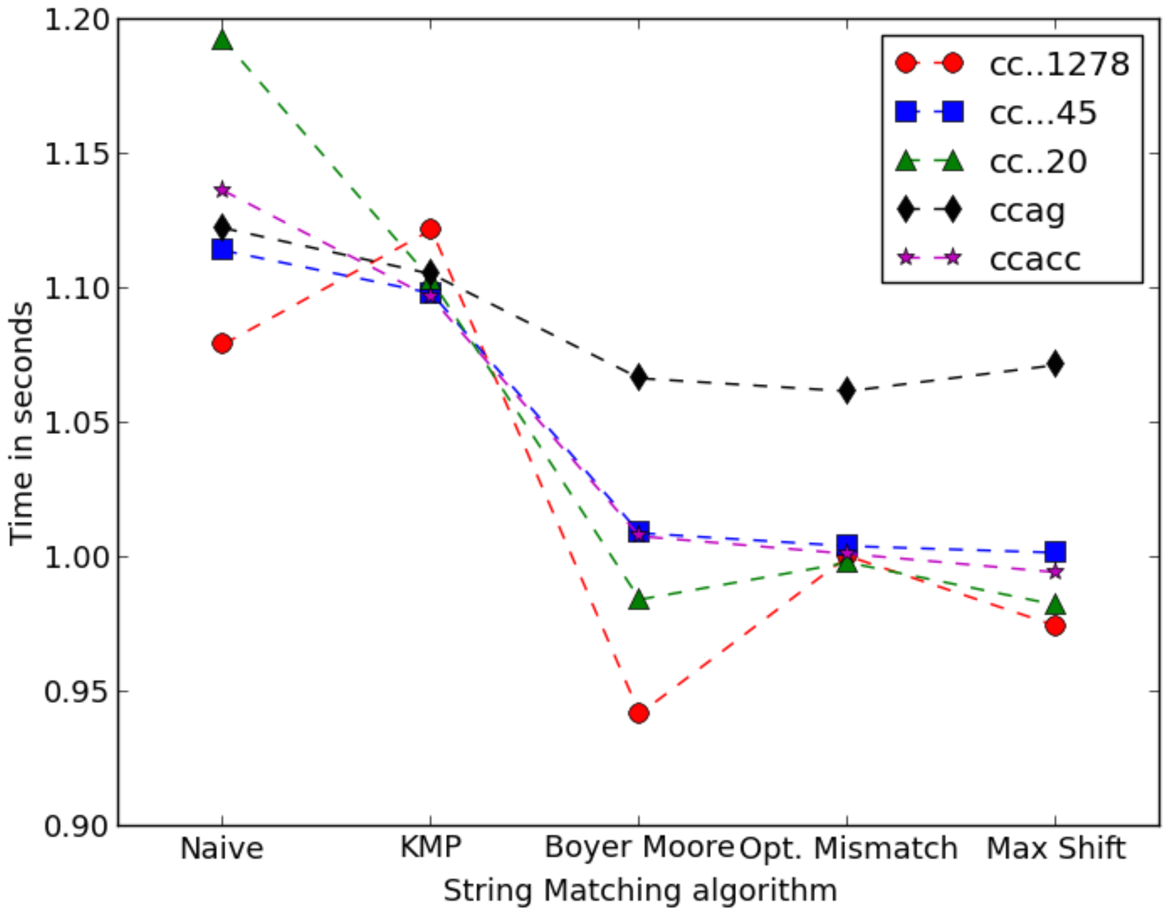

Figure 2. Time taken to search a string by different algorithms in Nucleotide Sequence database

Table 2: Speedup ratio in different cases for Nucleotide Sequence Database

Pat Length	Naïve/KMP	Naïve/BM	Naïve/OM	Naïve/MS	BM/OM	BM/MS
4	1.02	1.05	1.06	1.05	1.00	1.00
5	1.04	1.13	1.13	1.14	1.01	1.01
20	1.08	1.21	1.19	1.21	0.99	1.00
45	1.01	1.10	1.11	1.11	1.00	1.01
1278	0.96	1.15	1.08	1.11	0.94	0.97

In this case all algorithms perform better than the naïve algorithm in all patterns except for one case. For the pattern of length 1278 Knuth-Morris-Pratt algorithm is slower than naïve. As the length of the pattern increases, the speed-up obtained in Boyer-Moore increases and decreases after a level. But still, for the longest sequence Boyer-Moore gave better speedup than any of the other algorithms. The performance of Optimal Mismatch and Minimal Shift algorithm almost become similar to Boyer-Moore or less for such long, small alphabet sequences. This may be because the time taken to compare the pattern in the re-ordered fashion is much more than the gain due to faster mismatches, as here the alphabet is small and even though a mismatch is found faster, the shift might be very small.

Worst-case scenario:

The time taken by different algorithms for searching a pattern in the worst case scenario is shown in figure 3.

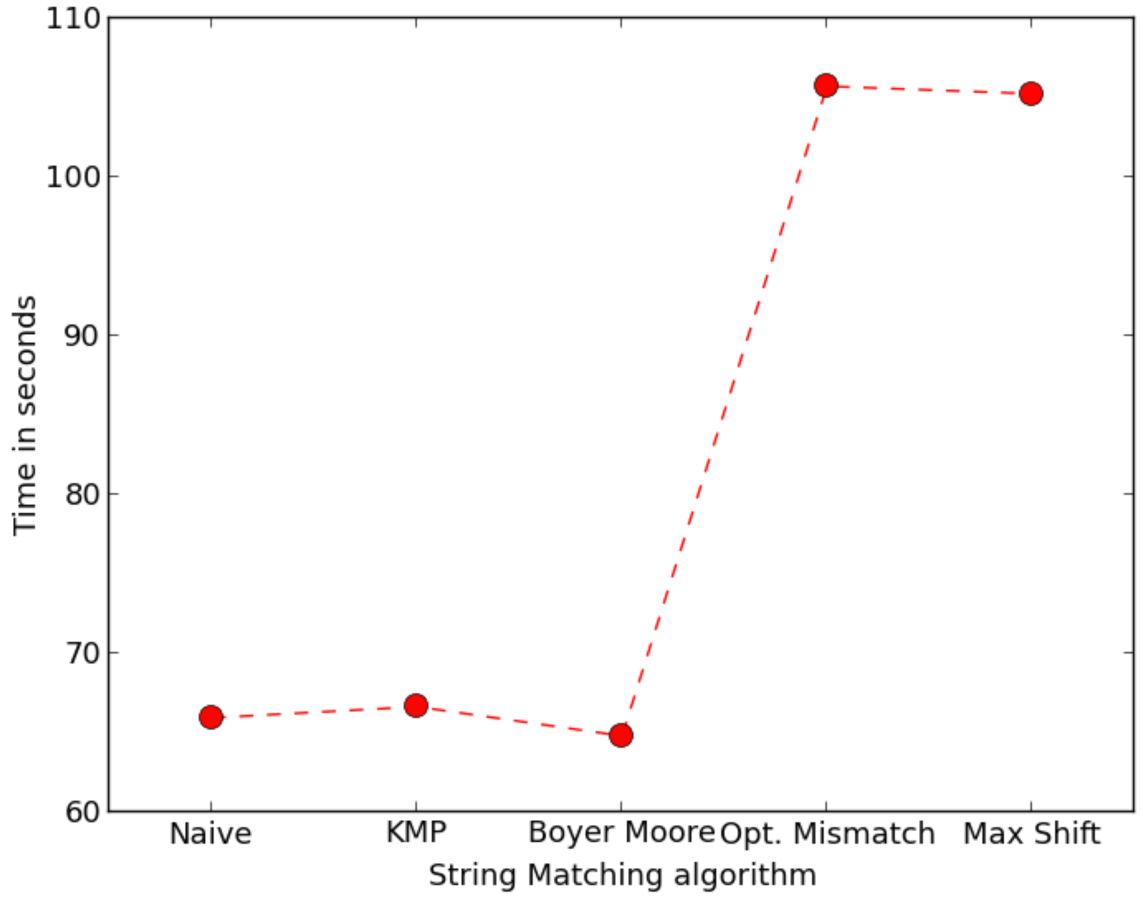

Figure 3. Time taken to search a string by different algorithms in worst case scenario

In the worst-case scenario, performance of Boyer-Moore, KMP and naïve are almost the same. But optimal mismatch and maximal shift algorithms perform really bad as they do not gain anything by re-ordering the pattern, but in turn add a lot of time for shifting forward and backward while comparing the pattern. Both these algorithms have quadratic worst-case time complexity.

Conclusions:

- In average cases optimal-mismatch algorithm seems to be faster than other algorithms.
- For longer patterns, Boyer-Moore performs better than the other algorithms.
- Optimal-mismatch algorithm is better than Boyer-Moore when the alphabet is large as seen in Newswire data
- When the alphabet is small, Boyer-Moore performs better than the rest of the algorithms. It is better than naïve and KMP as it has the required conditions to shift multiple spaces at early mismatches. The reordering of the pattern does not have much effect when the alphabet is small as the next occurrence of the character will be close by and therefore the shifts will be small. The time taken to shift among the pattern is more than that gained my seeing an early mismatch.

References:

1. Sunday D.M., 1990, A very fast substring search algorithm, Communications of the ACM. 33(8) : 132-142.
2. Gusfield, Dan., 1997, Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press.
3. String matching algorithms at http://www-igm.univ-mlv.fr/~lecroq/string/node1.html