CZ2001: Algorithms

AY20/21

Lab Group: SSP1

Lab 1 Report

Group Members:

NG CHI HUI (U1922243C)

LIM JIA EN (U1922802B)

KOO JIAN YANG (U1921728K)

RUSSELL LEUNG (U1920170K)

SUHANA GUPTA (U1923230B)

Introduction

Based on the research of our group, we have chosen to explore the KMP and Boyer Moore Algorithm to reduce the time complexity of the Brute Force Algorithm. Chi Hui, Jia En and Russell were involved in the report writing and research of the algorithm listed in the report. In addition, Suhana, Jian Yang and Jia En wrote the code based on the researched algorithm to supplement the findings. All in all, each member completed the tasks that we had assigned for each other including the PowerPoint slides.

1. Naive String Algorithm

Section 1.1: Code Implementation

For brute force sequence search, we implemented our own version of the code with the concept of brute force. In the implementation, we use a nested for loop to implement comparison of each index. When there is a character match for index 0 in Query in the Sequence, the algorithm will then compare the next index in Query against the next index in Sequence. If there is a mismatch of character, the current 'for-loop' will break and repeat comparison for the next index.

Section 1.2: Time Complexity of Naive String Search Algorithm

In this analysis, we will let N and M indicate the length of characters in Query and Sequence, respectively.

In the best-case scenario for brute force, the first character of Query is not found at all in the entire Sequence. Only comparisons of index 0 of Query occur throughout the entire length of Sequence. The maximum number of comparisons is M. Time complexity of best case is O(M).

The worst-case scenario occurs when all characters in Sequence are the same and Query characters are also the same. However, this is unlikely in our context of DNA. Hence the worst case will be when first N-1 characters are compared and matched except for index N in Query. There will be N number of comparisons for each index in the Sequence. Total number of comparisons will be NM. Time complexity of the worst case is O(NM).

2. Knuth-Morris-Pratt (KMP) Algorithm

Section 2.0: Brief Description of KMP Algorithm

The KMP differs from the brute force algorithm by keeping track of the information gained from previous comparisons. In the KMP algorithm, pre-processing is done in Pattern string P and an array of length 'm' is calculated. The brute force search will start from the beginning of a new alignment once a mismatch is found. However, in KMP function, the search does not necessarily commence from the beginning pattern of P. Although we reference the material we had researched, we came up with the implementation using the main concepts revolving around KMP.

Section 2.1: Code Implementation

Assume we declared a 'store' array. If x = store[i] it would translate that prefix P[0...x] = suffix P[(i-x)...i] where P[i...j] is a representation of a substring from index i to index j of string P.

For simplicity, let T be a string of characters "abcabfijkmn" and let P = "abcabg" as shown in Figure 01.

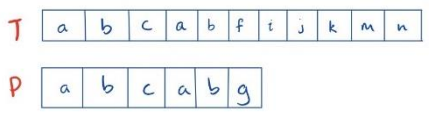


Figure 01: Sequences and Substring

From the first 5 characters of both strings, the characters match. However as shown in Figure 02, the mismatch occurs at index 5.

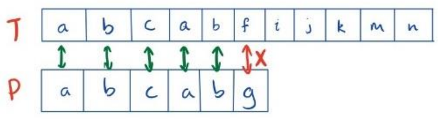
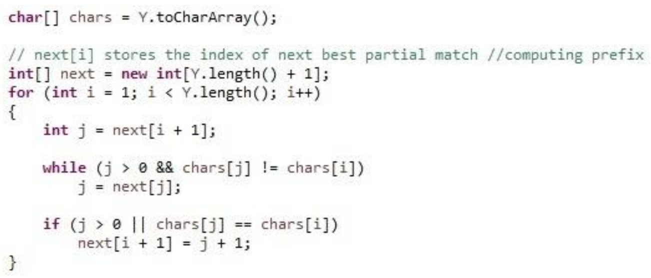


Figure 02: Mismatch

From index 5, the respective characters are ‘f’ and ‘g’ respectively. In other words, the system managed to successfully detect and match up to 4 indexes. Hence, the prefix of Pattern P that matches up to ‘abcab’. Let x be the representation of ‘abcab’ (x = ‘abcab’).

In x, we can see that ‘ab’ is both a prefix and a suffix in x. Therefore, the new search would not be starting from index 1 of T with index 0 of P. From the ‘store’ array we have y = store[4]. We note y = store[4] as 1. By doing that, the search can restart from index y + 1 of P and keep the index pointer in string T unchanged. This means the search will commence from index 5 of string T with index 2 of P. This similar concept is represented in our code in Figure 03. However, instead of declaring a ‘store’ array used in the explanation earlier, we declared an array ‘char[ ] chars = Y.toCharArray( )’ instead.



```
char[] chars = Y.toCharArray();

// next[i] stores the index of next best partial match //computing prefix
int[] next = new int[Y.length() + 1];
for (int i = 1; i < Y.length(); i++)
{
    int j = next[i + 1];

    while (j > 0 && chars[j] != chars[i])
        j = next[j];

    if (j > 0 || chars[j] == chars[i])
        next[i + 1] = j + 1;
}
```

Figure 03: Computing Prefixes and Suffixes

Section 2.2: Time Complexity of KMP algorithm:

Let n = length of the text and m = the length of the substring. It runs in O(n) time to find all occurrences of ‘m’ in ‘n’. However, we should also note the pre-processing (construction of the prefix and suffix table) will take twice the length of the pattern [2*length(m)] steps but not exceeding 2m-2 iteration. which can be represented by O(m). Therefore, the time complexity of the algorithm is O(n). In addition, factoring in the construction of the table the overall complexity would be O(m+n).

3. Boyer-Moore

Section 3.0: Brief Description of Boyer-Moore Algorithm

Boyer-Moore starts matching from the last character. Here, we implemented the Bad-character Heuristic as a group using the concept of Boyer-Moore we had researched and written on our own.

Section 3.1: Code Implementation

The character of the text which does not match with the current character of the pattern is called the Bad Character. In the following implementation, we pre-process the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern). Upon mismatch, we shift the pattern until –

1. The mismatch becomes a match
2. Pattern P moves past the mismatched character.

Case 1 – Mismatch becomes a match:

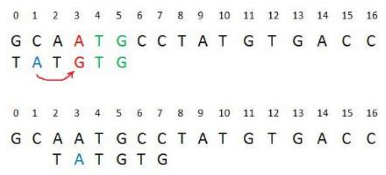


Figure 04: Mismatch becomes a match

In the above example, we got a mismatch at position 3. Here our mismatching character is "A". Now we will search for the last occurrence of "A" in pattern. We got "A" at position 1 in pattern (displayed in Blue) and this is the last occurrence of it. Now we will shift the pattern 2 times so that the "A" in pattern gets aligned with the "A" in text.

Case 2 – Pattern move past the mismatch character:

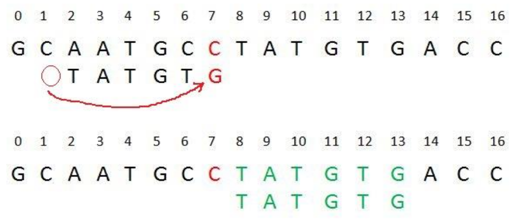


Figure 05: Pattern moves past the mismatch character

Here we have a mismatch at position 7. The mismatching character "C" does not exist in pattern before position 7 so we will shift pattern past to the position 7 and eventually in the above example we have got a perfect match of pattern (displayed in Green). We are doing this because, "C" does not exist in pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.

Section 3.2: Time Complexity of Boyer-Moore

Assume that the length of the pattern is M, and the length of text is N.

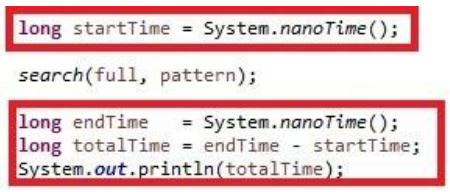
Best case scenario will be if there is no occurrence of mismatched character in the text in the pattern. So, the pattern need not compare with the M number of characters that it is aligned to currently. Thus, the pattern will be able to shift by M every time. Which means that the best case is O(M/N).

The worst-case scenario will be if the pattern consists of only a single type of character and the text also consists of the same single type of character. In this case, no matter where the pattern is, it would align with the text. At every point, it will never meet a mismatch, so it does not need to check where the last occurrence of a mismatch character is. It will only be able to move 1 step each time and check every character. Thus, this is no different than the time complexity of the naive method. Therefore, the worst-case scenario is O(MN).

Conclusion

The implemented algorithms proposed are improvements for the brute-force algorithm. They have similar characteristics such as pre-processing the substring and string before executing the actual search.

Through the experimental demonstration of string matching processing, we used 'System.nanotime( )' to calculate the runtime for each algorithm against 3 different datasets with the same substring pattern as shown in Figure 06.



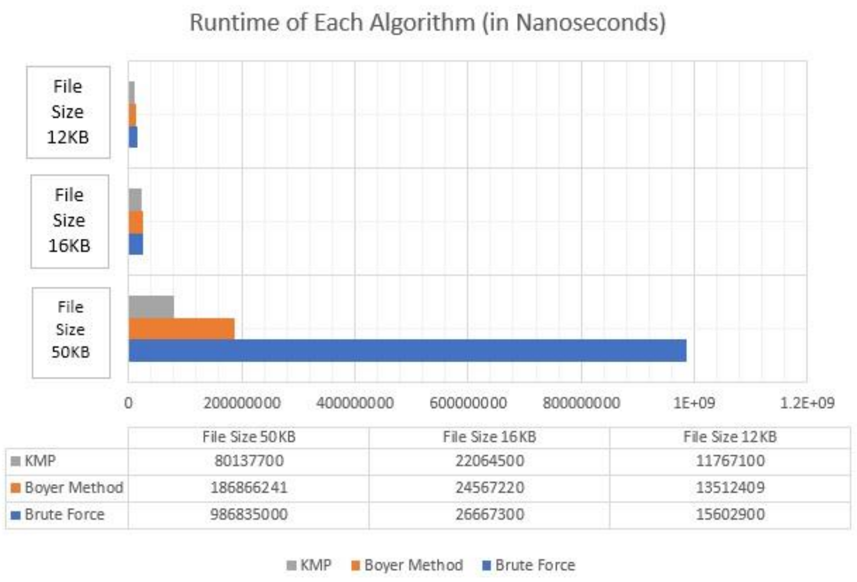
```
long startTime = System.nanoTime();

search(full, pattern);

long endTime   = System.nanoTime();
long totalTime = endTime - startTime;
System.out.println(totalTime);
```

Figure 06: Calculation of Runtime [System.nanotime( )]

From the graph our team had plotted in Figure 07, the efficiency of KMP is evident compared to the other 2 other algorithms.



	File Size 50KB	File Size 16KB	File Size 12KB
▨ KMP	80137700	22064500	11767100
▨ Boyer Method	186866241	24567220	13512409
▨ Brute Force	986835000	26667300	15602900

Figure 07: Runtime of Each Algorithm

The proposed algorithms have reduced the time-complexity of the brute force search algorithm. In worst case scenario, the brute force time-complexity is O(n*m) whereas the best proposed algorithm (KMP Algorithm) for the worst case is O(n+m).

Reference Materials

Regnier, M., 2020. *Knuth-Morris-Pratt Algorithm: An Analysis*. https://link.springer.com/chapter/10.1007/3-540-51486-4_90. [Accessed 6 September 2020]

Medium. 2020. *Kunth-Morris-Pratt (KMP) Algorithm For Pattern Searching*. [online] Available at: https://medium.com/@avishekp86/kunth-morris-pratt-kmp-algorithm-for-pattern-searching-552661788e6b [Accessed 6 September 2020].

Iopscience.iop.org. 2020. *Shieldsquare Captcha*. [online] Available at: https://iopscience.iop.org/article/10.1088/1742-6596/1345/4/042005/pdf [Accessed 8 September 2020]

Mathcs.emory.edu. 2020. [online] Available at: http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Text/Matching-Boyer-Moore1.html [Accessed 5 September 2020].

Tutorialspoint.com. 2020. *Bad Character Heuristic*. [online] Available at: https://www.tutorialspoint.com/Bad-Character-Heuristic#:~:text=The%20bad%20character%20heuristic%20method,approaches%20of%20Boyer%20Moore%20Algorithm.&text=When%20the%20mismatch%20has%20occurred,moves%20past%20the%20bad%20character [Accessed 8 September 2020].

GeeksforGeeks. 2020. *Boyer Moore Algorithm For Pattern Searching - Geeksforgeeks*. [online] Available at: https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/ [Accessed 10 September 2020].