NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Algorithms Challenge

Project 1: Searching Algorithm Report

Done By: Team SSP1-2

Chong Jing Hong

Tang Kai Wen, Alvin

Muhammad Nasran Hamza

Aaron Yap Jia Cheng

Ashwin Kurup

1. Analysis of Brute Force Sequential Search (Naïve String Search)

```
BruteForceStringMatch(text[0...n-1], pattern[0...m-1]) {
    result[ ] = [ ]
    for i ← 0 to n-m do
            j ← 0
            while j < m and pattern[j] = text[i+j] do
                    j++
            if j = m then append to result
    return result[ ] }
```

1.1 Best Case Analysis

The best case occurs when the pattern is not found in the text.

Example:
text = "AABCCAADDEE"
pattern = "FFF"

If the pattern is not found in the text, the for loop will run for n-m times. Thus, the time complexity of the best case is Θ(n).

1.2 Worst Case Analysis

The worst case occurs when:

1. When all characters of the text and pattern are the same.

   Example:
   text = "AAAAAAAAAAAAAAAAAA"
   pattern = "AAAAA"

   Since all of the characters in the pattern and the text are the same, the for-loop will run (n-m+1) times, and for each for-loop, the while-loop will run for m times. Hence, the time complexity of the worst case is O((n-m+1)*m) or Θ(mn).

2. The only last character is different.

   Example:
   text = "AAAAAAAAAAAAAAAAAB"
   pattern = "AAAAB"

   Since only the last character is different, the pattern will only be found in the text after (n-m+1) for-loops and m while-loops within each for-loops. Hence, the time complexity of the worst case is O((n-m+1)*m) or Θ(mn).

2. First-Last Pattern Matching (FLPM)

This algorithm is referenced from "Simple and Efficient Pattern Matching Algorithms for Biological Sequences" (Neamatollahi, Hadi, & Naghibzadeh, 2020). First-Last Pattern consists of 2 phases: Firstly, the pre-processing of the text and secondly, and the matching of the text with the pattern.

2.1 Pre-processing Phase

During the pre-processing phase, the first and last characters of the pattern are searched for in the text. The pre-processing phase also checks whether the two characters are separated with m-2 characters between them, where m is the length of the pattern. If the characters in the text satisfies the 2 conditions, the index of the characters will be stored in a window.

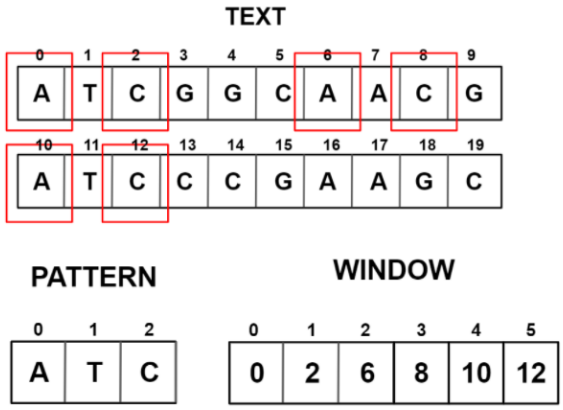

Figure 1

In figure 1, the first and last characters of the pattern are "A" and "C". The algorithm finds these 2 characters at index 0, 2, 6, 8, 10 and 12 in the text and stores these indexes into a window.

2.2 Matching Phase

During the matching phase, the remaining characters in the pattern, *P[1…m-2]*, are matched with the characters in-between the start indexes in the window, $T[W_i+1 \ldots W_i+m-2]$, found in the previous phase. If all the characters in-between match, the algorithm returns the start index where the pattern is found in the text.

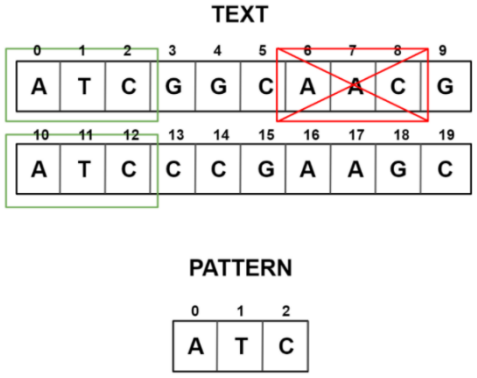

Figure 2

3. Analysis of FLPM

```
FLPM(text[0...n-1], pattern[0...m-1]) {
    //pre-processing phase
    for i ← 0 to n-m do
            window_index = 0
            if text[i] = pattern[0] and text[i+m-1] = pattern[m-1]
                    window[window_index] ← i
                    window_index ← window_index + 1
    //matching phase
    if window_size > 0
            for j ← 0 to window_size -1 do
                    x = window[j]
                    for y ← 1 to m-2
                    if pattern[y] != text[x+y]
                            break;
                    if y = m-2 then append j to window
    return window
```

3.1 Best Case Analysis

The best case occurs when the first and last characters of the pattern are not found at all in the text.

Example:
text = "EFGHIJKLMNOP"
pattern = "ABC"

As none of the characters in the text correspond to the first and last characters of the pattern, the size of the window will be 0 and the matching phase of the algorithm will not be executed. Thus, the time complexity will be O(n-m+1).

3.2 Worst Case Analysis

The worst case occurs when the text is filled with the pattern.

Example:
text = "ABCABCABCABCABC"
pattern = "ABC"

The algorithm will firstly run n-m+1 times to store all the indexes of the first and last characters in the window. Since the text is filled with the pattern, matching phase will run w * (m-2) times, w being the window size, to match the characters in between. Thus, the time complexity is O(n-m+1+w*(m-2)).

3.3 Average Case Analysis

We did not do an average case analysis because it is too complex to derive. Firstly, we would have to make assumptions to the probability of each individual character, “A”, “C”, “T” and “G”, appearing in the genome sequence. This assumption may not be true, hence, building an average case analysis around this assumption may not be meaningful. Secondly, we would have to consider the cases where the sequence does not appear, the sequence appears once, and the sequence appears multiple times. The probability of these may differ for different genome type. Hence, it is too complex to derive an equation from these cases.

4. Findings and Observations

1. FLPM is much faster than Brute Force Search as FLPM limits the number of queried sequences in the genome to those that are more likely to match with the user's input. This was confirmed when we conducted multiple tests to compare the speed of both algorithms as shown in figure 3. As the file size increased, Brute Force Search became increasingly slower. However, the size of the file we were only able to test to is 546MB.

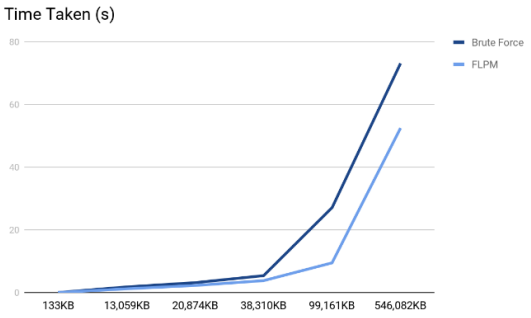

Figure 3

2. We also conducted tests to compare the speed of the FLPM algorithm when using for loops versus using while loops in Python. Figures 4 and 5 show the average time taken for the algorithms to run in the corresponding conditions.

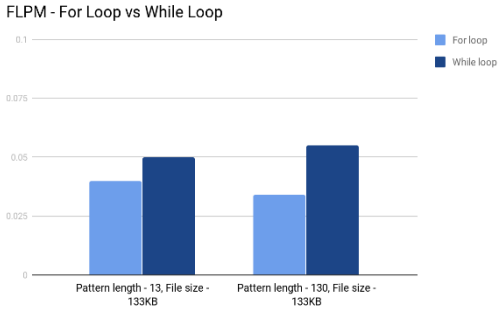

Figure 4

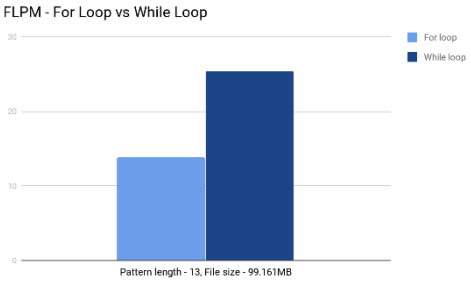

Figure 5

For loop is faster than the while loop in Python for all 3 comparisons because it comprises of 3 operations whereas the latter comprises of 10 operations in Python (Why is looping over range() in Python faster than using a while loop?, 2009).

3. We faced 2 problems when extracting the data from the .fna files:
   1. The first line is redundant and not part of the sequence.
   2. The sequences are separated by line breaks that are picked up by the scanner.

   Therefore, we have decided to remove the first line and all the line breaks

5. References

Neamatollahi, P., Hadi, M., & Naghibzadeh, M. (2020, January 23). Simple and Efficient Pattern Matching Algorithms for Biological Sequences. doi:10.1109

Why is looping over range() in Python faster than using a while loop? (2009, May). Retrieved from Stack Overflow: https://stackoverflow.com/questions/869229/why-is-looping-over-range-in-python-faster-than-using-a-while-loop

Contributions:

The pseudocode as well as the best and worst analysis of the brute force method was done by Jing Hong and Alvin. The first implementation of the brute force method was done by Nasran while the first-last pattern matching was by Ashwin. Initial testing and improvements of the algorithm was done by the whole team. The application was then further improved in terms of user friendliness and error handling by Alvin and Aaron. The report write-up was done by Jing Hong and Aaron. Jing Hong, Ashwin, and Alvin presented for the project as well as the demonstration of the program.