Project 1 : Searching Algorithm

CZ2001 Algorithm

Lab/Group: SSP3 Group 1

Ang Shu Hui	U1922145K
Bachhas Nikita	U1921630F
Kundu Koushani	U1922997B
Leonardo Irvin Pratama	U1920301J

Nanyang Technological University

Singapore, September 2020

In this report, two pattern-searching algorithms will be discussed in terms of algorithm process, time complexity, and comparisons to other algorithms when one searches for the substring occurrences in the given main string.

Algorithm Number 1: Finite Automata (FA Algorithm)

1. Introduction

Finite Automata is an algorithm set to recognize patterns within a given sequence. In this algorithm, we need to initially pre-process the algorithm and then build a *2D Finite State Automaton array*. The next step is the searching step, which consists of states for each character in the pattern and certain conditions must be met to move forward or backward in the states. It uses the concept of the state machine.

2. Pre-processing Stage

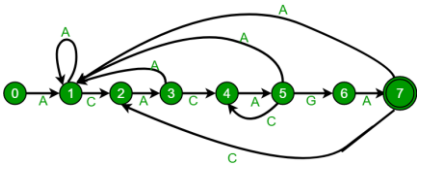
The pre-processing stage consists of 2 parts: first finding the states for each character in the pattern and computing the finite automata.

state	character A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

(i) Finding States: Determine the following state k according to the last occurrence of a similar pattern in the substring. For instance, if the substring is "ACACG" and the main string is "ACACACG". Although we have reached the state 4 in the character highlighted in red and the following character is not a 'G', we do not come back to state 0, but instead, proceed with state 3 because it is the start of a new possible substring "ACACACG".

(ii) Computing Finite Automata: Building of the array that represents every possible combination of the substring (as seen in the figure on the right).

3. Searching Stage



Consider the following example: we are required to find a sequence, ACACAGA, from a sequence of DNA. The algorithm first scans the code from the start until it matches with the letter A. State 0 is the default state of the counter. State 1 is achieved once letter A is found in the DNA sequence. If the next letter in the sequence matches the 2nd letter in the pattern, letter C, state increases from 1 to 2. If the next letter does not match the 2nd letter in the patter, the state may either remain at 1 (if it is a letter A) or fall to 0 (if it is a letter G or T). On the figure above, the arrows represent the destination state if the following letter is the one on the arrow. If the letter is not mentioned in the graph, it means that the state returns to 0.

The states increase or decrease along with the sequence. A full pattern is found when the number of the state reaches its final state, which is state number 7 in this example. The number of states in this algorithm will be m+1 (state 0 to state m) where m is the length of the pattern to be found. Every time the letter in the substring matches with the main string, the state will increase by 1.

4. Time Complexity

The time complexity of the search process is O(n), where n is the length of the main string.

The time complexity to construct the Finite State Automaton array is highly dependent on m, the length of the substring, and k which is the total number of possible characters in the pattern and sequence: O($k * m^3$)

However, since we know that there can only be 4 possible characters, A, C, G, and T, in the DNA sequence and in the pattern to be found, the time complexity of the construction of FA is O($4 * m^3$).

Hence, the total time complexity for the algorithm is O(n + $4m^3$). *m is not very significant in most cases.

This time complexity applies to all possible cases. Regardless of the combination, the pre-process will iterate through every possible combination of the 4 characters. In addition, the main process will iterate through every character, check the following state of the character, and compare whether the state number is equal to the length of the substring.

Best-case time complexity	*Worst-case time complexity*	*Average time complexity*
O(n) + O($4 * m^3$)	O(n) + O($4 * m^3$)	O(n) + O($4 * m^3$)

Algorithm Number 2: Boyer Moore Horspool Algorithm

1. **Introduction**

   The Boyer Moore Horspool algorithm is a simplified version of the Boyer Moore algorithm using only the bad character rule. Similar to the Finite Automata algorithm, Boyer Moore Horspool algorithm will pre-process the pattern initially and create an array. This algorithm then starts matching the given sequence with the pattern from the last character of the pattern.

2. **Pre-processing Stage**

   The pre-processing stage of Boyer Moore Horspool will find *the last occurrence of every letter in the substring*. In the case of substring 'ACTGACCC', the Bad Character Heuristic array will look as follow,

```
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, 4, -1, 7, -1, -1, -1, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 2, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1]
```

   The array has 256 indexes which comprise all the possible characters in the UTF-8 encoding. If a letter never appears in the substring, the value of array[ord(letter)] is -1. As seen above, we can interpret from the array that A is last seen in index 4, C is last seen in index 7, G is last seen in index 3, and T is last seen in index 2.

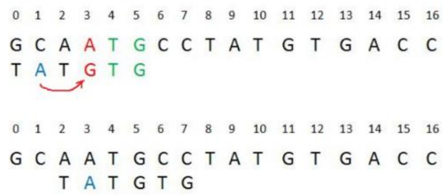
3. **Searching Stage**

   When the last character in the pattern matches the corresponding character of the given sequence, the previous character in the pattern will be compared similarly. After all characters are compared and there is a complete match between them, the full pattern is found.

   If a mismatch occurs, there will be two different cases: mismatched character exists in the pattern and complete mismatch where the mismatched character does not exist in the pattern.

   <u>Case 1: Mismatched character exists in the pattern</u>

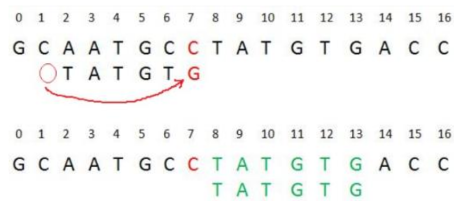
   In this case, the algorithm will search through the pattern for the last occurrence of the character. If the mismatched character exists in the pattern, the pattern will be shifted such that the character in the pattern aligns with the mismatched character.



   For example, if we are querying "TATGTG" in a sequence above, there is a mismatch found at index 3 where the character 'A' does not match with the corresponding character 'G' in the pattern. The algorithm then will search through the pattern. Since character 'A' exists in the pattern, the pattern will be shifted to the right such that at index 3, the characters match.

   <u>Case 2: Complete mismatch where the mismatched character does not exist in the pattern.</u>

   Similarly, the algorithm will search through the pattern again. If the mismatched character does not exist in the pattern, the algorithm will shift the pattern past the mismatched character.



   For example, in the figure, there is a mismatch at index 7. The character in the given sequence at index 7 is 'C' while that of the pattern is 'G'. The algorithm will initiate a search to find the last occurrence of 'C' in the pattern. If 'C' does not exist in the pattern, the pattern will be shifted to the right such that the entire sequence from index 0 to 7 is skipped.

4. **Time Complexity**

   The best case happens when the sequence does not contain any character from the pattern. In this case, the searching would always shift the letters by m.

   This way, the algorithm only requires iterating through n/m letters.

   Best-case analysis: $O(\frac{n}{m})$

$$\begin{aligned} \text{Time complexity} &= 5 + \left(\frac{n}{m} - 1\right)(2) \\ &= 5 + \frac{2n}{m} - 2 \\ &= O(\frac{n}{m}) \end{aligned}$$

The worst case happens when all characters in the pattern match with the sequence given. Hence, the algorithm will compare every character in the pattern, resulting in n times of comparison. The pattern will only be shifted by one index each time since the next character in the sequence given also matches. There will be a total of m shifts. For each shift, n times of comparison are carried out. Hence, the time complexity would be O(mn).

Index	0	1	2	3	4	5		m-5	m-4	m-3	m-2	m-1
Sequence (of length m)	T	T	T	T	T	T	...	T	T	T	T	T
Pattern (of length 4)	T	T	T	T								

An example is given above. We want to find the pattern 'TTTT' in the sequence of length m with all characters being 'T'. The algorithm will compare the characters starting from index 3. Since all characters match, the algorithm will carry out 4 comparisons in total.

After the first occurrence of the pattern is found, the pattern will be shifted by one index to the right and the algorithm will search through the sequence for the pattern. This example will give a time complexity of 4m.

Worst-case analysis: $O(mn)$

Notice the fact that if a letter in the main string does not exist in the substring, we can skip every possible combination that includes this one letter. Instead of 4 letters, if there are 256 types of characters in the main string, the probability of not having one of the types in the substring is much larger compared to 4 types. Hence, the average-case analysis depends on the possible type of characters allowed in the string structure.

Average-case analysis: $O(\frac{n}{4})$

Best-case time complexity	*Worst-case time complexity*	*Average time complexity*
O(n/m)	O(mn)	O(n/4).

Comparison of the Brute Force / Naïve Algorithm

1. **Introduction**
   The Brute Force (Naive) algorithm shifts the pattern over text one by one and checks for a match. If a match is found, then slides by 1 again to check for subsequent matches. Then, it will iterate through all possible m-letters combination in the main string, where m is the length of the substring.
   Therefore, it is not surprising that this algorithm is very slow and time–consuming as compared to the two algorithms mentioned above (i.e. Finite Automata and Boyer Moore Horspool), especially when searching for a pattern in a long genome sequence.
2. **Comparison to the Finite Automata Algorithm**
   Unlike Brute Force (Naïve) algorithm which checks every possible m-letters combination from the main string, Finite Automata checks every letter without having to shift backward. In the Brute Force algorithm, every letter may be checked multiple times, up to m times, whereas in the Finite Automata algorithm, every letter is strictly checked exactly once. This is possible due to the pre-processed array that resembles the finite state of every possible letter combination.
3. **Comparison to the Boyer-Moore Horspool Algorithm**
   Unlike Brute Force (Naïve) algorithm which requires the checking method to be done by shifting one letter at a time, Boyer-Moore Horspool is able to shift the searching by at most m letters. This way, if a letter that does not exist in the substring appears in the main string. We are certain that every possible combination that includes this one particular letter would not consist of our required substring.

Best-case time complexity	*Worst-case time complexity*	*Average time complexity*
O(n)	O(nm)	O(n+m).

Final Evaluation

1. **Time Complexity Comparison**

   Here are the average runtimes of *multiple algorithms* on *different inputs*. The test-bench used also has *two different file sizes*, which compare the algorithm's effectivity on strings of multiple lengths.

4 MB	ACTG	AAAAAAAAAA	CAAGTGCTATGTACAGCTGT
Brute Force (Naïve)	2.214 s	2.213 s	2.326 s
Finite Automata	0.550 s	0.529 s	0.571 s
Boyer Moore Horspool	1.420 s	0.263 s	0.993 s

92 MB	ACTG	AAAAAAAAAA	CAAGTGCTATGTACAGCTGT
Brute Force (Naïve)	46.604 s	48.738 s	48.121 s
Finite Automata	12.186 s	11.276 s	11.913 s
Boyer Moore Horspool	30.986 s	5.505 s	16.922 s

   Based on the table above, there are some interesting points we can extract:

   a. Finite Automata and Boyer Moore Horspool algorithm both outperform the Brute Force (Naïve) algorithm in terms of execution time.
   b. In the second test case where the input only consists of one letter, Boyer Moore Horspool algorithm significantly outperforms the other two. This happens because a lot of letter combinations that include 'C', 'T', 'G' will be automatically skipped as it is not contained in our substring/input.
   c. Interestingly for Boyer Moore Horspool and Finite Automata, the execution time remains stable for inputs of different sizes. The reason behind this is the length of our current input is rather short, we may see some differences should the input consist of more than 100 characters. In the case of Brute Force (Naïve), longer inputs mean a lesser number of iterations required. Hence, longer inputs do not significantly change the execution time.
   d. The worst case of any possible algorithms may not be lower than n ($\Omega(n)$). At the very least, we need to check every letter in the main string on cases where the substrings appear in every index of the main string (For instance, main string = 'CCCCCCCC' and substring = 'CCCC').

2. **Conclusion**

   a. Every algorithm has its own advantage and disadvantage. Some are suitable to be used in a certain condition, while some others are better at other conditions.
   b. Finite Automata (similar to the Knuth-Morris-Prath algorithm) is suitable for a general pattern searching condition. It has a stable time complexity that does not differ regardless of given main and substrings.
   c. Boyer Moore Horspool algorithm, on the other hand, is suitable on a search whose substring only consists of very few types of characters. Only in this case, we would be able to skip a lot of letters that are not included in the substring. We must be extremely careful as this algorithm would run at a similar time to Brute Force (Naïve) algorithm if it is used for its worst-case as explained above.

References

Debjit Dutta, Shrikant Pandey, Bhupendra Rathore. (2019). Finite Automata algorithm for Pattern Searching. Retrieved from https://www.geeksforgeeks.org/finite-automata-algorithm-for-pattern-searching/

Chitra Nayal, Shikha Singh, Bhupendra Rathore, Kirti Mangal, Nidhi. (2019). Boyer Moore Algorithm for Pattern Searching. Retrieved from https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

Sharon Christine. (2018). Naive Pattern Searching. Retrieved from https://www.tutorialspoint.com/Naive-Pattern-Searching

UC Irvine. (1996). ICS 161: Design and Analysis of Algorithms. Retrieved from https://www.ics.uci.edu/~eppstein/161/960227.html

Statement of Contribution

Name	Contribution
Ang Shu Hui	- Boyer-Moore-Horspool algorithm analysis (report) - Boyer-Moore-Horspool algorithm demonstration (presentation)
Bachhas Nikita	- Finite Automata algorithm analysis (report) - Comparisons between multiple algorithms (presentation)
Kundu Koushani	- Brute Force (Naïve) algorithm comparison (report) - References formatting (report) - Comparisons between multiple algorithms (presentation)
Leonardo Irvin Pratama	- Final Evaluation (report) - Adding missing details to the final report (report) - Code implementation of algorithms in Jupyter Notebook (submission) - Finite Automata algorithm demonstration (presentation)