NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CE2001 Algorithms

Project 1

(Add in names and matriculation no.)

In this report, we propose 2 search algorithms and compare its time and space complexity with the brute force algorithm. The 2 search algorithms are Knutt-Morris-Pratt(KMP) and Index Jump algorithms. These algorithms take in 2 inputs, the genome to be searched and the DNA/protein sequence we are looking for.

1.Brute Force

Code implementation

The Brute force algorithm imports the genome (m) and DNA/Protein string (n) as a string array. It then implements a for loop which generates a substring of length n starting from the first index of m. It then does a comparison operation with the target string. If a match occurs, the starting index of the match is added into an array. The code loop will then repeat for the length of genome - pattern length + 1

Time Complexity Analysis

Let n be length of text, p be length of pattern. Firstly, we note that after the index n-p+1, no comparisons are needed. Each character in the text are compared p times. Therefore, the total number of comparisons are (n-p+1)p.

It is a function of n and p, which is

$$f(n,p) = (n - p + 1)p = -p^2 + np + p$$

Assuming n is fixed in this case,

$$f(p) = -p^2 + np + p$$

The **best case** is when p = 1 and p = n,

$$f(1) = n,\ f(n) = n$$

The **worst case** is when $\frac{df(p)}{dp} = 0 - 2p + 1 + n = 0 p = \frac{n+1}{2}$

When $p = \frac{n+1}{2}$, $f(\frac{n+1}{2}) = \frac{(n+1)^2}{4}$

In order to find the **average case**, we first assume that the probability of the value of p is the same,

$$P(p = 1) = P(p = 2) = P(p = 3) \ldots\ldots = P(p = n) = \frac{1}{n}$$

The average case is

$$\sum_{p=1}^{n} \frac{1}{n}(n + 1 - p) = \sum_{p=1}^{n} p + \frac{1}{n}\sum_{p=1}^{n} p - \frac{1}{n}\sum_{p=1}^{n} p^2 = \frac{n(n+1)}{2} + \frac{n+1}{2} - \frac{(n+1)(2n+1)}{6} = \frac{(n+1)(n+2)}{6}$$

Therefore, the time complexity of all cases is as below:

Best Case: O(n)
Average Case: $O(n^2)$
Worst Case: $O(n^2)$

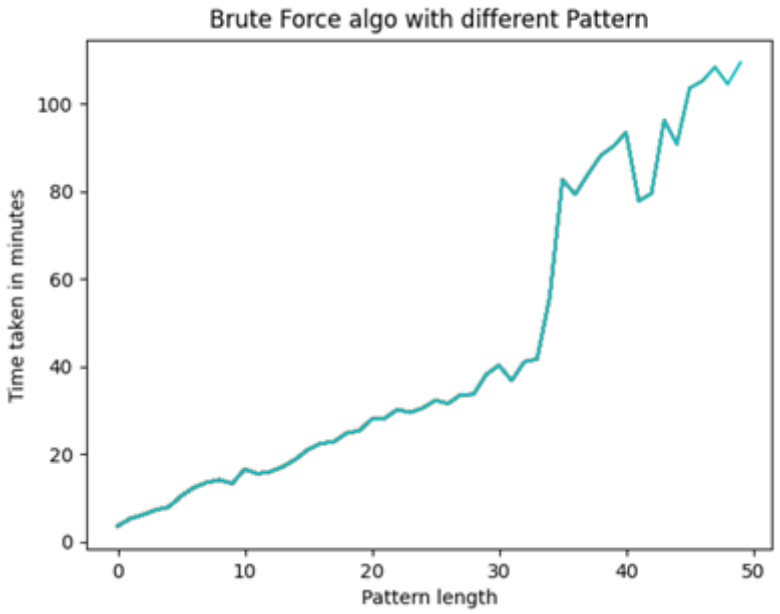


Fig. 1

To test whether the time complexity holds true, we tried to find the time taken under 2 conditions. One of them is to see how the time taken for the algorithm changes with respect to the length of the input string. And the other experiment is to identify the effect of the length of the pattern string on the time taken for the algorithm. From Fig. 1 we are able to see, when the pattern size or when the text file increases in size the time taken for the algorithm increases.

2. Knuth-Morris-Pratt(KMP)

Code Implementation

KMP is a string matching algorithm which tries to minimize the number of times manual matching is done unlike the brute force algorithm. The algorithm requires preprocessing and creates a failure function (lps[]), an array of the same length as the text. The failure function is understood as the longest prefix which is also a suffix. For ex. The lps array for the pattern “abcaby” would be “000120.” Ín this case, “ab” would be the suffix/prefix.

The algorithm will then do a comparison between the pattern and the text. Let’s consider the text “a b x a b c a b c a b y ” and the pattern “a b c a b y.” The first two

characters match. When there is a mismatch, the algorithm will look for the next point to compare the text and pattern. To do that, it will check the character before the mismatch character in lps array and check the lps array ‘’value’’. The comparison will then start from index “value” of the pattern.

In the above example, ‘’a’” , “b” match between the pattern and the text. “C’” and “x” do no match, the character before “c” is “b” in the pattern and the lps value for “b” is 0. So now, the comparison will start from index 0 of the pattern. “A” and “x” will be compared. ‘’A’ and “x” do not match, now the comparison will be between the next character of the pattern and the text which is “a” and “a.” The next four characters match between the pattern and the text. Now, “c” and “y” do not match. The algorithm will check the lps array value for the character before “y” in the pattern. It is 2. This 2 indicates that the prefix and suffix is of length 2 and the prefix “ab” should be present in the text. The algorithm skips the first two characters (“ab”) of pattern and starts the comparison from “c” in the pattern. This is how the KMP algorithms reduces the number of comparisons done in contrast to brute force algorithm.

Time Complexity Analysis

For preprocessing, each character in the pattern is compared once. Therefore the time complexity for deriving the failure function is O(p) where p is the length of the text. As this step is necessary and crucial, the time cannot be ignored.

The **best case** is when every single character in the text is only compared once.
Eg.Text: AGCT AGCT AGCT AGCT AGCT Pattern: AGCT

The **worst case** is when every single character is compared p times. Eg: Text: AAAA AAAA AAAA AAAA AAAA Pattern: AAA.

The following conditions were observed when the number of comparisons is equal to 2:
(i) The previous character must be a match

(ii) The targeted character must be a mismatch

When the length of text is large,
P(match) = 25% and P(mismatch)=75%

To find the **average case**, the following assumptions were made:
(i) Assume that the matches and mismatches are distributed evenly

(ii) Assume that the pattern is not repeated consecutively in the DNA sequences.

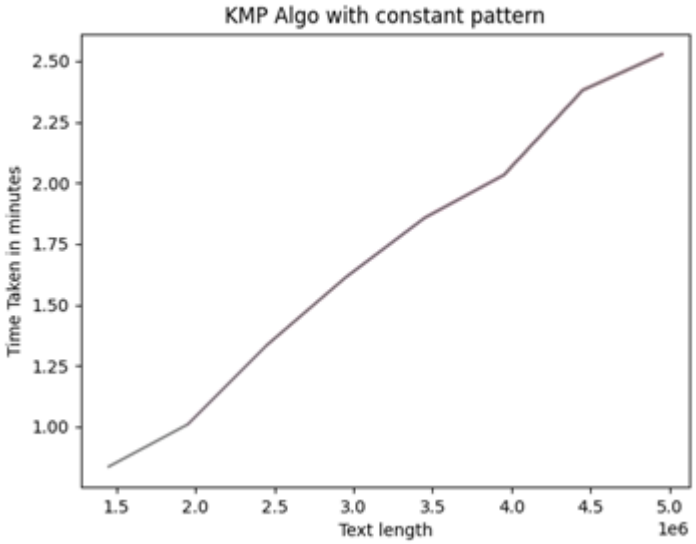
Every four characters will have 1 match and 3 mismatches. These 4 characters will contribute to 1 extra comparison. Therefore the total number of comparisons will be as follows:
N + N/4 = 5N/4

Therefore, the time complexity of index jump algorithm in all cases is as below:

Best Case: N comparisons. O(n) + O(p) = O(n+p)

Average Case: O(n)+O(n) = 2N comparisons. O(np) + O(p) = O(np+p)

Worst Case: $O(n^2)$+O(n) = 5N/4 comparisons. O(n) + O(p) = O(n+p)



3. 2-JUMP

This algorithm requires pre-processing prior to execution.

(i) Preprocessing

A genome will be a combination of 5 different characters (G,A,T,C,U). The Index Jump algorithm first creates an array for only the first letter of the target string.

(ii)Comparisons

The algorithm then searches the array for the first letter in the target string. Depending on the comparison, the algorithm decides if it can directly jump or not

to the next possible occurrence of the target by shifting to the next index in the line chosen. This occurs till we complete all indexes of that line.

Adv(?)

The advantage of this algorithm is that we are able to only focus on the possible strings instead of searching serially through the input. This, in turn, decreases the number of comparisons.

(ii) A comparison will be done between the pattern and text. If there is a match, the current indices iterate over the pattern and the text is incremented.

(iii) If there is no match, then the lps[] will be used to find the next new index in the pattern where the checking will continue against the mismatched character in the text.

The Space Complexity and time complexity for the KMP algorithm is O(n).

3. Index Jump

(i)Preprocessing

Time complexity

n=length of input string

p=length of pattern string

Best case (first letter in the pattern string does not match that in the input sequence)→ O(1)

Worst case (pattern contains all of the first letter of input string) →O(n^2)

Average case → O(n)+O(n)

Space Complexity

Preprocessing→ O(n)

The best case is when the first character of the pattern does not exist in the text file, therefore there will only be n comparisons. Since the first character is only compared to each of the characters in the text file only once. The worst case occurs when the pattern string matches exactly with the text file. Therefore, the time complexity of the worst case is O(np). For the average case we will have to take into consideration the number of occurrences of the first character of the pattern in the text file. Since each of the character has a probability of ¼ . The number of possible occurrence is n/4. Therefore, there will be n/4 sets of possible sets to compare with the pattern. From the second character onwards, each character has ¼ probability of matching with the subsequent character in the text file. As a result the time complexity of the average case will be n/12 (1-4^p). If we assume that the pattern will always exist in the text file, we are able to assume that p is a subset of the n, therefore p can expressed as a function of n. Thus the average case will simplified to O(n). And this is shown in the graph below as well, where the time taken changes linearly with the size of the text file.

3. Index Jump

Code Implementation

This algorithm is adapted from 2-Jump DNA Search Multiple Pattern Matching algorithm implemented by Raju Bhukya, DVLN Somayajulu from National Institute of Technology. This algorithm will process the text file and create a list of indexes where the first character in the pattern appears in the text file. If there is a match the algorithm will then compare the next few characters until there is a mismatch. If there is a mismatch, the algorithm will jump to the next occurrence of the first character of the pattern string.

Time Complexity Analysis

For **preprocessing**, time complexity will be (n-p+1) = O(n)

Similarly, the index after n-p+1 does not need any comparisons.

The **best case** is when the whole text does not have any character that matches with the first character. That is, the total time complexity will only consist of the pre-processing part, which is (n-p+1).

The **worst case** is when the whole text, and whole pattern only consist of one character. Eg: Text: “AAAAA….AAAAA”, Pattern: “AA”
The time complexity will be (n-p+1)p, as n-p+1 characters in the text are needed to compare p times.

To find the **average case**, we first assume that each of the characters (A,G,C,T) appears with the same probability, which is ¼.

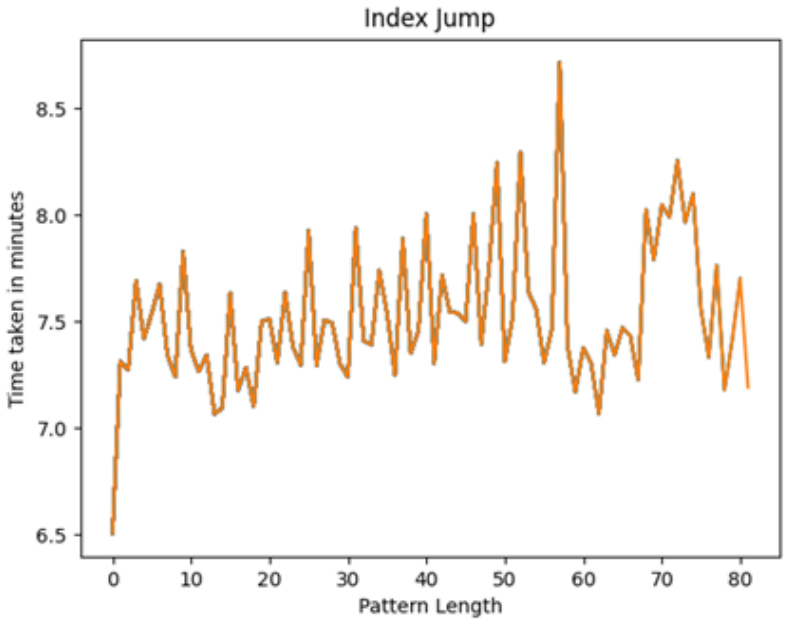
$$P(A) = P(C) = P(G) = P(T) = \frac{1}{4}$$

This also means that when n is large, there are n/4 times appearance for each character. We know that when the comparisons are only required when the previous comparison matches. For previous comparison to match, the probability is also ¼. Therefore, our total number of comparisons will be

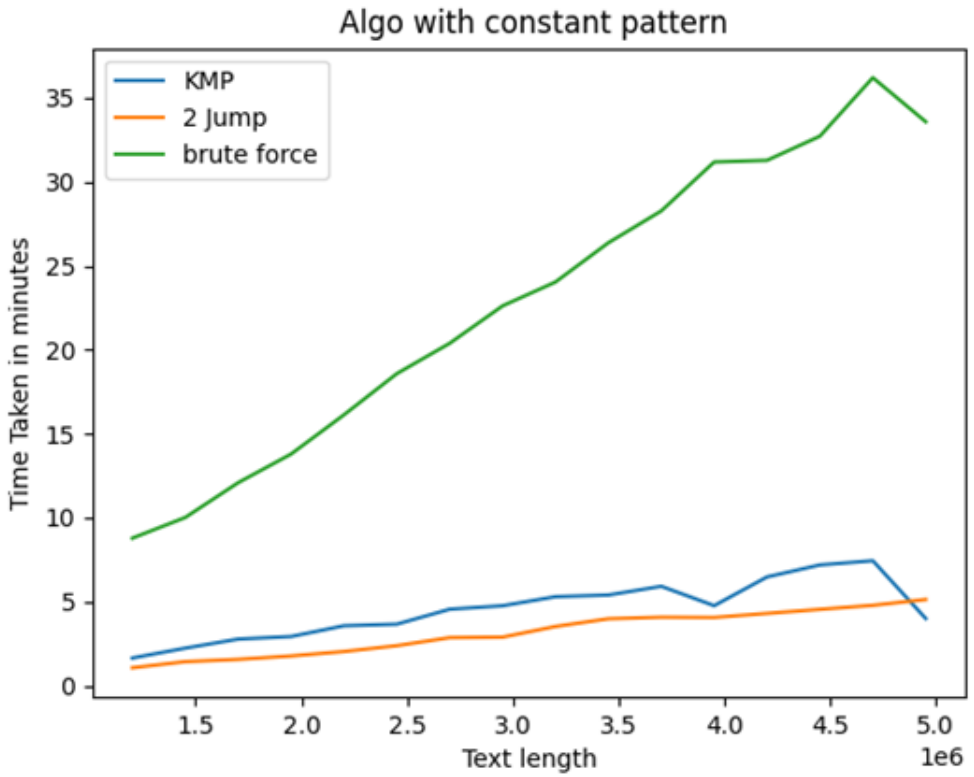
$$Average = \frac{1}{4}(n - p + 1) + \left(\frac{1}{4}\right)^2(n - p + 1) + \ldots + \left(\frac{1}{4}\right)^p(n - p + 1) = \frac{4}{3}(1 - \frac{1}{4^p})(n - p + 1)$$

Therefore, the time complexity of index jump algorithm in all cases is as below:

Best Case: O(n)
Average Case: O(n)+O(n) = O(n)
Worst Case: $O(n^2)$+O(n) = $O(n^2)$



Conclusion



From we can see that the KMP and the 2-Jump algorithm is able to increase linearly with increasing input text string. Another thing to note is that the increase in the time taken for the BruteForce algorithm is almost quadratic . Therefore our average case analysis for all the algorithm is further evident from this graph.We decided to reduce the number of sample size that we tested on because, when we increase the sample size there was a lot of unnecessary noise being reflected on the graph such as the one shown below.

References

KMP stuff(what was previously done)

KMP is a string matching algorithm where a pattern string P will be compared with a text string. The algorithm will then check if P can be found inside T.

KMP makes use of a failure function which searches for the next possible match within the current search. It does this by comparing each character searched with a last position suffix and stores this value to an array.

The LPS of the pattern is first calculated where each character in the pattern is assigned a value to where the search position searched is identical to the starting sequence of the pattern.

An example of this would be for the pattern “AABA” the Lps of the pattern searched would be [0,1,0,0]. When the string is searched for the pattern and fails, it checks the lps array for the value at the point of failure and continues searching at that point.

Considering the string “AAABA”.

i is the current search position in string

j is current search position in the pattern

If search fails resume search based on lps table[1]

The second function takes in 3 inputs, pattern, text and the failure function(

These are the components to KMP:

(i) Preprocessing

The pattern will be preprocessed by creating an integer array lps[] or otherwise known as the failure function.

Comparing the current character with the previous character. If the character in the input string is equivalent to the character in the pattern string, we will look at the next few characters. If there is a mismatch, we compare the current character in the input with the longest known prefix or suffix.

KMP TABLE

[0,1,2,3,4,5] “AAABCA” “A”	Success i=0 j=0	LPS of pattern is = [0,1,0,0]
[0,1,2,3,4,5] “AAABCA”	Success i=1	

[1] Table at the end of report

“AA”	j=1	
[0,1,2,3,4,5] “AAABCA” “AAB”	Fail i=2 j=2	As search fails at j=2 search continues at next highest value of the lps table, j is re assigned to highest value of lps table, j=1
[0,1,2,3,4,5] “AAABCA” “AA”	Success i=2 j=1	
[0,1,2,3,4,5] “AAABCA” “AAB”	Success i=3 j=2	
[0,1,2,3,4,5] “AAABCA” “AABA”	Fail i=4 j=3	In this case