CZ2001 Lab Project 1: Searching Algorithm Report

---

1. Introduction

1.1 Objective

As mentioned in the project briefing, DNA and protein sequence searching is one of the fundamental problems in bioinformatics research. Thus, when dealing with a large amount of data (longer sequences), efficient algorithms are important so as to save time during the searching process of a query sequence.

1.2 Overview

In this report, the design and complexity analysis of the brute force searching algorithm will first be discussed. In addition, 2 more searching algorithms, which were inspired by the Boyer-Moore-Horspool (BMH) algorithm and Knuth–Morris–Pratt (KMP) algorithm respectively, will be proposed, followed by the elaboration on the time and space complexity analysis of each algorithm. An overall comparison of the 3 proposed algorithms will also be discussed in later sections of the report.

---

2. Algorithms and Analyses

2.1 Brute Force Algorithm

2.1.1 Design and Execution

The brute force approach to the sequence searching problem is walking through the source sequence starting from the beginning and checking at each position if the resulting substring equals the query sequence. Thus, a nested for-loop is used to implement this algorithm, the outer loop is to check along all possible substrings, whereas the inner loop is to compare characters between the substring and the query sequence. (refer to Fig 1.)

```
occurrences = []    # create an empty list
# s1: query sequence; s2: source genome sequence
for i in range (len(s2)-len(s1)+1):
    match = True
    for j in range (len(s1)):
        if s2[i+j] != s1[j]: # compare characters
            match = False    # mismatch happened, exit from inner loop
            break
    # match found
    if match:       # append the position of the matched sequence in the list
        occurrences.append(i+1)
```

Fig 1. Brute Force Algorithm (Part)

2.1.2 Complexity Analysis

2.1.2.1 Time Complexity Analysis

For analysis purposes, let n denote the length of the source genome sequence, and m be the length of the query sequence, where (n>>m).

To analyze the time complexity of the brute force algorithm, we will be looking at the number of character comparisons.

- Best case scenario: O(n)

Every first-character comparison results in a mismatch, in other words, the inner loop is only executed once for each iteration. In this case, the total number of comparisons is only determined by the number of outer-loop iterations, which is (n-m+1), where only the first character of the query sequence is compared with every single character in the source genome sequence up until the $(n-m+1)^{th}$ character in the source genome sequence. (refer to Fig 2.)

Text: ACTGGTTCATGACCT

Pattern: BATGTC

Fig 2. In total there are 15-6+1=10 comparisons needed in this example

Since we assume that n>>m, the time complexity of the brute force algorithm in best case scenario can be represented by O(n).

- Worst case scenario: O(nm)

The total number of comparisons is the product of the number of outer-loop comparisons and inner-loop comparisons, which is m(n-m+1). (refer to Fig 3.)

Text: TTTTTTTTTT Text: TTTTTTTTTT

Pattern: TTTT Pattern: TTTA

Fig 3. In total there are 4×(10-4+1)=28 comparisons needed in both examples

In both cases, we assume that n>>m, the time complexity of the brute force algorithm in the worst case scenario can be represented by O(nm).

-Average case scenario: O(nm)

First, we consider the average number of comparisons that the outer loop is executed, since all possible substrings should be checked along the source string, thus the outer loop will always be executed (n-m+1) times.

Next, we consider the average number of comparisons in the inner loop. The number of comparisons ranges from 1 to m before breaking from the inner loop. Assuming each number of comparisons is equally likely to be chosen with probability of $\frac{1}{m}$. The expected number of comparisons is:

$$E(\text{Number of comparisons of inner loop}) = \frac{1}{m}\sum_{i=1}^{m} i = \frac{1}{m}\cdot\frac{(1+m)(m)}{2} = \frac{(1+m)}{2}$$

Therefore, the average time complexity of brute force algorithm is:

$$\text{Average number of comparisons} = \left(\frac{(1+m)}{2}\right)(n-m+1) = O(nm)$$

2.1.2.2 Space Complexity Analysis

In this algorithm, no pre-processing of strings (such as constructing additional arrays/tables to store information) is needed, thus only constant extra space is required to store the input genome sequence and query sequence.

Thus, the space complexity is constant, which can be represented by O(1).

2.2 BMH Algorithm

2.2.1 Design and Execution

Boyer–Moore–Horspool algorithm is a string searching algorithm which reduces searching time by comparing the pattern sequence with the text sequence from the end of the pattern sequence to the start. This allows the algorithm to save time as it can skip comparisons between the first few characters when the last few do not match.

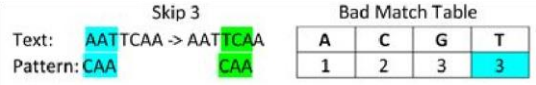


Fig 4. Single comparison between T and A (Skipped 2 comparisons at index 0 and 1)

The number of positions to skip with respect to the rightmost character compared in the text is stored in the Bad Match Table. (Fig 4.) To construct the Bad Match Table in Python, we iterate through the 4 possible alphabets to first construct an alphabet map which maps a character to its index in the bad match table.

```
self.alphabet_map = {} #{"A":0, "C":1, "G":2, "T":3}
for i in range(len(alphabet)):
  self.alphabet_map[self.alphabet[i]] = i
```

Next, we iterate through the pattern and for every character encountered, set the value in the bad match table to (pattern_length-pattern_index-1). All other characters not encountered before the last character in the pattern will be set to pattern length. The creation of the bad match table is completed after finishing this step.

```
self.bad_match_table = [self.pattern_len] * len(alphabet)
for i in range(self.pattern_len-1):
  char_to_index = self.alphabet_map[self.pattern[i]]
  self.bad_match_table[char_to_index] = self.pattern_len - i - 1
```

During searching, the algorithm starts comparing from the last character of the pattern and when a mismatch is found or when all characters have been compared, the rightmost character in the text is used to find the number of skips from the Bad Match Table. (Fig 5.)

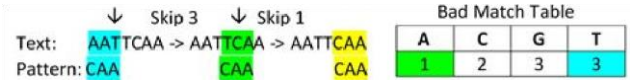


Fig 5. How Bad Match Table is used in searching (Arrows point to mismatched character)

2.2.2 Complexity Analysis

2.2.2.1 Time Complexity Analysis

Preprocessing Complexity O(m+ σ)

To make the bad match table, we first have to iterate through all possible alphabets (σ= 4). Next, we have to iterate through all but the last character in the pattern (m-1). Hence, preprocessing has a time complexity of O(m-1+σ)= O(m+ σ)

Searching Complexity

- Best case scenario: O($\frac{n}{m}$) (n: Text sequence, and m: Pattern sequence)

When the first compared character (rightmost) is a character not found in the pattern and pattern skips by m each time.

Fig 6. n: 9, m= 3: Only (n/m)=3 character comparisons = O($\frac{n}{m}$) -

Worse case scenario: O(nm)

When all characters in the pattern match or all but the last match and we skip by 1 each time.

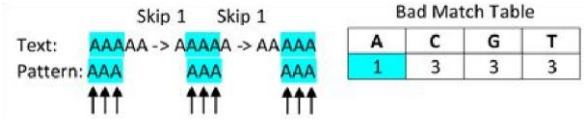


Fig 7. n: 5, m= 3: (n-m+1)(m)=9 character comparisons = O(nm) (Each arrow is a comparison)

- Average case scenario: O(n)

First, we calculate the average number of times the outer loop is executed. To find this, we have to calculate the average number of skips after executing each inner loop. The number of skips ranges from 1 to m. Assuming each skip is equally likely to be chosen, with probability of $\frac{1}{m}$. The expected of number of skips is:

$$E(\text{Number of skips}) = \frac{1}{m}\sum_{i=1}^{m} i = \frac{1}{m}\cdot\frac{(1+m)(m)}{2} = \frac{(1+m)}{2}$$

Hence, the average number of times the outer loop is executed can be estimate to be:

$$E(\text{Number of times outer loop is executed}) = \frac{n}{(1+m)/2} = 1\frac{2}{+}\frac{n}{m}$$

Next, we calculate the average number of comparisons for each outer loop. The number of comparisons ranges from 1 to m. Assuming each number of comparisons is equally likely to be chosen with probability of $\frac{1}{m}$. The expected number of comparisons is:

$$E(\text{Number of comparisons}) = \frac{1}{m}\sum_{i=1}^{m} i = \frac{1}{m}\cdot\frac{(1+m)(m)}{2} = \frac{(1+m)}{2}$$

Therefore, the average time complexity of Boyer-Moore-Horspool Algorithm is:

$$\text{Average number of comparisons} = (\_\_\_\_^{(1+}{}_2^{m)})(1\frac{2}{+}\frac{n}{m}) = n = O(n)$$

2.2.2.2 Space Complexity Analysis

In this algorithm, a bad matching table of size 4 (4 characters “ACGT” or “ACGU”) is needed. Since table size remains the same regardless of text or pattern length. The space complexity is constant, which can be represented by O(1).

2.3 KMP-Based Hybrid Algorithm

2.3.1 Design and Execution

The KMP algorithm has been widely used as a means of string searching and it eliminates the backtracking problem found in Boyer Moore. Using the idea behind KMP, we have created an original code which implements the bad match table mentioned in 2.2.1 to trigger lesser comparisons.

The basis of KMP is to prevent the re-examination of previously matched characters when a mismatch is detected after some matches. To achieve this, pre-processing of query patterns is necessary. An integer array lps[] which contains the count of characters to be skipped is constructed. This will be the first preprocessing step our code uses.
To implement the bad match table, our code will require a second preprocessing to create the bad match table(this process is explained in 2.2.1).

Preprocessing to form lps[]:

Construct an integer array lps[], where lps stands for "longest proper prefix that is also a suffix", (A proper prefix of a string is a prefix that is not equal to the string itself.)
E.g. Given a pattern ACTACG, we would like to construct its corresponding lps[]. (see Fig 8. & 9.)

Pattern: ACTACG
Take 'C' at index 4 as an example, we consider the proper prefixes and suffixes of ACTAC
"AC" is the lps at index 4,
thus lps[4]=2 (the length of lps)

Proper prefix	Proper suffix
A	C
AC	AC
ACT	TAC
ACTA	CTAC

Fig 8. Example of how to construct lps[]

Pattern: ACTACG lps arrary

A	C	T	A	C	G
0	0	0	1	2	0

Fig 9. Example of the lps array of a given pattern

The lps array is used to decrease the number of comparisons. If at index 5 there is a mismatch(Fig 10.) with the source code, we look at the previous index (index 4) and obtain the lps value of index 4⟶ lps[4]=2. This allows us to subsequently skip 2 comparisons at the same location of the source code as we compare current source index with pattern index 2.

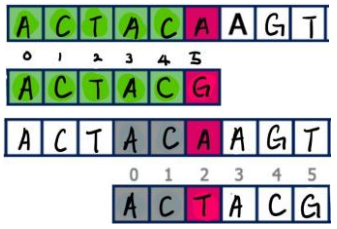
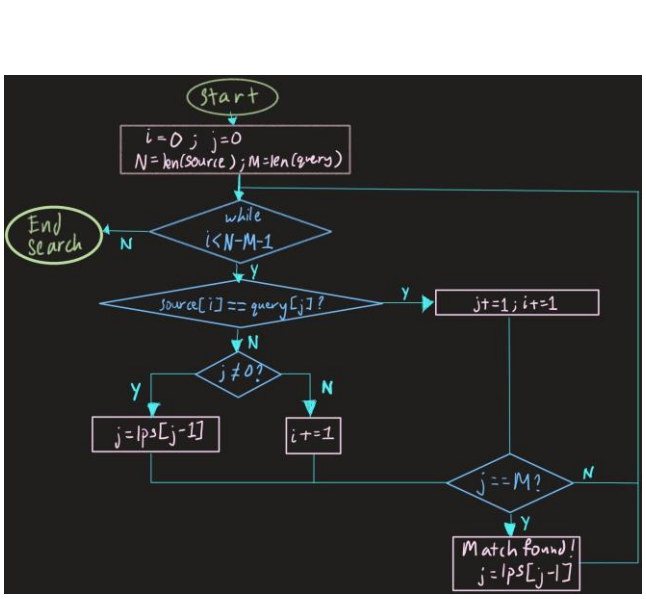


Fig 10. Example of the using lps to skip comparisons

Original KMP logic vs modified KMP logic:

The difference between the original and the modified lies in using bad character heuristic to determine if we should decrease comparison by changing i (which causes the program to iterate its next while loop) when the source character does not match query character; this code seeks to obtain maximum i increase per iteration.
Initially, in normal KMP, if a character in query pattern (excluding the first character) does not match with the respective character in source code, only the j value changes using the lps[]; this does not allow the loop to move onto the next iteration which is dependent on i and increases comparisons. (Fig 11.1)
However, in the newKMP, it finds the matching last character in the source code after lps[] jump and compares it with the last character of pattern using bad character heuristic. If it does not match, we change the i value using Bad-Match Table(BMT) and forgo changing j value with lps[] since we already know the last character does not match after lps[] jump.(Fig 11.2)



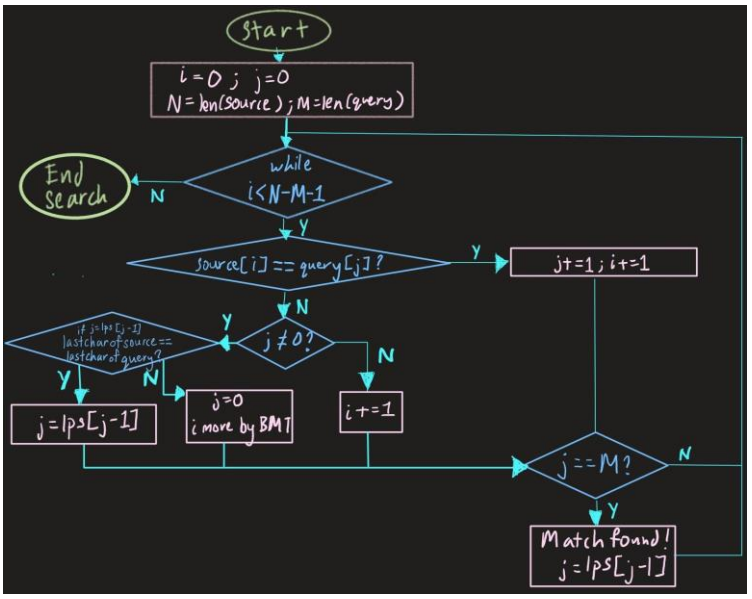


Fig 11.1 & 11.2 KMP logic flow vs modified KMP logic flow

2.3.2 Complexity Analysis

2.3.2.1 Time Complexity Analysis

Original KMP: Compute lps array: O(m); KMP search: O(n); Total: O(m+n)

Modified KMP:

Preprocessing Complexity

O(m+σ) for bad character table(refer to 2.2.2.1), O(m) for lps array(program runs through pattern with length of m)

Searching Complexity

-Best case scenario: O($_m{}^n$) (n: Text sequence, and m: Pattern sequence)

The modified algorithm tries to maximize the value of i shifts which symbolizes the movement along the source text. The maximum amount of movement of the bad character heuristic is m, so when the character does not match the last character of the pattern string and uses bad heuristic, the pattern shifts by m.

- Worse case scenario: O(n)

Source and search query look almost the same: Text- AAAAATAAAAATAAA... and Pattern-AAAAA

A. This is the same as the normal KMP's worst case scenario. LPS table of pattern would be [0,1,2,3,4,5] where at point of mismatch(highlighted red), code will run (m-1) more times. So each green sequence compare (m-1) times where all are matches(Fig 12.1), whereas red will cause comparisons from pattern index[5]to[4]to[3]to[2]to[1]to[0],all which are all mismatches, which is m mismatches(Fig 12.2). So total comparison of all green areas:(green strings in code)*(m-1 comparisons per green string)=(n/m)*(m-1)≈n, and all comparisons at red characters:(red strings in code)*(m comparisons per red string)=(n/m)*(m)=n. Thus a total of 2n comparisons were made and 2n = $O(n)$.

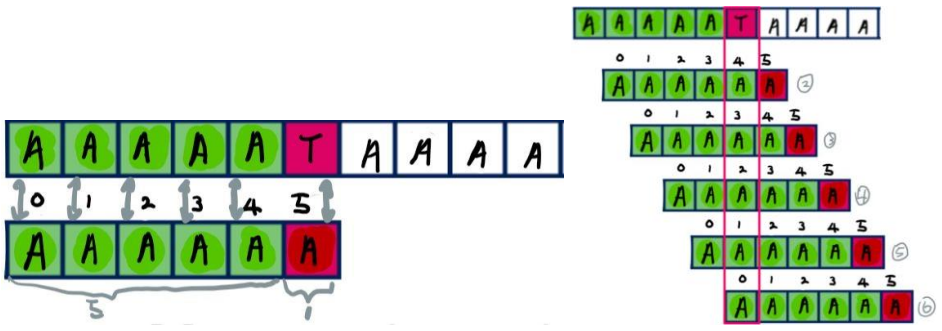


Fig 12.1 & 12.2 Green comparisons vs red comparisons

- Average case scenario O(n)

The first 3 if-else loops are of importance. Assuming that matching character from R/DNA has only ¼ chance of success and ¾ chance of failing(A==A vs A vs.G, A vs.T, A vs.C) and the probability for j==0 is 1/m (since j can take values 0 to m-1), referring to Fig.12.2, expectation of i jump includes when [txt==pattern],[j==0],[j!=0 and last char after jump match]. From 2.2.2.1, average skip from bad heuristic is:

$$E(Number\ of\ skips) = \frac{1}{m}\sum_{i=1}^{m} i = \frac{1}{m}\cdot\frac{(1+m)(m)}{2} = \frac{(1+m)}{2}$$

Expected i shifts per iteration

$$= \frac{1}{4} + (\frac{1}{m})(\frac{3}{4}) + (\frac{(1+m)}{2})(\frac{3}{4})(1-\frac{1}{m})(\frac{3}{4}) = \frac{1}{4} + \frac{15+9m^2}{32m}$$

Expected number of comparisons

$$= n/(\frac{15+8m+9m^2}{32m}) = O(n)$$

Although the average time complexity seems to be equal to that of Boyer Moore's, the expected comparison is slightly lesser (Refer to 2.2.2.1).

2.3.2.2 Space Complexity Analysis

Original KMP: Build Table: O(m); Matching Traversal: O(1); Overall: O(m)

Modified: Build lps Table: O(m); Build Bad character table(refer 2.2.2.2): O(1); Matching Traversal: O(1); Overall: O(m)

---

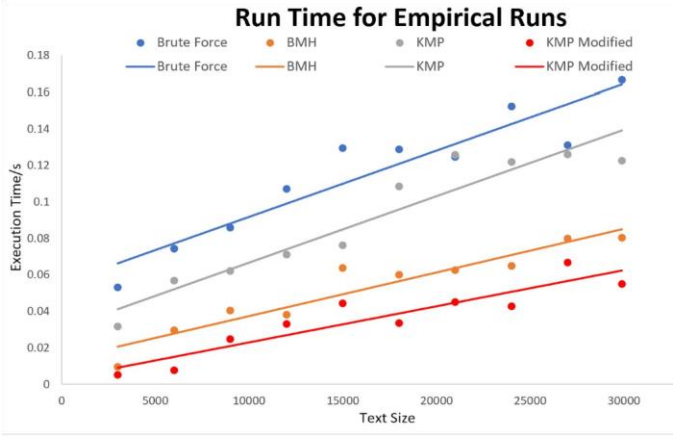
3. Comparison and Conclusion

3.1 Theoretical complexities

* n: Length of text, m: Length of pattern, σ: Number of alphabets

Algorithms	Preprocessing Time complexity		Best Case Time Complexity	Worst Case Time Complexity	Average Case Time Complexity	Preprocessing Space Complexity	Overall Space Complexity
Brute Force			O(n)	O(mn)	O(mn)	-	O(1)
BMH	O(m+σ) for bad character table		O($\frac{n}{m}$)	O(mn)	O(n)	O(1) for building bad character table	O(1)
KMP	O(m) for lps array		O(n)	O(n)	O(n)	O(m) for lps Table	O(m)
Modified KMP	O(m+σ) for bad character table	O(m) for lps array	O($\frac{n}{m}$)	O(n)	O(n)	O(m) for lps Table & O(1) for building bad character table	O(m)

Although the average time complexity for modified KMP seems to be the same as Boyer Moore Horspool's, the expected comparison is slightly lesser, as we can see in 3.2.



3.2 Run time for Empirical runs

In this section, we did 3 trial runs for each text size (3000, 6000, 9000 … 30000) with our 4 algorithms. Pattern is kept constant as "AGGTC", and we used the source DNA from test.fna. The average run time from each 3 trial runs is calculated to minimize the fluctuation in run time.

The graph clearly shows that with an increase in text size, execution time also increased linearly. While the brute force algorithm took the longest time to perform searching, modified KMP performs the best, followed by the BMH algorithm, and lastly original KMP algorithms.

*Preprocessing time is added with searching time for BMH

*** It is recommended to keep the input .fna file size to be smaller than a few MB for a smooth execution.**

Contributions

Shuwen: Time complexity analysis and space complexity analysis tables. Empirical analysis and graph for empirical analysis.

Chen: Implementation of file reading functions, Implementation of Brute Force Algorithm, Explanation of Brute Force algorithm. Time complexity analysis and space complexity analysis for brute force algorithm in report and slides, explanation and implementation of how LPS table was created.

Tiviatis: Explanation on how LPS table is used for pattern searching, Creation of Modified Knuth–Morris–Pratt (KMP) algorithm code, Logic flow and diagram of KMP versus Modified KMP, explanation of Modified KMP algorithm in report and slides, time and space complexity of Modified KMP in report and slides.

Lirong: Implementation of Boyer-Moore-Horspool (BMH) algorithm, Explanation of BMH in report and slides, Time and Space complexity analysis of BMH in report and slides.
