Introduction to Bioinformatics
Analysis of DNA String Searching Algorithms

Submitted on October 28, 2014

Dr. Rajamanickam Murugan

Adarsh A Tadimari EE12B003

Introduction

The following document analyzes three different algorithms for substring search in a large string which is a DNA sequence in this case.

1. Naive Search
2. Knuth-Morris-Pratt Algorithm
3. Boyer-Moore Algorithm

The algorithms have been explained and their performance on DNA data sets of different lengths have been reported. The source code for all the algorithms have been implemented in C++ by the author and are made available for evaluation at `https://github.com/tadarsh/Acad/tree/master/BT6090/DNA`
A DNA string search library was created, which has functions to generate random DNA sequence, Naive Search, Boyer-Moore Search and Knuth-Morris-Pratt Search algorithms for use.

Performance Measurement

Two approaches have been taken to measure the performance of each algorithm.

1. Number of Iterations
2. No of clock cycles (or) the time taken

The above approaches do not include the operations/time taken to load the data set.

Naïve Search

Naïve Search is a brute force method of searching for a substring in a large string.

Algorithm

Consider the sequence S[1...n] and the pattern P[1....m]

Algorithm 1 Naïve Search

```
1: procedure NAÏVE
2:     i ← 0
3: loop:
4:     if S[i....i + m − 1] = P[1...m] then
5:         match
6:     end if
7:     i ← i + 1
8:     if i = n − m + 2 then
9:         close
10:    end if
11:    goto loop.
12: end procedure
```

Performance

Time

The average time was calculated for DNA sequence of lengths ranging from 1 million to 1 billion and averaged over pattern length varying from 10 to 100.

DNA Sequence Length	Time taken(ms)
1 million	34.68
10 million	285.71
100 million	2962.60
1000 million	40222.20

Iterations

The number of iterations were calculated for DNA sequence of lengths ranging from 1 million to 1 billion and averaged over pattern length varying from 10 to 100.

DNA Sequence Length	Iterations
1 million	1332809
10 million	13332425
100 million	133331452
1000 million	1333378007

Complexity

Naïve search has a worst case complexity of **O((n - m)m)**, where n is the length of the sequence and m is the length of the pattern.

Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt (KMP) algorithm takes into account the degenerating property of the pattern, i.e there might be sub-patterns appearing more than once in the pattern. The KMP algorithm is a linear time algorithm but includes a preprocessing stage which is also linear with the size of the pattern.

Algorithm

Preprocessing Stage

Using the pattern we build an auxiliary array which contains information about the longest proper prefix (LPP) of the substring which is also a suffix.
Consider the following examples,

1. Pattern "ATGAGATG" , LPP = [0, 0, 0, 1, 0, 1, 2, 3]
2. Pattern "AAAGAAAG" , LPP = [0, 1, 2, 0, 1, 2, 3, 4]

The algorithm has been presented in the following page.

Algorithm 2 KMP Algorithm

```
1: procedure KMP Preprocessing
2:     i ← 1
3:     lenp ← 0
4:     LPP[0] ← 0
5:     M ← lengthofthepattern
6:     while i < M do
7:         if Pattern[i] = Pattern[lenp] then
8:             lenp ← lenp + 1
9:             LPP[i] ← lenp
10:            i ← i + 1
11:        else
12:            if lenp ≠ 0 then
13:                lenp = LPP[lenp − 1]
14:            else
15:                LPS[i] ← 0
16:                i ← i + 1
17:            end if
18:        end if
19:    end while
20: end procedure
21: procedure Find Matches
22:    M ← lengthofpattern
23:    N ← lengthofsubsequence
24:    j ← 0
25:    i ← 0
26:    while i < N do
27:        if Pattern[j] = Sequence[lenp] then
28:            j ← j + 1
29:            i ← i + 1
30:        end if
31:        if j = M then
32:            Match found
33:            j ← LPP[j − 1]
34:        else
35:            if Pattern[j] ≠ Sequence[i] then
36:                j ← LPS[j − 1]
37:                i ← i + 1
38:            end if
39:        end if
40:    end while
41: end procedure
```

Performance

Time

The average time was calculated for DNA sequence of lengths ranging from 1 million to 1 billion and averaged over pattern length varying from 10 to 100.

DNA Sequence Length	Time taken (ms)
1 million	33.3150
10 million	265.1308
100 million	2523.4023
1000 million	32104.2034

Iterations

The number of iterations were calculated for DNA sequence of lengths ranging from 1 million to 1 billion and averaged over pattern length varying from 10 to 100.

DNA Sequence Length	Iterations
1 million	1031907
10 million	10317640
100 million	103171564
1000 million	1031738910

Complexity

The algorithm has best case and worst case complexity of O(n)

Conclusions

Knuth-Morris-Pratt algorithm performs slightly better than Naïve search but the best case performance is not very good.

Boyer-Moore Algorithm

Boyer-Moore algorithm is an efficient string searching algorithm. Similar to the Knuth-Morris-Pratt algorithm, it preprocesses the the pattern and not the sequence. Unlike the above two algorithms, Boyer-Moore algorithm searches for mismatch from the end of the string.

Algorithm

Preprocessing Stage

Using the pattern, we build a table which will help us find number of steps to be skipped efficiently.
For example, consider the pattern "ATCGAT". The table we build will have four rows corresponding to each of the bases A, T, C, G. The number of columns will be the number of elements in the pattern. Entry in ith row and j column will represent the shift in the pattern needed to reach base pair i next.
n corresponds to the case where there the particular base pair is missing to the left of the given index from the end.
In the given example, the table will be the following.

A	1	0	3	2	1	0
T	0	3	2	1	0	n
C	3	2	1	0	n	n
G	2	1	0	n	n	n

Explanation for the first row: The first entry 1 represents the position of A with respect to the last element, the second entry corresponds to the position of A with respect to the second last element and so on.

Algorithm 3 Boyer-Moore Algorithm

```
1: procedure BOYER-MOORE MATCH SEARCH
2:     i ← 0
3:     N ← lengthofsequence
4:     M ← lengthofpattern
5:     while i ≤ N − M do
6:         j ← M − 1
7:         while j ≥ 0 do
8:             if Pattern[j] = Sequence[i + j] then
9:                 if j = 0 then
10:                    Match found
11:                    i ← i + 1
12:                end if
13:            else
14:                if Table[sequence[i + j]][M − j − 1] ≠ n then
15:                    i ← i + Table[sequence[i + j]][M − j − 1]
16:                else
17:                    igetsi + j + 1
18:                end if
19:            end if
20:            j ← j − 1
21:        end while
22:    end while
23: end procedure
```

Performance

Time

The average time was calculated for DNA sequence of lengths ranging from 1 million to 1 billion and averaged over pattern length varying from 10 to 100.

DNA Sequence Length	Time taken (ms)
1 million	19.4802
10 million	185.9633
100 million	1868.2345
1000 million	18528.2145

Iterations

The number of iterations were calculated for DNA sequence of lengths ranging from 1 million to 1 billion and averaged over pattern length varying from 10 to 100.

DNA Sequence Length	Iterations
1 million	376641
10 million	3767363
100 million	37645963
1000 million	376414095

Complexity

Boyer-Moore algorithm has a complexity of O(n). In the best case, only one in m (size of pattern) needs to be checked which reduces the complexity to O(n/m). It works better with larger m.

Conclusion

Through implementation and testing of the above algorithms, it was found that in most of the cases Boyer-Moore performs almost thrice as fast as the Naïve implementation of string search and twice as fast Knuth-Morris-Pratt Algorithm.

References

[1] Wikipedia Article on Boyer-Moore Algorithm,
`http://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm`

[2] Wikipedia Article on Knuth-Morris-Pratt Algorithm,
`http://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm`

[3] BT6090 Class Notes on String Search