ANALYSIS OF ALGORITHMS

Motif Finding Using KMP Algorithm

Abstract

Pattern searching holds much importance for biologists, for the understanding of DNA (and its functionality) can be more than a matter of satisfying curiosity, but also give answers to many issues such as medical conditions. However, there are number of ways of searching within a single chromosome, and the chosen method could depend on the specific task at hand, as well as the efficiency of the program (how fast it is, how much memory it requires) in respect to the computer system being used.

HIMAJA KOTTURU
50490169
himaja.kotturu@smail.astate.edu
AMAN HASAN SHAIK
50489517
amanhasa.shaik@smail.astate.edu

ANALYSIS OF ALGORITHMS PROJECT

HIMAJA KOTTURU
50490169
himaja.kotturu@smail.astate.edu

AMAN HASAN SHAIK
50489517
amanhasa.shaik@smail.astate.edu

Knuth-Morris-Pratt (KMP) Algorithm:

The object of string searching is to find the location of a specific text pattern within a larger body of text. The main considerations for string searching are speed and efficiency. One of the algorithm is Knuth-Morris-Pratt (KMP) Algorithm. It is the linear time string-matching algorithm that solves the string matching problem by preprocessing P in Θ(m) time. Its main idea is to skip some comparisons by using the previous comparison result.

But the drawback of this approach is if 'm' is the length of pattern 'p' and 'n' the length of string 'S', the matching time is of the order O(mn). This is a certainly a very slow running algorithm.

A matching time of O(n) is achieved by avoiding comparisons with elements of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs.

Components of KMP algorithm:

- **The prefix function, Π**

The prefix function, Π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern 'p'. In other words, this enables avoiding backtracking on the string 'S'.

- **The KMP Matcher**

With string 'S', pattern 'p' and prefix function 'Π' as inputs, finds the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrence is found.

Running - time analysis

- **Compute-Prefix-Function (Π)**

1. m ← length[p] //'p' pattern to be matched
2. Π[1] ← 0
3. k ← 0
4. **for** q ← 2 to m
5. **do while** k > 0 and p[k+1] != p[q]
6. **do** k ← Π[k]
7. **if** p[k+1] = p[q]
8. **then** k ← k +1
9. Π [q] ← k
10. **return** Π

In the above pseudocode for computing the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step 1 to step 3 take constant time. Hence the running time of compute prefix function is Θ(m).

ANALYSIS OF ALGORITHMS PROJECT

HIMAJA KOTTURU
50490169
himaja.kotturu@smail.astate.edu

AMAN HASAN SHAIK
50489517
amanhasa.shaik@smail.astate.edu

- **KMP Matcher**

1. n ← length[S]
2. m ← length[p]
3. Π ← Compute-Prefix-Function(p)
4. q ← 0
5. **for** i ← 1 to n
6. **do while** q > 0 and p[q+1] != S[i]
7. **do** q ← Π[q] **if** p[q+1] = S[i]
8. **then** q ← q + 1**if** q = m
9. **then** print “Pattern occurs with shift” i – m
10. q ← Π[q]

for loop beginning in step 5 runs ‘n’ times, i.e., as long as the length of the string ‘S’. Since step 1 to step 4 take constant time, the running time is dominated by this for loop. Thus running time of matching function is Θ(n).

ANALYSIS OF ALGORITHMS PROJECT

HIMAJA KOTTURU
50490169
himaja.kotturu@smail.astate.edu

AMAN HASAN SHAIK
50489517
amanhasa.shaik@smail.astate.edu

Program:

```
package kmp;

import java.io.FileNotFoundException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class KMPAlgorithm {
public String[] OpenFile() throws IOException {

  FileReader reader = new FileReader("bin\\input.fasta");
  BufferedReader textReader = new BufferedReader(reader);

  int numberOfLines = readLines();
  String[] textData = new String[numberOfLines];
  int BufferIndex = 0;
  String line;

  while ((line = textReader.readLine()) != null) {
    if (line.startsWith(">")) {

    }else{
       textData[BufferIndex] = line;
       BufferIndex = BufferIndex + 1;
    }
  }

  // close the line-by-line reader and return the data
  textReader.close();
  return textData;
}
int readLines() throws IOException {
  FileReader reader = new FileReader("bin\\input.fasta");
  BufferedReader textReader = new BufferedReader(reader);
  String line;
  int numberOfLines = 0;
// String[] textData;
  while ((line = textReader.readLine()) != null) {
    // I tried this:
  if (line.contains(">")) {
      // do nothing
    }else{
      numberOfLines++;
    }
  }
  reader.close();
  return numberOfLines;
}

private int[] computeTemporaryArray(char pattern[]){
```

ANALYSIS OF ALGORITHMS PROJECT

HIMAJA KOTTURU
50490169
himaja.kotturu@smail.astate.edu

AMAN HASAN SHAIK
50489517
amanhasa.shaik@smail.astate.edu

```
        int [] lps = new int[pattern.length];
        int index =0;
        for(int i=1; i < pattern.length;){
            if(pattern[i] == pattern[index]){
                lps[i] = index + 1;
                index++;
                i++;
            }else{
                if(index != 0){
                    index = lps[index-1];
                }else{
                    lps[i] =0;
                    i++;
                }
            }
        }
        return lps;
    }

    /**
     * KMP algorithm of pattern matching.
     */
    public boolean KMP(char []text, char[] pattern){

        int lps[] = computeTemporaryArray(pattern);
        int i=0;
        int j=0;
        while(i < text.length && j < pattern.length){
            if(text[i] == pattern[j]){
                i++;
                j++;
            }else{
                if(j!=0){
                    j = lps[j-1];
                }else{
                    i++;
                }
            }
        }
        if(j == pattern.length){
         for(int a=0;a<15;a++)
         {
         System.out.print(pattern[a]);
         }
            return true;
        }
        return false;
    }
public static void main(String[] args) throws IOException {
KMPAlgorithm obj = new KMPAlgorithm();
String [] mainfile= obj.OpenFile();
String pattern ;
char [] kmpmain = new char[738];
char [] patternarr = new char[82];
char [] smallpattern = new char[1];
```

ANALYSIS OF ALGORITHMS PROJECT

HIMAJA KOTTURU
50490169
himaja.kotturu@smail.astate.edu

AMAN HASAN SHAIK
50489517
amanhasa.shaik@smail.astate.edu

```
int a=0,b,p=0,z;
System.out.println("The input file is: ");
for(int i=1;i<10;i++)
{
kmpmain=mainfile[i].toCharArray();
System.out.println(kmpmain);
}
pattern = mainfile[0];
//mainfile = obj.OpenFile();
System.out.println("the pattern is :");
patternarr=pattern.toCharArray();
for(b=15;b<82;b++)
{
for(a=p,z=0;a<b&&z<15;a++,z++)
{
smallpattern[z]=patternarr[a];
System.out.print("Searching string :");
System.out.print(smallpattern);
}
}
boolean kmp = obj.KMP(kmpmain, smallpattern);
p=p+1;
  public static void KMP(String pat, String txt)
    {
        int M = pat.length();
        int N = txt.length();

        // create lps[] that will hold the longest
        // prefix suffix values for pattern
        int lps[] = new int[M];
        int j = 0;  // index for pat[]

        // Preprocess the pattern (calculate lps[]
        // array)
        computeLPSArray(pat,M,lps);

        int i = 0;  // index for txt[]
        while (i < N)
        {
            if (pat.charAt(j) == txt.charAt(i))
            {
                j++;
                i++;
            }
            if (j == M)
            {
                System.out.println("Found pattern "+
                            "at index " + (i-j));
                j = lps[j-1];
            }

            // mismatch after j matches
            else if (i < N && pat.charAt(j) != txt.charAt(i))
            {
                // Do not match lps[0..lps[j-1]] characters,
```

ANALYSIS OF ALGORITHMS PROJECT

HIMAJA KOTTURU
50490169
himaja.kotturu@smail.astate.edu

AMAN HASAN SHAIK
50489517
amanhasa.shaik@smail.astate.edu

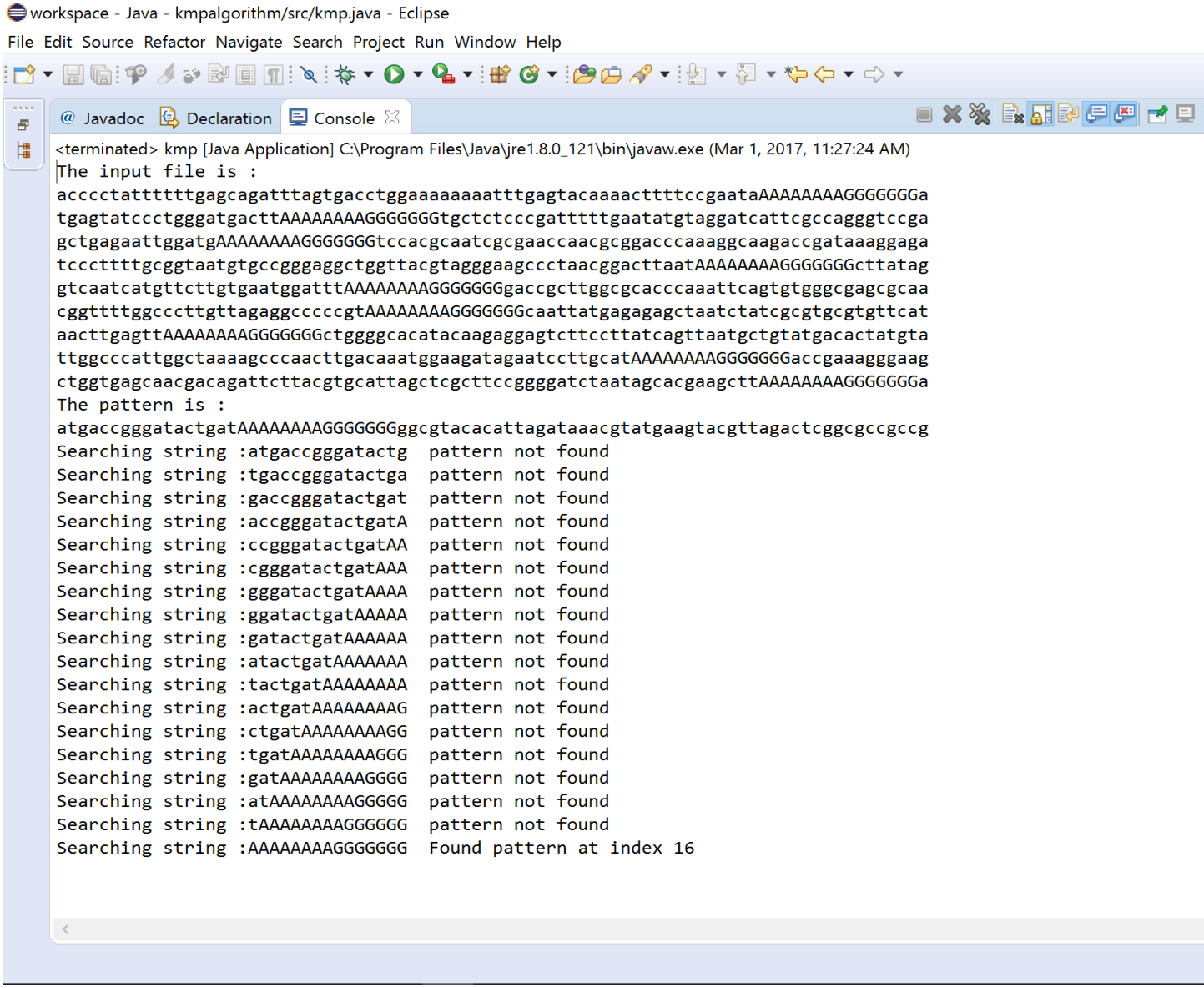
```
            // they will match anyway
          System.out.println("  pattern not found");
            if (j != 0)
                j = lps[j-1];
            else
                i = i+1;
        }
    }
  }
 void computeLPSArray(String pat, int M, int lps[])
  {
    // length of the previous longest prefix suffix
    int len = 0;
    int i = 1;
    lps[0] = 0;  // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M)
    {
        if (pat.charAt(i) == pat.charAt(len))
        {
            len++;
            lps[i] = len;
            i++;
        }
        else  // (pat[i] != pat[len])
        {
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
            // to search step.
            if (len != 0)
            {
                len = lps[len-1];

                // Also, note that we do not increment
                // i here
            }
            else  // if (len == 0)
            {
                lps[i] = len;
                i++;
            }
        }
    }
  }
}
}
}
```

ANALYSIS OF ALGORITHMS PROJECT

HIMAJA KOTTURU
50490169
himaja.kotturu@smail.astate.edu

AMAN HASAN SHAIK
50489517
amanhasa.shaik@smail.astate.edu

Output:



ANALYSIS OF ALGORITHMS PROJECT

HIMAJA KOTTURU
50490169
himaja.kotturu@smail.astate.edu

AMAN HASAN SHAIK
50489517
amanhasa.shaik@smail.astate.edu

Conclusion:

KMP is fast exactly when the pattern and text contain repeated patterns of characters. The running time of compute prefix function is Θ(m)and the matching function is Θ(n).The total running time of KMP algorithm is O(m+n). A real-time version of KMP can be implemented using a separate failure function table for each character in the alphabet. If a mismatch occurs on character in the text, the failure function table for character is consulted for the index in the pattern at which the mismatch took place. This will return the length of the longest substring ending at matching a prefix of the pattern, with the added condition that the character after the prefix is . With this restriction, character in the text need not be checked again in the next phase, and so only a constant number of operations are executed between the processing of each index of the text. This satisfies the real-time computing restriction.