Abstract

Recently, pattern matching on similar texts has drawn wide academic attention because of its important role in cataloging human genetic variation. Many different off-line version algorithms are proposed. However, the on-line version is rare in nowadays. But comparing with the off-line algorithm, on-line version has more benefits in practical application depending on its partial input. Recently, two on-line version algorithms are proposed --- EDSM algorithm and EDSM-BV algorithm[4]. They are not only the on-line version algorithm, but both of them have the better efficiency comparing with the previous on-line algorithm.

In my project, I am focusing on implementation of these two algorithms. Firstly, I implement these two algorithms in C++ depending on the pseudo code presented in [4]. Then I test them using the example data to ensure the correctness of my program. To show the efficiency of them, I design a naïve algorithm of pattern matching on similar texts. After that, I run these three algorithms using synthetic data and show their runtime of different pattern and text on graphs. At last, I input the real data into the EDSM and EDSM-BV algorithm and record their results to test the performance of them on real data.

Acknowledgements

I would like to extend my heartfelt gratitude to my supervisor, Solon P. Pissis. He always gives me valuable advice and helps during the project.

I would also like to thank Ahmad Retha for his help on data part of my project.

Table of Contents

1. Introduction

1.1. Overview

Multiple sequence alignment (MSA) algorithm can be used to align the closely-related sequences into one compact form, that can represent both non-polymorphic and polymorphic sites of MSA[1]. It can be represented by one string $\tilde{T}$ which consists of deterministic and non-deterministic segments. [2] defines this representation as an elastic-degenerate text. Finding all matching position of a deterministic pattern P in $\tilde{T}$ becomes a problem that is called Elastic-Degenerate String Matching (EDSM) problem. The simplest version of EDSM problem is that a degenerate segment can contain only one letter[3].

An elastic-degenerate text can be used in various fields. For example, a set of closely-related DNA sequences can be represented by it. There exist many different algorithms of off-line version for the pattern matching problem. Although there is not so much on-line algorithm exist, it also has itself benefits [4].

In [4], it presents two new algorithms to solve EDSM problems in an on-line manner. The time complexity of the first algorithm is O(nm^2+N), and it needs a preprocessing stage with time and space both O(m) where m is the length of the pattern, n is the length of $\tilde{T}$ and N is the size of $\tilde{T}$(N>m). The time complexity of the second algorithm is O(N· $\lceil m/\omega \rceil$)(ω is the size of the computer word in the RAM model), and it needs a preprocessing stage with time and space both O(m· $\lceil m/\omega \rceil$).

1.2. Aim and Objective

The main aims and objectives of this project are following:

- Understand what is EDSM problem and know relative knowledge
- Understand two on-line version algorithms in detail
- Implement two on-line version algorithms in C++
- Implement a naïve algorithm in C++
- Test three algorithms using synthetic data and compare their efficiency
- Using the real data to test and compare EDSM and EDSM-BV algorithm

The final achievement: three implementations of the algorithm in C++, a graph of the comparison of these three algorithms with synthetic data and a graph of the comparison of EDSM and EDSM-BV in real data.

1.3. Background and Literature Survey

1.3.1. Previous Study and Relative Knowledge

Assume there is a stable pattern P with length m and an *elastic-degenerate* text $\tilde{T}$ with length n and size N(N>m). In [2], it presents an algorithm for solving the EDSM problem in time O($\alpha\gamma$mn+N) and space O(N) where α represent the maximum number of strings in any degenerate segment of the text, γ represent the maximum number of degenerate segments spanned by any occurrence of the pattern in the text.

Although the previous algorithm can solve the EDSM problem, it is not efficient enough. Therefore, in [4], it improves the previous algorithm and presents two new algorithms. Both of them have smaller time complexity with O(nm^2+N) and O(N· $\lceil m/\omega \rceil$) respectively. Although both of them need a preprocessing stage, they are also better than previous one in total.

In the first algorithm, it uses suffix tree structure for fast searching. In the second algorithm, it uses the suffix tree structure and the bit vector structure for storing relative information. Constructing structures is a core part of the whole implementation when compiling a program in C++. In website "https://github.com /simongog/sdsl-lite", Succinct Data Structure Library(SDSL) is provided on it. This library provides the Compressed Suffix Trees structure and Integer Vector structure. They are used directly in my implementation instead of creating them by myself.

1.3.2. Background Theories

There are some definitions relative to the algorithm and some structures I used in the implementation. As following,

Definition:

- On-line algorithm[6]: receive a sequence of inputs and performs an immediate action in response to each input.
- Off-line algorithm[6]: receive the entire sequence before performing the response.
- Alphabet(Σ)[5]: a non-empty finite set of letters of size |Σ|.
- String[5]: a finite sequence of letters of Σ.
- Factor(substring)[5]: A string x is a factor of a string y if there exist two strings u and v such that y = uxv.
- ϵ is an empty string, $\sum^+$ are all strings on Σ[5].
- x[i…j][5]:the substring of x from position i to j, x[i…j]=x[i]…x[j].
- Proper factor[5]: a factor of string x and it is not equal x.
- Prefix[5]: if x is the factor of y, there exists a string v such that y=xv.
- Suffix[5]: if x is the factor of y, there exists a string u such that y=ux.
- Border[5]: a proper factor of a non-empty string which is both a prefix and a suffix of string.
- $B_{u,v}$[5]: a set contains all indices i, such that the prefix u[0..i] is the suffix of v.
- An elastic-degenerate string[5]:it can be seen as a two-dimensional array$\tilde{T}$=$\tilde{T}$[0] $\tilde{T}$[1]…$\tilde{T}[n-1]$. Each degenerate letter $\tilde{T}$[i] is a set of deterministic string $\tilde{T}$[i][j]. The length of $\tilde{T}$ is n, the total size N is sum of length of all $\tilde{T}$[i][j] (0≤i<n,0 ≤j<|$\tilde{T}$[i]|).
- Node[9]: a unit of reference in a data structure such tree.
- Root[9]: The initial node of a tree which does not have a parent.
- Internal node[9]: A node of tree has one or more than one children
- Parent[9]: An internal node has a non-empty child, this node is called the parent of its child.
- Child[9]: A node of a tree referred by a parent.
- Leaf[9]: A node in a tree which does not have any children.
- lexicographical order[9]: Alphabetical or "dictionary" order. For example, string AAC > AAA because the letter C is bigger than A in lexicographical order.
- Constructor[12]: a kind of member function which initializes an instance of its class. It has the same name as the class and do not return value.

Structure：

- List[7]: an ordered collection of elements.
- Stack[7]: a linear list in which insertion and removals take place at the same end.

- Vector[8]: it is sequence containers representing arrays that can change in size.
- Map(dictionary)[9]: An abstract data type storing items, or values. Value is accessed by an associated key.
- Tree[9]: A data structure which begins with a root node. Each node in a tree is either a leaf or an internal node. An internal node has equal or more than one child nodes and is called the parent of its child nodes.
- Trie[9]: A tree for storing strings in which there is one node for every common prefix. The strings are stored in extra leaf nodes.
- Suffix Tree[9]: A compact representation of a trie corresponding to the suffixes of a given string where all nodes with one child are merged with their parents[9].
- Suffix Array[9]: An array of all starting positions of suffixes of a string arranged in lexicographical order.

2. Overview of algorithm

Before introducing the detail of the algorithm, I want first to introduce the outline of the pattern matching in an elastic-degenerate text. It is divided into two parts --- "input" and "output".

2.1. Input

In general, there are two inputs:

1) A pattern string P of length m. P has a small length such as m=8, 16, 32 or 64.

2) An *Elastic-Degenerate* text $\tilde{T}$ of length n and size N (N is not less than m). In general, N is far greater than m. For example, a TXT file, which stores the $\tilde{T}$ of real data, can even reach the level of GB. However, the size of pattern is 1 KB (1GB =1000MB, 1 MB = 1024 KB) in general.

2.2. Output

This part is divided into two sections. The first one is about the meaning of "occurrence ending at position j". And the other one is about the output of Elastic-Degenerate String Matching problem.

The meaning of "occurrence ending at position j"[4]:

(1) Occurrence

Assumption the form of $\tilde{T}$ is $\tilde{T} = X[i][j](0 \le i \le n-1, 0 \le j \le |X[i]|)$ where X[i][j] is a string and X[i] is an array of the string. So, $\tilde{T}$ can be represented as following $\tilde{T} = X[0]X[1] \dots X[n-1]$.

We say a pattern P' of length m' match X=X[i]X[i+1]…X[i+m'-1], if following conditions are satisfied:

1) P' can be decomposed into $\mathrm{P'}_0\mathrm{P'}_1 \dots \mathrm{P'}_{m'-1}$

2) there exists a string $s \in$ X[i] and the suffix of s is equal to $\mathrm{P'}_0 (\mathrm{P'}_0 \neq \varepsilon)$

3) if m'>2, there exists $s \in$ X[j] for all $i < j < i + m' - 1$ such that s=$\mathrm{P'}_k$ for all $0 < k < m' - 1$

4) there exists a string $s \in$ X[i+m'-1] and the prefix of s is equal to $\mathrm{P'}_{m'-1} (\mathrm{P'}_{m'-1} \neq \varepsilon)$

(2) ending at position j

If a pattern string matches the text X[i]X[i+1]…X[j], we say that position j in $\tilde{T}$ is one occurrence of P ends. It means that the matching is end at j position.

The output of Elastic-Degenerate String Matching problem:

The output is all positions j in $\tilde{T}$ where at least one occurrence of P ends. For example[4],

The input is:

$$\tilde{T} = \{C\} \cdot \begin{Bmatrix} A \\ C \end{Bmatrix} \cdot \begin{Bmatrix} AC \\ ACC \\ CACA \end{Bmatrix} \cdot \begin{Bmatrix} C \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} A \\ AC \end{Bmatrix} \cdot \{C\}$$

P=ACACA

The output is 2 and 4.

3. EDSM Algorithm

3.1. Introduction

The pseudo code[4] of the EDSM algorithm is presented in APPENDIX B.

3.1.1 Basic Idea

(1) Check whether there exists a string $s \in \tilde{T}[i]$ which suffix is the prefix of the pattern. If the string exists, memories the prefix in a list.

(2) Check whether the partial occurrence, which starts at previous, can be extended at $\tilde{T}[i]$. Record the relative number in the list if the extension is possible.

(3) Finally, check whether the full occurrence is end at $\tilde{T}[i]$. If there is a full occurrence, check whether this position "i" is proposed before. Output position "i" if it never appears.

3.1.2. Input

①Pattern string "P" ②The length of pattern "m" ③Elastic-Degenerate text "$\tilde{T}$" ④The length of Elastic-Degenerate text "n"

3.1.3. Output

All positions j in $\tilde{T}$ where at least one occurrence of P ends

3.1.4. Algorithm

(1)Preprocessing and initial part:

Create a suffix tree of pattern string P in O(m)[4] and create an empty list named L_0 to record the relative number in $\tilde{T}[0]$.

(2)Process the $\tilde{T}[0]$:

At this step, each string in $\tilde{T}[0]$ should be processed. So, there is a loop to process each $\tilde{T}[0][j](0 \le j \le |\tilde{T}[0]|)$. The detailed steps are following: ①Check whether there exists a $\tilde{T}[0][j]$ and its suffix is the prefix of the pattern. This is the first point in the basic idea. If it exists, record the prefix in L_0. ②If the length of $\tilde{T}[0][j]$ is bigger than or equal "m", use KMP algorithm[10] to fast check whether there exists a matching. This is the third point in the basic idea. If matching is found, check the repeat and output "0". ③Go to $\tilde{T}[0][j+1]$ and repeat.

(3)Loop from $\tilde{T}[1]$ to $\tilde{T}[n-1]$:

The procedure for each $\tilde{T}[i](1 \le i \le n-1)$ is same. So, I just introduce one of them. In the beginning, a new list named L_i is created which is used to record the relative number of $\tilde{T}[i]$. I use $\tilde{T}[i][j]\left(1 \le \mathrm{i} \le \mathrm{n}-1, 0 \le \mathrm{j} \le \left|\tilde{T}[i]\right|\right)$ to represent each string in $\tilde{T}[i]$. The process for each $\tilde{T}[i][j]$ is same, and it is following: ①Check whether there exists a string $\tilde{T}[i][j]$ such that its suffix is the prefix of the pattern. This is the first point in the basic idea. If the string exists, record the prefix of it in L_i. ②If the length of $\tilde{T}[i][j]$ is smaller

than “m”, check whether the $\tilde{T}[i][j]$ is the factor of the pattern. If it is, record the start position in array “A”. Then try to find a pair $(\mathrm{p} \in L_{i-1}, \mathrm{x} \in \mathrm{A})$ such that $\mathrm{p}+1=\mathrm{x}$. It means that the outset position of $\tilde{T}[i][j]$ in the pattern is next to the position, which is the end of the prefix of the pattern matching the previous texts. So, record the $\mathrm{p}+|\tilde{T}[i][j]|$ into L_i. It means the factor of the pattern of length $\mathrm{p}+|\tilde{T}[i][j]|$ is matching the text. This is the second point in the basic idea. ③If the length of $\tilde{T}[i][\mathrm{j}]$ is bigger than or equal “m”, it is possible to find a full occurrence at here. So, I use KMP algorithm[10] to fast check whether there exists a matching. This is the third point in the basic idea. ④Compute the $B_{s,p}$ using suffix tree of the pattern. Then try to find a pair $(\mathrm{p} \in L_{i-1}, \mathrm{y} \in \mathrm{Bs}, \mathrm{p})$ such that $\mathrm{p}+\mathrm{y}+2=\mathrm{m}$ where y represents the length of a factor, which is both the prefix of $\tilde{T}[i][j]$ and the suffix of the pattern. If $\mathrm{p}+\mathrm{y}+2=\mathrm{m}$, it means that there is full matching. Because the p and y are beginning at index 0, both of them need to add one to compare with the pattern length. So if the (p,y) pair is found, check the repeat and output. ⑤Go to $\tilde{T}[i][j+1]$ and repeat above steps.

When processing each $\tilde{T}[i][j]$, there is a particular case need to be considered. When $\tilde{T}[i][j]$ represents an empty string(ε), It could be used as a connection symbol that connects the strings in $\tilde{T}[i-1]$ and $\tilde{T}[i+1]$ into one string. For example, the input is:

$$\tilde{T} = \{C\} \cdot \begin{Bmatrix} A \\ C \end{Bmatrix} \cdot \begin{Bmatrix} C \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} A \\ AC \end{Bmatrix}$$

P=AAC

The output is 3.

3.2. Implementation

There are two code files for EDSM algorithm. The first one is “KMP.h” file, and the other one is “EDSM.cpp” file. I store them in the folder “EDSM” in the supplemental file. In the code file, I define the “vector<vector<string>> T” to store the Elastic-Degenerate text. So, I use the variable “T” to represent the $\tilde{T}$ in this section. So, the $\tilde{T}[i][j]$ is represented by “T[i][j]” in this part.

3.2.1. KMP.h

The first step of writing program is calling Library. The table below shows the Library that is used in this file.

The Library Name	Description
string	Defines several functions to manipulate C strings and arrays[8]
iostream	Defines serval function used for output and input stream[8]
list	Defines the list container class[8]
vector	Defines the vector container class[8]

Table 1 The library used in "KMP.h"

After calling the library, I define the class. At this step, I use the file name “KMP” to define the class in order to ensure the correctness. There are four functions and one constructor defined in this file. They are following:

KMP():

This is an empty constructor. Call this constructor first before using its function in other files.

algorithm():

This is a function to implement the KMP searching[10]. The inputs of it are pattern string, pattern string length, Elastic-Degenerate text and Elastic-Degenerate text length. And the output is an integer value. The

basic idea is following. Firstly, compare each position of text with the pattern as the usual way. Then, jump pattern to the position, that store in a relative "KMP_next" array, when a mismatching occurs. The method of creating the "KMP_next" array will be introduced in following.

bordertable():

This function is used to create the border table to check whether there exists a string, that suffix is same to the prefix of the pattern. The inputs are pattern string and a vector. And it returns an integer array. The format of border table can be found in "Example 6" in [4]. In order to implement it, I first create a string named "comb", that stores the combination of the pattern and each string $\tilde{T}[i][j](0 \leq j \leq |\tilde{T}[i]|)$ with a symbol "$". After that, I calculate the border of this string and store it in an integer array "temp". At last, I return "temp". The method of calculating the border will be introduced following.

MP_N():

This a function used to calculate the border of a string and return the border table of it. The inputs are pattern string and its length. And it returns an integer array. Comparing with the usual method of calculating the border, this function uses a new idea, that the border of a border of a string x is the border of x, to decrease the runtime. If the character at index "i" of a string is different from the character at "j", the border of this one is same to previous because the repeat does not occur. Otherwise, the repeat occurs, so the border adds one. In addition, there are two points need to note here. One is that the array size is "m+1" because the correct number is stored in next index. The other one is that the first position of array stores "-1".

```
while((j >= 0) && (pattern[i] != pattern[j])){
    j = border[j];
}
border[i+1] = j + 1;
```

Figure 1 The core part of MP_N() function

KMP_N():

This function is used to calculate the "KMP_next" array depending on the border table and return it. The inputs of this function are pattern string and its length. And it returns an integer array. The aim of this array is that let the jump distance becomes bigger. For example, the pattern is "abacabacab" and text is"abacacabacccab"

Text: a b a c a c a b a c c c a b

Pattern: a b a c a b a c a b

Because the border of "abaca" is 1, so the jump is needed when the above case happens.

Text: a b a c a c a b a c c c a b

Pattern: a b a c a b a c a b

a b a c a b a c a b

From the above case, I can find that two different "b"("abacab"and "ab") are compared with the same "c". This causes low efficiency. So, the "KMP_next" array is used in order to avoid the above case happening. So, the basic idea is following:("MP_next" is an array that stores the border of the string)

$$\text{KMP\_next[i]} = \begin{cases} MP\_next[i], x[i] \neq x[MP\_next[i]] \ or \ i = m \\ KMP\_next[MP\_next[i]], x[i] = x[MP\_next[i]] \end{cases}$$

```
if(pattern[i] == pattern[MP_next[i]]){KMP_next[i] = KMP_next[MP_next[i]];}
else{
    KMP_next[i] = MP_next[i];
    do{MP_next[i] = KMP_next[MP_next[i]];}while((MP_next[i] >= 0) && (pattern[i] != pattern[MP_next[i]]));
}
```

Figure 2 The core part of KMP_N() function

3.2.2. EDSM.cpp

This is the core file of the EDSM algorithm. The libraries used in this file are the libraries of “KMP.h” plus below seven libraries.

The Library Name	Description
KMP.h	Call the “KMP.h” file to compute the border and implement KMP searching
sdsl/suffix_trees.hpp	Define the suffix tree container class in SDSL
sdsl/suffix_arrays.hpp	Define the suffix array container class in SDSL
stack	Defines the stack container class[8]
fstream	Input/output stream class to operate on files[8]
ctime	Define several functions to get and manipulate date and time information[8]
stdio.h	Define function for Input and Output operation[8]

Table 2 The library of "EDSM.cpp"

In this file, I use the structure suffix tree and the suffix array from the website “https://github.com/simongog/sdsl-lite”. I just download SDSL library and install it into the system. The instruction of how to install is presented on the website. In the beginning, I defined the suffix tree structure “cst” as a global variable. There are six functions defined in this file. They are following:

ReadText():

This is a function used to read the Elastic-Degenerate text from a TXT file into the program. The input of it is the path of the TXT file, and the output is a vector. The basic idea is that read character one by one until the end and perform different operation on various inputs. Before talking about the detail, I want first to introduce the format of The Elastic-Degenerate String in a TXT file. For example,

The Elastic-Degenerate String:

$$\tilde{T} = \{C\} \cdot \begin{Bmatrix} A \\ C \end{Bmatrix} \cdot \begin{Bmatrix} AC \\ ACC \\ CACA \end{Bmatrix} \cdot \begin{Bmatrix} C \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} A \\ AC \end{Bmatrix} \cdot \{C\}$$

And the format of it in a TXT file is “C{A,C}{AC,ACC,CACA}{C,E}{A,AC}C”.

In the beginning, I create a “vector<vector<string>> T” which can be seen as a two-dimensional array. So, “T[i][j]” can be used to call the string at position “j” of the array, which locates at index “i” of “T”. The aim is that save one segment(the strings in one parenthesis) into one “T[i]”. So, there are two cases of “T[i]”. One is that “T[i]” stores one string of length one. The other one is that “T[i]” stores more than one strings. So, the basic steps are following. ①Read character by character and store into “T” in order until meeting the “{“. At this point, next “T[i]” must store more than one strings because parenthesis is needed when there are more than one strings in one segment. So, some different operation is necessary here. I use a Boolean value “strat” to mark this point. ②When meet “{“, the characters between it and “,” belong a string. Therefore, I just combine them into one string. And continue to read. ③When meeting a comma, the letters between this comma and next comma or “}” belong one string. So, I just create a new string and repeat. ④Finally, when meeting the “}”, put the string, which is composed of the letters between the last “,” and “}”, into the vector and jump out this parenthesis(change the value of “start”). ⑤Repeat the above process until finish reading all characters.

ReadPattern():

This is the function used to read the pattern string into the program. The input is a string of path, and the output is a string of pattern. Because the pattern is just one string in TXT file, the function “getline()” is used to read the string into the program.

```
string P;
ifstream in(path);
getline(in,P);
```

Figure 3 The core part of ReadPattern() function

ComputeBsp():

This function is used to calculate all $B_{s,p}$ values and return them. The input is a string "T[i][j]" and the output is an integer list. Moreover, the global variable "cst" is used in this function. Depend on the definition, $B_{s,p}$ means that all index of the prefix of s which is also the suffix of p. So, I just need to search "s" in "p" using the suffix tree and check whether there exists matching.

The basic idea is that find the child node of a parent depending on the character in "T[i][j]" and check whether the factor represented by the node is same to the relative factor of "T[i][j]". The detailed steps are following, ①Using the "child()" function to get the node named "A" depend on the first letter of "T[i][j]". "child()" function returns the node, which is the child of a given node, and the first letter of the label of the connection edge is the given letter. If there does not exist this kind of node, return root node. ②Using "extract()" function to get the string that represents the node "A". And check whether this string is same to the relative factor of "T[i][j]" using a FOR loop. ③If two strings are same, check whether the previous "A" is a left. If it is a leaf, stop and put the value into the list. Because, if the node is a leaf, the factor represented by the node is a suffix of the pattern. In addition, I use the prefix of "T[i][j]" as the index for searching. Therefore, a $B_{s,p}$ is found when this condition happens. If two strings are not same, stop and return. ④If two strings are same but the node is not the leaf, justice whether it has a leaf node as its child. If it has, puts the relative value into the list. Because the factor represented by this kind of node is also a suffix depending on the definition of the suffix tree. ⑤If it does not have a leaf node as a child, try to find the child of node "A" depending on the relative letter in "T[i][j]". ⑥Repeat steps ② to ⑤ until each character in "T[i][j]" is used once or stop condition happens. ⑦At the end, check whether the node after the loop is a leaf or it has a leaf as a child to avoid missing.

SerachSinP():

This is a function used to get all the start position of "T[i][j]" in pattern string using suffix tree. The input is a string and it returns an integer list. This function also needs the global variable "cst" to search.

The aim of this function is finding all nodes, that the prefix of the string represented by the node is full matching with the input string "T[i][j]", in suffix tree. The basic steps is similar to function "ComputeBsp()". The detailed steps are following: ①Using the "child()" function to get the node named "A" depending on the first letter of "T[i][j]". ②Using "extract()" function to get the string that represents the node "A". And check whether this string is same to the relative factor of "T[i][j]" using a loop. ③If they are same, repeat using steps ① to find the children of node "A" and compare whether they are exactly matching until the end of string "T[i][j]". I call the stopping node "B". ④If two strings are full matching, all strings represented by the leaves of the subtree, which uses "B" as the root, is complete matching with the string "T[i][j]" because the string represented by the node "B" is the prefix of its children. So, return all values that are stored in the suffix array relative to those leaves.

EDSM():

This function is the main body of the EDSM algorithm. It has four inputs. They are pattern string, pattern string length, Elastic-Degenerate text and Elastic-Degenerate text length. And it returns an integer value which represents the number of occurrences. This is the main implementation of EDSM algorithm depends on the pseudo code.

The detailed steps are following: ①Use the "construct_im()" function to create the suffix tree. And Set

the clock variable to record the time. ②Initial some variables. Moreover, define the header file KMP into variable "k". ③Create a FOR loop to process "T[0]". Firstly, call the function "bordertable()" to get a specific border table and use it to record the $B_{s,p}$ into the list. Because the first number in borer table is "-1", the correct number is shifted one position to the right. So, I use variable "temp" to record the index and use "temp+1" to call the right number. Besides, add one to "temp" at the end of each round of loop to jump the symbol between two strings. In my implementation, I calculate $B_{s,p}$ for each "T[i][j]" at each round of loop instead of calculating all $B_{s,p}$ for "T[i]" in the beginning. It is also correct because $B_{s,p}$ is independent to each. So, there is no different between calculating them together or separately. Secondly, call the function "algorithm()" to implement KMP searching when the length of "T[i][j]" is bigger than length. Then I check repeat and output. In checking repeat part, I use the stack data structure. I define a stack named "out". Because the position will be output in sequence, I just need to check whether the new value is same to the top value of stack. If they are same, it means that there is a repeat. Otherwise, put this new value into the stack. ④Then I created a list named "previous" which is used to store the L_{i-1}. A list is needed instead of an array of the lists because the L_i is only directly depending on L_{i-1} instead of all previous lists. And let "previous" equal L_0. ⑤Create a FOR loop to process "T[i]"$(0 < \mathrm{i} < \mathrm{n})$. This procedure is very similar to process "T[0]". The procedure is same when the length of "T[i][j]" is bigger. The difference is following. Firstly, check whether there exists an empty string "E". If it exists, copy all value of L_{i-1} into L_i because the information of L_{i-1} is needed when calculating the L_{i+1}. Secondly, when the length of "T[i][j]" is smaller than the pattern, use the "SearchSinP()" function to get a list "A" that stores all starting position of "T[i][j]" in the pattern. Then use two FOR loops to traverse all values in "A" and "previous", and try to find a pair $(\mathrm{p} \in L_{i-1}, \mathrm{x} \in \mathrm{A})$ such that $\mathrm{p} + 1 = \mathrm{x}$. Thirdly, "ComputeBsp()" function is used to compute the $B_{s,p}$ at the end of each loop. And try to find a pair $(\mathrm{p} \in L_{i-1}, \mathrm{y} \in \mathrm{Bs}, \mathrm{p})$ such that $\mathrm{p} + \mathrm{y} + 2 = \mathrm{m}$. If it exists, check repeat and output. ⑥Repeat ⑤ until finished all segments. ⑦In the end, calculate the time and output.

main():

This is the main function used to call and test the EDSM algorithm. In this function, I read two texts and two patterns into the program(they are shown below). Then, input them into "EDSM()" function separately and output the results of the algorithm.

3.3. Test

After finished the implementation, I use two simple inputs to test whether the implementation is correct. They are following.

①The input is:

$$\tilde{T} = \{C\} \cdot \begin{Bmatrix} A \\ C \end{Bmatrix} \cdot \begin{Bmatrix} AC \\ ACC \\ CACA \end{Bmatrix} \cdot \begin{Bmatrix} C \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} A \\ AC \end{Bmatrix} \cdot \{C\}$$; n=6; P="ACACA"; m=5.

②The input is:

$$\tilde{T} = \{C\} \cdot \begin{Bmatrix} A \\ C \end{Bmatrix} \cdot \begin{Bmatrix} C \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} A \\ AC \end{Bmatrix}$$; n=4; P="AAC"; m=3.

In theory, the correct result of ① should be position 2 and 4, and the result of ② should be position 3.

The actual output is following

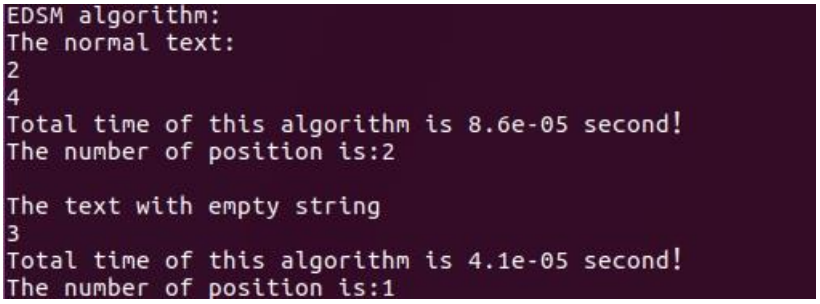


Figure 4 Actual output of EDSM algorithm

From the above picture, it is found that the outputs of the first input are 2 and 4 and the output of the second one is 3. This is same to the result of theoretical results. So, the implementation is correct.

4. EDSM-BV Algorithm

4.1. Introduction

The pseudo code[4] of the EDSM-BV algorithm is presented in APPENDIX C.

4.1.1. Basic Idea

The core idea is similar to EDSM algorithm. But, the difference is that EDSM-BV algorithm uses a bit-vector to store relative values about each segment $\tilde{T}[i]$ of Elastic-Degenerate text instead of the list. In addition, it uses an augmented suffix tree instead of the original one. The difference between these two suffix trees is that each node "u" in augmented suffix tree associated with a bit-vector "B_u". And the value of each "B_u" is following: $B_u[k-1]=1$ if and only if the factor, represented by the node "u", occurs at position k($0<\mathrm{k}<\mathrm{m}-1$) in pattern P. This is mean that there is an occurrence of factor represented by node "u" at terminal nodes in the subtree which use node "u" as root. What's more, create one more bit-vector I_c such that $I_c[k-1]=1$ $(0<k<m-1)$ if and only if P[k]=c.

4.1.2. Input

①Pattern string "P" ②The length of pattern "m" ③Elastic-Degenerate text "$\tilde{T}$" ④The length of Elastic-Degenerate text "n" ⑤The alphabet of text "Σ"

4.1.3. Output

All positions j in $\tilde{T}$ where at least one occurrence of P ends. This is same to EDSM algorithm.

4.1.4 Algorithm

(1)Preprocessing and initial part:

①Create an augmented suffix tree of pattern string P in O(m)[4]. ②Create a bit-vector I_c in O(m)[4]. ③Create a bit-vector B of length "m" with all zeros which plays the same role as L_0in EDSM algorithm.

(2)Process the $\tilde{T}[0]$:

This is step is very similar to process $\tilde{T}[0]$ in EDSM algorithm. The only difference is that save the $B_{p,s}$ in the bit-vector B(B$[B_{p,s}]=1$)instead of the list.

(3)Loop from $\tilde{T}[1]$ to $\tilde{T}[n-1]$:

The core idea of this steps is same to the process in EDSM algorithm. However, the detail of the operation is

different. The detailed steps are following: ①Compute the $B_{p,s}$ and store it into the bit-vector B_1 which is redefined at the beginning of each round of the outer loop. This step can be looked as using the B_1 to replace the L_i of EDSM algorithm. ②When the length of $\tilde{T}[i][\mathrm{j}]$ is smaller than the pattern, the process is also same. $\mathrm{OccVector}_p(\alpha)$: find a node "A" in suffix tree which represents string α, and returns the bit-vector of node "u" where "u" is the first explicit node in the subtree of root A. If string α is not a factor of pattern, return a bit-vector with m 0's. So, this function is used to find all start position of $\tilde{T}[i][\mathrm{j}]$ in the pattern. Moreover, utilize AND operation to check whether this $\tilde{T}[i]$ can be extended. And add it into B_1 if partial occurrence can be extended at here. This operation is just like that find pair $(\mathrm{p} \in L_{i-1}, \mathrm{x} \in \mathrm{A})$ such that $\mathrm{p}+1=\mathrm{x}$ in EDSM algorithm and add extended prefix index into the list. ③When the length of $\tilde{T}[i][\mathrm{j}]$ is bigger than the pattern, EDSM-BV algorithm uses the same KMP algorithm to fast check matching. ④At the end of each round, check whether there is a full occurrence depending on the bit-vector I_c. Firstly, AND the previous bit-vector with I_c and shift one position to check whether there exists a full occurrence. This operation is just like finding the pair $(\mathrm{p} \in L_{i-1}, \mathrm{y} \in \mathrm{Bs}, \mathrm{p})$ such that $\mathrm{p}+\mathrm{y}+2=\mathrm{m}$ in EDSM algorithm. ⑤Let bit-vector B equal B_1 that just like copy L_{i-1} into list "previous". Then go to $\tilde{T}[i][j+1]$ and repeat. In this algorithm, the empty string also exists. It also can be seen as a connection symbol.

4.2. Implementation

There are two code files for implementation of the EDSM-BV algorithm. The first one is "KMP.h" file, and the other one is "EDSM_BV.cpp" file. I store them in the folder "EDSM_BV" in the supplemental file. In the code file, I define the "vector<vector<string>> T" to store the Elastic-Degenerate text. So, I use the variable "T" to represent the $\tilde{T}$ in this section. So, the $\tilde{T}[i][j]$ is represented by "T[i][j]" here.

4.2.1. KMP.h

This file is entirely same to the "KMP.h" file of EDSM algorithm. The main aim of this file is implementing KMP algorithm and computing the border table. Because these parts are same for two algorithms, "KMP.h" is same.

4.2.2. EDSM_BV.cpp

This is the core file of the EDSM algorithm. The libraries used in this file are the libraries of "EDSM.cpp" plus below seven libraries.

The Library Name	Description
sdsl/bit_vectors.hpp	Define the bit_vector container class in SDSL
map	Defines the map and multimap container classes[8]
algorithm	Defines a collection of functions about algorithm[8]

Table 3 The library used in "EDSM_BV.cpp"

The suffix tree and bit-vector structure are also directly used from the website "https://github.com/simongog/sdsl-lite". In the beginning, I defined the suffix tree structure "cst" as a global variable. This file contains eight functions. They are following,

ReadText():

This is a function used to read the Elastic-Degenerate text from a TXT file into the program. The input of it is the path of the TXT file and output is a vector. This function is same to the "ReadText()" function in "EDSM.cpp" file.

ReadPattern():

This is the function used to read the pattern string into the program. The input is a string of path and returns a string of pattern. This is same to the “ReadPattern()” function in “EDSM.cpp” file.

ReadAlphabet():

This function is used to get alphabet of the Elastic-Degenerate text. The input is a vector that store the Elastic-Degenerate text and the output is a string of alphabet. The idea is that read each letter in the vector and add the different letter to a string and return this string at last.

CreateIc():

This is a function used to create bit-vector I_c for each letter “c”. The inputs are pattern string, the string of alphabet and the length of pattern and the output is a map that contains all letters in alphabet associated with a bit-vector. The basic idea is following: ①Create some all zero bit-vectors depending on the length of alphabet string. ②Loop from 0 to m-1, change the bit-vector I_c such that $I_c[k-1] = 1\ (0 < k < m-1)$ when P[k]=c.

OccVector():

This function is used to create the augmented suffix tree. The inputs are suffix tree, and the length of pattern and the output is the map that contains the string, that represented by the node in suffix tree, associated with a bit-vector. The basic idea is adding the bit-vector from the leaves to the root. I create a map “OccV” to store relative information. The index is the factor represented by the node and the value is the relative bit-vector.

The detailed steps are following: ①Create a FOR loop to traverse the node of the suffix tree from the bottom to top. ②Use “is_leaf()” function to check whether the node is a left. If it is, create a bit-vector such that B[k-1] = 1 where the string, represented by the node, is the suffix P[k…m-1]. Then, add the string and bit-vector into “OccV”③If the node is not a leaf, create a bit-vector which be gained by using OR operation between all bit-vectors of the children of the node. Because the sequence of calling node in the bottom-up method, all children’s bit-vector is done when calling a parent node. So, it can guarantee the correctness of this implementation. Then, add string and bit-vector into the map “OccV”.

In this function, there are two points need to be careful. Firstly, the bit-vector needs to be changed into an integer first and change back after the operation because OR and AND operation cannot be used between two bit-vector in this structure. Secondly, root node needs to be checked separately. Because the factor of the root node is empty, “extract()” function cannot be used for it. Therefore, I set a space as the index for the root node.

findOccVectorS():

This function is used to get the bit-vector in augment suffix tree depend on a given string. The input is the string T[i][j] and the output is a string that represented by the relative node. The function needs the global variable “cst”. Why use this function to get a string instead of directly using the “T[i][j]”? Because “T[i][j]” may be a factor of the pattern but it cannot be represented by a node in the suffix tree. For example, the pattern is “ACA” the suffix tree is following:

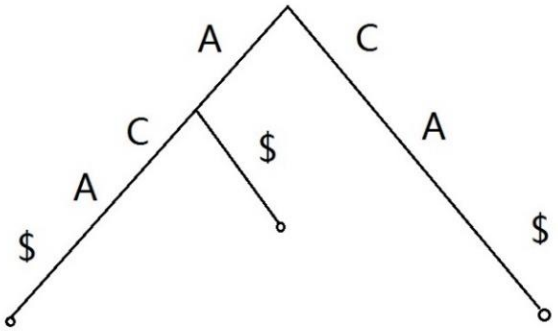


Figure 5 Suffix Tree of String "ACA"

And the “T[i][j]” is string “C”. In this case, there is no node in this suffix tree can direct represent the “T[i][j]” but it is the factor of “ACA”. So, use this function to return the nearest child node. In this case, it is “CA$”. Then depending on the return string to get the relative bit-vector.

The step of this function is similar to “SerachSinP()” in EDSM algorithm. The specific steps are: ①Use “child()” function to find the child node “A” of root depending on the first letter of string “T[i][j]” and check whether the factor, represented by the node, is same to the corresponding substring of “T[i][j]”. ②Find the child node of the node “A” depending on the relative letter in “T[i][j]” and check full matching again. ③Repeat the above process until the end of the “T[i][j]”. Then return the string represented by the node or empty if mismatching.

EDSM_BV():

This function is the main body of the EDSM-BV algorithm. It has five inputs. They are pattern string, pattern string length, Elastic-Degenerate text, Elastic-Degenerate text length and the alphabet. And it returns an integer value which represents the number of occurrences. This is the main implementation of the EDSM-BV algorithm depending on the pseudo code.

The detailed steps are following: ①Use the function “CreateIc()" to create the bit-vector for each letter in the alphabet. Create a suffix tree use function “construct_im()” and update it into an augmented suffix tree using “OccVector()” function. ②Set the clock to record the time and initial variables. ③Create a FOR loop to process “T[0]”. This step is similar to the process “T[0]” in EDSM algorithm. And the only difference is that store $B_{s,p}$ in bit-vector “vecB” instead of a list. ④Transfer the bit-vector “vecB” into an integer “B” before processing the next segment. Because this bit-vector structure cannot directly use OR and AND operation, transfer it into an integer is needed. ⑤Create a FOR loop to process “T[i]”$(0 < \mathrm{i} < \mathrm{n})$. This part is also similar to EDSM algorithm. The difference is following. Firstly, when the length of “T[i][j]” is smaller than the pattern, use the “findOccVector()” function and combine the map of augmented suffix tree to get the start position. Then, store it into variable “occ_vec”. Then, use it AND with “B” to check whether previous partial occurrence can be extended at here. Some transfers are needed at here because a bit-vector is used to store $B_{p,s}$. Secondly, when checking whether there is a full occurrence at the end of each round, combine the integer “B” and the bit-vector I_c to get the result. There is also need a transfer in order to run correctly. Apart from these two points, the rest is the same. ⑥Repeat ⑤ until finished all segments. ⑦At the end, calculate the time and output.

main():

This is the main function used to call and test the EDSM-BV algorithm. This is same to EDSM algorithm except to call EDSM_BV() function instead of EDSM().

4.3. Test

After finish the implementation, I use the same inputs of EDSM algorithm to test whether this implementation is correct. The actual output is

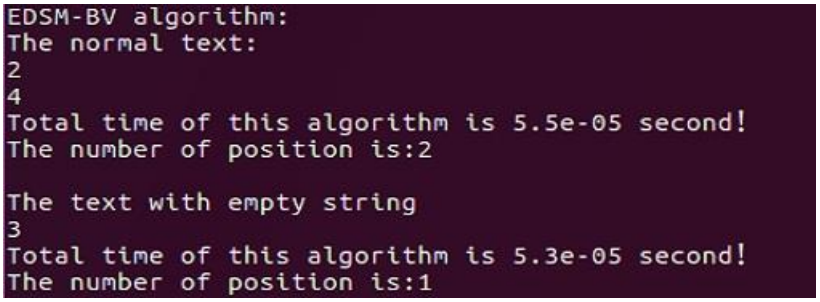


Figure 6 Actual output of EDSM-BV algorithm

It is also same to the result of theoretical results. Therefore, the implementation is correct.

5. NAIVE Algorithm

5.1. Introduction

5.1.1. Basic Idea

The core idea of the naïve algorithm is very simple that is comparing each possible string in Elastic-Degenerate text with the pattern once.

5.1.2. Input

①Pattern string "P" ②The length of pattern "m" ③Elastic-Degenerate text "$\tilde{T}$" ④The length of Elastic-Degenerate text "n"

5.1.3. Output

All positions j in $\tilde{T}$ where at least one occurrence of P ends.

5.1.4 Algorithm

There are three loops in this algorithm. The first one is a loop for $\tilde{T}[\mathrm{i}](0 \leq \mathrm{i} \leq \mathrm{n}-1)$. And the second one is for $\tilde{T}[\mathrm{i}][\mathrm{j}](0 \leq \mathrm{j} \leq |\tilde{T}[\mathrm{i}]|)$. The third loop is for $\tilde{T}[\mathrm{i}][\mathrm{j}][\mathrm{k}](0 \leq \mathrm{k} \leq |\tilde{T}[\mathrm{i}][\mathrm{j}]|)$. The detailed is following: ①Compare $\tilde{T}[\mathrm{i}][\mathrm{j}][0 \ldots \mathrm{k}](\mathrm{k} = |\tilde{T}[\mathrm{i}][\mathrm{j}]|)$ with the same length factor of pattern from index 0 to the smaller one of "k"(k =$\tilde{T}[\mathrm{i}][\mathrm{j}]$) and m(pattern length). ②If the length of $\tilde{T}[\mathrm{i}][\mathrm{j}]$ is smaller than the pattern, then go to $\tilde{T}[\mathrm{i}+1][0]$ and compare $\tilde{T}[\mathrm{i}+1][0][0 \ldots \mathrm{k}](\mathrm{k} = |\tilde{T}[\mathrm{i}+1][0]|)$ with the same length factor of remain pattern(those part that is not compared with $\tilde{T}[\mathrm{i}][\mathrm{j}][0 \ldots \mathrm{k}]$). If there are some parts of the pattern that is also not compared, go to $\tilde{T}[\mathrm{i}+2][0]$ and repeat. Repeat above process until all letter of the pattern is compared once. ③Assume, when the comparison is finished, the index stops at $\tilde{T}[\mathrm{i}+5][0]$, then try to call $\tilde{T}[\mathrm{i}+5][1]$and compare it with the substring of the pattern that is compared with $\tilde{T}[\mathrm{i}+5][0]$ and if the pattern is not full compared then go to $\tilde{T}[\mathrm{i}+6][0]$. ④If $\tilde{T}[\mathrm{i}+5][1]$ does not exist, back to $\tilde{T}[\mathrm{i}+4][\mathrm{j}+1]$ (j is the position in $\tilde{T}[\mathrm{i}+4]$ that already compared last time) and repeat ②③. ⑤If the $\tilde{T}[\mathrm{i}+1]$ does not exist, but the comparison of the pattern does not finish, I just record a mismatch and continue(for example

go to $\tilde{T}[\mathrm{i}][j+1]$). ⑥Whenever the comparison of all string in $\tilde{T}[\mathrm{i}]$ is finished, back to $\tilde{T}[\mathrm{i}-1]$ and repeats above steps until arriving the original one. ⑦When backing to $\tilde{T}[\mathrm{i}][\mathrm{j}]$, start to compare $\tilde{T}[\mathrm{i}][\mathrm{j}][1 \ldots \mathrm{k}](\mathrm{k} = |\tilde{T}[\mathrm{i}][\mathrm{j}]|)$with the same length substring of the pattern from the begin. And repeat the above steps until finish the comparison that uses $\tilde{T}[\mathrm{i}][\mathrm{j}][\mathrm{k} \ldots \mathrm{k}](\mathrm{k} = |\tilde{T}[\mathrm{i}][\mathrm{j}]|)$ as the start. ⑧Move to $\tilde{T}[\mathrm{i}][\mathrm{j}+1]$ to repeat the above steps until finish all comparison of $\tilde{T}[\mathrm{i}]$. ⑨Move to $\tilde{T}[\mathrm{i}+1]$ and repeat the above steps until all done.

For example, if the T is following and the length of pattern string is 3

$$\tilde{T} = \{C\} \cdot \begin{Bmatrix} A \\ C \end{Bmatrix} \cdot \begin{Bmatrix} AC \\ ACC \\ CACA \end{Bmatrix}$$

The sequence of the strings that compare with the pattern is following(The string in brackets is used to show which string is called at the end):

“CAA(C)”,”CAA(CC)”,”CAC(ACA)”,”CCA(C)”,”CCA(CC)”,”CCC(CACA)”,”AAC”,”AAC(C)”,”ACA(CA)”,

“CAC”,”CAC(C)”,”CCA(CA)”,”ACC”,”CAC(A)”,”ACA”.

5.2. Implementation

There is only one code file for the naïve algorithm which is named “NAÏVE.cpp”. I store it in the folder “NAIVE” in the supplemental file. In the code files, I define the “vector<vector<string>> T” to store the Elastic-Degenerate text. So, I use the variable “T” to represent the $\tilde{T}$ in this section. So, the $\tilde{T}[i][j]$ is represented by “T[i][j]” here.

5.2.1. NAIVE.cpp

This is the core file of the EDSM-BV algorithm. The libraries used in this file are the libraries of “KMP.h” plus below four libraries.

The Library Name	Description
stack	Defines the stack container class[8]
fstream	Input/output stream class to operate on files[8]
ctime	Define several functions to get and manipulate date and time information[8]
stdio.h	Define function for Input and Output operation[8]

Table 4 The library used in "NAIVE.cpp"

This file contains five functions. They are following,

ReadText():

This is a function used to read the Elastic-Degenerate text from a TXT file into the program. The input of it is the path of the TXT file and output is a vector. This function is totally same to the “ReadText()” function in “EDSM.cpp” file.

ReadPattern():

This is the function used to read the pattern string into the program. The input is a string of path and returns a string of pattern. This is same to the “ReadPattern()” function in “EDSM.cpp” file.

loop():

This is an iterative function which is used to call the next segment of Elastic-Degenerate text until finished all comparison. The inputs are a vector iterator, the length of the factor that already be compared, the pattern string, the index of the text and the length of the text. And the output is an integer list which stores the index of the full occurrence. The basic idea is that compare all string of one “T[i]” in a for loop and

use the “loop()” function to call next segment when needed.

The details are following: ①Initial variables to store relative value and create a loop for all strings in one “T[i]”. ②Check whether the string is an empty string. If it is, call the “loop()” function to go to next segment. ③If the string is not empty, using a FOR loop to compare each letter of this string with the relative factor of the pattern. If the comparison is not finished, use the “loop()” function to go to next segment. ④If comparison is completed, check whether there is a full occurrence. If the result is true, put this position into the list “appear”. ⑤Check whether there exists another full occurrence in next segment. This can be found in list “output”. If it is empty, there is no full occurrence in next segment. Otherwise, combine the list “output” and “appear” together. ⑥Return the list “appear”.

Naïve():

This is the main body of the NAÏVE algorithm. It has four inputs. They are pattern string, pattern string length, Elastic-Degenerate text and Elastic-Degenerate text length. And the output of it is the number of full occurrences.

The detailed steps are following: ①Set the clock to record the time and initial variables. ②Create two loops for “T[i]”$(0 \leq \mathrm{i} \leq \mathrm{n}-1)$ and “T[i][j]”$(0 \leq \mathrm{j} \leq |\mathrm{T}[\mathrm{i}]|-1)$ respectively. ③Check whether the “T[i][j]” is an empty string. If it is, go to “T[i][j+1]”. Otherwise, go to next step. ④Create a loop for “T[i][j][k]”$(0 \leq \mathrm{k} \leq |\mathrm{T}[\mathrm{i}][\mathrm{j}]|-1)$. And use a FOR loop to compare each letter of the relative factor, that is the substring of “T[i][j]” and the start position is “k”, with the corresponding factor of the pattern. If the length of “T[i][j]” smaller than the pattern, call the “loop()” function to finish the comparison. ⑤Check whether there exists the full occurrence depends on variable “end”(a Boolean value for marking the comparison end) and list “output”(store the position of full occurrence in following segments). This step is also included in the loop of ④. ⑥If there exists the full occurrence, check repeat and output. In checking repeat part, I cannot just compare the top value of the stack because the position does not output in order. There are four cases. The first one is that the stack “out”(used to store the position of full occurrence) is empty and there is a full occurrence in “T[i]”. In this case, put the index “i” into stack and output. The second case is that the “out” is not empty and there is a full occurrence in “T[i]”. In this case, check the index “i” with all values in “out”. If it does not repeat, put this value into stack and output. The third case is that the stack is empty and there are full occurrences in following segments. In this case, check the repeat in list “output” and add the non-repeating value into stack and output. The last case is that the stack is not empty and there are full occurrences in following segments. In this case, check each value in “output” with each value in “out”. If the value in “output” does not repeat, put it into the stack “out” and output. ⑦At the end, calculate the time and output.

main():

This is the main function used to call and test the NAIVE algorithm. This is same to EDSM algorithm except to call Naïve() function instead of EDSM().

5.3. Test

After finish the implementation, I use the same inputs of EDSM algorithm to test whether this implementation is correct. The actual output is

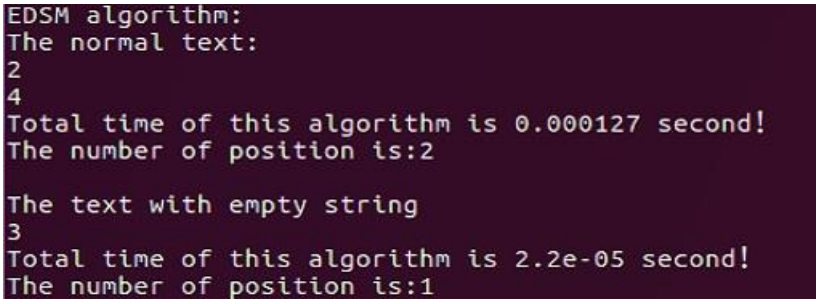


Figure 7 Actual output of NAIVE algorithm

It is also same to the result of theoretical results. Therefore, the implementation is correct.

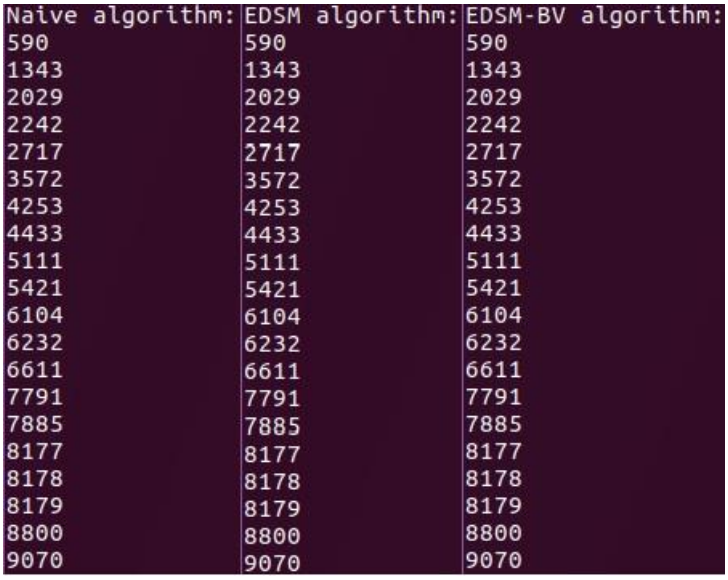
6. Comparison

6.1. Example Data Test

Before comparing the runtime, the correctness should be ensured for a larger data. So, I try to test these three algorithms using an example data. This data is downloaded directly from the following website "https://github.com/webmasterar/edsm/tree/master/exampledata". There are two TXT files, one is the text named "seq.txt" and the other one is the pattern named "pattern.txt". At this step, I combine three algorithms into one file "COMBINE.cpp" in order to compare more convenient. In addition, I can use the same compile method to compile them. And I store it in the folder "COMBINE_THREE_ALGORITHMS" in the supplemental file.

I change the "ReadText()" function, "ReadPattern()" function and "main()" function. The rest part is just copied from these three algorithms. For the "ReadText()" function, I just change the string, the output when the open file fails, from "Error opening file" into "Error opening file of text" and return an empty vector when an error happens. For the "ReadPattern()" function, I just add an IF condition that checks whether it read an empty pattern. If the string is empty, output "Error opening file of pattern". Why do I add it here? Because the path of the pattern is defined inside of the main function in the previous file, it must be correct. However, when I use the external parameter, there may occur some mistakes in the format of the path. So, the IF condition is used to check whether the pattern is loaded correctly. For the main function, I add two parameters "argc" and "argv[]" to read the outside input into this function. For my COMBINE.cpp file, I want the user to input three outside inputs. They are the path of text, the path of pattern and the type of algorithm. I depend on first two inputs to create the Elastic-Degenerate text "$\tilde{T}$" and pattern string "P". And choose to run which algorithm depending on the third inputs.

After the changing, I use the example data to test NAÏVE, EDSM and EDSM-BV algorithm and the result of them is shown below.



Naive algorithm:	EDSM algorithm:	EDSM-BV algorithm:
590	590	590
1343	1343	1343
2029	2029	2029
2242	2242	2242
2717	2717	2717
3572	3572	3572
4253	4253	4253
4433	4433	4433
5111	5111	5111
5421	5421	5421
6104	6104	6104
6232	6232	6232
6611	6611	6611
7791	7791	7791
7885	7885	7885
8177	8177	8177
8178	8178	8178
8179	8179	8179
8800	8800	8800
9070	9070	9070

Figure 8 The result of three algorithms depend on example data

From the above picture, it is found that the results of these three algorithms are totally same. So, I believe that the implementations of them are correct.

6.2. Runtime Comparison in Synthetic Data

6.2.1. Problem of Runtime

At the start, the implantations of EDSM and EDSM-BV algorithm are a little bit different. There are two inputs for the function ComputeBsp(), SerachSinP() and findOccVectorS() instead of one. The other input is the suffix tree structure "cst". However, there occurs a problem when using the synthetic data to test. The runtime of both EDSM and EDSM-BV algorithm is slower than the NAÏVE algorithm at some inputs. Therefore, I split the entire algorithm into a lot of small parts and test their runtime separately. After a long-time checking, I find that the function will spend some time to load some irrelevant functions of the structure whenever using the suffix tree structure as an input. So, some time redundancy will appear when calling the function. From the structure of the EDSM and EDSM-BV algorithm, it is found that all these three functions will be called once when processing each $\tilde{T}[i][j]$. Therefore, there is an enormous redundancy in total. So, it leads to a decline in efficiency.

After finding the problem, I try to remove the suffix tree input. But, all these three functions need to use the suffix tree structure inside. Therefore, I define the suffix tree structure "cst" as a global variable. In this way, the "cst" can be directly called at anywhere without causing the redundancy.

After changing, the suffix tree structure "cst" is defined as the global variable. And all of function ComputeBsp(), SerachSinP() and findOccVectorS() only have one input.

6.2.2. The Result of Comparison

In this part, I use the synthetic data to compare the efficiency of these three algorithms. The data is downloaded from the website "https://github.com/webmasterar/edsm/tree/master/experiments/syntheticdata". There are two folders in "syntheticdata". They are "randomPatterns" and "syntheticDatasets". There are four patterns("8.txt", "16.txt", "32.txt" and "64.txt") in "randomPatterns" folder. The name of the TXT is the length of pattern that store in the TXT file. In addition, there are five TXT files in "syntheticdata"

folder. The name of them are “100000_10.txt”, “200000_10.txt”, “400000_10.txt”, “800000_10.txt” and “1600000_10.txt”. The number in the name is the length of Elastic-Degenerate text that stores in the TXT file. It means that the “100000_10.txt” stores an Elastic-Degenerate text of length 100000. Therefore, combining different texts with different patterns, there are twenty sets of data for each algorithm.

I run each set three times and use the average of them as the final result of this set for one algorithm. And I plot the results of the same length pattern in one graph. So, I have four figures. Each figure contains five sets for one algorithm. They are shown below.

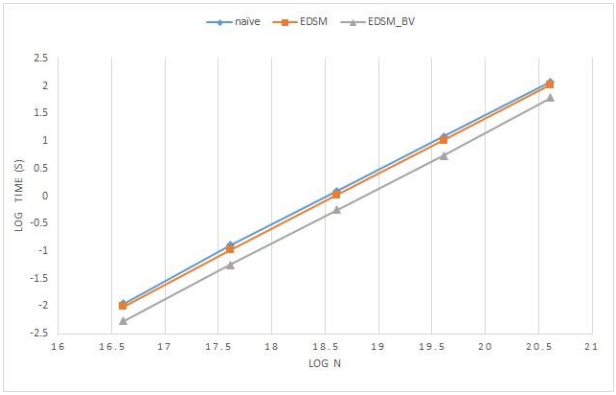


Figure 9 pattern of length m=8

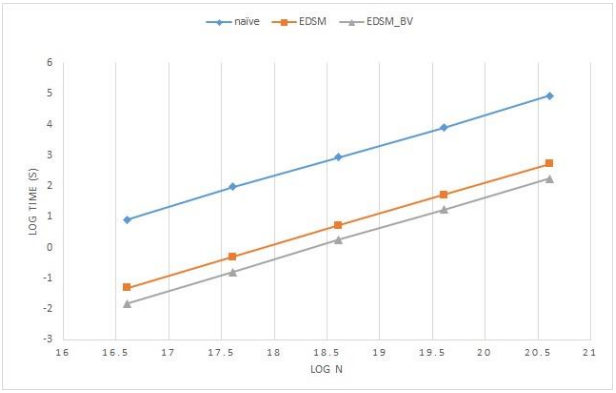


Figure 10 pattern of length m=16

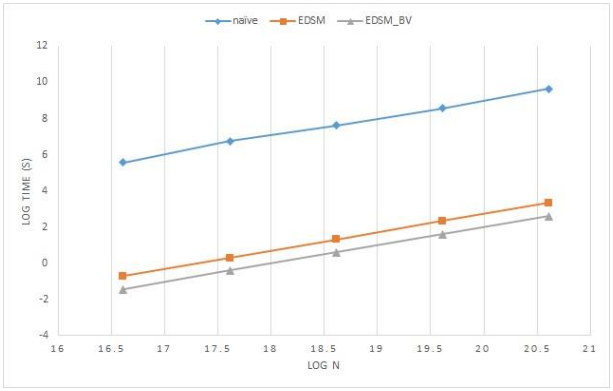


Figure 11 pattern of length m=32

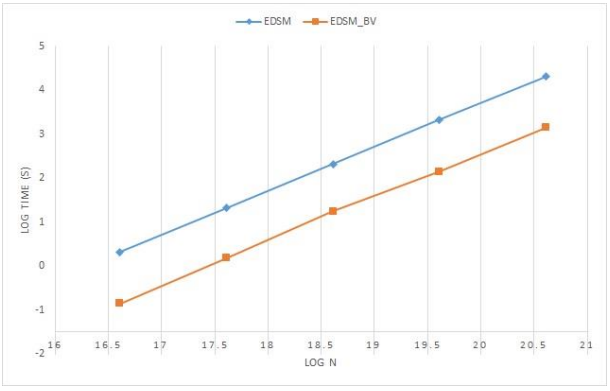


Figure 12 pattern of length m=64(there is no NAÏVE algorithm because its execution time is too long)

In the fourth figure, there is no NAÏVE Algorithm because it needs more than three hours to deal with the text “100000_10.txt” and pattern “64.txt”. It is very slow. So, I just ignore it in the last graph. The abscissa of four pictures is $log_2 n$(n is the length of Elastic-Degenerate text) and the ordinate is $log_2(time)$(the unit of time is second). From the above figures, it is found that the gap between the runtime of three algorithms becomes bigger when the length of the pattern increases. However, the sequence of them does not change that is NAÏVE Algorithm > EDSM Algorithm>EDSM-BV Algorithm. The theoretical runtime of EDSM algorithm is $\mathrm{O}(\mathrm{n}m^2 + \mathrm{N})$[4] and the theoretical runtime of EDSM-BV algorithm $\mathrm{O}(\mathrm{N} \cdot \lceil m/\omega \rceil)$. And the theoretical runtime of the NAÏVE algorithm is following. The best case is that each segment only has one string that means n=N. In this case, each string, which uses the different letter as the head, compares with the pattern once. So, the runtime is $\Omega(\mathrm{m} \cdot \mathrm{N})$. The worst case is that each segment contains the same amount of string(more than one) and each string only has one letter. In this case, the runtime is

$$\mathrm{O}\left(\mathrm{N} \times \left(\frac{N}{n}\right)^{m-1} \times m\right) = \mathrm{O}\left(\frac{m \cdot N^m}{n^{m-1}}\right)$$

Where N is the size of text, $(N/n)\hat{}(m-1)$ is the number of string use the same letter as the head and m

is the time complexity of the comparison.

Therefore, when the pattern length is small, there is not much difference in the runtime of these three algorithms. However, when the pattern length is bigger, the runtime of the NAÏVE algorithm increases exponentially. So, it is very slow when pattern length is large. In addition, Because the m^2 factor in the time complexity of EDSM algorithm becomes more and more significant when pattern length grows[4], EDSM is slower than EDSM-BV algorithm when pattern length is huge. So, this is why the sequence is NAÏVE > EDSM > EDSM-BV and the gap between them become larger and larger when pattern length increases.

6.3. Runtime Comparison in Real Data

In this part, I just compare the EDSM and EDSM-BV algorithm. Because the NAÏVE algorithm is an off-line algorithm, the program needs to read the entire Elastic-Degenerate text into the memory before processing the first segment. So, it is mean that all real data need to be stored in "<vector<vector<string>>> T" before calling the algorithm. However, the size of the real data is far greater than the synthetic data. The file, which stores the real text, can reach ten more GB. However, the maximum text file in synthetic data is "1600000_10.txt" which is only 6.5MB. We know 1GB=1000MB. So, there is a huge gap between the size two files. So, if I try to read all text into the program before processing, it needs a huge memory space to store them. It will reduce the efficiency even cause a crash if the memory space is not huge enough. Therefore, the NAÏVE algorithm is not considered in this section. But why there is no effect on the EDSM and EDSM-BV algorithm? Because both of them are the on-line algorithm. They all can perform an immediate action in response to each input. From the code of EDSM and EDSM-BV, it is found that the algorithm only needs the list(EDSM) or bit-vector(EDSM-BV) of previous one segment when dealing with a new one. So, each input can be processed independently. Therefore, they are the online algorithm.

Before testing, change is needed in "COMBINE.cpp" file because it reads all text first before performing the algorithm. I named the new file as "COMBINE_UPDATE.cpp". And I store it in the folder "COMBINE_EDSM_AND_EDSM-BV" in the supplemental file. The detailed changes are following: ①Delete everything about the NAÏVE algorithm. ②Delete the "ReadAlphabet()" function because the entire text is not known before running the algorithm. So, the string "ACGTN" is used as the default. ③Update the "ReadText()" function into two new functions in order to call the algorithm in an on-line way. ④Add a function named "ReadTextforEDSM()" function, this is a update version of "ReadText()" function. The changes are shown below. Firstly, create a suffix tree here instead of in "EDSM()" function. The reason is that, if the suffix tree is also created in "EDSM()", it will be created more than once because "EDSM()" is called many times. Secondly, calling the "EDSM()" function whenever finish reading one segment. And add the runtime into "totalTime"(a variable used to record total time). Thirdly, check whether the segment is the first one and set a Boolean variable "first". If the segment is the first one, it is TRUE. Otherwise, it is false. ⑤Add a function named "ReadTextforEDSM_BV()" function, it is also a update version of "ReadText()". It is similar to "ReadTextforEDSM()" function except following two points. Firstly, create bit-vector I_c and augmented suffix tree in this function. The reason is same as why creating a suffix tree in "ReadTextforEDSM()" function. Secondly, call "EDSM_BV()" function instead of "EDSM()" function. ⑥Change the "EDSM()" algorithm. Firstly, define the list "previous" and the index "i" into the global variable. Because they need to be used to store the value, that is transferred from the previous segment into the next. Secondly, change the third input from the Elastic-Degenerate text(vector<vector<string>>) into one segment(vector<string>) because "EDSM()" function only processes one segment whenever it is called. Thirdly, add a Boolean variable as the input which

is used to decide which part of the code will be executed in following. Fourth, removing all time-related function. ⑦Change the "EDSM_BV()" function. The changes are similar to "EDSM()". Firstly, define the integer "B" and the index "i" into the global variable. Secondly, change the third input from the Elastic-Degenerate text(vector<vector<string>>) into one segment(vector<string>) and delete the alphabet input. Thirdly, add a Boolean variable as the input in order to decide which part should be compiled. Fourth, removing all time-related function.

Apart from the above seven points, the rest part is basically same to "COMBINE.cpp" file. After changing, test correctness of the modification is needed. So, the example data(which is mentioned in "6.1. Example Data Test") is used to test the correctness of them. The results of them are following:

EDSM Algorithm:	EDSM-BV Algorithm:
590	590
1343	1343
2029	2029
2242	2242
2717	2717
3572	3572
4253	4253
4433	4433
5111	5111
5421	5421
6104	6104
6232	6232
6611	6611
7791	7791
7885	7885
8177	8177
8178	8178
8179	8179
8800	8800
9070	9070

Figure 13 The result of updated EDSM and EDSM-BV algorithm depend on example data

From the above result, it is found that it is totally same with the result in "6.1. Example Data Test". So, the updated algorithm should be correct.

This real data is downloaded directly from the following website "https://github.com/webmasterar/edsm/tree/master/experiments/realdata". There are two files, one is a folder named "randomPatterns" and the other is a MD file named "README". "randomPatterns" contains four TXT files, they are "8.txt", "16.txt", "32.txt" and "64.txt". The format of them is same to above. But the text for real data is different. From the "README.md" file, I know that the Elastic-Degenerate text is not just a TXT file. It can be seen as a combination of three different kinds of documents. There is an easy way to transfer them into the same format of previous text. There is a website "https://github.com/webmasterar/edso". It provides a little utility program that combines a reference FASTA file and its VCF file into an EDS format file. So, this program is used to transfer the real data into an EDS format file. After getting the EDS file, I directly change the extension from EDS into TXT. Because the format of the content is same for these two files. So, I can directly alter the extension name without changing the detail. Besides, it can be read faster if using the TXT file as the input. Then, I use the data from the website "ftp://ftp.ensembl.org/pub/release-75/fasta/homo_sapiens/dna/". This dataset is obtained from Phase 3 of the 1, 000 Genomes Project[11]. I choose the five smallest chromosomes as the text inputs. I first use the "edso" program to transfer these five sets of chromosomes into five EDS files. After that, I change them into five TXT files by changing the extension name. And I combine them with four patterns to get twenty sets of inputs. And I input these data into EDSM and EDSM-BV algorithm and record their runtime. During the test, the system should have enough free memory

to ensure running successfully. And I draw the runtime of the same pattern length in one graph. So, I get the following four figures.

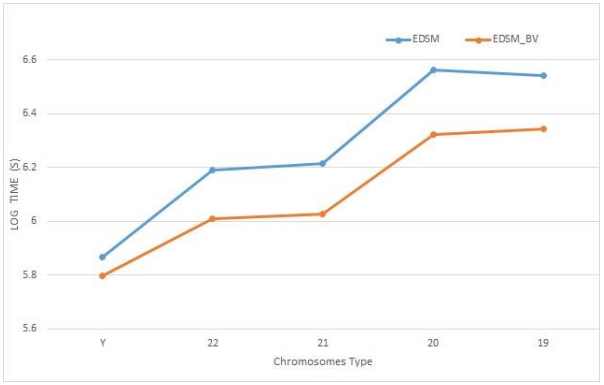


Figure 14 pattern of length m=8

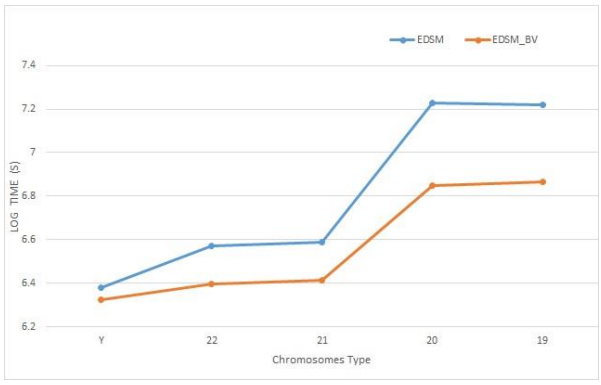


Figure 15 pattern of length m=16

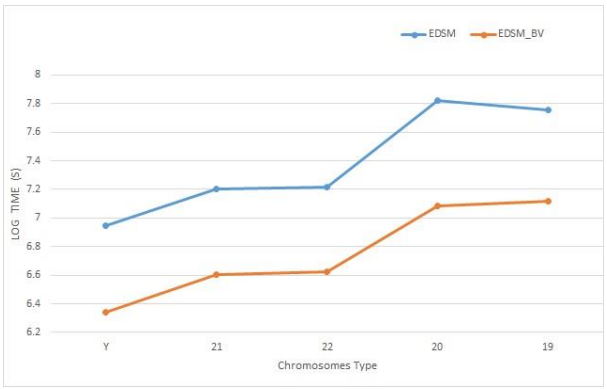


Figure 16 pattern of length m=32

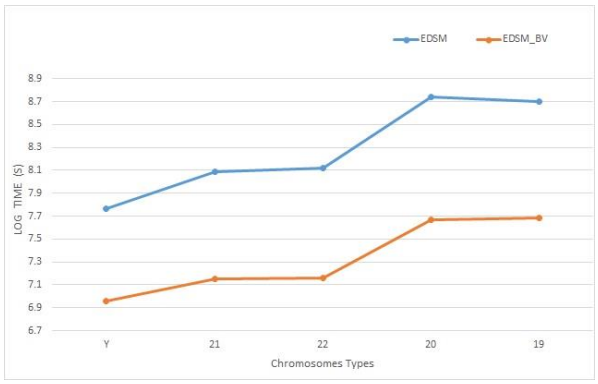


Figure 17 pattern of length m=64

The abscissa of above four graphs is “Chromosomes Types” and the ordinate is $log_2(time)$(the unit of time is second). The sequence of the “Chromosomes Types” is the number of the N', the total number of strings $\tilde{T}[i][j]$, that the length of it is smaller than the pattern and the previous list(EDSM) or bit-vector(EDSM-BV) is not empty, per chromosome from less to more. It means that the number of N' in the text is Y>22>21>20>19. From the above graphs, I can find that the EDSM-BV algorithm is more efficiency among these two algorithms. And the runtime increases as the number of N' increases. The reason why the result is this for EDSM-BV is explained in [4]. And the reason for EDSM is similar. It is also because update the list “previous” is time-consuming.

7. Conclusion

7.1. Summary of Project

At the beginning of the report, I stated the project’s aims and objectives that are understanding, implementation and comparison of two on-line algorithms of Elastic-Degenerate String Matching problem --- EDSM and EDSM-BV.

After that, I focus on introducing the core idea and implementation detail of these two algorithms. In addition, I introduce a naïve algorithm which is designed and created by myself. The reason of why the naïve algorithm is needed is that it is used to show the high efficiency of EDSM and EDSM-BV algorithm.

At last, I compare all three algorithms using the synthetic data and compare EDSM and EDSM-BV using real data. And the result is that both EDSM and EDSM-BV algorithm have the better efficiency comparing with the NAÏVE algorithm. And EDSM-BV algorithm performs better when using real data.

7.2. Future Work

The development and further plan of my project will be introduced following. I want to divide this section into two parts. The first one is about the development of my implementation and the other one is about the development of the algorithm.

Firstly, I will use larger real data to test EDSM and EDSM-BV algorithm and record the results. The biggest file used in my test is 60MB. Comparing with the level of GB, there exists an enormous gap. Secondly, I will try to decrease the runtime by changing the detail in implementation. From the graphs of real data, it is found that the algorithm needs a long time to run a large data. Therefore, the efficiency should be improved. Thirdly, I will try to change my implementation to decrease the use of memory space. For the large real data, my implementation may not run successfully when the memory space of a system is small. This disadvantage should be improved.

The above paragraph is about the development of the implementation, and I will introduce the something about updating the algorithm below. Multiple pattern matching (find all occurrences of a set of patterns in a fixed text) is as important as the single pattern matching in string matching problems. Both of EDSM and EDSM-BV are the algorithms for the single pattern matching in Elastic-Degenerate String. Therefore, the development is considering about how to update the EDSM and EDSM-BV into the algorithm that can deal with multiple pattern matching in Elastic-Degenerate String.

REFERENCE

[1] Huang, L., Popic, V. and Batzoglou, S., 2013. Short read alignment with populations of genomes. Bioinformatics, 29(13), pp.i361-i370.

[2] Iliopoulos, C.S., Kundu, R. and Pissis, S.P., 2017, March. Efficient pattern matching in elastic- degenerate texts. In International Conference on Language and Automata Theory and Applications (pp. 131-142). Springer, Cham.

[3] Holub, J., Smyth, W.F. and Wang, S., 2008. Fast pattern-matching on indeterminate strings. Journal of Discrete Algorithms, 6(1), pp.37-50.

[4] Grossi, R., Iliopoulos, C., Liu, C., Pisanti, N., Pissis, S., Retha, A., Rosone, G., Vayani, F. and Versari, L., 2017. On-line pattern matching on similar texts. In 28th Annual Symposium on Combinatorial Pattern Matching (CPM'17) (pp. 7-8).

[5] Crochemore, M., Hancart, C. and Lecroq, T., 2007. Algorithms on strings. Cambridge University Press.

[6] Karp, R.M., 1992, August. On-line algorithms versus off-line algorithms: How much is it worth to know the future?. In IFIP Congress (1) (Vol. 12, pp. 416-429).

[7] Sahni, S., 2005. Data structures, algorithms, and applications in C++. 2nd ed. Summit, N.J.: Silicon Press.

[8] Cplusplus.com. (2017). Reference - C++ Reference. [online] Available at: http://www.cplusplus.com/reference/ [Accessed 21 Apr. 2017].

[9] Xlinux.nist.gov., 2017. Dictionary of Algorithms and Data Structures. [online] Available at: https://xlinux.nist.gov/dads/ [Accessed 21 Apr. 2017].

[10] Knuth, D.E., Morris, Jr, J.H. and Pratt, V.R., 1977. Fast pattern matching in strings. SIAM journal on computing, 6(2), pp.323-350.

[11] 1000 Genomes Project Consortium, 2015. A global reference for human genetic variation. Nature, 526(7571), p.68.

[12] Msdn.microsoft.com. (2017). Constructors (C++). [online] Available at: https://msdn.microsoft.com/en-GB/library/s16xw1a8.aspx [Accessed 23 Aug. 2017].

APPENDIX A: TIME PLAN

	Task Name	Start	Finish	Duration
1	Choose Project Topic	Mon 16/11/28	Mon 17/1/23	41 days
2	Read Materials	Tue 17/1/24	Sat 17/3/25	45 days
3	Learn Realtived Knowledge	Tue 17/1/24	Sat 17/3/25	45 days
4	Implement EDSM Algorithm	Sun 17/3/26	Fri 17/5/5	31 days
5	Preliminary Project Report	Thu 17/4/20	Fri 17/4/28	7 days
6	Implement EDSM-BV Algorithm	Sat 17/5/6	Tue 17/6/20	33 days
7	First Project Progress Report	Sun 17/6/18	Fri 17/6/23	6 days
8	Debug Both Algorithm	Wed 17/6/21	Sat 17/7/1	9 days
9	Implement the Naïve algorithm	Sun 17/7/2	Thu 17/7/20	15 days
10	Compare the Efficiency in Synthetic Data	Thu 17/7/20	Fri 17/7/28	7 days
11	Second Project Progress Report	Sun 17/7/23	Fri 17/7/28	6 days
12	Compare the Efficiency in Real Data	Sat 17/7/29	Mon 17/8/7	7 days
13	Final Reprot	Tue 17/8/8	Fri 17/8/25	14 days
14	Presentation	Fri 17/8/25	Thu 17/8/31	5 days
15	Total	Mon 16/11/28	Thu 17/8/31	199 days

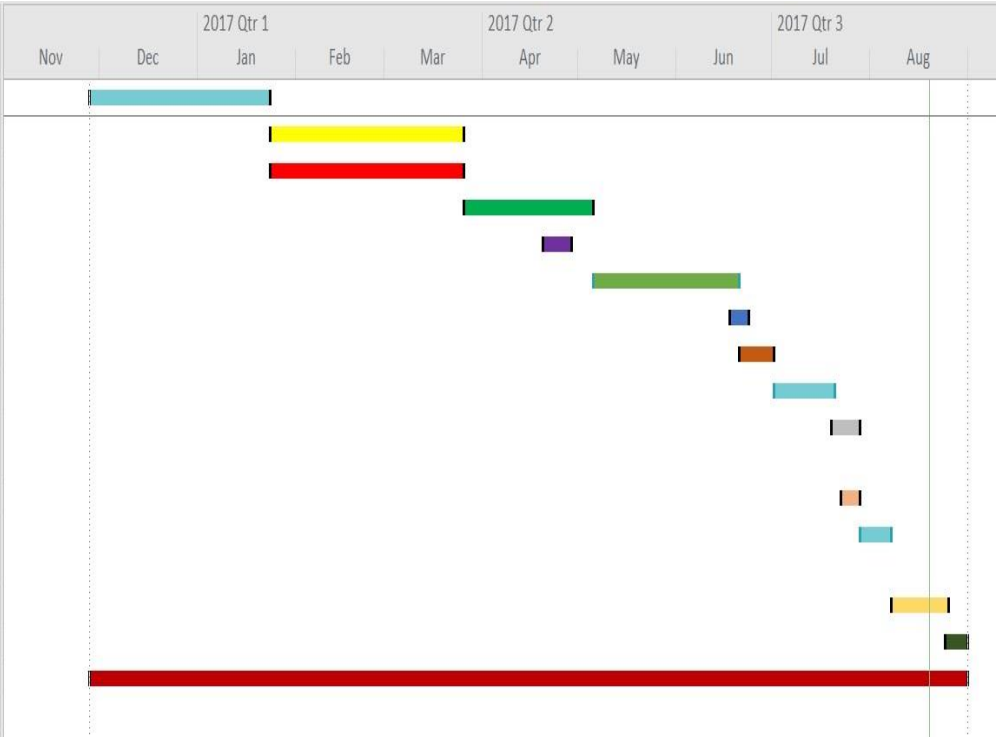


Figure 18 The Gantt chart of time plan

APPENDIX B: THE PSEUDO CODE OF EDSM ALGORITHM

```
1  Algorithm EDSM(P, m, T̃, n)
2      Construct 𝒮𝒯_P;
3      ℒ_0 ← EmptyList();
4      foreach S ∈ T̃[0] do
5          Compute ℬ_{P,S} using the border table; Insert(ℬ_{P,S}, ℒ_0);
6          if |S| ≥ m then
7              Search P in S using KMP and
8              report 0 if P occurs in S and CheckDuplicate(0);
9      foreach i ∈ [1, n − 1] do
10         ℒ_i ← EmptyList();
11         foreach S ∈ T̃[i] do
12             Compute ℬ_{P,S} using the border table; Insert(ℬ_{P,S}, ℒ_i);
13             if |S| < m then
14                 Search S in P using 𝒮𝒯_P; denote starting positions by 𝒜;
15                 foreach (p ∈ ℒ_{i−1}, j ∈ 𝒜) such that p + 1 = j do
16                     Insert({p + |S|}, ℒ_i);
17             if |S| ≥ m then
18                 Search P in S using KMP and
19                 report i if P occurs in S and CheckDuplicate(i);
20             Compute ℬ_{S,P} using 𝒮𝒯_P;
21             if there exists (p ∈ ℒ_{i−1}, j ∈ ℬ_{S,P}) such that p + j + 2 = m then
22                 Report i if CheckDuplicate(i);
```

Figure 19 The pseudo code of EDSM

APPENDIX C: THE PSEUDO CODE OF EDSM-BV ALGORITHM

```
1  Algorithm EDSM-BV(P, m, T̃, n, Σ)
2      Construct I_c, for all c ∈ Σ, and Occ-Vector_P;
3      B[0..m − 1] ← 0;
4      foreach S ∈ T̃[0] do
5          Compute B_{P,S} using the border table;
6          foreach b ∈ B_{P,S} do
7              B[b] = 1;
8          if |S| ≥ m then
9              Search P in S using KMP and
10             report 0 if P occurs in S and CheckDuplicate(0);
11     foreach i ∈ [1, n − 1] do
12         B_1[0..m − 1] ← 0;
13         foreach S ∈ T̃[i] do
14             Compute B_{P,S} using the border table;
15             foreach b ∈ B_{P,S} do
16                 B_1[b] = 1;
17             if |S| < m then
18                 B_2 ← B & Occ-Vector_P(S);
19                 B_1 ← B_1 | (B_2 ≫ |S|);
20             if |S| ≥ m then
21                 Search P in S using KMP and
22                 report i if P occurs in S and CheckDuplicate(i);
23             B_3 ← B;
24             foreach j ∈ [0, min{|S|, m − 1} − 1] do
25                 B_3 ← B_3 & I_{S[j]};
26                 B_3 ← B_3 ≫ 1;
27                 if B_3[m − 1] = 1 then
28                     Report i if CheckDuplicate(i);
29         B ← B_1;
```

Figure 20 The pseudo code of EDSM-BV

APPENDIX D: KMP.h

```
#include <string>
#include <iostream>
#include <list>
#include <vector>

using namespace std;

class KMP{
public:
	KMP();
	int algorithm(string pattern,string text,int n,int m);
	int* MP_N(string pattern,int m);
	int* KMP_N(string pattern,int m);
	int* bordertable(string pattern,vector<string> text);
};

/**
* @constructor
*/
KMP::KMP(){}

/**
* KMP Algorithm
*
* @param pattern The pattern string
* @param text The Elastic-Degenerate text
* @param n The length of text
* @param m The length of pattern
* @return -1 if the pattern is not found or the index of last character of pattern found in text
**/
int KMP::algorithm(string pattern,string text,int n,int m){
	int* KMP_next = KMP_N(pattern,m);
	int i = 0;
	int j = 0;
	while(j < n){
		while((i == m) || ((i >= 0) && (pattern[i] != text[j]))){
			i = KMP_next[i];
		}
		i++;
		j++;
		if(i == m){return j-1;}
	}
```

```
        return -1;
}

/**
* Create the border table
*
* @param pattern The pattern string
* @param array One segment of Elastic-Degenerate text(T[i])
* @return an integer array of border table
**/
int* KMP::bordertable(string pattern,vector<string> array){
        string comb;
        comb = comb + pattern;
        //create the index of border table
        for(int i = 0; i < array.size(); i++){
                if(array[i] == "E"){continue;}
                else{
                        comb = comb + "$";
                        comb = comb + array[i];
                }
        }
        //calculate the border of the index
        int* temp = MP_N(comb,comb.length());
        return temp;
}
/**
* Compute the border
*
* @param pattern The pattern string
* @param m The lenght of pattern
* @return the integer array whihc store the border table of pattern
**/
int* KMP::MP_N(string pattern,int m){
        int* border = new int[m+1];
                border[0]=-1;
                for(int i = 0; i < m; i++){
                        int j = border[i];
                        while((j >= 0) && (pattern[i] != pattern[j])){
                                j = border[j];
                        }
                        border[i+1] = j + 1;
                }
        return border;
}
```

```
/**
* Compute the KMP_next array
*
* @param pattern The pattern string
* @param m The length of pattern
* @return KMP_next array
**/
int* KMP::KMP_N(string pattern,int m){
	//compute the border
	int* MP_next = MP_N(pattern,m);
	//compute the KMP_next depend on border
	int* KMP_next = new int[m+1];
	KMP_next[0] = -1;
	int k = 0;
	for(int i = 0; i < m; i++){
		if(pattern[i] == pattern[MP_next[i]]){KMP_next[i] = KMP_next[MP_next[i]];}
		else{
			KMP_next[i] = MP_next[i];
			do{MP_next[i] = KMP_next[MP_next[i]];}while((MP_next[i] >= 0) && (pattern[i] !=
pattern[MP_next[i]]));
		}
		MP_next[i] = MP_next[i] + 1;
	}
	KMP_next[m] = MP_next[m];
	return KMP_next;
}
```

APPENDIX E: EDSM.cpp

```
#include "KMP.h"
#include <string>
#include <iostream>
#include <sdsl/suffix_trees.hpp>
#include <sdsl/suffix_arrays.hpp>
#include <vector>
#include <stack>
#include <fstream>
#include <ctime>
#include <list>
#include <stdio.h>

using namespace std;
using namespace sdsl;

//global varialbe for suffix tree
static sdsl::cst_sct3<>    cst;

//read the text into program
vector<vector<string>> ReadText(const char *);

//read the pattern intp program
string ReadPattern(string);

//compute the Bsp
list<int> ComputeBsp(string);

//search s in P
list<int> SerachSinP(string);

//EDSM algorithm
int EDSM(string,int,vector< vector<string> >,int);

//main
int main(){
	string path = "DATA/text.txt";
	string path2 = "DATA/text_empty.txt";
	const char *p = path.c_str();
	const char *p2 = path2.c_str();
	vector<vector<string>> T = ReadText(p);
	string P = ReadPattern("DATA/pattern.txt");
	cout<<"EDSM algorithm:"<<endl;
```

```
	cout<<"The normal text:"<<endl;
	int size = EDSM(P,P.length(),T,T.size());
	cout<<"The number of position is:"<<size<<endl;
	vector<vector<string>> T2 = ReadText(p2);
	string P2 = ReadPattern("DATA/pattern_empty.txt");
	cout<<endl;
	cout<<"The text with empty string"<<endl;
	int size2 = EDSM(P2,P2.length(),T2,T2.size());
	cout<<"The number of position is:"<<size2<<endl;
	return 0;
}

/**
* Read the text into program and store it into a vector
*
* @param path The path of the txt file which store the Elastic-Degenerate text
* @return a vector store the text
**/
vector<vector<string>> ReadText(const char *path){
	vector<vector<string>> T;
	FILE * pFile;
	int c;
	pFile = fopen(path,"r");
	if (pFile == NULL) perror ("Error opening file");
	else{
		//check whether meet a "{}"
		bool start = false;
		vector<string> tempvector;
		string tempstring;
		while((c = getc(pFile)) != EOF){
			//meet '{'
			if(c == '{'){
				start = true;
				tempvector.clear();
				tempstring.clear();
			}
			//meet ','
			else if(c == ','){
				tempvector.push_back(tempstring);
				tempstring.clear();
			}
			//meet '}'
			else if(c == '}'){
				start = false;
```

```
                    tempvector.push_back(tempstring);
                    T.push_back(tempvector);
                    tempvector.clear();
                }
                //otherwise
                else{
                    if(start){
                        tempstring = tempstring + (char)c;
                    }
                    else{
                        tempvector.clear();
                        tempstring = "S";
                        tempstring[0] = (char)c;
                        tempvector.push_back(tempstring);
                        T.push_back(tempvector);
                    }
                }
            }
        }
        fclose(pFile);
        return T;
}

/**
* Read the pattern into program
*
* @param path The path of the txt file that store the pattern string
* @return string The pattern
**/
string ReadPattern(string path){
        string P;
        ifstream in(path);
        getline(in,P);
        return P;
}

/**
* Compute the Bsp depend on the suffix tree of P
*
* @param s The string represented by T[i][j]
* @return an integer list that store all the Bsp value
**/
list<int> ComputeBsp(string s){
        list<int> Bsp;
```

```
    auto v = cst.child(cst.root(), s[0]);
    //check whether node v exist
    if(cst.size(v) != cst.size(cst.root())){
        string a = extract(cst, v);
        int len = a.length();
        int mark = 0;
        //check whehter two factor is same
        for(int i = 0; (i < len) && (a[i] != '\0'); i++){
            if(a[i] != s[i]){mark = 1;break;}
        }
        auto tempnode = v;
        if(mark == 0){
            //check whether it is a left
            if(cst.is_leaf(tempnode)){Bsp.push_back(len-2);}
            else{
                string temps;
                //repeat check whether match
                for(int j = len; j < s.length(); j++){
                    if(j == len){
                        auto child = cst.child(tempnode,'\0');
                        if((cst.size(child) != cst.size(cst.root())) || cst.is_leaf(tempnode)){
                            if(cst.is_leaf(tempnode)){
                                Bsp.push_back(len-2);break;
                            }
                            else{
                                Bsp.push_back(len-1);
                            }
                        }
                        tempnode = cst.child(tempnode,s[j]);
                        int tempnum = cst.size(tempnode);
                        if(tempnum == cst.size(cst.root())){mark = 1;break;}
                        temps = extract(cst, tempnode);
                        len = temps.length();
                    }
                    if((temps[j] !=s [j]) && (temps[j] != '\0')){mark = 1;break;}
                }
                //check after the loop finished
                auto child2 = cst.child(tempnode,'\0');
                if(cst.is_leaf(tempnode) && (len-2 < s.length()) && (mark == 0)){Bsp.push_back(len-2);}
                if((cst.size(child2)!= cst.size(cst.root())) && (len-1 < s.length()) && (mark ==
0)){Bsp.push_back(len-1);}
            }
        }
    }
```

```
	return Bsp;
}

/**
* Find all start positions of s(T[i][j]) in pattern P
*
* @param s The string represented by T[i][j]
* @return an integer array which contians all start positions
**/
list<int> SerachSinP(string s){
	list<int> ret;
	//strat from the root
	auto v = cst.child(cst.root(), s[0]);
	if(cst.size(v) != cst.size(cst.root())){
		string a = extract(cst, v);
		int len = a.length();
		int mark = 0;
		for(int ii = 0; (ii < len) && (a[ii] != '\0') && (ii < s.length()); ii++){
			if(a[ii] != s[ii]){mark = 1;break;}
		}
		auto tempnode = v;
		if(mark == 0){
			string temps;
			//repeat check the matching until the end of string
			for(int jj = len; jj < s.length(); jj++){
				if(jj == len){
					tempnode = cst.child(tempnode,s[jj]);
					int tempnum = cst.size(tempnode);
					if(tempnum == cst.size(cst.root())){mark = 1;break;}
					temps = extract(cst, tempnode);
					len = temps.length();
				}
				if(temps[jj] != s[jj]){mark = 1;break;}
			}
		}
		//if matching,store all relative value into list
		if(mark == 0){
			int left = cst.lb(tempnode);
			int right = cst.rb(tempnode);
			for(int kk = left; kk < right+1; kk++){
				ret.push_back(cst.csa[kk]);
			}
		}
	}
```

```
	return ret;
}

/**
* EDSM algorithm
*
* @param P The pattern string
* @param m The length of P
* @param T The Elastic-Degenerate text
* @param n The length of The
* @return the number of occurence
**/
int EDSM(string P,int m,vector< vector<string> > T,int n){
	//create the sufiix tree(preprocess)
	construct_im(cst,P,1);
	//start to record the time
	clock_t start,finish;
	double totalTime;
	start=clock();
	//initial part
	KMP k;
	stack<int> out;
	list<int> L0;
	//calculate the border table
	int* a0=k.bordertable(P,T[0]);
	int temp0=m;
	//process first segment T[0]
	for(int i0 = 0; (i0 < T[0].size()) && (T[0][i0] != ""); i0++){
		temp0 = temp0 + T[0][i0].length();
		int temp = temp0;
		while(a0[temp + 1] != 0){
			//add Bps to list
			L0.push_back(a0[temp + 1] - 1);
			temp = a0[temp + 1] - 1;
		}
		if(T[0][i0].length() >= m){
			//KMP serach
			int x = k.algorithm(P,T[0][i0],T[0][i0].length(),m);
			//check whether repeat and output
			if(x != -1){
				if((!out.empty() && (out.top() != 0))||(out.empty())){
					out.push(0);
					cout<<"0"<<endl;
				}
```

```
            }
        }
        //jump the mark symbol
        temp0++;
}
list<int> previous;
previous = L0;

//loop from T[1] to T[n - 1]
for(int i = 1; i < n; i++){
        list<int> L;
        int tempi = m;
        bool add = false;
        //compute border table
        int* ai = k.bordertable(P,T[i]);
        for(int j = 0; j < T[i].size(); j++){
                if(T[i][j] == "E"){add = true;}
                else{
                        tempi = tempi + T[i][j].length();
                        int temp = tempi;
                        while(ai[temp + 1] != 0){
                                //add Bps to list
                                L.push_back(ai[temp + 1] - 1);
                                temp = ai[temp + 1 ] - 1;
                        }
                        if(T[i][j].length() < m){
                                //find all start positions
                                list<int> ret;
                                ret = SerachSinP(T[i][j]);
                                //check whether can extend
                                if(!previous.empty() && !ret.empty()){
                                        list<int>::iterator pre_ite;
                                        list<int>::iterator A_ite;
                                        for(pre_ite = previous.begin(); pre_ite != previous.end(); ++pre_ite){
                                                int pre_num = *pre_ite;
                                                for(A_ite = ret.begin(); A_ite != ret.end(); ++A_ite){
                                                        int A_num = *A_ite;
                                                        if(pre_num + 1 == A_num){L.push_back(pre_num + T[i][j].length());}
                                                }
                                        }
                                }
                        }
                        if(T[i][j].length() >= m){
                                //KMP serach
```

```
                    int x = k.algorithm(P,T[i][j],T[i][j].length(),m);
                    //check repeat and output
                    if(x != -1){
                        if((!out.empty() && (out.top() != i))||(out.empty())){
                        out.push(i);
                        cout<<i<<endl;
                        }
                    }
                }
                //compute Bsp
                list<int> Bsp;
                Bsp=ComputeBsp(T[i][j]);
                //check whether exist full occurence
                if(!previous.empty() && !Bsp.empty()){
                    list<int>::iterator pre_ite2;
                    list<int>::iterator Bsp_ite;
                    for(pre_ite2 = previous.begin(); pre_ite2 != previous.end(); ++pre_ite2){
                        int pre_num = *pre_ite2;
                        for(Bsp_ite = Bsp.begin(); Bsp_ite != Bsp.end(); ++Bsp_ite){
                            int Bsp_num = *Bsp_ite;
                            if(pre_num + Bsp_num + 2 == m){
                                if((!out.empty() && (out.top() != i))||(out.empty())){
                                    out.push(i);
                                    cout<<i<<endl;
                                }
                            }
                        }
                    }
                }
                //jump the mark symbol
                tempi++;
            }
        }
        if(add){L.merge(previous);}//extend when empty exist
        previous = L;
    }
    //calculate the run time
    finish = clock();
    totalTime = (double)(finish-start) / (double)CLOCKS_PER_SEC;
    cout<<"Total time of this algorithm is "<<totalTime<<" second !  "<<endl;
    return out.size();
}
```

APPENDIX F: EDSM_BV.cpp

```
#include "KMP.h"
#include <string>
#include <iostream>
#include <sdsl/bit_vectors.hpp>
#include <sdsl/suffix_trees.hpp>
#include <sdsl/suffix_arrays.hpp>
#include <vector>
#include <map>
#include <algorithm>
#include <stack>
#include <fstream>
#include <ctime>
#include <list>
#include <stdio.h>

using namespace std;
using namespace sdsl;

//global variables for suffix tree
static sdsl::cst_sct3<>    cst;

//read the text into program
vector<vector<string>> ReadText(const char *);

//read the pattern into program
string ReadPattern(string);

//get the alphabet depend on the text
string ReadAlphabet(vector<vector<string>>);

// create the bit-vector Ic
map<char,bit_vector> CreateIc(string,string,int);

//create the augmented suffix tree
map<string,bit_vector> OccVector(cst_sct3<>,int);

//get relative bit-vector depend on the string
string findOccVectorS(string);

//EDSM-BV algorithm
int EDSM_BV(string,int,vector< vector<string> >,int,string);
```

```
//main
int main(){
	string path = "DATA/text.txt";
	string path2 = "DATA/text_empty.txt";
	const char *p = path.c_str();
	const char *p2 = path2.c_str();
	vector<vector<string>> T = ReadText(p);
	string P = ReadPattern("DATA/pattern.txt");
	string alphabet = ReadAlphabet(T);
	cout<<"EDSM-BV algorithm:"<<endl;
	cout<<"The normal text:"<<endl;
	int size = EDSM_BV(P,P.length(),T,T.size(),alphabet);
	cout<<"The number of position is:"<<size<<endl;
	vector<vector<string>> T2 = ReadText(p2);
	string P2 = ReadPattern("DATA/pattern_empty.txt");
	string alphabet2 = ReadAlphabet(T2);
	cout<<endl;
	cout<<"The text with empty string"<<endl;
	int size2 = EDSM_BV(P2,P2.length(),T2,T2.size(),alphabet2);
	cout<<"The number of position is:"<<size2<<endl;
	return 0;
}

/**
* Read the text into program and store it into a vector
*
* @param path The path of the txt file which store the Elastic-Degenerate text
* @return a vector store the text
**/
vector<vector<string>> ReadText(const char *path){
	vector<vector<string>> T;
	FILE * pFile;
	int c;
	pFile = fopen(path,"r");
	if (pFile == NULL) perror ("Error opening file");
	else{
		//check whether meet a "{}"
		bool start = false;
		vector<string> tempvector;
		string tempstring;
		while((c = getc(pFile)) != EOF){
			//meet '{'
			if(c == '{'){
				start = true;
```

```
                tempvector.clear();
                tempstring.clear();
            }
            //meet ','
            else if(c == ','){
                tempvector.push_back(tempstring);
                tempstring.clear();
            }
            //meet '}'
            else if(c == '}'){
                start = false;
                tempvector.push_back(tempstring);
                T.push_back(tempvector);
                tempvector.clear();
            }
            //otherwise
            else{
                if(start){
                    tempstring = tempstring + (char)c;
                }
                else{
                    tempvector.clear();
                    tempstring = "S";
                    tempstring[0] = (char)c;
                    tempvector.push_back(tempstring);
                    T.push_back(tempvector);
                }
            }
        }
    }
    fclose(pFile);
    return T;
}

/**
* Read the pattern into program
*
* @param path The path of the txt file that store the pattern string
* @return string The pattern
**/
string ReadPattern(string path){
    string P;
    ifstream in(path);
    getline(in,P);
```

```
	return P;
}

/**
* Get the alphabet depend on the text
*
* @param T The Elastic-Degenerate text stored in vector
* @return a string that is the alphabet of text
**/
string ReadAlphabet(vector<vector<string>> T){
	string all;
	vector<vector<string>>::iterator it;
	//read each letter
	for (it = T.begin(); it != T.end(); ++it){
		vector<string> temp = *it;
		for(int i = 0;i < temp.size(); i++){
			string temps = temp[i];
			if(temps == "E"){continue;}
			else{
				for(int j = 0;j < temps.length(); j++){
					bool repeat = false;
					if(all.empty()){all = all + temps[j];}
					else{
						for(int k = 0; k < all.length(); k++){
							if(all[k] == temps[j]){repeat = true;}
						}
						if(!repeat){all = all + temps[j];}
					}
				}
			}
		}
	}
	return all;
}

/**
* Create a map that contain bit-vectos Ic for each letter in alphabet
*
* @param P The pattern string
* @param all The string of all letters in alphabet
* @param m The lenght of pattern string
* @return a map the key is the letter and the value is the Ic for relative letter
**/
map<char,bit_vector> CreateIc(string P,string all,int m){
```

```
	map<char,bit_vector> Ic;
	for(int i = 0; i < all.length(); i++){
		bit_vector temp = bit_vector(m,0);
		Ic.insert(pair<char,bit_vector>(all[i],temp));
	}
	for(int j = 1; j < m; j++){
		bit_vector tempb = Ic[P[j]];
		Ic.erase(P[j]);
		tempb[j - 1] = 1;
		Ic.insert(pair<char,bit_vector>(P[j],tempb));
	}
return Ic;
}

/**
* Create the augmented suffix tree
*
* @param suffixtree The structure of suffix tree
* @param m The length of pattern
* @return a map the index is the string the represtned by the node of suffix tree and the value is relative
bit-vector
**/
map<string,bit_vector> OccVector(cst_sct3<> suffixtree,int m){
	map<string,bit_vector> OccV;
	cst_sct3<>::const_bottom_up_iterator it;
	for(it = suffixtree.begin_bottom_up(); it != suffixtree.end_bottom_up(); ++it){
	auto v = *it;
	bit_vector tempVec = bit_vector(m,0);
		if(v != suffixtree.root()){
			string temp = extract(suffixtree,v);
			if(suffixtree.is_leaf(v)){
				if((temp.length() != (m + 1)) && (temp.length() != 1)){
					tempVec[suffixtree.csa[suffixtree.id(v)] - 1] = 1;
				}
			}
			else{
				int up = suffixtree.degree(v);
				for(int i = 1;i <= up; i++){
					auto tempnode = suffixtree.select_child(v,i);
					string index = extract(suffixtree,tempnode);
					bit_vector indexvector = OccV[index];
					int temporig = tempVec.get_int(0,m);
					int tempnew = indexvector.get_int(0,m);
					int tempfinal = temporig | tempnew;
```

```
                    tempVec.set_int(0,tempfinal,m);
                }
            }
            OccV.insert(pair<string,bit_vector>(temp,tempVec));
        }
        else{
            int up = suffixtree.degree(v);
                for(int i=1;i<=up;i++){
                    auto tempnode = suffixtree.select_child(v,i);
                    string index = extract(suffixtree,tempnode);
                    bit_vector indexvector = OccV[index];
                    int temporig = tempVec.get_int(0,m);
                    int tempnew = indexvector.get_int(0,m);
                    int tempfinal = temporig | tempnew;
                    tempVec.set_int(0,tempfinal,m);
                }
            OccV.insert(pair<string,bit_vector>(" ",tempVec));
        }
    }
    return OccV;
}

/**
* Get the string represtned by the node in suffix tree depend on the T[i][j]
*
* @param S The string T[i][j]
* @return a string represtned by the relative node
**/
string findOccVectorS(string S){
    string index;
    auto v = cst.child(cst.root(), S[0]);
    if(cst.size(v) != cst.size(cst.root())){
        string a = extract(cst, v);
        int len = a.length();
        int mark = 0;
        for(int i = 0; (i < len) && (a[i] != '\0') && (i < S.length()); i++){
            if(a[i] != S[i]){mark = 1;break;}
        }
        auto tempnode = v;
        if(mark == 0){
            string temps;
            for(int j = len; j < S.length(); j++){
                if(j == len){
                    tempnode = cst.child(tempnode,S[j]);
```

```
                        int tempnum = cst.size(tempnode);
                        if(tempnum == cst.size(cst.root())){mark = 1;break;}
                        temps = extract(cst, tempnode);
                        len = temps.length();
                    }
                    if(temps[j] != S[j]){mark = 1;break;}
                }
            }
            if(mark == 0){
                index = extract(cst,tempnode);
            }
        }
        return index;
}

/**
* EDSM-BV Algorithm
*
* @param P The pattern string
* @param m The length of pattern string
* @param T The Elastic-Degenerate text
* @param n The length of Elastic-Degenerate text
* @param all The string of all letters in alphabet
* @return the number of occurrence
**/
int EDSM_BV(string P,int m,vector< vector<string> > T,int n,string all){
        //Preprocess
        //create Ic for all character
        map<char,bit_vector> Ic = CreateIc(P,all,m);
        //create the suffix tree
        construct_im(cst,P,1);
        //update into augmented suffix tree
        map<string,bit_vector> occvector = OccVector(cst,m);
        //strat to record time
        clock_t start,finish;
        double totalTime;
        start = clock();
        //initial part
        KMP k;
        stack<int> out;
        bit_vector vecB = bit_vector(m,0);
        //calcualte the border table
        int* a0 = k.bordertable(P,T[0]);
        int temp0 = m;
```

```
//process first segment T[0]
for(int i0 = 0; (i0 < T[0].size()) && (T[0][i0] != ""); i0++){
	temp0 = temp0 + T[0][i0].length();
	int temp = temp0;
	while(a0[temp + 1] != 0){
		//add Bps to bit-vector
		vecB[a0[temp + 1] - 1] = 1;
		temp = a0[temp + 1] - 1;
	}
	if(T[0][i0].length() >= m){
		//KMP
		int x = k.algorithm(P,T[0][i0],T[0][i0].length(),m);
		//check repeat and output
		if(x != -1){
			if((!out.empty() && (out.top() != 0)) || (out.empty())){
				out.push(0);
				cout<<"0"<<endl;
			}
		}
	}
	//jump the mark symbol
	temp0++;
}
int B = vecB.get_int(0,m);

//loop from T[1] to T[n - 1]
for(int i = 1; i < n; i++){
	int result;
	bit_vector B1 = bit_vector(m,0);
	int tempi = m;
	bool add = false;
	//compute border table
	int* ai = k.bordertable(P,T[i]);
	for(int j = 0; j < T[i].size(); j++){
		if(T[i][j] == "E"){
			add=true;
		}
		else{
		tempi = tempi + T[i][j].length();
		int temp = tempi;
		while(ai[temp + 1] != 0){
			//add Bps to bit-vector
			B1[ai[temp + 1] - 1] = 1;
			temp = ai[temp + 1] - 1;
```

```
}
if(T[i][j].length() < m){
	//get the bit-vector relative to T[i][j]
	bit_vector occ_vec = bit_vector(m,0);
	string index = findOccVectorS(T[i][j]);
	occ_vec = occvector[index];
	//AND operation and shift
	int B2 = 0;
	int tempocc = occ_vec.get_int(0,m);
	B2 = B & tempocc;
	B2 = B2 << T[i][j].length();
	//OR operation and add to B1
	int tempB1 = B1.get_int(0,m);
	tempB1 = B2 | tempB1;
	B1.set_int(0,tempB1,m);
}
if(T[i][j].length() >= m){
	//KMP
	int x = k.algorithm(P,T[i][j],T[i][j].length(),m);
	//check repeat and output
	if(x != -1){
		if((!out.empty() && (out.top() != i)) || (out.empty())){
		out.push(i);
		cout<<i<<endl;
		}
	}
}
//check whether there exist full occurence
int B3 = 0;
B3 = B;
int len = T[i][j].length();
int upbound = min(len,(m-1));
for(int l = 0; l < upbound; l++){
	bit_vector IcTemp = Ic[T[i][j][l]];
	int tempIc = IcTemp.get_int(0,m);
	B3 = B3 & tempIc;
	B3 = B3 << 1;
	bit_vector vecB3 = bit_vector(m,0);
	vecB3.set_int(0,B3,m);
	if(vecB3[m - 1] == 1){
		//check repeat and output
		if((!out.empty() && (out.top() != i)) || out.empty()){
			out.push(i);
			cout<<i<<endl;
```

```
                        }
                    }
                }
                //jump the mark symbol
                tempi++;
                }
            }
            int t2 = B1.get_int(0,m);
            //extend when empty exist
            if(add){
                result = B | t2;
                B = result;
            }
            else{B = t2;}
        }
        //calcualte the run time
        finish=clock();
        totalTime=(double)(finish-start)/(double)CLOCKS_PER_SEC;
        cout<<"Total time of this algorithm is "<<totalTime<<" second !  "<<endl;
        return out.size();
}
```

APPENDIX G: NAIVE.cpp

```
#include <string>
#include <iostream>
#include <vector>
#include <stack>
#include <fstream>
#include <ctime>
#include <list>
#include <stdio.h>

using namespace std;

//read the text into grogram
vector<vector<string>> ReadText(const char *);

//read the pattern into program
string ReadPattern(string);

//Iterative function to read next segment
list<int> loop(vector<vector<string>>::iterator,int,string,int,int);

//naive algorithm
int Naive(string,int,vector< vector<string> >,int);

int main(){
	string path = "DATA/text.txt";
	string path2 = "DATA/text_empty.txt";
	const char *p = path.c_str();
	const char *p2 = path2.c_str();
	vector<vector<string>> T = ReadText(p);
	string P = ReadPattern("DATA/pattern.txt");
	cout<<"EDSM algorithm:"<<endl;
	cout<<"The normal text:"<<endl;
	int size = Naive(P,P.length(),T,T.size());
	cout<<"The number of position is:"<<size<<endl;
	vector<vector<string>> T2 = ReadText(p2);
	string P2 = ReadPattern("DATA/pattern_empty.txt");
	cout<<endl;
	cout<<"The text with empty string"<<endl;
	int size2 = Naive(P2,P2.length(),T2,T2.size());
	cout<<"The number of position is:"<<size2<<endl;
	return 0;
}
```

```
/**
* Read the text into program and store it into a vector
*
* @param path The path of the txt file which store the Elastic-Degenerate text
* @return a vector store the text
**/
vector<vector<string>> ReadText(const char *path){
	vector<vector<string>> T;
	FILE * pFile;
	int c;
	pFile = fopen(path,"r");
	if (pFile == NULL) perror ("Error opening file");
	else{
		//check whether meet a "{}"
		bool start = false;
		vector<string> tempvector;
		string tempstring;
		while((c = getc(pFile)) != EOF){
			//meet '{'
			if(c == '{'){
				start = true;
				tempvector.clear();
				tempstring.clear();
			}
			//meet ','
			else if(c == ','){
				tempvector.push_back(tempstring);
				tempstring.clear();
			}
			//meet '}'
			else if(c == '}'){
				start = false;
				tempvector.push_back(tempstring);
				T.push_back(tempvector);
				tempvector.clear();
			}
			//otherwise
			else{
				if(start){
					tempstring = tempstring + (char)c;
				}
				else{
					tempvector.clear();
```

```
                        tempstring = "S";
                        tempstring[0] = (char)c;
                        tempvector.push_back(tempstring);
                        T.push_back(tempvector);
                    }
                }
            }
        }
        fclose(pFile);
        return T;
}

/**
* Read the pattern into program
*
* @param path The path of the txt file that store the pattern string
* @return string The pattern
**/
string ReadPattern(string path){
        string P;
        ifstream in(path);
        getline(in,P);
        return P;
}

/**
* An iterative function which is used to call next segment T[i+1]
*
* @param it The vector iterator of the text
* @param x The length of the substring of pattern that already be compared
* @param P The pattern string
* @param postion The index of the Elastic-Degenerate text. It is the i of T[i]
* @param n The length of the Elastic-Degenerate text
* @return the postion of the full occurence
**/
list<int> loop(vector<vector<string>>::iterator it,int x,string P,int position,int n){
            list<int> appear;
            bool justice;
            vector<string> temp = *it;
            for(int j    =0; j < temp.size(); j++){
                string temps = temp[j];
                justice = true;
                list<int> output;
                int k;
```

```
            //a iterator used to call next segment
            vector<vector<string>>::iterator next;
            next = it;
            //the bool value used to show that all letters in pattern is compared
            bool end = true;
            //when meet empty string call next segment
            if(temps == "E"){
                if(position + 1 != n){
                    next++;
                    output = loop(next,x,P,position + 1,n);
                    if(output.empty()){justice = false; continue;}
                }
            }
            if(temps != "E"){
                //compare whether two substring is same
                for(k = 0; (k < P.length() - x) && (k < temps.length()); k++){
                    if(P[x + k] != temps[k]){justice = false;}
                    //if not end call next segment
                    if(((k + 1) == temps.length()) && (k < (P.length() - x)) && ((k + 1) != (P.length() - x))){
                        end = false;
                        if(position + 1 != n){
                            next++;
                            output = loop(next,x + k + 1,P,position + 1,n);
                        }
                    }
                }
            }
            //check whether there is a full occurence
            if(!end && output.empty()){justice = false;}
            //add index of full occurence to the list
            if(justice){
                if((temps != "E" )&& (k <= temps.length()) && end){
                    appear.push_back(position);
                }
                if(!output.empty()){
                    output.sort();
                    appear.merge(output);
                }
            }
        }
    return appear;
}

/**
```

```
* Naive algorithm
*
* @param P The pattern string
* @param m The lenght of pattern
* @param T The Elastic-Degenerate text
* @param n The length of Elastic-Degenerate text
* @return the number of occurence
**/
int Naive(string P,int m,vector< vector<string> > T,int n){
	//set the clock to record time
	clock_t start,finish;
	double totalTime;
	start = clock();
	//initial
	bool appear;
	stack<int> out;
	vector<vector<string>>::iterator it;
	it = T.begin();
	//a loop for T[i]
	for(int index = 0; index < n; index++){
		// a loop for the T[i][j]
		for(int i = 0;i<T[index].size();i++){
			string temps = T[index][i];
			if(temps == "E"){continue;}
			else{
				//a loop for the T[i][j][k]
				for(int j = 0; j < temps.length(); j++){
					appear = true;
					vector<vector<string>>::iterator next;
					next = it;
					list<int> output;
					int k;
					bool end = true;
					//compare each letter of two factor
					for(k = 0; (k < m) && (k < temps.length()); k++){
						if(j + k < temps.length()){
							if(P[k] != temps[j + k]){appear = false;}
						}
						//if compare is not finish go to next segment
						if(((j + k + 1) == temps.length()) && (k < m) && ((k + 1) != m)){
							end = false;
							if((index + 1) != n){
								next++;
								output = loop(next,k + 1,P,index + 1,n);
```

```
            }
        }
    }
    //check whether there is a full occurence
    if(!end && output.empty()){appear = false;}
    if(appear){
        //check repeat and output
        if(((j + k) <= temps.length()) && end){
            if(out.empty()){
                out.push(index);
                cout<<index<<endl;
            }
            else{
                stack<int> temp = out;
                bool repeat = false;
                while(!temp.empty()){
                    int x = temp.top();
                    temp.pop();
                    if(x == index){repeat = true;}
                }
                if(!repeat){
                    out.push(index);
                    cout<<index<<endl;
                }
            }
        }
        else{
            if(out.empty()){
                out.push(output.front());
                cout<<output.front()<<endl;
                list<int>::iterator fo;
                fo = output.begin();
                fo++;
                for(fo; fo != output.end(); ++fo){
                    int tempint = *fo;
                    stack<int> temp = out;
                    bool repeat = false;
                    while(!temp.empty()){
                        int x=temp.top();
                        temp.pop();
                        if(x == tempint){repeat = true;}
                    }
                    if(!repeat){
                        out.push(tempint);
```

```
                                        cout<<tempint<<endl;
                                    }
                                }
                            }
                            else{
                                list<int>::iterator yy;
                                for(yy=output.begin(); yy != output.end(); ++yy){
                                    int tempint = *yy;
                                    stack<int> temp = out;
                                    bool repeat = false;
                                    while(!temp.empty()){
                                        int x = temp.top();
                                        temp.pop();
                                        if(x == tempint){repeat = true;}
                                    }
                                    if(!repeat){
                                        out.push(tempint);
                                        cout<<tempint<<endl;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
        it++;
    }
    //calculate the run time and output
    finish=clock();
    totalTime=(double)(finish-start)/(double)CLOCKS_PER_SEC;
    cout<<"Total time of this algorithm is "<<totalTime<<" second !  "<<endl;
    return out.size();
}
```

APPENDIX H: COMBINE.cpp

```
#include "KMP.h"
#include <string>
#include <iostream>
#include <sdsl/suffix_trees.hpp>
#include <sdsl/suffix_arrays.hpp>
#include <vector>
#include <map>
#include <algorithm>
#include <stack>
#include <fstream>
#include <ctime>
#include <list>
#include <stdio.h>

using namespace std;
using namespace sdsl;

//global variables for suffix tree
static sdsl::cst_sct3<>    cst;

//read the text into program
vector<vector<string>> ReadText(const char *);

//read the pattern into program
string ReadPattern(string);

//get the alphabet depend on the text
string ReadAlphabet(vector<vector<string>>);

//Iterative function to read next segment
list<int> loop(vector<vector<string>>::iterator,int,string,int,int);

//compute the Bsp
list<int> ComputeBsp(string);

//search s in P
list<int> SerachSinP(string);

// create the bit-vector Ic
map<char,bit_vector> CreateIc(string,string,int);

//create the augmented suffix tree
```

```
map<string,bit_vector> OccVector(cst_sct3<>,int);

//get relative bit-vector depend on the string
string findOccVectorS(string);

//naive algorithm
int Naive(string,int,vector< vector<string> >,int);

//EDSM algorithm
int EDSM(string,int,vector< vector<string> >,int);

//EDSM-BV algorithm
int EDSM_BV(string,int,vector< vector<string> >,int,string);

//main
int main(int argc, char *argv[]){
	if(argc != 4){cout<<"The number of input is wrong.Please run again."<<endl;return 0;}
	//first argument is path of text
	string text = argv[1];
	//the second argument is the path of pattern
	string pattern = argv[2];
	//the third argumrnt is the typr of algorithm
	string algorithm = argv[3];
	const char *p = text.c_str();
	vector<vector<string>> T = ReadText(p);
	if(T.empty()){return 0;}
	string P = ReadPattern(pattern);
	if(P.empty()){return 0;}
	if(algorithm == "NAIVE"){
		cout<<"Naive algorithm:"<<endl;
		int size = Naive(P,P.length(),T,T.size());
		cout<<"The number of position is:"<<size<<endl;
	}
	else if(algorithm == "EDSM"){
		cout<<"EDSM algorithm:"<<endl;
		int size = EDSM(P,P.length(),T,T.size());
		cout<<"The number of position is:"<<size<<endl;
	}
	else if(algorithm == "EDSM-BV"){
		string alphabet = ReadAlphabet(T);
		cout<<"EDSM-BV algorithm:"<<endl;
		int size = EDSM_BV(P,P.length(),T,T.size(),alphabet);
		cout<<"The number of position is:"<<size<<endl;
	}
```

```
	else{cout<<"The algorithm input is wrong."<<endl;}
	return 0;
}

/**
* Read the text into program and store it into a vector
*
* @param path The path of the txt file which store the Elastic-Degenerate text
* @return a vector store the text
**/
vector<vector<string>> ReadText(const char *path){
	vector<vector<string>> T;
	FILE * pFile;
	int c;
	pFile = fopen(path,"r");
	if (pFile == NULL){perror("Error opening file of text.");
		return T;
	}
	else{
		//check whether meet a "{}"
		bool start = false;
		vector<string> tempvector;
		string tempstring;
		while((c = getc(pFile)) != EOF){
			//meet '{'
			if(c == '{'){
				start = true;
				tempvector.clear();
				tempstring.clear();
			}
			//meet ','
			else if(c == ','){
				tempvector.push_back(tempstring);
				tempstring.clear();
			}
			//meet '}'
			else if(c == '}'){
				start = false;
				tempvector.push_back(tempstring);
				T.push_back(tempvector);
				tempvector.clear();
			}
			//otherwise
			else{
```

```
                    if(start){
                        tempstring = tempstring + (char)c;
                    }
                    else{
                        tempvector.clear();
                        tempstring = "S";
                        tempstring[0] = (char)c;
                        tempvector.push_back(tempstring);
                        T.push_back(tempvector);
                    }
                }
            }
        }
        fclose(pFile);
        return T;
}

/**
* Read the pattern into program
*
* @param path The path of the txt file that store the pattern string
* @return string The pattern
**/
string ReadPattern(string path){
        string P;
        ifstream in(path);
        getline(in,P);
        if(P.empty()){cout<<"Error opening file of pattern."<<endl;}
        return P;
}

/**
* Get the alphabet depend on the text
*
* @param T The Elastic-Degenerate text stored in vector
* @return a string that is the alphabet of text
**/
string ReadAlphabet(vector<vector<string>> T){
        string all;
        vector<vector<string>>::iterator it;
        //read each letter
        for (it = T.begin(); it != T.end(); ++it){
            vector<string> temp = *it;
            for(int i = 0;i < temp.size(); i++){
```

```
                string temps = temp[i];
                if(temps == "E"){continue;}
                else{
                    for(int j = 0;j < temps.length(); j++){
                        bool repeat = false;
                        if(all.empty()){all = all + temps[j];}
                        else{
                            for(int k = 0; k < all.length(); k++){
                                if(all[k] == temps[j]){repeat = true;}
                            }
                            if(!repeat){all = all + temps[j];}
                        }
                    }
                }
            }
        }
        return all;
}

/**
* An iterative function which is used to call next segment T[i+1]
*
* @param it The vector iterator of the text
* @param x The length of the substring of pattern that already be compared
* @param P The pattern string
* @param postion The index of the Elastic-Degenerate text. It is the i of T[i]
* @param n The length of the Elastic-Degenerate text
* @return the postion of the full occurence
**/
list<int> loop(vector<vector<string>>::iterator it,int x,string P,int position,int n){
            list<int> appear;
            bool justice;
            vector<string> temp = *it;
            for(int j    =0; j < temp.size(); j++){
                string temps = temp[j];
                justice = true;
                list<int> output;
                int k;
                //a iterator used to call next segment
                vector<vector<string>>::iterator next;
                next = it;
                //the bool value used to show that all letters in pattern is compared
                bool end = true;
                //when meet empty string call next segment
```

```
                if(temps == "E"){
                    if(position + 1 != n){
                        next++;
                        output = loop(next,x,P,position + 1,n);
                        if(output.empty()){justice = false; continue;}
                    }
                }
                if(temps != "E"){
                    //compare whether two substring is same
                    for(k = 0; (k < P.length() - x) && (k < temps.length()); k++){
                        if(P[x + k] != temps[k]){justice = false;}
                        //if not end call next segment
                        if(((k + 1) == temps.length()) && (k < (P.length() - x)) && ((k + 1) != (P.length() - x))){
                            end = false;
                            if(position + 1 != n){
                                next++;
                                output = loop(next,x + k + 1,P,position + 1,n);
                            }
                        }
                    }
                }
                //check whether there is a full occurence
                if(!end && output.empty()){justice = false;}
                //add index of full occurence to the list
                if(justice){
                    if((temps != "E" )&& (k <= temps.length()) && end){
                        appear.push_back(position);
                    }
                    if(!output.empty()){
                        output.sort();
                        appear.merge(output);
                    }
                }
            }
        return appear;
}

/**
* Compute the Bsp depend on the suffix tree of P
*
* @param s The string represented by T[i][j]
* @return an integer list that store all the Bsp value
**/
list<int> ComputeBsp(string s){
```

```
    list<int> Bsp;
    auto v = cst.child(cst.root(), s[0]);
    //check whether node v exist
    if(cst.size(v) != cst.size(cst.root())){
        string a = extract(cst, v);
        int len = a.length();
        int mark = 0;
        //check whehter two factor is same
        for(int i = 0; (i < len) && (a[i] != '\0'); i++){
            if(a[i] != s[i]){mark = 1;break;}
        }
        auto tempnode = v;
        if(mark == 0){
            //check whether it is a left
            if(cst.is_leaf(tempnode)){Bsp.push_back(len-2);}
            else{
                string temps;
                //repeat check whether match
                for(int j = len; j < s.length(); j++){
                    if(j == len){
                        auto child = cst.child(tempnode,'\0');
                        if((cst.size(child) != cst.size(cst.root())) || cst.is_leaf(tempnode)){
                            if(cst.is_leaf(tempnode)){
                                Bsp.push_back(len-2);break;
                            }
                            else{
                                Bsp.push_back(len-1);
                            }
                        }
                        tempnode = cst.child(tempnode,s[j]);
                        int tempnum = cst.size(tempnode);
                        if(tempnum == cst.size(cst.root())){mark = 1;break;}
                        temps = extract(cst, tempnode);
                        len = temps.length();
                    }
                    if((temps[j] !=s [j]) && (temps[j] != '\0')){mark = 1;break;}
                }
                //check after the loop finished
                auto child2 = cst.child(tempnode,'\0');
                if(cst.is_leaf(tempnode) && (len-2 < s.length()) && (mark == 0)){Bsp.push_back(len-2);}
                if((cst.size(child2)!= cst.size(cst.root())) && (len-1 < s.length()) && (mark ==
0)){Bsp.push_back(len-1);}
            }
        }
```

```
        }
        return Bsp;
}

/**
* Find all start positions of s(T[i][j]) in pattern P
*
* @param s The string represented by T[i][j]
* @return an integer array which contians all start positions
**/
list<int> SerachSinP(string s){
        list<int> ret;
        //strat from the root
        auto v = cst.child(cst.root(), s[0]);
        if(cst.size(v) != cst.size(cst.root())){
                string a = extract(cst, v);
                int len = a.length();
                int mark = 0;
                for(int ii = 0; (ii < len) && (a[ii] != '\0') && (ii < s.length()); ii++){
                        if(a[ii] != s[ii]){mark = 1;break;}
                }
                auto tempnode = v;
                if(mark == 0){
                        string temps;
                        //repeat check the matching until the end of string
                        for(int jj = len; jj < s.length(); jj++){
                                if(jj == len){
                                        tempnode = cst.child(tempnode,s[jj]);
                                        int tempnum = cst.size(tempnode);
                                        if(tempnum == cst.size(cst.root())){mark = 1;break;}
                                        temps = extract(cst, tempnode);
                                        len = temps.length();
                                }
                                if(temps[jj] != s[jj]){mark = 1;break;}
                        }
                }
                //if matching,store all relative value into list
                if(mark == 0){
                        int left = cst.lb(tempnode);
                        int right = cst.rb(tempnode);
                        for(int kk = left; kk < right+1; kk++){
                                ret.push_back(cst.csa[kk]);
                        }
                }
```

```
	}
	return ret;
}

/**
* Create a map that contain bit-vectos Ic for each letter in alphabet
*
* @param P The pattern string
* @param all The string of all letters in alphabet
* @param m The lenght of pattern string
* @return a map the key is the letter and the value is the Ic for relative letter
**/
map<char,bit_vector> CreateIc(string P,string all,int m){
	map<char,bit_vector> Ic;
	for(int i = 0; i < all.length(); i++){
		bit_vector temp = bit_vector(m,0);
		Ic.insert(pair<char,bit_vector>(all[i],temp));
	}
	for(int j = 1; j < m; j++){
		bit_vector tempb = Ic[P[j]];
		Ic.erase(P[j]);
		tempb[j - 1] = 1;
		Ic.insert(pair<char,bit_vector>(P[j],tempb));
	}
return Ic;
}

/**
* Create the augmented suffix tree
*
* @param suffixtree The structure of suffix tree
* @param m The length of pattern
* @return a map the index is the string the represtned by the node of suffix tree and the value is relative
bit-vector
**/
map<string,bit_vector> OccVector(cst_sct3<> suffixtree,int m){
	map<string,bit_vector> OccV;
	cst_sct3<>::const_bottom_up_iterator it;
	for(it = suffixtree.begin_bottom_up(); it != suffixtree.end_bottom_up(); ++it){
	auto v = *it;
	bit_vector tempVec = bit_vector(m,0);
		if(v != suffixtree.root()){
			string temp = extract(suffixtree,v);
			if(suffixtree.is_leaf(v)){
```

```
                if((temp.length() != (m + 1)) && (temp.length() != 1)){
                    tempVec[suffixtree.csa[suffixtree.id(v)] - 1] = 1;
                }
            }
            else{
                int up = suffixtree.degree(v);
                for(int i = 1;i <= up; i++){
                    auto tempnode = suffixtree.select_child(v,i);
                    string index = extract(suffixtree,tempnode);
                    bit_vector indexvector = OccV[index];
                    int temporig = tempVec.get_int(0,m);
                    int tempnew = indexvector.get_int(0,m);
                    int tempfinal = temporig | tempnew;
                    tempVec.set_int(0,tempfinal,m);
                }
            }
            OccV.insert(pair<string,bit_vector>(temp,tempVec));
        }
        else{
            int up = suffixtree.degree(v);
                for(int i=1;i<=up;i++){
                    auto tempnode = suffixtree.select_child(v,i);
                    string index = extract(suffixtree,tempnode);
                    bit_vector indexvector = OccV[index];
                    int temporig = tempVec.get_int(0,m);
                    int tempnew = indexvector.get_int(0,m);
                    int tempfinal = temporig | tempnew;
                    tempVec.set_int(0,tempfinal,m);
                }
            OccV.insert(pair<string,bit_vector>(" ",tempVec));
        }
    }
    return OccV;
}

/**
* Get the string represtned by the node in suffix tree depend on the T[i][j]
*
* @param S The string T[i][j]
* @return a string represtned by the relative node
**/
string findOccVectorS(string S){
    string index;
    auto v = cst.child(cst.root(), S[0]);
```

```
        if(cst.size(v) != cst.size(cst.root())){
            string a = extract(cst, v);
            int len = a.length();
            int mark = 0;
            for(int i = 0; (i < len) && (a[i] != '\0') && (i < S.length()); i++){
                if(a[i] != S[i]){mark = 1;break;}
            }
            auto tempnode = v;
            if(mark == 0){
                string temps;
                for(int j = len; j < S.length(); j++){
                    if(j == len){
                        tempnode = cst.child(tempnode,S[j]);
                        int tempnum = cst.size(tempnode);
                        if(tempnum == cst.size(cst.root())){mark = 1;break;}
                        temps = extract(cst, tempnode);
                        len = temps.length();
                    }
                    if(temps[j] != S[j]){mark = 1;break;}
                }
            }
            if(mark == 0){
                index = extract(cst,tempnode);
            }
        }
        return index;
}

/**
* Naive algorithm
*
* @param P The pattern string
* @param m The lenght of pattern
* @param T The Elastic-Degenerate text
* @param n The length of Elastic-Degenerate text
* @return the number of occurence
**/
int Naive(string P,int m,vector< vector<string> > T,int n){
        //set the clock to record time
        clock_t start,finish;
        double totalTime;
        start = clock();
        //initial
        bool appear;
```

```
stack<int> out;
vector<vector<string>>::iterator it;
it = T.begin();
//a loop for T[i]
for(int index = 0; index < n; index++){
    // a loop for the T[i][j]
    for(int i = 0;i<T[index].size();i++){
        string temps = T[index][i];
        if(temps == "E"){continue;}
        else{
            //a loop for the T[i][j][k]
            for(int j = 0; j < temps.length(); j++){
                appear = true;
                vector<vector<string>>::iterator next;
                next = it;
                list<int> output;
                int k;
                bool end = true;
                //compare each letter of two factor
                for(k = 0; (k < m) && (k < temps.length()); k++){
                    if(j + k < temps.length()){
                        if(P[k] != temps[j + k]){appear = false;}
                    }
                    //if compare is not finish go to next segment
                    if(((j + k + 1) == temps.length()) && (k < m) && ((k + 1) != m)){
                        end = false;
                        if((index + 1) != n){
                            next++;
                            output = loop(next,k + 1,P,index + 1,n);
                        }
                    }
                }
                //check whether there is a full occurence
                if(!end && output.empty()){appear = false;}
                if(appear){
                    //check repeat and output
                    if(((j + k) <= temps.length()) && end){
                        if(out.empty()){
                            out.push(index);
                            cout<<index<<endl;
                        }
                        else{
                            stack<int> temp = out;
                            bool repeat = false;
```

```
            while(!temp.empty()){
                int x = temp.top();
                temp.pop();
                if(x == index){repeat = true;}
            }
            if(!repeat){
                out.push(index);
                cout<<index<<endl;
            }
        }
    }
    else{
        if(out.empty()){
            out.push(output.front());
            cout<<output.front()<<endl;
            list<int>::iterator fo;
            fo = output.begin();
            fo++;
            for(fo; fo != output.end(); ++fo){
                int tempint = *fo;
                stack<int> temp = out;
                bool repeat = false;
                while(!temp.empty()){
                    int x=temp.top();
                    temp.pop();
                    if(x == tempint){repeat = true;}
                }
                if(!repeat){
                    out.push(tempint);
                    cout<<tempint<<endl;
                }
            }
        }
        else{
            list<int>::iterator yy;
            for(yy=output.begin(); yy != output.end(); ++yy){
                int tempint = *yy;
                stack<int> temp = out;
                bool repeat = false;
                while(!temp.empty()){
                    int x = temp.top();
                    temp.pop();
                    if(x == tempint){repeat = true;}
                }
```

```
                                        if(!repeat){
                                            out.push(tempint);
                                            cout<<tempint<<endl;
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
            it++;
        }
        //calculate the run time and output
        finish=clock();
        totalTime=(double)(finish-start)/(double)CLOCKS_PER_SEC;
        cout<<"Total time of this algorithm is "<<totalTime<<" second !  "<<endl;
        return out.size();
}

/**
* EDSM algorithm
*
* @param P The pattern string
* @param m The length of P
* @param T The Elastic-Degenerate text
* @param n The length of The
* @return the number of occurence
**/
int EDSM(string P,int m,vector< vector<string> > T,int n){
        //create the sufiix tree(preprocess)
        construct_im(cst,P,1);
        //start to record the time
        clock_t start,finish;
        double totalTime;
        start=clock();
        //initial part
        KMP k;
        stack<int> out;
        list<int> L0;
        //calculate the border table
        int* a0=k.bordertable(P,T[0]);
        int temp0=m;
        //process first segment T[0]
```

```
for(int i0 = 0; (i0 < T[0].size()) && (T[0][i0] != ""); i0++){
	temp0 = temp0 + T[0][i0].length();
	int temp = temp0;
	while(a0[temp + 1] != 0){
		//add Bps to list
		L0.push_back(a0[temp + 1] - 1);
		temp = a0[temp + 1] - 1;
	}
	if(T[0][i0].length() >= m){
		//KMP serach
		int x = k.algorithm(P,T[0][i0],T[0][i0].length(),m);
		//check whether repeat and output
		if(x != -1){
			if((!out.empty() && (out.top() != 0))||(out.empty())){
				out.push(0);
				cout<<"0"<<endl;
			}
		}
	}
	//jump the mark symbol
	temp0++;
}
list<int> previous;
previous = L0;

//loop from T[1] to T[n - 1]
for(int i = 1; i < n; i++){
	list<int> L;
	int tempi = m;
	bool add = false;
	//compute border table
	int* ai = k.bordertable(P,T[i]);
	for(int j = 0; j < T[i].size(); j++){
		if(T[i][j] == "E"){add = true;}
		else{
			tempi = tempi + T[i][j].length();
			int temp = tempi;
			while(ai[temp + 1] != 0){
				//add Bps to list
				L.push_back(ai[temp + 1] - 1);
				temp = ai[temp + 1 ] - 1;
			}
			if(T[i][j].length() < m){
				//find all start positions
```

```
        list<int> ret;
        ret = SerachSinP(T[i][j]);
        //check whether can extend
        if(!previous.empty() && !ret.empty()){
            list<int>::iterator pre_ite;
            list<int>::iterator A_ite;
            for(pre_ite = previous.begin(); pre_ite != previous.end(); ++pre_ite){
                int pre_num = *pre_ite;
                for(A_ite = ret.begin(); A_ite != ret.end(); ++A_ite){
                    int A_num = *A_ite;
                    if(pre_num + 1 == A_num){L.push_back(pre_num + T[i][j].length());}
                }
            }
        }
    }
    if(T[i][j].length() >= m){
        //KMP serach
        int x = k.algorithm(P,T[i][j],T[i][j].length(),m);
        //check repeat and output
        if(x != -1){
            if((!out.empty() && (out.top() != i))||(out.empty())){
            out.push(i);
            cout<<i<<endl;
            }
        }
    }
    //compute Bsp
    list<int> Bsp;
    Bsp=ComputeBsp(T[i][j]);
    //check whether exist full occurence
    if(!previous.empty() && !Bsp.empty()){
        list<int>::iterator pre_ite2;
        list<int>::iterator Bsp_ite;
        for(pre_ite2 = previous.begin(); pre_ite2 != previous.end(); ++pre_ite2){
            int pre_num = *pre_ite2;
            for(Bsp_ite = Bsp.begin(); Bsp_ite != Bsp.end(); ++Bsp_ite){
                int Bsp_num = *Bsp_ite;
                if(pre_num + Bsp_num + 2 == m){
                    if((!out.empty() && (out.top() != i))||(out.empty())){
                        out.push(i);
                        cout<<i<<endl;
                    }
                }
            }
```

```
                        }
                    }
                    //jump the mark symbol
                    tempi++;
                }
            }
            if(add){L.merge(previous);}//extend when empty exist
            previous = L;
        }
        //calculate the run time
        finish = clock();
        totalTime = (double)(finish-start) / (double)CLOCKS_PER_SEC;
        cout<<"Total time of this algorithm is "<<totalTime<<" second !  "<<endl;
        return out.size();
}


/**
* EDSM-BV Algorithm
*
* @param P The pattern string
* @param m The length of pattern string
* @param T The Elastic-Degenerate text
* @param n The length of Elastic-Degenerate text
* @param all The string of all letters in alphabet
* @return the number of occurrence
**/
int EDSM_BV(string P,int m,vector< vector<string> > T,int n,string all){
        //Preprocess
        //create Ic for all character
        map<char,bit_vector> Ic = CreateIc(P,all,m);
        //create the suffix tree
        construct_im(cst,P,1);
        //update into augmented suffix tree
        map<string,bit_vector> occvector = OccVector(cst,m);
        //strat to record time
        clock_t start,finish;
        double totalTime;
        start = clock();
        //initial part
        KMP k;
        stack<int> out;
        bit_vector vecB = bit_vector(m,0);
        //calcualte the border table
```

```
int* a0 = k.bordertable(P,T[0]);
int temp0 = m;
//process first segment T[0]
for(int i0 = 0; (i0 < T[0].size()) && (T[0][i0] != ""); i0++){
	temp0 = temp0 + T[0][i0].length();
	int temp = temp0;
	while(a0[temp + 1] != 0){
		//add Bps to bit-vector
		vecB[a0[temp + 1] - 1] = 1;
		temp = a0[temp + 1] - 1;
	}
	if(T[0][i0].length() >= m){
		//KMP
		int x = k.algorithm(P,T[0][i0],T[0][i0].length(),m);
		//check repeat and output
		if(x != -1){
			if((!out.empty() && (out.top() != 0)) || (out.empty())){
				out.push(0);
				cout<<"0"<<endl;
			}
		}
	}
	//jump the mark symbol
	temp0++;
}
int B = vecB.get_int(0,m);

//loop from T[1] to T[n - 1]
for(int i = 1; i < n; i++){
	int result;
	bit_vector B1 = bit_vector(m,0);
	int tempi = m;
	bool add = false;
	//compute border table
	int* ai = k.bordertable(P,T[i]);
	for(int j = 0; j < T[i].size(); j++){
		if(T[i][j] == "E"){
			add=true;
		}
		else{
		tempi = tempi + T[i][j].length();
		int temp = tempi;
		while(ai[temp + 1] != 0){
			//add Bps to bit-vector
```

```
        B1[ai[temp + 1] - 1] = 1;
        temp = ai[temp + 1] - 1;
    }
    if(T[i][j].length() < m){
        //get the bit-vector relative to T[i][j]
        bit_vector occ_vec = bit_vector(m,0);
        string index = findOccVectorS(T[i][j]);
        occ_vec = occvector[index];
        //AND operation and shift
        int B2 = 0;
        int tempocc = occ_vec.get_int(0,m);
        B2 = B & tempocc;
        B2 = B2 << T[i][j].length();
        //OR operation and add to B1
        int tempB1 = B1.get_int(0,m);
        tempB1 = B2 | tempB1;
        B1.set_int(0,tempB1,m);
    }
    if(T[i][j].length() >= m){
        //KMP
        int x = k.algorithm(P,T[i][j],T[i][j].length(),m);
        //check repeat and output
        if(x != -1){
            if((!out.empty() && (out.top() != i)) || (out.empty())){
            out.push(i);
            cout<<i<<endl;
            }
        }
    }
    //check whether there exist full occurence
    int B3 = 0;
    B3 = B;
    int len = T[i][j].length();
    int upbound = min(len,(m-1));
    for(int l = 0; l < upbound; l++){
        bit_vector IcTemp = Ic[T[i][j][l]];
        int tempIc = IcTemp.get_int(0,m);
        B3 = B3 & tempIc;
        B3 = B3 << 1;
        bit_vector vecB3 = bit_vector(m,0);
        vecB3.set_int(0,B3,m);
        if(vecB3[m - 1] == 1){
            //check repeat and output
            if((!out.empty() && (out.top() != i)) || out.empty()){
```

```
                        out.push(i);
                        cout<<i<<endl;
                    }
                }
            }
            //jump the mark symbol
            tempi++;
            }
        }
        int t2 = B1.get_int(0,m);
        //extend when empty exist
        if(add){
            result = B | t2;
            B = result;
        }
        else{B = t2;}
    }
    //calcualte the run time
    finish=clock();
    totalTime=(double)(finish-start)/(double)CLOCKS_PER_SEC;
    cout<<"Total time of this algorithm is "<<totalTime<<" second !  "<<endl;
    return out.size();
}
```

APPENDIX I: COMBINE_UPDATE.cpp

```
#include "KMP.h"
#include <string>
#include <iostream>
#include <sdsl/suffix_trees.hpp>
#include <sdsl/suffix_arrays.hpp>
#include <vector>
#include <map>
#include <algorithm>
#include <stack>
#include <fstream>
#include <ctime>
#include <list>
#include <stdio.h>

using namespace std;
using namespace sdsl;

//global variables for suffix tree
static sdsl::cst_sct3<>    cst;

//global varibales for the previous list
static std::list<int> previous;

//global variable for the bit-vector Ic
static std::map<char,bit_vector> Ic;

//global variable for augmented suffix tree
static std::map<string,bit_vector> occvector;

//global varibales for the stack out to store the position
static std::stack<int> out;

//global varaibales for the bit-vector B
static int B;

//global of the index of segment
static int i;

//read the text into program and call EDSM algorithm
void ReadTextforEDSM(const char *,string);

//read the text into program and call EDSM-BV algorithm
```

```
void ReadTextforEDSM_BV(const char *,string);

//read the pattern into program
string ReadPattern(string);

//compute the Bsp
list<int> ComputeBsp(string);

//search s in P
list<int> SerachSinP(string);

// create the bit-vector Ic
map<char,bit_vector> CreateIc(string,string,int);

//create the augmented suffix tree
map<string,bit_vector> OccVector(cst_sct3<>,int);

//get relative bit-vector depend on the string
string findOccVectorS(string);

//EDSM algorithm
void EDSM(string,int,vector<string>,int,bool);

//EDSM-BV algorithm
void EDSM_BV(string,int,vector<string>,int,bool);

//main
int main(int argc, char *argv[]){
	if(argc != 4){cout<<"The number of input is wrong.Please run again."<<endl;return 0;}
	//first argument is path of text
	string text = argv[1];
	//the second argument is the path of pattern
	string pattern = argv[2];
	//the third argumrnt is the typr of algorithm
	string algorithm = argv[3];
	const char *p = text.c_str();
	string patt = ReadPattern(pattern);
	if(patt.empty()){return 0;}
	if(algorithm == "EDSM"){
		ReadTextforEDSM(p,patt);
	}
	else if(algorithm == "EDSM-BV"){
		ReadTextforEDSM_BV(p,patt);
	}
```

```
        else{cout<<"The algorithm input is wrong."<<endl;}
        return 0;
}

/**
* Read the text into program and call the EDSM algorithm when one segment is finished
*
* @param path The path of the txt file which store the Elastic-Degenerate text
* @param pattern The pattern string
**/
void ReadTextforEDSM(const char *path,string pattern){
        cout<<"EDSM Algorithm:"<<endl;
        //create the sufiix tree(preprocess)
        construct_im(cst,pattern,1);
        FILE * pFile;
        int c;
        double totalTime = 0;
        bool first = true;
        i=0;
        pFile = fopen(path,"r");
        if (pFile == NULL)perror("Error opening file of text.");
        else{
                //check whether meet a "{}"
                bool start = false;
                vector<string> tempvector;
                string tempstring;
                while((c = getc(pFile)) != EOF){
                        //meet '{'
                        if(c == '{'){
                                start = true;
                                tempvector.clear();
                                tempstring.clear();
                        }
                        //meet ','
                        else if(c == ','){
                                tempvector.push_back(tempstring);
                                tempstring.clear();
                        }
                        //meet '}'
                        else if(c == '}'){
                                start = false;
                                tempvector.push_back(tempstring);
                                //set clock
                                clock_t start,finish;
```

```
                start=clock();
                //call EDSM algorithm
                EDSM(pattern,pattern.length(),tempvector,tempvector.size(),first);
                finish = clock();
                totalTime = totalTime + ((double)(finish-start) / (double)CLOCKS_PER_SEC);
                //go to next index
                i++;
                //check whether it is T[0]
                if(first){first = false;}
                tempvector.clear();
            }
            //otherwise
            else{
                if(start){
                    tempstring = tempstring + (char)c;
                }
                else{
                    tempvector.clear();
                    tempstring = "S";
                    tempstring[0] = (char)c;
                    tempvector.push_back(tempstring);
                    //set clock
                    clock_t start,finish;
                    start=clock();
                    //call EDSM algorithm
                    EDSM(pattern,pattern.length(),tempvector,tempvector.size(),first);
                    finish = clock();
                    totalTime = totalTime + ((double)(finish-start) / (double)CLOCKS_PER_SEC);
                    //go to next index
                    i++;
                    //check whether it is T[0]
                    if(first){first = false;}
                }
            }
        }
    }
    fclose(pFile);
    cout<<"Total time of this algorithm is "<<totalTime<<" second !  "<<endl;
    cout<<"The number of position is:"<<out.size()<<endl;
}

/**
* Read the text into program and call the EDSM-BV algorithm when one segment is finished
*
```

```
* @param path The path of the txt file which store the Elastic-Degenerate text
* @param pattern The pattern string
**/
void ReadTextforEDSM_BV(const char *path,string pattern){
	cout<<"EDSM-BV Algorithm:"<<endl;
	//create Ic for all character
	Ic = CreateIc(pattern,"ACGTN",pattern.length());
	//create the suffix tree
	construct_im(cst,pattern,1);
	//update into augmented suffix tree
	occvector = OccVector(cst,pattern.length());
	FILE * pFile;
	int c;
	double totalTime = 0;
	bool first = true;
	i=0;
	pFile = fopen(path,"r");
	if (pFile == NULL)perror("Error opening file of text.");
	else{
		//check whether meet a "{}"
		bool start = false;
		vector<string> tempvector;
		string tempstring;
		while((c = getc(pFile)) != EOF){
			//meet '{'
			if(c == '{'){
				start = true;
				tempvector.clear();
				tempstring.clear();
			}
			//meet ','
			else if(c == ','){
				tempvector.push_back(tempstring);
				tempstring.clear();
			}
			//meet '}'
			else if(c == '}'){
				start = false;
				tempvector.push_back(tempstring);
				//set clock
				clock_t start,finish;
				start=clock();
				//call EDSM algorithm
				EDSM_BV(pattern,pattern.length(),tempvector,tempvector.size(),first);
```

```
                finish = clock();
                totalTime = totalTime + ((double)(finish-start) / (double)CLOCKS_PER_SEC);
                //go to next index
                i++;
                //check whether it is T[0]
                if(first){first = false;}
                tempvector.clear();
            }
            //otherwise
            else{
                if(start){
                    tempstring = tempstring + (char)c;
                }
                else{
                    tempvector.clear();
                    tempstring = "S";
                    tempstring[0] = (char)c;
                    tempvector.push_back(tempstring);
                    //set clock
                    clock_t start,finish;
                    start=clock();
                    //call EDSM algorithm
                    EDSM_BV(pattern,pattern.length(),tempvector,tempvector.size(),first);
                    finish = clock();
                    totalTime = totalTime + ((double)(finish-start) / (double)CLOCKS_PER_SEC);
                    //go to next index
                    i++;
                    //check whether it is T[0]
                    if(first){first = false;}
                }
            }
        }
    }
    fclose(pFile);
    cout<<"Total time of this algorithm is "<<totalTime<<" second !  "<<endl;
    cout<<"The number of position is:"<<out.size()<<endl;
}

/**
* Read the pattern into program
*
* @param path The path of the txt file that store the pattern string
* @return string The pattern
**/
```

```
string ReadPattern(string path){
	string P;
	ifstream in(path);
	getline(in,P);
	if(P.empty()){cout<<"Error opening file of pattern."<<endl;}
	return P;
}

/**
* Compute the Bsp depend on the suffix tree of P
*
* @param s The string represented by T[i][j]
* @return an integer list that store all the Bsp value
**/
list<int> ComputeBsp(string s){
	list<int> Bsp;
	auto v = cst.child(cst.root(), s[0]);
	//check whether node v exist
	if(cst.size(v) != cst.size(cst.root())){
		string a = extract(cst, v);
		int len = a.length();
		int mark = 0;
		//check whehter two factor is same
		for(int i = 0; (i < len) && (a[i] != '\0'); i++){
			if(a[i] != s[i]){mark = 1;break;}
		}
		auto tempnode = v;
		if(mark == 0){
			//check whether it is a left
			if(cst.is_leaf(tempnode)){Bsp.push_back(len-2);}
			else{
				string temps;
				//repeat check whether match
				for(int j = len; j < s.length(); j++){
					if(j == len){
						auto child = cst.child(tempnode,'\0');
						if((cst.size(child) != cst.size(cst.root())) || cst.is_leaf(tempnode)){
							if(cst.is_leaf(tempnode)){
								Bsp.push_back(len-2);break;
							}
							else{
								Bsp.push_back(len-1);
							}
						}
```

```
                        tempnode = cst.child(tempnode,s[j]);
                        int tempnum = cst.size(tempnode);
                        if(tempnum == cst.size(cst.root())){mark = 1;break;}
                        temps = extract(cst, tempnode);
                        len = temps.length();
                    }
                    if((temps[j] !=s [j]) && (temps[j] != '\0')){mark = 1;break;}
                }
                //check after the loop finished
                auto child2 = cst.child(tempnode,'\0');
                if(cst.is_leaf(tempnode) && (len-2 < s.length()) && (mark == 0)){Bsp.push_back(len-2);}
                if((cst.size(child2)!= cst.size(cst.root())) && (len-1 < s.length()) && (mark ==
0)){Bsp.push_back(len-1);}
            }
        }
    }
    return Bsp;
}

/**
* Find all start positions of s(T[i][j]) in pattern P
*
* @param s The string represented by T[i][j]
* @return an integer array which contians all start positions
**/
list<int> SerachSinP(string s){
    list<int> ret;
    //strat from the root
    auto v = cst.child(cst.root(), s[0]);
    if(cst.size(v) != cst.size(cst.root())){
        string a = extract(cst, v);
        int len = a.length();
        int mark = 0;
        for(int ii = 0; (ii < len) && (a[ii] != '\0') && (ii < s.length()); ii++){
            if(a[ii] != s[ii]){mark = 1;break;}
        }
        auto tempnode = v;
        if(mark == 0){
            string temps;
            //repeat check the matching until the end of string
            for(int jj = len; jj < s.length(); jj++){
                if(jj == len){
                    tempnode = cst.child(tempnode,s[jj]);
                    int tempnum = cst.size(tempnode);
```

```
                        if(tempnum == cst.size(cst.root())){mark = 1;break;}
                        temps = extract(cst, tempnode);
                        len = temps.length();
                    }
                    if(temps[jj] != s[jj]){mark = 1;break;}
                }
            }
            //if matching,store all relative value into list
            if(mark == 0){
                int left = cst.lb(tempnode);
                int right = cst.rb(tempnode);
                for(int kk = left; kk < right+1; kk++){
                    ret.push_back(cst.csa[kk]);
                }
            }
        }
        return ret;
}

/**
* Create a map that contain bit-vectos Ic for each letter in alphabet
*
* @param P The pattern string
* @param all The string of all letters in alphabet
* @param m The lenght of pattern string
* @return a map the key is the letter and the value is the Ic for relative letter
**/
map<char,bit_vector> CreateIc(string P,string all,int m){
        map<char,bit_vector> Ic;
        for(int i = 0; i < all.length(); i++){
            bit_vector temp = bit_vector(m,0);
            Ic.insert(pair<char,bit_vector>(all[i],temp));
        }
        for(int j = 1; j < m; j++){
            bit_vector tempb = Ic[P[j]];
            Ic.erase(P[j]);
            tempb[j - 1] = 1;
            Ic.insert(pair<char,bit_vector>(P[j],tempb));
        }
return Ic;
}

/**
* Create the augmented suffix tree
```

```
*
* @param suffixtree The structure of suffix tree
* @param m The length of pattern
* @return a map the index is the string the represtned by the node of suffix tree and the value is relative
bit-vector
**/
map<string,bit_vector> OccVector(cst_sct3<> suffixtree,int m){
	map<string,bit_vector> OccV;
	cst_sct3<>::const_bottom_up_iterator it;
	for(it = suffixtree.begin_bottom_up(); it != suffixtree.end_bottom_up(); ++it){
	auto v = *it;
	bit_vector tempVec = bit_vector(m,0);
		if(v != suffixtree.root()){
			string temp = extract(suffixtree,v);
			if(suffixtree.is_leaf(v)){
				if((temp.length() != (m + 1)) && (temp.length() != 1)){
					tempVec[suffixtree.csa[suffixtree.id(v)] - 1] = 1;
				}
			}
			else{
				int up = suffixtree.degree(v);
				for(int i = 1;i <= up; i++){
					auto tempnode = suffixtree.select_child(v,i);
					string index = extract(suffixtree,tempnode);
					bit_vector indexvector = OccV[index];
					int temporig = tempVec.get_int(0,m);
					int tempnew = indexvector.get_int(0,m);
					int tempfinal = temporig | tempnew;
					tempVec.set_int(0,tempfinal,m);
				}
			}
			OccV.insert(pair<string,bit_vector>(temp,tempVec));
		}
		else{
			int up = suffixtree.degree(v);
				for(int i=1;i<=up;i++){
					auto tempnode = suffixtree.select_child(v,i);
					string index = extract(suffixtree,tempnode);
					bit_vector indexvector = OccV[index];
					int temporig = tempVec.get_int(0,m);
					int tempnew = indexvector.get_int(0,m);
					int tempfinal = temporig | tempnew;
					tempVec.set_int(0,tempfinal,m);
				}
```

```
                OccV.insert(pair<string,bit_vector>(" ",tempVec));
            }
        }
        return OccV;
}

/**
* Get the string represtned by the node in suffix tree depend on the T[i][j]
*
* @param S The string T[i][j]
* @return a string represtned by the relative node
**/
string findOccVectorS(string S){
        string index;
        auto v = cst.child(cst.root(), S[0]);
        if(cst.size(v) != cst.size(cst.root())){
            string a = extract(cst, v);
            int len = a.length();
            int mark = 0;
            for(int i = 0; (i < len) && (a[i] != '\0') && (i < S.length()); i++){
                if(a[i] != S[i]){mark = 1;break;}
            }
            auto tempnode = v;
            if(mark == 0){
                string temps;
                for(int j = len; j < S.length(); j++){
                    if(j == len){
                        tempnode = cst.child(tempnode,S[j]);
                        int tempnum = cst.size(tempnode);
                        if(tempnum == cst.size(cst.root())){mark = 1;break;}
                        temps = extract(cst, tempnode);
                        len = temps.length();
                    }
                    if(temps[j] != S[j]){mark = 1;break;}
                }
            }
            if(mark == 0){
                index = extract(cst,tempnode);
            }
        }
        return index;
}

/**
```

```
* EDSM algorithm
*
* @param P The pattern string
* @param m The length of P
* @param T A segment of Elastic-Degenerate text(T[i])
* @param n The size of T
* @param fisrt The mark to show whehter the segement is the first one
**/
void EDSM(string P,int m,vector<string> T,int n,bool first){
	//initial part
	KMP k;
	//T[0]
	if(first){
		list<int> L0;
		//calculate the border table
		int* a0=k.bordertable(P,T);
		int temp0=m;
		//process first segment T[0]
		for(int i0 = 0; (i0 < T.size()) && (T[i0] != ""); i0++){
			temp0 = temp0 + T[i0].length();
			int temp = temp0;
			while(a0[temp + 1] != 0){
				//add Bps to list
				L0.push_back(a0[temp + 1] - 1);
				temp = a0[temp + 1] - 1;
			}
			if(T[i0].length() >= m){
				//KMP serach
				int x = k.algorithm(P,T[i0],T[i0].length(),m);
				//check whether repeat and output
				if(x != -1){
					if((!out.empty() && (out.top() != 0))||(out.empty())){
						out.push(0);
						cout<<"0"<<endl;
					}
				}
			}
			//jump the mark symbol
			temp0++;
		}
		previous = L0;
	}
	//T[i]
	else{
```

```
list<int> L;
int tempi = m;
bool add = false;
//compute border table
int* ai = k.bordertable(P,T);
for(int j = 0; j < T.size(); j++){
	if(T[j] == "E"){add = true;}
	else{
		tempi = tempi + T[j].length();
		int temp = tempi;
		while(ai[temp + 1] != 0){
			//add Bps to list
			L.push_back(ai[temp + 1] - 1);
			temp = ai[temp + 1 ] - 1;
		}
		if(T[j].length() < m){
			//find all start positions
			list<int> ret;
			ret = SerachSinP(T[j]);
			//check whether can extend
			if(!previous.empty() && !ret.empty()){
				list<int>::iterator pre_ite;
				list<int>::iterator A_ite;
				for(pre_ite = previous.begin(); pre_ite != previous.end(); ++pre_ite){
					int pre_num = *pre_ite;
					for(A_ite = ret.begin(); A_ite != ret.end(); ++A_ite){
						int A_num = *A_ite;
						if(pre_num + 1 == A_num){L.push_back(pre_num + T[j].length());}
					}
				}
			}
		}
		if(T[j].length() >= m){
			//KMP serach
			int x = k.algorithm(P,T[j],T[j].length(),m);
			//check repeat and output
			if(x != -1){
				if((!out.empty() && (out.top() != i))||(out.empty())){
				out.push(i);
				cout<<i<<endl;
				}
			}
		}
		//compute Bsp
```

```
                    list<int> Bsp;
                    Bsp=ComputeBsp(T[j]);
                    //check whether exist full occurence
                    if(!previous.empty() && !Bsp.empty()){
                        list<int>::iterator pre_ite2;
                        list<int>::iterator Bsp_ite;
                        for(pre_ite2 = previous.begin(); pre_ite2 != previous.end(); ++pre_ite2){
                            int pre_num = *pre_ite2;
                            for(Bsp_ite = Bsp.begin(); Bsp_ite != Bsp.end(); ++Bsp_ite){
                                int Bsp_num = *Bsp_ite;
                                if(pre_num + Bsp_num + 2 == m){
                                    if((!out.empty() && (out.top() != i))||(out.empty())){
                                        out.push(i);
                                        cout<<i<<endl;
                                    }
                                }
                            }
                        }
                    }
                    //jump the mark symbol
                    tempi++;
                }
            }
            if(add){L.merge(previous);}//extend when empty exist
            previous = L;
        }
    }

    /**
    * EDSM-BV Algorithm
    *
    * @param P The pattern string
    * @param m The length of pattern string
    * @param T The Elastic-Degenerate text
    * @param n The length of Elastic-Degenerate text
    * @param all The string of all letters in alphabet
    * @return the number of occurrence
    **/
    void EDSM_BV(string P,int m,vector<string> T,int n,bool first){
        KMP k;
        if(first){
            bit_vector vecB = bit_vector(m,0);
            //calcualte the border table
            int* a0 = k.bordertable(P,T);
```

```
        int temp0 = m;
        //process first segment T[0]
        for(int i0 = 0; (i0 < T.size()) && (T[i0] != ""); i0++){
            temp0 = temp0 + T[i0].length();
            int temp = temp0;
            while(a0[temp + 1] != 0){
                //add Bps to bit-vector
                vecB[a0[temp + 1] - 1] = 1;
                temp = a0[temp + 1] - 1;
            }
            if(T[i0].length() >= m){
                //KMP
                int x = k.algorithm(P,T[i0],T[i0].length(),m);
                //check repeat and output
                if(x != -1){
                    if((!out.empty() && (out.top() != 0)) || (out.empty())){
                        out.push(0);
                        cout<<"0"<<endl;
                    }
                }
            }
            //jump the mark symbol
            temp0++;
        }
    int B = vecB.get_int(0,m);
    }
    else{
        int result;
        bit_vector B1 = bit_vector(m,0);
        int tempi = m;
        bool add = false;
        //compute border table
        int* ai = k.bordertable(P,T);
        for(int j = 0; j < T.size(); j++){
            if(T[j] == "E"){
                add=true;
            }
            else{
            tempi = tempi + T[j].length();
            int temp = tempi;
            while(ai[temp + 1] != 0){
                //add Bps to bit-vector
                B1[ai[temp + 1] - 1] = 1;
                temp = ai[temp + 1] - 1;
```

```
}
if(T[j].length() < m){
	//get the bit-vector relative to T[j]
	bit_vector occ_vec = bit_vector(m,0);
	string index = findOccVectorS(T[j]);
	occ_vec = occvector[index];
	//AND operation and shift
	int B2 = 0;
	int tempocc = occ_vec.get_int(0,m);
	B2 = B & tempocc;
	B2 = B2 << T[j].length();
	//OR operation and add to B1
	int tempB1 = B1.get_int(0,m);
	tempB1 = B2 | tempB1;
	B1.set_int(0,tempB1,m);
}
if(T[j].length() >= m){
	//KMP
	int x = k.algorithm(P,T[j],T[j].length(),m);
	//check repeat and output
	if(x != -1){
		if((!out.empty() && (out.top() != i)) || (out.empty())){
		out.push(i);
		cout<<i<<endl;
		}
	}
}
//check whether there exist full occurence
int B3 = 0;
B3 = B;
int len = T[j].length();
int upbound = min(len,(m-1));
for(int l = 0; l < upbound; l++){
	bit_vector IcTemp = Ic[T[j][l]];
	int tempIc = IcTemp.get_int(0,m);
	B3 = B3 & tempIc;
	B3 = B3 << 1;
	bit_vector vecB3 = bit_vector(m,0);
	vecB3.set_int(0,B3,m);
	if(vecB3[m - 1] == 1){
		//check repeat and output
		if((!out.empty() && (out.top() != i)) || out.empty()){
			out.push(i);
			cout<<i<<endl;
```

```
                        }
                    }
                }
                //jump the mark symbol
                tempi++;
                }
            }
            int t2 = B1.get_int(0,m);
            //extend when empty exist
            if(add){
                result = B | t2;
                B = result;
            }
            else{B = t2;}
        }
    }
```