《算法分析与设计》实验报告

——基于 BWT 压缩算法和 FM 索引技术的字符串匹配

姓名：王勇程

学号：2015202012

一、 实验目的和要求

字符串匹配是一个很常见的问题，解决字符串问题的思想也被广泛地应用。本次实验中，我们需要求得长度为m的短字符串在一个长度为n的长字符串中的精确匹配。

由于朴素的匹配算法时间复杂度将达到$O(m*n)$。当$n \gg m$时，算法的效率会很低本实验需要在用$O(m)$时间复杂度得到短串的精确匹配，其中。该匹配过程主要运用了 BWT(Burrows–Wheeler Transform)压缩算法和 FM 索引技术。

二、 实验环境

操作系统: Windows 8.1 专业版

CPU: Intel Core i5-5200U @ 2.2GHZ

RAM: 4.00 GB

三、 算法简述

1、 建立 BWT 结构

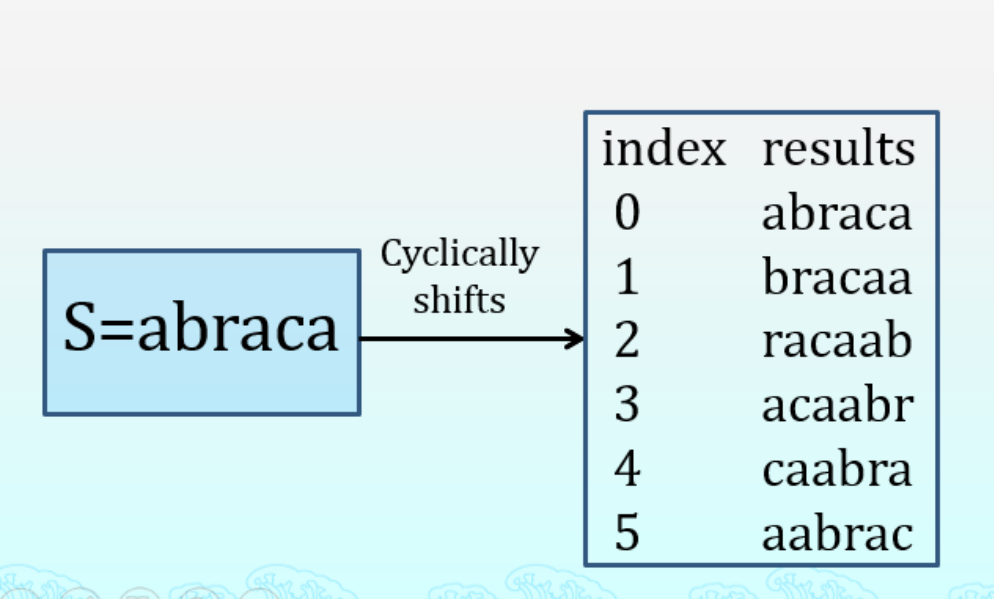

图 1. BWT 算法循环左移示意图

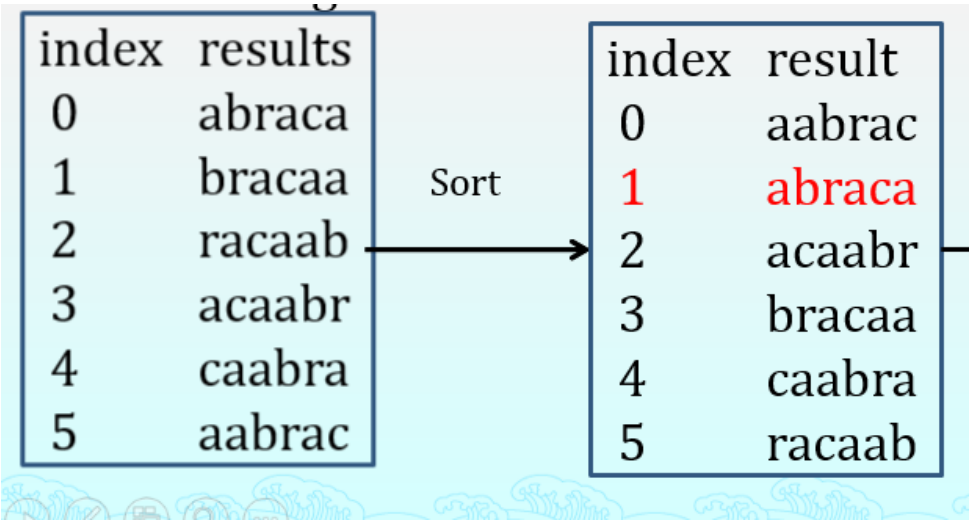

图 2. 字典排序结果示意图

首先对整个字符串每次左移一位生成新的字符串，所以对长度为n的字符串会生成n个字符串，并将这n个字符串依次编号$0, \dots, n-1$。如图 1 所示。然后对这n个字符串按字典序排序，如图 2 所示。这里我们引入一个数组 Suffix，Suffix[i]保存的是字典排序后的序列第 i 行字符串原来的 index 值。如图 2 中, Suffix[0]=5, Suffix[1]=0, Suffix[2]=3。排序之后我们将新矩阵的第一列和最后一列取出，用于计算另外两个数据结构，详见下文。需要注意的是上述预处理过程需要$O(n^2)$的空间（如果使用不用生成矩阵创建索引的方法，则至少需要$O(n)$的空间）。由于我们的长串文本最多长达 900M，全部读取再生成索引是不现实的。因此我们需要先将长串分段，再进行预处理。

在实验中，我们从原文本字符串中提取出长度固定为 10000 字节的一系列字符串（最后一个字符串的长度不一定）。然后用长度为m的短字符串对这些长度为 10000 字节的字符串依次进行精确匹配，并输出匹配结果。这里需要注意的是，段与段的衔接部分可能也存在短串的成功匹配，所以我们需要在段与段之间设置一定长度的重叠区域。重

叠部分的长度只要大于或等于短串的长度m(200 字节)即可。

2、 由 BWT 编码结果计算辅助数据结构

接下来我们用 BWT 编码结果中的第一列字符串与最后一列字符串计算辅助数据结构 C 数组和 OCC 数组。

其中 C 数组的元素 C[x]表示在字典序排序结果中第一列元素比 x 小的串多少个。C 数组的计算比较简单，只要遍历排序结果的第一列字符即可得到，这个过程需要$O(n)$的时间。OCC 数组是一个二维数组，它的元素 OCC[x][i]表示字典排序结果中的前 i 行（即第 0 至 i-1 行）字符串的最后一列里有几个字符 x。因此也只需遍历排序结果的最后一列字符串。这个过程需要的时间也是$O(n)$的。

例如在图 2 的数据集中,我们可以计算得到:

C[a]=0, C[b]=3, C[c]=4, C[r]=5。

OCC[a][3]=1, OCC[a][4]=2, OCC[a][5]=3, OCC[a][6]=3。

3、 通过以上数据结构进行精确匹配

我们从 BWT 矩阵的构建方法上可以得到该矩阵的两个重要性质：

第一个性质是，对于任意一行的首字符和尾字符，原序列中首字符是尾字符的后一个元素。所以我们可以把目标短串 S 从后往前匹配，即先匹配目标短串 S 的最后一个字符，然后计算出该匹配行的最后一个字符在 BWT 矩阵中作为首字符出现的对应行。跳到这个对应行之后，我们再继续类似的上述操作，检查短串 S 的倒数第二个字符是否与这

一行的首元素匹配，如果匹配则继续跳转，继续匹配……如此继续下去，如果中途匹配失败，则返回失败信息；如果精确匹配成功，则返回对应行的 Suffix 值，这个值就是目标串在长串中的起始位置偏移量，也是我们要求的结果。如图 3 所示。

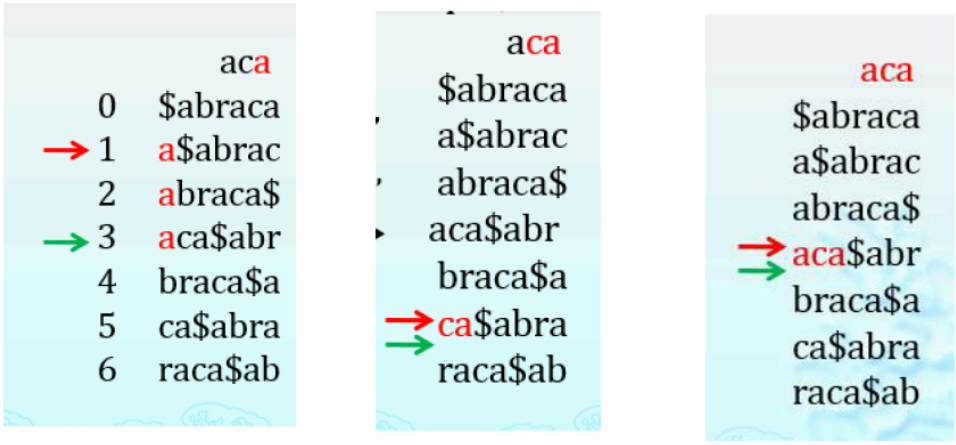

图 3. 匹配过程示意图

第二个性质是第一列字符串中的字符与最后一列字符串中的相同字符具有顺序不变性。例如图 2 中，第一列字符串中的第二个 a 与最后一列字符串中的第二个 a 在源字符串中的位置相同，即它们是同一个 a。通过这个性质我们就可以通过 OCC 数组和 C 数组计算出当前指针区间[sp,ep]需要跳转到的新区间[sp',ep']，且 sp'和 ep'满足下式：

sp' = C[ch] + OCC[ch][sp] ;　ep' = sp' + OCC[ch][ep+1] - OCC[ch][sp] -1;

在算法最后得到精确匹配区间时，我们再通过每一行对应的 Suffix 值即可得到目标短串在长串中所有精确匹配结果的起始位置。

4、 索引部分存储策略

本实验中，由于我们是将原文本分成长度固定为 10000 字节的一系列

字符串并创建数据结构，所以我们需要依次遍历所有这些长度为10000的字符串的索引数据，最终得到短串在整个长串文本中的精确匹配。

由于完整保存Suffix数组、OCC数组需要将近20G的内存空间，我们采用部分存储策略来减少索引的存储数据量。具体思想是：对于Suffix[i]和OCC[x][i]，只有当32整除i时才把它们存储下来，这样可以大大节省存储空间。我们在本地保存的索引数据是Suffix[0, 32, 64 … …]以及OCC[A][0,32,64 … …]，OCC[C][0,32,64 … …]，OCC[G][0,32,64……]，OCC[T][0,32,64……]。同时，我们还需要保存BWT矩阵的最后一列字符串，这样当我们需要使用的数据Suffix [i]和OCC[x][i]中32不整除i时，我们仍然可以借助最后一列字符串和本地存储的Suffix数组以及OCC数组计算出想要的数据。

注：本地保存的索引格式如下:

首先是C[0,…,5]。依次表示ACGT的起始位置和(T的结束位置+1)。

然后依次是 Suffix [0], OCC[A/C/T/G][0]; Suffix[32], OCC[A/C/T/G][32]; Suffix [64], OCC[A/C/T/G][64]……

最后再存储BWT矩阵的最后一列字符串。

计算Suffix [i]步骤如下：如果i整除32，则可以直接得到结果。否则需要先计算Suffix[i-1]，然后再推出Suffix [i]。当然为了得到Suffix [i-1]可能先要计算Suffix[i-2]……这是一个递归计算的过程。

计算OCC[A/C/T/G][i]的过程也类似，是一个递归的过程。

详细的算法将在下文中阐述。

四、 创建索引算法时间空间复杂度分析

1、 算法伪代码：

```
#define blocksize    10000    //每段长串长度设为 10000
#define coversize    200    //衔接部分设置一个长度为 200 的重复区间
for(i=0;i*(blocksize-coversize)<longStringLength;i++){
1.      longstring = GetLongString(longfilename,i);    //得到长串
        //生成两倍长的长串作为快排的原料
2.      string doublelongstring = longstring+longstring;
3.      string lastcolumn;                    //矩阵的最后一列字符串
4.      int len=longstring.length();          //len 是长串的长度

5.      for(k=0;k<len;k++)    suffix_array[k]=k;      //初始化 index 值
        //按字典序排序，计算 Suffix Array
6.      quickSort(suffix_array,0,len-1,doublelongstring);
        //计算 C 数组和 OCC 数组
7.      Count_C_OCC(longstring, C, OCC, suffix_array);
        //计算最后一列字符串
8.      for(int j=0; j <len; j++)
9.      lastcolumn+=longstring[(suffix_array[j]+len-1)%len];
        //将上述数据结构部分保存到本地
```

10. SAVE(suffix_array, C, OCC, lastcolumn, len);

}

2、 空间复杂度分析

设长串长度为n。（我们实验中n小于或等于 10000 字节）

步骤 1: $O(n)$

步骤 2: $O(n)$

步骤 3: $O(1)$

步骤 4: $O(1)$

步骤 5: $O(n)$

步骤 6: $O(n)$

步骤 7: $O(5n)$

步骤 8+9: $O(n)$

步骤 10: $O(n)$

综上，创建索引算法需要的空间为$O(n)$。具体一点说是$\Theta(5n)$。

3、 时间复杂度分析

设原文本长度为 N，分段后的长串长度为n。（实验中取n等于 10000 字节）

步骤 1: $O(n)$

步骤 2: $O(n)$

步骤 3: $O(1)$

步骤 4: $O(1)$

步骤 5: $O(n)$

步骤 6: $O(nlogn)$

步骤 7: $O(n)$

步骤 8+9: $O(n)$

步骤 10: $O(n)$

以上步骤的时间复杂度为$O(nlogn)$。由于 for 循环一共进行$O(\frac{N}{n})$次。所以总的时间复杂度为$O(Nlogn)$。

五、 精确匹配算法时间空间复杂度分析

1、 算法伪代码：

```
#define blocksize    10000    //每段长串长度设为 10000
#define coversize    200    //衔接部分设置一个长度为 200 的重复区间
for(int i=0;i*(blocksize-coversize)<longStringLength;i++){
1.      Read Suffix_Array,C,OCC and lastcolumn from index file;
        //运行匹配算法
2.      int len=longstring.length();
3.      int cur = shortstring.length()-1;
4.      char ch = ShortString[cur]; //ch is the last character of ShortString;
5.      Let sp and ep be the start pointer and end pointer of the matching
        region;
6.      Count sp and ep according to Array C and ch;
7.      while(sp<=ep){
8.          ch = shortstring[--cur];
```

```
9.              sp = C[ch] + GetOCC[ch][sp]    //递归计算 OCC[ch][sp]
10.             ep = sp + GetOCC[ch][ep+1] - GetOCC[ch][sp] -1;
11.             if(cur<=0)
12.                 if(sp<=ep) {
13.                     for(int j = sp; j <= ep; hh++)
14.                         cout<<GetSuffix(j)+i*(blocksize-coversize)<<endl;
15.                     return True;}
16.                 else return false
17.       } //end while
```

注：

递归计算 Suffix_Array[pos]的伪码如下，其运行时间摊还代价为$O(1)$

```
int GetSuffix(int pos){ //如果 32 整除 pos，则 Suffix Array 已被记录
    if(pos%32 == 0)
        return suffix[pos];
    else return GetSuffix(C[lastcolumn[pos]]+GetOCC(ch,pos))+1 ;
}
```

递归计算 OCC[ch] [pos]的伪码如下，其运行时间摊还代价为$O(1)$

```
int GetOCC(char ch, int pos){
    if(pos%32 == 0)      return OCC[ch][pos];
    int k = pos/32;
    int result = OCC[ch][k];
```

```
    for(int j=32*k; j<pos; j++)
        if(lastcolumn[j] == ch)
            result++;
    return result;
}
```

2、 空间复杂度分析

设长串长度为n。（我们实验中n小于或等于 10000 字节）

步骤 1: $O(n)$

步骤 2-6: $O(1)$

步骤 7-17: $O(n)$

综上，精确匹配算法需要的空间为$O(n)$。主要用于保存上一步中在本地部分存储的 Suffix 数组、OCC 数组、C 数组和 BWT 矩阵最后一列字符串的值。

3、 时间复杂度分析

设原文本长度为 N，分段后的长串长度为n。（实验中取n等于 10000 字节）

步骤 1: $O(n)$

步骤 2-6: $O(1)$

步骤 8-11: 每次 while 循环$O(1)$。总共$O(m)$。

步骤 12-17: 一般情况下$O(1)$，极端情况下$O(n)$

由于短串长度为m，因此 while 循环最多执行$O(m)$次。其中步骤 8-

11 最多执行$O(m)$次，所以运行时间总共为$O(m)$。而步骤 12-17 只在最后一次 while 循环中执行，且在一般情况下运行时间$O(1)$。因此，**在一般情况下**，每次 for 循环的时间复杂度为$O(m)$。

然而天下没有免费的午餐！部分存储的一个隐患就是**极端情况**下性能低下。例如：长串为'AAAAAAAAAAAAAAA……'，短串为'A'。如果我们完全存储 Suffix 数组，则只需返回 Suffix Array[0,…,n-1]。而如果采用了部分存储，在步骤 14 中，我们将调用 GetSuffix 函数递归计算所有 Suffix 的值，这需要$O(n)$的时间！但是一般情况下，调用 GetSuffix 函数计算对应 Suffix 数组值的平均时间是常数级别的。

综上，**在一般情况下**，每次 for 循环精确匹配短串的时间复杂度为$O(m)$。以上步骤循环$O(\frac{N}{n})$次。所以总的时间复杂度为$O\left(\frac{Nm}{n}\right)$。

如果完整存储三个数据结构，那么则所有情况下每次 for 循环的时间复杂度都是$O(m)$的。但是创建索引的过程会变得很慢，因为写文件到本地磁盘需要花费大量时间。

六、 实验源代码

建立索引程序源代码：Create_part_index.cpp

```
#include <iostream>
#include <string>
#include <stdlib.h>
#include <fstream>
#include <sstream>
```

```
#define blocksize    10000
#define coversize    200

using namespace std;

//快速排序获得 suffix array 数组
void quickSort(int s[], int l, int r, string sentence)
{
    int k;
    int len = sentence.length()/2;
     if (l< r) {
            int i = l, j = r;
            int x = s[l];
            string standard = sentence.substr(s[l],len);
            while (i < j) {
                while(i < j &&
sentence.substr(s[j],len).compare(standard)>=0)
                j--;
                if(i < j)
                    s[i++] = s[j];
                while(i < j &&
```

```
sentence.substr(s[i],len).compare(standard)<0)
                        i++;
                if(i < j)
                    s[j--] = s[i];
            }
            s[i] = x;
            quickSort(s, l, i - 1,sentence);
            quickSort(s, i + 1, r,sentence);
        }
}
//部分存储 s,C,OCC。每隔 32 位存一次。比如 s[0],s[32],s[64]...
void SAVE(int s[],int *C, int **OCC, string lastcolumn, int len){
    ofstream file;
    file.open("suffixArray2new.txt",ios::app);
    file<<C[0]<<" "<<C[1]<<" "<<C[2]<<" "<<C[3]<<" "<<C[4]<<" ";     //先
存 C 数组的五个元素
    for(int k=0;k<len;k++) {
        if(k%32==0){
            file<<s[k]<<" ";
            file<<OCC[0][k]<<"       "<<OCC[1][k]<<"       "<<OCC[2][k]<<"
"<<OCC[3][k]<<" ";
        }
```

```
        }
        file<<lastcolumn<<" "<<endl;
        file.close();
}
//读取目标长串，之前已经读了 pre_turns 次长串
string GetLongString(string longfilename,int pre_turns){
    char *file = (char *)longfilename.data();
    ifstream in(file);
    string temp;
    getline(in,temp);     //跳过第一行
    if(pre_turns!=0) {
        in.seekg(pre_turns*(blocksize-coversize),ios::cur);          //寻址到
目标长串起始位置
    }
    ostringstream buf;
    char ch;
    //读取目标长串
     while(buf){
        in.get(ch);
        if(ch<65||ch>90)       //不是 ACGT
            break;
        buf.put(ch);
```

```
        if(buf.str().size() == blocksize)
        break;
    }
    in.close();
    return buf.str();
}

//计算 C 数组和 OCC 数组
void Count_C_OCC(string longstring,int C[],int **OCC, int *suffix_array){
    int len=longstring.length();
    int offset;
    //OCC[0,1,2,3]分别对应 ACGT 的起始位置。OCC[4]是 T 的结束位置
+1
    C[0]=0; C[1]=0; C[2]=0; C[3]=0; C[4]=0;
    for(int j=0; j<len; j++){
        offset = suffix_array[j];
        char begin = longstring[offset];
        char end = longstring[(offset+len-1)%len];
        switch(begin){
            case 'A':C[1]++;C[2]++;C[3]++;C[4]++;
                break;
            case 'C':C[2]++;C[3]++;C[4]++;
```

```
            break;
        case 'G':C[3]++;C[4]++;
            break;
        case 'T':C[4]++;
            break;
    }
    OCC[0][j+1]=OCC[0][j];
    OCC[1][j+1]=OCC[1][j];
    OCC[2][j+1]=OCC[2][j];
    OCC[3][j+1]=OCC[3][j];
    switch(end){
        case 'A':
            OCC[0][j+1]+=1;
            break;
        case 'C':
            OCC[1][j+1]+=1;
            break;
        case 'G':
            OCC[2][j+1]+=1;
            break;
        case 'T':
            OCC[3][j+1]+=1;
```

```
                    break;
                default:
                    break;
            }
        }   // end for j
}

int main()
{
    string shortfilename = "search_SRR00001test.txt"; //自己的测试文件。注意只能有一个短串
    string longfilename = "SRR163132_rows_ATCG.txt"; //长串文件
    string    longstring;
    int i;

    //获取长串长度
    int longStringLength;
    char *file = (char *)longfilename.data();
    ifstream in(file);
    string s;
    getline(in,s);
    in.close();
```

```
longStringLength = atoi(s.c_str());
cout<<"here:"<<longStringLength<<endl;
//主循环，每次对长度为 10000 的长串子段创建索引，并存储到本地
for(i=0;i*(blocksize-coversize)<longStringLength;i++){
    //生成 suffix array
    longstring = GetLongString(longfilename,i);   //得到长串
    string doublelongstring = longstring;     //不用生成矩阵的快排的策略
    doublelongstring+=longstring;
    int len=longstring.length();
    int* suffix_array= (int *)malloc(sizeof(int)*len);
    int k;
    for(k=0;k<len;k++)
        suffix_array[k]=k;
    quickSort(suffix_array,0,len-1,doublelongstring);

    //生成 c 和 occ 数组
    int *C=(int *)malloc(5*sizeof(int));
    int **OCC=(int **)malloc(4*sizeof(int *));
    for(k=0;k<4;k++){
        OCC[k]=(int *)malloc((len+1)*sizeof(int));
```

```
            memset(OCC[k],0,(len+1)*sizeof(int));
        }
        Count_C_OCC(longstring, C, OCC, suffix_array);
        string lastcolumn;
        for(int j=0; j <len; j++){
            lastcolumn+=longstring[(suffix_array[j]+len-1)%len];
        }
        //保存到本地
        SAVE(suffix_array, C, OCC, lastcolumn, len);
        free (suffix_array);
        for(k=0;k<4;k++) free (OCC[k]);
        free(OCC);
        free(C);
    }
     return 0;
}
```

精确匹配程序源代码: **Match.cpp**

```
#include <iostream>
#include <string>
#include <stdlib.h>
#include <fstream>
```

```
#include <sstream>
#include <string.h>

#define blocksize    10000       //分段大小
#define coversize    200         //覆盖长度，需大于等于短串长度


using namespace std;

//读取待匹配短串
string GetShortString(string filename)
{
    char *file = (char *)filename.data();
    ifstream in(file);
    string s;
    getline(in,s);
    //cout<<"s="<<s<<endl;
    in.close();
    return s;
}
//读取目标长串，之前已经读了 pre_turns 次长串
string GetLongString(string longfilename,int pre_turns){
```

```
    char *file = (char *)longfilename.data();
    ifstream in(file);
    string temp;
    getline(in,temp);       //跳过第一行
    if(pre_turns!=0) {
        in.seekg(pre_turns*(blocksize-coversize),ios::cur);        //寻址到
目标长串起始位置
    }
    ostringstream buf;
    char ch;
    while(buf){
        in.get(ch);
        if(ch<65||ch>90)       //不是 ACGT
            break;
        buf.put(ch);
        if(buf.str().size() == blocksize)
        break;

    }
    in.close();
    return buf.str();
}
```

```
//求 OCC[case1][pos]
int GetOCC(int case1, int pos,int **OCC,string lastcolumn){
    int k = pos/32;
    if(pos%32 == 0)
        return OCC[case1][k];

    char ch;
    if(case1 == 0) ch='A';
    else if (case1 ==1) ch='C';
    else if (case1 ==2) ch='G';
    else if (case1 ==3) ch='T';

    int result = OCC[case1][k];
    for(int j=32*k; j<pos; j++){
        if(lastcolumn[j] == ch)
            result++;
    }
    return result;
}

//求 suffix[pos]
```

```
int GetSuffix(int pos,int *suffix,int *C,int **OCC,string lastcolumn){
    int k = pos/32;
    if(pos%32 == 0)
        return suffix[k];
    int case1;

    char ch=lastcolumn[pos];
    if(ch=='A') case1=0;
    else if(ch=='C') case1=1;
    else if(ch=='G') case1=2;
    else if(ch=='T') case1=3;

    int lastpos = C[case1]+GetOCC(case1,pos,OCC,lastcolumn);
    return GetSuffix(lastpos,suffix,C,OCC,lastcolumn)+1 ;
}


void BWT_MATCH(string longstring,string shortstring,int *C,int **OCC,int
*suffix_array,string lastcolumn,int max1,int row){
    int len=longstring.length();
    int offset;
    string temp1;
```

```
int cur = shortstring.length()-1;
int sp,ep;
char Last_OF_ShortString = shortstring[cur];
int case1;
if (Last_OF_ShortString == 'A') case1 =0;
else if (Last_OF_ShortString == 'C') case1 =1;
else if (Last_OF_ShortString == 'G') case1 =2;
else if (Last_OF_ShortString == 'T') case1 =3;
sp = C[case1];
ep = C[case1 +1]-1;

while(sp<=ep){
    if (shortstring[cur-1] == 'A') case1 =0;
    else if (shortstring[cur-1] == 'C') case1 =1;
    else if (shortstring[cur-1] == 'G') case1 =2;
    else if (shortstring[cur-1] == 'T') case1 =3;
    cur --;
    Last_OF_ShortString = shortstring[cur];
    int newsp = C[case1] + GetOCC(case1,sp,OCC,lastcolumn);
    int newep = newsp + GetOCC(case1,ep+1,OCC,lastcolumn) -
GetOCC(case1,sp,OCC,lastcolumn) -1;
```

```
            sp = newsp;
            ep = newep;

            if(cur<=0) {
                if(sp<=ep){
                    cout<<"Succeed! There is/are "<<ep-sp+1<<" match places
with offset(s):"<<endl;
                    for(int hh = sp; hh <= ep; hh++)

    cout<<"sp="<<sp<<","<<GetSuffix(hh,suffix_array,C,OCC,lastcolumn)
+row*(blocksize-coversize)<<endl;
                }
                else cout<<"No matched result."<<endl;
                break;
            }
    } //end while
}

int main()
{
    string shortfilename = "search_SRR00001test.txt"; // 待匹配短串文
件
```

```
string longfilename = "SRR00001_rows_ATCG.txt";        // 长串文件
string shortstring = GetShortString(shortfilename);   //读取短串
string    longstring;
int i,k;

ifstream in(longfilename);
string s;
getline(in,s);
in.close();
int longStringLength = atoi(s.c_str());

ifstream datafile("suffixArray1new.txt");
string datastream;

for(i=0;i*(blocksize-coversize)<longStringLength;i++){
    getline(datafile,datastream); //datastream 是一整行数据，依次包括 C[0...4], s[0],OCC[0][0...3] ...
    stringstream value(datastream);
    string temp1;

    longstring = GetLongString(longfilename,i);   //长串
    int len=longstring.length();
```

```
int *suffix_array = (int *)malloc((len/32+1)*sizeof(int));
int *C=(int *)malloc(5*sizeof(int));
int **OCC=(int **)malloc(4*sizeof(int *));
for(k=0;k<4;k++){
    OCC[k]=(int *)malloc((len/32+1)*sizeof(int));
    memset(OCC[k],0,(len/32+1)*sizeof(int));
}

for(int j=0; j<5; j++){    //首先读取 C 数组
    value >> temp1;
    C[j] = atoi(temp1.c_str());
}

//读取 suffix 数组和 OCC 数组
int max1;
if (len%32 == 0) max1 = len/32;
else max1 = len/32 + 1;
for(int j=0; j<max1 ; j++){
    value >> temp1;    suffix_array[j] = atoi(temp1.c_str());
    value >> temp1;    OCC[0][j] = atoi(temp1.c_str());
    value >> temp1;    OCC[1][j] = atoi(temp1.c_str());
    value >> temp1;    OCC[2][j] = atoi(temp1.c_str());
```

```
            value >> temp1;    OCC[3][j] = atoi(temp1.c_str());
        }
        string lastcolumn;
        value >> lastcolumn;

        //匹配短串

    BWT_MATCH(longstring,shortstring,C,OCC,suffix_array,lastcolumn,max1,i);
        for(k=0;k<4;k++) free (OCC[k]);
        free(OCC);
        free(C);
        free(suffix_array);
    }  // end i
    datafile.close();
} // end main
```

七、 实验结果（存储空间、查询时间以及结果截图）

1、 存储空间

对”SRR00001_rows_ATCG.txt”建立的的本地索引文件大小为227.4MB。

对”SRR163132_rows_ATCG.txt”建立的的本地索引文件大小为 905MB。

对”SRR311115_rows_ATCG.txt”建立的的本地索引文件大小为 1.65GB。

2、 创建索引时间

对”SRR00001_rows_ATCG.txt”建立索引的耗时为 30min。

对”SRR163132_rows_ATCG.txt” 建立索引的耗时约为 120min。

对”SRR311115_rows_ATCG.txt” 建立索引的耗时约为 220 min。

3、 查询时间

（i）对”SRR00001_rows_ATCG.txt”中第 6 个长度为 200 的子串进行精确匹配。得到偏移量为 1000，结果正确。耗时 57.73 s。

```
Succeed! There is/are 1 match places with offset(s):
sp = 3856, offset = 1000

--------------------------------
Process exited after 57.73 seconds with return value 0
请按任意键继续. . .
```

(ii) 对”SRR163132_rows_ATCG.txt”中第 4 个长度为 200 的子串进行精确匹配。得到偏移量为 600，结果正确。耗时 246.8 s。

```
Succeed! There is/are 1 match places with offset(s):
sp = 6013, offset = 600

--------------------------------
Process exited after 246.8 seconds with return value 0
请按任意键继续. . .
```

(iii)对”SRR311115_rows_ATCG.txt”中第 3 个长度为 200 的子串进行精确匹配。得到偏移量为 400，结果正确。耗时 434.7 s。

```
Succeed! There is/are 1 match places with offset(s):
sp = 8817, offset = 400

--------------------------------
Process exited after 434.7 seconds with return value 0
请按任意键继续. . .
```