大作业实验报告

邵宁录　2018202195

2020 年 12 月 15 日

目录

1 功能说明

1. 读入文件功能
2. 字符串匹配功能
3. 文件写入功能

注意事项：

1. 本项目使用 c++11 标准进行编写，编译时请加上 -std=c++11 参数。
2. 如果想要以最快速度运行本项目，请使用 -O3 参数进行编译优化，但进行编译优化后有可能出现错误的结果。

2 输入输出说明

2.1 输入

1. 算法初始化时，需要输入读入文件的文件名

```
(base)  lsn@localhost  ~/Course/大三上/DSA下/作业/大作业  ./a.out
[INFO] Please Input File Name: big.txt
[INFO] Load Start!
花费了1.14583秒
[INFO] Load Finished!
```

图 1: 输入读入文件示意图

2. 算法初始化完成后，需要输入想要匹配的字符串

```
[INFO] BWTDecode Start!
花费了28.5093秒
[INFO] BWTDecode Finished!
[INFO] Please Input String (Max 1k): accggtta
```

图 2: 输入匹配字符串示意图

2.2 输出

2.2.1 标准情况输出

标准情况下，输入匹配字符后，输出为匹配到的字符个数，并会将结果写到 *result.txt* 文件中去。

其中，匹配结果用换行符进行分隔。

```
[INFO] Please Input String (Max 1k): accggtt
[INFO] Start Matching...
[INFO] Match Finished! Total Result: 3281
花费了0.03321秒
[WARNING] Writing result to file, please not turn off the power...
[INFO] Write Finished!
```

图 3: 输出结果示意图

2.2.2 针对测试文件的输出

由于测试文件有 3 个，每个的都包含 1000 个长度为 200 的字符串。为了方便测试，我先用 python 将其连接成 3 个长度为 20w 字符串，然后再进行测试。

测试时需要修改一下代码 *main* 函数的部分，其中详细注释已在代码中给出。

匹配结果会写入输出文件 *result.txt* 中。

3 设计原理说明

设计原理分为以下几个部分：

1. 后缀排序部分
2. 分治思想部分
3. BWT 与 FM Index 构建部分
4. 精确匹配部分

其中，后缀排序部分是实现 BWT 与 FM Index 构建的基础。

3.1 Suffix Array Sorting

首先是 Suffix Array Sorting 的设计原理。Suffix Array Sorting 即后缀排序，我们需要定义什么是后缀：

定义 1. *后缀是指从某个位置 i 开始到这个串末尾结束的一个特殊子串。字符串 s 的从第 i 个字符开始的后缀表示为 $Suffix(i)$，也就是 $Suffix(i) = s[i:]$。*

在查阅论文与网络资料的时候，我发现解决后缀排序主要有以下几种方法：

1. 参考文献 **A Block-sorting Lossless Data Compression Algorithm** 中的后缀排序算法
2. 后缀树算法
3. 倍增算法
4. DC3 算法

在综合考虑了**时间复杂度**、**空间复杂度**、**实现难度和实际效果**后，我选择了倍增算法。

倍增算法的主要思路是：用倍增的方法对每个字符串开始的长度为 2^k 的子字符串进行排序，求出排名。k 从 0 开始，每次加 1，当 2^k 大于字符串长度 n 以后，每个字符开始的长度为 2^k 的子字符串便相当于所有的后缀。并且这些子字符串都一定已经比较出大小，那么此时就是最终的结果。

其中，排序以字典序作为依据，并且是用基数排序来保证稳定性以及相同关键字的元素被分进一个排名。

例 1. 下面以字符串 $s = aabaaaab$ 为例来演示倍增算法排序的过程。

记 $rank$ 数组里存放第 i 次排序的结果，该数组含义为排在第 $index$ 的后缀字符串为 $value$，即如果 $rank[1] = 3$，则说明 $s[3:]$ 的排名为 *1*。

x 和 y 数组分别存放第一关键字和第二关键字。

1. 第一次排序时，由于只有 a 与 b 两个不同的字母，因此排名只有 *1* 和 *2*，$rank$ 数组的结果为 11211112。

2. 第二次排序时，由于 $k = 0$，所以需要往后看一位，因此 x 数组与 $rank$ 数组相等，y 数组应该与 $rank$ 往后一位相等，不足部分补 *0*。然后再排序，因此 $rank$ 的值更新为 12411123。

3. 第三次排序时，由于 $k = 1$，所以需要往后看两位，因此 x 数组与 $rank$ 数组相等，y 数组应该与 $rank$ 往后移两位相等，不足部分补 *0*。然后再排序，因此 $rank$ 的值更新为 46812357。

4. 以此类推，直至 $2^k > n$ 时结束。此时可以保证 $rank$ 数组中所有的排名都是不同的。

详细过程如下图所示。

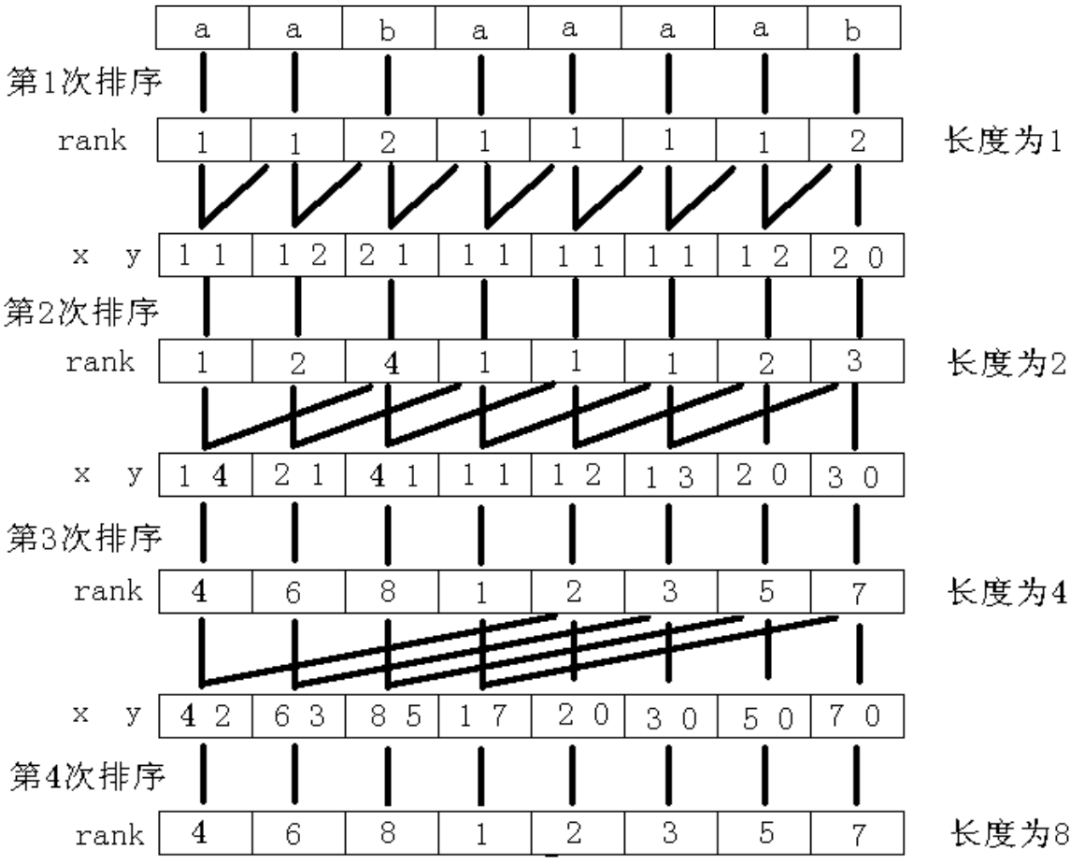

图 4: 倍增算法示意图

倍增算法使用基数排序，该排序是的复杂度为 $O(n)$ 。并且在最坏情况下，会排序 $\lg(n)$ 次。因此倍增算法的时间复杂度为 $O(n \lg(n))$ 。

3.2 分治思想

分治思想在本算法中起到了重要的作用。

由上一部分我们已经知道，倍增算法进行后缀排序的时间复杂度为 $O(n\lg(n))$，除了在特殊情况下才能使用的线性时间排序算法，这在排序中已经是一个理论最优的结果了。

但是，一次性排序一个大小为 10 亿的字符串还是为导致排序时间很长。因此我们可以通过分治的思想，将长串分为若干个短串。我个人将短串的长度设置为了 1000w。

下面我将通过计算说明分治思想所带来的的好处。

我们不妨设 1000w 为 n，那么 10 亿为 $100n$ 。

直接排序长串的时间大致为 $100n \times \lg(100n) = 100n\lg(n) + 100n\lg(100) \approx 100n\lg(n) + 600n$ 。

排序 100 个短串的时间大致为 $100 \times n\lg(n) = 100n\lg(n)$ 。

我们可以看到它们之间差了一个线性的时间，虽然在取近似时，这个线性时间是可以被忽略，但实际上线性项中的 n 非常的大，因此会对实际运行造成较大的影响，合理的分为较短的串会加快排序的速度。

3.3 BWT 与 FM Index

在该算法中，BWT 是 FM Index 过程中的一部分。FM Index 大致分为三个步骤：

1. BWT 算法
2. checkpoint 构建
3. 后缀排序构建 SA 数组

逻辑上，会按顺序执行以上三个步骤。但在实际执行中，可以直接根据后缀数组 SA 的结果得出 BWT 的结果。下面将按顺序介绍三个步骤。

3.3.1 BWT 算法

BWT 算法可以分为编码和解码两部分，我们这个算法只需要用到编码部分，因此下面将不涉及解码部分的原理。

其编码分为以下几个步骤：

1. 输入一个字符串 s。假设其中的所有字符都处于 $a-z$ 之间。
2. 在 s 的末尾加上一个标记字符 \$，选用 *dollar* 符的原因是它的 ASCII 码比 $a-z$ 的 ASCII 码都小。记这个新字符串为 s'。
3. 重复地将 s' 中的最后一个字符移到开头，每转移一次就得到一个新的字符串。重复该操作直至 \$ 回到原来的位置。
4. 将上一步得到的所有新字符从小到大按字典序排序，排序后的字符串数组记为 M。
5. 将 M 中的第一列的所有字符构成字符串 F，最后一列的所有字符构成字符串 L。
6. 其中字符串 L 是编码的输出结果，但在该算法中，字符串 F 也起到了作用，因此也需要保存下来。

例 2. 下面以字符串 $s = acaacg$ 为例演示这一过程。

1. 在 s 的最后加上标记字符 $\$$。

2. 不断地轮转字符串 s，再对他们进行排序，形成字符串数组 M。

3. 把第一列记为字符串 F，把最后一列记为字符串 L。

详细过程如下图所示。

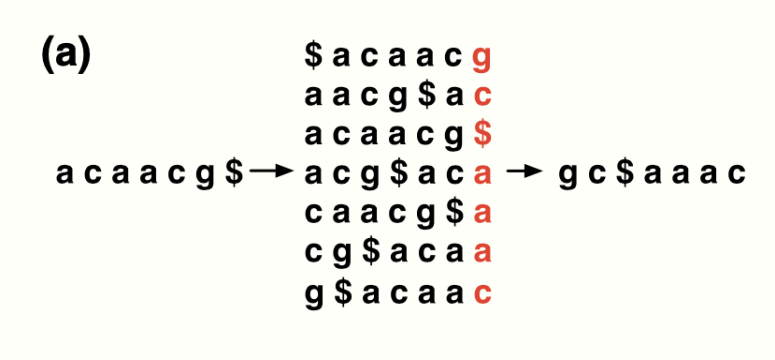

图 5: BWT 算法示意图

3.3.2 checkpoint 构建

checkpoint 是 FM Index 中的一种优化存储的手段。如果计算机内存够大或目标字符串很短的话，可以不考虑构建 checkpoint。

在介绍 checkpoint 之前，需要了解字典 C 和 Occ_buf 数组。

定义 2. 字典 C 记录了字符串 F 中各个字符第一次出现的位置。如 $F = \$aabbc$，则 $C = \{a : 0, b : 2, c : 4\}$。

定义 3. Occ_buf 数组记录了 L 中在第 row 行前，各个字符分别出现了多少次。如 $L = \$aabbc$，则 $Occ\_buf = [(a : 0, b : 0, c : 0), (a : 1, b : 0, c : 0), \ldots, (a : 2, b : 2, c : 1)]$。

我们可以设想，若对一个长度为 10 亿的字符串全都存储 Occ_buf 的话，则无疑存储空间消耗是十分巨大的，即空间复杂度巨大。但如果我们可以每隔一段数组去设置 checkpoint，这样就可以用少量的运算时间来换取大量的存储空间。

我参考了 *bowtie* 的 checkpoint 设置，它是以每隔 448 行设置一个 checkpoint，但实际上设置的更大也是可以的。

那么算法在运行时将会遇到以下两种情况：

1. 当选取的 row 刚好在 checkpoint 内。那么此时直接随机访问内部的值即可。

2. 当选取的 row 不在 checkpoint 内。那么此时访问比 row 小且离它最近的 checkpoint，之后对在遍历他们之间的字符进行计算。

3.3.3 后缀排序构建 SA 数组

SA 数组就是 3.1 中的排序结果 $rank$ 数组。下面给出它的详细定义：

定义 4. 后缀数组 SA 是一个一维数组，它保存 $1 \ldots n$ 的某个排列 $SA[1], SA[2], \ldots, SA[n]$，并且保证 $Suffix(SA[i]) < Suffix(SA[i+1]), 1 \le i < n$。也就是将字符串 s 的 n 个后缀从小到大进行排序之后把排好序的后缀的开头位置顺次放入 SA 中。

理论上我们可以直接对 3.3.1 中构建的 M 字符串数组来进行排序，就可以得到 SA 数组。

但实际上这是不太有可行性的。对于一个超长字符串，以 10 亿长度为例，单个所占的空间约为 $1G$。假如根据 M 字符串数组进行后缀排序，光考虑 M 的存储空间，都将是爆炸性的不足的。

因此我们应该按照 3.1 中给出的倍增算法的思想对原字符串 s 进行排序来获得 SA 数组。

3.3.4 总结

因此，总结一下 FM Index 构建的过程。逻辑上应当按照上述步骤顺次执行。

但实际上，我们应当先执行最后一步来获得 SA 数组。然后再根据 SA 数组来构建出 BWT 的结果字符串 F 和字符串 L。最后根据 F 和 L 来构建出 C 字典与 Occ_buf。

3.4 精确匹配

精确匹配需要基于 3.3.2 中的字典 C 和 Occ_buf 数组。先介绍一个引理：

引理 1. *$Last\ First\ Mapping(LF)$：可以根据字符串 L 中的字符 c，将其映射到到字符串 F 中，得到字符 c 在字符串 F 中索引，即在字符串数组 M 的哪一行。过程如下：*

1. *找到字典 C 中字符 c 的值，即 $C[c]$。*
2. *找到 Occ_buf 中该行 c 的值，即 $Occ\_buf[row][c]$。*
3. *将它们相加，即得到字符 c 在字符串 F 中索引，即在字符串数组 M 的哪一行。*

概括起来，精确匹配的过程就是通过 Lemma.1 的方法不断的缩小搜索范围，最后得出待匹配字符串在 s 中的位置。

具体操作过程如下：

1. 将 $s[-1]$ 赋值给字符 c。
2. 令起始位置 sp 为 $C[c]+1$，这样会使得 $s[sp]$ 取到的字符刚好与 c 相同。
3. 令结束位置 ep 为 $C[c+1]+1$，这样会使得 $s[ep]$ 取到的字符刚好与 c 之后的那个字符相同。
4. 令 i 为 $len(s)-1$，作为循环变量。
5. 当 $sp < ep\ and\ i \geq 1$ 时，令字符 $c = s[i]$，$sp = LF(c, sp)$，$ep = LF(c, ep)$。
6. 循环结束时，sp 和 ep 之间的值，就是待匹配字符串的位置。

我们可以看到，整个匹配过程是一个从后往前匹配的过程。

前面 4 个步骤的意思很明确，就是初始化一下所要用到的变量，让 sp 和 ep 定位到最后一个字符和比最后一个字符大一点的字符之间。

之后在后面的循环中，只要两个区间指针 sp 和 ep 的大小关系满足条件，并且字符串 s 中的字母还没被匹配完，就一直不断地依赖 Lemma.1 来缩小这个区间。

直至全部字符串 s 匹配完或者区间大小关系发生错误。

4 实现思路

算法的总体思路大致为：

1. 初始化所有变量
2. 读入文件，并按照**分治思想**，将其分为若干个 1000w 个字符串为一组的短串
3. 对所有的短串做 **Suffix Array Sorting**
4. 对所有的短串进行 **BWT 编码**和建立 **FM Index**
5. 精确匹配各个短串，并精确匹配各个短串连接处的字符串
6. 销毁所有变量

具体的流程主要分为以下几个部分。

4.1 初始化模块

- **功能**

  在算法开始时初始化整个算法需要用到的全局变量；并在算法结束时安全地销毁所有的全局变量。

- **实现思路**

  首先大致计算一下整个算法需要开辟的空间大小。最大的测试文件中的字符串包含 10 亿个字符，c++ 中一个 char 变量占 1 个字节，于是有 $10^9\ bit \div 1024 \div 1024 \approx 954\ M$。而 c++ 中栈空间在 Linux 下默认为 $8M$，这显然是远远不够的。因此，需要在堆空间中开辟内存。

  在堆空间开辟数组，一般有两种方法：

  I. 使用 new 函数并人工维护，在程序结束时 delete。

  II. 使用 c++ 的智能指针

  我使用的是第一种方法。

- **代码**

```
/**
 * @description: 初始化所有的全局变量，因为很大，所以只能new，用堆空间
 * @param {void}
 * @return {void}
 */
void init() {
    // 初始化 S F L
    S = new char*[MAX_N];
    for (int i = 0; i < MAX_N; i ++) {
        S[i] = new char[MAX_LEN];
        memset(S[i], 0, MAX_LEN * sizeof(char));
    }
    GLOBAL_F = new char*[MAX_N];
    for (int i = 0; i < MAX_N; i ++) {
        GLOBAL_F[i] = new char[MAX_LEN];
```

```
16          memset(GLOBAL_F[i], 0, MAX_LEN * sizeof(char));
17      }
18      GLOBAL_L = new char*[MAX_N];
19      for (int i = 0; i < MAX_N; i ++) {
20          GLOBAL_L[i] = new char[MAX_LEN];
21          memset(GLOBAL_L[i], 0, MAX_LEN * sizeof(char));
22      }
23      // 初始化 sa
24      SA = new int*[MAX_N];
25      for (int i = 0; i < MAX_N; i ++) {
26          SA[i] = new int[MAX_LEN];
27          memset(SA[i], 0, MAX_LEN * sizeof(int));
28      }
29      // 初始化 C
30      for (int i = 0; i < MAX_N; i ++) {
31          map <char, int>* tmp = new map <char, int>;
32          GLOBAL_C[i] = tmp;
33      }
34      // 初始化 Occ_buf
35      for (int i = 0; i < MAX_N; i ++) {
36          map <int, vector <int> >* tmp = new map <int, vector <int> >;
37          GLOBAL_Occ_buf[i] = tmp;
38      }
39      // 初始化杂七杂八
40      wa = new int[MAX_LEN];
41      memset(wa, 0, MAX_LEN * sizeof(int));
42
43      wb = new int[MAX_LEN];
44      memset(wb, 0, MAX_LEN * sizeof(int));
45
46      wv = new int[MAX_LEN];
47      memset(wv, 0, MAX_LEN * sizeof(int));
48
49      Ws = new int[MAX_LEN];
50      memset(Ws, 0, MAX_LEN * sizeof(int));
51  }
52
53
54  /**
55   * @description: 安全销毁所有全局变量
56   * @param {void}
57   * @return {void}
58   */
59  void destory() {
60      // 谨慎起见倒着释放内存
61      // 释放乱七八糟
62      delete [] wa;
63      wa = NULL;
64
65      delete [] wb;
66      wb = NULL;
67
```

```
68      delete [] wv;
69      wv = NULL;
70
71      delete [] Ws;
72      Ws = NULL;
73      // 释放 Occ_buf
74      map <int, map <int, vector <int> >* > ::iterator multitr_1;
75      map <int, vector <int> >* tmp_1 = NULL;
76      for (multitr_1 = GLOBAL_Occ_buf.begin(); multitr_1 != GLOBAL_Occ_buf.
            end(); multitr_1 ++) {
77          tmp_1 = multitr_1->second;
78          delete tmp_1;
79          tmp_1 = NULL;
80      }
81      // 释放 C
82      map <int, map <char, int>* > ::iterator multitr_2;
83      map <char, int>* tmp_2 = NULL;
84      for (multitr_2 = GLOBAL_C.begin(); multitr_2 != GLOBAL_C.end();
            multitr_2 ++) {
85          tmp_2 = multitr_2->second;
86          delete tmp_2;
87          tmp_2 = NULL;
88      }
89      // 释放 SA
90      for (int i = 0; i < MAX_N; i ++) {
91          delete [] SA[i];
92          SA[i] = NULL;
93      }
94      delete [] SA;
95      SA = NULL;
96      // 释放 L F S
97      for (int i = 0; i < MAX_N; i ++) {
98          delete [] GLOBAL_L[i];
99          GLOBAL_L[i] = NULL;
100     }
101     delete [] GLOBAL_L;
102     GLOBAL_L = NULL;
103
104     for (int i = 0; i < MAX_N; i ++) {
105         delete [] GLOBAL_F[i];
106         GLOBAL_F[i] = NULL;
107     }
108     delete [] GLOBAL_F;
109     GLOBAL_F = NULL;
110
111     for (int i = 0; i < MAX_N; i ++) {
112         delete [] S[i];
113         S[i] = NULL;
114     }
115     delete [] S;
116     S = NULL;
117 }
```

4.2 文件读入模块

- **功能**

  根据给定的文件路径，读入文件中的内容。

- **实现思路**

  文件读入并没有特殊之处，唯一需要注意的地方在于要将其分块变成短串，以供后续使用。

- **代码**

```
/**
* @description: 逐个字符读入文件，并根据大小分割成较小的小串
* @param {string} file: 读入文件的文件名
* @return {void}
*/
void ReadFile(string file) {
    int count = 0;
    ifstream infile;
    infile.open(file);
    if(infile.is_open()) {
        char c;
        while((c = infile.get()) && c != EOF) {
            S[STR_NUM][count] = c;
            count ++;
            if (count == MAX_TRUE_LEN) {
                count = 0;
                STR_NUM ++;
            }
        }
        if (count) {
            STR_NUM ++;
        }
    }
    infile.close();
}
```

4.3 后缀排序模块

- **功能**

  对短串进行后缀排序。

- **实现思路**

  由于字符串可以转换为的 ASCII 码有限，因此这里使用了基数排序加快排序速度。并且为了减少不必要的开销，可以把某些交换、赋值操作用指针操作来替代。

- **代码**

```
/**
* @description: 计算后缀排序
* @param {int} idx: 选择后缀排序的小串编号
* @return {void}
```

```
*/
void CalSA(int idx)
{
    // 变量初始化
    char *r = S[idx];
    int *sa = SA[idx];
    int m = 512, n = strlen(r);
    r[n] = '$';
    n ++;
    memset(wa, 0, MAX_LEN * sizeof(int));
    memset(wb, 0, MAX_LEN * sizeof(int));
    memset(wv, 0, MAX_LEN * sizeof(int));
    memset(Ws, 0, MAX_LEN * sizeof(int));

    int i,j,p,*x=wa,*y=wb,*t;
    for(i=0; i<m; i++) Ws[i]=0;
    for(i=0; i<n; i++) Ws[x[i]=r[i]]++;//以字符的ascii码为下标
    for(i=1; i<m; i++) Ws[i]+=Ws[i-1];
    for(i=n-1; i>=0; i--) sa[--Ws[x[i]]]=i;
    for(j=1,p=1; p<n; j*=2,m=p)
    {
        for(p=0,i=n-j; i<n; i++) y[p++]=i;
        for(i=0; i<n; i++) if(sa[i]>=j) y[p++]=sa[i]-j;
        for(i=0; i<n; i++) wv[i]=x[y[i]];
        for(i=0; i<m; i++) Ws[i]=0;
        for(i=0; i<n; i++) Ws[wv[i]]++;
        for(i=1; i<m; i++) Ws[i]+=Ws[i-1];
        for(i=n-1; i>=0; i--) sa[--Ws[wv[i]]]=y[i];
        for(t=x,x=y,y=t,p=1,x[sa[0]]=0,i=1; i<n; i++)
                x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;
    }
    return;
}
```

4.4 BWT 编码与 FM Index 模块

- **功能**

  构建 BWT 编码以及 FM Index。

- **实现思路**

  实现思路与设计原理给出的相同，详情请参考 3.3 的算法。

- **代码**

```
/**
* @description: BWT编码
* @param {int} idx: 选择编码的小串编号
* @return {void}
*/
void BWTEncode(int idx) {
    // 后缀排序
    CalSA(idx);
```

```
9
10      // 初始化一些变量
11      char *F = GLOBAL_F[idx];
12      char *L = GLOBAL_L[idx];
13      map <char, int>* C = GLOBAL_C[idx];
14      map <int, vector <int> >* Occ_buf = GLOBAL_Occ_buf[idx];
15      int n = strlen(S[idx]);
16      char *r = S[idx];
17      int *sa = SA[idx];
18
19      // 生成F和L字符串
20      for (int i = 0; i < n; ++ i) {
21          F[i] = r[sa[i]];
22          if (sa[i] == 0) {
23              L[i] = '$';
24          }
25          else {
26              L[i] = r[sa[i]-1];
27          }
28      }
29
30      // 生成C数组，基因组特化
31      int count = 0;
32      int j = 0;
33      char gene[5] = "ACGT";
34      for (int i = 1; i < n; ++ i) {
35          if (F[i] == gene[j]) {
36              C->insert(pair<char, int>(gene[j], count));
37              ++ j;
38              ++ count;
39              continue;
40          }
41          ++ count;
42      }
43      // 生成Occ_buffer，按照论文，生成间隔为448行
44      vector <int> count_list(4);
45      int step = 448;
46      for (int i = 0; i < n; ++ i) {
47          if (L[i] == '$') {
48              continue;
49          }
50          ++ count_list[gene2rank[L[i]]];
51          if (i % step == 0) {
52              Occ_buf->insert(pair<int, vector<int> >(i, count_list));
53          }
54      }
55  }
56
57
58  /**
59  * @description: 得到小串idx编码后的L字符串中，第row行的c字符出现的次数
60  * @param {int} idx: 选择小串的编号
```

```
* @param {char} c: 想要找的字符c
* @param {int} row: 行号
* @return {int}
*/
int Occ(int idx, char c, int row) {
    char *L = GLOBAL_L[idx];
    map <int, vector <int> >* Occ_buf = GLOBAL_Occ_buf[idx];
    int step = 448;
    int check_point = int(row / step) * step;
    int count = Occ_buf->at(check_point)[gene2rank[c]];
    for (int i = check_point + 1; i <= row; ++ i) {
        if (L[i] == c) {
            ++ count;
        }
    }
    return count;
}
```

4.5 精确匹配模块

- **功能**

  根据输入的字符串来精确匹配短串内的字符串，并将匹配到的结果记录下来。

- **实现思路**

  实现思路与设计原理给出的相同，详情请参考 3.4 的算法。

- **代码**

```
/**
* @description: 在小串idx里边精确匹配P字符串
* @param {int} idx: 选择小串的编号
* @param {string} P: 要匹配的字符串
* @return {void}
*/
void ExactMatch(int idx, string P) {
    // 变量初始化
    int *sa = SA[idx];
    char *F = GLOBAL_F[idx];
    int n = strlen(S[idx]);
    map <char, int>* C = GLOBAL_C[idx];

    int i = P.length() - 1;
    char c = P[i];
    int sp = C->at(c) + 1;
    int ep;
    if (c == F[n - 1]) {
        ep =  n - 1;
    }
    else {
        ep = C->at(rank2gene[gene2rank[c] + 1]);
    }

```

```
    while (sp <= ep && i >= 1) {
        c = P[i - 1];
        sp = C->at(c) + Occ(idx, c, sp - 1) + 1;
        ep = C->at(c) + Occ(idx, c, ep);
        -- i;
    }

    for (int j = ep; j > sp - 1; -- j) {
        pos_list.push_back(pair<int, int>(idx, sa[j]));
    }
    return;
}
```

4.6 连接处字符串匹配模块

- **功能**

  根据输入的字符串来精确匹配短串之间连接处的字符串，并将匹配到的结果记录下来。

- **实现思路**

  记待匹配字符串 s 的长度为 n，左边的短串为 s_l，右边的短串为 s_r。

  生成新的拼接字符串 $s_{joint} = s_l[-(n-1):] + s_r[:n-1]$。

  最后在 s_joint 中进行匹配即可。

- **代码**

```
/**
 * @description: 在小串idx_l和idx_r的连接出匹配字符串P
 * @param {int} idx_l: 左边的小串编号
 * @param {int} idx_r: 右边的小串编号
 * @param {string} P: 要匹配的字符串
 * @return {void}
 */
void JointMatch(int idx_l, int idx_r, string P) {
    // 构造连接后的字符串
    int len_P = P.size(), len_l = strlen(S[idx_l]) - 1, len_r = strlen(S[
        idx_r]) - 1;   //  不计算最后一个 $
    int offset = len_l - len_P + 1;
    string s_l(S[idx_l] + offset, len_P - 1);
    string s_r;
    if (len_r < len_P) {
        s_r = string(S[idx_r], len_r);
    }
    else {
        s_r = string(S[idx_r], len_P - 1);
    }
    string s_joint = s_l + s_r;
    // 用 STL 的内置函数循环找子串，因为连接后的字符串的长度为 2 * len_P，
        因此寻找的复杂度不会很高
    size_t pos = s_joint.find(P, 0);
    while (pos != s_joint.npos) {
        pos_list.push_back(pair<int, int>(idx_l, pos + offset));
```

```
        pos = s_joint.find(P, pos + 1);
    }
    return;
}
```

4.7 用户界面交互模块

- **功能**

  让用户可以在 shell 界面中使用该算法，并且具有较强的鲁棒性。

- **实现思路**

  该部分主要的目的是增加鲁棒性，因此我做了以下几个方面的优化：

  1. 限制读入字符串的长度，并将其转换为大写格式
  2. 对读入的字符串进行一次筛查，若该字符串 s 中存在字符 c 没有在某短串中出现过，则说明 s 不可能在该短串中出现。

- **代码**

```
/**
* @description: 运行的主函数
* @param {void}
* @return {void}
*/
void run() {
    // 初始化变量
    init();

    // 读入文件名
    string file;
    while(file.empty()) {
        PRINT_INFO("Please Input File Name: ", false);
        cin >> file;
    }

    // 读入文件
    PRINT_INFO("Load Start!");
    auto start = system_clock::now();
    ReadFile(file);     // 读入
    auto end = system_clock::now();
    auto duration = duration_cast<microseconds>(end - start);
    cout <<  "花费了"
        << double(duration.count()) * microseconds::period::num /
            microseconds::period::den
        << "秒" << endl;
    PRINT_INFO("Load Finished!");

    // BWTEncode
    PRINT_INFO("BWTEncode Start!");
    start = system_clock::now();
    for (int i = 0; i < STR_NUM; i ++) {
```

```
32          BWTEncode(i);   // 给各个小串编码
33      }
34      end = system_clock::now();
35      duration = duration_cast<microseconds>(end - start);
36      cout <<  "花费了"
37          << double(duration.count()) * microseconds::period::num /
                microseconds::period::den
38          << "秒" << endl;
39      PRINT_INFO("BWTEncode Finished!");
40
41      // 精确匹配字符串
42      string P;
43      while (true) {
44          PRINT_INFO("Please Input String (Max 1k): ", false);
45          cin >> P;
46          // 检查输入的内容长度
47          if (P.size() > MAX_MATCH_LEN) {
48              PRINT_ERROR("Content is too long!");
49          }
50          // 将输入的内容变为大写
51          transform(P.begin(), P.end(), P.begin(), ::toupper);
52          // 判断是否退出
53          if (P == "EXIT") {
54              cout << "[INFO] Exit System!" << endl;
55              break;
56          }
57          // 判断输入的字符是否不合法
58          set <int> skip_set;                         // skip_set, 里边记录
                P 绝对不会出现的小串编号
59          set <char> char_s(P.begin(), P.end());      // 字符串集合, 可能这
                样子不是很快, 可以再改
60          for (int i = 0; i < STR_NUM; i ++) {
61              for (auto c : char_s) {
62                  if (GLOBAL_C[i]->find(c) == GLOBAL_C[i]->end()) {
63                      // 如果 C 中找不到 P 中的某个字符, 这说明该小串中绝对
                            不会出现 P, 于是加入skip_list。但反之不成立。
64                      skip_set.insert(i);
65                      break;
66                  }
67              }
68          }
69          // 清空上一轮的 pos_list 结果并释放内存
70          vector <pair<int, int> >().swap(pos_list);
71          // Matching
72          PRINT_INFO("Start Matching...");
73          start = system_clock::now();
74          // ExcatMatch
75          for (int i = 0; i < STR_NUM; i ++) {
76              // 判断该小串是否绝对匹配不到 P
77              if (skip_set.find(i) == skip_set.end()) {
78                  ExactMatch(i, P);   // 在各个小串里搜索
79              }
```

```
80          }
81          // JointMatch
82          for (int i = 1; i < STR_NUM; i ++) {
83              JointMatch(i - 1, i, P);
84          }
85          end = system_clock::now();
86          duration = duration_cast<microseconds>(end - start);
87          PRINT_INFO(string("Match Finished! Total Result: ") + to_string(
                pos_list.size()));
88          cout <<  "花费了"
89              << double(duration.count()) * microseconds::period::num /
                    microseconds::period::den
90              << "秒" << endl;
91          // 将结果写入文件
92          PRINT_WARNING("Writing result to file, please not turn off the
                power...");
93          WriteResultToFile();
94          PRINT_INFO("Write Finished!");
95      }
96      PRINT_WARNING("Exiting, please not turn off the power...");
97      // 销毁变量
98      destory();
99      PRINT_INFO("Exit Successfully!");
100 }
```

5 实验结果与性能测试

5.1 1 亿小数据集测试结果

构建索引时间约为 **26s** 左右。

对于正常长度的精确匹配过程能在 **0.1s** 内完成；对长字符串也能在 **1s** 内完成。

```
(base) ninglu_shao@dou-PowerEdge-R730:~/code$ ./a.out
[INFO] Please Input File Name: big.txt
[INFO] Load Start!
花费了0.914396秒
[INFO] Load Finished!
[INFO] BWTEncode Start!
花费了26.3966秒
[INFO] BWTEncode Finished!
[INFO] Please Input String (Max 1k): aaattcatca
[INFO] Start Matching...
[INFO] Match Finished! Total Result: 319
花费了0.029231秒
[WARNING] Writing result to file, please not turn off the power...
[INFO] Write Finished!
[INFO] Please Input String (Max 1k):
```

图 6: 1 亿小数据集测试结果

5.2 10 亿大数据集测试结果

构建索引时间约为 **200s** 左右。

对于正常长度的精确匹配过程能在 **0.3s** 内完成；对长字符串也能在 **1s** 内完成。

```
(base) ninglu_shao@dou-PowerEdge-R730:~/code$ ./a.out
[INFO] Please Input File Name: very_big.txt
[INFO] Load Start!
花费了7.54357秒
[INFO] Load Finished!
[INFO] BWTEncode Start!
花费了194.417秒
[INFO] BWTEncode Finished!
[INFO] Please Input String (Max 1k): accggtacgt
[INFO] Start Matching...
[INFO] Match Finished! Total Result: 98
花费了0.186366秒
[WARNING] Writing result to file, please not turn off the power...
[INFO] Write Finished!
[INFO] Please Input String (Max 1k): acgtgtgtagctactgaaaccacggtacgt
```

图 7: 10 亿小数据集测试结果

5.3 测试文件结果测试

一共有 3 个测试文件，每个文件的字符串长度为 20w。

3 个文件的匹配结果为，0 个、1 个、1 个。具体位置信息如下（从 0 开始计数）：

文件名	位置
search_SRR00001.txt	-
search_SRR163132.txt	423989716
search_SRR311115.txt	0

匹配的速度也非常的快，所有的测试都在 2s 左右完成。

```
[INFO] Please Input File: s_1.txt
[INFO] Start Matching...
[INFO] Match Finished! Total Result: 0
花费了1.54746秒
[WARNING] Writing result to file, please not turn off the power...
[INFO] Write Finished!
[INFO] Please Input File: s_2.txt
[INFO] Start Matching...
[INFO] Match Finished! Total Result: 1
花费了2.42495秒
[WARNING] Writing result to file, please not turn off the power...
[INFO] Write Finished!
[INFO] Please Input File: s_3.txt
[INFO] Start Matching...
[INFO] Match Finished! Total Result: 1
花费了2.37448秒
[WARNING] Writing result to file, please not turn off the power...
[INFO] Write Finished!
[INFO] Please Input File:
```

图 8: 测试结果