

基于 SA-IS 算法、BWT 与 FM-index 的字符串模式匹配

(数据结构与算法 II 大作业 实验报告)

中国人民大学 信息学院 崔冠宇 2018202147

注：请使用支持 C++17 或以上标准的编译器！

1 符号约定

- $\Sigma(S)$: 字符串 S 的**字母表** (alphabet), 不引起歧义时, 简记为 Σ ;
- $|S|$: 集合 S 的**基数** (即元素个数), 或字符串 S 的长度;
- $c_1 < c_2$: 字符 c_1 按**字典序**小于/先于字符 c_2 ;
- $T[0 \dots n-1]$: **原串/被匹配串**, $|T| = n$;
- $P[0 \dots m-1]$: **模式串/匹配串**, 一般而言 $|P| = m \ll n$;
- $BWT(S)$: 字符串 S 的**BWT 结果**;
- $SA(S)$: 字符串 S 的**后缀数组** (suffix array)。

2 功能说明

本程序使用了 **SA-IS** 算法 [1] (一种线性时间后缀数组构建算法) 优化后的 **BWT** 算法 [2] 以及 **FM-index** 技术 [3], 实现了一个带有 $O(m)$ 时间字符串模式匹配功能的简单的交互式程序。

用户进入程序后, 可以输入各种命令, 从指定文件/终端加载被匹配串, 程序自动构建 BWT 与 FM 索引, 之后用户可以随意匹配子串。除此之外, 程序还支持多次加载不同文件, 而不必每次关闭程序再次打开程序以加载不同文件。

3 输入/输出说明

本程序是交互式的, 支持 **HELP** (帮助)、**FILELOAD** (从文件加载待匹配串)、**STRLOAD** (从终端加载待匹配串)、**BWT** (显示待匹配串的 BWT 结果)、**FILEMATCH** (从文件加载字符串匹配)、**STRMATCH** (从终端加载字符串匹配) 以及 **QUIT** (退出程序) 等命令。以下分别介绍各功能的输入/输出:

1. **HELP** 用来提示用户各个命令的含义, 不需要输入参数, 输出为各命令的用法;
2. **FILELOAD** 用来从文件加载被匹配的字符串 T , 输入参数为文件的绝对路径, 输出为加载成功与否以及所用时间 (允许覆盖加载);

3. **STRLOAD** 用来从终端读取加载被匹配的字符串 T , 输入参数为字符串 T , 输出为加载成功与否以及所用时间 (允许覆盖加载);
4. **BWT** 用来输出被匹配串 T 的 **BWT** 结果, 不需要输入参数, 输出为 T 的 **BWT** 结果 $BWT(T)$ (要求此时已经加载索引);
5. **FILEMATCH** 用来从文件加载子串进行匹配, 输入参数为模式串文件所在绝对路径, 输出为匹配成功的次数、各子串的位置以及子串及其前后若干字符 (要求此时已经加载索引);
6. **STRMATCH** 用来从终端加载子串进行匹配, 输入参数为模式串 P , 输出为匹配成功的次数、各子串的位置以及子串及其前后若干字符 (要求此时已经加载索引);
7. **QUIT** 用来退出程序, 不需要输入参数, 没有输出。

具体的程序运行截图和说明可参看5。

4 设计原理说明

由于 **SA-IS** 算法是一种线性时间后缀数组的构建方法 [1], 是对 **BWT** 算法中后缀数组构建步骤的一种改进; 而 **FM-index** 技术是在 **BWT** 算法的基础上附加了一些数据结构, 用来辅助查询 [4]; 所以本问题需要重点阐述的中心原理是 **BWT** 算法及其改进, 在此基础上再扩展介绍其他两种技术。

4.1 BWT 算法

BWT 算法是一种用于数据压缩的变换算法¹, 分为压缩变换和解压变换这两个方向相反的操作。由于后续建立 **FM-index** 进行字符串查询时只需要正向 **BWT**, 所以这里只说明正向 **BW** 变换的原理。

Algorithm: 压缩变换

以字符串 $T = \text{abaaba}$ 为例来说明算法的各步骤。

- (C1) 【添加结尾符】为了方便分辨循环移位后字符串的结尾, 在原字符串 T 的结尾加一个特殊的结尾符得到 T' 。要求结尾符不能在原字符串 T 中出现, 且它的字典序比 $\Sigma(T)$ 中各字符的字典序都小。这里将结尾符记作 '\$' (并不表示结尾符一定是 '\$')。例:

$$T' = \text{abaaba\$}$$

- (C2) 【循环移位构成矩阵】将 T' 依次循环左移形成的串构成一个矩阵 \mathbf{M} , 即 $\mathbf{M}_i = \text{ROL}(T', i)$, ($i = 0, 1, \dots, n =$

¹**BWT** 算法本身并不压缩数据, 而是将数据可逆地变为压缩效率更高的排列。

$|T'| - 1$)。同时，在 \mathbf{M} 每行前标记行号。例：

```

0 abaaba$
1 baaba$a
2 aaba$ab
M = 3 aba$aba
4 ba$abaa
5 a$abaab
6 $abaaba

```

(C3) 【矩阵各行按字典序排序】将 \mathbf{M} 各行字符串按字典序排序得到 \mathbf{M}' 。例：

```

6 $abaaba
5 a$abaab
2 aaba$ab
M' = 3 aba$aba
0 abaaba$
4 ba$abaa
1 baaba$a

```

(C4) 【取最后一列】 $BWT(T')$ 即为 \mathbf{M}' 的最后一列 $(\mathbf{M}'_{in})_{1 \times n}$ 。例：

$$BWT(T') = abba$aa$$

4.2 后缀数组及其构建方法

将 T' 、 \mathbf{M}' 前面的标号（先记作 $SA(T')$ ）与 $BWT(T')$ 放在一起观察：

```

0123456
T' = abaaba$
SA(T') = 6523041
BWT(T') = abba$aa

```

可以观察到以下规律：

$$BWT(T')[i] = \begin{cases} T'[SA(T')[i] - 1], & SA(T')[i] > 0 \\ \$, & SA(T')[i] = 0 \end{cases}$$

这种现象并非偶然，分析原因可以发现，构建 \mathbf{M}' 时对原字符串进行了循环移位和排序， \mathbf{M}' 的最后一列与排序前行号的关系恰好满足上面的关系，这也就说明了上面公式的正确性。

由于 \mathbf{M}' 中每行 \$ 前的内容正好就是 T' 的某个后缀 $T'[SA(T')[i]:]$ ，所以数组 $SA(T')$ 也被称为后缀数组。通过上面的讨论可以发现，如果有了后缀数组，则可以以 $O(n)$ ($n = |T'|$ ，下同) 的时间构建 $BWT(T')$ ，那么接下来一个很自然的问题就是：如何尽快地构建后缀数组 $SA(T')$ ？

经过自己的思考和阅读参考文献，我总共找到了三类方法，它们的时间复杂度由高到低。下面分别介绍三类构建后缀数组的方法。

4.2.1 平凡的 $O(n^2 \log n)$ 的方法

构建后缀数组最平凡的方法如下：

Algorithm 1 平凡后缀数组构建 Naive SA

Input: 含结束符的字符串 T'

Output: T' 的后缀数组 $SA(T')$

```

1: suffixes = []
2: for i = 0 to |T'| - 1 do
3:   suffixes.append((i, T'[i:]))
4: sort(suffixes) lexicographically
5: return [suffixes[i][0] for i = 0 to |T'|]
```

假定 $|T'| = n$ ，算法最耗时的部分是排序。如果选择最优的基于比较的排序，需要进行 $\Theta(n \log n)$ 次比较，而每次进行后缀间的比较时，耗时 $O(n)$ ，所以总时间复杂度 $T(n) = O(n^2 \log n)$ 。

4.2.2 更快的后缀数组构建算法——从 $O(n^2 \log n)$ 到 $O(n \log n)$ 再到 $O(n)$

虽然上面给出的 **Naive SA** 算法是一个有效的（多项式时间）算法，但是 $O(n^2 \log n)$ 的复杂度在 n 超大时还是很低效。据我的实验，当 n 约为 50 万时，构建后缀数组的时间已经不能承受，所以必须换用更快的算法。

参考文献 [2] 也考虑到了这个问题，于是作者提出了一种变形后的快速排序算法，平均情况下可以在 $O(n \log n)$ 的时间建立后缀数组。下面还是以之前的字符串 $T = \text{abaaba}$ 为例 ($n = |T| = 6$)，简要介绍作者提出的算法：

(Q1) **【扩展原字符串】** 设机器字长相当于 k 个字符²，在 T 后面增加 k 个结束符得到 T' 。例（为简便起见，这里设 $k = 2$ ）：

$$T' = \text{abaaba}\$ \$$$

(Q2) **【构造字 (word) 数组】** 构造包含 n 个字 (word) 的数组 W ，其中 $W[i] (i = 0, 1, \dots, n)$ 包含 $T'[i \dots i + k - 1]$ 。将

²一个 `char` 占用一个字节。如果仅考虑英文字符，则在 32 位机器上， $k = 4$ ；在 64 位机器上， $k = 8$ 。

字符“打包”成字，就将字符间的多次比较转换成了整数间的一次比较。例：

$$W = \begin{cases} W[0] = ab \\ W[1] = ba \\ W[2] = aa \\ W[3] = ab \\ W[4] = ba \\ W[5] = a\$ \\ W[6] = \$\$ \end{cases}$$

(Q3) 【初始化后缀数组】初始化后缀数组 $V[0 \dots n-1]$ ，使得 $V[i] = i$ 。最终若 $V[i] = j$ ，则表示所有后缀中字典序第 i 的后缀是以 $T'[j]$ 开头的后缀。即：

$$V[0] = 0, V[1] = 1, V[2] = 2, V[3] = 3, V[4] = 4, V[5] = 5, V[6] = 6$$

(Q4) 【基数排序】将各 $V[i]$ 进行基数排序，排序关键字是各后缀的前两个字符。例（其中 $V'[3]$ 、 $V'[4]$ 以及 $V'[5]$ 、 $V'[6]$ 的排序关键字相同）：

$$V'[0] = 6, V'[1] = 5, V'[2] = 2, V'[3] = 0, V'[4] = 3, V'[5] = 1, V'[6] = 4$$

(Q5) 【迭代】对字母表 (alphabet) Σ 中的每个字符 ch 迭代操作（本例中 $\Sigma = \{\$, a, b\}$ ）：

(Q5-1) 【对以字符 ch 开头的后缀进行快速排序】对每个 $ch' \in \Sigma$ ，对 V 中对应于开头是 ch 及 ch' 的后缀的元素 $V[i]$ 应用快速排序。在快速排序的实现中，通过访问数组 W 来比较 V 中的两个元素。在递归过程中，还需要动态记录它们已经比较过的相等的字符个数，这些字符就不需要再次比较了。经过此步骤后，所有以 ch 为首字符的后缀均已到位。

(Q5-2) 【更新排序关键字】对 $V[i]$ 中每个对应后缀开头是 ch 的元素（即 $T[V[i]] = ch$ ），设置 $W[V[i]]$ 的高位为 ch （与之前一样不变），低位为 i （即 Q2 中紧接着的 $k-1$ 个字符）。这样设置 $W[V[i]]$ 的新值保持原有顺序，但使得它的值与其它 $W[j]$ 各不相同。

复杂度分析：

- (Q1) 中扩展原字符串需要 $O(k)$ 的时间，而机器字长 k 是常数，所以时间为 $O(1)$ ；
- (Q2) 构建字数组的时间是 $O(nk) = O(n)$ 的；
- (Q3) 初始化后缀数组的时间显然是 $O(n)$ 的；
- (Q4) 基数排序是 $O(n)$ 的时间；

- 而 (Q5) 由于涉及到快速排序, 情况比较复杂。快速排序的**平均**总时间是 $\sum_{i=1}^{|\Sigma|} \sum_{j=1}^{|\Sigma|} O(n_{ij} \log n_{ij})$ (其中 n_{ij} 是开头两个字符为 $\Sigma(i)$ 与 $\Sigma(j)$ 的 $|T'|$ 的后缀数目, 故 $\sum_{i=1}^{|\Sigma|} \sum_{j=1}^{|\Sigma|} n_{ij} = n$)。当各 n_{ij} 占比差不多时, 平均总时间为 $|\Sigma|^2 O(\frac{n}{|\Sigma|^2} \log(\frac{n}{|\Sigma|^2})) = O(n \log \frac{n}{|\Sigma|^2}) = O(n \log n)$; 而当有某 n_{ij} 占绝大多数时, 平均总时间也为 $O(n \log n)$ 。但遇到快速排序的最坏情况时, 快速排序时间将退化到 $O(n^2)$, 此时最坏时间为 $O(n^2)$ 。

以上各步骤是串行的, 所以可以得到下面的结论:

作者改进的后缀数组构建算法在平均情况下的时间复杂度为 $O(n \log n)$, 最坏情况下的时间复杂度为 $O(n^2)$ 。

但这仍不是最好的结果, 因为还是有可能碰到最坏情况, 此时算法的效率仍然较低。

在后缀数组构建问题被提出后, 有不少学者在做此类研究, 试图降低算法的复杂性。比如 Manber 和 Myers[5] 提出了 $O(n \log n)$ 的算法, 但实际效率不高。N. Jesper Larsson [6] 于 2007 年也提出了一种 $O(n \log n)$ 的算法, 实践中的效率有了较大提升。

经过检索, 我发现了几篇有关线性时间内后缀数组构建方法的论文:

- Kärkkäinen 等人于 2006 年提出了一种名为 **DC3** 的算法, 能在 $O(n)$ 的时间内解决后缀数组构建问题 [7]。想要加速算法, 一个根本的改进是减少后缀比较的时间。

比较两个后缀 $S[i:]$ 和 $S[j:]$ 可以先比较第一个字符, 然后 (如果相等的话) 就转化为比较 $s[i+1:]$ 和 $s[j+1:]$ 。

这启发我们可以将所有 $i \equiv 0 \pmod 3$ 和 $i \equiv 1 \pmod 3$ 的后缀排序, 之后对任意两个后缀的比较都可以在常数时间内转化到比较这两种后缀的比较上, 而这两种后缀只占所有后缀的 $2/3$, 可以递归处理。

算法框架:

- 基数排序并将每个字符转化为其 rank。
- 令

$$T_0 = [(T[3i], T[3i+1], T[3i+2])](i = 0, 1, \dots)$$

$$T_1 = [(T[3i+1], T[3i+2], T[3i+3])](i = 0, 1, \dots)$$

$$T_2 = [(T[3i+2], T[3i+3], T[3i+4])](i = 0, 1, \dots)$$

(以上均为三元组的序列)

- 递归将 $[T_0, T_1]$ (即 T_0 后面接 T_1) 排序。
- 基数排序 T_2
- 合并 SA_0 、 SA_1 和 SA_2 得到 SA 。

算法的时间复杂度为 $T(n) = T(\frac{2}{3}n) + O(n) = O(n)$ 。

DC3 算法也有一个很明显的劣势：它的时间复杂度虽然是 $O(n)$ 的，但是常数系数很大，有时甚至劣于常数系数为 7-8 的 $O(n \log n)$ 的倍增算法。除此之外，**DC3** 算法的代码也相当难写，因此我不打算使用这种算法构建后缀数组。

- Nong G. 等人于 2010 年提出了两种高效算法，使得后缀数组构建能在线性时间内完成 [1]。其中一种算法被称为 **SA-IS** 算法，它具有 $O(n)$ 的最坏情况时间复杂度， $\max\{2n, 4|\Sigma|\}$ 的额外空间。

这个算法主要使用了“诱导排序”(induced sorting) 的概念，通过引入两种后缀类型 (S 型和 L 型) 和 LMS (LeftMost S-type) 子串这个概念，通过它们的一些性质快速构建后缀数组 SA 。

下面简单介绍一下论文引入的新概念：

- L 型与 S 型后缀：当 $S[i:] < S[i+1:]$ 时，称 $S[i:]$ 是 S 型后缀，否则是 L 型后缀；
- * 型字符与 LMS 子串：* 型字符是一连串 S 型后缀中最左边的一个字符，相邻两个 * 型字符之间的部分称为 LMS 子串；

SA-IS 算法的框架如下：

```
1 SA-IS(S):
2     t = bool []
3     S1 = int []
4     P = int []
5     bucket = int []
6
7     扫描倒序字符串确定每一个后缀的类型 -> t
8     扫描 t 数组确定所有的 LMS 子串 -> P
9     对所有的 LMS 子串进行诱导排序
10    对每一个 LMS 子串重新命名，生成新的串 S1
11
12    if S1 中的每一个字符都不一样：
13        直接计算 SA1
14    else
15        SA1 = SA-IS(S1) # 递归计算 SA1
16
17    利用 SA1 来进行诱导排序，计算 SA
18    return SA
```

根据 LMS 子串的定义，可以很容易的证明：除去结尾字符形成的 LMS 子串之外，其它 LMS 子串的长度大于 2。因此，字符串 S 中的 LMS 子串的数目不会超过 $\lceil |S|/2 \rceil$ ，于是算法的复杂度 $T(n) = T(\lceil n/2 \rceil) + \Theta(n) = \Theta(n)$ 。

SA-IS 算法更加具体具体的描述可以参看原论文 [1]，以及这个博客 [9]。

Yuta Mori 根据 Nong G. 等人的工作实现了其中的一种线性时间算法 **SA-IS**，并将其开源在网络上 [8]。为了最大程度上加速程序的效率，我选择了 Yuta 实现的 **sais** 库进行后缀数组和 $BWT(T')$ 的构建。

4.3 FM-index

FM 索引是在 **BW 变换**的基础上增加了部分数据结构，使得子串查询的时间复杂度能被降低到 $O(m)$ (其中 $m = |P|$ 是模式串的长度)。

4.3.1 LF-Mapping

在具体描述 **FM-index** 之前，先来关注一个有趣的现象。观察上面例子中 **M'** 的第一列与最后一列（为了方便观察，给原串加上下标）：

$$\begin{array}{l}
 T' = a_0b_0a_1a_2b_1a_3\$ \\
 \\
 \$abaaba_3 \\
 a_3\$abaab_1 \\
 a_1aba\$ab_0 \\
 \mathbf{M}' = \begin{array}{l} a_2ba\$aba_1 \\ a_0baaba\$ \\ b_1a\$abaa_2 \\ b_0aaba\$a_0 \end{array}
 \end{array}$$

可以发现，在 **M'** 的第一列 F 和最后一列 L 中，各字母的相对顺序保持不变。这现象并非偶然，由于矩阵 **M'** 是将字符串循环移位并排序得到的，所以前后两列在原串内本就相邻，于是自然有这性质。

如果我们改变编号方式，使得前后两列各字母的编号从上至下递增，不仅更符合我们的直觉，还有利于之后的子串定位：

$$\begin{array}{l}
 T' = a_3b_1a_1a_2b_0a_0\$ \\
 \\
 \$abaaba_0 \\
 a_0\$abaab_0 \\
 a_1aba\$ab_1 \\
 \mathbf{M}' = \begin{array}{l} a_2ba\$aba_1 \\ a_3baaba\$ \\ b_0a\$abaa_2 \\ b_1aaba\$a_3 \end{array} \\
 \\
 SA(T') = 6523041
 \end{array}$$

在上面的编号方式中，由于 **M'** 是按照字典序排序的，因此第一列中相同的字符会凑在一起。假如我们看到最后一列的字符 b_1 ，想判断 b_1 在第一列的下标位置，那我们只需要维护各个字符的起始下标（记作 **FCharRange**），然后加上所需字符的编号即可。就如上例中， $\$$ 的起始下标为 0， a 的起始下标是 1，而 b 的起始下标为 5，于是 b_1 所

在的下标就为 **b** 的起始下标 5 再加上编号 1，结果为 6。**M'** 的这种通过最后一列字符定位到第一列下标的性质被称为 **LF-Mapping**。

4.3.2 定位子串

现在就到了本问题的核心，如何利用上述结构快速定位子串？由于构建矩阵 **M'** 时使用的是后缀，因此查找子串时也要从后向前进行。下面以在上面的 T' 中查询 $P_1 = \text{aba}$ （查询成功）和 $P_2 = \text{bba}$ （查询失败）为例，说明查询子串的原理：

查询过程中需要动态维护一个范围 $[a, b)$ ，表示目前正在查询的子串后缀在后缀矩阵 **M'** 的范围。

1. 查询存在的子串 $P_1 = \text{aba}$ ：

- (a) 先查询 **aba** 中最后的字符 **a**：可以从 **FCharRange** 查得以 **a** 开头的后缀范围为 $[1, 5)$ 。
- (b) 再查询 **aba** 中倒数第二个字符 **b**：从上面查询的范围 $[1, 5)$ 中，**扫描**找出最后一列为 **b** 的行。在本例中为 $\{1, 2\}$ ，对应字符是 b_0 与 b_1 。将它们用 **LF-Mapping** 映射到第一列，得到范围 $[5, 7)$ （一定会形成范围，因为 **M'** 具有前后两列各字符相对顺序一致的性质）。
- (c) 再查询 **aba** 中倒数第三个字符 **a**：从上面得到的范围 $[5, 7)$ 中，**扫描**找出最后一列为 **a** 的行。在本例中为 $\{5, 6\}$ ，对应字符为 a_2 和 a_3 。将它们用 **LF-Mapping** 映射到第一列，得到范围 $[3, 5)$ 。
- (d) 查询至此已经耗尽了模式串 P_1 ，于是在 $SA(T')$ 中查得 **M'** 中行范围为 $[3, 5)$ 的后缀为 $\{3, 0\}$ 号，说明在 $T'[3]$ 和 $T'[0]$ 处匹配到了子串。

2. 查询不存在的子串 $P_2 = \text{baa}$ ：

- (a) 前两步与查询 P_1 相同。
- (b) 查询 **bba** 中倒数第三个字符 **b**：从得到的范围 $[5, 7)$ 中，**扫描**找出最后一列为 **b** 的字符。在本例中这样的行不存在，说明 T' 中不存在这样的模式串 P_2 。

4.3.3 优化时间

1. 查询第一列字符范围的优化：

在构建后缀数组 SA 时，我们还同时记录了字母表 **alphabet** (`std::set<char>` 类型) 以及字符-字符在字母表中顺序对 **charOrder** (`std::unordered_map<char, size_t>` 类型)。在子串查询过程中，第一步查询模式串最后一个字符 c 的行范围时，可以用 $O(1)$ 的时间查得 c 在字符表中的顺序 **charOrder** $[c]$ 。然后再以其为下标，从 **FCharRange** (`std::vector<size_t, size_t>` 类型) 中以 $O(1)$ 时间查询到范围。

2. 从范围中快速确定最后一列目标字符的范围：

在上面的示例中，我们在给定范围中定位最后一列为所需字符的行时，采用了按顺序**扫描**的方法，但是这种定位方法查询一个字符最坏情况需要扫描 n 个元素，时间复杂度为 $O(n)$ ，这是不能忍受的代价，因此必须要进行优化。我们可以采用空间换时间的方法，引入 **LCheckPoints** 矩阵。**LCheckPoints** 是一个 $n \times |\Sigma|$ 的矩

阵，它的某一行（按字符顺序排列）的第 i 行代表一个字符截止 $BWT(T')[i]$ 出现的总次数。如在上面的例子中：

$$T' = a_3b_1a_1a_2b_0a_0\$$$

	a	b
\$abaaba ₀	1	0
a ₀ \$abaab ₀	1	1
a ₁ aba\$ab ₁	1	2
a ₂ ba\$aba ₁	2	2
a ₃ baaba\$	2	2
b ₀ a\$abaa ₂	3	2
b ₁ aaba\$a ₃	4	2

在上面查询 $P_1 = aba$ 的过程中，已经定位到结尾的 **a** 出现于 $[1, 5)$ ，需要从中找到最后一列为 **b** 的行，于是可以利用 **LCheckPoints** 的 **b** 列的信息， $LCheckPoints['b'][0] = 0$ ， $LCheckPoints['b'][4] = 2$ ，因此可以断定 $[1, 5)$ 行行尾出现了 b_0 与 b_1 ，再利用 **LF-Mapping** 性质和 **FCharRange** 的信息以 $O(1)$ 的时间计算出下一个范围。

4.3.4 优化空间

1. 压缩矩阵 M' ：

在匹配子串时只与 M' 的第一列与最后一列有关，用不到中间部分，于是可以不保存 M' 的中间 $n - 2$ 列。这一措施将 M' 的空间占用由 n^2 字节（设一个字符占一个字节）降为 $2n$ 字节。

2. 压缩第一列：

由于第一行是按字典序排好的，相同字符会聚在一起，于是 **FCharRange** 只需要保留各字符出现的范围即可。这一措施将 M' 第一列的空间由 n 字节降为了占用 $|\Sigma|(1 + 2 \times \text{sizeof}(\text{index_type}))$ 字节。

3. 压缩后缀数组：

许多人认为后缀数组几乎是随机的、无序的，因此几乎是不可压缩的。但是 [4] 指出，后缀数组 $SA(T')$ 也是可以压缩的。不同于 **LCheckPoints** 固定下标间隔保存， $SA(T')$ 是**固定值间隔**保存，即只保存值为 $0, 0 + gap, 0 + 2 \times gap$ 等的元素。设保留比为 α ，于是压缩后的后缀数组剩余 αn 个元素，共占用 $2\alpha n \times \text{sizeof}(\text{index_type})$ 个字节（2 倍是因为需要存储它在完整的 SA 数组中的坐标）。在最后确定查询结果所在位置时，若对应后缀数组元素没有被保留，只需要多次使用 **LF-Mapping**，直到找到被保留的 $SA(T')$ 的元素，取出它的值，然后加上进行 **LF-Mapping** 的次数，即为没有保存的值。参考文献 [4] 称，在**固定值间隔**保存的前提之下，查找时间仍为 $O(1)$ 。

4. 压缩 **LCheckPoints**：

LCheckPoints 是一个 $n \times |\Sigma|$ 的矩阵，因此所需空间是 $n \times |\Sigma| \times \text{sizeof}(\text{index_type})$ 字节。根据 [4] 的提示，**LCheckPoints** 可以仅保存部分行，并以稀疏矩阵的方式存储。对于每一字符对应的列，按照固定间隔保留元素。

设保留比为 β , 于是压缩后的 `LCheckPoints` 将是 $(n\beta) \times |\Sigma|$ 的稀疏矩阵, 占用 $(n\beta) \times |\Sigma| \times \text{sizeof}(\text{index_type})$ 字节。在固定间隔保留的条件下, 若对应下标的元素没有保存, 只要从最近的保留点开始扫描到所需的位置, 然后通过保留点的值加减扫描时遇到的模式串当前字符的数目, 即可得到所需值。[4] 称, 在固定间隔的前提下, 查找时间仍为 $O(1)$ 。

4.4 程序结构说明

本项目的结构如下:

- `/sais-lite-2.4.1`: 目录。内容是从 [8] 下载解压得到的 Yuta Mori 实现的 **SA-IS** 算法库。构建后缀数组时只需要 `#include "sais.h"` (C) 或 `#include "sais.hxx"` (C++), 然后在程序内调用 `sais` (C) 或 `saisxx` (C++s) 函数即可。
- `BWT_FM.h`: 文件。内部定义了 `BWT_FM` 类, 对外提供 `bwt` 以及 `displayPatternQuery` 两个函数, 前者用于输出 *BW* 变换后的字符串, 后者用于查询子串位置并显示结果。
- `BWT_FM.cpp`: 文件。内部是 `BWT_FM` 类的各函数的实现。
- `main.cpp`: 文件。内部是主交互界面的实现以及各功能的调用。

4.5 程序复杂度分析

4.5.1 时间复杂度

设原串长 $|T| = n$, 模式串长 $|P| = m$ 。

- 加载 (`FILELOAD/STRLOAD`) 功能:
 1. 首先读入字符串: 时间复杂度为 $O(n)$ 。
 2. 调用 `saisxx` 以构建后缀数组: 时间复杂度为 $O(n)[8]$ 。
 3. 在构建后缀数组同时, 判断字母表是否有该字母, 若没有则向字母表插入: 由于字母表是 `set` 类型, 根据 [10], `set` 查找和插入元素的时间都为 $O(\log k)$ (其中 k 表示容器内已有的元素个数)。在构建后缀数组的过程中, 共进行了 n 次查询和 $|\Sigma|$ 次插入, 因此该步骤时间复杂度为 $O(n|\Sigma|) + O(\sum_{i=1}^{|\Sigma|} \log |\Sigma|) = O(|\Sigma|n + |\Sigma| \log |\Sigma|)$ 。由于 $|\Sigma|$ 为常数, 所以该步骤时间复杂度为 $O(n)$ 。
 4. 计算 `charOrder` 字符-字符在字母表的顺序: 由于 `charOrder` 是 `unordered_map<char, size_t>` 类型的, 根据 [10], 插入元素的平均时间复杂度为 $O(1)$, 最坏情况时间复杂度 $O(k)$ (其中 k 是容器内已有元素个数)。同时因为共进行了 $|\Sigma|$ 次插入, 所以平均时间为 $O(|\Sigma|)$, 最坏情况为 $\sum_{k=1}^{|\Sigma|} O(k) = O(|\Sigma|^2)$ 。由于 $|\Sigma|$ 为常数, 所以该步骤时间复杂度为 $O(1)$ 。
 5. 计算第一列各字母范围 `FCharRange`: 由于算法是从前向后扫描第一列的字符, 统计各字符出现范围, 于是该步骤的时间复杂度为 $O(n)$ 。

6. 根据后缀数组 $SA(T')$ 计算 $BWT(T')$: 由于需要对后缀数组进行扫描然后得到 $BWT(T')$, 所以时间复杂度为 $O(n)$ 。
7. 计算最后一列的 **LCheckPoints**: 由于需要对 $BWT(T')$ 进行扫描, 每个位置要对字母表中所有字母进行计数, 于是时间复杂度为 $O(|\Sigma|n) = O(n)$ 。

综上, 本功能的时间复杂度为 $O(n)$ 。

- **BWT 功能:**

由于加载部分已经预先算出 $BWT(T')$ 串, 所以没有单独的计算开销, 仅有输出开销。

- **匹配 (FILEMATCH/STRMATCH) 功能:**

匹配时将模式串 P ($|P| = m$) 从后往前扫描, 根据上面的优化时间/优化空间部分, 每次能以 $O(1)$ 时间计算出新范围, 于是匹配功能的时间复杂度为 $O(m)$ 。

除此之外, 为了方便用户查看显示结果, 程序还会对搜索结果 (位置对应的下标) 进行排序, 若有 k 个查询结果, 于是排序所需时间为 $O(k \log k)$ 。所以考虑排序的匹配功能的总时间复杂度为 $O(m + k \log k)$ 。

4.5.2 空间复杂度

1. 原串 **original**: `std::string` 类型。为了方便输出查询结果, 程序会保留原串 **original**, 共占用 $n \times \text{sizeof}(\text{char})$ 字节, 空间复杂度为 $O(n)$ 。
2. $BWT(T')$ 串 **BWTstr**: `std::string` 类型。为了方便进行 **LF-Mapping** 和输出查询结果, 程序会保留 **BWTstr**, 共占用 $n \times \text{sizeof}(\text{char})$ 字节, 空间复杂度为 $O(n)$ 。
3. 字符表 **alphabet**: `std::set<char>` 类型。存储了原串的字符表, 共占用 $|\Sigma| \times \text{sizeof}(\text{char})$ 字节, 空间复杂度为 $O(|\Sigma|) = O(1)$ 。
4. 字符-字符顺序映射 **charOrder**: `std::unordered_map<char, size_t>` 类型。用于快速获得字符在字符表中的顺序, 共占用 $|\Sigma|(\text{sizeof}(\text{char}) + \text{sizeof}(\text{index_type}))$ 字节, 空间复杂度为 $O(|\Sigma|) = O(1)$ 。
5. 第一列的字符范围 **FCharRange**: 由于它存储了每个字符的坐标范围 (下标对), 所以总共占用空间为 $|\Sigma|(\text{sizeof}(\text{char}) + 2 \times \text{sizeof}(\text{index_type}))$ 字节, 空间复杂度为 $O(|\Sigma|) = O(1)$ 。
6. 后缀数组 **suffixArray**: 根据上面的讨论, 需要将后缀数组按等值间隔保留, 因此需要同时保留原下标和数组值。假设保留元素比为 α , 于是压缩后的后缀数组占用内存为 $2\alpha n \times \text{sizeof}(\text{index_type})$ 字节, 空间复杂度为 $O(n)$ 。
7. 最后一列的检查点 **LCheckPoints**: 它是一个 $n \times |\Sigma|$ 矩阵。它的第 i 行存储了截至 $BWT(T')[i]$ 各字符累计出现次数。由于在上面讨论过可以压缩存储 **LCheckPoints**, 设保留比为 β , 由于可以利用整除运算直接计算出压缩后的下标, 所以不必专门保存下标, 因此共需要 $\beta(n|\Sigma| \times \text{sizeof}(\text{index_type}))$ 字节, 空间复杂度为 $O(|\Sigma|n) = O(n)$ 。
8. **saisxx** 函数工作空间: 根据论文 [1] 中的讨论, **SA-IS** 算法的工作空间为 $\max\{2n, 4|\Sigma|\} = O(n)$ 。

综上，整个程序的辅助空间复杂度为 $O(n)$ 。

5 操作说明与实验截图

本机的配置是：MacBook Pro (15-inch, 2018), 2.6 GHz Intel Core i7, 16 GB 2400 MHz DDR4, macOS Mojave 10.14.6, 在 `/Users/CuiGuanyu/Desktop/算法课大作业/codes/` 目录下放置有源代码文件。

1. 编译、运行程序：

使用下列 Shell（我使用的 Shell 是 zsh）命令来编译运行程序（注意开启了 -O3 优化）：

```
1 $ cd "/Users/CuiGuanyu/Desktop/算法课大作业/codes/" && g++ -std=c++17 -O3 main.cpp BWT_FM.cpp -o main && "/Users/CuiGuanyu/Desktop/算法课大作业/codes/"main
```

演示截图如下：



```
cd "/Users/CuiGuanyu/Desktop/算法课大作业/codes/" && g++ -std=c++17
Last login: Fri Dec  4 11:00:25 on ttys000
(base) CuiGuanyu@localhost ~ % cd "/Users/CuiGuanyu/Desktop/算法课大作业/codes/" && g++ -std=c++17 -O3 main.cpp BWT_FM.cpp -o main && "/Users/CuiGuanyu/Desktop/算法课大作业/codes/"main
-----|
|              BWT-FM              |
|          Written by G.Cui          |
|-----|
使用方法：
HELP：查看使用帮助。
FILELOAD + 文件名：加载文件内容并创建索引。
STRLOAD + 字符串：加载字符串内容并创建索引。
BWT：显示加载好的索引的BWT结果。
FILEMATCH + 文件名：从文件加载字符串进行模式匹配。
STRMATCH + 模式串：字符串模式匹配。
QUIT：退出。
-----|
>>> |
```

图 1: 编译运行结果

2. 输入 FILELOAD + 文件名加载长串：

在程序“命令行”中键入下列命令，程序将按照老师给定的长串文件的格式读取长串，即第一行为串长，第二行为长串：

```
1 >>> FILELOAD (待匹配串文件的绝对路径)
```

使用老师给的两个长串（SR00001 约为 1.3 亿字符，SR31115 约为 9.4 亿字符）作为文件、索引加载时间测试，结果如下：



```
cd "/Users/CuiGuanyu/Desktop/算法课大作业/codes/" && g++ -std=c++17

|          BWT-FM          |
|        Written by G.Cui        |
|-----|
使用方法：
HELP：查看使用帮助。
FILELOAD + 文件名：加载文件内容并创建索引。
STRLOAD + 字符串：加载字符串内容并创建索引。
BWT：显示加载好的索引的BWT结果。
FILEMATCH + 文件名：从文件加载字符串进行模式匹配。
STRMATCH + 模式串：字符串模式匹配。
QUIT：退出。
-----

>>> FILELOAD /Users/CuiGuanyu/Desktop/算法课大作业/data/SRR00001_rows_ATCG.txt
加载文件中...
加载索引中...
加载索引用时：62.6983s

>>> FILELOAD /Users/CuiGuanyu/Desktop/算法课大作业/data/SRR31115_rows_ATCG.txt
加载文件中...
加载索引中...
加载索引用时：526.183s

>>> |
```

图 2: 文件加载结果

其中较短的长串的加载时间为 60s 左右，较长的长串的加载时间为 520s 左右。运行前者时，索引建立完成后内存占用约为 423 MiB（原文本文件约为 129 MiB），后者内存占用约为 3.3 GiB（原文本文件约为 938 MiB），文件膨胀率约为 3.2-3.6：



图 3: 加载较短长串的内存占用



图 4: 加载较长长串的内存占用

3. 输入 FILEMATCH + 文件名从文件读取字符串匹配子串：

在程序“命令行”中键入下列命令，程序将按照老师给定的匹配串/模式串文件的格式读取短串，即将各行子串拼成一个长串作为模式串进行搜索：

```
1 >>> FILEMATCH （模式串文件的绝对路径）
```

继续之前的程序状态（已加载较长的长串），测试匹配三个短串，结果如下：

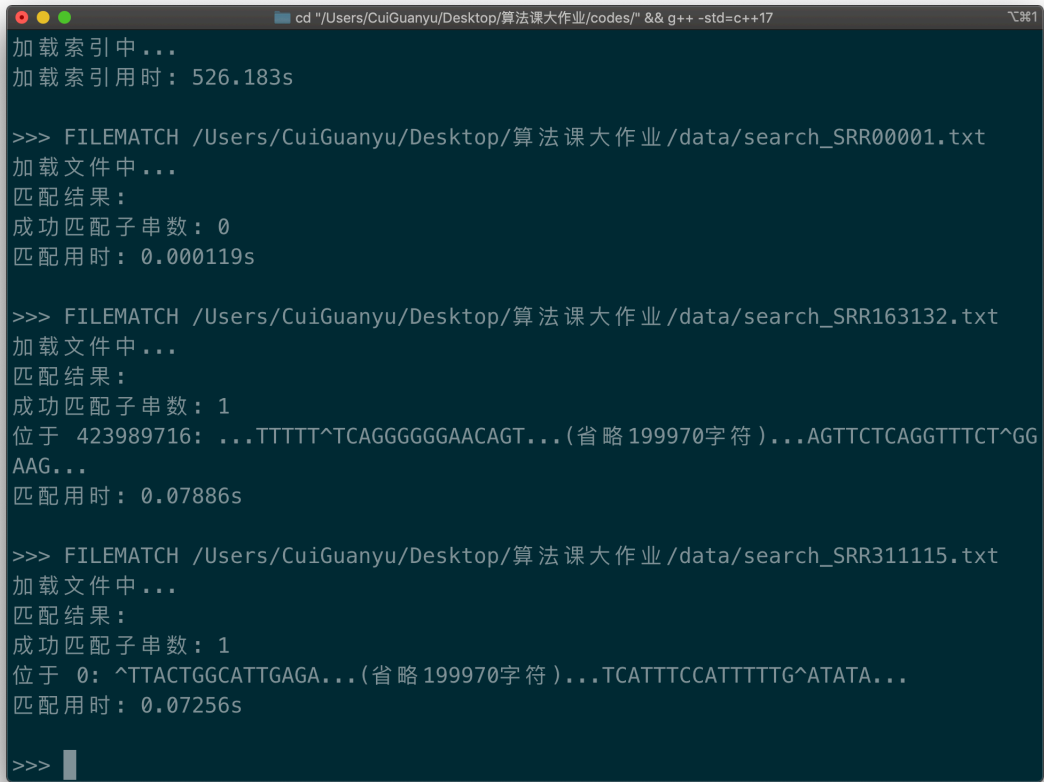
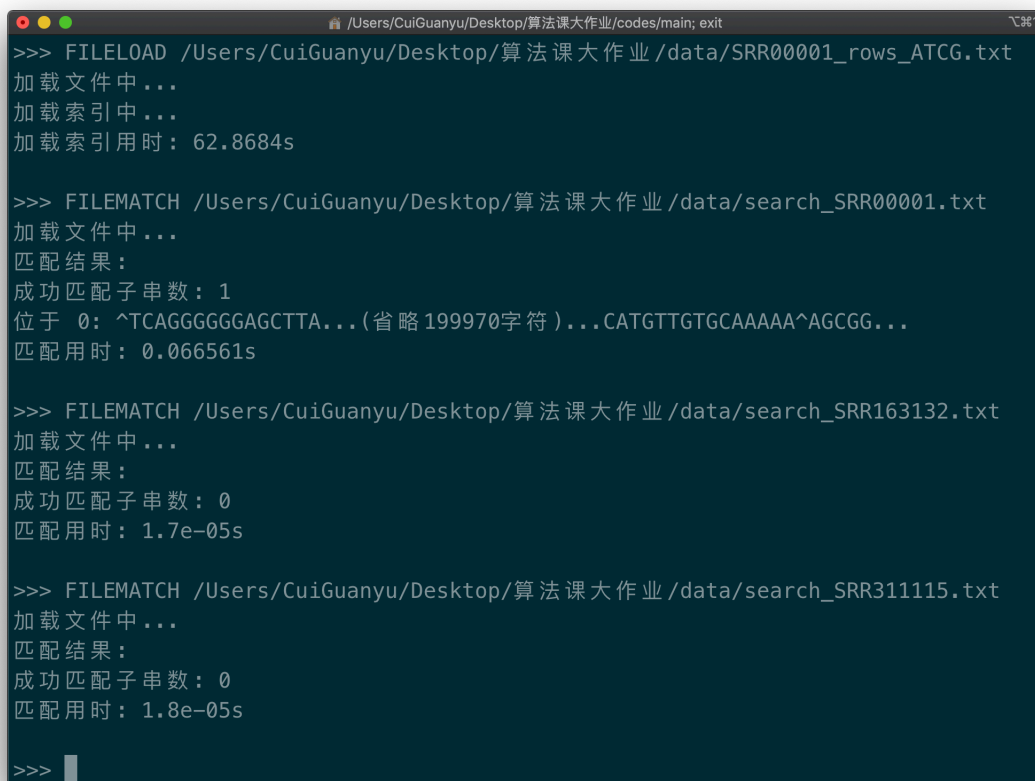


图 5: 三个子串的匹配结果

可以看到，三个子串的匹配时间都很快，均小于 0.1s。其中第一个子串由于很快就失配了，所以匹配时间很短。

以下是 1.3 亿字符的长串中三个子串的匹配结果：



```
>>> FILELOAD /Users/CuiGuanyu/Desktop/算法课大作业/codes/main; exit
加载文件中...
加载索引中...
加载索引用时：62.8684s

>>> FILEMATCH /Users/CuiGuanyu/Desktop/算法课大作业/data/search_SRR00001.txt
加载文件中...
匹配结果：
成功匹配子串数：1
位于 0: ^TCAGGGGGGAGCTTA...(省略199970字符)...CATGTTGTGCAAAA^AGCGG...
匹配用时：0.066561s

>>> FILEMATCH /Users/CuiGuanyu/Desktop/算法课大作业/data/search_SRR163132.txt
加载文件中...
匹配结果：
成功匹配子串数：0
匹配用时：1.7e-05s

>>> FILEMATCH /Users/CuiGuanyu/Desktop/算法课大作业/data/search_SRR311115.txt
加载文件中...
匹配结果：
成功匹配子串数：0
匹配用时：1.8e-05s

>>> 
```

图 6: 三个子串的匹配结果

同上，三个子串的匹配时间都很快，均小于 0.1s。

4. 输入 STRLOAD + 字符串加载长串：

或者也可以不通过文件读取长串，通过输入下列命令，直接从终端输入含有结束符的长串：（后面三个命令一同测试）

```
1 >>> STRLOAD (含结束符的字符串)
```

5. 输入 BWT 输出 $BWT(T)$ 结果：

在程序内输入下列命令，程序将输出 T 经过 BW 变换的结果：

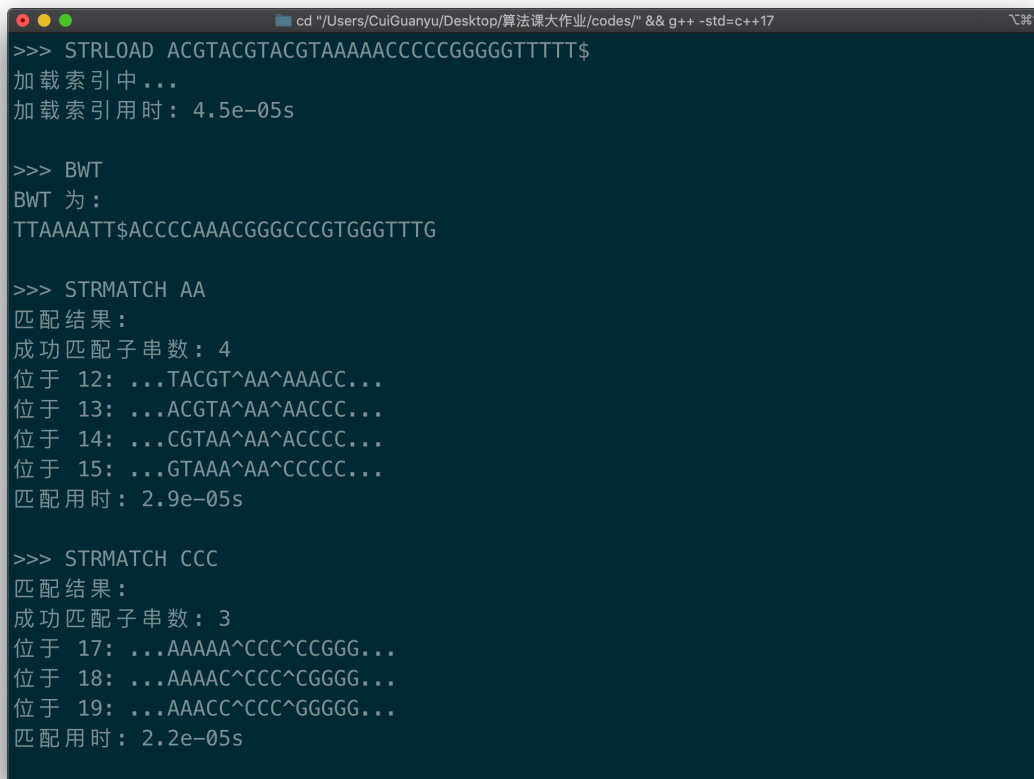
```
1 >>> BWT
```

6. 输入 STRMATCH + 字符串从终端读取字符串匹配子串：

用户也可以直接从终端输入字符串进行字符串匹配：


```
1 >>> STRMATCH (模式串)
```

由于老师给的长串过长，输出 *BWT* 结果不便，于是从终端自行加载一个短串，以此测试上面若干操作：



```
cd "/Users/CuiGuanyu/Desktop/算法课大作业/codes/" && g++ -std=c++17

>>> STRLOAD ACGTACGTACGTAAAAACCCCGGGGGTTTTT$
加载索引中...
加载索引用时：4.5e-05s

>>> BWT
BWT 为：
TTAAAATT$ACCCCAAACGGGCCCGTGGGTTTG

>>> STRMATCH AA
匹配结果：
成功匹配子串数：4
位于 12: ...TACGT^AA^AAACC...
位于 13: ...ACGTA^AA^AACCC...
位于 14: ...CGTAA^AA^ACCCC...
位于 15: ...GTAAA^AA^CCCCC...
匹配用时：2.9e-05s

>>> STRMATCH CCC
匹配结果：
成功匹配子串数：3
位于 17: ...AAAAA^CCC^CCGGG...
位于 18: ...AAAAC^CCC^CGGGG...
位于 19: ...AAACC^CCC^GGGGG...
匹配用时：2.2e-05s
```

图 7: 从终端加载字符串并进行匹配的结果

参考文献

- [1] Nong, G., Zhang, S., & Chan, W. H. (2010). Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10), 1471-1484.
- [2] Burrows, M., & Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm.
- [3] Ferragina, P., & Manzini, G. (2000, November). Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science* (pp. 390-398). IEEE.
- [4] [Lecture Note] Ben Langmead. Burrows-Wheeler Transform & FM Index. http://www.cs.jhu.edu/~langmea/resources/lecture_notes/10_bwt_and_fm_index_v2.pdf
- [5] Manber, U., & Myers, G. . (1993). Suffix arrays. *SIAM Journal on Computing*.
- [6] Larsson, N. J. , & Sadakane, K. . (1999). Faster suffix sorting. *Theoretical Computer Science*, 387(3), 258-272.

- [7] Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6), 918-936.
- [8] [Source Code Website] Yuta Mori. sais - An implementation of the induced sorting algorithm. <https://sites.google.com/site/yuta256/sais>
- [9] [Web] 诱导排序与 SA-IS 算法. <https://riteme.site/blog/2016-6-19/sais.html#fn:paper>
- [10] Working Draft, Standard for Programming Language C++.

A 程序源代码

A.1 BWT_FM 类的定义——**BWT_FM.h**

```
1 #ifndef BWT_FM_H
2 #define BWT_FM_H
3
4 #include <iostream>
5 #include <cmath>
6 #include <vector>
7 #include <string>
8 #include <map>
9 #include <set>
10 #include <algorithm>
11 #include <unordered_map>
12 #include <thread>
13
14 class BWT_FM
15 {
16     public:
17         BWT_FM(const std::string & str, char eol = '$', double alpha = 1.0 / 128,
18             double beta = 1.0 / 32);
19         // 返回经过 bwt 的串
20         std::string bwt();
21         // 展示子串查询结果
22         void displayPatternQuery(const std::string & pattern);
23     private:
24         // 原字符串(添加结尾符)
25         std::string original;
```

```
25 // 后缀数组 SA
26 typedef long satype;
27 // 因为是按值等间隔压缩，所以要保留下标
28 std::unordered_map<size_t, satype> suffixArray;
29 // suffixArray 的保留比
30 double alpha;
31 // BMT 的字符串
32 std::string BWTstr;
33 // 字母表
34 std::set<char> alphabet;
35 // 字母-相对顺序映射
36 std::unordered_map<char, size_t> charOrder;
37 // 各字符在排序好的后缀中出现的范围
38 std::vector<std::pair<size_t, size_t>> FCharRange;
39 // 为了实现O(m)子串查询
40 std::vector<std::vector<size_t>> LCheckPoints;
41 // LCheckPoints 的保留比
42 double beta;
43
44 // 建立后缀数组
45 // 建立各字符出现次数的索引
46 // 建立BWT处理的子串
47 void preprocess();
48
49 // 查询子串，返回所有子串起始点下标
50 std::vector<size_t> patternQuery(const std::string & pattern);
51 // 打印子串以及附近的情况
52 void printSubStr(size_t startPos, size_t subStrLen, size_t backFront = 5);
53
54 // 根据压缩后的LCheckPoints，返回 c 字符在 BWT[index] 位置的位次
55 size_t LCheckPointsFind(char c, size_t index);
56 // 计算 1 行的 LF-Mapping
57 size_t LFMapping(size_t l);
58 // 在压缩后的 SA 中计算 BWT[index] 对应的 SA 的位次
59 satype suffixArrayFind(size_t index);
60
61 // 不允许复制构造，拷贝构造
```

```
62     BWT_FM(const BWT_FM & rhs){}
63     void operator = (const BWT_FM & rhs){}
64 };
65
66 #endif
```

A.2 BWT_FM 类的实现——BWT_FM.cpp

```
1 #include "BWT_FM.h"
2 #include "../sais-lite-2.4.1/sais.hxx"
3
4 BWT_FM::BWT_FM(const std::string & str, char eol, double alpha, double beta)
5 : original(str), alpha(alpha), beta(beta)
6 {
7     // 去掉之前的结尾符
8     original.pop_back();
9     original.push_back(eol);
10    // 认为是较短的串的长度
11    const size_t shortStrLen = 1000;
12    // 认为是较长的串的长度
13    const size_t longStrLen = 100000;
14    // 长度太短，不压缩
15    if(original.length() <= shortStrLen)
16    {
17        this->alpha = this->beta = 1.00;
18    }
19    // 中间按线性变化
20    else if(original.length() <= shortStrLen + longStrLen)
21    {
22        this->alpha = (alpha - 1.00) / longStrLen * (original.length() -
shortStrLen) + 1.00;
23        this->beta = (beta - 1.00) / longStrLen * (original.length() -
shortStrLen) + 1.00;
24    }
25    // 够长，开始压缩
26    else
27    {
```

```
28         this -> alpha = alpha;
29         this -> beta = beta;
30     }
31     // 建索引等
32     preprocess();
33 }
34
35 std::string BWT_FM::bwt()
36 {
37     return BWTstr;
38 }
39
40 void BWT_FM::displayPatternQuery(const std::string & pattern)
41 {
42     // 做子串查询
43     auto ret = patternQuery(pattern);
44     std::cout << "成功匹配子串数: " << ret.size() << std::endl;
45     // 依次打印结果
46     for(size_t i = 0; i < ret.size(); i++)
47     {
48         printSubStr(ret[i], pattern.length(), 5);
49         std::cout << std::endl;
50     }
51 }
52
53 void BWT_FM::preprocess()
54 {
55     // 调用 sais 库, 可在 O(n) 时间内建立后缀数组
56     satype * pSA = new satype[original.length()];
57     saisxx<const char *, satype *, satype>(original.c_str(), pSA, original.length()
58 );
59
60     // 后缀数组的值间隔
61     satype SAValueGap = satype(1 / beta);
62
63     // 构建后缀数组同时构建字母表
64     // 循环 n 次, 每次 O(1), 共计 O(n)
```

```
64     for(size_t i = 0; i < original.length(); i++)
65     {
66         // 按值等间隔保存
67         if(pSA[i] % SAValueGap == 0)
68         {
69             suffixArray.insert(std::make_pair(i, pSA[i]));
70         }
71         // find 时间复杂度为  $O(\log k)$ ,  $k$  为已有字符数
72         // 因为是常数字母表, 所以  $k \leq |\Sigma| = O(1)$ 
73         if(alphabet.find(original[i]) == alphabet.end())
74         {
75             alphabet.insert(original[i]);
76         }
77     }
78
79     // 计算字母表的相对顺序
80     // 因为是常数字母表, 所以时间复杂度  $O(1)$ 
81     charOrder.reserve(alphabet.size());
82     {
83         size_t cOrder = 0;
84         for(auto i = alphabet.begin(); i != alphabet.end(); i++, cOrder++)
85         {
86             charOrder[(*i)] = cOrder;
87         }
88     }
89
90     // 初始化
91     FCharRange.resize(alphabet.size(), std::make_pair<size_t, size_t>(0, 0));
92     BWTstr.resize(original.length(), 0);
93     LCheckPoints.resize(alphabet.size(), std::vector<size_t>(size_t(std::ceil(
94         original.length() * alpha)), 0));
95
96     // LCheckPoints 的间隔
97     size_t LCheckGap = size_t(1 / alpha);
98     {
99         char prevChar = original[ pSA[0] ];
100         FCharRange[charOrder[prevChar]].first = 0;
```

```
100
101 // 临时的 LCheckPoints 的一行，压缩用
102 std::vector<size_t> tmpLCheckPointsRow(alphabet.size(), 0);
103 for(size_t i = 0; i < original.length(); i++)
104 {
105     // 计算F列各字母出现范围
106     // L 数组中 c 出现的下标范围 [begin, end)
107     // 从头扫到尾，时间复杂度 O(n)
108     char nowChar = original[ pSA[i] ];
109     if(nowChar != prevChar)
110     {
111         FCharRange[charOrder[ prevChar ]].second = i;
112         FCharRange[charOrder[ nowChar ]].first = i;
113         prevChar = nowChar;
114     }
115
116     // 计算 BWTstr
117     // 利用后缀数组建立BWTstr
118     // 显然也是 O(n)
119     BWTstr[i] = ( ( pSA[i] > 0 ) ? original[pSA[i] - 1] : '$');
120
121     // 计算L列的Checkpoints
122     // n 次循环，每次 O(Sigma)，共计 O(Sigma n)=O(n)
123     // Sigma次循环
124
125     for(auto j = alphabet.begin(); j != alphabet.end(); j++)
126     {
127         // 根据之前的行，变更Checkpoint的值
128         tmpLCheckPointsRow[ charOrder[*j] ] = (i == 0 ? 0 :
tmpLCheckPointsRow[ charOrder[*j] ]) + (BWTstr[i] == *j);
129     }
130     // 保留一部分
131     if(i % LCheckGap == 0)
132     {
133         for(auto j = alphabet.begin(); j != alphabet.end(); j++)
134         {
135             LCheckPoints[ charOrder[*j] ][i / LCheckGap] = (
```

```
        tmpLCheckPointsRow[ charOrder[*j] ]));
136         }
137     }
138 }
139 delete[] pSA;
140 FCharRange[charOrder[ prevChar ]].second = original.length();
141 }
142 }
143
144 std::vector<size_t> BWT_FM::patternQuery(const std::string & pattern)
145 {
146     auto i = pattern.rbegin();
147     // 略过空白
148     while(i != pattern.rend() && *i == '\\0')
149     {
150         i++;
151     }
152     if(alphabet.find(*i) == alphabet.end())
153     {
154         return {};
155     }
156
157     // 出事范围
158     std::pair<size_t, size_t> range = FCharRange[charOrder[*i]];
159     i++;
160     for(; i != pattern.rend(); i++)
161     {
162         char charToFind = *i;
163         // 找不到
164         if(alphabet.find(*i) == alphabet.end())
165         {
166             return {};
167         }
168         // 生成新范围
169         range = std::make_pair(
170             FCharRange[charOrder[charToFind]].first + LCheckPointsFind(charToFind,
range.first - 1),
```



```
171         FCharRange[charOrder[charToFind]].first + LCheckPointsFind(charToFind,
range.second - 1) - 1 + 1
172     );
173     // 如果是空范围
174     if(range.second <= range.first)
175     {
176         return {};
177     }
178 }
179 // 排序搜索到的子串位置
180 std::vector<size_t> ret;
181 for(size_t k = range.first; k < range.second; k++)
182 {
183     ret.push_back(suffixArrayFind(k));
184 }
185 std::sort(ret.begin(), ret.end());
186 return ret;
187 }
188
189 void BWT_FM::printSubStr(size_t startPos, size_t subStrLen, size_t backFront)
190 {
191     std::cout << "位于 " << startPos << ": ";
192     // 打印前半部分
193     if(startPos <= backFront)
194     {
195         std::cout << original.substr(0, startPos);
196     }
197     else
198     {
199         std::cout << "... " << original.substr(startPos - backFront, backFront);
200     }
201     // 当查找子串过长, 省略中间部分
202     const int N = 30;
203     assert(N % 2 == 0);
204     if(subStrLen <= N)
205     {
206         std::cout << "^" << original.substr(startPos, subStrLen) << "^";
```

```
207     }
208     else
209     {
210         std::cout << "^" << original.substr(startPos, N / 2) << "... (省略"
211             << subStrLen - N << " 字符)..."
212             << original.substr(startPos + subStrLen - N / 2, N / 2) << "^";
213     }
214     // 打印后半部分
215     if(startPos + subStrLen - 1 + backFront >= original.length())
216     {
217         std::cout << original.substr(startPos + subStrLen);
218     }
219     else
220     {
221         std::cout << original.substr(startPos + subStrLen, backFront) << "...";
222     }
223 }
224
225 // 在压缩过的 LCheckPoints 里定位
226 size_t BWT_FM::LCheckPointsFind(char c, size_t index)
227 {
228     size_t LCheckGap = (1 / alpha);
229     if ((index % LCheckGap) == 0)
230     {
231         return LCheckPoints[charOrder[c]][index / LCheckGap];
232     }
233     // 根据余数（即组间位置）确定最近的 CheckPoint 在高下标还是低下标
234     // 余数超过间隔的一半，最近的 CheckPoint 在高下标
235     else if ((index % LCheckGap) > (LCheckGap / 2) && ((index / LCheckGap) + 1 <
LCheckPoints.size()))
236     {
237         size_t nearestCP = ((index / LCheckGap) + 1) * LCheckGap;
238         // 从高下标往回扫
239         // 原基准
240         size_t base = LCheckPoints[charOrder[c]][(index / LCheckGap + 1)];
241         // 沿路遇到的该字符数目
242         size_t charCount = 0;
```

```
243 // 注意：此处 size_t 必定不会小于 LCheckGap / 2，所以不会溢出
244 for(size_t i = nearestCP; i > index; i--)
245 {
246     charCount += (BWTstr[i] == c);
247 }
248 return base - charCount;
249 }
250 // 余数没有超过间隔的一半，最近的CheckPoint在高下标
251 else
252 {
253     size_t nearestCP = (index / LCheckGap) * LCheckGap;
254     // 从低下标往后扫
255     // 原基准
256     size_t base = LCheckPoints[charOrder[c]][(index / LCheckGap)];
257     // 沿路遇到的该字符数目
258     size_t charCount = 0;
259     for(size_t i = nearestCP + 1; i <= index; i++)
260     {
261         charCount += (BWTstr[i] == c);
262     }
263     return base + charCount;
264 }
265 }
266
267 size_t BWT_FM::LFMapping(size_t l)
268 {
269     return FCharRange[charOrder[BWTstr[l]]].first + LCheckPointsFind(BWTstr[l], 1)
270     - 1;
271 }
272
273 BWT_FM::satype BWT_FM::suffixArrayFind(size_t index)
274 {
275     // 如果是保留的部分
276     if(suffixArray.find(index) != suffixArray.end())
277     {
278         return suffixArray[index];
279     }
```

```
279 // 跳跃次数
280 size_t jumpCount = 0;
281 size_t nowIndex = index;
282 // 循环跳
283 while (suffixArray.find(nowIndex) == suffixArray.end())
284 {
285     jumpCount++;
286     nowIndex = LFMapping(nowIndex);
287 }
288 return suffixArray[nowIndex] + jumpCount;
289 }
```

A.3 主界面的实现——**main.cpp**

```
1 #include <iostream>
2 #include <fstream>
3 #include <ctime>
4 #include <cctype>
5 #include <algorithm>
6 #include "BWT_FM.h"
7 // #include "BWT_FM.cpp"
8
9 int main()
10 {
11     // lambda 打印作者信息
12     auto printAuthor = []() -> void
13     {
14         std::cout << " |-----| "
15         << std::endl;
16         std::cout << " |               BWT-FM               | "
17         << std::endl;
18         std::cout << " |               Written by G.Cui               | "
19         << std::endl;
20         std::cout << " |-----| "
21         << std::endl;
22     };
23     // lambda 打印使用方法
```

```
20     auto printUsage = []() -> void
21     {
22         std::cout << "-----"
<< std::endl;
23         std::cout << "使用方法: " << std::endl;
24         std::cout << "HELP: 查看使用帮助。" << std::endl;
25         std::cout << "FILELOAD + 文件名: 加载文件内容并创建索引。" << std::endl;
26         std::cout << "STRLOAD + 字符串: 加载字符串内容并创建索引。" << std::endl;
27         std::cout << "BWT: 显示加载好的索引的BWT结果。" << std::endl;
28         std::cout << "FILEMATCH + 文件名: 从文件加载字符串进行模式匹配。" << std::
endl;
29         std::cout << "STRMATCH + 模式串: 字符串模式匹配。" << std::endl;
30         std::cout << "QUIT: 退出。" << std::endl;
31         std::cout << "-----"
<< std::endl;
32     };
33
34     // BWT_FM 单例
35     BWT_FM * bwtfm = nullptr;
36     // 先打印作者和用法
37     printAuthor();
38     printUsage();
39     // prompt 提示符
40     std::cout << std::endl << ">>> ";
41     // 第一次输入命令
42     std::string instr;
43     std::cin >> instr;
44     // 转换为大写
45     std::transform(instr.begin(), instr.end(), instr.begin(),
46         [](char c) -> char { return std::toupper(c); });
47     while(instr != "QUIT")
48     {
49         // 帮助
50         if(instr == "HELP")
51         {
52             printUsage();
53         }
```

```
54 // 从文件加载
55 else if(instr == "FILELOAD")
56 {
57     std::string fname;
58     // 删除额外空格
59     std::cin >> std::ws;
60     std::getline(std::cin, fname);
61     while(fname.back() == ' ' || fname.back() == '\r'
62           || fname.back() == '\n' || fname.back() == '\0')
63     {
64         fname.pop_back();
65     }
66     std::cout << "加载文件中..." << std::endl;
67     // 打开文件
68     std::ifstream file(fname);
69     if(!file.is_open())
70     {
71         std::cout << "无法打开文件!" << std::endl;
72         std::cout << std::endl << ">>> ";
73         std::cin >> instr;
74         std::transform(instr.begin(), instr.end(), instr.begin(),
75                        [](char c) -> char { return std::toupper(c); });
76         continue;
77     }
78     // 文件直接构建字符串
79     std::string text;
80     // 跳过第一行长度
81     std::getline(file, text);
82     std::getline(file, text);
83     // 关闭文件
84     file.close();
85     // 先删除原来的
86     delete bwtfm;
87     std::cout << "加载索引中..." << std::endl;
88     std::clock_t startTime = std::clock();
89     // 创建新的
90     bwtfm = new BWT_FM(text);
```

```
91         std::clock_t endTime = std::clock();
92         // 输出用时
93         std::cout << "加载索引用时: " << double(endTime - startTime) /
CLOCKS_PER_SEC << "s" << std::endl;
94     }
95     // 从字符串构建
96     else if(instr == "STRLOAD")
97     {
98         std::string text;
99         std::cin >> text;
100         delete bwtfm;
101         std::cout << "加载索引中..." << std::endl;
102         std::clock_t startTime = std::clock();
103         bwtfm = new BWT_FM(text);
104         std::clock_t endTime = std::clock();
105         std::cout << "加载索引用时: " << double(endTime - startTime) /
CLOCKS_PER_SEC << "s" << std::endl;
106     }
107     // 输出BWT结果
108     else if(instr == "BWT")
109     {
110         // 如果 bwtfm 不存在
111         if(bwtfm == nullptr)
112         {
113             std::cout << "尚未加载文件! " << std::endl;
114             std::cout << std::endl << ">>> ";
115             std::cin >> instr;
116             std::transform(instr.begin(), instr.end(), instr.begin(),
117                 [](char c) -> char { return std::toupper(c); });
118             continue;
119         }
120         std::cout << "BWT 为: " << std::endl;
121         std::cout << bwtfm -> bwt() << std::endl;
122     }
123     // 子串匹配
124     else if(instr == "FILEMATCH")
125     {
```

```
126 // 子串
127 std::string sub;
128 std::string fname;
129 // 删除额外空格
130 std::cin >> std::ws;
131 std::getline(std::cin, fname);
132 while(fname.back() == ' ' || fname.back() == '\r'
133        || fname.back() == '\n' || fname.back() == '\0')
134 {
135     fname.pop_back();
136 }
137 std::cout << "加载文件中..." << std::endl;
138 // 打开文件
139 std::ifstream file(fname);
140 if(!file.is_open())
141 {
142     std::cout << "无法打开文件!" << std::endl;
143     std::cout << std::endl << ">>> ";
144     std::cin >> instr;
145     std::transform(instr.begin(), instr.end(), instr.begin(),
146                    [](char c) -> char { return std::toupper(c); });
147     continue;
148 }
149 // 文件直接构建字符串
150 std::string text;
151 while(!file.eof())
152 {
153     std::getline(file, text);
154     while(text.back() == '\r' || text.back() == '\n')
155     {
156         text.pop_back();
157     }
158     sub += text;
159 }
160 // 关闭文件
161 file.close();
162 if(bwtfm == nullptr)
```



```
163         {
164             std::cout << "尚未加载文件!" << std::endl;
165             std::cout << std::endl << ">>> ";
166             std::cin >> instr;
167             std::transform(instr.begin(), instr.end(), instr.begin(),
168                 [](char c) -> char { return std::toupper(c); });
169             continue;
170         }
171         std::cout << "匹配结果:" << std::endl;
172         std::clock_t startTime = std::clock();
173         bwtfm -> displayPatternQuery(sub);
174         std::clock_t endTime = std::clock();
175         std::cout << "匹配用时:" << double(endTime - startTime) /
CLOCKS_PER_SEC << "s" << std::endl;
176     }
177     else if(instr == "STRMATCH")
178     {
179         // 子串
180         std::string sub;
181         std::cin >> sub;
182         if(bwtfm == nullptr)
183         {
184             std::cout << "尚未加载文件!" << std::endl;
185             std::cout << std::endl << ">>> ";
186             std::cin >> instr;
187             std::transform(instr.begin(), instr.end(), instr.begin(),
188                 [](char c) -> char { return std::toupper(c); });
189             continue;
190         }
191         std::cout << "匹配结果:" << std::endl;
192         std::clock_t startTime = std::clock();
193         bwtfm -> displayPatternQuery(sub);
194         std::clock_t endTime = std::clock();
195         std::cout << "匹配用时:" << double(endTime - startTime) /
CLOCKS_PER_SEC << "s" << std::endl;
196     }
197     else
```

```
198     {
199         std::cout << "无效指令!" << std::endl;
200         // 再次提示用法
201         printUsage();
202     }
203     std::cout << std::endl << ">>> ";
204     std::cin >> instr;
205     std::transform(instr.begin(), instr.end(), instr.begin(),
206         [](char c) -> char { return std::toupper(c); });
207 }
208 delete bwtfm;
209 return 0;
210 }
```