# 算法设计 Project：de Bruijn 图上的编辑距离

## 实验报告

计算机科学与技术
刘超颖 14307130318

## 1. 开发环境

使用 CLion 集成开发环境，使用 CMake 作为编译工具，使用 C++11 标准的 C++ 作为主要开发语言，辅助以 Python2.7.3 进行了极少量的文件处理操作。Task1和Task2运行环境均是macOS Sierra 10.12.5, 3 GHz Intel Core i7, 16GB的PC，Task3运行环境是Linux, 128GB的服务器。

## 2. Task1的设计思路与优化过程

### 2.1 编辑距离

编辑距离是通过计算将一个字符串转换为另一个字符串所需的最小操作数量，通常用编辑距离来量化两个字符串之间的不同。在生物信息学中，可以将两个字符串视为由字母A，C，G和T组成的字符序列，从而用编辑距离来量化DNA序列的相似性。不同的编辑距离定义使用不同的字符串操作集， Levenshtein 距离操作集中包括字符串中字符的删除、插入和替换。作为最常见的指标，通常使用 Levenshtein 距离来表示编辑距离。其定义为：

---

给出字符串 $a$ 与 $b$ ，编辑距离 $d(a,b)$ 指通过以下操作将 $a$ 转化为 $b$ 的最小操作次数，操作包括：

- **插入**：若 $a=uv$ ，插入字符 $x$ 得到 $a=uxv$
- **删除**：若 $a=uxv$ ，删除字符 $x$ 得到 $a=uv$
- **替换**：对于字符 $x,y$ 且 $x\neq y$ ，将 $a=uxv$ 转化为 $a=uyv$

---

目前解决编辑距离问题的一个主流方法是 Wagner-Fischer 算法[1]，这个算法的基本思想是：当我们已经得到两个字符串前缀字符串的编辑距离时，我们可以通过简单的计算得知将两个前缀字符串均延长一位字符后得到的两个字符串之间的编辑距离。根据这个思想，我们可以构造一个矩阵，这个矩阵以两个字符串的所有前缀字符串分别作为横、纵坐标，存储两个字符串的所有前缀字符串之间的编辑距离，然后通过前缀字符串之间的编辑距离计算延长新字符后的编辑距离，从而不断地更新并填充矩阵，直到算出两个完整字符串之间的编辑距离并将其作为计算的最后一个值。

前缀字符串之间的编辑距离与延长一个字符后的字符串之间的编辑距离计算关系如下：

$$d_{i0} = \sum_{k=1}^{i} w_{del}(b_k) \qquad for\ 1 \le i \le m$$

$$d_{0j} = \sum_{k=1}^{j} w_{ins}(a_k) \qquad for\ 1 \le j \le n$$

$$d_{ij} = \begin{cases} d_{i-1,j-1} & for\ a_j = b_i \\ min\begin{cases} d_{i-1,j} + w_{del}(b_i) \\ d_{i,j-1} + w_{ins}(a_j) \\ d_{i-1,j-1} + w_{sub}(a_j, b_i) \end{cases} & for\ a_j \ne b_i \end{cases} \qquad for\ 1 \le i \le m, 1 \le j \le n$$

在 Task1 中定义的编辑距离中，$w_{ins}$，$w_{del}$，$w_{sub}$ 的值都是 1，故我们可将上述公式改写为：

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & if\ \min(i,j) = 0 \\ min\begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{a_i \ne b_j} \end{cases} & otherwise \end{cases}$$

其中，$1_{a_i \ne b_j}$ 是一个指示函数，当 $a_i = b_j$ 时，该指示函数为 0；当 $a_i \ne b_j$ 时，该指示函数为 1。$lev_{a,b}(i,j)$ 是字符串 $a$ 取前 $i$ 个字符的子串和字符串 $b$ 取前 $j$ 个字符的子串之间的编辑距离[2]。

我们可以通过上述公式，很直观地得到一个递归求解编辑距离的方法，其伪代码如下：

```
int editDistance(string s, string t)
{
    int cost;

    if (s.length == 0)
        return t.length;
    if (t.length == 0)
        return s.length;

    if (s[s.length-1] == t[t.length-1])
        cost = 0;
    else
        cost = 1;

    return minimum(editDistance(s[0: -1], t) + 1,
                   editDistance(s, t[0: -1]) + 1,
```

```
                    editDistance(s[0: -1], t[0: -
1]) + cost);
}
```

## 2.2 Wagner-Fischer 算法及其滚动数组优化

我们发现，对于编辑距离问题，如果用以上递归的方法求解，时间复杂度为 $O((mn)^2)$ ，这是非常低效的。直观上，在用递归算法自顶向下对问题进行求解时，每次计算子串编辑距离的问题并不是新问题，这些子串的编辑距离会被重复计算多次。

这时，我们考虑之前提到的 Wagner-Fischer 算法，构造一个矩阵，这个矩阵以两个字符串的所有前缀字符串分别作为横、纵坐标，存储两个字符串的所有前缀字符串之间的编辑距离，然后通过前缀字符串之间的编辑距离计算延长新字符后的编辑距离，从而不断地更新并填充矩阵，直到算出两个完整字符串之间的编辑距离并将其作为计算的最后一个值。这一算法基于动态规划的思想，将每个子串的编辑距离计算出的结果存入矩阵中，下一次使用时不用重复计算，而是直接从矩阵中取出。其伪代码如下：

```
int editDistance(string s, string t)
{
    int d[s.length][t.length] = {0};

    for (int i = 0; i < s.length; i++)
        d[i, 0] = i;
    for (int j = 0; j < t.length; j++)
        d[0, j] = j;

    for (int j = 0; j < t.length; j++) {
        for (int i = 0; i < s.length; i++) {
            if (s[i] == t[j])
                subCost = 0
            else
                subCost = 1
            d[i, j] = minimum(d[i-1, j] + 1,
                              d[i, j-1] + 1,
                              d[i-1, j-1] + subCost)
        }
    }
  return d[m, n]
}
```

Wagner-Fischer 算法的时间复杂度为 $O(mn)$ ，可见在时间方面，这一算法的确对编辑距离产生了很大的优化，但与之同时，我们发现 Wagner-Fischer

算法在处理较长的字符串的时候，产生了极大的空间复杂度，因为需要存储一个 $m \times n$ 大小的二维数组。

接着，我们发现在计算过程中，每一次循环其实并不需要用到整个矩阵的所有数据。事实上，我们可以很直观地看出，$d[i,j]$ 的计算只和它左上、左侧、上侧的三个元素有关，而每一次循环只需要使用矩阵中的两行数据就够了。所以，我们并不需要 $m \times n$ 大小的空间来存储一个大矩阵，而只需要 $2 \times m$ 大小，将这一空间设置为静态内存，每次循环将其中的数据更新即可。其伪代码如下：

```
static int v0[t.length]
static int v1[t.length]

int LevenshteinDistance(string s, string t)
{
    for (int i = 0; i < t.length; i++)
        v0[i] = i
    for (int i = 0; i < s.length; i++) {
        v1[0] = i + 1

        for (int j = 0; j < t.length; j++) {
            if (s[i] == t[j])
                subCost = 0
            else
                subCost = 1
            v1[j + 1] = min(v1[j] + 1, v0[j + 1] + 1, v0
[j] + subCost)
        }
        swap v0 with v1
    }
    return v1[n]
}
```

至此，Task1 计算编辑距离的算法已经被优化至时间复杂度：$O(mn)$，空间复杂度：$O(m)$。其中，部分优化思路是十分重要的，在之后的 Task2 中也能体现。

## 2.3 其他算法细节与优化

除了使用上述算法简单地计算编辑距离外，我们还需要对每一次的操作内容进行记录，这是 Task1 的难点之一。在本文中，我们运用枚举类型来定义插入、删除、替换三种操作，在执行

```
v1[j + 1] = min(v1[j] + 1, v0[j + 1] + 1, v0[j] + subCos
t)
```

的时候的时候，使用 `if` 语句将三种情况分开，然后在 `if` 内对操作种类、操作位置和被操作数进行记录。

在实现了上述算法后，还有一些与 C++ 语言本身性质相关的细节。首先，我们应该尽量避免使用 vector 的使用。STL vector 的实现由于某些原因非常慢，使用 vector 会耗费大量时间，在 Task1 这种数据量较小，字符串本身也不长的情况下，完全可以开一个定长的大数组对所有信息进行存储。其次，在编译过程中，可以开启 -o2 对编译文件进行优化，这一优化在 Task1 中不是很明显，但在后续 Task 中十分有效

## 2.4 实验结果

代码运行结果见 $./task1/task1.out$ 。通过 `clock()` 函数可测试出递归算法、 Wagner-Fischer 算法、优化后的 Wagner-Fischer 算法和细节优化后的算法运行时间如下表：

| | **递归算法** | **Wagner-Fischer 算法** | **优化后的 Wagner-Fischer 算法** | **细节优化后的算法** |
|---|---|---|---|---|
| **运行时间** | 约 1 小时 | 2.63871 秒 | 1.38755 秒 | 0.72413 秒 |

# 3. Task2的设计思路与优化过程

## 3.1 de Bruijn 图

在图论中，一个 $k$ 阶 de Bruijn 图是一个用来表示序列交叠的有向图。图中的每个节点表示一个长度为 $k$ 的字符串，若两个节点的字符序列有 $k-1$ 的交叠即在这两个节点之间连一条有向边。因此在一个 $k$ 阶 de Bruijn 图上，一个长度为 $l$ 的路径可表示一个长度为 $k+l-1$ 的字符串。

## 3.2 一个简陋的算法

根据 de Bruijn 图的特性，我们可以直观地联想到将图中两个节点连接这一过程与之前编辑距离问题中在前缀字符串后添加字符进行对比，我们发现这两个过程是非常类似的，所以考虑是否能继承或沿用之前编辑路径问题的算法。

此处，我们先给出一个较为简陋的算法，其对时间和空间的消耗都是巨大的，我们会在后期对其进行改写和优化，这里只体现 de Bruijn 图问题是如何沿用编辑路径问题算法的。

首先，我们延续 Wagner-Fischer 算法的思想，仍考虑生成一个矩阵来存储每个节点和目标字符串之间的编辑距离。我们以目标字符串的所有前缀子串作为横坐标，以 de Bruijn 图的每个节点字符串作为纵坐标，通过与 Task1 中相似的动态规划思想计算矩阵每一个位置的编辑距离。在生成这样一个初始矩阵之后，我们寻找每个纵坐标内的 de Bruijn 图节点的邻接节点，然后在该节

点之后添加相应的字母，生成一个新的编辑距离矩阵，以此方式不断递推，直到节点不能继续向下延伸。我们发现，每一个新矩阵内的数据可以通过上一个矩阵内存储的数据计算得出，所以在实际运算过程中，只需要存储两个滚动矩阵即可。两个矩阵内数据的计算关系如下：

$$matrix_{now}(i,j) = \begin{cases} \max(i,j) & if\ \min(i,j) = 0 \\ min\begin{cases} matrix_{now}(i,j-1)+1 \\ matrix_{last}(i,j)+1 \\ matrix_{last}(i,j-1)+1_{a_i \neq b_j} \end{cases} & otherwise \end{cases}$$

以 http://datamining-iip.fudan.edu.cn/ppts/algo/pj2017/index.html 给出的 Task2&3 测试样例为例，生成的初始矩阵如下：

| | A | AA | AAC | AACC | AACCA | … |
|---|---|---|---|---|---|---|
| ACG | | | | | | |
| TCG | | | | | | |
| CGA | | | | | | |
| CGC | | | | | | |
| CTG | | | | | | |
| … | | | | | | |

在第一次循环后，新生成的矩阵如下：

| | A | AA | AAC | AACC | AACCA | … |
|---|---|---|---|---|---|---|
| ACGA | | | | | | |
| ACGC | | | | | | |
| TCGA | | | | | | |
| TCGC | | | | | | |
| CGAT | | | | | | |
| … | | | | | | |

这个简陋算法存在很多细节问题。首先，我们对于循环中止的判定条件是：de Bruijn 图的节点不能继续延伸，但事实上，我们考虑到当图中存在环状结构的情况下，循环中止条件永远不会被触发，会造成死循环。关于这个问题，我们可以加入另一循环中止条件，即当新生成的矩阵的最后一列的所有元素都大于上一个矩阵最后一列的最小元素时，循环中止。其次，我们发现矩阵与矩阵之间的对应关系不应是简单的角标对应，因为在节点向外延伸的过程中，后一个矩阵极有可能对原矩阵进行扩张或者缩减。关于这个问题，我们可以记录每个节点的来源，相应地进行计算。

在解决了上述问题以后，我们发现了关于这个算法更严重的问题，即时间空间复杂度太高。尤其是空间复杂度，在节点个数更多、目标字符串长度更长的情况下，不断生成新矩阵会使矩阵扩张到极大，尤其当 de Bruijn 图的结构是类似于多叉树的情况下，每一次矩阵都被扩张数倍，这样将会占用大量的内存空间，在数据量较大的情况下是不现实的。因此，我们基于这个算法的基础，重新考虑一个新的算法

## 3.3 使用全局 BFS 思想的算法

这一算法是我们目前想到的一个较优的算法，其时间复杂度为 $O(n*m*k)$，空间复杂度为 $O(n*m)$ 。

与之前的简陋算法相似的是，我们使用一个类似矩阵的结构来存储每一个前缀子串和 de Bruijn 图节点之间的最小编辑距离，但不同的是，我们将之前的矩阵扩张的思路改成了矩阵状态转移的方法。即使用 BFS 的思想，在矩阵的每一个位置存储前缀子串到当前节点的最小编辑距离，如果对一个节点的最小编辑距离有所更新，则将该节点加入队列，通过队列操作确保其编辑距离最小这一特点的正确性。

在这一算法中，我们以 de Bruijn 图的每个节点字符串作为横坐标，以目标字符串的所有前缀子串作为纵坐标，生成一个矩阵如下：

| | ACG | TCG | CGA | CGC | CTG | … |
|---|---|---|---|---|---|---|
| A | | | | | | |
| AA | | | | | | |
| AAC | | | | | | |
| AACC | | | | | | |
| AACCA | | | | | | |
| … | | | | | | |

通过不断更新这一矩阵中的数据来反映节点之间的转移过程，直到匹配到最后一行得到所有路径和目标字符串的最小编辑距离，再从中取出最小值，这一值就是全局的最小编辑距离。

其中，矩阵的每一个元素的值的计算方式为：

$$matrix_{now}(i,j)=\begin{cases}\max(i,j) & if\ \min(i,j)=0\\ min\begin{cases}matrix(i,src)+1\\ matrix(i-1,j)+1\\ matrix(i-1,src)+1_{a_i\neq b_j}\end{cases} & otherwise\end{cases}$$

其中， $src$ 代表该横坐标对应的节点与目标节点之间存在一条有向边。

通过上述计算公式，我们同样发现，对矩阵某一行元素的计算，只与该矩阵上一行内的元素有关。所以我们可以在此基础上对空间复杂度进行基于滚动数组的优化：在每次循环中，只保留矩阵中涉及计算的两行数据。这一操作将编辑距离计算过程中的空间复杂度从 $O(m*n)$ 优化成了 $O(m)$（ $m$ 为 de Bruijn 图节点个数， $n$ 为目标字符串长度)。

## 3.4 使用全局 BFS 思想算法的其他细节优化

### 3.4.1 邻接表的存储

在节点之间转移的操作中，我们需要判断一个节点是否与另一个节点之间存在有向边，这一判断我们可以用邻接表实现。这一过程没有什么难度，但是我们发现，在建立邻接表的过程中，我们需要在循环内嵌套字符串匹配的操

作。事实上，在 C++ 的自带的字符串 = 操作符重载中，匹配字符串的操作需要将字符串遍历一遍来判断两个字符串是否相同，如果将这一匹配嵌套在循环内，效率将会被大大降低。因此，我们考虑使用一个 hash 函数将每个字符串编码为一个 int 值，从而避免在循环中对字符串进行嵌套循环遍历匹配。

#### 3.4.2 路径的回溯

在这个算法中，一个比较困难的问题是对最小编辑距离相对应的路径的记录，在本文中，我们采用一个矩阵来存储截止每个节点的最短编辑距离的节点来源。该矩阵也是以 de Bruijn 图的每个节点字符串作为横坐标，以目标字符串的所有前缀子串作为纵坐标，不断记录并更新每个节点在之前计算公式中的 src 值。

当我们确认整体的最短编辑距离对应的尾节点时，再通过这一矩阵记录的来源节点编号进行一步步回溯。这个大矩阵的存储是这一算法中存储空间占用的主要原因，其空间复杂度为 $O(m*n)$，且目前没有想到好的方法将其降低。

以上操作也可以优化为在矩阵中存储节点与相应来源节点之间不同的那个字符，由于 char 类型占用空间为 $1Byte$，而 int 类型占用空间为 $4Byte$，用这种方式可以使这个大矩阵的存储占用空间下降 4 倍。但由于时间原因，这一思想未在代码中实现。

### 3.4 其他细节

首先，有一些与 C++ 语言本身性质相关的细节，如在字符串链接过程中，应尽量写成 $str+="a"$ 而避免 $str = str + "a"$，$str = str + "a"$ 加的运算产生的是一个新的对象，再把结果返回，而$str+="a"$ 涉及到的应该是对象的引用，操作之后直接返回引用，避免了产生新的对象[3]。因此，两者的性能有一定的差距，这一差距将近是 20 倍。

其次，我们通过观察和探索 Task2 中的输入数据，发现了数据的一些特征，如：所有节点所构成的 de Bruijn 图是一条单链，不存在环状结构或树状结构。对于这一优越的数据特性，我们可以对算法进行一些改写，此为针对特殊数据的奇技淫巧，不具有健壮性，在此不多赘述。

### 3.5 实验结果

代码运行结果见 $./task2/task2.out$ 。未对具体时间进行精确的测量，但优化后的基于全局 BFS 的运行时间比简陋算法快了约 20 倍。

## 4. Task3的设计思路与优化过程

### 4.1 基本问题与解决思路

在 Task3 中，我们面临的主要问题是时间和空间上的不足。事实上，在时间方面，我们通过估算可以得出，如果沿用 Task2 的算法，其运行时间大概为 10 小时左右，这个问题并不是特别大。更为严重的是，受制于硬件设备，我们

的存储空间将会大大不足，尤其是我们的算法不可避免地至少要存储一个 400G 左右的大矩阵。在这种情况下，我们考虑将这一大矩阵分块从硬盘读写。

## 4.2 初始化矩阵的处理

事实上，事情并不像我们在 4.1 中提到的那么乐观。在算法沿用伊始，对初始化矩阵的处理就产生了问题。在 Task2 的算法中，我们保存了一个 $m \times n$ 大小的初始化矩阵，而在 Task3 的大量数据的情况下，这一矩阵大小为 $1{,}000{,}000 \times 100{,}000$，占用内存空间约 400G。根据算法，由于这一大矩阵是按列生成的，而后续的所有矩阵生成操作都需要对这一初始化矩阵里的数据按行进行调用，对这一矩阵的按行读写是不可避免的，按列生成按行存储的模式将造成很大的矛盾。我们试图先将这一矩阵完全生成好再存储为.csv 文件，再使用 Python 中的 Pandas 库对大型文件进行按行分块处理，但其效果非常差，预计运行时间（仅文件分块）约为 91 小时。在此情况下，我们只能考虑放弃对整个初始化矩阵的存储。

通过对 Task3 输入数据的观察和探索，我们发现节点字符串的长度非常短，而目标字符串非常长，我们想到是否能减短目标字符串的长度（即横坐标的个数）。在本题数据中，我们发现只要截取目标字符串的前 221 位，便足以为后续的计算提供数据。这样，我们就将 $1{,}000{,}000 \times 100{,}000$ 的初始矩阵降到了 $1{,}000{,}000 \times 221$，生成和读写这样大小的矩阵是非常快速的。

## 4.3 路径回溯矩阵的处理

本算法中另外一个产生巨大空间开销的东西就是路径回溯矩阵，该矩阵必须存储 $m \times n$ 大小的节点路径信息。对于这一回溯矩阵，不存在上述初始化矩阵的问题，我们可以按行分块存储，再按行分块读取。由于内存不够，我们不能将这一操作在内存上执行（如果实现了 3.4.2 中的 char 类型优化则可以在内存上执行），只能将这一矩阵按行分块存储到硬盘，在实验室内存 128GB 的服务器上可以运行。

## 4.4 实验结果

代码运行结果见 $./task3/task3.out$ 。其运行时间约为 10 小时，占用存储空间约为 400G。

# 5. 其他算法思想和未来工作

## 5.1 寻找匹配节点扩展方法

通过 de Bruijn 图对基因组进行索引，使用基于散列表的索引，对目标字符串隐含的节点子串进行匹配。随着索引，采用种子和扩展策略。在种子阶段，将来自输入数据的一系列种子（节点字符串）与参照基因组（目标字符串）进行匹配，从匹配的位置推断出一组推定的读取位置（PRPs）；在扩展阶

段，选取读取位置周围的局部序列对种子进行扩展，以构成完整的路径字符串[4]。

其具体实现见 $./deBGA$。

### 5.2 通过并发、并行进行优化

考虑用 openmp 来进行优化，考虑将代码写成多线程并发模式[5]。

考虑使用 Hadoop 或 Spark 等计算引擎和集群进行分布式并行运算。

## 6. 其他

感谢室友王丹青同学，在与她的讨论与交流中产生了很多的灵感，产生了项目的一部分思路。感谢张睿哲同学，向我提供了一部分算法思路，并提供了较强的测试与调试数据。感谢陈镜融同学，在并发、并行方面的优化思想受到了他的启发。

## 参考文献

[1] https://en.wikipedia.org/wiki/Edit_distance
[2] https://en.wikipedia.org/wiki/Levenshtein_distance
[3] http://blog.csdn.net/xiaobaismiley/article/details/25962483
[4] Bo L, Guo H, Brudno M, et al. deBGA: read alignment with de Bruijn graph-based seed and extension[J]. Bioinformatics, 2016, 32(21):btw371.
[5] https://github.com/crazyboycjr/algorithm-course-project