# 算法设计Project：de Bruijn图上的编辑距离

**14计算机科学与技术 14307130356 卢颖**

## 0. 背景定义

关于编辑距离、de Bruijn图的定义，以及具体的问题描述和数据见这里

## 1. 开发运行环境

3个Task的代码均采用C++进行编写。编译时task1和task2使用-o和-O2(可以不开)选项进行编译，task3需要采用-O3选项。

### 1.1 运行环境

- Task1, Task2：
  - 系统：macOS Sierra
  - 处理器： 2.7 GHz Intel Core i5
  - 内存： 8 GB 1867 MHz DDR3
- Task3:
  - 系统：Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-87-generic x86_64)
  - 处理器：AMD Opteron(tm) Processor 6378
  - 内存：由于优化不够充分，请准备200G的空间

### 1.2 编译方法

```
g++ task1.cpp -o task1 -O2 -Wall
g++ task2.cpp -o task2 -O2 -Wall
g++ task3.cpp -o task3 -O3 -march=native -mtune=native -Wall -std=c++11 -g -mcmodel=large
```

### 1.3 运行方法

task1和task2直接运行即可：

```
time ./task1
time ./task2
```

task3需要比较大的堆空间：

```
ulimit -s unlimited
time ./task3
```

### 1.4 运行时间

Task1可以在1秒内得到结果，Task2实际运行时间约为11秒，其中预处理建de Bruijn图约10秒。
Task3实际运行时间约为8小时。

# 2. 算法细节

## 2.1 Task1 问题描述与算法分析

### 2.1.1 问题描述

- 给出字符串 $A$ 与 $B$
- 求出它们的编辑距离，并输出编辑操作过程
- 其中
  - 字符集 $|\Sigma| \leq 4$
  - $len(a) \leq 10000$
  - $len(b) \leq 10000$

### 2.1.2 算法细节

#### 2.1.2.1 求解编辑距离

采用动态规划的方法来求解编辑距离。（又称[Wagner–Fischer算法]）

设$A, B$ 两个字符串长度为$m$和$n$，其中$n \geq m$。

$f[i][j]$ 表示字符串A的长度为i的前缀$A[1..i]$和字符串B的长度为j的前缀$B[1..j]$之间的编辑距离。字符串下标从1开始。

- 边界情况：
  - $f[i][0] = i, f[0][j] = j$. 也就是说，一个字符串和一个空串之间的编辑距离为字符串本身的长度。
  - $f[0][0] = 0$
- 对于$1 \leq i \leq n, 1 \leq j \leq m$，$f[i][j]$的状态转移方程如下：

$$f[i][j] = min \begin{cases} f[i-1][j]+1 & \\ f[i][j-1]+1 & \\ f[i-1][j-1] & A[i]=B[j] \\ f[i-1][j-1]+1 & A[i] \neq B[j] \end{cases}$$

状态转移方程中的四种情况分别对应4种操作：

1. $f[i][j] = f[i-1][j]+1$ 表明$A[1..i]$在第i位进行了一次DEL操作，再变成$B[1..j]$。
2. $f[i][j] = f[i][j-1]+1$ 表明$A[1..i]$在第i位进行了一次INS操作，再变成$B[1..j]$。
3. $f[i][j] = f[i-1][j-1]$ 表明$A[1..i]$在第i位没有进行操作，$A[i]$和$B[j]$匹配。
4. $f[i][j] = f[i-1][j-1]+1$ 表明$A[1..i]$在第i位进行了一次SUB操作，再变成$B[1..j]$。

转移结束之后，$f[n][m]$就是最终的编辑距离。

状态转移的时间复杂度是$O(mn)$, 空间复杂度是$O(mn)$

#### 2.1.2.2 输出编辑操作

为了输出编辑操作步骤，在进行状态转移的同时记录转移的路径$edit[i][j]$。**节2.1.2.1**中详细描述了转移方法对应的操作。从$edit[n][m]$一路往回倒推到$edit[i][0]$或者$edit[0][j]$, 注意边界情况,$edit[i][0]$表明首先需要对字符串A进行i次DEL操作，$edit[0][j]$表明首先需要对字符串A进行i次INS操作(也就是插入字符串B的前j个字符).

输出编辑操作的时间复杂度是$O(m+n)$,由于需要栈来保存编辑操作，这部分的空间复杂度是$O(m+n)$

总体来说该算法的时间复杂度为$O(mn)$，空间复杂度为$O(mn)$。

### 2.1.3 时空复杂度分析

Backurs A, Indyk P. 在[1]中证明了，如果强指数时间猜想( Strong Exponential Time Hypothesis) 是正确的，编辑距离问题算法没有强次二项式时间(Strongly Subquadratic Time)内的解。也就是说，如果SETH成立，则不存在常数$\delta$，使得编辑距离在$O(n^{2-\delta})$的时间内可计算。

编辑距离的计算有一种比较经典的算法[2]，对于长度为m和n的两个串（$n \geq m$），算法的时间复杂度为 $O(n \cdot max(1, \frac{m}{\log n}))$。该方法需要将状态转移矩阵切割成若干个小的矩阵。针对Task1的数据量，Wagner–Fischer算法实现和调试的难度都比较小，时间复杂度$O(nm)$和空间复杂度$O(nm)$都在可接受范围之内。

Wagner–Fischer算法简单直观，但是计算编辑距离的时间复杂度和空间复杂度还可以进一步的降低。Task3中采用了Task1的优化版本，详细描述见Task3的描述部分。

## 2.2 Task2 问题描述与算法分析

### 2.2.1 问题描述

- 给出一个 K 阶 de Bruijn 图，以及字符串 A
- 求 de Bruijn 图上的一条路径，使其代表的字符串与 A 的编辑距离尽可能小，并输出编辑过程
- 其中
    - $len(A) \leq 10000$
    - $k \leq 30$
    - 字符集 $|\Sigma| \leq 4$
    - de Bruijn 图中节点数量 $\leq 10000$

### 2.2.2 算法细节

对于de Bruijn图中一条路径上的所有点，起始点之外的点每个点都唯一对应字符集里一个字符(也即字符串的最后一个字符)。和普通的编辑距离是类似。于是我们考虑在de Bruijn图上做DP。

#### 2.2.2.1 原始状态转移方程

设$A$ 字符串长度$len(A) = m$，结点数为n，de Bruijn图阶数为K。

$f[i][j][l]$ 表示以第i个结点为结尾、长度为l的de Bruijin图上的路径到$A[1..j]$的最小编辑距离。结点编号i从0开始。

首先从一个最直观的想法开始。

- 初始条件：
  在de Bruijn图上做DP的边界条件比较复杂。因为de Bruijn图上起始节点对应的字符串长度为K，和A进行匹配的时候不能直接进行简答的单点匹配转移，需要进行一定的预处理。

我们可能会从第一个结点的字符串上删除、插入、替换一些字符，从而变换到A。边界条件实际上处理的就是字符串A和单个点上字符串进行匹配的问题。因此我们让所有结点i上的字符串$str_i$及其前缀和字符串A计算编辑距离，并且记录$g[i][j]$。$g[i][j]$表示以第i个结点为起点，还未进行转移时候，和$A[1..j]$的编辑距离。

- 边界条件：
  借助于初始条件$g[i][j]$，我们可以给出$f[i][j][l]$的边界条件
  $f[i][j][0] = edit[K][j] \quad (1)$

- 状态转移方程：
  由于采用邻接表建图时将边$(u, v)$按照起始点u存储在一起，而DP转移的时候需要沿着图上的路径转移，所以在Task2中采用从$f[i][j][l]$向后转移的方法来考虑状态转移方程。结点si为i的后向结点，也即在de Bruijn图上存在边$(i, si)$。记点si上字符串的最后一个字符为ch[si]。
  $0 \le i \le n, 1 \le j \le m,\ f[i][j][l]$，考虑i的所有后继结点si，以及结点i本身，状态转移方程如下：

$$
\begin{gathered}
f[si][j][l+1] = f[i][j][l] + 1 \\
f[si][j+1][l+1] = \min_{si} \begin{cases} f[i][j][l] + 1 & ch[si] \ne A[j] \\ f[i][j][l] & ch[si] = A[j] \end{cases} \\
f[i][j+1][l] = f[i][j] + 1;
\end{gathered}
$$

状态转移方程中的四行分别对应4种情况/操作：

1. $f[si][j][l+1] = f[i][j][l] + 1$ 表明在图上走了一步，也即A在j+1位进行一个INS操作得到ch[si]。
2. $f[si][j+1][l+1] = f[i][j][l] + 1$ 表明A[j+1]进行了一次SUB操作，变成ch[si]。
3. $f[si][j+1][l+1] = f[i][j][l] + 1$表明A[j+1]和ch[si]匹配，不进行操作。
4. $f[i][j+1][l+1] = f[i][j][l] + 1$ 表明A在第j位进行了一次DEL操作。

转移结束之后，$\min\limits_{i,l} f[i][m][l]$就是最终的编辑距离，用和Task1类似的方法倒退回去，就可以得到de Bruijn图上的路径字符串和操作序列。但是DP的时候为了方便，记录的操作序列其实都是从路径字符串转移到A的操作。为了实现方便，在具体代码中倒推得到了路径字符串后，调用了一遍Task1来计算操作序列。

#### 2.2.2.2 时空优化

- 我们可以观察到，$f[i][j][l]$中维度$l$只会从$l$转移到$l+1$,因此我们可以简单地将数组进行滚动，仅维护一个二维数组$f[i][j]$。
- 由于字符集$|\Sigma| \le 4$，所以实际上所有结点最多只有4的出度，并且有的结点可能是不可达的。因此每次扫描一遍所有的结点i来进行状态转移是不划算的。因此对于每个A中的长度j，都可以维护一个队列Q[j]来存放当前能够和A[j]进行匹配的结点i。只有当结点i可达，才能作为匹配点。

优化之后的转移方程为：

$$f[si][j] = f[i][j] + 1$$
$$f[si][j+1] = \min_{si} \begin{cases} f[i][j]+1 & ch[si] \neq A[j] \\ f[i][j] & ch[si] = A[j] \end{cases}$$
$$f[i][j+1] = f[i][j] + 1;$$

### 2.2.3 时空复杂度分析

此处仅对优化过的方法做分析。记m为A串长度,K为de bruijn图阶数。

主要的操作有以下几个：

1. 构建debruijn图（初始化）
2. 根据动态规划转移方程进行求解
3. 输出打印序列

#### 2.2.3.1 时间复杂度分析

其中建图需要比较每一对结点，时间复杂度为$O(n^2)$
预处理时间复杂度为$O(K^2n)$
动态规划求解的时间复杂度为$O(4Cmn)$，其中C为结点i重复入队的次数，实际运行中平均约2次。
打印序列的时间复杂度为$O(nm)$

优化过后状态转移的时间复杂度是$O(n^2+4Cmn)$, 空间复杂度是$O(mn)$

可能由于我这部分代码优化不够，常数比较大，建图就跑了大约10s，代码总共大约跑了11s。

## 2.3 Task3 问题描述与算法分析

### 2.3.1 问题描述

- 与Task2问题相同，但是数据规模较大。
- 提示：答案中编辑距离 $d < 0.3 * len(a)$，且编辑步骤均匀分布
- 数据范围
  - $len(a) \leq 100000$
  - $k \leq 30$
  - 字符集 $|\Sigma| \leq 4$
  - de Bruijn 图中节点数量 $\leq 1000000$

### 2.3.2 算法细节

Task3和Task2相比区别在于：

1. 数据规模的扩大。m和n分别变成了100000和1000000
2. 编辑步骤均匀分布。
3. 编辑距离和A的长度相比，存在一个较小的上限
   基于上述几点，考虑对task2中的算法进行改进。

#### 2.3.2.1 编辑距离上线调整

考虑到提示中：编辑距离 $d < 0.3 * len(a)$，因此将状态转移中所有的最大值都设为0.3*len(a)

#### 2.3.2.2 状态转移优化

观察状态转移方程:

$$f[si][j]=f[i][j]+1$$
$$f[si][j+1]=\min_{si}\begin{cases}f[i][j]+1 & ch[si]\neq A[j]\\ f[i][j] & ch[si]=A[j]\end{cases}$$
$$f[i][j+1]=f[i][j]+1;$$

可以发现$f[i][j]$的值和第一维在i+1以后的数据都没有直接的联系，仅转移到i和i+1。因此可以将f数组进行滚动。减低空间复杂度到$O(2n)$

#### 2.3.2.3 编辑序列求解优化

在Task 3中，对求编辑距离的Task 1进行了优化。
观察状态转移方程：

$$f[i][j]=min\begin{cases}f[i-1][j]+1 & \\ f[i][j-1]+1 & \\ f[i-1][j-1] & A[i]=B[j]\\ f[i-1][j-1]+1 & A[i]\neq B[j]\end{cases}$$

可以发现$f[i][j]$的值和第一维在i-2以前的数据都没有直接的联系，仅从i和i-1转移过来。因此可以将f数组进行滚动。减低空间复杂度到$O(2m)$

### 2.3.3 时空复杂度分析

优化过的代码主要在空间复杂度上有了比较大的提升。
Task2的空间复杂度比较高，在实现中需要开5个大小为$m\times n$的数组，而经过优化的代码只需要3个大小为$m\times n$的数组。由于进行了空间存储结构上的优化，时间上也有了一定的提升。

虽然从时空复杂度上来看并没有很明显的区别。但是由于m和n都非常大，常数上的优化都对代码的效率产生了非常大的影响。

# 3. 总结和讨论

本次算法PJ通过不断优化Task1和Task2的方法从而得到一个Task3的解决方案。最终Task1计算编辑距离的时间复杂度为$O(nm)$，空间复杂度为$O(2m)$，Task3时间复杂度是$O(n^2+4Cmn)$, 空间复杂度是$O(mn)$

**更多的改进点**
虽然最终完成了3个Task的任务，但是task3的时间复杂度和空间复杂度还是非常的高，需要非常大的内存和较长的时间运行才能得到结果。以下为几个可能进行优化的点：

1. 对建图过程进行优化。
2. 滚动数组时应当将比较小的那一维当成第二维
3. 字符集只有4，可以考虑从这方面去除冗余
4. 算法并没有利用“编辑步骤均匀分布”这个条件，可以对编辑步骤进行统计进行优化。
5. 设计新的算法，引入随机性，通过多次的快速求解得到一个比较接近最优解的方案。

# 4. 感谢

非常感谢王丹青同学和刘超颖同学提供的task2测试数据。非常感谢张睿哲同学和陈镜融同学提供的task3算法优化思路和优化编译方法。

# 5. 参考文献

[1]: Backurs A, Indyk P. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)[C]//Proceedings of the forty-seventh annual ACM symposium on Theory of computing. ACM, 2015: 51-58.

[2]: Masek W J, Paterson M S. A faster algorithm computing string edit distances[J]. Journal of Computer and System sciences, 1980, 20(1): 18-31.

[3]https://en.wikipedia.org/wiki/Wagner%E2%80%93Fischer_algorithm

[4]https://en.wikipedia.org/wiki/Edit_distance