算法设计 Project 报告

de Bruijn 图上的编辑距离

14307130167 蒋骐泽

Task 1

显然用 DP 做。$f[i][j]$表示a串的前i个字符和b串的前j个字符的编辑距离。转移方式有：

1. $f[i][j-1]+1$ 表示在a第i个字符后面添加一个字符，这个字符即为b串第j位的字符。
2. $f[i-1][j]+1$ 表示将a串的第i个字符删去。
3. $f[i-1][j-1]$ 当a串的第i个字符和b串的第j个字符相等，此时不需要任何操作。
4. $f[i-1][j-1]+1$ 当a串的第i个字符和b串的第j个字符不相等，表示将a串的第i个字符替换成一个字符，这个字符即为b串的第j个字符。

时间复杂度$O(nm)$，n和m分别为字符串a和b的长度。由于要输出方案，另使用数组$fa[i][j]$表示$f[i][j]$是从哪个格子转移过来的。由于只有三种转移的格子，所以在$fa[i][j]$中记录$0,1,2$对应3种情况。其中，对应于$f[i-1][j-1]$的情况包含 3 和 4 两种子情况，这只需要判断a串i位置和b串j位置的字符是否相同即可区分。

```
1   Function Task1(a, b, n, m)
2       f[0][0] ← 0
3       for i ← 1 to n
4           f[i][0] ← i
5       for j ← 1 to m
6           f[0][j] ← i
7       for i ← 1 to n
8           for j ← 1 to m
9               if a[i] ← b[j] then subcost ← 0 else subcost ← 1
10              f[i][j] ← min{ f[i-1][j]+1, f[i][j-1]+1,
                               f[i-1][j-1]+subcost }
11              fa[i][j] ← which transferred to f[i][j]
12      step ← get edit step by fa
13      return f[n][m], step
```

Task 1 伪代码

Task 2

先考虑一种简单情况，数据的$de\ Bruijn$图有m个节点，并且是一条链，即除了一个节点没有前驱、一个节点没有后继以外，所有节点有且仅有一个前驱和后继。在这个情况下，最长的字符串显然就是整条链组成的字符串，长度为$m+k-1$。我们把这个字符串当做b串。我们可以发现，一条$de\ Bruijn$图上的路径即对应b串的一个长度大于等于k的子串。我们可以使用如下方法计算：首先对于每个节点p所代表的长度为k的字符串，我们将其和a串求最短编辑距离，$f_p[i][j]$表示a串的前i个字符和这个节点p对应字符串的前j个字符的编辑距离。对于一个节点，计算的时间是$O(nk)$，共有m个节点需要处理，所以预处理的总时间为$O(nmk)$。

```
1    Function Task2_Preprocess(a, b, n, m, k)
2        for i ← 1 to m
3            f_p ← array f in Task1(a, b[i], n, k)
4        return f_p
```

Task 2 预处理伪代码

接下来，我们用$g[i][j]$表示a串的前i个字符和b串中以第j个字符结尾的，长度大于等于k的所有子串的编辑距离中最小编辑距离。转移方式与两个串之间求编辑距离的转移类似。再次基础上再加上一个转移：

1. $f_{j-k}[i][k]$ 当p大于等于k时

由于之前转移而来的是以第j个字符结尾、串长度大于k的所有子串的最小编辑距离，因此用这个转移将串长度刚好为k的结果转移，使得$g[i][j]$满足我们的定义。现在考虑是图的情况。与一条链不同，每一个节点可以有多个前驱和后继。显然，我们只需要对于第二种 第三种转移对于其每个前驱都做一遍即可。这部分时间复杂度为$O(nm)$，总时间复杂度为$O(nmk)$。

```
1    for i ← 1 to n
2        for j ← 1 to m
3            g[i][j] ← min{ g[i-1][j]+1, g[i][j-1]+1,
                            g[i-1][j-1]+subcost, f_{j-k}[i][k] }
```

Task 2 简化问题的转移伪代码

现在考虑 DP 顺序的问题，由于一个点必须等到其前驱完成计算，这个点的转移才有意义，所以需要使用一个合理的顺序。首先，由于a串显然是有序的，DP 的二重循环我们先枚举a串前缀长度。对于内部循环，我们可以采用这样的顺序：首先将所有没有前驱的点加入队列中，它们显然是可以进行计算的。接下来，当一个点计算完后，如果这个点的答案发生了更新，那么对于这个点的后继点的答案也有可能发生更新，因此将其所有后继点加入队列中，然后将这个点从队列中删除。这样做对于一种特殊情况-环不能处理，因为环上每个点都有前驱。如果没有一条路径可以通过某一个起始点走到环上，那么这个环上的点将永远不会被加入队列。因此，在队列为空后，我们检查所有节点是否都进入过队列进行计算，对于没有进入过队列的点随机选择一个加入但队列并重复上述过程直到所有点都进行过了计算。幸运的是，在 Task 2 中数据并没有环，在 Task 3 中数据虽然有环但是环可以通过起始点走到，不进行上述处理也没有关系。

```
1   Function Task2_main(a, b, n, m, k)
2       f_p ← Task2_Preprocess(a, b, n, m, k)
3       startlist ← points set which have no father
4       for i ← 1 to n
5       processqueue ← startlist
6           for j in processqueue
7               g[i][j] ← min{f[i − 1][j] + 1, f_{j−k}[i][k]}
8               for jfa in father of j
9                   g[i][j] ← min{g[i][j], g[i][jfa] + 1, g[i − 1][jfa] + subcost}
10              fa[i][j] ← which transferred to g[i][j]
11              if g[i][j] is updated
12                  son of j add to processqueue
13      resm ← [0, m) with minimun g[n][resm]
14      step ← get edit step by fa, resm
15      return g[n][resm], step
```

Task 2 主函数伪代码

Task 3

与 Task 2 的区别即为n和m分别扩大了 10 倍和 100 倍。如果依旧使用 Task 2 的时间复杂度为$O(nmk)$的做法无法在合理的时间内计算完成。同时，计算需要的空间复杂度为$O(nmk)$，达到 TB 级，几乎没有设备可以拥有如此巨大的内存。接下来我们分别来解决这两个问题。

对于时间复杂度的优化，考虑到 Task 2 中使用的方法中，复杂度来自于最开始的预处理 DP，而不是之后的最优解计算 DP，我们需要想办法将预处理复杂度中比之后计算多出来的k给消去。在我们的预处理中，最终被利用的结果为每个节点上的 s 串和a串的每个前缀的最短编辑距离。我们可以注意到，当b串是a串的一个子序列，当且仅当的编辑距离为$length(a)-length(b)$。而如果b串已经是a串的前缀的子序列，显然b串必为a串的子序列，即两者的编辑距离为$length(a)-length(b)$。因此，当s串和a串的某一个前缀a'的编辑距离为$length(a')-length(s)$的时候，即s为a'的一个子序列，那么之后的 DP 都是没有必要的，因为之后与a串的另一个长度大于a'前缀a''的最短编辑距离必为$length(a'')-length(s)$。

现在考虑在这个优化下的时间复杂度。浏览数据后可以发现，虽然数据不是纯随机数据，但是其分布和概率和纯随机较为接近。我们以纯随机数据来考虑复杂度。由于a串长度为 100000，s串长度为 30，若s串为a串长度为 3000 的串的前缀，那么可以认为预处理的复杂度降为了$O(nm)$。经过简单计算，s串不是a串长度为 3000 的串的前缀的概率小于10^{-300}。实际测试中，前缀最长到 350 左右时所有s串已均为其子序列，因此复杂度成功降为了$O(nm)$。

```
1   Function fasterpart(a,b,n,m)
2       … same as Task1,line 2 to line 6
3       for i ← 1 to n
4           if f[i−1][m] ← i−1−m
5               f[i][m] ← i−m
6               continue
7           … same as Task1,line 8 to line 10
8       return f[∗][m]
9   Function Task3_Preprocess(a,b,n,m,k)
10      for i ← 1 to m
11          f_p ← fasterpart(a,b[i],n,k)
12      return f_p
```

Task 3 预处理加速伪代码

对于空间复杂度，由于转移时只和前一次动态规划的结果相关，因此我们可以对于动态规划数组使用滚动数组，这样预处理数组大小降到了$O(nk)$，主数组大小降到了$O(n)$。然而，由于最后需要给出方案，对于记录我们是从哪个位置转移过来的数组无法省略。考虑到一个点最多只有四个父亲，每个父亲最多有两个格子转移，再加上两个不是从父亲来的转移共有 10 种不同的转移。对其进行编号，只需要$4bit$即可。这样，转移方案需要$100000\times1000000\times4\div8\approx50GB$空间。如果拥有一台内存在$64GB$及以上的计算机，就可以在内存中完成存储计算。由于我没有这种设备，因此选择将每层的转移分层存储至硬盘。而在恢复时，和动态规划数组使用滚动数组原因类似，当移动到第i层时，就永远不会移动到$i+1$层及以上，可以将上面层的内存释放并载入新的一层转移数组。这样，我们只需要$O(nk)$的内存空间和$50GB$的磁盘空间。由于只是简单的数组重复利用和文件存储，主函数与 Task 2 基本相同，故省略伪代码。