算法 Project 报告

14307130245 张剑锋 2017

题目

Task 1: 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

Task 2: 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

Task 3: 同 Task 2

数据范围

共同: $|\Sigma| \leq 4$

Task 1: $len(a) \leqslant 10000$, $len(b) \leqslant 10000$

Task 2: $len(a) \leqslant 10000$, 节点数量≤ 10000, $k \leqslant 30$

Task 3: $len(a) \leqslant 100000$, 节点数量≤ 1000000, $k \leqslant 30$, 编辑距离 $d<0.3*len(a)$，且编辑步骤均匀分布

分析

Task 1

这是一个典型的动态规划问题。定义函数d(i, j)为a[0...i-1]和b[0...j-1]之间的编辑距离，可以写出状态转移函数：

```
d(i, j) = min {
            d(i-1, j-1) if a[i-1] == b[j-1]  // 直接匹配
            d(i-1, j-1) + 1,                 // SUB (i-1) b[j-1]
            d(i-1, j) + 1,                   // DEL (i-1)
            d(i, j-1) + 1                    // INS i     b[j-1]
          }
```

边界条件为：

```
d(i, 0) = i // DEL 0..i-1
d(0, j) = j // INS 0 b[0..j-1]
```

假设 n=len(a), m=len(b) 。状态空间大小为nm，所以时间复杂度和空间复杂度均为O(nm)，约10^8数量级。

Task 2

这个task可以继续依照动态规划的思路。定义d(i,j)是字符串a[0...i-1]匹配到de Bruijn图中j点时最少的修改次数。则状态转移函数变为：

```
d(i, j) = min {
            dist(a[0..i-1], str[j]),                        // a[0..i-1] 直接与 j 匹配
            d(i-1, j) + 1,                                  // DEL (i-1)
            d(i, prev(j)) + 1,                              // INS i     str[j][k-1]
            d(i-1, prev(j)) if str[j][k-1] == a[i-1],       // 恰好匹配
            d(i-1, prev(j)) + 1 if str[j][k-1] != a[i-1]    // SUB (i-1) str[j][k-1]
          }
```

其中dist(a, b)表示字符串a和b的编辑距离，prev(j)表示j的任何前趋节点。
边界条件为 d(0, j) = k 。

假设 n=len(a), m=节点数量 。
计算dist部分的时间复杂度总共 O(nmk) ，计算d的时间复杂度为 O(nm|Σ|) ，总时间复杂度是 O(nm(k+|Σ|)) 约为 O(nmk) 。
为了进一步降低时间复杂度，这里可以加入一个优化：当 dist(a[0..i-1], str[j]) = i-k 的时候，这时候说明直接匹配的所有操作都是DEL。继续增加i，dist的结果也不会比其它项更优了，那么可以停下对dist的计算。加入这个优化后dist部分的复杂度有一定下降。如果字符串中字符出现是随机的，那么复杂度变为 O(k|Σ|*mk)+O(nm|Σ|)=O((k^2+n)m|Σ|) ；由于 k^2<<n ，所以大概是 nm=10^8 数量级。

空间上，状态转移按照i递增的顺序计算，记录当前状态的空间复杂度仅为O(m)。在此之上若要记录修改过程，则需要O(nm)的空间记录。

Task 3

根据Task 2的分析，直接运行是nm=10^11数量级。根据Task 2程序运行时间，可以估算直接运行Task3的时间是数个小时。但是直接运行Task 2的程序，机器内存就不够了。考虑到在计算状态转移的过程中不需要历史修改过程，因此对于O(nm)的修改过程的空间可以放到硬盘上，内存占用只需O(m)。进一步，由于题目提示修改次数不超过30000，对于步骤数大于30000的状态就可以不记录了，再次减少了需要的硬盘空间。

实现

Task 1

使用Haskell写了简单实现，自定义了一个数据类型用于记录修改操作。由于Haskell约30倍的运行时开销运行时间为35.21s。

编译： `ghc -O2 task1.hs`

运行： `./task1 < task1.in`

Task 2

使用Go写了实现，耗时14.00s，内存占用1.3GiB。

编译： `go build task2.go`

运行： `./task2 < task2.in`

Task 3

使用C++写了一个类似Task 2的实现，并简单优化了一下。对于记录修改过程的部分，单独写了一个记录函数，并且使用了一个线程专门负责把修改记录存储到硬盘上。为了节省硬盘空间，使用minilzo压缩了硬盘文件，但是对于存储的数据结构没有进行压缩。实验程序运行时间约5h，内存占用2GiB，占用硬盘空间少于500GiB。

编译：

```
g++ -Ofast -mtune=native -pthread -lminilzo -o task3 task3.cc
g++ -lminilzo -o extractor extractor.cc
g++ -o genanswer genanswer.cc
```

运行：

```
./task3 < task3.in # 输出记录到当前目录，记下最优解记录编号
./extractor <reference> > logresult # 从磁盘提取修改记录
./genanswer task3.in < logresult # 生成答案
```

其它可能的优化和算法

当前实现Task 2 & 3的修改过程记录记录了上一个修改的指针，但是这其实是不需要的，因为节点度数有限所以可以只记录前一个节点字符串的首字母。如果进一步压缩存储应该能把Task3的硬盘需求降低至100GiB以内。

另一方面，由于期末时间关系没有仔细优化算法运行的常数。如果减少堆空间的分配释放，并加强程序局部性提高cache命中率应该能再把程序运行速度提高数倍。

观察Task 3的输出结果，字符串中随机位置连续多个字符都只有少数修改（连续25个字符基本上只有少于5个修改）。因此我认为Task3可以利用搜索+剪枝达到接近线性的效率。基本思路如下：

1. 随机在字符串中选取一个位置，取连续25个字符。
2. 将这个子串与图中每个节点计算编辑距离，选择距离最小的点作为参考。
3. 从这个点出发，拓展搜索树。搜索树每个节点下面都有INS、SUB、DEL三种转移的情况。搜索树往下拓展5层，得到243个叶子。
4. 根据之前分析，这些叶子中最好情况应该已经在字符串中往前匹配了约25个字符。选取最好的一种情况往回退4步的节点，继续第3步。
5. 如果搜索树往前拓展一段情况都非常糟糕（修改次数/子串长度增加），那么就回溯到之前某一个点选取次优的孩子继续搜索。

   从后往前搜索和从前往后对称。

   我认为使用这种方法可以在接近线性的时间内（搜索到的路径不会离开近似最优路径5步）得到接近最优的结果。