算法设计 Project：de Bruijn 图上的编辑距离

陈镜融
chenjr14@fudan.edu.cn

2017 年 6 月 23 日

1 Abstract

题目分为三个 tasks，task1 要求两个给定字符串的编辑距离；task2 要求在 k 阶 de Bruijn 图上找到一条路径，其对应的字符串和给定字符串的编辑距离最小；task3 是 task2 的数据范围放大版本。三个 task 都要输出具体方案。题目具体描述和数据在 http://datamining-iip.fudan.edu.cn/ppts/algo/pj2017/

这次大作业，我完成了全部三个 tasks，并尝试对 task3 进行了一定的优化。所有代码可以在 https://github.com/crazyboycjr/algorithm-course-project 上找到。

2 Compile

```
g++ edit_distance1.cpp -o edit_distance1 -O2 -Wall
g++ edit_distance2.cpp -o edit_distance2 -O2 -Wall
g++ edit_distance3.cpp -o edit_distance3 -O3 -march=native
                                    -mtune=native -Wall -std=c++11 -g -mcmodel=large
```

其中 `edit_distance3.cpp` 的编译可以增加`-fopenmp` 选项，不过由于种种原因 (估计是线程同步需要等待时间)，使用 openmp 并没有能提高速度。可以考虑接下来将整个程序静态编译并加上`-pg`，然后用 `gprof` 看一下分析结果。

3 Run

```
ulimit -s unlimited
time ./edit_distance3 < task3.in > task3.out
```

注意：`./edit_distance3` 的运行需要较大的堆空间，实际运行大约 60GB 左右，整个程序运行时间在 4h 左右（常数优化前）。另外可以用 `taskset` 把进程绑到一个处理器上执行。

4 Write-up

4.1 task 1

经典的 levenshtein distance 问题，求一个字符串 A 经过增加一个字符，删除一个字符，替换一个字符变成字符串 B 的代价。每个操作代价为 1。

用 $f[i][j]$ 表示将字符串 A 的前缀 $A[1..i]$ 变成字符串 B 的前缀 $B[1..j]$，最少需要多少次操作。

为了方便说明，令 $n = len(A)$, $m = len(B)$。$f[0][i]$ 和 $f[i][0]$ 表示从空串变成另一个串的代价。

显然有 $f[i][0] = i$; $f[0][j] = j$; 特殊的有 $f[0][0] = 0$;

接下来为了计算 $f[i][j]$，有最多四种可能的转移

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

第一种转移表示能直接匹配，不需要额外代价，剩下三种转移分别表示删除 $A[i]$，在 A 中增加字符 $B[j]$，替换 $A[i]$ 为 $B[j]$。转移结束后 $f[n][m]$ 就是最终答案。

为了输出方案，我们只需要记录转移的路径。另 $opt[i][j] = \text{NOP/DEL/ADD/SUB}$，分别对应四种转移情况，在实际发生转移时，更新 $opt[i][j]$ 为对应的值。这里要注意初始状态 $f[i][0] = i$ 的转移可以看成是由 $f[i-1][0]$ 转移而来，因此对应 DEL 转移，$f[0][i]$ 对应 ADD 转移。

有了 $opt[i][j]$ 的值，就可以通过 $opt[n][m]$ 往前倒推，每次根据当前是由上次那种转移过来，可以知道 n,m 的变化。同时可以知道当前在字符串 A 上做的操作是什么。这部分可以递归完成。当 $n + m == 0$ 时，说明转移倒推到最开始了。因为答案不会超过字符串长度，因此递归深度不会特别大。

时间复杂度 $O(n^2)$，空间复杂度 $O(n^2)$，其中 n 为字符串最大长度。这个算法的空间可以继续优化，我们往下看

4.2 task 2

这个任务和 task1 任务不同。task2 给出了一张图，图上每个节点是一个长度为 $k(k \leq 30)$ 的字符串，两个节点之间存在有向边，当且仅当第一个字符串的长度为 $k-1$ 的后缀，和第二个字符串长度为 $k-1$ 的前缀完全相同。可以想象，图上一条长度为 l 的路径，构成了一个长度为 $k+l-1$ 的字符串。现在给出字符串 A，和 m 个长度为 k 的字符串，希望在图中找到一条路径，使得路径所对应的字符串和 A 的编辑距离最小。

首先说建图,这个过程没有什么难度,可以简单考虑用两个 `unordered_map<string, vector<int>>[2]` 表示有哪些编号的字符串以 key 为前缀或者后缀，具体来说 `M[0][prefix]` 表示以字符串 prefix 为前缀的字符串编号列表，`M[1][suffix]` 表示以 suffix 为后缀的字符串编号列表。由于题目中约束字符集大小只有 4，在实际数据中没有重复字符串出现的情况下，这个列表长度不会超过 4。所以存图也可以用前向星，邻接表之类的方法存 (这里用什么方法存图会在后面常数优化章节进行比较)。

在不考虑空间的情况下，这个题目仍然可以考虑用动态规划，因为可以发现在图上走一步，其实只增加了一个字符，这与第一问中字符串上走一步，本质是没有什么不同的。于是我们修改转移状态，用 $f[i][j][k]$ 表示在字符串 A 的前缀 $A[1..i]$，变化成一个在图上走了 k 步的字符串，这个字符串最后停在 j 这个节点上，最少需要多少代价。

考虑初始状态和转移，我们发现在这个 dp 过程中，转移每次增加一个字符，而初始状态是上来一个字符串作为开头，没法归类到转移当中。于是初始状态需要提出来单独计算。

对于初始状态，实际上我们需要计算 $f[i][j][1]$，其中 i 和 j 是变值，即对每个字符串 j，计算任意长度的前缀和变换到字符串 j 的代价。这一部分我们可以调用 task1 中实现的函数，不做任何优化考虑，时间复杂度为 $O(nmk)$，其中 n 为字符串 A 的长度，m 为图中节点数，k 为每个节点上字符串的长度。

接下来仍然考虑四种转移，为了方便起见，用 $tlen$ 表示题目描述中的 k，即节点上每个字符串的长度，用 $ns[j]$ 表示第 j 个字符串

$$f[i][j][k] = min \begin{cases} f[i-1][y][k-1] & , A[i] == ns[j][tlen] \\ f[i-1][j][k-1] + 1 \\ f[i][y][k-1] + 1 \\ f[i-1][y][k-1] + 1 \end{cases}$$

y 表示字符串 j 的前一个可能的节点，转移的时候取较小值转移。类似的，第一种对应直接能匹配，第二种对应删除字符 $A[i]$，第三种对应在 A 中增加字符 $ns[j][tlen]$，第四种对应替换。

在这种情况下，答案等于这个数组中的最小值。但实际上我们并开不下空间存储这样的状态，于是考虑修改一下状态的含义。$f[i][j][k]$ 即路径不超过 k 的所有停在 j 上的字符串，变化到 A 的最小代价，相当于对前面的状态做一个继承。于是多一种转移 $f[i][j][k] = f[i][j][k-1]$ 即可。这样我们可以发现 $f[i][j][k]$ 一定是由 $f[x][y][k-1]$ 转移而来，于是第三维可以滚动。

但实际上，仅仅是上面这样的考量是不够的。考虑这样一种情况，有三个字符串分别为”abb”，”bbc”，”bbb”，字符串 2 在和 A 的某个前缀比较时，可能从字符串 1 转移过来，也可能从字符串 3 转移过来，而不恰当的转移顺序会丢掉 1 先增加一个字符转移到 3，再由 3 增加一个字符到 2 的情况。作为不，考虑转移顺序的一个 workaround，考虑图中可能出现环的情况，对 $f[i][x][]$ 最少需要重复转移 $tlen$ 次，才能保证 dp 的正确性。

对于方案的输出，则需要多记录上一次是从哪个节点转移过来的，一样可以递归输出。

因此该算法的时间复杂度为 $O(nmk)$，空间复杂度为 $O(nm)$, 对于 task2 的数据，可以在十几秒级别的时间出解。

4.3 task 3

面对 task3 的数据规模，我们主要解决 task2 算法时间和空间上的不足。
先考虑空间上的优化

1. task1 当中的 dp 数组是可以滚动的，所以这里只需要 $O(m)$ 的空间

2. task2 中的 dp 数组，第一维也是可以滚动的，于是这里的空间也只有 $O(m)$。

3. 对于方案的输出，由于一共有 $n * m$ 的状态，所以 $n * m$ 的空间必不可少，对于 $n = 100000$, $m = 1000000$ 来说，$n * m$ 个 int 需要大约 400GB 的内存空间，实际上，每种转移只有 4 种情况，对应 2bit，而转移要记录上一个结点，又因为字符集大小只有 4，且没有重复字符串，所以只关心上一个字符即可，也需要 2bit，所以对于转移方案的记录，一个需要 4bit，相比之前用 int 来存，我们节省了 8 倍的空间，于是这里空间大约需要 50GB，在一台小型的服务器上已经可以承载。

4. 另外还有 dp 初始化需要记录结果，一共有 $n * m$ 个结果需要保留，而实际上，由于数据随机，可以发现在 $ns[i]$ 已经在 A 串中完全出现之后，其后面的值不需要再次计算。这里我们用 m 个 vector，$bound[i][j]$ 表示第 i 个字符串，和 A 串的前缀 $A[1..j]$ 的编辑距离，对于 $bound[i][j] + tlen == j$ 之后的情况，可以不用存储。对于随机数据来说，在长度为 100 多的字符串中几乎就出现了 $ns[i]$。所以这不但节省了空间，而且节省了 dp 初始化的计算时间。初始化时间和空间都降低为了 $O(4mk^2)$。

在空间优化完成后，程序就已经可以跑了，经过估计大约需要 60h 可以出解。然而时间上任然可以继续优化。

1. 观察 task2 中 $f[i][j][k]$，已经对 k 这一维的转移结果进行继承了。为何不干脆把这一维直接优化掉？实践发现这样是可行的。于是记录状态的数组变成了 $f[i][j]$

2. 实际上当一个状态 $f[i][j]$ 在某次转移后值已经不变了，那么由它去再次转移后面的值没有意义。所以我们可以考虑用两个队列，每个队列里存放待转移的节点。第一个队列我们依次计算转移，当一个节点转移成功后，将其后继状态，也就是这个结点可以去更新的状态，加入到下一个队列中，等待下一轮转移。经过这个优化，结果发现从原来每个结点需要迭代 $tlen$ 次，降低到了平均迭代 2 次左右。在这个优化加上之后，跑出结果大约只需要 4h。

时间复杂度 $O(4mk^2 + 4Cnm)$，空间复杂度 $O(nm)$。这里的 C 最坏能达到 k，但实际情况下 C 只有 2 左右。

4.4 constant optimization

为了进一步优化，我们可以考虑从常数方面，和利于多线程并行方面进行。

1. 存图方面，因为需要大量次数的枚举图上相邻边，可以考虑用连续空间的存储代替前向星存储。

2. 考虑 cpu cache，dp 中的滚动数组应当将小的一维作为第二维。这样可以有效加快计算的速度。实测光是初始化阶段就快了几十倍。

3. 考虑用 openmp 来优化，这部分我做了一些工作，但效果不理想，我觉得主要有三个原因。

   (a) 第一个原因是每个线程做的任务太少，反而线程之间来回取任务占用了更多时间，这个已经通过增大每个线程执行的工作量来解决了，但也只能是一个模糊的数值

   (b) 第二个是线程之间同步，比如分出 12 个线程执行一个 for 循环，但 for 循环结束之后，需要等待所有线程都执行完，这里应该会花费不少时间

   (c) 第三个是 for 循环虽然并行了，但访问内存速度是瓶颈，由于数据巨大，多线程得写内存范围伤害了 cache 的局部性，导致速率得不到提高。

5 Others

感谢姜峻岩同学提供的数据，对我调试和测试提供了极大的帮助。