研究生算法课课堂笔记

上课日期：2016 年 12 月 22 日星期四　第(1)节课

组长学号及姓名：金丰羽 1601214459
组员学号及姓名：史博　1601214557

注意：请提交 Word 格式文档

本节课先回顾了之前学习过的 Sequence alignment 问题及其动态规划算法，然后就带负权边有向图中的最短路算法（Bellman-Ford 算法）进行了介绍及讨论。

一、Sequence alignment 问题的回顾：

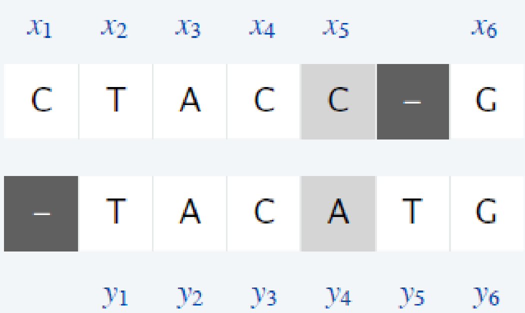
1、Sequence alignment 的问题描述

给定序列 $X = x_1x_2 \ldots x_m$ 以及 $Y = y_1y_2 \ldots y_n$。定义序列 X 和 Y 的一个匹配 M 为有序对$(x_i，y_j)$的集合，且满足

（1）特定的 x_i 或 y_j 在 M 中最多出现一次（可以不出现，即允许置空）；

（2）若$(x_i，y_j)\in M$，则不存在 u, v 使得 $u < i$，$v > j$ 并且$(x_u，y_v)\in M$（即必须按顺序匹配）

课件上给了如下匹配的例子：



an alignment of CTACCG and TACATG:

$$M = \{ x_2–y_1, x_3–y_2, x_4–y_3, x_5–y_4, x_6–y_6 \}$$

现给定匹配 M 的代价：

$$\text{cost(M)} = \sum_{(x_i,y_j)\in M \text{且} x_i \neq y_j} \alpha_{ij} + \sum_{i:x_i \text{未匹配}} \delta + \sum_{j:y_j \text{未匹配}} \delta$$

其中α_{ij}以及 δ 均为已知常数。上图的例子中$x_5 - y_4$就是一对误匹配。

求两个序列的最小匹配代价。

2、Sequence alignment 的求解思路

运用动态规划算法求解。

定义状态：$\mathrm{OPT}(i,j)$表示前缀序列 $x_1x_2 \ldots x_i$ 以及 $y_1y_2 \ldots y_j$ 的最小匹配代价。

初始条件：$\mathrm{OPT}(0,j) = j\delta$， $\mathrm{OPT}(i,0) = i\delta$（一个序列为空，显然另一个序列中的元素全部为未匹配）

转移方程：分三种情况，（1）x_i 和 y_j 匹配；（2）x_i 不和 Y 中的元素匹配；（3）y_j 不和 X 中的元素匹配。遍历这三种情况，求匹配代价最小值。综合初始条件以及转移方程，$\mathrm{OPT}(i,j)$有如下表达式：

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i=0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j=0 \end{cases}$$

要求的最小匹配代价：$\mathrm{OPT}(m,n)$。

具体实现可以朴素的开一个二维数组存储所有的$\mathrm{OPT}(i,j)$。如果要得到具体的匹配 M ，可以再开一个二维数组$\mathrm{Match}(i,j)$记录每个$\mathrm{OPT}(i,j)$是由 $\mathrm{OPT}(i-1,j-1)$，$\mathrm{OPT}(i-1,j)$，$\mathrm{OPT}(i,j-1)$(分别对应转移方程中的(1)~(3)情况)中的哪个转移过来的。OPT 数组是 m*n 维的，每个元素常数时间可以计算的解，所以总的时间复杂度是 O（mn）。

由于每个$\mathrm{OPT}(i,j)$只依赖前面求出的三种情况，所以可以把 m*n 的二维数组$\mathrm{OPT}(i,j)$ 压缩成两个长度为 m 和 n 的一维数组，分别记录对于 Y 序列匹配到 j-1 时 X 序列中每个元素的匹配代价以及 X 序列匹配到 i-1 时 Y 序列中每个元素的匹配代价（其实就是只记录前一轮的匹配代价）。对于最小匹配代价的求值没有影响，但是这样很难恢复匹配 M 的结构。

通过精巧设计的 Hirschberg’s algorithm 可以在内存为 O(m + n)的情况下还原匹配 M，但是比较复杂而且泛用性不广，课上也没具体解释。有兴趣的可以参考

课件。

二、Bellman-Ford 算法

Bellman-Ford 算法和 Dijkstra 算法类似，也是求有向图中点到给定点的最短路的算法（按照 algorithm design 课件上的定义，是到给定汇点的最短路。当需要求到给定源点的最短路时，可以将输入的有向图中的边保持权重全部反向，则转化为到给定汇点最短路的问题。也可以参考 Bellman-Ford 算法的思路重新定义状态，按照类似的方式进行状态转移）。它的优点是能够处理图中出现负权边的情况，并且本质上是一个“局部算法”，从而容易利用分布式计算加速。

1、Dijkstra 算法的局限性

Dijkstra 算法是一个贪心算法，通过优先队列实现时平均时间复杂度可以到达 O(m*log(n))。但是它不支持负权边。

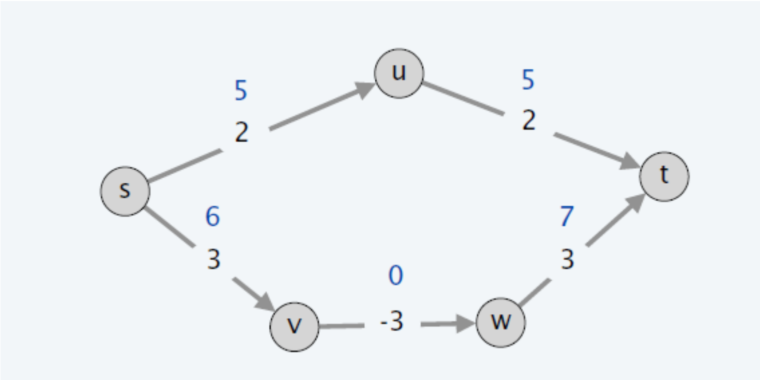
为了使 Dijkstra 算法支持负权边，有人提出了两种“启发式”的修正算法（然而实际上都是错误的）：

（1）负权边会影响当前“已求出到源点最短路的集合 S”中点的最短路，为此，只需要对 S 中的点也做松弛操作即可。

这种修正主要有两点错误。首先，Dijkstra 算法的优越性在于只对新加入的点有关联的其他点的距离做松弛，对集合 S 也做松弛显然增加了时间复杂度。另外关键的一点是，引入负权边的反复松弛会增加算法的运行时间：比如点 a，b 属于集合 S，c 是以负权边（c, a）关联到 S 的当前新加入的点，且图中存在有向边（a, b），(b, c），(即 a, b, c 构成一个环，有负权边，但是环的权值和大于 0）加入 c 后 a 到源点的距离可能比原来小，由于边（a, b）存在，b 的距离可能也变小，b 可能反过来再更新 c，导致进一步的迭代。

（2）所有边权都加上一个充分大的偏移量，使得所有边的权值都大于 0，在新图 G’上运用 Dijkstra 算法求解。

课件上给出了如下反例：



如图，原图的最短路是 s-v-w-t，然而加上偏移量 3 以后，最短路变成了 s-u-t。新图的最短路不一定是原图的最短路，问题的本质在于新旧图中最短路的边的条数不一定一样，同样的偏移量会在边多的那条最短路上产生较大影响。

小扩展：

对所有边权加一个相同的偏移量，如果某个子图原来是最小生成树，则它仍是新图中的最小生成树。这是因为生成树的边数固定是 n-1。另外，从 Kruskal 算法得到最小生成树的过程进行理解，对边做一个保序的函数操作，最小生成树不变。

2、Bellman-Ford 算法：单汇点求最短路径；负环检测

图中如果有负环，则最短路径不存在。图中如果没有负环，则存在一条不包含环的最短路径。Bellman-Ford 算法是一个动态规划算法，对于没有负环的图（允许存在负权边）能够求出所有点到单汇点的最短路。对于有负环存在的图，能够检测出负环。

定义状态：$\mathrm{OPT}(i, v)$表示顶点 v 通过小于等于 i 条边到达汇点 t 的最短距离。

初始条件：$\mathrm{OPT}(0, v) = +\infty$，一条边都不用，显然距离为无穷大。

转移方程：分两种情况更新$\mathrm{OPT}(i, v)$，（1）v 到汇点的实际最短路的边数小于等于 $i-1$，则直接$\mathrm{OPT}(i-1, v)$更新；（2）v 到汇点的实际最短路的边数可能等于 i，于是枚举 v 的所有出边连接的顶点 w,找到最小的 OPT(i −1, w)+C_{vw} 进行更新 由初始条件以及转移方程可得：

$$OPT(i,v) = \begin{cases} \infty & \text{if } i=0 \\ \min\left\{ OPT(i-1,\ v),\ \ \min\limits_{(v,w)\in E} \{ OPT(i-1,\ w)+c_{vw} \} \right\} & \text{otherwise} \end{cases}$$

课件中原文对于转移情况（2）的描述是恰好为 i，这其实不对，因为按照状态的定义，得到 OPT(i –1, w)的最短路不一定恰好包含 i – 1 条边；另外（1）和（2）并不是互斥的两种情况，但是（1）和（2）对应情况的“并集”包含了所有情况，这启发我们写转移方程的时候不一定要分互斥的情况进行转移，只需要“完备”即可。然而如果要求最短路的条数，我们必须写出由“互斥”的状态进行转移的方程。具体来说修改OPT(i, v)的定义为恰好由 i 条边构成 v 到汇点 t 的最短路的长度（如果到不了，那么距离就是无穷大）。

对于没有负环的图，所有顶点都必然存在一条由简单路径构成的最短路，这条路径的边数不超过 n – 1，所以最多更新 n – 1 轮，每轮最多扫描 m 条边（所有顶点的出边）就可以得到所有顶点到唯一汇点的最短路。所以循环 n – 1 轮后，**OPT（n – 1, v）**就是每个顶点的最短路

朴素的实现可以开一个二维数组存所有的OPT(i, v)，这时需要 n^2 的内存。然而事实上我们可以只开一个一维数组 OPT（v）表示当前循环找到的 v 到汇点 t 的最短路。同样还是循环 n – 1 轮，每轮遍历每个顶点的所有出边，更新 OPT（v）数组。

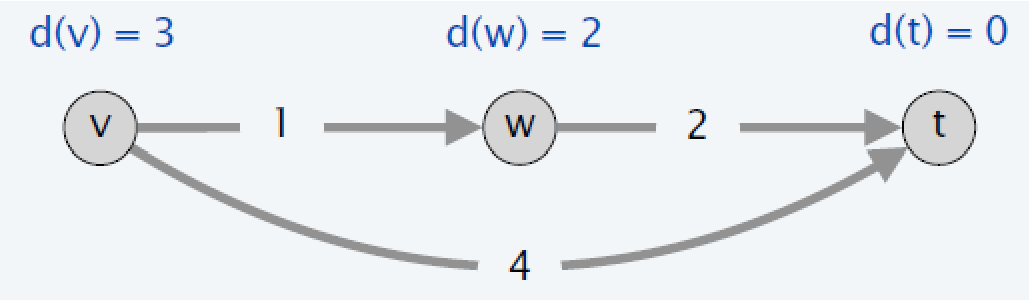
这时课上老师提了一个问题：

在做了内存优化的情况下，每轮遍历顶点的出边的顺序可否改变？

我当时联想以前背包问题做内存优化时不能改变遍历顺序的事实，猜想 Bellman-Ford 算法内存优化后，肯定跟固定的遍历顺序有关，于是不能每轮随意改变遍历顺序。

然而，事实并不是这样的。

做了内存优化后的递推式，并不严格按照之前的状态转移式的定义，然而它仍然在不断的松弛操作中得到了最短路。事实上，它收敛得更快了。课件上给出了如下例子：



假设更新顺序都是按照先 w 再 v。

(1)对于原始的递推式，初始化 d(0,t)=0, d(0,v)和 d(0, w)都是无穷大，第一轮循环 d(1, w)由 d(0,t)更新为 2，d(1,v)也由 d(0,t)更新为 4，第一轮循环得到的路径都是长度小于等于 1 的。第二轮循环 d(2,w)=d(1,w)不变，d(2,v)由 d(1,w)更新为 3。至此，算法结束。

(2)对于我们做了内存优化后的递推式。初始化 d(t)=0,d(v)和 d(w)均为无穷大。d(w)由 d(t)更新为 2，到目前为止和原始递推没有不同。然而在更新 d(v)的时候，由于 d(w)已经为 2（原始递推过程中 d(0,w)还是无穷大），d(v)直接由 d(w)更新为 3。此时实际上已经找到了 v 到 t 的最短路（虽然这条路径长度为 2）。

实际上，我们只关心最短路，而不关心最短路由几条边构成。内存优化后的每次松弛操作都让我们更新当前找到的最优路径，而最多 n-1 次循环必定能找到最短路。所以内存压缩后的递推式虽然不满足原始的状态转移定义，但仍然能帮助我们找到最短路。

从这个过程也可以看出，实际上每轮的顶点遍历顺序是可以随意选择的。

最后，关于负环检测。如果按照 Bellman-Ford 算法进行了 n-1 轮松弛操作后，第 n 轮松弛时仍然有顶点 v 到汇点 t 的距离缩小，则说明图中存在负环。

（以上是第一节课的所有内容。基于每轮是否有点被更新，可以提前结束 Bellman-Ford 算法，相关的进一步的优化内容是当天第二节算法课讲的，这里就交给后面的同学啦~）