研究生算法课课堂笔记

上课日期：2016 年 12 月 12 日　　　　　　　　第(2)节课

组员学号及姓名：申博 1601111290
郭怡欣 1601111266

内容概要

本节课主要讲解了算法模拟考试的两道题目（求逆序对数及 Heavy Transportation)，并在此基础上对所用到的算法（归并排序及 Dijkstra 算法）进行了分析与扩展。笔记内容安排如下：

1. 显著逆序对问题
2. Dijkstra 算法介绍
3. Dijkstra 算法实现及扩展
4. Heavy Transportation

详细内容

1. 逆序对问题变种：显著逆序对问题

问题描述

给定 n 个数的序列a_1，$a_2 \ldots \ldots a_n$,定义一个逆序对是一对 $i<j$,使得$a_i > a_j$。逆序对计算问题可以作为考察两个排序有多大差别的一个度量指标，但是这个指标可能太过敏感。我们定义 $i<j$ 且$a_i > 2a_j$这种为显著逆序对。设计一个 O(nlogn) 的算法计算在两个排序中的显著逆序对的个数。

问题分析

该问题为逆序对问题的变种，可以使用和逆序对相似的方法来完成。整体思路依旧是采用类似归并排序的方式来计算，差异在于在进行 MERGE-AND-COUNT 操作时，原始问题同时进行合并和计算逆序对个数的操作，而在计算重要逆序对的问题中，2 倍差异才算做逆序对，所以要将来两个操作分开进行。

算法伪代码

```
SORT-AND-COUNT (L)
IF 列表 L 只有一个元素
    RETURN (0, L).
把 list 分为两半 A 和 B
(rA , A) ← SORT-AND-COUNT(A).
(rB , B) ← SORT-AND-COUNT(B).
(rAB) ←COUNT(A, B).
(L’) ← MERGE (A, B).
RETURN (rA + rB + rAB , L’).
```

其中 L 为待处理的序列，L’为排序完毕的序列，rA，rB，rAB 分别为子段 A、B 的重要逆序对个数和两个子段之间的重要逆序对个数。

```
COUNT(A, B)
维护一个 Current 指针指向每个表，初始化指向首元素
维护一个变量 Count 用于逆序的个数，初始为 0
While 两个表都不空
    令a_i， b_j是由 Current 指针指向的元素
    IF a_i > 2b_j
        把 Count 加上 A 中剩余的元素
        把 B 中的 Current 指针前移
    ELSE
        将 A 中的 Current 指针前移
RETURN Count

MERGE(A,B)
维护一个 Current 指针指向每个表，初始化指向首元素
While 两个表都不空
    令a_i， b_j是由 Current 指针指向的元素
    IF a_i > b_j
        把 B 中 Current 指向的元素放入 L’
        把 B 中的 Current 指针前移
    ELSE
        把 A 中 Current 指向的元素放入 L’
        将 A 中的 Current 指针前移
若还有一个表不为空，将表中剩下所有元素放进 L’
RETURN L‘
```

算法时间复杂度

原问题中 MERGE-AND-CONUT 时间复杂度为 O(n)，变种问题中将 MERGE 和 COUNT 拆分为两步进行，遍历两遍，但时间复杂度同为 O(n)，所以算法总的时间复杂度为 O(nlogn)。

2. Dijkstra 算法介绍

Dijkstra 算法是 Edsger Dijkstra 在 1959 年提出的一个解决**边权非负图中单源最短路径问题**的**贪心**算法，用于计算一个节点到其他所有节点的最短路径，以起始点为中心向外层层扩展，直到扩展完所有点为止。

2.1 算法思想

设 G=(V,E)是一个带权有向图，把图中顶点集合 V 分成两组，维护两个集合 S 与 Q，分别记录当前已求得最短路径长度的点与未确定的点。维护一个数组 d[v]，表示对某一节点 v，s 到 v 的最短路径估计（上界）值为 d[v]。

初始时 S 中只有一个源点 s，以后每求得一条最短路径，就将路径终点加入

到集合 S 中，并更新数组所记录的最短路径长度估计值。循环直到全部顶点都加入，算法结束。

2.2 算法步骤

（1）初始时，S 只包括源点，即 S={s}，d[s]=0；Q 包括除 s 外其他顶点，对于某一顶点 v，若与 s 有边直接相连则 d[v]设为此边权值；若无则设为+∞;
（2）从 Q 中选取一个与 s 距离最小的顶点 u，将 u 加入 P 中；
（3）以 u 为中间点，更新 Q 中各点的最短路径长度估计值：若 s 经过 u 到某一顶点 v 的路径长度比原来 s 到 v 的距离要短（即 $d[u]+l_{uv}<d[v]$），则更新 d[v]为此时更短的长度；
（4）重复步骤（2）和（3），直到集合 Q 为空。

2.3 算法正确性证明

以下通过归纳法证明 Dijkstra 算法的正确性。

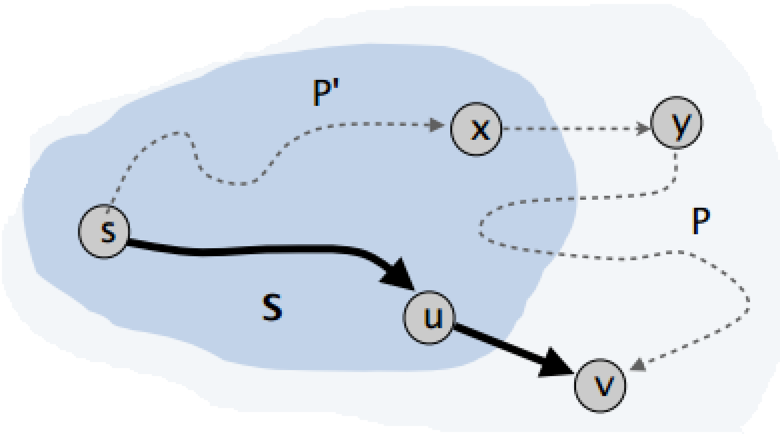
欲证明的假设：算法每次选择一条 s 到顶点 u 的路径时，这条路径比其他 s 到 u 的路径都要短，即算法选择的一定是最短路径。

归纳变元：S 中顶点的个数

（1） 当|S|=1 时，S={s}，d[s]=0，显然假设成立；

（2） 假设当|S|=k(k≥1)时假设成立，下一个加入到 S 中的点为 v，使得|S|=k+1，设(u,v)是 s-v 路径上最后一条边，则此时 s-u 的最短路径加上(u,v)边后是 s-v 的一条路径，设其长度为π(v)；

（3） 考虑任意 s-v 的其他路径 P，设(x,y)为第一条离开 S 的边（即 x∈S 且 y∈Q），设 P′为 s-x 的一条路径；



（4） 由于边权值非负，故 $L(P)\geq L(P')+l_{xy}$；因 s-x 的最短路径 d[x]已求得，故 L(P′)≥d[x]；假设 s-y 的最短路径长度为π(y)，可得出以下公式：

$$\ell(P) \geq \ell(P') + \ell(x,y) \geq d(x) + \ell(x,y) \geq \pi(y) \geq \pi(v)$$

由此可见，L(P)不可能比π(v)小，因此π(v)已是 s-v 最短路径长度。

（5） 至此已证明对于|S|=k+1 假设仍然成立。

3. Dijkstra 算法实现及扩展

对于有 n 个顶点，m 条边的图，初始化操作时间复杂度为 O(n)。通过 n-1 次

while 循环实现 Dijkstra 算法，每次循环中选择要加入的顶点 v 并加入 S。但若每次都考虑所有 $v \in Q$ 检查所有 S 与 v 间的边以确定最短路将耗费 O(m)，整体复杂度将为 O（mn），显然图越稠密，效率越低，最坏情况下为 $O(n^3)$。

因此，考虑使用恰当的数据结构进行优化，以下是使用优先级队列实现的算法伪代码：

DIJKSTRA (V, E, s)

Create an empty priority queue.

FOR EACH $v \neq s$: $d(v) \leftarrow \infty$; $d(s) \leftarrow 0$.

FOR EACH $v \in V$: *insert* v with key $d(v)$ into priority queue.

WHILE (the priority queue *is not empty*)

  $u \leftarrow$ *delete-min* from priority queue.

  FOR EACH edge $(u, v) \in E$ leaving u:

    IF $d(v) > d(u) + \ell(u, v)$

      decrease-key of v to $d(u) + \ell(u, v)$ in priority queue.

      $d(v) \leftarrow d(u) + \ell(u, v)$.

实现中需要注意三个关键操作：Insert，Extract_min，Decrease_key。

（1）Insert（n 次）：初始化时，以 d[v]为 key，将所有非源顶点 v 插入优先级队列中；

（2）Extract_min（n 次）：当优先级队列不为空时，从其中取出 key 最小的顶点 u；

（3）Decrease_key（m 次）：即常说的松弛操作。若 s 经过 u 到达 v 的路径长度（即 $d[u]+l_{uv}$）小于当前 s 到 v 的最短路径长度估计值（即 d[v]），则将 v 的 key 减小为 $d[u]+l_{uv}$。

这样实现的算法时间复杂度取决于优先队列的具体实现方式。采用哪种实现方式优化效果最好，要根据图的特征来确定。在此主要说明以下三种实现方式：

3.1 数组模拟优先级队列

直接用一维数组存储最短路径长度，每次选取下一个加入 S 的点时，需要遍历数组以找到距离最小的点，时间复杂度为 O(n)；而插入与修改操作都根据下标操作，时间复杂度都为 O(1)。算法整体复杂度为 $O(n^2)$。

3.2 使用 STL 中的 priority_queue

C++ STL<queue>中 priority_queue 的底层实现为 vector 容器，支持的取操作只有 top()，因此实现松弛操作后更新 key 时，可以另外维护一个标记数组，记录下一顶点是否以前出过队列，若是则不再处理（最短路径长度已确定），若

非则更新其 key。

使用标记数组对松弛操作时间复杂度的影响：由 lgn 变为 lgm。

（1） 当图为稀疏图时（即 m≈n），对复杂度无影响；

（2） 当图为稠密图时（即 m>>n），由于 m 最大为 n^2，lgm 最大为 2lgn，故从大 O 意义上来说，对复杂度也无影响。

3.3 基于堆实现优先队列

基于堆实现优先队列也有不同的方式，如二叉堆，d 路堆，斐波那契堆，布罗达尔队列等等，在此不一一展开，其三种关键操作及所实现的算法整体时间复杂度对比如下：

PQ implementation	insert	delete-min	decrease-key	total
unordered array	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log n)$
d-way heap (Johnson 1975)	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(m \log_{m/n} n)$
Fibonacci heap (Fredman-Tarjan 1984)	$O(1)$	$O(\log n)^{\dagger}$	$O(1)^{\dagger}$	$O(m + n \log n)$
Brodal queue (Brodal 1996)	$O(1)$	$O(\log n)$	$O(1)$	$O(m + n \log n)$

由此可见，对于稠密图，使用数组较优；对于稀疏图，二叉堆实现优先级队列较优；对性能要求较严格的情况下，可以选择 4 路堆实现优先级队列；斐波那契堆和布罗达尔队列理论上是最优的，但实现起来较为麻烦。

3.4 Dijkstra 算法扩展：

除了能求解单源最短路径问题，Dijkstra 算法还可以扩展来解决其他图搜索问题，如：

（1） 最大容量路径：边的权值表示该边能承受的容量/重量，需要求一条路径，使得该路径上的所有边的容量下界最大。

POJ1797:HeavyTransportation（http://poj.org/problem?id=1797）
POJ2253：Frogger(http://poj.org/problem?id=2253)

（2） 最大可靠性路径：边的权值表示该边的可靠性/安全系数，需要求一条路径，使得通过该路径到达终点的可靠性最大。

HDOJ1596：Find the Safest Road
（http://acm.hdu.edu.cn/showproblem.php?pid=1596）

4. Heavy Transportation

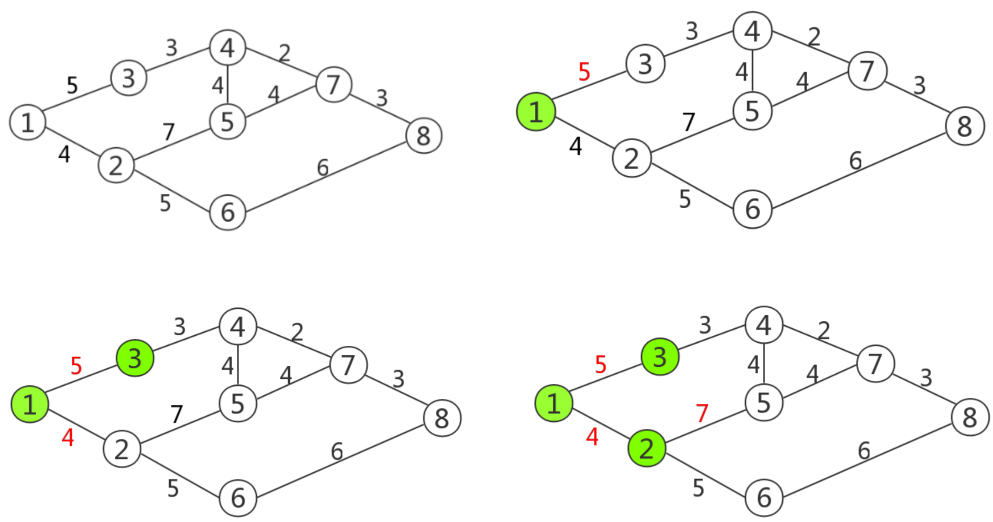
问题描述

有 1 到 n 的 n 个节点，每条边有能承受的最大重量，求可以从点 1 运送到点 n 的最大重量。

算法分析

该问题为 Dijkstra 最短路径问题的变种，由求最短路径改为求最大容量路径，算法整体思路与传统 Dijkstra 类似。从出发点开始，每次选择已知最大容量的点（集合 S 中的点）相连的点中，容量最大的边。

例：



第一步 起点 1 进入 S，d(2) d(3)由 0 变为 4,5

第二步 点 3 进入 S，d(4)由 0 变为 3

第三步 点 2 进入 S，d(5) d(6)由 0 变为 7,5

第四步 点 5 进入 S，d(7)由 0 变为 4；d(4)由 3 变为 4

由此类推

S 集合表示已经求出最大容量路径的点，c(u,v)表示从点 u 到点 v 的边的容量，d(u)表示从起点 1 到达该点的最大容量限制。

初始化：对所有 $u \neq s$，$d(u) = 0$；$d(s)=\infty$；

每次查找 d 最大的点 u，将其放入集合 S，对于所有和 u 相连的点，若 $d(v)<\min\{d(u),c(u,v)\}$，则将 d(v)改为 $\min\{d(u),c(u,v)\}$。

算法一：使用优先队列

```
DIJKSTRA (V, E, s)
产生一个空的优先队列
FOR EACH v ≠ s : d(v) ← ∞; d(s) ← 0.
FOR EACH v ∈ V : 将 v 以 d(v)为 key 插入优先队列
WHILE (优先队列不为空)
    删除优先队列中最大值 d(v)
```

```
FOR EACH edge (u, v) ∈ E leaving u:
    IF d(v)<min{d(u), c(u, v)}
        将优先队列中点 v 的值改为 min{d(u), c(u, v)}
        d(v) ← min{d(u), c(u, v)}
```

算法二：直接遍历查找最大值

```
对所有 u>1，d(u) ← 0
d(1)←∞
对所有 u，inS(u) ←false
WHILE S 集合元素少于 n 个
    遍历 d，找到最大值 d(u)
    inS(u) ←true
    S 集合元素数+1
    对所有和 u 相连且 inS(u)为 false 的点 v
        IF d(v)<min{d(u),c(u,v)}
            d(v)=←min{d(u),c(u,v)}
```