研究生算法课课堂笔记

上课日期：**20151221**　　第(2)节课

组长：陈怡然　　　　　　　　　　组员：宋文凯

课程内容：

1.解决上节课 Bellman-Ford 算法的相关问题

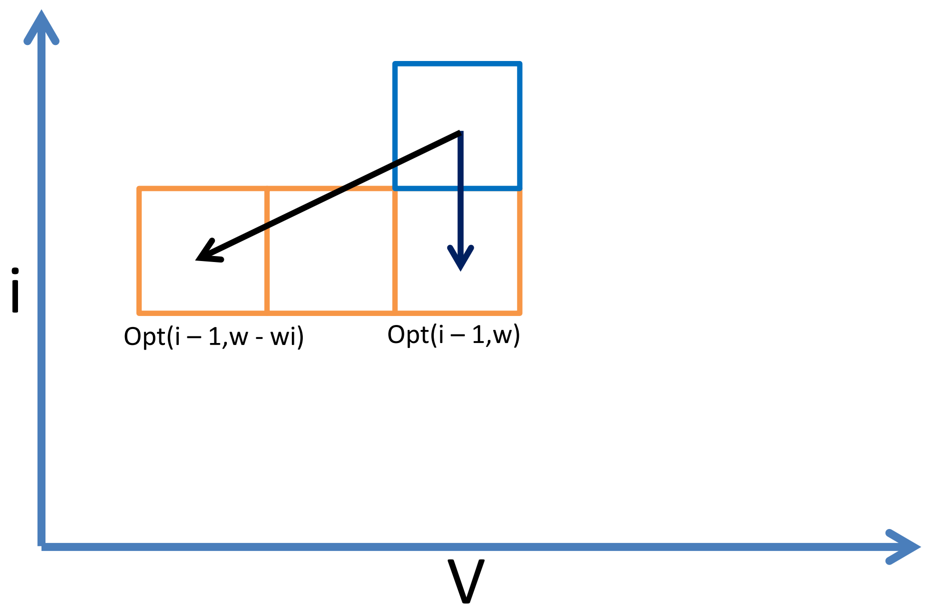
2.SPFA 算法及相关分析

1.Bellman-Ford 算法相关

1.1 路径压缩背景知识

前几节课说到的背包问题：

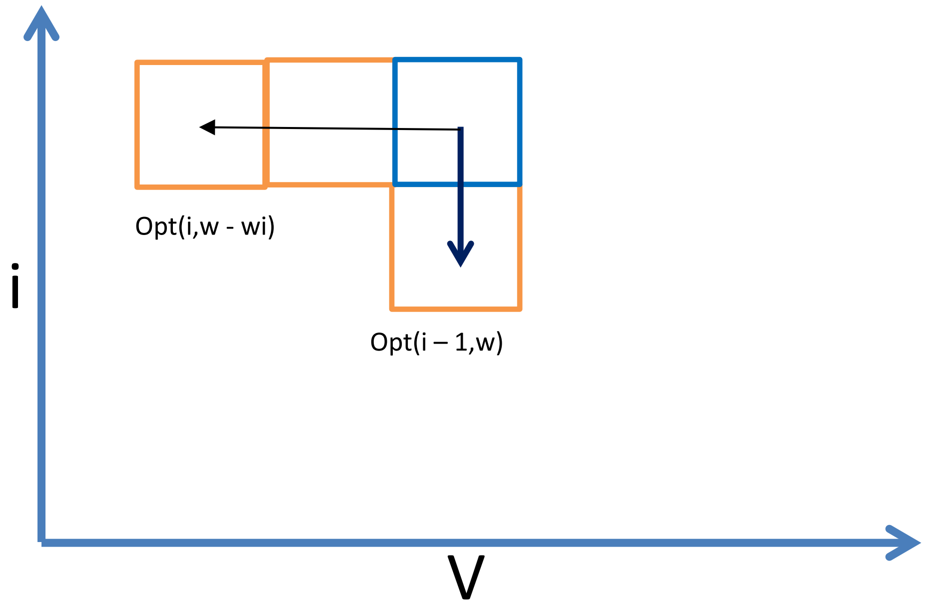
0-1 背包：根据状态转移公式，如果第 i 个物体没有装进来，则依赖方向向下，如果物体装进来，则依赖方向沿着 i 和 w 减小的方向。所有情况下 opt（i，w）都仅依赖于前一行 i-1。



所以压缩到一维时，w 需要从大往小循环，这样的话，此时使用的 w 较小的 opt 值还没有进行相关更新，自动使用了 opt（i-1,w）或者 opt（i-1，w-wi）。

完全背包：往下的依赖依旧存在，左下方向的依赖则改为向左的依赖，即第 i 个

物体装入时，依赖方向向左，依赖于 opt（i，w-wi）；第 i 个物体未装入时，依赖方向依旧向下



所以压缩到一维时，w 需要从小往大循环，这样就可以使 w 较小的 opt 值得到更新，如此可以得到 opt(i，w-wi)。否则会出现问题

1.2 Bellman-Ford 路径压缩

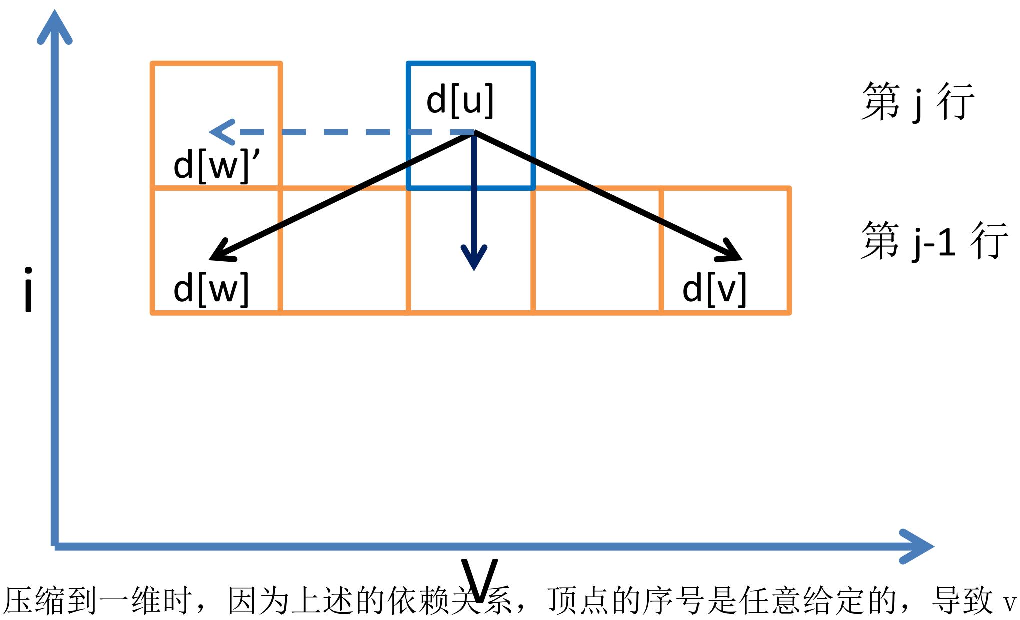
问题 1-1 Bellman-Ford 算法空间压缩的正确性

（对应上节课问题 2 和问题 3）

Bellman-Ford 的状态转移方程:

$$OPT(i,v) = \min\{OPT(i-1,v), \min_{(v,w)\in E}\{OPT(i-1,w) + c_{vw})\}\}$$

其中 Opt（i，v）：v 是顶点序号。依赖关系有两种，向下依赖：opt（i-1,v），没有使用 w 这个点；两边向下的依赖：opt（i-1,w），使用了 w 这个点，所以可以向左下也可以是右下



压缩到一维时，因为上述的依赖关系，顶点的序号是任意给定的，导致 v 无论从大到小还是从小到大循环，都会出现问题。

但是空间压缩算法依旧正确，为什么呢？

如图假设在第 j 轮循环中要更新 d[u]，则有如下三种可能：

(1) *OPT(j,u)*由 *OPT(j-1,u)*转移得到，则 d[u]保持不变，结果正确。

(2) *OPT(j,u)*由 *OPT(j-1,v)*转移得到，由于 d[v]在第 j 轮中尚未更新，所以其中仍保存着第 j-1 轮迭代后 v 到 t 的最短距离，所以更新 d[u]，结果正确。

(3) *OPT(j,u)*由 *OPT(j-1,w)*转移得到。即 *OPT(j,u)= OPT(j-1,w)+cost(u,w)*。由于在第 j 轮迭代过程中，w 比 u 先更新变为 *OPT(j,w)*（记为 d[w]’），且显然有：

OPT(j-1,w) = d[w] >= d[w]’ = *OPT(j,w)*。

然而按代码运行第 j 轮迭代后得到：

d[u] = *OPT(j,u)’* = d[w]’ + cost(u,w) <= *OPT(j,u)* = d[w] + cost(u,w)。

是一条花费更少的路径。由于本题目是求最短路径，所以从结果上看这样做是正确的。但是这样有一个问题：由于 d[w]’表示 w 可能经过 j 条边到达 t 的路径长度，则 d[u]实际表示 u 经过大于 j 条边到达 t，与状态转移方程中的 *OPT(j,u)*相悖。

但是每个点的更新次数一定小于 N，而且每次迭代至少有一个点到 t 的最短路径会永久确定下来，所以算法的正确性以及最短路径的输出不受影响。

问题 1-2 路径重复（对应上节课问题 1）

之前上节课还问了一个问题：$\mathrm{OPT}(\mathrm{i},\mathrm{v}) = \min_{(v,w)\in E}\{OPT(i-1,v), OPT(i-1,w)+c_{vw})\}$该式中的两个解会不会存在重复情况？

有可能，$OPT(i-1,v)$ 和 $\min_{(v,w)\in E}\{OPT(i-1,w)+c_{vw})\}$ 可能会有重复。例如 *OPT(i-1,w)*对应一条跳数小于等于 i-2 的路径，则 *OPT(i-1,w)*+c_{vw} 对应一条跳数小于等于 i-1 的路径，这可能与 *OPT(i-1,v)*重复。但是这并不影响算法正确性。

问题 1-3 最短路径数目求法

书上有道习题，求最短路径的条数，计算过程中，需要把最短路径的值计算出来，而且不能用空间压缩的方法来计算。

这种情况下，定义 opt（i，v）为顶点 v 到汇点 t 恰好用 i 条边的最短距离，这样它的状态转移方程就改为：

$$OPT(i,v)=\min_{(v,w)\in E}\{OPT(i-1,w)+c_{vw})\}$$

这时不能进行空间压缩，因为空间压缩之后就不能区分是 i 条边还是 i-1 条边。引入 cnt（i，v）表示顶点 v 到汇点 t 恰好用 i 条边的最短路径的条数，表达式如下：

$$cnt(i,v)=\sum_{w} cnt(i-1,w) \qquad \text{(w 为 v 所有最短路上的后继节点)}$$

表示对于所有满足最优解的 w,将它们的 cnt 值加在一起就得到了 v 的 cnt，注意这里只计算那些连到最短路径上的顶点的 cnt 值。

2．SPFA（shortest path fast algorithm）队列优化算法

2.1 算法介绍

功能：求单汇点最短路径+判断负环

方法：用 d[u]来记录当前情况下 u 到 t 的最短路径长度。设立一个队列每次取出队首结点 v，对所有 v 的入边 u->v，对 d[u]进行更新。若 d[u]发生改变且 u 点不在当前的队列中，就将 u 点放入队尾。这样不断从队列中取出结点来进行松弛操作，直至队列空为止。

2.2 代码实现

数据结构：

```
struct edge{
 int u, v; //u到v的边
 double c; //u到v的距离
};
double d[N]; //每个顶点当前求出的最短距离
bool valid[N]; //表示数组d里的值是否合法，到汇点是否可达，用来处理无穷
edge e[M]; //所有的边的大数组,下标从0开始
int head[N];//初始化为-1，链式前向星
```

队列初始化：

```
queue<int> q
q.push(t)
```

问题代码一：

```
while (!q.empty()){
 int u = q.front();//此处注意，q取队首元素的方法是用front而不是top
 q.pop();//取出元素后，将其pop出来
 for (int i = head[u]; i >= 0; i = e[i].next){
     int v = e[i].v;//e[i]表示u连到v的边
     double c = e[i].c;//c表示e[i]的cost
     double dist = d[v] + c;//此处不需要判断v点是否valid（valid[v]）,因为从队列中出来的点都是
valid的，某点只有存在某一条路径可达汇点时，才会有机会进队列。
     if (!valid[u] || d[u] > dist){//但是u尚未进入队列，所以要进行valid判断，
         d[u] = dist;
         s[u] = v;
         valid[u] = true;
         q.push(u);
     }
```

```
    }
  }
```

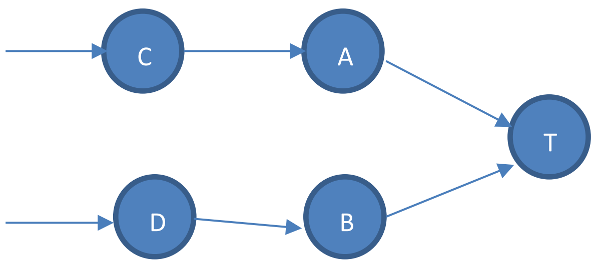
此时代码是按照单源点的方式来写的，需要将其改变成单汇点模式。

问题代码二：

```
  while (!q.empty()){
   int v = q.front();//这样改是因为我们需要把出边表模式，改成入边表。
   q.pop();//取出元素后，将其pop出来
   for (int i = head[v]; i >= 0; i = e[i].next){
       int u = e[i].u;//e[i]表示u连到v的边
       double c = e[i].c;//c表示e[i]的cost
       double dist = d[v] + c;//此处不需要判断v点是否valid ( valid[v] ),因为从队列中出来的点都是valid的，某点只有存在某一条路径可达汇点时，才会有机会进队列。
       if (!valid[u] || d[u] > dist){//但是u尚未进入队列，所以要进行valid判断，
           d[u] = dist;
           s[u] = v;
           valid[u] = true;
           q.push(u);
       }
    }
  }
```

通过上述代码我们实现了这样一个过程，

若连接图如下：



则入队列过程如下

T	A	B	C	D		

问题：我们如何利用这样的代码去找负环？

代码修改：

数据结构：

加入新的 bool 数组 cnt

```
bool inq[N]; //表示下标元素是否在队列 q 中
int cnt[N];//记录所有顶点入队列次数，初始化为 0
```

初始化：

```
inq[t] = true;//表示汇点 t 入队列
cnt[t]++;
```

更改后代码：（粗体表示更改项）

```
while (!q.empty()){
 int v = q.front();//此处注意，q取队首元素的方法是用front而不是top
 q.pop();//取出元素后，将其pop出来
 inq[v] = false;//v节点出队列，不在队列中
 for (int i = head[v]; i >= 0; i = e[i].next){
     int u = e[i].u;//e[i]表示u连到v的边
     double c = e[i].c;//c表示e[i]的cost
     double dist = d[v] + c;//此处不需要判断v点是否valid（valid[v]）,因为从队列中出来的点都是valid的，某点只有存在某一条路径可达汇点时，才会有机会进队列。
     if (!valid[u] || d[u] > dist){//但是u尚未进入队列，所以要进行valid判断，
         d[u] = dist;
         s[u] = v;
         valid[u] = true;
         if (!inq[u]){//如果u不在队列中，才让其进入队列
             q.push(u);
             inq[u] = true;//u节点进入队列
             if (++cnt[u] >= N) { return false; } //若入队次数大于等于N，则说明d[u]被更新至少N次，说明有负环
         }
```

```
        }
    }
  }
  return true;
```

2.3 相关分析

性能优化：

SPFA 算法中 while 的每次循环对应着 Bellman-Ford 算法中每次内循环。在 Bellman-Ford 算法中，每次内循环遍历所有边(u,v)。但是当 v 到 t 尚且不存在路径时，考虑(u,v)就没有意义了。而 SPFA 算法则规避了这一点，它利用近似 BFS 的思想，从汇点开始考虑入边，这样使得每次松弛的边(u,v)都满足 v 到 t 是可达的，而不是对所有边进行遍历，从而实现了性能上的优化。

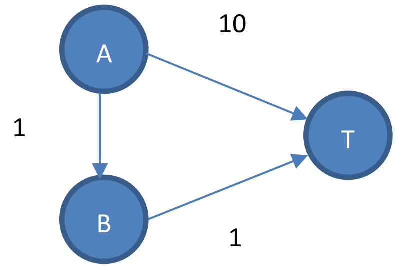
问题 2-1：队列的长度最多是多少呢？

最多为 N。因为队列当中每个顶点最多出现一次。

问题 2-2：每个顶点入队列的次数是多少呢？

该队列是一个普通队列。一般使用贪心法时才会使用优先队列，而且使用优先队列时，出队列后一般不再入队列。Bfs 搜索使用普通队列时，不会出现一个顶点入队列多次，只会出现某顶点不入队列。如果这个算法中所有顶点最多入队列一次，那根据算法可知，与点相关的所有入边最多遍历一次，那么此时该算法的时间复杂度为 O（m+n）。明显这种情况是不可能的。在最短路径求取过程中，一个顶点的最短距离有可能是反复更新的，所以每个顶点可以入队列多次。

举个例子



这种情况下，顶点 A 就会由于顶点 B 对其距离的更新而多次入队列。入队顺序依次为 T;A,B;A，即节点 a 入队两次。

问题 2-3：为什么 cnt 值大于 N 时，表示图中存在负环？

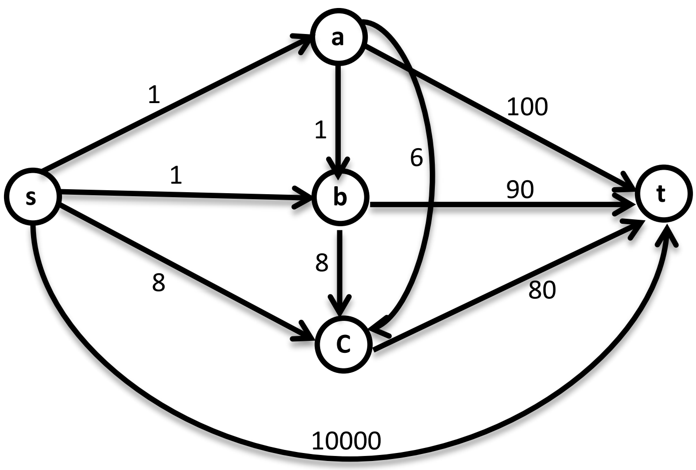
因为在 SPFA 算法中，除汇点外，队列中第一次出现某结点表明此时至少松弛了一遍了，此时 cnt 值为 1，以此类推，当某结点已经入队 n 次时，说明该算法应该运行到第 n 遍了，当进行到第 n 轮松弛结点距离依旧发生改变时，说明此处存在负环。

问题 2-4：可用栈或者其他结构来替换算法中使用的队列吗？

如果不使用先进先出的数据结构，就无法保证待处理结点的顺序性。只有保证了结点的顺序性，才能利用 cnt 值来判断负环。

问题 2-5：算法中的 bool 数组 inq 可否省略？

如果不考虑负环检测的话，用不用 inq 数组只是性能上的差别。如果考虑负环检测的话，若去除 inq 数组限制，会导致 cnt 条件判断出现问题。因为在一轮松弛过程中，一个顶点的最短距离有可能被松弛多次。举个例子：



入队顺序依次为 t;a,b,c,s;s,a,s,b,s,a;s。若没有 inq 数组，则 cnt[s]=5，返回 false 即有负环，然而实际构图中并不存在负环，甚至连负边都不存在。