北京大学暑期课《ACM/ICPC竞赛训练》

最短路算法

北京大学信息学院 郭炜
guo_wei@PKU.EDU.CN
http://weibo.com/guoweiofpku

本讲义参考一些网络资源改编而成，来源已不可考。仅用于内部授课

Dijkstra 算法

基本思想

- 解决无负权边的带权有向图或无向图的单源最短路问题
- 贪心思想，若离源点s前k-1近的点已经被确定，构成点集P，那么从s到离s第k近的点t的最短路径，$\{s,p_1,p_2\cdots p_i,t\}$ 满足 $s,p_1,p_2\cdots p_i\in P$。
- 否则假设pi∉P，则因为边权非负，pi到t的路径≥0，则d[pi]≤d[t],pi才是第k近。将pi看作t，重复上面过程，最终一定会有找不到pi的情况
- $d[i]=\min(d[p_i]+cost(p_i,i)), i\notin P, p_i\in P$

  $d[t]=\min(d[i]), i\notin P$

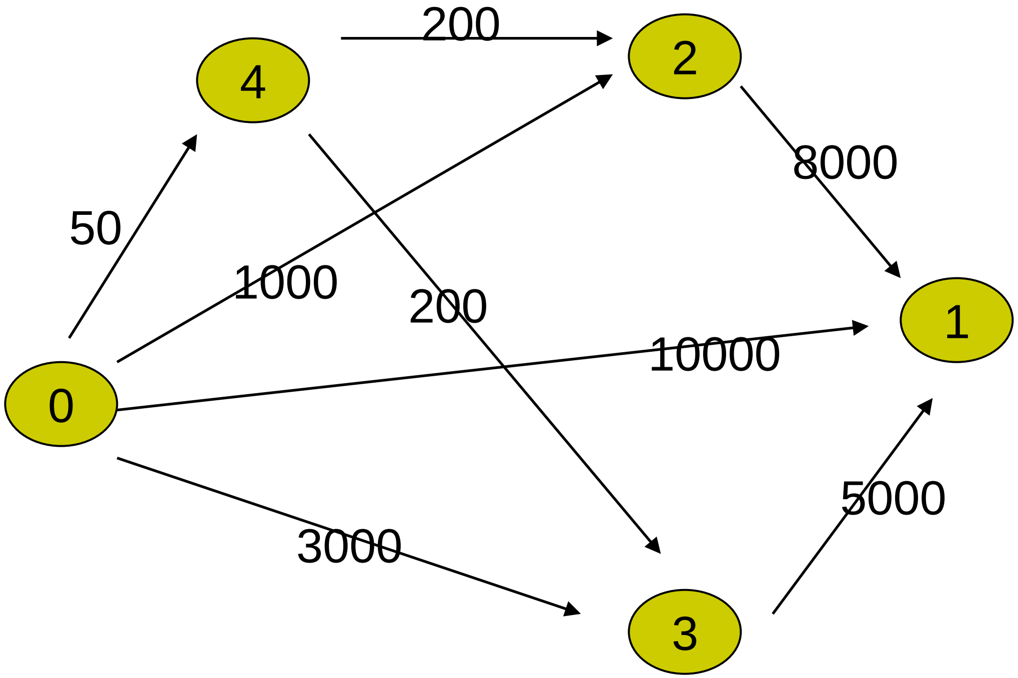
Dijkstra's Algorithm

- 初始令d[s]=0，d[i]=+∞，P=∅
- 找到点i∉P，且d[i]最小
- 把i添入P，对于任意j∉P，若d[i]+cost(i,j)<d[j]，则更新d[j]=d[i]+cost(i,j)。

Dijkstra's Algorithm

- 用邻接表，不优化，时间复杂度O(V^2+E)
- Dijkstra+堆的时间复杂度 o(ElgV)
- 用斐波那契堆可以做到O(VlogV+E)

- 若要输出路径，则设置prev数组记录每个节点的前趋点，在d[i]更新时更新prev[i]



源点0加入P后：

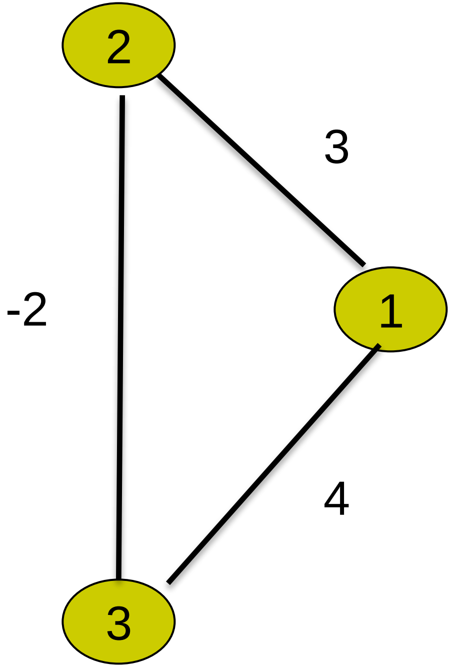
v	Dist[v]
0	0
1	[illegible]
2	[illegible]
3	[illegible]
4	50

Dijkstra's Algorithm

Dijkstra算法也适用于无向图。但不适用于有负权边的图。

d[1,2] = 2

但用Dijkstra算法求得 d[1,2] = 3



Dijkstra算法实现

- 已经求出到V0点的最短路的点的集合为T
- 维护Dist数组，Dist[i]表示目前Vi到V0的“距离”
- 开始Dist[0] = 0, 其他Dist[i] = 无穷大, T为空集

- 1) 若|T| = N，算法完成，Dist数组就是解。否则取Dist[i]最小的不在T中的点Vi, 将其加入T，Dist[i]就是Vi到V0的最短路长度。
- 2) 更新所有与Vi有边相连且不在T中的点Vj的Dist值：
- Dist[j] = min(Dist[j],Dist[i]+W(Vi,Vj))
- 3) 转到1）

POJ3159 Candies

有N个孩子（N<=3000)分糖果。
有M个关系(M<=150,000)。每个关系形如:

A B C

表示第B个学生比第A个学生多分到的糖果数目，不能超过C

求第N个学生最多比第1个学生能多分几个糖果

POJ3159 Candies

思路：30000点，150000边的稀疏图求单源最短路

读入　“A B C”，就添加A->B的有向边，权值为C

然后求1到N的最短路

用prioirty_queue实现 dijkstra + 堆的 POJ 3159 Candies

```
//by guo wei
#include <cstdio>
#include <iostream>
#include <vector>
#include <queue>
#include <cstring>
using namespace std;
struct CNode  {
        int k; //有向边的终点
        int w; //权值，或当前k到源点的距离
};
bool operator < ( const CNode & d1, const CNode & d2 )
{  return d1.w > d2.w; } //priority_queue总是将最大的元素出列
priority_queue<CNode> pq;
bool bUsed[30010]={0};

vector<vector<CNode> > v; //v是整个图的邻接表
const unsigned int INFINITE = 100000000;
```

```
int main()
{
        int N,M,a,b,c;
        int i,j,k;
        CNode p;
        scanf("%d%d", & N, & M );
        v.clear();
        v.resize(N+1);
        memset( bUsed,0,sizeof(bUsed));
        for( i = 1;i <= M; i ++ ) {
                scanf("%d%d%d", & a, & b, & c);
                p.k = b;
                p.w = c;
                v[a].push_back( p);
        }
        p.k = 1; //源点是1号点
        p.w = 0; //1号点到自己的距离是0
        pq.push (p);
```

```
    while( !pq.empty ()) {
            p = pq.top ();
            pq.pop();
            if( bUsed[p.k])   //已经求出了最短路
                    continue;
            bUsed[p.k] = true;
            if( p.k == N ) //因只要求求1-N的最短路，所以要break
                    break;
            for( i = 0, j = v[p.k].size(); i < j; i ++ ) {
                    CNode q; q.k = v[p.k][i].k;
                    if( bUsed[q.k] )  continue;
                    q.w = p.w + v[p.k][i].w ;
                    pq.push (q); //队列里面已经有q.k点也没关系
            }
    }
    printf("%d", p.w ) ;
    return 0;
}
```

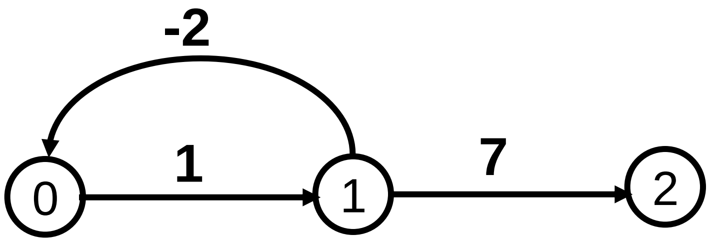
北京大學
PEKING UNIVERSITY

Bellman-Ford算法

Bellman-Ford算法

- 解决含负权边的带权有向图的单源最短路径问题
- 不能处理带负权边的无向图(因可以来回走一条负权边)
- 限制条件：

  要求图中不能包含权值总和为负值回路(负权值回路)，如下图所示。



(c)

Bellman-Ford算法思想

- 构造一个最短路径长度数组序列$dist^{1}$ [u]，$dist^{2}$ [u]，…，$dist^{n-1}$ [u]（u = 0,1…n-1,n为点数）
  - $dist^{1}$ [u]为从源点v到终点u的只经过一条边的最短路径长度，并有$dist^{1}$ [u] =Edge[v][u]；
  - $dist^{2}$ [u]为从源点v最多经过两条边到达终点u的最短路径长度；
  - $dist^{3}$ [u]为从源点v出发最多经过不构成负权值回路的三条边到达终点u的最短路径长度；
  - ……
  - $dist^{n-1}$ [u]为从源点v出发最多经过不构成负权值回路的n-1条边到达终点u的最短路径长度；
- 算法的最终目的是计算出$dist^{n-1}$ [u]，为源点v到顶点u的最短路径长度。

$dist^{k}[u]$的计算

●设已经求出 $dist^{k-1}[u]$，$u = 0, 1, \cdots, n-1$，即从源点v经过最多不构成负权值回路的$k-1$条边到达终点u的最短路径的长度

递推公式(求顶点u到源点v的最短路径)：

$dist^{1}[u] = Edge[v][u]$

$dist^{k}[u] = \min\{ dist^{k-1}[u], \min\{ dist^{k-1}[j] + Edge[j][u] \} \}$,

$j=0, 1, \cdots, n-1, j \neq u$

Dijkstra算法与Bellman-Ford算法的区别

- Dijkstra算法和Bellman算法思想有很大的区别：
  - Dijkstra算法在求解过程中，源点到集合S内各顶点的最短路径一旦求出，则之后不变了，修改的仅仅是源点到S外各顶点的最短路径长度。
  - Bellman-Ford算法在求解过程中，每次循环都要修改所有顶点的dist[ ]，也就是说源点到各顶点最短路径长度一直要到算法结束才确定下来。

负权回路的判断

如果存在从源点可达的负权值回路，则最短路径不存在，因为可以重复走这个回路，使得路径长度无穷小。

思路：在求出$dist^{n-1}[\ ]$之后，再对每条边$<u,k>$判断一下：加入这条边是否会使得顶点k的最短路径值再缩短，即判断：

$$dist[u]+w(u,k)<dist[k]$$

是否成立，如果成立，则说明存在从源点可达的负权值回路。

存在负权回路就一定能导致该式成立的证明略

负权回路的判断

证明：

如果成立，则说明找到了一条经过了n条边的从 s 到k的路径，且其比任何少于n条边的从s到k的路径都短。

一共n个顶点，路径却经过了n条边，则必有一个顶点m经过了至少两次。则m是一个回路的起点和终点。走这个回路比不走这个回路路径更短，只能说明这个回路是负权回路。

POJ3259 Wormholes

要求判断任意两点都能仅通过正边就互相可达的有向图(图中有重边）中是否存在负权环

Sample Input
2
3 3 1
1 2 2
1 3 4
2 3 1
3 1 3

3 2 1
1 2 3
2 3 4
3 1 8

Sample Output
NO
YES

2个test case
每个test case 第一行:

N M W (N<=500,M<=2500,W<=200)

N个点
M条双向正权边
W条单向负权边
第一个test case 最后一行
3 1 3
是单向负权边，3->1的边权值是-3

```
//by guo wei
#include <iostream>
#include <vector>
using namespace std;
int F,N,M,W;
const int INF = 1 << 30;
struct Edge {
        int s,e,w;
        Edge(int ss,int ee,int ww):s(ss),e(ee),w(ww) { }
        Edge() { }
};
vector<Edge> edges; //所有的边
int dist[1000];
```

```
int Bellman_ford(int v)   {
        for( int i = 1; i <= N; ++i)
                dist[i] = INF;
        dist[v] = 0;
        for( int k = 1; k < N; ++k) { //经过不超过k条边
                for( int i = 0;i < edges.size(); ++i) {
                        int s = edges[i].s;
                        int e = edges[i].e;
                        if( dist[s] + edges[i].w < dist[e])
                                dist[e] = dist[s] + edges[i].w;
                }
        }
        for( int i = 0;i < edges.size(); ++ i) {
                        int s = edges[i].s;
                        int e = edges[i].e;
                        if( dist[s] + edges[i].w < dist[e])
                                return true;
        }
        return false;
}
```

```
int main()  {
        cin >> F;
        while( F--) {
                edges.clear();
                cin >> N >> M >> W;
                for( int i = 0; i < M; ++ i) {
                        int s,e,t;
                        cin >> s >> e >> t;
                        edges.push_back(Edge(s,e,t)); //双向边等于两条边
                        edges.push_back(Edge(e,s,t));
                }
                for( int i = 0;i < W; ++i) {
                        int s,e,t;
                        cin >> s >> e >> t;
                        edges.push_back(Edge(s,e,-t));
                }
                if( Bellman_ford(1))//从1可达所有点
                        cout << "YES" <<endl;
                else cout << "NO" <<endl;
        }
}
```

问题

```
for( int k = 1; k < N; ++k) { //经过不超过k条边
    for( int i = 0;i < edges.size(); ++i) {
        int s = edges[i].s;
        int e = edges[i].e;
        if( dist[s] + edges[i].w < dist[e])
            dist[e] = dist[s] + edges[i].w;
    }
}
```

会导致在一次内层循环中，更新了某个 dist[x]后，以后又用dist[x]去更新dist[y],这样dist[y]就是经过最多不超过k+1条边的情况了

出现这种情况没有关系，因为整个 for(int k = 1; k < N; ++k) 循环的目的是要确保,对任意点u,如果从源s到u的最短路是经过不超过n-1条边的,则这条最短路不会被忽略。至于计算过程中对某些点 v 计算出了从s->v的经过超过N-1条边的最短路的情况，也不影响结果正确性。若是从s->v的经过超过N-1条边的结果比经过最多N-1条边的结果更小，那一定就有负权回路。有负权回路的情况下，再多做任意多次循环，每次都会发现到有些点的最短路变得更短了。

算法复杂度分析

- 假设图的顶点个数为n，边的个数为e
  - 使用邻接表存储图，复杂度O(n*e)
  - 使用邻接矩阵存储图，复杂度为O(n^3)；

Bellman-Ford算法改进

Bellman-Ford算法不一定要循环n-1次，n为顶点个数

只要在某次循环过程中，考虑每条边后，源点到所有顶点的最短路径长度都没有变，那么Bellman-Ford算法就可以提前结束了

例题

- POJ 1860 3259 2240

SPFA算法

Shortest Path Faster Algorithm

SPFA算法

- 快速求解含负权边的带权有向图的单源最短路径问题
- 是Bellman-Ford算法的改进版，利用队列动态更新dist[]

SPFA算法

●维护一个队列，里面存放所有需要进行迭代的点。初始时队列中只有一个源点S。用一个布尔数组记录每个点是否处在队列中。

●每次迭代，取出队头的点v，依次枚举从v出发的边v->u，若Dist[v]+len(v->u)小于Dist[u]，则改进Dist[u]（可同时将u前驱记为v）。此时由于S到u的最短距离变小了，有可能u可以改进其它的点，所以若u不在队列中，就将它放入队尾。这样一直迭代下去直到队列变空，也就是S到所有节点的最短距离都确定下来，结束算法。**若一个点最短路被改进的次数达到n，则有负权环(原因同B-F算法）。**可以用spfa算法判断图有无负权环

●**在平均情况下，SPFA算法的期望时间复杂度为O(E)。**

POJ3259 Wormholes 判断有没有负权环spfa

```
//by guo wei
#include <iostream>
#include <vector>
#include <queue>
#include <cstring>
using namespace std;
int F,N,M,W;
const int INF = 1 << 30;
struct Edge {
        int e,w;
        Edge(int ee,int ww):e(ee),w(ww) { }
        Edge() { }
};
vector<Edge> G[1000]; //整个有向图

int updateTimes[1000]; //最短路的改进次数

int dist[1000]; //dist[i]是源到i的目前最短路长度
```

```
int Spfa(int v) {
	for( int i = 1; i <= N; ++i)
		dist[i] = INF;
	dist[v] = 0;
	queue<int> que;  que.push(v);
	memset(updateTimes ,0,sizeof(updateTimes));
	while( !que.empty()) {
		int s = que.front();
		que.pop();
		for( int i = 0;i < G[s].size(); ++i) {
			int e = G[s][i].e;
			if( dist[e] > dist[s] + G[s][i].w ) {
				dist[e] = dist[s] + G[s][i].w;
				que.push(e); //没判队列里是否已经有e,可能会慢一些
				++updateTimes[e];
				if( updateTimes[e] >= N) return true;
			}
		}
	}
	return false;
}
```

```
int main(){
        cin >> F;
        while( F--) {
                cin >> N >> M >> W;
                for( int i = 1; i <1000; ++i)
                        G[i].clear();
                int s,e,t;
                for( int i = 0; i < M; ++ i) {
                        cin >> s >> e >> t;
                        G[s].push_back(Edge(e,t));
                        G[e].push_back(Edge(s,t));
                }
                for( int i = 0;i < W; ++i) {
                        cin >> s >> e >> t;
                        G[s].push_back(Edge(e,-t));
                }
                if( Spfa(1))
                        cout << "YES" <<endl;
                else    cout << "NO" <<endl;
        }
}
```

例题

POJ 2387
POJ 3256

弗洛伊德算法

弗洛伊德算法

- 用于求每一对顶点之间的最短路径。有向图，无向图均可，也可以有负权边

弗洛伊德算法

- 用于求每一对顶点之间的最短路径。有向图，无向图均可，也可以有负权边
- 假设求从顶点v_i到v_j的最短路径。如果从v_i到v_j有边，则从v_i到v_j存在一条长度为cost[i, j]的路径，该路径不一定是最短路径，尚需进行n次试探。

弗洛伊德算法

- 用于求每一对顶点之间的最短路径。有向图，无向图均可，也可以有负权边
- 假设求从顶点v_i到v_j的最短路径。如果从v_i到v_j有边，则从v_i到v_j存在一条长度为cost[i, j]的路径，该路径不一定是最短路径，尚需进行n次试探。
- 考虑路径（v_i，v_1，v_j）是否存在（即判别弧（v_i，v_1）和（v_1，v_j）是否存在）。如果存在，则比较cost[i, j]和（v_i，v_1，v_j）的路径长度，取长度较短者为从v_i到v_j的中间顶点的序号不大于1的最短路径，记为新的cost[i, j]。

弗洛伊德算法

- 用于求每一对顶点之间的最短路径。有向图，无向图均可，也可以有负权边
- 假设求从顶点v_i到v_j的最短路径。如果从v_i到v_j有边，则从v_i到v_j存在一条长度为cost[i, j]的路径，该路径不一定是最短路径，尚需进行n次试探。
- 考虑路径（v_i，v_1，v_j）是否存在（即判别弧（v_i，v_1）和（v_1，v_j）是否存在）。如果存在，则比较cost[i, j]和（v_i，v_1，v_j）的路径长度，取长度较短者为从v_i到v_j的中间顶点的序号不大于1的最短路径，记为新的cost[i, j]。
- 假如在路径上再增加一个顶点v_2，如果（v_i，…，v_2）和（v_2，…，v_j）分别是当前找到的中间顶点的序号不大于2的最短路径，那么（v_i，…，v_2，…，v_j）就有可能是从v_i到v_j的中间顶点的序号不大于2的最短路径。将它和已经得到的从v_i到v_j的中间顶点的序号不大于1的最短路径相比较，从中选出中间顶点的序号不大于2的最短路径之后，再增加一个顶点v_3，继续进行试探。依次类推。

弗洛伊德算法

- 在一般情况下，若（v_i，…，v_k）和（v_k，…，v_j）分别是从v_i到v_k和从v_k到v_j的中间顶点的序号不大于k-1的最短路径，则将（v_i，…，v_k，…，v_j）和已经得到的从v_i到v_j且中间顶点的序号不大于k-1的最短路径相比较，其长度较短者便是从v_i到v_j的中间顶点的序号不大于k的最短路径。这样，在经过n次比较后，最后求得的必是从v_i到v_j的最短路径。按此方法，可以同时求得各对顶点间的最短路径。
- 复杂度O(n^3)

弗洛伊德算法伪代码

```
for( int i = 1 ;i <= vtxnum; ++i )
   for( int j = 1; j <= vtxnum; ++j)  {
        dist[i][j] = cost[i][j]; // cost是边权值, dist是两点间最短距离
        if( dist[i][j] < INFINITE) //i到j有边
                path[i,j] = [i]+[j]; //path是路径
   }
for( k = 1; k <= vtxnum; ++k) //每次求中间点标号不超过k的i到j最短路
   for( int i = 1; i <= vtxnum; ++i)
        for(int j = 1; j <= vtxnum ; ++j)
                if( dist[i][k] + dist[k][j] < dist[i][j]) {
                        dist[i][j] = dist[i][k]+dist[k][j];
                        path[i,j] = path[i,k]+path[k,j];
                }
```

例题： POJ3660 Cow Contest

N个选手，如果A比B强,B比C强，则A必比C强

告知若干个强弱关系，问有多少人的排名可以确定

- **Sample Input**

5 5

4 3

4 2

3 2

1 2

2 5

5个人，5个胜负关系

4 比3强
4 比2强
3 比2强

…..

- **Sample Output**

2

例题： POJ3660 Cow Contest

如果一个点u，有x个点能到达此点，从u点出发能到达y个点，若x+y=N-1，则u点的排名是确定的。用floyd算出每两个点之间的距离，最后统计，若dist[a][b] 无穷大且dist[b][a]无穷大，则a和b的排名都不能确定。最后用点个数减去不能确定点的个数即可。

模版例题

POJ1125