SPFA算法

(跳转自SPFA)

目录 [隐藏]

算法简介

SPFA(Shortest Path Faster Algorithm)是Bellman-Ford算法的一种队列实现，减少了不必要的冗余计算。

算法流程

算法大致流程是用一个队列来进行维护。 初始时将源加入队列。 每次从队列中取出一个元素，并对所有与他相邻的点进行松弛，若某个相邻的点松弛成功，则将其入队。 直到队列为空时算法结束。

这个算法，简单的说就是队列优化的bellman-ford,利用了每个点不会更新次数太多的特点发明的此算法

SPFA——Shortest Path Faster Algorithm，它可以在O(kE)的时间复杂度内求出源点到其他所有点的最短路径，可以处理负边。SPFA的实现甚至比Dijkstra或者Bellman_Ford还要简单:

设Dist代表S到I点的当前最短距离，Fa代表S到I的当前最短路径中I点之前的一个点的编号。开始时Dist全部为+∞，只有Dist[S]=0，Fa全部为0。

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，设边的长度为len，判断Dist[v]+len是否小于Dist[u]，若小于则改进Dist[u]，将Fa[u]记为v，并且由于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到所有的最短距离都确定下来，结束算法。若一个点入队次数超过n，则有负权环。

SPFA 在形式上和宽度优先搜索非常类似，不同的是宽度优先搜索中一个点出了队列就不可能重新进入队列，但是SPFA中一个点可能在出队列之后再次被放入队列，也就是一个点改进过其它的点之后，过了一段时间可能本身被改进，于是再次用来改进其它的点，这样反复迭代下去。设一个点用来作为迭代点对其它点进行改进的平均次数为k，有办法证明对于通常的情况，k在2左右。

SPFA算法（Shortest Path Faster Algorithm），也是求解单源最短路径问题的一种算法，用来解决：给定一个加权有向图G和源点s，对于图G中的任意一点v，求从s到v的最短路径。 SPFA算法是Bellman-Ford算法的一种队列实现，减少了不必要的冗余计算，他的基本算法和Bellman-Ford一样，并且用如下的方法改进： 1、第二步，不是枚举所有节点，而是通过队列来进行优化 设立一个先进先出的队列用来保存待优化的结点，优化时每次取出队首结点u，并且用u点当前的最短路径估计值对离开u点所指向的结点v进行松弛操作，如果v点的最短路径估计值有所调整，且v点不在当前的队列中，就将v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、同时除了通过判断队列是否为空来结束循环，还可以通过下面的方法： 判断有无负环：如果某个点进入队列的次数超过V次则存在负环（SPFA无法处理带负环的图）。

SPFA算法有两个优化算法 SLF 和 LLL： SLF：Small Label First 策略，设要加入的节点是j，队首元素为i，若dist(j)<dist(i)，则将j插入队首，否则插入队尾。 LLL：Large Label Last 策略，设队首元素为i，队

列中所有dist值的平均值为x，若dist(i)>x则将i插入到队尾，查找下一元素，直到找到某一i使得dist(i)<=x，则将i出对进行松弛操作。 SLF 可使速度提高 15 ~ 20%；SLF + LLL 可提高约 50%。 在实际的应用中SPFA的算法时间效率不是很稳定，为了避免最坏情况的出现，通常使用效率更加稳定的Dijkstra算法。

伪代码

```
Procedure SPFA;

Begin
  initialize-single-source(G,s);
  initialize-queue(Q);
  enqueue(Q,s);
  while not empty(Q) do
    begin
      u:=dequeue(Q);
      for each v∈adj[u] do
        begin
          tmp:=d[v];
          relax(u,v);
          if (tmp<>d[v]) and (not v in Q) then
            enqueue(Q,v);
        end;
    end;
End;
```

代码

最基本的SPFA

```
bool        Relax(long        &w,long        m){return        m<w?(w=m,1):0;}//"松弛"操作
const        long        maxV=1000,maxE=999000;//最大顶点数,最大边数
long        m,H[maxV],D[maxV];//m为边数,初始化为0;H为链表头,初始化为-1;D为距离
struct        Edge{long        z,y,w;}E[maxE];//静态邻接表,w为权,y为边终点,z为静态指针
void        addE(long        x,long        y,long
w){E[m].y=y,E[m].w=w,E[m].z=H[x],H[x]=m++;}//加一条从x指向y,权为w的边
#include<cstring>
#include<queue>
void        SPFA(long        x=0){//默认计算从0点出发到达其他点的最短路
        bool        F[maxV]={};//初始为0的bool数组表示在不在队内
        std::queue<long>Q;//初始空队列

for(memset(D,0x3f,sizeof(D)),D[x]=0,F[x]=1,Q.push(x);!Q.empty();F[x]=0,Q.pop())//迭代到队列再次变空
                for(long        i=H[x=Q.front()],y;~i;i=E[i].z)//对于所有与x相邻的边
                        if(Relax(D[y=E[i].y],E[i].w+D[x])&&!F[y])
F[y]=1,Q.push(y);//如果松弛成功,则要确保y已入队
}
```

SPFA(slf优化)

```
void Spfa()
{
    d[S]=0;
    v[S]=true;
    deque <int> q;
    for(q.push_back(S);!q.empty();)
    {
        int x=q.front();
        q.pop_front();
        for(int k=head[x];k!=-1;k=el[k].next)
        {
            int y=el[k].y;
            if(d[y]>d[x]+el[k].c)
            {
                d[y]=d[x]+el[k].c;
                if(!v[y])
                {
                    v[y]=true;
                    if(!q.empty())
                    {
                        if(d[y]>d[q.front()])
                            q.push_back(y);
                        else
                            q.push_front(y);
                    }
                    else
```

```
                        q.push_back(y);
                    }
                }
            }
            v[x]=false;
        }
        return ;
}
```

```
procedure spfa;
begin
  fillchar(q,sizeof(q),0); h:=0; t:=0;//队列
  fillchar(v,sizeof(v),false);//v[i]判断i是否在队列中
  for i:=1 to n do
    dist[i]:=maxint;//初始化最小值

  inc(t);
  q[t]:=1;
  v[1]:=true;
  dist[1]:=0;//这里把1作为源点

  while h<>t do
    begin
      h:=(h mod n)+1;
      x:=q[h];
      v[x]:=false;
      for i:=1 to n do
        if (cost[x,i]>0) and (dist[x]+cost[x,i]<dist[i]) then
          begin
            dist[i]:=dist[x]+cost[x,i];
            if not(v[i]) then
              begin
                t:=(t mod n)+1;
                q[t]:=i;
                v[i]:=true;
              end;
          end;
    end;
end;
```

```
void SPFA(void)
{
 int i;
 queue list;
 list.insert(s);
 for(i=1;i<=n;i++)
  {
   if(s==i)
    continue;
   dist[i]=map[s][i];
   way[i]=s;
   if(dist[i])
   list.insert(i);
  }
 int p;
 while(!list.empty())
 {
  p=list.fire();
  for(i=1;i<=n;i++)
   if(map[p][i]&&(dist[i]>dist[p]+map[p][i]||!dist[i])&&i!=s)
    {
     dist[i]=dist[p]+map[p][i];
     way[i]=p;
     if(!list.in(i))
      list.insert(i);
    }
 }
}
```

各种加上了注释

```
/*
 * 单源最短路算法SPFA,时间复杂度O(kE),k在一般情况下不大于2,对于每个顶点使用可以在O(VE)的时间内算出每对节点之间的最短路
 * 使用了队列,对于任意在队列中的点连着的点进行松弛,同时将不在队列中的连着的点入队,直到队空则算法结束,最短路求出
 * SPFA是Bellman-Ford的优化版,可以处理有负权边的情况
 * 对于负环,我们可以证明每个点入队次数不会超过V,所以我们可以记录每个点的入队次数,如果超过V则表示其出现负环,算法结束
 * 由于要对点的每一条边进行枚举,故采用邻接表时时间复杂度为O(kE),采用矩阵时时间复杂度为O(kV^2)
 */
```

```
#include<cstdio>
#include<vector>
#include<queue>
#define MAXV 10000
#define INF 1000000000 //此处建议不要过大或过小,过大易导致运算时溢出,过小可能会被判定为真正的距离

using std::vector;
using std::queue;

struct Edge{
	int v; //边权
	int to; //连接的点
};

vector<Edge> e[MAXV]; //由于一般情况下E<<V*V,故在此选用了vector动态数组存储,也可以使用链表存储
int dist[MAXV]; //存储到原点0的距离,可以开二维数组存储每对节点之间的距离
int cnt[MAXV]; //记录入队次数,超过V则退出
queue<int> buff; //队列,用于存储在SPFA算法中的需要松弛的节点
bool done[MAXV]; //用于判断该节点是否已经在队列中
int V; //节点数
int E; //边数

bool spfa(const int st){ //返回值:TRUE为找到最短路返回,FALSE表示出现负环退出
	for(int i=0;i<V;i++){ //初始化:将除了原点st的距离外的所有点到st的距离均赋上一个极大值
		if(i==st){
			dist[st]=0; //原点距离为0;
			continue;
		}
		dist[i]=INF; //非原点距离无穷大
	}
	buff.push(st); //原点入队
	done[st]=1; //标记原点已经入队
	cnt[st]=1; //修改入队次数为1
	while(!buff.empty()){ //队列非空,需要继续松弛
		int tmp=buff.front(); //取出队首元素
		for(int i=0;i<(int)e[tmp].size();i++){ //枚举该边连接的每一条边
			Edge *t=&e[tmp][i]; //由于vector的寻址速度较慢,故在此进行一次优化
			if(dist[tmp]+(*t).v<dist[(*t).to]){ //更改后距离更短,进行松弛操作
				dist[(*t).to]=dist[tmp]+(*t).v; //更改边权值
				if(!done[(*t).to]){ //没有入队,则将其入队
					buff.push((*t).to); //将节点压入队列
					done[(*t).to]=1; //标记节点已经入队
					cnt[(*t).to]=1; //节点入队次数自增
					if(cnt[(*t).to]>V){ //已经超过V次,出现负环
						while(!buff.empty())buff.pop(); //清空队列,释放内
存
						return false; //返回FALSE
					}
				}
			}
		}
		buff.pop();//弹出队首节点
		done[tmp]=0;//将队首节点标记为未入队
	}
	return true; //返回TRUE
} //算法结束

int main(){ //主函数
	scanf("%d%d",&V,&E); //读入点数和边数
	for(int i=0,x,y,l;i<E;i++){
		scanf("%d%d%d",&x,&y,&l); //读入x,y,l表示从x->y有一条有向边长度为l
		Edge tmp; //设置一个临时变量,以便存入vector
		tmp.v=l; //设置边权
		tmp.to=y; //设置连接节点
		e[x].push_back(tmp); //将这条边压入x的表中
	}
	if(!spfa(0)){ //出现负环
		printf("出现负环,最短路不存在\n");
	}else{ //存在最短路
		printf("节点0到节点%d的最短距离为%d",V-1,dist[V-1]);
	}
	return 0;
}
```

图论及图论算法 [编辑]

图 - 有向图 - 无向图 - 连通图 - 强连通图 - 完全图 - 稀疏图 - 零图 - 树 - 网络

基本遍历算法: 宽度优先搜索 - 深度优先搜索 - A* - 并查集求连通分支 - Flood Fill

最短路: Dijkstra - Bellman-Ford (SPFA) - Floyd-Warshall - Johnson算法

最小生成树: Prim - Kruskal

强连通分支：Kosaraju - Gabow - Tarjan

网络流：增广路法（Ford-Fulkerson，Edmonds-Karp，Dinic） - 预流推进 - Relabel-to-front

图匹配 - 二分图匹配：匈牙利算法 - Kuhn-Munkres - Edmonds' Blossom-Contraction

1个分类：图论