Prim算法

目录 [隐藏]

基本思想

[编辑]

1. 在图G=(V, E) （V表示顶点 ，E表示边）中，从集合V中任取一个顶点（例如取顶点v0）放入集合 U中，这时 U={v0}，集合T(E)为空。
2. 从v0出发寻找与U中顶点相邻（另一顶点在V中）权值最小的边的另一顶点v1，并使v1加入U。即U={v0,v1 }，同时将该边加入集合T(E)中。
3. 重复2，直到U=V为止。
这时T(E)中有n-1条边，T = (U, T(E))就是一棵最小生成树。

PASCAL代码

[编辑]

```
procedure prim(v0:integer);

var
   lowcost,closest:array[1..maxn] of integer;
   i,j,k,min,ans:integer;
begin
   for i:=1 to n do begin
      lowcost[i]:=cost[v0,i];
      closest[i]:=v0;
   end;
   for i:=1 to n-1 do begin
   {寻找离生成树最近的未加入顶点k}
      min:=maxint;
      for j:=1 to n do
         if (lowcost[j]<min) and (lowcost[j]<>0) then begin
            min:=lowcost[j];
            k:=j;
         end;
      inc(ans, lowcost[k]); {把这条边加入生成树边集合T中}
      lowcost[k]:=0; {lowcost[k] = 0表示把顶点k加入集合U中}
      {修正各点的lowcost和closest值}
      for j:=1 to n do
         if cost[k,j]<lowcost[j] then begin
            lowcost[j]:=cost[k,j];
            closest[j]:=k;
         end;
   end;
 writeln(ans);
end;{prim}
```

C语言代码

[编辑]

```
int lowcost[MAXN],closest[MAXN];
int prim(int v0)
{
    int i,j,mindis,minone;
    int ans = 0;/*用来记录最小生成树的总长度*/
    /*各点距离初始化*/
    for(i = 0;i < n;i++)
    {
        lowcost[i] = cost[v0][i];
```

```
            closest[i] = v0;
        }
        for(i = 0;i < n-1;i++)
        {
            mindis = UPPERDIS;
            for(j = 0;j < n;j++)
              if(lowcost[j] && mindis > lowcost[j])
              {
                  mindis = lowcost[j];
                  minone = j;
              }
            /*将找到的最近点加入最小生成树*/
            ans += lowcost[minone];
            lowcost[minone] = 0;
            /*修正其他点到最小生成树的距离*/
            for(j = 0;j < n;j++)
              if(cost[j][minone] < lowcost[j])
              {
                  lowcost[j] = cost[j][minone];
                  closest[j] = minone;
              }
        }
        return ans;
}
```

C++代码

[编辑]

```
const        int        N=500;
unsigned        long        D[N],Q[N][N];//邻接阵表示距离,无边时此值为4294967295
bool        cmp(int        x,int        y){return        D[x]<D[y];}
unsigned        long        Prim(void){
        int        i=N;
        list<int>L;
        for(memcpy(D,Q[0],sizeof(D));--i;L.push_back(i));//以0号点为基准,L为还未进入最小生成树的点之集合
        for(list<int>::iterator        p;!L.empty();)//找到能见到的最短边,将新点从L中除去,然后以新点为基准,更新L中剩余点,直到L为空

for(p=min_element(L.begin(),L.end(),cmp),i=*p,L.erase(p),p=L.end();L.end()!=++p;D[*p]<?=Q[i][*p])

        return        accumulate(1+D,N+D,0LU);//对D求和,即为最小生成树总长
}
```

Prim算法的堆优化

[编辑]

朴素的Prim算法如果使用邻接矩阵来保存图的话，时间复杂度是O(N^2)，观察代码很容易发现，时间主要浪费在每次都要遍历所有点找一个最小距离的顶点，对于这个操作，我们很容易想到用堆来优化，使得每次可以在log级别的时间找到距离最小的点。下面的代码是一个使用二叉堆实现的堆优化Prim算法，代码使用邻接表来保存图。另外，需要说明的是，为了松弛操作的方便， 堆里面保存的顶点的标号，而不是到顶点的距离，所以我们还需要维护一个映射pos[x]表示顶点x在堆里面的位置。
使用二叉堆优化Prim算法的时间复杂度为O((V + E) log(V)) = O(E log(V))，对于稀疏图相对于朴素算法的优化是巨大的，然而100行左右的二叉堆优化Prim相对于40行左右的并查集优化Kruskal，无论是在效率上，还是编程复杂度上并不具备多大的优势。另外，我们还可以用更高级的堆来进一步优化时间界，比如使用斐波那契堆优化后的时间界为O(E + V log(V))，但编程复杂度也会变得更高。 --YangZX 22:17 2011年9月11日 (CST)

```
/*
二叉堆优化Prim算法
Author:YangZX
Date: 9.11 2011
*/
#include <iostream>
using namespace std;
const int MAXV = 10001, MAXE = 100001, INF = (~0u)>>2;
struct edge{
        int t, w, next;
}es[MAXE * 2];
int h[MAXV], cnt, n, m, heap[MAXV], size, pos[MAXV], dist[MAXV];
void addedge(int x, int y, int z)
{
        es[++cnt].t = y;
        es[cnt].next = h[x];
        es[cnt].w = z;
        h[x] = cnt;
}

void heapup(int k)
{
        while(k > 1){
```

```
			if(dist[heap[k>>1]] > dist[heap[k]]){
				swap(pos[heap[k>>1]], pos[heap[k]]);
				swap(heap[k>>1], heap[k]);
				k>>=1;
			}else
				break;
		}
}
void heapdown(int k)
{
		while((k<<1) <= size){
			int j;
			if((k<<1) == size || dist[heap[(k<<1)]] < dist[heap[(k<<1)+1]])
				j = (k<<1);
			else
				j = (k<<1) + 1;
			if(dist[heap[k]] > dist[heap[j]]){
				swap(pos[heap[k]], pos[heap[j]]);
				swap(heap[k], heap[j]);
				k=j;
			}else
				break;
		}
}
void push(int v, int d)
{
		dist[v] = d;
		heap[++size] = v;
		pos[v] = size;
		heapup(size);
}
int pop()
{
		int ret = heap[1];
		swap(pos[heap[size]], pos[heap[1]]);
		swap(heap[size], heap[1]);
		size--;
		heapdown(1);
		return ret;
}

int prim()
{
		int mst = 0, i, p;
		push(1, 0);
		for(i=2; i<=n; i++)
			push(i, INF);
		for(i=1; i<=n; i++){
			int t = pop();
			mst += dist[t];
			pos[t] = -1;
			for(p = h[t]; p; p = es[p].next){
				int dst = es[p].t;
				if(pos[dst] != -1 && dist[dst] > es[p].w){
					dist[dst] = es[p].w;
					heapup(pos[dst]);
					heapdown(pos[dst]);
				}
			}
		}
		return mst;
}
int main()
{
		cin>>n>>m;
		for(int i=1; i<=m; i++){
			int x, y, z;
			cin>>x>>y>>z;
			addedge(x, y, z);
			addedge(y, x, z);
		}
		cout<<prim()<<endl;
		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