Dijkstra算法

Dijkstra算法是一种求单源最短路的算法，即从一个点开始到所有其他点的最短路。其基本原理是：每次新扩展一个距离最短的点，更新与其相邻的点的距离。当所有边权都为正时，由于不会存在一个距离更短的没扩展过的点，所以这个点的距离永远不会再被改变，因而保证了算法的正确性。不过根据这个原理，用Dijkstra求最短路的图不能有负权边，因为扩展到负权边的时候会产生更短的距离，有可能就破坏了已经更新的点距离不会改变的性质。

如果用本算法求一个图中全部的最短路，则要以每个点为源调用一次Dijkstra算法。

适用条件与限制 [编辑]

- 有向图和无向图都可以使用本算法，无向图中的每条边可以看成相反的两条边。
- 用来求最短路的图中不能存在负权边。(可以利用拓扑排序检测)

算法流程 [编辑]

在以下说明中，s为源，w[u,v]为点u和v之间的边的长度，结果保存在dist[]

1. 初始化：源的距离dist[s]设为0，其他的点距离设为无穷大，同时把所有的点的状态设为没有扩展过。
2. 循环n-1次：
   1. 在没有扩展过的点中取一距离最小的点u，并将其状态设为已扩展。
   2. 对于每个与u相邻的点v，执行Relax(u,v)，也就是说，如果dist[u]+w[u,v]<dist[v]，那么把dist[v]更新成更短的距离dist[u]+w[u,v]。此时到点v的最短路径上，前一个节点即为u。
3. 结束。此时对于任意的u，dist[u]就是s到u的距离。

算法实现 [编辑]

== [编辑]

最简单的实现方法就是，在每次循环中，再用一个循环找距离最短的点，然后用任意的方法更新与其相邻

的边，时间复杂度显然为$O\left(n^2\right)$

对于空间复杂度：如果只要求出距离，只要n的附加空间保存距离就可以了（距离小于当前距离的是已访问的节点，对于距离相等的情况可以比较编号或是特殊处理下）。如果要求出路径则需要另外V的空间保存前一个节点，总共需要2n的空间。

二叉堆实现 [编辑]

使用二叉堆(Binary Heap)来保存没有扩展过的点的距离并维护其最小值，并在访问每条边的时候更新，可以把时间复杂度变成$O\left(|E|+|V|\log|V|\right)$。

用邻接表保存边，使得扩展边的总复杂度为O(E)，否则复杂度不会减小。

空间复杂度：这种算法需要一个二叉堆，及其反向指针，另外还要保存距离，所以所用空间为3V。如果保存路径则为4V。

具体思路：先将所有的点插入堆，并将值赋为极大值(maxint/maxlongint)，将原点赋值为0，通过松弛技术（relax）进行更新以及设定为扩展。

ddd

程序 [编辑]

二叉堆实现_Pascal

二叉堆实现_C

二叉堆实现_C++

扩展 [编辑]

第k短路 [编辑]

当k比较小时，可以直接在每个点保存k条最短路。更新的时候对每条能更新的路都更新一遍。此时每次更新的代价相当于把两个长度为k的表合并在一起，所以复杂度为纯Dijkstra实现的复杂度×O(k)。

曹氏短边法：每次将任意一条边赋值为MAX，重复计算数次后得到k短路径。

练习 [编辑]

Sweet Butter(USACO 3.2.6)

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

2个分类：版权无争议的内容 | 图论

Dijkstra 二叉堆实现 C

```
int  GraphDijk(struct Graph *g, int root, int *parent, int *distance)
{
        // 将除根结点之外的点都放入堆中，设置所有键为INFINITY
        // 遍历根结点发出的边，将其最短路径设为相应权值，并维持堆性质
        // RemoveTop，此结点已经取最短路径，如果为INFINITY，则终止算法
        // 否则，将其状态设为已标记，并设为根结点
        // loop back
        parent[root] = root;
        int reflection[g->V];
        int heap_real[g->V - 1];
        for (int i=0,j=0; i < g->V; i++) {
                if (i == root) {
                        distance[i] = 0;
                } else {
                        distance[i] = INFINITY;
                        heap_real[j++] = i;
                        reflection[i] = j;
                }
        }

        struct Edge *e;
        struct list_t *iter;
        int *heap = heap_real - 1;
        int base = 0;  /* euqal to distance[root]  */
        int size = g->V - 1;
        int length;
        do {
                iter = list_next(&(g->vertices + root)->link);
                for (; iter; iter = list_next(iter)) {
                        e = list_entry(iter, struct Edge, link);
                        length = base + e->weight;
                        if (length < distance[e->to]) {
                                HeapDecreaseKey(heap, size,
                                        distance, reflection,
                                        reflection[e->to], length);
                                parent[e->to] = root;
                        }
                }
                root = HeapRemoveTop(heap, size, distance, reflection);
                base = distance[root];

                if (distance[root] == INFINITY) {
                        /* remain nodes in heap  is not accessible */
                        return g->V - (size + 1); /* 返回强连通分支结点数 */
                }
        } while (size);

        /* successfull end algorightm  */
        return g->V;
}
```

导航

- 首页
- 社区主页
- 当前事件
- 最近更改
- 随机页面
- 使用帮助
- NOCOW地图
- 新手试练场

搜索

进入 搜索

工具箱

- 链入页面
- 链出更改
- 特殊页面
- 可打印版
- 永久链接

Dijkstra 二叉堆实现 C++

```
/*
ID:cmykrgb2
LANG:C++
TASK:butter
*/
/*
 * Problem: USACO Training 3.2.6
 * Author: Guo Jiabao
 * Time: 2009.4.6 10:20
 * State: Solved
 * Memo: Dijkstra + 堆
*/
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
#define MAXC 501
#define MAXN 801
#define MAXM 1451*2
#define INF 0x7FFFFFFF
using namespace std;
struct edge
{
        edge *next;
        int t,v;
}ES[MAXM];
struct HeapElement
{
        int key,value;
};
struct MinHeap
{
        HeapElement H[MAXN];
        int size;
        int Position[MAXN];
        void init(){H[size=0].value=-INF;}
        void ins(int key,int value)
        {
                int i,f;
                HeapElement p={key,value};
                for (i=++size;p.value<H[f=i>>1].value;i=f)
                {
                        H[i]=H[f];
                        Position[H[i].key]=i;
                }
                H[i]=p;
                Position[H[i].key]=i;
        }
        void decrease(int key,int value)
        {
                int i,f;
                HeapElement p={key,value};
                for (i=Position[key];p.value<H[f=i>>1].value;i=f)
                {
                        H[i]=H[f];
                        Position[H[i].key]=i;
                }
                H[i]=p;
                Position[H[i].key]=i;
        }
        void delmin()
        {
                int i,c;
                HeapElement p=H[size--];
                for (i=1;(c=i<<1)<=size;i=c)
                {
                        if (c+1<=size && H[c+1].value<H[c].value)
                                c++;
                        if (H[c].value<p.value)
                        {
                                H[i]=H[c];
                                Position[H[i].key]=i;
                        }
```

```
                        else
                                break;
                }
                H[i]=p;
                Position[H[i].key]=i;
        }
}H;
int N,M,C,EC=-1,Ans=INF;
int Cow[MAXC],sp[MAXN];
edge *V[MAXN];
inline void addedge(int a,int b,int c)
{
        ES[++EC].next=V[a];
        ES[EC].t=b;ES[EC].v=c;
        V[a]=&ES[EC];
}
void init()
{
        int i,a,b,c;
        freopen("butter.in","r",stdin);
        freopen("butter.out","w",stdout);
        scanf("%d%d%d",&C,&N,&M);
        for (i=1;i<=C;i++)
                scanf("%d",&Cow[i]);
        for (i=1;i<=M;i++)
        {
                scanf("%d%d%d",&a,&b,&c);
                addedge(a,b,c);
                addedge(b,a,c);
        }
}
void Dijkstra(int S)
{
        int i,j;
        sp[S]=0;
        H.decrease(S,0);
        for (i=S;;)
        {
                H.delmin();
                for (edge *k=V[i];k;k=k->next)
                {
                        if (sp[i]+k->v < sp[j=k->t])
                        {
                                sp[j]=sp[i]+k->v;
                                H.decrease(j,sp[j]);
                        }
                }
                if (H.size)
                        i=H.H[1].key;
                else
                        break;
        }
}
void solve()
{
        int i,j,Total;
        for (i=1;i<=N;i++)
        {
                H.init();
                for (j=1;j<=N;j++)
                {
                        H.ins(j,INF);
                        sp[j]=INF;
                }
                Total=0;
                Dijkstra(i);
                for (j=1;j<=C;j++)
                        Total+=sp[Cow[j]];
                if (Total<Ans)
                        Ans=Total;
        }
}
int main()
{
        init();
        solve();
        printf("%d\n",Ans);
        return 0;
}
```

Dijkstra 二叉堆实现 Pascal

```
{单源最短路径的Dijkstra算法。使用二叉堆挑选总复杂度O((e+v)logv)}
 const
   maxn=100;
 type
   link=^node;       //邻接表类型
    node=record
         v,w     :integer;
         next    :link;
       end;
  htype=record           //堆节点
          v,d,p  :integer;
        end;
 var
   n,s,hl          :integer;      //顶点数；源点；堆长度
   heap            :array[0..maxn]of htype;
   hpos            :array[1..maxn]of integer;    //hpos[v]:顶点v在堆中的位置
   g               :array[1..maxn]of link;     //邻接表
 procedure insert(u,v,w:integer);       //将权值为w的边(u,v)插入到邻接表
 var
   x      :link;
 begin
   new(x);
   x^.v:=v; x^.w:=w;
   x^.next:=g[u]; g[u]:=x;
 end;
 procedure init;         //初始化
 var
   u,v,w :integer;
 begin
   assign(input,'g.in');reset(input);
   readln(n,s);
   while not eof do
     begin
       readln(u,v,w);
       insert(u,v,w);insert(v,u,w);
     end;
 end;
 procedure swap(a,b:integer);            //交换堆中下标为a,b的节点
 begin
   heap[0]:=heap[a];heap[a]:=heap[b];heap[b]:=heap[0];
   hpos[heap[a].v]:=a;hpos[heap[b].v]:=b;
 end;
 procedure decrease(i:integer);          //减小键值并恢复堆性质
 begin
   while (i<>1)and(heap[i].d<heap[i div 2].d) do
     begin
       swap(i,i div 2);
       i:=i div 2;
     end;
 end;
 procedure heapify;         //恢复堆性质
 var
   i      :integer;
 begin
   i:=2;
   while i<=hl do
     begin
       if(i<hl)and(heap[i+1].d<heap[i].d) then inc(i);
       if heap[i].d<heap[i div 2].d then
         begin
           swap(i,i div 2);
           i:=i*2;
         end
       else break
     end;
 end;
 procedure relax(u,v,w:integer);         //松弛操作
 begin
   if w+heap[hpos[u]].d<heap[hpos[v]].d then
     begin
       heap[hpos[v]].p:=u;
       heap[hpos[v]].d:=w+heap[hpos[u]].d;
       decrease(hpos[v]);
     end;
```

```
end;
procedure dijkstra;          //主过程
var
  u     :integer;
  p     :link;
begin
  for u:=1 to n do          //初始化堆
    begin
      heap[u].v:=u;
      heap[u].d:=maxint;
      hpos[u]:=u;
    end;
  heap[s].p:=s;heap[s].d:=0;swap(1,s);
  hl:=n;
  while hl>0 do
   begin
     u:=heap[1].v;
     swap(1,hl);dec(hl);heapify;      //将堆的根节点移出堆并恢复堆性质
     p:=g[u];
     while p<>nil do
       begin
         if hpos[p^.v]<=hl then relax(u,p^.v,p^.w);    //对与u邻接且在堆中的顶点进行松弛操作
         p:=p^.next;
       end;
   end;
end;
procedure path(i:integer);
begin
 if heap[hpos[i]].p<>s then path(heap[hpos[i]].p);
 write('-->',i);
end;
procedure show;
var
 i     :integer;
begin
 for i:=1 to n do
   begin
     write(i:3,':',heap[hpos[i]].d:3,':',s);
     path(i);        //递归输出路径
     writeln;
   end
end;
{==========main===========}
begin
init;
dijkstra;
show;
end.
```