KMP：求模式字符串p与源字符串s的匹配位置

Next数组：除当前字符外的最长相同前缀和后缀:

‘’’’=’’’’ ==> Next[j]=k

Next数组求法：

Void GetNextval(char\* P,int next[])

{

Int pLen =strlen(P);

Next[0] = -1;

Int k = -1;

Int j = 0;

While(j<pLen-1)

{

If(k==-1||P[j]==P[k]) //P[k]表示前缀，P[j]表示后缀

{

++j;

++k;

If(P[j]!=P[k])

{

Next[j] = k;

}

Else//因为不能出现P[j] = P[next[j]],所以在出现的时候需要递归 k=next[k]=next[next[k]]

{

Next[j] = next[k];

}

}

Else

{

k=next[k];

}

}

}

求得next数组之后则KMP算法为：

Int KMPSearch(char\* S,char\* P)

{

Int i=0;

Int j=0;

Int sLen=strLen(S);

Int pLen=strLen(P);

While(i<sLen&&j<pLen)

{

If(j==-1||s[i]==p[j])

{

i++;

j++;

}

Else

{

j=next[j];

}

}

If(j==pLen)

{

Return i-j;

}

Else

{

Return -1;

}

}

Dijkstra算法

单源最短路径问题，即在图中求出给定顶点到其它任一顶点的最短路径。在弄清楚如何求算单源最短路径问题之前，必须弄清楚最短路径的最优子结构性质。

一.最短路径的最优子结构性质

   该性质描述为：如果P(i,j)={Vi....Vk..Vs...Vj}是从顶点i到j的最短路径，k和s是这条路径上的一个中间顶点，那么P(k,s)必定是从k到s的最短路径。下面证明该性质的正确性。

   假设P(i,j)={Vi....Vk..Vs...Vj}是从顶点i到j的最短路径，则有P(i,j)=P(i,k)+P(k,s)+P(s,j)。而P(k,s)不是从k到s的最短距离，那么必定存在另一条从k到s的最短路径P'(k,s)，那么P'(i,j)=P(i,k)+P'(k,s)+P(s,j)<P(i,j)。则与P(i,j)是从i到j的最短路径相矛盾。因此该性质得证。

二.Dijkstra算法

   由上述性质可知，如果存在一条从i到j的最短路径(Vi.....Vk,Vj)，Vk是Vj前面的一顶点。那么(Vi...Vk)也必定是从i到k的最短路径。为了求出最短路径，Dijkstra就提出了以最短路径长度递增，逐次生成最短路径的算法。譬如对于源顶点V0，首先选择其直接相邻的顶点中长度最短的顶点Vi，那么当前已知可得从V0到达Vj顶点的最短距离dist[j]=min{dist[j],dist[i]+matrix[i][j]}。根据这种思路，

假设存在G=<V,E>，源顶点为V0，U={V0},dist[i]记录V0到i的最短距离，path[i]记录从V0到i路径上的i前面的一个顶点。

1.从V-U中选择使dist[i]值最小的顶点i，将i加入到U中；

2.更新与i直接相邻顶点的dist值。(dist[j]=min{dist[j],dist[i]+matrix[i][j]})

3.知道U=V，停止。

代码实现:

/\*Dijkstra求单源最短路径 2010.8.26\*/

#include <iostream>

#include<stack>

#define M 100

#define N 100

using namespace std;

typedef struct node

{

int matrix[N][M]; //邻接矩阵

int n; //顶点数

int e; //边数

}MGraph;

void DijkstraPath(MGraph g,int \*dist,int \*path,int v0) //v0表示源顶点

{

int i,j,k;

bool \*visited=(bool \*)malloc(sizeof(bool)\*g.n);

for(i=0;i<g.n;i++) //初始化

{

if(g.matrix[v0][i]>0&&i!=v0)

{

dist[i]=g.matrix[v0][i];

path[i]=v0; //path记录最短路径上从v0到i的前一个顶点

}

else

{

dist[i]=INT\_MAX; //若i不与v0直接相邻，则权值置为无穷大

path[i]=-1;

}

visited[i]=false;

path[v0]=v0;

dist[v0]=0;

}

visited[v0]=true;

for(i=1;i<g.n;i++) //循环扩展n-1次

{

int min=INT\_MAX;

int u;

for(j=0;j<g.n;j++) //寻找未被扩展的权值最小的顶点

{

if(visited[j]==false&&dist[j]<min)

{

min=dist[j];

u=j;

}

}

visited[u]=true;

for(k=0;k<g.n;k++) //更新dist数组的值和路径的值

{

if(visited[k]==false&&g.matrix[u][k]>0&&min+g.matrix[u][k]<dist[k])

{

dist[k]=min+g.matrix[u][k];

path[k]=u;

}

}

}

}

void showPath(int \*path,int v,int v0) //打印最短路径上的各个顶点

{

stack<int> s;

int u=v;

while(v!=v0)

{

s.push(v);

v=path[v];

}

s.push(v);

while(!s.empty())

{

cout<<s.top()<<" ";

s.pop();

}

}

int main(int argc, char \*argv[])

{

int n,e; //表示输入的顶点数和边数

while(cin>>n>>e&&e!=0)

{

int i,j;

int s,t,w; //表示存在一条边s->t,权值为w

MGraph g;

int v0;

int \*dist=(int \*)malloc(sizeof(int)\*n);

int \*path=(int \*)malloc(sizeof(int)\*n);

for(i=0;i<N;i++)

for(j=0;j<M;j++)

g.matrix[i][j]=0;

g.n=n;

g.e=e;

for(i=0;i<e;i++)

{

cin>>s>>t>>w;

g.matrix[s][t]=w;

}

cin>>v0; //输入源顶点

DijkstraPath(g,dist,path,v0);

for(i=0;i<n;i++)

{

if(i!=v0)

{

showPath(path,i,v0);

cout<<dist[i]<<endl;

}

}

}

return 0;

}

最近公共祖先之Tarjan算法