# 求两点之间所有路径的算法

作者：finallyly 出处：博客园（如若转载请注明作者和出处）

最近在实现一个算法，算法之内有一个子算法是求有向图内两个定点（原点和目的点）之间的全部路径。在网上翻阅了大部分资料，发现给出的算法和代码要么只能解决DAG（有向无环图）的两定点之间所有路径问题，要么就是算法本身存在若干漏洞，连DAG图也无法解决。花费了一天的时间，自己写了个求简单有向图中（包括dag和非dag）两定点之间所有路径的算法，特共享出来。

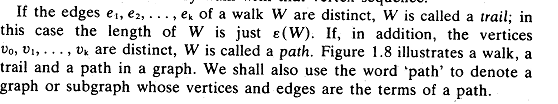
文章将按如下组织，首先给出path的定义，其次给出dag的定义，然后给出算法的伪代码，之后是算法的C++实现以及实验结果。

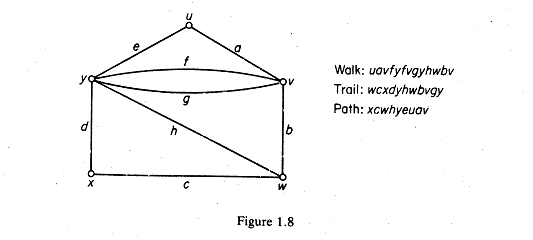
# Path的定义

Path的定义是建立在walk,基础上的。参见Bondy的《Graph Theory With Applications》









由上面的定义，我问可以得出path是一个结点和边交叠出现的序列，并且在这个序列中结点不能重复，边也不能重复。

# DAG的定义

DAG（Directed Acyclic Graph）:即不存在环路的有向图。或者说是DFS过程中不出现回边(backc edge)的图。如图2-1就是一个DAG。更一般的有向图见图2-2



2‑1 DAG

++++++++++++++++++++++++++++++++++++++

第1条路径是：0-->1-->4-->11

第2条路径是：0-->1-->3-->11

第3条路径是：0-->1-->2-->6-->10-->11

第4条路径是：0-->1-->2-->6-->9-->11

第5条路径是：0-->1-->2-->5-->8-->11

第6条路径是：0-->1-->2-->5-->7-->11

++++++++++++++++++++++++++++++++++++++

‑2 Digraph

该图对应的矩阵型存储格式为：

它的路径有：

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

第一条路径：0->1->2->3->4

第二条路径：0->1->2->4

第三条路径：0->1->3->2->4

第四条路径：0->1->3->4

第五条路径：0->1->4

第六条路径：0->2->1->3->4

第七条路径：0->2->1->4

第八条路径：0->2->3->1->4

第九条路径：0->2->3->4

第十条路径：0->2->4

第十一条路径：0->3->1->2->4

第十二条路径：0->3->1->4

第十三条路径：0->3->2->1->4

第十四条路径：0->3->2->4

第十五条路径：0->3->4

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# 算法设计

待求解问题是“求原点和目的点之间的全部路径”，求解问题的第一步，我们需要确定这是一个P问题还是NP问题。对于P问题，可以直接设计算法；对于NP问题，则需要一些近似手段。值得庆幸的是这是一个P问题。算法最大复杂度为

证明如下：

假定有向图为N个节点的简单完全图，即每个节点都与其他N-1个节点有边相连。起始结点和结束节点确定，那么我们需要排列中间的N-2个节点，对于第一个非固定的节点，它有N-2种可能取值。。。以此类推得到上述答案。

求两定点之间的全部路径，其根本是一个涉及到搜索和回溯的问题。我们设计算法时所关心的首要问题是：按照何种顺序搜索和回溯才能保证路径可以不重不漏地被全部找到。

如下是算法设计部分

图的存储结构：邻接矩阵。Arcs

工作结构：结点栈 mystack;

状态保存结构：

1. VertexStatus[]={0,0,0,1,1,…}。当结点未进栈或者已经出栈，则其对应的状态为0，否则状态为1；
2. ArcStatus[][]={0,0,1,0,1…..}当且仅当边的两个结点都在栈外时，边的状态才为0，否则为1。

注意我们只所以设计如上结点、边两个状态存储结构，就是依据于path的定义，结点不重复，边不重复。具有边状态存储结构，也是我的算法与其他算法根本上的不同。

不失一般性，我们假设原点的编号最小为0,目标点的编号最大N。我们的问题转换成了，求最小编号的节点与最大编号的节点之间的所有路径。

Intial :

Paths={}//路径集合

VertexStatus[]={0};//全部置0

ArcStatus[][]={0};////全部置0

mystack.push(0);

VertexStatus[0]=1;

While(!mystack.empty())

{

Int elem= mystack.top();//获得栈顶元素

if(elem==N)//找到了一条路径

{

path=Traverse(mystack);

Paths.add(path);

VertexStatus[elem]=0;

UpdateArcStatus();//更新ArcStatus[][]，使得所有两个端点都不在栈内的边的状态为0

mystack.pop();//移除栈顶元素

}

else

{

i=0;

For(;i<N;i++)

{ if(VertexStatus[i]=0&&ArcStatus[elem][i]=0&&Arcs.contain(elem,i))

{

VertexStatus[i]=1;

ArcStatus[elem][i]=1;

Mystack.push(i);//入栈

break;

}

}

if(i=N)//该节点没有符合要求的后续节点

{

VertexStatus[elem]=0;

UpdateArcStaus();////更新ArcStatus[][]，使得所有两个端点都不在栈内的边的状为0

Mystack.pop();//出栈

}

}

}

# 算法的C++实现

Editdistance.h

#pragma once

#include "common.h"

#include<map>

#include <cmath>

#include <stack>

class EditDistance

{

public:

EditDistance(wstring s, wstring t);

map<pair<int,int>,char>graphR;//int,int代表首尾节点编号，char,表示边

map<pair<int,int>,int>Vertex;//状态节点号对应表

void GetPaths(vector<vector<int>>&paths);//求两结点之间的所有边

}

Editdistance.cpp

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* 求原点到终点的所有路径，即所有最优结果 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void EditDistance:: GetPaths(vector<vector<int>>&paths)

{

stack<int>mystack;

vector<int> singlepath;

int \*vertexsStatus=new int[Vertex.size()];//0，未在栈内，已经在栈内。

map<pair<int,int>,int>arcstatus;

for (int i=0;i<Vertex.size();i++)

{

vertexsStatus[i]=0;

}

for (map<pair<int,int>,char>::iterator it=graphR.begin();it!=graphR.end();it++)

{

arcstatus[it->first]=0;

}

mystack.push(0);

vertexsStatus[0]=1;

int justpopup=-1;//保存刚刚出栈的元素

while(!mystack.empty())

{

int elem=mystack.top();

if (elem==Vertex.size()-1)//出栈的第一个条件找到了目的节点

{ //以下代码完成栈的遍历

while(!mystack.empty())

{

int tmp=mystack.top();

mystack.pop();

singlepath.push\_back(tmp);

}

for (vector<int>::reverse\_iterator rit=singlepath.rbegin();rit!=singlepath.rend();rit++)

{

mystack.push(\*rit);

}

paths.push\_back(singlepath);

singlepath.clear();

vertexsStatus[elem]=0;

for (int k=0;k<Vertex.size();k++)

{

if (vertexsStatus[k]==0)

{

if (arcstatus.count(make\_pair(elem,k)))

{

arcstatus[make\_pair(elem,k)]=0;

}

}

}

mystack.pop();

}

else

{ int i=0;

for (;i<Vertex.size();i++)

{

if (graphR.count(make\_pair(elem,i))&&vertexsStatus[i]==0&&arcstatus[make\_pair(elem,i)]==0)

{

mystack.push(i);

vertexsStatus[i]=1;

arcstatus[make\_pair(elem,i)]=1;

//graphR.erase(make\_pair(elem,i));

break;

}

}

if (i==Vertex.size())//出栈的第二个条件，没有可以往栈内添加的后续节点了。

{

int elemtmp=mystack.top();

vertexsStatus[elemtmp]=0;

for (int k=0;k<Vertex.size();k++)

{

if (vertexsStatus[k]==0)

{

if (arcstatus.count(make\_pair(elemtmp,k)))

{

arcstatus[make\_pair(elemtmp,k)]=0;

}

}

}

mystack.pop();

}

}

}

delete vertexsStatus;

}

Main.cpp

int \_tmain(int argc, \_TCHAR\* argv[])

{

wstring g=L"ace";

wstring s=L"aec";

EditDistance editdistance(s,g);

/\*

editdistance.Vertex[make\_pair(0,0)]=0;

editdistance.Vertex[make\_pair(0,1)]=1;

editdistance.Vertex[make\_pair(1,0)]=2;

editdistance.Vertex[make\_pair(1,1)]=3;

editdistance.Vertex[make\_pair(1,2)]=4;

editdistance.Vertex[make\_pair(2,1)]=5;

editdistance.Vertex[make\_pair(3,1)]=6;

editdistance.Vertex[make\_pair(4,1)]=7;

editdistance.Vertex[make\_pair(1,3)]=8;

editdistance.Vertex[make\_pair(1,4)]=9;

editdistance.Vertex[make\_pair(2,3)]=10;

editdistance.Vertex[make\_pair(4,5)]=11;

editdistance.graphR[make\_pair(0,1)]='e';;

editdistance.graphR[make\_pair(1,2)]='e';

editdistance.graphR[make\_pair(1,3)]='e';

editdistance.graphR[make\_pair(1,4)]='e';

editdistance.graphR[make\_pair(2,5)]='e';

editdistance.graphR[make\_pair(2,6)]='e';

editdistance.graphR[make\_pair(5,7)]='e';

editdistance.graphR[make\_pair(5,8)]='e';

editdistance.graphR[make\_pair(6,9)]='e';

editdistance.graphR[make\_pair(6,10)]='e';

editdistance.graphR[make\_pair(3,11)]='e';

editdistance.graphR[make\_pair(4,11)]='e';

editdistance.graphR[make\_pair(7,11)]='e';

editdistance.graphR[make\_pair(8,11)]='e';

editdistance.graphR[make\_pair(9,11)]='e';

editdistance.graphR[make\_pair(10,11)]='e';

\*/

editdistance.Vertex[make\_pair(0,0)]=0;

editdistance.Vertex[make\_pair(0,1)]=1;

editdistance.Vertex[make\_pair(1,0)]=2;

editdistance.Vertex[make\_pair(1,1)]=3;

editdistance.Vertex[make\_pair(1,2)]=4;

editdistance.graphR[make\_pair(0,1)]='e';

editdistance.graphR[make\_pair(0,2)]='e';

editdistance.graphR[make\_pair(0,3)]='e';

editdistance.graphR[make\_pair(1,0)]='e';

editdistance.graphR[make\_pair(1,2)]='e';

editdistance.graphR[make\_pair(1,3)]='e';

editdistance.graphR[make\_pair(1,4)]='e';

editdistance.graphR[make\_pair(2,0)]='e';

editdistance.graphR[make\_pair(2,1)]='e';

editdistance.graphR[make\_pair(2,3)]='e';

editdistance.graphR[make\_pair(2,4)]='e';

editdistance.graphR[make\_pair(3,1)]='e';

editdistance.graphR[make\_pair(3,2)]='e';

editdistance.graphR[make\_pair(3,0)]='e';

editdistance.graphR[make\_pair(3,4)]='e';

editdistance.graphR[make\_pair(4,1)]='e';

editdistance.graphR[make\_pair(4,2)]='e';

editdistance.graphR[make\_pair(4,3)]='e';

vector<vector<int>>paths;

editdistance.GetPaths(paths);

for (vector<vector<int>>::iterator it=paths.begin();it!=paths.end();it++)

{ cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*路径\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

for (vector<int>::iterator subit=it->begin();subit!=it->end();subit++)

{

cout<<\*subit<<" ";

}

cout<<endl;

}

cout<<"finish"<<endl;

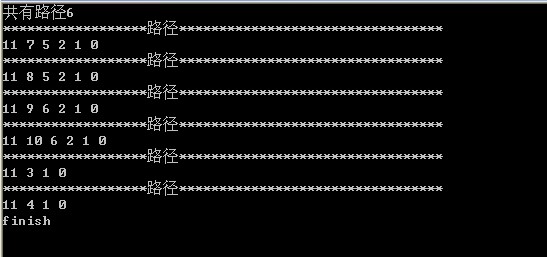
int f;

cin>>f;

}

# 实验结果

在DAG（图2-1）上运行算法的结果



在一般的Digraph(图2-2)上运行上述算法结果如下

