# 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。

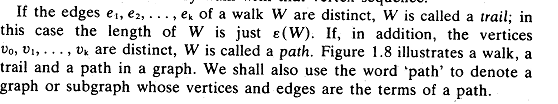
文章将按如下组织，首先给出path的定义，其次给出dag的定义，然后给出算法的伪代码，之后是算法的C++实现以及实验结果。

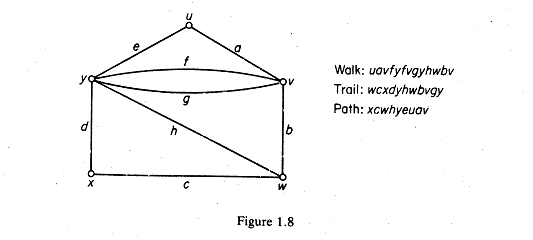
# Path的定义

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

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

2‑2 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

1. Editdistance.h
2. #pragma once
3. #include "common.h"
4. #include<map>
5. #include <cmath>
6. #include <stack>
7. **class** EditDistance
8. {
9. **public**:
10. EditDistance(wstring s, wstring t);
11. map<pair<**int**,**int**>,**char**>graphR;//int,int代表首尾节点编号，char,表示边
12. map<pair<**int**,**int**>,**int**>Vertex;//状态节点号对应表
13. **void** GetPaths(vector<vector<**int**>>&paths);//求两结点之间的所有边
14. }
15. Editdistance.cpp
16. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
17. /\* 求原点到终点的所有路径，即所有最优结果                                                                    \*/
18. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
20. **void** EditDistance:: GetPaths(vector<vector<**int**>>&paths)
21. {
22. stack<**int**>mystack;
23. vector<**int**> singlepath;
24. **int** \*vertexsStatus=**new** **int**[Vertex.size()];//0，未在栈内，已经在栈内。
25. map<pair<**int**,**int**>,**int**>arcstatus;
26. **for** (**int** i=0;i<Vertex.size();i++)
27. {
28. vertexsStatus[i]=0;
29. }
30. **for** (map<pair<**int**,**int**>,**char**>::iterator it=graphR.begin();it!=graphR.end();it++)
31. {
32. arcstatus[it->first]=0;
33. }

36. mystack.push(0);
37. vertexsStatus[0]=1;
38. **int** justpopup=-1;//保存刚刚出栈的元素
39. **while**(!mystack.empty())
40. {
41. **int** elem=mystack.top();
42. **if** (elem==Vertex.size()-1)//出栈的第一个条件找到了目的节点
43. {  //以下代码完成栈的遍历
44. **while**(!mystack.empty())
45. {
46. **int** tmp=mystack.top();
47. mystack.pop();
48. singlepath.push\_back(tmp);
49. }
50. **for** (vector<**int**>::reverse\_iterator rit=singlepath.rbegin();rit!=singlepath.rend();rit++)
51. {
52. mystack.push(\*rit);
54. }
55. paths.push\_back(singlepath);
56. singlepath.clear();
58. vertexsStatus[elem]=0;
59. **for** (**int** k=0;k<Vertex.size();k++)
60. {
61. **if** (vertexsStatus[k]==0)
62. {
63. **if** (arcstatus.count(make\_pair(elem,k)))
64. {
65. arcstatus[make\_pair(elem,k)]=0;
66. }
68. }
69. }

72. mystack.pop();
74. }
75. **else**
76. {   **int** i=0;
77. **for** (;i<Vertex.size();i++)
78. {
79. **if** (graphR.count(make\_pair(elem,i))&&vertexsStatus[i]==0&&arcstatus[make\_pair(elem,i)]==0)
80. {
81. mystack.push(i);
82. vertexsStatus[i]=1;
83. arcstatus[make\_pair(elem,i)]=1;

86. //graphR.erase(make\_pair(elem,i));
88. **break**;
89. }
90. }
91. **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。
92. {
93. **int** elemtmp=mystack.top();
94. vertexsStatus[elemtmp]=0;
95. **for** (**int** k=0;k<Vertex.size();k++)
96. {
97. **if** (vertexsStatus[k]==0)
98. {
99. **if** (arcstatus.count(make\_pair(elemtmp,k)))
100. {
101. arcstatus[make\_pair(elemtmp,k)]=0;
102. }
103. }
104. }

107. mystack.pop();

110. }
112. }



117. }

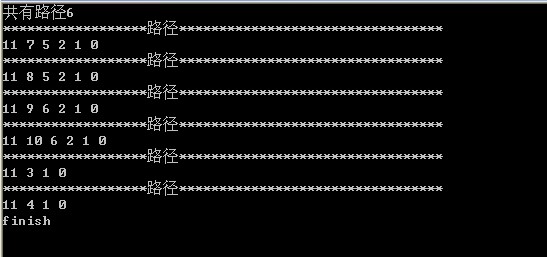
120. **delete** vertexsStatus;
122. }
123. Main.cpp
124. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
125. {
127. wstring g=L"ace";
128. wstring s=L"aec";
129. EditDistance editdistance(s,g);

132. /\*
133. editdistance.Vertex[make\_pair(0,0)]=0;
134. editdistance.Vertex[make\_pair(0,1)]=1;
135. editdistance.Vertex[make\_pair(1,0)]=2;
136. editdistance.Vertex[make\_pair(1,1)]=3;
137. editdistance.Vertex[make\_pair(1,2)]=4;
138. editdistance.Vertex[make\_pair(2,1)]=5;
139. editdistance.Vertex[make\_pair(3,1)]=6;
140. editdistance.Vertex[make\_pair(4,1)]=7;
141. editdistance.Vertex[make\_pair(1,3)]=8;
142. editdistance.Vertex[make\_pair(1,4)]=9;
143. editdistance.Vertex[make\_pair(2,3)]=10;
144. editdistance.Vertex[make\_pair(4,5)]=11;
145. editdistance.graphR[make\_pair(0,1)]='e';;
146. editdistance.graphR[make\_pair(1,2)]='e';
147. editdistance.graphR[make\_pair(1,3)]='e';
148. editdistance.graphR[make\_pair(1,4)]='e';
149. editdistance.graphR[make\_pair(2,5)]='e';
150. editdistance.graphR[make\_pair(2,6)]='e';
151. editdistance.graphR[make\_pair(5,7)]='e';
152. editdistance.graphR[make\_pair(5,8)]='e';
153. editdistance.graphR[make\_pair(6,9)]='e';
154. editdistance.graphR[make\_pair(6,10)]='e';
155. editdistance.graphR[make\_pair(3,11)]='e';
156. editdistance.graphR[make\_pair(4,11)]='e';
157. editdistance.graphR[make\_pair(7,11)]='e';
158. editdistance.graphR[make\_pair(8,11)]='e';
159. editdistance.graphR[make\_pair(9,11)]='e';
160. editdistance.graphR[make\_pair(10,11)]='e';
161. \*/
163. editdistance.Vertex[make\_pair(0,0)]=0;
164. editdistance.Vertex[make\_pair(0,1)]=1;
165. editdistance.Vertex[make\_pair(1,0)]=2;
166. editdistance.Vertex[make\_pair(1,1)]=3;
167. editdistance.Vertex[make\_pair(1,2)]=4;
168. editdistance.graphR[make\_pair(0,1)]='e';
169. editdistance.graphR[make\_pair(0,2)]='e';
170. editdistance.graphR[make\_pair(0,3)]='e';
171. editdistance.graphR[make\_pair(1,0)]='e';
172. editdistance.graphR[make\_pair(1,2)]='e';
173. editdistance.graphR[make\_pair(1,3)]='e';
174. editdistance.graphR[make\_pair(1,4)]='e';
175. editdistance.graphR[make\_pair(2,0)]='e';
176. editdistance.graphR[make\_pair(2,1)]='e';
177. editdistance.graphR[make\_pair(2,3)]='e';
178. editdistance.graphR[make\_pair(2,4)]='e';
179. editdistance.graphR[make\_pair(3,1)]='e';
180. editdistance.graphR[make\_pair(3,2)]='e';
181. editdistance.graphR[make\_pair(3,0)]='e';
182. editdistance.graphR[make\_pair(3,4)]='e';
183. editdistance.graphR[make\_pair(4,1)]='e';
184. editdistance.graphR[make\_pair(4,2)]='e';
185. editdistance.graphR[make\_pair(4,3)]='e';

188. vector<vector<**int**>>paths;
189. editdistance.GetPaths(paths);
190. **for** (vector<vector<**int**>>::iterator it=paths.begin();it!=paths.end();it++)
191. {   cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*路径\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;
192. **for** (vector<**int**>::iterator subit=it->begin();subit!=it->end();subit++)
193. {
194. cout<<\*subit<<" ";
195. }
196. cout<<endl;
197. }
199. cout<<"finish"<<endl;
200. **int** f;
201. cin>>f;
202. }

# 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

