

## 作业 4 图型结构及其应用

1. 分别实现无向图的邻接矩阵和邻接表存储结构的建立算法，分析和比较各建立算法的时间复杂度以及存储结构的占用情况。

为简化后续程序，本作业将邻接矩阵和邻接表储存在同一结构体中，具体如下：

```
20 struct graph
21 {
22     int n,e;//图的顶点数与边数
23     int edge[N][N] = {0}; //邻接矩阵
24     int vertex[N]; //顶点表
25     struct vertexNode vertexlist[N]; //顶点表
26 };
```

其中，struct vertexNode 为邻接表的表头数组，定义如下：

```
14 struct vertexNode
15 {
16     int vertex; //数据域
17     EdgeNode *phead; //边链表头指针
18 };
19
```

EdgeNode 为后续连接的节点，定义为：

```
7 struct EdgeNode
8 {
9     int adjvex; //邻接点域
10    int weight; //边权重
11    EdgeNode *next; //指针域
12 };
```

建立思路为输入顶点个数以及边数后，逐边对邻接矩阵和邻接表进行修改或者插入，具体建立过程如下：

A. 无向图的邻接矩阵的建立

```
108 void create_matrix(struct graph *G)
109 {
110     cout<<"图的顶点数为: ";
111     cin>>G->n;
112     cout<<"图的边数为: ";
113     cin>>G->e;
114     cout<<"此无向图的顶点为: ";
115     for(int i = 0; i < G->n; i++)
116     {
117         G->vertex[i] = i;
118         cout<<i<<" ";
119     }
120     cout<<endl;
121     for(int i = 0; i < G->e; i++)
122     {
123         int m,n,w;//边依附的两个顶点序号m, n; 边权值w
```

```

124         cout<<"第"<<i+1<<"条边的起止结点以及权值为: ";
125         cin>>m>>n>>w;
126         G->edge[m][n] = w;
127         G->edge[n][m] = w;
128     }
129     show_matrix(G);
130 }

```

## B. 无向图的邻接表的建立

```

101 void create_list(struct graph *G)
102 {
103     cout<<"图的顶点数为: ";
104     cin>>G->n;
105     cout<<"图的边数为: ";
106     cin>>G->e;
107     cout<<"此无向图的顶点为: ";
108     for(int i = 0; i < G->n; i++)
109     {
110         G->vertex[i] = i;
111         cout<<i<<" ";
112         G->vertexlist[i].vertax = i;
113         G->vertexlist[i].phead = NULL;
114     }
115     cout<<endl;
116     for(int i = 0; i < G->e; i++)
117     {
118         int head,tail,w;//边依附的两个顶点序号m, n; 边权值w
119         cout<<"第"<<i+1<<"条边的起止结点以及权值为: ";
120         cin>>head>>tail>>w;
121         EdgeNode *p = new EdgeNode;
122         p->adjvex = tail; p->weight = w;
123         p->next = G->vertexlist[head].phead;
124         G->vertexlist[head].phead = p;
125         EdgeNode *q = new EdgeNode;
126         q->adjvex = head; q->weight = w;
127         q->next = G->vertexlist[tail].phead;
128         G->vertexlist[tail].phead = q;
129     }
130     show_list(G);
131 }

```

由程序可得，设 $e$ 为图所含边数， $n$ 为顶点数目，无论邻接表或是邻接矩阵的建立时间复杂度均为 $O(e)$ ，但是邻接表的空间复杂度为 $O(n)$ ，而邻接矩阵的空间复杂度为 $O(n^2)$ 。

## 2. 实现无向图的邻接矩阵和邻接表两种存储结构的相互转换算法。

转换过程和建立过程相似，由邻接表或邻接矩阵读出边信息后再存入对应的矩阵或者表中即可，具体程序如下：

### A. 邻接表转换为邻接矩阵

```

133 void list_to_matrix(struct graph *G, struct graph *g)
134 {
135     for(int i = 0; i < G->n; i++)
136         g->vertex[i] = G->vertexlist[i].vertax;
137     g->n = G->n;
138     g->e = G->e;
139     for(int i = 0; i < G->n; i++)
140     {
141         EdgeNode *p;
142         p = G->vertexlist[i].phead;
143         while(p != NULL)
144         {
145             int m = i, n = p->adjvex;
146             g->edge[m][n] = p->weight;
147             p = p->next;
148         }
149     }
150     cout<<"邻接表转换为邻接矩阵....."<<endl;
151     show_matrix(g);
152 }

```

### B. 邻接矩阵转换为邻接表

```

132 void matrix_to_list(struct graph *G, struct graph *g)
133 {
134     for(int i = 0; i < G->n; i++)
135     {
136         g->vertexlist[i].vertax = G->vertex[i];
137         g->vertexlist[i].phead = NULL;
138     }
139     g->n = G->n;
140     g->e = G->e;
141     for(int i = 0; i < G->n; i++)
142         for(int j = 0; j < G->n; j++)
143             if(G->edge[i][j] != 0)
144             {
145                 int w = G->edge[i][j];
146                 EdgeNode *p = new EdgeNode;
147                 p->adjvex = j; p->weight = w;
148                 p->next = g->vertexlist[i].phead;
149                 g->vertexlist[i].phead = p;
150             }
151     cout<<"邻接矩阵转换为邻接表....."<<endl;
152     show_list(g);
153 }

```

由程序可知，设 $e$ 为图所含边数， $n$ 为顶点数目，则邻接表转换为邻接矩阵的时间复杂度为 $O(n + e)$ ，邻接矩阵转换为邻接表的时间复杂度为 $O(n^2)$ 。

3. 在上述两种存储结构上，分别实现无向图的深度优先搜索（递归和非递归）和广度优先搜索算法。并以适当的方式存储和展示相应的搜索结果，包括：深度优先或广度优先生成森林（或生成树）、深度优先或广度优先序列和深度优先或广度优先编号。并分析搜索算法的时间复杂度和空间复杂度。

## A. 邻接表

### 1. 深度优先搜索(递归)

```
245 void DFS_list_recursion(struct graph G)
246 {
247     int i, count = 1;
248     for(int j = 0; j < G.n; j++)
249         visited[j] = false;
250     cout<<"深度优先序列为（递归）: ";
251     int search_tree[N][N] = {0};
252     for(int i = 0; i < G.n; i++)
253         if(!visited[i]) DFS1(G,i,&count,search_tree);
254     cout<<endl;
255     cout<<"深度优先编号为（递归）: "<<endl;
256     show_search(dfn);
257     cout<<"深度优先生成树的邻接表为: "<<endl;
258     for(int i = 0; i < G.n; i++)
259         for(int j = 0; j < G.n; j++)
260             G.edge[i][j] = search_tree[i][j];
261     struct graph g;
262     matrix_to_list(&G, &g);
263     //clear(&g);
264 }
```

```
224 void DFS1(struct graph G, int i, int *count, int search_tree[][N])
225 {
226     EdgeNode *p = G.vertexlist[i].phead;
227     if(!visited[i])
228     {
229         cout<<G.vertexlist[i].vertax<<" ";
230         dfn[i] = (*count)++;
231         visited[i] = true;
232     }
233     while(p != NULL)
234     {
235         if(!visited[p->adjvex])
236         {
237             search_tree[i][p->adjvex] = p->weight;
238             search_tree[p->adjvex][i] = p->weight;
239             DFS1(G,p->adjvex,count,search_tree);
240         }
241         p = p->next;
242     }
243 }
```

### 2. 深度优先搜索(非递归)，基于栈结构

```
170 void DFS_list(struct graph G)
171 {
172     int i, count = 1;
173     stack<int> s;
174     for(int j = 0; j < G.n; j++)
175         visited[j] = false;
176     cout<<"深度优先序列为（非递归）: ";
177     int search_tree[N][N] = {0};
178     for(int i = 0; i < G.n; i++)
179     {
180         if(!visited[i])
181         {
182             cout<<G.vertexlist[i].vertax<<" ";
183             dfn[i] = count++;
184             visited[i] = true;
```

```

185         s.push(i);
186     }
187
188     while(!s.empty())
189     {
190         EdgeNode *p = G.vertexlist[s.top()].phead;
191         int m = G.vertexlist[s.top()].vertax;
192         while(p != NULL)
193         {
194             if(!visited[p->adjvex])
195             {
196                 search_tree[m][p->adjvex] = p->weight;
197                 search_tree[p->adjvex][m] = p->weight;
198                 cout<<p->adjvex<<" ";
199                 dfn[p->adjvex] = count++;
200                 visited[p->adjvex] = true;
201                 s.push(p->adjvex);
202                 m = G.vertexlist[p->adjvex].vertax;
203                 p = G.vertexlist[p->adjvex].phead;
204             }
205             else p = p->next;
206         }
207         p = G.vertexlist[s.top()].phead;
208         m = G.vertexlist[s.top()].vertax;
209         s.pop();
210     }
211 }
212 cout<<endl;
213 cout<<"深度优先编号为（非递归）： "<<endl;
214 show_search(dfn);
215 cout<<"深度优先生成树的邻接表为： "<<endl;
216 for(int i = 0; i < G.n; i++)
217     for(int j = 0; j < G.n; j++)
218         G.edge[i][j] = search_tree[i][j];
219 struct graph g;
220 matrix_to_list(&G, &g);
221 //clear(&g);

```

### 3. 广度优先搜索，基于队列结构

```

266 void BFS_list(struct graph G)
267 {
268     int i, count = 1;
269     queue<int> q;
270     for(int j = 0; j < G.n; j++)
271         visited[j] = false;
272     cout<<"广度优先序列为： ";
273     int search_tree[N][N] = {0};
274     for(int i = 0; i < G.n; i++)
275     {
276         if(!visited[i])
277         {
278             cout<<G.vertexlist[i].vertax<<" ";
279             bfn[i] = count++;
280             visited[i] = true;
281             q.push(i);
282         }
283     }
284     while(!q.empty())
285     {
286         int i = q.front();
287         q.pop();
288         EdgeNode *p = G.vertexlist[i].phead;
289         while(p != NULL)
290         {
291             if(!visited[p->adjvex])
292             {
293                 search_tree[i][p->adjvex] = p->weight;

```

```

293         search_tree[p->adjvex][i] = p->weight;
294         cout<<p->adjvex<<" ";
295         bfn[p->adjvex] = count++;
296         visited[p->adjvex] = true;
297         q.push(p->adjvex);
298     }
299     p = p->next;
300 }
301 }
302 }
303 cout<<endl;
304 cout<<"广度优先编号为: "<<endl;
305 show_search(bfn);
306 cout<<"广度优先生成树的邻接表为: "<<endl;
307 for(int i = 0; i < G.n; i++)
308     for(int j = 0; j < G.n; j++)
309         G.edge[i][j] = search_tree[i][j];
310 struct graph g;
311 matrix_to_list(&G, &g);
312 //clear(&g);
313 }

```

由程序可得，设 $n$ 为顶点数， $e$ 为边数，则对邻接表进行深度（广度）优先搜索的时间复杂度为 $O(n + e)$ ，对每个顶点都访问一次的时间复杂度为 $O(n)$ ，而图的遍历过程是对每个顶点通过边查找邻接顶点的过程，这一过程开销为 $O(e)$ ，而空间复杂度为 $O(n)$ ，这是因为最坏的情况下所有的顶点进入到栈或队列当中。

## B. 邻接矩阵

### 1. 深度优先搜索(递归)

```

223 void DFS_matrix_recursion(struct graph G)
224 {
225     int i, count = 1;
226     for(int j = 0; j < G.n; j++)
227         visited[j] = false;
228     cout<<"深度优先序列为(递归): ";
229     int search_tree[N][N] = {0};
230     for(int i = 0; i < G.n; i++)
231         if(!visited[i]) DFS2(G,i,&count,search_tree);
232     cout<<endl;
233     cout<<"深度优先编号为(递归): "<<endl;
234     show_search(dfn);
235     cout<<"深度优先生成树矩阵为: "<<endl;
236     for(int i = 0; i < G.n; i++)
237     {
238         for(int j = 0; j < G.n; j++)
239             cout<<search_tree[i][j]<<" ";
240         cout<<endl;
241     }
242 }

```

```

204 void DFS2(struct graph G, int i, int *count, int search_tree[][N])
205 {
206     if(!visited[i])
207     {
208         cout<<G.vertex[i]<<" ";
209         dfn[i] = (*count)++;
210         visited[i] = true;
211     }
212     for(int j = 0; j < G.n; j++)
213     {
214         if((G.edge[i][j] != 0) && (!visited[j]))
215         {
216             search_tree[i][j] = G.edge[i][j];
217             search_tree[j][i] = G.edge[i][j];
218             DFS2(G,j,count,search_tree);
219         }
220     }
221 }
222 }

```

## 2. 度优先搜索(非递归)，基于栈结构

```

155 void DFS_matrix(struct graph G)
156 {
157     int i, count = 1;
158     stack<int> s;
159     for(int j = 0; j < G.n; j++)
160         visited[j] = false;
161     cout<<"深度优先序列为（非递归）： ";
162     int search_tree[N][N] = {0};
163     for(int i = 0; i < G.n; i++)
164     {
165         if(!visited[i])
166         {
167             cout<<G.vertex[i]<<" ";
168             dfn[i] = count++;
169             visited[i] = true;
170             s.push(i);
171         }
172         int j = 0, f = i;
173         while(!s.empty())
174         {
175             while((j >= 0) && (j < G.n))
176             {
177                 if((G.edge[f][j] != 0) && (!visited[j]))
178                 {
179                     search_tree[f][j] = G.edge[f][j];
180                     search_tree[j][f] = G.edge[f][j];
181                     cout<<G.vertex[j]<<" ";
182                     dfn[j] = count++;
183                     visited[j] = true;
184                     s.push(j);
185                     f = j; j = 0;
186                 }
187                 else j++;
188             }
189             f = s.top();s.pop();
190             j = 0;

```

```

191     }
192 }
193 cout<<endl;
194 cout<<"深度优先编号为（非递归）： "<<endl;
195 show_search(dfn);
196 cout<<"深度优先生成树矩阵为： "<<endl;
197 for(int i = 0; i < G.n; i++)
198 {
199     for(int j = 0; j < G.n; j++)
200         cout<<search_tree[i][j]<<" ";
201     cout<<endl;
202 }
203 }

```

### 3. 广度优先搜索，基于队列结构

```

244 void BFS_matrix(struct graph G)
245 {
246     int i, count = 1;
247     queue<int> q;
248     for(int j = 0; j < G.n; j++)
249         visited[j] = false;
250     cout<<"广度优先序列为： ";
251     int search_tree[N][N] = {0};
252     for(int i = 0; i < G.n; i++)
253     {
254         if(!visited[i])
255         {
256             cout<<G.vertex[i]<<" ";
257             bfn[i] = count++;
258             visited[i] = true;
259             q.push(i);
260         }
261         while(!q.empty())
262         {
263             int j = 0, f = q.front();
264             q.pop();
265             while((j >= 0) && (j < G.n))
266             {
267                 if((G.edge[f][j] != 0) && (!visited[j]))
268                 {
269                     search_tree[j][f] = G.edge[f][j];
270                     search_tree[f][j] = G.edge[f][j];
271                     cout<<G.vertex[j]<<" ";
272                     bfn[j] = count++;
273                     visited[j] = true;
274                     q.push(j);
275                 }
276                 j++;
277             }
278         }
279     }

```

由程序可得，则对邻接表进行深度（广度）优先搜索的时间复杂度为 $O(n^2)$ ，因为需要对矩阵中每个位置进行遍历，而空间复杂度为 $O(n)$ ，这是因为最坏的情况下所有的顶点进入到栈或队列当中。

4. 对于无向图，采用“邻接表”存储结构，设计和实现计算每个顶点度的算法，并分析其时间复杂度。



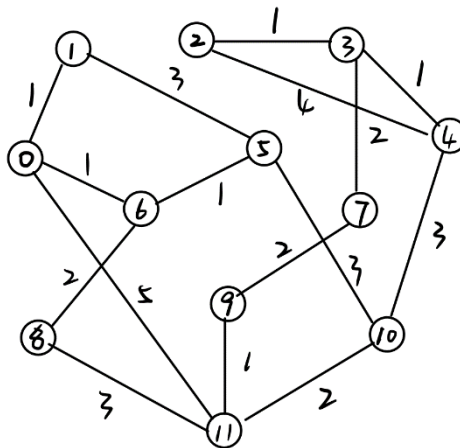
只需对每个顶点之后连接的顶点数进行遍历即可，具体程序如下：

```
314 void degree(struct graph G, int D[])
315 {
316     for(int i = 0; i < G.n; i++)
317     {
318         D[i] = 0;
319         EdgeNode *p;
320         p = G.vertexlist[i].phead;
321         while(p != NULL)
322         {
323             D[i]++;
324             p = p->next;
325         }
326     }
327     for(int i = 0; i < G.n; i++)
328         cout<<"第"<<i<<"个顶点的度为: "<<D[i]<<endl;
329 }
```

由程序可得，设 $n$ 为顶点数， $e$ 为边数，则对邻接表计算度数的时间复杂度为 $O(n + e)$ ，空间复杂度为 $O(1)$

5. 以适当的方式输入图的顶点和边，并显示相应的结果。要求顶点不少于 10 个，边不少于 13 个。

如图所示，图中共有 12 个顶点，16 条边，包含权值等信息。为方便观察，以下分为两个程序，一个展示以邻接表形式储存图以及遍历，另外一种以邻接矩阵形式储存图以及遍历。



## A. 以邻接表方式构建无向图：

### 1. 输入

```
C:\Windows\system32\cmd.exe
以邻接表方式构建无向图
=====
图的顶点数为：12
图的边数为：16
此无向图的顶点为：0 1 2 3 4 5 6 7 8 9 10 11
第1条边的起止结点以及权值为：0 1 1
第2条边的起止结点以及权值为：0 6 1
第3条边的起止结点以及权值为：0 11 5
第4条边的起止结点以及权值为：1 5 3
第5条边的起止结点以及权值为：2 3 1
第6条边的起止结点以及权值为：2 4 4
第7条边的起止结点以及权值为：3 4 1
第8条边的起止结点以及权值为：3 7 2
第9条边的起止结点以及权值为：4 10 3
第10条边的起止结点以及权值为：5 6 1
第11条边的起止结点以及权值为：5 10 3
第12条边的起止结点以及权值为：6 8 2
第13条边的起止结点以及权值为：7 9 2
第14条边的起止结点以及权值为：8 11 3
第15条边的起止结点以及权值为：9 11 1
第16条边的起止结点以及权值为：10 11 2
```

### 2. 显示以及转换

```
C:\Windows\system32\cmd.exe
第14条边的起止结点以及权值为：8 11 3
第15条边的起止结点以及权值为：9 11 1
第16条边的起止结点以及权值为：10 11 2
此无向图的邻接表为：
0->11->6->1
1->5->0
2->4->3
3->7->4->2
4->10->3->2
5->10->6->1
6->8->5->0
7->9->3
8->11->6
9->11->7
10->11->5->4
11->10->9->8->0
邻接表转换为邻接矩阵.....
此无向图的邻接矩阵为：
0 1 0 0 0 0 1 0 0 0 0 5
1 0 0 0 0 3 0 0 0 0 0 0
0 0 0 1 4 0 0 0 0 0 0 0
0 0 1 0 1 0 0 2 0 0 0 0
0 0 4 1 0 0 0 0 0 0 3 0
0 3 0 0 0 0 1 0 0 0 3 0
1 0 0 0 0 1 0 0 2 0 0 0
0 0 0 2 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 0 0 0 3
0 0 0 0 0 0 0 2 0 0 0 1
0 0 0 0 3 3 0 0 0 0 0 2
5 0 0 0 0 0 0 0 3 1 2 0
```

### 3. 搜索以及顶点度数

```
C:\Windows\system32\cmd.exe
深度优先序列为（递归）： 0 11 10 5 6 8 1 4 3 7 9 2
深度优先编号为（递归）：
0 1 2 3 4 5 6 7 8 9 10 11
1 7 12 9 8 4 5 10 6 11 3 2
深度优先生成树的邻接表为：
0->11
1->5
2->3
3->7->4->2
4->10->3
5->10->6->1
6->8->5
7->9->3
8->6
9->7
10->11->5->4
11->10->0
深度优先序列为（非递归）： 0 11 10 5 6 8 1 4 3 7 9 2
深度优先编号为（非递归）：
0 1 2 3 4 5 6 7 8 9 10 11
1 7 12 9 8 4 5 10 6 11 3 2
深度优先生成树的邻接表为：
0->11
1->5
2->3
3->7->4->2
4->10->3
5->10->6->1
6->8->5
7->9->3
```

```

C:\Windows\system32\cmd.exe
8->6
9->7
10->11->5->4
11->10->0
广度优先序列为: 0 11 6 1 10 9 8 5 4 7 3 2
广度优先编号为:
0 1 2 3 4 5 6 7 8 9 10 11
1 4 12 11 9 8 3 10 7 6 5 2
广度优先生成树的邻接表为:
0->11->6->1
1->0
2->4
3->4
4->10->3->2
5->6
6->5->0
7->9
8->11
9->11->7
10->11->4
11->10->9->8->0
第0个顶点的度为: 3
第1个顶点的度为: 2
第2个顶点的度为: 2
第3个顶点的度为: 3
第4个顶点的度为: 3
第5个顶点的度为: 3
第6个顶点的度为: 3
第7个顶点的度为: 2
第8个顶点的度为: 2
第9个顶点的度为: 2
第10个顶点的度为: 3
第11个顶点的度为: 4
请按任意键继续. . .

```

#### 4. 深度优先树以及广度优先树的图示

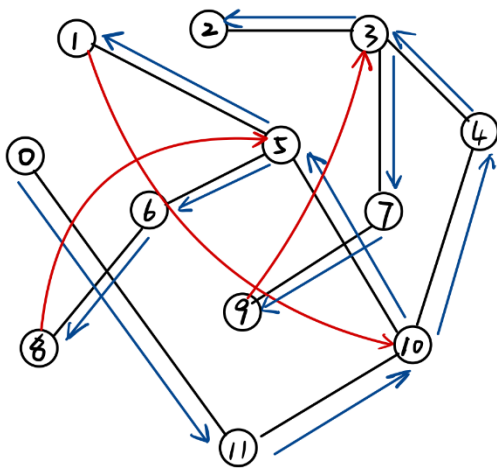


Figure 1 深度优先树

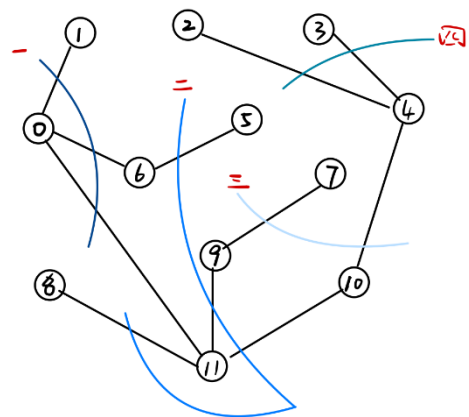


Figure 2 广度优先树

## B. 以邻接矩阵方式构建无向图：

### 1. 输入

```
C:\Windows\system32\cmd.exe
以邻接矩阵方式构建无向图
=====
图的顶点数为：12
图的边数为：16
此无向图的顶点为：0 1 2 3 4 5 6 7 8 9 10 11
第1条边的起止结点以及权值为：0 1 1
第2条边的起止结点以及权值为：0 6 1
第3条边的起止结点以及权值为：0 11 5
第4条边的起止结点以及权值为：1 5 3
第5条边的起止结点以及权值为：2 3 1
第6条边的起止结点以及权值为：2 4 4
第7条边的起止结点以及权值为：3 4 1
第8条边的起止结点以及权值为：3 7 2
第9条边的起止结点以及权值为：4 10 3
第10条边的起止结点以及权值为：5 6 1
第11条边的起止结点以及权值为：5 10 3
第12条边的起止结点以及权值为：6 8 2
第13条边的起止结点以及权值为：7 9 2
第14条边的起止结点以及权值为：8 11 3
第15条边的起止结点以及权值为：9 11 1
第16条边的起止结点以及权值为：10 11 2
```

### 2. 显示以及转换

```
C:\Windows\system32\cmd.exe
此无向图的邻接矩阵为：
0 1 0 0 0 0 1 0 0 0 0 5
1 0 0 0 0 0 3 0 0 0 0 0
0 0 0 1 4 0 0 0 0 0 0 0
0 0 1 0 1 0 0 2 0 0 0 0
0 0 4 1 0 0 0 0 0 0 3 0
0 3 0 0 0 0 1 0 0 0 3 0
1 0 0 0 0 1 0 0 2 0 0 0
0 0 0 2 0 0 0 0 0 0 2 0
0 0 0 0 0 0 2 0 0 0 0 3
0 0 0 0 0 0 0 2 0 0 0 1
0 0 0 0 3 3 0 0 0 0 0 2
5 0 0 0 0 0 0 0 0 3 1 2 0
邻接矩阵转换为邻接表.....
此无向图的邻接表为：
0->11->6->1
1->5->0
2->4->3
3->7->4->2
4->10->3->2
5->10->6->1
6->8->5->0
7->9->3
8->11->6
9->11->7
10->11->5->4
11->10->9->8->0
```

### 3. 搜索

```
C:\Windows\system32\cmd.exe
深度优先序列为（非递归）： 0 1 5 6 8 11 9 7 3 2 4 10
深度优先编号为（非递归）：
0 1 2 3 4 5 6 7 8 9 10 11
1 2 10 9 11 3 4 8 5 7 12 6
深度优先生成树矩阵为：
0 1 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 3 0 0 0 0 0 0
0 0 0 1 4 0 0 0 0 0 0 0
0 0 1 0 0 0 0 2 0 0 0 0
0 0 4 0 0 0 0 0 0 0 3 0
0 3 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 2 0 0 0
0 0 0 2 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 0 0 3
0 0 0 0 0 0 0 2 0 0 0 1
0 0 0 0 3 0 0 0 0 0 0 0
0 0 0 0 0 0 0 3 1 0 0 0
深度优先序列为（递归）： 0 1 5 6 8 11 9 7 3 2 4 10
深度优先编号为（递归）：
0 1 2 3 4 5 6 7 8 9 10 11
1 2 10 9 11 3 4 8 5 7 12 6
深度优先生成树矩阵为：
0 1 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 3 0 0 0 0 0 0
0 0 0 1 4 0 0 0 0 0 0 0
0 0 1 0 0 0 0 2 0 0 0 0
0 0 4 0 0 0 0 0 0 0 3 0
0 3 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 2 0 0 0
0 0 0 2 0 0 0 0 0 2 0 0
```

```

C:\Windows\system32\cmd.exe
0 0 0 0 0 0 2 0 0 0 0 3
0 0 0 0 0 0 0 2 0 0 0 1
0 0 0 0 3 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 3 1 0 0
广度优先序列为: 0 1 6 11 5 8 9 10 7 4 3 2
广度优先编号为:
0 1 2 3 4 5 6 7 8 9 10 11
1 2 12 11 10 5 3 9 6 7 8 4
广度优先生成树矩阵为:
0 1 0 0 0 0 1 0 0 0 0 5
1 0 0 0 0 3 0 0 0 0 0 0
0 0 0 0 4 0 0 0 0 0 0 0
0 0 0 0 0 0 2 0 0 0 0 0
0 0 4 0 0 0 0 0 0 0 3 0
0 3 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 2 0 0 0
0 0 0 2 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 0 0 0 0
0 0 0 0 0 0 0 2 0 0 0 1
0 0 0 0 3 0 0 0 0 0 0 2
5 0 0 0 0 0 0 0 0 1 2 0

请按任意键继续. . .

```

4. 深度优先树以及广度优先树的图示

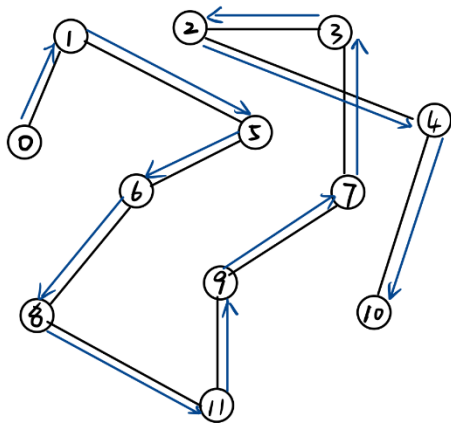


Figure 3 深度优先树

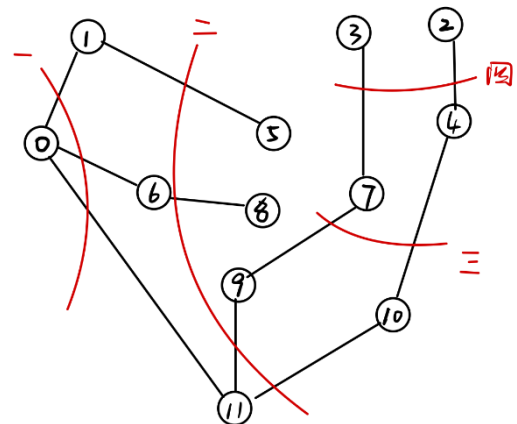


Figure 4 广度优先树