

北京交通大学

《数据结构（A）》第7章

“图论及其算法”基本作业与设计作业

专 业： 计算机科学与技术

班 级：

学生姓名：

学 号：

北京交通大学计算机与信息技术学院

2021 年 12 月 08 日

《数据结构 (A)》第 7 章基本与设计作业^①

提醒同学：本章基本作业与设计型作业合并在一起，即没有另外专门的设计作业题目。

1 作业题目

7.1 试基于图的深度优先搜索策略写一算法，判别以邻接表方式存储的有向图中是否存在由顶点 v_i 到顶点 v_j 的路径($i < j$)。注意：算法中涉及的图的基本操作必须在此存储结构上实现。

7.2 同 7.1 题要求，试基于广度优先搜索策略写一算法。

7.3 采用邻接表存储结构，编写一个判别无向图中任意给定的两个顶点之间是否存在一条长度为 k 的简单路径的算法。

7.4 输入任意的一个网，用普里姆(Prim)算法构造最小生成树

^① 这是《数据结构 (A)》第 7 章的基本与设计作业，第 12 周周四发布（2021 年 12 月 2 日星期四），学生提交的截止日期是 2021 年 12 月 12 日。

2 作业题目解答

7.1 试基于图的深度优先搜索策略写一算法，判别以邻接表方式存储的有向图中是否存在由顶点 v_i 到顶点 v_j 的路径($i < j$)。注意：算法中涉及的图的基本操作必须在此存储结构上实现。

答：

思路：

正常建立 `arcnode` & `vnode` 结构，以广度优先遍历找顶点。找到返回 1，找不到返回 0。找不到则继续下一个邻接点。直到邻接点为空，即遍历完所有路径时返回 0。

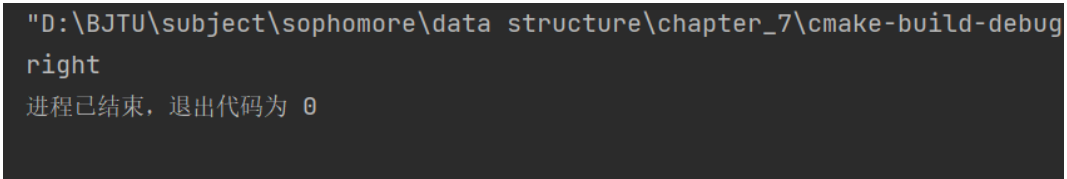
代码：

```
01:#include <stdlib.h>
02:#include <stdio.h>
03:typedef struct ArcNode{
04:    int adjvex;
05:    ArcNode * nextarc;
06:};
07:
08:typedef struct VNode{
09:    int data;
10:    ArcNode *firstarc;
11:};
12:
13:typedef struct ALGraph{
14:};
15:
16:
17:int DFS_find(int i, int j, VNode g[], bool visted[] ){
18:    visted[i] = true;
19:    ArcNode * p;
20:    if(i==j)
21:        return 1;
22:    else{
23:        p = g[i].firstarc;
24:        while (p){
```

```
25:         if(visted[p->adjvex]) p = p->nextarc;
26:         else if (!(DFS_find(p->adjvex, j, g, visted))) {
27:             p = p->nextarc;
28:         }
29:         else
30:             return 1;
31:     }
32:     return 0;
33: }
34:}
35:int main() {
36:     VNode g[6];
37:     for(int i= 0; i<6; ++i){
38:         g[i].data=i;
39:         g[i].firstarc = (ArcNode *)malloc(sizeof (ArcNode));
40:     }
41:     g[0].firstarc->adjvex=1;
42:     g[0].firstarc->nextarc = (ArcNode *)malloc(sizeof (ArcNode));
43:     ArcNode *p = g[0].firstarc->nextarc;
44:     p->adjvex =2;
45:     p->nextarc = NULL;
46:     g[1].firstarc->adjvex =0;
47:     g[1].firstarc->nextarc = NULL;
48:     g[2].firstarc->adjvex = 3;
49:     g[2].firstarc->nextarc = (ArcNode*) malloc(sizeof (ArcNode));
50:     p = g[2].firstarc->nextarc;
51:     p->adjvex =4;
52:     p->nextarc = (ArcNode*) malloc(sizeof (ArcNode));
53:     p = p->nextarc;
54:     p->adjvex = 0;
55:     p->nextarc = NULL;
56:     p= g[3].firstarc;
57:     p->adjvex = 2;
58:     p->nextarc = (ArcNode*) malloc(sizeof (ArcNode));
59:     p = p ->nextarc;
60:     p->adjvex = 5;
61:     p->nextarc = NULL;
62:     p = g[4].firstarc;
63:     p->adjvex = 2;
64:     p->nextarc = NULL;
65:     p = g[5].firstarc;
66:     p->adjvex = 3;
67:     p->nextarc = NULL;
```

```
68:
69:   bool visited[6] ;
70:   for(int i =0 ; i<6; ++i)
71:       visited[i] = false;
72:   if(DFS_find(0, 5 , g, visited))
73:       printf("right") ;
74:   else printf("0");
75:}
76:
```

调试正常:



```
"D:\BJTU\subject\sophomore\data structure\chapter_7\cmake-build-debug
right
进程已结束, 退出代码为 0
```

图 7.1

7.2 同 7.1 题要求, 试基于广度优先搜索策略写一算法。

思路:

建立一个简单的队列, 出队即把它所以未访问的邻接点入队。入队过程中, 判断是否为目的地, 是返回 1, 队列空了后即, 返回 0。

代码:

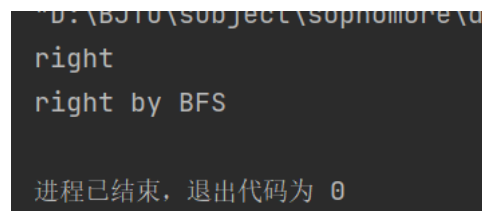
```
01:int BFS_find(int i, int j, VNode g[], bool visited[]){
02:   int queue_index[100];
03:   int front, rear;
04:   front = rear =0;
05:   visited[i] = true;
06:   VNode tem = g[i];
07:   queue_index[rear++] = tem.data;
08:
09:   while(front!=(rear+1)){
10:       tem = g[queue_index[front++]];
11:       ArcNode * p = tem.firstarc;
12://       while(p&& !visited[p->adjvex]) { the wrong method, thus divided
13://into two steps
14:       while(p){
15:           if(visited[p->adjvex]) {
```

```
16:         p=p->nextarc; // maybe the first arcnode is visited but the
17://next arcnode is not, then progress ends.
18:         continue;
19:     }
20:     if(p->adjvex== j) return 1;
21:     queue_index[rear++] = p->adjvex;
22:     visited[p->adjvex] = true;
23:     p = p->nextarc;
24: }
25: }
26: return 0;
27:}

28:int main() {
29:    VNode g[6];
30:    for(int i= 0; i<6; ++i){
31:        g[i].data=i;
32:        g[i].firstarc = (ArcNode *)malloc(sizeof (ArcNode));
33:    }
34:    g[0].firstarc->adjvex=1;
35:    g[0].firstarc->nextarc = (ArcNode *)malloc(sizeof (ArcNode));
36:    ArcNode *p = g[0].firstarc->nextarc;
37:    p->adjvex =2;
38:    p->nextarc = NULL;
39:    g[1].firstarc->adjvex =0;
40:    g[1].firstarc->nextarc = NULL;
41:    g[2].firstarc->adjvex = 3;
42:    g[2].firstarc->nextarc = (ArcNode*) malloc(sizeof (ArcNode));
43:    p = g[2].firstarc->nextarc;
44:    p->adjvex =4;
45:    p->nextarc = (ArcNode*) malloc(sizeof (ArcNode));
46:    p = p->nextarc;
47:    p->adjvex = 0;
48:    p->nextarc = NULL;
49:    p= g[3].firstarc;
50:    p->adjvex = 2;
51:    p->nextarc = (ArcNode*) malloc(sizeof (ArcNode));
52:    p = p->nextarc;
53:    p->adjvex = 5;
54:    p->nextarc = NULL;
55:    p = g[4].firstarc;
56:    p->adjvex = 2;
57:    p->nextarc = NULL;
58:    p = g[5].firstarc;
```

```
59:   p->adjvex = 3;
60:   p->nextarc = NULL;
61:
62:   bool visited[6] ;
63:   for(int i =0 ; i<6; ++i)
64:       visited[i] = false;
65:   if(DFS_find(0, 5 , g, visited))
66:       printf("right\n") ;
67:   else printf("0");
68:
69:
70:   for(int i= 0; i<6; ++i) visited[i] = false;
71:   if(BFS_find(0, 5, g, visited))
72:       printf("right by BFS\n");
73:   else printf("no way");
74:}
```

调试正常：



```
right
right by BFS
进程已结束，退出代码为 0
```

图 7.2

7.3 采用邻接表存储结构，编写一个判别无向图中任意给定的两个顶点之间是否存在一条长度为 k 的简单路径的算法。

思路：

直接在前两题基础上修改，加入一个路径计算数 **count**，最后 **count** 与给定的 **k** 比较即可（测试中 **k** 直接给出，实际中也可控制台输入）。

如果考虑有环情况，则 **k** 可以无限大，即一直转圈后出圈，这样其实没有意义，故 **k** 应该尽量小。即求在给定步数情况下能否到达目的地。

代码：

```
01:int BFS_find(int i, int j, VNode g[], bool visited[], int &count){
02:   int queue_index[100];
```



```
03:   int front, rear;
04:   front = rear = count = 0;
05:   visited[i] = true;
06:   VNode tem = g[i];
07:   queue_index[rear++] = tem.data;
08:
09:   while(front!=(rear+1)){
10:       tem = g[queue_index[front++]];
11:       ArcNode * p = tem.firstarc;
12://       while(p&& !visited[p->adjvex]) { the wrong method, thus divided
into two steps,
13:       while(p){
14:           if(visited[p->adjvex]) {
15:               p=p->nextarc; // maybe the first arcnode is visited but the
next arcnode is not, then progress ends.
16:               continue;
17:           }
18:           if(p->adjvex== j) return 1;
19:           queue_index[rear++] = p->adjvex;
20:           visited[p->adjvex] = true;
21:           p = p->nextarc;
22:       }
23:       count ++;
24:   }
25:   return 0;
26:}
27:   for(int i= 0; i<6; ++i) visited[i] = false;
28:   int count;
29:   int k = 3;
30:   if(BFS_find(0, 5 , g, visited, count)){
31:       printf("there is one way and the count is %d\n", count);
32:       if(count == k) printf("satisfy the num k\n");
33:   }
```

调试正常:

```
there is one way and the count is 3  
satisfy the num k
```

```
. 进程已结束, 退出代码为 0
```

图 7.3

7.4 输入任意的一个网，用普里姆(Prim)算法构造最小生成树

思路：

正常建立 **prim** 算法建立最小生成树。为测试方便，**MGraph** 结构体是针对测试的 **9*9** 矩阵设立的，其中无穷大以 **65535** 代替。逐次打印加入的最小权边。

代码：

```
01:typedef struct MGraph{  
02:    int numVertexes;  
03:    int (*arc)[9];// NIUDE ]  
04:};  
05:  
06:void minispantree_prim(MGraph g){  
07:    int adjvex[100];  
08:    int lowcost[100];  
09:    lowcost[0] =0;  
10:    adjvex[0] = 0;  
11:    for(int i=1; i<g.numVertexes; ++i){  
12:        lowcost[i] = g.arc[0][i];  
13:        adjvex[i] = 0;  
14:    }  
15:    for(int i =1; i<g.numVertexes; ++i){  
16:        int min = INFINITY;  
17:        int j=1, k=0;  
18:        while(j<g.numVertexes){  
19:            if(lowcost[j]!= 0 && lowcost[j] < min)  
20:                {  
21:                    min = lowcost[j];  
22:                    k = j;
```

```

23:         }
24:         ++j;
25:     }
26:     printf("(%d, %d)", adjvex[k], k);
27:     lowcost[k] = 0;
28:     for(j = 1; j<g.numVertexes; ++j){
29:         if(lowcost[j]!=0 && g.arc[k][j] < lowcost[j])
30:         {
31:             lowcost[j] = g.arc[k][j];
32:             adjvex[j] = k;
33:         }
34:     }
35: }
36:}
37:int main () {
38:     MGraph g;
39://     g.arc = (int **) malloc(sizeof(int *) *9);
40://     for(int i=0; i<9; ++i){
41://         g.arc[i] = (int *) malloc(sizeof (int ) *9);
42://     }
43:     int graph[9][9] = {
44:         {0,    10,    65535, 65535, 65535, 11,    65535, 65535, 65535},
45:         {10,    0,    18,    65535, 65535, 65535, 16,    65535, 12},
46:         {65535, 65535, 0,    22,    65535, 65535, 65535, 65535, 8},
47:         {65535, 65535, 22,    0,    20,    65535, 65535, 16,    21},
48:         {65535, 65535, 65535, 20,    0,    26,    65535, 7,    65535},
49:         {11,    65535, 65535, 65535, 26,    0,    17,    65535, 65535},
50:         {65535, 16,    65535, 65535, 65535, 17,    0,    19,    65535},
51:         {65535, 65535, 65535, 16,    7,    65535, 19,    0,    65535},
52:         {65535, 12,    8,    21,    65535, 65535, 65535, 65535, 0}
53: };
54:     g.numVertexes = 9;
55:     g.arc = graph;
56:     minispantree_prim(g);
57:}

```

调试正常:

```

(0, 1)(0, 5)(1, 8)(8, 2)(1, 6)(6, 7)(7, 4)(7, 3)
进程已结束, 退出代码为 0

```

图 7.4