



中国科学技术大学  
University of Science and Technology of China

## 数据结构实验报告 4

### 图及相关应用

姓名：\_\_\_\_\_ 高茂航 \_\_\_\_\_

学号：\_\_\_\_\_ PB22061161 \_\_\_\_\_

日期：\_\_\_\_\_ 2023 年 11 月 30 日 \_\_\_\_\_

# 实验报告 4

## 1 图的遍历

输入一个无向图，输出图的深度优先搜索遍历顺序与广度优先搜索遍历顺序。当有多个节点可以搜索时，优先去节点编号最小的那个。

### 1.1 算法描述

#### 1.1.1 数据结构

```
1     typedef int VertexType;
2     #define MaxVexNum 30
3     #define INFINITY 65535
4     typedef struct{
5         VertexType ves[MaxVexNum+1]; //顶点表
6         int arc[MaxVexNum+1][MaxVexNum+1]; //邻接矩阵
7         int VertexNum, EdgeNum; //分别是图中当前顶点数和边数
8     }MGraph;
9     int visited[MaxVexNum+1]={0}; //记录每个顶点是否被访问过，0为未访问，1为已访问
10    typedef struct QueueList{
11        int vertex; //存顶点序号
12        struct QueueList* next;
13    }Queue;
14
```

Listing 1: 数据结构

#### 1.1.2 程序结构

```
1     void CreateMGraph(MGraph &); //创建无向图的邻接矩阵结构
2     void DFS(MGraph &, int); //深度优先算法(邻接矩阵)
3     void DFSTraverse(MGraph &, int); //深度优先遍历操作(邻接矩阵)
4     void BFSTraverse(MGraph &, int); //广度优先遍历操作(邻接矩阵)
5     void EnQueue(Queue** Q, int n); //把序号为n的顶点入队
6     int DeQueue(Queue** Q); //将队首顶点出队，返回其序号
7     int QueueEmpty(Queue *Q); //判断队列是否为空
8     int main(){
9         MGraph G;
10        int a, i;
11        CreateMGraph(G);
12        cin >> a; //输入遍历起始的起始顶点
13        DFSTraverse(G, a);
14        for(i=0; i<=G.VertexNum; i++)
15            visited[i]=0;
16        BFSTraverse(G, a);
17    }
18
```

Listing 2: 程序结构

#### 1.1.3 主要算法

```
1     void CreateMGraph(MGraph &G){ //创建无向图的邻接矩阵结构
2         int i=0, j=0, k=0;
3         cin >> G.VertexNum >> G.EdgeNum;
4         for(i=1; i<=G.VertexNum; i++)
```

## 实验报告 4

```
5     G.ves[i]=i;//顶点序号从1开始
6     for(i=1;i<=G.VertexNum;i++){
7         for(j=1;j<=G.VertexNum;j++){
8             if(i==j)
9                 G.arc[i][j]=0;
10            else
11                G.arc[i][j]=INFINITY;
12        }
13    }
14    for(k=1;k<=G.EdgeNum;k++){
15        cin>>i>>j;
16        G.arc[j][i]=G.arc[i][j]=1;
17    }
18 }
```

Listing 3: 创建无向图的邻接矩阵结构

```
1     void DFS(MGraph &G,int a){//深度优先算法(邻接矩阵)
2         int i=0;
3         visited[a]=1;
4         cout<<G.ves[a];
5         for(i=1;i<=G.VertexNum;i++){
6             if(G.arc[a][i]==1&&!visited[i])
7                 DFS(G,i);
8         }
9     void DFSTraverse(MGraph &G,int a){//深度优先遍历操作(邻接矩阵)
10        int i=0;
11        for(i=1;i<=G.VertexNum;i++){
12            visited[i]=0;
13            DFS(G,a);
14        }
15 }
```

Listing 4: 深度优先搜索算法

```
1     void BFSTraverse(MGraph &G,int a){//广度优先遍历操作(邻接矩阵)
2         int i=0,j=0,ver=0;
3         for(i=1;i<=G.VertexNum;i++){
4             visited[i]=0;
5             Queue* Q=NULL;
6             visited[a]=1;
7             cout<<G.ves[a];
8             EnQueue(&Q,a);
9             while(!QueueEmpty(Q)){
10                i=DeQueue(&Q);
11                for(j=1;j<=G.VertexNum;j++){
12                    if(G.arc[i][j]==1&&!visited[j]){
13                        visited[j]=1;
14                        cout<<G.ves[j];
15                        EnQueue(&Q,j);
16                    }
17                }
18            }
19        }
20 }
```

Listing 5: 广度优先搜索算法

# 实验报告 4

## 1.2 测试结果

```
请依次输入顶点数(<30)和边数(<300)
5 6
每次输入两个顶点序号以表示其间有边相连
1 5
1 4
3 1
2 1
2 5
5 3
请输入遍历起始的起始顶点
1
图的深度优先搜索遍历结果是:12534
图的广度优先搜索遍历结果是:12345
```

## 2 求最小生成树

输入一个无向图,用 Prim 和 Kruskal 算法计算最小生成树并输出。

### 2.1 算法描述

#### 2.1.1 数据结构

```
1 typedef int VertexType;
2 typedef struct{
3     VertexType ves[MaxVexNum+1]; //顶点表
4     int arc[MaxVexNum+1][MaxVexNum+1]; //邻接矩阵
5     int VertexNum, EdgeNum; //分别是图中当前顶点数和边数
6 }MGraph;
```

Listing 6: 数据结构 (Prim)

```
1 typedef int VertexType;
2 typedef struct{
3     VertexType ves[MaxVexNum+1]; //顶点表
4     int arc[MaxVexNum+1][MaxVexNum+1]; //邻接矩阵
5     int VertexNum, EdgeNum; //分别是图中当前顶点数和边数
6 }MGraph;
7 typedef struct{
8     int begin;
9     int end;
10    int weight;
11 }Edge; //边集数组结构
12 Edge edges[MaxEdgeNum+1]; //边集数组
```

Listing 7: 数据结构 (Kruskal)

#### 2.1.2 程序结构

```
1 void CreateMGraph(MGraph &); //创建无向图的邻接矩阵结构
2 void Prim(MGraph); //Prim算法求最小生成树的各边的长度之和
3 int main(){
4     MGraph G;
5     CreateMGraph(G);
6     Prim(G);
7 }
```

Listing 8: 程序结构 (Prim)

# 实验报告 4

```
1 void CreateMGraph(MGraph &); //创建无向图的邻接矩阵结构
2 void Kruskal(MGraph); //Kruskal算法求最小生成树的各边的长度之和
3 void SortEdge(Edge *); //将边集数组按权值从小到大排序
4 int Find(int* parent, int i); //查找连线顶点的尾部下标
5 int main(){
6     MGraph G;
7     CreateMGraph(G);
8     Kruskal(G);
9 }
```

Listing 9: 程序结构 (Kruskal)

## 2.1.3 主要算法

```
1 void Prim(MGraph G){ //Prim算法求最小生成树的各边的长度之和
2     int min=0, length=0, i=0, j=0, k=0;
3     int adjvex[MaxVexNum+1]={0}; //记录已加入生成树的顶点序号, adjvex[i]的值是生成树上i顶点上
    一个连接的顶点序号
4     int lowcost[MaxVexNum+1]={0}; //记录已遍历顶点的邻边中最小的权重
5     lowcost[1]=0;
6     adjvex[1]=1;
7     for(i=2; i<=G.VertexNum; i++){
8         lowcost[i]=G.arc[1][i];
9         adjvex[i]=1;
10    }
11    for(i=1; i<G.VertexNum; i++){
12        min=INFINITY;
13        j=2;
14        k=1;
15        while(j<=G.VertexNum){
16            if(lowcost[j]&&lowcost[j]<min){
17                min=lowcost[j];
18                k=j;
19            }
20            j++;
21        }
22        length+=G.arc[k][adjvex[k]];
23        lowcost[k]=0; //此顶点已完成任务
24        for(j=2; j<=G.VertexNum; j++){
25            if(lowcost[j]&&G.arc[k][j]<lowcost[j]){
26                lowcost[j]=G.arc[k][j];
27                adjvex[j]=k;
28            }
29        }
30    }
31    cout<<"最小生成树的各边的长度之和为"<<length;
32 }
33
```

Listing 10: Prim 算法求最小生成树的各边的长度之和

```
1 void CreateMGraph(MGraph &G){ //创建无向图的邻接矩阵结构和边集数组
2     int i=0, j=0, k=0, w;
3     cout<<"请依次输入顶点数(<30)和边数(<300)"<<endl;
4     cin>>G.VertexNum>>G.EdgeNum;
5     for(i=1; i<=G.VertexNum; i++)
6         G.ves[i]=i; //顶点序号从1开始
7     for(i=1; i<=G.VertexNum; i++)
```

## 实验报告 4

```
8     for(j=1;j<=G.VertexNum;j++){
9         if(i==j)
10             G.arc[i][j]=0;
11         else
12             G.arc[i][j]=INFINITY;
13     }
14     for(k=1;k<=G.EdgeNum;k++){//输入边的信息
15         cin>>i>>j>>w;
16         G.arc[j][i]=G.arc[i][j]=w;
17         edges[k].begin=i;
18         edges[k].end=j;
19         edges[k].weight=w;
20     }
21     for(k=G.EdgeNum+1;k<=MaxEdgeNum;k++)
22         edges[k].weight=0;
23     for(k=1;k<=G.EdgeNum;k++)
24         cout<<edges[k].begin<<" "<<edges[k].end<<" "<<edges[k].weight<<endl;
25     cout<<endl;
26     SortEdge(edges);//将边集数组按权值从小到大排序
27     for(k=1;k<=G.EdgeNum;k++)
28         cout<<edges[k].begin<<" "<<edges[k].end<<" "<<edges[k].weight<<endl;
29 }
30 void SortEdge(Edge *edges){//将边集数组按权值从小到大排序
31     int i,j,temp;
32     for(i=1;i<=MaxEdgeNum;i++){
33         for(j=1;j<=MaxEdgeNum;j++){
34             if(edges[j].weight>edges[j+1].weight&&edges[j+1].weight){
35                 temp=edges[j].begin;
36                 edges[j].begin=edges[j+1].begin;
37                 edges[j+1].begin=temp;
38                 temp=edges[j].end;
39                 edges[j].end=edges[j+1].end;
40                 edges[j+1].end=temp;
41                 temp=edges[j].weight;
42                 edges[j].weight=edges[j+1].weight;
43                 edges[j+1].weight=temp;
44             }
45         }
46     }
47 }
48
```

Listing 11: 创建并排序编辑数组 (Kruskal)

```
1     void Kruskal(MGraph G){//Kruskal算法求最小生成树的各边的长度之和
2         int i,n,m,length=0;
3         int parent[MaxVexNum+1];//定义一数组用来判断边与边是否形成环路,parent[m]=n表示m与n在同一
           集合中,而不是表示m和n之间有边
4         for(i=0;i<=G.VertexNum;i++)
5             parent[i]=0;
6         for(i=1;i<=G.EdgeNum;i++){
7             n=Find(parent,edges[i].begin);
8             m=Find(parent,edges[i].end);
9             if(n!=m){//如果n与m不等,说明此边没有与现有生成树形成环路
10                 parent[n]=m;//将此边的结尾顶点放入下标为起点的parent中,表示此顶点已经在生成树集合中
11                 length+=edges[i].weight;
12             }
13         }
14         cout<<"最小生成树的各边的长度之和为"<<length;
15     }
```

# 实验报告 4

16

Listing 12: Kruskal 算法求最小生成树的各边的长度之和

## 2.2 测试结果

```
请依次输入顶点数(<30)和边数(<300)
4 5
每次输入两个顶点序号以表示其间有边相连,并输入该边的权值
1 2 2
1 3 2
1 4 3
2 3 4
3 4 3
最小生成树的各边的长度之和为7
```

## 3 求最短路径

输入一个无向铁路交通图、始发站和终点站,用 Dijkstra 算法计算从始发站到终点站的最短路径。

### 3.1 算法描述

#### 3.1.1 数据结构

```
1 typedef struct{
2     VertexType ves[MaxVexNum+1]; //顶点表
3     int arc[MaxVexNum+1][MaxVexNum+1]; //邻接矩阵
4     int VertexNum, EdgeNum; //分别是图中当前顶点数和边数
5 }MGraph;
6
```

Listing 13: 数据结构

#### 3.1.2 程序结构

```
1 int Patharc[MaxVexNum]; //Patharc[v]的值为最短路径下前驱顶点的坐标
2 int ShortestPath[MaxEdgeNum]; //储存起点到各点最短路径的权值和
3 void CreateMGraph(MGraph &); //创建无向图的邻接矩阵结构
4 void Dijkstra(MGraph, int start, int end); //Dijkstra算法求最短路径
5 int main(){
6     int start, end;
7     MGraph G;
8     CreateMGraph(G);
9     cin >> start >> end;
10    Dijkstra(G, start, end);
11 }
12
```

Listing 14: 程序结构

#### 3.1.3 主要算法

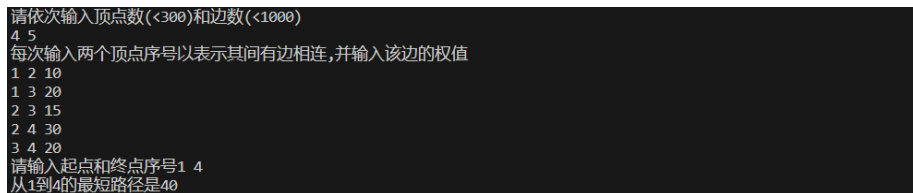
```
1 void Dijkstra(MGraph G, int start, int end){ //Dijkstra算法求从start到end最短路径
2     int i, j, k, min;
3     int Final[MaxVexNum]; //Final[v]为1表示已经求得从start到v的最短路径
4     for(i=1; i<=G.VertexNum; i++){ //初始化
```

## 实验报告 4

```
5     Final[i]=0;
6     ShortestPath[i]=G.arc[start][i];
7     Patharc[i]=-1;
8 }
9 ShortestPath[start]=0;//start到start的最短路径为0
10 Final[start]=1;//start到start的最短路径已经求得
11 for(i=1;i<G.VertexNum;i++){//每次循环求得从start到某个顶点v的最短路径
12     min=INFINITY;
13     for(j=1;j<=G.VertexNum;j++){//找到当前未求得最短路径的顶点中距离start最近的顶点
14         if(!Final[j]&&ShortestPath[j]<min){
15             k=j;
16             min=ShortestPath[j];
17         }
18     }
19     Final[k]=1;
20     for(j=1;j<=G.VertexNum;j++){//修正当前最短路径及距离
21         if(!Final[j]&&min+G.arc[k][j]<ShortestPath[j]){
22             ShortestPath[j]=min+G.arc[k][j];
23             Patharc[j]=k;
24         }
25     }
26 }
27 cout<<"从 "<<start<<"到 "<<end<<"的最短路径是 "<<ShortestPath[end];
28 }
29
```

Listing 15: Dijkstra 算法求 start 到 end 最短路径

### 3.2 测试结果



```
请依次输入顶点数(<300)和边数(<1000)
4 5
每次输入两个顶点序号以表示其间有边相连,并输入该边的权值
1 2 10
1 3 20
2 3 15
2 4 30
3 4 20
请输入起点和终点序号1 4
从1到4的最短路径是40
```

## 4 调试分析

本实验主要是图的一些基本操作,问题主要出现在处理各个数组的过程中,需要理清它们之间的关系以及它们的作用,否则很容易出错。

## 5 算法时空分析

由于本实验的图都采用邻接矩阵的方式存储,所以空间复杂度都是  $O(n^2)$ , DFS 和 BFS 的时间复杂度都是  $O(n^2)$ , Prim 算法的时间复杂度是  $O(n^2)$ , Kruskal 算法的时间复杂度是  $O(n \log n)$ , Dijkstra 算法的时间复杂度是  $O(n^2)$ 。

## 6 实验体会收获

通过本实验,我加深了对图的一些基本算法的理解,对图的遍历、求最小生成树和最短路径的操作更加熟悉,为将来进一步学习算法基础做了准备。