**《数据结构》**

**实验报告**

**实验名称：图形数据结构及其应用**

|  |  |
| --- | --- |
| **学院** | **网络空间安全学院** |
| **班级** | **2020级软件工程专业软卓2班** |
| **学号** | **202041404130** |
| **姓名** | **张 瀚** |
| **指导教师** | **周 坤 晓** |
| **时间** | **2021/05/31** |

# 需求分析

通过实验达到：

⑴ 理解和掌握图的基本概念、基本逻辑结构；

⑵ 理解和掌握图的邻接矩阵存储结构、邻接链表存储结构；

⑶ 理解和掌握图的DFS、BFS遍历操作的思想及其实现；

⑷ 加深对堆栈、队列的概念及其典型操作思想的理解；

⑸ 理解和掌握图的应用-最小生成树、最短路径算法的思想及其实现；

⑹ 掌握典型图操作算法的算法分析。

# 实验题目

实验题目：图的建立、遍历及其应用

设图结点的元素类型为ElemType(可以为char或int)，通过文件读取方式，建立一个不少于10个顶点的带权无向图G，实现以下图的各种基本操作的程序：

① 用邻接矩阵作为存储结构存储图G并输出该邻接矩阵；

② 用邻接链表作为存储结构存储图G并输出该邻接链表；

③ 按DFS算法输出图G中顶点的遍历序列；

④ 按BFS算法输出图G中顶点的遍历序列；

⑤ 用Prime算法(或者Kruskal算法)从某个指定的顶点出发输出图G的最小生成树；（要求把最小生成树的各条边输出成A-B-wight,或者(A,B,weight)的形式）；

⑥ 求从有向图的某个节点出发到其余各顶点的最短路径和最短路径值；（带权有向图）；

⑦ 主函数通过函数调用实现以上各项操作。

选做：

① 编写函数求邻接矩阵存储结构的有向图G中各顶点的入度和出度；

② 用狄克斯特拉（Dijkastra）算法或者Floyd算法求每对顶点之间的最短路径（带权有向图）；

## 数据结构设计

定义图（通过邻接矩阵存储）结构体如下：

typedef struct

{

SeqList Vertices; //存放顶点的顺序表

int edge[MaxVertices][MaxVertices]; //存放边的邻接矩阵

int numOfEdges; //边的条数

} AdjMGraph;

定义图（通过邻接矩阵存储）的边结构体如下：

typedef struct

{

int V1; //第一个顶点

int V2; //第二个顶点

int weight; //权值

} RowColWeight;

定义图（通过邻接表存储）的边结构体如下：

typedef struct Node

{

int dest; //邻接边的弧头顶点序号

int cost; //权值

struct Node \*next;

} Edge;

定义邻接表结构体如下：

typedef struct

{

ElemType data; //顶点数据元素

int source; //邻接边的弧尾顶点序号

Edge \*adj; //邻接边的头指针

} AdjLHeight;

定义图（通过邻接表存储）结构体如下：

typedef struct

{

AdjLHeight a[MaxVertices]; //邻接表数组

int numOfVerts; //顶点个数

int numOfEdges; //边个数

} AdjLGraph;

## 主要操作算法设计与分析

**通过文件读取图G的顶点和边信息算法描述：**

void readFile(FILE \*fp, ElemType V[], int \*n, RowColWeight E[], int \*e)

**返回类型：**无返回值

**是否含参数：**是，含有一个FILE类型的指针，一个顶点数组，一个边数组，两个int型数据的指针（分别用于保存顶点的个数和边的条数）

**步骤：**

A.从文件中读取字符ch，若读到文件尾则算法结束；

B.判断是否读取到换行符\n，遇到\n说明为顶点信息读取完毕，跳过步骤C；

C.将读取到的字符保存至顶点数组，顶点个数+1，转到步骤A读取下一顶点信息；

D.读取边信息，包括第一个顶点、第二个顶点和边的权值，若读到文件尾则算法结束；

E.将边信息保存至边数组，边条数+1，转到步骤D读取下一边信息；

**算法时间复杂度****：**O(V+E)（其中V为图G的顶点个数，E为图G的边的条数）

**用邻接矩阵存储图G算法描述：**

void CreatAdjMGraph(AdjMGraph \*G, ElemType V[], int n, RowColWeight E[], int e)

**返回类型：**无返回值

**是否含参数：**是，含有一个图G（通过邻接矩阵存储）的指针，一个顶点数组，一个边数组，两个int型数据（分别表示顶点的个数和边的条数）

**步骤：**

A.图G的顶点顺序表、邻接矩阵、边的条数初始化；

B.通过for循环，将顶点数组中的顶点全部保存到图G的顶点顺序表中；

C.通过for循环，根据边数组中每条边的信息，将所有边的权值保存到图G的邻接矩阵的相应位置中（图G为无向图，每次循环将保存两条反向的边至邻接矩阵中），算法结束。

**算法时间复杂度：**O(V+E)（其中V为图G的顶点个数，E为图G的边的条数）

**用邻接表存储图G算法描述：**

void CreatAdjLGraph(AdjLGraph \*G, ElemType V[], int n, RowColWeight E[], int e)

**返回类型：**无返回值

**是否含参数：**是，含有一个图G（通过邻接表存储）的指针，一个顶点数组，一个边数组，两个int型数据（分别表示顶点的个数和边的条数）

**步骤：**

A.图G的边的条数、顶点个数、邻接表数组初始化；

B.通过for循环，将顶点数组中的顶点全部保存到图G的邻接表中；

C.通过for循环，为边数组中的每条边申请邻接边单链表结点空间，将所有边信息存储到结点中并插入到单链表表头（图G为无向图，每次循环将保存两条反向的边至邻接表中），算法结束。

**算法时间复杂度：**O(V+E)（其中V为图G的顶点个数，E为图G的边的条数）

**按DFS算法输出图G（或图G的某一连通分量）中顶点的遍历序列算法描述：**

void DepthFSearch(AdjMGraph G, int v, int visited[])

**返回类型：**无返回值

**是否含参数：**是，含有一个结构体，一个int型数据（表示初始顶点），一个int型数组（用于标记各节点是否已经被访问，0表示未访问，1表示已访问）

**步骤：**

A.输出初始顶点v并标记为已访问；

B.查找顶点v的第一个邻接顶点w；

C.若顶点w不存在，则算法结束；

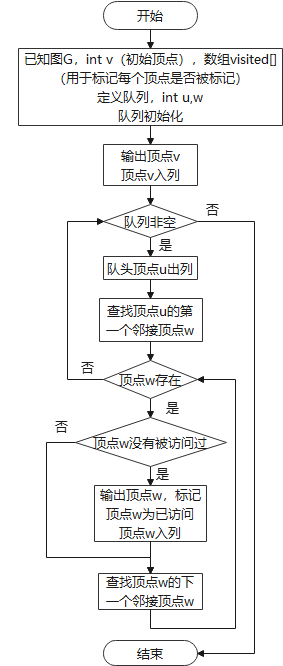
D.若顶点w未被访问，递归调用DepthFSearch函数访问顶点w；

E.查找顶点w的下一个邻接顶点w，转到步骤C；

**算法时间复杂度分析：**

查找每个顶点的邻接点所需时间为O(V)，要查找整个矩阵，故算法的时间复杂度为O(V²)（V为顶点个数）。

**按BFS算法输出图G（或图G的某一连通分量）中顶点的遍历序列算法流程图：**

****

**算法时间复杂度分析：**

查找每个顶点的邻接点所需时间为O(V)，要查找整个矩阵，故算法的时间复杂度为O(V²)（V为顶点个数）。

**按DFS算法或BFS算法输出图G中顶点的遍历序列算法描述：**

void DepthFirstSearch(AdjMGraph G)

void BroadFirstSearch(AdjMGraph G)

**返回类型：**无返回值

**是否含参数：**是，含有一个结构体

**步骤：**

A.创建动态数组visited[]，用于标记每个顶点是否被标记；

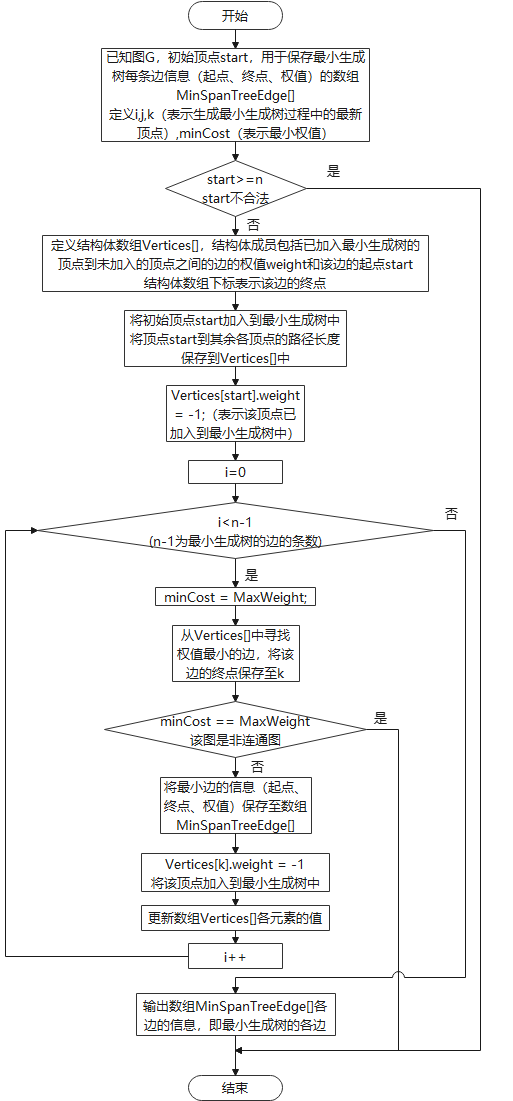
B.将数组visited[]中每个元素初始化为0；

C.通过for循环，依次判断每个顶点是否已被访问，若未被访问，调用DepthFSearch函数/ BroadFSearch函数访问该顶点；

D.销毁动态数组visited[]，算法结束；

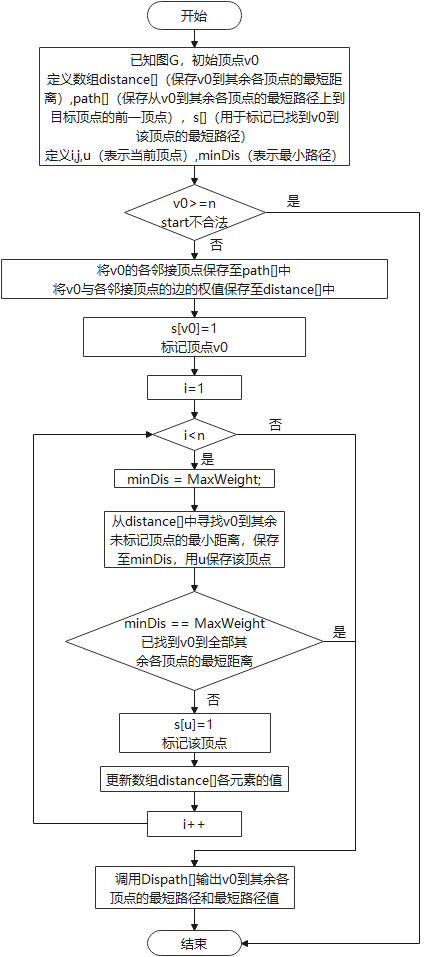
**算法时间复杂度：**O(V)（V为顶点数）

**用Prime算法从某个指定的顶点出发输出图G的最小生成树算法流程图：**

****

**算法时间复杂度：**O(n²)

**求从有向图的某个节点出发到其余各顶点的最短路径和最短路径值算法流程图：**

****

**算法时间复杂度：**O(n²)

**输出有向图的某个节点出发到其余各顶点的最短路径和最短路径值算法描述：**

void Dispath(AdjMGraph G, int distance[], int path[], int v)

**返回类型：**无返回值

**是否含参数：**是，含有一个结构体，两个int型数组，一个int型数据（表示各最短路径的起点）

**步骤：**

A.通过for循环，依次查找数组distance[]中的数据，若某元素的值为MaxWeight，表示v0到该顶点无通路，跳过步骤B；

B.输出v0到各顶点的最短路径长度，调用Ppath函数输出v0到各顶点的最短路径；

D.上述循环结束后，算法结束。

**算法时间复杂度：**O(V)（V为顶点数）

**输出有向图的某个节点出发到其余各顶点的最短路径算法描述：**

void Ppath(AdjMGraph G, int path[], int i)

**返回类型：**无返回值

**是否含参数：**是，含有一个结构体，一个int型数组，一个int型数据（表示最短路径中各顶点的上一顶点）

**步骤：**

A.如果path[i]等于-1，表示已经到达起点，算法结束；

B.递归调用Ppath函数，回溯路径的上一顶点；

C.输出下标为path[i]的顶点，算法结束。

**算法时间复杂度：**O(V)（V为顶点数）

**求邻接矩阵存储结构的有向图G中各顶点的入度和出度算法描述：**

void Degree(AdjMGraph G)

**返回类型：**无返回值

**是否含参数：**是，含有一个结构体

**步骤：**

A.创建数组Indegree[]，用于保存各个顶点的入度；

B.将数组Indegree[]中每个元素初始化为0；

C.通过for循环，依次寻找每个顶点的所有邻接顶点，每找到当前顶点的一个邻接顶点，当前顶点的出度+1，邻接顶点的入度+1；输出每个顶点的出度；

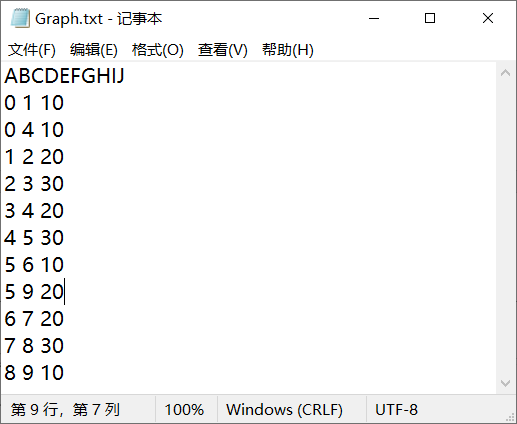
D.上述循环结束后，每个顶点的入度计算完毕，输出每个顶点的入度，算法结束。

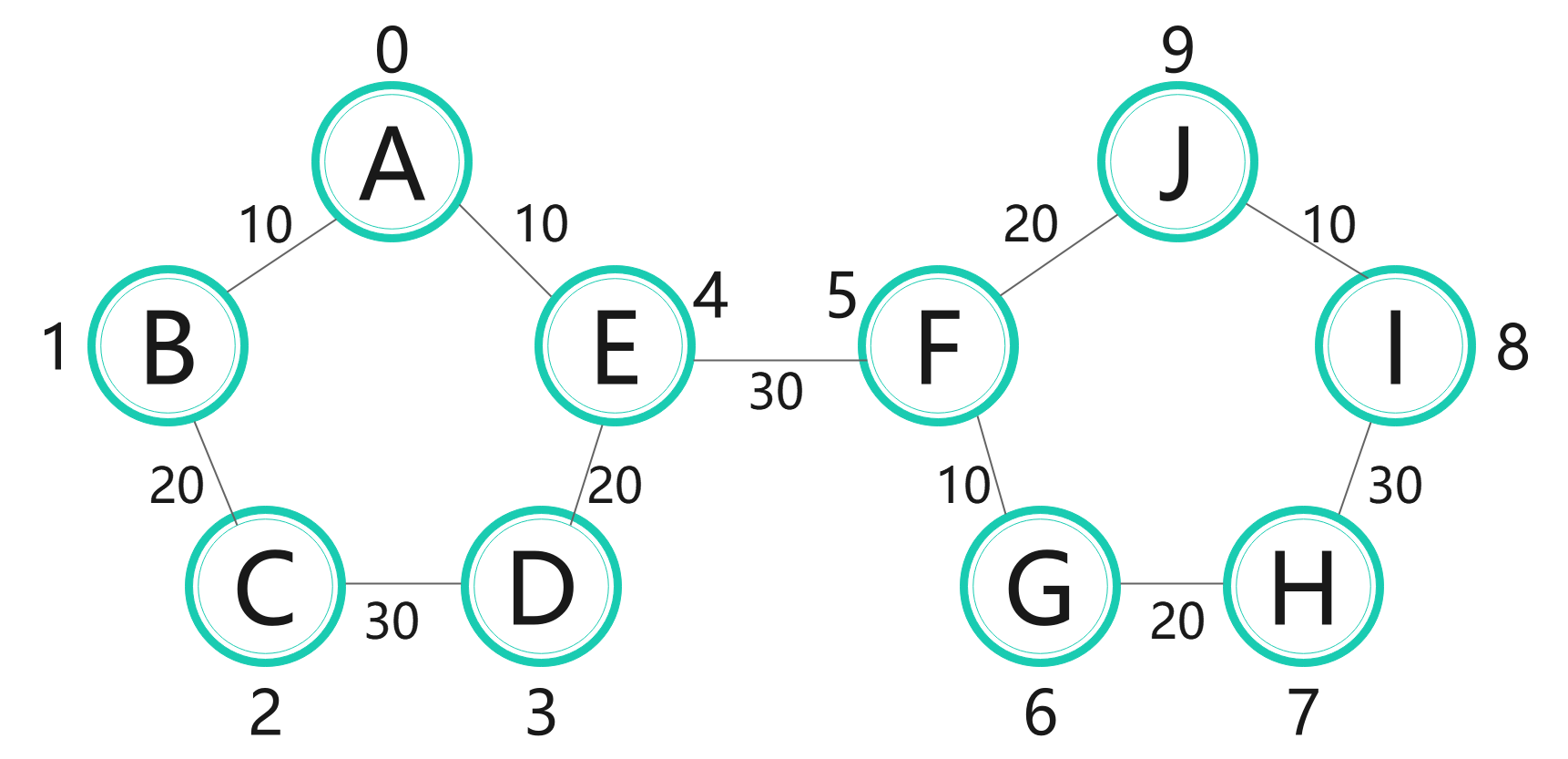
**算法时间复杂度：**O(V²)（V为顶点数）

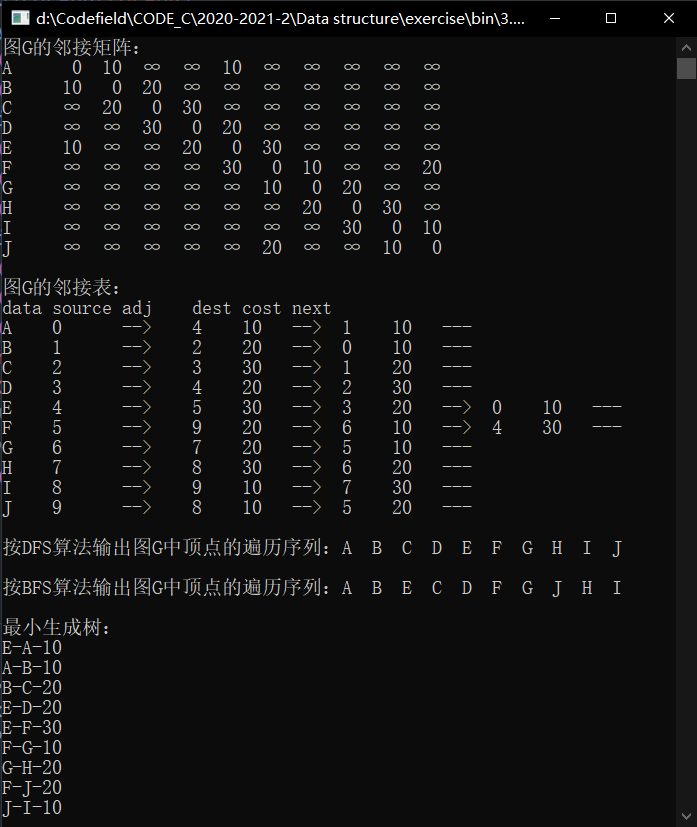
## 程序运行过程及结果（截屏粘贴）

测试数据1：

带权无向图，第一行为顶点信息，其余为边信息：



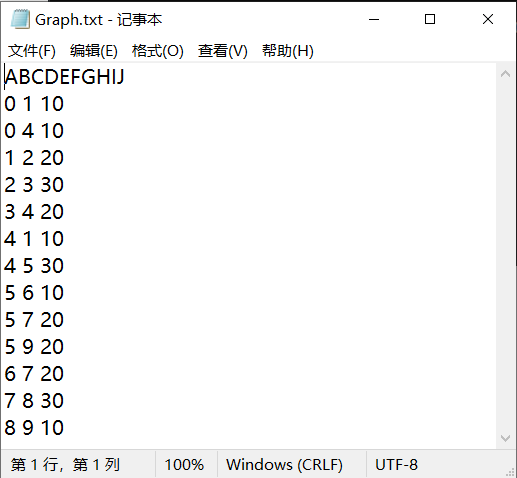


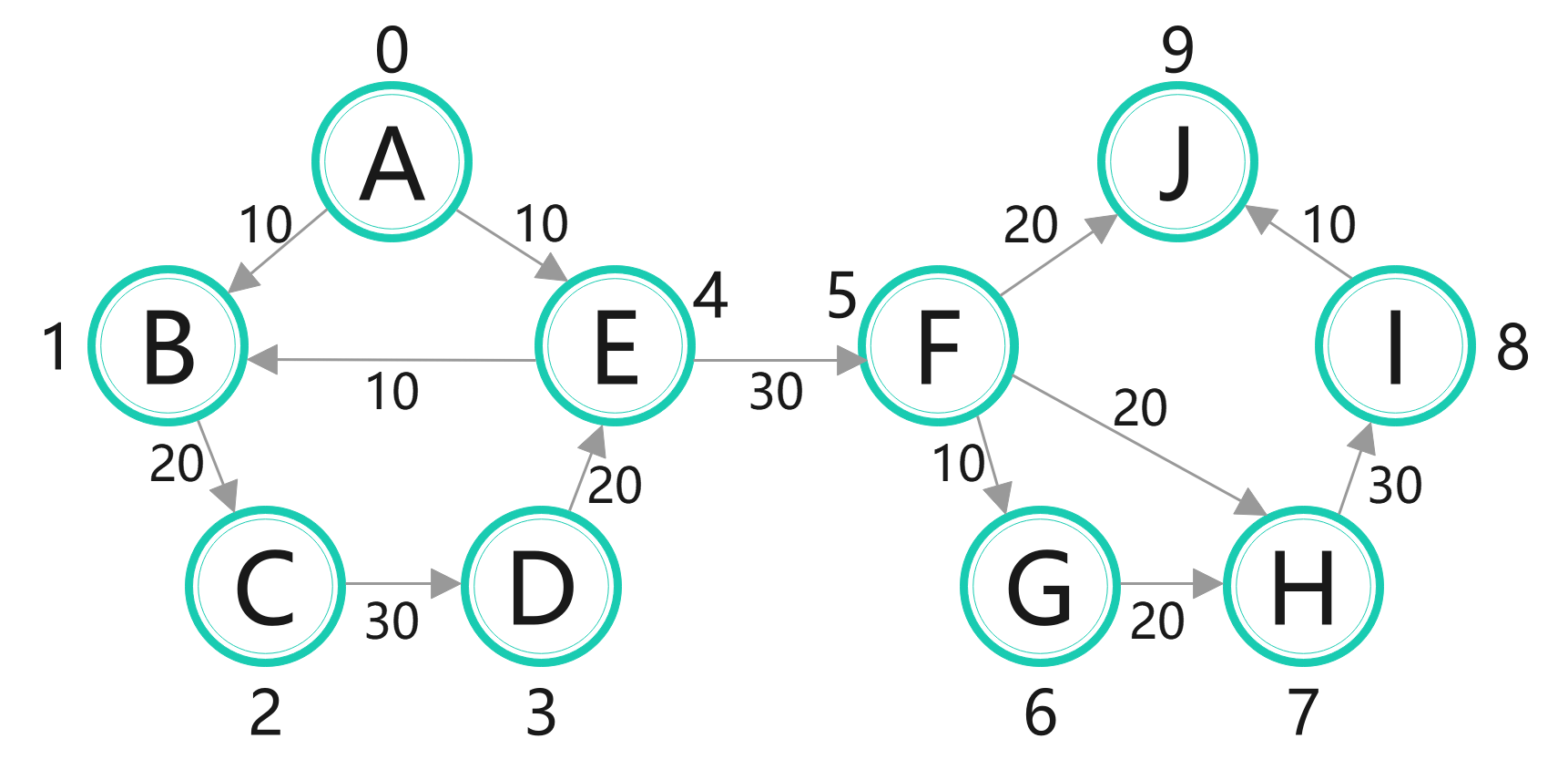


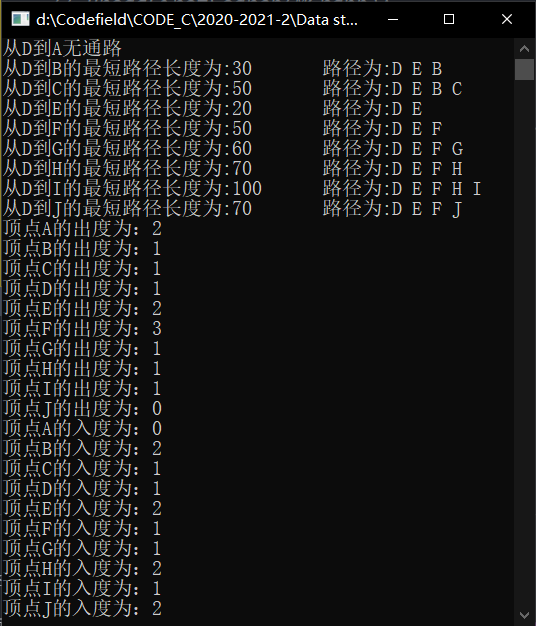
（注：生成最小生成树的初始顶点为E）

测试数据2：

带权有向图，第一行为顶点信息，其余为边信息：







（注：求最短路径的起点为D）

# 问题与总结

1.图形数据的结构比较复杂，有多种存储方式可以选择，不同的存储方式有不同的特点。

2.BFS算法与二叉树的前序遍历算法比较相似，可以通过类比、画图方法理解。

3.DFS算法可以借助队列方便地实现。

4. Prim算法较难理解，步骤复杂，可通过画图理解。

5. Dijkastra算法与Prim算法有许多步骤类似，可以通过类比理解。

6.计算有向图G各顶点的入度和出度，首先要了解入度和出度的概念。

7. 制作实验报告的过程中，学会制作流程图，可以方便、形象地看出算法的结构、运行步骤，比文字描述更容易理解。

# 附录：源代码

#include "SeqList.h"

#include "SeqCQueue.h"

typedef struct

{

SeqList Vertices;

int edge[MaxVertices][MaxVertices];

int numOfEdges;

} AdjMGraph;

typedef struct

{

int V1;

int V2;

int weight;

} RowColWeight;

typedef struct Node

{

int dest;

int cost;

struct Node \*next;

} Edge;

typedef struct

{

ElemType data;

int source;

Edge \*adj;

} AdjLHeight;

typedef struct

{

AdjLHeight a[MaxVertices];

int numOfVerts;

int numOfEdges;

} AdjLGraph;

typedef struct

{

VerT start;

VerT end;

int weight;

} MinSpanTree;

typedef struct

{

int start;

int weight;

} lowCost;

//从文件中读取图G的顶点信息和边信息

void readFile(FILE \*fp, ElemType V[], int \*n, RowColWeight E[], int \*e)

{

char ch;

int V1, V2, weight;

while (1)

{

if ((fscanf(fp, "%c", &ch)) == EOF)

return;

if (ch == '\n')

break;

V[\*n] = ch;

(\*n)++;

}

while ((fscanf(fp, "%d%d%d", &V1, &V2, &weight)) != EOF)

{

E[\*e].V1 = V1;

E[\*e].V2 = V2;

E[\*e].weight = weight;

(\*e)++;

}

}

//用邻接矩阵作为存储结构存储图G

void CreatAdjMGraph(AdjMGraph \*G, ElemType V[], int n, RowColWeight E[], int e)

{

ListInitiate(&G->Vertices);

int i, j;

for (i = 0; i < n; i++)

for (j = 0; j < n; j++)

{

if (i == j)

G->edge[i][j] = 0;

else

G->edge[i][j] = MaxWeight;

}

G->numOfEdges = 0;

for (i = 0; i < n; i++)

ListInsert(&G->Vertices, G->Vertices.size, V[i]);

for (i = 0; i < e; i++)

{

G->edge[E[i].V1][E[i].V2] = E[i].weight;

G->edge[E[i].V2][E[i].V1] = E[i].weight;

G->numOfEdges++;

}

}

//输出图G邻接矩阵

void PrintAdjMGraph(AdjMGraph G)

{

int i, j;

for (i = 0; i < G.Vertices.size; i++)

{

printf("%c ", G.Vertices.list[i]);

for (j = 0; j < G.Vertices.size; j++)

if (G.edge[i][j] != MaxWeight)

printf("%4d", G.edge[i][j]);

else

printf("%4s", "∞");

printf("\n");

}

}

//用邻接表作为存储结构存储图G

void CreatAdjLGraph(AdjLGraph \*G, ElemType V[], int n, RowColWeight E[], int e)

{

int i;

G->numOfEdges = 0;

G->numOfVerts = 0;

for (i = 0; i < MaxVertices; i++)

{

G->a[i].source = i;

G->a[i].adj = NULL;

}

for (i = 0; i < n; i++)

{

G->a[i].data = V[i];

G->numOfVerts++;

}

for (i = 0; i < e; i++)

{

Edge \*p, \*q;

p = (Edge \*)malloc(sizeof(Edge));

p->dest = E[i].V2;

p->cost = E[i].weight;

p->next = G->a[E[i].V1].adj;

G->a[E[i].V1].adj = p;

q = (Edge \*)malloc(sizeof(Edge));

q->dest = E[i].V1;

q->cost = E[i].weight;

q->next = G->a[E[i].V2].adj;

G->a[E[i].V2].adj = q;

G->numOfEdges++;

}

}

//输出图G邻接表

void PrintAdjLGraph(AdjLGraph G)

{

Edge \*p;

int i;

printf("%-5s%-7s%-7s%-5s%-5s%-5s\n", "data", "source", "adj", "dest", "cost", "next");

for (i = 0; i < G.numOfVerts; i++)

{

printf("%-5c%-7d", G.a[i].data, G.a[i].source);

if (G.a[i].adj == NULL)

printf("%-7s", "---");

else

{

printf("%-7s", "-->");

p = G.a[i].adj;

while (p != NULL)

{

printf("%-5d%-5d", p->dest, p->cost);

if (p->next != NULL)

printf("%-5s", "-->");

else

printf("%-5s", "---");

p = p->next;

}

}

printf("\n");

}

}

//销毁图G（通过邻接表存储）

void DestroyAdjLGraph(AdjLGraph \*G)

{

Edge \*p, \*tmp;

int i;

for (i = 0; i < G->numOfVerts; i++)

{

p = G->a[i].adj;

while (p != NULL)

{

tmp = p;

p = p->next;

free(tmp);

}

}

}

//取第一个邻接顶点

int GetFirstVex(AdjMGraph G, int v)

{

int col;

for (col = 0; col < G.Vertices.size; col++)

if (G.edge[v][col] > 0 && G.edge[v][col] < MaxWeight)

return col;

return -1;

}

//取下一个邻接顶点

int GetNextVex(AdjMGraph G, int v1, int v2)

{

int col;

for (col = v2 + 1; col <= G.Vertices.size; col++)

if (G.edge[v1][col] > 0 && G.edge[v1][col] < MaxWeight)

return col;

return -1;

}

//DFS算法

void DepthFSearch(AdjMGraph G, int v, int visited[])

{

int w;

printf("%c ", G.Vertices.list[v]);

visited[v] = 1;

w = GetFirstVex(G, v);

while (w != -1)

{

if (!visited[w])

DepthFSearch(G, w, visited);

w = GetNextVex(G, v, w);

}

}

//DFS算法

void DepthFirstSearch(AdjMGraph G)

{

int i;

int \*visited = (int \*)malloc(sizeof(int) \* G.Vertices.size);

for (i = 0; i < G.Vertices.size; i++)

visited[i] = 0;

for (i = 0; i < G.Vertices.size; i++)

if (!visited[i])

DepthFSearch(G, i, visited);

free(visited);

}

//BFS算法

void BroadFSearch(AdjMGraph G, int v, int visited[])

{

int u, w;

SeqCQueue queue;

printf("%c ", G.Vertices.list[v]);

visited[v] = 1;

QueueInitiate(&queue);

QueueAppend(&queue, v);

while (QueueNotEmpty(queue))

{

QueueDelete(&queue, &u);

w = GetFirstVex(G, u);

while (w != -1)

{

if (!visited[w])

{

printf("%c ", G.Vertices.list[w]);

visited[w] = 1;

QueueAppend(&queue, w);

}

w = GetNextVex(G, u, w);

}

}

}

//BFS算法

void BroadFirstSearch(AdjMGraph G)

{

int i;

int \*visited = (int \*)malloc(sizeof(int) \* G.Vertices.size);

for (i = 0; i < G.Vertices.size; i++)

visited[i] = 0;

for (i = 0; i < G.Vertices.size; i++)

if (!visited[i])

BroadFSearch(G, i, visited);

free(visited);

}

//Prime算法从某个指定的顶点出发输出图G的最小生成树

void Prim(AdjMGraph G, MinSpanTree MinSpanTreeEdge[], int start)

{

VerT x;

int i, j, k;

int n = G.Vertices.size, minCost;

if (start >= n)

return;

lowCost \*Vertices = (lowCost \*)malloc(sizeof(lowCost) \* n);

for (i = 0; i < n; i++)

{

Vertices[i].start = start;

Vertices[i].weight = G.edge[start][i];

}

Vertices[start].weight = -1;

for (i = 0; i < n - 1; i++)

{

minCost = MaxWeight;

for (j = 0; j < n; j++)

{

if (Vertices[j].weight < minCost && Vertices[j].weight > 0)

{

minCost = Vertices[j].weight;

k = j;

}

}

if (minCost == MaxWeight)

{

printf("该图不是连通图，无最小生成树!\n");

free(MinSpanTreeEdge);

free(Vertices);

return;

}

ListGet(G.Vertices, k, &x);

MinSpanTreeEdge[i].end = x;

ListGet(G.Vertices, Vertices[k].start, &x);

MinSpanTreeEdge[i].start = x;

MinSpanTreeEdge[i].weight = minCost;

Vertices[k].weight = -1;

for (j = 0; j < n; j++)

{

if (G.edge[k][j] < Vertices[j].weight)

{

Vertices[j].start = k;

Vertices[j].weight = G.edge[k][j];

}

}

}

for (i = 0; i < n - 1; i++)

printf("%c-%c-%d\n", MinSpanTreeEdge[i].start, MinSpanTreeEdge[i].end, MinSpanTreeEdge[i].weight);

free(MinSpanTreeEdge);

free(Vertices);

}

//输出最短路径

void Ppath(AdjMGraph G, int path[], int i)

{

char ch;

if (path[i] == -1)

return;

Ppath(G, path, path[i]);

ListGet(G.Vertices, path[i], &ch);

printf("%c ", ch);

}

//输出最短路径及最短路径值

void Dispath(AdjMGraph G, int distance[], int path[], int v)

{

char start, end;

int i;

ListGet(G.Vertices, v, &start);

for (i = 0; i < G.Vertices.size; i++)

{

if (i == v)

continue;

ListGet(G.Vertices, i, &end);

if (distance[i] == MaxWeight)

{

printf("从%c到%c无通路\n", start, end);

continue;

}

printf("从%c到%c的最短路径长度为:%d\t路径为:", start, end, distance[i]);

Ppath(G, path, i);

printf("%c\n", end);

}

}

//Dijkastra算法求从有向图的某个节点出发到其余各顶点的最短路径和最短路径值

void Dijkstra(AdjMGraph G, int v0)

{

int n = G.Vertices.size;

int minDis, i, j, u;

int s[MaxVertices];

int distance[MaxVertices];

int path[MaxVertices];

if (v0 >= G.Vertices.size)

return;

for (i = 0; i < n; i++)

{

distance[i] = G.edge[v0][i];

s[i] = 0;

if (i != v0 && distance[i] < MaxWeight)

path[i] = v0;

else

path[i] = -1;

}

s[v0] = 1;

for (i = 1; i < n; i++)

{

minDis = MaxWeight;

for (j = 0; j < n; j++)

if (s[j] == 0 && distance[j] < minDis)

{

u = j;

minDis = distance[j];

}

if (minDis == MaxWeight)

break;

s[u] = 1;

for (j = 0; j < n; j++)

if (s[j] == 0 && G.edge[u][j] < MaxWeight && distance[u] + G.edge[u][j] < distance[j])

{

distance[j] = distance[u] + G.edge[u][j];

path[j] = u;

}

}

Dispath(G, distance, path, v0);

}

//求邻接矩阵存储结构的有向图G中各顶点的入度和出度

void Degree(AdjMGraph G)

{

int Indegree[MaxVertices] = {0};

int Outdegree;

char ch;

int i, j;

for (i = 0; i < G.Vertices.size; i++)

{

Outdegree = 0;

ListGet(G.Vertices, i, &ch);

for (j = 0; j < G.Vertices.size; j++)

{

if (j != i && G.edge[i][j] < MaxWeight)

{

Outdegree++;

Indegree[j]++;

}

}

printf("顶点%c的出度为：%d\n", ch, Outdegree);

}

for (i = 0; i < G.Vertices.size; i++)

{

ListGet(G.Vertices, i, &ch);

printf("顶点%c的入度为：%d\n", ch, Indegree[i]);

}

}

//主函数

#include <stdio.h>

#include <stdlib.h>

typedef char ElemType;

typedef int DataType;

typedef char VerT;

#define MaxSize 10

#define MaxQueueSize 10

#define MaxVertices 10

#define MaxWeight 10000

#include "3.h"

int main()

{

int n = 0, e = 0;

AdjMGraph MGraph;

AdjLGraph LGraph;

MinSpanTree \*MinSpanTreeEdge;

ElemType V[MaxVertices];

RowColWeight rcw[MaxVertices \* MaxVertices];

FILE \*fp;

if ((fp = fopen("Graph.txt", "r")) == NULL)

{

printf("数据文件无法打开！");

return 0;

}

readFile(fp, V, &n, rcw, &e);

fclose(fp);

CreatAdjMGraph(&MGraph, V, n, rcw, e);

printf("图G的邻接矩阵：\n");

PrintAdjMGraph(MGraph);

printf("\n");

CreatAdjLGraph(&LGraph, V, n, rcw, e);

printf("图G的邻接表：\n");

PrintAdjLGraph(LGraph);

DestroyAdjLGraph(&LGraph);

printf("\n");

printf("按DFS算法输出图G中顶点的遍历序列：");

DepthFirstSearch(MGraph);

printf("\n");

printf("\n");

printf("按BFS算法输出图G中顶点的遍历序列：");

BroadFirstSearch(MGraph);

printf("\n");

printf("\n");

printf("最小生成树：\n");

MinSpanTreeEdge = (MinSpanTree \*)malloc(sizeof(MinSpanTree) \* (n - 1));

Prim(MGraph, MinSpanTreeEdge, 4);

// Dijkstra(MGraph, 0);

// Degree(MGraph);

system("pause");

return 0;

}

/\* 创建带权有向图函数略 \*/

/\* 队列头文件SeqCQueue.h和顺序表头文件SeqList.h略 \*/