《数据结构与算法实验》第 12 次实验

学院： 专业： 年级：
姓名 学号： 日期： 2022 年 6 月 10 日

第一部分 验证实验

一、 实验目的

1. 掌握两种最小生成树的算法：Prim 算法和 Kruskal 算法。

2. 掌握两种最短路径的算法：Dijkstra 算法和 Floyd 算法。

二、 实验内容

1. Prim 算法（参考课本 p164）

• 最小生成树问题：设 G=(V,E)是一个无向联通网，生成树上各边的权值之和称为该生成树的代价，在所有的生成树中，代价最小的生成树称为最小生成树。

2. Kruskal 算法（参考课本 p167）

• 最小生成树问题：设 G=(V,E)是一个无向联通网，生成树上各边的权值之和称为该生成树的代价，在所有的生成树中，代价最小的生成树称为最小生成树。

3. Dijkstra 算法（参考课本 p170）

• 最短路径问题：在非网图中，最短路径是指两顶点之间经历的边数最少的路径，路径上的第一个顶点称为源点，最后一个顶点称为终点。在网图中，最短路径是指两顶点之间经历的边上权值之和最小。在本次实验中，我们讨论有向网图的最短路径问题。

4. Floyd 算法（参考课本 p173）

• 最短路径问题：在非网图中，最短路径是指两顶点之间经历的边数最少的路径，路径上的第一个顶点称为源点，最后一个顶点称为终点。在网图中，最短路径是指两顶点之间经历的

边上权值之和最小。在本次实验中，我们讨论有向网图的最短路径问题。

三、 设计编码

1.Prim

Prim 算法的基本思想是：设 G=(V,E)是一个无向连通网，令 T=(U,TE)是 G 的最小生成树。T 的初始状态为 $U=v_0(v_0\in V)$，TE={}，然后重复执行下述操作：

在所有 u∈U，v∈V-U 的边中找一条代价最小的边(u,v)并入集合 TE，同时 v 并入 U，直至 U=V 为止。此时 TE 中必有 n-1 条边，则 T 就是一棵最小生成树。

Prim 算法的基本思想用伪代码描述如下：

```
1.初始化：U={v0}；TE={}；
2.重复下述操作直到 U=V：
  2.1 在 E 中寻找最短边 v,且满足 u∈U，v∈V-U；
  2.2 U=U+{v}；
  2.3 TE=TE+{(u,v)}；
```

显然，Prim 算法的关键是如何找到连接 U 和 V-U 的最短边来扩充生成树 T。设当前 T 中有 k 个顶点，则所有满足 u∈U，v∈V-U 的边最多有 k×(n-k)条，从如此之大的边集中选取最短边是不太经济的。利用 MST 性质，可以用下述方法构造候选最短边集：

对应 V-U 中的每个顶点，只保留从该顶点到 U 中某顶点的最短边，即候选最短边集为 V-U 中 n-k 个顶点所关联的 n-k 条最短边的集合。

下面讨论 Prim 算法基于的存储结构。

(1)图的存储结构：由于在算法执行过程中，需要不断读取任意两个顶点之间边的权值，所以，图采用邻接矩阵存储

(2)候选最短边集：设置一个数组 shortedge[n]表示候选最短边集，数组元素包括 adjvex 和 lowcost 两个域，分别表示候选最短边的邻接点和权值。

2.Kruskal 算法

Kruskal 算法的基本思想是：设无向连通网为 G=(V,E)，令 G 的最小生成树为 T=(U,TE)，其初态为 U=V,TE={}，T 中各顶点各自构成一个连通分量。然后按照边的权值由小到大的顺序，依次考察边集 E 中的各条边。若被考察边的两个顶点属于 T 的两个不同的连通分量，则

将此边加入到 TE 中，同时把两个连通分量连接为一个连通分量；若被考察边的两个顶点属于同一个连通分量，则舍去此边造成回路。如此下去，当 T 中的连通分量个数为 1 时，此连通分量便为 G 的一棵最小生成树。

Kruskal 算法的基本思想用伪代码描述如下：

```
1.初始化:U=V,TE={};
2.重复下述操作直到 T 中的连通分量个数为 1:
  2.1 在 E 中寻找最短边(u,v);
  2.2 如果顶点 u、v 位于 m 的两个不同连通分量,则
      2.2.1 将边(u,v)并入 TE;
      2.2.2 将这两个连通分量合为一个;
  2.3 在 E 中标记边(u,v),使得(u,v)不参加后续最短边的选取;
```

显然，实现 Kruskal 算法的关键之处是：如何判别被考察边的两个顶点是否位于两个连通分量（即是否与生成树中的边形成回路）。

3.Dijkstra 算法

Dijkstra 算法用于求单源点最短路径问题，问题描述如下：给定带权有向图 G=(V,E)和源点 v∈V，求从 v 到 G 中其余各顶点的最短路径。

Dijkstra 提出了一个按路径长度递增的次序产生最短路径的算法，其基本思想是：设置一个集合 S 存放已经找到最短路径的顶点，S 的初始状态只包含源点 v，对 vi∈V-S.

假设从源点 v 到 v_i 的有向边为最短路径。以后每求得一条最短路径就将 v 加入集合 S 中，并将路径 v,…,v_k,v_i 与原来的假设相比较，取路径长度较小者为当前最短路径。重复上述过程，直到集合 V 中全部顶点加入到集合 S 中。

4.Floyd 算法

Floyd 算法用于求每一对顶点之间的最短路径问题，问题描述如下：给定带权有向图 G=(V,E)，对任意顶点 v_i 和 v_j(i≠j)，求顶点 v 到顶点可的最短路径。

解决这个问题的一个办法是：每次以一个顶点为源点，调用 Dijkstra 算法 n 次，便可求得每一对顶点之间的最短路径，显然，时间复杂度为 $O(n^3)$。弗洛伊德提出了求每对顶点之间的最短路径算法——Floyd 算法，其时间复杂度也是 $O(n^3)$，但形式上要简单一些。

Floyd 算法的基本思想是：假设从 v_i 到 v_j 的弧（若从 v_i 到 v_j 的弧不存在，则将其弧的权值看成∞)是最短路径，然后进行 n 次试探。首先比较 v_i,v_j 和 v_i,v_0,v_j 的路径长度，取长度

较短者作为从 v_i 到 v_j 中间顶点的编号不大于 0 的最短路径。在路径上再增加一个顶点 v_1，将 $v_i, \cdots, v_1, \cdots, v_j$ 和已经得到的从 v_i 到 v_j 中间顶点的编号不大于 0 的最短路径相比较，取长度较短者作为中间顶点的编号不大于 1 的最短路径。以此类推，在一般情况下，若 $v_i, \cdots, v_k$ 和 $v_k, \cdots, v_j$ 分别是从 v_i 到 v_k 和从 v_k 到 v_j 中间顶点的编号不大于 k-1 的最短路径，则将 $v_i, \cdots, v_k, \cdots, v_j$ 和已经得到的从 vi 到 vj；中间顶点的编号不大于 k-1 的最短路径相比较，取长度较短者为从 v_i 到 v_j 中间顶点的编号不大于 k 的最短路径。经过 n 次比较后，最后求得的必是从 v_i 到 v_j 的最短路径。

四、具体代码（注：实验编码格式为 UTF-8/GB2312）

1.Prim

(1)MGraph.h

```
#ifndef MGraph_H
#define MGraph_H

const int MaxSize = 10;
extern int visited[MaxSize];

struct Candidate
{
    int adjvex;
    int lowcost;
};

int MinEdge(Candidate shortEdge[], int vertexNum);

class MGraph
{
    public:
        MGraph();
        ~MGraph();
        void Prim();
    private:
        int arc[MaxSize][MaxSize];
        int vertexNum, arcNum;
};

#endif
```

(2)MGraph.cpp

```
#include <iostream>
#include "MGraph.h"
using namespace std;

extern int visited[];

MGraph::MGraph()
{
    cout<<"输入顶点数和边数：";
    cin>>vertexNum>>arcNum;
    int i, j, m;
    for (i=0; i<vertexNum; i++)
        for (j=0; j<vertexNum; j++)
            arc[i][j]=1000;//以 1000 当做无穷
    for (int k=0; k<arcNum; k++)
    {
        cout<<"请输入边的两个顶点的序号和其权值：";
        cin>>i;
        cin>>j;
        cin>>m;
        arc[i][j]=m; arc[j][i]=m;
    }
}

MGraph::~MGraph()
{
    //empty
}

void MGraph::Prim()
{
    int k;
    Candidate shortEdge[MaxSize];
    for (int i = 0; i < vertexNum; i++)
    {
        shortEdge[i].lowcost = arc[0][i];
        shortEdge[i].adjvex = 0;
    }
    shortEdge[0].lowcost = 0;
    for (int i = 0; i < vertexNum-1; i++)
    {
        k = MinEdge(shortEdge, vertexNum);
        cout<<"("<<k<<", "<<shortEdge[k].adjvex<<") "<<shortEdge[k].lowcost<<endl;
```

```
        shortEdge[k].lowcost = 0;
        for (int j = 0; j < vertexNum-1; j++)
        {
            if (arc[k][j] < shortEdge[j].lowcost)
            {
                shortEdge[j].lowcost = arc[k][j];
                shortEdge[j].adjvex = k;
            }
        }
    }
}

int MinEdge(Candidate shortEdge[], int vertexNum)
{
    int min = 1000;
    int k;
    for (int i = 0; i < vertexNum; i++)
    {
        if (shortEdge[i].lowcost != 0 && shortEdge[i].lowcost < min)
        {
            min = shortEdge[i].lowcost;
            k = i;
        }
    }
    return k;
}
```

(3)MGraph_main.cpp

```
#include <iostream>
#include "MGraph.cpp"
using namespace std;

int visited[MaxSize] = {0};

int main()
{
    MGraph MG;
    cout<<"最小生成树的各边权值如下："<<endl;
    MG.Prim();
    return 0;
}
```

2. Kruskal

(1) EdgeGraph.h

```h
#ifndef EdgeGraph_H
#define EdgeGraph_H

const int MaxSize = 10;
const int MaxEdge = 100;
extern int visited[MaxSize];

struct EdgeType
{
    int from, to;
    int weight;
};

struct EdgeGraph
{
    EdgeType edge[MaxEdge];
    int vertexNum, edgNum;
};


void Initial(EdgeGraph &E);
void Kruskal(EdgeGraph &E);
int FindRoot(int parent[], int v);

#endif
```

(2) EdgeGraph.cpp

```cpp
#include <iostream>
#include "EdgeGraph.h"
using namespace std;

void Initial(EdgeGraph &E)          //构造图
{
    cout<<"输入顶点数和边数：";
    cin>>E.vertexNum>>E.edgNum;
    for (int i = 0; i < E.edgNum; i++)
    {
        cout<<"输入边上两个顶点和权值（权值从小到大输入）：";
        cin>>E.edge[i].from>>E.edge[i].to>>E.edge[i].weight;
    }
}
```

```
void Kruskal(EdgeGraph &E)
{
    int parent[E.vertexNum];
    for (int i = 0; i < E.vertexNum; i++)
        parent[i] = -1;
    int num = 0, vex1, vex2;
    for (int i = 0; i < E.edgNum; i++)
    {
        vex1 = FindRoot(parent, E.edge[i].from);
        vex2 = FindRoot(parent, E.edge[i].to);
        if (vex1 != vex2)
        {
            cout<<"("<<E.edge[i].from<<", "<<E.edge[i].to<<")
"<<E.edge[i].weight<<endl;
            parent[vex2] = vex1;
            num++;
            if (num == E.vertexNum - 1)  return;
        }
    }
}

int FindRoot(int parent[], int v)
{
    while (parent[v] > -1)  v = parent[v];
    return v;
}
```

(3)EdgeGraph_main.cpp

```
#include <iostream>
#include "EdgeGraph.cpp"
using namespace std;

int main()
{
    EdgeGraph E;
    Initial(E);
    cout<<"最小生成树的边及其权值如下："<<endl;
    Kruskal(E);
    return 0;
}
```

3.Dijkstra

(1)MGraph.h

```h
#ifndef MGraph_H
#define MGraph_H
#include <string>
using namespace std;
int const MaxSize = 10;

class MGraph
{
    public:
        MGraph(string a[], int n, int e);
        ~MGraph();
        friend void Dijkstra(MGraph G, int v);
    private:
        string vertex[MaxSize];
        int arc[MaxSize][MaxSize];
        int vertexNum, arcNum;
};

void Dijkstra(MGraph G, int v);

#endif
```

(2)MGraph.cpp

```cpp
#include <iostream>
#include <string>
#include "MGraph.h"
using namespace std;

extern int visited[];

MGraph::MGraph(string a[], int n, int e)
{
    int i, j, k;
    vertexNum = n, arcNum = e;
    for ( i = 0; i < vertexNum; i++)
        vertex[i] = a[i];
    for ( i = 0; i < vertexNum; i++)
        for ( j = 0; j < vertexNum; j++)
            arc[i][j] = 0;
    for ( k = 0; k < arcNum; k++)
    {
        cout<<"请输入边的两个顶点的序号及权值: ";
```

```
        cin>>i>>j;
        cin>>arc[i][j];
    }
}

MGraph::~MGraph()
{
    //empty
}

void Dijkstra(MGraph G, int v)
{
    int n = G.vertexNum;
    int dist[n];
    string path[n];
    for (int i = 0; i < G.vertexNum; i++)
    {
        dist[i] = G.arc[v][i];
        if (dist[i] != 1000)  path[i] = G.vertex[v] + G.vertex[i];
        else path[i] = " ";
    }
    int num = 1;
    while(num < n)
    {
        int k = 0;
        while (dist[k] == 0)  k++;
        for (int i = 0; i < n; i++)
        {
            if (dist[i] != 0 && dist[i] < dist[k])
                k = i;
        }
        cout<<G.vertex[v]<<"到"<<G.vertex[k]<<"的最短路径为："<<path[k]<<"，长度为
"<<dist[k]<<endl;
        num++;
        for (int i = 0; i < n; i++)
            if (dist[i] > dist[k] + G.arc[k][i])
            {
                dist[i] = dist[k] + G.arc[k][i];
                path[i] = path[k] + G.vertex[i];
            }
        dist[k] = 0;
    }
}
```

(3)MGraph_main.cpp

```cpp
#include <iostream>
#include "MGraph.cpp"
using namespace std;

int main()
{
    string ch[] = {"A","B","C","D","E"};
    cout<<"输入顶点数和边数：";
    int n, e;
    cin>>n>>e;
    MGraph G(ch, n, e);
    cout<<"请输入原点："<<endl;
    string yd;
    cin>>yd;
    int v = yd[0] - ch[0][0];
    Dijkstra(G, v);
    return 0;
}
```

4. Floyd

(1)MGraph.h

```cpp
#ifndef MGraph_H
#define MGraph_H
#include <string>
using namespace std;
int const MaxSize = 10;

class MGraph
{
    public:
        MGraph(string a[], int n, int e);
        ~MGraph();
        friend void Floyd(MGraph G);
    private:
        string vertex[MaxSize];
        int arc[MaxSize][MaxSize];
        int vertexNum, arcNum;
};
void Floyd(MGraph G);
#endif
```

(2)MGraph.cpp

```
#include <iostream>
#include <string>
#include <iomanip>
#include "MGraph.h"
using namespace std;

extern int visited[];

MGraph::MGraph(string a[], int n, int e)
{
    int i, j, k;
    vertexNum = n, arcNum = e;
    for ( i = 0; i < vertexNum; i++)
        vertex[i] = a[i];
    for ( i = 0; i < vertexNum; i++)
        for ( j = 0; j < vertexNum; j++)
            arc[i][j] = 1000;
    for ( i = 0; i < vertexNum; i++)
        arc[i][i] = 0;
    for ( k = 0; k < arcNum; k++)
    {
        cout<<"请输入边的两个顶点的序号及权值：";
        cin>>i>>j;
        cin>>arc[i][j];
    }
}

MGraph::~MGraph()
{
    //empty
}

void Floyd(MGraph G)
{
    int n = G.vertexNum;
    int dist[n][n];
    string path[n][n];
    //int i, j;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            dist[i][j] = G.arc[i][j];
```

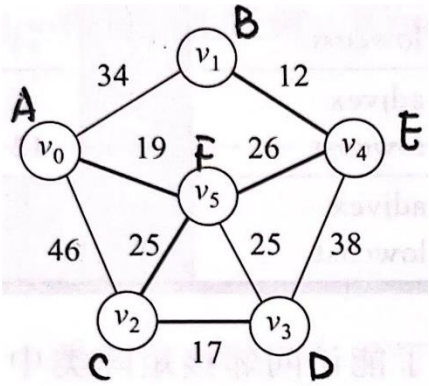
```
            if (dist[i][j] != 1000)  path[i][j] = G.vertex[i] + G.vertex[j];
            else path[i][j] = " ";
        }
    }
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (dist[i][j] > dist[i][k] + dist[k][j])
                {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    path[i][j] = path[i][k] + path[k][j].substr(1);
                }
    //输出
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
        {
            if(i != j)
            {
                if(dist[i][j] != 1000)
                    cout<<G.vertex[i]<<"到"<<G.vertex[j]<<"的最短路径为:
"<<setw(4)<<path[i][j]<<", 长度为"<<dist[i][j]<<endl;
                else
                    cout<<G.vertex[i]<<"到"<<G.vertex[j]<<"没有路径"<<endl;
            }
        }
}
```

(3)MGraph_main.cpp

```
#include <iostream>
#include "MGraph.cpp"
using namespace std;
int main()
{
    string ch[]={"A", "B", "C", "D", "E","F", "G", "H", "I", "J"};
    cout<<"输入顶点数和边数: ";
    int n, e;
    cin>>n>>e;
    MGraph G(ch, n, e);
    Floyd(G);
    return 0;
}
```

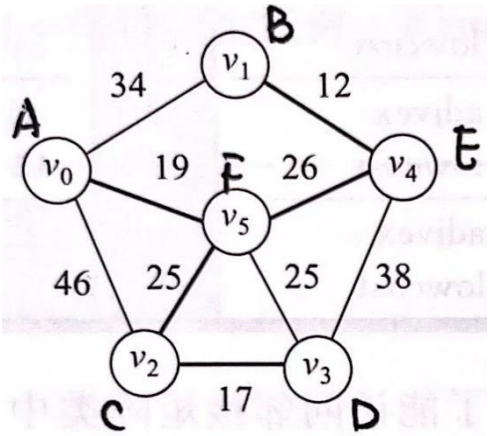
五、运行与测试

1.Prim



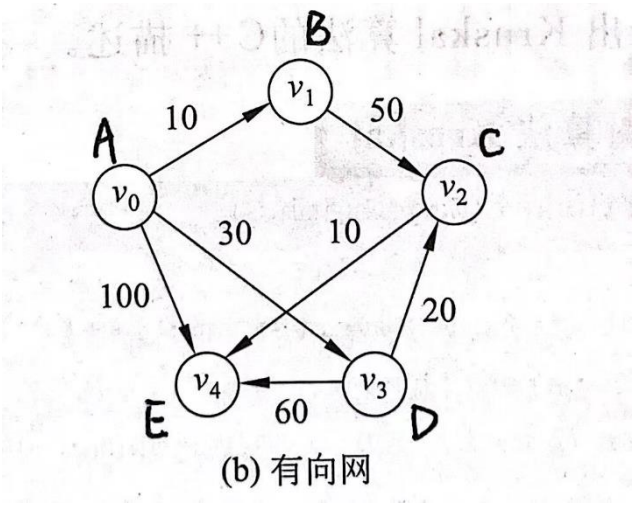

```
输入顶点数和边数：6 9
请输入边的两个顶点的序号和其权值：0 1 34
请输入边的两个顶点的序号和其权值：0 2 46
请输入边的两个顶点的序号和其权值：0 5 19
请输入边的两个顶点的序号和其权值：1 4 12
请输入边的两个顶点的序号和其权值：2 3 17
请输入边的两个顶点的序号和其权值：2 5 25
请输入边的两个顶点的序号和其权值：3 4 38
请输入边的两个顶点的序号和其权值：3 5 25
请输入边的两个顶点的序号和其权值：4 5 26
最小生成树的各边权值如下：
(5, 0) 19
(2, 5) 25
(3, 2) 17
(4, 5) 26
(1, 4) 12
```

2.Kruskal



```
输入顶点数和边数：6 9
输入边上两个顶点和权值（权值从小到大输入）：1 4 12
输入边上两个顶点和权值（权值从小到大输入）：2 3 17
输入边上两个顶点和权值（权值从小到大输入）：0 5 19
输入边上两个顶点和权值（权值从小到大输入）：2 5 25
输入边上两个顶点和权值（权值从小到大输入）：3 5 25
输入边上两个顶点和权值（权值从小到大输入）：4 5 26
输入边上两个顶点和权值（权值从小到大输入）：0 1 34
输入边上两个顶点和权值（权值从小到大输入）：3 4 38
输入边上两个顶点和权值（权值从小到大输入）：0 2 46
最小生成树的边及其权值如下：
(1, 4) 12
(2, 3) 17
(0, 5) 19
(2, 5) 25
(4, 5) 26
PS F:\DataStructure>
```

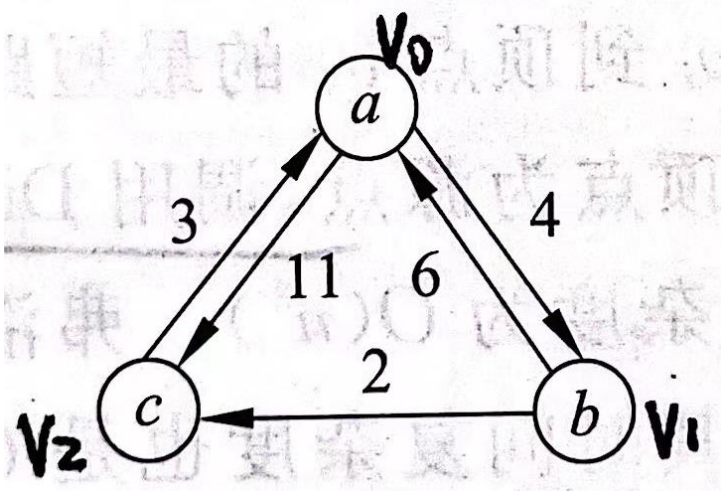
3.Dijkstra



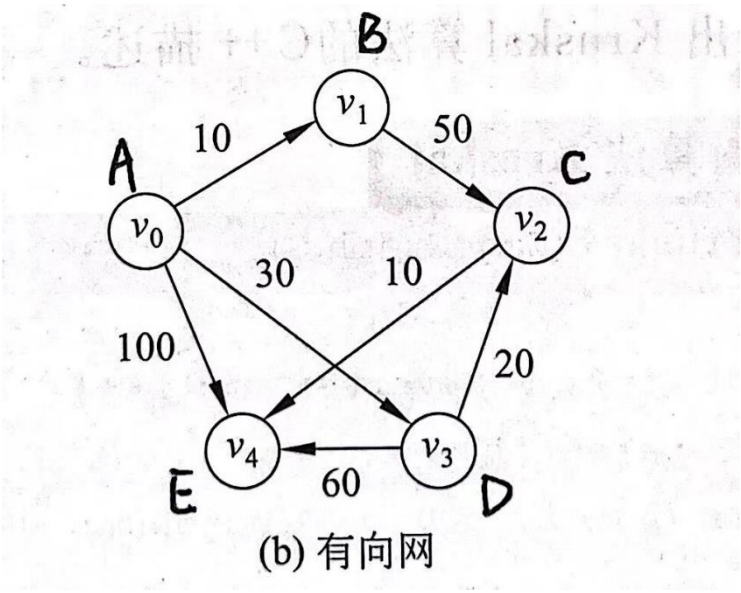
(b) 有向网

```
输入顶点数和边数：5 7
请输入边的两个顶点的序号及权值：0 1 10
请输入边的两个顶点的序号及权值：0 3 30
请输入边的两个顶点的序号及权值：0 4 100
请输入边的两个顶点的序号及权值：1 2 50
请输入边的两个顶点的序号及权值：2 4 10
请输入边的两个顶点的序号及权值：3 2 20
请输入原点：
A
A到B的最短路径为：AB，长度为10
A到D的最短路径为：ABD，长度为10
A到E的最短路径为：ABE，长度为10
A到的最短路径为：，长度为32767
```

4. Floyd




```
输入顶点数和边数：3 5
请输入边的两个顶点的序号及权值：0 1 4
请输入边的两个顶点的序号及权值：0 2 11
请输入边的两个顶点的序号及权值：1 0 6
请输入边的两个顶点的序号及权值：1 2 2
请输入边的两个顶点的序号及权值：2 0 3
A到B的最短路径为：  AB，长度为4
A到C的最短路径为：  ABC，长度为6
B到A的最短路径为：  BCA，长度为5
B到C的最短路径为：  BC，长度为2
C到A的最短路径为：  CA，长度为3
C到B的最短路径为：  CAB，长度为7
PS F:\DataStructure>
```



(b) 有向网

```
输入顶点数和边数: 5 7
请输入边的两个顶点的序号及权值: 0 1 10
请输入边的两个顶点的序号及权值: 0 3 30
请输入边的两个顶点的序号及权值: 0 4 100
请输入边的两个顶点的序号及权值: 1 2 50
请输入边的两个顶点的序号及权值: 2 4 10
请输入边的两个顶点的序号及权值: 3 2 20
请输入边的两个顶点的序号及权值: 3 4 60
A到B的最短路径为:   AB, 长度为10
A到C的最短路径为:  ADC, 长度为50
A到D的最短路径为:   AD, 长度为30
A到E的最短路径为: ADCE, 长度为60
B到A没有路径
B到C的最短路径为:   BC, 长度为50
B到D没有路径
B到E的最短路径为:  BCE, 长度为60
C到A没有路径
C到B没有路径
C到D没有路径
C到E的最短路径为:   CE, 长度为10
D到A没有路径
D到B没有路径
D到C的最短路径为:   DC, 长度为20
D到E的最短路径为:  DCE, 长度为30
E到A没有路径
E到B没有路径
E到C没有路径
E到D没有路径
PS F:\DataStructure>
```

第二部分 设计实验

一、实验目的

通过 2 个实验，解决 TSP 问题以及找出哈密顿路径问题，是对深度遍历算法和广度遍历算法的综合设计。

二、实验内容

1. TSP 问题：所谓 TSP 问题是指旅行家要旅行 n 个城市，要求各个城市经历且仅经历一次，并要求所走的路程最短。该问题又称为货郎担问题、邮递员问题、售货员问题，是图问题中最广为人知的问题。

2. 哈密顿路径：在图 G 中找出一条包含所有顶点的简单路径，该路径称为哈密顿路径。

三、实验内容

1. TSP 问题：参考实验书 p225

- 查找 TSP 问题的应用实例。
- 分析求 TSP 问题的全局最优解的时间复杂度。
- 设计一个求近似解的算法。
- 分析算法的时间复杂度。

编写伪代码如下：

```
1.任意选择某个顶点 v 作为出发点；
2.执行下述过程，直到所有顶点都被访问；
    2.1 v=最后一个被访问的顶点；
    2.2 在顶点 v 的邻接点中查找距离顶点 v 最近的未被访问的邻接点 k；
    2.3 访问顶点 k；
3.从最后一个访问的顶点直接回到出发点 v；
```

2. 哈密顿路径：参考实验书 p226

- 图 G 是非完全有向图。
- 设计算法判断图 G 是否存在哈密顿路径，如果存在，输出一条哈密顿路径。
- 分析算法的时间复杂度。

四、设计编码（注：实验编码格式为 UTF-8/GB2312）

1. TSP 问题

(1)MGraph.h

```
#ifndef MGraph_H
```

```
#define MGraph_H
#include <string>
using namespace std;
int const MaxSize = 10;

class MGraph
{
    public:
        MGraph(string a[], int n);
        ~MGraph();
        friend void TSP(MGraph &G, int v);
        friend int FindMin(MGraph &G, int v);
    private:
        string vertex[MaxSize];
        int arc[MaxSize][MaxSize];
        int vertexNum, arcNum;
};

void TSP(MGraph &G, int v);
int FindMin(MGraph &G, int v);

#endif
```

(2)MGraph.cpp

```
#include <iostream>
#include <string>
#include <iomanip>
#include "MGraph.h"
using namespace std;

extern int visited[];

MGraph::MGraph(string a[], int n)
{
    int i, j, k;
    vertexNum = n, arcNum = n*(n - 1);
    for ( i = 0; i < vertexNum; i++)
        vertex[i] = a[i];
    for ( i = 0; i < vertexNum; i++)
        for ( j = 0; j < vertexNum; j++)
            arc[i][j] = 1000;
    for ( i = 0; i < vertexNum; i++)
        arc[i][i] = 0;
```

```
    for ( k = 0; k < arcNum; k++)
    {
        cout<<"请输入边的两个顶点的序号及权值：";
        cin>>i>>j;
        cin>>arc[i][j];
    }
}

MGraph::~MGraph()
{
    //empty
}

void TSP(MGraph &G, int v)
{
    int p0 = v;
    string path = G.vertex[v];
    int length = 0;
    G.arc[v][v] = 1;
    int k;
    for (int i = 1; i < G.vertexNum; i++)
    {
        k = FindMin(G, v);
        path = path + G.vertex[k];
        length = length + G.arc[v][k];
        G.arc[k][k] = 1;
        v = k;
    }
    path = path + G.vertex[p0];
    length = length + G.arc[v][p0];
    cout<<G.vertex[p0]<<"走完所有顶点并回到自身的最短路径为："<<path<<"，长度为
"<<length<<endl;
}
int FindMin(MGraph &G, int v)
{
    int k = 0;
    while (G.arc[k][k] == 1 || k == v)  k++;
    for (int i = k; i < G.vertexNum; i++)
    {
        if (G.arc[i][i] != 1 && i != v &&G.arc[v][i] < G.arc[v][k])
            k = i;
    }
    return k;
}
```

(3)MGraph_main.cpp

```cpp
#include <iostream>
using namespace std;
#include <string>
#include "MGraph.cpp"

int main()
{
    string ch[ ]={"A", "B", "C", "D", "E","F"};
    cout<<"输入顶点数：";
    int n;
    cin>>n;
    MGraph G(ch, n);
    cout<<"请输入源点："<<endl;
    string yd;
    cin>>yd;
    int v=yd[0]-ch[0][0];
    TSP(G, v);
    return 0;
}
```

2. 哈密顿路径：

(1)ALGraph.h

```cpp
#ifndef MGraph_H
#define MGraph_H
#include <string>
using namespace std;
int const MaxSize = 10;

struct ShortEdge
{
    int lowcost;
    int adjvex;
};

class MGraph
{
    public:
        MGraph(string a[], int n, int e);
        ~MGraph();
        friend void Hamilton(MGraph &G, int v, int S[], int &count);
    private:
```

```
        string vertex[MaxSize];
        int arc[MaxSize][MaxSize];
        int vertexNum, arcNum;
};

void Hamilton(MGraph &G, int v, int S[], int &count);

#endif
```

(2) ALGraph.cpp

```
#include <iostream>
#include <string>
#include <iomanip>
#include "MGraph.h"
using namespace std;

extern int visited[MaxSize];
extern int count;
extern char S[MaxSize];

MGraph::MGraph(string a[], int n, int e)
{
    int i, j, k;
    vertexNum = n, arcNum = e;
    for ( i = 0; i < vertexNum; i++)
        vertex[i] = a[i];
    for ( i = 0; i < vertexNum; i++)
        for ( j = 0; j < vertexNum; j++)
            arc[i][j] = 0;
    for ( k = 0; k < arcNum; k++)
    {
        cout<<"请输入边的两个顶点的序号：";
        cin>>i>>j;
        arc[i][j] = 1;
    }
}

MGraph::~MGraph()
{
    //empty
}

void Hamilton(MGraph &G, int v, int S[], int &count)
```

```
{
    int j = 0;
    visited[v] = 1;
    S[count++] = v;
    if (count == G.vertexNum)
    {
        for (int i = 0; i < G.vertexNum; i++)
            cout << G.vertex[S[i]] << ' ';
        cout << endl;
        return;
    }
    for (j = 0; j < G.vertexNum; j++) {
        if (G.arc[v][j] != 0 && visited[j] == 0)
            Hamilton(G, j, S, count);
        if (count == G.vertexNum) return;
    }
    if (j == G.vertexNum)
    {
        visited[v] = 0;
        count--;
    }
    if (count == 0)
        cout << "no path" << endl;
}
```

(3)ALGraph_main.cpp

```
#include <iostream>
#include <string>
using namespace std;
#include "MGraph.cpp"

int visited[MaxSize] = { 0 };

int main() {
    string ch[4] = { "A","B","C","D" };
    MGraph G(ch, 4, 5);
    int count = 0; int *S = new int[4];
    cout << "从 A 点出发, ";
    Hamilton(G, 0, S, count);
    count = 0;
    for (int i = 0; i < MaxSize; i++)
        visited[i] = 0;
    cout << "从 B 点出发, ";
```

```
    Hamilton(G, 1, S, count);
    return 0;
}
```

五、运行结果

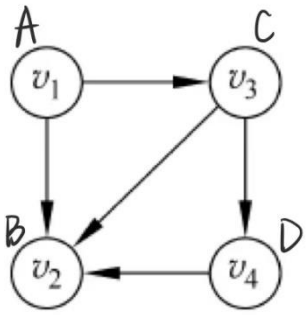
1. TSP 问题

$$C=\begin{bmatrix} \infty & 3 & 3 & 2 & 6 \\ 3 & \infty & 7 & 3 & 2 \\ 3 & 7 & \infty & 2 & 5 \\ 2 & 3 & 2 & \infty & 3 \\ 6 & 2 & 5 & 3 & \infty \end{bmatrix}$$

(a) 5城市的代价矩阵

```
输入顶点数: 5
请输入边的两个顶点的序号及权值: 0 1 3
请输入边的两个顶点的序号及权值: 0 2 3
请输入边的两个顶点的序号及权值: 0 3 2
请输入边的两个顶点的序号及权值: 0 4 6
请输入边的两个顶点的序号及权值: 1 0 3
请输入边的两个顶点的序号及权值: 1 2 7
请输入边的两个顶点的序号及权值: 1 3 3
请输入边的两个顶点的序号及权值: 1 4 2
请输入边的两个顶点的序号及权值: 2 0 3
请输入边的两个顶点的序号及权值: 2 1 7
请输入边的两个顶点的序号及权值: 2 3 2
请输入边的两个顶点的序号及权值: 2 4 5
请输入边的两个顶点的序号及权值: 3 0 2
请输入边的两个顶点的序号及权值: 3 1 3
请输入边的两个顶点的序号及权值: 3 2 2
请输入边的两个顶点的序号及权值: 3 4 3
请输入边的两个顶点的序号及权值: 4 0 6
请输入边的两个顶点的序号及权值: 4 1 2
请输入边的两个顶点的序号及权值: 4 2 5
请输入边的两个顶点的序号及权值: 4 3 3
请输入源点:
A
A走完所有顶点并回到自身的最短路径为: ADCEBA, 长度为14
```

2. 哈密顿路径




```
请输入边的两个顶点的序号：0 1
请输入边的两个顶点的序号：0 2
请输入边的两个顶点的序号：2 1
请输入边的两个顶点的序号：2 3
请输入边的两个顶点的序号：3 1
从A点出发，A C D B
从B点出发，no path
```

第三部分 综合实验 1：农夫过河

一、问题描述

一个农夫带一只狼、一棵白菜和一只山羊要从一条河的南岸过到北岸，农夫每次只能带一样东西过河，但是任意时刻如果农夫不在场时，狼要吃羊、羊要吃菜，请为农夫设计过河方案。

二、基本要求

- 为农夫过河问题设计抽象数据类型，体会数据模型在问题求解中的重要性。
- 设计一个算法求解农夫过河问题，并输出国和方案。
- 分析算法的时间复杂度。

三、设计思想

总的设计思想：要求解农夫过河问题，首先需要选择一个对问题中每个角色的位置进行描述的方法。一个很方便的办法是用四位二进制数顺序表示农夫、狼、白菜和羊的位置。例如，用 0 表示农夫或者某东西在河的南岸，1 表示在河的北岸，例如二进制数 0101 表示农夫和白

菜在河的南岸（南岸用 A 表示），而狼和羊在北岸（北岸用 B 表示）。0000 到 1111 共有 16 种状态，其中有些状态是禁止状态，如 0101 表示狼和羊在河的北岸。这样，问题转化为从初始状态二进制 0000（全部在河的南岸）出发，以二进制 1111（全部到达河的北岸）为最终目标，一步一步进行试探，每一步都搜索所有可能的选择，对每一步确定合适的选择再考虑下一步的各种方案，并且在序列中的每一个状态都可以从前一状态通过农夫可以带一样东西过河（即农夫的位置发生变化）到达。显然，这是一个对状态组成的图进行广度优先的遍历过程。

当然，也可以采用深度优先遍历思想求解农夫过河问题。

四、设计编码（注：实验编码格式为 UTF-8/GB2312）

1. MGraph.h

```
#ifndef MGraph_H
#define MGraph_H
#include <string>
using namespace std;
extern int visited[10];

class MGraph
{
    public:
        MGraph(string a[]);
        ~MGraph();
        void Path();
    private:
        string vertex[10];
        int arc[10][10];
        int vertexNum, arcNum;
};

#endif
```

2. MGraph.cpp

```
#include <iostream>
#include <string>
#include <iomanip>
#include "MGraph.h"
using namespace std;
```

```
MGraph::MGraph(string a[])
{
    int i, j, k;
    vertexNum = arcNum = 10;
    for ( i = 0; i < 10; i++)
        vertex[i] = a[i];
    for ( i = 0; i < 10; i++)
        for ( j = 0; j < 10; j++)
            arc[i][j] = 0;
    //邻接表(在这里排除不能取得的情况)
    arc[0][4] = arc[4][0] = 1;
    arc[1][6] = arc[6][1] = 1;
    arc[1][7] = arc[7][1] = 1;
    arc[2][6] = arc[6][2] = 1;
    arc[2][8] = arc[8][2] = 1;
    arc[3][4] = arc[4][3] = 1;
    arc[3][7] = arc[7][3] = 1;
    arc[3][8] = arc[8][3] = 1;
    arc[5][6] = arc[6][5] = 1;
    arc[5][9] = arc[9][5] = 1;
}

MGraph::~MGraph()
{
    //empty
}

void MGraph::Path()
{
    int S[10];
    int top = -1;
    visited[0] = 1;
    S[++top] = 0;
    int yes = 0;
    int t;
    while (top != -1 && yes == 0)
    {
        int p = 0;
        int i = S[top];
        while (arc[i][p] == 0)  p++;
        while (p < 10 && yes == 0)
        {
            t = p;
            if (t == 9)
```

```
            {
                S[++top] = t;
                yes = 1;
            }
            else if (visited[t] == 0)
            {
                visited[t] = 1;
                S[++top] = t;
                i = t;
                p = 0;
                while (arc[t][p] == 0)  p++;
            }
            else
            {
                p++;
                while (arc[i][p] == 0 && p < 10)  p++;
            }
        }
        if (p == 10)  top--;
    }
    string str;
    int count, p, bank;
    for (int i = 0; i < top; i++)
    {
        count = 0;
        p = 10;
        if (vertex[ S[i+1] ][0] == '1')  bank = 1;
        else bank = 0;

        for (int j = 1; j < 4; j++)
            if (vertex[ S[i+1] ][j] != vertex[ S[i] ][j])
            {
                p = j;
                count++;
                break;
            }
        if (p == 1) str = "wolf";
        else if (p == 2) str = "cabbage";
        else if (p == 3) str = "goat";
        if (count == 0)
        {
            if (bank == 1) cout<<"farmer -> B"<<endl;
            else cout<<"farmar -> A"<<endl;
        }
```

```
        else
        {
            if (bank == 1) cout<<"farmer + "<<str<<" -> B"<<endl;
            else cout<<"farmer + "<<str<<" -> A"<<endl;
        }
    }
}
```

3. MGraph_main.cpp

```
#include <iostream>
using namespace std;
#include "MGraph.cpp"

int visited[10];

int main()
{
    string ch[] = {"0000", "0100", "0010", "0001", "1001","0110", "1110", "1101",
"1011", "1111"};
    MGraph G(ch);
    for (int i = 0; i < 10; i++)
        visited[i] = 0;
    G.Path();
    return 0;
}
```

五、运行结果

邻接表如图。

	0000	0100	0010	0001	1001	0110	1110	1101	1011	1111
0000	0	0	0	0	1	0	0	0	0	0
0100	0	0	0	0	0	0	1	1	0	0
0010	0	0	0	0	0	0	1	0	1	0
0001	0	0	0	0	1	0	0	1	1	0
1001	1	0	0	1	0	0	0	0	0	0
0110	0	0	0	0	0	0	1	0	0	1
1110	0	1	1	0	0	1	0	0	0	0
1101	0	1	0	1	0	0	0	0	0	0
1011	0	0	1	1	0	0	0	0	0	0
1111	0	0	0	0	0	0	1	0	0	0

运行结果如图，验证方案可行。

```
farmer + goat -> B
farmar -> A
farmer + wolf -> B
farmer + goat -> A
farmer + cabbage -> B
farmar -> A
farmer + goat -> B
PS F:\DataStructure\12\T07-River>
```

第四部分 综合实验 2：医院选址

一、问题描述

n 个村庄之间的交通图可以用有向网图来表示，图中边<v_i,v_j>上的权值表示从村庄 i 到村庄 j 的道路长度。现在要从这 n 个村庄中选择一个村庄新建一所医院，问这所医院应建在哪个村庄，才能使所有的村庄离医院都比较近？

二、基本要求

- 建立数据模型，设计存储结构。
- 设计算法完成问题求解。
- 分析算法的时间复杂度。

三、设计思想

总的设计思想：医院选址问题实际上是求有向图的中心点。

首先定义顶点的偏心度。设图 G=(V,E)，对任一顶点 k，称 E(k)=max{dist<i,k>}(i∈V, dist<i,k>为顶点 i 到顶点 k 的代价)为顶点 k 的偏心度。显然，偏心度最小的顶点即为图 G 的中心点。

四、设计编码（注：实验编码格式为 UTF-8/GB2312）

1.MGraph.h

```
#ifndef MGraph_H
#define MGraph_H
#include <string>
using namespace std;
int const MaxSize = 10;

class MGraph
{
    public:
        MGraph(string a[], int n, int e);
        ~MGraph();
        friend void Floyd(MGraph G);
    private:
        string vertex[MaxSize];
        int arc[MaxSize][MaxSize];
        int vertexNum, arcNum;
};

void Floyd(MGraph G);

#endif
```

2. MGraph.cpp

```
#include <iostream>
#include <string>
#include <iomanip>
#include "MGraph.h"
using namespace std;

extern int visited[];

MGraph::MGraph(string a[], int n, int e)
{
    int i, j, k;
    vertexNum = n, arcNum = e;
    for ( i = 0; i < n; i++)
        vertex[i] = a[i];
    for ( i = 0; i < n; i++)
        for ( j = 0; j < n; j++)
            arc[i][j] = 1000;
    for ( i = 0; i < n; i++)
        arc[i][i] = 0;
    for ( k = 0; k < arcNum; k++)
```

```
    {
        cout<<"please enter the 2 points and weight:";
        cin>>i>>j;
        cin>>arc[i][j];
    }
}

MGraph::~MGraph()
{
    //empty
}

void Floyd(MGraph G)
{
    int n = G.vertexNum;
    int dist[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            dist[i][j] = G.arc[i][j];
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (dist[i][j] > dist[i][k] + dist[k][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
    int E[n];
    for (int k = 0; k < n; k++)
    {
        E[k] = dist[0][k];
        for (int j = 0; j < n; j++)
            if (dist[j][k] > E[k])  E[k] = dist[j][k];
    }
    cout<<"Point  Eccentricity"<<endl;
    for(int i = 0; i < n; i++)
        cout<<setw(3)<<G.vertex[i]<<setw(10)<<E[i]<<endl;
    int p = 0;
    for(int j = 1; j < n; j++)
        if(E[j] < E[p]) p=j;
    cout<<"The location of the hospital is "<<G.vertex[p]<<endl;
}
```

3. MGraph_main.cpp

```
#include <iostream>
using namespace std;
```

```cpp
#include "MGraph.cpp"

int main()
{
    string ch[ ]={"A", "B", "C", "D", "E"};
    cout<<"enter the amount of point and edge:";
    int n, e;
    cin>>n>>e;
    MGraph G(ch, n, e);
    Floyd(G);
    return 0;
}
```

五、运行结果

```
enter the amount of point and edge:5 7
please enter the 2 points and weight:0 1 1
please enter the 2 points and weight:1 2 2
please enter the 2 points and weight:2 3 2
please enter the 2 points and weight:2 4 4
please enter the 2 points and weight:3 1 1
please enter the 2 points and weight:3 2 3
  A      1000
  B         6
  C         8
  D         5
  E         7
The location of the hospital is D
```

六、总结与心得

这次的实验主要围绕图的性质以及他们的深度遍历和广度遍历实现，加入了综合实验的应用，需要在设计算法的时候就构思好数据结构和它们的用法，以及对于辅助顶点的运用。每一个实验项目的头文件和函数文件大同小异，主要根据实验目的进行函数的补充和删除，但是自己编程实现这些带有特定目的的函数，还是有一定的难度。

另外，本章节的实验对于调试的能力要求较高，需要自己对调试功能进行再次的熟悉和学习。