Lab3 实验报告

根据 lab3 评分细则中的样例输入，如图：

```
请输入无向网的顶点数和边数：
5 8
顶点、边输入完成
请输入顶点信息：
a b c d e
请输入边的信息：
a b 5
c a 12
a d 5
b e 8
c b 9
b d 7
e c 16
d e 8
```

一．图的创建

1. 邻接矩阵

```
void CreateUDN(AMGraph& G)
{
    cout << "请输入无向网的顶点数和边数：\n";
    int i = 0,j=0,k=0;
    char c = 0;
    cin >> G.vexnum;
    cin >> G.arcnum;
    cout << "顶点、边输入完成\n";
    cout << "请输入顶点信息：\n";
    for (i = 0;i < G.vexnum;i++)
    {
        cin >> G.vexs[i];
        c=getchar();
    }//输入顶点信息
    for (i = 0;i < G.vexnum;i++)
        for (j = 0;j < G.vexnum;j++)
            G.arcs[i][j] = 0;//边权值全部初始化为0
    cout << "请输入边的信息：\n";
    for (k = 0;k < G.arcnum;k++)
    {
        VerTexType v1, v2;
        int w;
        cin >> v1 >> v2 >> w;
        i = LocateVexUDN(G, v1);
        j = LocateVexUDN(G, v2);
        G.arcs[i][j] = w;
        G.arcs[j][i] = G.arcs[i][j];//赋值给边权
    }
    cout << "邻接矩阵创建完成！\n";
}//创建无向图的邻接矩阵
```

```
int LocateVexUDN(AMGraph& G, VerTexType v)
{
    int i = 0;
    for (i = 0;i < G.vexnum;i++)
    {
        if (G.vexs[i] == v)
            return i;
    }
    if (i == G.vexnum)
    {
        cout << "该点不是图中一点";
        return MaxInt;
    }
    else return MVNum;
}//找到邻接矩阵中顶点的对应编号
```

```
void printUDN(AMGraph& G)
{
    for (int i = 0;i < G.vexnum;i++)
    {
        for (int j = 0;j < G.vexnum;j++)
        {
            cout << G.arcs[i][j];
            cout << ' ';
        }
        cout << '\n';
    }
}//输出邻接矩阵
```

对于输入的边的信息，找到对应端点在所有点中的序号，将对应 n*n 的矩阵中的两个对应元素都赋值为边权值，记录完所有边以后再输出矩阵，结果如下:

```
邻接矩阵创建完成!
0 5 12 5 0
5 0 9 7 8
12 9 0 0 16
5 7 0 0 8
0 8 16 8 0
```

2. 邻接表

创建边节点和表头结点表的结构体，把输入的每条边的两个顶点 A 和 B 的第一个边结点分别修改为另一个结点 B（A)。把对应的边权值也存入边结点的相关信息中。

```
int LocateVexUDG(ALGraph G, VerTexType v)
{
    int i = 0;
    for (i = 0;i < G.vexnum;i++)
    {
        if (G.vertices[i].data == v)
            return i;
    }
    if (i == G.vexnum)
    {
        cout << "该点不是图中一点";
        return MaxInt;
    }
    else return MVNum;
}//找到邻接表中对应的点
void CreateUDG(ALGraph& G)
{
    cout << "请输入无向网的顶点数和边数：\n";
    int i = 0, j = 0, k = 0;
    VerTexType v1, v2;
    cin >> G.vexnum >> G.arcnum;//输入总顶点数，总边数
    cout << "顶点、边输入完成\n";
    cout << "请输入顶点的信息：\n";
    for (i = 0;i < G.vexnum;i++)
    {
        cin >> G.vertices[i].data;
        G.vertices[i].firstarc = NULL;
    }
```

```
    cout << "请输入边的信息，两顶点和边权值：\n";
    for (k = 0;k < G.arcnum;k++)
    {
        int w;
        cin >> v1 >> v2>>w ;//输入一条边依附的两个顶点
        i = LocateVexUDG(G, v1);
        j = LocateVexUDG(G, v2);
        ArcNode* p1;
        ArcNode* p2;//生成两个新的边节点
        p1 = new ArcNode;
        p1->adjvex = G.vertices[j].data;
        p1->info = w;
        p1->nextarc = G.vertices[i].firstarc;
        G.vertices[i].firstarc = p1;//新节点*p1插入顶点vi的边表头部
        p2 = new ArcNode;
        p2->adjvex = G.vertices[i].data;
        p2->info = w;
        p2->nextarc = G.vertices[j].firstarc;
        G.vertices[j].firstarc = p2;//新节点*p2插入顶点vj的边表头部
    }
    cout << "邻接表创建完成！\n";
}
```

```
void printUDG(ALGraph& G)
{
    for (int i = 0;i < G.vexnum;i++)
    {
        printf("%c ->", G.vertices[i].data);
        while (1)
        {
            if (G.vertices[i].firstarc == NULL)
            {
                printf("END");
                break;
            }
            else
            {
                printf(" %c %d ->", G.vertices[i].firstarc->adjvex, G.vertices[i].firstarc->info);
                G.vertices[i].firstarc = G.vertices[i].firstarc->nextarc;//依次输出边表中的每个点及对应边的权值
            }
        }
        cout<<"\n";
    }
}
```

二．图的遍历

1. 非递归的深度优先遍历

因为深度优先需要先进后出，所以用栈作为储存结构

具体过程如下:第一个顶点先入栈，然后进入 while 循环（循环条件为栈不空），循环中 pop 栈顶元素，遍历邻接矩阵，找到第一个与这个元素相邻且没有被访问到的节点，输出这个点、标记此点为已遍历并入栈，如果遍历了所有的点都没有找到满足条件的点，则把这个元素出栈并进入下一个 while 循环；否则，直接进入下一个 while 循环

```
void DFS_Notrecursion(AMGraph G, VerTexType v0)
{
    int v = LocateVexUDN(G, v0);
    int s;
    InitStack(&S);
    int visited[MVNum] = { 0 };//记录是否访问
    visited[v] = 1;//标记已访问结点
    Push(&S, v);
    cout << v0;
    while (StackEmpty(&S) == 0)
    {
        int i, k;
        Pop(&S, &k);
        //cout << k;
        for (i = 0;i < G.vexnum;i++)
            if (G.arcs[k][i] != 0 && visited[i] != 1)//没有访问过且存在弧
            {
                cout << '-';
                cout << '>';
                cout << G.vexs[i];
                Push(&S, i);//访问并入栈
                visited[i] = 1;
                break;
            }//把所有与k有边连接的点都访问
        if (i == G.vexnum)
            Pop(&S, &k);//弹出所有邻接结点都已访问过的点
    }
    cout << '\n';
}//非递归的深度优先遍历
```

```
请输入非递归深度优先遍历的第一个节点：b
b->a->c->e->d
```

2. 广度优先遍历

因为广度优先需要先进先出，所以用队列作为储存结构

队列非空的时候，一个元素出队。对于这个元素，遍历，找到第一个相邻节点，直到对于这一个节点来说的最后一个相邻节点，依次判断是否被访问过，若没有被访问，则输出，并按照顺序放进队列中（因为先遍历到的节点的相邻节点要先输出，符合先进先出）。

```
void BFS(AMGraph G, int v)
{
    int n = G.vexnum;
    int visited[MVNum] = { 0 };
    cout << G.vexs[v];
    visited[v] = 1;
    SqQueue Q;
    InitQueue(Q);
    EnQueue(Q, v);//队列的一系列操作
    while (!QueueEmpty(Q))
    {
        int u=DeQueue(Q), w;
        for (w = FirstAdjVex(G, u);w >= 0;w = NextAdjvex(G, u, w))
        {//找到第一个相邻节点，直到对于这一个节点来说的最后一个相邻节点，依次输出，并按照顺序放进队列中
            if (w == -1)
                break;
            if (!visited[w])
            {
                cout << '-';
                cout << '>';
                cout << G.vexs[w];
                visited[w] = 1;
                EnQueue(Q, w);
            }//若没访问过就输出并入队
        }
    }
}//广度优先遍历
```

```
请输入广度优先遍历的第一个节点：c
广度优先遍历的结果为：
c->a->b->e->d
```

三．Prim 算法求最小生成树

构造结构体 close[]，用于对于目前没有加入到生成树中的顶点，储存它们各自与已经加入到生成树中的顶点之间的最小边以及这些边的另一个端点。每次生成树中新加入一个点，都需要更新对应的最短边，并把新加入点的最短边赋值为 0。每次循环都找出现存的非最短边中最短的一个，输出所属的顶点，直至循环次数等于总的顶点数为止。

```
int Min(closedge close[], AMGraph G)
{
    int min = 0,k=0;
    for (int i = 0;i < G. vexnum;i++)
        if (close[i]->lowcost)
        {
            if (min == 0 || min > close[i]->lowcost)
            {
                min = close[i]->lowcost;
                k = i;
            }
        }
    return k;
}//找到对于每个顶点来说最短边的集合中最小的那一条，并且输出这条边的两个端点中没有被遍历过的那个
```

```
void MiniSpanTree_Prim(AMGraph G, VerTexType u)
{
    int k,j,i;
    int erase[MVNum][MVNum] = { 0 };
    k = LocateVexUDN(G, u);
    closedge close[MVNum];
    for(i=0;i<G.vexnum;i++)
    for (j = 0;j < G.vexnum;++j)
    {
        erase[i][j] = G.arcs[i][j];
    }
    for (i = 0;i < G.vexnum;i++)
        for (j = 0;j < G.vexnum;++j)
        {
            if(erase[i][j]==0)
            erase[i][j] = MaxInt;
        }//把0替换成无穷
    for (j = 0;j < G.vexnum;++j)
    {
        if (j != k)
        {
            close[j]->adjvex = u;
            close[j]->lowcost=erase[k][j];
        }//只遍历过一个顶点时，最短边就是它到每个顶点的弧
    }
```

```
        close[k]->lowcost = 0;//初始时只有一个顶点，它到自己的边为0，目前累计的权值为0
        for (i = 1;i < G.vexnum;++i)
        {
            k = Min(close,G);
            VerTexType u0, v0;
            u0 = close[k]->adjvex;
            v0 = G.vexs[k];
            cout << u0;
            cout << ' ';
            cout << v0;
            cout << ' ';
            cout<<close[k]->lowcost;
            cout << '\n';//输出对应顶点和弧长
            close[k]->lowcost = 0;
            for (j = 0;j < G.vexnum;++j)
                if (erase[k][j] < close[j]->lowcost)
                {
                    close[j]->adjvex = G.vexs[k];
                    close[j]->lowcost = erase[k][j];
                }//更新最短边
        }
//最小生成树
```

```
请输入prim算法的第一个节点：d
d a 5
a b 5
d e 8
b c 9
```

四．图的最短路径

分别用数组 D[]、S[]、Path[]存储当前已知的最短路径长度、标记最短路径是否已经被确定、最短路径的前驱（即前一个顶点）

对于与输入顶点直接相连的点，最短路径即是输入顶点和该点之间的边权值，前驱为输入顶点，对于的 S 都赋值为 true；对于其他点，把当前的最短路径都记为 MaxInt。

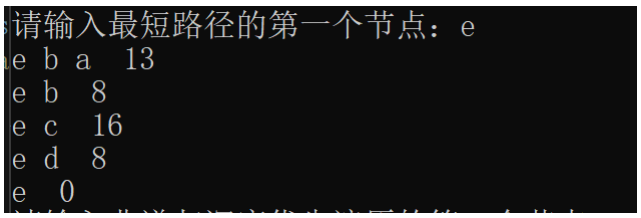
对于还没有确定最短路径的点 A1、A2……，循环对比当前记录的最短路径和（已经确定最短路径的点的最短路径长加两点之间距离），如果后者更小，则把 Ai 的当前最短路径更新为这个更小的值，并更改前驱。

完成所有的最短路径记录后，用 FindPath 函数递归输出所有最短路径上的节点。（其实就是因为 Path 数组储存的是前驱，所以要倒过来使用 Path，于是想到了递归输出。）

```
void FindPath(AMGraph G, int m, int v1,int Path[])
{
    if (m == v1)
        return;
    if (Path[m] != v1)
        FindPath(G,Path[m],v1,Path);
    cout << G.vexs[m];
    cout << ' ';
}
void ShortestPath_DIJ(AMGraph G, VerTexType v0)
{
    int v1 = LocateVexUDN(G, v0);
    int k, j, i;
    int erase[MVNum][MVNum] = { 0 };
    for (i = 0;i < G.vexnum;i++)
        for (j = 0;j < G.vexnum;++j)
        {
            erase[i][j] = G.arcs[i][j];
        }
    for (i = 0;i < G.vexnum;i++)
        for (j = 0;j < G.vexnum;++j)
        {
            if (erase[i][j] == 0)
                erase[i][j] = MaxInt;
        }
```

```
    int m = erase[1][0];
    int v = 0,n,n1;
    n = G.vexnum;
    int w = 0;
    int D[MVNum]={0}, Path[MVNum] = {0}, S[MVNum] = {0};//S记录从源点v0到终点vi最短
    for (v = 0;v < n;++v)//n个顶点依次初始化
    {
        S[v] = false;
        n1 = erase[v1][v];
        D[v] = n1;
        if (D[v] < MaxInt)
            Path[v] = v1;
        else
            Path[v] = -1;
    }//如果顶点v0和vi之间有弧，则将vi的前驱置为v0，否则数组存-1
    S[v1] = true;//true-已经找到最短路径
    D[v1] = 0;
    for (i = 1;i < n;i++)
    {
        int min = MaxInt;
        for(w=0;w<n;++w)
            if (!S[w] && D[w] < min)
            {
                v = w;
                min = D[w];
            }//选择当前最短的一条路径
```

```
        S[v] = true;
        for(w=0;w<n;++w)
            if (!S[w] && (D[v] + erase[v][w] < D[w]))
            {
                D[w] = D[v] + erase[v][w];
                Path[w] = v;
            }//更新D[w]，更改前驱
    }
    for (w = 0;w < n;w++)
    {
        int i=w,post[MVNum],k=0;
        cout << v0;
        cout << ' ';
        FindPath(G, w, v1, Path);
        //cout << G.vexs[w];
        cout << ' ';
        cout << D[w];
        cout << '\n';
    }
}//求最短路径
```


请输入最短路径的第一个节点：e
e b a 13
e b 8
e c 16
e d 8
e 0