图像处理

21307077

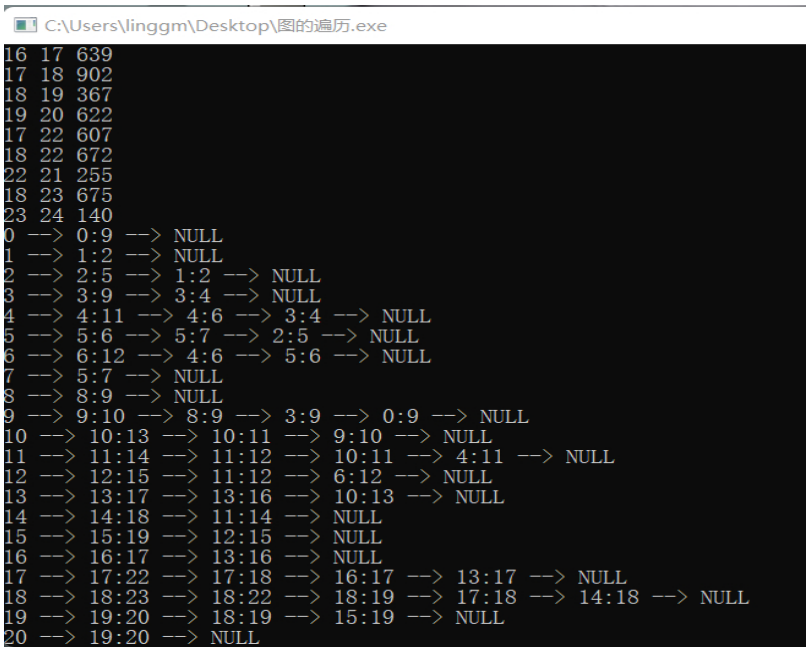
凌国明

程序功能说明

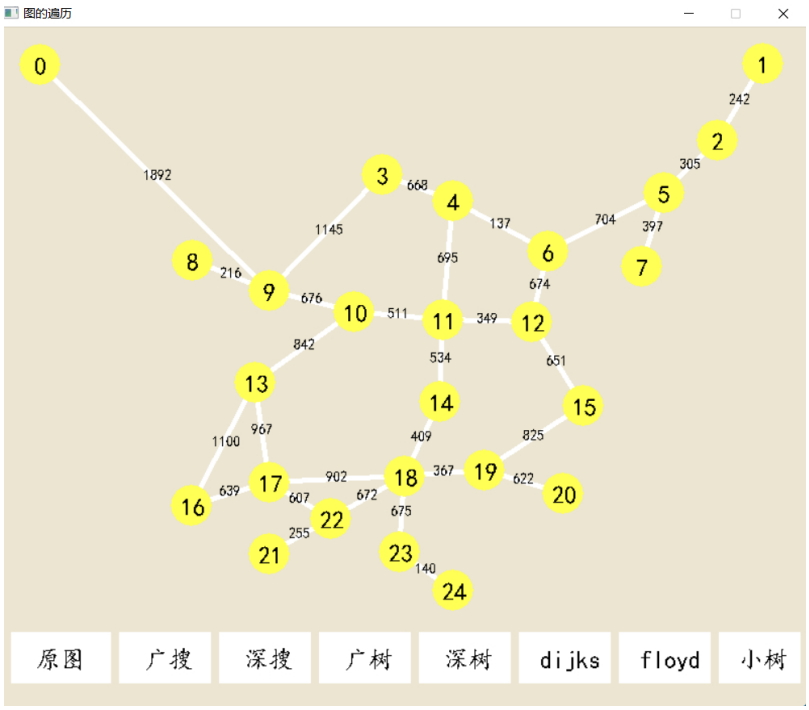
1. 以**邻接多重表**为存储结构，实现连通无向图的输入，插入等操作。
2. 借助于队列类型，用非递归算法实现**广度优先遍历**。
3. 借助于栈类型，用**非递归**算法实现**深度优先遍历**。
4. 以邻接表为存储结构，建立**深度优先生成树和广度优先生成树**，并以树形输出生成树。
5. 利用 $dijkstra$ 算法求某顶点到其他所有顶点的**最短路径**及其距离。
6. 利用 $floyd$ 算法求任意顶点到其他所有顶点的距离。
7. 利用 $kruskal$ 算法求**最小生成树**。
8. 以上功能均通过 $easyx$ 库**可视化实现**，通过每个步骤之间插入 $0.2s$ 的睡眠时间达到**过程可视化**的效果。

程序运行展示

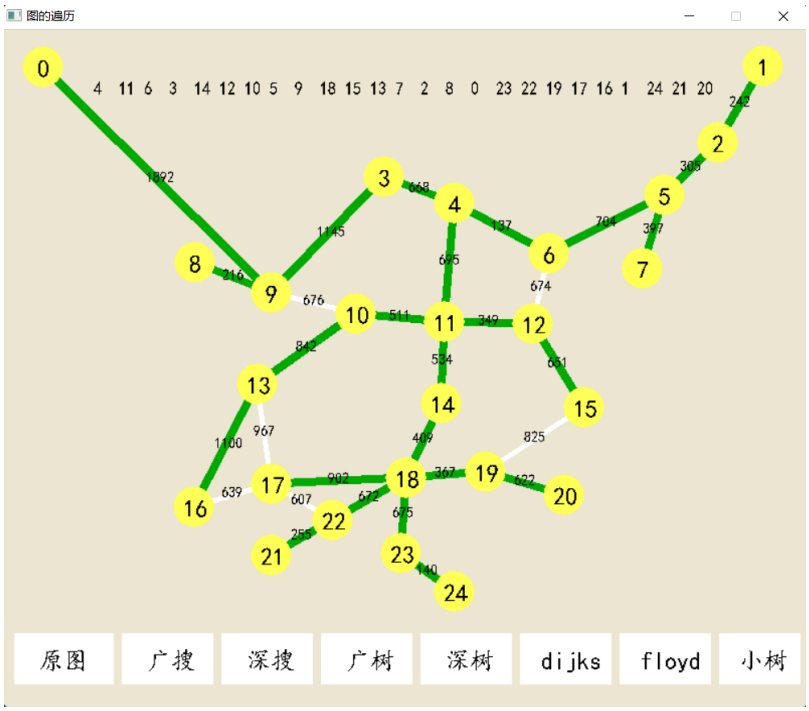
输入数据



原图展示

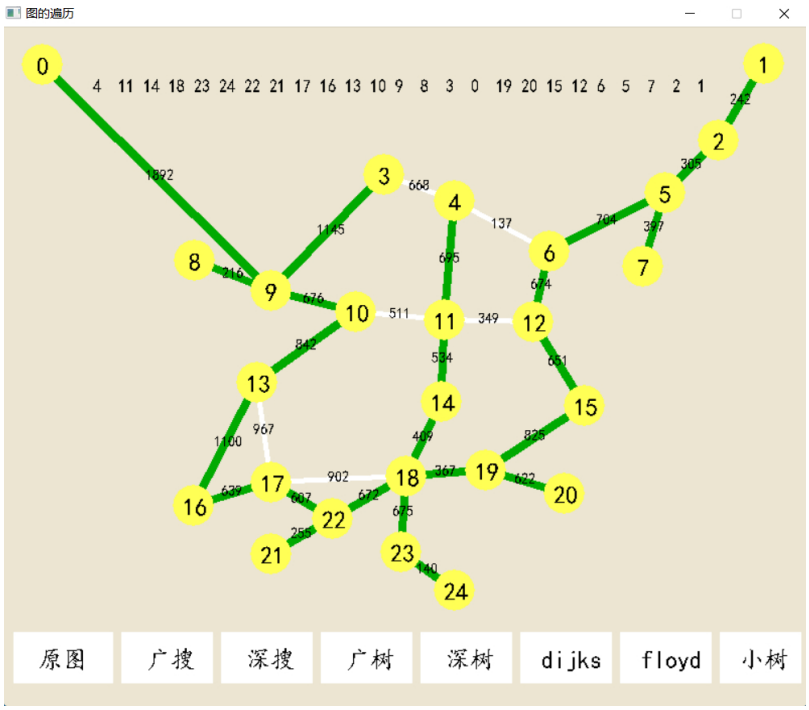


广度优先搜索



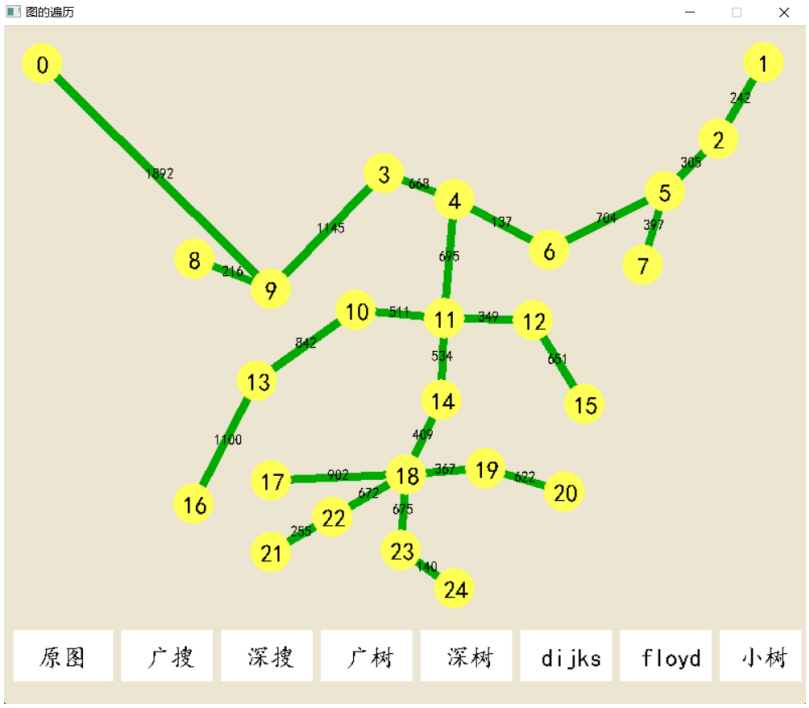
点击对应按钮，输入起点，开始搜索。

深度优先搜索

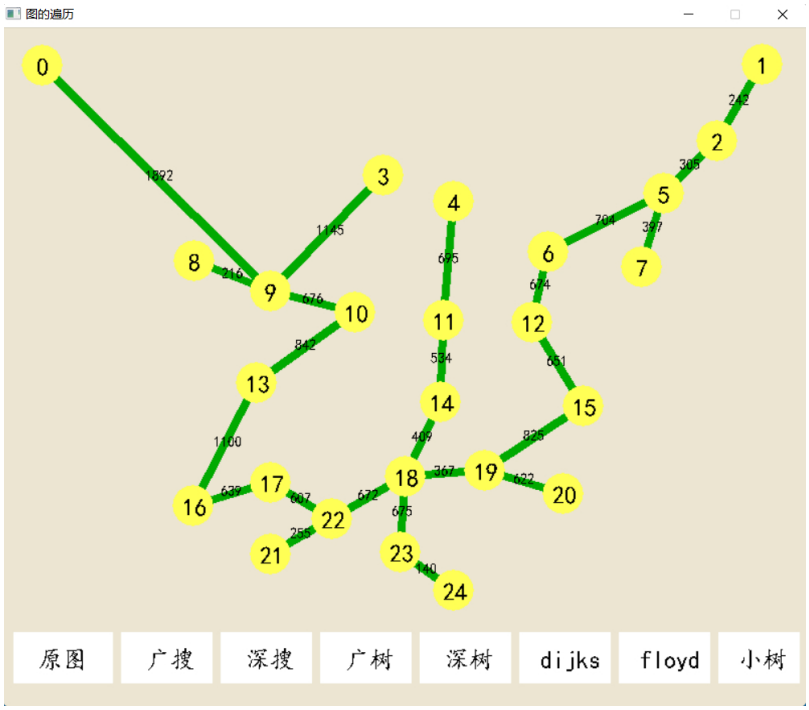


点击对应按钮，输入起点，开始搜索。

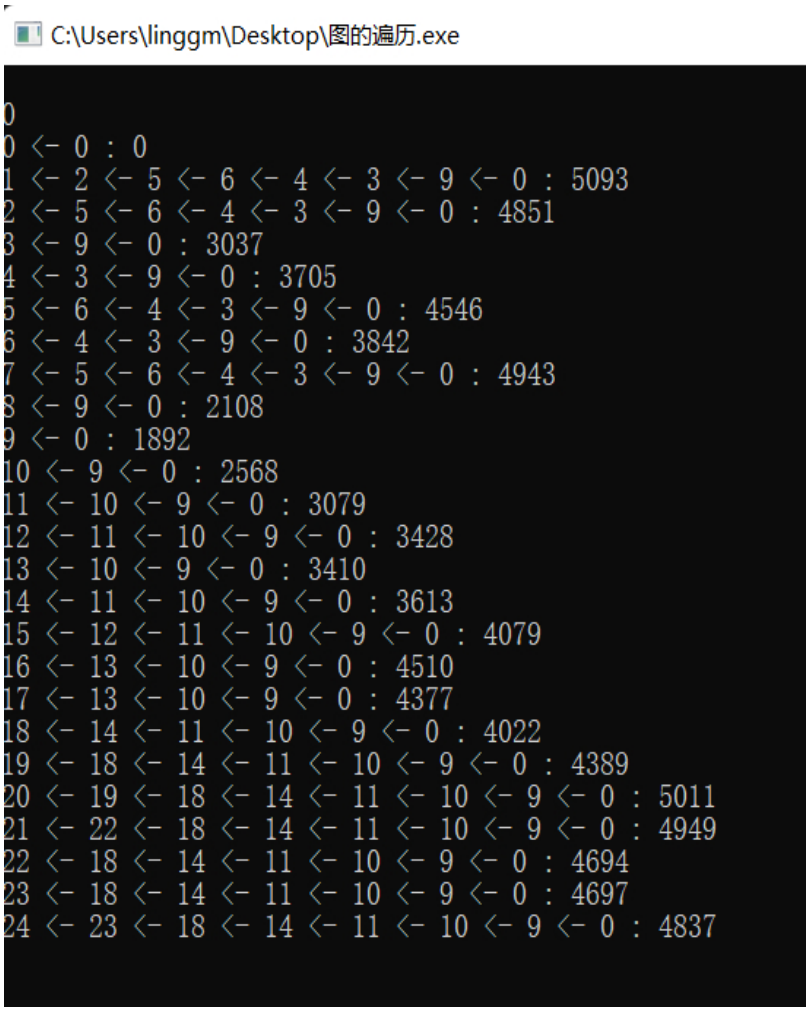
广度优先生成树

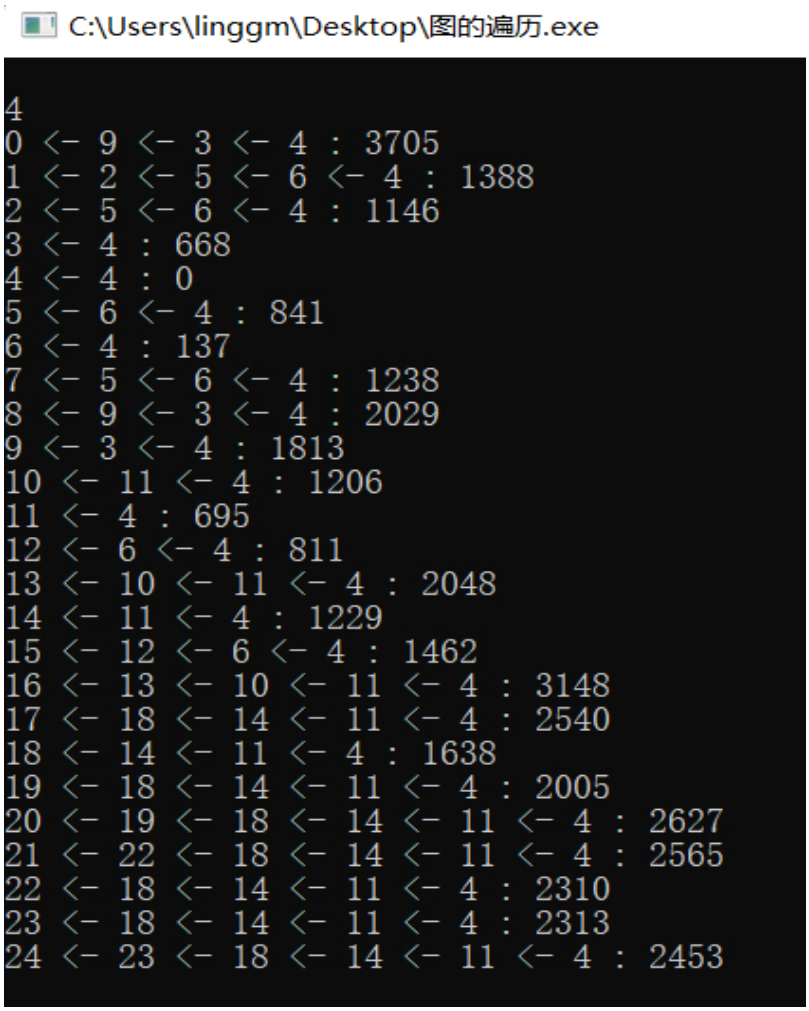


深度优先生成树



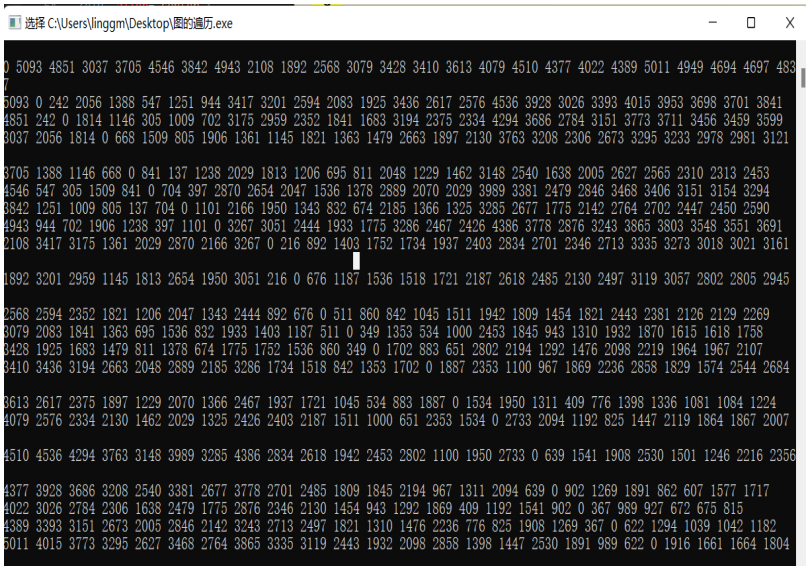
dijkstra



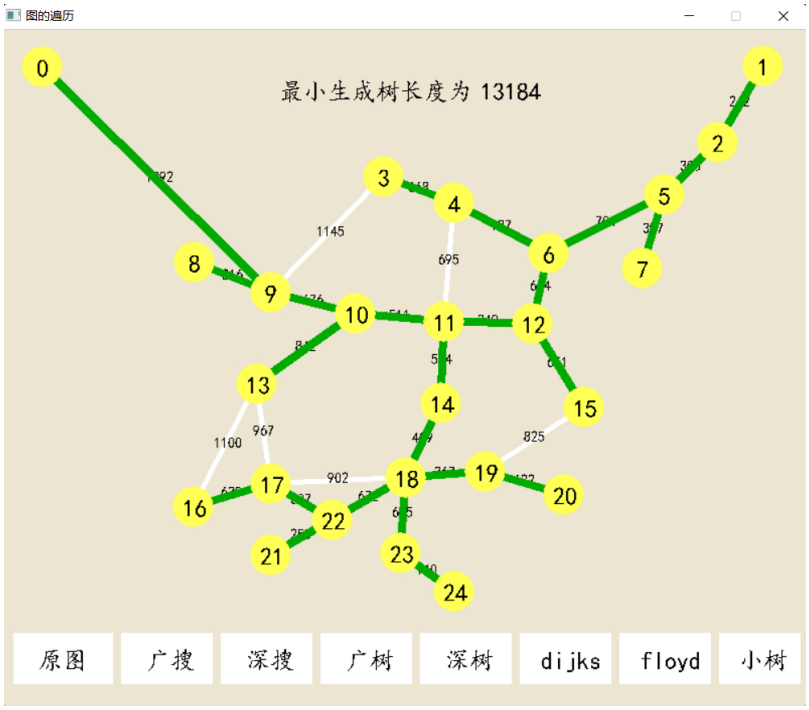


点击对应按钮，输入起点，开始搜索。

floyd



kruskal


图的遍历
最小生成树长度为 13184
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
1145
695
1100
967
902
825
原图
广搜
深搜
广树
深树
dijks
floyd
小树

部分关键代码及其说明

存储结构

```
template <typename T>
struct EdgeNode{
        bool mark; //是否被处理过
        T vtx1;
        T vtx2;
        EdgeNode<T> *path1; //与vtx1关联的下一条边
        EdgeNode<T> *path2; //与vtx2关联的下一条边
        int weight; //权重
        EdgeNode(T v1, T v2): vtx1(v1), vtx2(v2), path1(nullptr), path2(nullptr), weight(0), mar
        EdgeNode(T v1, T v2, int w): vtx1(v1), vtx2(v2), path1(nullptr), path2(nullptr), weight(
};

template <typename T>
struct VertexNode{
        T data;
        int x, y;
        EdgeNode<T> *fout; //第一条边
        VertexNode(): data(0), x(0), y(0), fout(nullptr){}
        VertexNode(T data): data(data), x(0), y(0), fout(nullptr){}
        VertexNode(T data, EdgeNode<T> *n): data(data), x(0), y(0), fout(n){}
};

template <typename T>
struct Graph{
        vector<vector<int>> adj; //邻接矩阵
    vector<vector<int>> adjlist; //邻接表
        VertexNode<T> **node; // 邻接多重表
        int order; //顶点数量
        int size; //边的数量

        Graph(int o, int s): order(o), size(s){
                node = new VertexNode<T>*[o];
                adj.resize(o);
        adjlist.resize(o);
                for(int i = 0; i < order; i++){
                        node[i] = new VertexNode<T>();
                        adj[i].resize(o, INF);
                        adj[i][i] = 0;
                }
    }
};
```

每个部分代表什么含义，在注释上有说明。

广度优先搜索部分代码

```
void bfs(T start){
    // 边框线条颜色函数
    setlinecolor(GREEN);
    // 边框线条形式函数
    setlinestyle(PS_SOLID, 8);
    if(start < 0 || start >= order)
        return;
    queue<T> que;
    vector<bool> visit(order, false);
    que.push(start);
    visit[start] = true;
    int counter = 0;
    settextstyle(18, 8, "楷体");
    while(!que.empty()){
        T t = que.front();
        cout << t << " ";
        outtextxy(90+25*counter++, 50, to_string(t).c_str() );
        que.pop();
        EdgeNode<T> *tmp = node[t]->fout;
        while(tmp != nullptr){

            if(tmp->vtx1 == t){
                if(!visit[tmp->vtx2]){
                    visit[tmp->vtx2] = true;
                    tmp->mark = true;
                    line(node[tmp->vtx1]->x, node[tmp->vtx1]->y, node[tmp->vtx2]->x, node[tmp->v
                    Sleep(200);
                    que.push(tmp->vtx2);
                }
                tmp = tmp->path1;
            }
            else{
                if(!visit[tmp->vtx1]){
                    visit[tmp->vtx1] = true;
                    tmp->mark = true;
                    line(node[tmp->vtx1]->x, node[tmp->vtx1]->y, node[tmp->vtx2]->x, node[tmp->v
                    Sleep(200);
                    que.push(tmp->vtx1);
                }
                tmp = tmp->path2;
            }
        }
    }
    cout << endl;
}
```

利用队列结构实现图的广度优先搜索（存储结构为邻接多重表）

深度优先搜索部分代码

```
void dfs(T start){
    // 边框线条颜色函数
    setlinecolor(GREEN);
    // 边框线条形式函数
    setlinestyle(PS_SOLID, 8);
    if(start < 0 || start >= order)
        return;
    stack<T> sta;
    vector<bool> visit(order, false);
    vector<EdgeNode<T>*> rec(order);
    for(int i = 0; i < order; i++){
        rec[i] = node[i]->fout;
    }
    sta.push(start);
    cout << start << ' ';
    visit[start] = true;
    int counter = 0;
    settextstyle(18, 8, "楷体");
    outtextxy(90+25*counter++, 50, to_string(start).c_str() );
    while(!sta.empty() ){
        T t = sta.top();
        EdgeNode<T> *tmp = rec[t];
        if(tmp == nullptr){
            sta.pop();
        }
        else{
            if(tmp->vtx1 == t){
                if(!visit[tmp->vtx2]){
                    visit[tmp->vtx2] = true;
                    tmp->mark = true;
                    sta.push(tmp->vtx2);
                    cout << tmp->vtx2 << ' ';
                    line(node[tmp->vtx1]->x, node[tmp->vtx1]->y, node[tmp->vtx2]->x, node[tmp->v
                    Sleep(200);
                    outtextxy(90+25*counter++, 50, to_string(tmp->vtx2).c_str() );
                }
                rec[t] = tmp->path1;
            }
            else if(tmp->vtx2 == t){
                if(!visit[tmp->vtx1]){
                    visit[tmp->vtx1] = true;
                    tmp->mark = true;
                    sta.push(tmp->vtx1);
                    cout << tmp->vtx1 << ' ';
                    line(node[tmp->vtx1]->x, node[tmp->vtx1]->y, node[tmp->vtx2]->x, node[tmp->v
                    Sleep(200);
                    outtextxy(90+25*counter++, 50, to_string(tmp->vtx1).c_str() );
                }
```

```
                rec[t] = tmp->path2;
            }
        }
    }
    cout << endl;
}
```

利用栈结构实现图的深度优先搜索（存储结构为邻接多重表）

dijkstra算法

```
void dijkstra(int start){
    if(start < 0 || start >= order)
        return;
    vector<int> dist(order, INF);
    vector<int> pre(order, -1);
    vector<bool> visit(order, false);
    visit[start] = true;
    for(int i = 0; i < order; i++){
        dist[i] = adj[start][i];
        if(dist[i] < INF && dist[i] != 0)
            pre[i] = start;
    }
    for(int e = 1; e < order; e++){
        int minn = INF, min_pos = -1;
        for(int i = 0; i < order; i++){
            if(!visit[i] && dist[i] < minn){
                minn = dist[i];
                min_pos = i;
            }
        }
        if(min_pos == -1)
            break;
        visit[min_pos] = true;
        for(int i = 0; i < order; i++){
            if((long long)dist[min_pos] + adj[min_pos][i] < dist[i]){
                dist[i] = dist[min_pos] + adj[min_pos][i];
                pre[i] = min_pos;
            }
        }
    }
    for(int i = 0; i < order; i++){
        int tmp = pre[i];
        cout << i;
        if(tmp == -1){
            cout << " <- " << start;
        }
        while(tmp != -1){
            cout << " <- " << tmp;
            tmp = pre[tmp];
        }
        if(dist[i] == INF)
            cout << " : " << "INF" << endl;
        else
            cout << " : " << dist[i] << endl;
    }
}
```

利用 $dijkstra$ 实现图的单源点最短路径搜索（存储结构为邻接矩阵）

floyd算法

```
void floyd(){
    int **dist = new int*[order];
    for(int i = 0; i < order; i++){
        dist[i] = new int[order];
        for(int j = 0; j < order; j++){
            dist[i][j] = adj[i][j];
        }
    }
    for(int e = 0; e < order; e++){
        for(int i = 0; i < order; i++){
            for(int j = 0; j < order; j++){
                if((long long)dist[i][e] + dist[e][j] < dist[i][j]){
                    dist[i][j] = dist[i][e] + dist[e][j];
                }
            }
        }
    }
    for(int i = 0; i < order; i++){
        for(int j = 0; j < order; j++){
            cout << dist[i][j] << " ";
        }
        cout << endl;
    }
    for(int i = 0; i < order; i++){
        delete[] dist[i];
    }
    delete[] dist;
}
```

利用 $floyd$ 实现图的多源点最短路径搜索（存储结构为邻接矩阵）

程序运行方式简要说明

1. 通过 $EasyX$ 库开发图形化交互界面，进行图的可视化展示，**通过插入睡眠时间实现算法过程的可视化。**
2. **鼠标左键按下按键**实现对应功能。如点击 “深度优先搜索” 后，根据要求输入起点，然后**图形窗口逐步展示算法的过程**，最后得出结果。
3. 广度优先搜索通过**队列**实现，深度优先搜索通过**栈**实现，两种搜索都在**邻接多重表**上进行。
4. $dijkstra$ ， $kruskal$ 和 $floyd$ 算法在**邻接矩阵**上进行。
5. 输出对应优先生成树时，在原图的基础上展示。