面向对象程序设计实验4
地图导航模拟算法

屈熙宸

55240425

任务：

- 地图导航模拟算法。在一个无向图中任意给定两点实现最短路径计算。要求采用dijkstra（深度或广度或弗洛伊德或者Bellman-Ford）算法以及堆排序算法或其他排序算法，利用递归、vector支持邻接表方式，对于路径和选择路径进行界面显示（可仅仅显示而不交互）

- 分析题目要求:
  - 要求在提供的无向图中利用相关算法，求出最短路径，并且显示

程序源码+运行结果

```cpp
#include<iostream>
#include<queue>
#include<vector>
#include<string>
using namespace std;

#define MAX 999
#define MVNum 20

typedef int VertexType;
typedef int ArcType;
```

You, 5天前 | 1 author (You)

```cpp
struct Edge {
    int dest;   // 目标顶点
    int weight; // 权重
};
```

```cpp
class Graph
{
public:
    void Create();
    int LocateVex(VertexType u);//查找Graph中的顶点u，并返回其对应在顶点表中的下标，未找到则返回-1
    int firstadj(int v);
    int nextadj(int v, int w);
    void Floyd();//Floyd算法
    void print_path();//打印路径
    void BuildAdjList();
    void SortAdjacentEdges();
    void QueryPath(int start, int end);
private:
    VertexType vexs[MVNum];//顶点表
    ArcType arcs[MVNum][MVNum];//邻接矩阵
    ArcType path[MVNum][MVNum];//保存路径
    int vexnum, arcnum;//图当前的顶点数和边数
    vector<vector<Edge>> adjList;
};
```

src > floyd.cpp > firstadj(int)

```cpp
int Graph::LocateVex(VertexType u)
{//查找Graph中的顶点u，并返回其对应在顶点表中的下标，未找到则返回-1
    int i;
    for (i = 0; i < this->vexnum; i++)
    {
        if (u == this->vexs[i])
            return i;
    }
    return -1;
}

int Graph::firstadj(int v)
{       You, 5天前 • 2025.5.6 …
    for (int i = 0; i < this->vexnum; i++)
    {
        if (this->arcs[v][i] != MAX)
            return i;
    }
    return -1;
}
```

```cpp
int Graph::nextadj(int v, int w)
{
    for (int i = w + 1; i < this->vexnum; i++)
    {
        if (this->arcs[v][i] != MAX)
            return i;
    }
    return -1;
}       You, 5天前 • 2025.5.6 …
```

```cpp
void Graph::Create()
{
    cout << "请输入总结点数和总边数:";
    cin >> this->vexnum >> this->arcnum;//输入总顶点数和总边数
    for (int i = 0; i < this->vexnum; i++)
    {
        this->vexs[i] = i + 1;
    }
    //初始化邻接矩阵
    for (int i = 0; i < this->vexnum; i++)
    {
        for (int j = 0; j < this->vexnum; j++)
        {
            if(i == j)
                this->arcs[i][j] = 0;
            else
                this->arcs[i][j] = this->arcs[j][i] = MAX;
        }
    }
    //构造邻接矩阵
    for (int i = 0; i < this->arcnum; i++)
    {
        int v1, v2, w;
        cout << "请输入第" << i + 1 << "条边的两个顶点及其对应的权值:";
        cin >> v1 >> v2 >> w;
        int m = LocateVex(v1);
        int n = LocateVex(v2);
        this->arcs[m][n] = w;
        this->arcs[n][m] = w;
    }
    //初始化路径
    for (int i = 0; i < this->vexnum; i++)
    {
        for (int j = 0; j < this->vexnum; j++)
        {
            this->path[i][j] = j;
        }
    }
    return;
}
```

```
void Graph::print_path()
{
    cout << "各个顶点对的最短路径: " << endl;
    int row = 0;
    int col = 0;
    int temp = 0;
    for (row = 0; row < this->vexnum; row++)
    {
        for (col = row + 1; col < this->vexnum; col++)
        {
            if (this->arcs[row][col] != MAX)
            {
                cout << "v" << to_string(row + 1) << "---" << "v" << to_string(col + 1) << " weight: " << this
                temp = path[row][col];
                //循环输出途径的每条路径。
                while (temp != col)
                {
                    cout << "-->" << "v" << to_string(temp + 1);
                    temp = path[temp][col];
                }
                cout << "-->" << "v" << to_string(col + 1) << endl;
            }
        }
        cout << endl;
    }
}
```

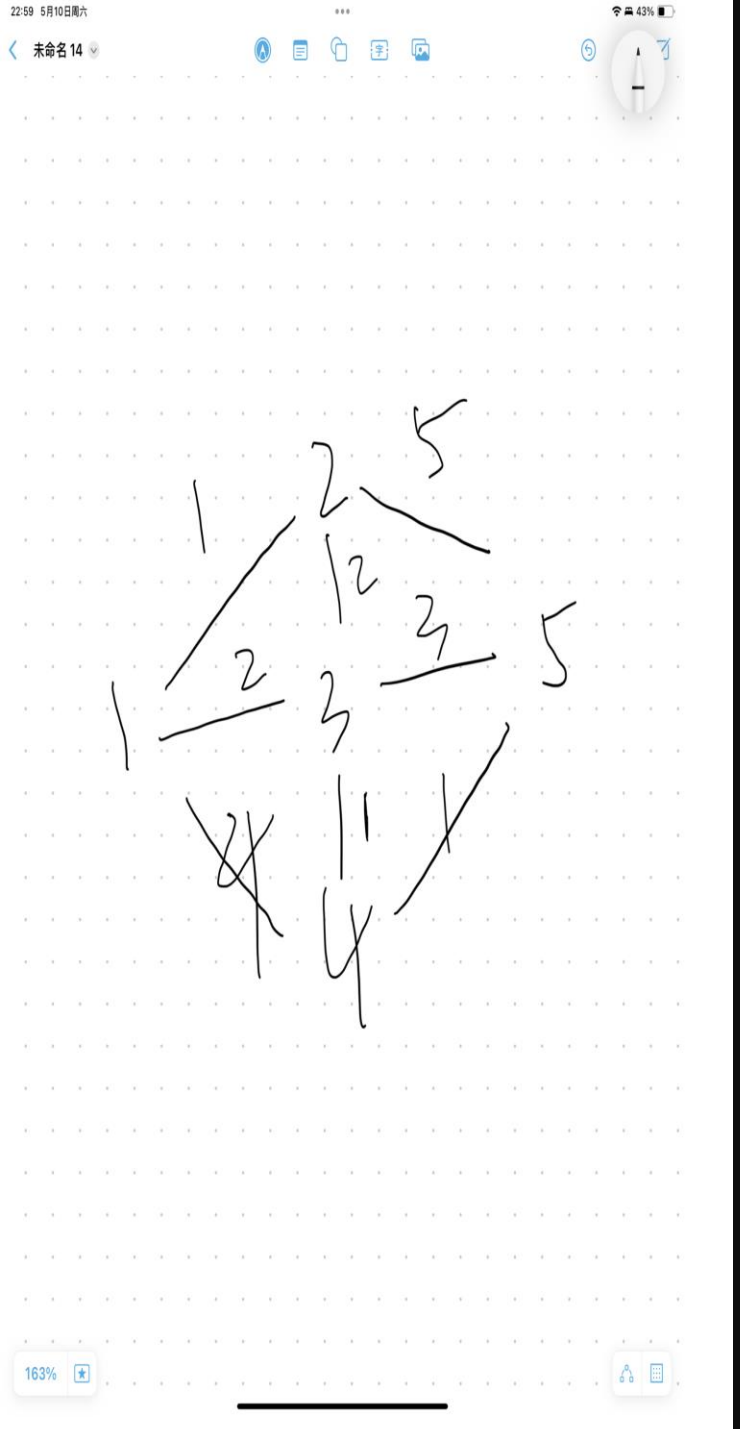
```cpp
void Graph::BuildAdjList() {
    // 初始化邻接表

    adjList.resize(vexnum);

    // 根据邻接矩阵构建邻接表
    for(int i = 0; i < vexnum; i++) {
        for(int j = 0; j < vexnum; j++) {
            if(arcs[i][j] != MAX && i != j) {
                Edge e;
                e.dest = j;
                e.weight = arcs[i][j];
                adjList[i].push_back(e);
            }
        }
    }
}
```

```cpp
void Graph::SortAdjacentEdges() {
    // 为每个顶点的邻接边按权重排序
    for(int i = 0; i < vexnum; i++) {
        if(!adjList[i].empty()) {
            // 提取权重到一个数组中供堆排序使用
            int size = adjList[i].size();
            int* weights = new int[size + 1]; // 堆排序从索引1开始

            // 复制权重
            for(int j = 0; j < size; j++) {
                weights[j + 1] = adjList[i][j].weight;
            }

            // 使用堆排序
            HeapSort(weights, size);

            // 根据排序后的权重重新组织邻接表
            vector<Edge> sortedEdges;
            for(int j = 1; j <= size; j++) {
                // 查找具有当前权重的边
                for(auto it = adjList[i].begin(); it != adjList[i].end(); ++it) {
                    if(it->weight == weights[j]) {
                        sortedEdges.push_back(*it);
                        adjList[i].erase(it);
                        break;
                    }
                }
            }

            // 更新邻接表
            adjList[i] = sortedEdges;

            delete[] weights;
        }
    }
}
```

```cpp
void Graph::QueryPath(int start, int end) {
    // 转换用户输入的顶点编号为内部索引      You, 5天前 • 2025.5.6
    start--;
    end--;

    // 验证输入
    if(start < 0 || start >= vexnum || end < 0 || end >= vexnum) {
        cout << "错误: 顶点编号无效! 有效范围为1至" << vexnum << endl;
        return;
    }

    if(arcs[start][end] == MAX) {
        cout << "从顶点v" << start+1 << "到顶点v" << end+1 << "没有可达路径。" << endl;
        return;
    }

    cout << "\n从顶点v" << start+1 << "到顶点v" << end+1 << "的最短路径:" << endl;
    cout << "距离: " << arcs[start][end] << endl;

    // 构建完整路径
    vector<int> fullPath;
    fullPath.push_back(start);

    int temp = start;
    while(temp != end) {
        temp = path[temp][end];
        fullPath.push_back(temp);
    }

    // 显示路径
    cout << "路径: ";
    for(size_t i = 0; i < fullPath.size(); i++) {
        cout << "v" << fullPath[i]+1;
        if(i < fullPath.size()-1) cout << " → ";
    }
    cout << endl;

    // 显示邻接表路径表示
    cout << "路径上的边:" << endl;
    for(size_t i = 0; i < fullPath.size()-1; i++) {
        int from = fullPath[i];
        int to = fullPath[i+1];
        int weight = arcs[from][to];
        cout << "  v" << from+1 << " → v" << to+1 << " (权重: " << weight << ")" << endl;
    }
```

```cpp
#include "heapsort.h"
#include <algorithm>
#include <iostream>

void HeapAdjust(int *L, int s, int m){
    int temp,j;
    temp = L[s];
    for(j = 2*s; j <= m; j*=2){
        if(j < m && L[j] < L[j+1])
            ++j;
        if(temp >= L[j])
            break;
        L[s] = L[j];
        s = j;
    }
    L[s] = temp;
}

void HeapSort(int *L, int x){        You, 5天前 • 2025.5.6
    int i;
    for(i = x/2; i > 0; i--)
        HeapAdjust(L,i,x);
    for(i = x; i > 1; i--){
        std::swap(L[1], L[i]);
        HeapAdjust(L,1,i-1);
    }
}
```

```cpp
int main() {
    Graph g;

    g.Create();

    g.BuildAdjList();

    g.SortAdjacentEdges();

    g.Floyd();

    g.print_path();

    int choice = 1;
    while(choice) {
        int start, end;
        cout << "\n请输入要查询的起点和终点 (顶点编号): ";
        cin >> start >> end;

        g.QueryPath(start, end);

        cout << "\n是否继续查询? (1:是, 0:否): ";
        cin >> choice;
    }


    system("pause");

    return 0;
}
```



```
请输入总结点数和总边数:5
8
请输入第1条边的两个顶点及其对应的权值:1 2 1
请输入第2条边的两个顶点及其对应的权值:1 3 2
请输入第3条边的两个顶点及其对应的权值:1 4 4
请输入第4条边的两个顶点及其对应的权值:2 5 5
请输入第5条边的两个顶点及其对应的权值:3 5 3
请输入第6条边的两个顶点及其对应的权值:4 5 1
请输入第7条边的两个顶点及其对应的权值:2 3 2
请输入第8条边的两个顶点及其对应的权值:3 4 1
各个顶点对的最短路径:
v1---v2 weight: 1 path:  v1-->v2
v1---v3 weight: 2 path:  v1-->v3
v1---v4 weight: 3 path:  v1-->v3-->v4
v1---v5 weight: 4 path:  v1-->v3-->v4-->v5

v2---v3 weight: 2 path:  v2-->v3
v2---v4 weight: 3 path:  v2-->v3-->v4
v2---v5 weight: 4 path:  v2-->v3-->v4-->v5

v3---v4 weight: 1 path:  v3-->v4
v3---v5 weight: 2 path:  v3-->v4-->v5

v4---v5 weight: 1 path:  v4-->v5



请输入要查询的起点和终点（顶点编号）: 2 4

从顶点v2到顶点v4的最短路径:
距离: 3
路径: v2 → v3 → v4
路径上的边:
  v2 → v3 (权重: 2)
  v3 → v4 (权重: 1)

是否继续查询? (1:是, 0:否): 0
Press any key to continue . . .
```

3.分析优缺点

- 优点:
- 对于要求全部节点之间的最短路径查询，使用floyd算法，提升代码易读性
- 缺点:
- 未能完全实现可视化编程，仅仅使用终端进行简单输出

4.没有完成:

- 未能清晰掌握可视化编程的含义，对于进行可视化编程的手段不足

5.收获

- 切身使用floyd算法进行实践，加深了对无向图以及最短路径算法的理解
- 加深了对堆排序的理解
- 更好的运用vector容器来进行存储
- 更深刻的理解了关于面向对象的思路