《算法设计与分析》

课程实验报告

专业：计算机科学与技术

班级：2021211307

姓名：陈朴炎

学号：2021211138

目录

1 作业内容

1.1 作业题目

作业 2 单元最短路径

- 利用“附件 1-1.基站图的邻接矩阵-v1-23”给出的 LTE 网络基站数据，以基站为顶点，以基站间的距离连线为边，组成图，计算图中的单源最短路径
- 图构造
  - 从昆明 LTE 网络中，选取部分基站，计算基站间的距离，在部分基站间引入边，得到
  - 22 个基站顶点组成的图
  - 42 个基站顶点组成的图

1.2 作业要求

- 对 22 个基站顶点组成的图，以基站 567443 为源点

【说明：可以选择其它顶点作为源点】

  - 1．计算 567443 到其它各点的单源最短路径
  - 2.计算 567443 到 33109 的最短路径

- 对 42 个基站顶点组成的图，以基站 565845 为源点

【说明：可以选择其它顶点作为源点】

  - 1．计算 565845 到其它各点的单源最短路径
  - 2．计算 565845 到 565667 的最短路径

1.3 评分参考标准

（1）针对上述典型问题，编程实现算法，程序能够针对一种输入正常运行，给出正确结果，并提交代码及实验报告

（2）算法程序能够面对多种输入和边界条件稳定运行，输出正确结果，算法性能符合预期，并且按期提交代码及实验报告

（3）在满足 2 的前提下，能够观察对比不同规模的输入数据下，算法的运行时间和空间占用的变化，分析算法时间和空间复杂性

（4）在满足 2、3 的前提下，提出改进算法性能的思路及方法，并付诸实现

2 算法设计

2.1 算法原理

1．设置集合 S={i}⊆V，记录已经得到最短路径的顶点 i（已经求出 v 至 i 的最短路径）

顶点 i∈S，当且仅当从源 v 到该顶点 i 的（全局）最短路径已知

初始时，S 中只有源点 v

2．对图 G(V，E)中某一个顶点 u∈V，将从源 v 到 u 且中间只经过 S 中的顶点的路径称为从源点 v 到 u 的特殊路径，用数组 dist 记录 v 到图中各点 u 的特殊路径长度，记为 dist[u]

v 到 u 的特殊最短路径，长度 dist[u]

从 v 到 u、中间只经过 S 中的顶点

v 到 u 的全局最短路径，长度 d(v,u)

d(v,u) ≤ dist[u]

可能经过不在 S 中的顶点，如 k

3．采用贪心选择策略，从 V-S 中挑选具有最小 dist[u_k]的顶点 u_k，将 u_k 加入 S，S={u_k}∪S

S_0={v}，S_1={v，u_1}，…，S_n={u_{n-1}}∪S_{n-1}

4．当 S=V 时，获得源点 v 至图中全部其它 n-1 个顶点的最短路径，算法结束

贪心策略

扩展集合 S 时，从 V-S 中取出具有最短特殊路径长度 dist[u]的顶点 u，将 u 添加到 S 中

S 发生改变，对数组 dist 应作必要修改，重新计算 v 到 S 外各点 u 的 dist[u]

2.2 算法正确性证明

2.2.1 问题形式化描述

对图 G(V，E)，源点 v，从三方面描述问题

（1）源点 v，图中顶点集合 V，，各顶点 u∈V 的全局最短路径及其长度 d(v，u)

算法迭代过程中保持不变

（2）用于构造最短路径的当前顶点集合 S_i ，

不断增加，定义了不同规模的子问题

（3）指标：相对于现有 S_i ，对各顶点 u∈V，dist[u]

随着 S_i 的变化， dist[u]可能需要不断更新

问题/子问题描述：[S_i，{<u,dist[u]>}]，S_i ⊆V，u∈V

最简子问题：[{u}，{<u，0>}]， S_i 只包含源点 u

原问题：[V, {<u,dist[u]>}] = [V, {<u,d(u,v)>}]，S_i =V

2.2.2 贪心选择性质

（1）贪心选择策略

循环/迭代计算中每一步，从 V-S_i 中选择具有最短特殊路径 dist[u]的顶点 u，加入 S_i 得到 S_{i+1}

（2）为证明贪心选择策略，需要证明

对任意已经在 Si 中的顶点 u，从 v 开始、经过 G 中任意顶点到达 u 的全局最短路径的长度 d(v, u) = 从 v 开始、只经过 S_i 中顶点到达 u 的最短路径的长度 dist(u)：u 一旦翻入 S，dist[u]就是 d(v, u)。

即：不存在另一条 v 到 u 的全局最短路径，该路径上某些节点 x 不在 V-S 中，且该路径长度 d(v,u)<dist[u]

（3）我们可以使用反证法来证明

在迭代求解过程中,顶点 u 是遇到的集合 S 中第 1 个满足 d(v,u) < dist[u] 的顶点，即 d(v,u) ≠ dist[u]，且全局最优路径经过 S 之外的顶点。从 v 到 u 的全局最短路径上，经过的第 1 个属于 V-S_i 的顶点为 x

对 v 到 u 的全局最短路径 d(v ,u) ，根据 d(v, x) + distance(x, u) = d(v ,u) ，由于 distance(x, u) >0，有

d(v, x) < d(v ,u)

进一步地，根据假设 d(v,u) < dist[u]，有

d(v, x) + distance(x, u) = d(v ,u) < dist[u]

由于 distance(x, u) >0，因此

dist[x]= d(v, x) ≤ d(v,u)= d(v, x) + distance(x, u) < dist[u],

即 dist[x]< dist[u]

但是根据路径 p 构造方法，在下图所示情况下，u、x 都在集合 S_i 之外，即 u、x 都属于 V-S_i，贪心选择 S 外顶点时，u 被选中，并没有选 x，根据扩展 S_i 的原则：选 dist 最小的顶点加入 S_i，说明此时

$$dist[u] \leq dist[x]$$

这与 前面推出的

$$dist[x] < dist[u]$$

相矛盾

所以我们可以说明这个问题的贪心选择性质。

2.2.3 最优子结构性质

对顶点 u，考察将顶点 u 加到集合 S_i 之前和之后，dist[u]的变化，添加 u 之前对应的顶点集合为 S_i，加入 u 之后的顶点集合为 S_{i+1}

（1）老 S_i：子问题

（2）新 S_{i+1}：原问题/规模更大子问题

要求：u 加到 S_i 中后，重新计算 V-S_i 中的各顶点 i 的 dist，dist[i]、dist[u]不增加

对另外 1 个 S_i 外的节点 i，考察 u 的加入对 dist[i]的影响：

（1）情况 1.假设添加 u 后，出现 1 条从 v 到 i 的新路，该路径先由 v 经过老 S_i 中的顶点到达 u，再从 u 经过一条直接边到达 i。如果 dist[u]+ c[u][i] < 原来的 dist[i]，则算法用 dist[u] + c[u][i] 替代 dist[i]，得到新的 dist[i]；否则， dist[i]不更新

（2）情况 2．如果新路径如下图所示，先经过 u，再回到 S_i 中的 x，由 x 直接到达 i。x 处于老的 S_i 中，故 dist[x]=d(v,x)已经是由 v 到 x 的全局最短路径的长度，x 比 u 先加入 S_i，dist[x]是全局最短路径，因此

$$dist[x] \leq dist[u] + path(u,x)$$

此时，从源点 v 到 i 的最短路径 dist[i]=dist[x]+c[x,i]小于路径（v，u，x，i）的长度，因此算法更新 dist[i]时不受路径(v，u，x，i)影响，即 u 的加入对 dist[i]无影响

因此，无论算法中 dist[u]的值是否变化，它总是关于当前顶点集合 S 的到顶点 u 的特殊最短路径

只针对子集 S，不一定是全局最优

也就是说：对于加入 u 之前、之后的新老 S 所对应的 2 个子问题，算法执行过程保证了 dist[u]始终是 u 相对于 S 的最优解

因此，算法结束时，S=V，dist(u)成为全局最优解

问题/子问题描述

$$S_i, \{<u,dist[u]>\}], S_i \subseteq V, u \in V$$

相对于当前的 S_i ，各顶点 u 的特殊路径的最短距离 dist[u]

最简子问题：[{u}，{<u，0>}]， S_i 只包含源点 u

原问题：[V，{<u,dist[u]>}] = [V，{<u,d(u,v)>}]，S_i =V

2.3 核心算法代码实现

```
1.  vector<double> dijkstra(vector<vector<double>> graph, int v){
2.      // 初始化 s 集合，只有一个起始点在里面
3.      int n = graph[0].size();
4.      vector<bool> s(n, false);
5.      vector<double> dist(n, MAXINT);
```

```
    s[v] = true;
    dist[v] = 0;
    cout << "dist 数组初始化完成" << endl;
    // 进行最短路径算法
    for(int i = 0; i < n; i++){
        // 考虑原点 v 之外的其他点 u，对于不同规模 si，逐步扩展
        int temp = MAXINT;
        int u = v;
        for(int j = 0; j < n; j++){
            //选取 s 外具有最小 dist 的点
            if(!s[j] && dist[j] < temp){
                u = j;
                temp = dist[j];
            }
        }
        s[u] = true;
        for(int j = 0; j < n; j++){
            if(!s[j] && graph[u][j] != MAXINT
            && dist[j] > graph[u][j]+dist[u]){
                dist[j] = graph[u][j] + dist[u];
            }
        }
    }
    return dist;
}
```

2.4 算法复杂性分析

2.4.1 时间复杂度

外层循环：外层循环执行了 n 次，其中 n 为图的顶点数量。

第一个内层循环：第一个内层循环执行了 n 次，用于选择未标记的顶点中距离源点最短的顶点。

第二个内层循环：第二个内层循环也执行了 n 次，用于更新源点到其他顶点的最短距离。

算法的其余部分所需要的时间不超过 O(n^2)。

因此，总体的时间复杂度为 O(n^2)。在最坏情况下，这个算法可能需要 O(n^2) 的时间来完成。

2.4.2 空间复杂度

使用了大小为 n 的布尔数组，用于标记哪些顶点已经被包含在 S 集合中。因此，空间复杂度为 O(n)。

使用了大小为 n 的 dist 数组，用于存储源点到各个顶点的最短距离。因此，空间复杂度为 O(n)。

使用了常量级别的其他局部变量，对总体空间复杂度没有显著影响。

总体来说，空间复杂度为 O(n)。

2.4.3 大规模输入下的算法性能分析

时间复杂度： 随着输入规模的增加，算法的运行时间将按照 O(n^2) 的复杂度增长。因为算法的性能主要受到两个嵌套的循环的影响，它在小规模的输入中性能良好，但对于大规模图可能会变得相对较慢。

空间复杂度： 由于使用了两个一维数组来存放算法中间步骤所需要的值，算法的空间占用是线性的，与顶点的数量成正比。对于大规模图来说，所需要的空间占用也呈线性增长。

2.5 改进算法——获得最短路径

在实际问题中，我们往往不仅仅关心两个点之间的最短路径是多少，还会关心怎么从这个点用最短的路径走到目的点。

2.5.1 算法思想

在贪心算法的基础上，每次当 dist 数组更新时，都会记录使这个距离更新的前一个节点的下标。最后通过回溯，来查询这条最短路径上的各个顶点。

初始化一个辅助数组 path，大小与图中节点数相同，用于记录在更新最短路径时的前驱节点。

使用贪心算法进行最短路径计算，与普通的 Dijkstra 算法类似，但在每次更新最短距离时，同时记录当前节点的前驱节点。

在回溯阶段，从目标节点开始，根据 path 数组不断追溯前驱节点，直至回溯到起始节点。这样就得到了最短路径上的各个顶点。

2.5.2 伪代码思路

```
function dijkstraWithPath(graph, start, end):
    // 初始化
    n = graph.size()
    dist = array of size n, initialized with infinity
    predecessors = array of size n, initialized with -1
    visited = array of size n, initialized with false

    dist[start] = 0

    // 贪心算法计算最短路径
    for i from 0 to n-1:
        u = vertex with the minimum dist value among unvisited vertic
es
        visited[u] = true

        for each neighbor v of u:
            if v is unvisited:
                newDist = dist[u] + weight(u, v)
                if newDist < dist[v]:
                    dist[v] = newDist
```

```
                    predecessors[v] = u

    // 回溯最短路径
    path = array or list to store the path
    current = end
    while current != start:
        path.push_back(current)
        current = predecessors[current]
    path.push_back(start)

    // 反转路径，使其从起始点到目标点
    reverse(path.begin(), path.end())

    return path
```

2.5.3 算法实现

```
vector<int> getPrePath(vector<vector<double>> graph, int v){
    // 初始化 s 集合，只有一个起始点在里面
    int n = graph[0].size();
    vector<bool> s(n, false);
    vector<double> dist(n, MAXINT);
    vector<int> path(n, -1);
    s[v] = true;
    dist[v] = 0;
    cout << "dist 数组初始化完成" << endl;
     // 进行最短路径算法
     for(int i = 0; i < n; i++){
         // 考虑原点 v 之外的其他点 u，对于不同规模 si，逐步扩展
         int temp = MAXINT;
         int u = v;
         for(int j = 0; j < n; j++){
             //选取 s 外具有最小 dist 的点
             if(!s[j] && dist[j] < temp){
                 u = j;
                 temp = dist[j];
             }
          }
          s[u] = true;
          for(int j = 0; j < n; j++){
              if(!s[j] && graph[u][j] != MAXINT
              && dist[j] > graph[u][j]+dist[u]){
```

```
                dist[j] = graph[u][j] + dist[u];
                path[j] = u;
            }
        }
    }
    return path;
}
vector<int> getPath(vector<int>path, int v, int end)
{
    vector<int> p;
    p.push_back(end);
    while(end != v){
        end = path[end];
        p.push_back(end);
    }
    reverse(p.begin(), p.end());
    return p;
}
```

2.5.4 算法复杂性分析

2.5.4.1 时间复杂度

外层循环：外层循环执行了 n 次，其中 n 为图的顶点数量。

第一个内层循环：第一个内层循环执行了 n 次，用于选择未标记的顶点中距离源点最短的顶点。

第二个内层循环：第二个内层循环也执行了 n 次，用于更新源点到其他顶点的最短距离。

算法的其余部分所需要的时间不超过 O(n^2)。

因此，总体的时间复杂度为 O(n^2)。在最坏情况下，这个算法可能需要 O(n^2) 的时间来完成。

2.5.4.2 空间复杂度

使用了大小为 n 的布尔数组，用于标记哪些顶点已经被包含在 S 集合中。因此，空间复杂度为 O(n)。

使用了大小为 n 的 dist 数组，用于存储源点到各个顶点的最短距离。因此，空间复杂度为 O(n)。

使用了常量级别的其他局部变量，对总体空间复杂度没有显著影响。

总体来说，空间复杂度为 O(n)。

2.6 改进算法——基于优先队列的 A*算法

2.6.1 A*算法思想

在实现迪杰斯特拉算法的时候，发现核心算法函数中使用了两层循环，并且循环中都是从 0 一直遍历到 n-1。这种实现方式过于暴力，在实际生活问题中往往计算速度并不快。比如说当基站数目不是几十个，而是几十万个的时候，普通的迪杰斯特拉算法的性能就会大大降低。

盲目搜索会浪费很多时间和空间， 所以我们在路径搜索时， 会首先选择最有希望的节点，如何选择最有希望的节点呢？那就要构造一个合适的启发函数，这个启发函数用来估计当前点到目标点的开销值。

这个减少盲目搜索的算法被称为 A*算法：

A*算法是一种基于启发式搜索的路径规划算法，其目标是找到从起始点到目标点的最短路径。它综合了实际代价和启发式估算代价，通过维护一个优先队列，每次选择具有最小综合代价的节点进行拓展。以下是 A 算法的主要步骤：

（1）初始化起始节点，启发式函数值，实际代价和综合代价。

（2）将起始节点加入优先队列。

（3）进入循环，直到队列为空或者找到目标节点：

1.从优先队列中选择具有最小综合代价的节点。

2.如果选择的节点是目标节点，返回路径。

3.否则，将该节点标记为已访问，遍历其相邻节点：

如果相邻节点未被访问，计算实际代价和综合代价，更新信息。

将相邻节点加入优先队列。

（4）如果循环结束且队列为空，表示未找到路径。

可以看到，A* 算法和迪杰斯特拉算法还是有一些相似之处的。它们都维护了一个已搜索过的集合，也都保存了原点到每个已搜索过集合元素的真实开销。在遍历相邻节点时，它们都会通过比较新路径的实际代价和已有信息，选择是否更新路径信息。如果新路径更短，则更新相关信息。

不同之处在于在 A*算法中，通过综合实际代价和启发式估算代价来选择扩展的节点。通过维护 f(n)值来选择下一个扩展的节点。A*算法使用一个启发函数（这里我采用曼哈顿距离）来估计当前节点到目标节点的代价。曼哈顿距离是一种在网格上的启发式函数，通过计算节点在网格中的行列差值之和来估计距离。

2.6.2 伪代码思路

```
function AStar(start, goal):
    openList = {start}    // 开放列表
    closedList = {}       // 关闭列表
    gScore[start] = 0     // 起始节点到当前节点的实际代价
    hScore[start] = heuristic(start, goal)  // 启发式函数的值
```

```
    fScore[start] = gScore[start] + hScore[start]  // 综合代价

    while openList is not empty:
        current = node with lowest fScore in openList
        if current == goal:
            return reconstructPath(cameFrom, goal)

        remove current from openList
        add current to closedList

        for each neighbor of current:
            if neighbor in closedList:
                continue
            tentativeGScore = gScore[current] + distance(current, neighbor)

            if neighbor not in openList or tentativeGScore < gScore[neighbor]:
                cameFrom[neighbor] = current
                gScore[neighbor] = tentativeGScore
                hScore[neighbor] = heuristic(neighbor, goal)
                fScore[neighbor] = gScore[neighbor] + hScore[neighbor]

                if neighbor not in openList:
                    add neighbor to openList

    return failure

function reconstructPath(cameFrom, current):
    totalPath = [current]
    while current in cameFrom:
        current = cameFrom[current]
        totalPath.prepend(current)
    return totalPath
```

2.6.3 核心算法实现

```
vector<int> reconstructPath(unordered_map<int, int> &cameFrom, int u)

{
    vector<int> totalPath = {u};
```

```
    while (cameFrom.find(u) != cameFrom.end())
    {
        u = cameFrom[u];
        totalPath.push_back(u);
    }
    reverse(totalPath.begin(), totalPath.end());
    return totalPath;
}

vector<int> AStar(vector<vector<double>> &graph, int start, int goal)

{
    int n = graph.size();

    set<int> openList;
    vector<bool> s(graph[0].size(), false);
    // g(n)代表从起始点到当前点 n 的开销
    unordered_map<int, double> dist;

    // h(n)代表当前点 n 到终点 goal 的估算开销
    unordered_map<int, double> h;

    // f(n) = g(n) + h(n)
    unordered_map<int, double> f;

    unordered_map<int, int> cameFrom;

    dist[start] = 0;
    h[start] = heuristic(graph, start, goal);
    f[start] = dist[start] + h[start];
    // 1. 先将起始点放入开放列表
    openList.insert(start);
    while (!openList.empty())
    {
        // 优先队列，优先级设置为 f(n)
        int u = *min_element(openList.begin(), openList.end(),
            [&](int a, int b){ return f[a] < f[b]; });

        if (u == goal)
        {
            return reconstructPath(cameFrom, goal);
        }
```

```
45.
46.         openList.erase(u);
47.         s[u] = true;
48.
49.         for (int j = 0; j < n; ++j)
50.         {
51.             // 查找不在关闭队列中的
52.             if (graph[u][j] != MAXINT && !s[j])
53.             {
54.                 double tempDist = dist[u] + graph[u][j];
55.                 if (openList.find(j) == openList.end() || tempDist <
    dist[j])
56.                 {
57.                     // 更新数据
58.                     cameFrom[j] = u;
59.                     dist[j] = tempDist;
60.                     h[j] = heuristic(graph, j, goal);
61.                     f[j] = dist[j] + h[j];
62.                     openList.insert(j);
63.                 }
64.             }
65.         }
66.     }
67.
68.     // 如果 openList 空了还没找到终点，就返回空
69.     return vector<int>();
70. }
```

2.6.4 复杂度分析

2.6.4.1 时间复杂度

在 A 算法中，主要操作是维护一个优先队列，每次从队列中选择 f(n)最小的节点进行拓展。在最坏情况下，每个节点可能都会进出队列一次，因此 A 算法的时间复杂度可以近似看作 O(N^2)。

由于 A 算法在迪杰斯特拉算法的基础上引入了启发式函数，可能提前剪枝，从而在实际应用中取得更好的性能。但总体而言，A 算法的时间复杂度与传统迪

杰斯特拉算法相当。

2.6.4.2 空间复杂度

该算法需要维护启发式函数值、实际代价、f(n)值等信息，因此额外的空间开销会比传统迪杰斯特拉算法稍多，但仍然是 O(N)级别的。

总体而言，A*算法在空间上的开销相对于传统迪杰斯特拉算法略有增加，但依然保持在 O(N)的水平。

3 程序设计实现

3.1 程序源码

```
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
#include <unordered_map>
#include <limits>
#include <fstream>
#include <string>
#include <sstream>
#include <math.h>
#include <stack>
#define MAXINT 1000000000
using namespace std;
// 将基站 id 转为数组的索引
int getIndexFromId(int id, vector<int>ids){
    for(int i = 0; i < ids.size();i ++){
        if(id == ids[i])
            return i;
    }
    return -1;
}
// 获取用户输入的终点
int getInputEnd(vector<int>ids){
    int id;
    for(int i = 0; i < ids.size();i++){
        cout << ids[i] << "\t";
```

```
    }
    cout << endl <<"请从以上基站 ID 中选择一个当做终点:";
    cin >> id;
    while(true){
        auto it = find(ids.begin(), ids.end(), id);
        if(it == ids.end()){
            cout << "输入不合法，请重新输入"<< endl;
            cin >> id;
        }else{
            break;
        }
    }
    return getIndexFromId(id, ids);
}
// 获取用户输入的起点
int getInputStart(vector<int>ids){
    int id;
    for(int i = 0; i < ids.size();i++){
        cout << ids[i] << "\t";
    }
    cout << endl <<"请从以上基站 ID 中选择一个当做起点:" ;
    cin >> id;
        while(true){
        auto it = find(ids.begin(), ids.end(), id);
        if(it == ids.end()){
            cout << "输入不合法，请重新输入"<< endl;
            cin >> id;
        }else{
            break;
        }
    }
    return getIndexFromId(id, ids);
}

int getIdFromIndex(int index, vector<int>ids){
    return ids[index];
}
// 完整的路径
vector<int> getPath(vector<int>path, int v, int end)
{
    vector<int> p;
    p.push_back(end);
```

```
69.     while(end != v){
70.         end = path[end];
71.         p.push_back(end);
72.     }
73.     reverse(p.begin(), p.end());
74.     return p;
75. }
76. 
77. vector<int> getPrePath(vector<vector<double>> graph, int v){
78.     // 初始化 s 集合，只有一个起始点在里面
79.     int n = graph[0].size();
80.     vector<bool> s(n, false);
81.     vector<double> dist(n, MAXINT);
82.     vector<int> path(n, -1);
83.     s[v] = true;
84.     dist[v] = 0;
85.     // 进行最短路径算法
86.     for(int i = 0; i < n; i++){
87.         // 考虑原点 v 之外的其他点 u，对于不同规模 si，逐步扩展
88.         int temp = MAXINT;
89.         int u = v;
90.         for(int j = 0; j < n; j++){
91.             //选取 s 外具有最小 dist 的点
92.             if(!s[j] && dist[j] < temp){
93.                 u = j;
94.                 temp = dist[j];
95.             }
96.         }
97.         s[u] = true;
98.         for(int j = 0; j < n; j++){
99.             if(!s[j] && graph[u][j] != MAXINT
100.                && dist[j] > graph[u][j]+dist[u]){
101.                    dist[j] = graph[u][j] + dist[u];
102.                    path[j] = u;
103.                }
104.            }
105.        }
106.        return path;
107.    }
108. 
109.    vector<double> dijkstra(vector<vector<double>> graph, int v){
110.        // 初始化 s 集合，只有一个起始点在里面
```

```
111.        int n = graph[0].size();
112.        vector<bool> s(n, false);
113.        vector<double> dist(n, MAXINT);
114.        s[v] = true;
115.        dist[v] = 0;
116.        // 进行最短路径算法
117.        for(int i = 0; i < n; i++){
118.            // 考虑原点 v 之外的其他点 u，对于不同规模 Si，逐步扩展
119.            int temp = MAXINT;
120.            int u = v;
121.            for(int j = 0; j < n; j++){
122.                //选取 s 外具有最小 dist 的点
123.                if(!s[j] && dist[j] < temp){
124.                    u = j;
125.                    temp = dist[j];
126.                }
127.            }
128.            s[u] = true;
129.            for(int j = 0; j < n; j++){
130.                if(!s[j] && graph[u][j] != MAXINT
131.                && dist[j] > graph[u][j]+dist[u]){
132.                    dist[j] = graph[u][j] + dist[u];
133.                }
134.            }
135.        }
136.        return dist;
137.    }
138.
139.    // 用曼哈顿距离作为启发函数
140.    double heuristic(vector<vector<double>> &graph, int u, int goal)
141.    {
142.        int n = graph.size();
143.        int uRow = u / n;
144.        int uCol = u % n;
145.        int goalRow = goal / n;
146.        int goalCol = goal % n;
147.
148.        return abs(uRow - goalRow) + abs(uCol - goalCol);
149.    }
150.
151.    vector<int> reconstructPath(unordered_map<int, int> &cameFrom, int u)
152.    {
```

```
153.        vector<int> totalPath = {u};
154.        while (cameFrom.find(u) != cameFrom.end())
155.        {
156.            u = cameFrom[u];
157.            totalPath.push_back(u);
158.        }
159.        reverse(totalPath.begin(), totalPath.end());
160.        return totalPath;
161.    }
162.
163.    vector<int> AStar(vector<vector<double>> &graph, int start, int goal)
164.    {
165.        int n = graph.size();
166.
167.        set<int> openList;
168.        vector<bool> s(graph[0].size(), false);
169.        // g(n)代表从起始点到当前点 n 的开销
170.        unordered_map<int, double> dist;
171.
172.        // h(n)代表当前点 n 到终点 goal 的估算开销
173.        unordered_map<int, double> h;
174.
175.        // f(n) = g(n) + h(n)
176.        unordered_map<int, double> f;
177.
178.        unordered_map<int, int> cameFrom;
179.
180.        dist[start] = 0;
181.        h[start] = heuristic(graph, start, goal);
182.        f[start] = dist[start] + h[start];
183.        // 1. 先将起始点放入开放列表
184.        openList.insert(start);
185.        while (!openList.empty())
186.        {
187.            // 优先队列，优先级设置为 f(n)
188.            int u = *min_element(openList.begin(), openList.end(),
189.                [&](int a, int b){ return f[a] < f[b]; });
190.
191.            if (u == goal)
192.            {
193.                return reconstructPath(cameFrom, goal);
194.            }
```

```
            openList.erase(u);
            s[u] = true;

            for (int j = 0; j < n; ++j)
            {
                // 查找不在关闭队列中的
                if (graph[u][j] != MAXINT && !s[j])
                {
                    double tempDist = dist[u] + graph[u][j];
                    if (openList.find(j) == openList.end() || tempDist < dist[j])
                    {
                        // 更新数据
                        cameFrom[j] = u;
                        dist[j] = tempDist;
                        h[j] = heuristic(graph, j, goal);
                        f[j] = dist[j] + h[j];
                        openList.insert(j);
                    }
                }
            }
        }

        // 如果 openList 空了还没找到终点，就返回空
        return vector<int>();
    }


    int main(){
        vector<vector<double>> graph;
        ifstream inputFile("./1-1-1.txt");
        if(!inputFile.is_open()){
            cerr << "文件打开失败" << endl;
            return 1;
        }
        string line;
        getline(inputFile, line);
        stringstream ss(line);
        // 读取第一行基站数据
        int id;
        double value;
        vector<int> ids;
```

```
236.        while(ss >> id){
237.            ids.push_back(id);
238.        }
239.        // 跳过第一列读取距离数据
240.        while(inputFile >> id){
241.            vector<double> row;
242.            for(int i = 0; i < ids.size(); i++){
243.                inputFile >> value;
244.                if(value+1 < abs(1e-4))
245.                    value = MAXINT;
246.                row.push_back(value);
247.            }
248.            graph.push_back(row);
249.        }
250.        inputFile.close();
251.
252.        // 用户输入起始位置
253.        int start = getInputStart(ids);
254.        vector<double>dist = dijkstra(graph, start);
255.        // int end = getInputEnd(ids);
256.        for(int i = 0 ; i < ids.size(); i++){
257.            int end = i;
258.            vector<int>prePath = getPrePath(graph, start);
259.            vector<int>path = getPath(prePath, start, end);
260.            cout << "==================以下是普通算法迪杰斯特拉的结果
    ==================="<< endl;
261.            cout << "dijkstra 到达终点的路径为:" ;
262.            for(int i = 0 ; i < path.size()-1; i++){
263.                cout << getIdFromIndex(path[i], ids) << " --> ";
264.            }
265.            cout << getIdFromIndex(end, ids) << endl << "最短的距离
    是: " << dist[end] <<endl;
266.
267.            cout << "==================以下是改进算法 A* 的结果
    ==================="<< endl;
268.            vector<int>aPath = AStar(graph, start, end);
269.            double d = 0;
270.            // 获取路径
271.            cout << " A*从起点到终点的路径为: " ;
272.            for (int i = 1; i < aPath.size(); i++){
273.                d += graph[aPath[i - 1]][aPath[i]];
274.            }
```

```
275.            for (int i = 0; i < aPath.size(); i++){
276.                if(i != aPath.size()-1)
277.                    cout << getIdFromIndex(aPath[i], ids) << " --> ";
278.                else
279.                    cout << getIdFromIndex(aPath[i], ids) << endl;
280.            }
281.            cout << "最短距离是:" << d << endl;
282.        }
283.
284.        return 0;
285.    }
```

3.2 代码说明

3.2.1 变量说明

`#define MAXINT 1000000000`

这个宏是表示不可达的距离的。当输入中图里两个点之间权值为-1 时，将用 MAXINT 替换。

`vector<vector<double>> graph;`

这是一个二维数组，用来存放输入图的邻接矩阵。

`string line;`

这个字符串用来存放输入数据中每一行的数据。

`vector<int> ids;`

这个数组用来存放基站的 ID，顺序是按照输入的第一行的顺序。

`int start = getInputStart(ids);`

start 变量用来存放用户输入的起点。用户将输入一个基站的 ID，程序会通过 ID 将 ID 转换为对应的数组下标

`vector<double>dist = dijkstra(graph, start);`

dist 数组用来存放从起始点到各个位置的最短距离。

`int end = getInputEnd(ids);`

end 变量用来存放用户输入的终点，用户输入的是 ID 号，程序会自动转为

数组的下标。

```
vector<int>prePath = getPrePath(graph, start);
```

prePath 数组存放了最短路径里各个点的前向点。

```
vector<int>path = getPath(prePath, start, end);
```

path 数组里存放了从起始点到终点的所需要经过的所有点。

3.2.2 函数说明

dijkstra 函数

传入图的邻接矩阵，以及起始点 v，函数将用贪心的策略来寻找从起始位置到图中各个点位置的最短路径。

getPrePath 函数

传入图的邻接矩阵以及起始点 v，函数将在搜索每个点的最短路径的同时保存每个点最短路径的上一个点。假如说从 A 到 D 的最短路径是 A→B→C→D，那么就会在 D 的位置上保存 C 这个前向点。

getPath 函数

这个函数通过输入从 getPrePath 函数获取得来的各个点的前向点，以及终点位置。该函数会从终点出发，从后往前将最短路径复现一遍，并将路径保存到数组中返回。

getIdFromIndex 函数

这个函数通过输入下标以及 ID 数组，来讲下标转为 ID 号

getIndexFromID 函数

这个函数通过 ID 号来获取对应的数组下标

getInputStart 函数

这个函数会指引用户输入起始的 ID 号，并检查 ID 号是否合法。然后将 ID 号转为数组下标返回。

getInputEnd 函数

这个函数会指引用户输入终点的基站 ID 号，并检查 ID 号是否合法。然后将 ID 号转为数组下标返回。

reconstructPath 函数

用来构造最短路径，作用同 getPath 函数。

AStar 函数

A*算法的具体实现，在 2.6.1 A*算法思想中有介绍

4 执行结果

4.1 附件 1-1-1 22 个端点图的执行结果

4.1.1 567443 -- 33109

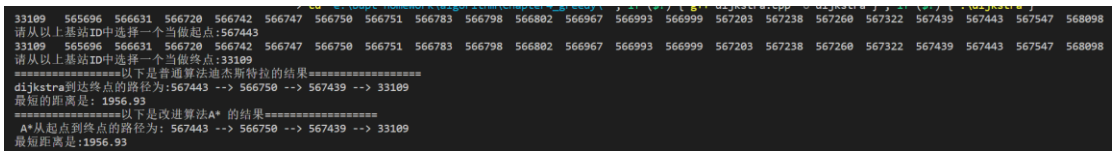

图 4-1 输入 1 执行结果 1

结果显示，两个算法都输出了 567443 --> 566750 --> 567439 --> 33109 这条路径，并且最短距离都是 1956.93

4.1.2 567443 -- 565696

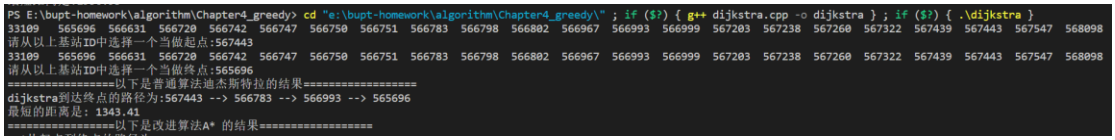

图 4-2 输入 1 执行结果 2

结果显示，两个算法都输出了 567443 --> 566783 --> 566993 --> 565696 这条路径，并且最短距离都是 1343.41

4.1.3 567443 -- 566631

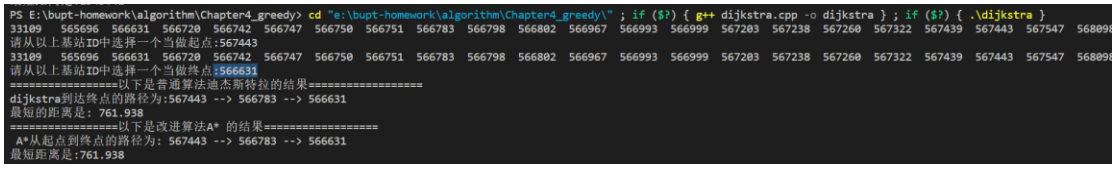

图 4-3 输入 1 执行结果 3

结果显示,两个算法都输出了 567443 --> 566783 --> 566631 这条路径，并且最短距离都是 761.938

4.1.4 567443 -- 566720

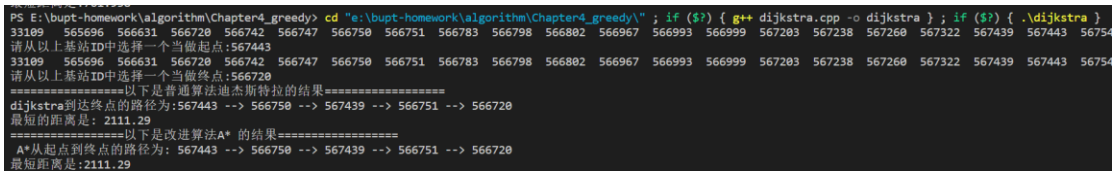

图 4-4 输入 1 执行结果 4

结果显示，两个算法都输出了 567443 --> 566750 --> 567439 --> 566751 --> 566720 这条路径，并且最短距离都是 2111.29

4.4.5 567443 -- 566993

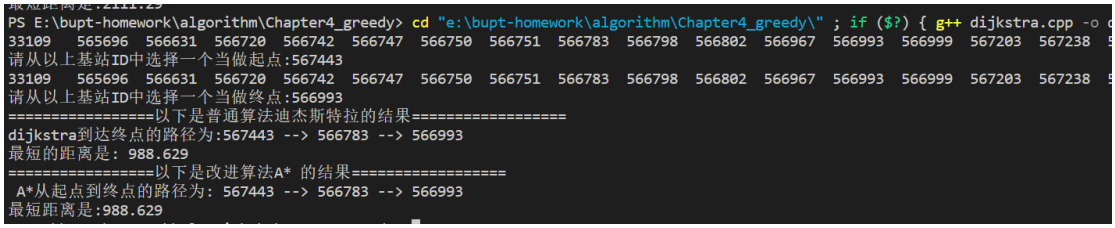

图 4-5 输入 1 执行结果 5

结果显示,两个算法都输出了 567443 --> 566783 --> 566993 这条路径,并且最短距离都是 988.629

4.4.6 567443 -- 568098

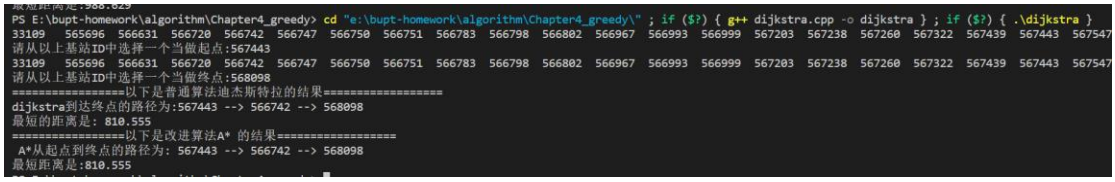

图 4-6 输入 1 执行结果 6

结果显示,两个算法都输出了 567443 --> 566742 --> 568098 这条路径,并且最短距离都是 810.555

4.4.7 其他用例

```
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566742
最短的距离是: 302.54
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566742
最短距离是:302.54
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566742 --> 566802 --> 567322 --> 566747
最短的距离是: 1988.14
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566742 --> 566802 --> 567322 --> 566747
最短距离是:1988.14
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566750
最短的距离是: 683.088
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566750
最短距离是:683.088
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566750 --> 567439 --> 566751
最短的距离是: 1622.91
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566750 --> 567439 --> 566751
最短距离是:1622.91
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566783
最短的距离是: 344.546
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566783
最短距离是:344.546
```

图 4-7 输入 1 执行结果 7

```
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566750 --> 567439 --> 566798
最短的距离是: 1778.06
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566750 --> 567439 --> 566798
最短距离是:1778.06
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566742 --> 566802
最短的距离是: 963.852
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566742 --> 566802
最短距离是:963.852
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566783 --> 566993 --> 566967
最短的距离是: 1562.25
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566783 --> 566993 --> 566967
最短距离是:1562.25
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566783 --> 566993
最短的距离是: 988.629
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566783 --> 566993
最短距离是:988.629
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566783 --> 566993 --> 566967 --> 566999
最短的距离是: 2072.92
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566783 --> 566993 --> 566967 --> 566999
最短距离是:2072.92
```

图 4-8 输入 1 执行结果 8

```
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566783 --> 566993 --> 567203
最短的距离是: 1592.31
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566783 --> 566993 --> 567203
最短距离是:1592.31
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566783 --> 567238
最短的距离是: 780.892
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566783 --> 567238
最短距离是:780.892
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 567260
最短的距离是: 244.053
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 567260
最短距离是:244.053
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:567443 --> 566742 --> 566802 --> 567322
最短的距离是: 1582.91
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 567443 --> 566742 --> 566802 --> 567322
最短距离是:1582.91
```

图 4-9 输入 1 执行结果 9

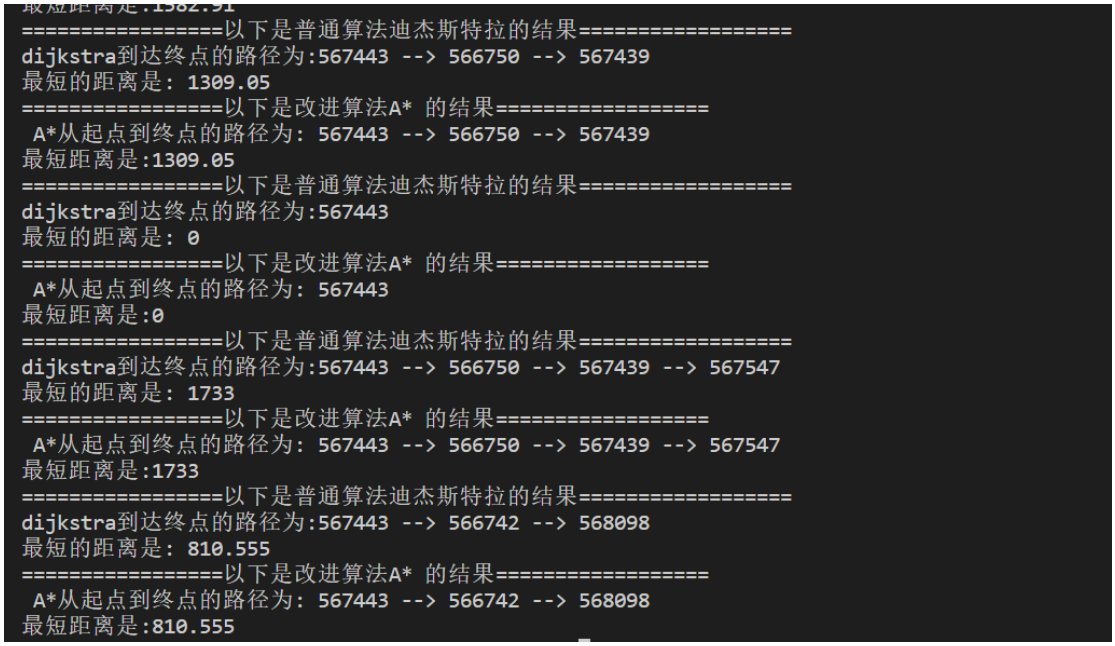

图 4-10 输入 1 执行结果 10

4.2 附件 1-1-2 44 个端点图的执行结果

4.2.1 565845 -- 565667

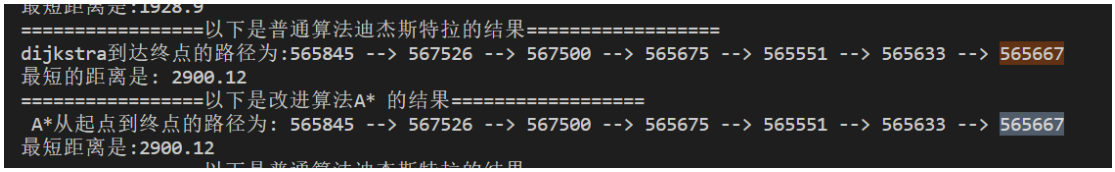

图 4-11 输入 1 执行结果 11

从结果可以看到，两个算法都输出了 565845 --> 567526 --> 567500 --> 565675 --> 565551 --> 565633 --> 565667 这条路径，并且最短距离都是 2900.12。

4.2.2 其他用例

```
65753   33566   566074  565648  567526  565551  565631  565608  567500  565531  565562  32788   567497
请从以上基站ID中选择一个当做起点:565845
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 567526 --> 567500 --> 565675
最短的距离是: 1369.37
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 567526 --> 567500 --> 565675
最短距离是:1369.37
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 566010 --> 565631 --> 565801 --> 565630 --> 565621
最短的距离是: 1928.9
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 566010 --> 565631 --> 565801 --> 565630 --> 565621
最短距离是:1928.9
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 567526 --> 567500 --> 565675 --> 565551 --> 565633 --> 565667
最短的距离是: 2900.12
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 567526 --> 567500 --> 565675 --> 565551 --> 565633 --> 565667
最短距离是:2900.12
```

图 4-12 输入 2 执行结果 1

```
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 567526 --> 567510
最短的距离是: 645.041
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 567526 --> 567510
最短距离是:645.041
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 566010 --> 565631 --> 565801
最短的距离是: 1153.11
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 566010 --> 565631 --> 565801
最短距离是:1153.11
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 566010
最短的距离是: 403.433
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 566010
最短距离是:403.433
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 567526 --> 567500 --> 565675 --> 565551 --> 567891
最短的距离是: 2401.9
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 567526 --> 567500 --> 565675 --> 565551 --> 567891
最短距离是:2401.9
```

图 4-13 输入 2 执行结果 2

```
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 567526 --> 567500 --> 565675 --> 565551 --> 565492
最短的距离是: 2223.01
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 567526 --> 567500 --> 565675 --> 565551 --> 565492
最短距离是:2223.01
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 567526 --> 567500 --> 565675 --> 565551 --> 565558
最短的距离是: 2171.29
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 567526 --> 567500 --> 565675 --> 565551 --> 565558
最短距离是:2171.29
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 567526 --> 567500 --> 565675 --> 565551 --> 565558 --> 565627
最短的距离是: 2697.46
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 567526 --> 567500 --> 565675 --> 565551 --> 565558 --> 565627
最短距离是:2697.46
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 567526 --> 567500 --> 566074 --> 565610 --> 565572
最短的距离是: 2440.92
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 567526 --> 567500 --> 566074 --> 565610 --> 565572
最短距离是:2440.92
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 567526 --> 567500 --> 566074 --> 565610
最短的距离是: 2025.89
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 567526 --> 567500 --> 566074 --> 565610
最短距离是:2025.89
```

图 4-14 输入 2 执行结果 3

```
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 567526 --> 565964 --> 567531 --> 565859
最短的距离是: 2050.98
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 567526 --> 565964 --> 567531 --> 565859
最短距离是:2050.98
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 566010 --> 565631 --> 565801 --> 565630
最短的距离是: 1468.96
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 566010 --> 565631 --> 565801 --> 565630
最短距离是:1468.96
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845 --> 567526 --> 567500 --> 565675 --> 565516 --> 565559
最短的距离是: 2381.34
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845 --> 567526 --> 567500 --> 565675 --> 565516 --> 565559
最短距离是:2381.34
==================以下是普通算法迪杰斯特拉的结果==================
dijkstra到达终点的路径为:565845
最短的距离是: 0
==================以下是改进算法A* 的结果==================
 A*从起点到终点的路径为: 565845
最短距离是:0
```

图 4-15 输入 2 执行结果 4

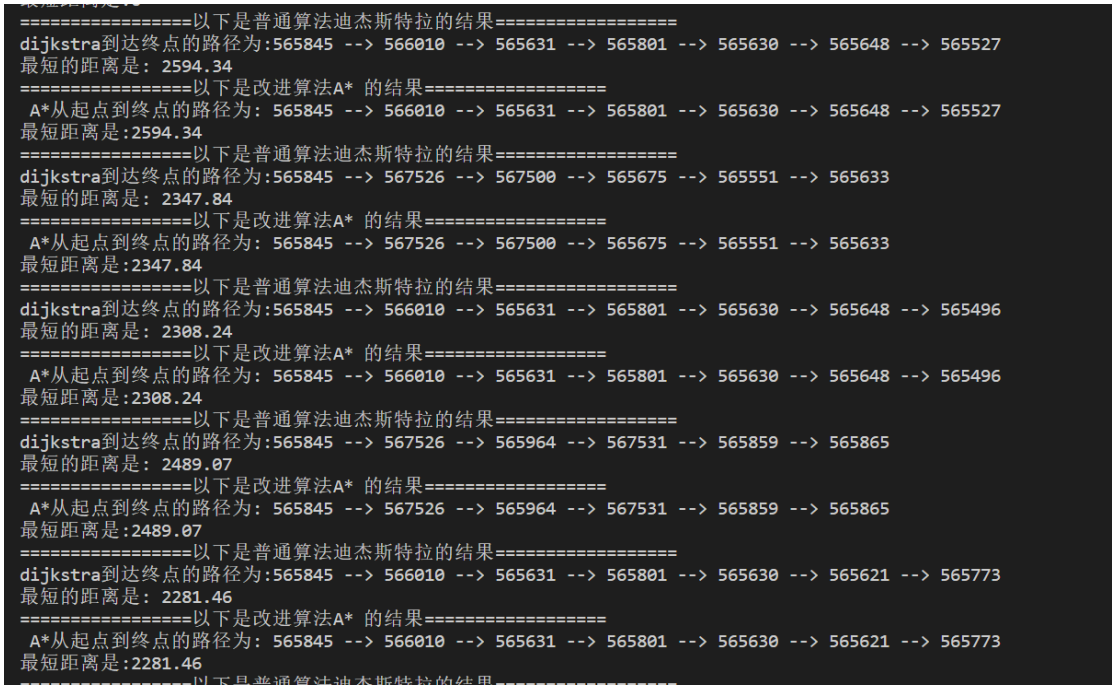

图 4-16 输入 2 执行结果 5

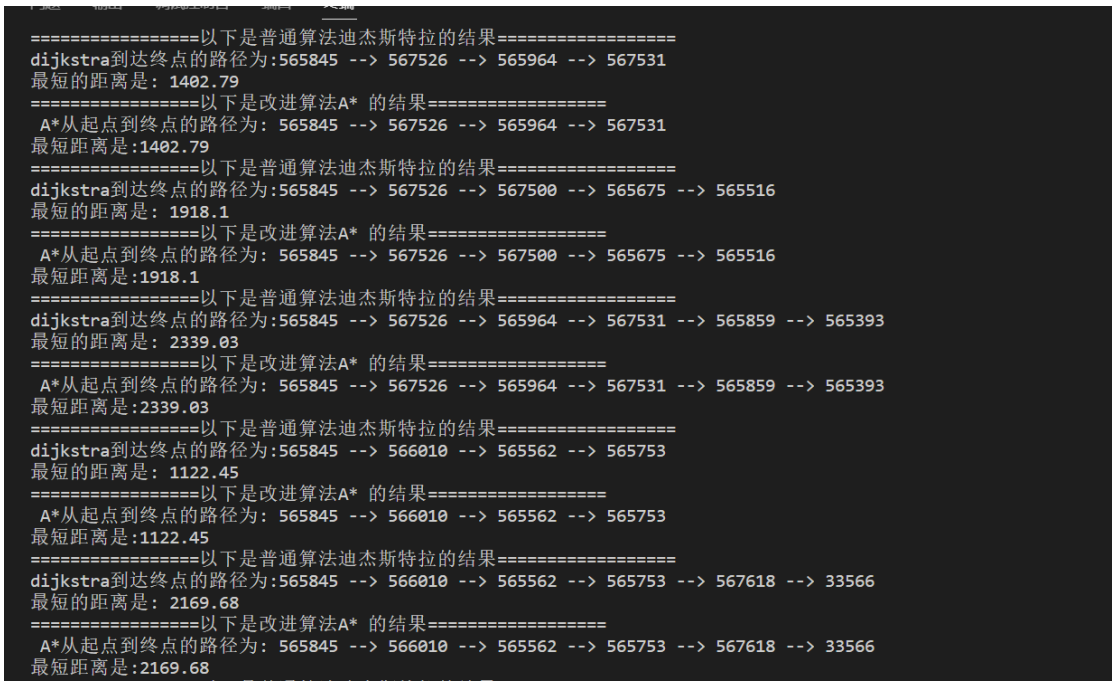

图 4-17 输入 2 执行结果 6

```
==========以下是普通算法迪杰斯特拉的结果==========
dijkstra到达终点的路径为:565845 --> 567526 --> 567500 --> 566074
最短的距离是: 1573.64
==========以下是改进算法A* 的结果==========
 A*从起点到终点的路径为: 565845 --> 567526 --> 567500 --> 566074
最短距离是:1573.64
==========以下是普通算法迪杰斯特拉的结果==========
dijkstra到达终点的路径为:565845 --> 566010 --> 565631 --> 565801 --> 565630 --> 565648
最短的距离是: 1997.17
==========以下是改进算法A* 的结果==========
 A*从起点到终点的路径为: 565845 --> 566010 --> 565631 --> 565801 --> 565630 --> 565648
最短距离是:1997.17
==========以下是普通算法迪杰斯特拉的结果==========
dijkstra到达终点的路径为:565845 --> 567526
最短的距离是: 488.237
==========以下是改进算法A* 的结果==========
 A*从起点到终点的路径为: 565845 --> 567526
最短距离是:488.237
==========以下是普通算法迪杰斯特拉的结果==========
dijkstra到达终点的路径为:565845 --> 567526 --> 567500 --> 565675 --> 565551
最短的距离是: 1806.75
==========以下是改进算法A* 的结果==========
 A*从起点到终点的路径为: 565845 --> 567526 --> 567500 --> 565675 --> 565551
最短距离是:1806.75
==========以下是普通算法迪杰斯特拉的结果==========
dijkstra到达终点的路径为:565845 --> 566010 --> 565631
最短的距离是: 843.923
==========以下是改进算法A* 的结果==========
 A*从起点到终点的路径为: 565845 --> 566010 --> 565631
最短距离是:843.923
==========以下是普通算法迪杰斯特拉的结果==========
dijkstra到达终点的路径为:565845 --> 566010 --> 565562 --> 565753 --> 567618 --> 565608
最短的距离是: 1883.38
==========以下是改进算法A* 的结果==========
 A*从起点到终点的路径为: 565845 --> 566010 --> 565562 --> 565753 --> 567618 --> 565608
最短距离是:1883.38
```

图 4-18 输入 2 执行结果 7

```
==========以下是普通算法迪杰斯特拉的结果==========
dijkstra到达终点的路径为:565845 --> 567526 --> 567500
最短的距离是: 1055.67
==========以下是改进算法A* 的结果==========
 A*从起点到终点的路径为: 565845 --> 567526 --> 567500
最短距离是:1055.67
==========以下是普通算法迪杰斯特拉的结果==========
dijkstra到达终点的路径为:565845 --> 566010 --> 565562 --> 565753 --> 567618 --> 565531
最短的距离是: 2161.48
==========以下是改进算法A* 的结果==========
 A*从起点到终点的路径为: 565845 --> 566010 --> 565562 --> 565753 --> 567618 --> 565531
最短距离是:2161.48
==========以下是普通算法迪杰斯特拉的结果==========
dijkstra到达终点的路径为:565845 --> 566010 --> 565562
最短的距离是: 853.566
==========以下是改进算法A* 的结果==========
 A*从起点到终点的路径为: 565845 --> 566010 --> 565562
最短距离是:853.566
==========以下是普通算法迪杰斯特拉的结果==========
dijkstra到达终点的路径为:565845 --> 567526 --> 565964 --> 567531 --> 565859 --> 32788
最短的距离是: 2187.66
==========以下是改进算法A* 的结果==========
 A*从起点到终点的路径为: 565845 --> 567526 --> 565964 --> 567531 --> 565859 --> 32788
最短距离是:2187.66
==========以下是普通算法迪杰斯特拉的结果==========
dijkstra到达终点的路径为:565845 --> 566010 --> 565562 --> 565753 --> 567497
最短的距离是: 1561.46
==========以下是改进算法A* 的结果==========
 A*从起点到终点的路径为: 565845 --> 566010 --> 565562 --> 565753 --> 567497
最短距离是:1561.46
```

图 4-19 输入 2 执行结果 8

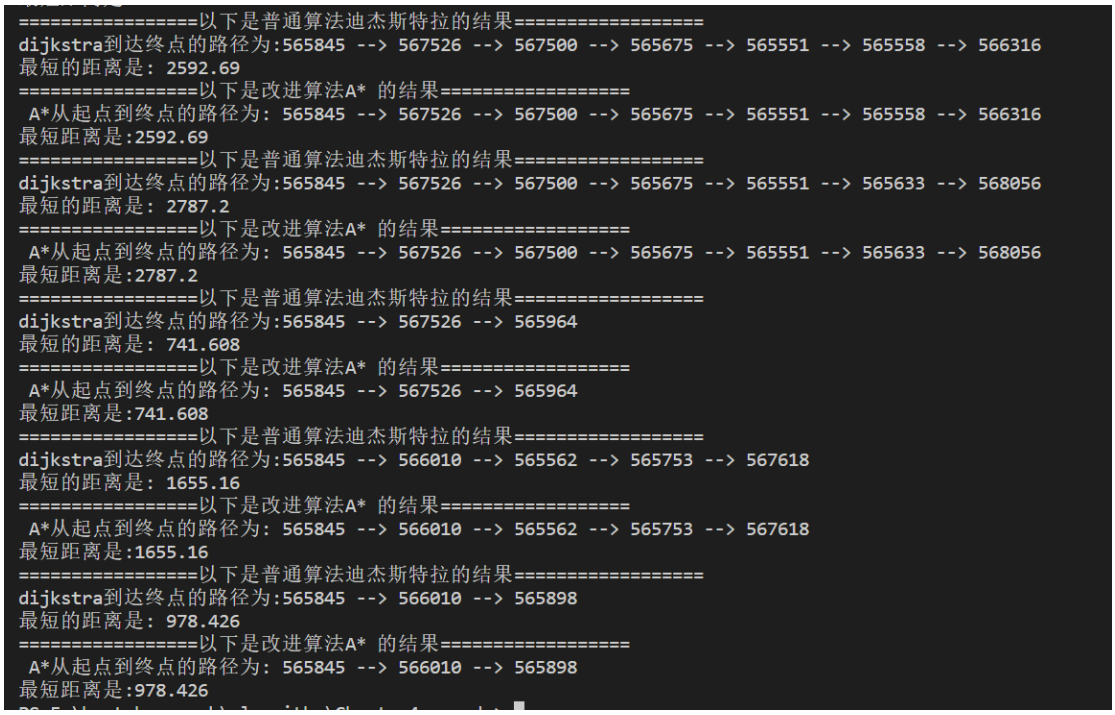

图 4-20 输入 2 执行结果 9

从结果显示，两个算法计算的路径都相同，计算的距离也相同。