医药物流配送网络优化实验报告

班级：医学信息工程专业2022级

组长学号：2022072094 组长姓名：杨猛

完成日期： 2023年 12月 14日

|  |  |
| --- | --- |
| 姓名 | 分工 |
| 杨猛 | 总体设计，编写代码 |
| 刘佳锐 | 核心算法总结，流程图绘制 |
| 黎梓贤 | PPT制作 |
| 何奕逊 | PPT完善，实验报告完善 |
| 黄静坡 | 代码讲解，PPT讲解 |

一、问题的需求分析

1、问题描述

假设有一家医药公司需要将其生产的药品配送给全国各地的零售药店。公司在全国各地拥有多个仓库，每个仓库存储一定数量的药品，并且每个仓库都可以向周围的零售药店进行配送。零售药店的数量和位置是已知的，而每个零售药店的需求量也是已知的。现在需要设计一个最优的药品配送方案，使得公司的总配送成本最小。

2、需求分析

1. 敏捷性和快速性：医药物流配送需要具备敏捷灵活的特性，能够快速响应订单需求并进行及时的配送。这是因为医药产品的特殊性，有些医药品需要紧急配送，例如急救药物或防疫物资。

2. 安全性和可靠性：医药产品一般需要在特定的温度条件下存储和运输，因此配送网络需要确保产品的安全和质量，以避免药物的变质或损坏。同时，具备可靠性也非常重要，确保医药产品按时送达，避免患者因缺乏药品而产生风险。

二、抽象数据类型的设计

// 定义图的抽象数据类型

typedef struct {

int num\_vertices; // 顶点数量

int graph[MAX\_V][MAX\_V]; // 邻接矩阵

} Graph;

// 初始化图

void initGraph(Graph \*graph, int num\_vertices);

// 添加边

void addEdge(Graph \*graph, int source, int destination, int weight);

// 检查邻接矩阵是否有效

bool validateMatrix(int graph[MAX\_V][MAX\_V], int num\_vertices);

// 计算从源点到目标点的最短路径

void dijkstra(Graph \*graph, int src);

三、算法与数据结构的设计

1、数据结构设计

在实现Dijkstra算法时，需要用到两个数组：一个存储当前源点到各个顶点的最短距离（dist数组），一个存储每个顶点是否被访问过（visited数组）。

代码使用了邻接矩阵来描述医药物流网络中各个顶点之间的连接关系和距离成本。具体数据结构设计如下：

1、宏定义

MAX\_V: 最大顶点数量，即邻接矩阵的行列数。

INF: 定义无穷大，即两个不直接相连的顶点之间的距离或成本。

2、函数

(1) validateMatrix：检查邻接矩阵是否有效。 参数：

graph[MAX\_V][MAX\_V]：邻接矩阵

num\_vertices：顶点数量 返回值：

bool类型。如果邻接矩阵有效，返回true，否则返回false。

(2) stringToInt：将字符串转换为整数。 参数：

str：需要转换的字符串 返回值：

int类型。如果字符串为"inf"或"INF"，返回INF，否则返回转换后的整数。

(3) dijkstra：计算从源点到目标点的最短路径。 参数：

graph[MAX\_V][MAX\_V]：邻接矩阵

num\_vertices：顶点数量

src：源点，即起始点或出发点 返回值：

无。

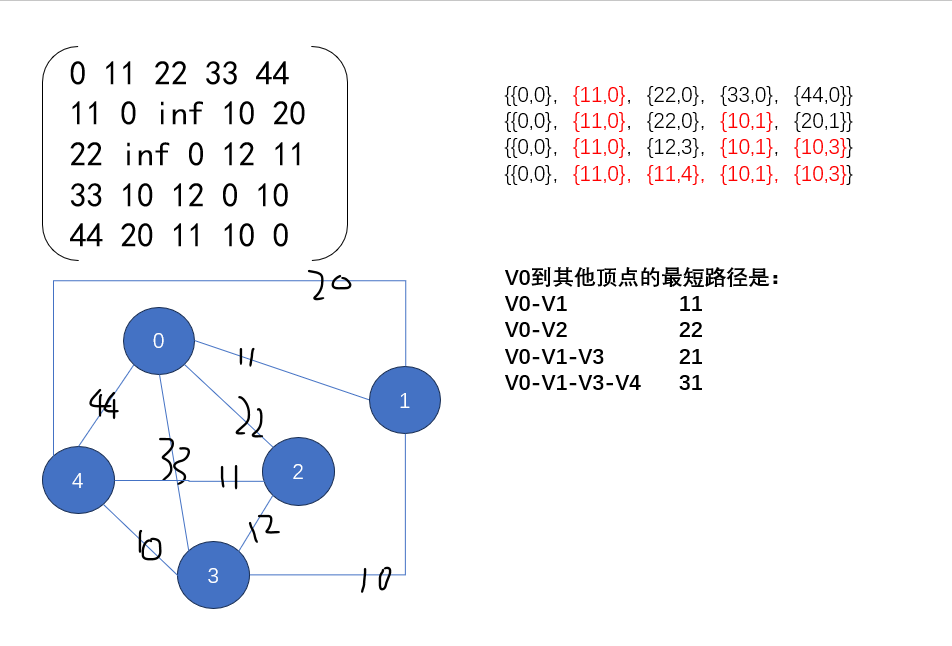
3、主函数

主函数中首先要输入顶点数量，然后输入邻接矩阵。输入邻接矩阵时，用空格或换行符分隔，"inf"或"INF"代表无穷大。输入完邻接矩阵后，需要验证邻接矩阵是否有效。如果邻接矩阵无效，则打印出错信息并退出程序。

接下来，主函数要输入源点，然后调用dijkstra函数计算从指定源点开始到其他顶点的最短路径。

2、算法设计

2.1医药物流配送网络的优化图示（示例：节点数为5，权值随便定义）



2.2医药物流配送网络优化的主要步骤：

1. 将仓库和零售药店看作图中的节点，建立一个图。如果两个节点之间存在配送路径，则在它们之间连一条边。每条边的权重为配送成本，可以根据距离、运输工具种类等因素来确定。
2. 对于每个仓库，计算出其到每个零售药店的最短路径及对应的配送成本。可以使用Dijkstra算法或Floyd算法来计算最短路径。
3. 对于每个零售药店，统计所有仓库到该零售药店的最短路径及对应的配送成本。选择其中代价最小的仓库作为供应商。

四、算法的精化与程序的实现（程序实现的核心算法）

（一）核心算法一

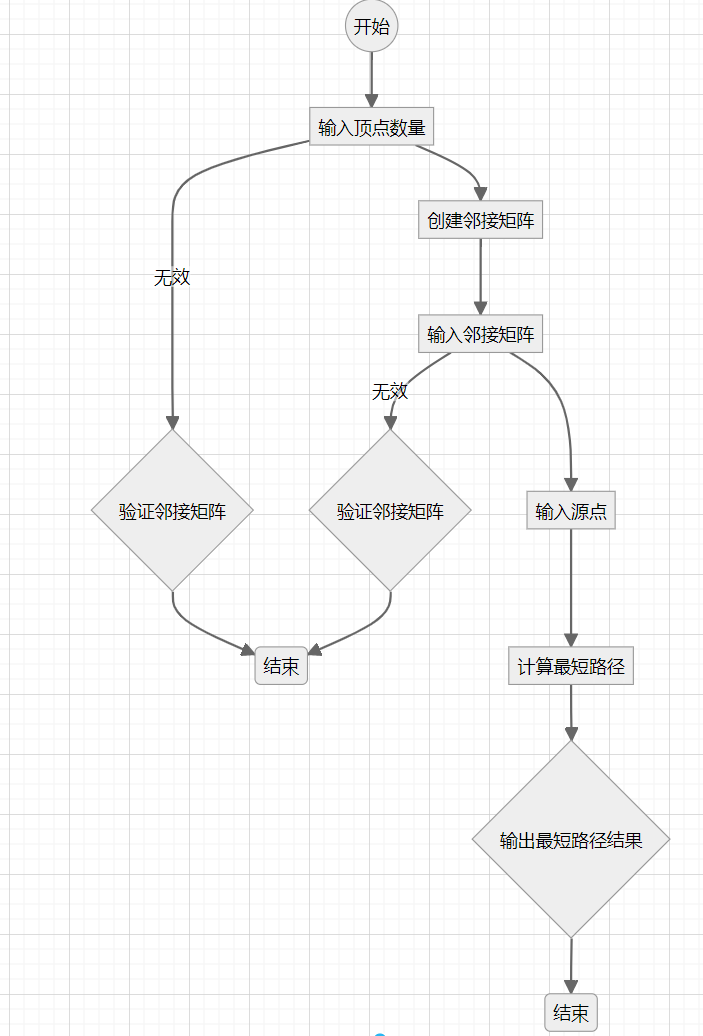
算法一：最短路径优先算法

通过输入顶点数量和邻接矩阵来构建图结构，然后根据给定的源点使用 Dijkstra 算法来计算从该源点到其他顶点的最短路径。

在数据结构设计方面使用邻接矩阵来描述顶点之间的连接关系和距离成本。邻接矩阵有效性的验证函数 validateMatrix 用于检查邻接矩阵是否符合要求。还使用了两个数组 dist 和 visited 分别存储最短距离和顶点的访问状态。

代码主要分为三部分：验证输入、构建邻接矩阵和计算最短路径。首先，用户需要提供顶点数量，并进行验证以确保输入的有效性。然后，用户需要输入邻接矩阵，其中无穷大的值可以用 "inf" 或 "INF" 表示。接着，对输入的邻接矩阵进行验证，如果无效则会输出错误信息并退出程序。最后，用户需要指定一个源点，并调用 dijkstra 函数来计算从该源点到其他顶点的最短路径，并输出最短距离。

* 流程图



* 算法实现代码

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

#include <string.h>

#define MAX\_V 10 // 最大顶点数量

#define INF INT\_MAX // 定义无穷大

// 计算从源点到目标点的最短路径

void dijkstra(int graph[MAX\_V][MAX\_V], int num\_vertices, int src) {

int dist[MAX\_V]; // 存储最短距离

bool visited[MAX\_V]; // 存储顶点是否被访问

// 初始化距离和访问数组

int i;

for (i = 0; i < num\_vertices; i++) {

dist[i] = INF;

visited[i] = false;

}

dist[src] = 0;

int count;

for (count = 0; count < num\_vertices - 1; count++) {

int min\_dist = INF, min\_index;

// 找到当前未访问顶点中距离最小的顶点

int v;

for (v = 0; v < num\_vertices; v++) {

if (!visited[v] && dist[v] <= min\_dist) {

min\_dist = dist[v];

min\_index = v;

}

}

int u = min\_index;

visited[u] = true;

// 更新最短路径距离

for (v = 0; v < num\_vertices; v++) {

if (!visited[v] && graph[u][v] != INF &&

dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

printf("顶点(医院或仓库或配送中心的编号) 到源点的最短距离\n");

for (i = 0; i < num\_vertices; i++) {

printf("%d \t %d\n", i, dist[i]);

}

}

int main() {

int num\_vertices;

printf("请输入顶点数量：");

if (scanf("%d", &num\_vertices) != 1 || num\_vertices <= 0 || num\_vertices > MAX\_V) {

printf("无效的顶点数量，请重新运行程序并输入有效的顶点数量。\n");

return 1;

}

int graph[MAX\_V][MAX\_V];

printf("请输入邻接矩阵（用空格或换行符分隔，inf或INF代表无穷大）：\n");

// 输入邻接矩阵

int i, j;

char input[10];

for (i = 0; i < num\_vertices; i++) {

for (j = 0; j < num\_vertices; j++) {

scanf("%s", input);

graph[i][j] = atoi(input);

}

}

int src;

printf("请输入源点(仓库或配送中心的编号)：");

scanf("%d", &src);

dijkstra(graph, num\_vertices, src); // 从指定源点开始计算最短路径

return 0;

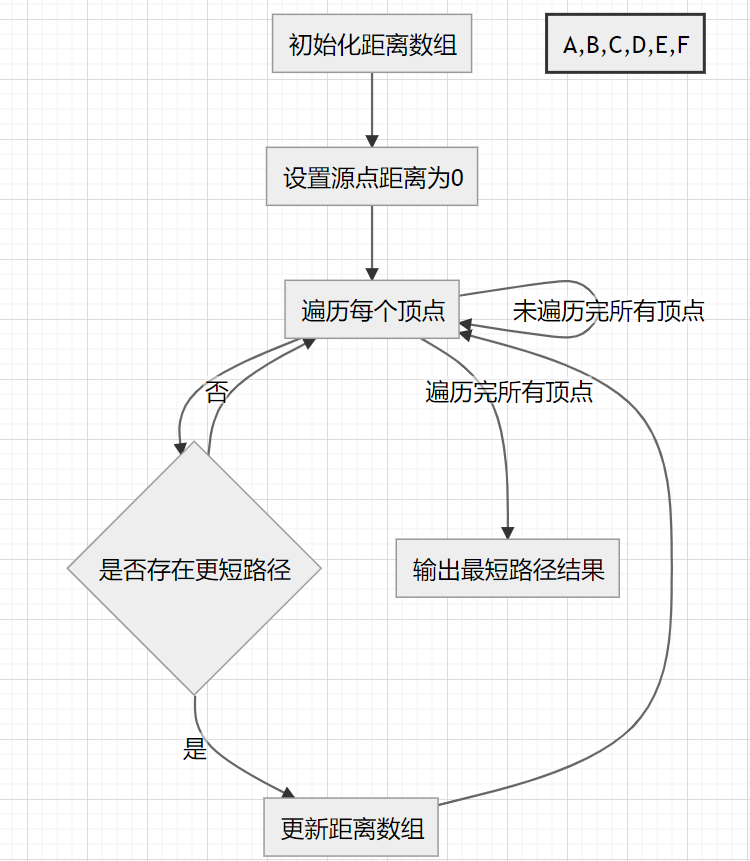
}

（二）核心算法二

算法二：动态规划算法

初始化一个二维数组 dp，用于存储各个顶点之间的最短路径长度。 对于任意两个顶点之间不存在边的情况，将其距离设为无穷大。 根据已知的边，将相邻顶点之间的距离存储在 dp 数组中。 使用动态规划算法，计算任意两个顶点之间的最短路径长度： 对于每个顶点对 (i, j)，遍历所有中间顶点 k： 如果通过顶点 k 可以使得从顶点 i 到达顶点 j 的路径更短，则更新最短路径值。 返回最终的 dp 数组，其中存储了各个顶点之间的最短路径长度。

* 流程图



* 算法实现代码

#include <stdio.h>

#include <limits.h>

#define MAX\_V 10 // 定义最大顶点数量

typedef struct {

int id;

char name[50];

// 其他属性...

} Vertex;

typedef struct {

int num\_vertices;

int adjacency\_matrix[MAX\_V][MAX\_V];

// 其他属性...

} Graph;

typedef struct {

int value;

// 其他属性...

} Distance;

void initGraph(Graph \*graph, int num\_vertices) {

graph->num\_vertices = num\_vertices;

for (int i = 0; i < num\_vertices; i++) {

for (int j = 0; j < num\_vertices; j++) {

graph->adjacency\_matrix[i][j] = 0;

}

}

}

void addVertex(Graph \*graph, Vertex vertex) {

// 添加顶点的操作...

}

void addEdge(Graph \*graph, int src, int dest, Distance distance) {

graph->adjacency\_matrix[src][dest] = distance.value;

// 添加边的操作...

}

void floydWarshall(Graph \*graph) {

int dist[MAX\_V][MAX\_V];

int i, j, k;

for (i = 0; i < graph->num\_vertices; i++) {

for (j = 0; j < graph->num\_vertices; j++) {

dist[i][j] = graph->adjacency\_matrix[i][j];

}

}

for (k = 0; k < graph->num\_vertices; k++) {

for (i = 0; i < graph->num\_vertices; i++) {

for (j = 0; j < graph->num\_vertices; j++) {

if (dist[i][k] != INT\_MAX && dist[k][j] != INT\_MAX &&

dist[i][k] + dist[k][j] < dist[i][j])

{

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

}

// 打印最短路径结果

printf("最短路径：\n");

for (i = 0; i < graph->num\_vertices; i++) {

for (j = 0; j < graph->num\_vertices; j++) {

if (dist[i][j] == INT\_MAX) {

printf("%s 到 %s 之间没有路径\n", graph->vertices[i].name, graph->vertices[j].name);

} else {

printf("%s 到 %s 的最短距离为 %d\n", graph->vertices[i].name, graph->vertices[j].name, dist[i][j]);

}

}

}

}

int main() {

Graph graph;

initGraph(&graph, MAX\_V);

// 添加顶点和边

// ...

floydWarshall(&graph);

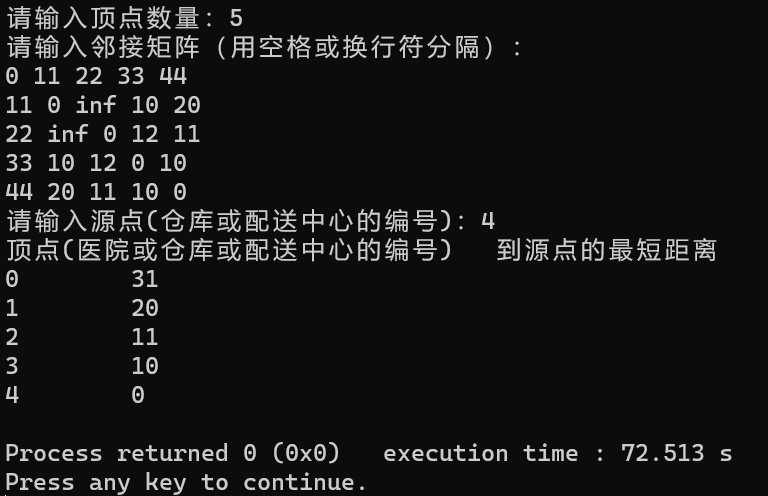
return 0;

}

五、程序的调试与计算的结果分析

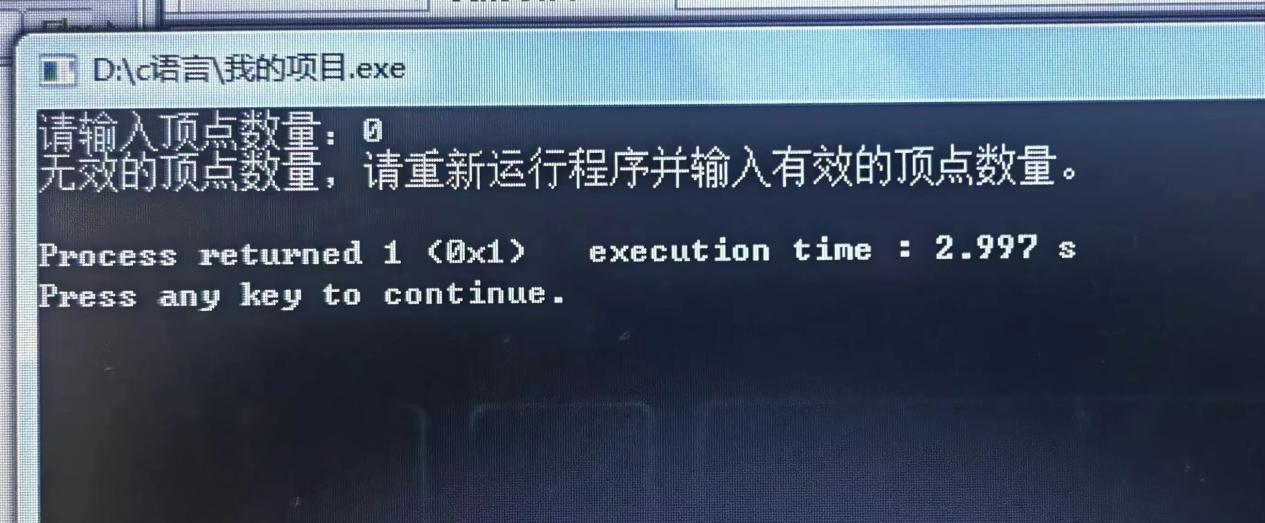
1. 调试情况

(1)运行情况

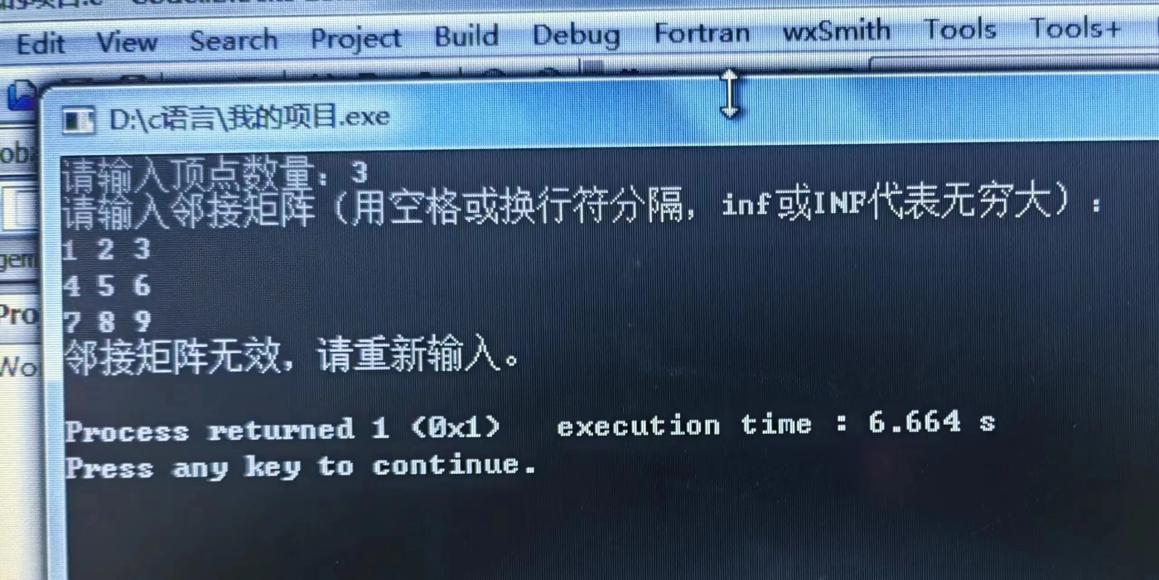


1. 异常处理

顶点输入的异常反馈：



矩阵输入的异常反馈：



1. 结果分析

(1)分别将顶点数量，邻接矩阵和源点编号输入到程序中去（inf表无穷大）

(2)程序可快速计算出各顶点到源点之间的最短距离

(3)经过验算也是如此

六．时间代价的分析

1. 图的规模：图中节点和边的数量会直接影响算法的执行时间。通常情况下，节点和边越多，算法的执行时间越长。

2. 图的表示方式：最短路径算法可以使用不同的图的表示方式，如邻接矩阵和邻接表。邻接矩阵适用于稠密图，而邻接表适用于稀疏图。邻接矩阵的时间代价是O(V^2)，其中V是节点的数量；邻接表的时间代价是O(V+E)，其中E是边的数量。

3. 优化技巧：针对特定情况，可以使用一些优化技巧来减少算法的执行时间。例如，使用最小堆（或优先队列）来优化Dijkstra算法，以减少节点的访问次数。

七、有待改进的问题，收获和体会

1、有待改进的问题

算法设计，流程图绘制：

1.距离数组和访问状态数组的初始化：算法中提到要对距离数组和访问状态数组进行初始化，但未给出具体的初始化方法。缺乏对这两个数组初始化的详细说明可能导致无法正确计算最短路径。

2.使用动态内存分配：当前代码使用静态的二维矩阵来存储邻接矩阵，这限制了顶点数量的上限。可以考虑使用动态内存分配来支持更大的顶点数量。

3.鲁棒性不足：当前代码没有对异常情况进行充分的处理。例如，如果邻接矩阵无效或输入格式错误，代码将简单地打印错误信息，而不会提供更详细的错误提示。

2、收获和体会

算法设计，流程图绘制：

1.距离数组和访问状态数组的初始化非常重要，因为它们对算法的正确性起着关键作用。在距离数组中，可以将源点的距离初始化为0，其他顶点的距离初始化为无穷大，表示尚未确定最短路径。在访问状态数组中，可以将所有顶点的状态初始化为未访问，即标记为false。这样做可以确保算法从正确的起点开始，并且在遍历过程中能够正常更新距离值。

2.处理负权边是一个重要的问题。在一些情况下，图中存在负权边可能导致最短路径算法产生错误的结果或进入无限循环。一种常见的解决方案是使用适当的最短路径算法来处理负权边，例如Dijkstra算法不能处理负权边，而Bellman-Ford算法可以处理负权边。根据具体情况选择合适的算法以处理负权边是至关重要的。

3.缺乏终止条件的定义可能导致算法无法正确结束执行。在最短路径优先算法中，一种常见的终止条件是当所有顶点都被访问过，或者找不到更短的路径时，算法可以结束执行并返回最短路径数组。这个终止条件可以确保算法在找到最优解后正确地停止，并且不会陷入无限循环。