暨南大学本科上机课实验报告

学生姓名：肖健成　　学号：2021103268

学院：网络空间安全　　专业：网络空间安全

课程名称:《算法分析与设计》　　实验名称：基于多段图问题的动态规划算法设计

实验类型：验证-设计-综合　　实验时间：2023 年 11 月 3 日晚　　地点：N516

上机内容：2.4
利用动态规划策略设计具有数据输入、处理和输出功能的多段图规划问题：求源点 S 到汇点 T 的最小成本路径的求解算法，实现算法的编译和运行，记录实验过程并整理实验结果。根据原理图，连接实验线路。

算法描述：

多段图问题的动态规划算法是一种用于解决图论中多段有向图的最短路径问题的方法。这个问题通常涉及到一个有向图，分为多个段，每个段内的结点可以互相到达，但段与段之间的结点只能单向移动。目标是找到从起点到终点的最短路径，使得总成本最小。

多段图问题通常用向前处理和向后处理两种算法来解决。

问题描述：

有向图：首先，给定一个有向图，包括结点和边，其中每条边都有一个成本（权重）。多段：图被分成多个段，每个段包含一组结点，结点之间可以互相到达，但段与段之间的结点只能单向移动。

起点和终点：问题的目标是找到从起点到终点的最短路径，使得总成本最小。

先前处理算法：

向前处理算法是一种动态规划算法，它从起点出发，逐段段地计算到达每个结点的最小成本，并维护每个结点的前驱结点，以便最后构建最短路径。

算法采用两层循环。外层循环遍历所有段（从第一段到最后一段），内层循环遍历每个段内的结点。

对于每个结点，算法尝试从前一段的所有结点出发，计算到达当前结点的最小成本。如果找到更短的路径，就更新最小成本和前驱结点。

最终，算法计算出从起点到每个结点的最小成本和相应的前驱结点，从而构建出最短路径。

步骤：

1.初始化：设置起点的成本为 0，其他结点的成本为无穷大（或足够大）。

2.外层循环：遍历每个段，从第一段到最后一段。

3.内层循环：遍历当前段内的每个结点。

4.对于每个结点，计算从前一段的所有结点出发到达当前结点的成本，选择最小的成本，并更新成本和前驱结点。

5.最后，根据前驱结点信息，构建最短路径。

算法的复杂度：

时间复杂度：多段图向前处理算法的时间复杂度是 O(V^2)，其中 V 是结点的数量。

空间复杂度：空间复杂度取决于存储成本和前驱结点的数据结构，通常是 O(V)。

向后处理算法：

问题描述：

有向图：与向前处理算法一样，首先给定一个有向图，包括结点和边，每条边都有一个成本（权重）。

多段：图被分成多个段，每个段包含一组结点，结点之间可以互相到达，但段与段之间的结点只能单向移动。

起点和终点：问题的目标是找到从起点到终点的最短路径，使得总成本最小。

算法概述：

向后处理算法从终点出发，逐段段地逆向计算到达每个结点的最小成本，并维护每个结点的后继结点，以便最后构建最短路径。

算法同样采用两层循环。外层循环遍历所有段（从最后一段到第一段），内层循环遍历每个段内的结点。

对于每个结点，算法尝试从后一段的所有结点出发，计算从当前结点到达终点的最小成本。如果找到更短的路径，就更新最小成本和后继结点。

最终，算法计算出从起点到每个结点的最小成本和相应的后继结点，从而构建出最短路径。

步骤：

1.初始化：设置终点的成本为 0，其他结点的成本为无穷大（或足够大）。

2.外层循环：遍历每个段，从最后一段到第一段。

3.内层循环：遍历当前段内的每个结点。

4.对于每个结点，计算从后一段的所有结点出发到达终点的成本，选择最小的成本，并更新成本和后继结点。

5.最后，根据后继结点信息，构建最短路径。

算法的复杂度：

时间复杂度：多段图向后处理算法的时间复杂度也是 O(V^2)，其中 V 是结点的数量。

空间复杂度：与向前处理算法类似，取决于存储成本和后继结点的数据结构，通常是 O(V)。

下面是伪代码部分

向前处理算法：

```
function ForwardProcessingAlgorithm(graph, start, end):
    nodeCount = graph.numberOfNodes  // 获取图中结点的数量
    costList = initialize an array of size nodeCount with all elements set to infinity
    predecessorList = initialize an array of size nodeCount with all elements set to -1 (indicating no predecessor)

    costList[start] = 0  // 起点到自身的成本为 0
```

```
    for segment from 1 to nodeCount - 1:  // 从第一段到倒数第二段
        for currentNode from 0 to nodeCount - 1:  // 遍历当前段内的所有结点
            for previousNode from 0 to nodeCount - 1:  // 遍历前一段的所有结点
                if there is an edge from previousNode to currentNode in the current segment:  // 如果在当前段内存在从前一段到当前结点的边
                    potentialCost = costList[previousNode] + cost of the edge from previousNode to currentNode  // 计算从前一段到当前结点的潜在成本
                    if potentialCost < costList[currentNode]:  // 如果潜在成本小于当前已知成本
                        costList[currentNode] = potentialCost  // 更新当前结点的最小成本
                        predecessorList[currentNode] = previousNode  // 记录当前结点的前驱结点

    // Reconstruct the shortest path
    path = []  // 初始化用于存储最短路径的列表
    currentNode = end  // 从终点开始反向构建路径
    while currentNode != start:  // 当前结点不是起点时
        path.prepend(currentNode)  // 将当前结点添加到路径的开头
        currentNode = predecessorList[currentNode]  // 获取当前结点的前驱结点
    path.prepend(start)  // 将起点添加到路径的开头

    return path, costList[end]  // 返回最短路径和最小成本
```

向后处理算法：

```
function BackwardProcessingAlgorithm(graph, start, end):
    nodeCount = graph.numberOfNodes  // 获取图中结点的数量
    costList = initialize an array of size nodeCount with all elements set to infinity
    successorList = initialize an array of size nodeCount with all elements set to -1 (indicating no successor)

    costList[end] = 0  // 终点到自身的成本为 0

    for segment from nodeCount - 2 down to 0:  // 从倒数第二段到第一段，反向遍历每个段
        for currentNode from 0 to nodeCount - 1:  // 遍历当前段内的所有结点
            for nextNode from 0 to nodeCount - 1:  // 遍历下一段的所有结点
                if there is an edge from currentNode to nextNode in the current segment:  // 如果在当前段内存在从当前结点到下一段的边
                    potentialCost = costList[nextNode] + cost of the edge from currentNode to nextNode  // 计算从当前结点到下一段的潜在成本
                    if potentialCost < costList[currentNode]:  // 如果潜在成本小于当前已知成本
                        costList[currentNode] = potentialCost  // 更新当前结点的最小成本
```

successorList[currentNode] = nextNode // 记录当前结点的后继结点

// Reconstruct the shortest path
path = [] // 初始化用于存储最短路径的列表
currentNode = start // 从起点开始正向构建路径
while currentNode != end: // 当前结点不是终点时
    path.append(currentNode) // 将当前结点添加到路径
    currentNode = successorList[currentNode] // 获取当前结点的后继结点
path.append(end) // 将终点添加到路径

return path, costList[start] // 返回最短路径和最小成本

源程序：

```
#include <stdio.h>
#define Max_Node 20
#define Max_Cost 100

// 有向图的结构定义
typedef struct DirectedGraph {
    int edges[Max_Node][Max_Node]; // 邻接矩阵表示的有向图的边信息
    int NodeCount; // 结点数量
    int EdgeCount; // 边数量
};

// 初始化有向图
DirectedGraph InitializeGraph();
// 从后向前计算最小成本路径
void CalculateBackwardMinCostPath(DirectedGraph g, int costList[], int
predecessorList[]);
// 从前向后计算最小成本路径
void CalculateForwardMinCostPath(DirectedGraph g, int costList[], int
successorList[]);
// 初始化前向成本列表
void InitializeForwardCostList(DirectedGraph g, int costList[]);
// 初始化后向成本列表
void InitializeBackwardCostList(DirectedGraph g, int costList[]);
// 显示有向图
void DisplayGraph(DirectedGraph g);
// 显示从前向后的最小成本路径
void DisplayForwardMinCostPath(DirectedGraph g, int predecessorList[], int
costList[]);
// 显示从后向前的最小成本路径
```

```
void DisplayBackwardMinCostPath(DirectedGraph g, int successorList[], int
costList[]);

int main() {
    int costList[Max_Node], predecessorList[Max_Node] = { 0 };
    DirectedGraph G = InitializeGraph();
    InitializeForwardCostList(G, costList);
    CalculateForwardMinCostPath(G, costList, predecessorList);
    DisplayForwardMinCostPath(G, predecessorList, costList);
    printf("\n");
    InitializeBackwardCostList(G, costList);
    CalculateBackwardMinCostPath(G, costList, predecessorList);
    DisplayBackwardMinCostPath(G, predecessorList, costList);
    return 0;
}

// 初始化有向图
DirectedGraph InitializeGraph() {
    DirectedGraph g;
    int nodes = 0;
    int edges = 0;
    printf("Enter the number of nodes and edges (separated by a space):\n");
    scanf_s("%d %d", &nodes, &edges);
    g.NodeCount = nodes;
    g.EdgeCount = edges;
    for (int i = 0; i < Max_Node; i++) {
        for (int j = 0; j < Max_Node; j++) {
            g.edges[i][j] = 0; // 初始化所有边的成本为 0
        }
    }
    int x, y, cost;
    for (int i = 0; i < edges; i++) {
        printf("Enter the start node, edge cost, and end node for edge %d:\n", i +
1);
        scanf_s("%d %d %d", &x, &cost, &y);
        g.edges[x][y] = cost; // 设置边的成本
    }
    DisplayGraph(g);
    return g;
}

// 从后向前计算最小成本路径
void CalculateBackwardMinCostPath(DirectedGraph g, int costList[], int
predecessorList[]) {
```

```
    for (int j = 1; j < g.NodeCount; j++) { // 外层循环遍历除起点外的所有结点
        for (int i = 0; i < j; i++) { // 内层循环从前向后遍历所有可能的前趋结点
            if (g.edges[i][j] != 0 && (costList[i] + g.edges[i][j] < costList[j])) {
                costList[j] = costList[i] + g.edges[i][j]; // 如果两结点相通且到达结点 i 的成本+i 到 j 的成本 < 到达结点 j 的成本
                predecessorList[j] = i; // j 点的前一个点是 i
            }
        }
    }
}

// 从前向后计算最小成本路径
void CalculateForwardMinCostPath(DirectedGraph g, int costList[], int successorList[]) {
    for (int j = g.NodeCount - 1; j >= 0; j--) { // 外层循环遍历除终点外的所有结点
        for (int i = g.NodeCount - 2; i >= 0; i--) { // 内层循环从后向前遍历所有可能的后继结点
            if (g.edges[i][j] != 0 && (costList[j] + g.edges[i][j] < costList[i])) {
                costList[i] = costList[j] + g.edges[i][j]; // 如果两结点相通且结点 j 到终点的成本+i 到 j 的成本 < 结点 i 到终点的成本
                successorList[i] = j; // i 点的后一个点是 j
            }
        }
    }
}

// 初始化后向成本列表
void InitializeBackwardCostList(DirectedGraph g, int costList[]) {
    costList[0] = 0; // 起始节点到达自身的成本为 0
    for (int i = 1; i < g.NodeCount; i++) {
        costList[i] = Max_Cost; // 将成本初始化为最大成本
    }
}

// 初始化前向成本列表
void InitializeForwardCostList(DirectedGraph g, int costList[]) {
    costList[g.NodeCount - 1] = 0; // 终止节点到达自身的成本为 0
    for (int i = 0; i < g.NodeCount - 1; i++) {
        costList[i] = Max_Cost; // 将成本初始化为最大成本
    }
}

// 显示从后向前的最小成本路径
void DisplayBackwardMinCostPath(DirectedGraph g, int successorList[], int costList[])
```

```
{
    printf("Backward Processing Algorithm:\n");
    for (int i = 0; i < g.NodeCount; i++) {
        printf("Minimum cost to reach node %d is %d, and the corresponding path
is: %d", i, costList[i], i);
        int predecessorNode = successorList[i]; // 此结点的前驱结点
        while (predecessorNode != 0) { // 有向图的起点为 0
            printf(" <- %d", predecessorNode);
            predecessorNode = successorList[predecessorNode]; // 更新“最新”结点，最
新的结点为 connect_list[旧结点]
        }
        printf(" <- 0\n");
    }
}

// 显示从前向后的最小成本路径
void DisplayForwardMinCostPath(DirectedGraph g, int predecessorList[], int
costList[]) {
    printf("Forward Processing Algorithm:\n Minimum cost to reach the end node
is: %d, and the corresponding path is: 0", costList[0]); // 输出起点及其到达终点的成
本
    int successorNode = predecessorList[0]; // 起点的后继结点
    while (successorNode != g.NodeCount - 1) { // 有向图的终点为最后一个结点
        printf(" -> %d", successorNode);
        successorNode = predecessorList[successorNode]; // 更新“最新”结点，最新的结
点为 connect_list[旧结点]
    }
    printf(" -> %d\n", successorNode);
}

// 显示有向图
void DisplayGraph(DirectedGraph g) {
    printf("Adjacency matrix of the directed graph:\n");
    for (int i = 0; i < g.NodeCount; i++) {
        for (int j = 0; j < g.NodeCount; j++) {
            printf("%2d\t", g.edges[i][j]);
            if (j == g.NodeCount - 1) { // 矩阵转行
                printf("\n");
            }
        }
    }
}
```

复杂度分析：

多段图向前处理算法

时间复杂度分析：

多段图向前处理算法的时间复杂度也主要由两部分组成：

第一部分：计算从源点到各个顶点的最短路径长度，同样包括两层嵌套循环。外层循环执行 m-1 次，内层循环对所有入边进行计算，每条入边只计算一次。如果图的边数为 n，则该部分的时间复杂度为 O(n)。

第二部分：输出最短路径经过的顶点。如果多段图划分为 k 段，该部分的时间复杂度为 O(k)。

综上所述，多段图向前处理算法的时间复杂度为 O(n + k)。

空间复杂度分析：

多段图向前处理算法同样使用了两个额外的空间来存储相关数据信息。因此，空间复杂度为 O(n)。

多段图向后处理算法

时间复杂度分析：

多段图向后处理算法的时间复杂度主要由两部分组成：

第一部分：计算从源点到各个顶点的最短路径长度。它包括两层嵌套循环。外层循环执行 m-1 次，内层循环对所有入边进行计算。在所有循环中，每条入边只计算一次。如果图的边数为 n，则该部分的时间复杂度为 O(n)。

第二部分：输出最短路径经过的顶点。如果多段图划分为 k 段，该部分的时间复杂度为 O(k)。

综上所述，多段图向后处理算法的时间复杂度为 O(n + k)。

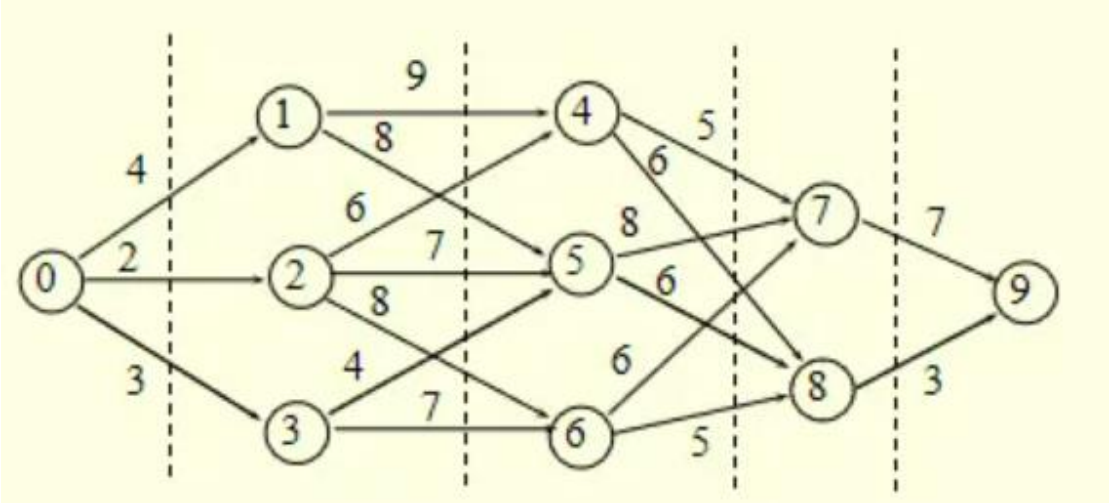
空间复杂度分析：

多段图向后处理算法使用了两个额外的空间来存储相关数据信息。因此，空间复杂度为 O(n)。

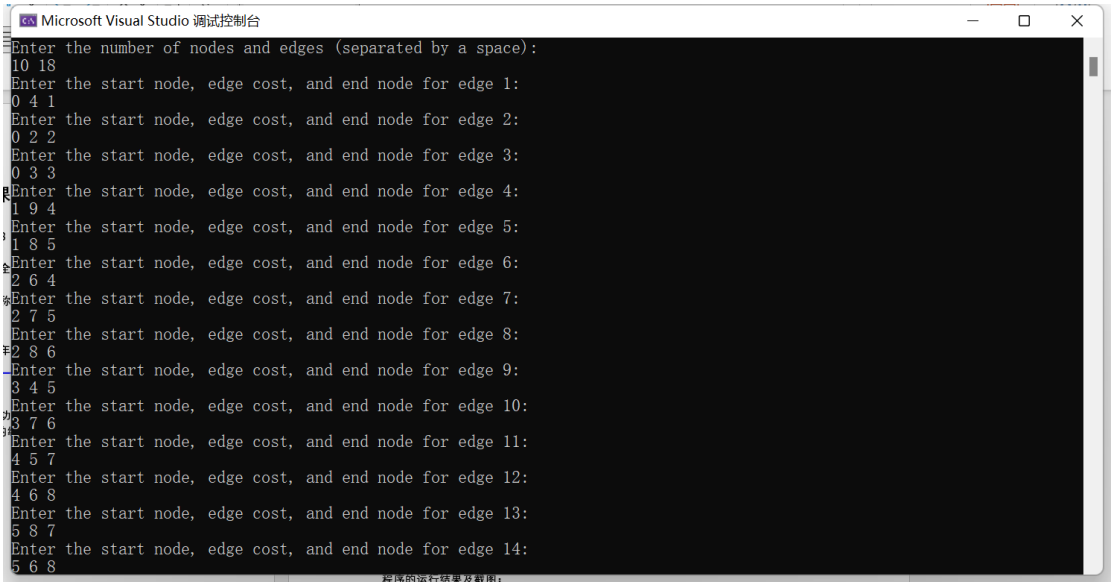
向前处理时间复杂度	向前处理空间复杂度	向后处理时间复杂度	向后处理空间复杂度
O(n+k)	O(n)	O(n+k)	O(n)

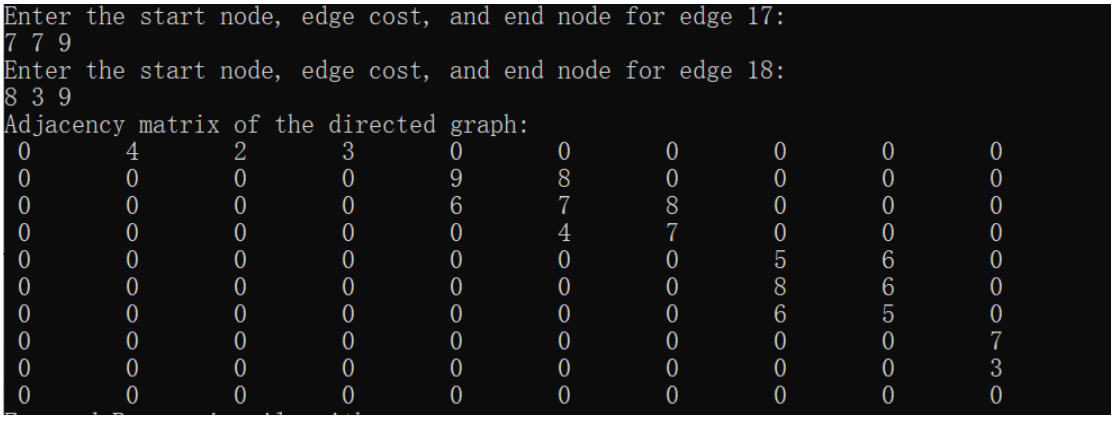
程序的运行结果及截图：

测试 1：



我们以这个图作为测试案例





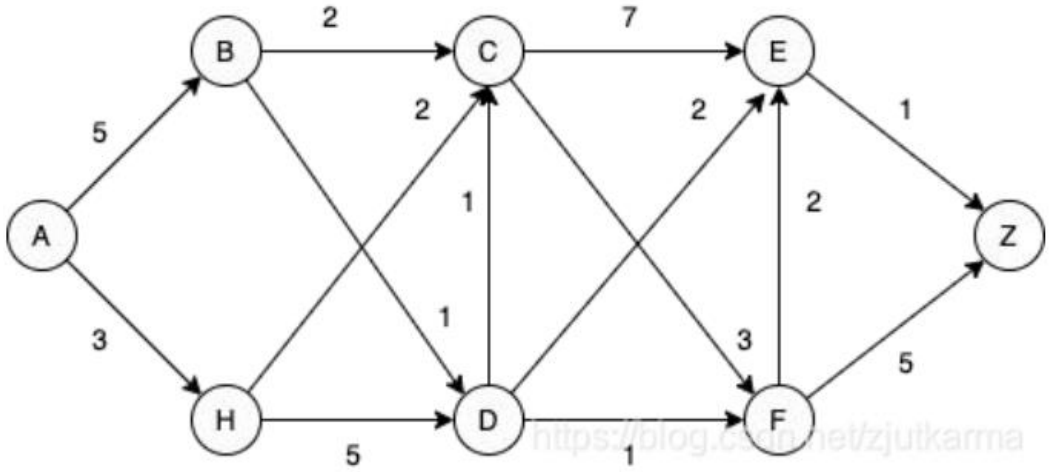
这是输入与生成邻接矩阵部分

```
Forward Processing Algorithm:
 Minimum cost to reach the end node is: 16, and the corresponding path is: 0 -> 3 -> 5 -> 8 -> 9

Backward Processing Algorithm:
Minimum cost to reach node 0 is 0, and the corresponding path is: 0 <- 3 <- 0
Minimum cost to reach node 1 is 4, and the corresponding path is: 1 <- 0
Minimum cost to reach node 2 is 2, and the corresponding path is: 2 <- 0
Minimum cost to reach node 3 is 3, and the corresponding path is: 3 <- 0
Minimum cost to reach node 4 is 8, and the corresponding path is: 4 <- 2 <- 0
Minimum cost to reach node 5 is 7, and the corresponding path is: 5 <- 3 <- 0
Minimum cost to reach node 6 is 10, and the corresponding path is: 6 <- 2 <- 0
Minimum cost to reach node 7 is 13, and the corresponding path is: 7 <- 4 <- 2 <- 0
Minimum cost to reach node 8 is 13, and the corresponding path is: 8 <- 5 <- 3 <- 0
Minimum cost to reach node 9 is 16, and the corresponding path is: 9 <- 8 <- 5 <- 3 <- 0

D:\vs work\Project44\x64\Debug\Project44.exe (进程 23716)已退出，代码为 0。
按任意键关闭此窗口. . .
```

这是输出结果部分

实验结果正确。

测试 2：



我们以这个图作为测试案例

```
Enter the number of nodes and edges (separated by a space):
8 13
Enter the start node, edge cost, and end node for edge 1:
0 5 1
Enter the start node, edge cost, and end node for edge 2:
0 3 6
Enter the start node, edge cost, and end node for edge 3:
1 2 2
Enter the start node, edge cost, and end node for edge 4:
1 1 3
Enter the start node, edge cost, and end node for edge 5:
2 7 4
Enter the start node, edge cost, and end node for edge 6:
2 3 5
Enter the start node, edge cost, and end node for edge 7:
3 1 2
Enter the start node, edge cost, and end node for edge 8:
3 2 4
Enter the start node, edge cost, and end node for edge 9:
3 1 5
Enter the start node, edge cost, and end node for edge 10:
4 1 7
Enter the start node, edge cost, and end node for edge 11:
5 2 4
Enter the start node, edge cost, and end node for edge 12:
5 5 7
Enter the start node, edge cost, and end node for edge 13:
6 2 2
```

```
Adjacency matrix of the directed graph:
 0       5       0       0       0       0       3       0
 0       0       2       1       0       0       0       0
 0       0       0       0       7       3       0       0
 0       0       1       0       2       1       0       0
 0       0       0       0       0       0       0       1
 0       0       0       0       2       0       0       5
 0       0       2       0       0       0       0       0
 0       0       0       0       0       0       0       0
```

这是输入与生成邻接矩阵部分

```
Forward Processing Algorithm:
 Minimum cost to reach the end node is: 9, and the corresponding path is: 0 -> 1 -> 3 -> 4 -> 7

Backward Processing Algorithm:
Minimum cost to reach node 0 is 0, and the corresponding path is: 0 <- 1 <- 0
Minimum cost to reach node 1 is 5, and the corresponding path is: 1 <- 0
Minimum cost to reach node 2 is 7, and the corresponding path is: 2 <- 1 <- 0
Minimum cost to reach node 3 is 6, and the corresponding path is: 3 <- 1 <- 0
Minimum cost to reach node 4 is 8, and the corresponding path is: 4 <- 3 <- 1 <- 0
Minimum cost to reach node 5 is 7, and the corresponding path is: 5 <- 3 <- 1 <- 0
Minimum cost to reach node 6 is 3, and the corresponding path is: 6 <- 0
Minimum cost to reach node 7 is 9, and the corresponding path is: 7 <- 4 <- 3 <- 1 <- 0
```

这是输出结果部分
实验结果正确。

个人总结：
向前处理算法：

概述：向前处理算法是解决多段图问题的一种动态规划算法。它的目标是找到从起点到终点的最短路径，考虑了多段图的特殊结构。

操作过程：向前处理算法首先将多段图按照段数划分，并根据问题的特点，从终点逆向计算每段中各结点的最小成本路径。它通过动态规划的方式逐段计算每段内的最短路径。

时间复杂度：O(n + k)，其中 n 是图的边数，k 是多段图的段数。

空间复杂度：O(n)，需要存储两个额外的数组用于成本和前驱结点。

特点：向前处理算法适用于多段图的情况，其中段之间的成本受限制。它的时间复杂度较低，适用于解决小到中等规模的多段图问题。

向后处理算法：

概述：向后处理算法也是解决多段图问题的一种动态规划算法。它的目标是找到从起点到终点的最短路径，同样考虑多段图的特殊结构。

操作过程：向后处理算法将多段图按段数划分，然后从起点开始正向计算每段中各结点的最小成本路径。它通过动态规划的方式逐段计算每段内的最短路径。

时间复杂度：O(n + k)，其中 n 是图的边数，k 是多段图的段数。

空间复杂度：O(n)，需要存储两个额外的数组用于成本和后继结点。

特点：向后处理算法同样适用于多段图的情况，其中段之间的成本受限制。它的时间复杂度较低，适用于解决小到中等规模的多段图问题。

向前处理算法和向后处理算法都是有效解决多段图问题的动态规划算法，选择哪种算法取决于问题的具体特点和图的规模。它们都考虑了多段图的结构，以提高解决问题的效率。

在这次的实验中，我温习了数据结构，实践了使用动态规划法解决多段图问题，使我受益良多。