图论是数学的一个分支，它研究图的性质和图之间的关系。图由节点（顶点）和边组成，而图论的最短路径问题是其中一个重要的算法问题。最短路径问题的目标是找到图中两个节点之间的最短路径，即连接这两个节点的路径中权重之和最小的路径。

最短路径问题有两种主要的情景：单源最短路径和全源最短路径。

1. 单源最短路径问题

单源最短路径问题是指在图中找到从一个特定的源节点到图中所有其他节点的最短路径。经典的算法包括：

1.1 Dijkstra算法

Dijkstra算法是一种贪心算法，通过维护一个优先队列来逐步确定从源节点到其他节点的最短路径。算法步骤如下：

a. 初始化源节点的距离为0，其他节点的距离为无穷大。

b. 从未处理的节点中选择距离最小的节点，标记为已处理。

c. 更新与该节点相邻的节点的距离，如果新路径更短，则更新距离。

d. 重复步骤b和c，直到所有节点都被处理。

```
#include <iostream>
#include <limits.h>
#include <vector>

using namespace std;

// 定义无穷大
#define INF INT_MAX

// Dijkstra算法函数
void dijkstra(vector<vector<int>>& graph, int start) {
    int num_nodes = graph.size();

    // 初始化距离数组，表示从起始节点到各节点的距离
    vector<int> distances(num_nodes, INF);
    distances[start] = 0;

    // 记录节点是否已经被访问
    vector<bool> visited(num_nodes, false);
```

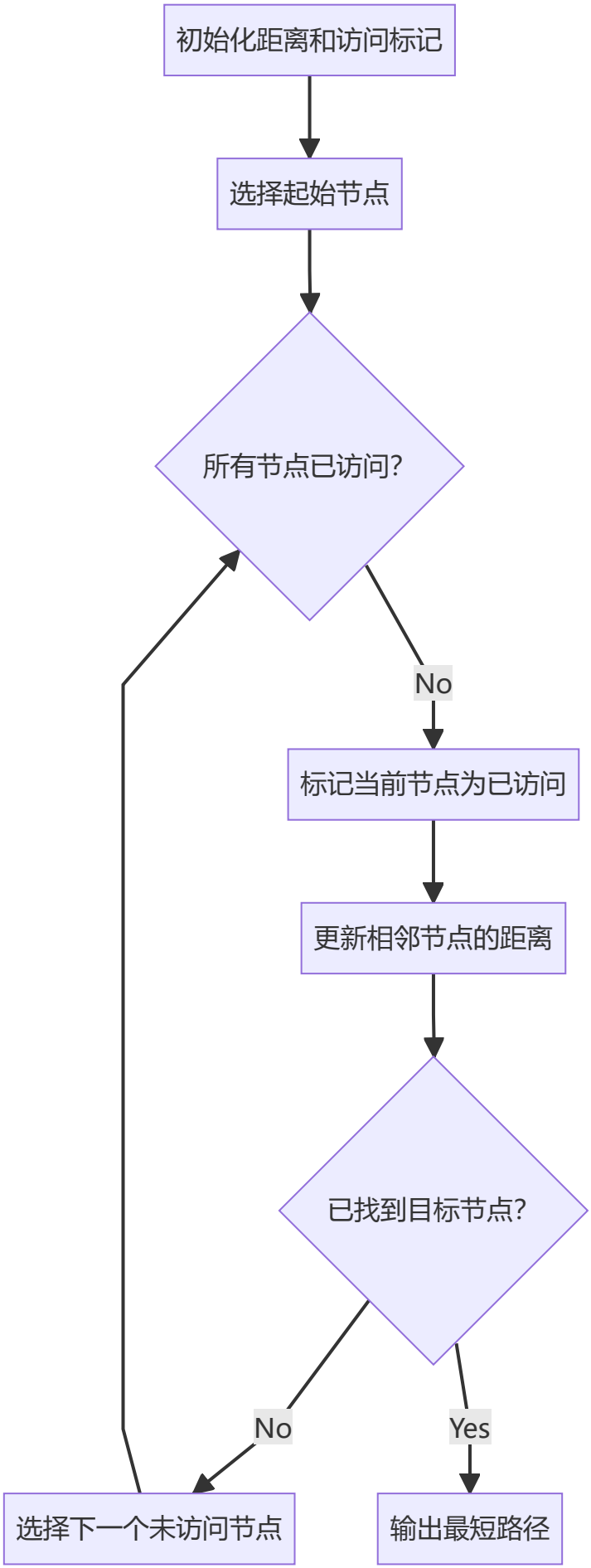
```
    for (int _ = 0; _ < num_nodes; ++_) {
        // 从未处理的节点中选择距离最小的节点
        int min_distance = INF;
        int min_index = -1;

        for (int i = 0; i < num_nodes; ++i) {
            if (!visited[i] && distances[i] < min_distance) {
                min_distance = distances[i];
                min_index = i;
            }
        }

        // 标记选择的节点为已处理
        visited[min_index] = true;

        // 更新与选择节点相邻的节点的距离
        for (int j = 0; j < num_nodes; ++j) {
            if (!visited[j] && graph[min_index][j] != 0 &&
distances[min_index] != INF &&
                distances[min_index] + graph[min_index][j] <
distances[j]) {
                distances[j] = distances[min_index] +
graph[min_index][j];
            }
        }
    }

    // 输出结果
    for (int i = 0; i < num_nodes; ++i) {
        cout << "从节点 " << start << " 到节点 " << i << " 的最
短距离为 " << distances[i] << endl;
    }
}

int main() {
    // 例子
    // 表示一个带权有向图的邻接矩阵
    vector<vector<int>> graph = {
        {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
```

```
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };

    int start_node = 0;
    dijkstra(graph, start_node);

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits.h>

using namespace std;

// 边的结构体，包含目标节点和权重
struct Edge {
    int target;
    int weight;

    Edge(int t, int w) : target(t), weight(w) {}
};

// Dijkstra算法函数
void dijkstra(vector<vector<Edge>>& graph, int start) {
    int num_nodes = graph.size();

    // 初始化距离数组，表示从起始节点到各节点的距离
    vector<int> distances(num_nodes, INT_MAX);
    distances[start] = 0;

    // 创建小顶堆，存储节点和距离的pair
    priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> minHeap;
    minHeap.push({0, start});
```

```
    while (!minHeap.empty()) {
        // 取出当前距离最小的节点
        int current_distance = minHeap.top().first;
        int current_node = minHeap.top().second;
        minHeap.pop();

        // 更新相邻节点的距离
        for (const Edge& edge : graph[current_node]) {
            int neighbor = edge.target;
            int weight = edge.weight;

            if (current_distance + weight <
distances[neighbor]) {
                distances[neighbor] = current_distance +
weight;
                minHeap.push({distances[neighbor], neighbor});
            }
        }
    }

    // 输出结果
    for (int i = 0; i < num_nodes; ++i) {
        cout << "从节点 " << start << " 到节点 " << i << " 的最
短距离为 " << distances[i] << endl;
    }
}

int main() {
    // 例子
    // 表示一个带权有向图的邻接表
    vector<vector<Edge>> graph = {
        {{1, 4}, {7, 8}},
        {{0, 4}, {2, 8}, {7, 11}},
        {{1, 8}, {3, 7}, {5, 4}, {8, 2}},
        {{2, 7}, {4, 9}, {5, 14}},
        {{3, 9}, {5, 10}},
        {{2, 4}, {3, 14}, {4, 10}, {6, 2}},
        {{5, 2}, {8, 6}, {7, 1}},
        {{0, 8}, {1, 11}, {6, 1}, {8, 7}},
        {{2, 2}, {6, 6}, {7, 7}}
    };
```

```
    int start_node = 0;
    dijkstra(graph, start_node);

    return 0;
}
```


初始化距离和访问标记
选择起始节点
所有节点已访问?
No
标记当前节点为已访问
更新相邻节点的距离
已找到目标节点?
No
Yes
选择下一个未访问节点
输出最短路径

1.2 Bellman-Ford算法

Bellman-Ford算法适用于含有负权边的图。它通过对每条边进行松弛操作来逐步逼近最短路径。算法步骤如下：

a. 初始化源节点的距离为0，其他节点的距离为无穷大。

b. 对每条边进行松弛操作，即尝试通过该边缩短路径。

c. 重复步骤b，直到没有路径可以被缩短。

2. 全源最短路径问题

全源最短路径问题是指在图中找到任意两个节点之间的最短路径。最著名的算法是：

2.1 Floyd-Warshall算法

Floyd-Warshall算法使用动态规划的思想，通过中间节点逐步更新所有节点对之间的最短路径。算法步骤如下：

a. 初始化路径矩阵，使得矩阵的元素表示直接相连的节点的路径长度，不相连的节点路径长度为无穷大。

b. 对每个中间节点进行遍历，尝试通过该中间节点缩短路径。

c. 重复步骤b，直到所有可能的中间节点都被考虑。

这些算法在不同场景下有不同的适用性和性能表现，选择合适的算法取决于图的特性以及问题的具体要求。