

数据结构 Project 项目报告

【城市交通路径规划系统】



学生姓名： 邓必成

学 号： 23307130144

专 业： 计算机科学与技术

日 期： 2024/12/28

一、项目内容

(1) 基本要求:

1. 实现基础数据结构

设计记录城市交通网络的数据结构，并设计记录车辆信息的数据结构。

2. 为多个车辆提供最短时间规划

给定多个车辆的起始地点和目的地点，系统需求得所有车辆从起点到终点的时间之和最短。车辆每单位时间可以行进的道路长度为1。

(2) 高级要求: 在进行路径规划时，需要考虑道路的**最大车流量**。如果某条道路的车流量超过了其最大承载能力(例如，某条道路已经饱和，无法再容纳更多车辆)，则该道路将被排除。因此，需要系统在基础要求的基础上增加对道路车流量的考虑，以确保交通网络的流畅性。

二、解题思路与重要算法模块展示

(1) 解题思路综述: 考虑到高级要求的存在，我在写源文件代码时兼顾了初级和高级要求，使得程序能同时解决满足这两个要求的案例。首先，我通过深度优先搜索（DFS）为每辆车（由题目要求，我们一共有 $M \leq 20$ 辆车）查找所有可能的路径。例如，第一辆车从起点到终点有 3 条路可以选择，第二辆车有 4 条路可以选择等等。此时由于 DFS 的限制，我们并未考虑“一条车可以多次走过同一条边的情况”。完成 DFS 后，我们对每一辆车可走的所有路径，按照路径的成

本，对每一辆车可走的所有路径进行排序。代码中的 `all_paths[i]` 表示车 `i` 可走的所有路径。

然后，利用优先队列（最小堆）和哈希表，按照总花费（所有车当前的路径长度之和）从小到大的顺序，枚举所有可能的路径组合，并通过模拟，逐个检查每个组合是否满足流量限制。由于优先队列的性质，一旦当前检查的组合满足流量限制，那么它**必定是最优解**。其中流量限制的模拟法检查是通过模拟车辆沿路径行驶的过程，逐步更新道路上的流量并确保不超过交通流量限制来验证的。整个算法利用了深度优先搜索来生成路径、优先队列来优化路径组合的枚举、哈希函数来保证各个路径组合在入队时的不重复，同时也使用模拟法，保证了流量限制的合法性，从而有效解决了这个多车辆路径规划问题。

解题所用的主要数据结构和算法有**优先队列、最小堆、哈希函数、DFS**等。

（2）用 DFS 寻找每辆车的所有可走的道路（在一辆车不能重复走过一条边的情况下）：这一功能在函数 `find_all_paths` 中实现，通过深度优先搜索（DFS）从起点到终点找出所有可能的路径，并将这些路径存储在 `all_paths` 数组中。由于 DFS 的限制，我们无法访问已经走过的边，也就是默认每条边对于每辆车只能走一次。代码如下：

```
// 深度优先搜索构建所有路径
void find_all_paths(int src, int dest, bool visited[], int current_path[], int path_length, int vehicle_index) {
    visited[src] = true;
    current_path[path_length++] = src;
```

```
if (src == dest) {
    Path new_path;
    new_path.length = path_length;
    new_path.cost = 0;
    for (int i = 0; i < path_length; i++) {
        new_path.nodes[i] = current_path[i];
    }
    for (int i = 0; i < path_length - 1; i++) {
        new_path.cost += road_length[current_path[i]][current_path[i + 1]];
    }
    all_paths[vehicle_index][path_count[vehicle_index]++] = new_path;
} else {
    for (int v = 0; v < N; v++) {
        if (!visited[v] && road_length[src][v] > 0) {
            find_all_paths(v, dest, visited, current_path, path_length, vehicle_index);
        }
    }
}
visited[src] = false;
}
```

(3) 通过优先队列（最小堆）和哈希函数，每次枚举当前还未判断过的组合中最优的组合，并将更多的组合入队：这一功能在 `find_best_combination_priority_queue` 中实现，通过优先队列（最小堆），枚举所有可能的路径组合，并寻找满足流量限制的最优路径组合，使得所有车辆的总路径成本最小。

该函数首先初始化一个优先队列，其中包含初始的路径组合（每辆车选择的第一条路径，即每辆车的最短路径，对应基础要求）。然后，在每次从队列中取出一个路径组合时，函数会检查该组合是否满足流量限制（这利用了（4）中的函数模块），如果合法，则返回该组合的总成本，并将其作为最优解。如果不合法，函数会继续扩展当前组合，尝试替换路径并更新其成本，直到找到合法的组合或遍历所

有可能的组合。扩展入队方式为逐步将每辆车选择的道路改为下一条。例如，如果 $M=3$ ，且原组合是 $\{0, 0, 0\}$ ，即每辆车都选择自己的第一条路，且这种选择不合法，则下一次扩展时会将 $\{1, 0, 0\}$ ， $\{0, 1, 0\}$ ， $\{0, 0, 1\}$ 三个组合中尚未入队的组合入队，入队后维护最小堆，使得下一次挑选（pop）出来的组合是总花费最小的。判断是否已入过队的方式是哈希函数。

通过这种方式，函数高效地枚举路径组合，确保最终选出的路径组合不仅满足流量限制，而且具有最小的总成本。代码如下：

```
int find_best_combination_priority_queue(int M, Path all_paths[MAX_N][MAX_PATHS], int
path_count[MAX_N], Path result_paths[MAX_N]) {
    priority_queue<Combination, vector<Combination>, CompareCombination> pq; // 优先队列，用于存
    储所有合法路径组合，按照总成本从小到大排序
    unordered_map<Combination, bool, CombinationHash, CombinationEqual> map; // 用于记录已经访问
    过的组合
    Combination initial_comb;
    memset(initial_comb.indices, 0, sizeof(initial_comb.indices));
    initial_comb.total_cost = 0;
    for (int i = 0; i < M; i++) {
        initial_comb.total_cost += all_paths[i][0].cost;
    }
    pq.push(initial_comb);
    map[initial_comb] = true;

    while (!pq.empty()) {
        Combination current = pq.top();
        pq.pop();
        Path selected_paths[MAX_N];
        for (int i = 0; i < M; i++) {
            selected_paths[i] = all_paths[i][current.indices[i]];
        }
        //cout << "Now check a new combination,the indices are: ";
        for (int i = 0; i < M; i++) {
            selected_paths[i] = all_paths[i][current.indices[i]];
            //cout << current.indices[i] << " ";
        }
        //cout << '\n';
    }
}
```

```
    if (check_combination(selected_paths, M)) {
        for (int i = 0; i < M; i++) {
            result_paths[i] = selected_paths[i];
        }
        return current.total_cost;
    }
    else
        //cout << "This combination is illegal!Come to the next one" << '\n';

    for (int i = 0; i < M; i++) {
        if (current.indices[i] + 1 < path_count[i]) {
            Combination next = current;
            next.indices[i]++;
            next.total_cost += all_paths[i][next.indices[i]].cost -
all_paths[i][current.indices[i]].cost;
            if(!map[next]){
                pq.push(next);
                map[next] = true;
            }
        }
    }
}
return -1;
}
```

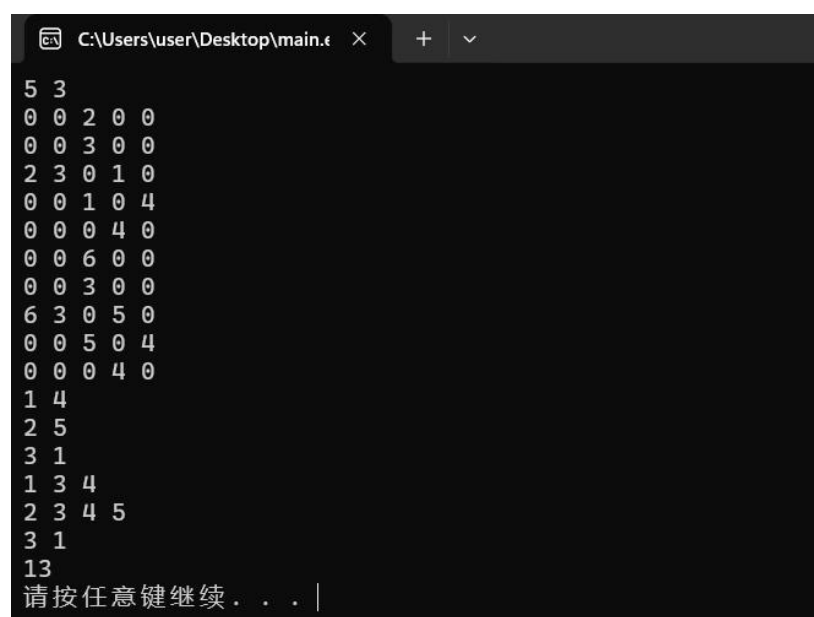
(4) 用模拟法判断当前道路选择是否满足流量限制：这一功能在 `check_combination` 中实现，模拟多个车辆沿其路径行驶的过程，并检查给定路径组合是否满足交通流量限制。函数通过维护一个 `time_usage` 矩阵来记录每条道路的流量，并模拟每辆车沿路径行驶时，在道路上占用的时间。对于每辆车的路径，函数逐步检查每个路段的流量，确保道路上的流量在任何时刻不会超过其限制。如果在行驶过程中某条道路的流量超出了其限制，函数立即返回 `false`，表示该路径组合不合法；如果所有路径都合法，函数最终返回 `true`，表示该组合满足流量限制。因此，`check_combination` 函数的核心功能


```
        last_progress[i][1] = progress[i][1];
        for (int j = 0; j < paths[i].length - 1; j++) {
            if (paths[i].nodes[j] == current_node) {
                next_index = j + 1;
                break;
            }
        }
        if (next_index != -1) {
            progress[i][0] = current_node;
            progress[i][1] = paths[i].nodes[next_index];
            progress[i][2] = road_length[progress[i][0]][progress[i][1]]; }
    } } }
    if (all_done) break;
}
return true;
}
```

三、部分样例检测

我构建了一些样例，包括“流量不发生冲突”和“发生冲突”两类的，同时也包括了 N、M 较小时的情况和最大时的情况，均能在 20 秒内输出正确答案。下面给出我尝试的样例中具有代表性的三个：

样例一：



```
C:\Users\user\Desktop\main.€ x + v
5 3
0 0 2 0 0
0 0 3 0 0
2 3 0 1 0
0 0 1 0 4
0 0 0 4 0
0 0 6 0 0
0 0 3 0 0
6 3 0 5 0
0 0 5 0 4
0 0 0 4 0
1 4
2 5
3 1
1 3 4
2 3 4 5
3 1
13
请按任意键继续. . . |
```


样例解释：这是示例样例，N, M 较小，且不发生流量冲突

样例二：

```
5 3
0 99 99 1 0
99 0 0 99 0
99 0 0 0 9
1 99 0 0 99
0 0 9 99 0
0 99 1 1 0
99 0 0 99 0
1 0 0 0 1
1 99 0 0 99
0 0 1 99 0
1 4
1 4
5 3
1 4
1 2 4
5 3
208
请按任意键继续. . . |
```

样例解释：这是一个会有流量冲突且较复杂的样例，第二辆车因为流量冲突既不能选择最短的 1-4（与第一辆车冲突），也不能选择第二短的 1-3-5-4（因为与第三辆车冲突），只能选择最长的 1-2-4。

样例三：

```
20 13
0 0 0 0 0 0 0 0 0 0 0 0 0 38 0 40 0 46 0 0 0 13
0 0 0 0 0 0 0 0 0 96 0 0 0 0 83 23 81 0 26 0
0 0 57 0 0 0 0 0 0 0 0 0 28 0 0 0 0 0 0
0 0 0 100 0 47 0 84 0 15 65 54 0 0 0 0 43 0 0
0 0 0 100 0 0 0 0 0 0 0 57 0 0 0 0 0 0 85
0 0 0 0 0 0 10 88 0 0 0 69 58 0 0 0 5 18 0
0 0 0 47 0 0 0 0 0 26 0 0 0 0 0 0 0 64
0 0 0 0 10 0 0 0 93 0 0 0 0 0 3 0 0 0
0 96 0 84 0 88 0 0 0 0 0 44 0 0 0 4 0
0 0 0 0 0 0 93 0 10 0 0 0 0 0 0 0 0
0 0 0 15 0 0 26 0 0 0 0 0 0 0 0 56 57
38 0 0 65 57 0 0 0 0 0 0 0 80 0 0 0 0
0 0 28 54 0 69 0 0 0 0 0 89 0 0 0 27 20 0
40 0 0 0 0 58 0 0 44 0 0 89 55 0 0 46 0 54
0 83 0 0 0 0 0 0 0 0 0 0 0 96 0 62 0
46 23 0 0 0 0 0 0 0 80 0 0 0 0 0 0 0
0 81 0 0 0 0 3 0 0 0 0 96 0 0 15 0
0 0 0 43 0 5 0 0 0 0 27 46 0 0 25 0 0
0 26 0 0 18 0 0 4 0 56 0 20 0 62 0 15 0 24 46
13 0 0 85 0 64 0 0 0 57 0 0 54 0 0 0 46 0
```

```
0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 1
0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 0 1 0
0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 1 0 1 0 1 1 1 0 0 0 1 0 0
0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1
0 0 0 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0
0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0
0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0
0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 1
1 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 1 1 0 1 0 0 0 0 0 0 1 0 0 0 1 1 0
1 0 0 0 0 1 0 0 1 0 0 0 1 1 0 0 0 1 0 1
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0
0 0 0 1 0 1 0 0 0 0 0 0 1 1 0 0 0 1 0 0
0 1 0 0 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1
1 0 0 0 1 0 1 0 0 0 1 0 0 1 0 0 0 0 1 0
```

```
3 7
4 8
4 10
4 12
4 17
5 13
5 15
6 8
6 9
6 16
7 11
7 20
9 16
3 13 4 11 7
4 18 6 8
4 9 19 17 8 10
4 12
4 11 19 17
5 4 13
5 20 19 15
6 8
6 19 9
6 18 13 19 2 16
7 11
7 20
9 19 2 16
1154
```

样例解释：这是一个十分庞大的有流量冲突的样例，并且可以证明若全选择最短路径会发生流量冲突。在程序运行约 6-7 秒后给出了答案。