中国石油大学（北京）克拉玛依校区

2023—2024 学年第三学期

《数据结构与程序综合实践》

实 践 报 告

班级：计算机类 23-3 班

学号：2023015509

姓名：胡林森

指导教师：李国和 董丹丹

完成时间：2024 年 7 月 4 日

一、题目：启发式搜索（九宫格应用）

摘要：本文介绍了笔者在《数据结构与程序综合实践》课程中完成的图搜索项目，图搜索算法的实现和应用。介绍了三种种图搜索算法（DFS、BFS、A*算法），并以八数码九宫格问题为例，展示了 A*算法在求解最优路径问题上的优势，并且比较启发式搜索算法与其他算法。

关键词：数据结构，图搜索，八数码问题，最优路径

目 录

二、目的与目标

在计算机科学中，图是表示对象及其相互关系的一种数学结构。图搜索算法是一系列用于遍历或搜索图中节点的算法，它们对于路径查找、最短路径问题等问题至关重要。

1目的

掌握数据结构和算法的应用： 通过实践项目，深入学习并掌握图等数据结构以及深度优先搜索（DFS）、广度优先搜索（BFS）、迪杰斯特拉算法、A*算法等常用算法的应用。

提升编程能力： 通过编写代码实现文件加密解密、树搜索、图搜索等功能，提升编程能力和问题解决能力。

培养综合实践能力： 将所学理论知识与实际问题相结合，通过实践项目锻炼综合实践能力，为未来学习和工作打下基础。

2 目标

理解算法原理： 深入理解 DFS、BFS、迪杰斯特拉算法、A*算法等图搜索算法的原理和适用场景。

分析算法性能： 能够分析不同算法的性能，并根据实际问题选择合适的算法。

拓展应用场景： 将图搜索算法应用于九宫格问题、迷宫问题、TSP 问题等实际问题，并探讨算法的拓展应用。

三、实践内容

1. 算法学习与理解:
   a) 深入学习并理解深度优先搜索(DFS)、广度优先搜索(BFS)、迪杰斯特拉算法、A*算法等图搜索算法的原理、实现方法和适用场景。
   b) 分析不同算法的优缺点和性能特点，并能够根据实际问题选择合适的算法。
2. 数据结构设计:
   a) 设计并实现图数据结构，包括图的存储方式（邻接矩阵、邻接表等）和节点表示方法。
   b) 设计九宫格问题的数据结构，包括状态表示、移动操作等。
3. 算法实现:
   a) 使用编程语言，实现 DFS、BFS、A*算法等图搜索算法。
   b) 实现九宫格问题的求解程序,利用所学的图搜索算法求解从初始状态到目标状态的最优路径。
4. 实验与分析:
   a) 在九宫格问题上进行实验，比较不同图搜索算法的效率和解的质量。
   b) 分析算法的性能特点，并探讨算法对性能的影响。
5. 应用拓展:
   a) 将图搜索算法应用于迷宫问题、TSP 问题等实际问题，并进行算法改进和优化。
   b) 探讨图搜索算法在其他领域的应用，例如路径规划、网络优化等。
6. 总结与展望:
   a) 总结实践过程中所学到的知识和经验，并撰写实践报告。
   b) 探讨图搜索算法的进一步研究方向和应用前景。

四、总体设计

1 问题描述

在一个 3x3 的网格中，通过上下左右移动八个数字和一格空位，从初始状态转变为目标

状态。在移动过程中，每次只能将一个数字滑动到空格位置，直到整个九宫格的数字排列与目标排列一致。

2 数据结构

```
// 常见图的结构定义
typedef struct Graph {
    int n;  // 顶点数量
    char** nodes;  // 存储顶点字符串数组
    int** adjMatrix;  // 邻接矩阵
} Graph;
```

3 算法设计

启发式函数（也称为代价预估函数）在搜索算法中用于估计从当前状态到目标状态的代价。它在启发式搜索算法（如 A*算法）中起着关键作用，帮助算法更有效地找到最优解。

常见的代价预估函数包括：

1. 曼哈顿距离：在二维网格中，计算从当前位置到目标位置的行差和列差的绝对值之和。
2. 欧几里得距离：在二维网格中，计算从当前位置到目标位置的直线距离。
3. 切比雪夫距离：在二维网格中，计算从当前位置到目标位置的行差和列差的最大值。

五、详细设计

1 数据结构

A*算法中用于优先队列的数据结构

```
// 节点结构体
typedef struct Node {
    int state[N][N];  // 当前状态
    int g;  // 实际代价
    int h;  // 预估代价（曼哈顿距离）
    struct Node* parent;  // 父节点指针
    } Node;
```

Node 是用来在 A*算法中表示搜索树中的节点的。具体来说，它包含以下几个部分：

1. int state[N][N]：表示当前的状态，通常是一个 N×N 的二维数组，用来存储九宫格拼图或其他类似问题的当前布局。
2. int g：表示从起始状态到当前状态的实际代价，即已经走过的步数。
3. int h：表示从当前状态到目标状态的预估代价，通常使用曼哈顿距离来计算。
4. struct Node* parent：指向父节点的指针，用于回溯路径。

这个数据结构在 A* 算法中非常重要，它包含了搜索过程中每个节点的关键信息，帮助算法决定下一步应该扩展哪个节点。

2 核心算法

(1)深度优先搜索（DFS）

深度优先搜索是一种用于遍历或搜索树或图结构的算法。这个算法会尽可能深地搜索图的分支。当节点 v 的所在边都已被探寻过，搜索将回溯到发现节点 v 的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。

如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。DFS 主要用在需要遍历所有节点并且需要尽可能深地搜索图中节点的场景，例如解决迷宫问题、拓扑排序以及找出图中所有的连通分量。

(2)广度优先搜索（BFS）

与 DFS 不同，广度优先搜索是一种从根节点开始，沿着图的宽度遍历节点的算法。它对每一个节点先访问它的所有邻接节点，然后再逐层向外扩展。BFS 通常使用队列来实现，并且适用于需要在不需要考虑路径成本的情况下找到两个节点之间的最短路径的问题。

(3)A*搜索算法

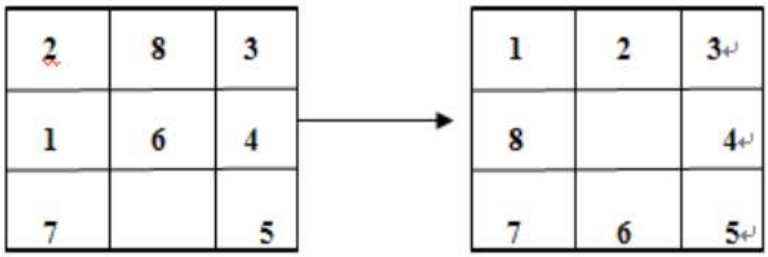
A*搜索算法是一种启发式搜索算法，通过维护一个代价函数评估每个节点的代价，该代价由两部分组成：一部分是起点到当前节点的实际代价，另一部分是从当前节点到目标点的估计代价。启发函数的设计对 A 算法的性能有很大影响。A*算法广泛应用于路径规划和游戏 AI 中的角色移动计算。

A*算法是一种在图中寻找从初始节点到目标节点最短路径的启发式搜索算法。它结合了 Dijkstra 算法的确保性（保证找到一条最短路径）和贪心算法的高效性（快速找到目标）。A*算法通过评估函数 f(n)=g(n)+h(n)来工作，其中 g(n)是从起始点到任何顶点 n 的实际成本，而 h(n)是从顶点 n 到目标的估计最低成本，通常用启发式函数来计算，这个函数需要事先设计来反映实际的地形或环境特征。理想情况下，h(n)应该不会高估实际的成本，这种情况下，A*算法保证找到一条最低成本路径。算法的性能和准确性高度依赖于启发式函数的选择。在实际应用中，A*算法广泛应用于各类路径规划问题，如机器人导航、地图定位服务和游戏中的 AI 路径寻找等场景。通过适当选择和调整启发式函数，A*算法能够在复杂的环境中有效地寻找最短路径，同时保持计算上的可行性和效率。

(4)小结

DFS 和 BFS 是最基本的图遍历方法，其中 DFS 适合目标明确但求解空间大的情况，而 BFS 适合找到最短路径或近距离的目标。A*算法是在 Dijkstra 的基础上增加了启发信息，使得搜索更加高效。在选择合适的图搜索算法时，需要根据具体问题的需要以及图的性质来决定。

3 应用方法



图表 1 左方为初始状态，右方为目标状态

九宫格的所以排列有 9!种，也就是 362880 种排法，同一幅图，通过移动方式也有$\frac{9!}{2}$种。

为了节约步骤，我们可以使用启发式搜索——A*算法，但会使用较多的空间，换取最优路径。

六、实验结果

1程序运行截图

(1)A*算法

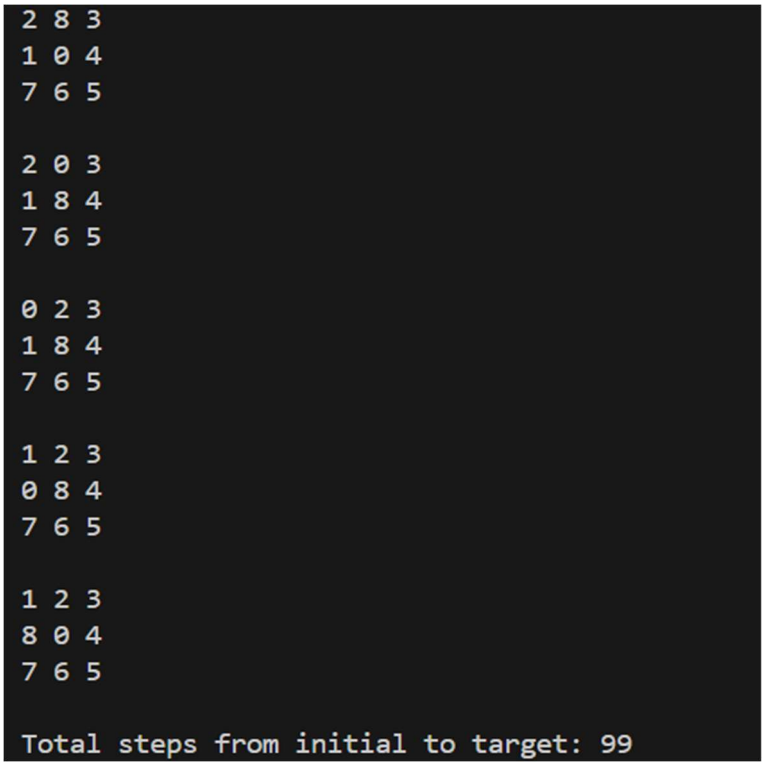
```
PS E:\CUPK-lib\XXQ1>  & 'c:\Users\15638\.vscode\extensions\ms-vscod
rosoft-MIEngine-In-n2skleib.wwr' '--stdout=Microsoft-MIEngine-Out-m
ine-Pid-4mpr41gq.1yj' '--dbgExe=F:\MinGW64\mingw64\bin\gdb.exe' '--
2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

Total steps from initial to target: 4
```

(2)BFS 算法

问题　输出　调试控制台　端口　终端　SQL CONSOLE

```
2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

Total steps from initial to target: 5
```

(3)DFS 算法



算法名称	所用时间	所用步骤
A*算法	0.03747s	4
BFS 算法	0.03938s	5
DFS 算法	0.1131s	99

七、结论

1 启发式搜索算法优势明显：

• A*算法在求解八数码问题中表现出显著的性能优势，用时最短，步骤最少。

• 这归功于 A*算法结合了 Dijkstra 算法的确保性和贪心算法的高效性，并通过启发式函数有效降低了搜索空间和时间复杂度。

2 启发式函数设计至关重要：

• 启发式函数的设计直接影响了 A*算法的性能。

• 高效的启发式函数能够准确评估当前状态与目标状态的差距，指导搜索过程，避免不必要的搜索。

• 设计高效的启发式函数需要深入理解问题特性，并进行大量实验和调整。

3 算法选择要从实际应用出发：

• DFS 算法在求解空间较大时效率较低，但在某些情况下可以找到最优解。

• BFS 算法能够找到最短路径，但时间复杂度较高。

• 选择合适的算法需要根据具体问题的特点进行权衡。

4 不足与改进方案：

•使用堆栈进行深度优先搜索容易出现内存溢出，虽然代码中没有显式地定义和使用堆栈数据结构，但 DFS 的递归实现依赖于系统堆栈来管理递归调用。

改进措施：可以考虑将递归实现转换为使用显式堆栈的迭代实现。

•未处理非法输入：代码没有处理非法输入的情况，例如初始状态或目标状态中包含非

法数字（不在 0 到 8 之间），可能会导致未定义行为。

改进措施：增加对非法输入的处理

5 未来努力方向：

- 进一步研究启发式函数的设计方法，以提高算法效率和求解质量。
- 探索其他启发式搜索算法，并将其应用于更复杂的图搜索问题。
- 将图搜索算法与其他算法相结合，解决更复杂的优化问题。

八、图搜索问题的应用拓展

图搜索算法是通过图中的边对图中所有顶点进行系统访问的过程。除了八数码九宫格以为，常见的图搜索应用还有迷宫问题、旅行商问题(TSP),公路铺设、油气管道建设规划一定程度上可以抽象为 TSP 问题。

1 迷宫问题

迷宫问题是指在一个二维的网格中，找到从起点到终点的一条路径。网格中的某些格子是可通行的，而某些则是不可通行的障碍物。迷宫的目标是在满足约束的情况下找到一条从起点到终点的路径。

(1)迷宫问题的解法——DFS

1.定义迷宫和相关参数：

a) 使用一个二维数组 maze 表示迷宫，其中 0 表示可以通行的路径，1 表示不可通行的障碍。

b) 定义一个与迷宫同等大小的二维数组 visited 用来标记已访问的节点。

c) 定义起点 (startX, startY) 和终点 (endX, endY) 的坐标。

d) 定义四个移动方向：上（{-1, 0}），下（{1, 0}），左（{0, -1}），右（{0, 1}）。

2.输入与初始化：

a) 用户输入迷宫数据，即每个格子是 0 或 1。

b) 用户输入起点和终点的坐标。

c) 初始化访问数组 visited，将所有节点标记为未访问状态。

3.DFS 算法：(3.1.1 中已做介绍，此处不做过多阐述)

a) 从起点出发，标记起点为已访问。

b) 尝试移动到四个方向的邻接节点：

c) 检查新位置是否越界、是否是路径且未被访问（通过 isSafe 函数）。

d) 如果可以移动，递归调用 DFS 函数处理新位置。

e) 如果到达终点，则输出路径并结束递归。

f) 如果无法到达终点，则回溯，继续尝试其他方向。

4.输出结果：

a) 如果 DFS 找到路径，则依次打印路径上的坐标。

b) 如果 DFS 未找到路径，则输出提示“没有路径可以到达目的地”。

(2)DFS 算法的改进——最优路径 DFS

普通 DFS 会沿着树的深度进行搜索，直到找到叶子节点或没有路径为止，然后再回溯到上一个节点，继续搜索其他未访问的路径。因此，DFS 的特点是沿着路径尽可能深入搜索。

而最优路径 DFS 是在普通 DFS 的基础上，改进其能找到最短路径或满足特定条件的路径。例如，在求解迷宫问题时，不仅仅是找到一条路径，更希望找到最短路径。这种情况下，可以记录每次找到的路径长度，若找到更短的路径则更新最优路径。

优化普通 DFS 以达到查找最优路径只需要调整 3 个关键步骤：

1.使用一个变量来记录最佳路径长度和路径。

2.每当找到终点时，比较当前路径的长度与记录的最优路径长度。

3.如果当前路径更短，则更新最优路径。

(3)迷宫问题的解法——A*算法

在迷宫问题中，A*算法可以如下实现：

1.使用优先级队列（通常是最小堆）保存节点，按照 f(n)=g(n)+h(n)的值进行排序。

2.从起点开始，将其加入优先级队列中，并初始化 g(n)值。其中 g(n)：从起点到节点 n 的实际路径长度；h(n)：节点 n 到目标节点的估计路径长度

3.从优先级队列中取出 f(n)最小的节点进行扩展，检查其是否为终点。

4.对每个邻居节点，计算其新的 g(n)和 f(n)值，并更新优先级队列。

5.重复上述过程直到找到终点或者优先级队列为空。

2 TSP 问题——以管道建设规划为例

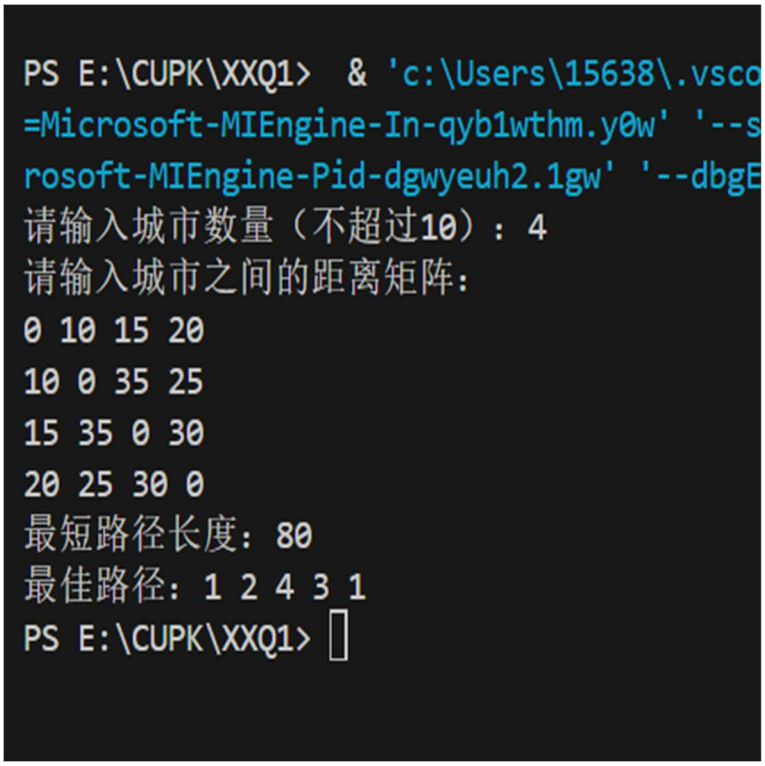
对于管道建设，一般会考虑站点间距离、建设代价(建设难度、费用)。在计算机科学中，旅行商问题（Traveling Salesman Problem，简称 TSP）是一个经典的组合优化问题。给定一个有 n 个城市的集合和每对城市之间的距离，目标是找到一条旅行路线，使得一个旅行商能够访问每个城市恰好一次并返回出发城市，使得所走的总距离最短。

3 运行结果

(1)迷宫问题

```
请输入迷宫(5 x 5)，0表示可通，1表示不可通：
0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
迷宫如下：
0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
请输入起点(x，y)和终点(x，y)的坐标（如：0 0 4 4）：
0 0 4 4
路径如下：
(4,4)
(3,4)
(2,4)
(1,4)
(0,4)
(0,3)
(0,2)
(1,2)
(2,2)
(2,1)
(2,0)
(1,0)
(0,0)
```

(2) TSP 问题



九、致谢

首先，我要衷心感谢我的指导老师李国和老师和董丹丹老师半个月来的教育与指导。在项目过程中，李国和老师不仅从交给我知识，更教给我治学的道理，严谨的态度。教给我们如何学习。

道阻且长，行则将至。

十、附录：代码

1 DFS

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N 3  // 九宫格大小
#define MAX_DEPTH 100  // 设置最大递归深度，防止无限递归

int count = 0;  // 状态计数

// 目标状态
int target[N][N] = {
    {1, 2, 3},
    {8, 0, 4},
    {7, 6, 5}
};

// 节点结构体
typedef struct Node {
    int state[N][N];  // 当前状态
    struct Node* parent;  // 父节点
    int depth;  // 当前深度
} Node;

// 节点比较以检查两个状态是否相同
int is_same_state(int state1[N][N], int state2[N][N]) {
    return memcmp(state1, state2, N * N * sizeof(int)) == 0;
}

// 生成一个新节点
Node* create_node(int state[N][N], Node* parent, int depth) {
    Node* new_node = (Node*) malloc(sizeof(Node));
    if (new_node != NULL) {
        memcpy(new_node->state, state, N * N * sizeof(int));
        new_node->parent = parent;
        new_node->depth = depth;
    }
    return new_node;
}

// 获取空格位置
void get_blank_pos(int state[N][N], int* blank_x, int* blank_y) {
```

```
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            if (state[i][j] == 0) {
                *blank_x = i;
                *blank_y = j;
                return;
            }
        }
    }
}

// 打印路径
void print_path(Node* node) {
    if (node == NULL) return;
    print_path(node->parent);
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            printf("%d ", node->state[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

// 释放路径
void free_path(Node* node) {
    if (node == NULL) return;
    free_path(node->parent);
    free(node);
}

// 深度优先搜索算法求解函数
int dfs_search(Node* current) {
    if (is_same_state(current->state, target)) {
        print_path(current);
        count = current->depth;
        return 1;
    }

    if (current->depth >= MAX_DEPTH) {
        return 0;  // 达到最大深度，停止递归
    }

    int blank_x, blank_y;
```

```
    get_blank_pos(current->state, &blank_x, &blank_y);
    int directions[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    for (int i = 0; i < 4; ++i) {
        int new_x = blank_x + directions[i][0];
        int new_y = blank_y + directions[i][1];
        if (new_x >= 0 && new_x < N && new_y >= 0 && new_y < N) {
            int new_state[N][N];
            memcpy(new_state, current->state, N * N * sizeof(int));
            new_state[blank_x][blank_y] = new_state[new_x][new_y];
            new_state[new_x][new_y] = 0;
            Node* neighbor = create_node(new_state, current,
current->depth + 1);
            if (neighbor != NULL) {
                if (dfs_search(neighbor)) {  // 递归搜索
                    return 1;
                }
                free(neighbor);
            }
        }
    }
    return 0;
}

int main() {
    int initial[N][N] = {
        {2, 8, 3},
        {1, 6, 4},
        {7, 0, 5}
    };

    Node* initial_node = create_node(initial, NULL, 0);
    if (initial_node == NULL) {
        printf("Failed to create initial node.\n");
        return -1;
    }

    if (!dfs_search(initial_node)) {
        printf("No solution found\n");
    }

    printf("Total steps from initial to target: %d\n", count);
    free_path(initial_node);
```

```
    return 0;
}
```

2 BFS

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N 3  // 九宫格大小
#define MAX_LIST_SIZE 10000  // 最大列表大小

int count = 0;  // 状态计数

// 目标状态
int target[N][N] = {
    {1, 2, 3},
    {8, 0, 4},
    {7, 6, 5}
};

// 节点结构体
typedef struct Node {
    int state[N][N];  // 当前状态
    struct Node* parent;  // 父节点
    int depth;  // 当前深度
} Node;

// 节点比较以检查两个状态是否相同
int is_same_state(int state1[N][N], int state2[N][N]) {
    return memcmp(state1, state2, N * N * sizeof(int)) == 0;
}

// 生成一个新节点
Node* create_node(int state[N][N], Node* parent, int depth) {
    Node* new_node = (Node*) malloc(sizeof(Node));
    if (new_node != NULL) {
        memcpy(new_node->state, state, N * N * sizeof(int));
        new_node->parent = parent;
        new_node->depth = depth;
    }
    return new_node;
}

// 获取空格位置
```

```
void get_blank_pos(int state[N][N], int* blank_x, int* blank_y) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            if (state[i][j] == 0) {
                *blank_x = i;
                *blank_y = j;
                return;
            }
        }
    }
}

// 生成相邻节点
void get_neighbors(Node* current, Node** neighbors, int* count) {
    int blank_x, blank_y;
    get_blank_pos(current->state, &blank_x, &blank_y);
    int directions[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    *count = 0;

    for (int i = 0; i < 4; ++i) {
        int new_x = blank_x + directions[i][0];
        int new_y = blank_y + directions[i][1];
        if (new_x >= 0 && new_x < N && new_y >= 0 && new_y < N) {
            int new_state[N][N];
            memcpy(new_state, current->state, N * N * sizeof(int));
            new_state[blank_x][blank_y] = new_state[new_x][new_y];
            new_state[new_x][new_y] = 0;
            Node* neighbor = create_node(new_state, current,
current->depth + 1);
            if (neighbor != NULL) {
                neighbors[*count] = neighbor;
                (*count)++;
            }
        }
    }
}

// 打印路径
void print_path(Node* node) {
    if (node == NULL) return;
    print_path(node->parent);
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            printf("%d ", node->state[i][j]);
```

```
        }
        printf("\n");
    }
    printf("\n");
}

// 释放路径
void free_path(Node* node) {
    if (node == NULL) return;
    free_path(node->parent);
    free(node);
}

// 广度优先搜索算法求解函数
void bfs_search(int initial[N][N]) {
    Node* open_list[MAX_LIST_SIZE];
    int open_count = 0;

    Node* closed_list[MAX_LIST_SIZE];
    int closed_count = 0;

    Node* initial_node = create_node(initial, NULL, 0);
    if (initial_node == NULL) {
        printf("Failed to create initial node.\n");
        return;
    }
    open_list[open_count++] = initial_node;

    while (open_count > 0) {
        Node* current = open_list[0];
        if (is_same_state(current->state, target)) {
            print_path(current);
            count = current->depth;
            free_path(current);
            return;
        }

        for (int i = 0; i < open_count - 1; ++i) {
            open_list[i] = open_list[i + 1];
        }
        open_count--;

        closed_list[closed_count++] = current;
```

```
        Node* neighbors[4];
        int neighbor_count;
        get_neighbors(current, neighbors, &neighbor_count);

        for (int i = 0; i < neighbor_count; ++i) {
            Node* neighbor = neighbors[i];
            int is_in_closed_list = 0;
            for (int j = 0; j < closed_count; ++j) {
                if (is_same_state(neighbor->state,
closed_list[j]->state)) {
                    is_in_closed_list = 1;
                    break;
                }
            }
            if (!is_in_closed_list) {
                open_list[open_count++] = neighbor;
            } else {
                free(neighbor);
            }
        }
    }

    printf("No solution found\n");
}

int main() {
    int initial[N][N] = {
        {2, 8, 3},
        {1, 6, 4},
        {7, 0, 5}
    };

    bfs_search(initial);  // 调用BFS 算法进行搜索
    printf("Total steps from initial to target: %d\n", count);
    return 0;
}
```

3 A*算法

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N 3  // 九宫格大小
#define MAX_LIST_SIZE 10000  // 加大列表以防越界
```

```
// 目标状态
int target[N][N] = {
    {1, 2, 3},
    {8, 0, 4},
    {7, 6, 5}
};

int count = 0;  // 状态计数

// 节点结构体
typedef struct Node {
    int state[N][N];  // 当前状态
    int g;  // 实际代价
    int h;  // 预估代价（曼哈顿距离）
    struct Node* parent;  // 父节点指针
} Node;

// 曼哈顿距离计算
int manhattan(int state[N][N]) {
    int distance = 0;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            if (state[i][j] != 0) {  // 忽略空格位置
                int target_x = (state[i][j] - 1) / N;
                int target_y = (state[i][j] - 1) % N;
                distance += abs(target_x - i) + abs(target_y - j);
            }
        }
    }
    return distance;
}

// 生成一个新节点
Node* create_node(int state[N][N], int g, int h, Node* parent) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    if (new_node != NULL) {  // 检查内存分配是否成功
        memcpy(new_node->state, state, N * N * sizeof(int));  // 复制状态
        new_node->g = g;  // 设置实际代价
        new_node->h = h;  // 设置预估代价
        new_node->parent = parent;  // 设置父节点
    }
    return new_node;
}
```

```
// 节点比较函数（用于优先队列）
int compare_nodes(const void* a, const void* b) {
    Node* nodeA = *(Node**)a;
    Node* nodeB = *(Node**)b;
    return (nodeA->g + nodeA->h) - (nodeB->g + nodeB->h);  // 按f = g + h 排序
}

// 获取空格位置
void get_blank_pos(int state[N][N], int* blank_x, int* blank_y) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            if (state[i][j] == 0) {
                *blank_x = i;
                *blank_y = j;  // 找到空格位置
                return;
            }
        }
    }
}

// 生成相邻节点
void get_neighbors(Node* current, Node** neighbors, int* count) {
    int blank_x, blank_y;
    get_blank_pos(current->state, &blank_x, &blank_y);  // 获取空格的位置
    int directions[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};  // 上下左右四个方向
    *count = 0;

    // 遍历四个方向，生成合法的相邻节点
    for (int i = 0; i < 4; ++i) {
        int new_x = blank_x + directions[i][0];
        int new_y = blank_y + directions[i][1];
        if (new_x >= 0 && new_x < N && new_y >= 0 && new_y < N) {  // 检查新坐标是否在边界内
            int new_state[N][N];
            memcpy(new_state, current->state, N * N * sizeof(int));  // 复制当前状态
            new_state[blank_x][blank_y] = new_state[new_x][new_y];  // 交换空格与目标位置
            new_state[new_x][new_y] = 0;
            Node* neighbor = create_node(new_state, current->g + 1, manhattan(new_state), current);  // 生成新节点
```

```
            if (neighbor != NULL) {  // 检查节点生成是否成功
                neighbors[*count] = neighbor;
                (*count)++;
            }
        }
    }
}

// 判断两个状态是否相同
int is_same_state(int state1[N][N], int state2[N][N]) {
    return memcmp(state1, state2, N * N * sizeof(int)) == 0;
}

// 打印路径
void print_path(Node* node) {
    if (node == NULL) return;
    print_path(node->parent);  // 递归打印父节点的路径
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            printf("%d ", node->state[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

// 释放路径
void free_path(Node* node) {
    if (node == NULL) return;
    free_path(node->parent);  // 递归释放父节点
    free(node);
}

// A* 算法求解函数
void astar_search(int initial[N][N]) {
    Node* open_list[MAX_LIST_SIZE];
    int open_count = 0;

    Node* closed_list[MAX_LIST_SIZE];
    int closed_count = 0;

    Node* initial_node = create_node(initial, 0, manhattan(initial),
NULL);  // 生成初始节点
    if (initial_node == NULL) {
```

```
        printf("Failed to create initial node.\n");
        return;
    }
    open_list[open_count++] = initial_node;  // 初始节点加入开启列表

    while (open_count > 0) {
        qsort(open_list, open_count, sizeof(Node*), compare_nodes);  //
根据f值对开启列表排序

        Node* current = open_list[0];  // 取出f值最小的节点
        if (is_same_state(current->state, target)) {  // 检查是否达到目标
状态
            print_path(current);  // 打印路径
            count = current->g;  // 记录从初始状态到目标状态所经过的步数
            free_path(current);  // 释放路径
            return;
        }

        // 从开启列表中移除当前节点
        for (int i = 0; i < open_count - 1; ++i) {
            open_list[i] = open_list[i + 1];
        }
        open_count--;

        closed_list[closed_count++] = current;  // 将当前节点加入关闭列表

        Node* neighbors[4];
        int neighbor_count;
        get_neighbors(current, neighbors, &neighbor_count);  // 生成当前
节点的所有邻居节点

        // 遍历所有邻居节点
        for (int i = 0; i < neighbor_count; ++i) {
            Node* neighbor = neighbors[i];
            int is_in_closed_list = 0;
            for (int j = 0; j < closed_count; ++j) {
                if (is_same_state(neighbor->state,
closed_list[j]->state)) {
                    is_in_closed_list = 1;  // 如果邻居节点已经在关闭列表中
则跳过
                    break;
                }
            }
            if (!is_in_closed_list) {
```

```
                open_list[open_count++] = neighbor;  // 将合法的邻居节点加入开启列表
            } else {
                free(neighbor);  // 若邻居节点在关闭列表中则释放节点
            }
        }
    }

    printf("No solution found\n");  // 没有找到解决方案
}

int main() {
    int initial[N][N] = {
        {2, 8, 3},
        {1, 6, 4},
        {7, 0, 5}
    };

    astar_search(initial);  // 调用A*算法进行搜索
    printf("Total steps from initial to target: %d\n", count -1);  // 打印步数
    return 0;
}
```

4 迷宫问题

```
#include <stdio.h>

// 定义迷宫大小
#define SIZE 5

int maze[SIZE][SIZE]; // 迷宫数组
int visited[SIZE][SIZE]; // 标记已访问节点
int startX, startY, endX, endY; // 起点和终点

// 定义四个可能的移动方向
int direction[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

// 打印迷宫
void printMaze() {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            printf("%d ", maze[i][j]);
        }
        printf("\n");
```

```
    }
}

// 检查当前位置是否越界，是否可以访问
int isSafe(int x, int y) {
    return (x >= 0 && x < SIZE && y >= 0 && y < SIZE && maze[x][y] == 0
&& !visited[x][y]);
}

// 深度优先搜索
int dfs(int x, int y) {
    if (x == endX && y == endY) {
        printf("(%d,%d)\n", x, y);
        return 1;
    }

    // 标记当前节点已访问
    visited[x][y] = 1;

    // 尝试四个方向的移动
    for (int i = 0; i < 4; i++) {
        int newX = x + direction[i][0];
        int newY = y + direction[i][1];
        if (isSafe(newX, newY)) {
            if (dfs(newX, newY)) {
                printf("(%d,%d)\n", x, y);
                return 1;
            }
        }
    }

    // 回溯
    visited[x][y] = 0;
    return 0;
}

int main() {
    // 输入迷宫
    printf("请输入迷宫(%d x %d)，0 表示可通，1 表示不可通：\n", SIZE, SIZE);
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            scanf("%d", &maze[i][j]);
        }
    }
```

```
    // 打印迷宫
    printf("迷宫如下: \n");
    printMaze();

    // 输入起点和终点
    printf("请输入起点(x, y)和终点(x, y)的坐标（如: 0 0 4 4）: \n");
    scanf("%d %d %d %d", &startX, &startY, &endX, &endY);

    // 初始化访问数组
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            visited[i][j] = 0;
        }
    }

    printf("路径如下: \n");
    if (!dfs(startX, startY)) {
        printf("没有路径可以到达目的地\n");
    }

    return 0;
}
```

5 TSP 问题

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

#define MAX_CITIES 10  // 假设最多有 10 个城市

// 记录最短路径长度
int minCost = INT_MAX;

// 路径数组，用于记录当前路径
int path[MAX_CITIES];
// 最佳路径数组，用于记录最短路径
int bestPath[MAX_CITIES];

// 计算旅行费用
void tsp(int graph[MAX_CITIES][MAX_CITIES], bool visited[MAX_CITIES],
int n, int pos, int cost, int count) {
    // 如果已访问所有城市并回到起点，更新最小费用
    if (count == n && graph[pos][0] > 0) {
```

```
        if (cost + graph[pos][0] < minCost) {
            minCost = cost + graph[pos][0];
            for (int i = 0; i < n; i++) {
                bestPath[i] = path[i];
            }
            bestPath[n] = 0;  // 回到起点
        }
        return;
    }

    // 考虑下一个城市
    for (int i = 0; i < n; i++) {
        if (!visited[i] && graph[pos][i] > 0) {
            visited[i] = true;
            path[count] = i;

            // 递归调用tsp函数
            tsp(graph, visited, n, i, cost + graph[pos][i], count + 1);

            // 回溯
            visited[i] = false;
        }
    }
}

int main() {
    int n;  // 城市数量
    printf("请输入城市数量（不超过%d）: ", MAX_CITIES);
    scanf("%d", &n);

    int graph[MAX_CITIES][MAX_CITIES];
    printf("请输入城市之间的距离矩阵: \n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    // 记录已访问城市
    bool visited[MAX_CITIES] = { false };
    visited[0] = true;  // 从第一个城市开始
    path[0] = 0;

    // 调用tsp函数
```

```
    tsp(graph, visited, n, 0, 0, 1);

    // 输出最短路径及最短路径长度
    printf("最短路径长度：%d\n", minCost);
    printf("最佳路径：");
    for (int i = 0; i <= n; i++) {
        printf("%d ", bestPath[i] + 1);  // 1-indexed
    }
    printf("\n");

    return 0;
}
```