中国科学院大学

数据结构实习报告

姓名 朱希研 学号 2023Kxxxxxxxxxx

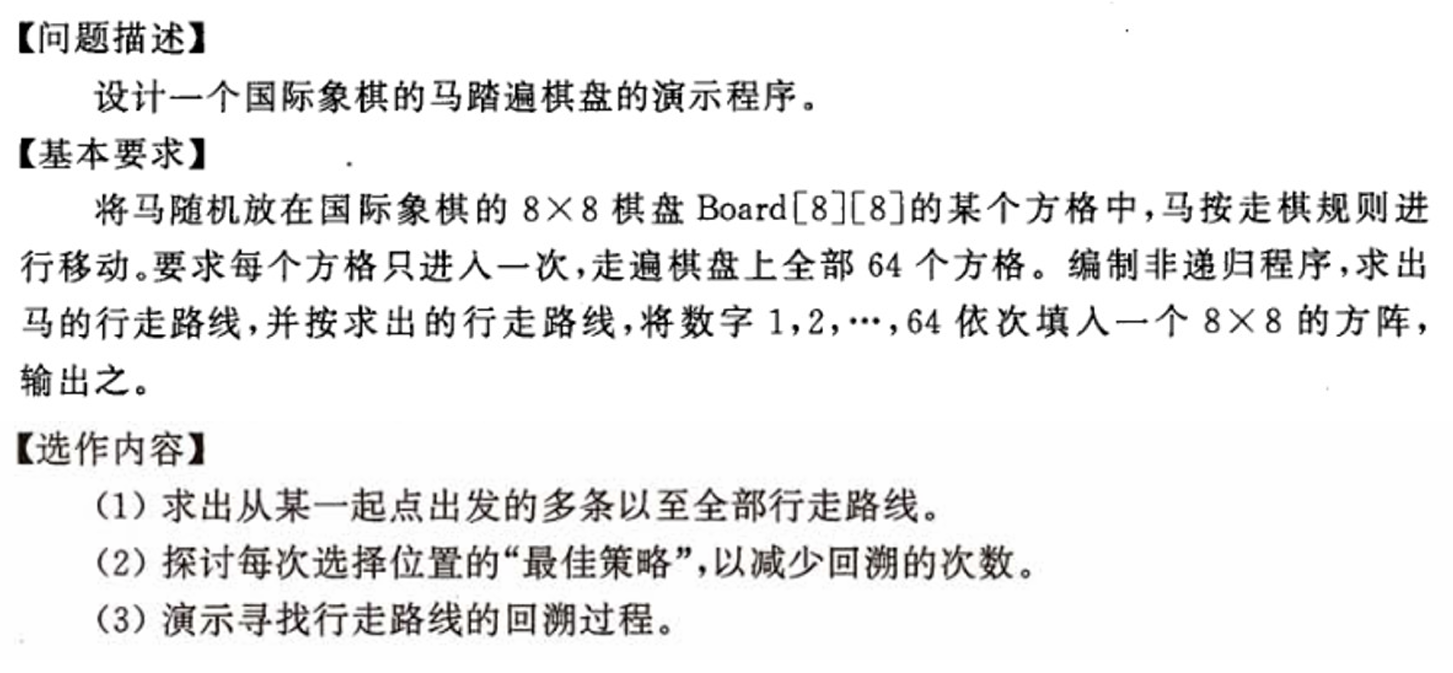
姓名 朱首赫 学号 2023K8009906029

姓名 罗子墨 学号 2023Kxxxxxxxxxx

实习项目 马踏棋盘问题和平衡二叉树操作演示

**一、马踏棋盘**

**1. 问题描述及项目情况概述**



本项目已完成基本内容+所有选作内容。

本项目实现了两个版本的马踏棋盘问题求解器：

（1）一个轻量化的版本（naive版）：可由用户任选起点，包含寻找单路径并输出运行时间、寻找所有路径两种模式。其中寻找所有路径模式中，可选择由用户手动确认寻找下一路径，或者程序自动寻找所有路径。此版本直接在终端通过字符组合输出棋盘，较简陋。

（2）一个可视化窗口版本（visualize版）：使用WINAPI实现窗口展示运行动画，可指定起点、演示动画速度，并可进行多次不同起点的演示。界面较为美观。

**2. 关键数据结构及全局变量**

typedef struct {

    int x; // 横坐标

    int y; // 纵坐标

    int tried\_times; // 已经尝试过的方向数

    int dir\_seq[8]; // 排序好的方向，被忽略的方向设为IGNORE

} point;

int board[8][8] = {0};           // 棋盘数组，存储每个位置的步数

int visited[8][8] = {0};         // 访问标记数组，记录位置是否已被访问

point path\_stack[MAXSTACKVOL] = {0}; // 路径栈，存储马走过的路径点

int top = -1;                    // 栈顶指针，-1表示栈空

int back\_num = 0;                // 回溯次数计数器

以上代码即为算法实现所用到的所有全局变量。

定义了一个8×8的棋盘格局数组board，初始化为0，当马走到某一格时记录从起点走到这一格的步数。visited数组用于标记棋盘上各点是都被访问过。back\_num用于记录寻路过程中的回溯次数。

path\_stack为一个栈，用来存储马走过的路径信息，是实现非递归回溯的关键。栈指针top指向栈顶元素，初始时栈为空，故top初值为-1。栈元素为point类型，包含该点的坐标信息，在此格局下下一步可走的位置dir\_seq（经启发式排序后的序列），以及在此格局中尝试过的次数tried\_times。

**3. 算法实现**

结合贪心算法和回溯法实现，大致思路为：对马当前所在的位置，求出周围八个方向分别有几个合法的下一步，每次选择下一步最少的且没尝试过的合法方向，若无路可走，则通过弹栈回溯到之前状态。直到走完整个棋盘（成功），或者栈空（失败），程序终止。具体实现及代码在下面给出。

**（1）栈操作**

#define FAIL 0

#define SUCCESS 1

#define MAXSTACKVOL 64

int push(int x, int y) {

    if (++top >= MAXSTACKVOL)

    {

        return FAIL;

    }

    point p;

    p.x = x;

    p.y = y;

    p.tried\_times = 0; // 初始化尝试次数为0

    path\_stack[top] = p; // 将点压入栈

    visited[x][y] = 1; // 标记为已访问

    board[x][y] = top + 1; // 更新棋盘上的步数

    return SUCCESS;

}

int pop() {

    if (top < 0) {

        return FAIL; // 栈为空

    }

    int x = path\_stack[top].x;

    int y = path\_stack[top].y;

    visited[x][y] = 0; // 恢复为未访问

    board[x][y] = 0; // 恢复棋盘上的步数

    top--; // 移动栈顶指针

    return SUCCESS;

}

栈的最大容量为MAXSTACKVAL=64，即路径最大长度为64。push和pop操作与一般的栈基本一致，略有不同的是加入了更新和回溯棋盘状态的操作。

**（2）启发式函数及排序操作**

**get\_next\_count()函数：**获取给定位置的下一步可走位置数。其中可走步数为0的方向的next\_count值会被设置为MY\_IGNORE = 9，表示后续不再考虑。但在最后一步时为了防止无限循环需要特殊处理。

int get\_next\_count(int x, int y) {

    // 下一步无可走位置的位置不考虑，设为MY\_IGNORE

    int next\_count = 0;

    for (int dir = 0; dir < 8; dir++)

    {

        int sx = x + DIR[dir][0];

        int sy = y + DIR[dir][1];

        if (is\_valid(sx, sy)){

            next\_count++;

        }

    }

    if (top == 62)

    {

        // 最后一个位置没有可走步了，将next\_count设为0而不是MY\_IGNORE，确保其会排在最前面且不被忽略（否则会不断回溯导致死循环）

        return next\_count;

    }else{

        return next\_count ? next\_count : MY\_IGNORE; // 如果没有可走位置，返回MY\_IGNORE

    }

}

**init\_and\_sort()函数：**初始化入栈结点的合法方向序列seq\_dir，并根据next\_count中的值对合法方向按从小到大顺序进行选择排序。同时保证了标记为被忽略的点会排在最后

void init\_and\_sort(int dir\_seq[8], int next\_count[9]) {

    // 初始化方向序列

    for (int i = 0; i < 8; i++)

    {

        if (next\_count[i] == MY\_IGNORE) {

            dir\_seq[i] = DIR\_IGNORE; // DIR\_IGNORE = 8

            // 被忽略的方向索引指向MY\_IGNORE = 9，这样就会被排在后面，

            // 不直接使用MY\_IGNORE的原因是为了防止下面以dir\_seq[i]为索引访问next\_count时越界

        }else {

            dir\_seq[i] = i; // 其他合法方向设为对应的索引

        }

    }

    // 选择排序

    for (int i = 0; i < 7; i++)

    {

        for (int j = i + 1; j < 8; j++)

        {

            // 将next\_count值小的方向放在前面

             if (next\_count[dir\_seq[i]] > next\_count[dir\_seq[j]]) {

                int temp = dir\_seq[i];

                dir\_seq[i] = dir\_seq[j];

                dir\_seq[j] = temp;

            }

        }

    }

    // 得到的dir\_seq数组按下一步可走位置从小到大排列，且被忽略的位置会排在后面

}

**（3）贪心+回溯算法实现**

代码在文字下方给出，大体思路与上面的描述一致，具体实现中需细节如下：

退出条件：若只寻找单条路径，则走完整个棋盘，即栈指针top=63时停止。若寻找所有路径，到达终点时立即回溯，只有用户取消或找完所有路径才停止。（使用宏定义选择只寻找单条路径还是多条路径）

生成方向序列及尝试：若栈顶结点的tried\_times为0，说明这个结点刚刚入栈，需要根据当前棋盘格局生成seq\_dir序列；若不为0，说明是回溯到了该结点，根据尝试次数选择seq\_dir中没尝试过的方向，直到试完所有未被忽略的方向，说明无路可走需要回溯。

通过存储已生成的seq\_dir序列，可以防止回溯时重复生成。这样用空间换取时间的性价比如何，取决于回溯发生的频率，以及棋盘的规模。后续通过实验发现，只寻找一条路径时，该贪心算法大多数时候能实现零回溯，对以往的候选方向序列的需求不大。不过，对于要寻找所有路径的情况，这样做的性价比还是比较高的。

void my\_dfs() {

    while (1)

    {

        // 退出条件：走到倒数第二个位置

        if (top >= 63) {

            #ifdef SPEED

            // 只寻找一条路径

            printf("成功到达终点！\n");

            break;

            #endif

            display\_board(); // 显示棋盘

            // 寻找所有路径

            printf("成功找到第%d条路径！\n", ++success\_count);

            #ifdef ONEWAY

            // 询问是否继续

            if (!if\_continue()) {

                return; // 终止寻找

            }

            #endif

            if (pop() == FAIL) { // 弹出终点，回溯，寻找其他路径

                printf("没有可走位置了，回溯失败！\n");

                return ; // 回溯失败，退出

            }

            back\_num++; // 回溯次数加1

            continue;

        }

        // 获取栈顶（当前点）位置

        int x = path\_stack[top].x;

        int y = path\_stack[top].y;

        // 如果尝试次数为0，需要重新生成并排序可走序列dir\_seq；否则，可以复用之前的。

        if (!path\_stack[top].tried\_times)

        {

            int next\_count[9]; // 记录合法位置的下一步可走数量

            next\_count[8] = MY\_IGNORE; // 最后一个位置设为MY\_IGNORE，表示没有可走位置

            for (int dir = 0; dir < 8; dir++)

            {

                // 检查当前点的8个方向是否合法，若合法，计算合法位置的下一步可走数量

                int sx = x + DIR[dir][0];

                int sy = y + DIR[dir][1];

                if(is\_valid(sx, sy)){

                    next\_count[dir] = get\_next\_count(sx, sy);

                }else{

                    //不合法的位置不考虑，设为MY\_IGNORE

                    next\_count[dir] = MY\_IGNORE;

                }

            }

            // 排序下一步可走位置不为0的方向

            init\_and\_sort(path\_stack[top].dir\_seq, next\_count);

        }

        int start = path\_stack[top].tried\_times++; // 尝试dir\_seq[start]的方向，并增加尝试次数

        if (start >= 8) { // 如果尝试次数≥8，说明该点的所有方向都尝试过了，需要回溯

            // 回溯

            if (pop() == FAIL) {

                printf("没有可走位置了，回溯失败！\n");

                return ; // 回溯失败，退出

            }

            back\_num++; // 回溯次数加1

            continue;

        }

        int try\_dir = path\_stack[top].dir\_seq[start];  // 本次尝试方向

        if (try\_dir == DIR\_IGNORE) { // 如果是被忽略的方向，说明没有可走位置了,需要回溯

            // 回溯

            if (pop() == FAIL) {

                printf("没有可走位置了，回溯失败！\n");

                return ; // 回溯失败，退出

            }

            back\_num++; // 回溯次数加1

            continue;

        }

        // 尝试下一个点

        int sx = x + DIR[try\_dir][0]; // 下一个点的横坐标

        int sy = y + DIR[try\_dir][1]; // 下一个点的纵坐标

        // 压入下一个点（push函数内部设置棋盘步数、标记为已访问、以及初始化该点的信息）

        if (push(sx, sy) == FAIL) {

            printf("栈满，无法继续走！\n");

            return; // 栈满，退出

        }

    }

}

**4. 时空复杂度分析**

本部分按照只寻找一条和寻找所有路径两种情况讨论时空复杂度。设棋盘规模为n×n。

（1）只寻找一条路径

需要注意的是，最多可走方向不随棋盘规模改变，始终为8。

时间复杂度：由于大多数时候可以零回溯，只需要走等同于棋盘格子数的步数，且每次需要排序的可走方向最多为8个，故总时间复杂度为O(n2)。

空间复杂度：棋盘格局O(n2)+栈空间O(n2)+中间变量O(1)，故总空间复杂度也为O(n2)。但根据上一部分的讨论，若棋盘规模较小，在只寻找一条路径时，可以去掉point结构体中的seq\_dir数组成员，能够节省相当一部分空间且不会增加多少运行时间。

（2）寻找所有路径

时间复杂度：设马在每个位置合法下一步数量的平均值为分支因子k（k≤8，查阅到的实验值约为6；由于启发式排序的存在会触发剪枝，可进一步降低至3左右），棋盘共有n2个格子，则时间复杂度为O()，不低于O()。

空间复杂度：记录棋盘格局和栈使用的空间仍然为O(n2)，采用每成功一次便输出一次路径的方式，空间复杂度仍为O(n2)。

**5. 可视化方法说明**

使用WINAPI实现窗口展示与交互。

（1）界面设计

窗口包含的要素有：棋盘区域：8×8网格，带行列标号；控制面板：输入起点坐标、按钮控制启动和重置；速度控制：通过滑块调整动画速度；信息显示：回溯次数统计；马位置、已走过的格子及对应步数

颜色方面，使用灰色边框，采用奶油色和海棠红交替显示初始棋盘；若格子被访问，则重绘为淡蓝色，并显示步数。符号方面，马位置使用“♘”符号标注，已访问的的格子则直接标注数字。字体方面，采用微软雅黑字体。

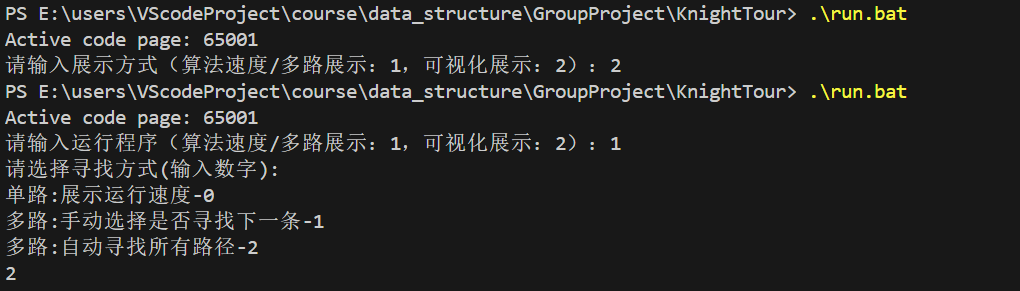
（2）动画控制机制

使用Windows定时器控制动画触发，每次定时器触发时贪心算法执行单步移动，每次移动后仅重绘棋盘区域。使用滑块输入调整动画延迟时间，以控制最终的呈现速度。

**6. 成果展示及亮点分析**

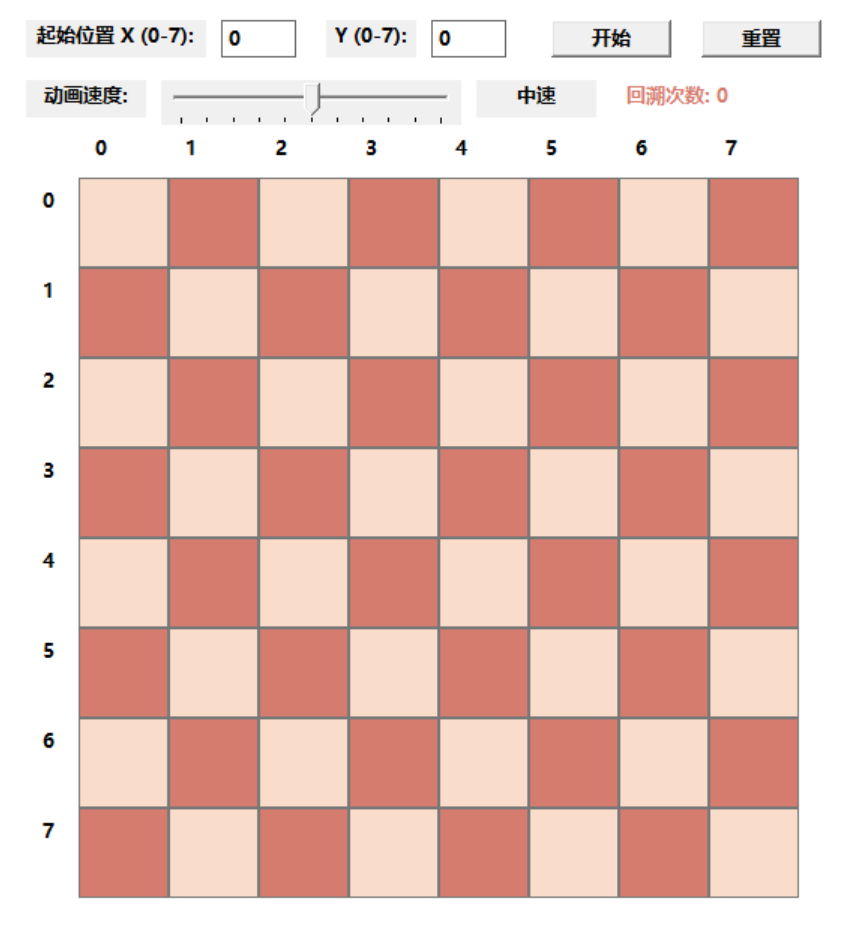
**成果展示：**

（1）选择界面

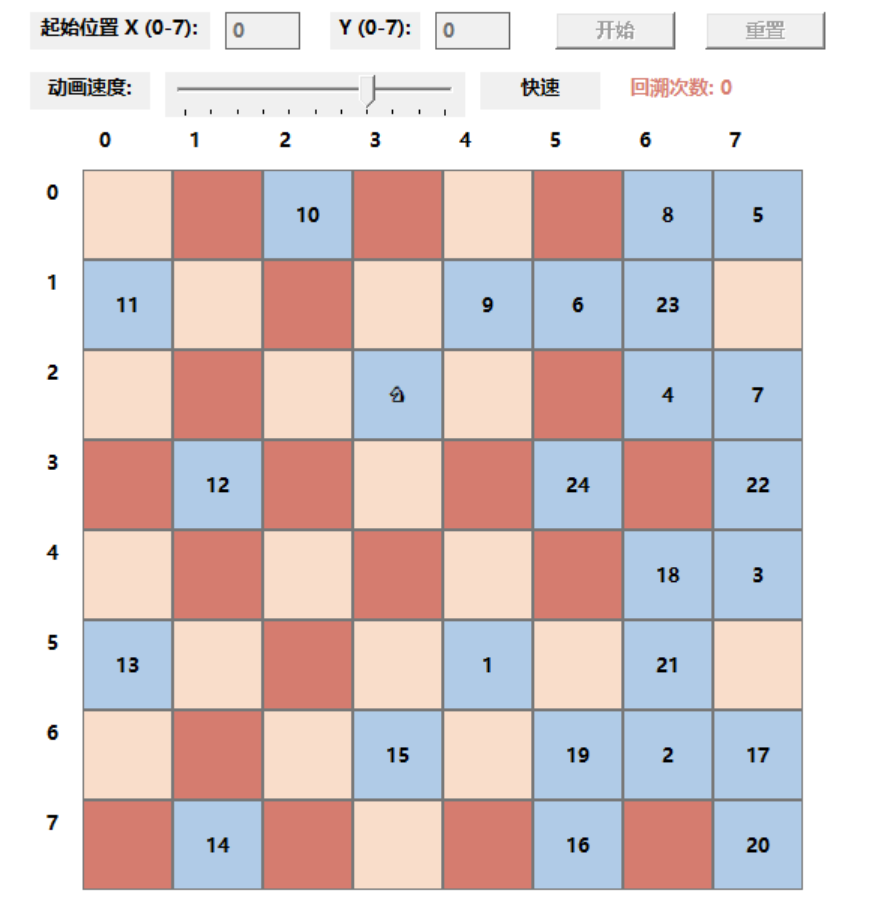


**图 1 程序及模式选择**

（2）图形化展示界面

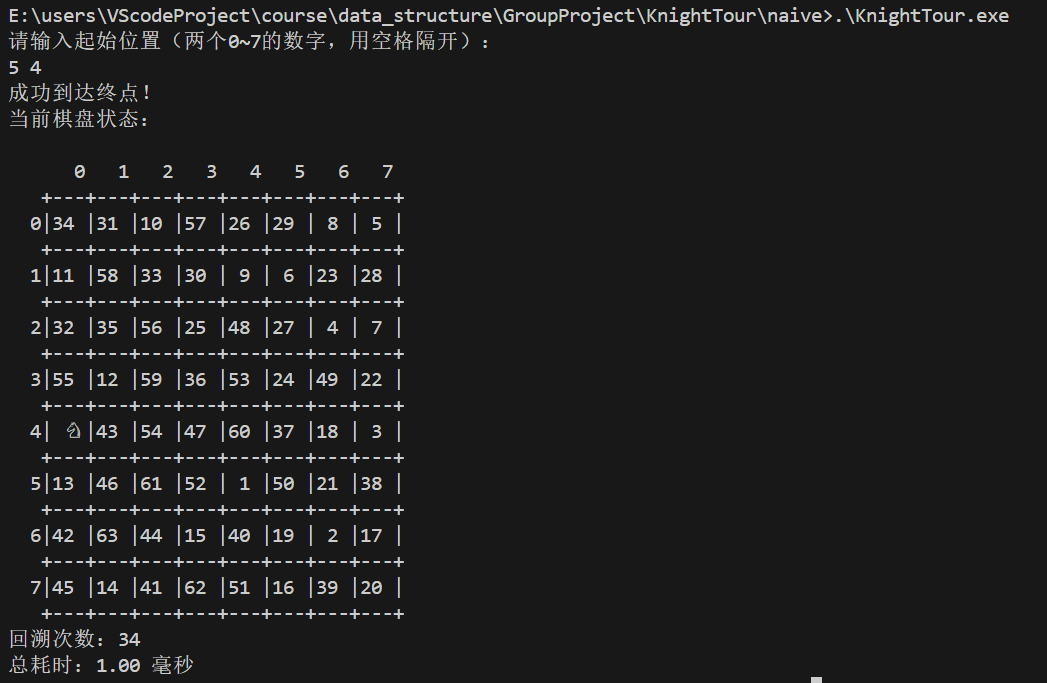


**图 2 初始界面**



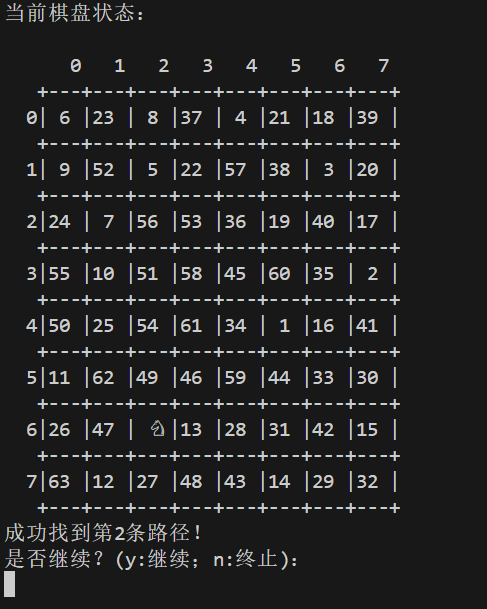
**图 3 运行中动画**

（3）单路寻找速度

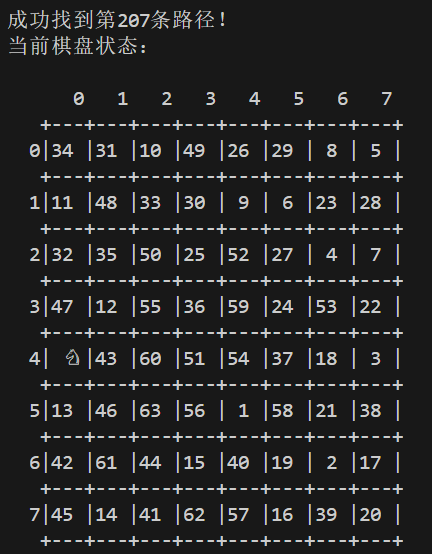
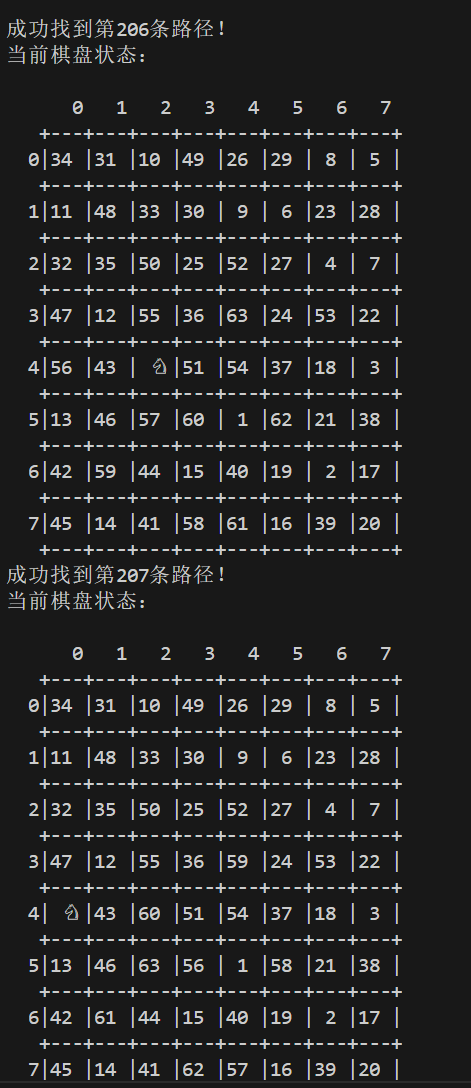


**图 4 单路寻找速度展示**

（4）所有路径寻找



**图 5 手动多路寻找**



**图 6 自动多路寻找**

**亮点分析：**

1. 贪心算法优化：复用历史生成的候选方向序列，避免重复生成和排序。

2. 健壮性：对预期范围之外的输入能够正确检测并处理，不会出现死循环或异常退出的情况。

3. 可视化功能强大：呈现方式直观美观，可进行多次演示及个性化调整。

4. 功能丰富：可以根据模式选择的不同，充分发挥算法优越的速度或强大的可视化功能。可以支持用户修改展示速度及是否手动跟进下一步。