数据结构与算法 I 实验报告

实验 1：栈和队列的应用

龚舒凯 2022202790 应用经济-数据科学实验班

https://github.com/GONGSHUKAI

2023 年 9 月 30 日

骑士巡逻

1 需求分析

问题描述：骑士巡逻（Knight's tour）是指在按照国际象棋中骑士的规定走法走遍整个棋盘的每一个方格，而且每个网格只能够经过一次。假若骑士能够走回到最初位置，则称此巡逻为“封闭巡逻”，否则，称为“开巡逻”。国际象棋中骑士的走法与中国象棋中马的走法相似，呈“日”字型或“L”字型。如下图所示。即每走一步，其行列坐标，一个变化 1，另一个变化 2。

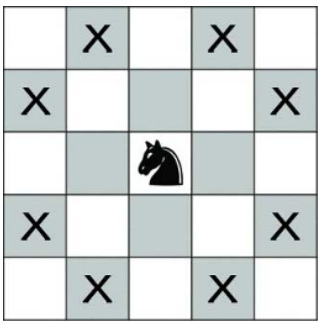


图 1: 骑士可走的 8 个方向

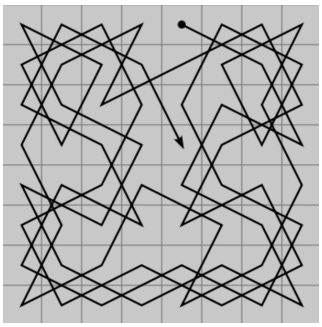


图 2: 一种“开巡逻”

基本要求：将骑士放在给定大小（$n \times n$）的国际象棋棋盘的给定位置上（某一方格中），骑士按照走棋规则进行移动，要求每个方格只进入一次，走遍棋盘上的所有方格。分别编写**一个递归和一个非递归程序**，求出骑士的行走路线，将数字 1，2，3，...，$n \times n$ 依次填入这个棋盘上的所有方格，数字表示路线上的第几步。

输入形式：棋盘的大小 n（只考虑正方形的棋盘，即棋盘上共有 $n \times n$ 个方格）；骑士在棋盘上的起始位置 (x, y)。

输出形式：骑士完成一次巡逻的路线（即每个方格标注了从 1 到 $n \times n$ 不同数字的棋盘）。例如，对于一个 5×5 的棋盘，若骑士从 $(2, 2)$ 开始巡逻，则一种巡逻路线可以为：

$$\begin{bmatrix} 21 & 2 & 7 & 12 & 23 \\ 8 & 13 & 22 & 17 & 6 \\ 3 & 20 & 1 & 24 & 11 \\ 14 & 9 & 18 & 5 & 16 \\ 19 & 4 & 15 & 10 & 25 \end{bmatrix}$$

2 概要设计

1. 递归程序：考虑使用深度优先搜索遍历（DFS）实现。设置目标搜索深度为 $n \times n$（棋盘格数），从起始位置开始，向骑士能走的 8 个方向搜索，如果搜索失败就回溯。当搜索深度达到 n^2 时输出棋盘巡逻结果。

2. 非递归程序：考虑使用栈实现。设置目标搜索深度为 $n \times n$（棋盘格数），从起始位置开始，向骑士能走的 8 个方向搜索，将每次骑士走的坐标入栈，如果搜索失败就将栈顶坐标弹出。当搜索深度达到 n^2 时输出栈中每一个坐标。

2.1 抽象数据类型

在本程序中，用到的**抽象数据类型**定义如下所示：

ADT `coord`{
    数据对象：$D = \{c | c \in$ `int`$\}$
    数据关系：$R = \{\langle e_1, e_2 \rangle | e_1, e_2 \in$ `int`$, e_1$表示骑士所在行，e_2表示骑士所在列$\}$
    基本操作：
        `Position(row, col)` 操作结果：返回一个`coord`类型的坐标`(row,col)`
}
ADT `stack`{
    数据对象：$D = \{c | c \in$ `int`$\}$
    数据关系：$R = \{\langle e_1, e_2 \rangle | e_1$是一个`int[]`类型的数组，$e_2$是一个指向栈顶的指针$\}$
    基本操作：
        `InitStack()` 操作结果：创建一个字符串类型的栈
        `StackPush(s, value)` 操作结果：入栈骑士的坐标
        `StackPop(s)` 操作结果：将栈顶元素弹出
        `PrintStack(s, ChessBoard[N][N], n, sum)`
        操作结果：打印栈中储存的骑士巡逻坐标，n 为棋盘格大小，`sum`为骑士巡逻的方法数。
}

2.2 主程序流程

1. 创建棋盘`ChessBoard[N][N]`并将所有元素置 0，创建一个储存骑士坐标的栈`patrol`。
2. 输入棋盘的边长和骑士巡逻的初始位置。
3. 调用`KnightPatrol_Recursion(ChessBoard, row, col, n)`和`KnightPatrol_Stack(ChessBoard, row, col, n, patrol)`，输出骑士巡逻的所有可能路线。

3 详细设计

坐标coord和栈stack的基本操作设计在附录中有详细代码实现。这里主要阐述**两种骑士巡逻实现函数（递归回溯实现和栈实现）的详细设计**。

3.1 递归回溯法实现骑士巡逻路线

递归回溯法寻找骑士巡逻路线的函数定义如下：

```
void KnightPatrol_Recursion(int ChessBoard[N][N], int row, int col, int n);
```

其中ChessBoard[N][N]为大棋盘，row为骑士所在行，col为骑士所在列，n为棋盘边长。设N为全局变量，表示棋盘边长的上限。

算法设计如下：由于骑士最终必然要遍历完 n^2 个棋盘格，因此设置最大搜索深度为depth = n * n

1. 如果到达了最大搜索深度：
   (a) 当搜索到一种巡逻路线时，首先将这个巡逻路线打印出来。
   (b) 然后递归回溯，寻找其他的巡逻路线。
2. 如果没有到达最大搜索深度：向八个方向移动骑士，如果移动位置合法：
   (a) 标记骑士巡逻过这个棋盘格，并递归搜索。
   (b) 如果深度优先搜索一直没搜到解，则将搜索过的棋盘格置 0，并递归回溯，表示从这一步搜索退回来。

代码实现如下：

```
void KnightPatrol_Recursion(int ChessBoard[N][N], int row, int col, int n){
        //递归回溯法解决骑士巡逻问题
        //row, col 是骑士的行列
        if (depth >= n * n){ //搜索深度达到 n*n 说明已经搜到答案
                sum++;//骑士巡逻方法数 +1
                PrintChessboard(ChessBoard, n, sum);
                flag = 1;//骑士巡逻路线存在
                return;//回溯
        }
        for (int i = 0 ; i < 8 ; i++){//骑士分别向 8 个方向移动
                int new_row = row + step[i][0];
                int new_col = col + step[i][1];
                //如果移动方向合法，且该方向之前没有走过
                if (ValidIndex(new_row, new_col, n) == true && ChessBoard[new_row][new_col] == 0){
                        ChessBoard[new_row][new_col] = ++depth;
                        KnightPatrol_Recursion(ChessBoard, new_row, new_col, n);//从新位置开始搜索
                        //到这里仍然没搜到，则说明这条路走不通，需回溯
                        depth--;//搜索深度-1
```

```
                        ChessBoard[new_row][new_col] = 0;//搜索过的位置置零
                }
        }
}
```

3.2 依托栈的非递归法寻找骑士巡逻路线

依托栈的非递归法寻找骑士巡逻路线的函数定义如下：

```
void KnightPatrol_Stack(int ChessBoard[N][N], int row, int col, int n, stack *patrol);
```

其中ChessBoard[N][N]为大棋盘，row为骑士所在行，col为骑士所在列，n为棋盘边长，patrol为储存骑士移动位置的栈。设N为全局变量，表示棋盘边长的上限。

算法设计如下：由于骑士最终必然要遍历完 n^2 个棋盘格，因此设置最大搜索深度为depth = n * n

1. 如果到达了最大搜索深度：
   (a) 当搜索到一种巡逻路线时，首先将这个巡逻路线打印出来。
   (b) 然后递归回溯，寻找其他的巡逻路线。
2. 如果没有到达最大搜索深度：向八个方向移动骑士，如果移动位置合法：
   (a) 标记骑士巡逻过这个棋盘格，并将当前位置入栈，从这个位置开始下一步的巡逻。
   (b) 如果深度优先搜索一直没搜到解，则将搜索深度减 1，把栈顶元素弹出，表示从这一步搜索退回来。

代码实现如下：

```
void KnightPatrol_Stack(int ChessBoard[N][N], int row, int col, int n, stack *patrol){
        //非递归法解决骑士巡逻问题
        //row, col 是骑士的行列
        if (depth2 == n * n){ //搜索深度达到 n*n 说明已经搜到答案
                sum2++;//骑士巡逻方法数 +1
                PrintStack(patrol, ChessBoard, n, sum2);
                flag2 = 1;//骑士巡逻路线存在
                return;//回溯
        }
        for (int i = 0 ; i < 8 ; i++){//骑士分别向 8 个方向移动
                int new_row = row + step[i][0];
                int new_col = col + step[i][1];
                //如果移动方向合法，且该方向之前没有走过
                if (ValidIndex(new_row, new_col, n) == true && ChessBoard[new_row][new_col] == 0){
                        ChessBoard[new_row][new_col] = ++depth2;
                        coord *nextpos = Position(new_row, new_col);//记录新位置
                        StackPush(patrol, *nextpos);//将新位置入栈
```

```
18                          KnightPatrol_Stack(ChessBoard, new_row, new_col, n, patrol);//从新位置开始搜索
19                          //到这里仍然没搜到，则说明这条路走不通，需回溯
20                          ChessBoard[new_row][new_col] = 0;//搜索过的位置置零
21                          depth2--;//搜索深度-1
22                          StackPop(patrol);//将走不通的位置弹出栈
23                  }
24          }
25  }
```

4 用户使用说明

1. 程序运行后，首先显示”请输入棋盘的边长 n：”

- 输入一个正整数 n：n 的范围为`3 <= n <= 8`，骑士的走法决定其无法在边长 < 3 的棋盘中巡逻。当 $n > 8$ 时，递归回溯法的运算时间过长。

2. 其次显示”请输入骑士在棋盘上的起始位置`row, col`：”

- 输入两个正整数`row, col`：`row, col`的范围为`0 <= row, col <= n`。
- 两个正整数以空格分割。一个合法的输入为`0, 0`
- `0,0`、`00`等等都是不合法的输入。

4.1 测试结果

有以下测试样例供参考：

第一组：输入：`5`
`2 2`

第二组：输入：`8`
`0 0`

以第一组测试结果为例：

```
骑士巡逻方法 1:递归回溯法:
骑士巡逻路线 1
23 10 15 4 25
16 5 24 9 14
11 22 1 18 3
6 17 20 13 8
21 12 7 2 19

...(省略 63 个骑士巡逻路线结果)

骑士巡逻方法 2:依托栈实现的非递归法:
骑士巡逻路线 1
23 10 15 4 25
16 5 24 9 14
11 22 1 18 3
6 17 20 13 8
21 12 7 2 19

...(省略 63 个骑士巡逻路线结果)
```

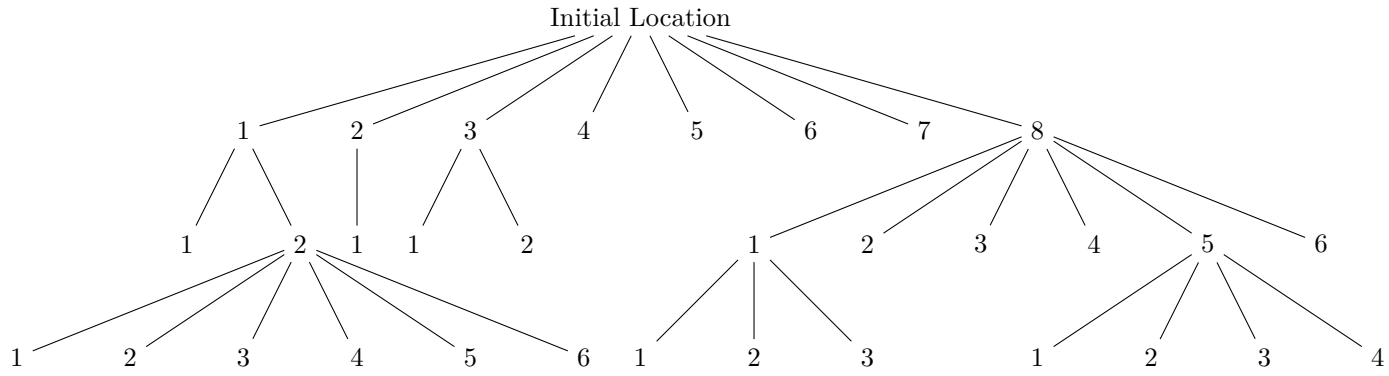
5 调试分析

5.1 算法的时空分析

5.1.1 递归程序

1. 时间复杂度：

我们可以将递归搜索的过程看作树的遍历：我们将每个坐标视作树的结点，由于骑士可以往 8 个方向移动，因此初始结点(x,y)的子节点最多有 8 个，每个子节点又有 8 个子节点，以此类推... 由于搜索深度为 n^2（棋盘格数），因此树的最大深度为 n^2。



尽管不是每个节点都有 8 个子节点（有一些位置下标越界或已经被访问过），但我们可以计算一个上界，即这个算法最坏情况下会生成 8^{n^2} 个状态空间，时间复杂度 $O(8^{n^2}) = O(2^n)$，是指数级别的。

棋盘的边长为算法运行时间的主要影响因素，当 $n > 5$ 时，算法的计算时间显著变长甚至不可计算（例如棋盘边长 $n = 6$，初始巡逻位置 $(0, 0)$ 时，共有 524486 种巡逻路线，耗时 991.126 秒）。

此外，骑士的初始位置队算法的运行时间也有影响。当 $n = 5$，初始位置为 $(0, 0)$ 的骑士巡逻路线共有 304 条，而当 $n = 5$，初始位置为 $(2, 2)$ 的骑士巡逻路线共有 64 条，显然初始位置在 $(0, 0)$ 的计算时间更长。

2. 空间复杂度：

如果我们用树的观点看待递归回溯法求解骑士巡逻问题的话，该算法占用结点数的上界为 8^{n^2}。因此算法的最坏空间复杂度为 $O(8^{n^2}) = O(2^n)$。

5.1.2 非递归程序

1. 时间复杂度：

类似地，从初始点开始搜索骑士巡逻路径，有最多 8 个方向可以走，选择一个方向后把这个位置压入栈，又有最多 8 个方向可以走... 以此类推，骑士共需要尝试 n^2 个位置，从而时间复杂度为 $O(8^{n^2}) = O(2^n)$。

2. 空间复杂度：

由于我们用栈保存骑士每次巡逻的坐标，用一个数组`ChessBoard[N][N]`保存骑士的巡逻顺序，因此算法的空间复杂度为 $O(N^2)$，其中 N 为棋盘的最大边长。

6 附录

详细的代码实现如下所示。也可以通过https://github.com/GONGSHUKAI/Data_Structure/tree/main/Lab_Code/Lab_1/Sept.29_Lab下载代码原文件。

```
#include <iostream>
#include <ctime>
#define N 8
#define MAXSIZE 1000

using namespace std;

//骑士的 8 个移动方向用 step[][] 记录
//第一个分量表示行位移，第二个分量表示列位移
int step[8][2]={{2,1},{1,2},{-1,2},{-2,1},{-2,-1},{-1,-2},{1,-2},{2,-1}};
int depth = 0;//搜索深度，最大为 n*n(棋盘格大小)
int depth2 = 0;//搜索深度，最大为 n*n(棋盘格大小)
int sum = 0;//记录骑士巡逻的方法数（回溯法）
int sum2 = 0;//记录骑士巡逻的方法数（非递归法）
int flag = 0;//判断骑士巡逻路线是否存在（回溯法）
int flag2 = 0;//判断骑士巡逻路线是否存在（非递归法）

typedef struct coord{
        int row;
        int col;
}coord;//存放骑士的位置 (row, col)

typedef struct stack{
        coord data[MAXSIZE];
        int top;
}stack;//存放骑士位置的栈

stack* InitStack();//初始化栈
bool StackEmpty(stack *s);//判断栈空
coord StackTop(stack *s);//判断栈满
void StackPush(stack *s, coord value);//入栈骑士的位置 (row, col)
void StackPop(stack *s);//弹出栈顶元素
//打印栈中储存的骑士巡逻坐标，n 为棋盘格大小，sum 为骑士巡逻的方法数
void PrintStack(stack *s, int ChessBoard[N][N], int n, int sum);

coord* Position(int row, int col);//输入骑士所在行列，返回骑士所在坐标
bool ValidIndex(int i, int j,int n);//判断 [i][j] 是否为一个合法的位置
//打印整个棋盘，n 为棋盘格大小，sum 为骑士巡逻的方法数
```

```
void PrintChessboard(int ChessBoard[N][N], int n, int sum);
//递归回溯法寻找骑士巡逻路线
void KnightPatrol_Recursion(int ChessBoard[N][N], int row, int col, int n);
//依托栈的非递归法寻找骑士巡逻路线
void KnightPatrol_Stack(int ChessBoard[N][N], int rol, int col, int n, stack *patrol);

int main(){
        int ChessBoard[N][N];
        int n;//棋盘大小
        int row, col;//骑士的起始位置
        memset(ChessBoard, 0, sizeof(ChessBoard));//将棋盘置 0

        cout << " 请输入棋盘的边长 n: ";
        cin >> n;
        cout << " 请输入骑士在棋盘上的起始位置 row, col: ";
        cin >> row >> col;

        cout << " 骑士巡逻方法 1: 递归回溯法: " << endl;
        clock_t startTime = clock();//计时开始
        ChessBoard[row][col] = ++depth;
        KnightPatrol_Recursion(ChessBoard, row, col, n);
        if (flag == 0) cout << " 不存在骑士巡逻路线" << endl;
        clock_t endTime = clock();//计时结束
        cout << " 运行时间: " <<(double)(endTime - startTime) / CLOCKS_PER_SEC << "s" << endl;

        memset(ChessBoard, 0, sizeof(ChessBoard));//将棋盘置 0
        cout << " 骑士巡逻方法 2: 依托栈实现的非递归法: " << endl;
        clock_t startTime2 = clock();//计时开始
        stack *patrol = InitStack();//创建一个存放骑士坐标的栈
        coord *init = Position(row, col);//骑士的初始位置
        StackPush(patrol, *init);
        ChessBoard[row][col] = ++depth2;
        KnightPatrol_Stack(ChessBoard, row, col, n, patrol);
        if (flag2 == 0) cout << " 不存在骑士巡逻路线" << endl;
        clock_t endTime2 = clock();//计时结束
        cout << " 运行时间: " <<(double)(endTime2 - startTime2) / CLOCKS_PER_SEC << "s" << endl;

}

stack* InitStack(){
        stack *s = new stack;
        s->top = -1;//栈顶指针（即数组下标）赋初值-1
```

```
        return s;
}

bool StackEmpty(stack *s){//如果栈空返回 1，否则返回 0
        if (s->top == -1) return true;
        else return false;
}

coord StackTop(stack *s){//返回栈顶元素
        if (s->top > -1 && s->top < MAXSIZE) return s->data[s->top];
        else{
                coord wrongpos;
                wrongpos.row = -1;
                wrongpos.col = -1;
                return wrongpos;
        }
}

void StackPush(stack *s, coord value){//压入栈
        if (s->top == MAXSIZE - 1) return;
        else s->data[++s->top] = value;
}

void StackPop(stack *s){//弹出栈顶元素
        if (s->top == -1) return;
        else{
                s->top--;
        }
}

coord* Position(int row, int col){
        coord *pos = new coord;
        pos->row = row;
        pos->col = col;
        return pos;
}

bool ValidIndex(int i, int j,int n){//n 是棋盘的大小
        if (i >= 0 && j >= 0 && i < n && j < n) return true;
        else return false;
}
```

```
void PrintChessboard(int ChessBoard[N][N], int n, int sum){
        cout << " 骑士巡逻路线" << sum << endl;
        for (int i = 0 ; i < n ; i++){
                for (int j = 0 ; j < n ; j++){
                        cout << ChessBoard[i][j] << " ";
                }
                cout << endl;
        }
        cout << endl;
}

void PrintStack(stack *s, int ChessBoard[N][N], int n, int sum){
        for (int i = 0 ; i <= s->top ; i++){
                ChessBoard[s->data[i].row][s->data[i].col] = i+1;
        }
        PrintChessboard(ChessBoard, n, sum);
}

void KnightPatrol_Recursion(int ChessBoard[N][N], int row, int col, int n){
        //递归回溯法解决骑士巡逻问题
        //row, col 是骑士的行列
        if (depth >= n * n){ //搜索深度达到 n*n 说明已经搜到答案
                sum++;//骑士巡逻方法数 +1
                PrintChessboard(ChessBoard, n, sum);
                flag = 1;//骑士巡逻路线存在
                return;//回溯
        }
        for (int i = 0 ; i < 8 ; i++){//骑士分别向 8 个方向移动
                int new_row = row + step[i][0];
                int new_col = col + step[i][1];
                //如果移动方向合法，且该方向之前没有走过
                if (ValidIndex(new_row, new_col, n) == true && ChessBoard[new_row][new_col] == 0){
                        ChessBoard[new_row][new_col] = ++depth;
                        KnightPatrol_Recursion(ChessBoard, new_row, new_col, n);//从新位置开始搜索
                        //到这里仍然没搜到，则说明这条路走不通，需回溯
                        depth--;//搜索深度-1
                        ChessBoard[new_row][new_col] = 0;//搜索过的位置置零
                }
        }
}

void KnightPatrol_Stack(int ChessBoard[N][N], int row, int col, int n, stack *patrol){
```

```
        //非递归法解决骑士巡逻问题
        //rol, col 是骑士的行列
        if (depth2 == n * n){ //搜索深度达到 n*n 说明已经搜到答案
                sum2++;//骑士巡逻方法数 +1
                PrintStack(patrol, ChessBoard, n, sum2);
                flag2 = 1;//骑士巡逻路线存在
                return;//回溯
        }
        for (int i = 0 ; i < 8 ; i++){//骑士分别向 8 个方向移动
                int new_row = row + step[i][0];
                int new_col = col + step[i][1];
                //如果移动方向合法，且该方向之前没有走过
                if (ValidIndex(new_row, new_col, n) == true && ChessBoard[new_row][new_col] == 0){
                        ChessBoard[new_row][new_col] = ++depth2;
                        coord *nextpos = Position(new_row, new_col);//记录新位置
                        StackPush(patrol, *nextpos);//将新位置入栈
                        KnightPatrol_Stack(ChessBoard, new_row, new_col, n, patrol);//从新位置开始搜索
                        //到这里仍然没搜到，则说明这条路走不通，需回溯
                        ChessBoard[new_row][new_col] = 0;//搜索过的位置置零
                        depth2--;//搜索深度-1
                        StackPop(patrol);//将走不通的位置弹出栈
                }
        }
}
```