题目 机器人的运动范围

考点 回溯法 热点指数 26084 通过率 22.71%

具体题目

地上有一个m行和n列的方格。一个机器人从坐标0,0的格子开始移动，每一次只能向左，右，上，下四个方向移动一格，但是不能进入行坐标和列坐标的数位之和大于k的格子。 例如，当k为18时，机器人能够进入方格（35,37），因为3+5+3+7 = 18。但是，它不能进入方格（35,38），因为3+5+3+8 = 19。请问该机器人能够达到多少个格子？

```
/**算法本质：
* DFS||BFS 寻找连通分量
*
 * 题目分析：
 * 机器人在一个矩阵上的m*n个格子上移动，可进入的格子的集合可抽象为以下点集：
 * { (row, col) | (i%10+i/10+j%10+j/10) <= threshold }。且路径节点可重复，无步数限制。
 * 问：机器人能到达多少个格子？
 *
 * 题目抽象：
 * 倘若我们把矩阵的每一个“格子”抽象成一个“结点”，把“格子相邻”抽象为“结点连通”（结点之间存在无向边），
 * 把“无法进入的格子”抽象成“与所有普通结点都不连通（不存在无向边）的孤点”，则整个问题可以抽象为：
 * 从某个结点出发，寻找无向图的连通分量的节点个数。很显然，可以使用DFS或者BFS进行实现
 *
 * 算法实现：
 * 这里选择DFS进行实现。
 * 设置两个辅助boolean矩阵：visited与isWall。前者是DFS中的典型辅助矩阵，记录每个节点是否已访问过。
 * 后者用来表示每个节点是否是不能进入的“孤点”。
 * 设置静态变量nodeCnt，用于在DFS的过程中记录访问过的结点数
 * DFS递归函数的出口条件设置为：
 * (outOfBoundary(rows, cols, row, col) || visited[row][col] || isWall[row][col] )
 * 即：“若超过边界（到矩阵之外）”或“访问过”或“是无法进入的结点” 则 return
 * 然后进行DFS。
 * */
int nodeCnt = 0;
boolean[][] visited;
boolean[][] isWall;
int threshold;
int rows;
int cols;
public int movingCount(int threshold, int rows, int cols){
    if (threshold<0 || rows<=0 || cols<=0) //robust
        return 0;//牛客示例是0
    //init
    this.nodeCnt = 0;
    this.threshold = threshold;
    this.rows = rows;
    this.cols = cols;
    this.visited = new boolean[rows][cols];
    this.isWall = new boolean[rows][cols];
    for (int i=0;i<rows;i++){
        for (int j=0;j<cols;j++){
            this.visited[i][j]=false;
            if ( (i%10+i/10+j%10+j/10) > threshold )
                this.isWall[i][j]=true;
            else
                this.isWall[i][j]=false;
        }
    }
```

```
        //body
        DFS(0,0);
        return this.nodeCnt;
    }
    public void DFS(int row, int col){
        if (   outOfBoundary(rows, cols, row, col)
            || visited[row][col]
            || isWall[row][col] )
            return;
        //visit
        visited[row][col]=true;
        nodeCnt++;
        //DFS
        DFS(row+1, col);
        DFS(row-1, col);
        DFS(row, col+1);
        DFS(row, col-1);
    }
    public boolean outOfBoundary(int rows, int cols, int row, int col){
        return ( row<0 || row>=rows || col<0 || col>=cols );
    }
}
```

---

public class Solution {

```
public int movingCount(int threshold, int rows, int cols) {
   int[][] flag = new int[rows][cols];
   return moving(threshold, rows, cols, flag, 0, 0);
}

public int moving(int threshold, int rows, int cols, int[][] flag, int i, int j){
    if(threshold <= 0 || i >= rows || i< 0 || j >= cols || j < 0 || (flag[i][j] == 1) ||
(sum(i) + sum(j) > threshold)){
        return 0;
    }
    flag[i][j] = 1;
    return moving(threshold, rows, cols, flag, i - 1, j)
        +moving(threshold, rows, cols, flag, i + 1, j)
        +moving(threshold, rows, cols, flag, i, j - 1)
        +moving(threshold, rows, cols, flag, i, j + 1)
        + 1;
}

public int sum(int i ){
    if(i == 0){return i ;}
    int sum = 0;
    while(i != 0){
        sum += i % 10;
        i /= 10;
    }
    return sum;
}

}
```

【java】和上一题类似，本题依然用DFS来解题，依然提供递归和非递归两种方法，了解一下！ 方法一：非递归 思路：不带记忆的DFS搜索 + 限定条件 = 普通的DSF例题 1.需要记录已经遍历过的节点，用辅助矩阵visited[rows * cols] 2.每次加入栈时，count++，标记已经遍历，这样下一个节点就不会遍历了 入栈条件： 1.每一位的和小于等于threshold： 2.x和 y 的边界条件 3.没有遍历过

```
 public int movingCount(int threshold, int rows, int cols)
    {
       if(rows <= 0 || cols <= 0 || threshold < 0) return 0;
        Stack<Integer> s = new Stack<>();
        boolean[] visited = new boolean[rows * cols];
        int[][] xoy = {{0,1,0,-1},{1,0,-1,0}};
        int count = 0;
        s.add(0);
        visited[0] = true;
        while(!s.empty()) {
            int  cur = s.pop();
            count++;
            for (int i = 0; i < 4; i++) {
                int x = cur % cols + xoy[0][i];
                int y = cur / cols + xoy[1][i];
                int sum = getDigitSum(x) + getDigitSum(y);
                if(x >= 0 && x < cols && y >= 0 && y < rows
                        && sum <= threshold && !visited[x + y * cols]) {
                    s.add(x + y * cols);
                    visited[x + y * cols] = true;
                }
            }
        }
        return count;
    }
    private int getDigitSum(int i) {//获取位的和
        int sum = 0;
        while(i > 0) {
            sum += i % 10;
            i /= 10;
        }
        return sum;
    }
```

方法二：递归 * 递归的方式更加简单了，比上一题简单 * 出口： * 0:不满足边界条件 ；已经遍历过；位数和大于阈值 * 1.递： * 1.1标记遍历 * 1.2上下左右递归 * 2.归：返回count public int movingCount(int threshold, int rows, int cols) { if(rows <= 0 || cols <= 0 || threshold < 0) return 0; boolean[] visited = new boolean[rows * cols]; return dfs(threshold,rows,cols,visited,0,0); } private int dfs(int threshold, int rows, int cols, boolean[] visited, int x, int y) { if(x < 0 || x >= cols || y < 0 || y >= rows || getDigitSum(x) + getDigitSum(y) > threshold || visited[x + y * cols]) return 0;//出口 visited[x + y * cols] = true;//标记 return 1 + dfs(threshold, rows, cols, visited, x, y - 1)//归

- dfs(threshold, rows, cols, visited, x + 1, y)
- dfs(threshold, rows, cols, visited, x, y + 1)
- dfs(threshold, rows, cols, visited, x - 1, y);

  private int getDigitSum(int i) { int sum = 0; while(i > 0) { sum += i % 10; i /= 10; } return sum; }

---

import java.util.*; public class Solution { ArrayList result = new ArrayList(); public int movingCount(int threshold, int rows, int cols) { int len = rows*cols; int[] state= new int[len]; return core(threshold,0,0,state,0,rows,cols);

```
}
public int core(int k,int i,int j ,int[] state,int step,int rows,int cols){
    int count = 0;
    if(canIn(k,i,j ,state,step,rows,cols)){
        state[i*cols+j]=1;
```

```
            count = 1+core(k,i+1,j ,state,step,rows,cols)+core(k,i-1,j ,state,step,rows,cols)+
                core(k,i,j+1 ,state,step,rows,cols)+core(k,i,j-1 ,state,step,rows,cols);
        }
        return count;
    }
    public boolean canIn(int k,int i,int j ,int[] state,int step,int rows,int cols){
        int index = cols*i+j;
        if(i>=0&&j>=0&&i<rows&&j<cols&&state[index]!=1&&getSum(i,j)<=k){return true;}
        return false;
    }
    public int getSum(int i ,int j ){
        int sum = 0;
        while(i>0){
            sum+=i%10;
            i/=10;
        }
        while(j>0){
            sum+=j%10;
            j/=10;
        }
        return sum;
    }

}
```

用回溯法实现，从起点出发，从每个点的左右上下开始寻找，如果任何一个方向已经寻找过或者超出边界或者不满足条件，则停止这个方向的寻找，从另外一个方向开始寻找每次满足条件，则满足条件个数+1，这样一直找，直到没有满足条件的点，用栈来存储满足条件的点

```
public class Solution {
   int count = 0;    //需要一个矩阵判断是否访问过    boolean[][] visited;
   public int movingCount(int threshold, int rows, int cols)    {        visited = new boolean[rows][cols];        //dfs深度优先遍历，从0,0下标开始        countGrid(0,0,threshold,rows,cols);        return count;    }
   public void countGrid(int i,int j,int k,int rows,int cols){        //越界，或不满足进入格子条件，或已经访问过，均不再递归        //dfs中，一个访问过的节点，以其开始的所有路径必定已经被访问过了，因此需要过滤        if(i < 0 || j < 0 || i>=rows || j >= cols || (getSum(i) + getSum(j)>k) || visited[i][j])            return;
       ++count;        visited[i][j] = true;        //上        countGrid(i-1,j,k,rows,cols);        //下        countGrid(i+1,j,k,rows,cols);        //左        countGrid(i,j-1,k,rows,cols);        //右        countGrid(i,j+1,k,rows,cols);
}    //获取指定下标数位和    public int getSum(int index){        int sum = 0;        while(index != 0){            sum += index % 10;            index /= 10;        }        return sum;    } }
```