题目 矩阵中的路径

考点 回溯法 热点指数 25170 通过率 21.84%

具体题目

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则之后不能再次进入这个格子。 例如 a b c e s f c s a d e e 这样的3 X 4 矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

```
/**
用一个状态数组保存之前访问过的字符，然后再分别按上，下，左，右递归
*/
public class Solution {
    public boolean hasPath(char[] matrix, int rows, int cols, char[] str) {
        int flag[] = new int[matrix.length];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (helper(matrix, rows, cols, i, j, str, 0, flag))
                    return true;
            }
        }
        return false;
    }
    private boolean helper(char[] matrix, int rows, int cols, int i, int j, char[] str, int
k, int[] flag) {
        int index = i * cols + j;
        if (i < 0 || i >= rows || j < 0 || j >= cols || matrix[index] != str[k] ||
flag[index] == 1)
            return false;
        if(k == str.length - 1) return true;
        flag[index] = 1;
        if (helper(matrix, rows, cols, i - 1, j, str, k + 1, flag)
                || helper(matrix, rows, cols, i + 1, j, str, k + 1, flag)
                || helper(matrix, rows, cols, i, j - 1, str, k + 1, flag)
                || helper(matrix, rows, cols, i, j + 1, str, k + 1, flag)) {
            return true;
        }
        flag[index] = 0;
        return false;
    }
}
```

回溯 基本思想：
0.根据给定数组，初始化一个标志位数组，初始化为false，表示未走过，true表示已经走过，不能走第二次
1.根据行数和列数，遍历数组，先找到一个与str字符串的第一个元素相匹配的矩阵元素，进入judge

2. 根据i和j先确定一维数组的位置，因为给定的matrix是一个一维数组
3. .确定递归终止条件：越界，当前找到的矩阵值不等于数组对应位置的值，已经走过的，这三类情况，都直接false，说明这条路不通
4. .若k，就是待判定的字符串str的索引已经判断到了最后一位，此时说明是匹配成功的
5. 下面就是本题的精髓，递归不断地寻找周围四个格子是否符合条件，只要有一个格子符合条件，就继续再找这个符合条件的格子的四周是否存在符合条件的格子，直到k到达末尾或者不满足递归条件就停止。
6. 走到这一步，说明本次是不成功的，我们要还原一下标志位数组index处的标志位，进入下一轮的判断。

```
public class Solution {
    public boolean hasPath(char[] matrix, int rows, int cols, char[] str)
    {
        //标志位，初始化为false
        boolean[] flag = new boolean[matrix.length];
        for(int i=0;i<rows;i++){
            for(int j=0;j<cols;j++){
                 //循环遍历二维数组，找到起点等于str第一个元素的值，再递归判断四周是否有符合条件的----回溯
法
                 if(judge(matrix,i,j,rows,cols,flag,str,0)){
                     return true;
                 }
            }
        }
        return false;
    }

    //judge(初始矩阵，索引行坐标i，索引纵坐标j，矩阵行数，矩阵列数，待判断的字符串，字符串索引初始为0即先判断
字符串的第一位)
    private boolean judge(char[] matrix,int i,int j,int rows,int cols,boolean[] flag,char[]
str,int k){
        //先根据i和j计算匹配的第一个元素转为一维数组的位置
        int index = i*cols+j;
        //递归终止条件
        if(i<0 || j<0 || i>=rows || j>=cols || matrix[index] != str[k] || flag[index] ==
true)
            return false;
        //若k已经到达str末尾了，说明之前的都已经匹配成功了，直接返回true即可
        if(k == str.length-1)
            return true;
        //要走的第一个位置置为true，表示已经走过了
        flag[index] = true;

        //回溯，递归寻找，每次找到了就给k加一，找不到，还原
        if(judge(matrix,i-1,j,rows,cols,flag,str,k+1) ||
           judge(matrix,i+1,j,rows,cols,flag,str,k+1) ||
           judge(matrix,i,j-1,rows,cols,flag,str,k+1) ||
           judge(matrix,i,j+1,rows,cols,flag,str,k+1)  )
        {
            return true;
        }
        //走到这，说明这一条路不通，还原，再试其他的路径
        flag[index] = false;
        return false;
    }
}
```

核心思路：回溯法 1.先将matrix字符串映射为字符矩阵； 2.从原字符串中找到第一个跟str[0]相等的字符，得到其对应的在矩阵中的位置[r,c] 1）从[r,c]开始按 上、左、右、下的顺序搜索； 2）每当搜索到一个节点，先判断path是否包括它，包括就说明已经经过此节点，不能 再经过了；如果不包括，就将其加入path容器 3）直到搜索到str[length - 1]节点，说明成功找到，标记result为true，标记 isFinished为true,尽快结束所有的递归操作 4）如果某节点起的 上、左、右、下 都搜索过但还没找到结果，说明经过此节点的路 径都不满足题意，将其从path中删除，回溯到上一层继续找。（PS:确实是回溯法，不过代码有点长，实现的有点繁杂） 运行时间：51ms 占用内存：625k */ //   private char[][] data; //   private int rows; //   private int cols; //   private LinkedList path = new LinkedList(); //   private boolean result = false; //   private boolean isFinished = false; // //   public boolean hasPath(char[] matrix, int rows, int cols, char[] str){ //   this.rows = rows; //   this.cols = cols; //   data = new char[rows][cols]; //   for (int i = 0,k = 0; i < rows; i ++) { //   for (int j = 0; j < cols; j ++) { //   data[i][j] = matrix[k ++]; //   } //   } // //   int r,c; //   for (int i = 0; i < matrix.length; i++) { //   if (matrix[i] == str[0] && !isFinished){ //   r

= i / cols; // c = i % cols; // tryPath(r,c,str,0); // } // } // // return result; // } // // public void tryPath(int r,int c,char[] str,int index){ // if (isFinished) return; // if (path.contains(r * cols + c)) return; // // path.addLast(r * cols + c); // // if (index == str.length - 1) { // isFinished = true; // result = true; // } // else { // for (int i = 0; i < 4; i++) { // switch (i) { // case 0: // if (r - 1 >= 0 && data[r - 1][c] == str[index + 1]) { // tryPath(r - 1, c, str, index + 1); // } // break; // case 1: // if (c - 1 >= 0 && data[r][c - 1] == str[index + 1]) { // tryPath(r, c - 1, str, index + 1); // } // break; // case 2: // if (r + 1 < rows && data[r + 1][c] == str[index + 1]) { // tryPath(r + 1, c, str, index + 1); // } // break; // case 3: // if (c + 1 < cols && data[r][c + 1] == str[index + 1]) { // tryPath(r, c + 1, str, index + 1); // } // break; // } // } // } // path.removeLast(); // }

【java】标准的带记忆DFS搜索，提供递归和非递归 两种 方法，了解一下。 一，标准的DFS，非递归，了解一下 思路：带记忆的BFS或者DFS， 需要辅助容器帮助记录路径，选用栈stack，还需要标记是否遍历过，用boolean[] visited 1.DFS深度优先， 进：peek一次，str的位子index++，对应位子visited[i+j*rows]=true， 并且把周围合适的点（上下左右&&字符匹配&&未遍历）加入到stack中 退:当前遍历过，复位：visited设为false，并且s.pop移除当前元素，str的位置减一 2.如果str匹配成功就返回true 是不是看起来很简单，接下来看实现 public boolean hasPath(char[] matrix, int rows, int cols, char[] str) { if(matrix == null || matrix.length != rows * cols || str == null || str.length == 0 || str.length > matrix.length) return false; boolean[] visited = new boolean[matrix.length]; for (int j = 0; j < rows; j++) { for (int i = 0; i < cols; i++) {//每个节点都有可能是起点 if(dfs(matrix,rows,cols,str,i,j,visited)) return true; } } return false; } //这里方便遍历上下左右 private static int[] x = {0,1,0,-1};//顺时针 private static int[] y = {1,0,-1,0};//顺时针 //这里复用了boolean[] visited 减少内存开销 private boolean dfs(char[] matrix, int rows, int cols, char[] str, int i, int j, boolean[] visited) { if(matrix[i + j * cols] != str[0]) return false;//第一个字符必须相等 Stack s = new Stack<>();//存的是坐标 int index = 0;//当前str的索引 s.push(i + j * cols); while(!s.empty()) { int location = s.peek(); if(visited[location] == true){//访问过,全部复位 visited[location] = false;//取消访问记录 s.pop();//退出该节点 if(--index < 0) return false; continue;//防止该路径再次遍历 } visited[location] = true;//标记已访问 if(++index == str.length) return true;//如果这个字符恰好是最后一个字符，直接返回true /*

- 将当前节点周围(上下左右)符合标准的点加入到s中,
- 1.边界条件：i = location % cols j = location / cols i和j判断边界
- 2.必须未遍历过visited[cur] == false
- 3.当前字符匹配matrix[cur] == str[index]

for (int k = 0; k < 4; k++) { int xn = location % cols + x[k]; int yn = location / cols + y[k]; int cur = xn + yn * cols; if(xn >= 0 && xn < cols && yn >= 0 && yn < rows && visited[cur] == false && matrix[cur] == str[index]) { s.push(cur); } } } return false; } 二,前面大佬都是递归，没有新意，为了保证完整性，还是加上，递归也确实容易理解，代码简单 用递归来实现DFS 1.确定出口： false:1.边界条件不满足，2.当前字符不匹配，3.已经遍历过 true：字符串str已经遍历结束 2.递：设置访问过 递归方式，按照上下左右递归 3.归：复位，未访问 public boolean hasPath(char[] matrix, int rows, int cols, char[] str) { if(matrix == null || matrix.length != rows * cols || str == null || str.length == 0 || str.length > matrix.length) return false; boolean[] visited = new boolean[matrix.length]; for (int j = 0; j < rows; j++) { for (int i = 0; i < cols; i++) {//每个节点都有可能是起点 if(dfs(matrix,rows,cols,str,i,j,0,visited)) return true; }//这里多了个k=0来充当str的索引 } return false;

} //递归开始，真是短啊 private boolean dfs(char[] matrix, int rows, int cols, char[] str, int i, int j, int k, boolean[] visited) { if(i < 0 || i >= cols || j < 0 || j >= rows || visited[i + j * cols] || matrix[i + j * cols] != str[k]) return false; if(k == str.length - 1) return true;//出口 visited[i + j * cols] = true;//递 if(dfs(matrix, rows, cols, str, i, j - 1, k + 1, visited) || dfs(matrix, rows, cols, str, i + 1, j, k + 1, visited) || dfs(matrix, rows, cols, str, i, j + 1, k + 1, visited) || dfs(matrix, rows, cols, str, i - 1, j, k + 1, visited))

return true;　　visited[i + j * cols] = false;//归　　return false;　}
```

---

```
public class Solution {
    public boolean hasPath(char[] matrix, int rows, int cols, char[] str){
        for(int i = 0; i < rows; i ++){
            for(int j = 0; j < cols; j ++){
                if(isPath(matrix, str, 0,
                          i, j, rows, cols)){
                    return true;
                }
            }
        }
        return false;
    }
    //   i 当前行   j 当前列   len已经计算的长度
    public boolean isPath(char[] matrix, char[] str, int len,
                          int i, int j, int rows, int cols){
        //先根据i和j计算匹配的第一个元素转为一维数组的位置
        int index = i * cols + j;
        //不等...该路径不是
        if(str[len] != matrix[index]){
            return false;
        }
        //长度够了...该路径是
        if(str.length == len + 1){
            return true;
        }
        char c = matrix[index];
        //访问过了改个状态
        matrix[index] = '*';
        int[] di = {-1, 0, 1, 0}, dj = {0, 1, 0, -1};
        for(int k = 0; k < 4; k ++){
            int a = i + di[k];
            int b = j + dj[k];
            if((a > -1) && (a < rows) && (b > -1) && (b < cols)
               && (len + 1 < str.length)){//不越界就搞下一个上下左右走
                if(isPath(matrix, str, len + 1, a, b, rows, cols)){
                    return true;
                }
            }
        }
        //状态改回来
        matrix[index] = c;
        return false;
    }
}
```
```