【回溯】-找到它

题目描述与示例

题目描述

找到它是个小游戏，你需要在一个矩阵中找到给定的单词

假设给定单词 `HELLOWORLD` ，在矩阵中只要能找 `HELLOWORLD` 就算通过

注意区分英文字母大小写，并且你只能上下左右行走，不能走回头路

输入描述

输入第一行包含两个整数 `N M (0 < N, M < 21)`

分别表示 `N` 行 `M` 列的矩阵

第二行是长度不超过 `100` 的单词 `W`

在整个矩阵中给定单词 `W` 只会出现一次

从第 `3` 行到第 `N+2` 是只包含大小写英文字母的长度为 `M` 的字符串矩阵

输出描述

如果能在矩阵中连成给定的单词，则输出给定单词首字母在矩阵中的位置为第几行第几列
否则输出 `NO`

示例一

输入

```
5 5
HELLOWORLD
CPUCY
EKLQH
CHELL
LROWO
DGRBC
```

输出

```
1 3 2
```

示例二

输入

```
1 5 5
2 Helloworld
3 CPUCh
4 wolle
5 orldO
6 EKLQo
7 PGRBC
```

输出

```
1 NO
```

解题思路

> 注意，本题和LeetCode79. 单词搜索几乎完全一致，唯一的区别在于前者只要求判断是否能够找到该单词，本题还需要输出起始位置。

状态更新和回滚写在横向遍历for循环内的回溯写法

我们需要思考的是，对于这种二维网格如何进行回溯？换句话说，如何构建回溯函数？

在回溯过程中我们需要知道以下信息：

1. 当前进行到了单词中的哪一个字符?
2. 当前在网格中搜索到了哪一个位置?
3. 由于网格中的字符不能重复使用，那么哪一些字符是已经使用过的?
4. 是否已经在网格中找到了这个单词?

对于第一点，我们的回溯函数中需要存在参数 `word_idx` ，来表示待搜索的单词此时遍历到的索引位置。

对于第二点，我们的回溯函数需要传入当前搜索的点的位置 `(x, y)`

对于第三点，这个在二维网格类型的搜索问题中是非常常用的技巧，即构建一个大小和 `grid` 一样的 `check_list`

对于第四点，我们可以直接声明一个全局变量 `isFind` 来表示是否已经找到该单词

除了这些参数之外，我们还需要传入二维矩阵 `grid` 本身，它的大小 `N` 和 `M` 等等。

容易构建出回溯函数如下

```
DIRECTIONS = [(0,1), (1,0), (-1,0), (0,-1)]

def backtracking(grid, N, M, check_list, x, y, word, word_idx):
    global isFind
    if word_idx == len(word) - 1:
        isFind = True
        return
    for dx, dy in DIRECTIONS:
        nx, ny = x+dx, y+dy
        if (0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and
        check_list[nx][ny] == False and grid[nx][ny] == word[word_idx+1]):
            check_list[nx][ny] = True
            backtracking(grid, N, M, check_list, nx, ny, word, word_idx+1)
            check_list[nx][ny] = False
```

容易发现，此处回溯函数的写法，和我们用DFS做二维网格搜索类型的题目是非常类似的。

换句话说，在当前点 `(x, y)` 的近邻点上下左右四个方向的选取的这个 `for` 循环，实际上就对应着回溯过程中状态树的横向遍历。

和常规DFS解法的区别在于，我们现在搜索的是一条路径，所以我们需要在递归调用 `backtracking()` 函数的前后，进行 `check_list[nx][ny]` 的状态更新和回滚，来表示近邻点 `(nx, ny)` 已经被使用过以及回滚之后再次可以被使用的情况。

在递归调用回溯函数的时候，我们需要将下一个点 `(nx, ny)` 以及 `word` 的下一个字符索引 `word_idx+1` 传入函数中。

回溯的终止条件也非常简单，就是当 `word_idx` 已经等于 `len(word)-1` 了，说明整个 `word` 的所有字符都能够在二维网格中找到，那么修改 `isFind` 为 `True` ，同时退出搜索。

而递归入口则需要这样调用

```
isFind = False
check_list = [[False] * M for _ in range(N)]

for i in range(N):
    for j in range(M):
        if grid[i][j] == word[0]:
            check_list[i][j] = True
            backtracking(grid, N, M, check_list, i, j, word, 0)
            check_list[i][j] = False
            if isFind:
                print("{} {}".format(i+1, j+1))
                break
    if isFind:
        break
```

注意到，由于在回溯函数中修改 `check_list` 始终是对 `(nx, ny)` 进行修改，所以我们在做起始点搜索的双重循环的时候，在递归函数入口处，需要对起始点 `(i, j)` 额外地进行 `check_list` 的状态更新和回滚。

状态更新和回滚写在横向遍历for循环外的回溯写法

如果你想直接修改 `check_list[x][y]` 而不是修改 `check_list[nx][ny]`，那么回溯函数也可以改成这样

```
DIRECTIONS = [(0,1), (1,0), (-1,0), (0,-1)]

def backtracking(grid, N, M, check_list, x, y, word, word_idx):
    global isFind
    if word_idx == len(word) - 1:
        isFind = True
        return

    check_list[x][y] = True

    for dx, dy in DIRECTIONS:
        nx, ny = x+dx, y+dy
        if (0 <= nx < len(grid) and 0 <= ny < len(grid[0])
        and check_list[nx][ny] == False and grid[nx][ny] == word[word_idx+1]):
            backtracking(grid, N, M, check_list, nx, ny, word, word_idx+1)

    check_list[x][y] = False
```

这样就跟常规的DFS解法更加接近，但和常规的回溯题目的相似性就没那么高了。

因为状态更新和回滚写在了横向遍历 `for` 循环的外部。

对应的，由于此处状态的是 `(x, y)` 而非 `(nx, ny)`，那么在递归入口处就可以不用单独进行 `(i, j)` 的更新了。即递归入口可以写为

```
isFind = False
check_list = [[False] * M for _ in range(N)]

for i in range(N):
    for j in range(M):
        if grid[i][j] == word[0]:
            backtracking(grid, N, M, check_list, i, j, word, 0)
            if isFind:
                print("{} {}".format(i+1, j+1))
                break
    if isFind:
        break
```

代码

Python

```python
# 题目：2024E-找到它
# 分值：200
# 作者：许老师-闭着眼睛学数理化
# 算法：回溯
# 代码看不懂的地方，请直接在群上提问


# 全局的方向数组，表示上下左右移动四个方向
DIRECTIONS = [(0,1), (1,0), (-1,0), (0,-1)]

# 构建回溯函数，各个参数的含义为
# grid:          原二维矩阵
# N,M:           原二维矩阵的行数、列数
# check_list:    大小和grid一样的检查列表，用于判断某个点是否已经检查过
# x,y:           当前在grid中的点的坐标
# word:          待搜索的单词
# word_idx:      待搜索的单词此时遍历到的索引位置
def backtracking(grid, N, M, check_list, x, y, word, word_idx):
    # 声明全局变量isFind
    global isFind
    # 若此时word_idx等于word的长度-1
    # 说明word中的所有字母都在grid中找到了
    # 修改isFind为True，同时终止递归
    if word_idx == len(word) - 1:
        isFind = True
        return
    # 遍历四个方向，获得点(x,y)的近邻点(nx,ny)
    for dx, dy in DIRECTIONS:
        nx, ny = x+dx, y+dy
        # (nx,ny)必须满足以下三个条件，才可以继续进行回溯函数的递归调用
        # 1. 不越界；2. 尚未检查过；
        # 3.在grid中的值grid[nx][ny]为word的下一个字符word[word_idx+1]
        if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and check_list[nx][ny] == False and grid[nx][ny] == word[word_idx+1]:
            # 状态更新，将点(nx,ny)在check_list中的状态更新为True
            check_list[nx][ny] = True
            # 回溯，将点(nx,ny)传入回溯函数中，注意此时word_idx需要+1
```

```
            backtracking(grid, N, M, check_list, nx, ny, word, word_idx+1)
            # 回滚，将点(nx,ny)在check_list中的状态重新修改回False
            check_list[nx][ny] = False


# 输入行数和列数
N, M = map(int, input().split())
# 输入待查找的单词
word = input()
# 构建二维网格
grid = list()
for _ in range(N):
    grid.append(input())

# 构建全局变量isFind，初始化为False
isFind = False
# 构建大小和grid一样的检查数组check_list
# 用于避免出现重复检查的情况
check_list = [[False] * M for _ in range(N)]
# 双重遍历整个二维网格grid
for i in range(N):
    for j in range(M):
        # 找到点(i,j)等于word的第一个字母
        # 则点(i,j)可以作为递归的起始位置
        if grid[i][j] == word[0]:
            # 将点(i,j)在check_list中设置为已检查过
            check_list[i][j] = True
            # 回溯函数递归入口
            backtracking(grid, N, M, check_list, i, j, word, 0)
            # 将点(i,j)在check_list中重置为未检查过，因为本次回溯不一定找到答案
            check_list[i][j] = False
            # 如果在回溯中，全局变量isFind被改为True，说明找到了单词
            if isFind:
                # 输出行数和列数，注意在问题中行数和列数是从1开始计数的
                # 所以存在一个+1操作
                print("{} {}".format(i+1, j+1))
                # 同时可以直接退出循环
                break
    if isFind:
        break

if not isFind:
    print("NO")
```

Java

```java
import java.util.Scanner;

public class Main {
    // Global directions array to represent four directions: up, down, left, right
    private static final int[][] DIRECTIONS = {{0, 1}, {1, 0}, {-1, 0}, {0, -1}};
    private static boolean isFind = false;

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int N = scanner.nextInt();
        int M = scanner.nextInt();
        scanner.nextLine(); // Consume newline
        String word = scanner.nextLine();

        char[][] grid = new char[N][M];
        for (int i = 0; i < N; i++) {
            String row = scanner.nextLine();
            for (int j = 0; j < M; j++) {
                grid[i][j] = row.charAt(j);
            }
        }

        boolean[][] checkList = new boolean[N][M];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < M; j++) {
                if (grid[i][j] == word.charAt(0)) {

                    checkList[i][j] = true;
                    backtracking(grid, N, M, checkList, i, j, word, 0);
                    checkList[i][j] = false;

                    if (isFind) {
                        System.out.println((i + 1) + " " + (j + 1));
                        return;
                    }
                }
            }
        }

        if (!isFind) {
            System.out.println("NO");
        }
    }

```

```
    private static void backtracking(char[][] grid, int N, int M, boolean[][] checkList, int x, int y, String word, int wordIdx) {
        if (wordIdx == word.length() - 1) {
            isFind = true;
            return;
        }

        for (int[] dir : DIRECTIONS) {
            int nx = x + dir[0];
            int ny = y + dir[1];

            if (nx >= 0 && nx < N && ny >= 0 && ny < M && !checkList[nx][ny] && grid[nx][ny] == word.charAt(wordIdx + 1)) {
                checkList[nx][ny] = true;
                backtracking(grid, N, M, checkList, nx, ny, word, wordIdx + 1);
                checkList[nx][ny] = false;
            }
        }
    }
}

```

C++

```
#include <iostream>
#include <vector>

using namespace std;

// Global directions array to represent four directions: up, down, left, right
const vector<pair<int, int>> DIRECTIONS = {{0, 1}, {1, 0}, {-1, 0}, {0, -1}};
bool isFind = false;

// Backtracking function
void backtracking(vector<vector<char>>& grid, int N, int M, vector<vector<bool>>& checkList, int x, int y, string& word, int wordIdx) {
    if (wordIdx == word.length() - 1) {
        isFind = true;
        return;
    }

    for (const auto& dir : DIRECTIONS) {
        int nx = x + dir.first;
        int ny = y + dir.second;

```

```cpp
        if (nx >= 0 && nx < N && ny >= 0 && ny < M && !checkList[nx][ny] && grid[nx][ny] == word[wordIdx + 1]) {
            checkList[nx][ny] = true;
            backtracking(grid, N, M, checkList, nx, ny, word, wordIdx + 1);
            checkList[nx][ny] = false;
        }
    }
}

int main() {
    int N, M;
    cin >> N >> M;
    string word;
    cin >> word;

    vector<vector<char>> grid(N, vector<char>(M));

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            cin >> grid[i][j];
        }
    }

    vector<vector<bool>> checkList(N, vector<bool>(M, false));

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            if (grid[i][j] == word[0]) {

                checkList[i][j] = true;
                backtracking(grid, N, M, checkList, i, j, word, 0);
                checkList[i][j] = false;

                if (isFind) {
                    cout << i + 1 << " " << j + 1 << endl;
                    return 0;
                }
            }
        }
    }

    if (!isFind) {
        cout << "NO" << endl;
    }

    return 0;
}
```

```
67
```

时空复杂度

时间复杂度：`O(NM3^L)`。其中`L`为单词`word`的长度，这是一个比较宽松的上界，回溯过程中每一个点都最多有三个分支可以进入。

空间复杂度：`O(NM)`。`check_list`所占空间。