79. Word Search

Jan. 26, 2020 | 42.6K views

Average Rating: 4.88 (51 votes)

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example:

```
board =
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]

Given word = "ABCCED", return true.
Given word = "SEE", return true.
Given word = "ABCB", return false.
```

Constraints:

- `board` and `word` consists only of lowercase and uppercase English letters.
- `1 <= board.length <= 200`
- `1 <= board[i].length <= 200`
- `1 <= word.length <= 10^3`

Solution

Approach 1: Backtracking

Intuition

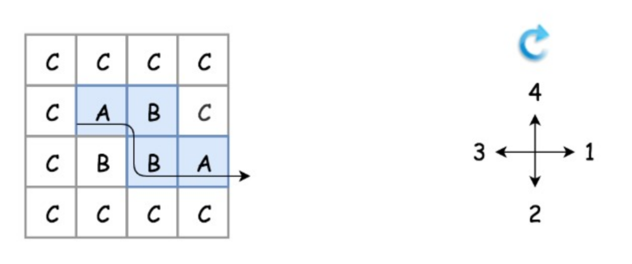
This problem is yet another 2D grid traversal problem, which is similar with another problem called 489. Robot Room Cleaner.

Many people in the discussion forum claimed that the solution is of ***DFS*** (Depth-First Search). Although it is true that we would explore the 2D grid with the DFS strategy for this problem, it does not capture the entire nature of the solution.

> We argue that a more accurate term to summarize the solution would be ***backtracking***, which is a methodology where we mark the current path of exploration, if the path does not lead to a solution, we then revert the change (*i.e.* backtracking) and try another path.

As the general idea for the solution, we would walk around the 2D grid, at each step we *mark* our choice before jumping into the next step. And at the end of each step, we would also revert our marking, so that we could have a *clean slate* to try another *direction*. In addition, the exploration is done via the *DFS* strategy, where we go as further as possible before we try the next direction.

Searching for word "ABBA"



Algorithm

There is a certain code pattern for all the algorithms of backtracking. For example, one can find one template in our Explore card of Recursion II.

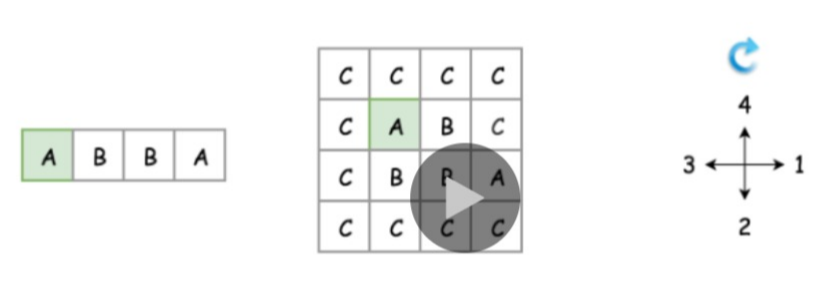
The skeleton of the algorithm is a loop that iterates through each cell in the grid. For each cell, we invoke the *backtracking* function (*i.e.* `backtrack()`) to check if we would obtain a solution, starting from this very cell.

For the backtracking function `backtrack(row, col, suffix)`, as a DFS algorithm, it is often implemented as a *recursive* function. The function can be broke down into the following four steps:

- Step 1). At the beginning, first we check if we reach the bottom case of the recursion, where the word to be matched is empty, *i.e.* we have already found the match for each prefix of the word.
- Step 2). We then check if the current state is invalid, either the position of the cell is out of the boundary of the board or the letter in the current cell does not match with the first letter of the word.
- Step 3). If the current step is valid, we then start the exploration of backtracking with the strategy of DFS. First, we mark the current cell as *visited*, *e.g.* any non-alphabetic letter will do. Then we iterate through the four possible directions, namely *up*, *right*, *down* and *left*. The order of the directions can be altered, to one's preference.
- Step 4). At the end of the exploration, we revert the cell back to its original state. Finally we return the result of the exploration.

We demonstrate how it works with an example in the following animation.

Searching for word "ABBA": DFS



1. Find the first matching letter: A

1 / 6

Java | Python | Copy

```python
class Solution(object):
    def exist(self, board, word):
        """
        :type board: List[List[str]]
        :type word: str
        :rtype: bool
        """
        self.ROWS = len(board)
        self.COLS = len(board[0])
        self.board = board

        for row in range(self.ROWS):
            for col in range(self.COLS):
                if self.backtrack(row, col, word):
                    return True

        # no match found after all exploration
        return False


    def backtrack(self, row, col, suffix):
        # bottom case: we find match for each letter in the word
        if len(suffix) == 0:
            return True

        # Check the current status, before jumping into backtracking
        if row < 0 or row == self.ROWS or col < 0 or col == self.COLS \
```

Notes

There are a few choices that we made for our backtracking algorithm, here we elaborate some thoughts that are behind those choices.

> Instead of returning directly once we find a match, we simply *break* out of the loop and do the cleanup before returning.

Here is what the alternative solution might look like.

Java | Python | Copy

```python

    def backtrack(self, row, col, suffix):
        """
            backtracking with side-effect,
              the matched letter in the board would be marked with "#".
        """
        # bottom case: we find match for each letter in the word
        if len(suffix) == 0:
            return True

        # Check the current status, before jumping into backtracking
        if row < 0 or row == self.ROWS or col < 0 or col == self.COLS \
                or self.board[row][col] != suffix[0]:
            return False

        # mark the choice before exploring further.
        self.board[row][col] = '#'
        # explore the 4 neighbor directions
        for rowOffset, colOffset in [(0, 1), (-1, 0), (0, -1), (1, 0)]:
            # sudden-death return, no cleanup.
            if self.backtrack(row + rowOffset, col + colOffset, suffix[1:]):
                return True

        # revert the marking
        self.board[row][col] = suffix[0]

        # Tried all directions, and did not find any match
```

As once notices, we simply `return True` if the result of recursive call to `backtrack()` is positive. Though this minor modification would have no impact on the time or space complexity, it would however leave with some "side-effect", *i.e.* the matched letters in the original board would be altered to `#`.

> Instead of doing the boundary checks before the recursive call on the `backtrack()` function, we do it within the function.

This is an important choice though. Doing the boundary check within the function would allow us to reach the bottom case, for the test case where the board contains only a single cell, since either of neighbor indices would not be valid.

Complexity Analysis

- Time Complexity: $\mathcal{O}(N \cdot 4^L)$ where N is the number of cells in the board and L is the length of the word to be matched.
  - For the backtracking function, its execution trace would be visualized as a 4-ary tree, each of the branches represent a potential exploration in the corresponding direction. Therefore, in the worst case, the total number of invocation would be the number of nodes in a full 4-nary tree, which is about 4^L.
  - We iterate through the board for backtracking, *i.e.* there could be N times invocation for the backtracking function in the worst case.
  - As a result, overall the time complexity of the algorithm would be $\mathcal{O}(N \cdot 4^L)$.
- Space Complexity: $\mathcal{O}(L)$ where L is the length of the word to be matched.
  - The main consumption of the memory lies in the recursion call of the backtracking function. The maximum length of the call stack would be the length of the word. Therefore, the space complexity of the algorithm is $\mathcal{O}(L)$.

Rate this article:

Previous Next

Comments: 18

Sort By

Type comment here... (Markdown is supported)

Preview Post

NeosDeus 268 February 26, 2020 11:35 PM

The quality of these articles are getting better and better. Very well written!

43 Share Reply

SHOW 1 REPLY

Haomin0817 30 February 4, 2020 8:38 AM

I hope this is more understandable for some people since I am a beginner.

```
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        def dfs(word, i, j, match_idx, directions, board, visited):
```

Read More

11 Share Reply

SHOW 2 REPLIES

yozzangga 9 March 16, 2020 12:43 PM

I think the overall time complexity is closer to O(N*3^min(L, N)).
Each cell has only 3 directions to be potentially explored because one direction has been already visited by its parent.
So, the worst case can be expressed by N * 4 * 3^min(L - 1, N - 1) (4 means the beginning point) and by big O,

Read More

9 Share Reply

SHOW 1 REPLY

jordan34 265 March 13, 2020 7:04 AM

One thing that got me is the early return `ret`. I got a TLE due to not have the ret block.

Something like this doesn't work, since we'll still recurse all those different options, even when we have already found a valid path.

Read More

6 Share Reply

SHOW 2 REPLIES

CoolWarM 52 February 4, 2020 8:37 AM

Time complexity is not thorough. If L is super long, 4^L will be bounded by N. So better expressed as O(N*(4^L+N)), plus means whichever larger.

5 Share Reply

SHOW 3 REPLIES

papaaced 9 February 2, 2020 4:43 AM

Made quick video explaining solution with code: https://youtu.be/gEXBOPnSBfk

Read More

3 Share Reply

user1484j 1 February 19, 2020 11:01 AM

a BFS based solution might be easier to understand.

1 Share Reply

SHOW 2 REPLIES

dingli 19 February 1, 2020 5:14 PM

Should be

```
self.board[row][col] != suffix[0]:
```

Read More

1 Share Reply

SHOW 1 REPLY

Iwillcrackit 6 July 3, 2020 9:14 AM

Java Solution:

```
class Solution {
    public boolean exist(char[][] board, String word) {
```

Read More

0 Share Reply

ats13 41 July 1, 2020 8:53 PM

DFS solution would have been more direct!

0 Share Reply

< 1 2 >