2018. Check if Word Can Be Placed In Crossword

Medium Array Enumeration Matrix Leetcode Link

Problem Description

This problem involves a matrix representing a crossword puzzle, which includes lowercase English letters, spaces (represented by `' '`) for empty cells, and the `'#'` character for blocked cells. The goal is to determine if a given word can be placed in the puzzle following certain conditions. The word can be placed either horizontally or vertically and must adhere to the following rules:

1. The word cannot be placed in cells that contain the `'#'` character (blocked cells).
2. Each letter of the word must either fill an empty cell (designated by spaces `' '`) or match an existing letter on the board.
3. If the word is placed horizontally, there should not be any empty cells or other letters immediately to the left or right of the word.
4. If the word is placed vertically, there should not be any empty cells or other letters immediately above or below the word.

The task is to return `true` if the word can be placed on the board according to the rules, or `false` otherwise.

Intuition

To decide whether the given word can be positioned within the board, we should check each cell of the matrix where the first letter of the word could potentially be placed. This checking has to take into account the orientation (horizontal and vertical) and also the direction (from left to right, right to left, top to bottom, and bottom to top).

The intuition behind the provided solution is to systematically iterate over every cell in the matrix and try to match the word considering all possible starting positions and directions that adhere to the crossword rules. For every potential starting position, we examine whether the word would fit without violating any constraints such as running into blocked cells or mismatching existing letters. This involves:

- Checking horizontally to the right (left_to_right) and to the left (right_to_left).
- Checking vertically downwards (up_to_down) and upwards (down_to_up).

If any of these checks succeed, indicating that the word fits without issue, the function will return `true`. If no such fitting place is found across the entire board, the function will return `false`.

The solution efficiently prunes the search by ensuring the word's placement does not start or end next to an empty cell or a different letter when placing words horizontally or vertically. As such, it starts the placement from the border of the board or next to a blocked cell and checks if every letter of the word can be placed in a suitable position.

Solution Approach

The solution uses a nested loop to iterate through every cell in the `board`. For each cell, it checks if this cell could be a potential starting point for the word by following these steps:

1. It checks if the current cell and its immediate neighbor in the opposite direction of the check are either on the edge of the board or blocked by a `'#'`. This ensures that we only start at valid positions according to the rules of the puzzle.
2. If the starting position is valid, it then invokes the `check` function which will attempt to place the word starting from that position, moving either horizontally or vertically and either forwards or backwards depending on the check being performed.

The `check` function is designed to validate the placement of the word by iterating over each letter of the word and checking the following conditions:

- The current position is within the bounds of the board.
- The current board cell is not blocked (i.e., not `'#'`).
- The current board cell is either empty (i.e., `' '`) or matches the corresponding letter in the word.

The `check` function also ensures that the letter after the last one of the word (calculated by `x, y = i + a * k, j + b * k`) is either out of bounds or blocked. This ensures that the word does not end next to a cell that could violate the horizontal or vertical placement rules.

The data structures used in the solution are:

- The `board` matrix which stores the characters as a 2D list.
- Variables `m` and `n` which represent the number of rows and columns in the `board`, respectively.
- Variable `k` which is the length of the `word`.

The solution approach does not use any additional complex algorithms or patterns. It simply leverages careful iteration and checking of board states to determine if the word can be placed.

Here's a code snippet encapsulating that logic:

```python
for i in range(m):
    for j in range(n):
        left_to_right = (j == 0 or board[i][j - 1] == '#') and check(i, j, 0, 1)
        right_to_left = (j == n - 1 or board[i][j + 1] == '#') and check(i, j, 0, -1)
        up_to_down = (i == 0 or board[i - 1][j] == '#') and check(i, j, 1, 0)
        down_to_up = (i == m - 1 or board[i + 1][j] == '#') and check(i, j, -1, 0)
        if left_to_right or right_to_left or up_to_down or down_to_up:
            return True
return False
```

The `and` operator in `left_to_right`, `right_to_left`, `up_to_down`, and `down_to_up` checks combines the start position validation and the `check` function call. If any of these conditions return `True`, it means the word can be placed in the board following the puzzle rules.

Example Walkthrough

Consider a `3 x 3` crossword puzzle board and the word "cat":

```
board = [
    ['#', 'c', '#'],
    [' ', 'a', ' '],
    ['#', 't', '#']
]
```

We want to check if we can place the word "cat" on this board.

Using our solution approach, we will check each cell starting from `(0,0)` to `(2,2)` to find a valid placement for "cat".

First, we check horizontally to the right (`left_to_right`). Starting from `(0,0)` we find it's a blocked cell (`'#'`), so we move to `(0,1)`. The cell `(0,1)` contains the first letter of the word "cat", `'c'`, and cell `(0,0)` is blocked, which is a valid start position. However, we cannot place "cat" horizontally to the right because there is no space to fit the entire word "cat".

Next, we check horizontally to the left (`right_to_left`). This direction is not applicable in this case as we are looking for starting positions.

Then, we check vertically downwards (`up_to_down`). From `(0,1)`, we realize `(0,0)` is blocked, providing a potential starting position for "cat". We can successfully match 'c' with 'c', then move to the next cell `(1,1)` and match 'a' with 'a', and finally move to `(2,1)` and see that 't' can match 't'. Thus, "cat" can be placed vertically from `(0,1)` to `(2,1)`.

Since we found a valid placement for "cat", we do not need to check vertically upwards (`down_to_up`) from `(0,1)`. The check function would return true, and the algorithm would confirm that "cat" can be placed on the board.

So in our algorithm, as soon as it runs the `up_to_down` check starting at `(0,1)`, it will return `true`, indicating that the word "cat" can indeed be placed on the board vertically. The result is obtained without having to check other cells or directions, as the solution is efficiently designed to stop once a valid position is found.

Python Solution

```python
from typing import List

class Solution:
    def place_word_in_crossword(self, board: List[List[str]], word: str) -> bool:
        # Function to check if the word fits starting from position (i, j) in the direction specified by (delta_i, delta_j)
        def is_valid_placement(i, j, delta_i, delta_j):
            # Move to the end of the word in the specified direction to check if it is within bounds or blocked by '#'
            end_i, end_j = i + delta_i * word_length, j + delta_j * word_length
            if not (0 <= end_i < rows and 0 <= end_j < cols) or (board[end_i][end_j] == '#'):
                return False

            # Iterate through each character of the 'word' to check for a valid placement
            for char in word:
                # Check for out of bounds or if the current board cell is blocked or does not match the word character
                if (
                    i < 0 or i >= rows or
                    j < 0 or j >= cols or
                    (board[i][j] != ' ' and board[i][j] != char)
                ):
                    return False

                # Move to the next cell in the specified direction
                i, j = i + delta_i, j + delta_j

            return True

        rows, cols = len(board), len(board[0])  # Get the dimensions of the board
        word_length = len(word)  # Get the length of the word

        # Iterate over every cell in the board
        for i in range(rows):
            for j in range(cols):
                # Check all four directions from the current cell: left to right, right to left, top to bottom, bottom to top
                left_to_right = (j == 0 or board[i][j - 1] == '#') and is_valid_placement(i, j, 0, 1)
                right_to_left = (j == cols - 1 or board[i][j + 1] == '#') and is_valid_placement(i, j, 0, -1)
                top_to_bottom = (i == 0 or board[i - 1][j] == '#') and is_valid_placement(i, j, 1, 0)
                bottom_to_top = (i == rows - 1 or board[i + 1][j] == '#') and is_valid_placement(i, j, -1, 0)

                # If the word can be placed in any direction, return True
                if left_to_right or right_to_left or top_to_bottom or bottom_to_top:
                    return True

        # If no valid placement is found, return False
        return False

# Example usage:
# solution = Solution()
# result = solution.place_word_in_crossword(board=[['#',' ','#'],[' ',' ','#'],['#','c',' ']], word="abc")
# print(result)  # Output will be True or False based on if the word can be placed on the board
```

Java Solution

```java
class Solution {
    private int rows;
    private int cols;
    private char[][] board;
    private String word;
    private int wordLength;

    // Method to check if the word can be placed in the crossword
    public boolean placeWordInCrossword(char[][] board, String word) {
        rows = board.length;
        cols = board[0].length;
        this.board = board;
        this.word = word;
        wordLength = word.length();

        // Traverse the board to check every potential starting point
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // Check four possible directions from the current cell
                // Left to right
                boolean leftToRight = (j == 0 || board[i][j - 1] == '#') && canPlaceWord(i, j, 0, 1);
                // Right to left
                boolean rightToLeft = (j == cols - 1 || board[i][j + 1] == '#') && canPlaceWord(i, j, 0, -1);
                // Up to down
                boolean upToDown = (i == 0 || board[i - 1][j] == '#') && canPlaceWord(i, j, 1, 0);
                // Down to up
                boolean downToUp = (i == rows - 1 || board[i + 1][j] == '#') && canPlaceWord(i, j, -1, 0);

                // If any direction is possible, return true
                if (leftToRight || rightToLeft || upToDown || downToUp) {
                    return true;
                }
            }
        }
        // If no direction is possible, return false
        return false;
    }

    // Helper method to check if the word can be placed starting from (i, j) in the specified direction (a, b)
    private boolean canPlaceWord(int i, int j, int rowIncrement, int colIncrement) {
        int endRow = i + rowIncrement * wordLength;
        int endCol = j + colIncrement * wordLength;

        // Check if the word goes out of bounds or is not terminated properly
        if (endRow < 0 || endRow >= rows || endCol < 0 || endCol >= cols || board[endRow][endCol] != '#') {
            return false;
        }

        // Check each character to see if the word fits
        for (int p = 0; p < wordLength; ++p) {
            if (i < 0 || i >= rows || j < 0 || j >= cols
                || (board[i][j] != ' ' && board[i][j] != word.charAt(p))) {
                return false;
            }
            i += rowIncrement;
            j += colIncrement;
        }
        return true;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to determine if a word can be placed in a crossword
    bool placeWordInCrossword(vector<vector<char>>& board, string word) {
        int numRows = board.size(), numCols = board[0].size(); // board dimensions
        int wordLen = word.size(); // length of the word to be placed

        // Lambda function to check if the word fits in the given direction
        auto check = [&](int row, int col, int deltaRow, int deltaCol) {
            int endRow = row + deltaRow * wordLen, endCol = col + deltaCol * wordLen;
            // Check if the end position is not blocked by '#'
            if (endRow >= 0 && endRow < numRows && endCol >= 0 && endCol < numCols && board[endRow][endCol] != '#') {
                return false;
            }
            // Iterate over each character in the word
            for (char& c : word) {
                // Check boundaries and match the character with the board or wildcard
                if (row < 0 || row >= numRows || col < 0 || col >= numCols || (board[row][col] != ' ' && board[row][col] != c)) {
                    return false;
                }
                row += deltaRow;
                col += deltaCol;
            }
            return true;
        };

        // Iterate over each cell in the board
        for (int i = 0; i < numRows; ++i) {
            for (int j = 0; j < numCols; ++j) {
                // Check four possible directions where the word can be placed
                bool leftToRight = (j == 0 || board[i][j - 1] == '#') && check(i, j, 0, 1);
                bool rightToLeft = (j == numCols - 1 || board[i][j + 1] == '#') && check(i, j, 0, -1);
                bool upToDown = (i == 0 || board[i - 1][j] == '#') && check(i, j, 1, 0);
                bool downToUp = (i == numRows - 1 || board[i + 1][j] == '#') && check(i, j, -1, 0);

                // If the word can be placed in any direction, return true
                if (leftToRight || rightToLeft || upToDown || downToUp) {
                    return true;
                }
            }
        }
        // Return false if the word can't be placed on the board in any direction
        return false;
    }
};
```

Typescript Solution

```typescript
// Function to check if a word can be placed in a crossword
function placeWordInCrossword(board: char[][], word: string): boolean {
    const numRows = board.length; // Number of rows in the board
    const numCols = board[0].length; // Number of columns in the board
    const wordLen = word.length; // Length of the word to be placed

    // Helper function to check if the word fits in the given direction
    const check = (row: number, col: number, deltaRow: number, deltaCol: number): boolean => {
        const endRow = row + deltaRow * wordLen;
        const endCol = col + deltaCol * wordLen;
        // Check if the end position is outside the boundary or blocked by '#'
        if (endRow >= 0 && endRow <= numRows && endCol >= 0 && endCol <= numCols && (board[endRow] === undefined || board[endRow][e
            // Iterate over each character in the word
            for (let index = 0; index < wordLen; ++index) {
                const char = word[index];
                // Check boundaries and match the character with the board or wildcard
                if (row < 0 || row >= numRows || col < 0 || col >= numCols || (board[row][col] !== ' ' && board[row][col] !== char)
                    return false;
                }
                row += deltaRow;
                col += deltaCol;
            }
            return true;
        }
        return false;
    };

    // Iterate over each cell in the board
    for (let i = 0; i < numRows; ++i) {
        for (let j = 0; j < numCols; ++j) {
            // Check four possible directions where the word can be placed
            const leftToRight = (j === 0 || board[i][j - 1] === '#') && check(i, j, 0, 1);
            const rightToLeft = (j === numCols - 1 || board[i][j + 1] === '#') && check(i, j, 0, -1);
            const upToDown = (i === 0 || board[i - 1][undefined] === '#' || board[i - 1][j] === '#') && check(i, j, 1, 0);
            const downToUp = (i === numRows - 1 || board[i + 1] === undefined || board[i + 1][j] === '#') && check(i, j, -1, 0);

            // If the word can be placed in any direction, return true
            if (leftToRight || rightToLeft || upToDown || downToUp) {
                return true;
            }
        }
    }
    // Return false if the word can't be placed on the board in any direction
    return false;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(m * n * k)`, where `m` is the number of rows in the board, `n` is the number of columns in the board, and `k` is the length of the word to be placed. This complexity arises because the code iterates over all cells of the board (`m * n`) and for each cell, it attempts to place the word in all four directions. The check function, which is called for each direction, runs a loop up to the length of the word (`k`).

Additionally, when evaluating the board for possible placements, the algorithm checks the perpendicular cells to ensure placement is at the beginning or end of a word sequence (which is a constant time check). Due to these operations being constant in time, they do not impact the linear relationship between the time complexity and the number of cells times the length of the word.

Space Complexity

The space complexity of the code is `O(1)` (constant space complexity). This is because the algorithm only uses a fixed amount of extra space for variables that store the dimensions of the board and indices during the checks regardless of the input size. No additional space proportional to the input size is required beyond what is used to store the board and word, which are inputs and not counted towards the space complexity.