Word Search Problem

Algorithm Design

CSX 3009 Algorithm Design - Section 541 - Term Project

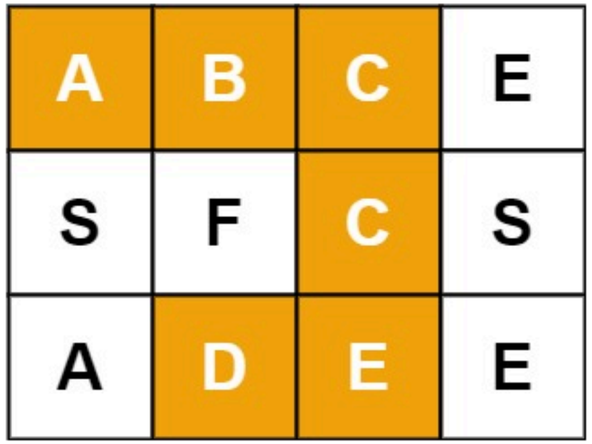
6632108 - Khant Nyi Thu

Introduction

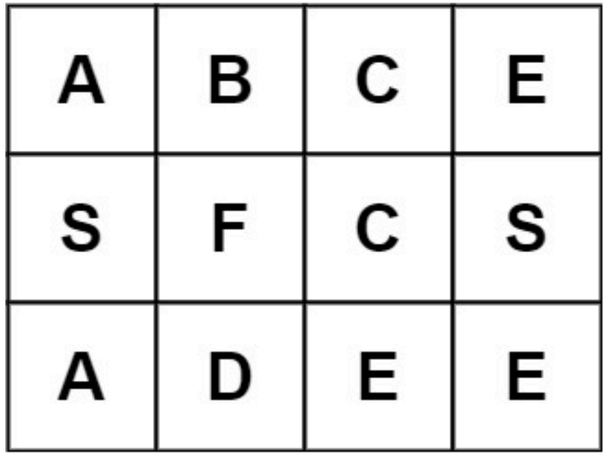
- Given an ***m x n*** grid of characters board and a string word, return true if **word** exists in the grid.

- You can move up, down, left, right one step at a time, cannot reuse a cell in one path, and must decide whether the word can be formed by a path through the grid.

Example 1:




```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
Output: true
```

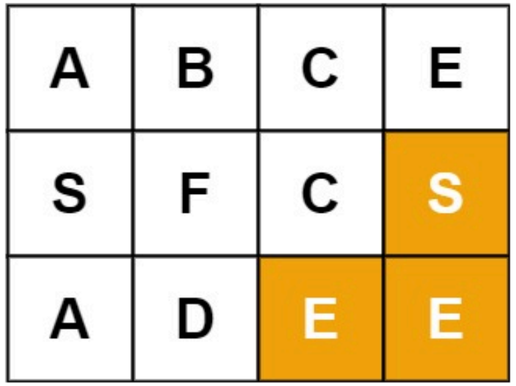



```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"
Output: false
```

Constraints

- `m == board.length`
- `n = board[i].length`
- `1 <= m, n <= 6`
- `1 <= word.length <= 15`
- `board` and `word` consists of only lowercase and uppercase English letters.

Example 2:




```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"
Output: true
```

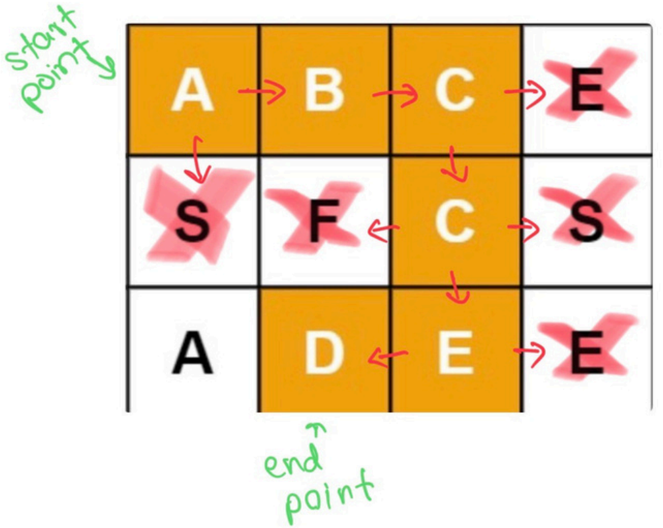
Approaching the problem logically

1. **Start point**:
   - Look for places in the grid where the word's first letter appears.

2. Exploring paths:
   - From each starting point, try moving step by step to see if you can follow the word's letters in order.

3. Restrictions:
   - You can't step on the same box twice for the same word attempt, so if you already used a letter, you must backtrack (go back and try another route).

4. Stopping early:
   - If at any point the letter doesn't match, stop going down that path and try a different one.



word = "ABCCED"

Approaches We Consider

For this Word Search problem, we are going to consider 2 approaches: **Bread First Search (BFS)** and **Depth First Search (DFS)**

In **BFS**, we treat the grid as a graph where each cell is a node. We explore this graph level by level, where a "level" corresponds to the length of the partial word we have formed.

In **DFS**, we treat the grid as a graph where each cell is a node. We explore this graph by diving deeply into one path at a time, backtracking when a path fails, to find a sequence of cells that forms the given word.

BFS

BFS

column

row	0	1	2	3
0	F	U	R	E
1	E	N	C	A
2	X	C	O	C
3	T	E	D	T

word to find

OCRE

1. Start a queue to hold the paths.

2. Find starting word O

   found at (2, 2)

3. Populate the queue

   Q = {[(2,2)]}

4. Dequeue the Path and continue

5. Look for next word C in neighbors

   found C at (1,2), (2,1) and (2,3)

6. Enqueue new paths

   Q_{new} = {[(2,2), (1,2)], [(2,2), (2,1)], [(2,2), (2,3)]}

7. Update queue Q = Q_{new}

8. Loop through Q to find next word R

   Path 1 = [(2,2), (1,2)]

   found at (0, 2)

   Path 2 = [(2,2), (2,1)] Not found

   Path 3 = [(2,2), (2,3)] Not found

9. Update Q = {[(2,2), (1,2), (0,2)]} (O, C, R)

10. Loop through Q to find next word E

    Path = {[(2,2), (1,2), (0,2)]}

    found E at (0,3)

11. Update Q = {[(2,2), (1,2) (0,2), (0,3)]}

    Found OCRE

BFS Code

```python
from collections import Counter
from collections import deque

def exist(board: list[list[str]], word: str) -> bool:
    """
    Solves the Word Search problem using Breadth-First Search (BFS)
    """
    if not board:
        return False

    rows, cols = len(board), len(board[0])

    # --- Pruning: Frequency pre-check ---
    need = Counter(word)
    have = Counter(ch for row in board for ch in row)
    if any(have[ch] < cnt for ch, cnt in need.items()):
        return False

    # Handle single-letter words
    if len(word) == 1:
        for r in range(rows):
            for c in range(cols):
                if board[r][c] == word[0]:
                    return True
        return False

    def bfs(start_row: int, start_col: int) -> bool:
        # Queue stores: (row, col, index, path)
        queue = deque([(start_row, start_col, 0, {(start_row, start_col)})])

        # Define the four possible directions (up, down, left, right)
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

        while queue:
            row, col, index, path = queue.popleft()

            # Base case: we've matched the entire word
            if index == len(word) - 1:
                return True

            # Explore neighbors for the next letter in the word
            for dr, dc in directions:
                next_row, next_col = row + dr, col + dc

                # Check if the neighbor is valid
                if 0 <= next_row < rows and 0 <= next_col < cols:
                    # Check if the neighbor has not been visited and matches the next character
                    if (next_row, next_col) not in path and board[next_row][next_col] == word[index + 1]:
                        # Create a new path set for this branch
                        new_path = path | {(next_row, next_col)}
                        queue.append((next_row, next_col, index + 1, new_path))

        return False

    # Try starting from each cell that matches the first letter of the word
    for r in range(rows):
        for c in range(cols):
            if board[r][c] == word[0]:
                if bfs(r, c):
                    return True

    return False


# Test cases
board1 = [["F","U","R","E"],
          ["E","N","C","A"],
          ["X","C","O","C"],
          ["T","E","D","T"]]
word1 = "OCRE"
print(f"Does '{word1}' exist? {exist(board1, word1)}")  # Expected: True

board2 = [
    ["S", "P", "R", "I", "N", "G"],
    ["T", "I", "M", "E", "F", "L"],
    ["O", "W", "E", "R", "S", "B"],
    ["L", "O", "O", "M", "I", "N"],
    ["G", "A", "R", "D", "E", "N"],
    ["P", "A", "T", "H", "W", "A"],
    ["Y", "S", "U", "N", "N", "Y"],
    ["D", "A", "Y", "S", "K", "I"]
]
word2 = "PATHNSK"
print(f"Does '{word2}' exist? {exist(board2, word2)}")  # Expected: True

board3 = [
    ["T", "H", "E", "Q", "U", "I"],
    ["C", "K", "B", "R", "O", "X"],
    ["F", "O", "X", "J", "U", "M"],
    ["P", "S", "O", "V", "E", "R"],
    ["L", "A", "Z", "Y", "D", "O"],
    ["G", "W", "A", "V", "E", "S"]
]
word3 = "SOZYE"
print(f"Does '{word3}' exist? {exist(board3, word3)}")  # Expected: False
```

BFS Analysis

In BFS, we treat the grid as a graph where each cell is a node. We explore this graph level by level, where a "level" corresponds to the length of the partial word we have formed.

Characteristics:

- Uses a queue data structure
- Explores neighbors layer by layer
- Requires tracking the entire path to avoid revisiting cells
- Memory intensive due to storing multiple states

Time Complexity: O(M × N × 4^L) where M×N is board size and L is word length
Space Complexity: O(M × N × L) for the queue and path tracking

DFS

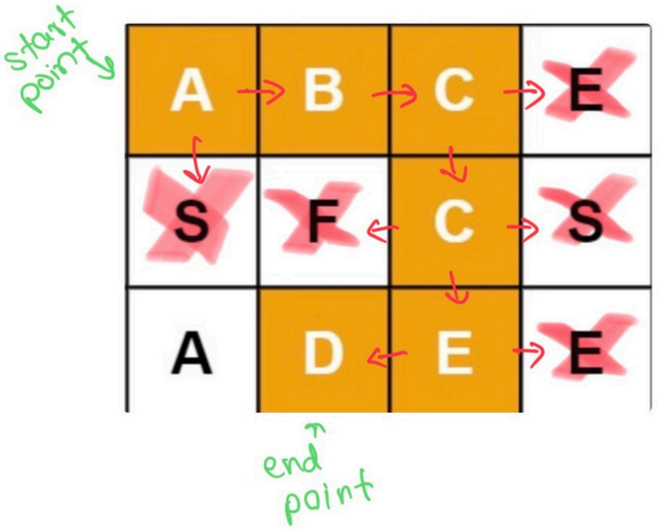
From every starting letter, check every possible path until you either find the word or exhaust all options.

Backtracking

- If the path doesn't work out, undo the step (mark the cell as unvisited again). Go back to the previous position and try another direction.

Pruning

- Letter mismatch: If the next cell's letter doesn't match the word's next letter → stop immediately.
- Revisited cell: If we already used this cell in the current path → don't use it again.
- Frequency check: If the board doesn't have enough of some letters (like 3 "A"s in the word but only 2 on the board) → say "no" before starting.



word = "ABCCED"

Backtracking

Example 3:

A	B	C	E
S	F	C	S
A	D	E	E

```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"
Output: false
```

Not enough "B" in the board

DFS Code

```python
from collections import Counter

def exist(board: list[list[str]], word: str) -> bool:
    """
    Solves the Word Search problem using Depth-First Search (DFS) with frequency pruning.
    """
    if not board:
        return False

    rows, cols = len(board), len(board[0])

    # --- Pruning: Frequency pre-check ---
    need = Counter(word)
    have = Counter(ch for row in board for ch in row)
    if any(have[ch] < cnt for ch, cnt in need.items()):
        return False

    def dfs(row: int, col: int, index: int, path: set) -> bool:
        # Base case: we've matched the entire word
        if index == len(word) - 1:
            return True

        # Define the four possible directions (up, down, left, right)
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

        # Explore neighbors for the next letter in the word
        for dr, dc in directions:
            next_row, next_col = row + dr, col + dc

            # Check if the neighbor is valid
            if 0 <= next_row < rows and 0 <= next_col < cols:
                # Check if the neighbor has not been visited in the current path
                if (next_row, next_col) not in path:
                    # Check if the neighbor's character matches the next character in the word
                    if board[next_row][next_col] == word[index + 1]:
                        # Add the new cell to the path
                        path.add((next_row, next_col))

                        # Recursively search from this new position
                        if dfs(next_row, next_col, index + 1, path):
                            return True

                        # Backtrack: remove the cell from the path
                        path.remove((next_row, next_col))

        return False

    # Try starting from each cell that matches the first letter of the word
    for r in range(rows):
        for c in range(cols):
            if board[r][c] == word[0]:
                # If the word is just one letter, we found it
                if len(word) == 1:
                    return True

                # Start DFS from this cell
                if dfs(r, c, 0, {(r, c)}):
                    return True

    return False


# Test cases
board1 = [["F","U","R","E"],
          ["E","N","C","A"],
          ["X","C","O","C"],
          ["T","E","D","T"]]
word1 = "OCRE"
print(f"Does '{word1}' exist? {exist(board1, word1)}")  # Expected: True

board2 = [
    ["S", "P", "R", "I", "N", "G"],
    ["T", "I", "M", "E", "F", "L"],
    ["O", "W", "E", "R", "S", "B"],
    ["L", "O", "O", "M", "I", "N"],
    ["G", "A", "R", "D", "E", "N"],
    ["P", "A", "T", "H", "W", "A"],
    ["Y", "S", "U", "N", "N", "Y"],
    ["D", "A", "Y", "S", "K", "I"]
]
word2 = "PATHNSK"
print(f"Does '{word2}' exist? {exist(board2, word2)}")  # Expected: True

board3 = [
    ["T", "H", "E", "Q", "U", "I"],
    ["C", "K", "B", "R", "O", "X"],
    ["F", "O", "X", "J", "U", "M"],
    ["P", "S", "O", "V", "E", "R"],
    ["L", "A", "Z", "Y", "D", "O"],
    ["G", "W", "A", "V", "E", "S"]
]
word3 = "SOZYE"
print(f"Does '{word3}' exist? {exist(board3, word3)}")  # Expected: False

# Additional test case to demonstrate pruning
board4 = [["A", "B"], ["C", "D"]]
word4 = "AAA"
print(f"Does '{word4}' exist? {exist(board4, word4)}")  # Expected: False (pruning kicks in)
```

DFS Analysis

In DFS, we treat the grid as a graph where each cell is a node. We explore this graph by diving deeply into one path at a time, backtracking when a path fails, to find a sequence of cells that forms the given word.

Characteristics:

- Uses a recursive call stack data structure.
- Explores one path to its completion before backtracking to try alternative paths.
- Requires tracking the current path to avoid revisiting cells within that path.
- Memory efficient as it only stores the current path and recursion stack.

Time Complexity: O(M × N × 4^L) where M×N is board size and L is word length
Space Complexity: O(L) for the recursion and path

BFS and DFS Comparison Table

Aspect	BFS	DFS
Exploration Strategy	Level-by-level	Depth-first (one path to completion, then backtrack)
Data Structure	deque for queue	Recursive call stack
Time Complexity	O(M * N * 4^L)	O(M * N * 4^L)
Space Complexity	O(M * N * L) (queue + path sets)	O(L) (recursion stack + single path set)
Memory Usage	High (stores many states per level)	Low (stores only current path)
Path Tracking	Each queue state has its own path set	Single path set modified and restored
Single-Letter Words	O(M * N), avoids BFS	O(M * N), avoids DFS
Practical Efficiency	Slower due to queue management and copying sets	Faster due to simpler recursion and less memory overhead
Use Case	Guarantees shortest path (in steps), but not needed here	Preferred for memory efficiency and simplicity

BFS and DFS Complexity Comparison

Aspect	BFS	DFS
Best Case Time	O(M * N + L) (pre-check triggers or L=1)	O(M * N + L) (pre-check or early success)
Worst Case Time	O(M * N * 4^L)	O(M * N * 4^L)
Average Case Time	O(M * N * k^L), k < 4	O(M * N * k^L), k < 4
Best Case Space	O(L + M * N) (pre-check only)	O(L) (early success or pre-check)
Worst Case Space	O(M * N * L) (queue many states + each with a path size of O(L))	O(L) (recursion + path set)
Average Case Space	O(k * M * N)	O(L)
Practical Efficiency	Slower due to queue and set copying	Faster due to recursion and single path set
Memory Usage	High (many states in queue)	Low (single path)

M = number of rows on the board

N = numbers of columns on the board

k = number of branching factors

L = length of the word

- **Best Case** (Board doesn't contain letter or single letter word is found immediately)
- **Average Case** (Given word may or may not exist)
- **Worst Case** (Word doesn't exist but the board has many cells matching each letter, search continues)

Conclusion

DFS is superior for word search problems. The identical time complexity means both algorithms explore similar search spaces, but DFS is memory efficient and feels natural.