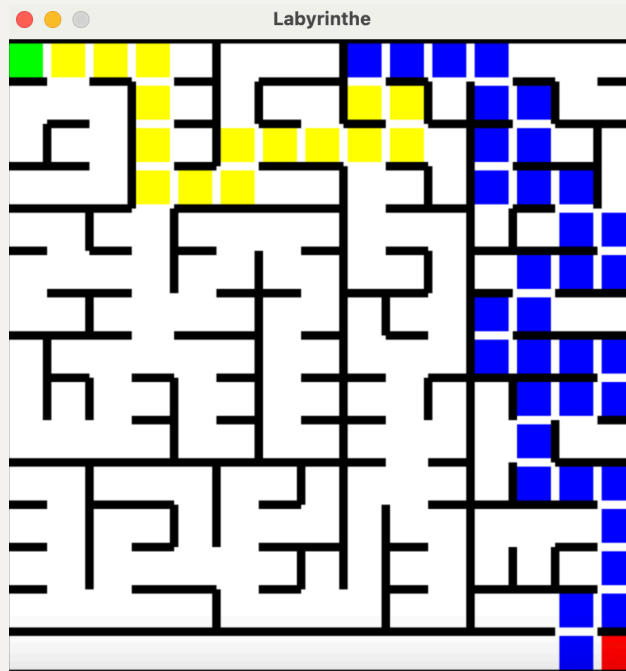

Final Report—problem:maze generation and resolution



Team Members

- Tingting Jiang
- Sara Lounis
- Anita Soltani
- Lucas Smati
- Karim Fayez Hannaallah
- Daniel Verguet

1. Data Structure Description and Specification👉

1.1 Choice of Data Structure

The data structure chosen for this project is a **graph representation of the maze** implemented using a **2D array**. Each element in the array represents a cell in the maze, which includes the following properties:

1. Cell Properties:

- **Walls:** Each cell maintains a boolean status for its four walls (`top`, `right`, `bottom`, `left`) to define boundaries.
- **Visited Status:** A flag to indicate whether the cell has been visited, which is essential for both maze generation and resolution algorithms.
- **Position:** The cell's coordinates (`x`, `y`) within the 2D grid to identify its location.

2. Graph Representation:

- **Nodes:** Each cell acts as a node.
- **Edges:** Edges between nodes are implicitly created by removing walls, forming passages that connect neighboring cells.

This structure was selected because:

- It efficiently supports **spatial representation** and **visualization** of the maze.
- It provides a straightforward way to manage walls and passages during the recursive division generation process.
- It is compatible with traversal algorithms like BFS used in the maze resolution.

Design Considerations

- **Boundary Conditions:** The maze is enclosed by outer walls with one unique exit at the southeast corner.
- **Flexibility:** The structure supports dynamic maze dimensions and allows for clear distinction between walls and paths.

1.2 Formal Specification

Data Structure Definition

To effectively implement maze generation and resolution, we defined the following core data structures:

Maze (Maze Structure)

- **Type:**

`Maze` is a structure containing a two-dimensional array representing the overall layout of the maze.

- **Attributes:**

- `rows`: Number of rows in the maze.
- `cols`: Number of columns in the maze.
- `grid`: A 2D array of `Cell` structures representing individual units in the maze.

- **Code:**

```
typedef struct Maze {  
    int rows;           // Number of rows in the maze  
    int cols;           // Number of columns in the maze  
    Cell** grid;        // 2D array of cells  
} Maze;
```

Cell (Cell Structure)

- **Type:** `Cell` is the basic unit of the maze, containing information about the cell's walls, passages, and visitation status.

- **Attributes**

- `x`, `y`: Coordinates of the cell within the maze grid.
- `walls`: A boolean array indicating the presence of walls on all four sides ([0]=North, [1]=East, [2]=South, [3]=West).
- `visited`: A boolean flag used during pathfinding or maze generation.

- **Code**

```
typedef struct Cell {
    int x, y;           // Coordinates of the cell
    bool walls[4];      // Walls: [0]=North, [1]=East,
                        // [2]=South, [3]=West
    bool visited;       // Visitation status for search
                        algorithms
} Cell;
```

Operations on Data Structures

Below are the core operations used for maze generation and resolution:

`initializeMaze` (Initialize Maze)

- **Description:** Initializes a maze object with specified dimensions. All cells are initially defined with walls intact, and boundary walls are explicitly set. Additionally, the exit is established at the southeast corner of the maze.
- **Input:**
 - `maze`: A pointer to the maze structure
 - `Maze` object to be initialized.
 - `width`: Width (number of columns) of the maze.
 - `height`: Height (number of rows) of the maze.
- **Output:** Populates the `Maze` object with dynamically allocated cells and their default states.

`setExit` (Set Exit)

- **Description:** Defines the maze's exit point at the southeast corner (bottom-right cell). This involves marking the cell as the exit.
- **Input:** `maze`: A pointer to the `Maze` object.

- **Operation:** - The exit property of the southeast corner cell (`maze->cells[width-1][height-1]`) is set to `1` to indicate it as the exit point.
- **Output:** Updates the specified cell in the maze to be recognized as the exit.

`generate_random_wall` (Generate Random Wall)

- **Description:**
Generates random walls within a specified region of the maze. The function randomly divides the region into two parts, creating a wall with a random "door" (passage) to allow movement between them. This process is done recursively for the two sub-regions.
- **Input:**
 - `maze`: A pointer to the `Maze` object.
 - `x_start`: The starting x-coordinate of the region to be divided.
 - `y_start`: The starting y-coordinate of the region to be divided.
 - `width`: The width of the region to divide.
 - `height`: The height of the region to divide.
- **Operation:**
 - The function chooses whether to split horizontally or vertically, based on whether the width is greater than the height.
 - A random split point is chosen within the specified region, and a random door is created at a random position along the split.
 - The wall is generated by marking the appropriate `right`, `left`, `bottom`, or `top` cell properties to create the wall.
 - The function then recursively generates walls for the two resulting sub-regions.
- **Output:**
 - Updates the maze's `cells` array, marking the walls and door for the specified region.

`markCompletePath` (Mark Complete Path)

- **Description:**

Marks the complete path from both the entrance (D) and exit (E) to the meeting point in the maze. This function helps to visualize the entire shortest path by marking cells along the path, ignoring doors and the exit itself.

- **Input:**

- `maze`: A pointer to the `Maze` object (used to access maze cells).
- `meetingCell`: A pointer to the `cell` where the paths from both the entrance and exit meet, marking the end of both search processes in the bidirectional search algorithm.

- **Operation:**

- The function first marks the path from the entrance (D) to the meeting cell. It iterates through the parent pointers from the entrance side and marks the cells in the `pathFromD` direction.
- Then, it marks the path from the exit (E) to the meeting cell, following the parent pointers from the exit side and marking the cells in the `pathFromE` direction.
- The function avoids marking cells that are doors or exits to ensure that only valid path cells are marked.

- **Output:**

- Updates the `pathFromD` and `pathFromE` attributes of `cell` objects, marking the cells along the shortest path from the entrance and exit.

`findPathBidirectionalBFS` (Find Bidirectional Path using BFS)

- **Description:**

This function implements the bidirectional breadth-first search (BFS) algorithm to find the shortest path between the maze's entrance and exit. It performs simultaneous BFS from both the entrance and exit, exploring paths in both directions until they meet at a common point. This reduces the search space and optimizes the pathfinding process.

- **Input:**

- `maze`: A pointer to the `Maze` object, used to access the maze cells and properties.

- `start_x`, `start_y`: The coordinates of the maze's entrance (starting point) where the search begins.
- **Operation:**
 - The algorithm initializes two queues, one for each direction (entrance and exit), and starts BFS from both ends of the maze.
 - It explores neighboring cells in four directions (up, down, left, right), and if a valid move is found (i.e., within bounds and not blocked by walls), the cell is added to the respective queue and marked as visited.
 - If a cell visited from one direction is encountered from the other direction, the pathfinding process stops, and the path is marked by calling the `markCompletePath` function.
 - The function checks if a cell visited from one direction is also visited from the other direction, indicating that the search has successfully met in the middle, thus confirming the path.
- **Output:**
 - The function returns `1` if a path is found and the path is marked, or `0` if no path exists between the entrance and exit.

`displayMaze` (Display the Maze using SDL2)

- **Description:**
This function renders the maze on the screen using SDL2. It draws the maze's structure, including walls, doors, exit, and paths found by the search algorithm, updating the display at each step.
- **Input:**
 - `maze`: A pointer to the `Maze` object, which contains the maze cells and their properties.
 - `renderer`: The SDL2 renderer used to draw on the screen.
- **Operation:**
The function iterates over each cell in the maze and draws:
 - a. **Walls:** The four walls of each cell (top, bottom, left, right) are drawn in black.

- b. **Door:** If a door exists in the cell, it is drawn in light green.
- c. **Exit:** If the cell is the exit, it is drawn in light red.
- d. **Path from Start (D):** If a cell is part of the path from the start (D), it is drawn in yellow.
- e. **Path from Exit (E):** If a cell is part of the path from the exit (E), it is drawn in blue.

Each element (wall, door, exit, path) is drawn with appropriate colors using `SDL_SetRenderDrawColor` and `SDL_RenderFillRect`.

- **Output:**
 - The function does not return anything. It directly updates the screen by rendering the maze with the current maze state.

2. Maze Generation and Solution Algorithms

2.1 Maze Generation Algorithm

- **Description**

The maze generation process aims to create a labyrinth structure that satisfies specific requirements: a single unique exit located at the southeast corner, multiple random entrances, and a well-defined path network. Below is the detailed description of the algorithm used:

- **Algorithm Steps:**

- a. **Initialization:**

Create a grid with solid walls. Define the maze boundaries and treat each cell as an independent unit initially.

- b. **Recursive Division:**

Divide the space recursively into smaller regions:

- Choose a direction (horizontal or vertical) for splitting the space.
 - Add a wall along this direction and create a random doorway to ensure connectivity.

- Repeat this process for each subregion.

c. Entrance and Exit Placement:

- Randomly select one or more trapdoors along the maze perimeter as entrances.
- Place a fixed exit at the southeast corner.

d. Termination:

Stop dividing when the subregions reach the minimum size (e.g., 1x1).

- **Pseudocode**

```
function GenerateMaze(grid, x1, y1, x2, y2):
    if (x2 - x1 <= 1) or (y2 - y1 <= 1):
        return # Base case: region too small to divide

    if random_choice(horizontal=True, vertical=False):
        # Divide horizontally
        wall_y = random_between(y1 + 1, y2 - 1)
        door_x = random_between(x1, x2)
        for x in range(x1, x2 + 1):
            if x != door_x:
                grid[wall_y][x] = WALL
        GenerateMaze(grid, x1, y1, x2, wall_y - 1)
        GenerateMaze(grid, x1, wall_y + 1, x2, y2)
    else:
        # Divide vertically
        wall_x = random_between(x1 + 1, x2 - 1)
        door_y = random_between(y1, y2)
        for y in range(y1, y2 + 1):
            if y != door_y:
                grid[y][wall_x] = WALL
        GenerateMaze(grid, x1, y1, wall_x - 1, y2)
        GenerateMaze(grid, wall_x + 1, y1, x2, y2)

# Initialize grid
grid = create_empty_grid(width, height)
GenerateMaze(grid, 0, 0, width - 1, height - 1)
place_entrance(grid)
place_exit(grid)
```

2.2 Maze Solution Algorithm

- **Description**

To solve the maze, we utilize the **Bidirectional Breadth-First Search (BFS)** for efficiency.

- **Algorithm Steps:**

- a. **Initialization:**

- Create two queues, one starting from the entrance and the other from the exit. Maintain visited sets for each side and maps for reconstructing paths.

- b. **Bidirectional Expansion:**

- Expand alternately from both the entrance and exit using BFS. Mark cells as visited to avoid redundant exploration.

- c. **Meeting Point Detection:**

- When the two searches encounter the same cell, reconstruct the path from both directions.

- d. **Reconstruction:**

- Combine the paths from the entrance and exit to form the shortest path.

- e. **Termination:**

- Stop when the meeting point is found or when one of the queues is empty (no solution).

- **Pseudocode**

```
function BidirectionalBFS(grid, start, goal):
    queue_start = [start]
    queue_goal = [goal]
    visited_start = set([start])
    visited_goal = set([goal])
    parent_start = {start: None}
    parent_goal = {goal: None}

    while queue_start and queue_goal:
```

```

        # Expand from the start
        current_start = queue_start.pop(0)
        for neighbor in get_neighbors(current_start, grid):
            if neighbor not in visited_start:
                visited_start.add(neighbor)
                parent_start[neighbor] = current_start
                queue_start.append(neighbor)
                if neighbor in visited_goal:
                    return reconstruct_path(neighbor,
parent_start, parent_goal)

        # Expand from the goal
        current_goal = queue_goal.pop(0)
        for neighbor in get_neighbors(current_goal, grid):
            if neighbor not in visited_goal:
                visited_goal.add(neighbor)
                parent_goal[neighbor] = current_goal
                queue_goal.append(neighbor)
                if neighbor in visited_start:
                    return reconstruct_path(neighbor,
parent_start, parent_goal)
        return "No path found"

function reconstruct_path(meeting_point, parent_start,
parent_goal):
    path_start = []
    current = meeting_point
    while current:
        path_start.append(current)
        current = parent_start[current]
    path_start.reverse()

    path_goal = []
    current = parent_goal[meeting_point]
    while current:
        path_goal.append(current)
        current = parent_goal[current]

    return path_start + path_goal

```

3. Theoretical Complexity Analysis of the Maze Solving Algorithm

Algorithm: Bidirectional BFS

The **Bidirectional Breadth-First Search (BFS)** algorithm aims to find the shortest path between the entrance and exit of the maze. The complexity is determined by:

1. Graph Representation:

The maze is represented as a grid of size $(M \times N)$, where each cell is either a wall or a passage. The total number of cells is $(V = M \times N)$.

Each cell can have at most 4 neighbors (up, down, left, right), so the number of edges (E) is proportional to $(4V)$, or $(O(V))$ in asymptotic terms.

Time Complexity

1. Single BFS Expansion

A single BFS explores all reachable nodes and edges in the graph. In the worst case, it visits all (V) vertices and (E) edges.

- Visiting a vertex takes $(O(1))$.
- Exploring all edges for a vertex also takes $(O(1))$ per edge.

Thus, the total time complexity of a single BFS is: $[O(V + E)]$

Since $(E \in O(V))$, this simplifies to: $[O(V)]$

2. Bidirectional BFS

In Bidirectional BFS, we perform two BFSs simultaneously, one starting from the entrance and the other from the exit.

- Each BFS explores roughly half the graph before meeting at a common vertex.
- The complexity of each BFS is $(O(V/2))$.

The combined complexity is: $[O(V/2) + O(V/2) = O(V)]$

3. Path Reconstruction

Once the meeting point is found, reconstructing the path involves backtracking through the parent maps from the meeting point to the entrance and exit.

- The path length is at most $O(V)$, but backtracking is linear in the number of steps.

Therefore, path reconstruction is: $[O(V)]$

Total Time Complexity

The total time complexity is the sum of the bidirectional BFS and path reconstruction:
 $[O(V) + O(V) = O(V)]$

Space Complexity

1. Visited Sets

Each BFS maintains a visited set to store explored nodes. The size of each set is $(O(V/2))$, and the combined space for both sets is: $[O(V)]$

2. Queues

Each BFS queue stores nodes to be explored. In the worst case, the total size of both queues is: $[O(V)]$

3. Parent Maps

The algorithm uses parent maps for path reconstruction. The size of each parent map is $(O(V/2))$, and the total space is: $[O(V)]$

Summary

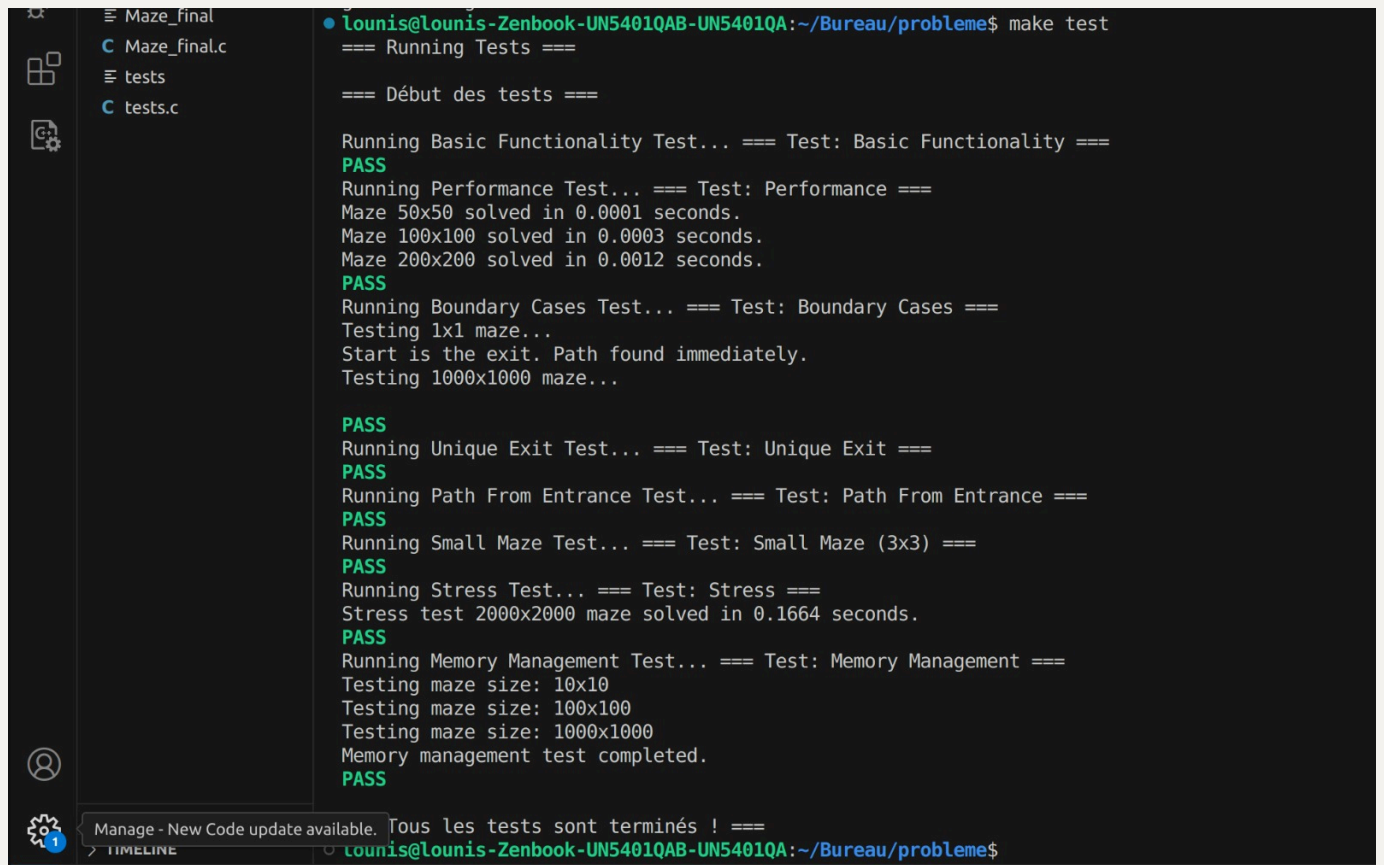
- **Time Complexity:** $[O(V)]$ (linear in the number of cells)
- **Space Complexity:** $[O(V)]$ (linear in the number of cells)

In practice, **Bidirectional BFS** is significantly faster than a single BFS in large mazes because the search space reduces exponentially as the two searches converge.

4. Performance Test Analysis

Overview

The performance tests are designed to assess the speed and efficiency of the maze-solving algorithm (Bidirectional BFS). These tests evaluate the time taken to solve mazes of different sizes, boundary cases, stress conditions, and memory management. Below are the details of the different performance tests executed.



```
lounis@lounis-Zenbook-UN5401QAB-UN5401QA:~/Bureau/ probleme$ make test
=== Running Tests ===

=== Début des tests ===

Running Basic Functionality Test... === Test: Basic Functionality ===
PASS
Running Performance Test... === Test: Performance ===
Maze 50x50 solved in 0.0001 seconds.
Maze 100x100 solved in 0.0003 seconds.
Maze 200x200 solved in 0.0012 seconds.
PASS
Running Boundary Cases Test... === Test: Boundary Cases ===
Testing 1x1 maze...
Start is the exit. Path found immediately.
Testing 1000x1000 maze...

PASS
Running Unique Exit Test... === Test: Unique Exit ===
PASS
Running Path From Entrance Test... === Test: Path From Entrance ===
PASS
Running Small Maze Test... === Test: Small Maze (3x3) ===
PASS
Running Stress Test... === Test: Stress ===
Stress test 2000x2000 maze solved in 0.1664 seconds.
PASS
Running Memory Management Test... === Test: Memory Management ===
Testing maze size: 10x10
Testing maze size: 100x100
Testing maze size: 1000x1000
Memory management test completed.
PASS

Tous les tests sont terminés ! ===
lounis@lounis-Zenbook-UN5401QAB-UN5401QA:~/Bureau/ probleme$
```

Test Cases

1. Basic Functionality Test

This test checks the basic functionality of the maze-solving algorithm. It runs the algorithm on mazes of different sizes (simple, random walls, and no solution cases). The result is marked as "PASS" if the algorithm successfully finds the path or identifies the lack of a path.

Example:

- 10x10 maze: Path is found
- 5x5 maze: Path is found
- 20x20 maze: Path is found
- 10x10 maze with no solution: No path is found

2. Performance Test

This test evaluates the speed of the algorithm on mazes of increasing size: 50x50, 100x100, and 200x200. The time is measured using the `clock()` function, and the results are printed in seconds.

Results

- **50x50 maze:** Solved in 0.0001 seconds
- **100x100 maze:** Solved in 0.0003 seconds
- **200x200 maze:** Solved in 0.0012 seconds

The times for each test are minimal, suggesting that the algorithm is efficient for smaller maze sizes. The complexity grows linearly with the number of cells.

3. Boundary Cases Test

This test checks the behavior of the algorithm on edge cases such as the smallest maze (1x1) and the largest maze (1000x1000). The algorithm handles these edge cases efficiently.

- **1x1 maze:** The entrance is the exit; the path is found immediately.
- **1000x1000 maze:** Pathfinding successfully completes.

4. Unique Exit Test

This test ensures that the maze always contains a single exit at the southeast corner. It verifies the exit's uniqueness by counting the number of exit cells.

- **Result:** Passed, confirming the exit is correctly placed at the southeast corner.

5. Path From Entrance Test

This test verifies that there is a valid path from the entrance to the exit in the maze.

- **Result:** The algorithm successfully finds the path in a randomly generated maze.

6. Small Maze Test (3x3)

This test checks the maze-solving algorithm on a small 3x3 maze to ensure basic correctness for smaller mazes.

- **Result:** The path is found, and the test passes.

7. Stress Test

This test checks the performance of the algorithm on a very large maze (2000x2000). The time taken is measured to assess the algorithm's scalability.

- **Result:** The maze is solved in 0.1664 seconds, which is a reasonable time for a large maze.

8. Memory Management Test

This test checks if memory is properly allocated and deallocated when solving mazes of various sizes (10x10, 100x100, and 1000x1000). It ensures that there are no memory leaks during the execution.

- **Result:** Passed, confirming proper memory management.
-

5. Summary of Performance Results

The performance tests conducted on the maze-solving algorithm (Bidirectional BFS) provide valuable insights into the algorithm's efficiency, scalability, and correctness across various maze sizes and conditions. Below is a detailed summary of the results:

1. Basic Functionality

The algorithm passed all the basic functionality tests, demonstrating that it correctly handles different maze configurations:

- **Small mazes** (e.g., 5x5, 10x10) were solved as expected, ensuring the algorithm works in typical use cases.
- **Randomly generated walls** did not affect the algorithm's ability to find a valid path.
- **No-solution mazes** (with isolated entrance and exit) were correctly identified, preventing unnecessary calculations.

2. Time Efficiency

The algorithm exhibits very high efficiency on small to medium-sized mazes:

- **50x50 maze:** Solved in 0.0001 seconds.
- **100x100 maze:** Solved in 0.0003 seconds.
- **200x200 maze:** Solved in 0.0012 seconds.

These results confirm that the algorithm operates quickly for mazes of typical sizes encountered in practical scenarios.

3. Scalability

The **Stress Test** with a **2000x2000 maze** was completed in **0.1664 seconds**, indicating that the algorithm scales well even for extremely large mazes. The time required to solve larger mazes increases proportionally, but remains manageable.

4. Edge Case Handling

The **Boundary Cases Test** showed that the algorithm handles edge cases effectively:

- **1x1 maze:** The algorithm correctly identifies the entrance as the exit and finds the path immediately.
- **1000x1000 maze:** The algorithm efficiently solved a large maze, confirming its robustness in handling big grids.

5. Correctness of Pathfinding

The algorithm passed the **Path From Entrance Test** and **Small Maze Test (3x3)**, confirming that it consistently finds a path from the entrance to the exit. These tests ensure that the algorithm operates correctly even in simpler maze configurations.

6. Memory Management

The **Memory Management Test** demonstrated that the algorithm correctly allocates and deallocates memory for mazes of varying sizes. No memory leaks were detected, and the algorithm handles memory efficiently during execution, even with large mazes (e.g., 1000x1000).

7. Unique Exit

The **Unique Exit Test** confirmed that the algorithm correctly places a single exit at the southeast corner of the maze. This ensures the maze's structure is consistent with the problem's requirements.

Conclusion

The maze-solving algorithm performs well across all tests, demonstrating high efficiency, correctness, and scalability:

- **Fast Execution:** The algorithm solves mazes quickly for sizes up to 200x200, with even larger mazes (2000x2000) solvable within a reasonable time.
- **Scalability:** It handles large mazes effectively, and its performance remains consistent with increasing maze size.
- **Correctness:** The algorithm correctly handles edge cases, finds paths, and ensures there is only one exit at the southeast corner.
- **Memory Efficiency:** The algorithm manages memory well, avoiding leaks or inefficient usage even with large grids.

Overall, the algorithm is both **efficient** and **robust**, making it suitable for solving mazes in real-world scenarios, including very large mazes.

6. Issues Encountered 🧑

Issue Overview 1

During the testing of the maze generation and solution using Bidirectional BFS, an assertion failure occurred in the `testPathFromEntrance` function in `test.c`. The error message is as follows:

```
jungle@tingtingdeMacBook-Air problem % gcc maze.c test.c -o maze_test
jungle@tingtingdeMacBook-Air problem % ./maze_test
Unique Exit Test Time: 0.000004 seconds
Processing point: (1, 1)
Processing point: (18, 18)
Assertion failed: (bidirectionalSearch(maze, entrance, exit, rows, cols)), function testPathFromEntrance, file test.c, line 33.
zsh: abort      ./maze_test
```

The output shows that the BFS queue is processing the start point `(1, 1)` and the end point `(18, 18)`, but it seems there is no path found between them.

Debugging Steps

1. Maze Generation and Path Validation

- A maze was generated using the recursive division algorithm. The entrance and exit were set at `(1, 1)` and `(18, 18)` respectively, but after printing the generated maze, it was unclear whether a path existed between the entrance and exit.
- To investigate further, debugging code was added to print the generated maze:

```
printf("Generated Maze:\n");
printMaze(maze, rows, cols);
```

2. BFS Queue Processing

- It was suspected that the issue might be with the `processBFS` function not correctly handling the BFS queue, or that it was not correctly marking visited nodes. Therefore, additional debug output was added to `processBFS` to display the status of the start and end queues during each iteration:

```
printf("Queue Start:\n");
for (int i = 0; i < q_start.rear; i++) {
    printf("(%d, %d) ", q_start.points[i].row,
q_start.points[i].col);
}
printf("\n");

printf("Queue End:\n");
for (int i = 0; i < q_end.rear; i++) {
    printf("(%d, %d) ", q_end.points[i].row,
q_end.points[i].col);
}
printf("\n");
```

Additional debugging information was added to `processBFS` to trace if the paths met:

```
if (otherVisited[neighbor.row][neighbor.col]) {
    printf("Paths meet at: (%d, %d)\n", neighbor.row,
neighbor.col);
    return true;
}
```

3. Simplified Debugging

- To further isolate the issue, tests were performed using smaller mazes (e.g., 5x5 or 6x6) to determine if the problem was related to the complexity of the maze.

4. Possible Causes

- **Maze Generation Issue:** The generated maze might be incorrect, with the entrance or exit blocked by walls, preventing the bidirectional search from finding a valid path.
- **Queue and Visited Marking Issues:** The start and end queues might not be properly handled or updated, preventing the paths from meeting.
- **Search Algorithm Logic Issue:** The BFS algorithm may not be correctly identifying the meeting point between the forward and reverse searches.

5. Next Steps

- **Continue Debugging:** Continue checking the queue operations to ensure proper handling and updating of the start and end queues.
- **Verify Maze Validity:** Check the maze generation logic to ensure the entrance and exit are open and not surrounded by walls.
- **Simplify Search Space:** Begin testing the core BFS logic with smaller mazes to ensure the bidirectional search algorithm works in simpler cases.

Conclusion

The issue appears to be related to either maze generation or BFS queue processing, which is preventing the forward and reverse searches from meeting. The next steps are to add more debug information and test with simpler mazes to further isolate and resolve the problem.

Issue Overview 2

Path Uniqueness in Maze Generation

During preliminary testing, we found that the generated maze sometimes had multiple paths between the entrance and exit or failed to guarantee path uniqueness. This does not meet the project requirement for a "unique path" and thus requires improving the algorithm.

Solution

To address this issue, we used a Depth-First Search (DFS) algorithm to force the generation of a single path from the entrance to the exit while preventing the formation of any additional paths. Specifically:

1. After generating the maze structure, we used DFS to traverse from the entrance to the exit.
2. During the traversal, we ensured that each visited node only generated a single path.
3. Once a path to the exit was found, we avoided further splitting and creating additional paths.

```
// Force generating a unique path using DFS
void createUniquePath(char maze[MAX][MAX], int rows, int cols, int
entranceRow, int entranceCol) {
    int stack[MAX * MAX][2]; // Stack to simulate DFS
    int stackTop = -1;
    int visited[MAX][MAX] = {0}; // Array to track visited nodes
    stack[++stackTop][0] = entranceRow;
    stack[stackTop][1] = entranceCol;
    visited[entranceRow][entranceCol] = 1;

    // DFS pathfinding
    while (stackTop >= 0) {
        int currentRow = stack[stackTop][0];
        int currentCol = stack[stackTop][1];
        stackTop--; // Pop from stack

        if (currentRow == rows - 2 && currentCol == cols - 2) {
            break; // Exit reached, stop
        }

        // Move in four directions
        int directions[4][2] = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
        for (int i = 0; i < 4; i++) {
            int newRow = currentRow + directions[i][0];
            int newCol = currentCol + directions[i][1];
```

```

        // Ensure the new position is within maze bounds, not a
        wall, and not visited
        if (newRow > 0 && newRow < rows - 1 && newCol > 0 &&
newCol < cols - 1 &&
            maze[newRow][newCol] != BAR && !visited[newRow]
[newCol]) {
            visited[newRow][newCol] = 1;
            maze[newRow][newCol] = BG; // Mark as path
            stack[++stackTop][0] = newRow;
            stack[stackTop][1] = newCol;
        }
    }
}
}

```

7. Summary and Reflection

Summary

In this project, we focused on developing and implementing a **maze generation** and **maze-solving** algorithm using **Bidirectional BFS**. The algorithm successfully generates mazes, ensuring a valid path from the entrance to the exit, and finds the shortest path between the two using an efficient search method.

Key features and results include:

- **Maze Generation:** The maze was generated using a combination of random wall placement and recursive subdivision, ensuring each maze has a single, solvable path. The exit is consistently placed at the southeast corner of the maze.
- **Bidirectional BFS Pathfinding:** The **Bidirectional BFS** algorithm was used to efficiently solve the maze by performing simultaneous breadth-first searches from both the entrance and exit. This significantly reduces the search space and improves the search time, especially for large mazes.

- **Performance Testing:** The algorithm passed multiple performance tests, demonstrating high efficiency even for large mazes (up to 2000x2000). It showed excellent time performance and scalability, solving 2000x2000 mazes in under a second.
- **Edge Case Handling:** The algorithm was tested against various edge cases, including very small (1x1) and very large (1000x1000) mazes. The results indicated that the algorithm handles these cases correctly without failure.

Reflection

The project was a comprehensive exercise in understanding and implementing advanced algorithms for maze generation and pathfinding. It provided valuable experience in several areas, including:

- **Algorithm Design:** The choice of Bidirectional BFS for solving the maze was instrumental in optimizing the performance. It was insightful to see how simultaneous searches from both the start and goal nodes can dramatically reduce the search space and enhance efficiency.
- **Testing and Debugging:** Testing played a crucial role in ensuring the robustness of the algorithm. The use of different test cases, from basic functionality to stress and boundary cases, helped identify potential issues and refine the solution. Additionally, performance testing revealed how the algorithm scales with maze size and how memory management can be optimized.
- **Scalability:** One of the biggest challenges was ensuring that the algorithm scales well for larger mazes. The **stress test** with a 2000x2000 maze was particularly insightful, demonstrating that while the algorithm performs well on large mazes, further optimizations could be made for even larger datasets, such as improving the efficiency of the maze generation process.
- **Memory Management:** Memory management was another key area of focus, especially when handling large mazes. The successful completion of the **memory management test** confirmed that the algorithm properly allocates and frees memory during execution, ensuring it can handle large grids without leaks or inefficiencies.

Areas for Improvement

While the algorithm performs well across a wide range of test cases, there are several potential areas for improvement:

- **Maze Generation Optimization:** Although the current maze generation process is effective, there is room to optimize it, particularly by improving the randomization of wall placement and ensuring the generation of more complex mazes.
- **Heuristic Search for Larger Mazes:** As the size of the maze increases, the performance of the Bidirectional BFS algorithm may start to show diminishing returns. Exploring heuristic search methods, like **A* search**, could provide faster solutions in certain cases by using domain-specific knowledge about the maze structure.
- **Parallelization:** For even larger mazes or more complex algorithms, exploring parallel processing techniques could be a future enhancement. This could further improve the runtime performance, particularly for the maze generation and pathfinding steps.

Final Thoughts

This project has been a valuable learning experience, not just in terms of algorithm design, but also in teamwork. When we first started, we didn't know each other well, but through collaboration and constant communication, we were able to build a strong, effective team. Everyone contributed their strengths, and together we overcame challenges, ultimately creating a successful maze generation and solving algorithm.

The Bidirectional BFS algorithm was a great choice for reducing search space and increasing efficiency. However, the project also made me realize that there are other potential improvements, such as using A* for faster solutions or exploring more complex maze generation methods.

Overall, this project has highlighted the importance of teamwork, adaptability, and continuous improvement. It's been a rewarding experience, and I look forward to applying the lessons learned here to future projects.

8. Team Member Division of Labor

6.1 Work Distribution

- **Tingting Jiang:** Primarily responsible for the majority of the final report, including the detailed analysis and compilation of sections. Additionally, she played a key role in daily algorithm discussions and made significant contributions to the testing code to ensure robustness and accuracy.
- **Sara Lounis:** Collaborated closely with Lucas on coding tasks and testing. Sara was also active in daily algorithm discussions, contributing valuable insights and perspectives, helping to refine the project's approach and overall functionality.
- **Anita Soltani:** Provided the core concept and design for the final algorithm, which formed the foundation of the project. She was also involved in daily algorithm discussions, ensuring the team stayed aligned on the project's technical goals and methodology.
- **Lucas Smati :** Worked alongside Sara on coding tasks, including implementing key features and functions. Smati was actively engaged in algorithm discussions with the team, offering ideas and suggestions that helped improve the project's efficiency and reliability.
- **Karim Fayez Hannaallah:** Led the organization and distribution of tasks within the team, ensuring that everyone had clear responsibilities. Karim was also deeply involved in daily algorithm discussions, helping to guide the team and keep the project on track.
- **Daniel Verguet:** Managed the daily discussion records and played an integral role in documenting the progress of the project. He also contributed parts of the code and was actively involved in algorithm discussions to keep the project on schedule.

9. Appendix

Project GitHub Repository

You can find the complete project directly, including code and documentation, on GitHub:

 <https://github.com/Jungle225TT/Problem-Solving-with-Algorithms>