COSC2658 - Data Structures and Algorithms/COSC2469 - Algorithms and Analysis/COSC2203 - Algorithms and Analysis

Group Project (SAMPLE)

Assignment Overview:

This assignment involves designing and implementing a system for an autonomous robot to navigate through a dark maze without any prior knowledge of its structure. The robot will start at a random point and can move in four directions (UP, DOWN, LEFT, RIGHT), receiving feedback after each move (true for a successful move, false for hitting a wall, and win upon finding the exit). The maze is defined as a 2D grid with walls, empty spaces, and a single exit. The task is to develop a `Robot` class with a `navigate()` method that will guide the robot through the maze effectively and efficiently.

Preparation Advice:

1. **Understand the Problem**: Make sure you fully understand the maze structure, robot movements, and the type of feedback received after each move.
2. **Study Examples**: Review the examples/solutions shared on your course's GitHub page to understand the base code you are allowed to extend.
3. **Java Basics**: Brush up on your Java skills, especially focusing on object-oriented concepts since you cannot use the Java Collection Framework.
4. **Algorithmic Thinking**: Practice algorithmic design paradigms like brute force, greedy algorithms, and others that may be relevant to this problem.
5. **Data Structures Knowledge**: As you can't use the Java Collection Framework, you need to be comfortable creating custom data structures.

Problem Requirements:

- Create a `Robot` class without using the Java Collection Framework.
- Develop a `navigate()` method to move the robot through the maze using a `go()` method that returns movement feedback.
- Implement custom data structures to track visited locations and maze walls.
- Your solution should be as efficient as possible, minimizing the number of moves to reach the exit.
- The maze and robot's starting point are unknown to your program and will be tested with unseen mazes.

Step-by-Step Solution with Detailed Explanation:

1. Data Structures:

Since you cannot use the Java Collection Framework, you'll need to design your own data structures.

- **Maze Representation**:

```
public class MazeNode {
    boolean isWall;
    boolean isExit;
    boolean visited;
    // Constructor, getters and setters
}

public class CustomMaze {
    private MazeNode[][] grid;
    // Initialize the maze grid with the size of 1000x1000
    // Constructor and necessary methods
}
```

- **Tracking Visited Nodes**:

```
public class VisitedTracker {
    private boolean[][] visited;
    // Constructor and methods to mark visited nodes and check if a node is visited
}
```

2. Maze Navigation Algorithm:

A good algorithm for maze navigation could be a modified Depth-First Search (DFS) with backtracking.

- **DFS Algorithm**:

```
public void navigateDFS(int currentRow, int currentCol, CustomMaze maze) {
    if (!isValid(currentRow, currentCol, maze)) {
        return;
    }
    if (maze.getNodeAt(currentRow, currentCol).isExit) {
        // Exit found
        return;
    }
    maze.visit(currentRow, currentCol);
    // Recursive calls for UP, DOWN, LEFT, RIGHT
    navigateDFS(currentRow - 1, currentCol, maze); // UP
    navigateDFS(currentRow + 1, currentCol, maze); // DOWN
    navigateDFS(currentRow, currentCol - 1, maze); // LEFT
    navigateDFS(currentRow, currentCol + 1, maze); // RIGHT
    // Backtracking
    maze.unvisit(currentRow, currentCol);
}
```

  - The `isValid` method will check if the current position is within the bounds of the maze and not a wall.
  - The `isExit` attribute will be set to true for the node that represents the exit gate.
- **Backtracking Mechanism**: Backtracking is inherent to the recursive nature of the DFS. After visiting a node, the function will recursively visit adjacent nodes. If a path doesn't lead to the exit, it will backtrack.

3. Movement Logic:

You will need to implement a method that interacts with the provided `Maze` class and its `go()` method.

- **Movement Strategy**:

```
public void navigate() {
    CustomMaze maze = new CustomMaze();
    // Assume the starting point is provided, or start from (0,0)
    navigateDFS(startRow, startCol, maze);
}
```

4. Integration:

Ensure the `Robot` class is well-integrated with the `navigate()` method.

- **Robot Class**:

```
public class Robot {
    public void navigate() {
        // Your navigation code here
    }
    // Additional helper methods if needed
}
```

5. Complexity Analysis:

- **DFS Complexity**:
  - Time Complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph representing the maze. In the worst case, this is O(n * m) for an n x m maze.
  - Space Complexity: O(n * m) due to the recursive stack and the storage of visited nodes.

6. Testing and Evaluation:

Develop test cases based on different maze configurations and evaluate the performance.

- **Testing**:

```
public class RobotTest {
    public static void main(String[] args) {
        Robot robot = new Robot();
        robot.navigate();
        // Further testing and validation logic
    }
}
```

7. Documentation and Reporting:

Comment your code thoroughly and create a detailed report as specified.

- **Code Documentation**:

```
/**
 * Represents a node within the maze.
 * ...
 */
public class MazeNode {
    // ...
}

/**
 * Custom maze class for representing the maze grid.
 * ...
 */
public class CustomMaze {
    // ...
}

/**
 * Main robot class with navigation logic.
 * ...
 */
public class Robot {
    // ...
}
```

8. Finalization:

Prepare all deliverables, ensuring the README.txt and the video link are included.

This outline provides a structure for the task with explanations of the algorithms and data structures involved, as well as the complexities. You'll need to flesh out the actual code, ensure that it compiles and runs correctly, and validate it against test cases.

Code Editorial

```
// A class to represent a cell in the maze.
class MazeCell {
    boolean isWall;
    boolean isVisited;
    boolean isExit;
    // Constructor, initializes the cell with default values.
    MazeCell() {
        this.isWall = false;
        this.isVisited = false;
        this.isExit = false;
    }
}

// The main Robot class that will navigate through the maze.
public class Robot {
    private MazeCell[][] maze;
    private int rows;
    private int cols;

    // Constructor, initializes the maze with the given dimensions.
    public Robot(int rows, int cols) {
        this.rows = rows;
        this.cols = cols;
        this.maze = new MazeCell[rows][cols];
        initializeMaze();
    }

    // Initializes the maze with MazeCell objects.
    private void initializeMaze() {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                maze[i][j] = new MazeCell();
                // You'll need to set the wall and exit cells based on your maze input.
            }
        }
    }

    // The navigate method that contains the main logic for moving the robot.
    public void navigate() {
        // Start from a random point or a fixed point, as per your assignment needs.
        int startRow = 0; // Example starting point
        int startCol = 0; // Example starting point
        navigateFrom(startRow, startCol);
    }

    // Recursive method to navigate from a given cell.
    private void navigateFrom(int row, int col) {
        // Check bounds and wall condition
        if (!isSafe(row, col)) {
            return;
        }

        // Check if the cell is the exit
        if (maze[row][col].isExit) {
            System.out.println("Exit found at " + row + ", " + col);
            return;
        }

        // Mark the current cell as visited
        maze[row][col].isVisited = true;

        // Recursively explore the neighboring cells
        navigateFrom(row - 1, col); // Move UP
        navigateFrom(row + 1, col); // Move DOWN
        navigateFrom(row, col - 1); // Move LEFT
        navigateFrom(row, col + 1); // Move RIGHT

        // Backtrack if no exit is found in this path
        maze[row][col].isVisited = false;
    }
    // Checks if the cell is safe to visit (i.e., not a wall and within bounds).
    private boolean isSafe(int row, int col) {
        return row >= 0 && row < rows && col >= 0 && col < cols && !maze[row][col].isWall && !maze[row][col].isVisited;
    }

    // Entry point of the program
    public static void main(String[] args) {
        // Initialize robot with maze dimensions (rows, cols)
        Robot robot = new Robot(30, 100); // Example dimensions
        robot.navigate();
    }
}
```

In this skeleton:

- The `MazeCell` class represents a cell in the maze.
- The `Robot` class contains the maze and the navigation logic.
- The `navigate()` method starts the navigation process.
- The `navigateFrom()` is a recursive method that attempts to move to each adjacent cell.
- The `isSafe()` method checks whether a cell is within bounds and not a wall or already visited.

You'll need to fill in the logic for initializing the maze with walls and the exit based on your assignment's input format. Additionally, you'll need to handle the interaction with the `Maze` class's `go()` method as specified in your assignment description, along with any additional functionality required.

Remember to also handle the `false`, `true`, and `win` signals that the `go()` method returns, which isn't reflected in this simple skeleton. This is an essential part of the assignment that will require additional logic to be added to the `Robot` class.

Please use this as a starting point and build upon it to create a complete solution that meets all of your assignment's requirements.

Advices to write Readme.txt and Video presentation When creating your `README.txt` and preparing your video presentation, it's important to communicate clearly and effectively, ensuring that you cover all the essential aspects of your project. Here are some tailored advices for each:

Advice for Writing README.txt:

1. **Start with the Basics**: Begin with the project's title, the team members' names, and the purpose of the project.
2. **Include Installation and Running Instructions**: Provide step-by-step instructions for installing and running your software, including any prerequisites or dependencies.
3. **Describe the File Structure**: Briefly explain the organization of your project directory and the contents of each file.
4. **Usage**: Clearly describe how to use the program, including any command-line arguments or user inputs.
5. **Contribution Details**: If it’s a team project, describe each member's contributions to give proper credit and show teamwork.
6. **Link to Video**: Ensure the video link is correct and accessible. The video should be public or unlisted but not private.
7. **Contact Information**: Provide a way to reach out in case there are questions or feedback.
8. **Known Issues and Limitations**: If there are any known bugs or limitations, list them. It shows honesty and self-awareness of your project's scope.
9. **Acknowledgments**: If you have used resources such as code snippets, libraries, or if you’ve received help from someone, acknowledge their contribution.
10. **Formatting**: Keep your README.txt well-formatted. Use bullet points, headings, and clear paragraph breaks to make it easy to read.

Advice for Video Presentation:

1. **Structure Your Presentation**: Have a clear introduction, body, and conclusion. Start with an overview of the project, discuss the main points in the body, and conclude with a summary of what was achieved.
2. **Demonstrate the Software**: Show your software in action. Demonstrate how to start it, and walk through a typical use case.
3. **Explain the Technical Aspects**: Briefly discuss the algorithms, data structures, and design patterns used in your project. Use layman's terms to make it accessible.
4. **Discuss Challenges and Solutions**: Talk about any significant challenges you faced and how you overcame them.
5. **Highlight Unique Features**: Point out what makes your project stand out. It could be efficiency, scalability, or something unique about your implementation.
6. **Keep it Concise**: Since the video should be short, practice what you want to say to stay within the time limit.

7. **Good Audio and Video Quality**: Ensure that the audio is clear and the video is well-lit. If you're showing code, make sure it's readable in the video.
8. **Practice Your Delivery**: Rehearse your presentation to make it smooth and ensure you cover all the points without rushing.
9. **Engage the Audience**: Try to make the presentation engaging. Speak with enthusiasm and confidence.
10. **End with a Call to Action**: Invite viewers to try the software, contribute, or provide feedback.
11. **Edit for Clarity**: If possible, edit the video to cut out any mistakes or unnecessary parts to keep it professional and polished.

Remember, your README.txt and video presentation are often the first things people will engage with. They should both reflect the quality and professionalism of your work on the project.

Highlights and Advice for the Maze Navigation Robot Assessment:

Highlights:

1. **Algorithm Selection**: Choosing the right algorithm is crucial. Depth-First Search (DFS) and Breadth-First Search (BFS) are good starting points for maze navigation, but consider variations or entirely different algorithms based on efficiency and maze characteristics.
2. **Data Structures**: Designing your own data structures is a great exercise in understanding memory and data organization. Pay attention to how you track visited cells and walls as it's central to the efficiency of your robot's navigation.
3. **Efficiency**: Strive for an efficient solution, both in terms of algorithm complexity and the number of moves made by the robot. This is not just about finding a path but finding a good path quickly.
4. **Testing**: Thorough testing with various maze configurations is necessary to ensure that your algorithm works in all scenarios, not just the simplest cases.
5. **Documentation**: Proper documentation and clear code are essential. They help you keep track of your logic and make your code understandable to others, including your instructor.

What-to-Avoid:

1. **Overlooking Edge Cases**: Don't assume the maze will be straightforward; design your solution to handle complex scenarios, including loops, multiple paths, and dead ends.
2. **Ignoring Feedback**: The robot receives feedback after each move. Make sure your code listens to this feedback to avoid repeating the same mistakes, such as walking into walls.
3. **Reinventing the Wheel**: While you can't use Java's Collection Framework, that doesn't mean you can't apply the underlying principles of those data structures. Don't make your solution more complicated than it needs to be.
4. **Inefficiency**: Avoid brute-force approaches that don't consider the path already taken. This can lead to an exponential number of unnecessary moves.
5. **Neglecting Time Complexity**: It's easy to get a solution that works but performs poorly on larger mazes. Always consider the time complexity of your algorithms.
6. **Hardcoding Solutions**: Don't hardcode paths based on the sample maze. The solution should be dynamic and adaptable to any maze configuration.
7. **Underestimating Backtracking**: If not handled properly, backtracking can lead to a lot of redundant computation. Make sure to mark visited paths and possibly use a stack to keep track of the path history.
8. **Poor Code Organization**: Avoid writing monolithic code. Break down your solution into classes and methods with single responsibilities for easier maintenance and readability.

Additional Advice for Students:

- **Start Early**: Don't wait until the last minute. Complex problems require time to understand and solve effectively.
- **Collaborate Wisely**: If this is a group project, communicate and divide work efficiently. Ensure everyone's code integrates well.
- **Seek Feedback**: Don't hesitate to ask for help or feedback from peers or instructors. Fresh perspectives can be very helpful.
- **Reflect and Iterate**: After your initial implementation, step back and consider what could be improved, both in terms of code quality and algorithmic efficiency.

Remember, this assessment is not only about getting to the solution but also about the quality of the journey. Good coding practices, a thoughtful approach, and robust testing are just as important as a working end product.