Project 3 Report

Corinna Chang
Paul Chon

November 2020

Note:

We mainly used Intellij to code, but we used Eclipse for the JUnit testing and exporting the project.

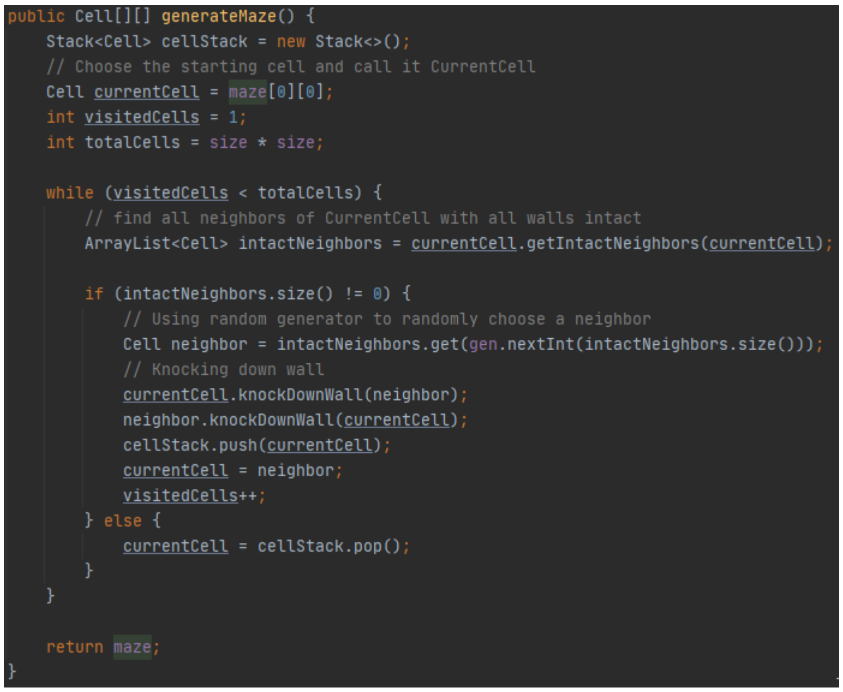
A Generating Maze

A.1 OBJECTIVE:

The objective of this part is to generate a perfect maze, which is a maze where there are no cycles, no inaccessible sections, and no open areas. Essentially there is only one path from any one point to another.

A.2 IMPLEMENTATION:

The maze is represented by a 2D array of size n by n, as specified in the argument. The method createGrid() generates a grid for the maze, and the method generateMaze(), which can be seen in Figure 1, creates a perfect maze by using depth-first search. We use a variable visitedCells to make sure that all cells are visited before the while loop terminates. We also included a stack to keep track of the order the cells are visited and backtrack if there are no eligible walls to knock down. The algorithm starts with the starting cell then it will find intact neighbors, which are neighboring cells that have all four of their walls intact, and pick a random intact neighbor to knock down the shared wall between them. This is repeated until all cells are visited and a perfect maze is generated.



```java
public Cell[][] generateMaze() {
    Stack<Cell> cellStack = new Stack<>();
    // Choose the starting cell and call it CurrentCell
    Cell currentCell = maze[0][0];
    int visitedCells = 1;
    int totalCells = size * size;

    while (visitedCells < totalCells) {
        // find all neighbors of CurrentCell with all walls intact
        ArrayList<Cell> intactNeighbors = currentCell.getIntactNeighbors(currentCell);

        if (intactNeighbors.size() != 0) {
            // Using random generator to randomly choose a neighbor
            Cell neighbor = intactNeighbors.get(gen.nextInt(intactNeighbors.size()));
            // Knocking down wall
            currentCell.knockDownWall(neighbor);
            neighbor.knockDownWall(currentCell);
            cellStack.push(currentCell);
            currentCell = neighbor;
            visitedCells++;
        } else {
            currentCell = cellStack.pop();
        }
    }

    return maze;
}
```

Figure 1: Generating the maze using depth-first search.

B Solving and Printing Maze

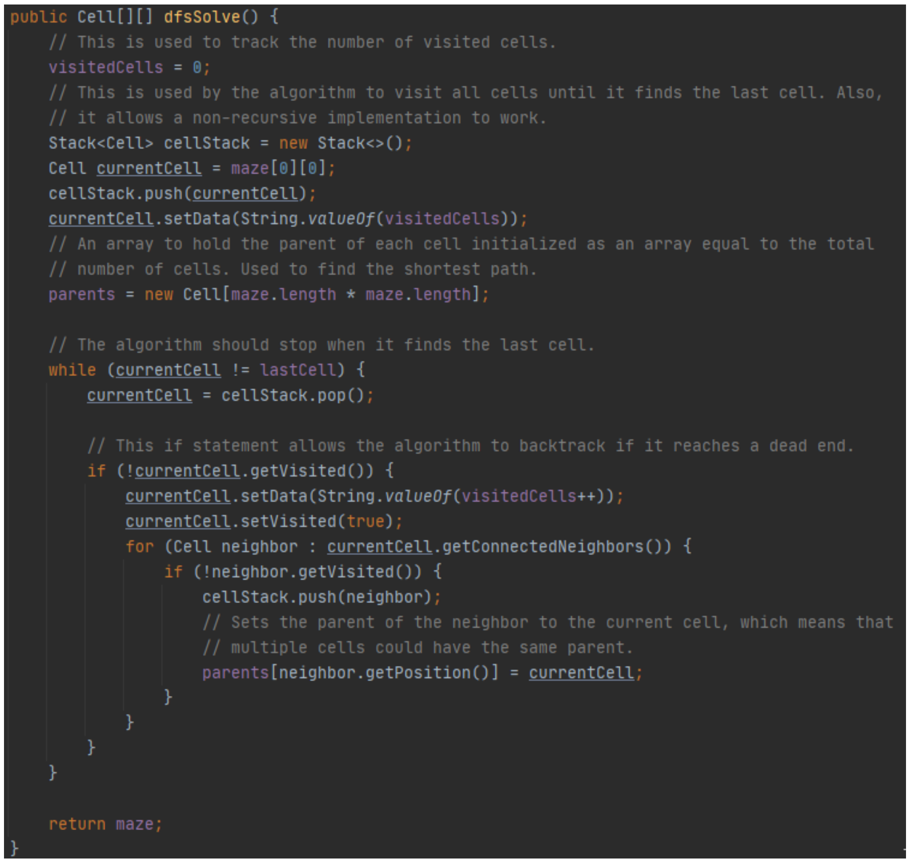
B.1 OBJECTIVE:

The objective of this part is to solve the maze by finding the shortest path from the starting room to the finishing room using depth-first search and breadth-first search. For each searching method, two mazes (one with the order the cells are visited and the other with the shortest path), coordinates of the shortest path, length of path, and cells visited are printed.

B.2 IMPLEMENTATION:

B.2.1 Depth-First Search:

Depth-first search uses a stack to keep track of the visited cells and the order they are visited. We traverse the maze from the starting room, which is pushed on the stack and visited is set as true. The current cell is popped from the stack. If it has not been visited, visited is set as true and it pushes a connected neighbor that has not yet been visited into the stack. The parent of the connected neighbor is set as the current cell. Otherwise, we pop the next cell in the stack and repeat the while loop until the stack is empty or if the finishing room has been reached.

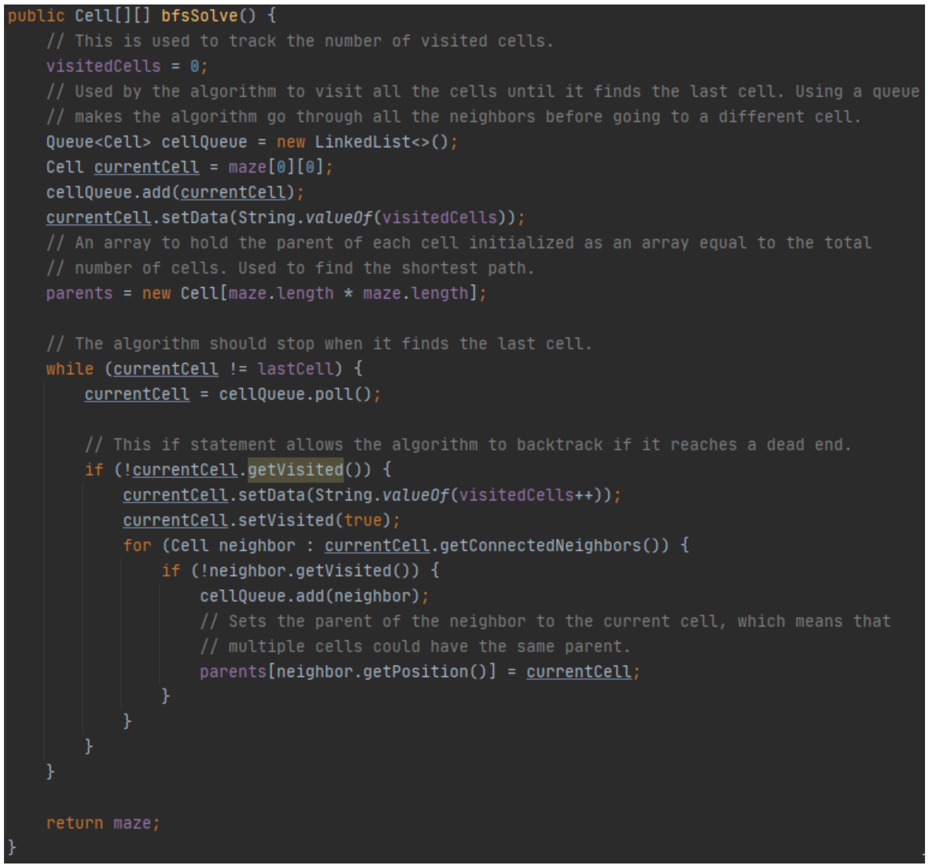


```java
public Cell[][] dfsSolve() {
    // This is used to track the number of visited cells.
    visitedCells = 0;
    // This is used by the algorithm to visit all cells until it finds the last cell. Also,
    // it allows a non-recursive implementation to work.
    Stack<Cell> cellStack = new Stack<>();
    Cell currentCell = maze[0][0];
    cellStack.push(currentCell);
    currentCell.setData(String.valueOf(visitedCells));
    // An array to hold the parent of each cell initialized as an array equal to the total
    // number of cells. Used to find the shortest path.
    parents = new Cell[maze.length * maze.length];

    // The algorithm should stop when it finds the last cell.
    while (currentCell != lastCell) {
        currentCell = cellStack.pop();

        // This if statement allows the algorithm to backtrack if it reaches a dead end.
        if (!currentCell.getVisited()) {
            currentCell.setData(String.valueOf(visitedCells++));
            currentCell.setVisited(true);
            for (Cell neighbor : currentCell.getConnectedNeighbors()) {
                if (!neighbor.getVisited()) {
                    cellStack.push(neighbor);
                    // Sets the parent of the neighbor to the current cell, which means that
                    // multiple cells could have the same parent.
                    parents[neighbor.getPosition()] = currentCell;
                }
            }
        }
    }

    return maze;
}
```

Figure 2: Non-recursive depth-first search.

B.2.2 Breadth-First Search:

Breadth-first search uses a queue to keep track of the visited cells and the order they are visited. We traverse the maze from the starting room, which is added to the queue. In the while loop, the first cell is the queue is retrieved. If the cell has not yet been visited, we get the connected neighboring cells that had not been visited and add them to the queue. The parent of the connected neighbor is set as the current cell. The loop runs until the queue is empty or the finishing room is reached.

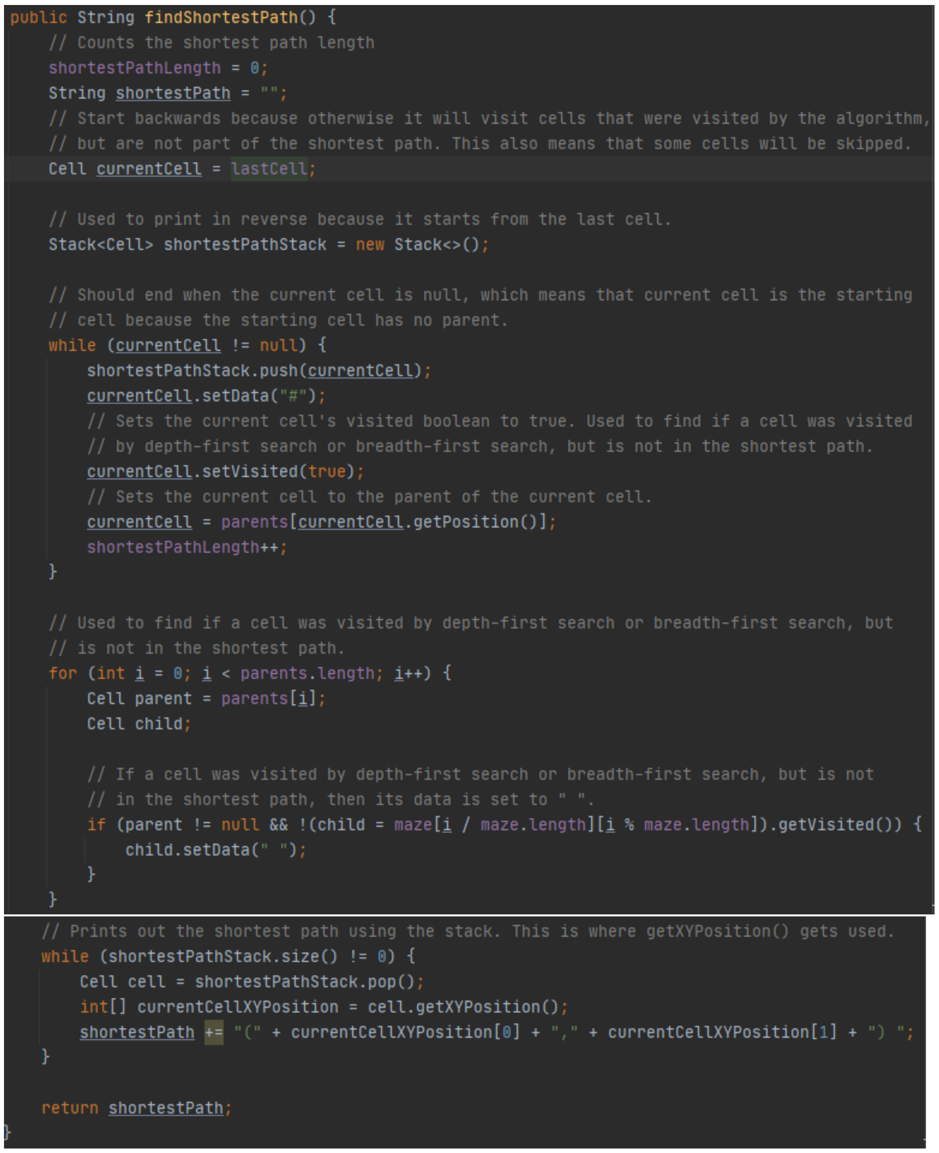


```java
public Cell[][] bfsSolve() {
    // This is used to track the number of visited cells.
    visitedCells = 0;
    // Used by the algorithm to visit all the cells until it finds the last cell. Using a queue
    // makes the algorithm go through all the neighbors before going to a different cell.
    Queue<Cell> cellQueue = new LinkedList<>();
    Cell currentCell = maze[0][0];
    cellQueue.add(currentCell);
    currentCell.setData(String.valueOf(visitedCells));
    // An array to hold the parent of each cell initialized as an array equal to the total
    // number of cells. Used to find the shortest path.
    parents = new Cell[maze.length * maze.length];

    // The algorithm should stop when it finds the last cell.
    while (currentCell != lastCell) {
        currentCell = cellQueue.poll();

        // This if statement allows the algorithm to backtrack if it reaches a dead end.
        if (!currentCell.getVisited()) {
            currentCell.setData(String.valueOf(visitedCells++));
            currentCell.setVisited(true);
            for (Cell neighbor : currentCell.getConnectedNeighbors()) {
                if (!neighbor.getVisited()) {
                    cellQueue.add(neighbor);
                    // Sets the parent of the neighbor to the current cell, which means that
                    // multiple cells could have the same parent.
                    parents[neighbor.getPosition()] = currentCell;
                }
            }
        }
    }

    return maze;
}
```

Figure 3: Breadth-first search implementation

B.2.3 Shortest Path

To find the shortest path through the maze, we use a stack to keep track of the order each cells are visited. We start from the finishing room as current cell and get its parent. The parent is then set as current cell, and we continue this while loop until the starting room is reached. Next, we use a for loop to remove any unused cell to ensure that only the shortest path shows up when we print the maze. Finally, we pop the cells from the stack, which gives us the exact cells of the shortest path.

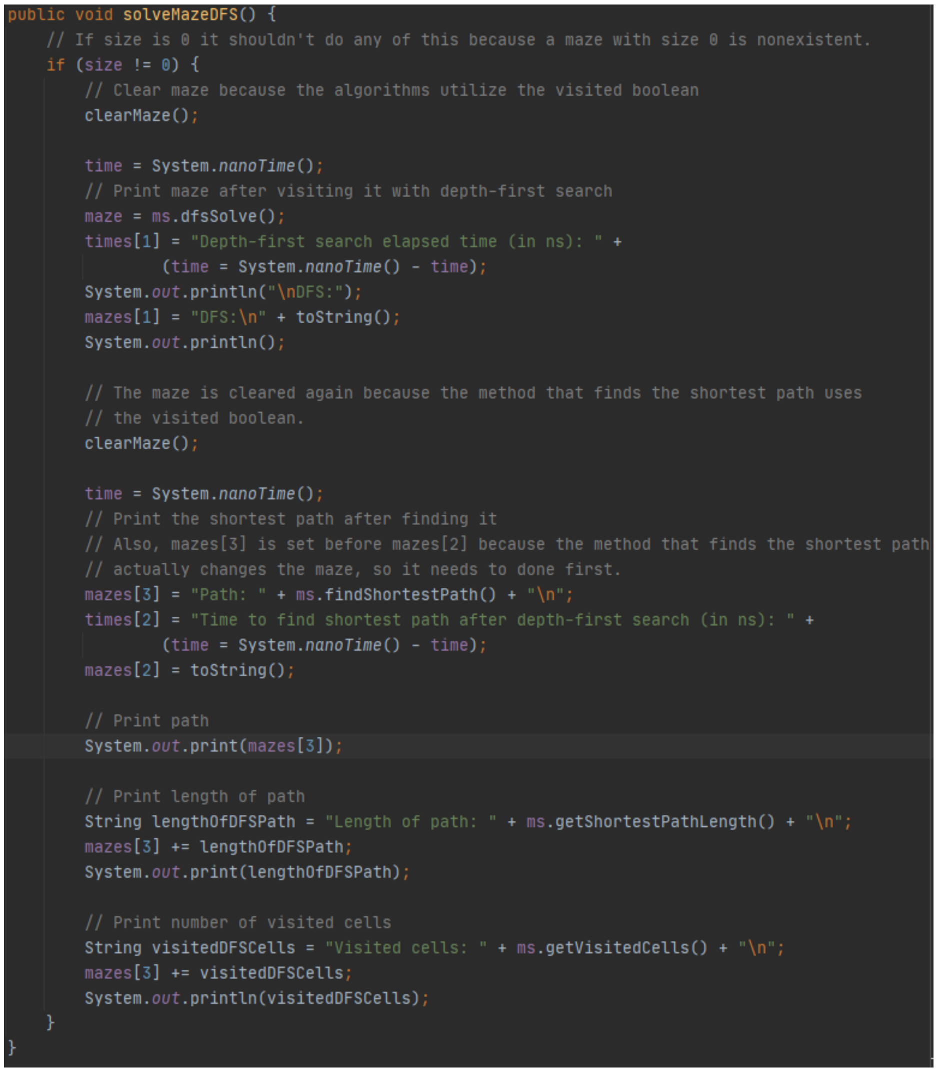


```java
public String findShortestPath() {
    // Counts the shortest path length
    shortestPathLength = 0;
    String shortestPath = "";
    // Start backwards because otherwise it will visit cells that were visited by the algorithm,
    // but are not part of the shortest path. This also means that some cells will be skipped.
    Cell currentCell = lastCell;

    // Used to print in reverse because it starts from the last cell.
    Stack<Cell> shortestPathStack = new Stack<>();

    // Should end when the current cell is null, which means that current cell is the starting
    // cell because the starting cell has no parent.
    while (currentCell != null) {
        shortestPathStack.push(currentCell);
        currentCell.setData("#");
        // Sets the current cell's visited boolean to true. Used to find if a cell was visited
        // by depth-first search or breadth-first search, but is not in the shortest path.
        currentCell.setVisited(true);
        // Sets the current cell to the parent of the current cell.
        currentCell = parents[currentCell.getPosition()];
        shortestPathLength++;
    }

    // Used to find if a cell was visited by depth-first search or breadth-first search, but
    // is not in the shortest path.
    for (int i = 0; i < parents.length; i++) {
        Cell parent = parents[i];
        Cell child;

        // If a cell was visited by depth-first search or breadth-first search, but is not
        // in the shortest path, then its data is set to " ".
        if (parent != null && !(child = maze[i / maze.length][i % maze.length]).getVisited()) {
            child.setData(" ");
        }
    }
    // Prints out the shortest path using the stack. This is where getXYPosition() gets used.
    while (shortestPathStack.size() != 0) {
        Cell cell = shortestPathStack.pop();
        int[] currentCellXYPosition = cell.getXYPosition();
        shortestPath += "(" + currentCellXYPosition[0] + "," + currentCellXYPosition[1] + ") ";
    }

    return shortestPath;
}
```

Figure 4: Finding the shortest path after solving with either DFS or BFS.

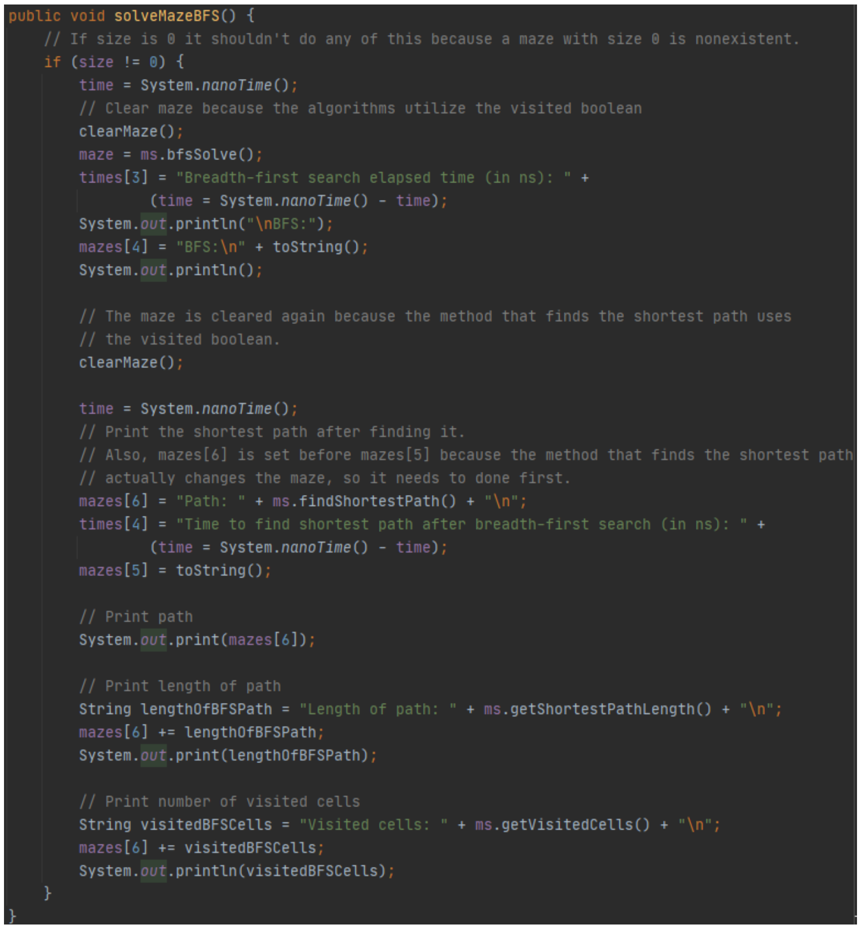
B.2.4 Maze Printing and Saving

To print the maze, we need to go through each row three times: first time for the north wall, second for the east and west walls, and third for the south wall. Only the first row prints out the north wall, since the south wall of the row above is the current row's north wall. Similarly, when printing the east and west walls, only the first column prints out the west wall. Each cell's data will be printed out as the path through the maze. Thus, unvisited cells will remain blank. The maze is stored as a string and printed out using PrintWriter. Also, the Maze class combines everything and prints out the output and saves the output to the file as well shown in Figure 5, Figure 6, and Figure 7.



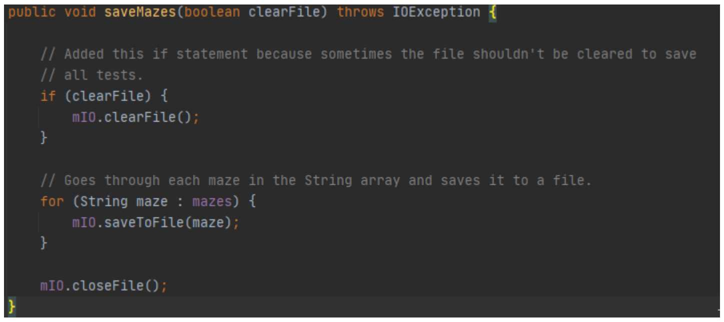
```java
public void solveMazeDFS() {
    // If size is 0 it shouldn't do any of this because a maze with size 0 is nonexistent.
    if (size != 0) {
        // Clear maze because the algorithms utilize the visited boolean
        clearMaze();

        time = System.nanoTime();
        // Print maze after visiting it with depth-first search
        maze = ms.dfsSolve();
        times[1] = "Depth-first search elapsed time (in ns): " +
                (time = System.nanoTime() - time);
        System.out.println("\nDFS:");
        mazes[1] = "DFS:\n" + toString();
        System.out.println();

        // The maze is cleared again because the method that finds the shortest path uses
        // the visited boolean.
        clearMaze();

        time = System.nanoTime();
        // Print the shortest path after finding it
        // Also, mazes[3] is set before mazes[2] because the method that finds the shortest path
        // actually changes the maze, so it needs to done first.
        mazes[3] = "Path: " + ms.findShortestPath() + "\n";
        times[2] = "Time to find shortest path after depth-first search (in ns): " +
                (time = System.nanoTime() - time);
        mazes[2] = toString();

        // Print path
        System.out.print(mazes[3]);

        // Print length of path
        String lengthOfDFSPath = "Length of path: " + ms.getShortestPathLength() + "\n";
        mazes[3] += lengthOfDFSPath;
        System.out.print(lengthOfDFSPath);

        // Print number of visited cells
        String visitedDFSCells = "Visited cells: " + ms.getVisitedCells() + "\n";
        mazes[3] += visitedDFSCells;
        System.out.println(visitedDFSCells);
    }
}
```

Figure 5: Maze class printing and saving DFS output.



```
public void solveMazeBFS() {
    // If size is 0 it shouldn't do any of this because a maze with size 0 is nonexistent.
    if (size != 0) {
        time = System.nanoTime();
        // Clear maze because the algorithms utilize the visited boolean
        clearMaze();
        maze = ms.bfsSolve();
        times[3] = "Breadth-first search elapsed time (in ns): " +
                (time = System.nanoTime() - time);
        System.out.println("\nBFS:");
        mazes[4] = "BFS:\n" + toString();
        System.out.println();

        // The maze is cleared again because the method that finds the shortest path uses
        // the visited boolean.
        clearMaze();

        time = System.nanoTime();
        // Print the shortest path after finding it.
        // Also, mazes[6] is set before mazes[5] because the method that finds the shortest path
        // actually changes the maze, so it needs to done first.
        mazes[6] = "Path: " + ms.findShortestPath() + "\n";
        times[4] = "Time to find shortest path after breadth-first search (in ns): " +
                (time = System.nanoTime() - time);
        mazes[5] = toString();

        // Print path
        System.out.print(mazes[6]);

        // Print length of path
        String lengthOfBFSPath = "Length of path: " + ms.getShortestPathLength() + "\n";
        mazes[6] += lengthOfBFSPath;
        System.out.print(lengthOfBFSPath);

        // Print number of visited cells
        String visitedBFSCells = "Visited cells: " + ms.getVisitedCells() + "\n";
        mazes[6] += visitedBFSCells;
        System.out.println(visitedBFSCells);
    }
}
```

Figure 6: Maze class printing and saving BFS output.



```
public void saveMazes(boolean clearFile) throws IOException {

    // Added this if statement because sometimes the file shouldn't be cleared to save
    // all tests.
    if (clearFile) {
        mIO.clearFile();
    }

    // Goes through each maze in the String array and saves it to a file.
    for (String maze : mazes) {
        mIO.saveToFile(maze);
    }

    mIO.closeFile();
}
```

Figure 7: Maze class saving output to file.

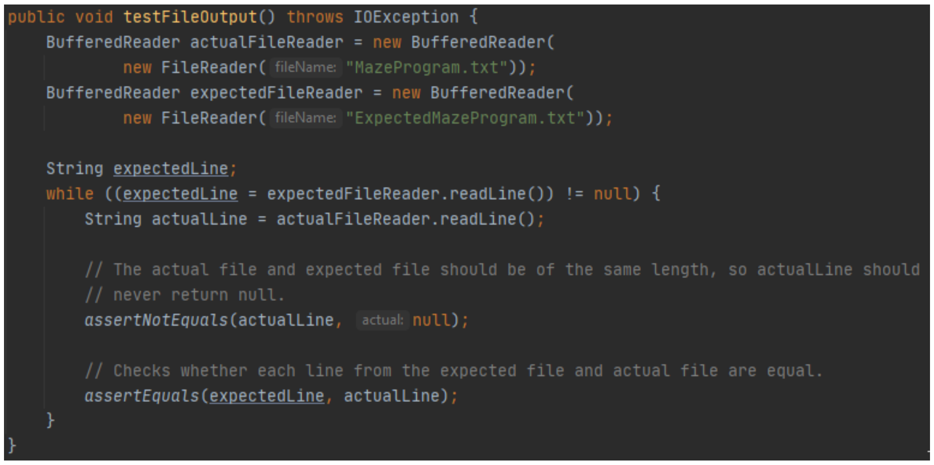
C Testing Maze

C.1 OBJECTIVE:

The objective for this part is to test that our algorithms generate a perfect maze and solve it with the shortest possible path. The tests are conducted using our own mazes.

C.2 IMPLEMENTATION:

We test mazes with size 0 to 10. The expected mazes are generated first and saved as a text file. We manually checked them to ensure that the mazes are correct. Next, we use JUnit for testing. We run the program again to generate a second text file of actual mazes. The actual file is compared with the expected file line by line using BufferedReader shown in Figure 8. Also, the running time of the program is printed out all in nanoseconds: maze generation, solving the maze with DFS, finding the shortest path after DFS, solving the maze with BFS, and finding the shortest path after BFS.



```
public void testFileOutput() throws IOException {
    BufferedReader actualFileReader = new BufferedReader(
            new FileReader( fileName: "MazeProgram.txt"));
    BufferedReader expectedFileReader = new BufferedReader(
            new FileReader( fileName: "ExpectedMazeProgram.txt"));

    String expectedLine;
    while ((expectedLine = expectedFileReader.readLine()) != null) {
        String actualLine = actualFileReader.readLine();

        // The actual file and expected file should be of the same length, so actualLine should
        // never return null.
        assertNotEquals(actualLine,  actual: null);

        // Checks whether each line from the expected file and actual file are equal.
        assertEquals(expectedLine, actualLine);
    }
}
```

Figure 8: Testing expected file and actual file.

Input Size	Elapsed Time (in ns)
1	153600
2	64200
3	35400
4	67400
5	77800
6	142300
7	194400
8	167500
9	383600
10	287700

Figure 9: Time to generate maze.

Input Size	Elapsed Time (in ns)
1	28200
2	66100
3	28300
4	45000
5	30900
6	59500
7	89900
8	84400
9	64500
10	78300

Figure 10: Time to solve maze using DFS.

Input Size	Elapsed Time (in ns)
1	1000500
2	45000
3	34100
4	51800
5	36700
6	47400
7	90700
8	102300
9	123500
10	826800

Figure 11: Time to find shortest path after DFS.

Input Size	Elapsed Time (in ns)
1	132100
2	36000
3	22600
4	47400
5	38500
6	80400
7	74300
8	85600
9	60500
10	216500

Figure 12: Time to solve maze using BFS.

Input Size	Elapsed Time (in ns)
1	132900
2	25000
3	31900
4	49700
5	59600
6	161500
7	79500
8	94800
9	104600
10	112700

Figure 13: Time to find shortest path after BFS.

Operation Name	Average time (in ns)
Generating Maze	157390
Depth-first search	57510
Finding shortest path after depth-first search	235880
Breadth-first search	79390
Finding shortest path after breadth-first search	85220

Figure 14: Average elapsed times for different operations.