Submission Information

Course:	CS 7375 – Artificial Intelligence
Student Name:	Marion Garrett Sisk
Student ID:	000942002
Assignment #:	1
Due Date:	2/21/21
Signature:	
Score:	

Table of Contents

Agent Design

The AI agent implemented within this assignment is designed to operate in a specific environment, and complete one task. The agent will solve for a path through a maze, given a starting point, and a goal, or end point. There are four elements to an AI agent: Performance metric(s), Environment, Actuators, and Sensors (or PEAS, as given in the textbook).

The performance metric provides the agent a framework from which to make decisions. In this instance, we are to solve a maze from an arbitrary starting point. The overall idea behind this task is we want to move closer to the goal. In other words, we want to reduce our distance between us and the goal as much as possible with each step taken in the maze. To accomplish

this, the agent uses a straight-line distance between the current location and the goal. When deciding where to move next, the agent picks the location with the shortest distance to the goal. Details on how this distance is implemented and computed are described alongside the Environment the agent is designed to work within.

The environment, in its most basic sense, is a grid maze with randomly generated paths. This is implemented using a custom *gridGraph* class. This data structure is based on a graph, where each node of the graph is a "cell" within the grid maze, and each cell has four "walls" which are given as edges between each node. The initial cells and walls are built using the constructor method such that the initial state is a full grid, with each "wall" present around and between each "cell." Using this initial state, the maze is generated by using a randomized version of Prim's Algorithm based off the description given in source [1]. This produces a spanning tree that produces "passages" through the "walls" of a "cell" when two are connected through this tree. The effect, when drawn onto the screen, shows a random maze with many short branching corridors. The agent utilizes attributes of each instance of "cell" to determine its state, and which direction to move. The way in which these attributes are examined and implemented is explained subsequently.

The "actuators" implemented within the agent is the process of moving into a cell closer to the goal cell. The choice is made to move into the cell based on the process described in the performance metric.

Lastly, the sensors of this agent include the ability to determine if a cell has been visited before, if a wall is a passage or not, and whether or not the current cell is the goal cell. The combination of these sensing abilities gives the agent the proper percept such that the appropriate decisions with the goal in mind is made.

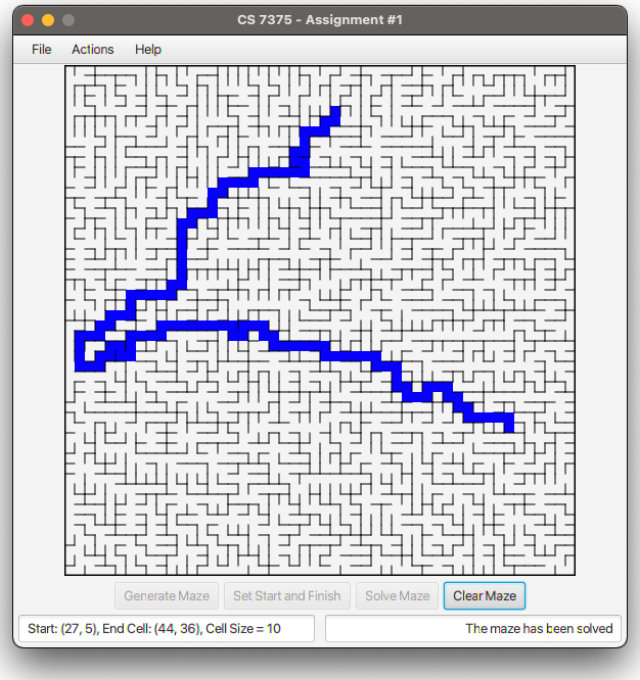
The way in which the agent works is akin to a greedy algorithm. It makes the best choice based on what is available to it at the moment, though does has some prescience since the location of the goal cell with respect to the current location is known. However, the agent will not know whether a path will result in a dead end until a dead end is reached. Additionally, the agent has issues when the path finding must cross the start cell to begin a new direction. The application handles the error by telling the user to reset the maze and try again. The pseudocode

of the algorithm used to implement the agent is given below. Note that straight-line distance is defined as the distance using the Pythagorean theorem, and the differences in the X and Y coordinates between the current cell and the goal cell. This quantity is used to compute the best option and subsequent cell to move forward to.

Figure 1: AI Agent Pseudocode.

```
WHILE the current cell is not equal to the goal cell
      Mark the current cell as visited
      Determine all possible moves
      IF there are possible moves available THEN
            Compute all distances
            Move to cell with shortest distance
            PUSH the cell to the path stack
      ELSE
            POP the current cell off the path stack
      END IF
END WHILE
```

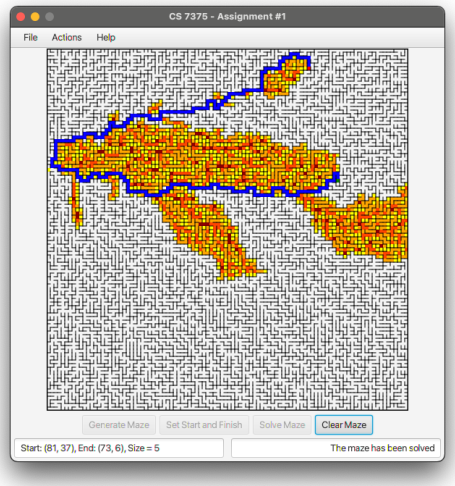
Output Screenshot

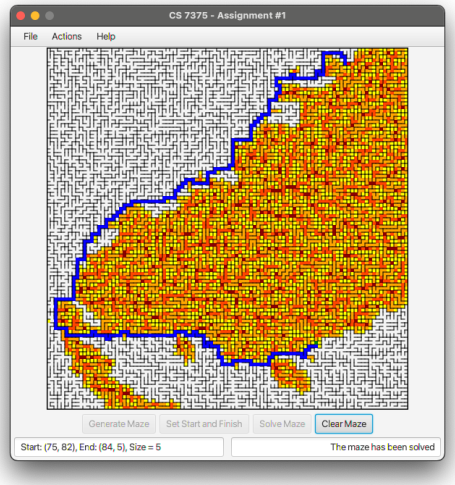


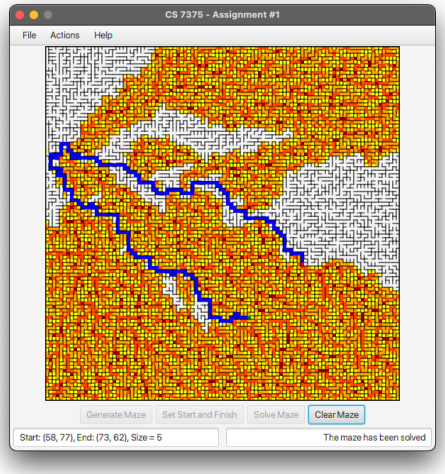
Tasks the Agent Can Solve

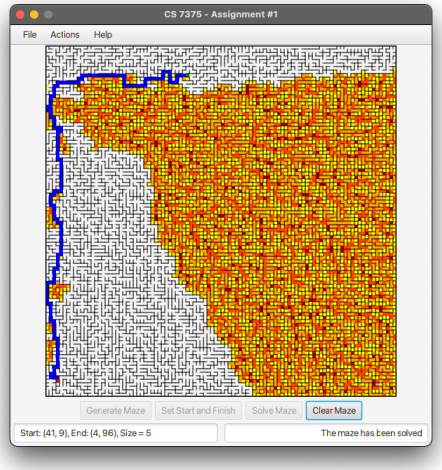
This particular agent can only solve for a path within a specific data structure. Using this agent outside of this particular environment will result in unpredictable behavior and may not

even work at all. Additionally, there was a last-minute addition of a feature that permits the user to see how efficient (or inefficient, as the case may be) the agent is at solving the problem. By using the visit count of each cell that keeps track of how many times the agent accesses that cell, it becomes possible to see where the agent has been and how many times. Below are several screen shots that illustrate this using the 100x100 grid maze. Note that the final solution path is given in blue, but the yellow to red cells indicate where the agent has visited within the maze. The darker the color, the more often the agent has visited a particular cell. It becomes obvious when looking at these images that the agent can be extremely inefficient at solving a maze, and as such suggests there is ample room for improvement in its behavior.









Video Demonstration

A video demonstration of the application can be found at the following YouTube link:

https://youtu.be/IZ10ZvcHrn0

GitHub Repository

All source files and associated binary files can be found on my personal GitHub page at:

https://github.com/mgarrettsisk/mazeSolver

References

[1] https://en.wikipedia.org/wiki/Maze_generation_algorithm#Randomized_Prim's_algorithm

Source Code

main.java

```
package assignment01;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root =
FXMLLoader.load(getClass().getResource("mainWindow.fxml"));
        primaryStage.setTitle("CS 7375 - Assignment #1");
        primaryStage.setScene(new Scene(root, 600, 600));
        primaryStage.setResizable(false);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

aiAgent.java

```
package assignment01;

import java.util.ArrayList;
import java.util.LinkedList;

public class aiAgent {
    // attributes
    private gridGraph.cell goalCell;
    private final LinkedList<gridGraph.cell> solutionPathStack = new
LinkedList<>();
    private final ArrayList<gridGraph.cell> possibleMoves = new
ArrayList<>();

    // constructor methods
    aiAgent(){
        // null constructor
    }
    aiAgent(gridGraph.cell startCell, gridGraph.cell goalCell) {
        // this constructor takes a start cell and goal cell and solves the
maze, producing a solutionPath
        // first, push the start cell onto the solutionPathStack list
        this.solutionPathStack.push(startCell);
        // second, set the goal cell attribute
        this.setGoalCell(goalCell);

    }
    // public methods
    public void solveMaze() {
        // this method actually does the solving, and creates the solution
path along the stack
        // define starting cell
        gridGraph.cell currentCell = this.solutionPathStack.peek();
        while (!(currentCell.equals(goalCell))) {
            // determine which cells can be moved to
            currentCell.visit();
            determinePossibleMoves(currentCell);
            if (possibleMoves.isEmpty()) {
                // if no possible moves at this cell, pop off current cell
from stack and repeat for previous cell
                solutionPathStack.pop();
                currentCell = solutionPathStack.peek();
                //System.out.println("Stack size reduced by one. New Size: "
+ solutionPathStack.size());
            } else {
                // else choose the best possible move, and add the new cell
to the stack and redo loop
                currentCell = computeBestMove(possibleMoves, goalCell);
                solutionPathStack.push(currentCell);
                //System.out.println("X: " + currentCell.getX() + "; Y: " +
currentCell.getY());
                //System.out.println("Stack size increased by one. New Size:
" + solutionPathStack.size());
            }
        }
    }
```

```
    public void setGoalCell(gridGraph.cell inputCell) {
        // method sets the attribute for the goal cell
        this.goalCell = inputCell;
    }
    public gridGraph.cell getGoalCell() {
        // returns the cell object of the goal as provided in the input
        return this.goalCell;
    }
    public void setCurrentCell(gridGraph.cell inputCell) {
        // method to set the current cell attribute
        this.solutionPathStack.push(inputCell);
    }
    public gridGraph.cell getCurrentCell() {
        // returns the current cell on which the AI agent is acting
        return this.solutionPathStack.peekLast();
    }
    public void setPreviousCell(gridGraph.cell inputCell) {
        // method to keep track of previous cell. Adds the inputCell to the
top of the solutionPathStack
        this.solutionPathStack.push(inputCell);
    }
    public gridGraph.cell getPreviousCell() {
        // method pops the top cell off the solutionPath stack
        return this.solutionPathStack.pop();
    }
    public LinkedList<gridGraph.cell> getSolutionPath() {
        // method returns the list of cells that compose the solution path
from start to finish
        return this.solutionPathStack;
    }
    // private methods
    private void determinePossibleMoves(gridGraph.cell inputCell) {
        // this method takes a cell as input, and adds the cells that are
possible to move to to the possibleMoves array
        // clear the array first, such that there are no other cells present
        this.possibleMoves.clear();
        // examine each wall and add the neighboring cell to the possible
moves list if the wall is a passage
        if (inputCell.getTopWall().isPassage()) {
            gridGraph.cell topNeighbor = inputCell.getNeighbors()[0];
            if (topNeighbor.getVisitCount() == 0) {
                this.possibleMoves.add(topNeighbor);
            }
        }
        if (inputCell.getRightWall().isPassage()) {
            gridGraph.cell rightNeighbor = inputCell.getNeighbors()[1];
            if (rightNeighbor.getVisitCount() == 0) {
                this.possibleMoves.add(rightNeighbor);
            }
        }
        if (inputCell.getBottomWall().isPassage()) {
            gridGraph.cell bottomNeighbor = inputCell.getNeighbors()[2];
            if (bottomNeighbor.getVisitCount() == 0) {
                this.possibleMoves.add(bottomNeighbor);
            }
        }
```

```
        if (inputCell.getLeftWall().isPassage()) {
            gridGraph.cell leftNeighbor = inputCell.getNeighbors()[3];
            if (leftNeighbor.getVisitCount() == 0) {
                this.possibleMoves.add(leftNeighbor);
            }
        }
    }
    private gridGraph.cell computeBestMove(ArrayList<gridGraph.cell>
inputCellList, gridGraph.cell goalCell) {
        // this method takes the current possible moves list, and the goal
cell as inputs, and determines which cell
        // should be used next in the path
        int goalXpos = goalCell.getX();
        int goalYpos = goalCell.getY();
        int outputIndex = -1;
        gridGraph.cell outputCell = null;
        double shortestDistance = -1.0;
        // find the index of the cell with the shortest straight line
distance to goal
        for (int listIndex = 0; listIndex < inputCellList.size();
listIndex++) {
            // get the current (x,y) coordinates of neighbor cell
            int currentXpos = inputCellList.get(listIndex).getX();
            int currentYpos = inputCellList.get(listIndex).getY();
            // compute the pythagorean distance between the current cell and
the goal cell
            double radicand = Math.pow((goalXpos - currentXpos),2) +
Math.pow((goalYpos - currentYpos),2);
            double distance = Math.sqrt(radicand);
            if (shortestDistance == -1.0) {
                // this is the first cell, and we set the shortest distance
as the distance
                shortestDistance = distance;
                outputIndex = listIndex;
            } else if (distance <= shortestDistance) {
                // the newly computed distance is shorter, so set the
shortest distance and the output index
                shortestDistance = distance;
                outputIndex = listIndex;
            }
        }
        if (outputIndex == -1) {
            return null;
        } else {
            outputCell = inputCellList.get(outputIndex);
            return outputCell;
        }
    }
}
```

gridGraph.java

```
package assignment01;

import java.util.ArrayList;

public class gridGraph {

    // Attributes
    ArrayList<cell> cells = new ArrayList<>();
    ArrayList<wall> walls = new ArrayList<>();

    // Constructor Methods
    gridGraph(int x, int y) {
        // constructor method that creates the data structure
        generateGraphStructure(x, y);
    }
    // Public Methods
    public cell getCell(int index) {
        // returns a cell object given a particular index on the cells
ArrayList.
        return cells.get(index);
    }
    public int getCellsSize() {
        // returns the cardinality of the cells set
        return cells.size();
    }
    public wall getWall(int index) {
        // returns a wall object given a particular index on the walls
ArrayList
        return walls.get(index);
    }
    public int getWallsSize() {
        // returns the cardinality of the walls set
        return walls.size();
    }
    // Private Methods
    private void generateGraphStructure(int xSize, int ySize) {
        /* this method takes a 2D size parameter (as two separate integer
values) and populates the data structure with
            the following arrangement:
                Each "pixel" in the graph structure is a cell. Each cell has
at most four walls. To generate every
                "pixel" we must iterate over the size of the canvas given as
inputs to this method.
         */
        for (int yIndex = 1; yIndex <= ySize; yIndex++) {
            for (int xIndex = 1; xIndex <= xSize; xIndex++) {
                int[] currentPosition = {xIndex, yIndex};
                cells.add(new cell(currentPosition));
            }
        }
        /*
            The next step in the creation is to generate all the appropriate
walls in the grid. There are four walls per
            cell, however the method will place only the right and bottom
walls by default. This leaves the outside and
```

```
            corner cells requiring special consideration. This arrangement
will avoid creation of duplicates and will
            prevent needing to run a search for a particular wall to avoid
duplicates. Additionally, the adjacency
            lists for each cell are created. This is done through the
cell.addNeighbors(cell c) method.
         */
            for (int listIndex = 0; listIndex < cells.size(); listIndex++) {
                // get the current cell from list
                cell workingCell = cells.get(listIndex);

                // get working cell's position on the grid
                int cellX = workingCell.getX();
                int cellY = workingCell.getY();

                // add walls and neighbors to working cell
                // add top wall and neighbor
                if (cellY == 1) {
                    wall workingTop = new wall(workingCell, null);
                    walls.add(workingTop);
                    workingCell.setTopWall(workingTop);
                    workingCell.setTopNeighbor(null);
                } else {
                    wall workingTop = new wall(workingCell,
cells.get(listIndex - xSize));
                    if (walls.contains(workingTop)) {

workingCell.setTopWall(walls.get(walls.indexOf(workingTop)));
                    } else {
                        workingCell.setTopWall(workingTop);
                        walls.add(workingTop);
                    }
                    workingCell.setTopNeighbor(cells.get(listIndex - xSize));
                }
                // add right wall and neighbor
                if (cellX == xSize) {
                    wall workingRight = new wall(workingCell, null);
                    walls.add(workingRight);
                    workingCell.setRightWall(workingRight);
                    workingCell.setRightNeighbor(null);
                } else {
                    wall workingRight = new wall(workingCell,
cells.get(listIndex + 1));
                    if (walls.contains(workingRight)) {

workingCell.setRightWall(walls.get(walls.indexOf(workingRight)));
                    } else {
                        workingCell.setRightWall(workingRight);
                        walls.add(workingRight);
                    }
                    workingCell.setRightNeighbor(cells.get(listIndex + 1));
                }
                // add bottom wall and neighbor
                if (cellY == ySize) {
                    wall workingBottom = new wall(workingCell, null);
                    walls.add(workingBottom);
                    workingCell.setBottomWall(workingBottom);
```

```
                    workingCell.setBottomNeighbor(null);
                } else {
                    wall workingBottom = new wall(workingCell,
cells.get(listIndex + xSize));
                    if (walls.contains(workingBottom)) {

workingCell.setBottomWall(walls.get(walls.indexOf(workingBottom)));
                    } else {
                        workingCell.setBottomWall(workingBottom);
                        walls.add(workingBottom);
                    }
                    workingCell.setBottomNeighbor(cells.get(listIndex +
xSize));
                }
                // add left wall and neighbor
                if (cellX ==1) {
                    wall workingLeft = new wall(workingCell, null);
                    walls.add(workingLeft);
                    workingCell.setLeftWall(workingLeft);
                    workingCell.setLeftNeighbor(null);
                } else {
                    wall workingLeft = new wall(workingCell,
cells.get(listIndex - 1));
                    if (walls.contains(workingLeft)) {

workingCell.setLeftWall(walls.get(walls.indexOf(workingLeft)));
                    } else {
                        workingCell.setLeftWall(workingLeft);
                        walls.add(workingLeft);
                    }
                    workingCell.setLeftNeighbor(cells.get(listIndex - 1));
                }
            }
        }
    // Internal Classes
    public class cell {
        // Attributes
        private int[] position;
        private int visitCount = 0;
        private wall topWall = null;
        private wall leftWall = null;
        private wall rightWall = null;
        private wall bottomWall = null;
        private final cell[] neighbors = new cell[4];
        // Methods
        cell(int[] coordinates) {
            // sets the coordinates based on the input array. Once set, this
cannot be changed from outside the object
            // scope.
            this.setPosition(coordinates);
        }
        private void setPosition(int[] orderedPair) {
            // this method is used solely in the constructor method as the
position of individual cells should not
            // change after creation
            this.position = orderedPair;
        }
```

```
        protected int getX() {
            // returns only the X coordinate of the cell
            return this.position[0];
        }
        protected int getY() {
            // returns only the Y coordinate of the cell
            return this.position[1];
        }
        protected void setTopWall(wall inputWall) {
            // method to place the top wall into its appropriate place
            this.topWall = inputWall;
        }
        protected wall getTopWall() {
            // returns the top wall object
            return this.topWall;
        }
        protected void setLeftWall(wall inputWall) {
            // method to place the left wall into its appropriate place
            this.leftWall = inputWall;
        }
        protected wall getLeftWall() {
            // returns the left wall object
            return this.leftWall;
        }
        protected void setRightWall(wall inputWall) {
            // method to place the right wall into its appropriate place
            this.rightWall = inputWall;
        }
        protected wall getRightWall() {
            // returns the right wall------- object
            return this.rightWall;
        }
        protected void setBottomWall(wall inputWall) {
            // method to place the bottom wall into its appropriate place
            this.bottomWall = inputWall;
        }
        protected wall getBottomWall() {
            // returns the bottom wall object
            return this.bottomWall;
        }
        protected void visit() {
            // increases the visit count by 1 every time the method is called
            this.visitCount++;
        }
        protected int getVisitCount() {
            // returns the visit count when called
            return this.visitCount;
        }
        protected void setTopNeighbor(cell c) {
            // this method takes a cell as input and adds as the top neighbor
            this.neighbors[0] = c;
        }
        protected void setRightNeighbor(cell c) {
            // this method takes a cell as input and adds as the right
neighbor
            this.neighbors[1] = c;
        }
```

```
        protected void setBottomNeighbor(cell c) {
            // this method takes a cell as input and adds as the bottom
neighbor
            this.neighbors[2] = c;
        }
        protected void setLeftNeighbor(cell c) {
            // this method takes a cell as input and adds as the left
neighbor
            this.neighbors[3] = c;
        }
        protected cell[] getNeighbors() {
            // returns the array list of neighboring cells
            // The array is of format:
            //  0 = top
            //  1 = right
            //  2 = bottom
            //  3 = left
            return this.neighbors;
        }
        @Override
        public boolean equals(Object obj) {
            if (!(obj instanceof cell)) {
                return false;
            } else {
                cell compareCell = (cell) obj;
                return (this.getX() == compareCell.getX() && this.getY() ==
compareCell.getY());
            }
        }
    }
    public class wall {
        // Attributes
        private final cell cellOne;
        private final cell cellTwo;
        private boolean passage = false;
        // Methods
        wall(cell cellOne, cell cellTwo) {
            // constructor method assigns each cell to an end of an edge.
These two cells exist on each side of the
            // "wall" and are divided by this object
            this.cellOne = cellOne;
            this.cellTwo = cellTwo;
        }
        public cell getCellOne() {
            // returns the first cell in the edge (or wall, such as it is)
            return this.cellOne;
        }
        public cell getCellTwo() {
            // returns the second cell in the edge (or wall)
            return this.cellTwo;
        }
        public void setPassage(boolean tf) {
            // sets the passage parameter in the wall, if the wall is meant
to be "knocked down" use this to specify
            this.passage = tf;
        }
        public boolean isPassage() {
```

```
            // returns the value that determines whether this edge is a
passage or not. The default is 'false' which
            // indicates this wall is non-passable.
            return this.passage;
        }
        @Override
        public boolean equals(Object obj) {
            if (!(obj instanceof wall)) {
                return false;
            } else {
                return ((this.getCellOne().equals(((wall)obj).getCellOne())
|| this.getCellOne().equals(((wall)obj).getCellTwo())) &&
                        (this.getCellTwo().equals(((wall)obj).getCellOne())
|| this.getCellTwo().equals(((wall)obj).getCellTwo())));
            }
        }
    }
}
```

mainController.java

```
package assignment01;

import javafx.fxml.Initializable;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.Button;
import javafx.scene.control.MenuItem;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import java.net.URL;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Random;
import java.util.ResourceBundle;

public class mainController implements Initializable {
    // GUI Objects
    public BorderPane borderPane;
    public Canvas centerCanvas;
    public TextField notificationText;
    public TextField dataTextField;
    public Button generateMazeButton;
    public Button clearMazeButton;
    public Button solveMazeButton;
    public Button setStartFinishButton;
    public MenuItem generateMazeMenuButton;
    public MenuItem clearMazeMenuButton;
    public MenuItem solveMazeMenuButton;
    public MenuItem setStartFinishMenuButton;
    public MenuItem aboutMenuButton;
    public GraphicsContext canvasGc;
    // private attributes
    private int pixelSize;
    private gridGraph graph;
    private ArrayList<gridGraph.cell> mazePath = new ArrayList<>();
    private LinkedList<gridGraph.cell> solutionPath = new LinkedList<>();
    private gridGraph.cell startCell;
    private gridGraph.cell goalCell;
    private String dataString = "No maze present.";

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        // get the Graphics Context of the center canvas
        canvasGc = centerCanvas.getGraphicsContext2D();
        // draw the grid upon startup with default pixel size equal to 20
pixels
        setPixelSize20();
        drawOutline(canvasGc);
        // set initial UI configuration
        updateDataTextArea(dataString);
        clearMazeButton.setDisable(true);
        clearMazeMenuButton.setDisable(true);
        solveMazeButton.setDisable(true);
```

```
        solveMazeMenuButton.setDisable(true);
        setStartFinishButton.setDisable(true);
        setStartFinishMenuButton.setDisable(true);
    }
    // Public Event Handling Methods
    public void generateMaze() {
        // method used to generate the maze object and display it on the
mainCanvas object
        // method variables
        int canvasWidth = (int)canvasGc.getCanvas().getWidth();
        int canvasHeight = (int)canvasGc.getCanvas().getHeight();
        int gridWidth = canvasWidth/this.pixelSize;
        int gridHeight = canvasHeight/this.pixelSize;
        ArrayList<gridGraph.wall> wallList = new ArrayList<>();
        // Start with a grid full of walls.
        graph = new gridGraph(gridWidth, gridHeight);
        // pick a random cell and add it to the maze and add walls to wall
list
        // for this, we want the start to be on the left hand row, so will
need to pick a cell w/ xPos == 1
        Random random = new Random();
        int startCellXpos = random.nextInt(gridHeight);
        gridGraph.cell startCell = graph.getCell((startCellXpos *
gridWidth));
        startCell.visit();
        mazePath.add(startCell);
        addWalls(startCell, wallList);
        // while there are walls left in the list
        while (!(wallList.isEmpty())) {
            // pick a random wall from the list
            Random rand = new Random();
            int choice = rand.nextInt(wallList.size());
            gridGraph.wall workingWall = wallList.get(choice);
            // get cells adjacent to wall
            gridGraph.cell cellOne = workingWall.getCellOne();
            gridGraph.cell cellTwo = workingWall.getCellTwo();
            // if adjacent cell is in the path, add the other to the path and
mark wall as passage
            if (!(mazePath.contains(cellTwo))) {
                mazePath.add(cellTwo);
                workingWall.setPassage(true);
                // add new cell's walls to wall list
                addWalls(cellTwo, wallList);
                wallList.remove(workingWall);
            } else if (!(mazePath.contains(cellOne))) {
                mazePath.add(cellOne);
                workingWall.setPassage(true);
                addWalls(cellOne, wallList);
                wallList.remove(workingWall);
            } else {
                // as both cells are already in the path, remove wall from
list
                wallList.remove(workingWall);
            }
        }
        // draw the actual maze in the GUI and give notification maze has
been generated
```

```
        updateNotificationArea("Maze successfully generated with a grid unit
size of " + this.pixelSize + " pixels.");
        dataString = "Size = " + this.pixelSize;
        updateDataTextArea(dataString);
        drawMaze(canvasGc, mazePath);
        drawOutline(canvasGc);
        // change UI configuration
        setStartFinishButton.setDisable(false);
        setStartFinishMenuButton.setDisable(false);
        clearMazeButton.setDisable(false);
        clearMazeMenuButton.setDisable(false);
        generateMazeButton.setDisable(true);
        generateMazeMenuButton.setDisable(true);
    }
    public void clearMaze() {
        // method used to reset the application to its initial state

canvasGc.clearRect(0,0,centerCanvas.getWidth(),centerCanvas.getHeight());
        mazePath.clear();
        drawOutline(canvasGc);
        updateNotificationArea("Maze cleared");
        dataString = "No maze present.";
        updateDataTextArea(dataString);
        // change UI configuration
        setStartFinishButton.setDisable(true);
        setStartFinishMenuButton.setDisable(true);
        clearMazeButton.setDisable(true);
        clearMazeMenuButton.setDisable(true);
        solveMazeButton.setDisable(true);
        solveMazeMenuButton.setDisable(true);
        generateMazeMenuButton.setDisable(false);
        generateMazeButton.setDisable(false);
        generateMazeButton.requestFocus();
    }
    public void solveMaze() throws NullPointerException {
        // clear the canvas
        try {
            canvasGc.clearRect(0, 0, centerCanvas.getWidth(),
centerCanvas.getHeight());
            // this method invokes the AI agent that will solve the maze
            aiAgent solver = new aiAgent(this.startCell, this.goalCell);
            solver.solveMaze();
            this.solutionPath = solver.getSolutionPath();
            updateNotificationArea("The maze has been solved");
            for (int drawIndex = 0; drawIndex < this.solutionPath.size();
drawIndex++) {
                gridGraph.cell drawnCell = solutionPath.get(drawIndex);
                int xPos = drawnCell.getX();
                int yPos = drawnCell.getY();
                drawPixel(canvasGc, xPos, yPos, "light blue");
            }
            drawPixel(canvasGc, startCell.getX(), startCell.getY(), "green");
            drawPixel(canvasGc, goalCell.getX(), goalCell.getY(), "red");
            // redraw the grid above the path plots
            drawMaze(canvasGc, mazePath);
            drawOutline(canvasGc);
            // change UI configuration
```

```
            solveMazeButton.setDisable(true);
            solveMazeMenuButton.setDisable(true);
            clearMazeButton.setDisable(false);
            clearMazeMenuButton.setDisable(false);
        } catch (NullPointerException ex) {
            updateNotificationArea("An error occurred. Clear the maze and
start over.");
            clearMazeButton.setDisable(false);
            clearMazeMenuButton.setDisable(false);
            solveMazeButton.setDisable(true);
            solveMazeMenuButton.setDisable(true);
        }
    }
    public void setStartFinishCells() {
        // this method sets the start and goal cells from the gridGraph
object that are then used to solve the maze.
        Random startChoice = new Random();
        Random endChoice = new Random();
        this.startCell = mazePath.get(startChoice.nextInt(mazePath.size()));
        this.goalCell = mazePath.get(endChoice.nextInt(mazePath.size()));
        drawPixel(canvasGc, startCell.getX(), startCell.getY(), "green");
        drawPixel(canvasGc, goalCell.getX(), goalCell.getY(), "red");
        String formerString = dataString;
        dataString = "Start: (" + startCell.getX() + ", " + startCell.getY()
+
                "), End: (" + goalCell.getX() + ", " + goalCell.getY() + "),
"
                + formerString;
        updateDataTextArea(dataString);
        // change UI configuration
        setStartFinishButton.setDisable(true);
        setStartFinishMenuButton.setDisable(true);
        solveMazeButton.setDisable(false);
        solveMazeMenuButton.setDisable(false);
    }
    public void showAbout() throws Exception {
        // this method calls the about window and displays the result
        aboutController about = new aboutController();
        about.showWindow();
    }
    public void closeProgram() {
        // this method ensure the program closes appropriately
        Stage activeStage = (Stage) this.borderPane.getScene().getWindow();
        activeStage.close();
    }
    public void setPixelSize5() {
        // used as part of the radio menu selector to set the appropriate
pixel size
        this.pixelSize = 5;
        clearMaze();
        //drawGrid(canvasGc);
    }
    public void setPixelSize10() {
        // used as part of the radio menu selector to set the appropriate
pixel size
        this.pixelSize = 10;
        clearMaze();
```

```
        //drawGrid(canvasGc);
    }
    public void setPixelSize20() {
        // used as part of the radio menu selector to set the appropriate
pixel size
        this.pixelSize = 20;
        clearMaze();
        //drawGrid(canvasGc);
    }
    // Private Methods
    private void updateNotificationArea(String notification) {
        // this method takes a string as input and displays it in the
notification text field
        notificationText.setText(notification);
    }
    private void updateDataTextArea(String data) {
        // method takes a string as input and displays it in the data text
field
        dataTextField.setText(data);
    }
    private void addWalls(gridGraph.cell inputCell, ArrayList<gridGraph.wall>
inputList) {
        // this method takes a cell and a list and adds all walls that are
not already on the list
        if (!(inputCell.getTopWall().getCellTwo() == null)) {
            gridGraph.wall workingWall = inputCell.getTopWall();
            if (!(inputList.contains(workingWall))) {
                inputList.add(workingWall);
            } else {
                // do nothing
            }
        }
        if (!(inputCell.getRightWall().getCellTwo() == null)) {
            gridGraph.wall workingWall = inputCell.getRightWall();
            if (!(inputList.contains(workingWall))) {
                inputList.add(workingWall);
            } else {
                // do nothing
            }
        }
        if (!(inputCell.getBottomWall().getCellTwo() == null)) {
            gridGraph.wall workingWall = inputCell.getBottomWall();
            if (!(inputList.contains(workingWall))) {
                inputList.add(workingWall);
            } else {
                // do nothing
            }
        }
        if (!(inputCell.getLeftWall().getCellTwo() == null)) {
            gridGraph.wall workingWall = inputCell.getLeftWall();
            if (!(inputList.contains(workingWall))) {
                inputList.add(workingWall);
            } else {
                // do nothing
            }
        }
    }
```

```
    private void drawMaze(GraphicsContext contextInput,
ArrayList<gridGraph.cell> inputMaze) {
        // draws a grid with cell size in pixels, size of pixel can be
changed with the parameters given below
        ArrayList<gridGraph.wall> drawnWalls = new ArrayList<>();
        // iterate over each cell in mazePath array
        for (int cellIndex = 0; cellIndex < inputMaze.size(); cellIndex++) {
            // get the particular cell from the mazePath array
            gridGraph.cell workingCell = inputMaze.get(cellIndex);
            // iterate over each wall in the cell
            // top wall
            if (workingCell.getTopWall().getCellTwo() != null &&
!(workingCell.getTopWall().isPassage())) {
                if (!(drawnWalls.contains(workingCell.getTopWall()))) {
                    // draw the top wall line
                    drawGridLine(contextInput, workingCell, "top");
                    // add wall to drawn wall list
                    drawnWalls.add(workingCell.getTopWall());
                }
            }
            // right wall
            if (workingCell.getRightWall().getCellTwo() != null &&
!(workingCell.getRightWall().isPassage())) {
                if (!(drawnWalls.contains(workingCell.getRightWall()))) {
                    drawGridLine(contextInput, workingCell, "right");
                    drawnWalls.add(workingCell.getRightWall());
                }
            }
            // bottom wall
            if (workingCell.getBottomWall().getCellTwo() != null &&
!(workingCell.getBottomWall().isPassage())) {
                if (!(drawnWalls.contains(workingCell.getBottomWall()))) {
                    drawGridLine(contextInput, workingCell, "bottom");
                    drawnWalls.add(workingCell.getBottomWall());
                }
            }
            // left wall
            if (workingCell.getLeftWall().getCellTwo() != null &&
!(workingCell.getLeftWall().isPassage())) {
                if (!(drawnWalls.contains(workingCell.getLeftWall()))) {
                    drawGridLine(contextInput, workingCell, "left");
                    drawnWalls.add(workingCell.getLeftWall());
                }
            }
        }
    }
    private void drawGridLine(GraphicsContext inputContext, gridGraph.cell
inputCell, String direction) {
        // method draws a line according to the location of the wall given an
input cell and direction string
        // set drawing parameters
        int pixelSize = this.pixelSize;
        int gridXpos = inputCell.getX()-1;
        int gridYpos = inputCell.getY()-1;
        inputContext.setLineWidth(1.0);
        inputContext.setStroke(Color.BLACK);
        // define the relative coordinates of each corner
```

```
        int topLeftXpos = gridXpos * pixelSize;
        int topLeftYpos = gridYpos * pixelSize;
        int topRightXpos = topLeftXpos + pixelSize;
        int topRightYpos = topLeftYpos;
        int bottomLeftXpos = topLeftXpos;
        int bottomLeftYpos = topLeftYpos + pixelSize;
        int bottomRightXpos = topRightXpos;
        int bottomRightYpos = bottomLeftYpos;
        // draw the actual lines on the canvas given the appropriate
direction
        if (direction.equalsIgnoreCase("top")) {
            inputContext.strokeLine(topLeftXpos, topLeftYpos, topRightXpos,
topRightYpos);
        }
        if (direction.equalsIgnoreCase("right")) {
            inputContext.strokeLine(topRightXpos, topRightYpos,
bottomRightXpos, bottomRightYpos);
        }
        if (direction.equalsIgnoreCase("bottom")) {
            inputContext.strokeLine(bottomRightXpos, bottomRightYpos,
bottomLeftXpos, bottomLeftYpos);
        }
        if (direction.equalsIgnoreCase("left")) {
            inputContext.strokeLine(bottomLeftXpos, bottomLeftYpos,
topLeftXpos, topLeftYpos);
        }
    }
    private void drawOutline(GraphicsContext context) {
        // method draws an outline around the entire canvas
        // Set the stroke color
        context.setLineWidth(3.0);
        context.setStroke(Color.BLACK);
        // draw lines around entire canvas
        context.strokeLine(0, 0, context.getCanvas().getWidth(), 0);
        context.strokeLine(0,0,0, context.getCanvas().getHeight());
        context.strokeLine(context.getCanvas().getWidth(), 0,
context.getCanvas().getWidth(),
                context.getCanvas().getHeight());
        context.strokeLine(0, context.getCanvas().getHeight(),
context.getCanvas().getWidth(),
                context.getCanvas().getHeight());
    }
    private void drawPixel(GraphicsContext contextInput, int x, int y, String
color) {
        // creates a pixel that is then drawn onto the particular canvas.
Pixel size and color can be defined below.
        // Set the color of the pixel
        if (color.equalsIgnoreCase("blue")){
            contextInput.setFill(Color.BLUE);
        } else if (color.equalsIgnoreCase("red")) {
            contextInput.setFill(Color.RED);
        } else if (color.equalsIgnoreCase("green")) {
            contextInput.setFill(Color.GREEN);
        } else if (color.equalsIgnoreCase("light blue")) {
            contextInput.setFill(Color.LIGHTBLUE);
        } else {
            contextInput.setFill(Color.BLACK);
```

```
        }
        // Define the Size of the pixel
        int pixelHeight = this.pixelSize;
        int pixelWidth = this.pixelSize;
        // Define the maximum dimensions of the intended canvas
        double verticalSize = contextInput.getCanvas().getHeight();
        double horizontalSize = contextInput.getCanvas().getWidth();
        // Determine location of top right hand corner of pixel from input
(X,Y)
        int canvasXcoord = (x-1) * pixelWidth;
        int canvasYcoord = (y-1) * pixelHeight;
        // Display error if computed coordinate goes beyond the canvas
dimensions
        if ((canvasXcoord > horizontalSize) || (canvasYcoord > verticalSize))
{
            System.out.println("The computed coordinate is beyond the
canvas.");
        }
        // Write the actual "pixel" to the canvas

contextInput.fillRect(canvasXcoord,canvasYcoord,pixelWidth,pixelHeight);
    }
}
```

aboutController.java

```
package assignment01;

import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class aboutController {
    // Initialize GUI Elements
    public Button closeAboutButton;
    public Stage activeStage = new Stage();
    // Public Methods
    public void showWindow() throws Exception {
        Parent root =
FXMLLoader.load(getClass().getResource("aboutWindow.fxml"));
        activeStage.setTitle("About");
        activeStage.setScene(new Scene(root, 350, 250));
        activeStage.setResizable(false);
        activeStage.show();
    }
    public void closeWindow() {
        Stage currentStage = (Stage) closeAboutButton.getScene().getWindow();
        currentStage.close();
    }
}
```

mainWindow.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.canvas.Canvas?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Menu?>
<?import javafx.scene.control.MenuBar?>
<?import javafx.scene.control.MenuItem?>
<?import javafx.scene.control.RadioMenuItem?>
<?import javafx.scene.control.SeparatorMenuItem?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.control.ToggleGroup?>
<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.layout.VBox?>

<BorderPane fx:id="borderPane" maxHeight="-Infinity" maxWidth="-Infinity"
minHeight="-Infinity" minWidth="-Infinity" prefHeight="600.0"
prefWidth="600.0" xmlns="http://javafx.com/javafx/15.0.1"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="assignment01.mainController">
   <right>
      <VBox BorderPane.alignment="CENTER" />
   </right>
   <left>
      <VBox BorderPane.alignment="CENTER" />
   </left>
   <top>
      <MenuBar BorderPane.alignment="CENTER">
        <menus>
          <Menu mnemonicParsing="false" text="File">
            <items>
              <MenuItem fx:id="closeMenuButton" mnemonicParsing="false"
onAction="#closeProgram" text="Close" />
            </items>
          </Menu>
            <Menu mnemonicParsing="false" text="Actions">
              <items>
                <MenuItem fx:id="generateMazeMenuButton"
mnemonicParsing="false" onAction="#generateMaze" text="Generate Maze" />
                  <MenuItem fx:id="setStartFinishMenuButton"
mnemonicParsing="false" onAction="#setStartFinishCells" text="Set Start and
Finish" />
                  <MenuItem fx:id="solveMazeMenuButton"
mnemonicParsing="false" onAction="#solveMaze" text="Solve Maze" />
                  <MenuItem fx:id="clearMazeMenuButton"
mnemonicParsing="false" onAction="#clearMaze" text="Clear Maze" />
                  <SeparatorMenuItem mnemonicParsing="false" />
                  <Menu mnemonicParsing="false" text="Set Pixel Size">
                    <items>
                        <RadioMenuItem fx:id="pixelSize5"
mnemonicParsing="false" onAction="#setPixelSize5" text="5">
                           <toggleGroup>
                              <ToggleGroup fx:id="pixelSizeGroup" />
                           </toggleGroup>
```

```
                        </RadioMenuItem>
                        <RadioMenuItem fx:id="pixelSize10"
mnemonicParsing="false" onAction="#setPixelSize10" text="10"
toggleGroup="$pixelSizeGroup" />
                        <RadioMenuItem fx:id="pixelSize20"
mnemonicParsing="false" onAction="#setPixelSize20" selected="true" text="20"
toggleGroup="$pixelSizeGroup" />
                     </items>
                  </Menu>
               </items>
            </Menu>
            <Menu mnemonicParsing="false" text="Help">
               <items>
                  <MenuItem fx:id="aboutMenuButton" mnemonicParsing="false"
onAction="#showAbout" text="About" />
               </items>
            </Menu>
         </menus>
      </MenuBar>
   </top>
   <center>
      <Canvas fx:id="centerCanvas" height="500.0" width="500.0"
BorderPane.alignment="CENTER" />
   </center>
   <bottom>
      <VBox BorderPane.alignment="CENTER">
         <children>
            <HBox alignment="TOP_CENTER" spacing="5.0">
               <children>
                  <Button fx:id="generateMazeButton" mnemonicParsing="false"
onAction="#generateMaze" text="Generate Maze" />
                  <Button fx:id="setStartFinishButton"
mnemonicParsing="false" onAction="#setStartFinishCells" text="Set Start and
Finish" />
                  <Button fx:id="solveMazeButton" mnemonicParsing="false"
onAction="#solveMaze" text="Solve Maze" />
                  <Button fx:id="clearMazeButton" mnemonicParsing="false"
onAction="#clearMaze" text="Clear Maze" />
               </children>
               <padding>
                  <Insets bottom="5.0" left="5.0" right="5.0" top="5.0" />
               </padding>
            </HBox>
            <HBox>
               <children>
                  <TextField fx:id="dataTextField" editable="false"
HBox.hgrow="ALWAYS">
                     <HBox.margin>
                        <Insets left="5.0" right="5.0" />
                     </HBox.margin>
                  </TextField>
                  <TextField fx:id="notificationText"
alignment="CENTER_RIGHT" editable="false" text="Must first click
&amp;quot;Generate Maze&amp;quot;" HBox.hgrow="ALWAYS">
                     <HBox.margin>
                        <Insets left="5.0" right="5.0" />
                     </HBox.margin></TextField>
```

```
               </children>
               <VBox.margin>
                  <Insets bottom="5.0" />
               </VBox.margin>
            </HBox>
         </children>
      </VBox>
   </bottom>
</BorderPane>
```

aboutWindow.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.text.Font?>
<?import javafx.scene.text.Text?>

<Pane fx:id="aboutPane" maxHeight="-Infinity" maxWidth="-Infinity"
minHeight="-Infinity" minWidth="-Infinity" prefHeight="250.0"
prefWidth="350.0" xmlns="http://javafx.com/javafx/15.0.1"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="assignment01.aboutController">
   <children>
      <Button fx:id="closeAboutButton" layoutX="148.0" layoutY="209.0"
mnemonicParsing="false" onAction="#closeWindow" text="Close" />
      <Text layoutX="52.0" layoutY="71.0" strokeType="OUTSIDE"
strokeWidth="0.0" text="CS 7375 - Assignment 1">
         <font>
            <Font size="24.0" />
         </font>
      </Text>
      <Text layoutX="120.0" layoutY="39.0" strokeType="OUTSIDE"
strokeWidth="0.0" text="Marion Garrett Sisk" />
      <Text layoutX="48.0" layoutY="108.0" strokeType="OUTSIDE"
strokeWidth="0.0" text="The maze generator uses a &#10;Randomized Prim's
Algorithm implementation. &#10;The AI Agent is designed to navigate&#10;the
generated maze. It can find a solution,&#10;but may not be the most efficient
means." textAlignment="CENTER" />
   </children>
</Pane>
```