## 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.

Beginning with the provided start cell, the agent determines which directions are passable, and which are blocked by a wall or edge of the maze. Once the possible moves have been determined, each move has a computed distance from the goal. This distance is the straight line, or Pythagorean, distance between the intended next cell and the goal cell. The agent will then move to the cell with the lowest value and repeat the process until reaching the goal cell.

The pseudocode for how the agent accomplishes this is given below in Figure 1.**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;  
 }  
 }  
}**