I. Labyrinth Game Rules

Labyrinth is a web-based implementation of a popular board game by the same name. It was developed by the Computer Science department of the Rochester Institute of Technology and was funded by the NSF grant #1044721. The game is a one to four player game, played on a seven by seven tile board. Board tiles combine to form maze corridors. Each player has a pawn of a certain color. Players start out positioned on the corner tiles of the board corresponding to the color of their pawn and make moves in a round robin fashion. The game has an extra tile, which is used to alter the maze by inserting it into one of the designated insertion points in any orientation. Insertion of the extra tile shifts the entire row or column of the board. The tile that is shifted out of the board becomes the next insertion tile. There are three insertion points per edge of the board, for a total of twelve insertion points. The point where a tile is shifted out becomes ineligible as the next insertion point, to prevent circular moves. Thus, all subsequent turns have eleven possible insertion points. Some of the tiles also have treasures on them in addition to being a part of the maze. Each player also starts out with a six treasure card pile, representing treasures that they need to collect by visiting the corresponding tiles on the board. Only the top treasure card of each player is seen. Once a player visits a tile with that treasure, the top treasure card is removed from that player's treasure card pile and the next treasure card is revealed. A move consists of two parts. First, the player inserts the extra tile in such a way as to make it easier for them to reach their next treasure tile or to block the opponents from reaching their treasure tiles. Second, the player moves to any other tile to which it has a connecting pathway. The player can stay of the same tile if they so choose. The player that collects all of their treasures and returns to their starting position first, wins.

II. Independent Study Goals

The Labyrinth game developed by the Computer Science department of the Rochester Institute of Technology has a human player implementation only. Students in certain CS courses at RIT are required to implement an Artificial Intelligence player for this game. However, it is somewhat difficult to test the work of these students since there is no AI player against which the students' AI players can be benchmarked. Additionally, the existing Labyrinth game engine itself needs to be tested better. The aim of this independent study is therefore to implement an AI player for the Labyrinth game in order to address these concerns.

III. AI Player Strategy

a. Initial Strategy

The first fully functional AI player that was developed as part of this independent study performed random valid moves. A randomly oriented extra tile would be inserted into one of the randomly selected valid insertion locations. The player's pawn would be moved to a neighboring tile (up, down, left, right) randomly selected from all neighboring tiles to which a connecting labyrinth path exists. While this was a fully functional player, this player was very easy to beat by a human, and when two such players were pitted against one another, the game could go on for over 1,000 moves before there would finally be a winner.

b. Offense Without Defense

In the next iteration of the AI player, all insertion locations and extra tile orientations were considered. For each insertion, best reachable tiles were found, where best is defined as closest to the goal tile based on Manhattan distance. The goal tile is either a tile with the next treasure that the player needs to collect or the player's home tile, if the player already collected all of their treasures. Manhattan distance is simply the sum of horizontal and vertical distances between two coordinates. If multiple equally good reachable tiles were found for a particular insertion, one was chosen at random. Similarly, if multiple insertions produced equally good best reachable tiles, one was chosen at random.

This iteration of the AI player was quite good offensively. There was usually a winner within 20 or so moves:

However, there were a couple issues with this AI player. When multiple equally good moves existed for the player, a random one was chosen, instead of choosing the move that would put the next opponent in the worst situation. Also, sometimes the pawn would get stuck for a while. This would occur because the AI player was always trying to minimize the Manhattan distance between its pawn and its next goal. Sometimes the player may need to move *away* from its next goal, so that it could be in a better position to actually reach the goal tile during its next move. Simple reliance on minimum Manhattan distance during a single move would not allow the player to "see" this. This iteration of the player was also tested by an external subject who also found the AI player to be quite good offensively, but also observed the opponent getting stuck as described above, allowing the tester to catch up and even beat the AI player frequently.

c. Good Offense with Some Defense

This evolution of the AI player saw the AI player starting to consider the next opponent's next move when calculating its next move. All insertion locations and extra tile orientations were still considered to find the best reachable tile for the player as described in the previous iteration of the player. However, *when multiple equally good offensive moves were found for the current player*, instead of choosing one at random, as was done previously, these moves were compared. The next opponent's best reachable tile was found considering the game boards resulting from each of the equally good offensive moves for the current player, using the same Manhattan distance measurement. The move causing the *largest* Manhattan distance between the opponent's best reachable location and opponent's goal tile was then chosen. This

improved the AI player's intelligence. However, the next opponent's insertions were still not being considered at this stage. Since a move consists of both an extra tile insertion *and* a pawn move, considering only the pawn move component of the opponent's total move would clearly not result in always finding the move that was actually worst for the opponent.

d. Offense with Better Defense

This was the final AI player developed as part of this study. It had an improved defensive strategy in that all of the next opponent's possible insertions were now being considered. Offense was still the priority over defense. As was the case before, if only one best move was found from the offensive perspective, that would be the move that was chosen. When the player found multiple equally good offensive moves, each of these moves were still being considered, as before. The next opponent's best reachable tile was found considering the game boards resulting from each of the equally good offensive moves for the current player. However, in this iteration of the AI player, the next opponent's full move, including all of their possible insertions, were considered when finding their best reachable tile. This caused this AI player to be able to accurately determine the set of next opponent's possible next moves, and, therefore, to pick the worst one for the next opponent from the equally good ones for the current player. It became increasingly hard to beat this AI player. Due to AI player's reliance on Manhattan distance, it would still sometimes become stuck for a while. This was really the only chance to beat this AI player. Two external subjects also played against this AI player and neither were able to beat it.

IV. Code Organization

a. AIPlayer

AIPlayer implements the PlayerModule interface. It is called by the Labyrinth engine to:

- Initialize the player
- Get the player's next move when it is the player's turn
- Update the player of the move that was made (including the player's own move)

It calls into the GameController to perform these functions.

b. GameController

GameController is initialized by the AIPlayer's init method. It maintains the following information:

- Board object
- Extra tile object

- Player id
- Next opponent's player id

It is the component responsible for finding the next best move and updating the board when the game engine notifies of a move that was made by any player.

c. Board

Board class maintains the following information:

- Double dimensional array of tile objects, representing the state of the board
- Set of coordinates representing the board locations which are valid for tile insertion at the time
- Coordinate object representing one of the tile insertion locations which is not valid for tile insertion at the time. This is because the location directly across the board from where the tile was inserted at most recently, is not valid as an insertion location for the next move to prevent undoing the previous player's move.
- Each player's home coordinate
- Each player's current pawn location
- Treasure piles of each player, which are represented as queues. Treasures are removed from the beginning of the respective players' queues as the players collect them.
- Current location of each treasure

It handles the function of inserting a tile, which shifts a row or a column of the double dimensional array and updates player and treasure locations and which locations are valid and invalid for tile insertion for the next move. It also handles the job of moving the pawn of each player.

d. Tile

Tile objects represent each tile of the game board. They hold the following information about each tile:

- Maze path type of the tile
- Maze path orientation of the tile
- Treasure type contained on the tile, if any
- Set of integers representing the player ids of the players currently located on the tile

Tile class also has a method to see if the tile has a maze path leading out in one of the four compass directions.

e. Enumerations

- Maze path type, which can be either ‘I’, ‘L’, or ‘T’
- Maze path orientation, which can be 0, 90, 180, or 270 degrees
- Treasure type, which is a value 0-23, or -1, to represent “no treasure”
- Compass direction, which can be north, east, south and west

V. Optimizations

a. Arrays as Look up Tables

As mentioned above, the Tile class has a method which determines if the tile has an exit in a particular direction. This method is called extensively when finding all other tiles that can be reached from the player’s current position. As this method is called extensively, efficiency is particularly important here. Initially, that method consisted of many switch statements, which switched on the maze path of the tile, the orientation of the tile and the compass direction. It was then changed to a hash set object, to see if there would be any improvement. The performance was much worse however, because even though hash sets have a lookup performance of O(1), the k constant is significant due to needing to hash the input strings, which eventually serve as keys into the hash set. Professor Heliotis suggested using a plain three dimensional array as a look up table, where the indices of the dimensions are represented by the maze type path, orientation of the tile and compass direction. This indeed was the most efficient solution and is found in the final implementation of the AI player.

b. Cloner Utility

The board and extra tile objects should not be modified as part of finding the best move method, as doing so would modify the state of the game. However, in that same method, it was necessary to consider all extra tile insertion combinations in order to pick the best one. Furthermore, in the later evolutions of the player, it was necessary to consider all insertion combinations of the next opponent’s next move, as described in the previous sections. As such, a copy of the board and extra tile objects were needed so as not to modify the existing board. To achieve that, the objects were serialized and then deserialized. While this worked to create copies of objects, it was inefficient. Since tile insertion was done many times per a single call to find the best move, efficiency here once again is particularly important. Professor Heliotis had previously created a Cloner utility for just such a purpose. While it is necessary to create ClonerCommand classes for all custom objects that need to be cloned, the utility handles cloning primitives, arrays and most of the Java collection types out of the box. Ten-fold and

hundred-fold improvements were seen after switching the AI player to use the Cloner utility for the board and extra tile objects, respectively.

c. Skipping Certain Tile Orientations

Maze path type of 'I' truly only has two orientations, as it is symmetrical vertically and horizontally. Therefore, only 0 and 90 degree orientations were considered whenever the extra tile was of maze path type 'I', cutting the time needed to find the best move in half.

VI. Engine Issues Found

- playerHomes is a list of coordinates, where each coordinate is the starting position of each player. It is passed to each player module by the engine during the player module initialization. It was found that the playerHomes list always contains four coordinates, even when the number of players is not four, and therefore cannot be used to determine the number of players playing the game.
- It was found that the data the engine is passing to the player modules is mixing both zero-based indexing (such as for treasure ids) and one-based indexing (such as for player ids). It would be preferable to maintain a single standard (although since the player id is displayed to the user, it is probably not desirable to have a "player zero").
- Tiles can be in one of four orientation positions. Since tiles are named after letters I, T and L, it would make sense if the upright position of each of these letters would be considered to have the orientation of zero degrees. This is only true for maze path type of 'I'. Instead, zero degree orientations of 'L' and 'T' maze path types are _| and -|, respectively, which is not intuitive.
- Maze path type of 'I' is symmetrical vertically and horizontally. Therefore, the engine should only need to use two orientations to represent its two orientation states. Instead the engine uses all four orientation types for the 'I' maze path type.

VII. Challenges and Future Work

a. Manhattan Distance Not Always Best

If there is no move that would allow the AI player to reach its next goal, it will always pick a move that will place its pawn as close as possible to its next goal, where the distance is measured as a Manhattan distance. As mentioned previously, this could cause the player to become stuck for a period of time. There are a couple possible solutions that could be implemented to alleviate this. The AI player could track several of its prior positions. If it detects that it has not moved in a few turns, it could force the row or the column that the player is on, or which is adjacent to the player, to shift. This

would be a relatively simple and possibly naïve remedy. As a more sophisticated solution, the AI player could project two or more of its own future moves, applying the Manhattan distance measurement on last move. The AI player is already projecting its own move and the next opponent's next move. Considering one more move, assuming it is a two player game, would help to eliminate the problem of getting stuck significantly.

b. Difficulty Levels

The AI player strategies described above were the actual progressions of the AI player as it was built up. The end product AI player is the last one listed. It would have been nice to have them all, each corresponding to a difficulty level. This could be especially helpful when grading students' AI player implementations. Students' AI players should have no trouble beating the AI player which performs random valid moves, for example.

Appendix A - Source Code:

AIPlayer.java

```
package Players.AIPlayer;

import Engine.Logger;
import Interface.Coordinate;
import Interface.PlayerModule;
import Interface.PlayerMove;

import java.util.List;
import java.util.Set;

/**
 * AIPlayer Player
 */
public class AIPlayer implements PlayerModule
{
   private Logger l;
   private int playerId;
   private GameController gameController;

   /**
    * Initializes the AIPlayer with the specified parameters
    *
    * @param logger - reference to the logger class
    * @param playerId- the id of this player
    * @param playerHomes - starting locations for each player, in order
    * @param treasures - ordered list of treasures for each player
    * @param board - 2-d list of [Tile ID, Rotation, Treasure]
    * Tile IDs:  0 = L tile, 1 = T tile, 2 = I tile
    * Rotations: 0 = 0 degrees, 1 = 90 degrees, 2 = 180 degrees,
    *            3 = 270 degrees, all clockwise
    * Treasures: -1 = no treasure, 0-23 = corresponding treasure
    * @param extra - contains [Extra Tile ID, Treasure]
    */
   public void init(final Logger logger,
                    final int playerId,
                    final List<Coordinate> playerHomes,
                    final List<List<Integer>> treasures,
                    final List<List<List<Integer>>> board,
                    final List<Integer> extra) {
      this.l = logger;
```

```
    this.playerId = playerId;

    //SHOULDN'T NEED TO DO playerHomes.subList(...) but engine is always passing playerHomes containing 4 elements at
    //the moment
    this.gameController = new GameController(playerId, playerHomes.subList(0, treasures.size()),
              treasures, board, extra);

    log("Loaded");
}

/**
 * Called when it's the AI player's turn to make a move.  The AI player generates
 * and returns its next move.
 *
 * @return the move that the AI player wants to make
 */
public PlayerMove move() {
    log("Move was requested...");

    return this.gameController.findBestMove();
}

/**
 * Notifies the AI player that a specified move was just made.  The AI player updates
 * the state of the game with this move.  It is assumed that all moves are given in
 * the order that they are made.  It is also assumed that all passed moves are valid.
 *
 * @param m - the move that was just made
 */
public void lastMove(final PlayerMove m) {
    log("Last move: " + m.toString());

    this.gameController.handlePlayerMove(m);
}

/**
 * Notifies the AI player that the specified opponent player made a bad move and has
 * been invalidated
 *
 * @param playerId - the id of the invalid player
 */
public void playerInvalidated(final int playerId) {
    log("Player " + Integer.toString(playerId) + " was invalidated :(");

}
```

```
    /**
     * Returns all the reachable cells which are adjacent to the specified coordinate.
     * This is not implemented as it is not called by the engine.
     *
     * @param c - the coordinate whose reachable adjacent cells are to be returned
     *
     * @return a set of reachable coordinates which are adjacent to the specified
     * coordinate
     */
    public Set<Coordinate> getNeighbors(final Coordinate c) {
        return null;
    }

    /**
     * Returns any valid path between the two specified coordinates.  This is not
     * implemented as it is not called by the engine.
     *
     * @param start - the starting coordinate from which any valid path to the
     * specified end coordinate should be returned
     * @param end - the end coordinate to which any valid path from the specified
     * start coordinate should be returned
     *
     * @return an ordered list of Coordinate objects representing a path from the
     * specified start coordinate to the specified end coordinate
     */
    public List<Coordinate> getPath(final Coordinate start, final Coordinate end) {
        return null;
    }

    private void log(final String msg) {
        String message =
            this.l.msg("AI (P" + Integer.toString(this.playerId) +")", msg);

        this.l.writeln(message);
        System.out.println(message);
    }
}
```

GameController.java

```
package Players.AIPlayer;

import Interface.Coordinate;
import Interface.PlayerMove;

import java.util.*;

/**
 * Represents game controller
 */
class GameController {
    private int playerId;
    private int nextOpponentPlayerId;
    private Board board;
    private Tile extraTile;

    /**
     * Initializes the GameController with the specified parameters
     *
     * @param playerId - the id of this player
     * @param playerHomes - starting locations for each player, in order
     * @param treasures - ordered list of treasures for each player
     * @param board - 2-d list of [Tile ID, Rotation, Treasure]
     * Tile IDs:  0 = L tile, 1 = T tile, 2 = I tile
     * Rotations: 0 = 0 degrees, 1 = 90 degrees, 2 = 180 degrees,
     *            3 = 270 degrees, all clockwise
     * Treasures: -1 = no treasure, 0-23 = corresponding treasure
     * @param extraTile - contains [Extra Tile ID, Treasure]
     */
    GameController(final int playerId,
                   final List<Coordinate> playerHomes,
                   final List<List<Integer>> treasures,
                   final List<List<List<Integer>>> board,
                   final List<Integer> extraTile) {
        this.playerId = playerId;
        this.nextOpponentPlayerId = this.playerId % playerHomes.size() + 1;
        this.board = new Board(playerHomes, treasures, board);
        this.extraTile = new Tile(MazePathType.fromId(extraTile.get(0)),
                                  TreasureType.fromId(extraTile.get(1)));
    }

    /**
     * Generates and returns the player's next best move
```

```
 *
 * @return player's next best move
 */
PlayerMove findBestMove() {
    final List<PlayerMove> bestPlayerMoves = new ArrayList<>();
    int myBestManhattanDistanceToGoal = Integer.MAX_VALUE;
    int nextOpponentWorstManhattanDistanceToGoal = 0;

    //Consider all valid tile insertion locations
    for(Coordinate tileInsertionLocation : this.board.getValidTileInsertionLocations()) {
        //Consider all extra tile orientations
        for(MazePathOrientation mazePathOrientation : MazePathOrientation.values()) {
            //Ignore 180 and 270 degree maze path orientation for 'I' maze type path, as they are equivalent
            //to 0 and 90 degree maze path orientations.
            if(extraTile.getMazePathType().equals(MazePathType.I) &&
                    (mazePathOrientation.equals(MazePathOrientation.ONE_HUNDRED_EIGHTY) ||
                            mazePathOrientation.equals(MazePathOrientation.TWO_HUNDRED_SEVENTY))) {
                continue;
            }

            //Create a copy of the current board and extra tile and insert the extra tile in the chosen insertion
            //location with the chosen tile orientation
            final Board tempBoard = (Board)Cloner.deepCopy(this.board);
            final Tile tempExtraTile = (Tile)Cloner.deepCopy(this.extraTile);
            tempExtraTile.setMazePathOrientation(mazePathOrientation);
            final Tile newTempExtraTile = tempBoard.insertTile(tempExtraTile, tileInsertionLocation);

            //Get the next goal coordinate for the player, which is the coordinate of the tile having the player's
            //next treasure or, if the player already collected all of its treasures, the coordinate of the
            //player's home tile.
            final Coordinate myNextGoalCoordinate = getNextGoalCoordinateForPlayer(tempBoard, this.playerId,
                newTempExtraTile);

            //Goal coordinate would be null if it is not reachable, which would happen if player's next treasure
            //is on the resulting extra tile after extra tile insertion
            if(myNextGoalCoordinate == null) {
                continue;
            }

            final Coordinate myCurrentLocationCoordinate = tempBoard.getPlayerLocation(this.playerId);
            final List<Coordinate> myPathTowardsNextGoal;

            //If the player is already on the tile in needs to reach next, the current location is the desired path
            //(no pawn move)
            if(myNextGoalCoordinate.equals(myCurrentLocationCoordinate)) {
```

```
            List<Coordinate> path = new ArrayList<>();
            path.add(myCurrentLocationCoordinate);
            myPathTowardsNextGoal = path;
        //If the player's goal coordinate is not the player's current location, find the best path towards the
        //next goal
        } else {
            myPathTowardsNextGoal = findBestPathTowardsNextGoalForPlayer(tempBoard, this.playerId,
            myNextGoalCoordinate);
        }

        int nextOpponentManhattanDistanceToGoal = -1;

        //If the last coordinate in the path is the goal coordinate, that means we have a path to the goal
        //coordinate
        if(myPathTowardsNextGoal.get(myPathTowardsNextGoal.size() - 1).equals(myNextGoalCoordinate)) {
            //If this is the first path we found leading all the way to the goal coordinate
            if(myBestManhattanDistanceToGoal != 0) {
                bestPlayerMoves.clear();
                myBestManhattanDistanceToGoal = 0;
                nextOpponentWorstManhattanDistanceToGoal = 0;
            }

            //Calculate next opponent's best Manhattan distance to their next goal coordinate, considering all
            //possible insertions that the next opponent would be allowed to do after this move
            nextOpponentManhattanDistanceToGoal = calculateNextOpponentBestManhattanDistanceToGoal(tempBoard,
                newTempExtraTile);
        //If the last coordinate in the path is not the goal coordinate, the goal coordinate is not reachable
        } else {
            //Calculate Manhattan distance from the closest approach to the next goal coordinate to the goal
            //coordinate
            final int myManhattanDistanceToGoal =
                calculateManhattanDistance(myPathTowardsNextGoal.get(myPathTowardsNextGoal.size() - 1),
                    myNextGoalCoordinate);

            //If the Manhattan distance from the closest approach to the next goal coordinate to the goal
            //coordinate is better than or equal than the approach to the goal coordinate from previously
            //considered insertions
            if(myManhattanDistanceToGoal <= myBestManhattanDistanceToGoal) {
                //If the Manhattan distance from the closest approach to the next goal coordinate to the goal
                //coordinate is better than the approach to the goal coordinate from previously considered
                //insertions
                if(myManhattanDistanceToGoal < myBestManhattanDistanceToGoal) {
                    bestPlayerMoves.clear();
                    myBestManhattanDistanceToGoal = myManhattanDistanceToGoal;
                    nextOpponentWorstManhattanDistanceToGoal = 0;
```

```
                    }

                    //Calculate next opponent's best Manhattan distance to their next goal coordinate, considering
                    //all possible insertions that the next opponent would be allowed to do after this move
                    nextOpponentManhattanDistanceToGoal =
                        calculateNextOpponentBestManhattanDistanceToGoal(tempBoard, newTempExtraTile);
                }
            }

            //If the opponent's best Manhattan distance distance to their next goal coordinate is worse than or
            //equal than the approach to their goal coordinate from previously considered insertions
            if(nextOpponentManhattanDistanceToGoal >= nextOpponentWorstManhattanDistanceToGoal) {
                //If the opponent's best Manhattan distance distance to their next goal coordinate is worse than
                //the approach to their goal coordinate from previously considered insertions
                if(nextOpponentManhattanDistanceToGoal > nextOpponentWorstManhattanDistanceToGoal) {
                    bestPlayerMoves.clear();
                    nextOpponentWorstManhattanDistanceToGoal = nextOpponentManhattanDistanceToGoal;
                }

                //Save this move
                bestPlayerMoves.add(new PlayerMove(this.playerId, myPathTowardsNextGoal, tileInsertionLocation,
                    mazePathOrientation.ordinal()));
            }
        }
    }

    //Return a random move from the list of equally good moves
    return bestPlayerMoves.get(new Random().nextInt(bestPlayerMoves.size()));
}

/**
 * Notifies the player that a specified move was just made.  The AI player updates
 * the state of the game with this move.  It is assumed that all moves are given in
 * the order that they are made.  It is also assumed that all passed moves are valid.
 *
 * @param playerMove - the move that was just made
 */
void handlePlayerMove(final PlayerMove playerMove) {
    //Set extra tile to the orientation specified in the specified player move
    this.extraTile.setMazePathOrientation(MazePathOrientation.fromId(playerMove.getTileRotation()));
    //Insert the extra tile at the insertion location specified in the specified player move
    this.extraTile = this.board.insertTile(this.extraTile, playerMove.getTileInsertion());

    final List<Coordinate> playerPath = playerMove.getPath();
```

```
    //Move the player specified in the specified player move to the destination location specified
    //in the specified player move
    this.board.movePlayer(playerMove.getPlayerId(), playerPath.get(playerPath.size() - 1));
}

private Coordinate getNextGoalCoordinateForPlayer(final Board board, final int playerId,
                                                  final Tile extraTile) {
    final TreasureType nextTreasureForPlayer = board.getNextTreasureForPlayer(playerId);

    ///If the player collected all treasures
    if(nextTreasureForPlayer == null) {
        return board.getPlayerHome(playerId);
    //If the player hasn't collected all of their treasures and the next treasure they need
    //to collect is on the board (not on extra tile)
    } else if(!extraTile.getTreasureType().equals(nextTreasureForPlayer)) {
        return board.getNextTreasureLocationForPlayer(playerId);
    }

    //If the next treasure player needs to collect is on the extra tile
    return null;
}

private int calculateNextOpponentBestManhattanDistanceToGoal(final Board board,
                                                             final Tile extraTile) {
    int nextOpponentBestManhattanDistanceToGoal = Integer.MAX_VALUE;

    for(Coordinate tileInsertionLocation : board.getValidTileInsertionLocations()) {
        for(MazePathOrientation mazePathOrientation : MazePathOrientation.values()) {
            //Ignore 180 and 270 degree maze path orientation for 'I' maze type path, as they
            //are equivalent to 0 and 90 degree maze path orientations.
            if(extraTile.getMazePathType().equals(MazePathType.I) &&
                    (mazePathOrientation.equals(MazePathOrientation.ONE_HUNDRED_EIGHTY) ||
                            mazePathOrientation.equals(MazePathOrientation.TWO_HUNDRED_SEVENTY))) {
                continue;
            }

            //Create a copy of the current board and extra tile and insert the extra tile in the chosen insertion
            //location with the chosen tile orientation
            final Board tempBoard = (Board)Cloner.deepCopy(board);
            final Tile tempExtraTile = (Tile)Cloner.deepCopy(extraTile);
            tempExtraTile.setMazePathOrientation(mazePathOrientation);
            final Tile newTempExtraTile = tempBoard.insertTile(tempExtraTile, tileInsertionLocation);

            final Coordinate nextOpponentNextGoalCoordinate = getNextGoalCoordinateForPlayer(tempBoard,
                    this.nextOpponentPlayerId, newTempExtraTile);
```

```
            int nextOpponentManhattanDistanceToGoal = Integer.MAX_VALUE;

            if(nextOpponentNextGoalCoordinate != null) {
                final Coordinate nextOpponentCurrentLocationCoordinate = tempBoard.getPlayerLocation(
                        this.nextOpponentPlayerId);

                if(nextOpponentCurrentLocationCoordinate.equals(nextOpponentNextGoalCoordinate)) {
                    return 0;
                } else {
                    final List<Coordinate> nextOpponentPathTowardsNextGoal = findBestPathTowardsNextGoalForPlayer(
                            tempBoard, this.nextOpponentPlayerId, nextOpponentNextGoalCoordinate);

                    if(nextOpponentPathTowardsNextGoal.get(nextOpponentPathTowardsNextGoal.size() - 1)
                            .equals(nextOpponentNextGoalCoordinate)) {
                        return 0;
                    } else {
                        nextOpponentManhattanDistanceToGoal = calculateManhattanDistance(
                                nextOpponentPathTowardsNextGoal.get(nextOpponentPathTowardsNextGoal.size() - 1),
                                nextOpponentNextGoalCoordinate);
                    }
                }
            }

            if(nextOpponentManhattanDistanceToGoal < nextOpponentBestManhattanDistanceToGoal) {
                nextOpponentBestManhattanDistanceToGoal = nextOpponentManhattanDistanceToGoal;
            }
        }
    }

    return nextOpponentBestManhattanDistanceToGoal;
}

private List<Coordinate> findBestPathTowardsNextGoalForPlayer(final Board board,
                                                              final int playerId,
                                                              final Coordinate goalCoordinate) {
    final Map<Coordinate, Coordinate> reachableCoordinates = findAllReachableCoordinates(board,
            null, board.getPlayerLocation(playerId), goalCoordinate, new HashMap<>());

    final List<Coordinate> bestReachableCoordinates = new ArrayList<>();

    if(reachableCoordinates.containsKey(goalCoordinate)) {
        bestReachableCoordinates.add(goalCoordinate);
    } else {
        int bestManhattanDistanceReachableCoordinateToGoal = Integer.MAX_VALUE;
```

```
        for(Coordinate reachableCoordinate : reachableCoordinates.keySet()) {
            final int manhattanDistanceReachableCoordinateToGoal = calculateManhattanDistance(reachableCoordinate,
                    goalCoordinate);

            if(manhattanDistanceReachableCoordinateToGoal <= bestManhattanDistanceReachableCoordinateToGoal) {
                if(manhattanDistanceReachableCoordinateToGoal < bestManhattanDistanceReachableCoordinateToGoal) {
                    bestReachableCoordinates.clear();
                    bestManhattanDistanceReachableCoordinateToGoal = manhattanDistanceReachableCoordinateToGoal;
                }

                bestReachableCoordinates.add(reachableCoordinate);
            }
        }
    }

    Stack<Coordinate> bestReversePathTowardsNextGoal = new Stack<>();
    Coordinate pathCoordinate = bestReachableCoordinates.get(new
        Random().nextInt(bestReachableCoordinates.size()));

    do {
        bestReversePathTowardsNextGoal.push(pathCoordinate);
        pathCoordinate = reachableCoordinates.get(pathCoordinate);
    }
    while(pathCoordinate != null);

    List<Coordinate> bestPathTowardsNextGoal = new ArrayList<>();

    while(!bestReversePathTowardsNextGoal.empty()) {
        bestPathTowardsNextGoal.add(bestReversePathTowardsNextGoal.pop());
    }

    return bestPathTowardsNextGoal;
}

private Map<Coordinate, Coordinate> findAllReachableCoordinates(final Board board,
        final Coordinate arrivedFromLocationCoordinate,
        final Coordinate currentLocationCoordinate,
        final Coordinate nextGoalCoordinate,
        final HashMap<Coordinate, Coordinate> reachableCoordinates) {
    final Tile currentLocationTile = board.getTile(currentLocationCoordinate.getRow(),
        currentLocationCoordinate.getCol());

    reachableCoordinates.put(currentLocationCoordinate, arrivedFromLocationCoordinate);
```

```
//check neighboring tile to the north
if(currentLocationCoordinate.getRow() > 0) {
    final Coordinate northTileCoordinate = new Coordinate(currentLocationCoordinate.getRow() - 1,
            currentLocationCoordinate.getCol());
    final Tile northTile = board.getTile(northTileCoordinate.getRow(), northTileCoordinate.getCol());

    if(currentLocationTile.hasExit(CompassDirection.NORTH) && northTile.hasExit(CompassDirection.SOUTH)) {
        if(northTileCoordinate.equals(nextGoalCoordinate)) {
            reachableCoordinates.put(northTileCoordinate, currentLocationCoordinate);

            return reachableCoordinates;
        } else if(!reachableCoordinates.containsKey(northTileCoordinate)) {
            reachableCoordinates.putAll(findAllReachableCoordinates(board, currentLocationCoordinate,
                    northTileCoordinate, nextGoalCoordinate, reachableCoordinates));
        }
    }
}

if(!reachableCoordinates.containsKey(nextGoalCoordinate)) {
    //check neighboring tile to the south
    if (currentLocationCoordinate.getRow() < Coordinate.BOARD_DIM - 1) {
        final Coordinate southTileCoordinate = new Coordinate(currentLocationCoordinate.getRow() + 1,
                currentLocationCoordinate.getCol());
        final Tile southTile = board.getTile(currentLocationCoordinate.getRow() + 1,
                currentLocationCoordinate.getCol());

        if (currentLocationTile.hasExit(CompassDirection.SOUTH) && southTile.hasExit(CompassDirection.NORTH)) {
            if (southTileCoordinate.equals(nextGoalCoordinate)) {
                reachableCoordinates.put(southTileCoordinate, currentLocationCoordinate);

                return reachableCoordinates;
            } else if (!reachableCoordinates.containsKey(southTileCoordinate)) {
                reachableCoordinates.putAll(findAllReachableCoordinates(board, currentLocationCoordinate,
                        southTileCoordinate, nextGoalCoordinate, reachableCoordinates));
            }
        }
    }

    if(!reachableCoordinates.containsKey(nextGoalCoordinate)) {
        //check neighboring tile to the west
        if(currentLocationCoordinate.getCol() > 0) {
            final Coordinate westTileCoordinate = new Coordinate(currentLocationCoordinate.getRow(),
                    currentLocationCoordinate.getCol() - 1);
            final Tile westTile = board.getTile(currentLocationCoordinate.getRow(),
                    currentLocationCoordinate.getCol() - 1);
```

```
                if(currentLocationTile.hasExit(CompassDirection.WEST) && westTile.hasExit(CompassDirection.EAST)) {
                    if(westTileCoordinate.equals(nextGoalCoordinate)) {
                        reachableCoordinates.put(westTileCoordinate, currentLocationCoordinate);

                        return reachableCoordinates;
                    } else if(!reachableCoordinates.containsKey(westTileCoordinate)) {
                        reachableCoordinates.putAll(findAllReachableCoordinates(board, currentLocationCoordinate,
                                westTileCoordinate, nextGoalCoordinate, reachableCoordinates));
                    }
                }
            }

            if(!reachableCoordinates.containsKey(nextGoalCoordinate)) {
                //check neighboring tile to the east
                if(currentLocationCoordinate.getCol() < Coordinate.BOARD_DIM - 1) {
                    final Coordinate eastTileCoordinate = new Coordinate(currentLocationCoordinate.getRow(),
                            currentLocationCoordinate.getCol() + 1);
                    final Tile eastTile = board.getTile(currentLocationCoordinate.getRow(),
                            currentLocationCoordinate.getCol() + 1);

                    if(currentLocationTile.hasExit(CompassDirection.EAST) &&
                           eastTile.hasExit(CompassDirection.WEST)) {
                        if(eastTileCoordinate.equals(nextGoalCoordinate)) {
                            reachableCoordinates.put(eastTileCoordinate, currentLocationCoordinate);

                            return reachableCoordinates;
                        } else if(!reachableCoordinates.containsKey(eastTileCoordinate)) {
                            reachableCoordinates.putAll(findAllReachableCoordinates(board,
                                currentLocationCoordinate, eastTileCoordinate, nextGoalCoordinate,
                                reachableCoordinates));
                        }
                    }
                }
            }
        }
    }

    return reachableCoordinates;
}

private int calculateManhattanDistance(final Coordinate coordinate1, final Coordinate coordinate2) {
    return Math.abs(coordinate2.getRow() - coordinate1.getRow()) +
            Math.abs(coordinate2.getCol() - coordinate1.getCol());
```

```
        }
    }
```

Board.java

```java
package Players.AIPlayer;

import Interface.Coordinate;

import java.util.*;

/**
 * Represents the game board
 */
class Board {
    private Tile[][] board;
    private Set<Coordinate> validTileInsertionLocations;
    private Coordinate invalidInsertionLocation;
    private Map<Integer, Coordinate> playerHomes;
    private Map<Integer, Coordinate> playerLocations;
    private Map<Integer, Queue<TreasureType>> playerTreasures;
    private Map<TreasureType, Coordinate> treasureLocations;

    /**
     * Initializes the Board with the specified parameters
     *
     * @param playerHomes - starting locations for each player, in order
     * @param treasures - ordered list of treasures for each player
     * @param board - 2-d list of [Tile ID, Rotation, Treasure]
     * Tile IDs:  0 = L tile, 1 = T tile, 2 = I tile
     * Rotations: 0 = 0 degrees, 1 = 90 degrees, 2 = 180 degrees,
     *            3 = 270 degrees, all clockwise
     * Treasures: -1 = no treasure, 0-23 = corresponding treasure
     */
    Board(final List<Coordinate> playerHomes,
          final List<List<Integer>> treasures,
          final List<List<List<Integer>>> board) {
        initValidTileInsertionLocations();
        initBoard(playerHomes, treasures, board);
    }

    /**
     * Initializes the Board object without any data
     */
    Board() {
    }

    /**
```

```
 * Returns the board array of tiles representing this Board
 *
 * @return board array of tiles representing this Board
 */
Tile[][] getBoard() {
    return this.board;
}

/**
 * Sets the board array of tiles representing this Board
 *
 * @param board - board array of tiles representing this Board
 */
void setBoard(final Tile[][] board) {
    this.board = board;
}

/**
 * Gets the set of valid tile insertions locations
 *
 * @return set of valid tile insertion locations
 */
Set<Coordinate> getValidTileInsertionLocations() {
    return this.validTileInsertionLocations;
}

/**
 * Sets the set of valid tile insertion locations
 *
 * @param validTileInsertionLocations - set of valid tile insertion locations
 */
void setValidTileInsertionLocations(final Set<Coordinate> validTileInsertionLocations) {
    this.validTileInsertionLocations = validTileInsertionLocations;
}

/**
 * Gets the insertion location not valid for insertion at this time
 *
 * @return insertion location not valid for insertion at this time
 */
Coordinate getInvalidInsertionLocation() {
    return this.invalidInsertionLocation;
}

/**
```

```
 * Sets the insertion location not valid for insertion at this time
 *
 * @param invalidInsertionLocation - insertion location not valid for insertion
 * at this time
 */
void setInvalidInsertionLocation(final Coordinate invalidInsertionLocation) {
    this.invalidInsertionLocation = invalidInsertionLocation;
}

/**
 * Gets the map mapping each player to their home location
 *
 * @return map mapping each player to their home location
 */
Map<Integer, Coordinate> getPlayerHomes() {
    return this.playerHomes;
}

/**
 * Sets the map mapping each player to their home location
 *
 * @param playerHomes - map mapping each player to their home location
 */
void setPlayerHomes(final Map<Integer, Coordinate> playerHomes) {
    this.playerHomes = playerHomes;
}

/**
 * Gets the map mapping each player their current location
 *
 * @return map mapping each player to their current location
 */
Map<Integer, Coordinate> getPlayerLocations() {
    return this.playerLocations;
}

/**
 * Sets the map mapping each player to their current location
 *
 * @param playerLocations - map mapping each player to their current location
 */
void setPlayerLocations(final Map<Integer, Coordinate> playerLocations) {
    this.playerLocations = playerLocations;
}
```

```java
/**
 * Gets the map mapping each player to their treasure pile
 *
 * @return mapping mapping each player to their treasure pile
 */
Map<Integer, Queue<TreasureType>> getPlayerTreasures() {
    return this.playerTreasures;
}

/**
 * Sets the map mapping each player to their treasure pile
 *
 * @param playerTreasures - map mapping each player to their treasure pile
 */
void setPlayerTreasures(final Map<Integer, Queue<TreasureType>> playerTreasures) {
    this.playerTreasures = playerTreasures;
}

/**
 * Gets the map mapping each treasure to their current location
 *
 * @return map mapping each treasure to their current location
 */
Map<TreasureType, Coordinate> getTreasureLocations() {
    return this.treasureLocations;
}

/**
 * Sets the map mapping each treasure to their current location
 *
 * @param treasureLocations - map mapping each treasure to their current location
 */
void setTreasureLocations(final Map<TreasureType, Coordinate> treasureLocations) {
    this.treasureLocations = treasureLocations;
}

/**
 * Inserts the specified tile at the location identified by the specified coordinate
 * and returns the tile that was pushed out as as result the inserting the specified tile.
 * Illegal argument exception is thrown if the location where the tile should be inserted
 * at is not a valid insertion location.
 *
 * @param tileToInsert - tile to be inserted
 * @param tileInsertionLocation - location where the tile to be inserted will be inserted at
 *
```

```
     * @return tile that was pushed out as a result of the inserted tile
     *
     * @throws IllegalArgumentException if the location where the tile should be inserted at
     * is not a valid insertion location
     */
    Tile insertTile(final Tile tileToInsert, final Coordinate tileInsertionLocation) {
        final Tile shiftedOutTile;

        if(!this.validTileInsertionLocations.contains(tileInsertionLocation)) {
            throw new IllegalArgumentException("Specified tile insertion location is not a valid tile" +
                    "insertion location.");
        }

        if(this.invalidInsertionLocation != null) {
            this.validTileInsertionLocations.add(this.invalidInsertionLocation);
        }

        final int rowToInsertTileAt = tileInsertionLocation.getRow();
        final int columnToInsertTileAt = tileInsertionLocation.getCol();

        Tile tileToShiftIn = tileToInsert;

        //Insert on the north side of the board
        if(rowToInsertTileAt == 0) {
            Tile tileToShiftDown = null;

            for(int rowIndex = 0; rowIndex < Coordinate.BOARD_DIM; rowIndex++) {
                tileToShiftDown = this.board[rowIndex][columnToInsertTileAt];
                this.board[rowIndex][columnToInsertTileAt] = tileToShiftIn;
                tileToShiftIn = tileToShiftDown;
            }

            shiftedOutTile = tileToShiftDown;
            this.invalidInsertionLocation = new Coordinate(Coordinate.BOARD_DIM - 1, columnToInsertTileAt);
        //Insert on the south side of the board
        } else if(rowToInsertTileAt == (Coordinate.BOARD_DIM - 1)) {
            Tile tileToShiftUp = null;

            for(int rowIndex = Coordinate.BOARD_DIM - 1; rowIndex >= 0; rowIndex--) {
                tileToShiftUp = this.board[rowIndex][columnToInsertTileAt];
                this.board[rowIndex][columnToInsertTileAt] = tileToShiftIn;
                tileToShiftIn = tileToShiftUp;
            }

            shiftedOutTile = tileToShiftUp;
```

```
            this.invalidInsertionLocation = new Coordinate(0, columnToInsertTileAt);
        //Insert on the west side of the board
        } else if(columnToInsertTileAt == 0) {
            Tile tileToShiftRight = null;

            for(int columnIndex = 0; columnIndex < Coordinate.BOARD_DIM; columnIndex++) {
                tileToShiftRight = this.board[rowToInsertTileAt][columnIndex];
                this.board[rowToInsertTileAt][columnIndex] = tileToShiftIn;
                tileToShiftIn = tileToShiftRight;
            }

            shiftedOutTile = tileToShiftRight;
            this.invalidInsertionLocation = new Coordinate(rowToInsertTileAt, Coordinate.BOARD_DIM - 1);
        //Insert on the east side of the board
        } else {
            Tile tileToShiftLeft = null;

            for(int columnIndex = Coordinate.BOARD_DIM - 1; columnIndex >= 0; columnIndex--) {
                tileToShiftLeft = this.board[rowToInsertTileAt][columnIndex];
                this.board[rowToInsertTileAt][columnIndex] = tileToShiftIn;
                tileToShiftIn = tileToShiftLeft;
            }

            shiftedOutTile = tileToShiftLeft;
            this.invalidInsertionLocation = new Coordinate(rowToInsertTileAt, 0);
        }

        this.validTileInsertionLocations.remove(this.invalidInsertionLocation);

        if(shiftedOutTile.hasPlayer()) {
            this.board[rowToInsertTileAt][columnToInsertTileAt].addPlayers(shiftedOutTile.getPlayers());
            shiftedOutTile.removeAllPlayers();
        }

        updatePlayerAndTreasureLocations();

        return shiftedOutTile;
    }

    /**
     * Moves the specified player to the location identified by the specified destination location
     *
     * @param player - player which should be moved to the location identified by the specified
     * destination location
     * @param destinationLocation - location to which the specified player should be move to
```

```
 */
void movePlayer(final int player, final Coordinate destinationLocation) {
    final Coordinate currentPlayerLocation = this.playerLocations.get(player);

    this.board[currentPlayerLocation.getRow()][currentPlayerLocation.getCol()].removePlayer(player);

    final Tile destinationTile = this.board[destinationLocation.getRow()][destinationLocation.getCol()];

    destinationTile.addPlayer(player);

    if(destinationTile.getTreasureType().equals(this.playerTreasures.get(player).peek())) {
        this.playerTreasures.get(player).poll();
    }

    this.playerLocations.put(player, destinationLocation);
}

/**
 * Gets the home location of the specified player
 *
 * @param player - player whose home location is to be returned
 *
 * @return home location of the specified player
 */
Coordinate getPlayerHome(final int player) {
    return this.playerHomes.get(player);
}

/**
 * Gets the current location of the specified player
 *
 * @param player - player whose current location is to be returned
 *
 * @return current location of the specified player
 */
Coordinate getPlayerLocation(final int player) {
    return this.playerLocations.get(player);
}

/**
 * Gets the next treasure that needs to be collected by the specified player
 *
 * @param player - player whose next treasure to be collected is to be returned
 *
 * @return next treasure that needs to be collected by the specified player
```

```
 */
TreasureType getNextTreasureForPlayer(final int player) {
    return this.playerTreasures.get(player).peek();
}

/**
 * Gets the location of the next treasure that needs to be collected by the
 * specified player
 *
 * @param player - player whose location of the next treasure to be collected
 * is to be returned
 *
 * @return location of the next treasure that needs to be collected by the
 * specified player
 */
Coordinate getNextTreasureLocationForPlayer(final int player) {
    return this.treasureLocations.get(this.playerTreasures.get(player).peek());
}

/**
 * Gets the tile located at the row and column location identified by the
 * specified row and column indices
 *
 * @param rowIndex - identifies the row where the tile to be returned is located at
 * @param columnIndex - identifies the column where the tile to be returned is located at
 *
 * @return tile located at the row and column location identified by the
 * specified row and column indices
 */
Tile getTile(final int rowIndex, final int columnIndex) {
    return this.board[rowIndex][columnIndex];
}

private void initValidTileInsertionLocations() {
    this.validTileInsertionLocations = new HashSet<>();

    for(int index = 1; index < Coordinate.BOARD_DIM; index+=2) {
        //North side
        this.validTileInsertionLocations.add(new Coordinate(0, index));
        //South side
        this.validTileInsertionLocations.add(new Coordinate(6, index));
        //East side
        this.validTileInsertionLocations.add(new Coordinate(index, 6));
        //West side
        this.validTileInsertionLocations.add(new Coordinate(index, 0));
```

```
    }
}

private void initBoard(final List<Coordinate> playerHomes,
                       final List<List<Integer>> treasures,
                       final List<List<List<Integer>>> board) {
    final Map<Coordinate, Integer> playerHomeToIdMap = new HashMap<>();
    this.playerHomes = new HashMap<>();
    this.playerLocations = new HashMap<>();

    for(int player = 1; player <= playerHomes.size(); player++) {
        final Coordinate playerHome = playerHomes.get(player - 1);

        playerHomeToIdMap.put(playerHome, player);
        this.playerHomes.put(player, playerHome);
        this.playerLocations.put(player, playerHome);
    }

    initPlayerTreasures(treasures);

    this.board = new Tile[Coordinate.BOARD_DIM][Coordinate.BOARD_DIM];
    this.treasureLocations = new HashMap<>();

    for(int rowIndex = 0; rowIndex < Coordinate.BOARD_DIM; rowIndex++) {
        for(int columnIndex = 0; columnIndex < Coordinate.BOARD_DIM; columnIndex++) {
            final List<Integer> tileInfoList = board.get(rowIndex).get(columnIndex);

            final Coordinate currentTileCoordinate = new Coordinate(rowIndex, columnIndex);

            final TreasureType treasureType = TreasureType.fromId(tileInfoList.get(2));

            if(!treasureType.equals(TreasureType.NONE)) {
                this.treasureLocations.put(treasureType, currentTileCoordinate);
            }

            if(playerHomeToIdMap.containsKey(currentTileCoordinate)) {
                this.board[rowIndex][columnIndex] = new Tile(MazePathType.fromId(tileInfoList.get(0)),
                                                             MazePathOrientation.fromId(tileInfoList.get(1)),
                                                             TreasureType.fromId(tileInfoList.get(2)),
                                                             playerHomeToIdMap.get(currentTileCoordinate));
            } else {
                this.board[rowIndex][columnIndex] = new Tile(MazePathType.fromId(tileInfoList.get(0)),
                                                             MazePathOrientation.fromId(tileInfoList.get(1)),
                                                             TreasureType.fromId(tileInfoList.get(2)));
            }
```

```
            }
        }
    }

    private void initPlayerTreasures(List<List<Integer>> playerTreasures) {
        this.playerTreasures = new HashMap<>();

        for(int player = 1; player <= playerTreasures.size(); player++) {
            final Queue<TreasureType> playerTreasuresQueue = new LinkedList<>();

            for(Integer treasureId : playerTreasures.get(player - 1)) {
                final TreasureType treasureType = TreasureType.fromId(treasureId);
                playerTreasuresQueue.add(treasureType);
            }

            this.playerTreasures.put(player, playerTreasuresQueue);
        }
    }

    private void updatePlayerAndTreasureLocations() {
        this.playerLocations.clear();
        this.treasureLocations.clear();

        for(int rowIndex = 0; rowIndex < Coordinate.BOARD_DIM; rowIndex++) {
            for(int columnIndex = 0; columnIndex < Coordinate.BOARD_DIM; columnIndex++) {
                final Tile tile = this.board[rowIndex][columnIndex];
                final Coordinate coordinate = new Coordinate(rowIndex, columnIndex);

                for(int player : tile.getPlayers()) {
                    this.playerLocations.put(player, coordinate);
                }

                final TreasureType treasureType = tile.getTreasureType();

                if(!treasureType.equals(TreasureType.NONE)) {
                    this.treasureLocations.put(treasureType, coordinate);
                }
            }
        }
    }
}
```

Tile.java

```
package Players.AIPlayer;

import java.util.HashSet;
import java.util.Set;

/**
 * Represents a tile of the game board
 */
class Tile  {
    private MazePathType mazePathType;
    private MazePathOrientation mazePathOrientation;
    private TreasureType treasureType;
    private Set<Integer> players;
    private static boolean[][][] hasExitLookupTable;

    /**
     * Initializes hasExitLookupTable to return true for the MazeTypePath, MazePathOrientation and CompassDirection
     * combinations which cause the tile to have an exit
     */
    static {
        hasExitLookupTable = new boolean[MazePathType.values().length]
                [MazePathOrientation.values().length]
                [CompassDirection.values().length];

        for(int mazePathType = 0; mazePathType < MazePathType.values().length; mazePathType++) {
            for(int mazePathOrientation = 0; mazePathOrientation < MazePathOrientation.values().length;
                mazePathOrientation++) {
                for(int compassDirection = 0; compassDirection < CompassDirection.values().length;
                    compassDirection++) {
                    hasExitLookupTable[mazePathType][mazePathOrientation][compassDirection] = false;
                }
            }
        }

        hasExitLookupTable[MazePathType.I.ordinal()][MazePathOrientation.ZERO.ordinal()]
                [CompassDirection.NORTH.ordinal()] = true;
        hasExitLookupTable[MazePathType.I.ordinal()][MazePathOrientation.ZERO.ordinal()]
                [CompassDirection.SOUTH.ordinal()] = true;
        hasExitLookupTable[MazePathType.I.ordinal()][MazePathOrientation.ONE_HUNDRED_EIGHTY.ordinal()]
                [CompassDirection.NORTH.ordinal()] = true;
        hasExitLookupTable[MazePathType.I.ordinal()][MazePathOrientation.ONE_HUNDRED_EIGHTY.ordinal()]
                [CompassDirection.SOUTH.ordinal()] = true;
        hasExitLookupTable[MazePathType.I.ordinal()][MazePathOrientation.NINETY.ordinal()]
```

```
        [CompassDirection.EAST.ordinal()] = true;
hasExitLookupTable[MazePathType.I.ordinal()][MazePathOrientation.NINETY.ordinal()]
        [CompassDirection.WEST.ordinal()] = true;
hasExitLookupTable[MazePathType.I.ordinal()][MazePathOrientation.TWO_HUNDRED_SEVENTY.ordinal()]
        [CompassDirection.EAST.ordinal()] = true;
hasExitLookupTable[MazePathType.I.ordinal()][MazePathOrientation.TWO_HUNDRED_SEVENTY.ordinal()]
        [CompassDirection.WEST.ordinal()] = true;
hasExitLookupTable[MazePathType.L.ordinal()][MazePathOrientation.ZERO.ordinal()]
        [CompassDirection.WEST.ordinal()] = true;
hasExitLookupTable[MazePathType.L.ordinal()][MazePathOrientation.ZERO.ordinal()]
        [CompassDirection.NORTH.ordinal()] = true;
hasExitLookupTable[MazePathType.L.ordinal()][MazePathOrientation.NINETY.ordinal()]
        [CompassDirection.NORTH.ordinal()] = true;
hasExitLookupTable[MazePathType.L.ordinal()][MazePathOrientation.NINETY.ordinal()]
        [CompassDirection.EAST.ordinal()] = true;
hasExitLookupTable[MazePathType.L.ordinal()][MazePathOrientation.ONE_HUNDRED_EIGHTY.ordinal()]
        [CompassDirection.EAST.ordinal()] = true;
hasExitLookupTable[MazePathType.L.ordinal()][MazePathOrientation.ONE_HUNDRED_EIGHTY.ordinal()]
        [CompassDirection.SOUTH.ordinal()] = true;
hasExitLookupTable[MazePathType.L.ordinal()][MazePathOrientation.TWO_HUNDRED_SEVENTY.ordinal()]
        [CompassDirection.SOUTH.ordinal()] = true;
hasExitLookupTable[MazePathType.L.ordinal()][MazePathOrientation.TWO_HUNDRED_SEVENTY.ordinal()]
        [CompassDirection.WEST.ordinal()] = true;
hasExitLookupTable[MazePathType.T.ordinal()][MazePathOrientation.ZERO.ordinal()]
        [CompassDirection.SOUTH.ordinal()] = true;
hasExitLookupTable[MazePathType.T.ordinal()][MazePathOrientation.ZERO.ordinal()]
        [CompassDirection.WEST.ordinal()] = true;
hasExitLookupTable[MazePathType.T.ordinal()][MazePathOrientation.ZERO.ordinal()]
        [CompassDirection.NORTH.ordinal()] = true;
hasExitLookupTable[MazePathType.T.ordinal()][MazePathOrientation.NINETY.ordinal()]
        [CompassDirection.WEST.ordinal()] = true;
hasExitLookupTable[MazePathType.T.ordinal()][MazePathOrientation.NINETY.ordinal()]
        [CompassDirection.NORTH.ordinal()] = true;
hasExitLookupTable[MazePathType.T.ordinal()][MazePathOrientation.NINETY.ordinal()]
        [CompassDirection.EAST.ordinal()] = true;
hasExitLookupTable[MazePathType.T.ordinal()][MazePathOrientation.ONE_HUNDRED_EIGHTY.ordinal()]
        [CompassDirection.NORTH.ordinal()] = true;
hasExitLookupTable[MazePathType.T.ordinal()][MazePathOrientation.ONE_HUNDRED_EIGHTY.ordinal()]
        [CompassDirection.EAST.ordinal()] = true;
hasExitLookupTable[MazePathType.T.ordinal()][MazePathOrientation.ONE_HUNDRED_EIGHTY.ordinal()]
        [CompassDirection.SOUTH.ordinal()] = true;
hasExitLookupTable[MazePathType.T.ordinal()][MazePathOrientation.TWO_HUNDRED_SEVENTY.ordinal()]
        [CompassDirection.EAST.ordinal()] = true;
hasExitLookupTable[MazePathType.T.ordinal()][MazePathOrientation.TWO_HUNDRED_SEVENTY.ordinal()]
        [CompassDirection.SOUTH.ordinal()] = true;
```

```
    hasExitLookupTable[MazePathType.T.ordinal()][MazePathOrientation.TWO_HUNDRED_SEVENTY.ordinal()]
            [CompassDirection.WEST.ordinal()] = true;
}

/**
 * Initializes the Tile with the specified parameters.  No tile orientation is
 * set for the tile and it is set to have no players.  It is used to initialize
 * the extra tile, as the extra tile has no orientation and no players on it.
 *
 * @param mazePathType - maze path type of the tile
 * @param treasureType - treasure type located on the tile
 */
Tile(final MazePathType mazePathType,
     final TreasureType treasureType) {
    this.mazePathType = mazePathType;
    this.treasureType = treasureType;
    this.players = new HashSet<>();
}

/**
 * Initializes the Tile with the specified parameters.  It is set to have
 * no players.
 *
 * @param mazePathType - maze path type of the tile
 * @param mazePathOrientation - maze path orientation of the tile
 * @param treasureType - treasure type located on the tile
 */
Tile(final MazePathType mazePathType,
     final MazePathOrientation mazePathOrientation,
     final TreasureType treasureType) {
    this(mazePathType, treasureType);
    this.mazePathOrientation = mazePathOrientation;
}

/**
 * Initializes the Tile with the specified parameters
 *
 * @param mazePathType - maze path type of the tile
 * @param mazePathOrientation - maze path orientation of the tile
 * @param treasureType - treasure type located on the tile
 * @param player - player located on the tile
 */
Tile(final MazePathType mazePathType,
     final MazePathOrientation mazePathOrientation,
     final TreasureType treasureType,
```

```
     final int player) {
    this(mazePathType, mazePathOrientation, treasureType);
    this.players.add(player);
}

/**
 * Gets the maze path type of the tile
 *
 * @return maze path type of the tile
 */
MazePathType getMazePathType() {
    return this.mazePathType;
}

/**
 * Gets the maze path orientation of the tile
 *
 * @return maze path orientation of the tile
 */
MazePathOrientation getMazePathOrientation() {
    return this.mazePathOrientation;
}

/**
 * Sets the maze path orientation of the tile
 *
 * @param mazePathOrientation - maze path orientation of the tile
 */
void setMazePathOrientation(final MazePathOrientation mazePathOrientation) {
    this.mazePathOrientation = mazePathOrientation;
}

/**
 * Gets the treasure type located on the tile
 *
 * @return treasure type located on the tile
 */
TreasureType getTreasureType() {
    return this.treasureType;
}

/**
 * Gets the set of players located on the tile
 *
 * @return set of players located on the tile
```

```
 */
Set<Integer> getPlayers() {
    return this.players;
}

/**
 * Adds the specified set of players to the set of players located on the tile
 *
 * @param players set of players to be added to the set of players located on
 * the tile
 */
void addPlayers(final Set<Integer> players) {
    this.players.addAll(players);
}

/**
 * Add the specified player to the set of players located on the tile
 *
 * @param player player to be added to the set of players located on the tile
 */
void addPlayer(final int player) {
    this.players.add(player);
}

/**
 * Removes the specified player from the set of players located on the tile
 *
 * @param player player to be removed from the set of players located on the tile
 */
void removePlayer(final int player) {
    this.players.remove(player);
}

/**
 * Removes all players from the tile
 */
void removeAllPlayers() {
    this.players.clear();
}

/**
 * Checks if the tile has any players on it
 *
 * @return True if the tile has any players on it; false otherwise
 */
```

```
    boolean hasPlayer() {
        return !this.players.isEmpty();
    }

    /**
     * Checks if the tile has an exit in the specified compass direction.  Throws
     * IllegalStateException if the tile does not have orientation set.
     *
     * @param compassDirection - compass direction for which the tile is to be
     * checked to see if it has an exit
     *
     * @return True if the tile has an exit in the specified compass direction; false
     * otherwise
     *
     * @throws IllegalStateException if the tile does not have orientation set
     */
    boolean hasExit(final CompassDirection compassDirection) {
        if(this.mazePathOrientation == null) {
            throw new IllegalStateException("This tile does not have an orientation set, therefore checking " +
                "if it has an exit in a particular compass direction is not valid.");
        }

        return Tile.hasExitLookupTable[this.mazePathType.ordinal()][this.mazePathOrientation.ordinal()]
                [compassDirection.ordinal()];
    }
}
```

CompassDirection.java

```java
package Players.AIPlayer;

/**
 * Represents four standard compass directions
 */
enum CompassDirection {
    NORTH,
    EAST,
    SOUTH,
    WEST
}
```

MazePathOrientation.java

```java
package Players.AIPlayer;

/**
 * Represents a maze path orientation
 */
enum MazePathOrientation {
    ZERO,
    NINETY,
    ONE_HUNDRED_EIGHTY,
    TWO_HUNDRED_SEVENTY;

    /**
     * Gets the value of MazePathOrientation matching the specified id.  Throws
     * IllegalArgumentException if none of the values of MazePathOrientation match
     * the specified id.
     *
     * @param id - identifies the value of MazePathOrientation which is to be returned
     *
     * @return value of MazePathOrientation matching the specified id
     *
     * @throws IllegalArgumentException if none of the values of MazePathOrientation
     * match the specified id
     */
    public static MazePathOrientation fromId(final int id) {
        for(MazePathOrientation mazePathOrientation : MazePathOrientation.values()) {
            if(mazePathOrientation.ordinal() == id) {
                return mazePathOrientation;
            }
        }

        throw new IllegalArgumentException("Can not cast id " + id + " into a MazePathOrientation");
    }
}
```

MazePathType.java

```
package Players.AIPlayer;

/**
 * Represents maze path type
 */
enum MazePathType {
    L,
    T,
    I;

    /**
     * Gets the value of MazePathType matching the specified id.  Throws
     * IllegalArgumentException if none of the values of MazePathType match
     * the specified id.
     *
     * @param id - identifies the value of MazePathType which is to be returned
     *
     * @return value of MazePathType matching the specified id
     *
     * @throws IllegalArgumentException if none of the values of MazePathType
     * match the specified id
     */
    public static MazePathType fromId(final int id) {
        for(MazePathType mazePathType : MazePathType.values()) {
            if(mazePathType.ordinal() == id) {
                return mazePathType;
            }
        }

        throw new IllegalArgumentException("Can not cast id " + id + " into a MazePathType");
    }
}
```

TreasureType.java

```
package Players.AIPlayer;

/**
 * Represents a treasure type
 */
enum TreasureType {
    NONE(-1),
    T1(0),
    T2(1),
    T3(2),
    T4(3),
    T5(4),
    T6(5),
    T7(6),
    T8(7),
    T9(8),
    T10(9),
    T11(10),
    T12(11),
    T13(12),
    T14(13),
    T15(14),
    T16(15),
    T17(16),
    T18(17),
    T19(18),
    T20(19),
    T21(20),
    T22(21),
    T23(22),
    T24(23);

    private int id;

    TreasureType(final int id) {
        this.id = id;
    }

    /**
     * Gets the id associated with the TreasureType
     *
     * @return id associated with the TreasureType
     */
```

```
    public int getId() {
        return this.id;
    }

    /**
     * Gets the value of TreasureType whose id matches the specified id.  Throws
     * IllegalArgumentException if none of the values of TreasureType have an id
     * matching the specified id.
     *
     * @param id - identifies the id of the value of TreasureType which is
     * to be returned
     *
     * @return value of TreasureType whose id matches the specified id
     *
     * @throws IllegalArgumentException if none of the values of TreasureType have
     * an id matching the specified id
     */
    public static TreasureType fromId(final int id) {
        for(TreasureType treasureType : TreasureType.values()) {
            if(treasureType.getId() == id) {
                return treasureType;
            }
        }

        throw new IllegalArgumentException("Can not cast id " + id + " into a TreasureType");
    }
}
```

Appendix B - Activity Log:

Date	Time Spent (Hours)	Activities Performed
Jan 24	1	Discussed and agreed upon the Independent Study topic, planned work, deliverables and evaluation criteria
Jan 25	1.5	Filled out the Independent Study Form and wrote the Independent Study Proposal Document
Jan 31	1	Reviewed Adam Oest's Labyrinth Web Engine documentation; familiarized myself with the main rules of the Labyrinth game
Feb 13	2.5	Setup IntelliJ, imported LabyrinthJavaClient project into the IDE, created a GitHub project for the LabyrinthJavaClient project and pushed the intial project to GitHub
Mar 2-3	6	Wrote AI Player Data and Strategy Design document
Mar 17	2	Started work on AIPlayer.java
Mar 28	5	Implemented MazePathOrientation, MazePathType and TreasureType enums. Added Tile class. Created initial Board and GameState classes. Made more changes to the AIPlayer class. Added ability to insert a tile with checking if insertion coordinate is legal. Added some commenting to enums. Initialization of the board state now initializes player homes as well. Added functionality that prohibits insertion of a tile at a location where the tile was pushed out in the preceding move.
Mar 29	2	GameState now stores and initializes extra tile and players' treasure piles. Tile creation was simplified.
Mar 30	1	Handle scenario where tile being pushed out has players, so the players are wrapped around. Other refactoring. When players wrap around, use add players, not set players.
Apr 2	1	Fixing bugs. Accounting for the fact that Coordinate object x and y are zero based. This now allows the AIPlayer to run without error. Generating random valid insertion point and handling player moves. performBestMove method now returns a PlayerMove. handlePlayerMove now accounts for extra tile orientation when updating the board state. Generating random valid tile orientation for insertion. Using player id instead of hardcoded value.
Apr 3	4	
Apr 4	4	When generating best move, I didn't need to update my board because that happens when I'm notified of my own move. Added ability to print the board. Now tracking player locations on board initialization and after each tile insertion. Added ability to move player and get player location to Board class. handlePlayerMove in the GameController now properly updates the board not only for the tile insertion but also for the pawn move. Added getTile to Board class. Added CompassDirection enum. GameController generate random path now actually checks neighboring tiles to see if path to them exists. Dummy implementation of Tile.hasExist(CompassDirection). Fixed error in the Board class movePlayer method where I was using currentPlayerLocation instead of destinationLocation when moving the player. Also added createCopy method to the Board class. In the GameController, generateRandomPath now uses the copy of the board with the extra tile inserted at the randomly calculated insertion position. Tile.hasExit(CompassDirection) is now implemented.

		Updated GameController.generateRandomPath(..) so that the pawn isn't just moved to the north/south/west/east direction in that precedence order (meaning if north and south moves are available, previously north would always get chosen). It now finds all possible moves and randomly chooses one of them.
Apr 5	1.5	Simplified Tile.hasExit(CompassDirection) to eliminate unnecessary branches and converted to using switch statements instead of ifs.
Apr 6	1.5	Tile.hasExit(CompassDirection) now uses a static look up hash set instead of a bunch of switch statements for efficiency.
Apr 10	5	Started coding to replace generateRandomMove with findBestMove. GameController now tries to find the best move based on which insertion point and pawn move takes the player closest to their next treasure in terms of Manhattan distance. Board class now tracks player treasures and treasure locations on initialization and on tile insertion and player moves. Added final modifiers where they were missing. getNextTreasureForPlayer added to Board. Dealing with situation where there's no distance possible to calculate because player's next treasure is off the board (on extra tile). Other temporary debugging changes. Debugging a strange issue causing a critical error.
Apr 11	3	Changed hasExitLookupTable in Tile class to store integer values rather than strings for efficiency. Continued debugging a strange issue causing a critical error.
Apr 12	2	Tile class now has a createCopy method and all methods/constructors in Tile are now package-private. GameController now creates and uses a copy of the extra tile in findBestMove(). Some code in findBestPathToNextTreasure(Board) was previously commented out to make the method more deterministic in order to debug a critical issue. The issue was fixed now with creating a copy of the extra tile, so uncommented this commented out code. Board class now tracks and exposes player homes. GameController now handles the situation where all treasures have been collected by the player and the player should now try to reach their home location.
Apr 13	3	hasExitLookupTable has been converted to a boolean[][][] from Set<Integer>. GameController.findBestMove() stops searching for best move once it already found a move that takes it directly to the treasure. findBestPathTowardsNextGoal(..) no longer just returns a random reachable neighbor.
Apr 14	2	Full-fledged player implementation. GameController.findBestPathTowardsNextGoal(..) is now fully implemented. Organized imports. Removed unused imports. Now calculating list of best insertions and best pawn moves in case there are multiple best moves, in which case a random best move is chosen rather than the first best move.
Apr 15	2	Optimization of finding next best path.
May 1	3	Work in progress to choose worst move for opponent from equally good moves for current player.
May 2	2	Implemented feature where worst move for next opponent is considered when current player has equally good moves.
May 3	2	Significantly simplified GameController.findBestMove() by combining code branches that were very similar and pulling out some common code into a private method. Fixed a bug where the opponent's Manhattan distance

		was being measured between incorrect points.
May 4	3	When determining the next move, next opponent's all possible insertions are now considered to find the current player's move that will make it worst for the opponent.
May 6	2	Tested AI player.
May 7	3	Switched to use Professor Heliotis' cloner utility to clone objects instead of serializing/deserializing. Added more comments to the code. Re-tested the AI player. Removed System.out lines that were added for testing purposes.
May 8	0.5	Created Final Report Outline document
May 12	7	Wrote the Final Report document
May 13	3	Added Javadoc comments to the code where they were missing and created Appendix A – Source Code and Appendix B – Activity Log for the Final Report document