Université Jean Monnet

Machine Learning & Data Mining

Intro. to AI: Prolog Project

Amazing Labyrinth

Edward Beeching
Jorge Chong
Sejal Jaiswal
Dimitris Tsolakidis

January, 8th 2017

Table of Contents

1 Introduction

The goal of this project is to build a two player game using Prolog such that it is possible for a human player to play against the AI agent or let two AI agents play against each other with different heuristics. There are five heuristics we have implemented (described in detail in section 2.1). Each heuristic has a different approach about how to win the game.

2 Amazing Labyrinth

2.1 The Game

Amazing Labyrinth is a board game that can be played with a maximum of four players. However, we have modified the game such that only a maximum of two players can play at a time. The board is a 7 * 7 grid that consists of 16 fixed tiles and 33 movable tiles. Amongst the 16 fixed tiles, there are 12 different treasures. In the beginning of the game, the movable tiles are set up randomly.

The objective of the game is for each player to search the Labyrinth for a set of treasures by carefully moving through the constantly changing maze. There are five treasures randomly assigned to each player. The first player to find all of their treasures and then return to the starting square is the winner.

Each player has to adjust the configuration of the maze in order to create a path to his next treasure or to reach a specific position. The configurations can be adjusted either by moving a column of tiles either up or down, or by moving a row of tiles left or right. Not all rows and columns are movable.

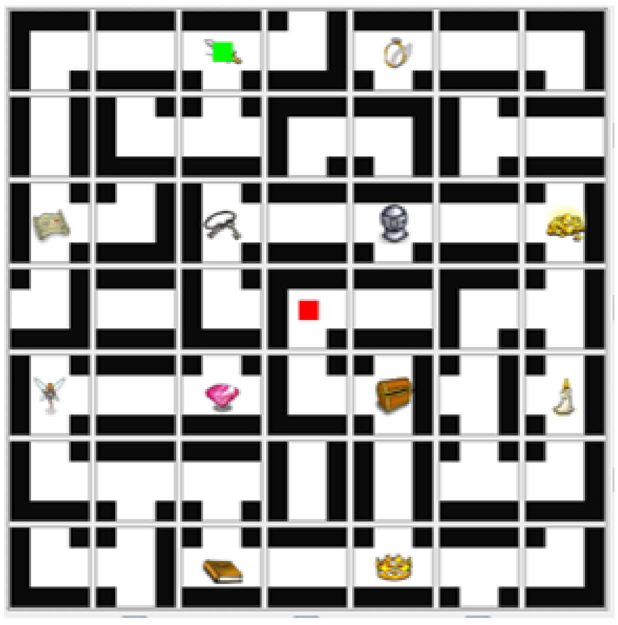
2.2 Rules

- For each turn a player gets two moves:
  1. (Compulsory) Shift any one row or column.
  2. Move your piece to any connected position or stay at the same position.
- The board is wrapped around, which means any tile that leaves the board from one end will re-enter into the same row or column from the other end.
- When a row or column is being shifted, any player on that particular row or column also changes position according to that particular move.
- A player must collect the treasures in the order assigned at the beginning of the game.

2.3 Implementation: Prolog

The state of the game is represented by: a board, two players, a history of states, valid moves, list of treasures and the index of the current treasure.

The board is represented as a list of lists in Prolog. Each element of the list is a row. Each element of a row is a binary representation of the tile.



```
[
 [0110, 0101, 0111, 1001, 0111, 0101, 0011],
 [1010, 1100, 1101, 0110, 0011, 1110, 0101],
 [1110, 1001, 1110, 0101, 0111, 0101, 1011],
 [1001, 0101, 1100, 0110, 0101, 0110, 1011],
 [1110, 0101, 1101, 1100, 1011, 1011, 1011],
 [1100, 0111, 0111, 1010, 1010, 1101, 1001],
 [1100, 1011, 1101, 0101, 1101, 0111, 1001]
]
```

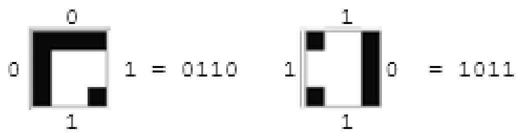


Figure 1: Board Setup

Players are represented by predicate `player(Player, Heuristic, PositionI/PositionJ)`. There are two players a and b. Examples: `player(a,h1,1/1)`: Player a using heuristic h1 in row 1 column 1.

The game also has a list of game states saved (History). Each element of history is represented as: `Board/a/A2/A3/A4/b/B2/B3/B4`, where: Board is the matrix representing the board, `A2` the player a's heuristic, `A3` the player a's `PositionI/PositionJ`, `A4` the player a's Target, `B2` the player b's heuristic, `B3` the player b's `PositionI/PositionJ`, and `B4` the player b's `Target`.

The state of the game is saved as a predicate game_state. This is necessary to validate the rule of the game that specifies the order in which a player can move.

The game is played with the following predicates:

- `setup(_)`: To set up the board for a game. This calls other predicates internally that setup the tiles, sets the treasures for each players, etc.
- `create_shifted_player(I,J,Move,NewI,NewJ)`: This predicate implements the rule that when a particular row or column is shifted, the players on that particular row or column are also moved.

- `create_shifted_board(CurrentBoard,Move,NewBoard)`: This predicate generates a new board after making the shifts as specified by a player, ie the board after the row left or right shift or column up or down shift.
- `graph_search_BFS(NewBoard,NewI,NewJ,ListOfVisitedNodes)`: This predicate returns a list of positions that are accessible to a particular player from a certain position. The available positions are in the form of a list of i/j, where i and j are the coordinates.
- `try_and_make_move(Player, Heuristic)`: Evaluates the best board shift to apply for a given heuristic and then applies it.
- `make_best_local_move(Player, Heuristic)`: Evaluates and applies the best move for the current board, for a given heuristic.

2.4 Implementation: Java

The UI for the game has been implemented using Java. The UI provides an option to choose the players either human vs computer (using a heuristic) or both players can be the computer (choosing different heuristics or the same one). The UI also provides an option to choose the number of games that can be played as well as the time delay between different moves.

`QueryProlog.java`: The `QueryProlog` class is used to communicate between the Labyrinth user interface and the Game AI in Prolog, using the JPL library. For further details of the Java implementation, consult the Appendix for the UML diagram.

2.5 Heuristics (Ordered by increasing complexity)

2.5.1 Heuristic H1

This heuristic searches one step ahead of its current position. To do so, it tries to evaluate each to the 12 possible rotations of rows and columns. For each of these rotations, it applies a breadth first search algorithm on a potential board to get the list of all reachable tiles. For every reachable tile, it calculates a score using a function based on the Manhattan distance (Max distance = 12). This heuristic selects as the next move, a combination of the board's rotation and a move that maximises this score. Note: The rotations could possibly move the tile in which the player is located. The AI takes that into account as well.

Complexity: 12 rotations multiplied by the number of tiles reachable from the graph search for each rotations. See Appendix 1 for a picture showing just the first move.

2.5.2 Heuristic H3

Searches two steps ahead of it's current position. As if trying to make two moves in a row. It can be described as follows:

- From the current player's position try to make each of the 12 rotations. Then for every one of these, do a graph search and get the list of reachable tiles. Let's call each position `I/J`.
- For every one of these new possible positions, try to make again the 12 board rotations.

- Once again make a graph search to retrieve the list of reachable tiles given the previous board rotation.
- Score each one of these according to a Manhattan distance, in the same way H1 scores each tile.
- Now get the maximum score and associate with `I/J` as `Score/I/J`. Again, for all positions `I/J`, calculate the maximum and associate it with the Move as `Move/Score/I/J`.
- Finally, to execute this move, rotate the board using `Move`, and move to the position `I/J`.

There are a large number of possible game states explored when looking two steps ahead, as there are 12 possible board shifts, multiplied by between 1 and 49 possible places a player can then reach, which is again multiplied by 12 possible board rotations. In an average case where 15 locations are connected to the player, this will lead to 2160 possible board configurations. Each configuration is then searched with a BFS algorithm.

2.5.3 Heuristic H4

This heuristic implements H3 and also includes a score based on the closeness of the opponent to their target. H4 is based on H3, but during the evaluation of the second step, for every one of them, do a graph search from the perspective of the opponent, and score each position based on the manhattan distance to his own target. Let's call the original score as `ScoreA` and this new score as `ScoreB`. The resulting score is: `Score = (ScoreA - (weight_factor x ScoreB))`. The rest is the same as in H3, that is we get the `Move/Score/I/J` that maximises the combined score. After some trial and error, an appropriate weight_factor was found to be 0.5.

This heuristic can be interpreted as trying to make the move that maximises the closeness from the player to the target in the second move; and also tries to minimise the closeness of the opponent to his own target in this second move.

2.5.4 Heuristic H2

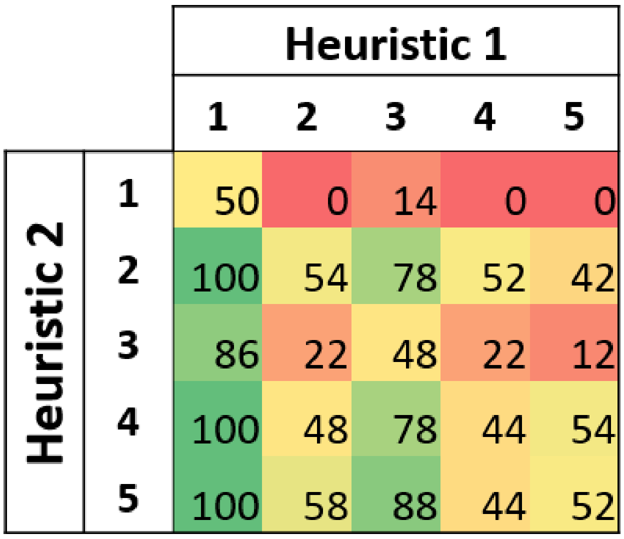
This is similar to H4, but differs only in the choice of the distance function. The distance function used here to calculate the score is a Circular Manhattan Distance. The circular Manhattan Distance is described as: `min(abs(x1-x2), 7 - abs(x1-x2)) + min(abs(y1-y2), 7 - abs(y1-y2))`. Because the wrapping around property of the board, some moves require just a rotation in the opposite direction suggested by a score based on normal manhattan distance.

2.5.5 Heuristic H5

This improves upon Heuristic H2 but also scoring moves based on their ability to create a path from the current target to the next target. The scoring function is now: `Score = (ScoreA + (weight1 x ScoreC) - (weight2 x ScoreB))`. `ScoreA` and `ScoreB` are the same as in H2. `ScoreC` is the new score. The weights selected are `weight1 = 0.25` and `weight2 = 0.5`.

3 Tests and Results

The heuristics were tested by a combination of games against a human player and many games of heuristic vs. heuristic. Figure 2 summarises the results of 50 games of all combinations of the 5 heuristics playing against each other. Note that due to the random variation of the assignment of the treasure lists, even when a heuristic plays itself, the result is not always 50:50. Run times of the heuristics ranges from 5-10 ms for heuristic 1, to 200-1500 ms for heuristics 2, 3, 4 & 5, depending on the board configuration.



		Heuristic 1				
		1	2	3	4	5
Heuristic 2	1	50	0	14	0	0
	2	100	54	78	52	42
	3	86	22	48	22	12
	4	100	48	78	44	54
	5	100	58	88	44	52

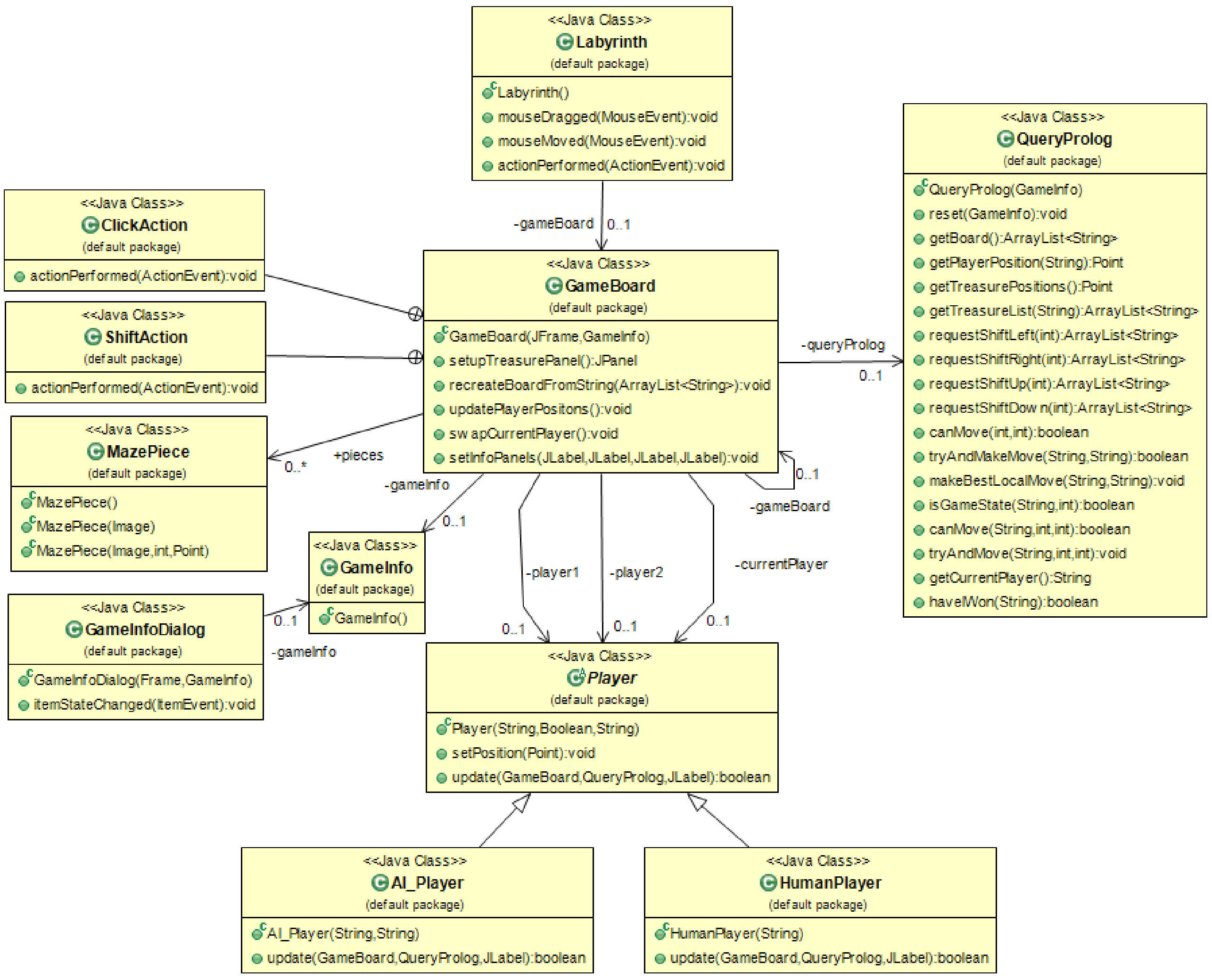
Figure 2: Heuristics comparison with percentage win rate over 50 games

4 Conclusion

Five convincing, different and challenging heuristics were implemented. An intuitive, responsive and user-friendly UI was created, with the ability to play heuristics against each other. There was good teamwork and a good mutual understanding of the implemented heuristics.

The main challenges of this project was the implementation of a complicated game with a large branching factor. Some heuristics were initially observed to get stuck and make repeated moves, this was resolved by recording a history of all explored game states. Using the JPL library on Mac OS.X took a while to get working but was achieved in the end.

Appendices



<<Java Class>>
Labyrinth
(default package)
Labyrinth()
mouseDragged(MouseEvent):void
mouseMoved(MouseEvent):void
actionPerformed(ActionEvent):void
<<Java Class>>
QueryProlog
(default package)
QueryProlog(GameInfo)
reset(GameInfo):void
getBoard():ArrayList<String>
getPlayerPosition(String):Point
getTreasurePositions():Point
getTreasureList(String):ArrayList<String>
requestShiftLeft(int):ArrayList<String>
requestShiftRight(int):ArrayList<String>
requestShiftUp(int):ArrayList<String>
requestShiftDown(int):ArrayList<String>
canMove(int,int):boolean
tryAndMakeMove(String,String):boolean
makeBestLocalMove(String,String):void
isGameState(String,int):boolean
canMove(String,int,int):boolean
tryAndMove(String,int,int):void
getCurrentPlayer():String
haveWon(String):boolean
<<Java Class>>
ClickAction
(default package)
actionPerformed(ActionEvent):void
<<Java Class>>
ShiftAction
(default package)
actionPerformed(ActionEvent):void
<<Java Class>>
GameBoard
(default package)
GameBoard(JFrame,GameInfo)
setupTreasurePanel():JPanel
recreateBoardFromString(ArrayList<String>):void
updatePlayerPositons():void
swapCurrentPlayer():void
setInfoPanels(JLabel,JLabel,JLabel,JLabel):void
-gameBoard
0..1
-queryProlog
0..1
<<Java Class>>
MazePiece
(default package)
MazePiece()
MazePiece(Image)
MazePiece(Image,int,Point)
+pieces
0..*
-gameInfo
0..1
0..1
-gameBoard
<<Java Class>>
GameInfo
(default package)
GameInfo()
-player1
-player2
-currentPlayer
0..1
0..1
0..1
<<Java Class>>
GameInfoDialog
(default package)
GameInfoDialog(Frame,GameInfo)
itemStateChanged(ItemEvent):void
0..1
-gameInfo
<<Java Class>>
Player
(default package)
Player(String,Boolean,String)
setPosition(Point):void
update(GameBoard,QueryProlog,JLabel):boolean
<<Java Class>>
AI_Player
(default package)
AI_Player(String,String)
update(GameBoard,QueryProlog,JLabel):boolean
<<Java Class>>
HumanPlayer
(default package)
HumanPlayer(String)
update(GameBoard,QueryProlog,JLabel):boolean

Figure 3: UML diagram for the game

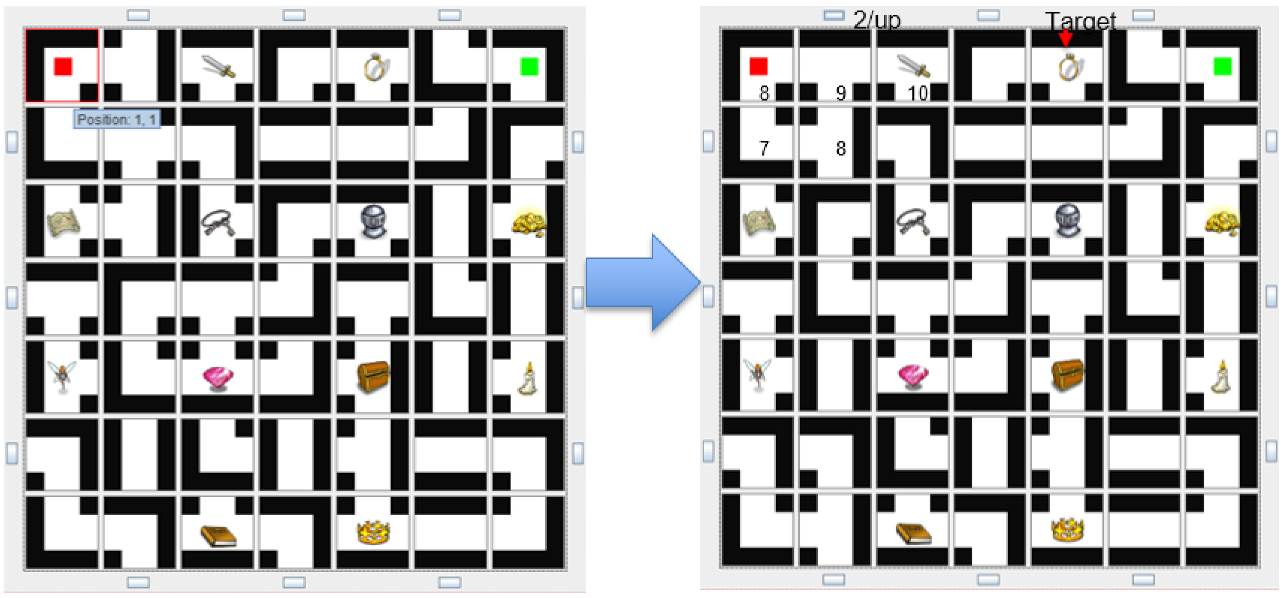


Figure 4: Evaluation of the first shift for H1