Snails game with AI

ARTIFICIAL INTELLIGENCE FOR GAMES (DR. JUNAID AKHTAR)

Muhammad Usman
12036782

Introduction

There was a time when computer and games were considered to be two opposite ends of a huge technological river. But during the span of the 1950s, the developers came up with the idea of 'Computer Games'. That is, people were able to play games on computers too. Then, another critical breakthrough was given by a scientist named 'Arthur Samuel' who gave the idea of making the computer able to think so that it can play games. For this purpose, he studied the 'Checkers game' and gave a case study for building an 'Intelligent Agent' to play this game (Samuel, 2000).

After the new path guided by Samuel, a new dimension of research was opened for the coming scientists. Initially, the agents were simpler and were using the rule-based mechanism which was working fine for small board games, but it was difficult to tackle complex games like Chess with that procedure. Therefore, next progress came in the form of 'Decision tree' and Min-Max in which every possible move was being checked and Maximizing the profit (in terms of win/loss) during agent's turn and Minimizing that otherwise did the task.

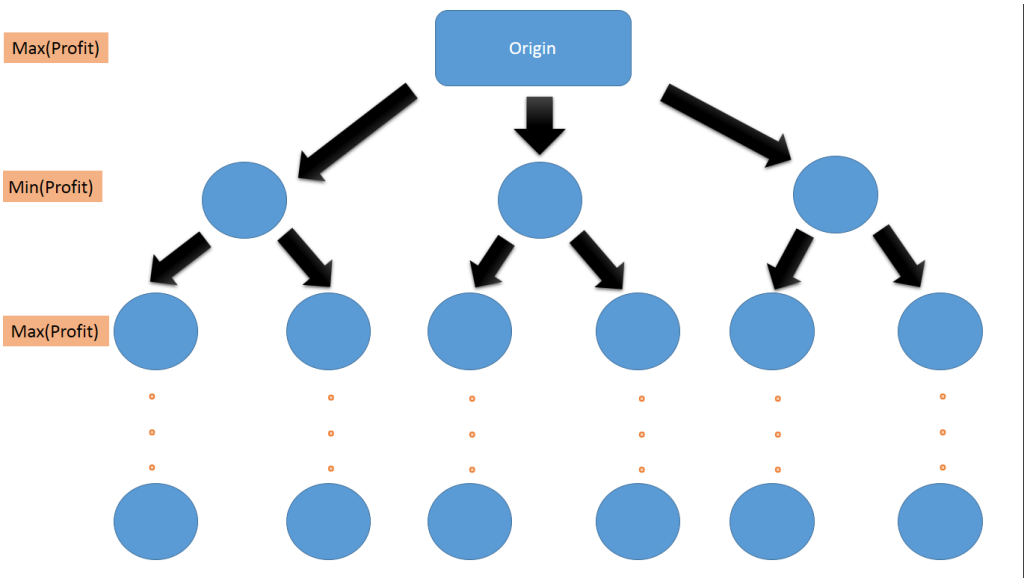

Figure 1 Min-Max Algorithm

As shown in figure 1, at any given state of the board, children were created by exploring all possible moves of player at that time till the endgame was reached and result (win and loss represented by maximum and minimum values respectively) was sent back to the parent node. Also, when the turn was agent's it was selecting the child having maximum value and when the agent was making a turn as a human (to complete tree by reaching the next level), it was selecting the children with least value.

The report is written to explain the implementation of Snails game with an Artificial Intelligent (AI) agent. The mentioned game is a played on a board of usually N x N between two games, and one of the players was an artificially intelligent agent in our scenario. We have already discussed the game user interface and possible moves in the section 'User Manual'.

For the implementation of computer agent, we have used a decision tree, Min-Max algorithm, and an intelligent heuristic function to evaluate the state of the board at a given time.

Game Implementation

First of all, user interface (front-end) and back-end of the game was separated so that user can play the game in a user-friendly environment and the game logic can be implemented easily on the back-end. To fulfill this purpose back-end and front-end of the game was built in the following way:

Backend Implementation

To implement the back-end of this game, state of the game was stored in an 8x8 matrix of integers. Also, it was noticed that there were 5 possible states of a particular cell in the Snails game, therefore, 5 different integer values were used to occupy every possible state. Possible states and the used integer for that state is shown in Table 1:

Table 1 Integer representation of cell states

State of the cell	Representative Integer value
Player one current position	1
Player one splash (player one's territory)	11
Player two current position	2
Player two splash (player two's territory)	22
Empty cell	0

Also, splashes of both players were saved in a 1x2 matrix in such a manner that 1st index was containing the splash of player one, and the 2nd index was having the integer value for the other splash. By using this matrix, process of placing splashes was made more efficient, as it saved us from checking the current turn before placing a particular splash, and we just did ***splash(TURN)*** where TURN was 1 or 2.

Frontend Implementation

For the implementation of front end of the game, we need to make an 8 x 8 grid where the user can play the game by using a mouse. To make this grid, 400 x 400 matrix of ones (to make a white board) was used having zeros on every nth column and row (n is a multiple of 50). This was done to draw black lines on the white board. (400/8 = 50)

Also, four different images were used to represent four possible states other than an empty cell.

Game Logic

To develop any general game, there are certain methods which are needed to be working in a certain order. For example, a general board game would be having following steps:

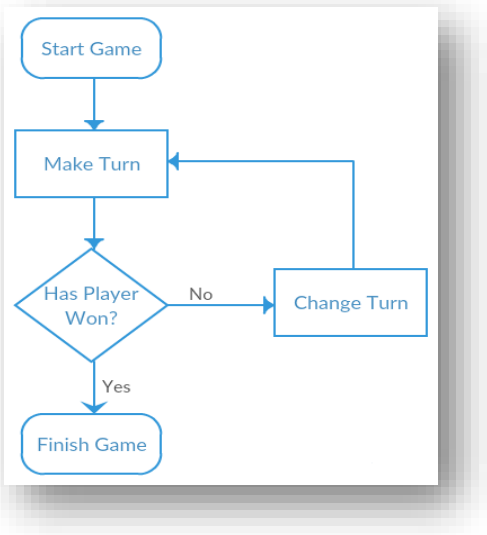

Figure 2 General game flow

Keeping this flow of events in view, the provided game was divided into subparts and each of the game subpart was implemented and tested separately to accomplish component based design. At last, all the game components were assembled into the main driver function. The overall flow of game logic that was used for the development of this game is as follows (AI agent was developed in the end):

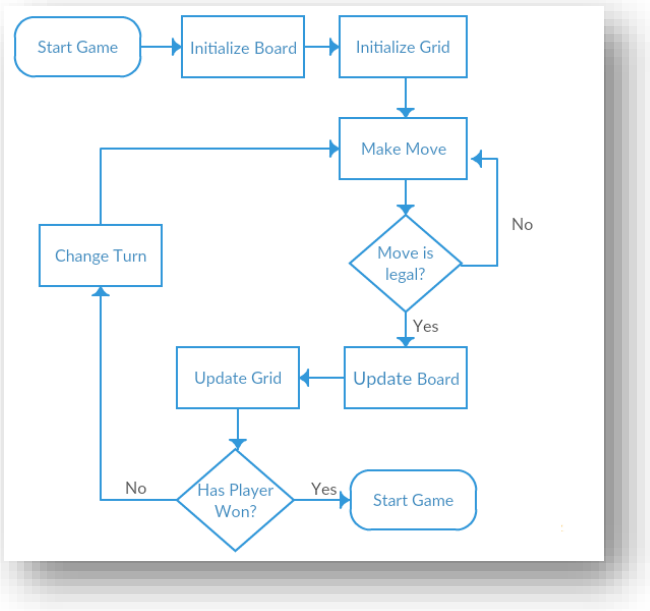

Figure 3 Snails game flow

Initialize board

To initialize the board, a matrix of 8 x 8 was created and initialized with all zeros. Then left-bottom corner and the top-right corner was initialized with player one and player two initial positions respectively. This was done using a function ***initBoard()*** that took rows and columns on the board as parameters and returned the board representing the start of the 'Snails game'. The returned matrix, storing back-end of game, was looking like this:

0	0	0	0	0	0	0	2
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0

Initialize grid

After initializing the board, the grid was initialized. For this purpose, a function initGrid() was written that accepts number of horizontal and vertical lines as parameters and returns the grid in its initial form, that is, empty cells of grid. This function initialized a matrix of 400 x 400 by ones (for white background) and placed zeros (black lines) in all those rows and columns which were multiple of 50 using the following code:

```
grid(rowInterval:rowInterval:HEIGHT,:) = 0;      % Making rows, which are multiple of 50, black
grid(:,colInterval:colInterval:WIDTH) = 0;       % Making cols, which are multiple of 50, black
```

Here, rowInterval and colInterval were equal to 50. The value 50 was obtained by dividing the total size of the matrix (400 x 400) with total horizontal/vertical lines (8).

Update grid

When the grid was initialized, players' current position was not being shown (it was having all empty cells). Therefore, to synchronize the grid with the given state of the board, a separate function ***updateGrid()*** was written that got the current state of the board, previous grid, grid box height, grid and box width as parameters and returned the updated grid by placing the images of size (box-height) x (box-width). The function iterated through each cell of the board and placed unique images for every non-zero state of the game. Placement of images was done using a separate function ***placeImages ().***
This function was used in the later stage of game too, where after every move, board was updated and this function was being called with the updated state of the board.

Function placeImages ()

This function received boxY, boxX, BOXWIDTH, BOXHEIGHT, grid, flag as parameters. boxY and boxX were used to get the column index and row index respectively where the image needs to be placed. Similarly, BOXWIDTH and BOXHEIGHT were elaborating the number of columns and number of rows for one cell in the grid matrix. Flag parameter was used to tell the function that which of the image should be placed.

That means separate flags were used for every image. After analyzing the flag value, the function was placing the image on required rows and columns of the grid, and returning the updated grid.

Update Board

After every legal move, the board was needed to be updated. This was done by a function named 'Update Board'. The function took previous board position, current turn, destination x and destination y as parameters and returned the updated board position. This function checks the destination cell selected by the user, and if the destination is an empty cell, then simply moves the player from source to destination by leaving a splash on the source position. But, if destination already holds a splash of the same player, then it forwards the player position till the last splash.

Evaluate board

After every update in board and before changing the current turn, we needed to check if the game is still continue or the player has won/lost A function ***evaluateBoard()*** was written to evaluate a particular state of the game with respect to winning/lose/draw. The function was accepting the state of the game in the form of a board and turn of the AI agent as parameters and was returning -64 for loss, 64 for the win, 32 for a draw and 0 for continue from agent's perspective. Another function ***isWin()*** was written to assist the evaluate board function. ***isWin()*** took board and agent's turn as parameters and founded splashes of the player using ***find*** function of MatLab:

```
player = find(board == turn);
```

After calculating the length of vector returned from the mentioned function. It was returning 64 if AI player had got more than half splashes, -64 if AI player had got less than half splashes and 0 if both players had got equal splashes.

Legal move

A function isLegal() was written to check the validity of a given move. The function received board, destination X coordinate for board, destination Y coordinate, current turn, and splash matrix as parameters and returned 1 or 0 for representing a valid and an invalid move respectively. To illustrate, the function was firstly looking for a destination that is either 0 (empty) or a splash of the current TURN. Also, it was ensuring that either destination row and source row or destination column and source column should be same.

	OPPONENT SPLASH		
	PLAYER CURRENT POSITION		

For example, if the current position of the player is in the mid of above-mentioned piece of board, then the player can move in the green boxes only. That is, it can move one cell in right, left, or below, but cannot move above as the above cell is already conquered by the opponent.

Turn and turn change

To track the current turn of the game, a variable named 'TURN' was used and was initialized with a random value between 1 and 2 (inclusive). This was done to give the game more professional look where first turn can belong to any of the player:

```
TURN = randint(1,1,[1 2]);                    % Start with a random value
```

Also, the overall logic of the game was written in such a way that this variable could have one of the two values (1 or 2) at any given time. After every successful turn, turn was changed using a function ***changeTurn()*** that received the current turn and returned the next turn using:

```
turn = mod(agentTurn,2) + 1;
```

Human-turn

The human turn was made using a built-in function of MatLab ***ginput()*** that gave us the X and Y coordinate of the grid where the user has clicked**.** This ginput() command was called in a loop until the destination selected by the user is valid.

Grid to Board

The function ginput() gave us the position of the grid where the user had clicked but to map this interaction with the backend of our game, we used another function ***gridToBoard()*** that was converting the grid values (rows and columns) to the corresponding indices of the board. The function received height of grid cell, width of grid cell, x coordinate where the user has clicked, y coordinate where the user has clicked, total rows in the grid, total columns in the grid as parameters and returned the x and y index of board. That is, this function is **mapping frontend to backend**.

AI-Agent for Snails game

Firstly, we tried to implement the Artificial Intelligent agent for our game using 'Decision tree'.

Decision/Search Tree for making move

A method named 'searchTree()' was written for this in which a tree was generated from the current state of the game. This function received the board, current player turn (for which children would be created), agentTurn (for which Min-Max would be decided), depth. In this method, children states (immediate next states of a particular state) were generated from every state in such a way that next state must be one of the generated children states. To achieve this functionality, another function was written named '***generateChildren()***' that received parent state of the board, a matrix of splashes flags, and next (upcoming) turn as parameters and return children boards in the form of N-dimensional matrix, where N is the number of children. The process of generating children nodes was repeated until mentioned method 'evaluateBoard()' returned anything other than '0', that is, the tree has been expanded till the end of the game, or the depth of tree given from main driver function has been achieved. Moreover, to mimic the actual scenario where after every move of our agent, there will be human player move, generateChildren() was called with changed turn after every level and 'Min-Max' algorithm was implemented that was maximizing our agent's profit when the turn was equal to AI agent's turn and minimizing it otherwise.

Problem with Search Tree

After this implementation of 'searchTree()' that needed to return the updated board and outcome of that board in terms of the score after making the turn from the AI player, it was seen that this method was taking too much time for traversing the whole tree, especially in the beginning of the game.

Heuristic with limited depth search of search tree

To tackle the mentioned problem, it was decided to limit the depth of the tree to some level D, and when that height is achieved. The best possible move was decided from another function named ***'evaluateHeuristic()'*** that took a game state as an argument and returned a corresponding heuristic score for that board. Then the children with the better score were picked in case of the turn of AI agent. Heuristic evaluation was also an evolutionary process that started with a simple idea that was returning area covered by the player as the score. For example, the score returned by that heuristic for the player, having snail with a line over it, would be 10 (2 rows of 5 columns are covered).

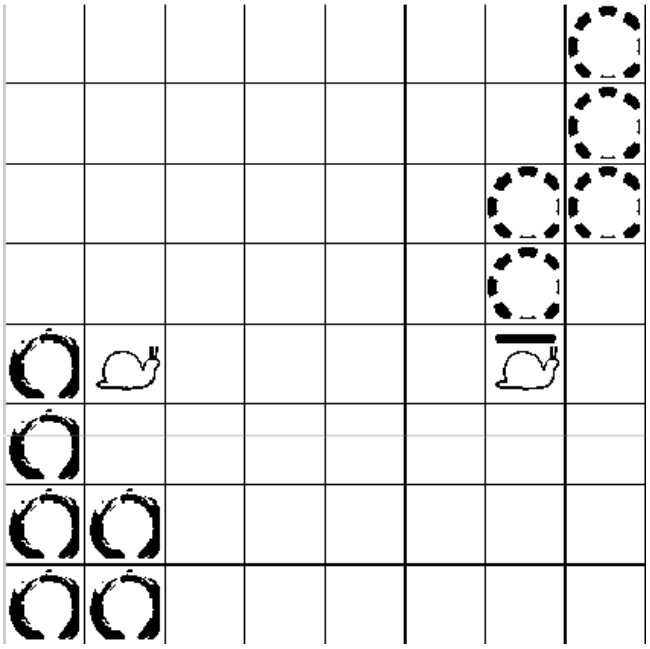

Figure 4 Board position at a given instance

Then, this function was improved by introducing two more numbers which were added into the possible covered area. These numbers were calculated by finding the difference of rows and columns between current positions of players and then dividing these distances by 2. This gave us a broader area that can the conquered by the agent at any cost. For example, in the mentioned example board (figure 3), those two number for rows and columns would be calculated in the following manner:

```
diffX1 = abs(x1 - rows1);
diffY1 = abs(y1 - cols1);
```

Here, x1 and y1 are the positions of player one row and column respectively, and rows1 and cols1 are the positions for player two. Then half of the diffX1 was used as scale value for rows and same for the other value. This strategy returned score of **20** for the mentioned game state (2 was used as a scale in the column value).

The flaw of this technique was that it was unable to distinguish between conquered and empty boxes within the covered area. This was causing our agent to stuck at a point somewhere after mid of the board (figure 3 is explaining that problem).

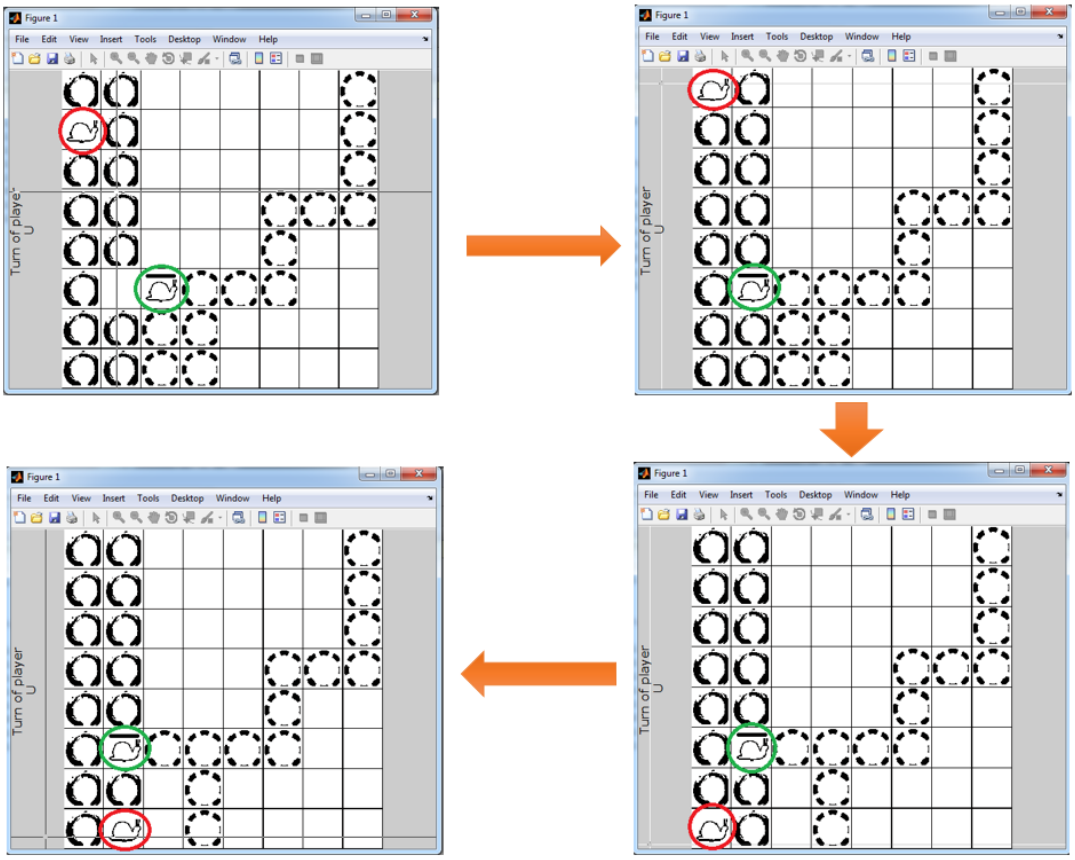

Figure 5 Old AI agent

Therefore, score was again calculated in such a way that after finding the area, splashes of player were found using find() function, then empty boxes were found by subtracting the splashes from (area – 1). Then final score was calculated by:

Final score = (Number of splashes x 1) + (Number of Empty cells x 0.5)

In this way, we were able to give priority to those paths where the agent can cover more cells and agent was able to prioritize splashes over empty cells too within the conquered area. Figure 4 is showing the moves of an improved agent which is does not get stuck. Final score for mentioned game state is:

Final score = (5 x 1) + (14 x 0.5) = 12

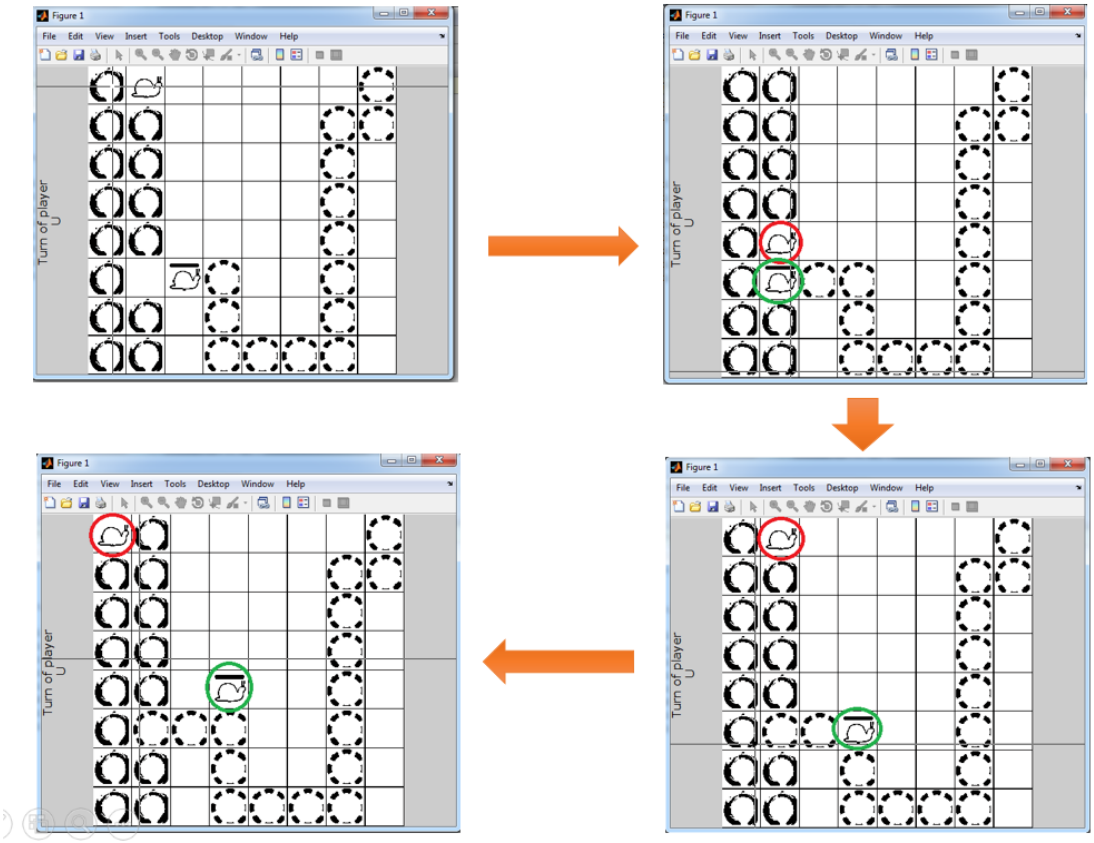

Figure 6 Improved Version of AI agent

Main driver function

All these components of game were combined using a main() function that was controlling order of calls to these functions. This main function was called from the console with dimension of the board. After initializing the board and the grid, the function was entering into a loop that was ending when evaluate board was returing win/loss/draw. In this loop, current player was making his move (calling the search tree function if it was agent's turn) and after ensuring the validity of move, update board function was called followed by update grid.

Conclusion

In this work, an efficient Artificial agent was created for Snails game using a depth limited decision tree with a heuristic function. Creation of agent was an evolutionary process that was able to provide desired results at the end, that is, the agent was able to make moves intelligently in an efficient way.

References

Samuel, A. .L. 2000. Some Studies in Machine Learning Using the Game of Checkers. IBM Journal of Research and Development. 44(1/2), pp. 206-226.