Snails AI

Abdul Malik, Shakir Ullah, Badar Shazad and
Abdur Rehman Khan Niazi

March 30, 2018

Abstract

AI in games is making computer controlled chracters behave like human. Algorithms like path finding, making plans, or learning from the opponent player are used in such games to make these characters intelligent. In short, AI is a term that refers to a set of Algorithms that make the game agents look intelligent (Bura, 2012). This documents investigates some algorithms that are used to implement an AI for a game called Snails. Efficiency and performance analysis of implemented algorithms have also been discussed in this document.

1 Introduction

The aim of our project was to develop a Snails game including an AI agent which tries to maximize its score and eventually win the game. Snails is a game which can be played like other games by two players against each other or a single player against an AI agent. To make the single player experience better, we will be implementing different approaches and algorithms to make our AI agent as close to a human as possible. In upcoming passages, we will discuss several approaches to implement and design sunch a game. Such games already exist which contain AI implemented agents which is why in the upcoming subsections we will focus on our approach to design an AI agent for Snails. Moreover, we will conclude how our agent works and how we tried to maximize its strength to win.

2 Background

2.1 Overview

Snails is a desktop game. It is similar to regular two-player board games like Tic Tac Toe, Ludo, Chess etc. Firstly, a description is provided related to the game and then the approaches toward AI implementation are discussed in this section.

2.2 Game Rules

Like most other basic board games, there are conditions like win, lose and draw. A player can just move once in a turn and then the turn changes. Players can move “their snail” up, right, down and left but cannot move diagonally. A player can win only if he gets a bigger score as compared to his opponent or the game ends in a draw if the score is equal. In this game, the additional constraints make the game more interesting. The score increases if the player (snail) visits the not-visited boxes of the board. The score is not counted if the player visits already-visited coordinates. Moreover, if the player has already visited the neighbour boxes in straight steep then it will not just move one step, it will move to the end position i.e. the last coordinate of the explored area.

2.3 AI Agent

In this game, an AI agent has also been implemented. The agent will be smart enough to try to get maximum score on each single move. The agent moves each step, from its available set of moves which is its best move. The possible best moves could be calculated with different techniques by implementing already designed algorithms or by one of our own. To make an AI agent perfect, we could train an agent by learning from opponents moves but in this game we are not going to implement this approach. The agent will be intelligent enough since the start of the game, to tackle each single opponent move with its best move. In the upcoming passages we will discuss in detail, how the best moves are being generated.

2.4 Min-Max Algorithm

Minmax is a recursive algorithm which is usually used to take next move in a game. It generates all possible moves for an AI agent as well as for opponent player hence, making a tree structure (Figure 1) of all possible moves in a game. It starts tracking back to root node returning optimal values from each branch of a tree, once it reaches to the leaf nodes. Generally Min-Max does the following things:

- Returns a value i-e +1, 0 or -1, if a leaf node is found. Leaf nodes calculate their values based on agent’s condition i-e win, lose, draw or continue.
- Visits all boxes on the board.
- The minmax function is called on each box.
- On each node it evaluates the returning values from each branch of that node and returns the best value.

Min-Max algorithm is also used in game theory and decision making to find the best move for a player, supposing that the opponent also plays optimally (*Minimax Algorithm in Game Theory*, 2018). This algorithm is mostly used in two-player turn-based games i.e. Tic-Tac-Toe, Chess, Snails etc.

In Min-Max algorithm, the two players are titled as maximizer and minimizer. The maximizer attempts to get the highest possible score while the minimizer is supposed to get the lowest possible score. A value is associated with every board. If it is the maximizer’s turn then, the score of the board will tend to be some positive or highest value and if it is the minimizer’s turn, then the score of the board will tend to be some negative or minimum value.

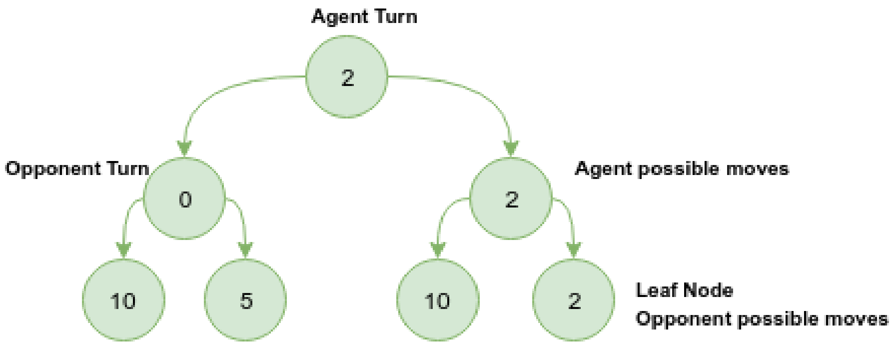

Figure 1: Min-Max Tree

Consider a scenario in which we have four leaf nodes(Figure 1) and each node is associated with a value i-e 10, 5, 2. Now these values will be returned to parent nodes depending on turn i-e minimizer or maximizer, using depth first search. Let's imagine maximizer is at the root level of the tree and the minimizer is at the next level. According to the algorithm discussed, we will pick the lowest possible value from leaf nodes because the algorithm ends with minimizer's turn. Now, there are two branches originated from each node at second level of the tree therefore, two values will be picked from leaf nodes. The minimizer has now choice to choose from 10, 5 and 10, 2. As it is a minimizer, it will choose 0 from one branch and 2 from other branch. Now it's the maximizer's turn and being the maximizer, you would choose the largest possible value, which is 2.

2.5 A* Algorithm

A* is the one of the most widely used and popular methods for finding the shortest path between two points. A* algorithm is an extension of Dijkstra's algorithm with some features of breadth-first-search. It introduces a heuristic into a regular searching algorithm, basically planning ahead at each period so that the best decision is made.

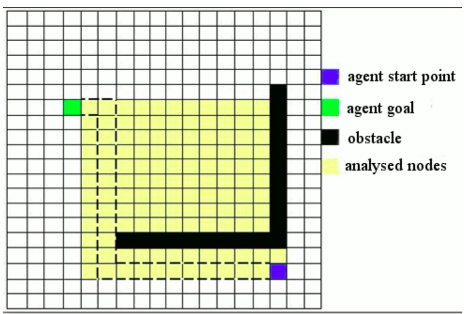

Figure 2: A* Algorithm

The important features of the A* algorithm are the construction of a "closed list" to record areas previously evaluated, a "fringe list" to record areas that are being evaluated, and the calculation of distances toured from the start point with estimated distances to the final point (goal). The fringe list, also called the open list, is a list of all places directly adjacent to the agent coordinate. The closed list is a record of all places which have been discovered and evaluated. Figure 2 gives an idea about working of A*. The detailed discussion about working of A* in this game is discussed in the upcoming passages.

3 Game Implementation

3.1 Backend Game Representation

The backend is basically how the game works, updates and changes. As Snails is a two players' game, there should be two different representations for each player at the backend because both players try to maximize the number of boxes and we have to maintain the visited boxes for each player and also have to maintain unvisited boxes, separately. To maintain it all, in the beginning, we create an 8x8 board and initialize it with zeros (not visited) and place the two players at two corners, first player at lower left corner which is board's coordinate (8, 1) and second player at top right corner which is board's coordinate (1, 8). The representation for each player is his turn, to keep things simple. First player's turn is 11 and second player's turn is 22, so we place these numbers at their positions on the board. As the players make their move, the previous box will be marked as "visited" by that player. Backend representation of the first player's visited boxes is 1 and for the second player is 2. Under given matrix shows how board in this game is being represented at the backend.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 22 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 11 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 3: Backend Game Board Representation

3.2 UI of the Game

We have already discussed the detailed representation of game at the backend. Initially game loads an image with a background (blue wall paper) and against every number in the board, we use an images to show it to the users. Below is the graphical representation of the board:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 22 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 11 & 0 & 0 & 0 & 0 \end{bmatrix}$$

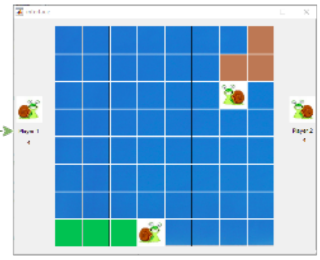

Figure 4: Front end Representation

In the figure above, the blue color represent unvisited boxes, brown color represents the boxes visited by the second player (agent) which is 22 and the green boxes represent the boxes visited by the first player which is 11.

3.3 AI Implementation

Several approaches are implemented in this game to make AI agent behave like human. Under given sections give a detail overview of each implemented technique.

3.3.1 First Approach(Min-Max + Euclidean Distance)

The Min-Max algorithm always make an optimal move if we let it complete its operations i-e leaving it to reach the leaf nodes and then return a best value from the leaf nodes. It can be used for games like Tic-Tac-Toe because in this case it is a small board (3x3) but a game like Snails which has a board of size 8x8, we cannot let the Min-Max algorithm reach the leaf nodes because it will probably take hours to make a move for a single turn as discussed in the under given paragraph. So, after considering the response time, we should select a reasonable depth for the Min-Max algorithm.

Applying Min-Max algorithm on this game without specifying the depth takes a lot of time. It may also go up to infinity at certain stages of the game. We want it to make a reasonable optimal move in a reasonable time so that the players do not have to wait too long. To gauge the time the algorithm will spend, we tried different depths in Min-Max algorithm. The different depths with the time taken to make a move are:

- Depth 1: With depth equal to 1, it responded very quickly i.e. in milliseconds.
- Depth 3: With depth equal to 3, it took some time to respond i.e. a few seconds.
- Depth 5: With depth 5, the response time was around couple of seconds.
- Depth 8: With depth 8, it took more than 4 seconds.
- Depth 15: Increasing the depth to 15, we waited for almost half an hour and did not get the response.

Now we know that Min-Max algorithm will reach the specified depth and will return a value but Min-Max algorithm will not take the best optimal move because we are only letting it to reach a certain depth (specified depth). To take an optimal move, we have to generate a heuristic which will be capable enough to take an optimal move or close to the optimal move. For that, we generated a heuristic which was to get closer to the opponent snail and follow it i.e. block it. To get close to the opponent we used Euclidean distance formula to compute minimum possible steps to reach to the opponent coordinates from agent coordinates. Once the Min-Max reaches to its specified depth it adds the generated heuristic into the current score of agent and starts backtracking. Under given diagram shows how this approach generates heuristic and adds it to the returning score in Min-Max.

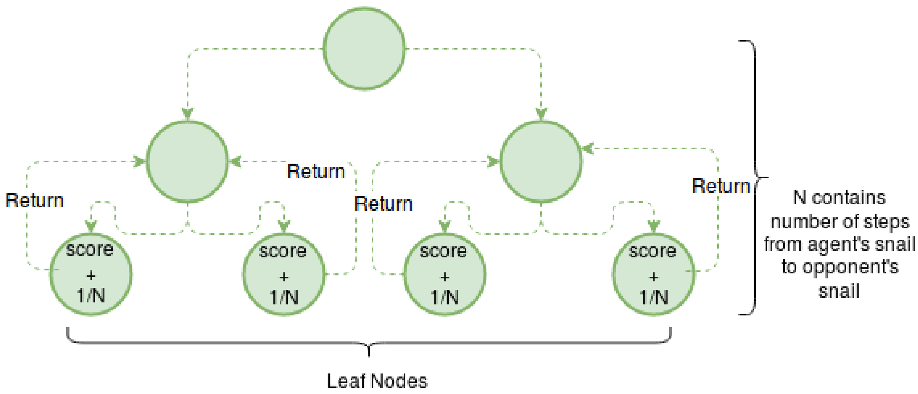

Figure 5: Approaching to the Opponent

The distance between two nodes is calculated according to the following formula.

$$N = \sqrt{abs(x_2 - x_1) + abs(y_2 - y_1)} \tag{1}$$

Limitations in this approch: At the start, the heuristic seemed to be working well but at a certain stage we noticed that the AI agent was just trying to get closer to the opponent snail and was following it but never trying to win the game and to trap the opponent player i.e. to maximize its score.

3.3.2 Second Approach (Min-Max + A*)

We have already discussed how Min-Max algorithm works in this game but in this scenario the startegy for generating heuristic has changed. This discussion explains how A* is being implemented to generate heuristic to make AI agent more efficient. The heuristic is generated by finding the distances from AI agent's snail and opponent's snail to every empty box in the board and also find whether agent can reach there first or not. If the steps required to reach to the empty block of AI agent is equal to or less than the steps needed by the opponent then make increment in the score of AI agent by one for each block and return it when all the distances are calculated.

When the heuristic is generated for each leaf node in Min-Max tree, it is added to the value returned by the Min-Max algorithm and on the basis of that value (Min-Max value + heuristic value), the AI agent decides where to move the snail.

Limitations in this approach: The second approach is better than the first approach in terms of efficiency but it takes a couple of minutes to generate next move for AI agent becuase keeping depth to 8 in Min-Max takes some time and time taken by A* algorithm to compute distance from snail to every block for each leaf node is three to four times more then Min-Max algorithm. Keeping depth of tree up to 2 takes a resonable time but then we have to compromise on the efficiency of AI agent.

3.3.3 Third Approach (A*)

We have seen that the previous approach of using Min-Max with A* was causing some problems. In this approach decision of next move is totally based on the heuristic calculated by A* algorithm. First it finds out all of the possible children of a snail and for each children A* algorithm is used to calculate distance from a particular child to all possible free blocks. This calculated distance is also compared with the distance of opponent snail to a free block. If the distance of that child is less then the distance of opponent then this block is added in to the scores of agent. The decision of next move is based upon the child with maximum number of blocks.

Let's consider a scenario in which the coordinates of a child are (2, 7) and we want to compute the distance to a free block of coordinates (4, 4). Under given figure shows how A* finds shortest distance from a child to a free block.

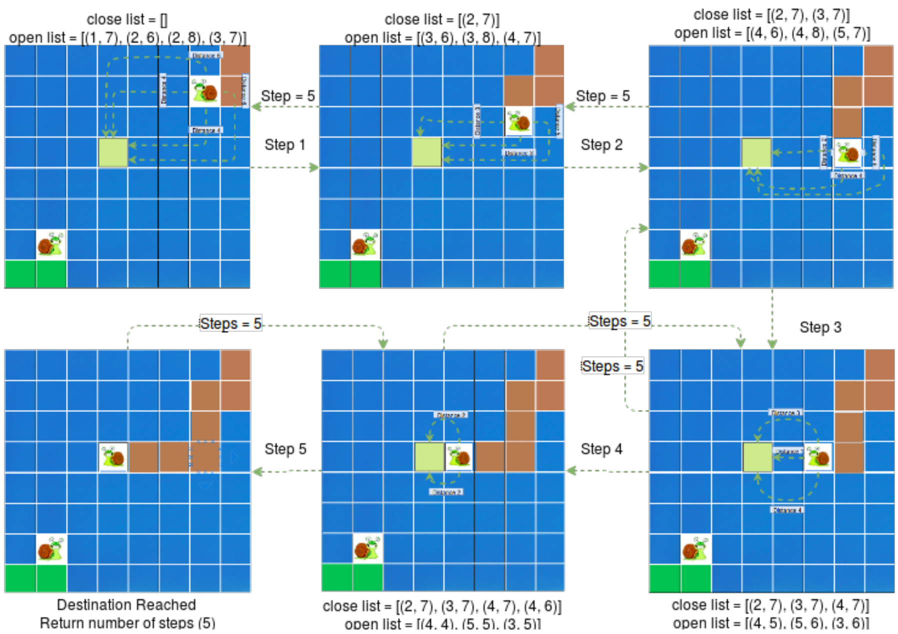

Figure 6: A* shortest path finding

Initially the close list is empty. A* generates all possible moves and put them in the open list and if any of the box among these moves is in close list it discards that move. It chooses a block which takes it more rapidly to the destination i-e minimum distance to the destination. In the same way it generates children at every single stage, picks the optimum path and makes increment in number of steps until it reaches to the destination. We see that in the above case 5 steps are required to snail to reach a free block. If these steps are less than or equal to the number of steps required by opponent to reach the free blocks then the score of agent will be incremented by 1. In the same way it repeates that process and calculates number of steps to reach all of the free blocks. Decision of next move depends upon child with maximum scores.

3.3.4 Performance Analysis

We have discussed all the techniques that we have implemented in this game to make AI agent look intelligent. The first technique was efficient in term of time but it was not intelligent enough to pick the best moves. The second approach was more intelligent than the first one but it takes a lot of time to make a move which does not make a game interactive. The third technique that is used is more intelligent and more efficient than the other two techniques. Under given graphs show the comparative analysis of all implemented techniques in term of time and efficiency.

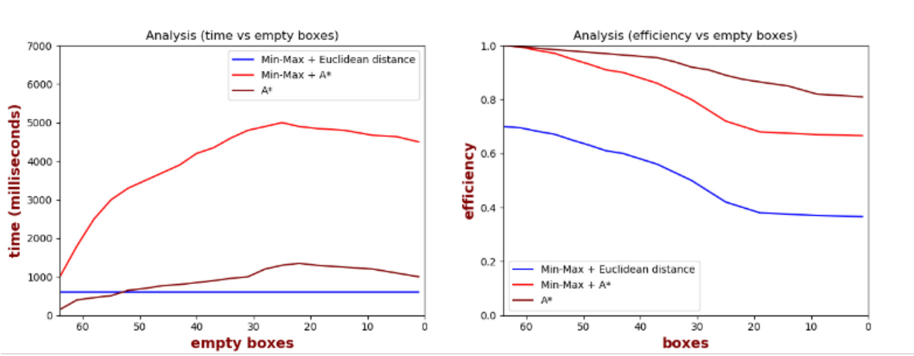

Figure 7: Performance Analysis

After analyzing all the techniques the final version of game is based on third approach.

4 Testing

AI agent was tested against human to figure out what decisions are made by agent at different stages of the game. Under given pictures show some moves of AI agent against human.

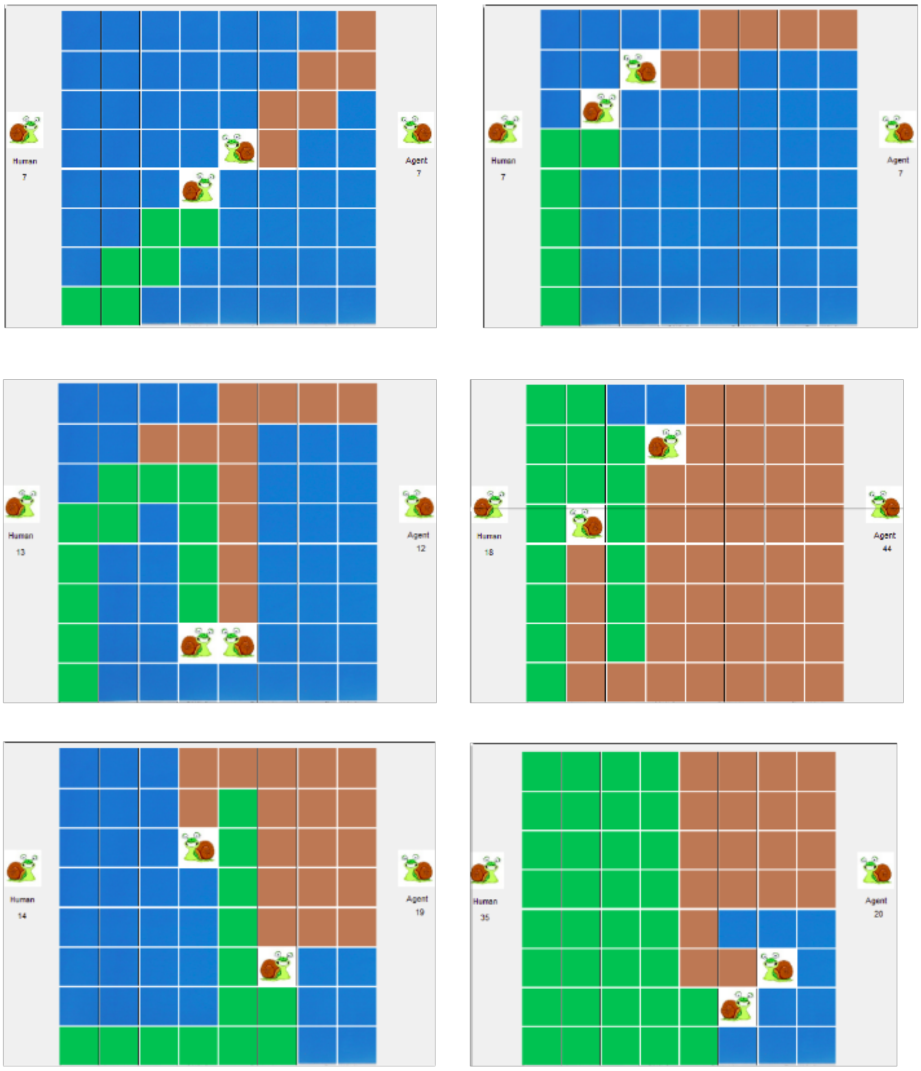

Figure 8: Test Cases

5 Appendix

Since, many functions are working at the backend therefore, sequence of each function call is shown in under given flow chart.

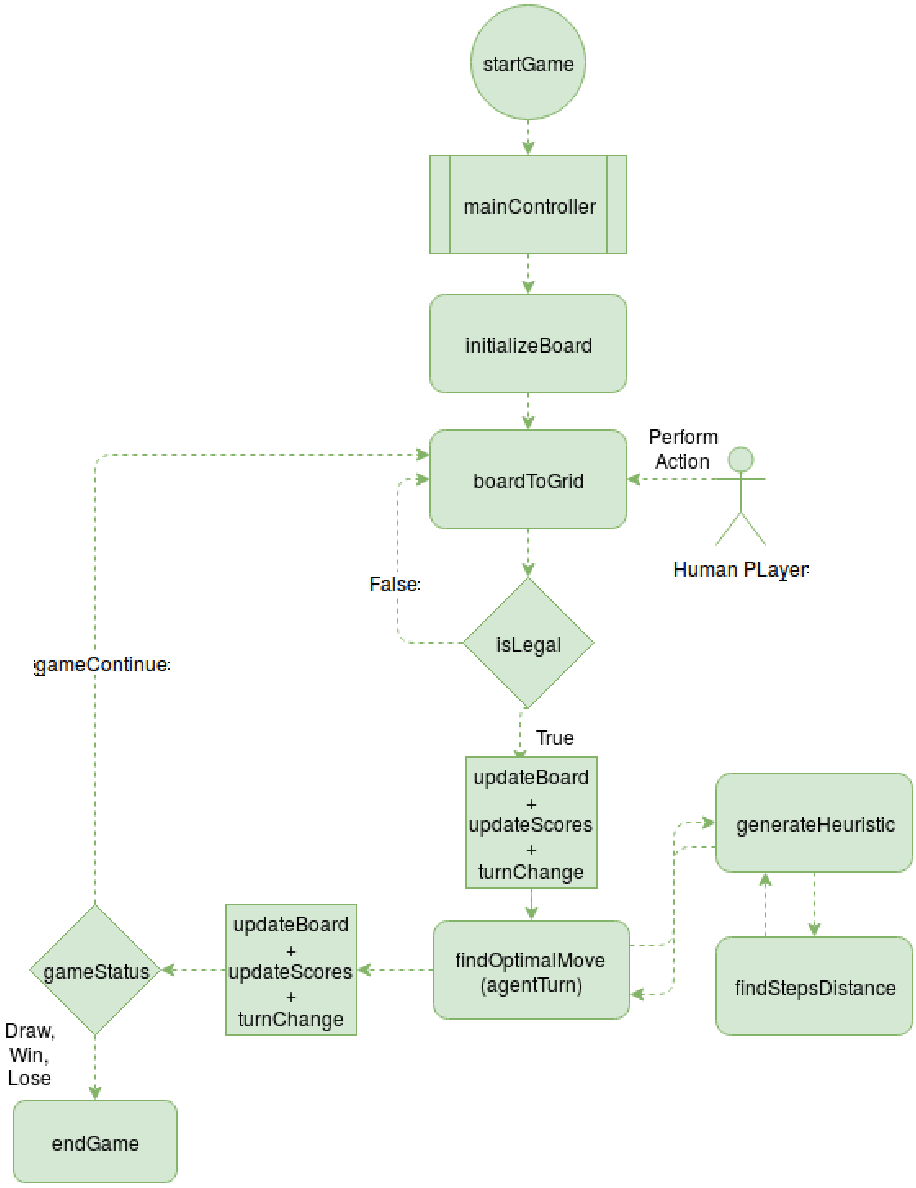

Figure 9: Sequence of function calls

The main functions with their code are listed below.

Start Game

```
function varargout = startGame(varargin)
% INTERFACE MATLAB code for interface.fig
%      INTERFACE, by itself, creates a new INTERFACE or raises the existing
%      singleton*.
%
%      H = INTERFACE returns the handle to a new INTERFACE or the handle to
%      the existing singleton*.
%
%      INTERFACE('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in INTERFACE.M with the given input arguments.
%
%      INTERFACE('Property','Value',...) creates a new INTERFACE or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before interface_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to interface_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help interface

% Last Modified by GUIDE v2.5 27-Mar-2018 11:19:35

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @interface_OpeningFcn, ...
                   'gui_OutputFcn',  @interface_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
```

```
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before interface is made visible.
function interface_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to interface (see VARARGIN)

% Choose default command line output for interface
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes interface wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = interface_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;
```

```
mainController(handles)
```

Main Controller

```
function mainController(handle)
% initilizing game setup
rows = 8; cols = 8;
board = initializeBoard(rows, cols);
board(1, 8) = 22;
board(8, 1) = 11;
grid = boardToGrid( board );
imshow( grid );

turn = 11;
agentTurn = 22;
depth = 1;

% initially agent score and opponent score is 1
scoreOpp = 1;
scoreAgent = 1;
snail1 = imread('snail1.jpg');
imshow(snail1, 'Parent', handle.axes1);
snail2 = imread('snail2.jpg');
imshow(snail2, 'Parent', handle.axes2);
% while game continuos
while 1
    % if it is not agent
    if( turn ~= agentTurn )
        [x, y] = ginput(1);
        temp = y;
        y = floor(x/100)+1;
        x = floor(temp/100)+1;

        [xx, yy] = find (board == 11);
        [ islegal, movement ] = isLegal(board, x, y, 11); % if its a legal move
        if ( islegal == false )
            msgbox('Invalid Input', 'Error','error')
        else
            board(xx, yy) = 1;
            [ x, y ] = slideSnail( board, x, y, turn, movement );% slide snail if possible
            % if board(x, y) ~= 1
                % scoreOpp = scoreOpp + 1;
            % end
            board(x, y) = 11;
            turn = changeTurn(turn); % change turn
            updateScores(board, handle);
        end
    % if it is agent move
```

```
    else
       % [ board, value, scoreAgent ] = searchTree( board,
           turn, agentTurn, depth, scoreAgent, scoreOpp );
       [ board, scoreAgent ] = findOptimalMove( board,
          turn, agentTurn, scoreAgent, scoreOpp ); % find
          optimal move using heuristic
       turn = changeTurn(turn); % chagne turn
    end

    grid = boardToGrid(board); % convert board to grid
    imshow(grid); % showing grid
    updateScores(board, handle);

    % stop game if no possible move
    score = gameStatus( board, agentTurn );
    if score == 10 || score == -10
        if score == 10
            disp('agent won');
        else
            disp('opponent won');
        end
        break;
    end
end
end

function updateScores(board, handle)
       % calculating score of agent
       [ agent_blocks, dummy ] = find(board == 2);
       [ scoreAgent, dummy ] = size(agent_blocks);
       scoreAgent = scoreAgent + 1;
       set(handle.Player_2,'String',scoreAgent);

       % calculating score of opponent
       [ oppo_blocks, dummy ] = find(board == 1);
       [ scoreOpp, dummy ] = size(oppo_blocks);
       scoreOpp = scoreOpp + 1;
       set(handle.Player_1, 'String', scoreOpp);
end
```

Board to Grid

```
function grid = boardToGrid(board)
    [rows, cols] = size(board);
    size_rows = rows * 100;
    size_cols = cols * 100;

    %read images
    grid = imread('boardImage.jpg');
    turn1 = imread('snail1.jpg');
```

```
    turn2 = imread('snail2.jpg');
    snail1_mark = imread('snail1_mark.jpg');
    snail2_mark = imread('snail2_mark.jpg');
    no_mark = imread('no_mark.jpg');

    %place icon of second player
    [x, y] = find(board == 22);
    x = (x * 100) - 100;
    y = (y * 100) - 100;
    if x == 0
        x = x + 1;
    end
    if y == 0
        y = y + 1;
    end

    %place icon
    grid(x:(x-1) + 100, y:y+100, :) = turn2(1:100, 1:101,
        :);

    %place icon of first player
    [x, y] = find(board == 11);
    x = (x * 100) - 100;
    y = (y * 100) - 100;
    if x == 0
        x = x + 1;
    end
    if y == 0
        y = y + 1;
    end
    %place icon
    grid(x:(x-1) + 100, y:y+100, :) = turn1(1:100, 1:101,
        :);

    %find marks of first players
    [ x, y ] = find(board == 1);
    [ numberOfMarks, dummy ] = size(x);
    %place marks on grid
    for i = 1:numberOfMarks
        x_Cord = x(i); y_Cord = y(i);
        x_Cord = (x_Cord * 100) - 100;
        y_Cord = (y_Cord * 100) - 100;
        if x_Cord == 0
            x_Cord = x_Cord + 1;
        end
        if y_Cord == 0
            y_Cord = y_Cord + 1;
        end
        %place icon
        grid(x_Cord:(x_Cord-1) + 100, y_Cord:y_Cord+100,
```

```
                :) = snail1_mark(1:100, 1:101, :);
    end

    %find marks of second players
    [ x, y ] = find(board == 2);
    [ numberOfMarks, dummy ] = size(x);
    %place marks on grid
    for i = 1:numberOfMarks
        x_Cord = x(i); y_Cord = y(i);
        x_Cord = (x_Cord * 100) - 100;
        y_Cord = (y_Cord * 100) - 100;
        if x_Cord == 0
            x_Cord = x_Cord + 1;
        end
        if y_Cord == 0
            y_Cord = y_Cord + 1;
        end
        %place icon
        grid(x_Cord:(x_Cord-1) + 100, y_Cord:y_Cord+100,
            :) = snail2_mark(1:100, 1:101, :);
    end

    % find place where no player can reach
    [ x, y ] = find(board == -1);
    [ numberOfMarks, dummy ] = size(x);
    %place marks on grid
    for i = 1:numberOfMarks
        x_Cord = x(i); y_Cord = y(i);
        x_Cord = (x_Cord * 100) - 100;
        y_Cord = (y_Cord * 100) - 100;
        if x_Cord == 0
            x_Cord = x_Cord + 1;
        end
        if y_Cord == 0
            y_Cord = y_Cord + 1;
        end
        %place icon
        grid(x_Cord:(x_Cord-1) + 100, y_Cord:y_Cord+100,
            :) = no_mark(1:100, 1:101, :);
    end

    %draw lines or make board
    printLine = 100;
    for i = 1:rows-1
        %draw vertical lines
        printLineColumn = printLine * i;
        grid(:, printLineColumn, :) = 0;
        grid(:, (printLineColumn+1), :) = 255;
        grid(:, (printLineColumn+2), :) = 255;
        grid(:, (printLineColumn+3), :) = 255;
```

```

        %draw horizontal lines
        printLineRow = printLine * i;
        grid(printLineRow, :, :) = 255;
        grid((printLineRow+1), :, :) = 255;
        grid((printLineRow+2), :, :) = 255;
        grid((printLineRow+3), :, :) = 255;
    end
end
```

Is Legal Move

```
function [ islegal, movement ] = isLegal(board, x, y, turn)
    movement = 'wrongMove';
    if turn == 22
        myMark = 2;
    else
        myMark = 1;
    end

    [xx, yy] = find (board == turn);
    if (x > 8 || x < 1 || y > 8 || y < 1)
         islegal = false;
         return;
    end
    %down movement
    if ((x == xx+1 && y == yy) && (board(x, y) == 0 || board(x, y) == myMark))
        islegal = true;
        movement = 'down';
        return;
    end
    %right movement
    if ((x == xx && y == yy+1) && (board(x, y) == 0 || board(x, y) == myMark))
        islegal = true;
        movement = 'right';
        return;
    end
    %up movement
    if ((x == xx-1 && y == yy) && (board(x, y) == 0 || board(x, y) == myMark))
        islegal = true;
        movement = 'up';
        return;
    end
    %left movement
    if ((x == xx && y == yy-1) && (board(x, y) == 0 || board(x, y) == myMark))
```

```
            islegal = true;
            movement = 'left';
            return;
        end

        %code for long jump

        %horizontal jump
        if ( x == xx && y > yy) %horizontal jump toward right
            for i = (yy + 1):y
                if(board(x, i) ~= myMark)
                    islegal = false;
                    return;
                end
            end
            islegal = true;
            movement = 'right';
            return;
        end
        if (x == xx && y < yy) %horizontal jump left
            for i = y:(yy - 1)
                if(board(x, i) ~= myMark)
                    islegal = false;
                    return;
                end
            end
            islegal = true;
            movement = 'left';
            return;
        end

        %vertical jump
        if (y == yy && x > xx) %jump down
            for i = (xx + 1):x
                if(board(i, y) ~= myMark)
                    islegal = false;
                    return;
                end
            end
            islegal = true;
            movement = 'down';
            return;
        end

        if (y == yy && x < xx) %jump up
            for i = x:(xx - 1)
                if(board(i, y) ~= myMark)
                    islegal = false;
                    return;
                end
```

```
        end
        islegal = true;
        movement = 'up';
        return;
    end
    islegal = false;
end
```

Find Optimal Move

```
function [ bestBoard, score ] = findOptimalMove( board,
    turn, agentTurn, scoreAgent, scoreOpp )

    % generate all possible immidiate children
    [ children, scores ] = generateChildren( board, turn,
        agentTurn, scoreAgent, scoreOpp );
    [ r c l ] = size( children );
    maxScoreList = zeros(1, l);

    % find heuristic for all children
    for i=1:l
        [ maxScore, modifiedBoard ] = generateHeuristic(
            children(:, :, i), turn );
        children(:, :, i) = modifiedBoard;
        maxScoreList(1, i) = maxScore + scores(1, i);
    end

    % find children with maximum heuristic
    [ maxScore, index ] = max(maxScoreList);
    if maxScoreList(1, :) == maxScoreList(1, index)
        [ dummy, index ] = max(scores);
    end

    % return board which can give you the best heuristic
        with scores
    bestBoard = children(:, :, index);
    score = scores(1, index);
end
```

Generate Children

```
function [ children, scores ] = generateChildren( board,
    turn, agentTurn, scoreAgent, scoreOpp )
   [ snail_x, snail_y ] = find( board == turn );
   % initialize children list
   children = zeros(8, 8, 1);
   % initialize scores list
   scores = zeros(1, 1);

```

```
%child generated by up movement
temp_x = snail_x - 1;
[ islegal , movement ] = isLegal( board , temp_x ,
    snail_y , turn );

% make a temp children list to save data while
    increasing children list
% size
tempChildren = children;
if islegal
    % increment in scores based on turn
    mark = floor(turn/10);
    if turn == agentTurn
        if board(temp_x , snail_y) == mark
            scores(1, 1) = scoreAgent;
        else
            scores(1, 1) = scoreAgent + 1;
        end
    else
        if board(temp_x , snail_y) == mark
            scores(1, 1) = scoreOpp;
        else
            scores(1, 1) = scoreOpp + 1;
        end
    end

    % generate children
    tempChildren(:, :, 1) = board(:, :);
    if turn == 11
     tempChildren(snail_x , snail_y , 1) = 1;
    else
     tempChildren(snail_x , snail_y , 1) = 2;
    end
    %check sliding
    [ x, y ] = slideSnail( tempChildren(:, :, 1),
        temp_x, snail_y , turn , movement );
    tempChildren(x, y, 1) = turn;
    children(:, :, 1) = tempChildren;
end

%child generated due to right move
temp_y = snail_y + 1;
[ islegal , movement ] = isLegal(board , snail_x , temp_y
    , turn);
tempScores = scores;

if islegal

    % increment in score based on condition
    if scores(1, 1) ~= 0
```

```
            scores = zeros(1, 2);
            scores(1, 1) = tempScores(1, 1);
        end

        [ temp length ] = size(scores);
        mark = floor(turn/10);
        if turn == agentTurn
            if board(snail_x, temp_y) == mark
                scores(1, length) = scoreAgent;
            else
                scores(1, length) = scoreAgent + 1;
            end
        else
            if board(snail_x, temp_y) == mark
                scores(1, length) = scoreOpp;
            else
                scores(1, length) = scoreOpp + 1;
            end
        end

        % generation of children
        if tempChildren(snail_x, snail_y, 1) ~= 0
            children = zeros(8, 8, 2);
            children(:, :, 1) = tempChildren(:, :, 1);
        end

        [ temp temp length ] = size(children);
        children( :, :, length ) = board(:, :);

         if turn == 11
             children(snail_x, snail_y, length) = 1;
         else
             children(snail_x, snail_y, length) = 2;
         end
        %check sliding
        [ x, y ] = slideSnail( children(:, :, 1), snail_x, temp_y, turn, movement );
        children(x, y, length) = turn;
        tempChildren = children;
    end

    %child generated due to down movement
    temp_x = snail_x + 1;
    [ islegal, movement ] = isLegal(board, temp_x, snail_y, turn);
    tempScores = scores;

    if islegal

        % increment in score
```

```
        [ temp length ] = size(scores);
        if scores(1, 1) ~= 0
            length = length + 1;

            scores = zeros(1, length);
            for score = 1:length - 1
                scores(1, score) = tempScores(1, score);
            end
        end

        mark = floor(turn/10);
        if turn == agentTurn
            if board(temp_x, snail_y) == mark
                scores(1, length) = scoreAgent;
            else
                scores(1, length) = scoreAgent + 1;
            end
        else
            if board(temp_x, snail_y) == mark
                scores(1, length) = scoreOpp;
            else
                scores(1, length) = scoreOpp + 1;
            end
        end

        % generation of scores
        tempScores = scores;
        [ temp temp length ] = size(tempChildren);
        if tempChildren(snail_x, snail_y, 1) ~= 0
            length = length + 1;

            children = zeros(8, 8, length);
            for child=1:length - 1
                children(:, :, child) = tempChildren(:, :,
                    child);
            end
        end

        children(:, :, length) = board(:, :);
        if turn == 11
            children(snail_x, snail_y, length) = 1;
        else
            children(snail_x, snail_y, length) = 2;
        end

        %check sliding
        [ x, y ] = slideSnail( children(:, :, 1), temp_x,
           snail_y, turn, movement );
        children(x, y, length) = turn;
        tempChildren = children;
```

```
    end

    %child generated due to left movement
    temp_y = snail_y - 1;
    [ islegal , movement ] = isLegal(board, snail_x , temp_y
        , turn);
    tempScores = scores;

    if islegal

        % increment in socres
        [ temp length ] = size(scores);
        if scores(1, 1) ~= 0
            length = length + 1;

            scores = zeros(1, length);
            for score = 1:length - 1
                scores(1, score) = tempScores(1, score);
            end
        end

        mark = floor(turn/10);
        if turn == agentTurn
            if board(snail_x , temp_y) == mark
                scores(1, length) = scoreAgent;
            else
                scores(1, length) = scoreAgent + 1;
            end
        else
            if board(snail_x , temp_y) == mark
                scores(1, length) = scoreOpp;
            else
                scores(1, length) = scoreOpp + 1;
            end
        end

        % generation of children
        [ temp temp length ] = size(tempChildren);
        if tempChildren(snail_x , snail_y , 1) ~= 0
            length = length + 1;

            children = zeros(8, 8, length);
            for child=1:length - 1
                children(:, :, child) = tempChildren(:, :,
                    child);
            end
        end

        children(:, :, length) = board(:, :);
        if turn == 11
```

```
            children(snail_x, snail_y, length) = 1;
        else
            children(snail_x, snail_y, length) = 2;
        end

        %check sliding
        [ x, y ] = slideSnail( children(:, :, 1), snail_x,
            temp_y, turn, movement );
        children(x, y, length) = turn;
    end
end
```

Slide Snail

```
function [ tempX, tempY ] = slideSnail( board, x, y, turn
    , movement )
    if turn == 22
        myMark = 2;
    else
        myMark = 1;
    end
    tempX = x; tempY = y; isSlide = false;

    %if need to slide down
    if strcmp(movement, 'down') == 1
        while board(tempX, tempY) == myMark
            isSlide = true;
            tempX = tempX + 1;
            if tempX == 9
                break;
            end
        end
        if isSlide
            tempX = tempX - 1;
        end
        return;
    end
    %if need to slide up
    if strcmp( movement, 'up') == 1
        while board(tempX, tempY) == myMark
            isSlide = true;
            tempX = tempX - 1;
            if tempX == 0
                break;
            end
        end
        if isSlide
            tempX = tempX + 1;
        end
        return;
```

```
    end

    %if need to slide right
    if strcmp( movement, 'right') == 1
        while board(tempX, tempY) == myMark
            isSlide = true;
            tempY = tempY + 1;
            if tempY == 9
                break;
            end
        end
        if isSlide
            tempY = tempY - 1;
        end
        return;
    end

    %if need to slide left
    if strcmp( movement, 'left') == 1
        while board(tempX, tempY) == myMark
            isSlide = true;
            tempY = tempY - 1;
            if tempY == 0
                break;
            end
        end
        if isSlide
            tempY = tempY + 1;
        end
        return;
    end
end
```

Generate Heuristic

```
function [ maxScore, board ] = generateHeuristic( board,
    turn )

    % initialize variables
    maxScore = 0;
    [x, y] = find(board == 0);

    % find opponent turn i-e not agent
    if turn == 11
        oppo_turn = 22;
    else
        oppo_turn = 11;
    end

    % find number of empty spaces in board
```

```
    [ length dummy ] = size([x, y]);
    % check distancea of every free block from agent and opponent
    for i=1:length
        [ snail_x , snail_y ] = find(board == turn); % finding position of snail
        % find how many steps agent is away from free block
        [ my_steps , possibility_1 ] = findStepDistance_dummy( board , [ snail_x , snail_y ], [x(i), y(i)], turn , 0, 0, 500, 0 );
        % if there exist a possible shortest path from agent to free block
        if possibility_1 == 0
            % verifying that there is no block from free block to agent
            dummy_board = board;
            dummy_board( snail_x , snail_y ) = 0;
            dummy_board( x(i), y(i) ) = turn;
            % check path possibility from free block to snail
            [ my_steps , possibility_1 ] = findStepDistance_dummy( dummy_board , [x(i), y(i)], [ snail_x , snail_y ], turn , 0, 0, 500, 0 );
            % if possibility1 == 0
            %    board(x(i), y(i)) = floor(oppo_turn/10);
            % end
        end
        [ snail_x , snail_y ] = find(board == oppo_turn); % finding position of opponent snail
        % finding distance from opponent snail to free block
        [ oppo_steps , possibility_2 ] = findStepDistance_dummy( board , [ snail_x , snail_y ], [x(i), y(i)], oppo_turn , 0, 0, 500, 0 );
        if possibility_2 == 0 % if there is no path from opponent snail to free block
            dummy_board = board;
            dummy_board( snail_x , snail_y ) = 0;
            dummy_board( x(i), y(i) ) = oppo_turn;
            % also check path possibility from free block to snail
            [ my_steps , possibility_2 ] = findStepDistance_dummy( dummy_board , [x(i), y(i)], [ snail_x , snail_y ], oppo_turn , 0, 0, 500, 0 );
            % if possibility2 == 0
            %    board(x(i), y(i)) = floor(turn/10);
```

```
            % end
        end

        % if there is no possible path for both snail to the free block
        % then this block comes under no territory
        if possibility_1 == 0 && possibility_2 == 0
            board(x(i), y(i)) = -1;
        else if possibility_1 == 0 && possibility_2 == 1 % if only opponent can reach
                board(x(i), y(i)) = floor(oppo_turn/10);
            else if possibility_1 == 1 && possibility_2 == 0 % if only agent can reach
                board(x(i), y(i)) = floor(turn/10);
                end
            end
        end

        if my_steps <= oppo_steps && possibility_1 == 1
            maxScore = maxScore + 1;
        end
    end
end
```

Find Step Distance

```
function [ steps, possibility ] = findStepDistance_dummy( board, snail, free_block, turn, CLOSE_LIST, steps, depth, N )

    if depth == 0 || steps > 20
        possibility = 0;
        return;
    end
    OPEN_LIST = zeros(1, 1); % maintains list where snail can move
    %STEP_DISTANCE = zeros(1, 1); % distance of each neighbour block to free block
    MAX_MOVEMENTS = 4; % maximum number of movements that a snail can have
    MOVEMENTS = zeros(1, 1);

    % get snail coordinates
    snail_x = snail(1, 1);
    snail_y = snail(1, 2);

    % get coordinates of free blocks
    free_block_x = free_block(1, 1);
    free_block_y = free_block(1, 2);

```

```
    % check all possible movements
    while MAX_MOVEMENTS ~= 0
        if MAX_MOVEMENTS == 4 % check up movment
            x = snail_x - 1; y = snail_y;
        else if MAX_MOVEMENTS == 3 % checks right movement
                x = snail_x; y = snail_y + 1;
        else if MAX_MOVEMENTS == 2 % check down movement
                x = snail_x + 1; y = snail_y;
            else % check left movement
                x = snail_x; y = snail_y - 1;
            end
            end
        end

        [ islegal, movement ] = isLegal(board, x, y, turn);
        [ checkSlide_x, checkSlide_y ] = slideSnail(board, x, y, turn, movement);
        block_number = (checkSlide_x - 1)*8 + checkSlide_y;
        member = ismember(block_number, CLOSE_LIST);
        if member == 1 % if it is not already visited
            islegal = false;
        end
        if islegal % if it is a valid move
            %if we have reached to the destination
            if (x == free_block_x) && (y == free_block_y)
                possibility = 1;
                steps = steps + 1;
                return;
            end

            % need to increse the size of open list if we get one more
            % possible move
            [ dummy length ] = size(OPEN_LIST);
            if OPEN_LIST(1, length) ~= 0
                length = length + 1;
            end

            % calculate block number and place it into the open list
            block_number = (x - 1)*8 + y;
            member = ismember(block_number, CLOSE_LIST);
            if member == 0 % if it is not already visited
                OPEN_LIST(1, length) = block_number;
                if strcmp(movement, 'up') == 1
                    MOVEMENTS(1, length) = 1;
                else if strcmp(movement, 'right') == 1
```

```
                        MOVEMENTS(1, length) = 2;
                    else if strcmp(movement, 'down') == 1
                            MOVEMENTS(1, length) = 3;
                        else
                            MOVEMENTS(1, length) = 4;
                        end
                    end
                end
            end
        end
        MAX_MOVEMENTS = MAX_MOVEMENTS - 1;
    end

    % if there is no possible way to reach ot a particular coordinate
    if OPEN_LIST(1, 1) == 0
        possibility = 0;
        return;
    end

    distances = zeros(1, 1);
    rows = zeros(1, 1);
    cols = zeros(1, 1);
    indexes = zeros(1, 1);

    % find optimal next block to get benifit of sliding as well
    [ dummy length ] = size(OPEN_LIST);
    for i=1:length
        minStepDistanceBlock = OPEN_LIST(1, i);
        best_movement = MOVEMENTS(1, i); % finding best movement number based on index
        row = ceil(minStepDistanceBlock/8); % getting row number of that block
        col = minStepDistanceBlock - (row - 1) * 8;% getting column number of that block

        best_direction = findBestDirection( best_movement );
        [ row, col ] = slideSnail( board, row, col, turn, best_direction );
        %calculate minimum steps taken from x, y to free_x, free_y
        step_x = free_block_x - row;
        step_y = free_block_y - col;

        %if step_x < 0
        %   step_x = step_x * (-1);
        %end
        %
```

```
        %if step_y < 0
        %   step_y = step_y * (-1);
        %end

        distance = sqrt(step_x * step_x + step_y * step_y
            ); % total steps distance
        indexes(1, i) = i;
        distances(1, i) = distance;
        rows(1, i) = row;
        cols(1, i) = col;
    end

    % add snail coorinate into visited or close list
        blocks
    [ dummy, length ] = size( CLOSE_LIST );
    snail_block_number = (snail_x - 1) * 8 + snail_y;
    if CLOSE_LIST(1, length) == 0
        CLOSE_LIST(1, 1) = snail_block_number;
    else
        length = length + 1;
        CLOSE_LIST(1, length) = snail_block_number;
    end

     %[ dummy, index ] = min(distances);
    % distances = sort(distances);
    [ distances, rows, cols ] = sortDistances(distances,
        rows, cols);
    [ dummy, length ] = size(distances);
    for index = 1:length

        %if index ~= 1
        %     if distances(1, index - 1) ~= distances(1,
            index)
        %          possibility = 0;
        %          return;
        %     end
             %disp('index has changed');
             %index
        %end
        % based on best movement number find direction
        best_movement = MOVEMENTS(1, index);
        best_direction = findBestDirection( best_movement
            );

        dummy_board = board;
        [ row, col ] = slideSnail( dummy_board, rows(1,
            index), cols(1, index), turn, best_direction )
            ;
        dummy_board(row, col) = turn;
        if turn == 11
```

```
            dummy_board(snail_x, snail_y) = 1;
        else
            dummy_board(snail_x, snail_y) = 2;
        end
        dummy_steps = steps + 1;

        dummy_snail = [row, col]; % update position of snail
        % call this function again to find shortest path
        dummy_depth = depth - 1;
        dummy_N = N + 1;
        [ dummy_steps, possibility ] = findStepDistance_dummy(dummy_board, dummy_snail, free_block, turn, CLOSE_LIST, dummy_steps, dummy_depth, dummy_N);

        % if there exists a path to reach to the destination
        if possibility == 1
            steps = dummy_steps;
            return;
        else
            if N == 0 % if it is a parent node then check another child for possibility
                continue;
            else % else move toward parent i-e back recurrsion
                possibility = 0;
                steps = dummy_steps;
                return;
            end

        end
    end
    % if we donot find any possiblity to reach to the destination
    possibility = 0;
    return;
end

function [ distances, rows, cols ] = sortDistances(distances, rows, cols)

    [ dummy, length ] = size(distances);
    for i=1:length
        for j=i+1:length
            if distances(1, i) > distances(1, j)
                dummy = distances(1, i);
                distances(1, i) = distances(1, j);
                distances(1, j) = dummy;
```

```
                dummy = rows(1, i);
                rows(1, i) = rows(1, j);
                rows(1, j) = dummy;

                dummy = cols(1, i);
                cols(1, i) = cols(1, j);
                cols(1, j) = dummy;
            end
        end
    end
end
```

References

Bura, J. (2012). Ai in games. In *Pro android web game apps* (pp. 513–540). Springer.

Minimax algorithm in game theory. (2018).