AI IN GAME

The Snail Grid World

Muhammad Bilal

Sunday 5th December, 2021

Acknowledgement

We would like to express my special thanks of gratitude to my honorable lab engineer Mr. Faisal Zeeshan who provided me a golden opportunity to perform this wonderful project and gain the experience of implementing AI in games. This helped me in doing a lot of Research and i came to know about so many new things. We are really thankful to them. Secondly, we would also like to thanks my friends who helped me a lot in finalizing this project within the limited time frame.

Abstract

Artificial intelligence is progressing imperatively in field of computer science. It is making machines capable of intelligently solving problems. One of the earliest applications of AI is in the zone of game development. Particularly, artificial intelligence is regularly utilized to form opponent players in games. Early shapes of AI players frequently appeared in conventional board games, such as chess, checkers, backgammon, and tic-tac-toe. Diversions of this sort give a fully discernible and deterministic view of the game progress at any state. This permits an AI player the capacity to analyze all conceivable moves. We have embedded such intelligence in a Bot player against Human. The game is called Snail game.

Table of Contents

List of Figures

1 Player Snail Game

Snail Game is a 2D board game that is held between only two players. Both players play their turns one after the other until any of the player Won. The game could end as a draw as well.

Figure 1: 2D Snail Game.

2 Frame Work

Since the game is digitally implemented it must have been developed over a digital framework and language. Our 2P snail game is developed using Python 3.90 language under the framework of Arcade. Arcade is a powerful Python library which have robust resources like illustrations and sounds to build 2D games.

3 Literature Review

The core of the game is Intelligent Player playing against Human. Our implemented AI agent is intelligent enough to give opposition a tough time. For making this agent intellectually strong we need to implement some searching techniques with the help of which it can predetermine its future steps. Two of such techniques are named as Min and Max algorithm and Heuristic search which are explained in the coming section.

3.1 Artificial Intelligent agent

One of the two players has been made artificially intelligent to beat the human player. Mini Max and heuristic search are two algorithms working behind it. AI agent needs to be clever enough to choose best path among the possibilities at the time of its turn same as a human do. In this regard Min Max helps it out. But Mini Max consumes a lot of memory figuring out maths for each possibility. Here we have use heuristic search where instead of drawing all possibilities AI agent take decision on few initial possibility graph.

3.2 MIN–MAX Algorithm

Mini Max is a recursive backtracking algorithm that is used in decision making. It spreads the whole game theory until the leaf nodes and follow Depth First Search Traversal. Mini Max assumes that the opponent player is also making optimal moves like it. However, this is not guaranteed every time. The human player may play without considering stats of the game. In such cases AI agent has high probability of winning the game. Mini Max has two parts that is the essence of this simple yet optimal algorithm. Let's call one of the player of snail game mini and the other max. what does mini do is always look for lowest number by selecting which it can reduce the probability of winning of the opponent. While Max always choose the path through which its score can be maximized.

3.3 Heuristic Algorithm

Heuristic algorithms belongs to class of decision problem. It is a search algorithm which provides fast search speed compromising over precision. In our snail game we have used this algorithm along with MINI MAX. Mini Max Algorithm alone is not enough to search the optimal Path because it consumes a lot of memory space to draw the whole graph until

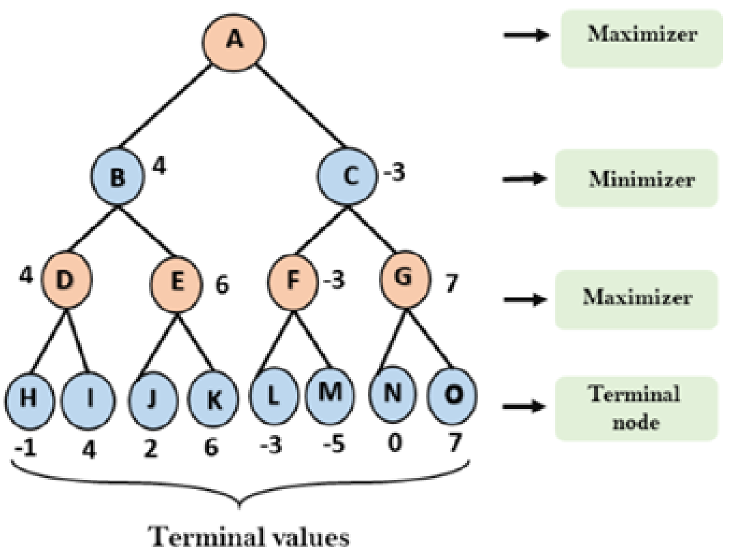


Figure 2: Min-Max Explanation

the end of game which makes it computationally expensive. For this reason, Heuristic Algorithm is used along with Mini Max which draws the graph to a limited capacity and then opt for foremost promising path. Further in Heuristic algorithm we have used Alpha Beta pruning heuristic search. Alpha Beta pruning is a Depth limited mini max which takes current state of AI as Input and predicts best move. The accuracy is totally dependent on heuristic function implementation.

3.3.1 Alpha Beta Pruning

Alpha beta speeds up the process of finding optimal path by reducing recursive computations which including ignoring complete unfavorable sub graphs. This technique uses two sub techniques Alpha and Beta. Both works in hand with Mini Max algorithm. Alpha is the best choice along the path of Maximizer player and Beta is the best choice along the path of Minimizer. Initial values of Alpha is set as negative infinity and Beta is set as positive infinity.

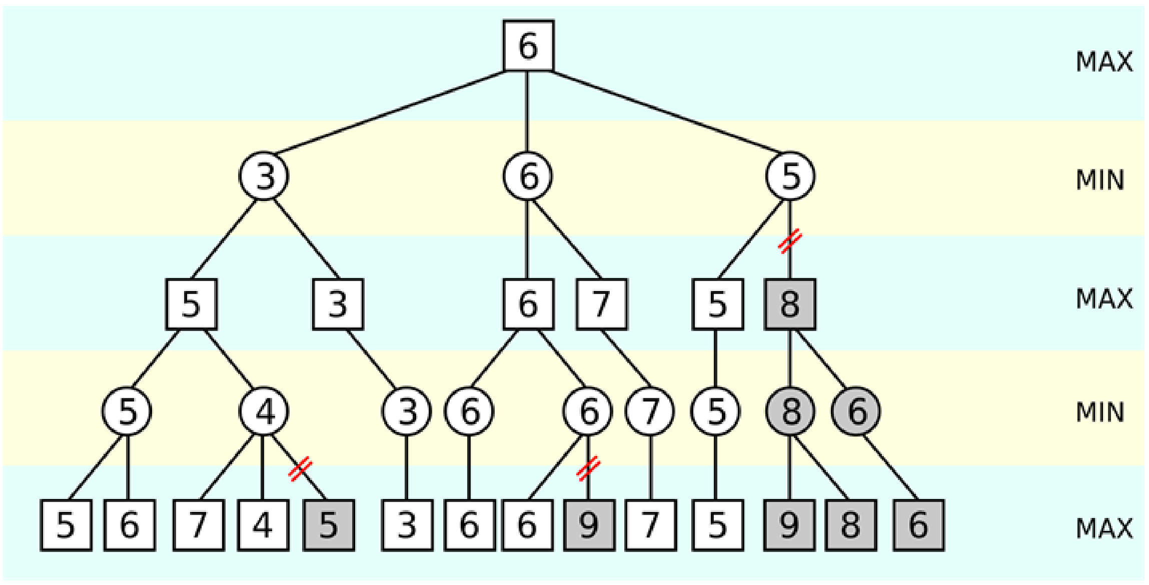


Figure 3: Alpha-Beta Pruning Example

4 Game Overview

4.1 Grid Initialization

A 10*10 grid is initialized, each box of 60*60 dimension. The grid is mapped on a back board with the help of a nested array. Each player has unique integer representation. Player 1 and player 2 are represented by 1 and 2 respectively in back grid. Both players are placed in the grid by default at alternative positions. Player 1 is placed at (0,9) and player 2 is placed at (9,0) position.

4.1.0.1 Implementation

```
self.board=[[1,0,0,0,0,0,0,0,0,0],
            [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],
            [0,0,0,0,0,0,0,0,0,0],
            [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],
            [0,0,0,0,0,0,0,0,0,2]]
```

4.2 Snail Movement

On the front end both players are animated as snails. When a player makes a call for move, his respective snail changes its position on the front end according to player's call. Blocks occupied by a snail are represented by splashes of respective player. Movement of the snails are governed by some rules which are as follows:

1. Each snail can make one move at a time and cannot be undone.
2. Snails can only move horizontally or vertically. Diagonal movement is not allowed.
3. If a snail steps into its splash. The snail is relocated to the end of splash in the respective direction.
4. If a snail steps into its opponent's splash or out of the grid boundary, then its turn will be lost.
5. Stepping into an empty space gives an increment of 1 in score.
6. The snail which scores 50 first is declared winner

5 Code Implementation

This section explains how the game is implemented. Tools have been already discussed in beginning as well as different game views. the actual game logic is applied in Game view. It follows a game loop explained below.

5.1 Game Loop

The game proceeds in a loop. Initially back-end board is initialized and synchronized with front end grind. Each time when player move its turn the boards are updating. This is exhibited in a loop and ends when time ends or a winner is decided. Flow chart is given below.

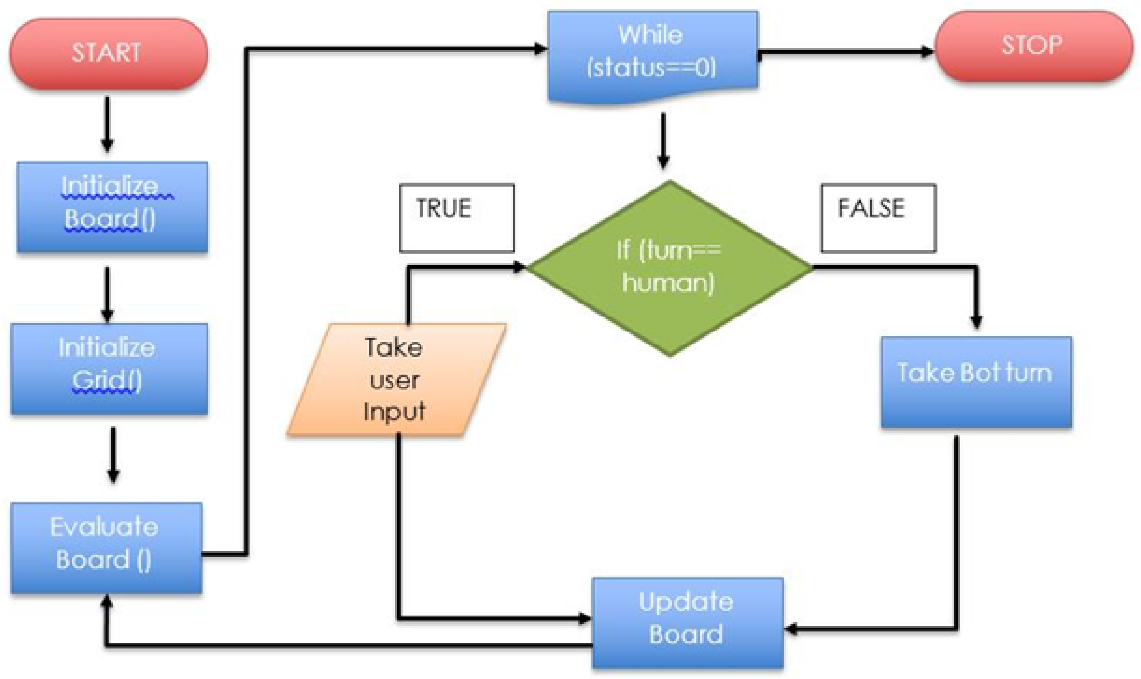


Figure 4: Flowchart of the Snail Game

5.2 Helping Function

Below is the explanation of some helping function used in the 2D Snail Game.

5.2.1 Setup()

This function will set-up both the snails sprite in Arcade.SpriteList(). We are then using this list onward in our program.

5.2.1.1 Implementation

```
def setup(self):
    self.snail_player1 = arcade.Sprite("player1.png", 0.15)
    self.snail_player1.center_x = ROW_SPACING/2
    self.snail_player1.center_y = COLUMN_SPACING/2
    self.snail_list.append(self.snail_player1)
    self.snail_player2 = arcade.Sprite("player2.png", 0.18)
    self.snail_player2.center_x =(SCREEN_HEIGHT)-(ROW_SPACING/2)
```

```
        self.snail_player2.center_y =(SCREEN_HEIGHT)-(COLUMN_SPACING/2)
        self.snail_list.append(self.snail_player2)
```

5.2.2 SlideOnSplash()

This function will activate if a snail moves to its own splash. Then the sprite's position will be adjusted towards the end of the splashes on that specific direction using this function. The function will get the current player splash val i.e for human it is 100 and for bot it is 200. Moreover, it take the direction of splash so that the snail can move only in its splash's direction.

5.2.2.1 Implementation

```
    def slide_on_splash(self,player_splash,x,y,isLeft,isRight,isUp,isDown):
        if isLeft:
            while (y>0):
                if self.board[x][y-1] != player_splash:
                    break
                y -=1
            return x,y
        if isRight:
            while (y<9):
                if self.board[x][y+1] != player_splash:
                    break
                y +=1
            return x,y
        if isUp:
            while (x<9):
                if self.board[x+1][y] != player_splash:
                    break
                x +=1
            return x,y
        if isDown:
            while (x>0):
                if self.board[x-1][y] != player_splash:
                    break
                x -=1
            return x,y
```

5.2.3 Evaluate()

This function will get the current board and return the current situation whether any of the player is in winning state or not. If human player is winning 10 will be return and if the bot is wining -10 will be return. If none of the player is in winning condition 0 will be returned.

5.2.3.1 Implementation

```
    def evaluate(self,board):
        count_p1 = 0
        count_p2 = 0
        for row in range(len(board)):
            for col in range(len(board)):
                if board[row][col]==100:
                    count_p1+=1
                elif board[row][col]==200:
                    count_p2+=2
        if count_p1>count_p2 and self.timer!=60:
            return 10
        elif count_p2>count_p1 and self.timer!=60:
            return -10
        # Else if none of them have won then return 0
        return 0
```

5.2.4 IsLegalMove()

We know that there is a constraint on snails movement. It can move on 4 possible direction i.e up,down,left,right. This function will check that whether each. This function is only being by AI agent to check its move whether legal or not.

5.2.4.1 Implementation

```
def isLegalMove(self,newi,newj,currenti,currentj):
    if (newi+1==currenti and newj==currentj) or (newi-1==currenti and newj==curre
        return True
    else:
        return False
```

5.3 AI Specific Function

5.3.1 FindBestMove()

This is initial step towards the implementation of AI in our game. This function will takes the current board and pass it to the MIN-MAX function after checking that the move is legal for AI. After checking all the possible moves it will return the best moves. First it will check for all possible zero's available at legal move if no one fits for best move. It will moves to its splashes and check there for best move.

5.3.1.1 Implementation

```
def findBestMove(self,board,ismax,last_max) :
    bestVal = -1000
    moveVal=  -1000
    bestMove = (-1, -1)

    # Traverse all cells, evaluate minimax function for
    # all empty cells. And return the cell with optimal
    # value.
```

```
        for i in range(len(board)) :
            for j in range(len(board)) :
                if self.isLegalMove(i,j,last_max[0],last_max[1]):

                # Check if cell is empty
                    if (board[i][j] == 0):
                    # Make the move
                        board[last_max[0]][last_max[1]]=200
                        board[i][j] = 2


                        # compute evaluation function for this
                        # move.
                        moveVal = self.minimax(board, 4, ismax,last_max[0],last_max[1

                        # Undo the move
                        board[i][j] = 0
                        board[last_max[0]][last_max[1]]=2
                        # If the value of the current ]move is
                        # more than the best value, then update
                        # best/
                    if (moveVal > bestVal) :
                        bestMove = (i, j)
                        bestVal = moveVal
        if bestMove !=(-1,-1):
            return bestMove
        else:
            for i in range(len(board)) :
                for j in range(len(board)) :
                    if self.isLegalMove(i,j,last_max[0],last_max[1]):
                        if (board[i][j] == 200):
```

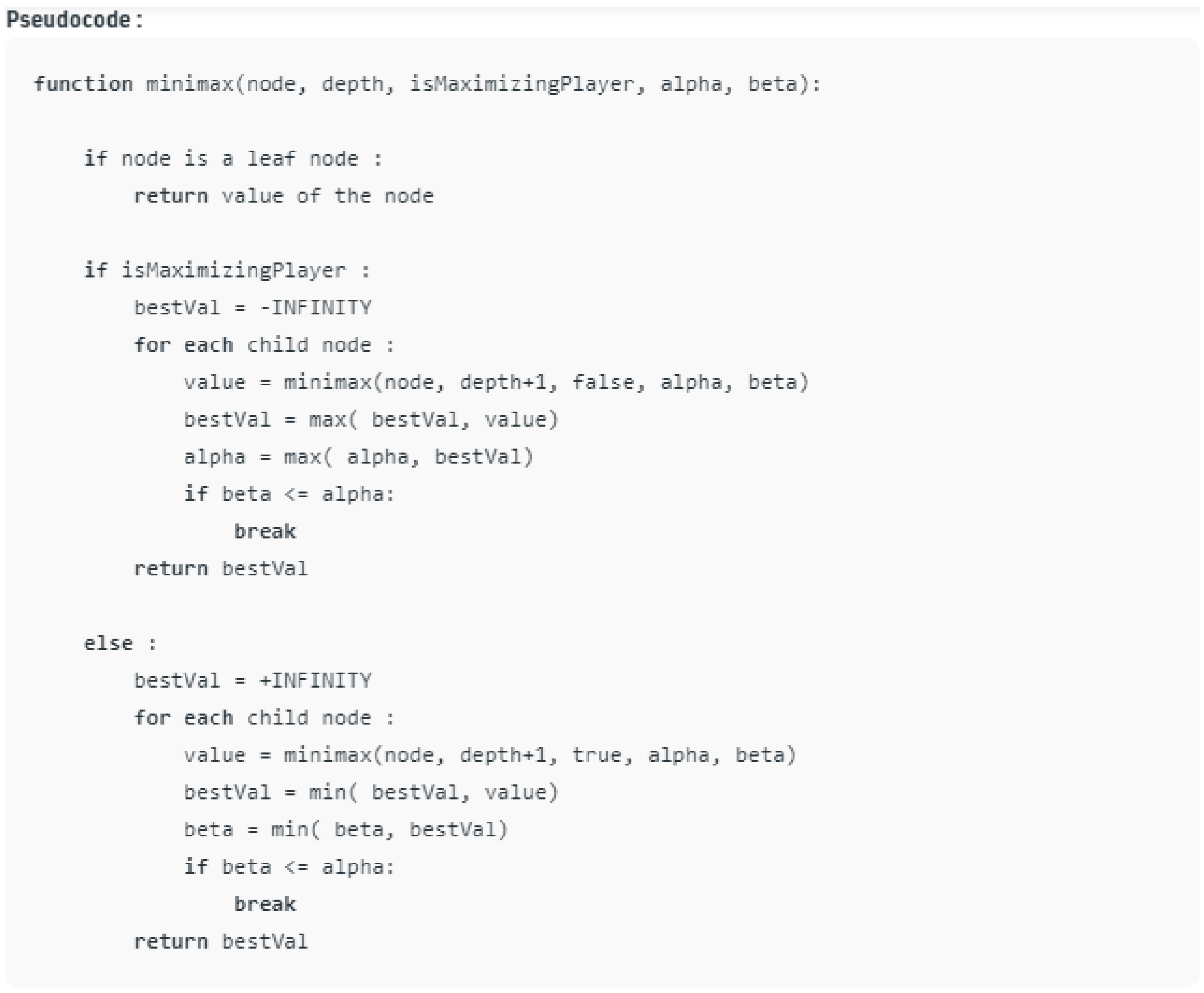
```
                    # Make the move
                        board[last_max[0]][last_max[1]]=200
                        board[i][j] = 2

                        # compute evaluation function for this
                        # move.
                        moveVal = self.minimax(board, 4, ismax,last_max[0],last_m

                        # Undo the move
                        board[i][j] = 200
                        board[last_max[0]][last_max[1]]=2
                    if (moveVal > bestVal):
                        bestMove = (i, j)
                        bestVal = moveVal
        if last_max[0]+1==bestMove[0]:
            newMove= self.slide_on_splash(200,last_max[0],last_max[1],False,False
        elif last_max[0]-1==bestMove[0]:
            newMove= self.slide_on_splash(200,last_max[0],last_max[1],False,False
        elif last_max[1]+1==bestMove[1]:
            newMove= self.slide_on_splash(200,last_max[0],last_max[1],False,True,
        elif last_max[1]-1==bestMove[1]:
            newMove= self.slide_on_splash(200,last_max[0],last_max[1],True,False,
        return newMove
```

5.3.2 Min-Max(Using Alpha-Beta Pruning)()

In Min-Max one player maximizes the score and other minimize it. Our AI agent is the maximizing player. We cannot traverse through all the board so we defined a limit on depth. In our case the depth is 8. Moreover, we have used the Alpha-Beta Pruning in our MIN-MAX algorithm so that we can smartly prune the unfavourable sub graphs. Here is the pseudo code for explanation of the algorithm.


Pseudocode :

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

Figure 5: Min-Max Algorithm Using Alpha-Beta Pruning

5.3.3 Heuristic()

Due to the big size of board, it is not feasible to generate all possible children of the board and figure out the best move for AI agent. Therefore, this function predicts the best possible move without processing all possibilities. The main purpose of heuristic function is to indicate, whether a particular move is in the right direction or not. To ensure that this is happening, a variable "winningChances" is initialized. The function will add some value to this variable by fulfilling the following criteria.

1. Calculate the number of visited boxes by AI Agent and add them to the variable 'winnigChances'.

2. Calculate the number of empty boxes around the current position of AI Agent and

this count to the variable ‘winnigChances’.

3. If the AI Agent is in around center of the board then add a constant value to the variable ‘winnigChances’. The constant number in our case is 10.

5.3.3.1 Implementation

```
def heuristic(self, board,i,j,player_splash):
    winningChances = 0

    # First Condition
    for x in range(10):
        for y in range(10):
            if self.board[x][y]==player_splash:      #updating score of players e
                winningChances +=1

    # 2nd Condition
    currentRow, currentCol = i,j

    # If below box is empty
    if currentRow+1 < len(board):
        if board[currentRow+1][currentCol] == 0:
            winningChances += 1

    # If above box is empty
    if currentRow-1 > 0:
        if board[currentRow-1][currentCol] == 0:
            winningChances += 1
    # If left box is empty
    if currentCol-1 > 0:
        if board[currentRow][currentCol-1] == 0:
            winningChances += 1
    # If right box is empty
    if currentCol+1 < len(board):
        if board[currentRow][currentCol+1] == 0:
            winningChances += 1



    # 3rd Condition
    rangeMin = (len(board)//2) - 3
    rangeMax = (len(board)//2) + 3

    if [rangeMin,rangeMin] <= [currentRow, currentCol] <= [rangeMax,rangeMax]:
        winningChances += 10



    return winningChances
```

6 Game Screens

6.1 Start screen

The game starts with an interactive screen holding name of game in a Halloween Specific theme. With a mouse click start screen transitions into instruction screen.

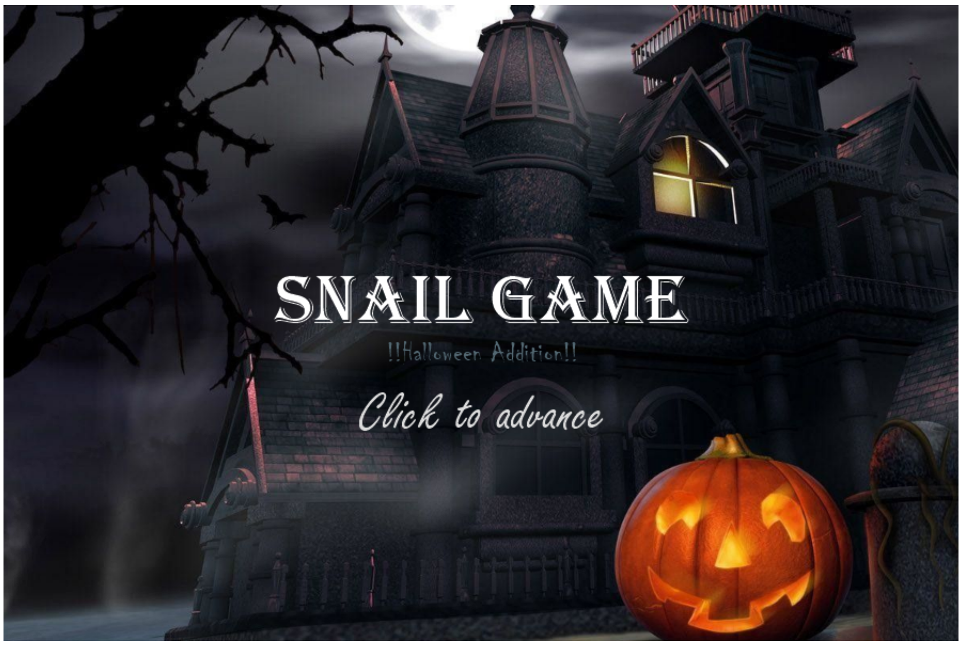
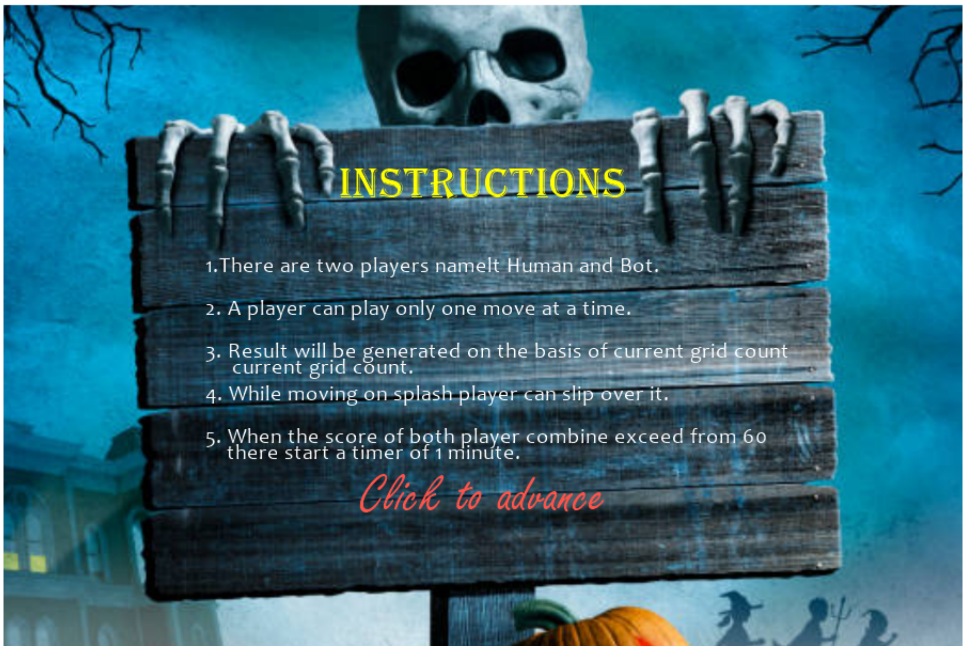


Figure 6: Start Screen of Snail Game

6.2 Instruction Screen

With a mouse click or keyboard press The screen shifts from Start to Instruction View. This screen displays all the rules of game and tells how it will proceed.



INSTRUCTIONS

1.There are two players namelt Human and Bot.

2. A player can play only one move at a time.

3. Result will be generated on the basis of current grid count
current grid count.

4. While moving on splash player can slip over it.

5. When the score of both player combine exceed from 60
there start a timer of 1 minute.

Figure 7: Instruction Screen of Snail Game

6.3 Game Screen

After the instruction main game view appears. It contains 10*10 Grid with in which the two player moves. A panel at the right side shows all the statistics like time remaining and scores of each player.

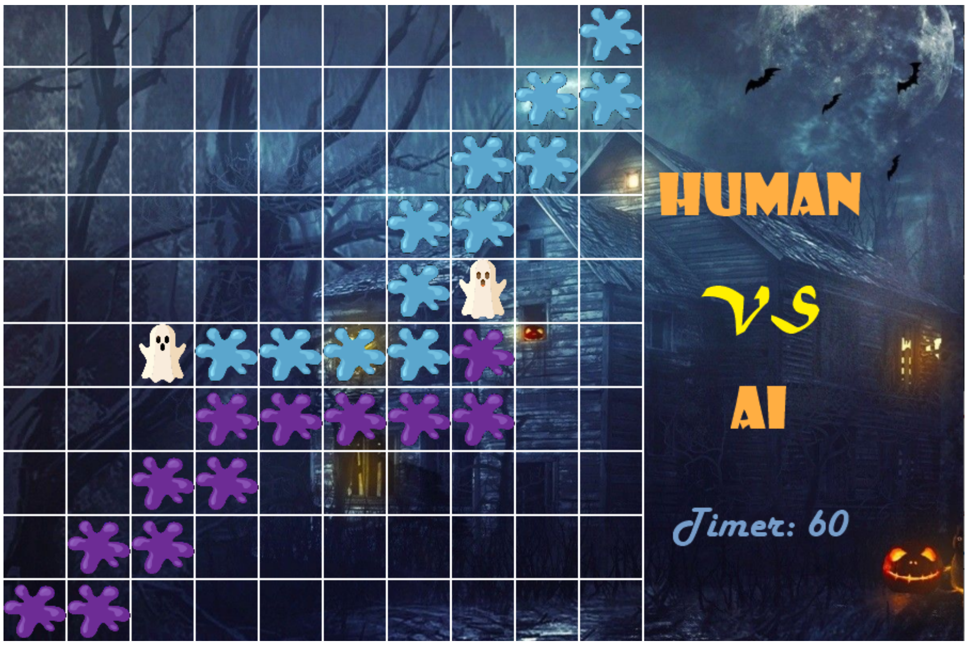


Figure 8: Main Screen of Snail Game

6.4 End screen

The game end screen shows which player has won. Not every time there is a win – lose situation but draw can also take place between two players. It also allows to restart the game.

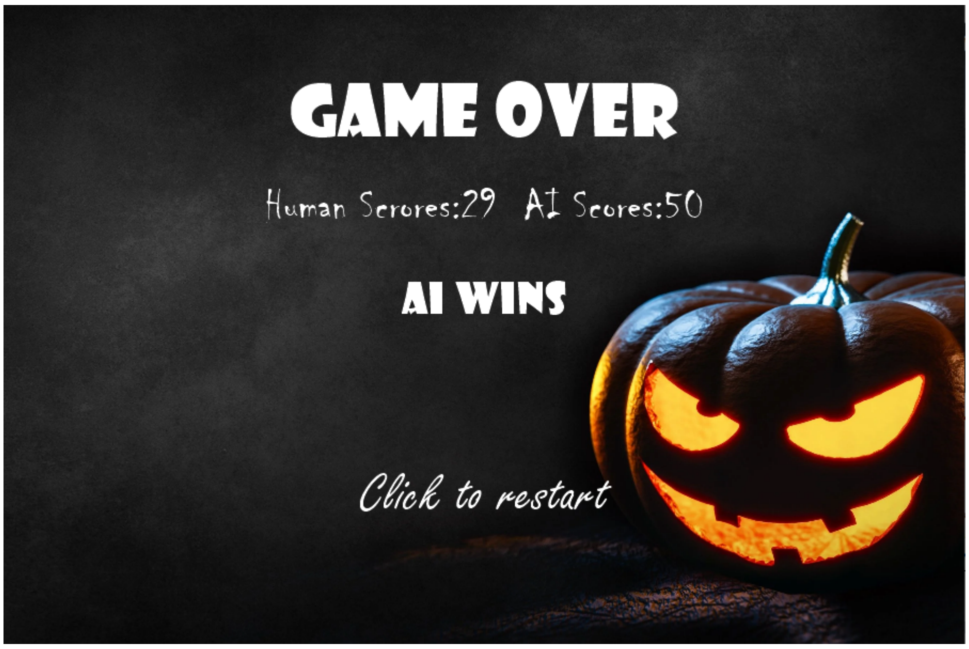


Figure 9: Game Over Screen of Snail Game

7 Conclusion & Recommendation

Games like chess, checkers, backgammon, tic-tac-toe, and our snail uses techniques like MIN-MAX to find the optimal best move in an efficient and fastest way. Some techniques can also be used along with Min-Max to enhance its functionality such as Chasing-Snail, A-Star, Heuristic and Alpha Beta pruning are the some of the technique which can b merged to enhance the efficiency of the AI Agent. Moreover, this snail game can also be improved by using strong heuristic algorithms with high accuracy and probability of winning game. The interface can also be enhanced or modified to other versions for instance we produced Halloween addition.

8 Citation & Reference

1. GeeksforGeeks. “Minimax Algorithm in Game Theory — Set 1 (Introduction).” GeeksforGeeks, 5 Dec. 2021

2. Alpha–beta pruning - Wikipedia, 2021

3. Minimax Algorithm in Game Theory — Set 4 (Alpha-Beta Pruning) - GeeksforGeeks, 2021