COURSE CODE: UE16CS325

ARTIFICIAL INTELLIGENCE

PROJECT REPORT

AI based GAMES

TEAM MEMBERS

Kiran S Hombal– 01FB16ECS167
Lokesh Kumar T N – 01FB16ECS180
Jaydeep Kudumula- 01FB16ECS170
Hemanth SVS - 01FB16ECS134

BATTLESHIP

N-QUEEN

Sudoku

Ultimate
Tic-Tac-Toe

A.I MINI-PROJECT:BATTLESHIP

INTRODUCTION

This part of the project deals with solving a well-known board game called battlefield using basic game theory models.

PROBLEM STATEMENT

Battleships is a 2-player guessing game originally published by the Milton Bradley Company in 1943. Each player uses two square grids of size 10, one to place the own ships, one to record the shots fired at the opponent. A total of 10 ships of various sizes has to be placed by every player in such a way that no two ships occupy adjacent squares. After the initial placement phase, players alternate in shooting at enemy ships. The methods of solving guessing games that include a search component can have widespread real-world applications. Many robot-robot or robot-human interactions can be represented as adversarial games with uncertainty. Gaining insight into the structure and solution processes with toy problems such as Battleships will help us apply these methods to other problems of a similar structure.

GOALS

Arrive at a stage in a game where the following criteria is meet:

- The goal of the game is to sink all enemy ships before one's own ships are sunk.
- No invalid moves are made.
- No overlapping of the ships.

APPROACH

I intend to split the problem into two sections:

1. The Ship placement
    a. The best possible placement maximizes entropy and therefore favors random placement of the ships(according to [1])
2. Actual game playing
    a. The game will be represented as a search problem with chance nodes and a heuristic function that guides the search towards squares that contain a ship with high probability. We have encoded both exploration of the board and exploitation in our heuristic, by taking into account the distance to the goal state (i.e. missing hits till every enemy ship is sunk) and the positions and orientation of found and already sunk ships
    b. As opposed to rule-based expert systems and machine learning approaches, our approach is expected to perform well without any previous training and knowledge about the opponent.
    c. Rules are implicitly encoded in the heuristic function guiding the search, thus enabling it to react to unforeseen situations.

SPECIFICATIONS

To be able to run this code, any web browser is all that it requires.
Input: hit block on the grid each time it is the users turn. (GUI input)
Output: The state of the game for both sides(GUI)
Language used: JS
AI models used : search problem with chance nodes and a heuristic function.

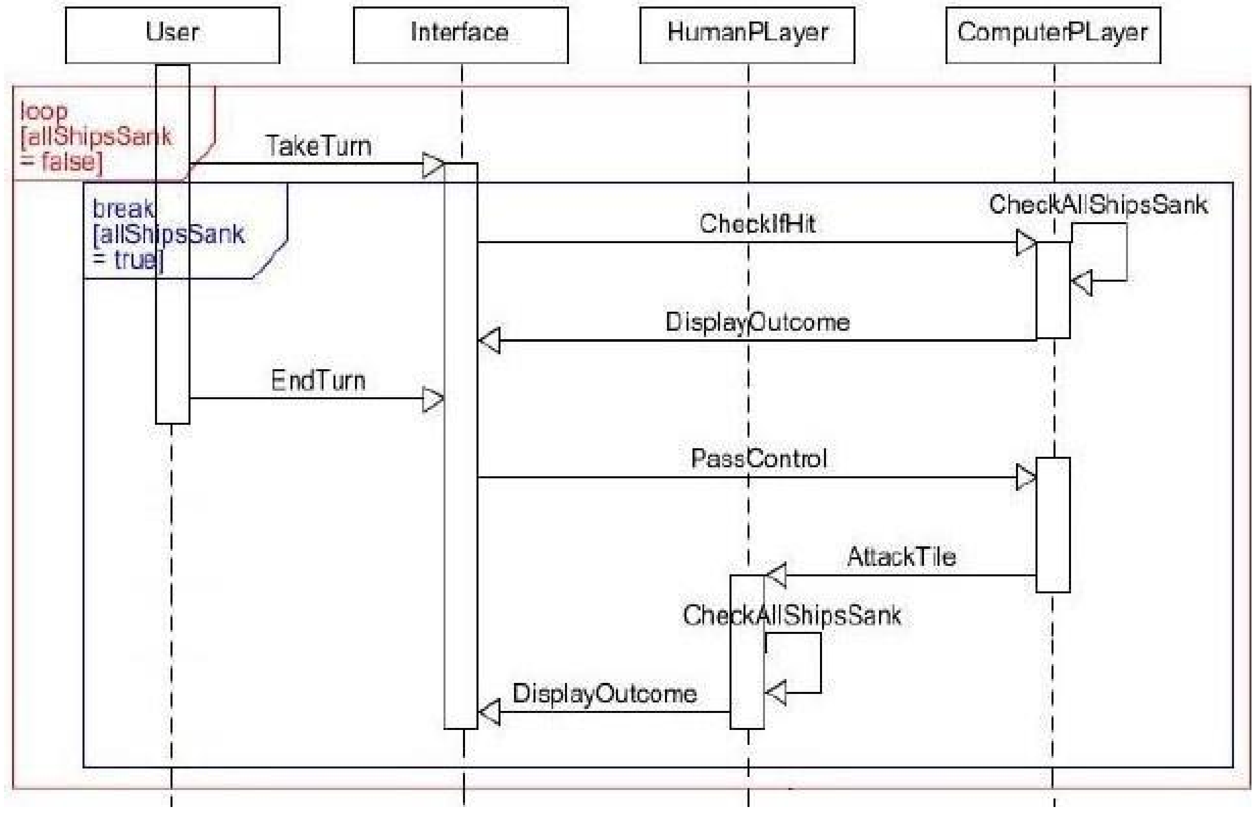


Figure 1.1

ALGORITHM

Random AI

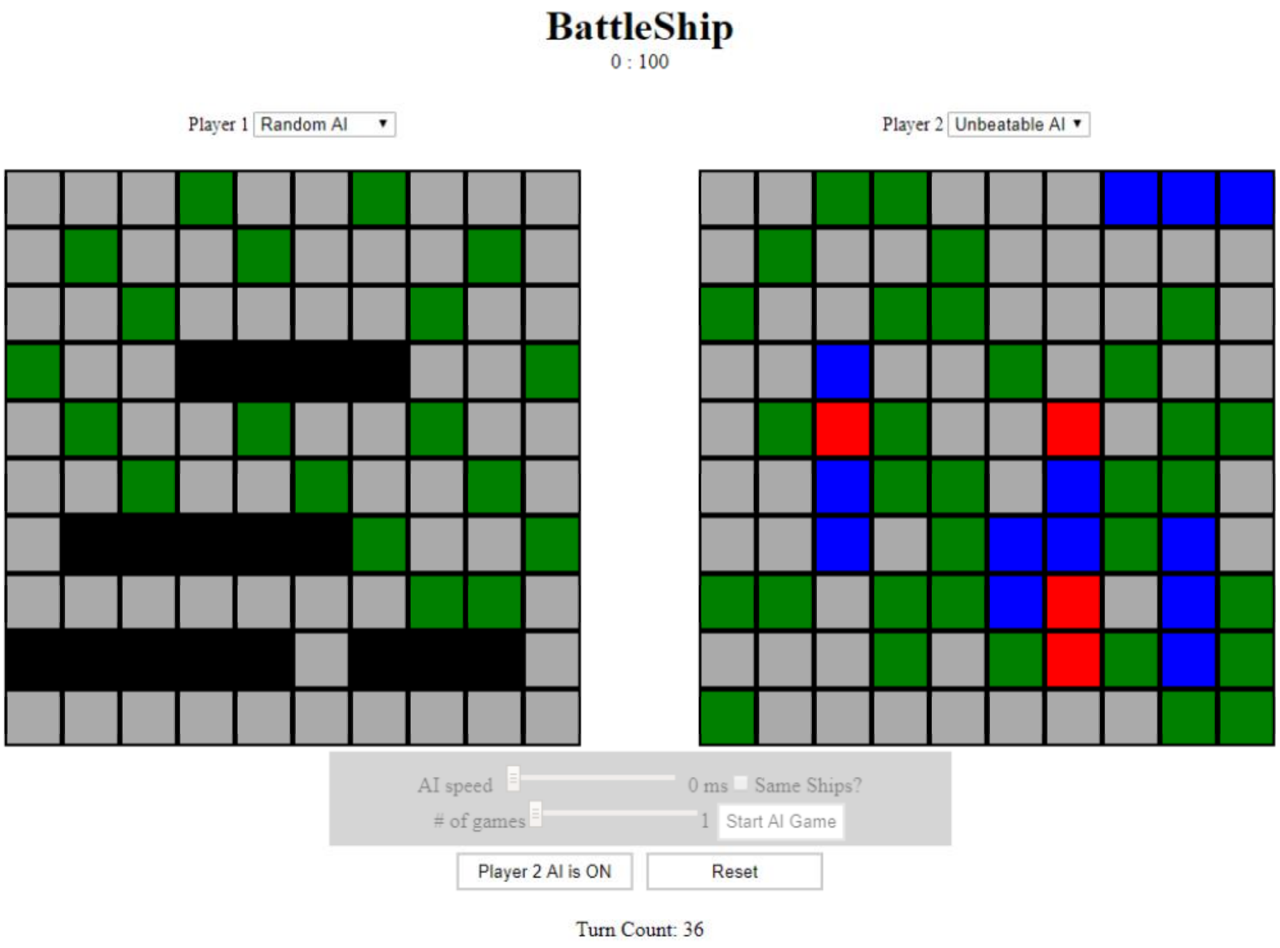
```
function randAI(state){
                var open = state.getOpen();
                return open[Math.floor((Math.random() * open.length))];
}
```

Unbeatable AI

```
function unbeatableAI(state){
 if(hitNoSunk.length > 0){
  for(var h = 0; h < hitNoSunk.length; h++){
   for(var s = 0; s < state.ships.length; s++){
    var shipLen = state.ships[s].len;
    for(var i = 0; i < 10; i++){
     for(var j = 0; j < 10; j++){
      var ship = new Ship(j,i, shipLen, true);
      if(ship.validLoc() && !shipsContainPoints(state.miss, [ship]) && shipsContainPoints([hitNoSunk[h]], [ship])
               && !intersectOtherShips(ship, state.sunk)){
       for(var p = 0; p < ship.getPoints().length; p++){
        var x = ship.getPoints()[p][1];
        var y = ship.getPoints()[p][0];
        probs[y][x] += 1;   }                  }
      var ship = new Ship(j,i, shipLen, false);
      if(ship.validLoc() && !shipsContainPoints(state.miss, [ship]) && shipsContainPoints([hitNoSunk[h]], [ship])
               && !intersectOtherShips(ship, state.sunk)){
       for(var p = 0; p < ship.getPoints().length; p++){
        var x = ship.getPoints()[p][1];
        var y = ship.getPoints()[p][0];
        probs[y][x] += 1;
       }
      }
     }
    }
   }
  }
 }
 else{
  for(var s = 0; s < state.ships.length; s++){
   var shipLen = state.ships[s].len;
   for(var i = 0; i < 10; i++){
    for(var j = 0; j < 10; j++){
     var ship = new Ship(j,i, shipLen, true);
     if(ship.validLoc() && !shipsContainPoints(state.miss, [ship]) && !intersectOtherShips(ship, state.sunk)){
        for(var p = 0; p < ship.getPoints().length; p++){
         var x = ship.getPoints()[p][1];
         var y = ship.getPoints()[p][0];
         probs[y][x] += 1;}  }
     var ship = new Ship(j,i, shipLen, false);
     if(ship.validLoc() && !shipsContainPoints(state.miss, [ship]) && !intersectOtherShips(ship, state.sunk)){
        for(var p = 0; p < ship.getPoints().length; p++){
         var x = ship.getPoints()[p][1];
         var y = ship.getPoints()[p][0];
         probs[y][x] += 1;
        }
     }
    }
   }
  }
 }
```

CONCLUSION

After trying to solve a well know, simple and inherently fun games using AI gives us a unique perspective into the vast application of AI on a great scale. In the figure 1.2 we should note that our Unbeatable AI wins a 100% of times against random AI. This observation allows us to come to an inference saying sometimes a well-defined AI can outperform any kind of user.


BattleShip

0 : 100

Player 1 Random AI

Player 2 Unbeatable AI

AI speed 0 ms

Same Ships?

# of games 1

Player 2 AI is ON

Turn Count: 36

Figure 1.2

REFERENCES

[1]J. Bridon, Z. Correll, C. Dubler, and Z. Gotsch, "An artificially intelligent battleship player utilizing adaptive firing and placement strategies," 2009. [Online]. Available: www.cores2.com/files/FinalResearchPaper.pdf

[2] www.academia.edu "Battleship: Software feasibility, analysis, design, testing & debugging Report"

A.I MINI-PROJECT: N-QUEEN

INTRODUCTION

In this project we try to apply different modules of Artificial Intelligence that we have learned in the course, to get a interesting time which is either existing or a modification of the existing game. This gives us an opportunity to get our hands dirty of Artificial Intelligence modules and the pros and cons of the decisions we make, which are sometimes critical.

PROBLEM STATEMENT

The game which I am trying to build is derived from the famous N queens problem. The game is a two player game. Each player plays alternatively and has to place a queen in a cell such that it is not attacked by any of the queens placed so far, either by the player or the bot. The player(or bot) who is not able to place a queen safely is the winner.

GOALS

There are a number of decisions while considering the game development. Some of them are as follows. We need to keep track of both the cells where queens are actually placed and the cells which are being attacked by an existing queen. Though if you place a queen in either of the cell, the player will lose the game, but you can still place a queen in such a way that the cells which are being attacked are attacked again. So there should be something to distinguish between the two. Also we need to keep track of the count of the queens placed on the board. If it is equal to 8, then it is declared a draw.

APPROACH

So there are multiple things to consider while developing this game. The approach is to first let the player make his move. Then bot will check all the cells which are being attacked by this move. Then bot will check all the cells in which are left in the domain and find out the number of non-attacked cells that are going to be attacked by placing the queen here. At the end, bot will select the cell in such a way, that placing a queen there will have the most number of non-attacking cells attacked. And again check the cells which are being attacked by this new queen and remove them from the bot domain. Repeat the whole process, until either one of them places a queen such that it is being attacked(obviously bot is not going to do this) or there are ten queens placed in the board. In case if there are ten queens being placed(a rare occurrence) the game is declared a draw. Otherwise the player who made a wrong move or is unable to place the queen, loses the game.

SPECIFICATIONS

To be able to run this code, any web browser is all that it requires. You can open it with the browser of your choice and the game starts when you click on the game icon. You need to place the queen first.

ALGORITHM

Main() <-do:

```
While(!game.over){
    x,y <- input() //the coordinates of the cell
    r=attack(x,y)
    if(r==-1){
        print "Sorry, you lost the game"
        game.over=true
    }
    Else{
        move<-move + 1
        r,x,y <-huristic()
        if(r==1){
            print "Congratulations you won the game"
            game.over=true
        }
        Else{
            move<-move+1
        }
    }
    If(move=10){
        print "Match drawn!"
        game.over=true
    }
}
```

attack(x,y) <- do:

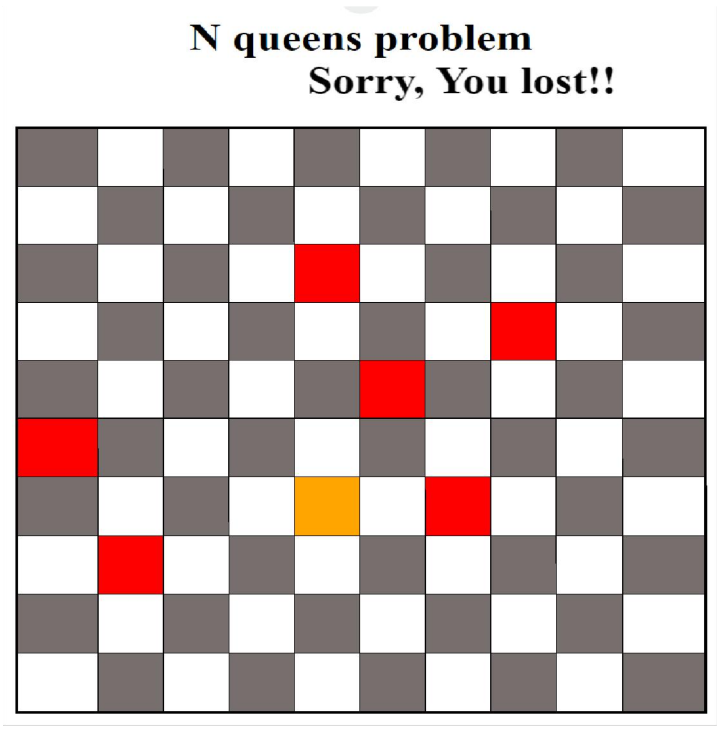
```
if(Graph[x][y] is attacked by atleast one of the queens){
    return -1
}
If(Graph[x][y] is already occupied by another queen){
    return -1
}
Graph[x][y] <- queen placed
for all p,q in cells attacked by Graph[x][y]:
    Graph[p][q] <- attacked by queen
return 0
```

huristic() <- do:

```
maximum<-  -1
for all p,q in cells not attacked by any of the queens and queen not placed in that cell{
      current <- number of new cells which are going to be attacked by placing the queen in Graph[p][q]
      if(maximum < current){
          maximum <-  current
          idx,idy <- p,q
  }
}
If(maximum=-1){
   return -1,-1,-1
}
for all p,q cells which are attacked by Graph[idx][idy]{
     Graph[p][q] <- attacked by queen
}
Graph[idx][idy] <- queen placed
return 0,idx,idy
}
```

CONCLUSION

The approach used to develop this game was to minimize the domain of the other player and who decreased it the most won the game. There is a higher chance of bot winning the game since it always places the queen in the optimized cell. This also shows that constraint specific problems can also be used to develop interesting problems.



A.I MINI-PROJECT: SUDOKU SOLVER

INTRODUCTION

An inherent approach to learning fundamental concepts of AI can be done, though using AI techniques for playing ad solving games and puzzles such as the age-old Sudoku. With fundamental approaches to problem solving like **Constraint Satisfaction** and **Search**, AI algorithms give us the ability to tackle the expanse of these problems.

PROBLEM STATEMENT

Design and implement an algorithm for solving a given (9*9) Sudoku puzzle. A puzzle is said to be solved if each square in every row, column and box in the puzzle has a unique value from 1 to 9.

GOALS

Arrive at a solution to the input puzzle defined by:

- A puzzle is solved if the squares in each unit are filled with a permutation of the digits 1 to 9.
- If unsolvable, retain none/a set of possible values at the squares for which a solution cannot be obtained.

APPROACH

1. Convert the input to the solver into a dictionary (parse grid), with letters (A-I) denoting the rows and numbers 1-9 denoting the columns as the key and the input as the value (0 for an empty box).
2. Constraint satisfaction - Based upon 2 fundamental rules:
    a. If a square can have only 1 possible value, then eliminate that value from the square's peers.
    b. If a unit has only one possible place for a value, then put the value there, having eliminated it from its peers.

3. Search - For completion:
    a. In case the constraint satisfiability cannot be met, use search to eliminate the impossible values from the set of unresolvable values at a square, for each square.

b. For increasing the chances of reaching the goal faster, start selection from a set of possible values at a square with least number of possible values.

SPECIFICATIONS

Nature of solution: complete due to having the addition of a search which traverses all possible solutions when the CS approach halts.

Input: String of number in row-major order with '0' or '.' Representing a blank square.

Output: Sudoku grid with solution in text format.

Language used: python 2.x

ALGORITHM

Assign: assign(values, s, d) updates the incoming values by eliminating the other values than d from the square s calling the function eliminate(values, s, d).

```
def assign(values, s, d):

    """Eliminate all the other values (except d) from values[s] and
      propagate.
    Return values, except return False if a contradiction is detected."""
    other_values = values[s].replace(d, '')
    if all(eliminate(values, s, d2) for d2 in other_values):
        return values
    else:
        return False
    else:
        return False
```

Eliminate: Implement the rules defined for the CSP and eliminate values for the squares.

```
def eliminate(values, s, d):
    """Eliminate d from values[s];”””.
    if d not in values[s]:
        return values ## Already eliminated
    values[s] = values[s].replace(d,'')
    # Rule 1 : If a square s is reduced to one value d2, then eliminate d2
       from the peers.
    if len(values[s]) == 0:
       return False ## Contradiction: removed last value
    elif len(values[s]) == 1:
        d2 = values[s]
        if not all(eliminate(values, s2, d2) for s2 in peers[s]):
            return False
    #Rule 2: If a unit u is reduced to only one place for a value d, then put
         it there.
    for u in units[s]:
        dplaces = [s for s in u if d in values[s]]
        if len(dplaces) == 0:
            return False ## Contradiction: no place for this value
        elif len(dplaces) == 1:
            if not assign(values, dplaces[0], d):
                return False
    return values
```

Search: Search all possible remaining solutions to see if goal is reached.

```
def search(values):
    "Using search and propagation, try all possible values."
    if values is False:
        return False #Failed
    if all(len(values[s]) == 1 for s in squares):
        return values #Solved!
    ## Chose the unfilled square s with the fewest possibilities
    n,s = min((len(values[s]), s) for s in squares if len(values[s]) > 1)
    return some(search(assign(values.copy(), s, d))
                for d in values[s])
```

CONCLUSION

Developing an algorithm to arrive at a solution to problems such as the 'Sudoku solver' and many other simple and inherently fun puzzles games gives us a unique retrospective into application of AI on a greater scale. The above, serves a similar purpose and paves way for more advanced algorithms which find their way into important real life applications.

TIC TAC TOE

Introduction

Tic Tac Toe is a game for two players,X and O who take turns in marking the spaces in a 3 X 3 grid. The player who succeeds in marking three positions in the grid in a horizontal, vertical or diagonal row wins the game.

This game can be described as zero-sum game which means that each opponent's gain or loss of utility is exactly balanced by the losses or gains of the utility of the other participants.

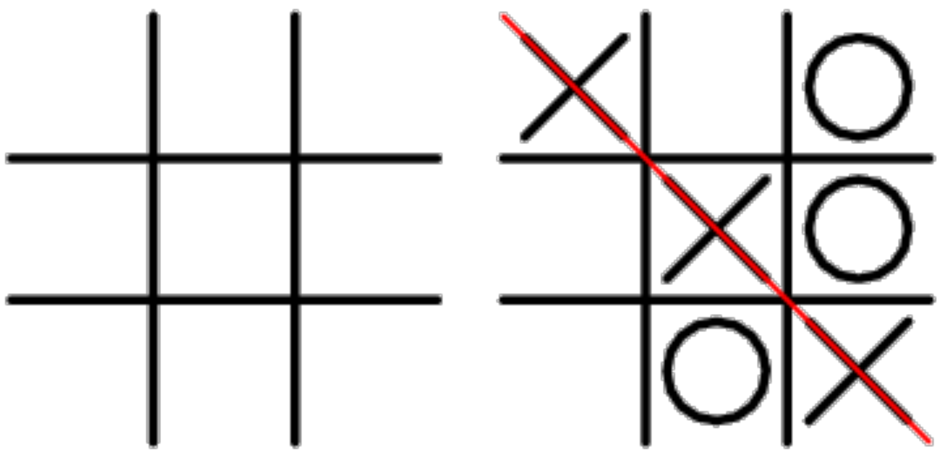
In the TIC TAC TOE game implemented here we have used the famous algorithm "THE MINIMAX".

Problem Statement

Creating a game of TIC TAC TOE using the MINIMAX algorithm

GOALS

The Goal of the game is for the player to win by marking three successive markers of their own making a horizontal, vertical or diagonal row before the other opponent makes a row using his own marker.



Approach

THE MINIMAX ALGORITHM :

Minimax is an artificial intelligence algorithm applied in two player games, such as tic tac toe, checkers, chess and go. These games are known as zero-sum games because in a mathematical representation when one player wins (+1) the other player loses (-1) or if both don't win (0).

 Since the chance to play the game is alternative, after every-time the humans plays his turn it's the AI turn.When it is AI's turn the ai_turn() is called in the program which in turn uses the Minimax algorithm and computes the best move by recursively calling the minimax function and picks the best move of the lot with the highest score tagged to it.So the AI always makes the best possible move and the AI never loses. The game always end's in a draw unless the human doesn't make the best possible move always.

Algorithm : Understanding the Algorithm “MINIMAX”

```
minimax(state, depth, player)

    if (player = max) then
        best = [null, -infinity]
    else
        best = [null, +infinity]

    if (depth = 0 or gameover()) then
        score = evaluate this state for player
        return [null, score]

    for each valid move m for player in state s do
        execute move m on s
        [move, score] = minimax(s, depth - 1, -player)
        undo move m on s

        if (player = max) then
            if score > best.score then best = [move,
score]
        else
            if score < best.score then best = [move,
score]

    return best
end
```

The above algorithm which is minimax has been used in the tic tac toe game.

state : the current grid
depth : index of the node in the game tree
player : may be a MAX player or MIN player

Both players start with their worst score

```
If player == MAX :
    return [-1,1,-inf]
else:
     return [-1,-1,+inf]
```

```
if depth ==  0 or game_over(state):
        score = evaluate(state)
        return score
```

If the depth is equal to zero, then the board doesn’t have any new cells to play. If a player wins, then the game ended for MAX or MIN. So the score for that state will be returned.

If MAX won : we return +1
If MIN won : return -1
Else : return 0 (draw)

Now the main part of the code that contains recursion

```
for cell in empty_cells(state):
    x, y = cell[0],cell[1]
    state[x][y] = player
    score = minimax(state, depth - 1, -player)
    state[x][y] = 0
    score[0], score[1] = x, y
```

So, for each valid move which is an empty cell :

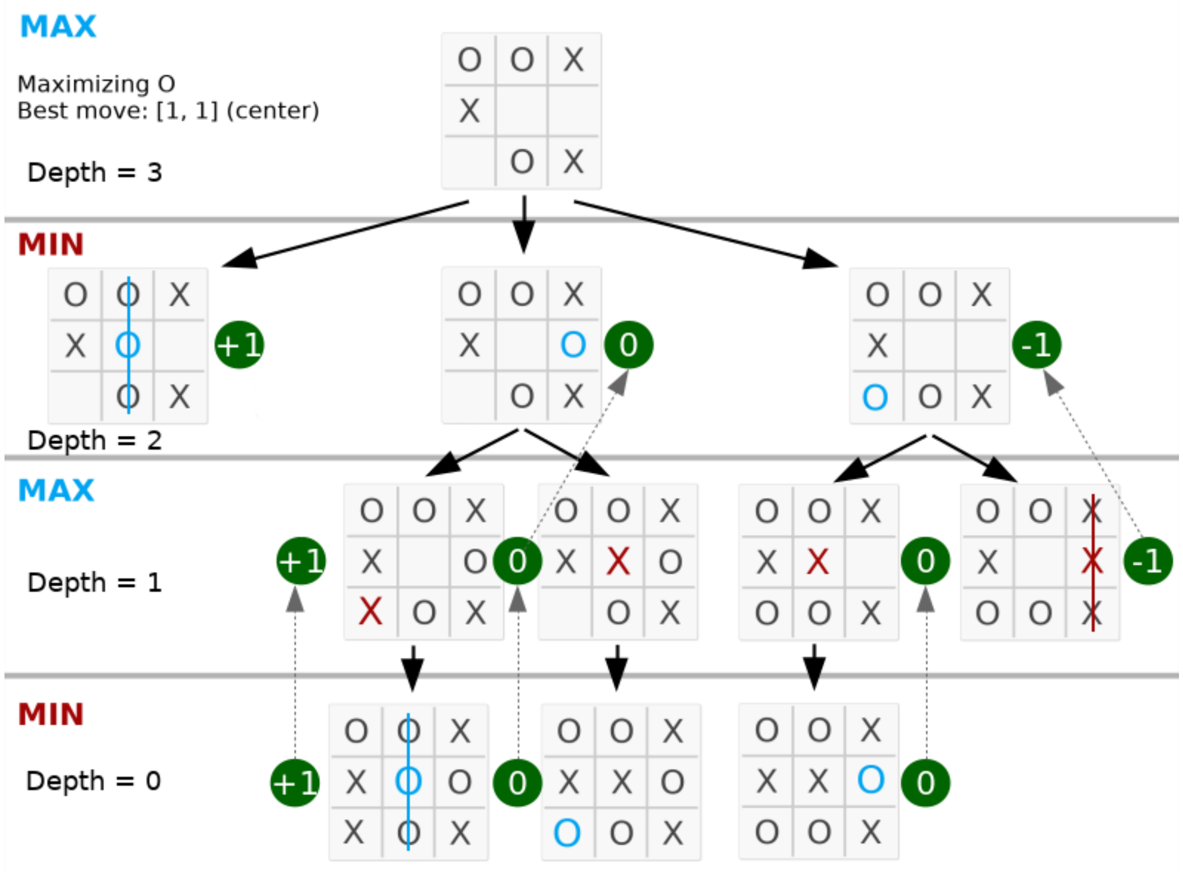
x will be row index

y will be column index
state[x][y] is board[available_row][available_column] will have MAX or MIN player
score = minimax(state, depth - 1,-player) the depth is decreased by one everytime.
    state is the current board in recursion
    depth is the index of the next state
    -player is if a player is MAX it'll be (+1) or else MIN it'll be (-1).

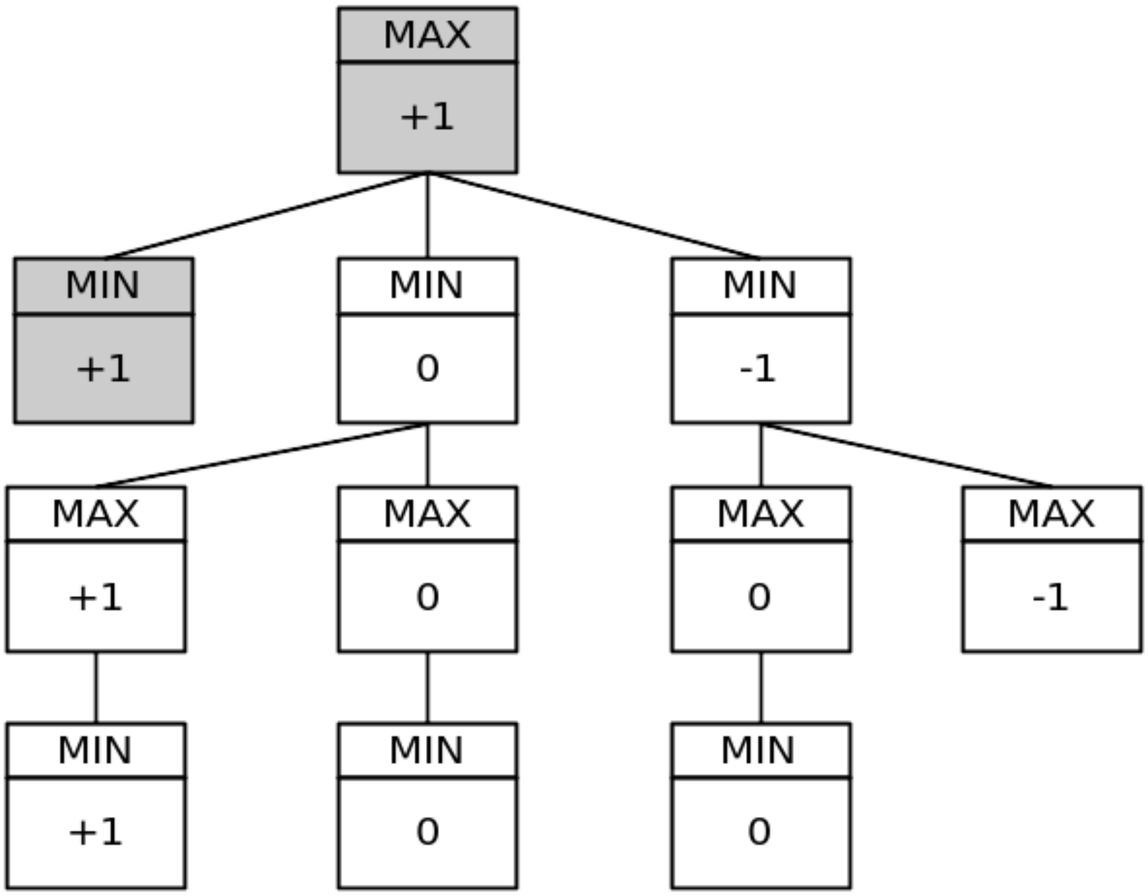
The final step is to compare and set the best move for the best variable which can be seen in the minimax algorithm.

For a MAX player, a bigger score will be received. For a MIN player, a lower score will be received and in the end, the best move is returned.

Now we can see the game tree and how the best move is calculated using the tree which is created using the recursive calling of the function in the minimax.



The below one is a simplified game tree :



In a more complex game, such as chess, it's hard to search whole game tree. However, Alpha–beta Pruning is an optimisation method to the minimax algorithm that allows us to prune or ignore some branches in the search tree, because it removes some of the irrelevant nodes from which no move or decision can be made.

Conclusion

So Here we can see how AI has been used to create a game of TIC TAC TOE which is unbeatable. Similarly many games can be created using minimax algorithm like Chess, Checkers,GO etc.The AI here always chooses the best possible move unlike human where he might stumble when he has to.So AI is more reliable here in this case than humans.