The Best AI Algorithm Defeats The Human In Board Game

Vigneshraj Perumal Raja
School of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, 5000, Australia.
a1787474@student.adelaide.edu.au

Abstract:

Artificial Intelligence (AI) serves for the various scope of computer technology. Games are the essential key element that helps in improving the strength and structure of the algorithms. Recently, games got many branches like the Board game, Video game and Virtual reality game. Board games have always been a testbed for artificial intelligence techniques. Most of the developers are undergoing some constant search techniques to automatically create computer games in order to minimise the developer's task. In this paper, we consider designing an optimal method to obtain the next move in a two player board player where a human player is played by opposing the AI. The traditional search algorithm Minimax is considered which results in consuming a lot of time to make the next move in the Five-in-a-row board game. So, I have designed an algorithm of combining Minimax with an evaluation function to reduce the time and space complexity. Experiment shows that search speed is highly improved by this algorithm which is tested under different depths.

Keyword: Artificial Intelligence (AI), Minimax, Evaluation Function, Five-in-a-row

1 Introduction

1.1 Background

Artificial intelligence (AI) is a scientific discipline, helps in the study of developing intelligent machine and software to perform tasks requiring human intelligence [1]. AI's computer game playing concepts were initiated with a chess-playing machine proposed by Shannon in 1950 to a Deep blue chess system by IBM that beats human world chess champion [2]. To make the computer respond to the human player's input, strong algorithms are required. Among all algorithms, the Minimax search strategy was proposed to determine good moves to oppose human players [2]. Applying fixed-depth minimax tree search requires a function to evaluate the state of the game [6]. Heuristic search is an evaluation function helps to acquire the state and required decision to make a move [6]. The heuristic search does not require any domain-based knowledge [6]. Experiments are undertaken to show the efficiency of the proposed method and the corresponding results state the improvement of the search algorithm with the combination of the evaluation function. This proposed algorithm is compatible with any computer having at least 4GB of RAM.

1.2 Goal of thesis paper

This paper explains the method and experimentation result of applying the Minimax algorithm with Heuristic search to Five-in-a-row board game and comparing the result with Alpha-Beta pruning's strategy, which was discussed in my teammate Zhuroan Yang's paper [9].

1.3 Structure of thesis paper

Section 2 discusses the overview and existing approach of AI board game, Minimax, and heuristic search method. Following that, I have discussed the design and results of implementing those methods in a Five-in-a-row computer board game.

This paper contains some Pseudocode in the following sections to provide a high-level description of an algorithm to the readers.

2 Literature review

2.1 AI Board game

The science of AI was invented soon after world war II in 1956 [3]. The competitive conditions are the reason to create interests of the AI agent to examine games [3]. Game playing concept was the first activity performed by Konrad Zuse (discoverer of the first computer program), Claude Shannon (innovator of the theory of knowledge), Norbert Wiener (father of modern systems engineering), and Norbert Wiener (father of modern control systems) [3].

For better understanding and to experiment with the algorithm, in this paper we consider the Five-in-a-row board game. This game consists of a board with 15×15 squares providing the human player with "X" labelled coins and the AI player with "O" labelled coins. In this game, both the players will be targeted to arrange unbroken Five in a row/ column/ diagonal in their turn to win the game.

In the below sections, I have provided the explanation and methodology of implementing the Minimax and Heuristic search concept to the Five-in-a-row game as per the goal of this paper.

2.2 AI algorithm

The strength of the programmable AI machine is measured depending upon their respective AI algorithm. With its help, the machines will determine the response of the corresponding human player's input [3].

The game starts by initializing the board. The main control of the game switches the system control to the human player and computer player. After each move irrespective of the player, the main control judges the outcome, if someone wins, the game draws or ends [3].

The turn-taking method is followed to give chance for both human and AI consecutively [3]. The input is received from the human player generated by a left click of the mouse. The mouse click is converted into a binary number using the in-built function *onClick* from the *tkinter* library. Now, the AI generates its input with the combination of two modules [3]. They are:

1. Search method [3]: Minimax algorithm
2. Evaluation function [3]: Heuristic search

2.2.1 Minimax algorithm

The Minimax algorithm is also named Negamax, which is considered the most adaptable method for two player game [10]. This provides an optimal method for two player game by understanding the optimal strategy of the opponent [7]. The Minimax algorithm considers the whole game as a tree [7]. But this nature requires extreme effort to implement but results in reducing the search space [7]. The player's move as per game state is considered as a node in the game tree. According to the Five-in-a-row game rule, as per the positions of the pieces, the child nodes are generated [2]. In general, Minimax scoring is always based on the current player’s point of view with an extra code of tracking who is the current player [11]. This

algorithm denotes the two players as MAX and MIN. Here, I consider MAX's move first followed by MIN's move and changes consecutively. MAX is referred to as the maximizing player who tends to increase his winning chance and MIN is referred to as minimising the opponent's score [2].

As entering the minimax loop, the game tree search objective is to obtain the winning strategy for MAX that results in attaining the good next move irrespective of its MIN node [2]. As per the minimax algorithm's structure, MAX has the highest chance of winning the game. Here the node traverse is carried out by a predefined depth function called the depth-first method. This function evaluates the leaf values and propagates the values level-by-level in accordance with the minimax principle [2]. The depth-first search refers to various search orders, the instances can be used to find the origin of the node implementation to get an optimal solution [5]. The technique of bounding the depth of the tree with the nth lookahead strategy is implemented, where n is considered as the number of explored levels of the tree [7].

Starting from 0th depth, the odd number of depths are considered as maximising node (MAX) and the even number of depths are considered as minimizing node (MIN).

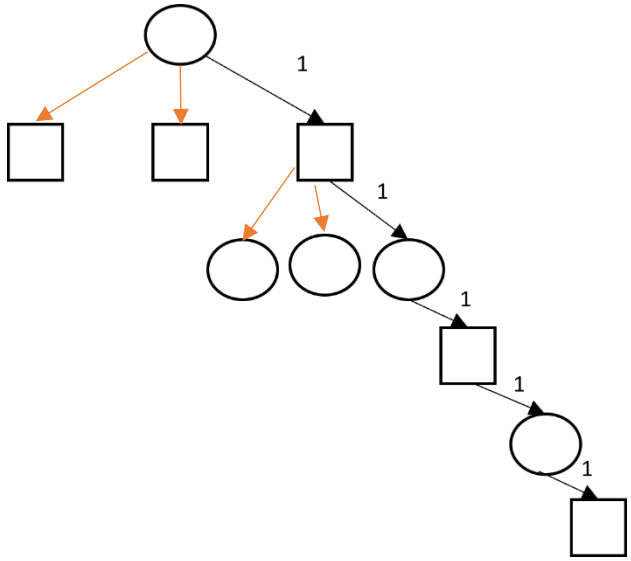
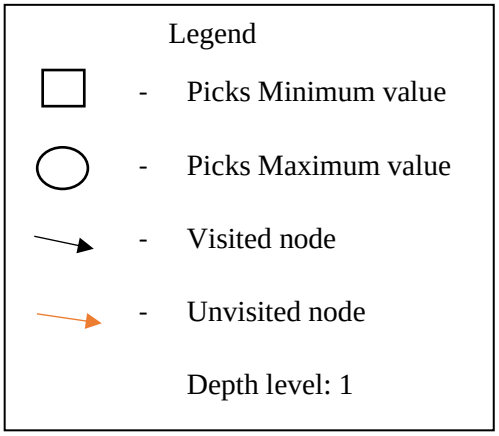


Fig. 1. Depth First Search [7].



The AI agent generates all possible moves in the respective depth of the game's board using a minimax algorithm. All the resulting game board is evaluated using a rule-based evaluation

function. For instance, in depth-1 minimum value from the evaluated value is selected and the maximum value is considered in depth-0 [4].

2.2.2 Heuristic Search

Numerous approaches have been proposed to improve the efficiency of the minimax strategy [2]. Artificial Intelligence (AI) encompasses a set of algorithms each serving for different purposes such as the formal method for reasoning and knowledge, algorithms for machine learning and more, and heuristic algorithm for searching, planning, and evaluating [1].

Since I implemented the Minimax algorithm, I had to incorporate an evaluation function mechanism to eliminate the randomness effect near the end of the game or to manage any other difficult situation [5]. Drazen Draskovic included the search algorithm in his Software system for learning AI algorithms (SAIL) to find the path from the initial to the target node [1].

Several search algorithms use the heuristic function, where SAIL uses the dynamic programming principle with a combination of realized algorithm to execute the algorithm to get better search performance [1].

An AI algorithm without a heuristic can only search in the game tree. The heuristic involves searching and protects from computational effort [5]. With the help of the *Valtorta* theorem, it is proved that the heuristic is helpful only if the auxiliary search is required to evaluate the node's weightage [5].

Heuristic search system performs the agent-centred search, which performs according to board position and current state and position of the player [5]. Here the game-playing programs are being conducted using the Minimax search with a lookahead option for limited stages to perform the next move. McCarthy invented this methodology in the AI chess game.

The purpose of defining a limited search is to reduce the search time and reasonable space because, in a realistic game, an unimaginable amount of time and memory is utilised to perform the complete search process [5]. Even though, this results in finding a non-deterministic search of the opponent's future move, but it helps in information limitation [5]. So, performing a heuristic with agent-centred search focuses on selecting the most relevant move in a reasonable time [5].

Generally, the heuristic search function assigns weightage to each and every node as it traverses [4]. The weightage allocation is always depending upon the state of the game that could be changed with that particular node/ move [4].

All the nodes that are generated in section 2.2.1 are visited by the heuristic function in a reverse DFS method and the weightages are allocated respectively. From this, the MAX and MIN values are opted according to the node's depth [2]

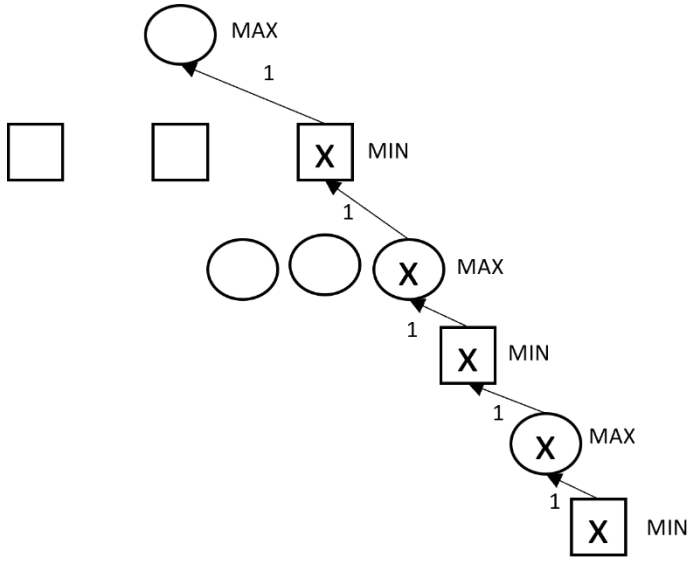
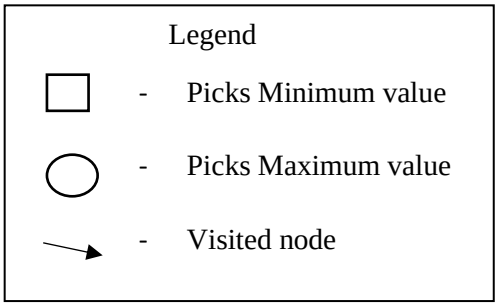


Fig. 2. Reverse Depth First Search [7].



3 Aim and objectives

This project aims to design an algorithm with the combination of a genetic search algorithm and an evaluation function. Project objectives are as follows,

1. Design Minimax algorithm
2. Design the Heuristic function that calls Minimax function
3. Frame parameters to evaluate the result
4. Design a Multithreading process to compare the algorithm designed by my teammate Zhuroan Yang

4 Design

4.1 Combination of Minimax and Heuristic

As we discussed in section 2, with the help of research I have designed the Five-in-a-row game with the combination of minimax and heuristic search function in the Python 21.0.1 platform. I preferred Python instead of MATLAB being an Electronics student because Python got all the necessary libraries inbuilt whereas MATLAB does not.

To achieve it in Python, declaring Minimax and heuristic in separate function is much easier and faster for computation.

In this entire function, I consider even number of depths as AI's move (i.e., player "o", depth level: 0,2,4,6,...), considering themselves as a Maximizing player who tends to increase their self's score. Accordingly, the odd number of depths are considered as the human player (i.e., player "x", depth level:1,3,5,...), considering as a Minimizing player who tends to reduce the AI's score.

Pseudo code for Minimax algorithm:

```
def minimax(board, player, current)

get available moves

if (current+1)>=depth:

    if ai=="x":

        opponent="o"

    else:

        opponent="x"
    return

elseif length in available moves == 0:

    return 0

  else:

    if player==ai:

        declare best score

       declare next player as human

    else:

        declare best score

       declare next player as AI

    for i in range(len(availMoves))

        move=availMoves[i]

        board[move[0]][move[1]]=player
```

```
return( board, player, current)

board[move[0]][move[1]]=0
```

Minimax function receives board and player's current status as an input.
With those inputs, the function checks all available move moves by creating a tree with the respective parent and child node. This process takes places for all the required depth level as specified. Depth First Search methodology is used here to traverse and create the node. Once the nodes are created, the heuristic function is called to produce all those nodes with an appropriate value.

This is an essential process, that traverse to all the node in the reverse Depth-First search process starting from the lowest level in-depth 2. All the previously generated nodes have a board that would be created with that particular from the available move list.

Pseudo code for Heuristic function:

```
def heuristic
declare all weightage values w1, w2……..wn
define conditions according to the game in a variable n1, n2……`
match condition variable == weightage

for i in range(size):

    for j in range(size):

        if board[i][j]==player:
            dt=evaluate_rows(i,j,player,board)

rt=(n1*w1+n2*w2+n3*w3+n4*w4+n5*w5+n6*w6+n7*w7+n8*w8+n9*w9)
    return rt
```

By visiting a particular node, it checks the board in it and the score. To score it, I follow a set of conditions as below:

1. Single "o" piece blocked by "x" in all the side- Score of o: 1
2. Single "o" piece having one empty cell next to it- Score of o: 10
3. Two consecutive "o" but blocked by "x" to match with third "o"- Score of o: 100
4. Two consecutive "o" having the possibility to match with third "o"- Score of o: 1000
5. Three consecutive "o" but blocked by "x" to match with fourth "o"- Score of o: 10000
6. Three consecutive "o", having the possibility to match with fourth "o"- Score of o: 100000
7. Four consecutive "o" but blocked by "x" to match with fifth "o"- Score of o: 1000000
8. Four consecutive "o", having possibility match with fifth "o"- Score of o: 10000000
9. Five consecutive "o"- Score of o: 1000000000

All nine conditions are checked horizontally, vertically and diagonally according to the game's rule. If a board matches with two or more condition the total score is the summation of all conditions score. By following these steps, "o" is scored in odd-numbered depths. The same steps are followed in even-numbered depths and "x" is scored by considering "o" as the opponent.
For example, consider the below board in depth= 1;

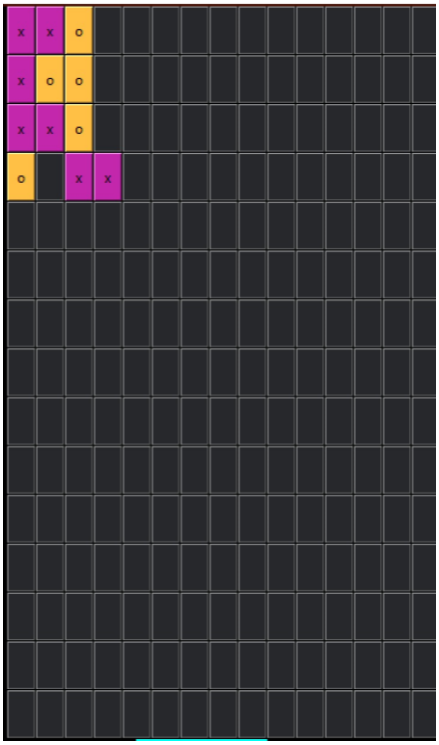


Fig. 3. Five-in-a-row board at depth 1.

Score of "x"= (Sum of Horizontals) + (Sum of Verticals) + (Sum of Diagonals)
= (10+1+10+100) + (10000+1+10+10+10) +
(1+1+100+1+1+100000+10+1+1000+10+10+1)
= 102288

After scoring all the nodes in respective depths, it returns to the minimax function that helps to compare the node and returns the MAX or MIN value respectively to the lower depth level. Finally, the move is selected from the node of depth 0.

For example, consider the below tree status with depth= 2

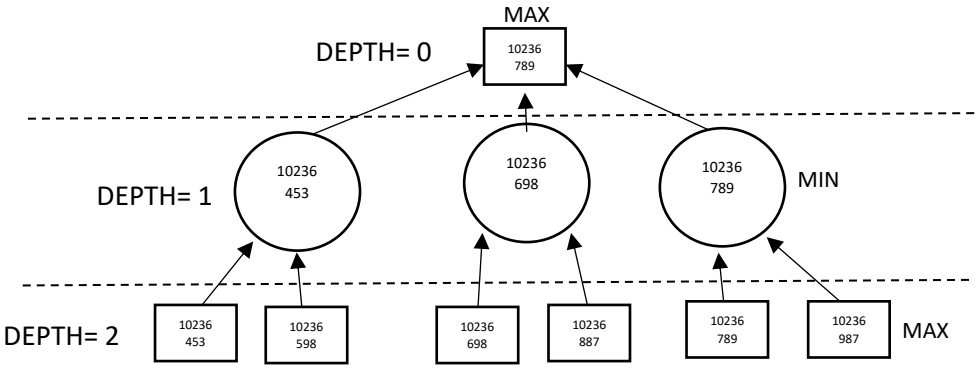


Fig. 4. Tree status at depth 2.

From depth 2, the MAX values are compared, and the minimum value is stored in depth 1's node. Then, the MIN values are compared, and the maximum is stored in depth 0's node. Now, this move is selected.

5 Experimental results

The above design procedure is followed to create the Five-in-a-row board game with the combination of Minimax and Heuristic search algorithm according to the research. To verify the research, I have implemented few steps, such as:

1. Time taken to make a move
2. Number of moves evaluated before making one move

I have implemented these steps in an evaluate function rather than implementing in the same function. A small modification is made to achieve this process. Instead of running the Minimax function initially after receiving the input from the human player, the program initiates the *evaluation* function. The actual minimax function is called from here. Before calling this loop *perf_counter()* is used that returns the system time and it is stored as the starting time. As soon as the Minimax function returns a move to be implemented the *pref_counter* is used to store the end time.

$$\textit{Time taken to find a best move} = \textit{End time} - \textit{Start time} \qquad \text{[Eq.1]}$$

By this formula, the time taken calculation is achieved.

The number of moves evaluated to implement one best move is attained by declaring a variable (*ni*) and incrementing it as traversing to each node. After generating the best move, the *ni* variable is returned and reinitiated to zero.

Initially, when the first AI winning algorithm was designed, humans felt that it is the end of game competition between interlevel players though AI is going to the pro. But later, the developers invented a method of tracking the human player's progress by letting the same winning algorithm evaluate the human brilliancy in making a move. With this achievement, even human players started supporting AI's development in the gaming stream.

To add some interesting element to this game, I have implemented a graph showing the winning possibility of human (i.e., AI's opponent).

Calculating the human winning possibility is achieved by subtracting the Heuristic value of depth 1's minimum value and the Heuristic value of the board with human player's move.

$$\textit{AI score} = \textit{Heuristic(Depth 1's MIN value)} - \textit{Heuristic(Current board)} \qquad \text{[Eq.2]}$$

With those verification steps, I have compared the Five-in-a-row game in three different depth conditions.

For instance: Depth: 0, 2, 4.

From Fig. 5: Depth 0:

Average time taken to decide a move = $\frac{0.13 + 0.13 + 0.17 + 0.15 + 0.17}{5} = 0.15 \longrightarrow 1$

Average number of moves calculated to get one best move = $\frac{3+5+7+10+10}{5} = 7 \longrightarrow 2$

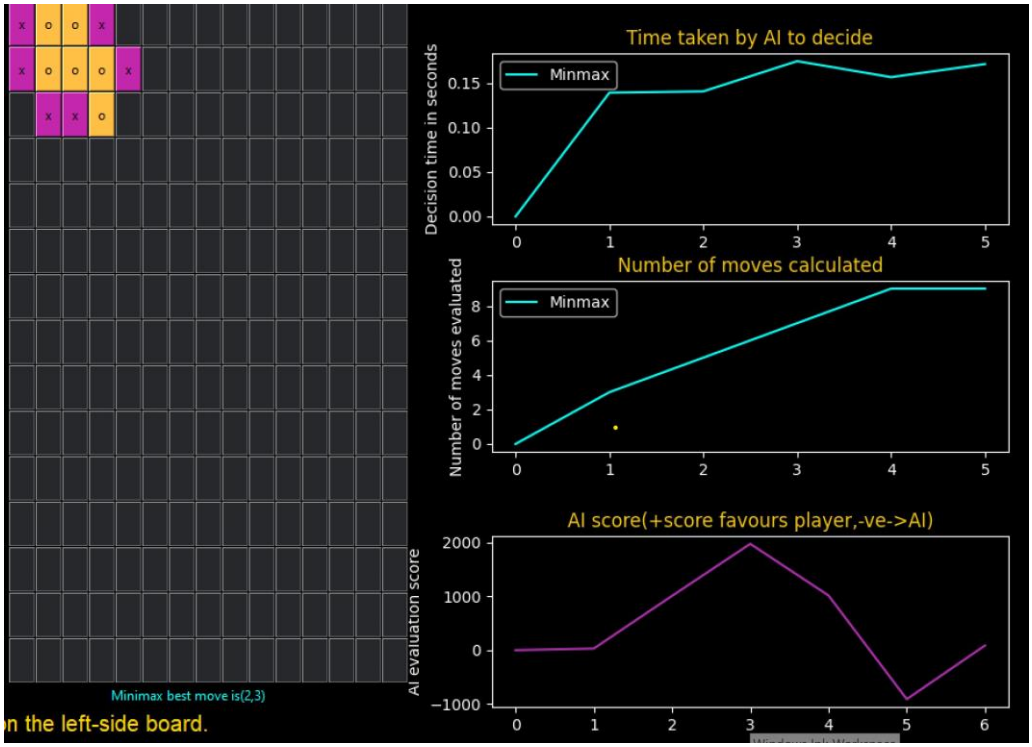


Fig. 5. Five-in-a-row board at depth 0.

From Fig. 6: Depth 2:

Average time taken to decide a move = $\frac{0.2 + 0.25 + 0.27 + 0.30 + 0.7}{5} = 0.344 \longrightarrow 3$

Average number of moves calculated to get one best move = $\frac{25+50+50+75+150}{5} = 70 \longrightarrow 4$

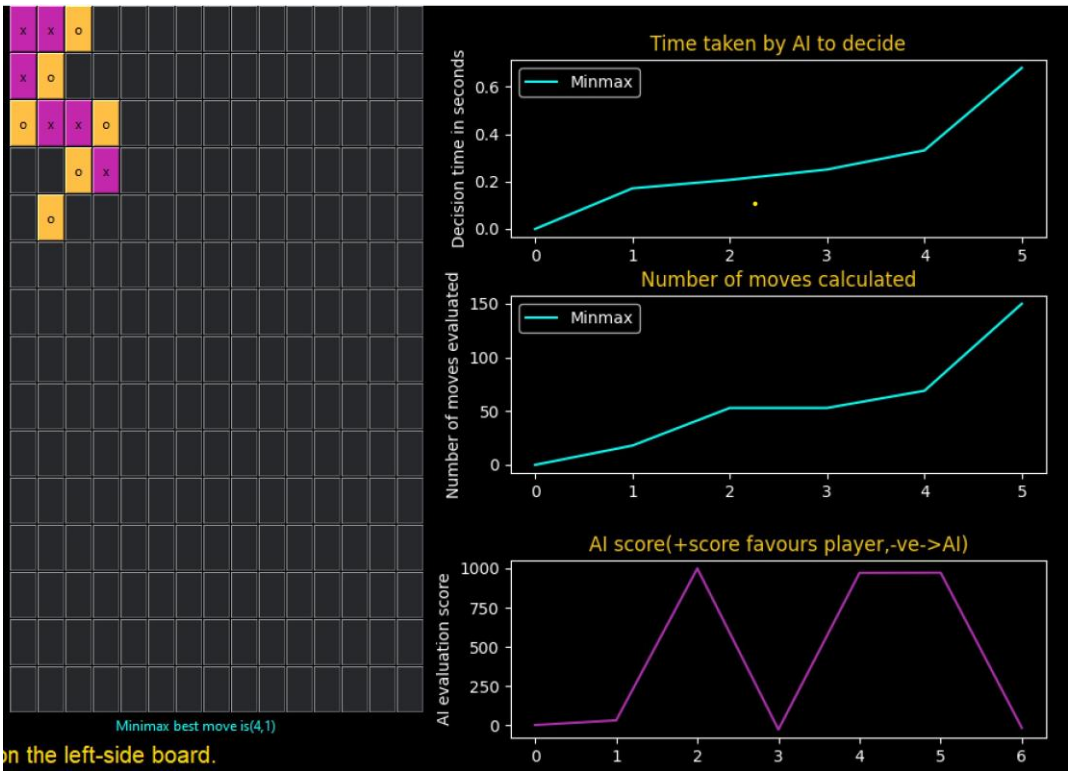


Fig. 6. Five-in-a-row board at depth 2.

From Fig. 7: Depth 4:

Average time taken to decide a move = $\frac{2+15+17+30+40}{5} = 20.8 \longrightarrow 5$

Average number of moves calculated to get one best move = $\frac{625+7500+7500+10000+15000}{5}$

$= 8125 \rightarrow 6$

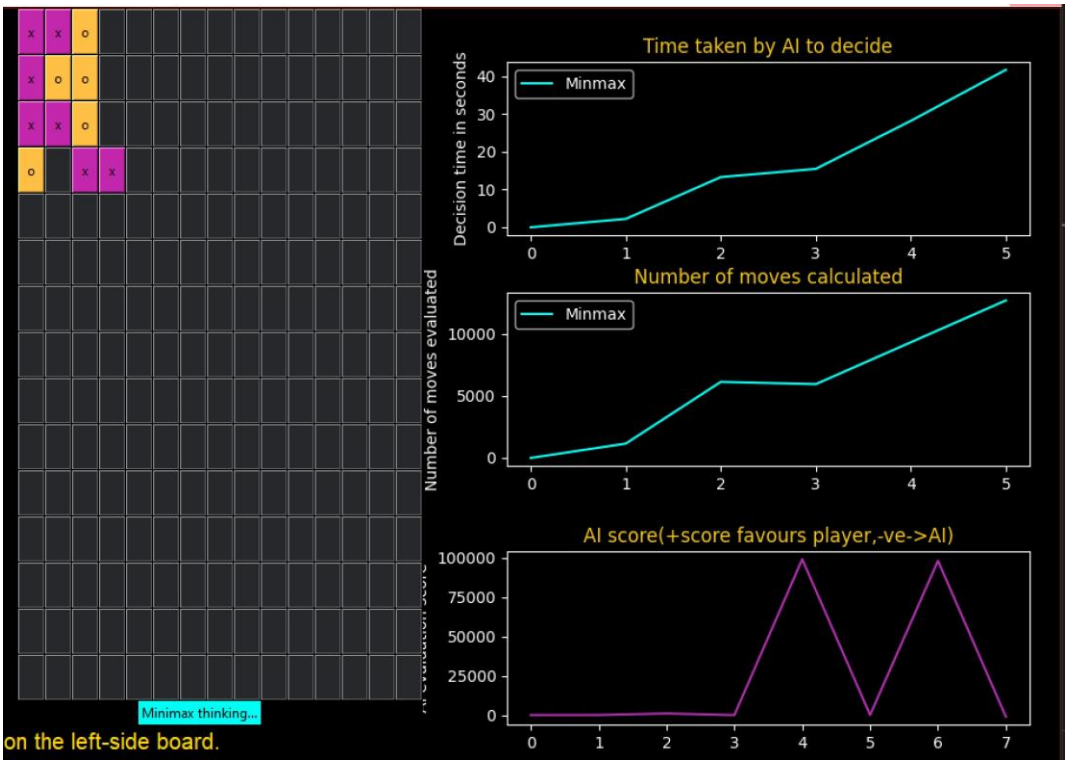


Fig. 7. Five-in-a-row board at depth 4.

In this project, my teammate Zhuoran Yang has implemented the Five-in-arow game with the AB pruning technique with the Heuristic search algorithm [9]. To compare both the algorithms it is better to run both the game (1. Minimax Five in a row, 2. AB pruning Five in a row) parallelly instead of running them separately. In this case, we used the Multithreading methodology. Multithreading is a process that helps in the efficient parallelisation of the packet classification algorithm on a multicore central processing unit using a Multithreading application program interface [8]. The categorisation of tasks takes place under different parameter. This method started to evolve due to the increase in people need for multitasking [8]. This could be implemented only in the Multiprocessor systems.

To achieve this both the algorithms are fed to a separate variable such as p1= *evaluateMinimax*, p2= *evaluateABpruning* and these variables are passed as threads to the processing unit.

Now, the player is allowed to give input only on the left side (ABPruning) of the board which is shared to the right side (Minimax) of the board as well. After receiving the input, the functions are passed in as threads and the respective outputs (moves) are returned. By following the same conditions as Minimax, AB-pruning's parameters are also displayed on the same graph for better comparison. Fig. 8. Shows the Graphical user interface of running AB pruning and Minimax algorithm parallelly with the comparison chart.

For example:

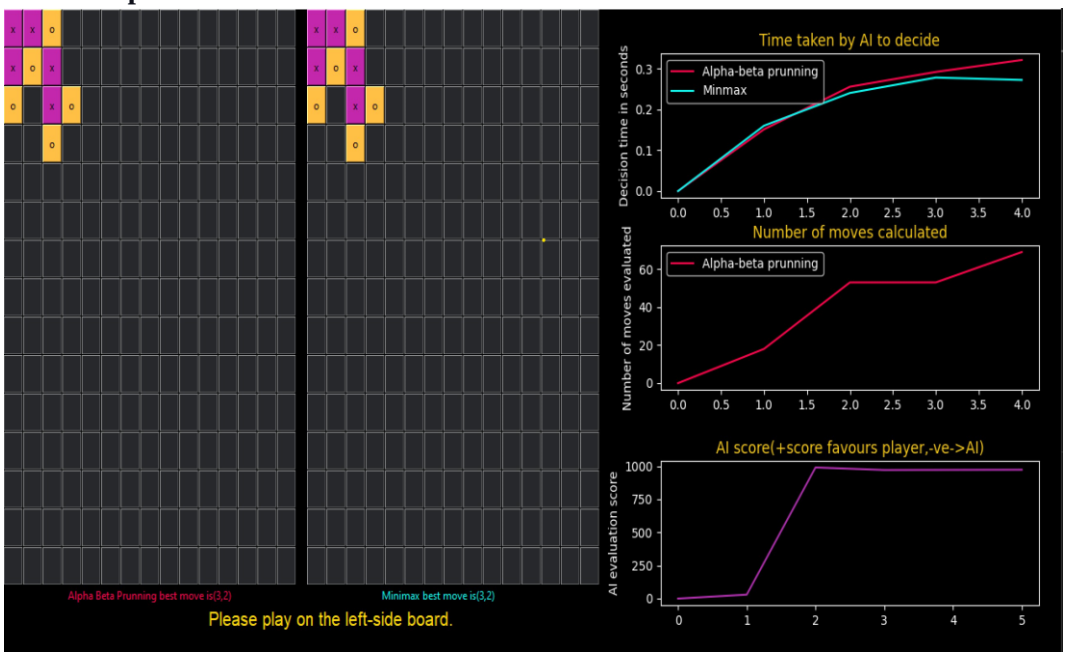


Fig. 8. Five-in-a-row board at depth 2.

From Fig. 8: Depth= 2:

Average time taken to decide a move = $\frac{0.15 + 0.25 + 0.3 + 0.35 + 0.7}{4} = 0.4375 \longrightarrow 7$

Average number of moves calculated to get one best move = $\frac{20+50+50+70}{4} = 47.5 \longrightarrow 8$

Findings:

1. As per the study, AB-pruning must take less time than Minimax in finding the best move. By comparing 3 and 7, Minimax takes less time than AB-pruning. This is because of the Multithreading. Here, multithreading would have passed Minimax's thread first and later AB-pruning would have passed.
2. In the case of number of moves calculated, AB pruning has evaluated lesser moves than Minimax, which satisfies the study

6 Conclusion

Artificial Intelligence is a stream that incorporates numerous complex algorithms to help in solving many different problems. These concepts are also being helpful for humans in playing a game during their leisure time and for some other human player who wishes to expertise in that particular game.

In this thesis paper, I have implemented an AI algorithm that fuses minimax and heuristic search to generate an efficient Five-in-a-row game with an additional feature of providing the human player with a score graph for every move that helps in tracking one's progress.

According to the result, Minimax with heuristic search (evaluation function) is a better combination of AI algorithm to run the game efficiently whereas we have found AB-pruning with Heuristic search is less time consuming by declining several unwanted moves during the evaluation process irrespective of the Multithreading methodology.

In the future, I will attempt to apply a combination of two genetic search algorithms (eg: Minimax with Monte Carlo Tree search) rather than combining the search algorithm with an evaluation function.

7 Acknowledgement

I would like to thank my Supervisor Peng Shi, Technical Advisor Xin Yuan and Yang Fei. They took in-charge of this project from the second phase. They boosted me and my team very well and provided us with exceptional guidance and support.

My other two team members have also stood along with me to achieve the project goals instead of getting deviated from the project scope and to meet the deadline as per the course requirement.

Finally, I am extremely thankful to the University of Adelaide for providing me with the necessary facilities in the workspace with computers, WIFI and library access to complete my research project successfully. This study has helped me to acquire a better understanding of the essential technology called Artificial Intelligence apart from regular Electronics course-based knowledge.

References

[1] Draskovic, D., M. Cvetanovic, and B. Nikolic, *SAIL—Software system for learning AI algorithms.* Computer applications in engineering education, 2018. **26**(5): p. 1195-1216.

[2] Tzung-Pei, H., H. Ke-Yuan, and L. Wen-Yang, *A genetic minimax game-playing strategy*. 1998, IEEE. p. 690-694.

[3] Venkateswara Reddy, L., et al., *Design and development of artificial intelligence (AI) based board game (Gobang) using android.* Materials today : proceedings, 2021.

[4] Halim, Z., *Evolutionary Search in the Space of Rules for Creation of New Two-Player Board Games.* 2014.

[5] Edelkamp, S. and S. Schroedl, *Heuristic Search : Theory and Applications*. 2011, San Francisco: Elsevier Science & Technology.

[6] Aljazzar, H. and S. Leue, *$K*$: A heuristic search algorithm for finding the k shortest paths.* Artificial intelligence, 2011. **175**(18): p. 2129-2154.

[7] Garcia Diez, S., J. Laforge, and M. Saerens, *Rminimax: An Optimally Randomized MINIMAX Algorithm.* IEEE Transactions on Cybernetics, 2013. **43**(1): p. 385-393.

[8] Abbasi, M. and M. Rafiee, *Efficient parallelisation of the packet classification algorithms on multi-core central processing units using multi-threading application program interfaces.* IET computers & digital techniques, 2020. **14**(6): p. 313-321.

[9] Zhuoran Yang, *A study of game search algorithms in Chinese chess*, 2020.

[10] Abdelbar, A.M., Alpha-Beta Pruning and Althöfer's Pathology-Free Negamax Algorithm. Algorithms, 2012. 5(4): p. 521-528.

[11] Millington, I., Artificial intelligence for games. 2nd ed. ed, ed. J.D. Funge. 2009, Boca Raton, Florida ;: CRC Press.

Appendix A

In this appendix, I have attached the source code:

A. Home file

```
     from tkinter import *
from tkinter.ttk import *
from PIL import ImageTk, Image
import sys
import os

root=Tk()
root.geometry('800x600')

# Welcome label
welcome = Label(root, text="Welcome!", padding=(30,20),font=("Helvetica", "16"
, "bold"))
welcome.pack()

#Select game
selectGame = Label(root, text="Select Game:", padding=(30,20), font=("Helvetic
a", "20", "bold"))
selectGame.pack()

#Function for play button
def playTicTacToe():
    #print("tic tac toe")
    global size
    size=3
    playButton.pack(pady=60)

def playFiveInARow():
    #print("Five in a row")
    selectDepth.pack()
    drop.pack()
    global size
    size=15
    playButton.pack(pady=60)

def play():
    #print(size)
    if size==3:
        root.destroy()
        os.system('python ./heuristics/multi_ttt.py')

    if size==15:
        root.destroy()
        #print("depth: "+str(clicked.get()))
        global depth
        depth=clicked.get()
```

```
        #path = 'python ./heuristics/heuristic_a_b.py '+str(depth)
        os.system("python ./heuristics/heuristic_a_b.py {depth}".format(depth=
depth))

def getSize():
    #print(size)
    return size

def getDepth():
    return depth

#Button1
pic1 = Image.open("TicTacToe.jpg")
resized1 = pic1.resize((50,50), Image.ANTIALIAS)
newPic1 = ImageTk.PhotoImage(resized1)
Button(root, text = 'Tic Tac Toe', image = newPic1,
                    compound = LEFT, command=playTicTacToe).pack(side = TOP, p
ady=10)

#Button2
pic2 = Image.open("5inarow.png")
resized2 = pic2.resize((50,50), Image.ANTIALIAS)
newPic2 = ImageTk.PhotoImage(resized2)
Button(root, text = '5 in a row', image = newPic2,
                    compound = LEFT, command=playFiveInARow).pack(side = TOP,
pady=10)

#Depth dropdown
selectDepth = Label(root, text="Select Depth:", padding=(30,20), font=("Helvet
ica", "12", "bold"))

options = ["0", "2", "4", "6"]
clicked = StringVar()
clicked.set( "2" )
drop = OptionMenu( root , clicked , *options)

#PlayButton
playButton = Button(root, text="Play", command=play, padding=(10,10))

mainloop()
```

B. Minimax Function

```
def ominimax(board,player,current):
    global minmax_ni
    availMoves=getReducedMoves(board)
    availMoves=sortMoves(availMoves,board,player)
    if (current+1)>=depth:
        if ai=="x":
            opponent="o"
        else:
            opponent="x"
        return (heuristic(ai,board)-heuristic(opponent,board))

    elif len(availMoves) == 0:          # if drawn, no reward
        return 0
    else:
        if player==ai:
            best_score=-9999999999
            nextPlayer=human
        else:
            best_score=9999999999
            nextPlayer=ai
        for i in range(len(availMoves)):
            minmax_ni+=1
            move=availMoves[i]
            board[move[0]][move[1]]=player
            #print("move played ",move,"by ", player)
            result=ominimax(board,nextPlayer,current+1)
            board[move[0]][move[1]]=0
            if player==ai:
                #if result>=10:
                #    return result
                if result > best_score:
                    best_score=result
            else:
                #if result <=-10:
                #    return result
                if result<best_score:
                    best_score=result
        return best_score
```

C. Heuristic Function

```
def heuristic(player,board):
    w1=1
```

```
    w2=10
    w3=100
    w4=1000
    w5=10000
    w6=100000
    w7=1000000
    w8=10000000
    w9=1000000000
    n1=0
    n2=0
    n3=0
    n4=0
    n5=0
    n6=0
    n7=0
    n8=0
    n9=0
    for i in range(size):
        for j in range(size):
            if board[i][j]==player:
                dt=evaluate_rows(i,j,player,board)
                if dt!=None:
                    n1+=dt["closed_one"]
                    n2+=dt["single_mark"]
                    n3+=dt["closed_two"]
                    n4+=dt["two_in_row"]
                    n5+=dt["closed_three"]
                    n6+=dt["three_in_row"]
                    n7+=dt["four_in_row"]
                    n8+=dt["straight_fours"]
                    n9+=dt["five_in_row"]
                dt=evaluate_cols(i,j,player,board)
                if dt!=None:
                    n1+=dt["closed_one"]
                    n2+=dt["single_mark"]
                    n3+=dt["closed_two"]
                    n4+=dt["two_in_row"]
                    n5+=dt["closed_three"]
                    n6+=dt["three_in_row"]
                    n7+=dt["four_in_row"]
                    n8+=dt["straight_fours"]
                    n9+=dt["five_in_row"]
                dt=evaluate_diags(i,j,player,board)
                if dt!=None:
                    n1+=dt["closed_one"]
                    n2+=dt["single_mark"]
                    n3+=dt["closed_two"]
                    n4+=dt["two_in_row"]
                    n5+=dt["closed_three"]
```

```
                    n6+=dt["three_in_row"]
                    n7+=dt["four_in_row"]
                    n8+=dt["straight_fours"]
                    n9+=dt["five_in_row"]
    rt=(n1*w1+n2*w2+n3*w3+n4*w4+n5*w5+n6*w6+n7*w7+n8*w8+n9*w9)
    return rt
```

D. To Evaluate values for graph

```
def evaluateMinMax():
    global canPlay,minmax_ni
    availMoves=getReducedMoves(data_board_copy)
    availMoves=sortMoves(availMoves,data_board_copy,ai)
    board=deepcopy(data_board_copy)
    best_score=-9999999999
    best_move=None
    start=perf_counter()
    mm_label.configure(text=mm_text+"thinking...",bg="#00fff5",fg="black")
    for l in availMoves:
        #print("inside play eval minmax. avail moves are",availMoves)
        board[l[0]][l[1]]=ai
        minmax_ni+=1
        #print("move played ",l," by O")
        result=ominimax(board,human,0)
        board[l[0]][l[1]]=0
        if result > best_score:
            best_move=l
            best_score=result
    end=perf_counter()
    ttaken=end-start
    minmax_time_array.append(ttaken)
    minmax_moves_array.append(minmax_ni)
    minmax_ni=0
    mm_label.configure(text=mm_text+"best move is({0},{1})".format(best_mov
e[0],best_move[1]),bg="black",fg="#00fff5")
    makeMoveInLabelBoard(best_move)
    canPlay=True
```