Hanoi University of Science and Technology
School of Information and Communication Technology

==========

PROJECT REPORT

TOPIC: Tic Tac Toe game with AI player

Class: 136462 (IT3160E)
Lecturer: Ph. D Nguyen Nhat Quang

Students

Nguyễn Mạnh Cường	20215184
Phan Công Đại	20210147
Bùi Phương Nam	20215227
Nguyễn Trọng Huy	20210451

Hanoi, 2023

TABLE OF CONTENTS

ABSTRACT

Artificial intelligence (AI) is shaping the future of humanity across nearly every industry. It is already the main driver of emerging technologies like big data, robotics and IoT, and it will continue to act as a technological innovator for the foreseeable future.

"Introduction to Artificial Intelligence – IT3160E" course provides us with fundamental general understandings about the AI world. In our project, we applied adversarial search algorithms we have learnt and some variants in our progam named: Tic Tac Toe.

In this report, we will explain the theoretical basis of Minimax with alpha-beta pruning, Negamax with alpha-beta pruning and Monte-Carlo tree searching (MTCs) and how we implement all these algorithms. After that, we analyse the effectiveness of each one and make comparisons.

Keywords: adversarial serach, zero-sum game, minimax, negamax, alpha-beta pruning, MCTs.

CHAPTER 1: INTRODUCTION

I. History of Tic Tac Toe

The early Roman version was known as terni lapilli, or ***three pebbles at a time***. In the mid-1800s, Britain used the name ***noughts and crosses***, with ***nought*** referring to the O's (or zeros) used in the game. The phrase ***tic-tac-toe*** wasn't used until the late 1800s, and there's some findings that report it might have been used to describe a completely different game.

The United States officially adopted the name ***tic tac toe*** in the 20th century. Historians aren't completely clear about the origin of the game's name, but it could be referring to the noise of repetitive ticking or writing that goes along with the game's play.

II. Rule of Tic Tac Toe

Tic tac toe is one of the most popular games. When playing Tic-Tac-Toe, there are several simple and easy-to-follow rules that you should observe. The rules that you need to be aware of include the following:

- The traditional version of Tic-Tac-Toe must be played on a 3×3 grid that contains 9 squares.

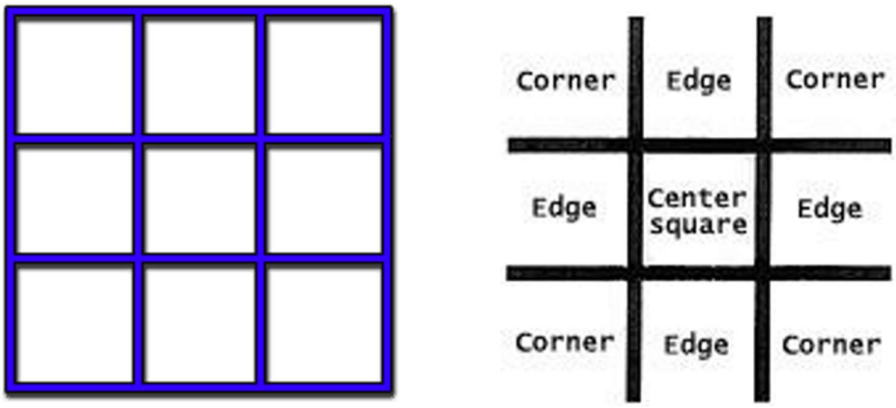

Figure 1: Table sample

- Players must select their mark before the game starts (either X or O) and must play the entire game using the same mark.

- Players must take turns, making only 1 mark with each turn.

- Marks can only be placed in empty squares, and once they are placed, it is permanent.

Assume, Player 1 – X and Player 2 – O. Having a sequence:

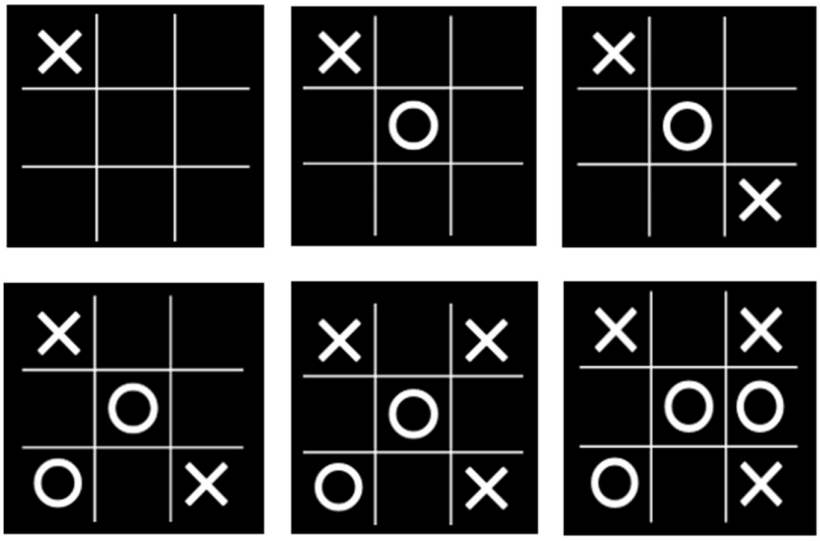

Figure 2: A sequence list of a game sample

- The winner is the first player to get 3 of their marks in a straight line (the line can be positioned diagonally, vertically, or horizontally).

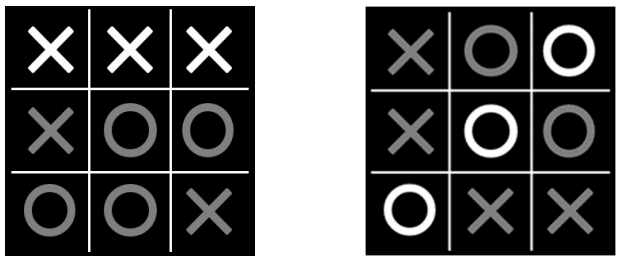

Figure 3: Player 1 wins (left) – Player 2 wins (right)

- The game is over when all 9 squares are filled with marks, even if none of the players have a straight line of 3 marks.

- If neither player has a straight line of 3 marks, it is considered a tie.

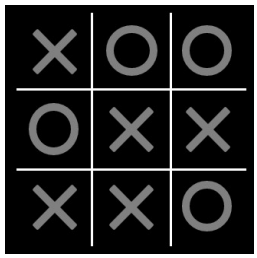

Figure 4: A tie

III. Problem descriptions

We will implement an intelligent agent that can play Tic Tac Toe. This agent has the following properties:

- Performance measure: high winning rate, good consumption time, clever places

- Environment: a 3x3 grid including 9 squares, X and O dots, players

- Actuator: make a move
- Sensor: Observe the state on 3x3 grid

The game is perfect information, competitive multiagent, deterministic, static, discrete, and sequential.

Based on game rules, this is a zero – sum game. With the goal to win against the human player or the other computer player.

From those properties, the agent uses adversarial search.

The game can be represented as follows:

- Initial state: The board is represented as a 3x3 matrix. Each element is filled with ‘0’ value to indicate a blank space. Every time the ‘X’ player puts a mark on a square, then the value corresponding to that square coordinates will change to ‘1’. Similarly, the ‘O’ player will change the value corresponding to that square coordinate to ‘-1’.
- Successor function: A list of available legal moves and corresponding state of that turn.
- Goal test: Everytime, a player finishes his/her turn then the function evaluation run to check if the game is over or not.
- Evaluation: A set of all possible win moves.

CHAPTER 2: ALGORITHMS

To implement the agents, we apply these following algorithms:

- Random move strategy
- Minimax with alpha-beta pruning
- Negamax with alpha-beta pruning – a variance of Minimax strategy
- Monte-Carlo Tree search

In our project, we will evaluate the effectiveness of each algorithm and make comparisons between them.

I. Random move strategy

As its name recommends, in our program, this strategy simply returns a possible random move on the board for the computer player.

Time and space complexity: O (1), assume that the implementation of random function is negligible

```
def random_move(self, aimove):
    x_pos = random.randint(0,2)
    y_pos = random.randint(0,2)
    while (self.markers[x_pos][y_pos]!=0):
        x_pos = random.randint(0,2)
        y_pos = random.randint(0,2)
    self.markers[x_pos][y_pos]=aimove
```

Figure 5: The simple implementation of random move strategy

In this function, every time taking in the (x_pos, y_pos) coordinates, it will check if this move is possible or not and returns the first possible one.

II. Minimax with alpha-beta pruning

a. Minimax algorithm:

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, if your opponent also plays optimally.

In Minimax, the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. In each state if the maximizer has upper hand, then, the score of the board will tend to have some positive value. If the minimizer has the upper hand in that board state, then it

will tend to have some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Figure 6: Peseudo code for Minimax algorithm

Since, its ***time complexity*** is O(b^d), where b is the branching factor and d is count of depth or ply of graph or tree. ***Space complexity*** is O(bd) where b is branching factor into d is maximum depth of tree like DFS.

However, this raises a problem. In pure minimax algorithm might face the exploding problem which means it will search all the branches in tree that contains reverse results for the computer player. Indeed, alpha-beta pruning is reagard as a suitable trick to solve kind of this problem.

b. Alpha-beta pruning:

Alpha-Beta pruning is not actually a new algorithm, but rather an optimization technique for the minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

Alpha is the best value that the maximizer currently can guarantee at that level or above.

Beta is the best value that the minimizer currently can guarantee at that level or below.

If the move ordering for the search is optimal (meaning the best moves are always searched first), its ***time complexity*** is estimated about O(b^(d/2))

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            if value > β then
                break (* β cutoff *)
            α := max(α, value)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            if value < α then
                break (* α cutoff *)
            β := min(β, value)
        return value
```

```
(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
```

Figure 7: the simple pseudo code for minimax with ab pruning

However, in case of the first player is computer, its first move is by default of the function at the first top left corner of the board, so that, we implement this move is a random move. The full implementation of minimax with alpha-beta pruning is presented in our source code.

III. Negamax with alpha-beta pruning

Negamax search is a variant form of minimax search that relies on the zero-sum property of a two-player game.

This algorithm relies on the fact that max (a, b) = - min (-a, -b) to simplify the implementation of the minimax algorithm. More precisely, the value of a position to player A in such a game is the negation of the value to player B. Thus, the player on the move looks for a move that maximizes the negation of the value resulting from the move: this successor position must have been valued by the opponent. The reasoning of the previous sentence works regardless of whether A or B is on the move. This means that a single procedure can be used to value both positions. This is a coding simplification over minimax, which requires that A selects the move with the maximum-valued successor while B selects the move with the minimum-valued successor.

Algorithm optimizations for minimax are also equally applicable for Negamax. Alpha-beta pruning can decrease the number of nodes the negamax algorithm evaluates in a search tree in a manner similar to its use with the minimax algorithm.

```
function negamax(node, depth, α, β, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node

    childNodes := generateMoves(node)
    childNodes := orderMoves(childNodes)
    value := −∞
    foreach child in childNodes do
        value := max(value, −negamax(child, depth − 1, −β, −α, −color))
        α := max(α, value)
        if α ≥ β then
            break (* cut-off *)
    return value
```

```
(* Initial call for Player A's root node *)
negamax(rootNode, depth, −∞, +∞, 1)
```

Figure 8: the pseudo code for negamax algorithm with ab pruning

In fact, the negamax algorithm by defenitions works the same as minimax algorithms but in clearer and shorter code lines. The full implementation of negamax algoritm with alpha-beta pruning is also presented in our source code.

IV. Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a best-first search method that does not require a positional evaluation function. It is based on a randomized exploration of the search space.

In our project, we will introduce the very first simple implementation of Monte-Carlo Tree Search.

The focus of MCTS is on the analysis of the most promising moves, expanding the search tree based on random sampling of the search space. The application of Monte Carlo tree search in games is based on many playouts, also called rollouts. In each playout, the game is played out to the very end by selecting moves at random. The final game result of each playout is then used to weigh the nodes in the game tree so that better nodes are more likely to be chosen in future playouts.

MCTS consists of four main steps, repeated ‘NoOfSimulations’ times. The steps are as follows. (1) In the selection step the tree is traversed from the root node until we reach node E, where we select a child that is not part of the tree yet. (2) Next, in the expansion step this child of E is added to the tree. (3) Subsequently, during the simulation step moves are played in self-play until the end of the game is reached. The result R of this “simulated” game is +1 in case

of a win for Black (the first player in Go), 0 in case of a draw, and −1 in case of a win for White. (4) In the backpropagation step, R is propagated backwards, through the previously traversed nodes. Finally, the move played by the program is the child of the root with the highest visit count. An outline of the four steps of MCTS are depicted in the figure below.

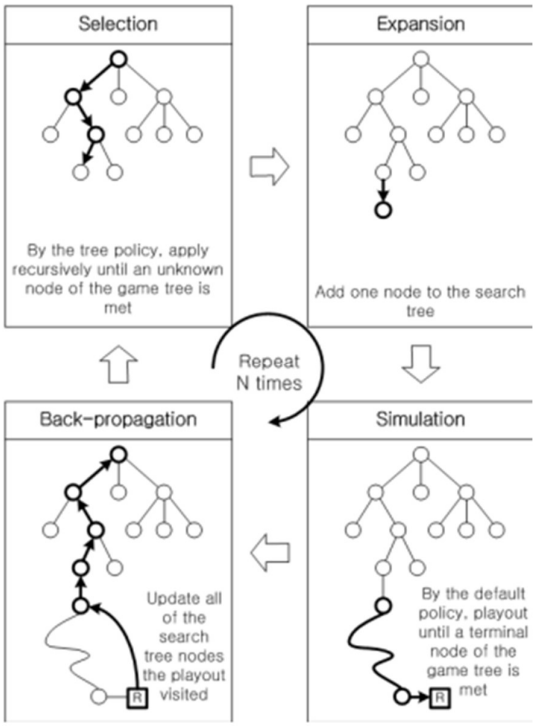

Figure 9: Four steps of MCTs algorithm

Here is the pseudo code for MCTs algorithm:

```
function MCTS(Position, NoOfSimulations)
  for each child n available from Position
    reward[n] = 0
    for i from 1 to NoOfSimulations
      POS = copy(Position)
      while (POS is terminal node)
        MCTS(POS, random child from POS)
      end while
      reward[n] += result(POS)
    end for
    reward[n] /= NoOfSimulations
  end for
  return(child i with highest reward[i])
end function
```

Figure 10: The basic pseudo code for MCTs algorithm

If MCTS is used in its basic form without any improvements, it may fail to suggest reasonable moves. It may happen if nodes are not visited adequately which results in inaccurate estimates. In this case, if we increase the value of

NoOfSimulations, the higher accuracy of computer in the game. However, it means that the time to excecute the code also increases.

Although it has been proven that the evaluation of moves in Monte Carlo tree search converges to minimax, the basic version of Monte Carlo tree search converges only in so called "Monte Carlo Perfect" games. However, Monte Carlo tree search does offer significant advantages over alpha–beta pruning and similar algorithms that minimize the search space.

CHAPTER 3: ANALYSIS

In this section, we compare the running time supported by actual runtime data collected from running each algorithm against itself. Both theoretical and practical clock time analyses have been done for the algorithms as described below.

To analyze the performance of algorithms, we have set up the matches as follows:

- AI versus AI:

We let the algorithms play against each other. The fixtures are shown in this table:

Random	Minimax with α,β pruning	40 games for each match
Minimax with α,β pruning	Negamax with α,β pruning	
Minimax with α,β pruning	Monte Carlo Tree Search	
Monte Carlo Tree Search	Random	

Table 1: AI fixtures

- AI versus human player: We only use Minimax with α,β prunning and Monte Carlo Tree Search to play with human players.

People that we have asked for help can be divided into 2 subgroups:

- Novice: These players understand the rules and have played at least 10 games. They have no or less experience, and do not have any specific strategies.

- Intermediate: These players understand the rules and have been played at least 20 games. They have more experience than Novice group, know some strategies.

Everybody is asked to play at least 50 games with our AI on both sides. Especially, we have 100 games between Minimax (with alpha-beta pruning) and Monte-Carlo Tree Search (with 10000 simulations)

After processing the data from all matches, we obtain the following results:

- AI versus AI

Player 1	Player 2	Player 1 wins (games)	Draw (games)	Player 2 wins (games)
Random	Minimax with with α,β pruning	0	1	39
Minimax with α,β	Negamax with	0	40	0

pruning	α,β pruning			
Minimax with α,β pruning	Monte Carlo Tree Search	6	34	0
Monte Carlo Tree Search	Random	37	3	0
Monte Carlo Tree Search (10000 simulations)	Minimax with α,β pruning	0	71	29

Table 2: AI versus AI - Match records

- AI versus human

Player 1	Player 2	Player 1 wins (games)	Draw (games)	Player 2 wins (games)	Total games
Novice	Minimax with α,β pruning	0	38	12	50
Intermediate	Minimax with α,β pruning	0	37	3	40
Novice	Monte Carlo Tree Search	0	23	17	40
Intermediate	Monte Carlo Tree Search	13	27	8	48

Table 3:AI versus human - Match records

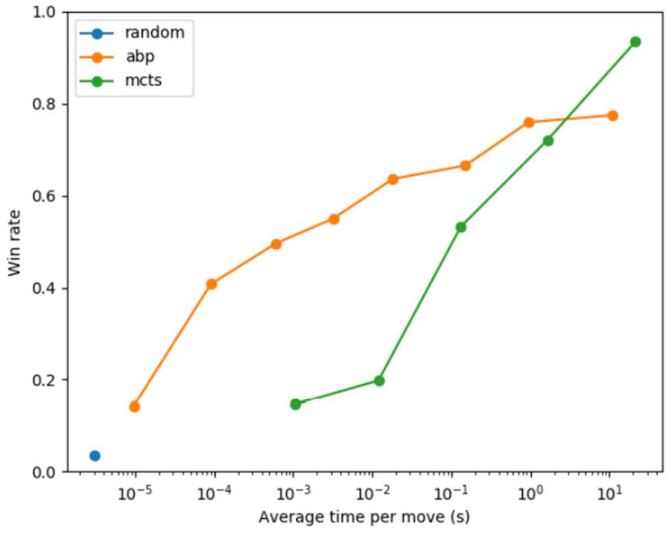

Figure 11: The average running time and average win rate on every algorithm

The figure below shows the analysis of MCTS based on the number of simulations (x10 times) (including Random Move). In the graph: The X represents MCTS AI player while O is the Random move one.

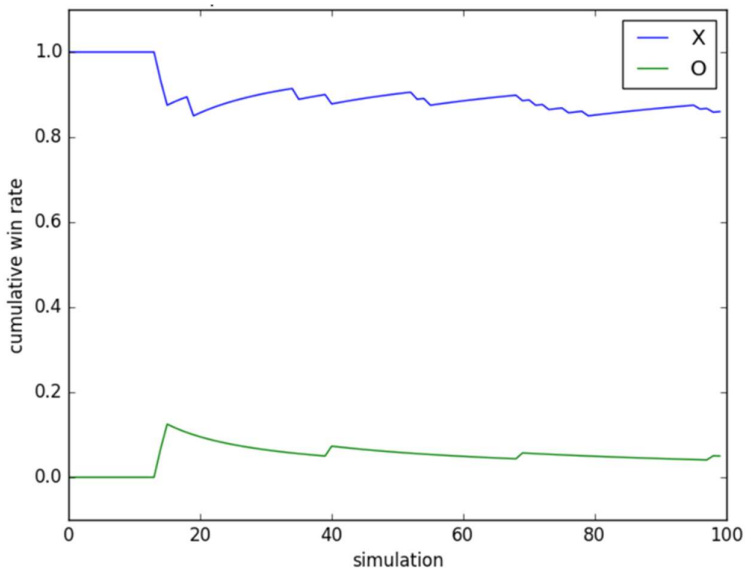

Figure 12: Win Rate for for the match between Random move and MCTS (based on different number of simulations)

We also test the match between MCTS and Minimax, with 100 Trials (the number of simulations):

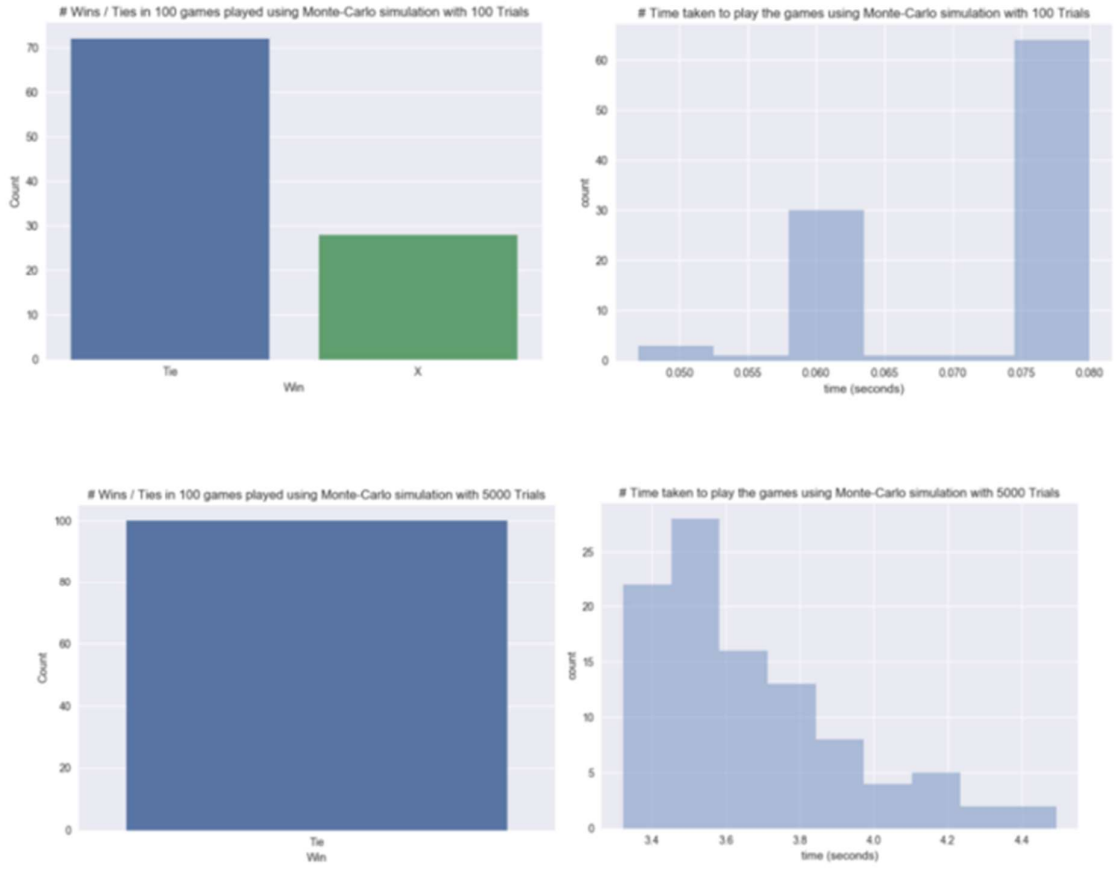

Figure 13: The number of winning games and the running time of MCTS algorithms for the match between MCTS AI player versus Minimax AI Player in every 100 matches

For each of the following different numbers of simulations, 100 different TicTacToe games are played between the 2 AI players. We have increased the number of simulations by 1000 for each test, and we got the best result at around 10000. The results obtained are shown below, they show the trade-off clearly. As the number of simulations increases:

- Number of draw games increases.
- Total time to finish the game increases on average.

With many trials, as large as 10000 each step, all the 40 games end in ties. The only drawback is that the running time is quite large, compared to the other algorithms (Minimax, Negamax), it needs total time ~ 3-4 seconds on average to play each game.

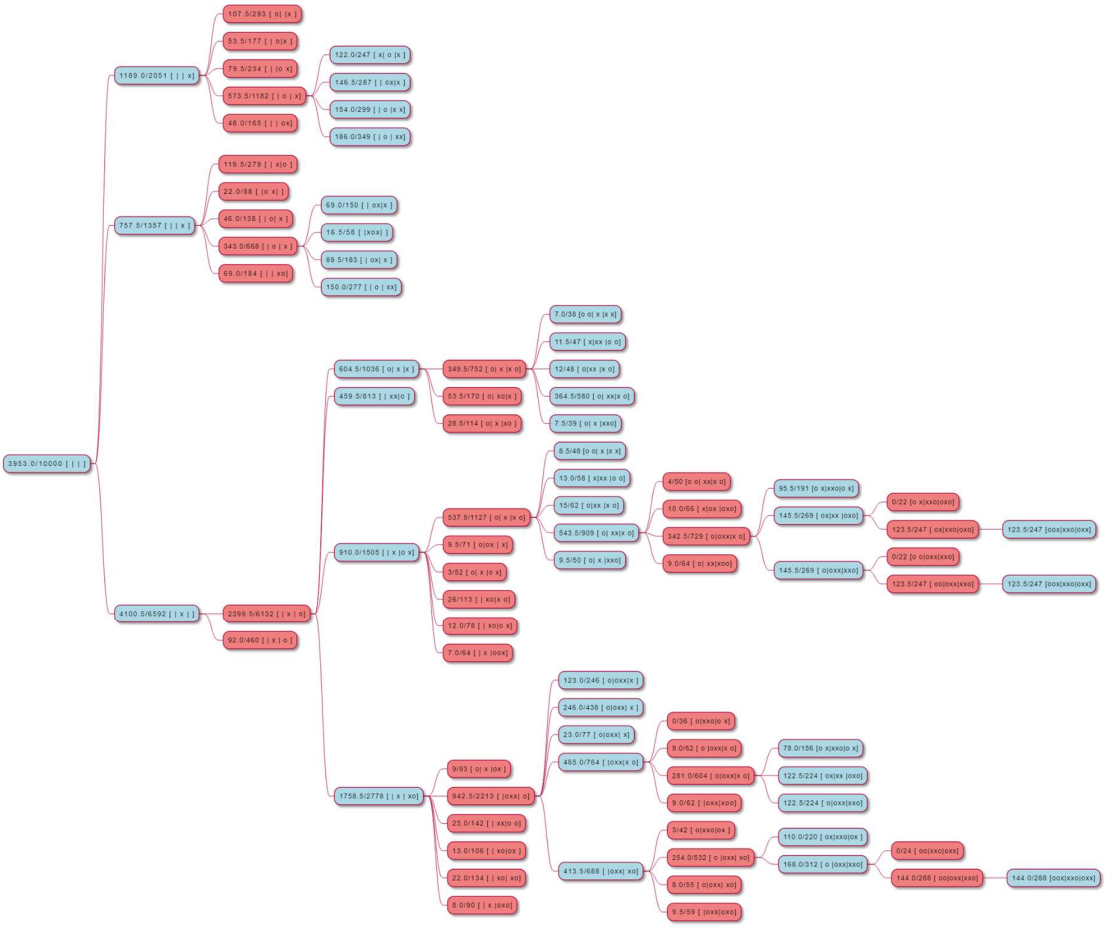

Figure 14: The tree about the ratio of wins to total possible playouts

By observing the results in all games, we have following comments:

- Minimax (with Alpha-Beta Pruning) has the best win rate - almost win or draw

- Alpha – Beta pruning works very efficiently. Minimax and Negamax show no difference in their confrontation.

- The result of MCTS is not that good as we expected, and its win-rate depends heavily on the number of simulations.

Some basic advantages of MCTS over Minimax (and its many extensions, like Alpha-Beta pruning and all the other extensions over that) are:

- MCTS does not **need** a heuristic evaluation function for states. It can make meaningful evaluations just from random playouts that reach terminal game states where you can use the loss/draw/win outcome. So, if you're faced with a domain where you have absolutely no heuristic domain knowledge that you can plug in, MCTS is likely a better choice. Minimax **must** have a heuristic evaluation function for states (**exception**: if you game is so simple that you can afford to compute the complete game tree and reach all terminal game states immediately from the the initial game state, you don't need heuristics). If you do have strong evaluation functions, you can still incorporate them and use them to improve MCTS too; they're just not strictly necessary for MCTS.

- MCTS has simpler **anytime** behavior; you can just keep running iterations until you run out of computing time, and then return the best move. Typically, we expect the performance level of MCTS to grow with computation time / iteration count relatively smoothly (not always 100% true, but intuitively you can usually expect something like this). You can sort of achieve anytime behavior in minimax with **iterative deepening**, but that's usually a bit less "smooth", a bit more "bumpy"; this is because every time you increase the search depth, you need *significantly* more processing time than you did for the previous depth limit. If you run out of time and have to abort your current search at your current depth limit, that last search will be completely useless; you'll have to discard it and stick to the results from the previous search with the previous depth limit.

A difference, which is not necessarily an advantage or disadvantage either way in the general case (but can be in specific cases):

- The computation time of MCTS is generally dominated by running (semi-)random playouts. This means that functions for computing legal move lists, and applying moves to game states, typically dictate how fast or slow your MCTS runs; making these functions faster will generally make your MCTS faster. On the other hand, the computation time of Minimax is generally dominated by copying game states (or "undoing" moves, which is an operation that in most games will require additional memory usage for game states to be possible) and heuristic evaluation functions (though the latter are likely to also become important in terms of computation cost in MCTS if you choose to

include them there). In some games it will be easier to provide efficient implementations for one of these, and in other games it may be different.

A basic advantage of Minimax over MCTS:

In settings where MCTS can only run **very few** iterations relative to the branching factor (or in the extreme case, fewer iterations than there are actions available in the root node), MCTS will perform extremely poorly / close to random play. We've noticed this being the case for quite a decent number of games in our general game system Ludii (where the "general game system" often implies that games are implemented less efficiently than they could be in a dedicated single-game-specific program) with low time controls (like 1 second per move). This same general game setting often makes it difficult to find super strong heuristics, but it's generally still possible to come up with some relatively simple ones (like just a simple material heuristic in chess). An alpha-beta search with just a couple of search plies and a basic, simple heuristic will often outperform a close-to-random MCTS if the MCTS can't manage to run significantly more iterations than it has legal moves in the root node.

CHAPTER 4: CONCLUSIONS

I. Experimental results

a) Monte Carlo

At first glance, it's difficult to trust that an algorithm relying on random choices can lead to smart AI. However, thoughtful implementation of MCTS can indeed provide us with a solution which can be used in many games as well as in decision-making problems. There are several advantages:

- It does not necessarily require any tactical knowledge about the game
- A general MCTS implementation can be reused for any number of games with little modification
- Focuses on nodes with higher chances of winning the game
- Suitable for problems with high branching factor as it does not waste computations on all possible branches
- Algorithm is very straightforward to implement
- Execution can be stopped at any given time, and it will still suggest the next best state computed so far

If MCTS is used in its basic form without any improvements, it may fail to suggest reasonable moves. It may happen if nodes are not visited adequately which results in inaccurate estimates. However, MCTS can be improved using some techniques. It involves domain specific as well as domain-independent techniques. In domain specific techniques, simulation stage produces more realistic play outs rather than stochastic simulations. Though it requires knowledge of game specific techniques and rules.

b) Minimax

Minimax algorithm is one of the most popular algorithms for computer board games. It is widely applied in turn-based games. It can be a good choice when players have complete information about the game. It may not be the best choice for the games with exceptionally high branching factor. Nonetheless, given a proper implementation, it can be a pretty smart AI.

In there, we have studied adversarial – search algorithm Minimax by implementing it in the game Tic Tac Toe. We have analyzed the game features to create the heuristic evaluation function to estimate the utility of game state for searching. We observed the performance of the Minimax algorithm in different matches against various opponents, and through observation, we have comments about its strengths and weaknesses.

II. Future improvements

We derive some proposals for future development:

- Improving the heuristic evaluation function, as well as considering different periods of the game (beginning, middle, end) to make the evaluation more flexible and accurate.

- Minimax is a type A strategy – considers all possible moves to a certain depth, then estimates the utility of that state. We can consider using some type B strategy methods, which ignore moves that look bad, and follow the branch that looks “reasonable” as far as possible.

Use some techniques in Machine Learning like Reinforcement Learning, Deep Learning, … can be used to create a more powerful AI. AlphaGo is the shining example that uses these learning techniques to defeat human players in Tic Tac Toe

- Our program currently does not limit the time for each turn. We want to extend the functionality of our program by adding the time limit for the players. In that case, we can use some searching techniques with time constraints, like Minimax search with Iterative Deepening…

- MCTs is an intelligent agent but has many disadvantages in case of no more development. However, MCTs still has many potentials to a more extended table of Tic-Tac-Toe. In fact, MCTs is a milestone when applied in Machine Learning, artificial neural networks (a deep learning method).

REFERENCES

[1] N. N. Quang, Introduction to Artificial Intelligent - IT3160E slides.

[2] P. d. G. Weiss, Monte-Carlo Tree Search, G.M.J-B. Chaslot, 2010.

[3] B.-D. Lee, "Enhanced strategic Monte-Carlo Tree Search algorithm," Journal of Korea Game Society, 2016.

[4] A. J. Paul, "Randomised fast no-loss expert system to play tic-tac-toe like a human," *Cognitive Computation and Systems,* 2020.

[5] "Monte Carlo Tree search," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search. [Accessed 2022].

[6] "Minimax," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Minimax. [Accessed 2022].

[7] "Negamax," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Negamax. [Accessed 2022].

[8] "Alpha-beta pruning," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning. [Accessed 2022].

[9] "Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning)," geeksforgeeks, [Online]. Available: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/.

[10] "ML | Monte Carlo Tree Search (MCTS)," geeksforgeeks, [Online]. Available: https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/.

[11] S. B, "Analysis of Minimax Alogrithm using Tic-Tac-Toe," The authors and IOS Press, 2020, pp. 528-532.

[12] "Minimax Algorithm in Game Theory | Set 1 (Introduction)," geeksforgeeks, [Online]. Available: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/.

[13] D. AVELLAN-HULTMAN and E. G. QUERAT, "A comparison of two tree-search based algorithms for playing 3-dimensional Connect Four," DEGREE PROJECT TECHNOLOGY, STOCKHOLM, SWEDEN, 2021.

APPENDIX A: Project contribution

Team's member	Task description
Nguyễn Mạnh Cường	Implement negamax with alpha-beta pruning and design Monte-Carlo Tree Searching theoretical basis. Write chapter 3: Analysis (report)
Phan Công Đại	Design game pattern and game theory basis. Write chapter 1: Introduction and chapter 4: Conclusion (report)
Bùi Phương Nam	Design negamax with alpha-beta theoretical basis. Test and analyse algorithms. Write chapter 3: Analysis (report)
Nguyễn Trọng Huy	Design UI. Design Minimax with alpha-beta pruning theorectical basis and implementation. Implement random-move and Monte-Carlo Tree Searching. Write chapter 2: Algorithm (report)

APPENDIX B: How to run our program

a. Set up the environments:

Our program was written in Python 3

To execute our program, you, at first, install Python compiler to your computer.

Then, you need to install all extended libraries we utilise in our program. All of that was packaged in `requirement.txt` file. You must excecute the following command in terminal:

```
pip –install requirement.txt
```

Please note that the installation process might be varied due to different operating systems and the version of pip also.

After successfully running all the requirements, you navigate to the destination folder concluding the source code. Execute our program by following command:

```
python main.py
```

Or

```
python3 main.py
```

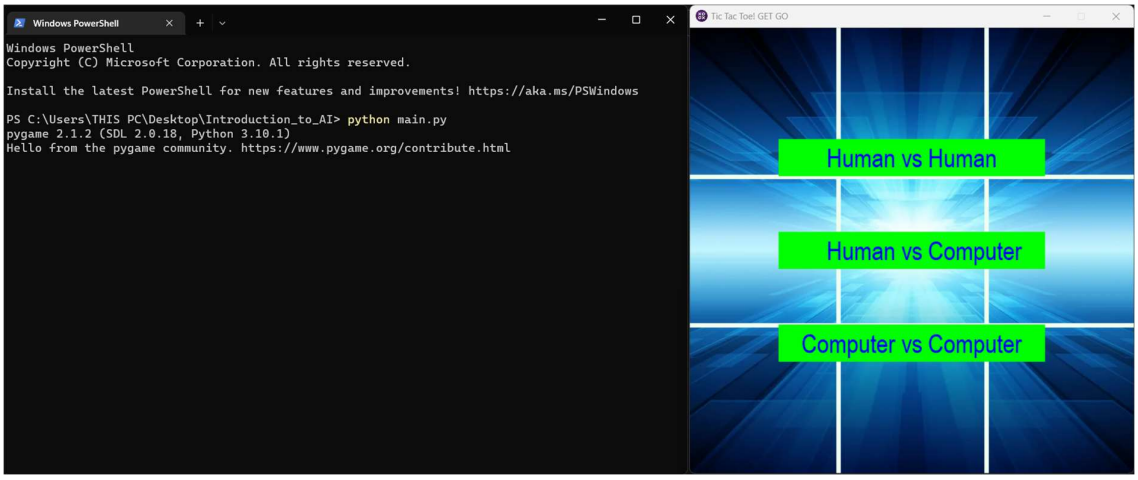

Figure 15: Our program starts

b. Run the program:

Then, you have three modules to select to start the game:

1. Human versus human:

You can play with the other human player. The first player is always defined as ‘X’ player or ‘Player 1’, and the other one is ‘O’ or ‘player 2’.

2. Human versus Computer:

After you select this module, you may need to define you are player 1 and the computer is player 2 or vice versa.

Then you must select one computer algorithm to battle with you. You have 4 choices:

- Random move
- Minimax with alpha – beta pruning
- Negamax with alpha – beta pruning
- Monte-carlo Tree (10000 NoOfSimulations)

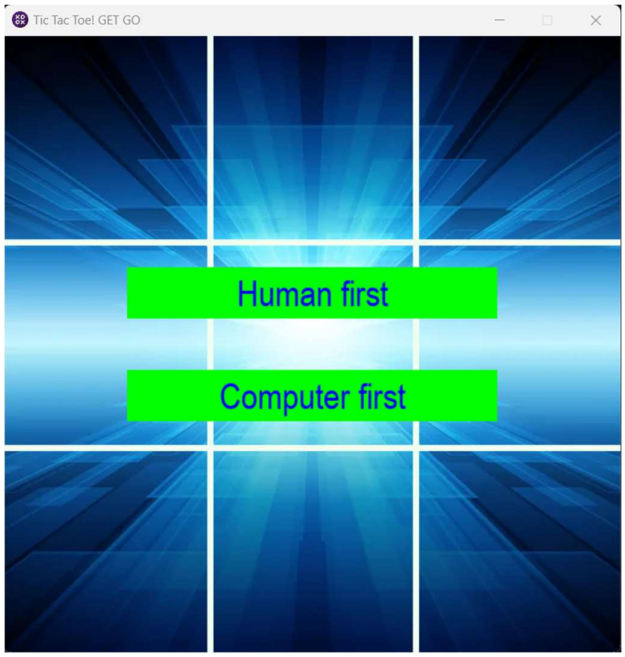

Figure 16: You need to select your turn

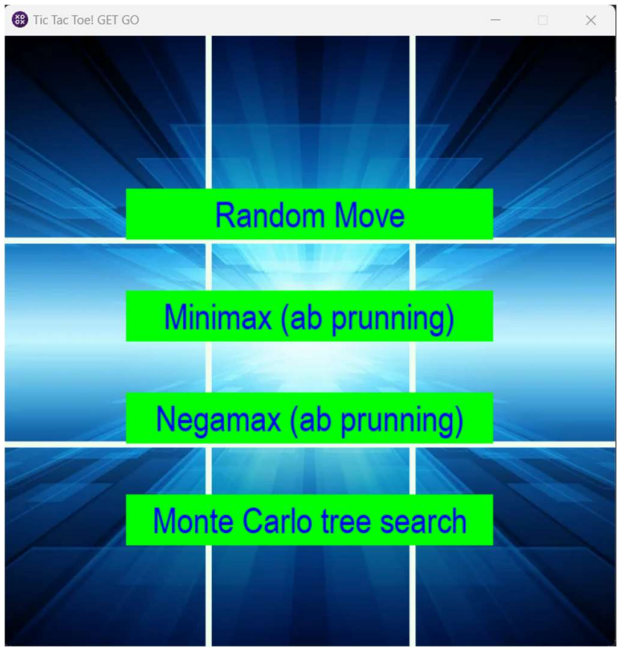

Figure 17:Then you select the computer's algorithm

But please note that the Minimax (with ab pruning) and Negamax (with ab pruning) are unbeatable.

3. Computer versus computer:

When selecting this module, you might have a chance to see that battle between two computer players with two distinct algorithms. In this module, you have to select the algorithm for the first and the second players consecutively. Depending on the time complexity of each algorithm, then you will see the result of the battle in a certain time.

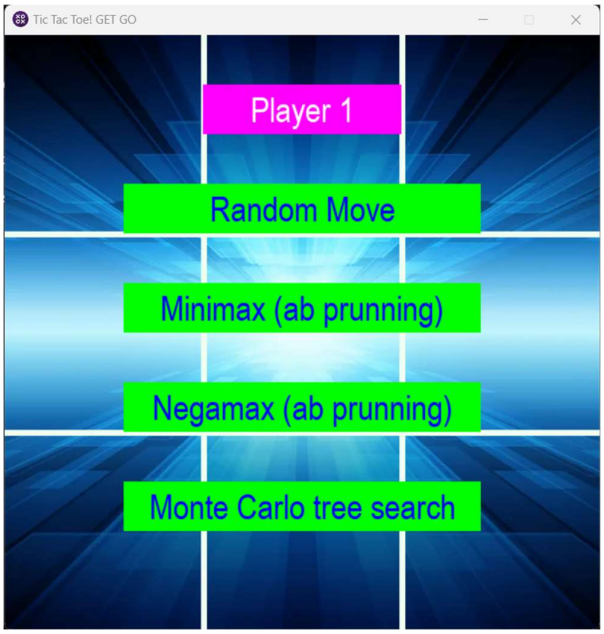

Figure 18: You have to select the algorithms for two players

c. Game's result:

After you finish a game, the result is displayed on the application's screen. We offer you the selection to restart the game.

Figure 19: The result's window

Or if you want to quit the game, you can click on the ESC button on the top right of the screen.