2023 IEEE Symposium Series on Computational Intelligence (SSCI)
Mexico City, Mexico. December 5-8, 2023

Optimizing Strategy Games: Ant Colony Optimization vs. Minimax Algorithm

Yoqsan Angeles-García
Computational Cognitive Sciences Laboratory-CIC
Instituto Politécnico Nacional
Mexico City, Mexico
http://orcid.org/0009-0004-0886-5540
yangelesg2020@cic.ipn.mx

Valeria Karina Legaria-Santiago
Centro de Investigación en Computación (CIC)
Instituto Politécnico Nacional
Mexico City, Mexico
http://orcid.org/0009-0008-4843-4971

Álvaro Azueto-Ríos
U. Prof. Interdisciplinaria en Ingeniería y Tecnologías Avanzadas
UPIITA, Instituto Politécnico Nacional
Mexico City, Mexico
https://orcid.org/0000-0003-1627-0323

Hiram Calvo
Computational Cognitive Sciences Laboratory-CIC
Instituto Politécnico Nacional
Mexico City, Mexico
https://orcid.org/0000-0003-2836-2102
hcalvo@cic.ipn.mx

***Abstract*—This article proposes an Ant Colony Optimization (ACO) algorithm, an optimization method to find paths in graphs, adapted to solve strategic games. The games of study are Tic-Tac-Toe (also known as noughts and crosses, three in a row, or Xs and Os), and Chess. The algorithms' performance is contrasted by contending ACO against the Minimax algorithm, in different setups of Tic-Tac-Toe and Chess. The performance is explained in terms of average time response, correctness of the move choice, and memory used when executing the function. Results reveal a slightly better average performance by the ACO algorithm compared to Minimax. These findings highlight the ability of ACO in decision-making algorithms without requiring knowledge of previous games. Furthermore, the results suggest that the ACO-based path optimization approach can be an effective alternative to improve the efficiency of decisions made by intelligent systems in environments that require rapid response.**

***Keywords*—Strategic games, Ant Colony Optimization, Intelligent Agents.**

I. Introduction

One of the characteristics that define intelligence is the ability to adapt to various environments. [1]. It implies the capacity to solve problems, reason logically, and make effective decisions [2]. Moreover, artificial intelligence (AI) refers to the system's property of adapting its behavior in order to achieve a goal by analyzing how its previous actions affect the environment [3]. Decision-making algorithms have the aim to learn and improve, in terms of objectivity, the decision-making of a human. The accuracy of the intelligent systems' decisions can be tested through their performance in strategic games.

In [4] the authors define strategic play as the understanding and evaluation of the available options and their application within the context of the game. In this way, there are algorithms based on neural networks or in reinforcement learning to play strategy games, yielding favorable outcomes. However, these techniques require knowledge from previous games from expert systems (such as humans or other AI systems) or compete against themselves until specializing in a specific game [5]–[7]. On the other hand, there are algorithms for decision-making, that do not require previous knowledge, such as Minimax (widely employed to teach AI agents how to play turn-based strategy games) and the Ant Colony Optimization Algorithm (ACO).

The ideal in a decision-making algorithm is to find the optimal value, however, as the eligible options increase, the search space increases exponentially along with the complexity, increasing the resources needed to find the optimal value. When there are limited resources (such as memory, storage, response time, etc.), it is advantageous to have alternative options that provide solutions, even if they are suboptimal, as long as they have a response within the requirements of the context. Examples of this type of scenario are time trial games, online games, or real-time strategy (RTS) games [8], where a quick response is needed; In the case of physically constructed AI agents, such as exploration robots, the hardware could determine the limitations of the algorithms that can be implemented.

The Minimax algorithm is complete, which means it will certainly find a solution (if one exists) in the finite search tree. Alternatively, ACO uses ant agents to traverse a search tree, incrementally constructing solutions guided by pheromone levels and problem-specific information to find optimal or suboptimal solutions in complex combinatorial optimization problems. In this paper, the initial conditions for both algorithms are the only knowledge of the state of the board, the rules of the game, and a board evaluation function. The parameters to compare are the search depth and the execution time.

This paper aims to explore the performance of ACO and Minimax algorithms as decision-making tools to play strategy games without relying on prior game knowledge, contrasting

the resources used to assess their advantages and disadvantages.

The organization of this paper is as follows: Section 2 presents the related work. Section 3 explains the basis of the Minimax algorithm. Section 4 describes the proposed method to adapt the ACO algorithm for gaming. Section 4 shows the evaluation function to guide the decision-making process. Section 5 shows the experiments and their results. Section 6 offers a brief discussion about the results. Section 7 gives the conclusions.

II. RELATED WORK

This section shows some articles as related work that discusses different algorithmic approaches for developing strategic game-playing systems, with a focus on the game of Tic-Tac-Toe. The mentioned methods utilize various techniques such as ensemble-based boosting, rule-based inference, T-AlphaBeta search algorithm, artificial neural networks, genetic algorithms, and ant colony optimization.

In [6] the authors propose an algorithmic solution by combining an ensemble-based boosting approach and rule-based inference to build a probabilistic expert system that strategically chooses the best optimal move for the next possible state of the game. Nevertheless, they train their systems with 255,168 unique game states of Tic-Tac-Toe.

In [8] the authors propose a T-AlphaBeta search algorithm for RTS games and return better results at the same time of fast search. The algorithm has a balance between search depth and search time. The game for the test they propose was SpajCraft.

In [9] the author proposes a method to evolve strategies for the Tic-Tac-Toe game using artificial neural networks and genetic algorithms. The genetic algorithm generates a population of solutions that are evaluated with the neural network to calculate how strong the selected movement is. Experiments showed that the method was able to evolve tic-tac-toe players who could consistently win or draw against a human opponent.

In [10] the author proposes evolutionary programming as a technique to train neural networks for playing Tic-Tac-Toe effectively. It uses an initial population of 50 neural networks with random weights and connections and evaluates them based on their performance against human opponents or against other neural networks in a competitive process. New populations of neural networks are generated that show improvements. The evolutionary process continues until a neural network is obtained that can play effectively and win consistently against human opponents.

In [11] a method is proposed to evolve game strategies using ant colony optimization (ACO) and neural networks. The neural network is used to assess the fitness of each solution in the ACO search. The fitness of a solution is determined by how well it performs against a set of human opponents. The solutions with the highest fitness are used to create the next generation of solutions. This process is repeated until a solution is found that can consistently win against a human opponent.

III. MINIMAX ALGORITHM

The Minimax algorithm is a method used in many areas such as game theory and decision-making to determine the best strategy in zero-sum game situations [12] [13]. Its main objective is to minimize the maximum possible loss or maximize the minimum possible gain in a game between two players. Its space complexity is the same as the depth-first search algorithm $O(d)$. However, its time complexity is $O(b^d)$, (where b is the branching factor and d is the depth of the tree), which means that for complex search spaces, its execution time will be exponential making the response time really long, and in some cases with no possibility of terminating.

The Minimax algorithm operates on the principle that players will always choose their most beneficial move, maximizing their gain while minimizing the opponent's. By assessing the complete game tree, it determines the best move considering potential responses at every level. When the entire tree is mapped, the algorithm identifies the optimal move, ensuring a win or draw for the user. In extensive games, exploration is typically limited to a specified depth, not always reaching the end [14].

As an example of application, in [15], Baxter describes a chess program called KnightCap that uses a combination of temporal difference learning (TDL) and Minimax search to improve its play.

IV. ANT COLONY OPTIMIZATION ALGORITHM FOR PLAYING BOARD GAMES

This section presents the proposal of this paper, starting with the description of how to translate a game board to a graph. Then the evaluation function to guide the decision-making process and the pheromone actualization is explained. Finally, the pseudocode of the algorithm is shown.

Its time complexity is $O(b*d*a*e)$ where a is the number of ants and e is the number of epochs.

A. How to See the Game as a Graph

In the case of board games, each state of the board can be seen as a node, and the weight of each edge can be determined by calculating the value of the board relative to the player. Fig. 1 shows a graph where each node represents a board state. Each edge connects the current board state with a potential future board state if the move is made. Each edge has the board's value after the potential move. The weight of each edge is determined by the evaluation function that calculates the value of the potential future board state when the move is applied.

The value of each edge can be positive, negative, or zero. The value will be negative if the board benefits the opponent more than the player, positive if the board provides winning possibilities for the player, and zero in the case of a draw.

Each node has four properties: move, weight, τ (pheromone level), and children. A move gives the coordinates $\{$x,y$\}$ of the

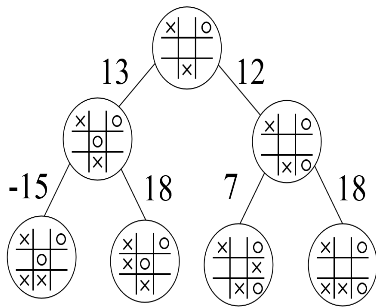

Fig. 1. Graph of the Tic-Tac-Toe game

board to put the player's symbol. Children are the descendant nodes. However, the descendants are not necessarily all the possible moves because each ant can only see a randomly selected set of n moves from all possible moves. Each new move is added to the node's children. Therefore, the first ant chooses among n moves it can see, but if the second ant sees another n moves, all different from those of the first ant, the second ant will choose among $2n$ possible moves. This allows for greater exploration without significantly increasing computational cost.

As each ant advances, it generates descendants from the current node, expanding the graph only in the nodes it visits. This presents a disadvantage compared to Minimax and other algorithms due to its inability to consider all options, therefore, it does not guarantee to find the optimal move every time. However, it has an advantage in terms of computational cost. This allows it to explore deeper in the same amount of time, enabling it to find moves further into the future and thereby outperform other algorithms with more limited future move vision.

In the beginning, the value of τ is the same for all edges, so the random selection is based solely on the probabilities according to the weights of each edge. The ants will traverse the graph sequentially, not in parallel. An epoch is when all the ants finish their path. The maximum depth reached by each ant is specified, and when all ants have finished their path, the process of updating τ begins. As the graph is expanding, for negative board's values when the opponent is playing, we take the value as positive, this is assuming the opponent is playing the best move it can, so when we take the probabilities of each move, the most negative value will be chosen by the opponent. The sum of the path considers negative values as negative so the final path's value is negative when the play is bad for the player.

The total value of the path is calculated by summing the weight of all edges that each ant passed through. If the total value is negative, the pheromones of each edge that the ant traversed are erased, meaning $\tau = 0$, to prevent the ants from passing through there again in the future. If the value is positive, the corresponding τ is updated.

After a relatively low number of epochs (between 10 and 20), the pheromone values are updated and differentiated, with the best moves having much higher pheromone values compared to other moves. To choose the move on the board, we utilize the information provided by the algorithm in the following manner:

- Take the first move chosen by each ant in the last epoch.
- Calculate the total value of the path for each of these ants and sum the path values that started with the same move.
- The move with the highest sum of path values is chosen.

For example, for an algorithm with 5 ants:

- In the last epoch, the ants chose the following initial moves: {3,2}, {2,1}, {3,2}, {1,3}, {1,3}.
- They obtained the path values: {19, 22, 14, 20, 23}.
- The paths that started with the move {3,2} had a total of 33. The paths that started with the move {2,1} had a total of 22, and the paths that started with the move {1,3} had a total of 43. Therefore, for this example, the move {1,3} is chosen.

B. Evaluation Function for Tic Tac Toe

For the evaluation function, three main aspects were considered: the number of rows, columns, or diagonals (any line) where it is still possible to have a line of the same symbol to win, the number of lines that are only one cell away from winning, and lastly, whether the board has already been won or lost. The scores for each aspect were defined empirically.

For each line that can be used to win, one point is awarded. For each line that is only one cell away from the player winning, 50 points are awarded. If the opponent will be the winner, 100 points are subtracted. Lastly, for each line where the opponent has all the cells in the row occupied by the player and one cell occupied by the opponent, 40 points are added. This was done to prioritize a tie over a risky move with a low probability of winning.

Finally, if the board shows the player as the winner, the value will be 100/depth, where depth is the level at which the graph reached the move. Dividing by depth assigns higher priority to moves that result in a faster win. If the board shows the opponent as the winner, the value will be -100/depth. In the case of a tie, the value will be 0. Fig. 2 shows some examples of boards and their respective values.

C. Evaluation Function for Chess

In this section we will only explain how to evaluate the board for Chess to get the values the algorithm needs to choose a play, the rest of the algorithm is the same for both games.

The function assigns different values to each piece: pawn = 1, knight = 3, bishop = 3, rook = 15, queen = 25, king = 0. It accumulates the value and number of pieces for each color. It also adds the number of opponent's pieces attacked and subtracts the number of pieces under attack by the player. It evaluates control of the center of the board and adds a value of one if the pieces are on central squares. It checks for check, and if the piece giving check is not in danger, 500 points are added. In the case of checkmate, 2000 points are added. Depending on the turn, the score is adjusted to reflect control of the game, assigning a positive value if the current player's

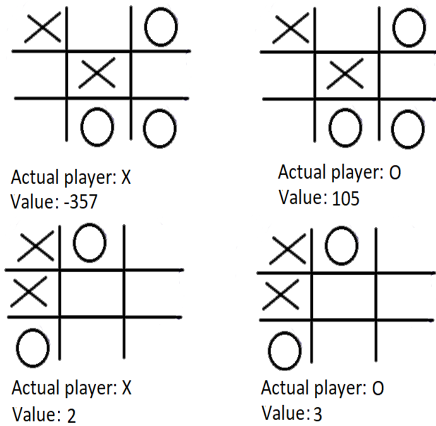

Fig. 2. Evaluation function with different board states

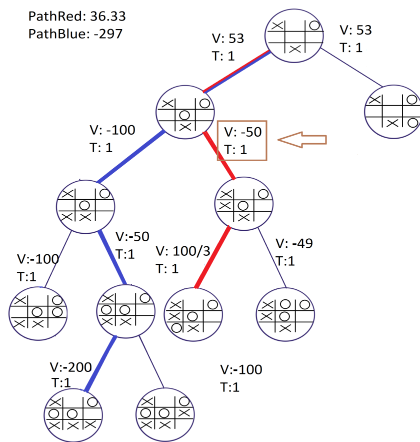

Fig. 3. Graph before pheromone actualization. Letter V represents the board's value, T represents the pheromone's value

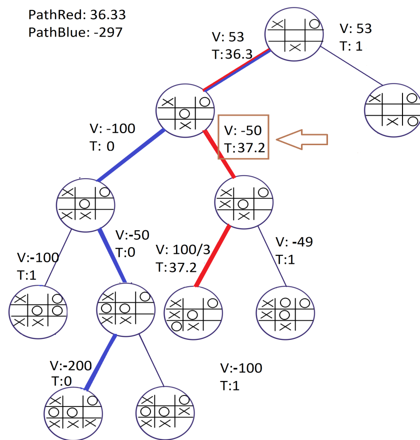

Fig. 4. Graph after pheromone actualization. Letter V represents the board's value, T represents the pheromone's value

pieces have the advantage over the opponent and a negative value otherwise.

D. Pheromone Actualization

First, it is important to note that two variables are used: τ (the current pheromone value of the edge) and deltaTau, an auxiliary value that stores the amount to be added to the current τ, initialized to 0.

The process of updating the pheromones on each edge is as follows:

- The edges traversed by each ant are examined.
- Check the value of the ant's path used on this edge.
- If it is greater than zero, deltaTau = Q * (ant's path value) + child.deltaTau.
- If $deltaTau \leq 0$, $deltaTau = 0$
- Finally, each node is revisited and updated using the formula: $child.tau = (1-rho)*child.tau+child.deltaTau$

Fig. 3 shows an example of two ants, one red and one blue, playing with the 'O', the root node is the actual board. After they complete their path, the red ant chose a path that finished with a win for the player, while the blue ant finished with a win for the opponent. Fig. 4 shows pheromone actualization, the board's values are the same but τ value has changed.

Algorithm 1 Main Game Loop

```
1: d1 ← depthACO
2: d2 ← depthMinimax
3: E ← epochs
4: board ← gen_tab(n)
5: #check if the game finished
6: Gameover ← check_winner(board)
7: while not Gameover do
8:     Print('ACO Plays')
9:     move ← ACO(board, d1, n, Q, rho, ants, E, True)
10:    board ← apply_move(board, move, True)
11:    Gameover ← check_winner(board)
12:    if not Gameover then
13:        Print('MiniMax Plays')
14:        move ← compMove(board, False, d2)
15:        board ← apply_move(board, move, False)
16:        Gameover ← check_winner(board)
17:    end if
18: end while
```

V. Experiments and Results

The pseudocode for ACO to play against Minimax, both in Tic-Tac-Toe and in Chess, is shown in Algorithm 1 box. The first scenario used three different game modes of Tic-tac-toe:

- Tic-Tac-Toe 3x3: The game is played on a square board consisting of 9 cells arranged in 3 rows and 3 columns. Two players participate in the game, with one using the symbol "X" and the other using the symbol "O."
- Tic-Tac-Toe 5x5: Same as Tic-Tac-Toc 3x3 but with a square board of 25 cells arranged in 5 rows and 5 columns.

Algorithm 2 ACO

1: **function** ACO($board, d, n, Q, \rho, ants, epochs, player$)
2: **for** $epoch \leftarrow 1$ to $epochs$ **do**
3: $CostPath \leftarrow []$
4: **for** $ant \leftarrow 1$ to $ants$ **do**
5: $AccumWeight \leftarrow []$
6: **for** $current_depth \leftarrow 1$ to d **do**
7: $Gameover \leftarrow$ CHECK_WINNER($board$)
8: **if** $\neg Gameover$ **then**
9: # EXPAND function gives possible moves
10: $moves, weights \leftarrow$ EXPAND($board$)
11: **if** $moves \notin node.children$ **then**
12: ADD_CHILD($node, child$)
13: **end if**
14: $visibility \leftarrow [child.weight \times child.tau$ for $child$ in $descendants]$
15: $visibility \leftarrow [-value$ if $value < 0$ else $value$ for $value$ in $visibility]$
16: **end if**
17: $Probabilities \leftarrow [value/\sum(visibility)$ for $value$ in $visibility]$
18: # ROULETTE makes a selection by roulette
19: $index \leftarrow$ ROULETTE($Probabilities$)
20: $node \leftarrow descendants[index]$
21: $AccumWeight$.append($node.weight$)
22: $CostPath$.append($\sum(AccumWeight)$)
23: **end for**
24: UPDATE_TAU()
25: **end for**
26: **end for**
27: $moves \leftarrow$ *Path[0] for each ant*
28: $sums \leftarrow []$
29: **for** $move$ **in** $moves$ **do**
30: $sum_value \leftarrow 0$
31: **for** $cost, move$ **in** COSTPATH() **do**
32: **if** $move =$ list($move$) **then**
33: sum_value += $cost$
34: **end if**
35: **end for**
36: $sums$.append(sum_value)
37: **end for**
38: $max_sum \leftarrow$ max($sums$)
39: $index_max_sum \leftarrow$ index of max_sum in $sums$
40: $movement \leftarrow$ list($moves[index_max_sum]$)
41: **return** $movement$
42: **end function**

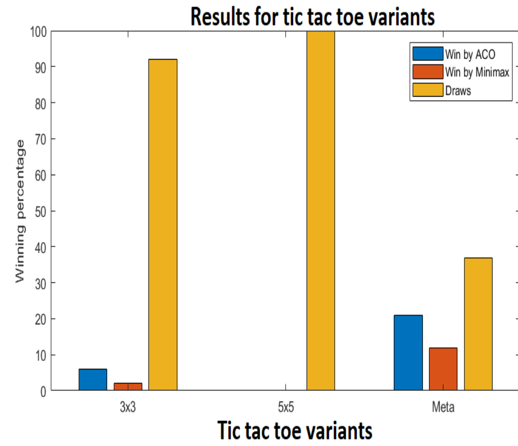

Fig. 5. Results (in percentage) of playing 500 times the configuration of Tic-Tac-Toe 3x3

- Meta Tic-Tac-Toe 3x3: This is a game that extends the 3x3 game, where each of the nine cells is, in turn, a 3x3 board. The way to select a cell on the larger board is by winning in the corresponding sub-board of that cell.

For each of the three Tic-Tac-Toe variants, 500 games were played, pitting the proposed ACO algorithm against Minimax with Alpha-Beta pruning. Experimental results indicate that ACO outperforms Minimax in terms of speed, despite both algorithms being allocated approximately one second of computation time. Minimax achieves a depth of three, whereas ACO explores a depth of seven. The ACO algorithm's parameters are detailed below:

- depthACO = 7 depth for ACO algorithm
- depthMinimax = 3 depth for Minimax algorithm
- n = 20 number of moves chosen randomly
- Q = 1 learning ratio
- rho = 0.1 Pheromone Decay Rate
- ants = 5
- epochs = 15

The algorithm's parameters were empirically selected after experimentation with various parameter sets. The results for the three Tic Tac Toe's variants are depicted in Fig 5.

- Win by ACO: 6% of the games Win by Minimax: 2% of the games Draw: 92% of the games
- Win by ACO: 0% of the games Win by Minimax: 0% of the games Draw: 100% of the games
- Win by ACO: 21% of the games Win by Minimax: 12% of the gamesDraw: 67% of the games

In the case of Chess, two distinct experiments were conducted:ACO playing against Minimax with a depth of five for the ACO algorithm, while a depth of three was used for Minimax. The other experiment used the Kaufman Test with a depth of five for ACO and three for Minimax. Subsequently, both algorithms were evaluated at a depth of three.

The Kaufman Test, originally introduced by Larry Kaufman, comprises a set of 25 chosen positions extracted from real

chess games. Participants are presented with these positions and are tasked with providing a move for each position [16].

In the context of the Kaufman test, when the test conditions impose an identical time constraint on both algorithms, the observed outcome indicates that ACO correctly identifies the optimal move for four boards, whereas Minimax attains the correct move for two boards. Conversely, when the constraint is placed on the depth of search, ACO identifies the correct move for three boards, while Minimax achieves correctness for two boards. The average time for ACO to give the move of the 25 boards is 0.757s, the average time for Minimax is 1.6s. The average memory used for ACO to give the move of the 25 boards is 115.365Mb, the average memory for Minimax is 116.08Mb.

Both algorithms were also pitted against each other as was done in the case of Tic Tac Toe; however, in these games, Minimax exhibited a significantly higher win rate, winning 80% of the time, while 20% resulted in draws.

VI. Discussion

The Ant Colony Optimization algorithm performs slightly better than the Minimax algorithm in two out of three Tic Tac Toe variants. However, in the 5x5 Tic Tac Toe variant, all matches resulted in draws. This could be due to the increased complexity of achieving victory in the 5x5 variant, where strategic planning is more intricate due to the opponent's enhanced ability to block potential paths to victory. In contrast to the 3x3 format, where opportunities for victory prevail, the 5x5 board limits offensive and defensive strategies, resulting in a higher likelihood of draws as players prioritize blocking victory paths over gaining a definitive advantage.

In the context of chess, a game with a significantly larger search space, it becomes evident that ACO exhibits improved performance during the Kauffman test but performs less optimally when competing against Minimax. This discrepancy may arise from an evaluation function that inadequately assigns a sufficiently high value when victory is attainable. Given the extensive search space inherent to chess, the probability of selecting the optimal value diminishes, contributing to this observed difference in performance.

VII. Conclusions

The ACO approach demonstrates a greater ability to explore different options and find suboptimal moves in a limited time.

However, it is important to note that these conclusions may vary depending on the specific game configurations. While adding variants may complicate the game, further research and testing with other strategy games are recommended to validate these results in different contexts.

While this article explored various strategy games, it is feasible to implement the algorithm in diverse decision-making problems. As shown in Algorithm 2, the code is not specific to the game being played, with the only difference being in the EXPAND function. By adapting the evaluation function to the specific context, it can be implemented without the need for prior information. Examples of such problems include risk identification and management in investment, route and logistics planning, trajectory planning for robotics, and vehicle routing. While these problems can be solved using Minimax, having an alternative algorithm can be advantageous when resources are limited.

Acknowledgments

The authors wish to thank the support of the Instituto Politécnico Nacional (COFAA, SIP-IPN) and the Mexican Government (CONAHCyT, SNI).

References

[1] R. J. Sternberg, "A theory of adaptive intelligence and its relation to general intelligence," J. Intell., vol. 7, no. 4, pp. 1–17, 2019, doi: 10.3390/jintelligence7040023.
[2] R. J. Sternberg, "The Theory of Successful Intelligence," Gifted Education International, vol. 15, no. 1. pp. 4–21, 2000, doi: 10.1177/026142940001500103.
[3] High-Level Expert Group on Artificial Intelligence, "A definition of AI: Main capabilities and scientific disciplines," 2018. European Commission. [Online]. Available: https://ec.europa.eu/futurium/en/system/files/ged/ai_hleg_definition_of_ai_18_december_1.pdf
[4] S. Dor, "Game Studies - Strategy in Games or Strategy Games: Dictionary and Encyclopaedic Definitions for Game Studies," Int. J. Comput. game Res., vol. 18, no. 1, 2018, Accessed: Jun. 19, 2023. [Online]. Available: https://gamestudies.org/1801/articles/simon_dor.
[5] D. Silver et al., "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," Science (80-.)., vol. 362, no. 6419, pp. 1140–1144, Dec. 2018, doi:10.1126/SCIENCE.AAR6404/SUPPL_FILE/AAR6404_DATAS1.ZIP.
[6] M. S. K. Inan, R. Hasan, and T. T. Prama, "An Integrated Expert System with a Supervised Machine Learning based Probabilistic Approach to Play Tic-Tac-Toe," 2021 IEEE 12th Annu. Ubiquitous Comput. Electron. Mob. Commun. Conf. UEMCON 2021, pp. 116–120, 2021, doi: 10.1109/UEMCON53757.2021.9666728.
[7] D. Meng, B. Jianbo, Q. Yizhong, F. Yao, and L. Shuqin, "Design of Amazon Chess Game System Based on Reinforcement Learning," Proc. 31st Chinese Control Decis. Conf. CCDC 2019, pp. 6337–6342, Jun. 2019, doi: 10.1109/CCDC.2019.8832999.
[8] S. Chen, H. Yan, and L. Wang, "Adaptation of Alpha-Beta Search Algorithm for Real-Time Strategy Game," Proc. IEEE Int. Conf. Softw. Eng. Serv. Sci. ICSESS, vol. 2018-Novem, pp. 995–998, 2019, doi: 10.1109/ICSESS.2018.8663793.
[9] B. Al-khateeb, "An evolutionary tic-tac-toe player," vol. 4, no. 4, pp. 182–185, 2012.
[10] D. B. Fogel, "Using evolutionary programming to create neural networks that are capable of playing tic-tac-toe," IEEE Int. Conf. Neural Networks - Conf. Proc., vol. 1993-Janua, pp. 875–880, 1993, doi: 10.1109/ICNN.1993.298673.
[11] X. Qi et al., "Collective intelligence evolution using ant colony optimization and neural networks," Neural Comput. Appl., vol. 33, no. 19, pp. 12721–12735, Oct. 2021, doi: 10.1007/S00521-021-05918-7/FIGURES/8.
[12] B. Bošanský, V. Lisý, M. Lanctot, J. Čermák, and M. H. M. Winands, "Algorithms for computing strategies in two-player simultaneous move games," Artif. Intell., vol. 237, pp. 1–40, Aug. 2016, doi: 10.1016/J.ARTINT.2016.03.005.
[13] Y. Hu and G. Zuo, "Expectation Minimax Algorithm for the Two-Player Military Chess Game," Proc. 31st Chinese Control Decis. Conf. CCDC 2019, pp. 6357–6362, Jun. 2019, doi: 10.1109/CCDC.2019.8833085.
[14] Strong, G. (2011). The minimax algorithm. Trinity College Dublin.
[15] J. Baxter, A. Tridgell, and L. Weaver, "KnightCap: A chess program that learns by combining TD(lambda) with game-tree search," Jan. 1999, Accessed: Jun. 22, 2023. [Online]. Available: https://arxiv.org/abs/cs/9901002v1.
[16] (1992). Computer chess reports. Computer Chess, 19(1):1–2