Alex Latz

Dr. Olid and Mr. Griest

Honors Independent Study

6 August 2021

Artificial Intelligence in Adversarial Environments

As long as computers have existed, many have dreamed of using them to simulate intelligence. However, the programmatic nature of computers makes the creation of a general decision-making AI nearly impossible with current technology. Computers struggle in environments with hard-to-understand choices and limited knowledge, making AI development very narrow in its focus. Because of this limitation, one of the simplest applications for artificial intelligence is in board games like chess, where there are a set number of legal moves and well-defined rules for computers to understand. Creating and refining artificial intelligence for these simple environments can lead to large improvements for other, more practical artificial intelligence programs (Purves). Alan Turing, a British mathematician and early computer scientist considered by many to be the “father of artificial intelligence,” realized this limitation while working with David Champernowe to create Turochamp, one of the first AI programs in 1948. Turochamp was a complex algorithm that contained a formula for evaluating a chessboard to determine which side had an advantage. It evaluated each possible move in turn and decided on the optimal move based on this formula (Kasparov and Friedel). Although it only looked one move into the future, the program was much too complex to run on computers of its time. To remedy this, Turing and Champernowe copied their algorithm onto a notebook and played against each other on a physical chessboard, evaluating each move by hand according to the formula. They quickly realized the potential benefits of optimizing this algorithm, as it required vast amounts of repetitive actions that wasted time and caused it to not be feasible to run on a computer (Newell et al.).

Artificial intelligence for games, while seemingly pointless, serves as a textbook example of decision-making AI. The limitations of games help create a closed environment to perfect AI approaches and tweak decision-making based on easily predictable future outcomes (Purves). Without the advancements in AI programming first discovered through creating simple game AI, complex approaches like Monte Carlo Tree Search and Deep Learning would not be possible.

Additionally, while the simplicity of these environments suggest that perfect and efficient artificial intelligence is simple to solve, new approaches are still being developed that allow for less computation time or more efficient strategies to this day. For example, until 2008, all traditional decision making AI used an evaluation function to determine the merit of each decision. Monte Carlo Tree Search instead randomly simulates a full game at a time, assigning preference values to moves based on the percentage of wins in fully-played games that result from the move (Chaslot et al.). Without continued advances in so-called “traditional” AI algorithms tested in seemingly unproductive environments, all decision-making AI would suffer.

To manage the many possibilities in a typical game environment, traditional AIs use techniques from game theory and applied mathematics. Each environment has a finite set of possible end states, which determine the final outcome of the game. These are typically represented by three numbers: 1 for a computer win, 0 for a draw, and -1 for a computer loss (Michie). This makes it simple to indicate that a computer should strive for non-negative outcomes, with higher values being more favorable. However, if an AI considers the outcomes of possible moves and their subsequent moves until it reaches a preferred end state, it must backtrack to the first imagined move to continue down the path. This is represented in a tree, a data structure in computer science. A tree consists of a series of nodes, which contain some kind of data and a set of links to other nodes. Each link points in only one direction, which allows a computer to sort each node by how many links it must follow from the root, or starting, node to reach the current node (Sedgewick and Kevin Daniel Wayne). This data structure works well for considering branching possibilities, and when used to simulate a game it is called a game tree. To create a game tree, each node must contain some representation of the current state of the game, like a list of pieces and their positions on a chess board. Each of its links points towards a child node, which contains the representation with one possible move made from the parent node. This allows the computer to survey each possible child move, and the child moves of that node, and so on until it reaches an end to the game.

Creating an AI to look through a game tree to pick the quickest path to a win presents a problem: it does not know which paths lead to wins, meaning it has to manually work through every possible path to find an answer. As early computer scientist Claude Shannon quickly discovered when building out his attempt at a chess AI, on average the game of chess has a branching factor of 35. This means that on average, each position on the chess board has 35 possible moves that could be made (Newell et al.). As each node of the game tree has on average 35 children, the time taken to explore each level of the tree (first moves, second moves, and so on) expands by a power of 35 for each level. Chess games take on average 40 moves to reach an end, so the average chess game tree would consist of 59 novemdecillion (or 5.961) nodes. As Newell et al. point out in their 1958 paper *Chess-Playing Programs and the Problem of Complexity*, “Now [scanning the tree] might have been the "wheel" of chess - with the adventure ended almost before it had started - if the tree were not so large that even current computers can discover only the minutest fraction of it in years of computing” (Newell et al.). To alleviate this, an evaluation function is necessary. An evaluation function acts as an estimate of the win probability at that point in the tree. This means a node with a sufficiently bad score from the evaluation function can be skipped, saving millions of potential nodes in the future from needing to be evaluated (Shannon). Evaluation functions require detailed knowledge of the environment, and are usually the most fallible component of a traditional AI as it is impossible to perfectly approximate the win probability of each node. While the use of an evaluation function is almost universal in traditional decision-making AI, deciding when to use the evaluation function is very different for each prominent method.

Many simple testbeds for AI development are so-called “zero-sum” environments. These environments consist of 2 or more agents (players) competing for resources. However, the defining characteristic of a zero-sum environment is that each agent’s best interests are completely perpendicular to the others’. For example, blocking a possible four-in-a-row in Connect 4 is directly beneficial to one player, preventing a loss and adding more possible pieces for them to work with on the board, and detrimental to the other: preventing a win. Alan Turing, in creating his chess AI, used this characteristic to his advantage by utilizing minimaxing, a procedure where the AI attempts to make the best possible move to minimize the opponent’s ability to make the best possible move. However, while attempting to reproduce Turing’s AI, Kasparov and Friedel found that Turing deviated from his algorithm at times. Instead of calculating paths that only led to worse moves, Turing skipped them entirely (Kasparov and Friedel). This approximates a later improvement to minimaxing called alpha-beta pruning. Alpha-beta pruning scans each set of children to determine whether a better move can be found, and skips all nodes that lead to worse moves than the known best move (Fuller et al.). This results in an average 25% decrease in nodes evaluated per level in chess games, which greatly decreases the amount of time needed to evaluate a move (Marckel). Alpha-beta improvements to minimax searching are widely regarded as a reliable and traditional method to solve a game tree.

However, the dominance of minimax search with alpha-beta pruning came to a close in 2008 with the discovery of Monte Carlo Tree Search (MCTS). MCTS forgoes the standard evaluation function to avoid searching nodes, instead relying on stochastic simulation. Stochastic simulation refers to random simulation of nodes, meaning that MCTS randomly explores possibilities branching from nodes and estimates the probability of a win from the explored outcomes for each node. This probability is used instead of an evaluation function to decide which moves should be played. MCTS uses a four-stage method to search through possibilities: selection, expansion, simulation, and backpropagation (Chaslot et al.). The first stage, selection, determines whether to continue looking at other possibilities or to make a move. It does this with an algorithm called Upper Confidence Bounds Applied to Trees (UCT), which decides between exploration of further nodes or exploitation (choosing the node with the highest expected payoff). UCT treats selection as a multi-armed bandit problem, a model for allocating limited resources between different choices in probability theory, deciding based on a confidence interval. As MCTS functions (with less accuracy) regardless of the ratio when exactly to explore or exploit, UCT has been shown to optimize the accuracy of MCTS, allowing it to compete in much harder to approximate environments - like more complex board and video games and real life (Wang and Gelly). If UCT indicates that exploration is the better choice, it then travels down the game tree by choosing random moves until it reaches an end state. Finally, it backpropagates by moving up the tree and updating the number of wins or losses resulting from each node each time it moves up a node. Once it reaches the root node, it has updated the ratio of wins to losses resulting from the node, and the cycle repeats, creating more accurate probabilities for each child move until UCT indicates that it is confident in the positive outcome of one of the possible children (Chaslot et al.). The discovery of MCTS revolutionized AI for more complicated environments, especially in the Chinese game of Go, which has a 19x19 grid for its board, compared to the 8x8 board in chess. MCTS efficiently searches through vastly more possibilities than minimaxing with alpha-beta pruning, allowing AI development to function in less controlled environments, leading to real-life decision making AI like in driverless vehicles.

One major difference between these two decision-making algorithms and real-world applications is the lack of domain knowledge. Popular implementations of these algorithms like Stockfish, the world champion traditional chess engine, and AlphaGo, a neural network-MCTS hybrid created by DeepMind that became the first program to beat the Go world champion, require vast amounts of information on the rules and typical strategy of the game in question (Silver et al., “Mastering the Game of Go with Deep Neural Networks and Tree Search”). Stockfish, much like a human chess master, does not resort to figuring out new strategies using MCTS until after the first few moves. These opening moves, or “book moves,” have a limited number of quality responses due to the clustered starting arrangement of the chessboard. Stockfish catalogues these moves, many of which are named after their famous users, and chooses from them to begin a match (Ray). This saves valuable processing time in the beginning of the match and guarantees avoiding early blunders. During the match, Stockfish stores information on previously explored nodes in a transposition table, a list of board layouts and what possibilities come from each one, allowing it to avoid re-evaluating moves and their outcomes. Many different evaluation functions, hand-tweaked to follow common strategies, guide Stockfish on which child nodes to explore first. In the final moves, Stockfish then resorts to a similar by-the-book strategy, as every endgame for up to 8 pieces remaining on the board has been calculated in full and stored into Stockfish’s memory (Ray). AlphaGo, on the other hand, uses a neural network trained with games played by expert Go players to guide its selection. Neural networks are a black-box method of developing an AI, modeled after the way humans learn. They use a self-modifying set of “weights,” values that determine how it ranks certain inputs and categorizes them into outputs (Bose and Liang). By attempting to match the strategy of vast amounts of Go games, the neural network becomes better and better at synthesizing moves of its own. However, the neural network cannot think into the future. AlphaGo uses this to its advantage, combining the neural network with MCTS to allow it to choose from moves without manually modifying thousands of evaluation functions (Silver et al., “Mastering the Game of Go with Deep Neural Networks and Tree Search”). While these algorithms work well on their own, adding domain knowledge can improve the accuracy of decision-making AI even further.

One of the main issues with these implementations of decision-making AI in more complex environments, like self-driving cars, is that they either require millions of recordings of past decisions or thousands of hours tweaking and building evaluation functions manually to fit a human’s approach to solving the problem. Even after accomplishing either of these monumental tasks, decision-making AI built with these methods often mimic human bias and mistakes. By learning from humans instead of creating their own rules, AI are limited by the domain knowledge that they are given. Traditional AI, while incredibly successful in many complex environments, lacks the capacity to learn and innovate that neural network-powered AI has (Silver et al., “Mastering the Game of Go without Human Knowledge”). One prominent example of a neural-network based AI that requires no domain knowledge is AlphaZero. AlphaZero, also developed by DeepMind, uses a technique called reinforcement learning to create its own training data. To train with reinforcement learning, the AI plays millions of games against itself, originally playing random moves to discover what happens, but eventually discovers strategies that help it to succeed. After playing 1.2 billion chess games against itself, AlphaZero surpassed Stockfish in skill, playing strategies that no human or traditional AI had been able to conceive of (Silver et al., “A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play”). Another advantage to the lack of domain knowledge in AlphaZero is its ability to pivot between many environments without much manual work, only needing a rulebook for possible moves and what constitutes an end state. AlphaZero and other AIs using this technique have found success in everything from simple, contained environments like chess to more complex and varied environments like Atari video games from the 80s (Schrittwieser et al.). AIs using these techniques are becoming more capable with each new discovery, and will soon be able to operate in the most complex environment of all - real life.

All of these incredibly successful AIs rely on precise, optimized algorithms built to function in their environment. While the neural networks used to power some of the more prominent game AIs are generally flashier and more prominent due to their black-box design, their accomplishments wouldn’t be possible without the underlying algorithms still being improved on today. If we do not continue to optimize and discover new algorithms that function in seemingly frivolous environments, we may never see decision-making AI capable of working to help better humanity.

The original component to my independent study seeks to find a correlation between characteristics of an environment and the effectiveness of several common decision-making algorithms. I will select or recreate environments and build different implementations of AI for the environments. These environments will be two-player and most likely zero-sum, so I will be able to test the AIs in competition against each other to test the effectiveness of different modifications. As I continue to build my project, I hope I can shed light on what characteristics of an AI environment are most beneficial to different approaches to implementing a traditional AI.

Works Cited

Bose, N K, and Ping Liang. Neural Network Fundamentals with Graphs, Algorithms, and Applications. New York ; London, Mcgraw-Hill, 1996.

Chaslot, Guillaume, et al. “Monte-Carlo Tree Search: A New Framework for Game AI.” Association for Computing Machinery Digital Library, 22 Oct. 2008, dl.acm.org/doi/10.5555/3022539.3022579, 10.5555/3022539.3022579. Accessed 30 July 2021.

Fuller, Samuel H., et al. “Analysis of the Alpha-Beta Pruning Algorithm.” Kilthub.cmu.edu, 1 Nov. 2010, kilthub.cmu.edu/articles/journal\_contribution/Analysis\_of\_the\_alpha-beta\_pruning\_algorithm/6603488, 10.1184/R1/6603488.v1. Accessed 30 July 2021.

Kasparov, Garry, and Frederic Friedel. “Reconstructing Turing’s ‘Paper Machine.’” Easychair.org, 14 Sept. 2017, easychair.org/publications/preprint/WjKW, 10.29007/g4bq. Accessed 30 July 2021.

Marckel, Otto. “Alpha-Beta Pruning in Chess Engines.” , 9 June 2017.

Michie, Donald. “Game-Playing and Game-Learning Automata.” Advances in Programming and Non-Numerical Computation, 1966, pp. 183–200, 10.1016/b978-0-08-011356-2.50011-2. Accessed 30 July 2021.

Newell, Allen, et al. “Chess-Playing Programs and the Problem of Complexity.” IBM Journal of Research and Development, vol. 2, no. 4, Oct. 1958, pp. 320–335, 10.1147/rd.24.0320.

Purves, Dale. “Opinion: What Does AI’s Success Playing Complex Board Games Tell Brain Scientists?” Proceedings of the National Academy of Sciences, vol. 116, no. 30, 23 July 2019, pp. 14785–14787, 10.1073/pnas.1909565116. Accessed 28 Sept. 2019.

Ray, Catherine. How Stockfish Works: An Evaluation of the Databases behind the Top Open-Source Chess Engine. George Mason University, 2012.

Schrittwieser, Julian, et al. “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model.” Nature, vol. 588, no. 7839, 23 Dec. 2020, pp. 604–609, 10.1038/s41586-020-03051-4.

Sedgewick, Robert, and Kevin Daniel Wayne. Algorithms. 4th ed., Addison-Wesley Professional, 3 Apr. 2011.

Shannon, Claude E. “Programming a Computer for Playing Chess.” Computer Chess Compendium, New York, NY, Springer, 1988, pp. 2–13, archive.computerhistory.org/projects/chess/related\_materials/text/2-0%20and%202-1.Programming\_a\_computer\_for\_playing\_chess.shannon/2-0%20and%202-1.Programming\_a\_computer\_for\_playing\_chess.shannon.062303002.pdf. Accessed 19 May 2021.

Silver, David, et al. “A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play.” Science, vol. 362, no. 6419, 6 Dec. 2018, pp. 1140–1144, 10.1126/science.aar6404.

---. “Mastering the Game of Go with Deep Neural Networks and Tree Search.” Nature, vol. 529, no. 7587, Jan. 2016, pp. 484–489, 10.1038/nature16961.

---. “Mastering the Game of Go without Human Knowledge.” Nature, vol. 550, no. 7676, Oct. 2017, pp. 354–359, www.nature.com/articles/nature24270, 10.1038/nature24270.

Wang, Yizao, and Sylvain Gelly. “Modifications of UCT and Sequence-like Simulations for Monte-Carlo Go.” 2007 IEEE Symposium on Computational Intelligence and Games, 2007, 10.1109/cig.2007.368095. Accessed 30 July 2021.

‌