Alex Latz

Mr. Griest and Dr. Olid

Honors Independent Study

3 December 2021

Algorithmic Artificial Intelligence in Game Environments

Table of Contents

Literature Review

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 5.9^{61}) 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.

Implementations and Results

Artificial intelligence in its purest form is a decision-making problem. How can a computer, given information about its surroundings and time to think, make an intelligent decision? By representing choices as a branching tree, a computer can visualize and weigh options and their consequences, creating the foundation of a simple artificial intelligence. However, to scale this small decision-making process to complex scenarios like a self-driving car or an assembly-line robot, the process must be optimized and tuned for a specific type of decision. Luckily, simple board and video games have proven remarkably helpful for tuning and testing new techniques such as stochastic simulation and heuristic evaluation due to their clear repeatability and limited move sets. By creating and optimizing AIs for these simple decision-making environments, we can improve and tune AI for much more complex scenarios on a large scale.

For my research, I selected three common games as environments. Two of these games, Connect 4 and Checkers, are what are known as "solved" games, meaning that a mathematician has proved a strategy exists that can guarantee a win or a draw no matter what the opposing player does (Allis) (Schaeffer et al.). However, the third environment, a video game called Clickomania (a variant of SameGame, a widely adapted 1980s game), has challenged computer programmers for years. It is yet to be solved, with computer scientist Therese Biedl proving that it belongs to a class of problems called "NP-complete" (Biedl). NP-complete problems have solutions that can be quickly verified, but there is no way to guarantee finding a solution in varying starting conditions. This clear limitation differentiates Clickomania from the other environments chosen, and makes an optimal AI much harder to fine-tune. Additionally, while

Checkers and Connect 4 are two-player games, Clickomania is a single-player puzzle. This complicates AI development, as algorithmic AI also has to simulate the opposing player and guess moves that their opponent is likely to make. To determine the relative effectiveness of each AI, I used a programming challenge site called HackerRank to simulate both the Clickomania and Checkers AIs against other opponents and with randomized puzzles.

The development process for Clickomania was challenging, and I soon found that I had made assumptions that slowed down my AI and biased it towards incorrect choices. After completing my literature review and reviewing the rules of Clickomania, I knew that I could model the game with a disjoint-set data structure.

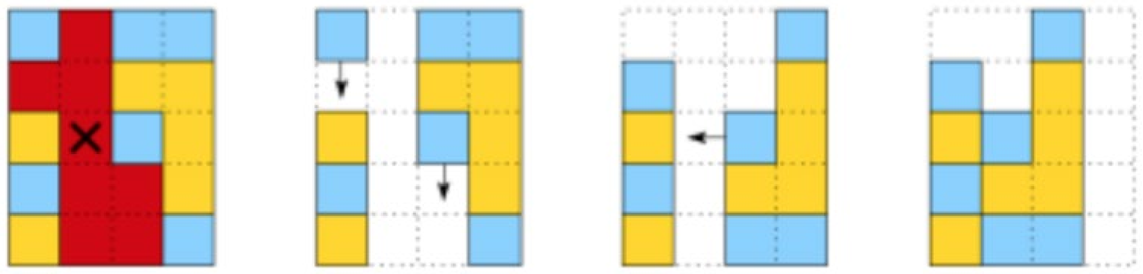


Figure 1: A Clickomania board and the result of eliminating a group of blocks. Ravindranath, Dheeraj. "Click-o-Mania." HackerRank, www.hackerrank.com/challenges/click-o-mania. Accessed 28 Nov. 2021.

By parsing through the board and only allowing the AI to pick from a list of clusters instead of each individual block, I could "prune" vast numbers of potential moves from each point on the game tree. To convert a list of coordinates and colors into states of a game tree, I had to model Clickomania through code, but soon I had a working disjoint-set generation process that would be able to attach separately to any AI. As Clickomania is a single-player game, I knew a simple graph traversal algorithm like A* would be capable of choosing moves. One of the differentiators between the A* algorithm and a simple search is the presence of a heuristic (a mathematical function that approximates the “cost” or benefit of making each move) (Sedgewick and Kevin Daniel Wayne). With a heuristic, A* can avoid exploring paths that it estimates to be too costly. However, after testing several options and doing additional background research, I found that an admissible heuristic for Clickomania is yet to be discovered (meaning that A* search cannot exactly estimate the cost of each move). A* search requires an underestimating heuristic, one that will always give an estimate less than or equal to the actual cost. However, I did manage to find several combinations of factors that gave a slightly overestimating heuristic, which tended to work well even without being exact. This combination of an A* algorithm with an overestimating heuristic was capable of solving only very simple 10x20 grid problems, including the first of four problems on the HackerRank simulation.

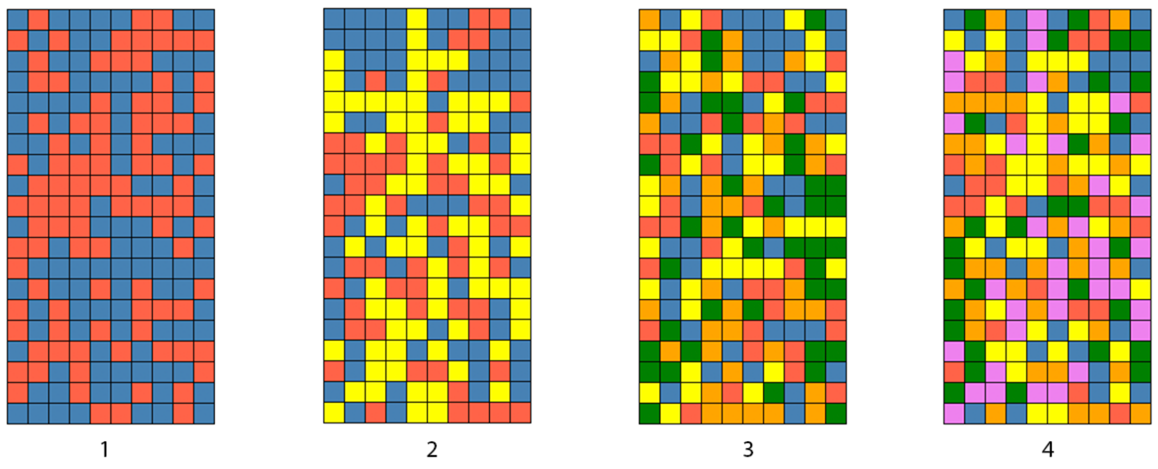


Figure 2: The four Clickomania boards used to test each iteration of the A and MCTS AIs on HackerRank. Ravindranath, Dheeraj. “Click-o-Mania.” HackerRank, www.hackerrank.com/challenges/click-o-mania. Accessed 28 Nov. 2021.*

After several iterations on the heuristic, I had managed to solve two of the four difficult puzzles that HackerRank offered. With a limitation of 2 seconds per move, it could only explore up to 6 moves from its current position before running out of time, making it much harder to solve problems with more than 3 colors of blocks. I experimented with various optimizations, but soon learned that they could not extend the search depth by enough to solve the third problem, as the five available colors greatly increased the number of potential moves. I also implemented an IDA* algorithm approach (iterative deepening A*, which uses less memory but takes longer to run), but it struggled more than the A* implementation due to the limited search time (Korf).

The need for a heuristic was the largest barrier in solving complex problems with an A* or IDA* approach, so I returned to the drawing board to build an AI that did not require an admissible heuristic. After researching non-heuristic search options, I settled on a modified version of SP-MCTS (single player Monte-Carlo tree search) (Schadd et al.). Monte-Carlo Tree Search, unlike most search algorithms, uses stochastic (random) simulation to estimate the average potential of each choice. For a more thorough explanation of Monte-Carlo Tree Search, please refer to my Literature Review. The main modification from Monte-Carlo Tree Search to enable single player games is a change in the UCT formula to accommodate a scoring system instead of the average win percentage.

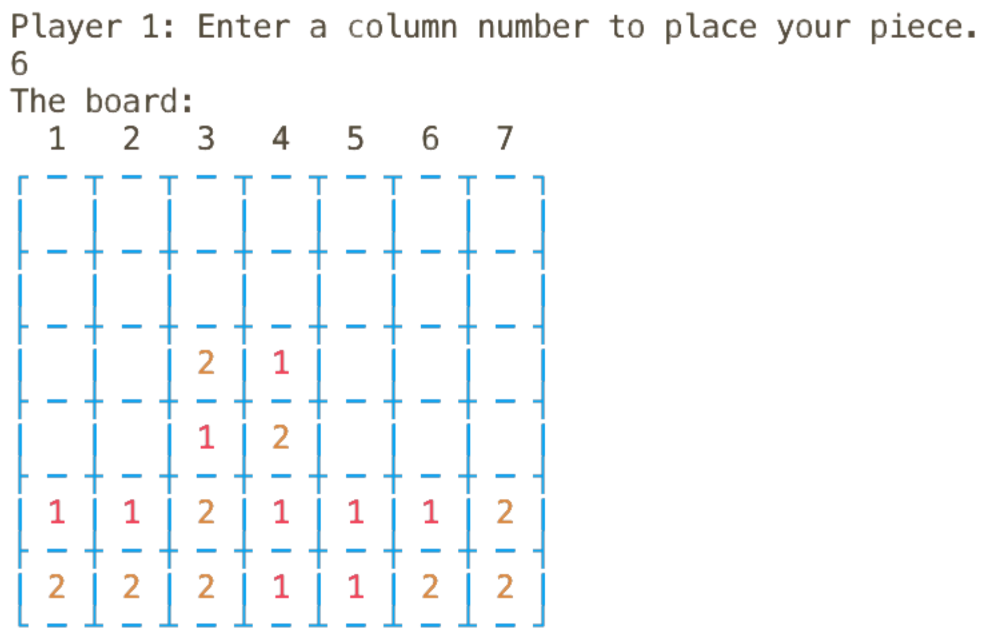
$$\overline{X} \quad + \quad C \cdot \sqrt{\frac{ln\ t\left(N\right)}{t\left(N_i\right)}} \quad + \quad \sqrt{\frac{\sum x^2 - t\left(N_i\right) \cdot \overline{X}^2 + D}{t\left(N_i\right)}}$$

Figure 3: The modified UCT formula for SP-MCTS. C and D are arbitrary constants, while t(N) is the number of parent node visits and t(N_i) is the number of child node visits. $\overline{x}$ is the average score of the node. Schadd, Maarten P. D., et al. "Single-Player Monte-Carlo Tree Search for SameGame." Knowledge-Based Systems, vol. 34, Oct. 2012, pp. 3–11.

However, Clickomania does not have an agreed-upon scoring system. To remedy this, I devised a system that encourages the MCTS algorithm to focus on a solution regardless of the number of moves taken to reach it. Each removal awards $(N\text{-}2)^2$ points, where N is the total number of blocks removed in one move. Unsolvable boards are penalized 5,000 points, while solved boards are given a 10,000-point bonus. This scoring system had immediate success, finding much simpler solutions for the first and second Clickomania boards with a 2-second runtime. Additionally, it was able to utilize its time more efficiently, exploring a broad number of outcomes instead of locking itself into one move at a time like the A* and IDA* approaches.

One of the important parts of tuning and building a MCTS-style AI is modifying the arbitrary exploration and exploitation constants in the modified UCT formula. As Schadd et al. explain, “The pair of constants (C; D) of [singleplayer UCT], must be modified in order to investigate which balance between exploitation and exploration gives the best results… The parameter pair (0.1; 32) represents exploitation, (1; 20,000) performs exploration, and (0.5; 10,000) is a balance between the other two” (Schadd et al.). After several iterations on the C and D constants, I found that the pair (0.4; 10,000) helped the AI balance between exploration and exploitation best with my scoring system. This pair of constants was able to complete the third board on HackerRank, putting my AI in the top 9% of HackerRank submissions. I made significant progress towards completing the fourth board, but was not able to complete it without restructuring the AI significantly, so I concluded my study of Clickomania.

For my Connect 4 project, I decided to use a previous implementation of a minimax AI I constructed in AP Computer Science A as part of my accelerated program while preparing for this Independent Study. Connect 4, being a two-player game, requires very different strategies to be solved by an AI.



```
Player 1: Enter a column number to place your piece.
6
The board:
  1   2   3   4   5   6   7
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   | 2 | 1 |   |   |   |
|   |   | 1 | 2 |   |   |   |
| 1 | 1 | 2 | 1 | 1 | 1 | 2 |
| 2 | 2 | 2 | 1 | 1 | 2 | 2 |
```

Figure 4: My Connect 4 renderer displaying the board from the computer's perspective.

While it can still be viewed under the lens of a game tree, each level of the tree must correspond to a certain player's turn, with child nodes alternating between each player. A simple tree search does not work on a two-player game tree, as it does not account for the other player's moves and winning probabilities. To account for this, the minimax algorithm was created. Minimax uses a standard tree search on a multiplayer game tree, aiming to pick the move that minimizes the opponent's ability to play beneficial moves while maximizing its ability to play beneficial moves (Fuller et al.). This may not result in the most beneficial move with each turn, as minimax search instead attempts to maximize its chance of winning out over the opponent. However, minimax search struggles against suboptimal players, as it aims to protect itself assuming the opponent chooses intelligent moves. This can lead to significant issues when facing novices, so minimax search is generally only used in high-level AI for multiplayer games.

While implementing minimax search in my Java Connect 4 game and renderer, I decided to use the alpha-beta pruning optimization to limit the potential number of nodes considered at one time (Fuller et al.). Alpha-beta pruning, as explained in my Literature Review, was able to significantly increase performance during Connect 4 games, and it was able to best novice players on several online Connect 4 websites. Like the A* AI for Clickomania, it suffered from several flaws due to its need for a heuristic function that was both time-intensive to compute and non-optimal. After a large amount of tuning, the AI was able to play at an average human level, settling at around 1000 ELO (ranking points) on the papergames.io Connect 4 leaderboard (Galvis).

To implement a stochastic search, I decided against using Connect 4 as it is a strongly solved game. Even a badly-implemented MCTS would be unbeatable in a strongly solved game as long as it had the first-player advantage, so I decided on using Checkers, a weakly-solved game (Schaeffer et al.). Checkers took over 18 years of computation to weakly solve (meaning that there is a list of moves that can be played to guarantee a win from every position), but its large game tree depth and number of possible moves make finding this solved set of moves unrealistic during normal play, especially with a reasonable time limit on the MCTS search.

MCTS was originally designed for two-player games like Checkers, and instead of a scoring system, regular MCTS uses the average win percentage at each node (Chaslot et al.). This makes the simulation stage much simpler as it does not need to calculate the score with each move played on the board. It also does not suffer from the same drawbacks as minimax search, as it does not assume the opposing player will always play an optimal move. I used the HackerRank Checkers simulation to test my AI, as it can compare relative results of my AI implementation against other AIs.

My MCTS implementation for Checkers was fairly straightforward until I arrived at the simulation stage. In Checkers, promoted pieces are able to move back and forth, allowing an infinite game tree depth. This meant that the MCTS simulation was unable to finish as there was an equal chance of promoted pieces moving forwards and backwards. To remedy this, I added the ability to detect whether two moves on a board were identical and kept a list of played moves in each simulation run-through. However, this did not fully eliminate the problem, as many simulations contained at least one unreasonable or completely pointless move. I attempted several different solutions to weed out these unreasonable moves, finally settling on a solution that used a heuristic function to sort potential moves, enticing pieces to move to the center of the board or the opposite side to be promoted.

This solution, while improving the results of the Checkers AI, was only barely able to compensate for the random simulation phase's ability to play meaningless moves. By wasting time during each simulation moving at least one piece backwards, it was not able to run a reasonable number of simulations to accurately rank the potential of its potential moves.

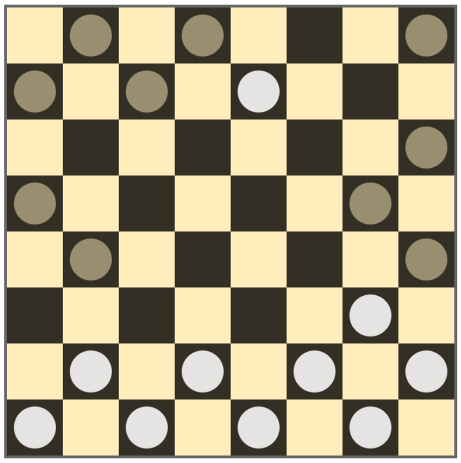


Figure 5: The Checkers AI played very aggressively, as it did not have enough data to know these moves were risky (my AI player is playing as white). Ravindranath, Dheeraj. "Checkers." HackerRank, www.hackerrank.com/challenges/checkers. Accessed 30 Nov. 2021.

Although I did not have time to fully finish fixing this error, my temporary fix allowed the Checkers AI to play at a novice level, even while incorrectly selecting and simulating moves. In the future, I would love to investigate techniques like transposition tables and storing results of simulated moves to improve the simulation performance of MCTS in games like Checkers with repeatable moves (Slate and Atkin).

Through building and testing these three AIs, I have learned a great deal about the process of assembling algorithmic AIs and modeling environments in both Java and C++ to fit a game tree model. I have also been able to collect a vast amount of data on the effectiveness of stochastic and heuristic search methods for AI in different environments. Heuristic search methods, while effective for simple problems or solved games, struggle greatly in games where an optimal heuristic is not feasible. Stochastic simulation solves many of these problems while also introducing new issues, as its random-sampling method allows it to approximate an ideal heuristic without actually solving for it. In the vast majority of simple environments, stochastic simulation can outperform heuristic simulation due to its lack of a heuristic function and ability to adapt easily to both singleplayer and multiplayer environments.

Works Cited

Allis, Victor. “A Knowledge-Based Approach of Connect-Four.” *ICGA Journal*, vol. 11, no. 4, Dec. 1988, pp. 165–65, https://doi.org/10.3233/icg-1988-11410.

Biedl, Therese C., et al. “The Complexity of Clickomania.” *ArXiv:cs/0107031*, July 2001, arxiv.org/abs/cs/0107031.

Bose, N. K., and Ping Liang. *Neural Network Fundamentals with Graphs, Algorithms, and Applications*. Mcgraw-Hill, 1996.

Chaslot, Guillaume, et al. “Monte-Carlo Tree Search: A New Framework for Game AI.” *Association for Computing Machinery Digital Library*, Oct. 2008, https://doi.org/10.5555/3022539.3022579.

Fuller, Samuel H., et al. “Analysis of the Alpha-Beta Pruning Algorithm.” *Kilthub.cmu.edu*, Nov. 2010, https://doi.org/10.1184/R1/6603488.v1.

Galvis, David. “Connect 4 Online.” *Papergames.io*, papergames.io/en/connect4. Accessed 30 Nov. 2021.

Kasparov, Garry, and Frederic Friedel. “Reconstructing Turing’s ‘Paper Machine.’” *Easychair.org*, Sept. 2017, https://doi.org/10.29007/g4bq.

Korf, Richard E. “Depth-First Iterative-Deepening.” *Artificial Intelligence*, vol. 27, no. 1, Sept. 1985, pp. 97–109, https://doi.org/10.1016/0004-3702(85)90084-0.

Marckel, Otto. “Alpha-Beta Pruning in Chess Engines.” *UMinn Morris Senior Seminars*, 9 June 2017, umm-csci.github.io/senior-seminar/seminars/spring2017/marckel.pdf.

Michie, Donald. “Game-Playing and Game-Learning Automata.” *Advances in Programming and Non-Numerical Computation*, 1966, pp. 183–200, https://doi.org/10.1016/b978-0-08-011356-2.50011-2.

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. 320–35, https://doi.org/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, July 2019, pp. 14785–87, https://doi.org/10.1073/pnas.1909565116.

Ravindranath, Dheeraj. “Checkers.” *HackerRank*, www.hackerrank.com/challenges/checkers. Accessed 30 Nov. 2021.

---. “Click-o-Mania.” *HackerRank*, www.hackerrank.com/challenges/click-o-mania. Accessed 28 Nov. 2021.

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

Schadd, Maarten P. D., et al. “Single-Player Monte-Carlo Tree Search for SameGame.” *Knowledge-Based Systems*, vol. 34, Oct. 2012, pp. 3–11, https://doi.org/10.1016/j.knosys.2011.08.008.

Schaeffer, J., et al. “Checkers Is Solved.” *Science*, vol. 317, no. 5844, July 2007, pp. 1518–22, https://doi.org/10.1126/science.1144079.

Schrittwieser, Julian, et al. “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model.” *Nature*, vol. 588, no. 7839, Dec. 2020, pp. 604–9, https://doi.org/10.1038/s41586-020-03051-4.

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

Shannon, Claude E. “Programming a Computer for Playing Chess.” *Computer Chess Compendium*, Springer, 1988, pp. 2–13, https://doi.org/10.1007/978-1-4757-1968-0_1.

Silver, David, et al. “A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play.” *Science*, vol. 362, no. 6419, Dec. 2018, pp. 1140–44, https://doi.org/10.1126/science.aar6404.

---. “Mastering the Game of Go with Deep Neural Networks and Tree Search.” *Nature*, vol. 529, no. 7587, Jan. 2016, pp. 484–89, https://doi.org/10.1038/nature16961.

---. “Mastering the Game of Go without Human Knowledge.” *Nature*, vol. 550, no. 7676, Oct. 2017, pp. 354–59, https://doi.org/10.1038/nature24270.

Slate, David J., and Lawrence R. Atkin. “CHESS 4.5—the Northwestern University Chess Program.” *Chess Skill in Man and Machine*, 1983, pp. 82–118, https://doi.org/10.1007/978-1-4612-5515-4_4.

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, https://doi.org/10.1109/cig.2007.368095.