Thesis Journal

# Artifact

My current working title for thesis is Deckbuilding Game AI Using Monte Carlo Tree Search. For my artifact I’d be building out the base game of Dominion. Dominion is a deckbuilder where every turn you buy cards from 10 different types. The goal is end the game with the most victory points from cards that you buy. Most cards don’t give victory points but will help you to get more money to eventually buy the highest victory point cards. To play the game with an AI I’ll be using the Monte Carlo Tree Search algorithm. To make it more thesis worthy I’ll be comparing different selection algorithms like Upper Confidence Bounds for Trees (UCT) and Upper Confidence Bounds (UCB). To show mastery I will compare by Monte Carlo AI versus simple but strong Dominion AI strategies like “Big Money” or “Single Witch” to prove my algorithm is strong. I can also compare my results against the results of other papers I have found.

The below are the sources I’ll use. The first will be my main source since it’s a MCTS implementation of Dominion as well. I’ll use their way of using finite state machines as comparison. The “Big Money” source is a great explanation of these simple Ais. The third source is “Game Ais with Minimax and Monte Carlo Tree Search” which explains how MCTS works and how I can make a basic AI first for Tic Tac Toe. My other sources are augmentations to my first source in applications of MCTS.

# Annotated Bibliography

|  |  |
| --- | --- |
| [1] | J. V. Jansen and R. Tollisen, "An AI for Dominion Based on Monte-Carlo Methods," 2 June 2014. [Online]. Available: https://pdfs.semanticscholar.org/28b6/ada13e948cfaee4af5138ee667d404eb01ac.pdf. [Accessed 9 July 2020].  The paper starts by saying Dominion is difficult to make an AI because of hidden information and stochastic elements. The paper uses Upper Confidence Bounds (UCB) and Upper Confidence Bounds applied to Trees (UCT). The author explains the basic rules of Dominion and also states a big challenge is that each game will have different starting cards which cause different behaviors. Other Dominion AIs use state machines. UCT and UCB are used as part of the selection formula to determine which tree path to explore. The difference being UCT has a exploration constant C that can allow for more tuning. They created finite state machines for comparison being AI-BigMoney, AI-SingleWitch, and AI-Random. They found SingleWitch is by far the best of these finite state machines, so they compared it agains UCB and UCT. UCB started beating Single Witch at 10,000 simulations whereas UCT needed 100,000 with a tuned C of 5. When played against each other, UCB outperformed UCT.  The paper’s analysis shows that UCB and UCT are great ways of tweaking MCTS to find the optimal algorithm for Dominion. Its use of “Big Money” and “Single Witch” are good for how I can compare my algorithm against traditional strategies. |
| [2] | "Big Money," 20 October 2018. [Online]. Available: http://wiki.dominionstrategy.com/index.php/Big\_money. [Accessed 9 July 2020].  Big Money is a strategy used to play Dominion. Its focus is on buying Treasure cards with very few action cards. The pure version of it is ONLY buying basic cards. Because of the simplicity of this strategy its useful as a baseline AI. The pure version follows these rules: Buy Provinces at 8 money, buy Gold with 6-7, buy silver with 3-5, and buy nothing otherwise. There are simple optimizations that can be made like buying Duchies at 5 money if there are 5 provinces left. The reason this strategy is good is beginners like to buy the “fancy Actions” even if they don’t fully understand how the game works. Big Money is a way of confirming if your strategy is good enough. Another benefit of Big Money is it works in every set of Dominion and requires very little thinking. It is also a good way of iterating on strategies. If you do Big Money plus one action, you’ll normally beat Big Money. What about 2 or 3? The most common way of making Big Money better is “Terminal draw”. This means adding action cards that add lots of draw but don’t give more actions themselves. The big example is Smithy which gives +3 cards only. This allows for drawing more treasure cards which you should have really good ones of.  For my thesis Big Money can be super useful in testing my AI. I can write it first and then use it to tune my AI to what works best. I can also use variations of Big Money to get more variation for playtesting. I could dive in a little bit on a state machine to come up with better strategies. |
| [3] | P. Muens, "Game AIs with Minimax and Monte Carlo Tree Search," 3 April 2019. [Online]. Available: https://towardsdatascience.com/game-ais-with-minimax-and-monte-carlo-tree-search-af2a177361b0. [Accessed 9 July 2020].  Philipp Muens starts by saying there are two modes of play: Aggressive where you make a move to win the game/set up a winning situation or Defensive where you prevent the opponent from winning the next round/set up to prevent future winning situations. This is perfect for explaining Tic-Tac-Toe to choose your next move. A game tree used for algorithms would be all possible moves the player can make and all possible moves then the opponent will make. He then takes a dive into Minimax. If you take chess and add up all of the white piece values while subtracting the black piece values, you’ve created an evaluation function. Of course you’d need a more complicated evaluation algorithm, but this would allow you to determine which is your best move. You can’t look one level deep though because chess is back and forth, so you must choose your best move while also assuming the opponent will choose their best move. White is maxing and black is minimizing. You want to look as deep as possible, but the amount of moves makes this difficult. To make it more manageable Alpha-Beta Pruning is used to remove the worst moves, so you don’t have to traverse as far down the tree. Deep Blue which beat Garry Kasparove heavily used parallized Alpha-Beta search algorithms.  The big problem with Minimax is its evaluation function. Montecarlo tries to solve this with randomly choosing moves and finding out which worked best. Monte Carlo uses the law of large numbers to guess that the one move that wins the most is probably the best. It uses the “Upper Confidence Bound 1 formula which is xi + C sqrt( ln( N ) / ni ) where xi is the average value of the game state, C is a constant “temperature” (he sets to 1.5), N isparent node visits and ni is current node visits. We start with a tree of only one move, then play a game from each of the possible moves and plug in the results of the moves. Unexplored moves have values approaching infinity, so you always explore those first. Once you’ve found a node that wins more, you extend its tree and continue again. To handle the high computation of this method, C can be changed. A high C means more exploring unvisited nodes, but a low C will hit visited nodes more to gather more info.  This paper is useful in explaining UCB, UCT, and why even using Monte Carlo in the first place. It is helpful in learning how it works in the background and how I can make a simple AI as a test piece. |
| [4] | G. J. B. Roelofs, "Monte Carlo Tree Search in a Modern Board Game Framework," 25 January 2012. [Online]. Available: https://project.dke.maastrichtuniversity.nl/games/files/bsc/Roelofs\_Bsc-paper.pdf. [Accessed 9 July 2020].  The paper describes a framework for playing complex board games with Settlers of Catan as a proof-of-concept. MCTS is used because it doesn’t require knowledge of how the game is played. Two changes are used, a simplification of Chance Node model from Expectimax and move groups to allow for use on a non-deterministic game. An avenuse of research is General Game Playing (GGP) where one algorithm can play multiple different types of games. A problem with MCTS is high branching can hide possible replies. He explains the graph state and cycle using “Placeables” as all pieces that can be placed. There are rule changes implemented to remove some of the branching and hidden moves. Monte Carlo uses four phases: Selection, Playouts, Expansion, and Backpropagation. Selection selects a node to research, Playouts plays a game from a specific node, Expansion is the new node that is added to the tree, and Backpropagation where information from the games traverses back up the tree. Selection uses UCT as the algorithm (research further). Playout uses pure random play since the focus of the paper is on no domain knowledge. For Backpropagation, Average was chosen from between Max, Average, Informed Average, and Mix since they say it’s the best. A chance node function replaces the backpropagation function when it is their turn. The function is the sum of the chance of the child nodes \* the value of the child nodes. An alternative is Grouped Chance Model where the chance node is grouped with the child nodes. This allows for faster converging of the expected average, but it can hide moves. For the experiment, there were no cutoff for playouts and 2 seconds per move. Roughly 1300 playours per second were calculated. The results showed MCTS outperformed the Group Chance Model changes. Looking into RAVE as a possible online learning model is a good place to look for improvements.  This paper shows more ways of tweaking the four phases of MCTS, and it also shows the results of the tweaks they made. These can be used for my own tweaks when comparing different variations of MCTS. |
| [5] | "Expectiminimax," Wikipedia, 21 April 2020. [Online]. Available: https://en.wikipedia.org/wiki/Expectiminimax. [Accessed 9 July 2020].  Expectiminimax is a variation of minimax where chance is part of the game. Instead of just the AI maxing and then minimizing for the player turn, a “chance” node is added which represents the random effect. Using a theoretical game of a die is thrown followed by a decision by the ai and opponent, the order would be “chance”, “max”, “min”. A proposed algorithm is shown which is a recursive tree traversal which has cases for the terminal node and the above 3. This information is useful to explain the other papers being read.  Because Dominion has stochastic elements, chance nodes could be useful for augmenting Monte Carlo to handle these elements. |
| [6] | C. Huchler, "An MCTS Agent For Ticket To Ride," 8 July 2015. [Online]. Available: https://project.dke.maastrichtuniversity.nl/games/files/msc/Huchler\_thesis.pdf. [Accessed 9 July 2020]. |

This thesis written by Carina Huchler focuses on using Monte-Carlo Tree Search for Ticket To Ride. Carina explains the difficulties of writing an AI for Ticket to Ride since the game has a lot of unknown information. You don’t know what cards your opponents have and you don’t know what cards you will draw. She explains different search algorithms and the MCTS (Monte-Carlo Tree Search) is a best-first search algorithm. You build a tree and then choose the most promising one. MCTS has seen success in Catan and Bridge. The paper explains the rules of Ticket To ride, MCTS, and then how MCTS could be used for Ticket to Ride. She then explains how different types of MCTS agents compared against different players like Single Observer and Cheating Player. In conclusion MCTS is successful in Ticket to Ride with Flat Monte-Carlo working best. The thesis only focused on two players, so further research would be necessary to test if MCTS would work with more.

This is another paper of MCTS being used on a game with unknown information. I can use it to help find ways of augmenting MCTS.