Graphical user interface, website

Description automatically generated

Technical Design Document

**Dominion AI Using Monte Carlo Tree Search (MCTS)**

Version 1.0

|  |  |  |
| --- | --- | --- |
| **Position Title** | **Name** | **Signature** |
| Programmer | *Jonathan Sarasua* |  |
| Advisor | *Matt Butler* |  |

Table of Contents

[Table of Figures 3](#_Toc71045923)

[Introduction and Motivation 4](#_Toc71045924)

[Overview 5](#_Toc71045925)

[Scope 5](#_Toc71045926)

[Product 5](#_Toc71045927)

[Deliverables 7](#_Toc71045928)

[Table of deliverables 7](#_Toc71045929)

[Technology Sources 8](#_Toc71045930)

[Background and Previous Work 9](#_Toc71045931)

[MCTS 9](#_Toc71045932)

[Dominion 9](#_Toc71045933)

[Theory 12](#_Toc71045934)

[Monte Carlo Tree Search 12](#_Toc71045935)

[Selection 12](#_Toc71045936)

[Expansion 13](#_Toc71045937)

[Simulation 13](#_Toc71045938)

[Backpropagation 13](#_Toc71045939)

[Choosing the Best Move 14](#_Toc71045940)

[Implementation 14](#_Toc71045941)

[Data Structure 14](#_Toc71045942)

[Selection 15](#_Toc71045943)

[Expansion 16](#_Toc71045944)

[Simulation and Backpropagation 16](#_Toc71045945)

[Multithreading 16](#_Toc71045946)

[Victory Point Nudge 17](#_Toc71045947)

[Results 17](#_Toc71045948)

[Lesson Learned 18](#_Toc71045949)

[RandomPLUS 19](#_Toc71045950)

[MCTS Using Big Money 19](#_Toc71045951)

[Chaos Chance 19](#_Toc71045952)

[Future Work 20](#_Toc71045953)

[Conclusion 20](#_Toc71045954)

[Bibliography 21](#_Toc71045955)

# Table of Figures

[Figure 1: Dominion Artifact 9](#_Toc69832943)

[Figure 2: Curse and Victory cards [2] 10](#_Toc69832944)

[Figure 3: Treasure cards [2] 10](#_Toc69832945)

[Figure 4: Examples of Action cards [2] 10](#_Toc69832946)

[Figure 5: MCTS Steps [3] 12](#_Toc69832947)

[Figure 6: UCT Formula 13](#_Toc69832948)

[Figure 7: Node Legend Figure 8: MCTS Tree Data structure 14](#_Toc69832949)

[Figure 9: Node data structure 15](#_Toc69832950)

[Figure 10: UCT Formula 15](#_Toc69832951)

[Figure 11: Modified UCT Formula for Dominion 15](#_Toc69832952)

[Figure 12:Get next game state from game 16](#_Toc69832953)

[Figure 13:Thread diagram 17](#_Toc69832954)

[Figure 14: MCTS 100,000 iterations versus Big Money 18](#_Toc69832955)

[Figure 15: Village Card [5] Figure 16: Witch Card [5] 18](#_Toc69832956)

[Figure 17: MCTS RandomPLUS versus Big Money 19](#_Toc69832957)

[Figure 18: MCTS Big Money versus Big Money 19](#_Toc69832958)

[Figure 19: MCTS 15% Chaos Chance versus 0% Chaos Chance 19](#_Toc69832959)

# Introduction and Motivation

At SMU Guildhall the curriculum focuses on making real time games that are easy to show off common features like physics and rendering. Turn based card games like Dominion was an interesting choice to try a new area that was not covered in previous classes. Many AI strategies were looked at as possibilities for a thesis topic, but the one that kept popping up was Monte Carlo Tree Search. It was used in Google’s AI, AlphaZero, to play games like Chess and Go by combining it with a neural network. [1] Also, the strongest version of a Hierarchical Portfolio Search was found to use MCTS as the overarching brain to search over the created portfolios of move combinations. [2] This shows MCTS can be a strong AI strategy on its own, but it can also be combined with other AI strategies to make an even more powerful AI.

Dominion is a deckbuilding game where every turn you try to play and buy cards to make your deck better with the eventual goal of buying more victory point cards than your opponents. The artifact is both a playable game of Dominion and an AI using Monte Carlo Tree Search (MCTS) as its algorithm. MCTS had to be modified to handle the hidden information challenges present in Dominion. For example when you draw a hand of cards each turn, you will not draw the same hand every time.

# Overview

Deckbuilding games like Dominion have built in complexity with handling hidden information like not knowing what cards will be drawn or what cards the opponents have. Every card added to the player’s deck changes the way the deck plays. MCTS is an exciting AI strategy because of not relying on domain knowledge. The focus can then be on making minor changes instead of trying to solve the entire game.

## Scope

Building an AI first requires building the game of Dominion. All of Dominion is a massive game, so it was simplified into only using a subset of cards and the cards would not be changed out each game like in regular Dominion. The game would be fully playable for a two-player game. Both players can have an AI play moves, and an auto play feature is available for running tests. The MCTS AI will be fully tweakable, and two different MCTS AIs can be set up to play against each other.

## Product

The product is both the game of Dominion along with the MCTS AI.

Monte Carlo Tree Search AI

* Tree
  + Data structure to store game state, pointer parent, child nodes, and metadata
* MCTS tree building iteration loop doing the following steps
  + Select
  + Expand
  + Simulate/Rollout
  + Backpropagate
* Use a separate thread to run the AI to allow for more user friendly experience
* Use separate threads for the simulation step to increase the speed of the iteration loop
* Implement saving and loading of the tree to allow for the AI to continue to learn
* Variables exposed in menus to change settings in the AI

Dominion

* Cards that can be bought and played
  + Treasure cards
    - Copper
    - Silver
    - Gold
  + Victory cards
    - Estate
    - Duchy
    - Province
  + Curse cards
    - Curse
  + Action cards
    - Village
    - Smithy
    - Laboratory
    - Festival
    - Market
    - Witch
    - Council Room
* Player Actions
  + Play Card
  + Buy Card
  + End Phase
* UI
  + Mouse controlled UI System
  + Controls to play moves, switch between players
  + Controls to modify the AIs playing for player 1 and 2

# Deliverables

## Table of deliverables

|  |  |
| --- | --- |
| **Asset** | **Description** |
| *Engine/Code/Engine/…* | Personal C++ engine, Visual Studio project included |
| *Engine/Code/ThirdParty/…* | External tools STB Image and TinyXML2 |
| *Dominion/Code/Game/…* | Dominion game C++ source code |
| *Dominion/Code/Game/…* | Monte Carlo Tree Search C++ source code |
| *Dominion /Run/Fonts/…* | Bitmap font used by game and editor |
| *Dominion/Run/Dominion.sln* | Dominion solution file |
| *Dominion/Run/Data/Images/…* | Dominion card textures |
| *Dominion/Run/Data/Shaders/…* | Default shader use by game |
| *Dominion/Run/fmod(64).dll* | 2 FMod audio DLLs for x86 and x64 compilations |
| *Dominion/Run/Dominion\_x64.exe* | Game executable (x64) |

# Technology Sources

The Dominion project was developed using Visual Studio 2019. *STB Image* was used for image loading the card textures. *TinyXml2* was used for reading XML.

# Background and Previous Work

### MCTS

MCTS has been used with many different games like Alpha Zero with chess. [1] It is commonly mixed with other AI strategies with thousands of papers covering a wide range of use cases. MCTS with Dominion though has been researched very little. Only one paper was found on the topic by Jansen and Tollisen. [1] They changed the simulation step to use a greedy heuristic to solve issues random has with the game of Dominion. They tried out a multitude of tweaks to try to find what works best for Dominion.

Graphical user interface, website

Description automatically generated

Figure 1: Dominion Artifact

### Dominion

**Game setup for a two-player game:**

* Card piles are set up for each card being used as shown in the above image
* Treasure cards use their entire pile size
* Each victory card pile has a size of eight
* Each action card pile has a size of ten

**Goal:**

* Have the most victory points from victory cards by the end of the game

**Card Types:**

A picture containing text

Description automatically generated

Figure 2: Curse and Victory cards [2]

A picture containing text

Description automatically generated

Figure 3: Treasure cards [2]

A picture containing calendar

Description automatically generated

Figure 4: Examples of Action cards [2]

**Game Ends When:**

* Any three piles are empty or the Province card pile is empty

**Start of game:**

* Each player starts with a deck of seven coppers and three estates
* The decks are shuffled, and each player then draws five cards

**Each Turn:**

Player’s will take turns doing the following in order

1. Action Phase
   1. Play one action card
2. Buy Phase
   1. Buy one card from one of the card piles
      1. The card bought goes to the player’s discard pile
3. Cleanup Phase
   1. All cards in the player’s hand and play area are discarded to their discard pile
   2. A new hand of five cards are drawn
   3. All resources gained this turn are reset back to their starting amounts
      1. Money is set to zero
      2. Number of actions and buys is set to one

**Strategy:**

* Players want to buy action and treasure cards to build up enough money to buy victory cards
* The province pile running out is the normal way most games will end
* Trying to reach eight money consistently, so the player can repeatedly buy provinces each turn is the most common strategy

**Simple AI Strategies**

These are common baseline strategies players use to test if a strategy they are trying is good.

* Big Money [2]
  + If you have eight money, buy a Province
  + Else if you have six money buy a Gold
  + Else if you have three money buy a Silver
  + Else do nothing
* Single Witch
  + If you have five money and don’t have a witch, buy a Witch
  + Else follow Big Money
* Double Witch
  + If you have money and don’t have two witches, buy a witch
  + Else follow Big Money

# Theory

## Monte Carlo Tree Search

MCTS is an AI strategy that tries to fuse the benefits of a tree and other AIs. In pure MCTS the other AI is random moves, but it could be a heuristic, genetic algorithm, or even a neural network. A tree in games is a data structure where each node contains a game state and a list of child nodes with an input move to reach that node. There is a starting root node that becomes a massive branching tree from all its child nodes and their child nodes and so forth.

If a complete tree of all game states existed for a game, you could solve for which move is the best. Unfortunately, most games are too large to completely map in a tree, and a tree search by itself cannot give a result without being complete. MCTS is a method of trying to build a tree while always able to return what the tree believes is the best move. It will explore the most promising moves in a game first while still exploring other moves. MCTS will do the following four steps done in a loop as many times as possible. The AI becomes more and more confident in its best move the more iterations of these four steps it has done.

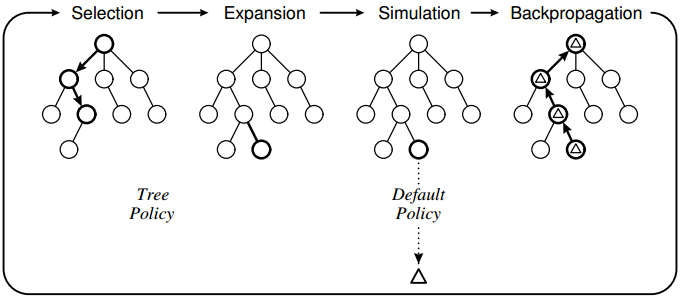


Figure 5: MCTS Steps [3]

### Selection

The first step in MCTS is selection. Here the AI moves down the tree looking for the best node to expand. Starting with the root node, two questions are asked:

Do all possible moves from the current node have a corresponding child node?

If yes, we have found our node we need to expand, so we move the expansion step.

If no, what child node is the best to continue down the tree?

The first question is yes only if the number of child nodes is equal to the number of possible moves. If not all moves have been expanded, then a random move of the remaining choices is chosen for the expansion step. [1]

To choose the best child node to continue the selection process, the Upper Confidence Bounds Applied to Trees (UCT) formula is used to calculate a value for each child node. Whichever node has the highest value will be the new current node and we repeat the questions listed above.

##### Upper Confidence Bounds Applied to Trees (UCT)

A picture containing diagram

Description automatically generated

Figure 6: UCT Formula

The UCT formula is the idea that exploitation should be balanced with exploration. [4] Exploitation is looking deep into the tree focusing on the best move. The best move is normally considered the one with the highest win rate. Exploration is looking wide into the tree. The UCT formula is relative to the current node in the tree. Score can be any method of evaluating a node, but it is normally the win rate at the current node. C is the exploration value which can be changed to effect if the selection should go deeper or wider. It is theoretically √2. [4] The exploration value is everything on the right side and is roughly a value between zero and one, so it can be compared against the score. The simulation data is gathered in the simulation and backpropagation steps. Each node stores the number of wins and the number of simulations of itself and all its children. Because of this, the exploration value gets larger if the current node has been visited very few times relative to its siblings. For example if a node and its siblings have been simulated ten times, then their parent node will save eleven in its simulation count. If one of the child nodes has only been visited once, it will have a higher exploration value than another child node that has been visited five times.

### Expansion

Once the selection step has chosen the best node to expand, the expansion will get a random move that can be made that is not already a child. It will then play that move and create a new node with the new game state stored inside. This new node will be added as a child to the currently selected node. The new node is passed along to the simulation step.

### Simulation

Given the node from the expansion step, a game of random moves for both players is played. The simulation step will return the result of the game played and will then be passed to the backpropagation step. Instead of random moves, any strategy of playing the game could also be used. Using other AIs could speed up how quickly MCTS finds a strong move.

### Backpropagation

Backpropagation takes the result from the simulation and saves it in the current node. Each node has the number of wins and simulations saved. The result is then sent up the tree where each parent also saves the result. This metadata of the number of wins and simulations can be used to calculate the win rate of a move and is used in the selection step. Note that the number of wins is relative to the player that played the move to reach the current game state, so as the result gets sent up the tree it will flip between a win or loss depending on the current node.

### Choosing the Best Move

One of the benefits of MCTS is the ability to ask the tree at any point what it believes the best move is. The two most common methods are getting the move with the highest win rate or the move that has been simulated the most. Using win rate means the result may change more often especially at lower simulation counts since it has not explore many different paths. Using the most simulated means you are putting your faith in the selection algorithm on which it believes it has as the most confident and thus the most winning move.

# Implementation

Implementing MCTS in Dominion presented many challenges that required changes to the four steps in MCTS as well as how the data is stored. Below are the changes made.

### Data Structure

Shown in the diagram below is the change to the data structure for a game like Dominion. Now any move could lead to multiple possible outcomes. An example scenario is player 1 ends his turn and draws his new hand of cards. They could draw four coppers and one estate, but they also could draw three coppers and two estates.

Chart, bubble chart

Description automatically generatedDiagram

Description automatically generated with medium confidence

Figure 7: Node Legend Figure 8: MCTS Tree Data structure

Graphical user interface, diagram

Description automatically generated

Figure 9: Node data structure

### Selection

Selection changes the most of all the steps for a game like Dominion. Now when using the UCT formula to calculate which is the best move, the sum of wins and simulations must be used. The score changes from (Wins at node)/(Current node sims) to (∑Outcome wins)/(∑Outcome sims). The formula changes as shown below:

A picture containing diagram

Description automatically generated

Figure 10: UCT Formula

Diagram

Description automatically generated with medium confidence

Figure 11: Modified UCT Formula for Dominion

When moving down the tree after a move has been selected, now there is the problem of deciding which outcome will become the current node. Jansen and Tollisen used sampling to determine which outcome would be taken. [1] While this solution would work, it was another place where bugs could occur. Fortunately, all the code to decide the outcome was already implemented in the Dominion game code. The only extra change needed is randomizing the unknown information like deck order and the opponent’s deck and hand. Graphical user interface

Description automatically generated

Figure 12:Get next game state from game

The new game state is compared against the game states stored in the array for the best move. If the outcome is found, we have found the new current node to repeat the selection process. If it does not exist, then we have found the node to expand.

### Expansion

Previously the expand step only expanded a completely new move. Now it must also be able to add a new outcome passed by the select step.

### Simulation and Backpropagation

Simulation and backpropagation required no changes at this point to get the game running. Changes were made to the simulation step after looking at results.

### Multithreading

The AI was multithreaded in two ways. The first was creating a separate thread for running MCTS to allow for a better user experience. While AI is running iterations, the game would freeze for long periods of time making the artifact difficult to use. The second was creating threads to run the simulation step. The simulations were found to be the slowest step, so five threads were spun up to improve performance.

A picture containing text

Description automatically generated

Figure 13:Thread diagram

Now one iteration loop changes from Selection -> Expansion -> Simulation -> Backpropagation to Selection -> Expansion -> Post Simulation Job -> Check for completed simulation jobs to backpropagate result. A fear when creating this new process was the main MCTS thread could get too far ahead of the simulation threads. To solve this, the main MCTS thread would wait if ever there were more than ten simulation jobs to run. Also, since the meta data is extremely important during the first few steps when there is no meta data, the first one hundred iterations were set to run without using the simulation threads. The step may not be necessary, but it was a reassuring step.

The complexity of threads caused a multitude of bugs, and the simulation threads themselves did not seem to improve performance too much. There may be better ways of multithreading the AI than what was done.

### Victory Point Nudge

A problem found running the AI was MCTS would be making intelligent moves, but suddenly start making random moves when the game was about to end. The issue is if a player has an overwhelming advantage, all moves appear to give the same outcome of a win or loss. MCTS decides all moves are equal, so it will choose a random move. To solve the problem a very small victory point nudge is added to the UCT formula, so if all else is equal getting more victory point cards should be favored. This also covers the case of MCTS deciding a game is over even if there is a slim chance of victory.

# Results

After implementing the AI as shown above, the MCTS AI was tested against the Big Money AI strategy. Preliminary results showed Big Money winning every match against MCTS, so a test in an optimal situation was done doing 100,000 iterations per move. Previous tests had done 1-10,000 iterations per move with poor results.



Figure 14: MCTS 100,000 iterations versus Big Money

A 20% win rate was a big shock especially considering by the end of each game rough 17 GB of RAM was being used. A higher number of iterations per move meaning a larger tree would become infeasible as the computer to run the tests has 32 GB RAM total. Another interesting fact is that each game averaged 42 minutes with all the time being used by the MCTS AI. A normal game of Dominion should take 30 minutes.

### Lessons Learned

Watching what happened during each game showed MCTS scored all action cards besides the Witch card very poorly. This makes sense since most actions are only useful if they are played in a specific order. If the below cards are in hand, the player should always play the Village card first because it gives more actions to be able to play the Witch. If the Witch is played first, no more actions can be played since there is a limit of one action per turn. Also, it was clear random moves ends the game by exhausting three piles whereas Big Money ends the game by exhausting the Province card pile. The result was MCTS using random moves would buy early Duchy victory cards giving an early advantage, but Big Money would always catch up because MCTS could never end the game fast enough.

A screenshot of a video game

Description automatically generated with low confidence A picture containing text

Description automatically generated

Figure 15: Village Card [5] Figure 16: Witch Card [5]

### RandomPLUS

After watching MCTS random play, RandomPLUS was created to try solving the problems. RandomPLUS is the same as using random moves, but during the action step cards giving extra actions like Village are prioritized over other cards. Also, playing an action is favored over ending the action phase. With just this change the AI was again tested against Big Money.



Figure 17: MCTS RandomPLUS versus Big Money

Only 20,000 iterations per move were done instead 100,000 iterations, but it improved performance from a 20% winrate to 61.5%. This is a massive improvement using only minor game knowledge.

### MCTS Using Big Money

Another test was using Big Money as the simulation step. Using Big Money as the simulation step tries to demonstrate that MCTS can make another AI strategy better.

A picture containing graphical user interface

Description automatically generated

Figure 18: MCTS Big Money versus Big Money

An interesting outcome is doing 1,000 iterations per move MCTS using Big Money performed worse than Big Money by itself. This shows that a certain number of iterations are required before MCTS can have confidence in a move. The doubling of the win rate using 10,000 iterations per move demonstrates MCTS can make an AI better.

### Greedy

Another possible heuristic is a greedy one where the most expensive cards are bought. Greedy assumes that the most expensive cards are the best cards. Greedy is what Jansen and Tollisen used in their AI. [3]

Background pattern

Description automatically generated with medium confidence

Figure MCTS Greedy versus other AI

Greedy crushes Big Money and does well against Double Witch. It also outperforms MCTS using RandomPLUS and goes even with MCTS using Sarasua1. This last test against Sarasua1 is very interesting because it shows that having a stricter but better AI does not necessarily mean it is better for the simulation step. It also may show that Sarasua1 may perform worse with cards it would not normally buy.

### Expand Using Heuristics

Changing the simulation step may tell us if one game state is better than another but changing what nodes we expand can limit the most in the tree to just the good ones. This can reduce the branching factor significantly. Determining what is a good move can be difficult, and there will be cases where the best move is not in the list from the heuristics.

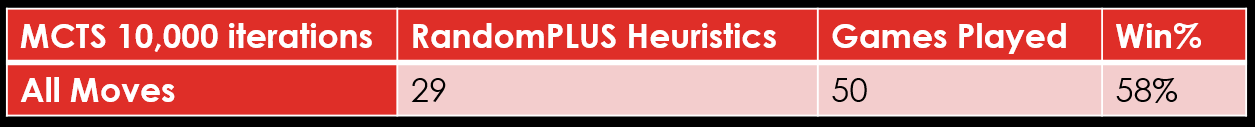


Figure : MCTS RandomPLUS using heuristics versus all moves

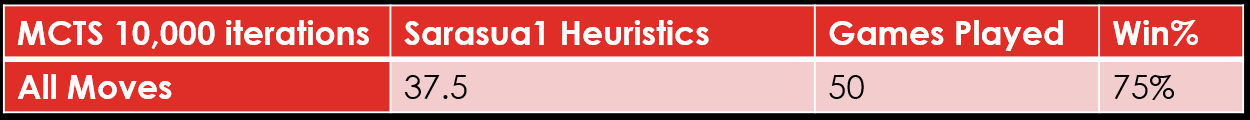


Figure : MCTS Sarasua1 using heuristics versus all moves

Heuristics showed modest improvements using RandomPLUS for simulations, but it was a massive improvement using Sarasua1 for simulations. This shows that some AIs need the reduction of branching more than others.

### Chaos Chance

A feature tested by Jansen and Tollisen was adding a chance of doing a random move instead of the specified heuristic like Big Money. [3] The idea is random moves will make the heuristic more flexible in trying possible outcomes. The results are shown below:

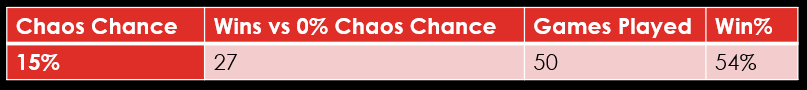


Figure 22: MCTS 15% Chaos Chance versus 0% Chaos Chance

The chaos chance improving performance slightly shows it is a useful feature, but it most likely requires fine tuning for any heuristic.

# Future Work

#### Simulation Step Changes

After gathering results, it became clear that changing the simulation step can cause massive improvements. It would be interesting to try using a genetic algorithm, Markov chain, or other AI strategy.

#### Strategies Versus Moves

Currently the AI searches over individual moves, but it could be changed to search over strategies. This has been already partially done by expanding only over moves given by strategies, but it would allow for a more complicated game to use MCTS. An example is how Prismata combined a Hierarchical Portfolio Search with MCTS to play a game that has an even greater branching factor than Dominion. [6]

#### More Cards

The next level of the AI would be to add in more cards to the game and see how it performs. The power of MCTS is not that it can play a game well but that it can adapt to changes to the game. Adding in the full set of Dominion cards or even expansions of Dominion would show a single AI can play the game even when different strategies are used to win the game.

#### Different Deckbuilding Game

Because the AI does not care or even know that it is playing Dominion, the AI could be changed relatively easily to play a different deckbuilding game. The hardest part would be coding a different deckbuilding game.

# Conclusion

After testing the AI it is clear that pure MCTS is a poor choice for Dominion. MCTS needs some help, so it can more quickly learn the game. This help can come in many different areas. The expansion step can be changed, so only good moves are added to the tree. The simulation step can be changed to use any number of heuristics to play the game that are better than just random moves. It can also be modified to have a chaos chance to avoid issues with the heuristic being too rigid to accurately evaluate a game state. With all these changes, MCTS can be a powerful AI at quickly deciding if a move is good. Because MCTS does not care what it is playing, it can easily change strategies when changes in the game occur.

# 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]. |
| [2] | "Dominion Beginners Guide | Tutorial," UltraBoardGames, [Online]. Available: https://www.ultraboardgames.com/dominion/beginners-guide.php. [Accessed 9 February 2021]. |
| [3] | "Monte Carlo Tree Search: Implementation for Tic-Tac-Toe," Stack Overflow, 22 April 2014. [Online]. Available: https://stackoverflow.com/questions/23803186/monte-carlo-tree-search-implementation-for-tic-tac-toe. [Accessed 9 February 2021]. |
| [4] | L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *Machine Learning: ECML*, 2006. |
| [5] | "How Does Monte Carlo Search Tree work?," Stack Overflow, June 2017. [Online]. Available: https://stackoverflow.com/questions/44230911/how-does-monte-carlo-search-tree-work. [Accessed July 2020]. |
| [6] | "Big Money," 20 October 2018. [Online]. Available: http://wiki.dominionstrategy.com/index.php/Big\_money. [Accessed 9 July 2020]. |
| [7] | "Dominion\_(card\_game)," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Dominion\_(card\_game). [Accessed 9 February 2021]. |
| [8] | "Monte Carlo Tree Search," Wikpedia, [Online]. Available: https://en.wikipedia.org/wiki/Monte\_Carlo\_tree\_search. [Accessed 9 February 2021]. |
| [9] | "Dominion Strategy Wiki," 4 March 2021. [Online]. Available: http://wiki.dominionstrategy.com/index.php/Main\_Page. [Accessed 20 4 2021]. |