An AI for Dominion
Based on Monte-Carlo Methods

by
Jon Vegard Jansen and Robin Tollisen

Supervisors:
Morten Goodwin, Associate Professor, Ph.D
Sondre Glimsdal, Ph.D Fellow

June 2, 2014

Abstract

To the best of our knowledge there exists no Artificial Intelligence (AI) for Dominion which uses Monte Carlo methods, that is competitive on a human level. This thesis presents such an AI, and tests it against some of the top Dominion strategies available. Although in a limited testing environment, the results show that our AI is capable of competing with human players, while keeping processing time per move at an acceptable level for human players. Although the approach for our AI is built on previous knowledge about Upper Confidence Bounds (UCB) and UCB applied to Trees (UCT), an approach for handling the stochastic element of drawing cards is presented, as well as an approach for handling interaction between players. Our best solutions win 87.5% games against moderately experienced human players, and outperforms the successful, rule-based, Dominion strategies SingleWitch and DoubleWitch both with a win percentage of 68.5%.

Keywords: Dominion, UCT, UCB, AI, Multi-Armed Bandit Problem, Monte-Carlo, Tree Search

Contents

1 Introduction **1**
1.1 Scope . . . 2
1.2 Solution Summary . . . 2
1.3 Result Summary . . . 3
1.4 Background . . . 3
1.4.1 Artificial Intelligence (AI) . . . 3
1.4.2 Multi-Armed Bandit Problems . . . 4
1.4.3 Trees as a Data Structure . . . 5
1.4.4 Monte-Carlo Methods . . . 6
1.5 The Game of Dominion . . . 8
1.5.1 Card Mechanics . . . 8
1.5.2 Game Rules . . . 9
1.5.3 Game Turn Example . . . 11
1.5.4 Important Characteristics and Game Concepts . . . 11

2 State of the Art **13**
2.1 AI in Dominion . . . 13
2.1.1 Official Dominion Game Site . . . 13
2.1.2 Artificial Neural Networks based Dominion AI . . . 14
2.1.3 Provincial AI . . . 15
2.2 Monte-Carlo Tree Search (MCTS) . . . 15
2.3 Upper Confidence Bounds applied to Trees (UCT) . . . 16
2.3.1 Selection . . . 16
2.3.2 Expansion . . . 18
2.3.3 Simulation . . . 18
2.3.4 Backpropagation . . . 19
2.4 Exploration versus Exploitation . . . 21
2.4.1 Dynamic C Value . . . 21
2.5 UCT Enhancements . . . 22
2.5.1 PaG enhancement A: State Hashing . . . 23
2.5.2 PaG Enhancement B: Move Pruning . . . 23
2.5.3 Bandit Enhancement A: Thompson Sampling . . . 23
2.5.4 Bandit Enhancement B: UCB1-Tuned . . . 24
2.5.5 Selection Enhancement A: First Play Urgency . . . 25
2.5.6 Selection Enhancement B: Expert Knowledge . . . 25
2.5.7 Selection Enhancement C: Search Seeding . . . 25
2.5.8 Selection Enhancement D: History Heuristics . . . 25
2.5.9 Simulation Enhancement A: Rule Based Simulation . . . 26
2.5.10 Simulation Enhancement B: Machine Learning Simulations 26
2.5.11 Scoring Enhancements: Minimax and Expectimax . . . 26
2.6 Parallelization . . . 26

3 Approach **28**
3.1 Brute Force Search Tree 30
3.2 Dealing with Stochastic Card Draws 33
3.3 Interaction Between Players 35
3.4 UCT Variants: UCT_{orig} and UCT_{mod} 36
3.4.1 Dominion in the Search Tree 37
3.4.2 Propagation and Scoring 38
3.4.3 Rollout 41
3.4.4 Must Play Plus Actions First (MPPAF) 43
3.4.5 Simulations 45
3.4.6 Minimum Visits 47
3.4.7 Parallelization 50
3.5 UCB Variants: UCB_{orig} and UCB_{mod} 52
3.5.1 Simulations 53
3.5.2 Minimum Visits 55
3.5.3 Parallelization 55
3.6 Framework and Implementation 56

4 Experiments **57**
4.1 Unbalanced Kingdom Card 57
4.2 Optimal Playstyle for Single-player 59
4.3 Playing the Variants Against Each Other 60
4.4 MCDomAI versus Random AI 63
4.5 MCDomAI versus SingleWitch 65
4.6 MCDomAI versus DoubleWitch 67
4.7 MCDomAI versus Human Players 71
4.8 Speed: Time per Move 72

5 Conclusion **75**
5.1 Future Work 76

Appendices **80**

A Big Money Finite-State Machine (BMFSM) Algorithm **80**

B SingleWitch and DoubleWitch Algorithm **81**

C Cards Chosen for the Test-Bed **82**
C.1 Action Cards 82
C.2 Treasure Cards 85
C.3 Victory Cards 86
C.4 Curse Cards 87

1 Introduction

The game of Dominion [22] takes place in a medieval era setting, where each player assumes control of their own kingdom or dominion. The players start out with a small amount of resources, and have to choose what is best for their own kingdom, depending on the existing possibilities. Some players act like greedy monarchs, choosing to amass as much gold and money as possible, by employing woodcutters and hosting festivals, while other monarchs consult mystics and witches to spread curse and despair among the other kingdoms. While there are different ways to expand the kingdom and many potential strategies, in the end, the player with the most land (represented by victory cards) wins.

Dominion is played as a card game for two to four players, where each player's kingdom is represented by their own deck of cards. Each player starts with ten cards, and will throughout the game expand their kingdom by adding more cards to their deck. For each game there are different possibilities and resources available, thus on the table there are laid out ten different stacks of kingdom cards, which can be swapped out for different cards between games.

In addition to the ten kingdom cards there are seven more stacks, which are present in every game, and these 17 stacks of cards are referred to as the *supply*. The game is over when three or more card stacks in the supply are empty, and the winner is the one with the most land area, or victory cards, in the card deck (see detailed game rules in Section 1.5.2).

Compared to many of the other games that employ Monte-Carlo methods to create a game Artificial Intelligence (AI), Dominion is notably different. This is because Dominion is both a non-deterministic game and a game of imperfect information. An example of a stochastic game element is the drawing of cards, where each card has a certain probability for being drawn.

Imperfect information denotes that there are elements regarding the game state, which is not known to all players. For instance, the hidden cards each player has in his or her hand is an element of imperfect information. Other games, like *Chess* [35] and *Go* [36], which also use Monte-Carlo methods, do not have such cases where it is not known whether a game piece is in a certain position. This greatly increases the complexity of a Dominion AI, since the AI will have to make decisions without knowing the outcome of each action with full assurance.

From our point of view, there are five main challenges to create a Dominion AI:

1. How to decide which cards to play from hand, and in what order?
2. How to decide which cards to buy?
3. When to stop buying intermediate cards and start buying victory points (VPs)?
4. How to handle interaction between players?
5. How to handle the stochastic element of drawing cards?

The approach is to apply Monte-Carlo methods to take care of the challenges (1-4), and a sampling approach to handle (5).

1.1 Scope

The goal of this thesis was to create the best possible Dominion AI utilizing Monte-Carlo methods. Though the AI may be applicable to the complete game of Dominion, only a specific set of cards was chosen as test-bed. The cards are listed in Appendix C. The performance of the solution is measured by running a series of experiments, where the AI is playing against different opponents and is tested in different scenarios. How well the AI handles these opponents and situations is determining the performance.

1.2 Solution Summary

The solution AI that were created has four different variants, each behaving similarily to each other, but also with some differences. Each variant was given their own name, while their collective term was named MCDomAI. All variants apply either *flat Upper Confidence Bounds* (UCB) or *UCB applied to Trees* (UCT) together with either the UCB1 formula or a modified version, as listed below:

1. UCB_{orig} applies flat UCB, using the UCB1 formula during selection to find the best move.

2. UCB_{mod} uses a modified version of the UCB1 formula, but is otherwise similar to (1).

3. UCT_{orig} finds the best move by using UCT, thus creating a search tree, which is different from (1) and (2). This variant also uses a slightly different propagation system than (3), in order to resemble a Minimax tree search.

4. UCT_{mod} is similar to (3), but uses UCT together with the modified UCB1 formula for selection, as well as a different propagation system.

In addition, some new enhancements are presented in order to deal with the Dominion-specific challenges, such as the stochastic element of drawing cards, where proportional sampling is applied to the UCT approaches.

For the UCT variants, interaction between players is taken care of in the search tree by using a Minimax resembling propagation system. The flat UCB variants however, do not need to take special care of the player interaction, since no search tree is created.

Parallelization is also applied for all AI variants by utilizing root parallelization. This is further supporting the UCT card sampling approach, as well as increasing playing strength and reducing time spent per move.

1.3 Result Summary

For playing against the successful SingleWitch and DoubleWitch Dominion strategies, the best solution was UCB_{orig}, capable of winning 68.5% games against both strategies, while UCT_{mod} had the weakest results, winning in 35% and 31% of the games.

Against moderately experienced human players, UCB_{mod} won in 87.5% games, while UCT_{mod} won in 25% games, which were the only two variants tested in this experiment. Due to time limits, not enough games were played to be conclusive on the performance against human players.

The main configurations for variant UCB_{orig} and UCT_{orig} both use 1.2 seconds on average per move, while UCB_{mod} and UCT_{mod} respectively use 8.4 and 9.1 seconds per move, which can be reduced by increasing the number of threads. For instance will two threads almost cut in half the time used per move, while still achieving the same performance in playing strength.

Due to time limits, a sole variant could not be concluded best, however the flat UCB variants seem to perform better than the UCT variants for this test-bed of cards. However, UCT may perform better when tested with other sets of Dominion cards, as the UCT variants seem to play more cards in sequence than the flat UCB variants. Dominion in general may possibly also be favoring greedy, or locally optimized choices, which flat UCB is good at, instead of planning many turns ahead, which is one the UCT strengths.

MCDomAI is also capable of recognizing good cards in terms of both buying cards and playing them. This is tested by increasing the value of cards beyond reasonable game balance.

1.4 Background

This section aims to provide some background information that will enable the reader to better understand the different elements of MCDomAI. The section expands upon the topics mentioned in the introduction, and covers the basic concepts this thesis builds upon.

The first subsection gives a brief explanation of what an AI is, focusing mainly on AI for board games.

Secondly, multi-armed bandit problems are presented, which are a specific category of problems, also applicable to Dominion, within the field of AI.

Thirdly, we explain how to use trees as a data structure, since search trees are essential to UCT.

The last subject is Monte-Carlo methods, which is a methodology for solving problems. This method includes UCB and UCT, and can be applied to solving multi-armed bandit problems.

1.4.1 Artificial Intelligence (AI)

AI is a branch in computer science that aims to create human-like intelligence in computer programs. While the main focus here is on game AI, specifically board games, there are other interesting fields for AI as well.

AI is becoming more and more widespread, finding uses in almost any field, including physics, construction, mathematics, medical research and many more. Great AIs have been created, most commonly known is probably computer programs playing the game of Chess [35], surpassing the skill of human players. The world-famous Chess-playing computer *Deep Blue* beat world champion Garry Kasparov in 1997 [29].

When using the term AI, we distinguish between simple AIs, referred to as finite-state machine-based AIs (FSM-based AIs) and more advanced AIs employing advanced algorithms such as neural networks, UCB and UCT. The FSM-based AIs usually consist of multiple conditional if-statements, such as the rule-based Dominion strategies MCDomAI plays against in this thesis, see Appendix A and B for full descriptions.

1.4.2 Multi-Armed Bandit Problems

A problem in AI with certain characteristics can be classified as a multi-armed bandit problem. When making a decision between several different options of which the problem is to find the best one, we can consider them as arms on a multi-armed bandit, or slot machines, where each machine has its own probability for winning.

Several techniques have been developed to attempt to find which machine gives the highest reward, and when this abstraction can be applied to a problem, it can be considered a multi-armed bandit problem.

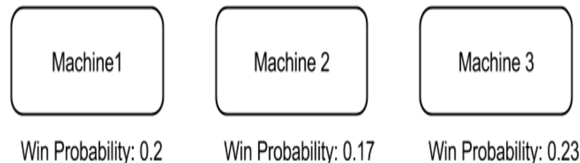

Figure 1: Multi-armed bandits.

Figure 1 illustrates three machines with differing probabilities for winning. Solving a multi-armed bandit problem is to find out which machine that gives the highest reward, when the win probability is not known in advance. Optimally, the problem should be solved in as few tries as possible, to minimize the "money spent", commonly named *regret*, on the sub-optimal, or even losing, slot machines. One of the main problems in such algorithms is to balance the exploration and exploitation. The question is that whenever a good machine is found, should one continue to exploit/spend money on it, or should other machines be explored in an attempt to find a better one?

Several AIs for games have abstracted the game as a multi-armed bandit problem. Notably, the game of Go employs UCT in the world class AI *MoGo*

[25]. Commonly, board games are abstracted to a multi-armed bandit problem by considering each possible move in the current game state as an arm, and attempt to find which one is most likely to give us a favorable outcome, thus win the game.

1.4.3 Trees as a Data Structure

Trees are commonly used as a data structure in computer science. This thesis puts trees in relation to some of the techniques presented later, so this section aims to provide a more general overview. The most relevant use of trees for MCDomAI is UCT, which utilizes trees to represent a Monte-Carlo approach to solve multi-armed bandit problems.

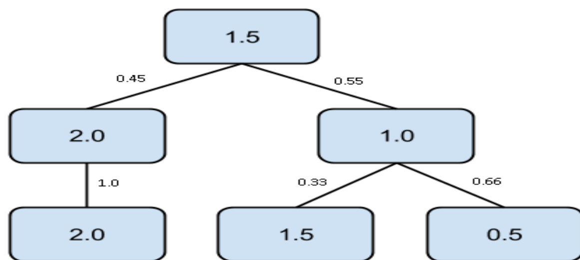

Figure 2: Simple tree example.

As shown in Figure 2, a tree is similar to the trees outdoors, but branches downwards instead of upwards. A tree consists of an amount of *nodes* containing some data. This can be any data, but in Figure 2 the data is a decimal number. Nodes are related through *edges*, as indicated by the lines between nodes. The nodes above are considered the *parent nodes* of the *child nodes* below. Nodes without children are considered *terminal nodes* or *leaf nodes*.

Building a tree can be useful in the case of Dominion to plan sequences of possible moves. The root node on top could contain the current game state, the edges down to the children could represent the currently available moves and the top child nodes could be new states that are changed from the outcome of each move. These child nodes could also have children of their own, each with an associated edge.

Another useful feature is the ability to assign information to the edges between nodes. The information can be used, for instance as probability for which child node to select when traversing the tree, or as some weight value to be used by the parent node.

Section 3.4.1 describes how UCT utilizes trees in detail, and Section 3.2 describes how we attempted to utilize edge information to handle the stochastic element of drawing cards.

1.4.4 Monte-Carlo Methods

Monte-Carlo methods are a part of so called experimental mathematics, where results are inferred based on observations [27]. When applied for computational algorithms, Monte-Carlo methods utilize the speed of computers to run many simulations, and make a conclusion based on the outcomes of each simulation. The following list of steps summarizes the Monte-Carlo approach:

1. Define the problem with possible outcomes.
2. Generate (simulate) random outcomes for the problem.
3. Perform a deterministic computation on the results.
4. Aggregate and infer a solution.

By performing Monte-Carlo methods for bandit problems, we can create probabilistic models, and then infer which arm that will maximize the reward.

When implementing this technique in a game AI, each possible move is considered as an arm. Then, simulated games are run for each arm, and an estimated score value is given for each option. The best move is the one with the most visits or the best score, depending on the application.

We will illustrate this, by summarizing how each step can be applied to determine good moves in Dominion with flat UCB.

1. Problem: Determine the currently best move for Dominion. Possible outcomes: Move results in a game win or a game loss.
2. Simulate full games from each possible move, according to a rollout policy, to see if games were won or lost.
3. Score each move as an average of each move's simulated games.
4. Best move is the one with the highest average score (or highest number of visits if a selection formula is used).

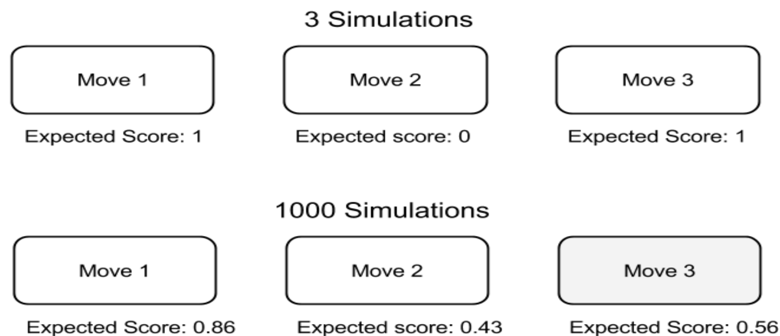

Figure 3: Solving multi-armed bandits with UCB.

Figure 3 shows a summary of how Monte-Carlo methods are employed when solving a multi-armed bandit problem. This example is a representation of how Monte-Carlo methods are used for flat UCB. The UCT variant is similar, and explained in Section 2.2. Each possible option is represented by a node, which can be considered a slot machine. By simulating games from that node (pulling the arm), we can use the values returned by these simulated games to estimate how good that arm is.

UCB and UCT are Monte-Carlo methods used to solve multi-armed bandit problems. Both methods use UCB1, in Equation 1, as a selection formula to select the most appropriate option to simulate, while balancing exploration and exploitation.

By simulating and scoring all options at least once, we can then perform more simulations to receive a more accurate value for each option. By tweaking the exploration constant in the UCB formula, exploration or exploitation can be weighed more heavily, as discussed in Section 2.2.

Monte-Carlo methods have also been extensively employed in other fields, such as operational research and nuclear physics as early as in the 1960s, with the first known implementations in the 1940s [27, 30]. With the exponential growth in computing power, Monte-Carlo methods are now applied more frequently, such as for creating board game AIs, where computers run hundreds of thousands of simulated games before deciding upon each move.

1.5 The Game of Dominion

Dominion uses a set of kingdom cards that is used in addition to the basic supply. One can mix and match these kingdom cards, as there are many cards to choose from. For this thesis however, we have chosen a set of ten cards, which is used as a test-bed for all experiments. There are mainly four reason why we have chosen these ten cards:

1. Some of the cards synergize well with each other, and should test our AI in utilizing them together.

2. The set of cards contains some interaction between players.

3. There are many potential strategies present.

4. The cards were fairly easy to implement.

A picture of the cards is displayed in Figure 4, with full descriptions available in Appendix C. Section 1.5.1 explains the default mechanics for most of the cards.

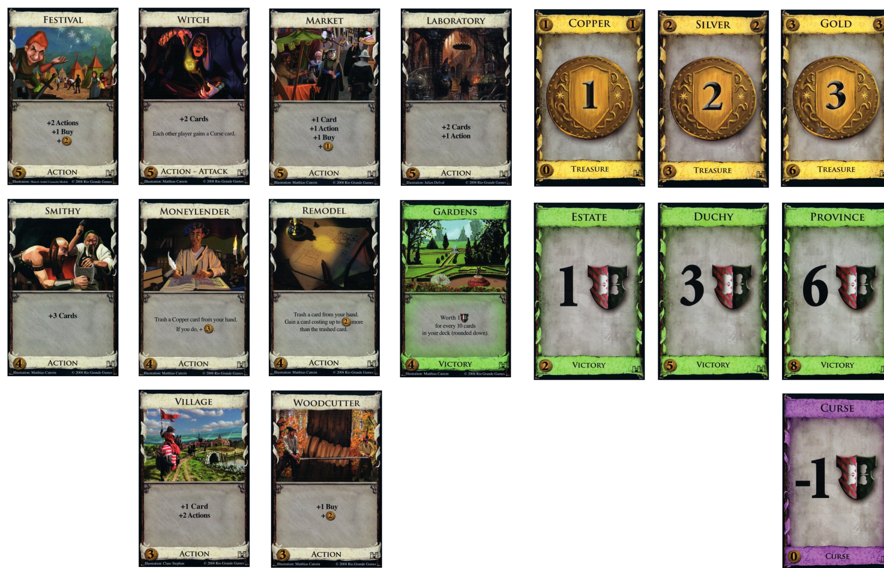

Figure 4: The cards in Dominion, using the test-bed cards, see Appendix C for a full description of each card. [3, 15]

1.5.1 Card Mechanics

Cards in dominion vary a lot, but there are 4 core mechanics which are usually present in cards.

Figure 5: Village: +1 Card, +2 Actions; Woodcutter: +1 Buy, +2 Coins.

The cards in Figure 5, Village and Woodcutter, contain all of these 4 core mechanics.

- +X Card(s) - The player who plays this card can draw X card(s) from his or her deck.
- +X Action(s) - When played, grants the player X additional action(s), which means that he or she can play X more action cards that turn.
- +X Buy(s) - Grants the player additional buys. The number of buys are equal to the number of cards that can be bought during the buy phase.
- +Coins - As seen on the Woodcutter card in Figure 5 there is a plus (+) sign and a coin with the number '2' printed inside. This means that the player has two extra coins to spend on cards during the buy phase.

There are other mechanics as well, because cards can have sepcific text printed on them, explaining their unique mechanics. An example is the Witch card, which makes all other players gain a Curse card, when played.

1.5.2 Game Rules

In this section follows a summary of the game rules. The complete and offical game rules can be found in online [40].

Dominion is initially set up with the different cards from Figure Figure 4 spread out on a table. Each card comes with many copies to be placed as a stack of card. For a two-player game, each stack of victory cards has initially eight cards, while the action and Curse stacks have ten in each stack. The treasure

cards are usually abundant, with respectively 44, 30 and 20 cards for Copper, Silver and Gold card stacks.

Each player starts with a deck of seven Copper cards and three Estate cards, and should expand the deck by acquiring more cards from the supply as the game progresses.

Players also have their own discard pile, which contains all the previously played and bought cards. Whenever the player needs to draw a card from deck, and the deck is empty, the discard pile is shuffled and made into the player's deck, allowing cards to be reused over and over.

The goal of the game is to have more VPs than the opponent player at the end of the game. VPs are achieved by buying victory cards, which are worth the printed amount of VPs. There are four types of cards in the game:

1. **Victory cards**, representing land area by a number of VPs.
2. **Treasure cards**, played to give coins used to purchase other cards.
3. **Action cards**, doing a variety of things, such as giving more coins or drawing more cards into hand.
4. **Curse cards**, which are worth -1 VP. Some action cards, such as Witch, distribute Curse cards to other players, while the Remodel card can be used to remove (trash) Curse cards.

The starting player is picked at random, and the game progresses by each player taking their turn, continuing clockwise around the table. In successive games the starting player can be the one positioned on the left side of the previous winner, as the starting player has a small advantage, as shown in the experiments in Section 4.3, 4.5 and 4.6, as well as in another paper [21].

A game turn consists of three phases:

1. **Action phase**, where the player starts out with one action, thus enabling the play of one action card. Playing a card is done by putting it from hand onto the table. This card may however give +X actions, allowing play of X more action cards. After all actions or action cards have been used, or the player does not want to play the remaining action cards in hand, the player enters the next phase.
2. **Buy phase**, where the player can play any number of treasure cards from hand. The player is granted one buy, allowing the player to buy one card from all the coins gathered from both the played action cards (if applicable) and treasure cards . If the player played an action card giving +X buys (such as the Woodcutter card), then the player can split the total money, and buy multiple cards. All bought cards are usually put into the player's discard pile, which is later reshuffled as the player's deck.
3. **Clean-up phase**, which is simply to put all cards played, and the rest of the cards from hand into the discard pile, no cards are saved on hand for

the next turn. Then the player draws five new cards from deck into hand, shuffling the discard pile if necessary. Note that the treasure cards used to buy cards are not leaving the player, but put into the discard pile to be reused later.

The game ends when either all the Province cards have been bought, or three other supply piles are empty. The winner is then the player with the most VPs acquired throughout the game.

If the highest scoring players have equal amounts of VPs, then the player with the least turns taken wins, but in cases where they have the same number of turns, the game is tied between them.

1.5.3 Game Turn Example

For reasons of clarity, a game turn example is presented:

Anna is taking her turn against Frank, and has a Village card, a Woodcutter card, two Copper cards and an Estate card in hand.

During her action phase she can either play the Woodcutter or the Village card. She chooses to play the Village card, first giving +1 card, so she draws the top card from deck into hand, and sees that it is a Silver card. The Village card also gives +2 actions, so now she can also play the Woodcutter card. When she does this she gains +1 buy and +2 coins for the buy phase. She still has one more action, but since she has no more action cards, her action phase is finished, and she enters the buy phase.

In the buy phase she already has +1 buy and +2 coins from the action phase, in addition to the one buy she always receives during buy phase, for a total of two buys and two coins. She can now play the treasure cards from her hand, and she chooses to play them all (this happens in most cases when playing the standard game of Dominion). She had two Copper cards and one Silver card to play, giving four more coins in addition to the two coins from the Woodcutter card.

Now she has a total of two buys and six coins, where she can choose to buy a multitude of card combinations, by either picking one card costing up to six, or two cards costing up to a total of six, or no cards at all. Even though she can buy two Silver cards for instance, she chooses to buy one Gold card instead. She then puts the Gold card into her discard pile and enters the clean-up phase.

In the clean-up phase she puts all the played cards into the discard pile, as well as the Estate card from hand, and draws five new cards from deck. However, she only has two cards left in the deck, so she draws those two, then shuffles the discard pile, puts the discard pile where the deck was, and then draws three more cards. Her turn is now complete, and the next player takes his turn.

1.5.4 Important Characteristics and Game Concepts

According to the list of attributes for combinatorial games, Dominion can be categorized as follows [6]:

- Zero-sum: At least when there are only two players, because moves that make one player win, will make the other player lose. Some cards not in this test-bed may have positive side-effects for the other player as well though.
- Imperfect information: Since the cards on a player's hand are not visible to the other player. What the other player has bought is not directly visible, but can be memorized as all moves made are visible to the public.
- Non-deterministic: Due to the stochastic element of drawing cards, but the stochastic element is usually more proportional than flipping a coin. Although the drawn cards reflect what the player bought on earlier turns, it is possible to lose a game with an inferior deck, due to unlucky card draws. In order to flatten out the stochastic element, most of the experiments test their settings over many games.
- Sequential: This is a turn-based card game, where each player takes its turn before the other.
- Discrete: All moves are discrete, and can be considered separate.

The way we see it, mastering the game of Dominion as a human player is difficult mainly because of the following two reasons:

1. The wide variety of different kingdom cards: Requires players to familiarize themselves with new game setups, applying general knowledge to new situations, in order to buy and play cards that synergize well.
2. The dynamic game duration: Requires players to time when to start buying victory cards over other cards.

Other than being worth VPs, the victory cards have no other function throughout the game, thus it is wise to not buy these too early in the game, since they take up a space in your hand. Since it is hard to determine when the game is going to end, choosing when to buy victory cards over other cards can be difficult. This and the fact that the kingdom cards in supply are often replaced between games, make the game of Dominion more dynamic and difficult to master, requiring players to apply general Dominion knowledge to new setups.

Strategies in Dominion often first buy cards that will enable the player to buy VPs, and then at a later point start utilizing the built deck to buy VPs.

As a parallel to the exploration versus exploitation, there is first a phase where one is required to explore or gather the necessary pieces, either information or cards, before exploiting this to achieve the highest gain or score available.

Note that the wide variety of different kingdom cards is not extensively tested further in this thesis, as the testing environment uses only a specific set of kingdom cards. However, as shown by the experiment in Section 4.1, MCDomAI should still perform well when implementing more cards.

2 State of the Art

This section presents the current state of the art in the areas this thesis contains.

First, we will look at the current AIs Dominion, which are to the best of our knowledge limited to a single scientific paper [21], and a few AIs with no corresponding papers [16, 20].

Secondly, Monte-Carlo and UCT are presented, and UCT is explained into detail.

Finally we will look at different implementations and enhancements for the UCT algorithm.

2.1 AI in Dominion

There has not been done a lot of academic work in the area of Dominion AIs. To the best of our knowledge, there is only one AI by Fynbo et al. [21], which utilizes artificial neural networks. There are two other Dominion AIs available on two web pages [16, 20], but they are without any academic publications. Having looked at them and achieved a fair grasp of how they work, they are also presented.

2.1.1 Official Dominion Game Site

The FSM-based AIs used at the official Dominion game site [16] have no corresponding papers. The site contains several FSM-based AIs, which all appear to employ some heuristic and rule-based tactics. This means that they are likely a series of conditional actions. These are relatively simple, but we were unable to obtain the exact rules they use. For instance, the *Banker* FSM may behave according to: "Buy the most expensive treasure or victory card possible". A more specific example is given in Figure 6.

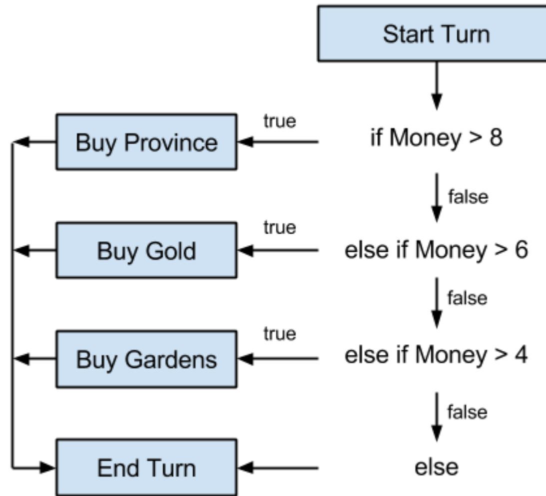

Figure 6: Example of a simple rule based FSM.

FSM-based AIs are used later as test opponents for our AI. While we do not have the exact rules used for the FSM-based AIs on the official dominion site, we have created similar ones for testing purposes.

2.1.2 Artificial Neural Networks based Dominion AI

The only academic paper we were able to find on Dominion AI is written by Fynbo and Nelleman, which describes a Dominion AI that utilizes artificial neural networks trained to play Dominion [21].

Their AI achieves approximately a 48% win percentage versus a rule based FSM, and 60% win percentage against a human player. They do unfortunately not describe the rule-based FSM in detail, so we were unable to test MCDomAI against it.

One weakness of the AI they present is that according to themselves it is not extendable to the full Dominion game, as the technique chosen is not applicable as the complexity rises, at least not on current hardware [21].

Another weakness is the inability to achieve good win percentage against FSM-based AIs [21]. Using neural networks as an approach for a Dominion AI seems to be an inefficient method overall, but as they mention, the AI might win more games when more evolutionary runs are used [21]. However, with the available hardware, the evolved tactics appear to be worse than that of FSM-based AIs.

For MCDomAI, we have chosen an approach which should be applicable regardless of the cards in setup, being able to expand for all Dominion cards, and not just the set of cards in the test-bed.

2.1.3 Provincial AI

The Provincial Dominion AI is published on a web page at Matt's Webcorner [20]. The Provincial AI generates strategies to play by, using a genetic algorithm. There are some prior strategies, which are known to be good, used to develop new strategies. New strategies are played against the old ones, and the strongest ones replace weaker ones.

Provincial provides no statistics on performance against other AIs or humans, although it claims that: "Overall Provincial is quite a powerful AI that develops strategies fairly similar to those used by very experienced players." [20].

2.2 Monte-Carlo Tree Search (MCTS)

This section presents the state of the art of MCTS, and more specifically the UCT implementation.

MCTS appeared in different versions in 2006 [9], where the UCT variant was proposed by Kocsis and Szepesvri [32]. UCT is MCTS using any UCB selection formula, such as UCB1 in Equation 1. When using the UCB1 formula, UCT is often referred to as plain UCT, but since we use different selection formulas, we always use the term UCT in this thesis [6].

$$v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} \tag{1}$$

where v_i is the value of the currently evaluated node, C is the exploration constant, n_p is how many times the parent of node i has been visited and n_i is the number of times node i has been visited.

The UCB1 formula used for UCT is slightly different than the original formula proposed, as shown in Equation 2 [1]. The difference between the two formulas is the inclusion of the exploration and exploitation term, C. The original formula in Equation 2 is always using $\sqrt{2}$ as value for C, while the formula in Equation 1 is more flexible, allowing this value to be manually set depending on the application.

$$v_i + \sqrt{\frac{2 \ln n_p}{n_i}} \tag{2}$$

where v_i is the value of the currently evaluated node, $n_p$is how many times the parent of node i has been visited and n_i is the number of times node i has been visited.

After the original was proposed, the UCB1 formula has been subject to changes and improvements, such as UCB1-Tuned and UCB Improved [25, 2]. Although UCT is an extension of flat UCB, UCT was shown to sometimes be overly optimistic compared to flat UCB [13].

The outline of the UCT algorithm is given in Figure 7 and explained in Section 2.3 [32].

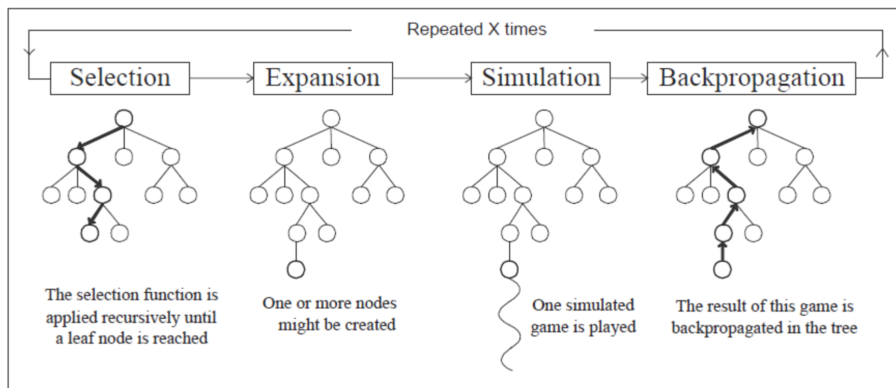

Figure 7: The flow of the UCT algorithm [9].

UCT after being introduced by Kocsis and Szepesvri has been applied to AIs in many games, the most notable research done for the game Go, but some work has been done on games more relevant to Dominion, most notably *Magic: The Gathering* (M:TG) [12] and *The Settlers of Catan* (Settlers) [26].

M:TG is a card game quite similar to Dominion in nature, containing the same stochastic elements relating to card draws [41]. They considered utilizing UCT, but decided to utilize flat UCB due to the stochastic nature making tree creation difficult. Although in a later paper, they apply MCTS with Ensemble Determinization with some success [14].

Szita et al. apply MCTS to Settlers [38], however they are unable to achieve high performance against human players. Still, their work shows that MCTS is both applicable for Settlers, and they also claim that "MCTS is a suitable tool for achieving a strong Settlers of Catan player" [38].

2.3 Upper Confidence Bounds applied to Trees (UCT)

This section explains UCT in detail, describing each of the four steps in its own section. The steps are equal to those in Figure 7. This shows how UCT works in detail, and should also give an indication for how MCTS works in general. The algorithm is described according to some of the academic papers [32, 9].

Before these steps, for the sake of example, we assume that a root node is created along with a child for each possible move. The root node is actually the only node created during initialization, then the selection process expands the initial child nodes as well.

2.3.1 Selection

Selection is what enables UCT to be such a successful method for MCTS. The selection is done by the UCB1 algorithm, as shown in equation 1, to determine which child node to select for expansion.

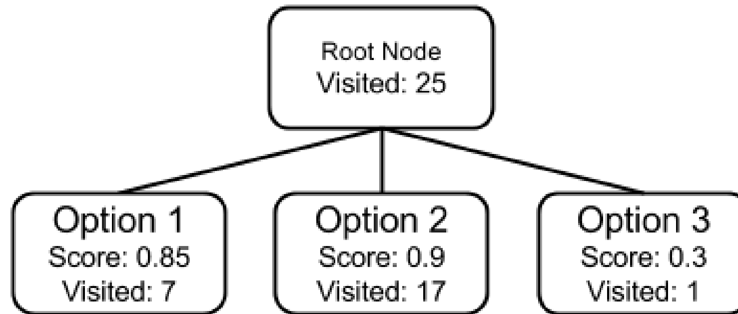

Figure 8: Example excerpt of a UCT tree.

Consider the tree in Figure 8. When the values are inserted into the UCB1 equation an option is selected according to the following values. Here C is set to 0.3 as the exploration constant.

1. $0.85 + 0.3 \times \sqrt{\frac{\ln 25}{7}} = 1.05$
2. $0.9 + 0.3 \times \sqrt{\frac{\ln 25}{17}} = 1.03$
3. $0.3 + 0.3 \times \sqrt{\frac{\ln 25}{1}} = 0.84$

The selected node would be option 1, as it maximizes the UCB1 equation. An important term mentioned a lot throughout this thesis is exploration and exploitation. By modifying C in the Equation 1 different options will be selected. The following example uses the same nodes, but with C valued at 1.0, which would mean more exploration and less exploitation.

1. $0.85 + 1.0 \times \sqrt{\frac{\ln 25}{7}} = 1.53$
2. $0.9 + 1.0 \times \sqrt{\frac{\ln 25}{17}} = 1.34$
3. $0.3 + 1.0 \times \sqrt{\frac{\ln 25}{1}} = 2.09$

As seen in the above results, the algorithm would now select option 3. By setting the exploration constant higher, we will be able to explore currently weaker options more, as opposed to attempting to exploit the ones which seem strong right now. Balancing exploration and exploitation is extremely important for node selection in UCT, as well as in all multi-armed bandit problems. If the exploration constant is set too low we might exclude good options too early based on unlucky simulations.

After selecting this node, UCT would then perform the selection process on the children of the selected node. Option 1 would virtually be placed where the root node is in Figure 8 with child nodes below.

We continue this selection process until a leaf node or a node with unvisited children is reached.

2.3.2 Expansion

After having selected the appropriate leaf node in the current tree comes the expansion phase. This phase adds a single new node containing a unique option below the selected node.

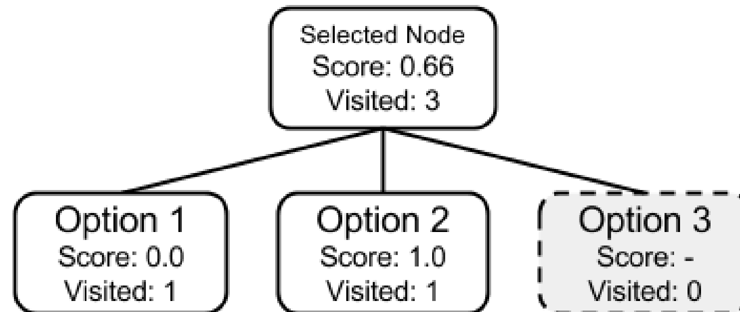

Figure 9: Expanding the tree to contain a new node. The "Option 3" node from Figure 8 is now the "Selected Node".

Figure 9 shows that the selected node had two visited and one unvisited child options. The unvisited node is added to the tree, and is shown as option 3 in Figure 9.

2.3.3 Simulation

When a new node is added, UCT simulates a game below it. The game is simulated according to a defined rollout policy, which can range from completely random to using more heuristics and rules. Using this policy, the game is simulated until completion, which is then scored appropriately. Although the simulation may stop before the game ends, it becomes more difficult to score the state at which we stop in.

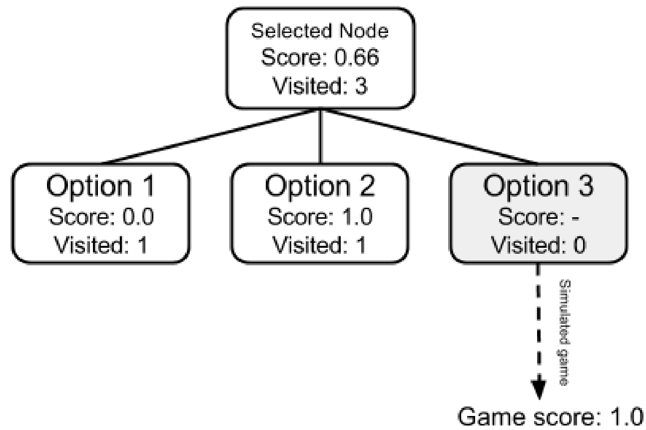

Figure 10: Simulating a game.

The strength of a UCT algorithm is directly connected to the amount of simulations run. If an infinite amount of simulations were used, a random rollout should be converging towards the best move, however utilizing a more deterministic rollout policy could increase performance drastically, by not needing as many simulations to find the best move. A common policy used is *greedy*, where the game is simulated by each play choosing what gives the most benefit for the next turn only.

For a Dominion AI, the most greedy option can be hard determine, but buying the most expensive card affordable, or play the most expensive action card on hand could be a greedy

Games are simulated until they end, and then scored according to some score system. Commonly used for games is a win/loss score system, which gives an amount of points for winning the game, and a given amount for a losing, for instance one and zero respectively. Figure 10 implies a won game for option 2 scored with a 1.0, as opposed to the lost game for option 1, which receives a score of 0.0.

2.3.4 Backpropagation

Finally we backpropagate the result to parent nodes in the tree, which then calculates the value into its own value, using for instance the average score of all simulation for self and all children. Another approach is to use Max, which only uses the value of the best node [9].

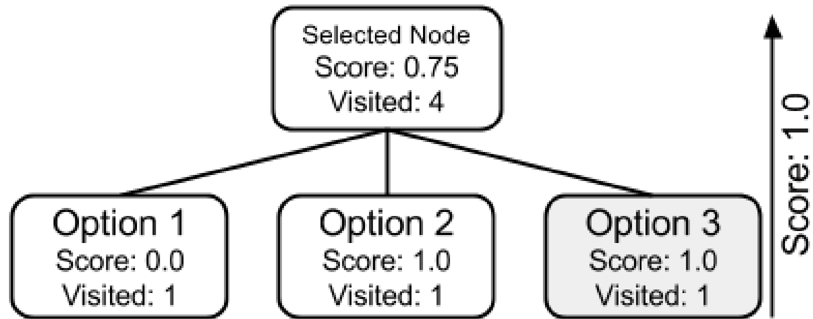

Figure 11: Backpropagating the score from the simulation.

The score from the simulated game is set as the value for the node that was visited for the first time. Then, the score is backpropagated to the parent, which updates its value and visited number accordingly, as shown in Figure 11. The parent node will then backpropagate it to its own parent, until the root node of the tree is reached.

Now one loop of the UCT algorithm is complete.

The process, starting with selection, is then repeated, until a limit is reached, such as a maximum number of simulations or time limit. The best move can then be inferred from the results.

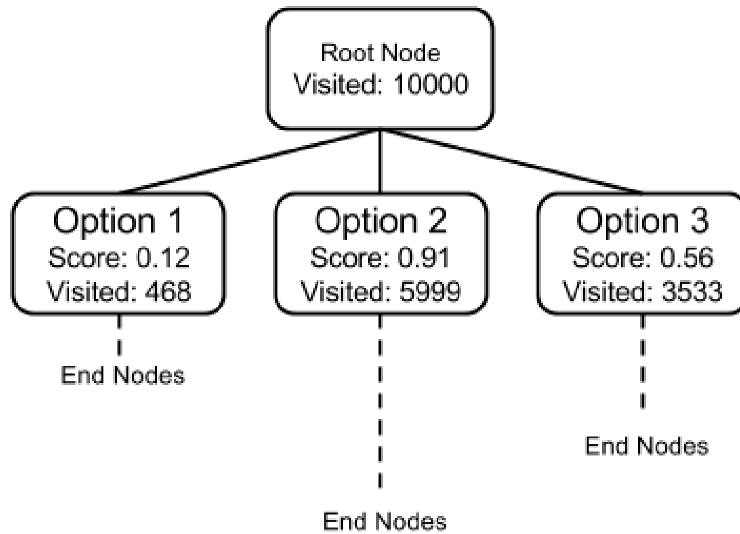

Figure 12: A possible final result after UCT has run its course. The stippled lines represent tree nodes below each option.

As shown in Figure 12, the size of the tree varies, which is due to the explo-

ration and exploitation constant in the UCB1 equation. The tree below option 2 is larger, as it has had more visits. In this case it will have been in the expansion phase 5998 times, thus having 5998 nodes below it, since the first visit was the simulation when it was created. The others have less nodes, 467 and 3532 respectively, as seen by the visited count.

From this UCT can infer that option 2 is likely the better of the three, and thus choose to play option 2 this turn.

2.4 Exploration versus Exploitation

One of the largest challenges with multi-armed bandits and UCT is to balance exploration and exploitation. The UCB1 formula for selecting which node to expand in UCT is given in Equation 1, adding the parameter C that can be tweaked to balance exploration and exploitation.

The UCB1 formula ensures that even though a node has a better score than its siblings, the others will still be visited after a while. Selecting a good value for C is crucial to ensure that UCT works well, and this section talks about some papers discussing different techniques to handle this.

The number of simulations used helps explore the tree in multiple branches, depending on the exploration and exploitation balancing. One way of putting it is that the number of simulations is the search credits available for both exploration and exploitation, and one has to balance this to receive the maximum intelligence from the available simulations.

2.4.1 Dynamic C Value

A paper on methods of MCTS shows, among other things, how one can have a C which modifies itself while the algorithm is running [33]. The two following approaches to having a dynamic C are presented:

1. Time based Decay
2. Variance based Decay

Time based decay was tested with the formula in Equation 3.

$$c = max(0.01, min(0.25, \frac{k}{\sqrt{n}}) \tag{3}$$

where n is the number of visits, and k is a constant to fit your score system. The minimum and maximum values 0.01 and 0.25 may also differ depending on your score system.

This approach attempts to handle the problem with initial scores being less accurate and improving as you get more runs to average from and as such less exploration is needed as time passes.

Variance based was tested using the formula in Equation 4.

$$c = max(0.01, min(0.25, k(\frac{1}{n}\sum_{t=1}^{n} X_i^2) - \overline{X}^2) \tag{4}$$

where k is again a constant preset to fit your scores, n is number of visits, $\overline{x}^2$ is the average results of rollouts, and X_i is the result of the ith rollout.

This is to some extent what UCB1-Tuned, in Equation 6, attempts to do. This let the runs be more evenly distributed, if the score variance is large. Options that might receive an initial score which is much worse than what it would normally average to, is then run again, which might not happen if the C was set to a static value.

The results showed some success, however the conclusion is that the results vary too much, and are not feasible to use in the long run.

2.5 UCT Enhancements

Some papers on UCT attempt to make their own enhancements to improve on their specific domain implementation [6]. This section explores some of these enhancements. The survey on MCTS techniques divides enhancements into two groups [6]:

1. Domain Independent
2. Domain Dependent

The difference between the two is that domain independent enhancements are general enhancements for any application of MCTS, while domain dependent enhancements are created specifically for a certain domain or game. The following enhancements are all domain independent, as they hold the most interest for us when considering whether to apply them to our Dominion implementation of UCT. We further divide the enhancements into the following categories.

1. Performance and General Enhancements. These enhancements generally help achieve better performance in terms of speed. This will enable the algorithm to run more simulations per move, and as such improve performance in terms of play accuracy as well.

2. Bandit Enhancements: There are many different functions to select which node to expand from. By replacing or modifying the UCB1 function you get different levels of exploration and exploitation.

3. Selection Enhancements: Another common thing to do is to alter the way the tree is explored through other means than modifying the bandit function. Some of the most common ones are presented, two of which we have utilized versions of for MCDomAI.

4. Simulation Enhancements: These are approaches to enhance the rollout policy used during simulations. The initial method of simulating in MCTS was to select a random option from the available ones, until end conditions are met. This can be greatly improved to give a more accurate score with less simulations. For flat UCB, random rollouts may also result in non-optimal moves being chosen incertain situations [5]. It is thus normal for implementations of UCT to enhance the rollout algorithm.

5. Scoring Enhancements: Improves how each simulated game is scored, and possibly propagated up through the search tree.

2.5.1 PaG enhancement A: State Hashing

When building large trees for games, such as Chess and Go, the same position of pieces may occur more than once throughout a single game. Running all simulations from the original game could become a black hole, as the game would be stuck in a repeating loop. By hashing the current game state, and comparing hashes, the algorithm can identify equal states, and when a state is reached which already exists, it can point to the first of the equal nodes and continue expanding from there. For the basic set of Dominion, this can however not happen since all moves are happening in a forward direction, with the exception of not taking an action at all, which would not cause the player to win, unless other players are also making very poor choices.

2.5.2 PaG Enhancement B: Move Pruning

Removal of bad options in the UCT tree to prevent wasting simulations on them is another commonly used technique. This can improve performance by enabling more exploitation. Several approaches have been taken to move pruning, and techniques have been developed for MCTS specifically. A survey divides these into two categories; soft pruning of moves and hard pruning of moves [6]. Soft pruning are approaches where the pruned moves can be revisited later if they prove to be beneficial, while hard pruning means complete removal of the options. The main problem with pruning is to accurately evaluate the states to not remove moves which may be good, and preferably not keep bad moves.

There are several approaches to move pruning:

1. Progressive Unpruning/Widening Pruning [10]
2. Absolute and Relative Pruning [28]
3. Pruning with Domain Knowledge [28]

The first approach, progressive unpruning/widening, is a heuristic technique which soft prunes based on previous knowledge.

The second approach, Absolute and Relative pruning, are techniques where one excludes options which no longer have the chance of becoming the most visited option. Relative pruning has an estimated number of visits before the other options are pruned, while absolute pruning waits until it is certain that no other moves can overtake it.

The third approach, Pruning with domain knowledge, means to remove options which are known to be poor, using expert knowledge of the field.

2.5.3 Bandit Enhancement A: Thompson Sampling

Thompson sampling was proposed in the 1933, and has recently risen in popularity for use in bandit algorithms and computer intelligence [39]. When performing

Thompson sampling, a beta distribution is created, based on the amount of tries and successes for each choice.

$$Beta(p1 + successes, p2 + tries - successes) \quad (5)$$

where p1 and p2 are some priors selected in advance.

A random variable is then selected from the beta distribution in Equation 5, which is further used as the value for the option. The option which has the highest value returned is the one selected.

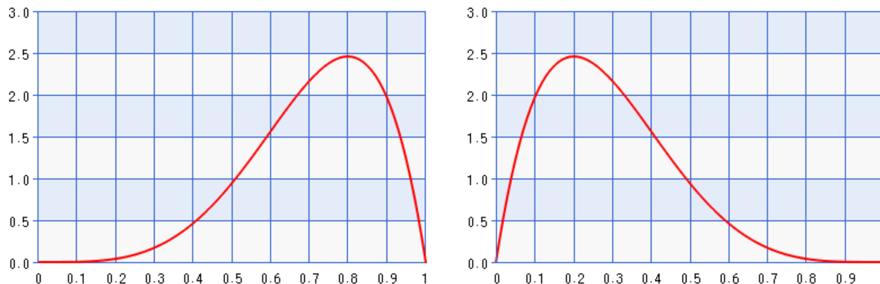

Figure 13: Two sample beta distributions.

The graphs in Figure 13 are two examples of Beta distributions. The left graph shows an option with five tries and four successes (wins), while the graph to the right has five tries and one successful run. As can be seen, the graph to the left with four successful runs has a far higher chance to pick a high number, than the graph to the right with only one successful run.

2.5.4 Bandit Enhancement B: UCB1-Tuned

UCB1-Tuned, in Equation 6 and 7 is a version of the original UCB1 algorithm proposed [1, 25]. UCB1-Tuned is showed to be performing better than UCB1 in Go, by taking the variance of the empirical values into consideration before selecting a node to expand. This UCB1 variant replaces the upper confidence bound with a variance variable.

$$v_i + C \times \sqrt{\frac{\ln n_p}{n_i} \times min\left\{\frac{1}{4}, V_i(n_i)\right\}} \quad (6)$$

where the first part is the same as in Equation 1, and $V_i(n_i)$ part is variance variable in Equation 7, which equals to

$$V_i(n_i) = \left(\frac{1}{n_i}\sum_{t=1}^{n_i} R_{i,t,j}^2 - v_i^2 + \sqrt{\frac{2\ln n_p}{n_i}}\right) \quad (7)$$

where the variables are the same as in Equation 1 and the new variable $R^2_{i,t,j}$ is the tth payoff in node i for the player j. Which expresses the estimated upper variance bound of machine i. In [9] this performs better than UCB1 in all their tests.

2.5.5 Selection Enhancement A: First Play Urgency

First Play Urgency (FPU) is a modification intended to enable exploitation further down the search tree [25]. It is based on the principle of not changing a winning tactic. When an arm on the multi-armed bandit is winning, there is no reason to explore other options unless the arm starts losing. As the tree grows proportionally as the algorithm gets deeper, there will be less simulations available per option, this is intended to utilize the limited simulations better.

2.5.6 Selection Enhancement B: Expert Knowledge

Supplying expert knowledge for the UCT algorithm to use is commonly done when UCT is used as a game AI [23, 41, 6]. This can be supplied as an opening book, which is a collection of fixed moves early in the game which is known to be optimal or close to optimal, or as heuristics and rules. MCDomAI utilizes some expert knowledge in the form of heuristics and rules to improve rollout accuracy, time per move and general play quality.

2.5.7 Selection Enhancement C: Search Seeding

Search seeding is to initialize the nodes in the UCT tree with values based on some heuristics. This can reduce the need for exploration of nodes which are already known to be poor choices. Gelly et al. generate prior data for their Go AI, which improved play notably [24].

2.5.8 Selection Enhancement D: History Heuristics

Storing information from previous simulations can also be beneficial, as it enables the UCT algorithm to more accurately judge where to explore or exploit considering how these moves performed previously. History heuristics is based on the same technique as used in $\alpha\beta$ tree search [33].

Kozelek differentiates between utilizing history heuristics on a tree-playout level and a tree-tree level [33]. Tree-playout level is to utilize history heuristics to tune simulation policy, while tree-tree level is to utilize it to improve action selection. Kozelek shows in experiments that utilizing tree-tree level history heuristics significantly improves performance compared to no history heuristics, it is however not measured up against tree-playout level history heuristics. Finnsson also describes significant improvements when utilizing history heuristics, while it is unclear exactly what heuristics he gathers, he states that history heuristics are better with 99% statistical significance [17].

2.5.9 Simulation Enhancement A: Rule Based Simulation

By providing expert knowledge as rules for the simulations run during rollout, one can greatly improve the accuracy, with virtually no loss of performance in time per move.

2.5.10 Simulation Enhancement B: Machine Learning Simulations

Some research has also been done on having the rollout algorithm learn by experience which options to play, based on previous options. There are two different related techniques which are applicable in general:

1. Move-Average Sampling Technique (MAST) creates a table for each option containing updated reward values for each option. When in the rollout phase these values are used to guide option selection towards the better moves by using a Gibbs distribution [19].
2. Predicate-Average Sampling Technique (PAST), which is related to MAST, the difference being that PAST store different predicates paired with actions [18].

2.5.11 Scoring Enhancements: Minimax and Expectimax

When scoring games based on their rollouts, it is important to score them in such a way that UCT will make good choices. This is especially important when dealing with more than one player, having to consider their score against your own.

- Minimax - Attempting to maximize your own score, while minimizing your opponents, and assuming they will do the same [6].
- Expectimax - For stochastic games, where the value for each node will be a weighted value based on child nodes and the probability that it will occur [6].

2.6 Parallelization

There has been done some of work on the subject of multi-threading UCT trees, [4, 7, 8, 11, 34]. We have chosen to focus on the three following, as they seemed to be the most used ones:

1. Root Parallelization or Single-Run Parallelization
2. Tree Parallelization or Multiple-Runs Parallelization
3. Leaf Parallelization or At-the-leaves Parallelization

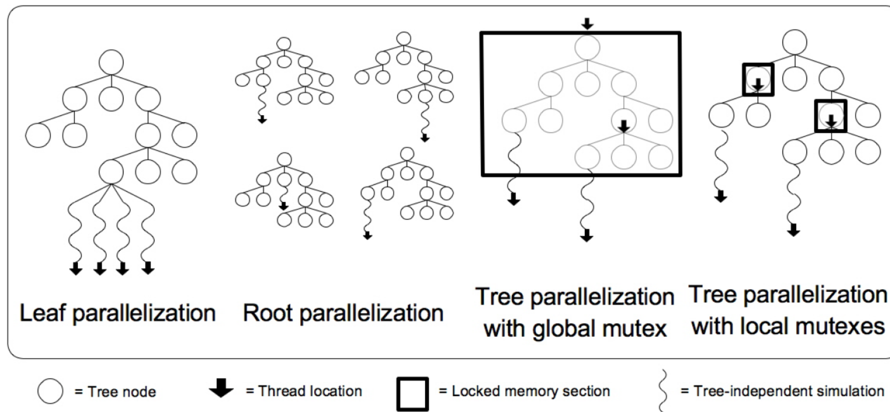

Figure 14: The three approaches to apply parallelization to UCT [11]. Note that tree parallelization can be done with either local or global mutexes.

As shown in Figure 14, root parallelization creates multiple trees, and merges the results for the initial options afterwards.

Tree parallelization uses one global tree and lets each thread modify the tree, using locks to keep consistency.

Leaf parallelization runs multiple simulations or rollouts instead of just one when expanding a leaf node, giving a larger sample size.

In tests leaf parallelization proved to be slower, and not yield much better results [11]. Root parallelization is the one which has shown the most success, although some research on tree parallelization claim that it might be able to yield equal or better results [4, 11].

3 Approach

This section introduces the four variants of MCDomAI, where two of them use UCT and the other two use flat UCB. The variants are similar, the differences are described below:

1. UCB_{orig}: Applies flat UCB, using the UCB1 formula, in Equation 1, during selection to find the best move.

2. UCB_{mod}: This variant uses a modified version of the UCB1 formula as shown in Equation 8, but is otherwise similar to UCB_{orig}.

3. UCT_{orig}: UCT is used to find the best move, thus creates a search tree compared to (1) and (2). Is also using a slightly different propagation system than UCT_{mod}, in order to resemble a Minimax tree search.

4. UCT_{mod}: Similar to UCT_{orig}, but uses UCT together with the modified version of the UCB1 formula for selection, as well as a modified propagation system, which does not pay as much regard to the opponent player.

For clarity, Table 1 shows the differences and similarities when it comes to selection formulas for the different variants.

	UCB1 Formula	**Modified UCB1 Formula**
UCT	UCT_{orig}	UCT_{mod}
Flat UCB	UCB_{orig}	UCB_{mod}

Table 1: A matrix overview of all variants and which selection formula they use.

Since the variants are very similar to each other, UCB_{orig} and UCB_{mod} is presented together, showing the difference between them in appropriate sections. The same is done for UCT_{orig} and UCT_{mod}, although there is a more significant difference between these variants, which is also presented where appropriate.

Figure 15 shows the main approach of MCDomAI in general, where the goal is to find the best of all possible moves in the current state. This is done by testing every move many times, as a multi-armed bandit problem, where the best move is the one that in the end has received the most visits, due to the UCB1 formula.

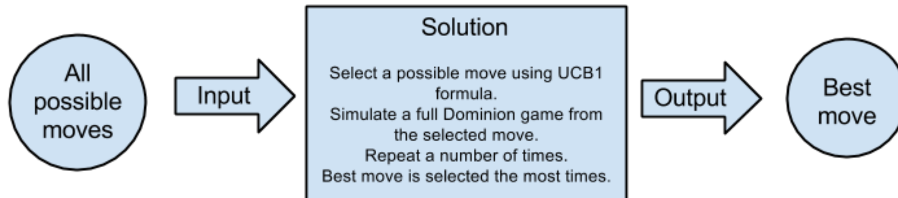

Figure 15: The general approach for all variants of MCDomAI.

Throughout the rest of this section, as well as Section 4, there is a number of configurations and results presented. These experiments are mostly adjusting on a few parameters at a time, so a single configuration table is presented for each experiment. An example table is presented in Table 2. To save space, the settings are not explained in detail for each table, so a general explanation is provided:

- Sims.: Abbreviation for Simulations. This is the number of simulations used for each game. See Section 3.4.5 for more information on the UCT variants and Section 3.5.1 for the flat UCB variants.
- Scoring: Scoring system refers to which scoring system from Section 3.4.2 is used during simulation.
- C: Also refers to Section 3.4.2, where C is an adjustable constant to balance exploration and exploitation.
- Rollout: Which policy is used during the rollout phase. See Section 3.4.3.
- Epsilon: Epsilon is the percentage of how much of the rollout is randomized. Only applicable to Epsilon Greedy rollout policy. See Section 3.4.3 for more information.
- Min. Vis.: Abbreviation for Minimum Visits. The minimum amount of times a top node should be visited. Used to combat cases where nodes are only visited once. See Section 3.4.6.
- MPPAF: Abbreviation for Must Play Plus Actions First. Column indicates whether the enhancement that forces MCDomAI to play cards that give +X Actions before other action cards is enabled. See Section 3.4.4.
- Thrs.: Abbreviation for Threads. Refers to the amount of threads used to run experiment. See Section 3.4.7.
- Its.: Abbreviation for Iterations, and refers to the number of iterations used to achieve better performance and sampling. Multiply with Threads to find number of search trees created. See Section 3.5.3.
- Starting Player: Denotes which player starts the game. Does not apply to the "Total" rows, because "Total" is the measured values added together, so that both players start 50% of the games each.
- Wins: Displays how many games that were won by the tested configuration.
- Losses: Displays how many games that were lost by the tested configuration.
- Ties: Displays how many games that were tied by the tested configuration.

- Win Perc.: Abbreviation for Win Percentage. It is the number of wins divided by the total number of games. Ties are counted negatively against the win percentage.

Simulations	100 000
Scoring	Win+Diff
C	0.5
Rollout	Epsilon H. Greedy
Epsilon	10
Min. Vis.	1
MPPAF	True
Thrs.	1
Its.	1

Table 2: An example configuration table.

As mentioned in Section 1.1, only a specific set of kingdom cards was chosen as test-bed for MCDomAI. The reasoning behind this is mainly that since this is a relatively new approach for Dominion, the goal is not to implement as many cards as possible, but create and benchmark a Dominion AI, which can be further improved and used for the complete game. The list of used cards is located in Appendix C.

To save time, MCDomAI always plays all treasure cards during buy phase. Based on domain knowledge, it is very close to always bad to buy a Curse card, so those moves have been completely removed / pruned.

The number of games played in further experiments may seem a bit odd, but this is because each experiment had to run for several days, and would sometimes have to be stopped and started again, so some experiments have sometimes been kept going after convergence. Instead of removing the extra data, it is included, as more data is better.

Most of the parameter setups run MCDomAI against a simple and conditional, finite-state machine-based AI, named Big Money Finite-State Machine (BMFSM). Note that BMFSM is not an actual finite-state machine only similar to one. The algorithm for BMFSM can be found in Appendix A.

Due to time limits, we could not test all four variants with each configuration, so UCT_{mod} is used as a representative variant to find the best scoring system, value for C, rollout policy and whether to use the Must Play Plus Actions First (MPPAF) enhancement.

3.1 Brute Force Search Tree

It is of interest to first research whether brute force can be applied to play Dominion, because if it is, then an optimal solution may possibly be derived from it, thus one could be capable of creating an optimal AI. The naïve or brute force approach to playing Dominion might be possible, but is arguably

infeasible. To demonstrate this, a brute force tree was created, showing every possible game state of a very simplified game of Dominion from start until a limited number of turns. The simplification made use of only treasure and victory cards, and also limiting the game to single-player mode. Each node in the created tree represents a different game state, as either a ’draw node’ (red squares), effectively constituting a player’s hand, or a ’buy node’ (blue circles), showing only which card the player bought, if he something at all.

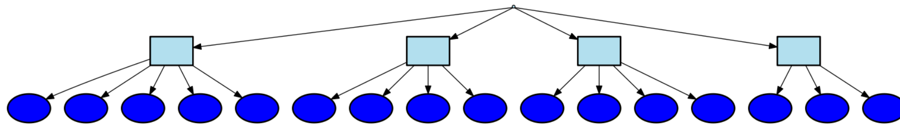

Figure 16: The first turn in the brute force tree containing 21 nodes. The squares represent the different combinations of card draws, while the circles represent different buy options.

Figure 16 shows the brute force tree for the first turn, where the top node is a dummy root node, having four children, which represent the four possible ways to draw a combination of five cards from a player’s starting deck. Each ’draw node’ will give the player an amount of money to buy a card for, respectively 5, 4, 3 or 2 coins. Using this money, the player may then buy a card costing up this amount, the different choices represented by the leaf nodes or ’buy nodes’. Note that the amount of money will limit the set of cards available for purchase. Graphing the first turn in Figure 16 is relatively straightforward, but this becomes exponentially more difficult as the number of turns increases. Due to the nature of Dominion, the branching will increase more at times when the deck has to be shuffled, and the player has to draw many cards from the newly shuffled deck. In the case of turn two, there is only one possible combination of cards left in the deck, as there will always be exactly five cards remaining. Even though the node size is increasing from 21 to 99 nodes when graphing the first two turns, this branching will become much larger whenever there is need for a shuffle.

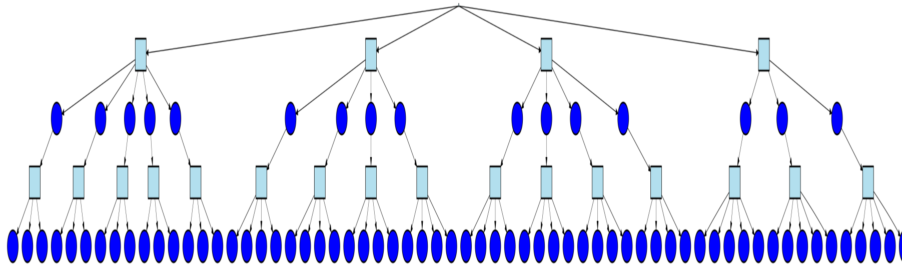

Figure 17: The second turn in the brute force tree containing 99 nodes. The squares represent the different combinations card draws, while the circles represent different buy options. The figure is purposely chaotic and unclear to illustrate the infeasible amount of nodes the brute force tree has to work with.

Figure 17 shows the increasing amount of nodes needed to graph the first two turns, however, at turn 2 there is the need to shuffle cards, and with between 10-12 cards in deck, with between 2-4 different card types, the number of nodes quickly increases. The number of nodes used to create a brute force tree including turn 3 requires 2077, which is more than 20 times as many as turn 2.

Table 3 contains the number of nodes needed for a tree of depth up to turn six, together with the branching factor for each turn increase.

Turns	**Nodes**	**Approximate Branching Factor**
First turn	21	N/A
First 2 turns	99	4.7
First 3 turns	2 077	21.0
First 4 turns	23 631	11.4
First 5 turns	681 365	28.8
First 6 turns	15 366 943	22.6

Table 3: The number of nodes needed for a Dominion brute force search tree for the first 1-6 turns.

The two main reasons for the big branching factor are:

1. Each player has roughly 10 choices each turn, depending on which cards are on player's hand. For each of these choices, there will be 10 new choices, etc. as we explore the brute force tree. This is less than what is expected in a game of e.g. Go, where in a 19x19 game each player has 361 - number of turns played moves available, (this amount further increases, when groups of stones are captured).

2. The second reason, which is by far the most important, is the stochastic element associated with drawing one or multiple cards from a deck. After each turn the players draw five new cards to constitute their hand for the next turn. With a deck consisting of 8 different cards, and 5 of each type,

which is not an unlikely size for the end game of Dominion, there are 52 different card combinations.

Requiring 15 366 943 nodes for the six first turns of a game of about 30 turns in a single-player game of Dominion without the ten kingdom cards, the brute force approach is arguably infeasible.

3.2 Dealing with Stochastic Card Draws

In a game of Dominion each player has to draw cards from a shuffled deck at the end of each turn, and when playing some cards, like the Village card. At the end of each turn the player is required to draw a total of five new cards. Since the deck of cards is turned face-down and the card order is unknown, there is a certain stochastic element when drawing a card. This is a challenge for UCT, because the UCB1 formula in Equation 1 is normally based on complete information, thus each node in the tree should ideally represent complete knowledge about a game state, and the edges should have guarantees for transitioning between nodes. However, since this is not the case for Dominion we present two new methods we tried for dealing with this for the UCT variants:

1. Create a search node for each possible combination of card draws and associate a probability with each node.
2. Proportionally sample a single card draw, and use it for the rest of the search.

The first approach, see Figure 18, requires MCDomAI to create a lot of nodes, and is not very far from a brute force solution, since the number of card draws is the biggest factor to increasing tree size. For example, in a deck of 20 cards, with 4 different types, and 5 cards of each type, there are a total of 56 card combinations. Towards the end of a game of Dominion it is not unusual to have a deck of 40 cards, with possibly 17 different types, although not 5 of each type. Note that the order of cards does not matter here, only how many cards of each type, thus we only have to create a node for each *combination* of drawn cards, not each permutation. This reduces the amount of nodes that needs to be created, but not by enough.

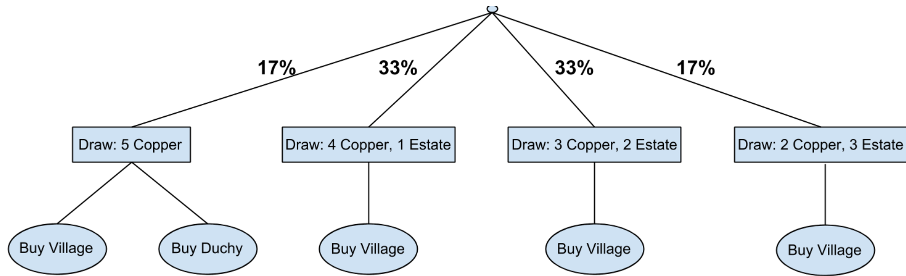

Figure 18: Create a node for each possible card draw, and assign the edges of each card draw an associated probability.

In order to create draw nodes, all possible card draws must first be found along with their associated probability, so instead of expanding one node at a time, when encountering a node that has card draws beneath, all card draws are created at once, and then one of them is selected afterwards using the associated probability together with score and the UCB1 formula, to let likely nodes be explored more. However, since each node should be visited at least once, this approach soaks up a lot of simulations, exploring unlikely combinations of cards. One solution is to set each draw node to set the visited counter to 1 upon creation, so that all do not need to be explored. Still, the number of nodes needed for this approach to be effective will likely be enormously high, in fact it will exceed the number of atoms on Earth!

This makes the second solution more attractive, which also the one used for the UCT variants Whenever there is need to draw cards in the search tree, the probabilities for drawing card combinations are used, resulting in a single card draw. This card draw will then be used for all further simulations, even though the game may never progress to that specific card draw. This works because the card draws are made using proportional sampling, and can be further increased using parallelization to create more realistic card draws, see Section 3.4.7 for experiments on sampling. Thus, instead of only having one set of card draws, multiple search trees will view different possibilities of card draws, resulting in better game estimation. Figure 19 shows how multiple search trees increases card draw sampling.

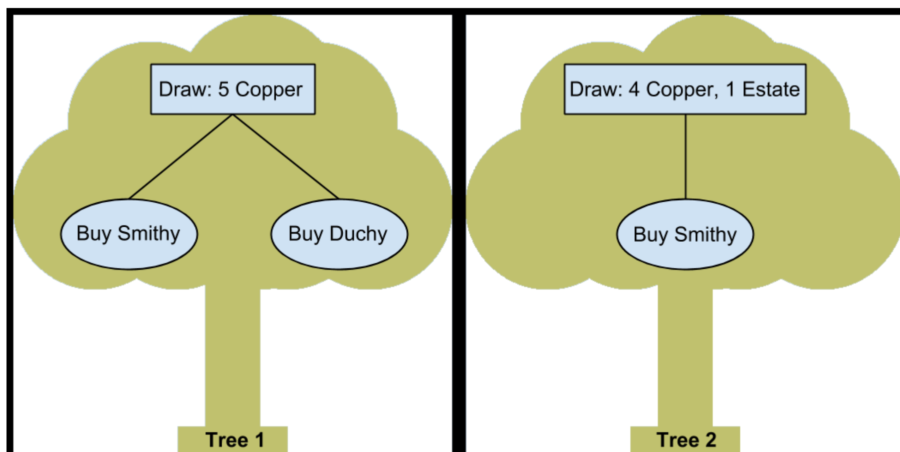

Figure 19: Illustrating how different search trees sample multiple draws.

This is resembling the idea behind Thompson sampling in Section 2.5.3, where one would create a probability distribution over the different card draws, and then pick one using mentioned distribution, and then work with that specific sample. Using Thompson sampling for dealing with card draws is however different from the bandit enhancement itself, but the idea is the same.

Due to the lack of a search tree, card draws are not such a big challenge for the UCB variants, because all card draws are taken care of during rollout phase. Card draws during the rollout phase are simulated as if it was a real game, where random cards are selected from the deck according to their proportional probabilities, similar to the sampling approach for the UCT variants. However, these samples are not kept across simulations due to the states beyond the current one not being saved, as opposed to in UCT when a node is created with a fixed state based on the previous card draws.

3.3 Interaction Between Players

For the UCT variants of MCDomAI, the interaction between players is taken care of at three places:

1. UCT tree nodes: The UCT search tree creates nodes for both players, which is the way most UCT implementations work.
2. Propagation: Normally UCT would create nodes for the opponent's moves, and score them based on the UCT player's score. The UCT_{mod} variant simply skips propagating whenever the opponent makes a move. UCT_{orig} uses a slightly different approach by propagating the opponent's score to the opponent's nodes, as well as increasing the number of visits in all nodes on the path to the leaf, instead of skipping propagating entirely. UCT_{orig}

resembles a Minimax behavior (normally Minimax is done during selection), since the algorithm will try to explore where both players perform well.

3. Rollout: The rollout policy is the same for both players, but could have been modified to resemble the opponent player. For MCDomAI however, one assumes that the opponent will also use the epsilon heuristic greedy policy.

For the flat UCB variants, interaction between players is much more subtle. Because there is no search tree, there are no nodes representing the opponent's moves, nor is the propagation present. So the rollout is the only place where the opponent's behavior could be modeled, possibly by using different policies for the opponent and MCDomAI. However like the UCT variants, the same epsilon, heuristic, greedy rollout policy is used for both players.

Even without any specific handling of the opponent, the flat UCB variants seem to handle interaction between players well.

For all variants we note that complete information about both players' discard piles is known to both players. While it is possible to hide some cards during the clean-up phase by discarding the whole hand at once, leaving only the top card visible, we considered the benefit of this neglectable. Hiding cards can possibly be of some use for the complete game, but when using the cards in test-bed it should not matter. Drawing cards from the deck is random, and the order is not known. Information regarding which cards are in which pile thus gives little advantage.

Also, since draws are done at the end of each turn, information about the opponent's next draw is available during rollout. This is a simplification that may have a larger impact than knowledge about the discard piles. However, for this setup of cards, it should generally not matter much since there are few cards which interact with each other.

3.4 UCT Variants: UCT_{orig} and UCT_{mod}

This section presents the two UCT variants, UCT_{orig} and UCT_{mod}. They are similar in most aspects, however there are the following differences:

1. UCT_{mod} uses a modified UCB1 selection formula. See Section 3.4.2.

2. UCT_{mod} needs to use a higher number of simulations, due to the modified selection formula. See Section 3.4.5.

3. UCT_{orig} uses a Minimax-like propagation system. See Section 3.3.

When playing Dominion there are a number of times during a player's turn that the player is required to make a choice. At all such times this approach runs the logic from Figure 20. By first examining all possible moves, then simulating full games of Dominion from each of these moves, and building a search tree as the process runs, this approach is capable of finding the a very good move,

possibly the optimal one. Note that cards where a player is required to take an action during another player's turn are not implemented, due to the extra work in rollout and expert knowledge required, but this should be feasible to implement in the future.

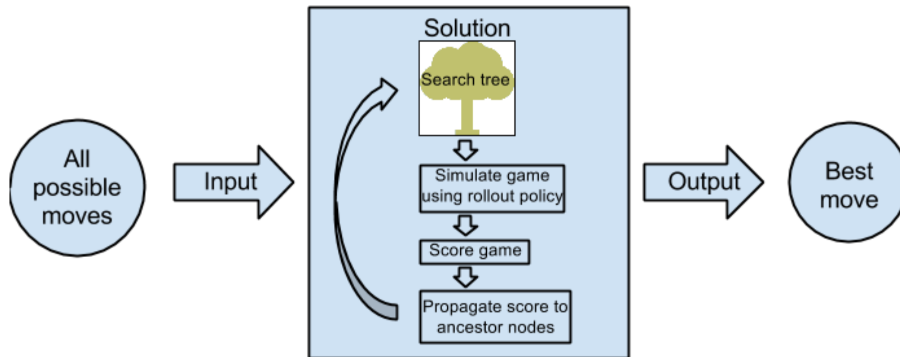

Figure 20: The approach to find the best move from the set of possible moves, by using UCT.

3.4.1 Dominion in the Search Tree

In order to use UCT, one needs to build a search tree, which should be a subtree of the brute force search tree. The game state is represented by nodes, in the same way as the brute force approach in Section 3.1, but extended with nodes for playing action cards. Since a turn can be longer or shorter, depending on the player's choices, there is no direct link between the tree depth and the number of turns. However, the tree depth is equal to the number of moves that were made to reach the specific node or game state. An example of a small search tree is given in Figure 21.

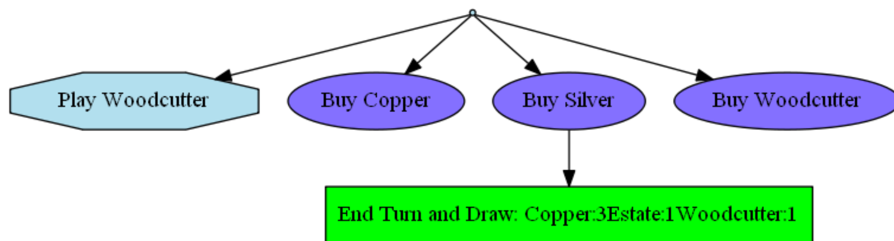

Figure 21: A small search tree, showing how the Dominion game state is changed.

Each of the top nodes represents a choice or a move, and finding the optimal move can be viewed as a multi-armed bandit problem. The idea is try out the four different top moves, while also expanding the search tree with one child

node after every try, selecting a node using the formula in Equation 1. Then, this node runs a simulated game, receiving a score, which is then later used during the next node selection. After a given number of loops, see Section 3.4.5, the best move is the one with the highest amount of visits. The best node value could also be chosen as the best move, however the difference seems to be small, but there are cases where the best node may not have been visited the most times, due to the selection formula also calculating some exploration into the total score. In the future, one could do some experiments to see if this has an effect on performance.

3.4.2 Propagation and Scoring

When selecting a leaf node to expand, the UCB1 formula in Equation 1 is used for UCT_{orig}, while the modified formula in Equation 8 is used for UCT_{mod}.

$$v_i + C \times \sqrt{\ln \frac{n_p}{n_i}} \tag{8}$$

where the variables are the same as for Equation 1.

The difference between using Equation 1 and 8 as selection formula is mainly that the when using the modified formula, good options are exploited more in the beginning, while exploring is done more towards the end of a search. Similar to first play urgency (FPU), as mentioned in Section 2.5.5.

The value set for C in Equation 1 and 8 is also dependent on the scoring system, which gives each simulation a score. This score is then propagated back through each ancestor until reaching root. The score is further used when selecting a new child node to expand. Note that the propagation is actually not passing through each ancestor, only each ancestor belonging to the same player as the leaf node, see Section 3.3 for details. Normally when using UCT the scoring system only uses Pure Win/Loss, listed as solution (1). However, this solution alone seemed to have some problems in "sure" win or lose cases. In order to combat this, an improved scoring system, listed as (2), was also tested.

1. Pure Win/Loss: The following approach is the one often used in other UCT applications. It only keeps track of the number of wins and how many times a node has been visited. When a game is won, the value of 1 is propagated upwards, while also increasing the amount of visits in each propagated node, including the leaf node. This scoring system uses the formulas in Equations 9 and 10, depending on the simulation result in Equation 11.

$$\text{Score} = \text{Winpoint} = \begin{cases} 1, & \text{if Diff} > 0 \quad (9) \\ 0, & \text{otherwise} \quad (10) \end{cases}$$

where

$$\text{Diff} = \text{P}_{\text{UCT}} - \text{P}_{\text{Opponent}} \tag{11}$$

and P_{UCT} is the current player's VPs in the simulated node and $\text{P}_{\text{Opponent}}$ is the opponent's VPs in the simulated node.

2. Win/Loss + percentage of difference in score (Win+Diff): Works like the scoring system 1, but adds a percentage of the difference in the players' VPs to the total score. The formula is given in Equation 12.

$$\text{Score} = \frac{\text{Diff}}{100} + \text{Winpoint} \tag{12}$$

where Diff is equal to the formula in Equation 11, and P_{UCT} is the current player's VPs in the simulated node and $\text{P}_{\text{Opponent}}$ is the opponent's VPs in the simulated node. Winpoint is equal to the formulas in Equations 9 and 10, depending on the simulation result in Equation 11.

The second scoring system is an improvement to the first one. The reason for the improvement came from some observations of random moves in situations where MCDomAI already had either clearly lost or won. In these cases, MCDomAI would buy or play a bad card, instead of ensuring victory or trying to catch up. This is probably because the first scoring system does not distinguish what is a small or huge victory/loss, because the same score is given no matter which option is taken, when all options lead to either loss or victory. Consider the following example: MCDomAI has currently 60 VP, while the opponent has 20VP. There is only one Province card and one Smithy card left in the supply, and all other victory cards have been bought. MCDomAI will win the game, no matter what happens next, so on its turn, it may finish the game buy buying the last Province card, but it might as well buy the Smithy card, because both options will lead to victory for MCDomAI.

This is something to be avoided, because even if MCDomAI is going to win/lose it could be mistaken, which is why it is important to distinguish between good and bad options in these cases too. In order to solve this, one could make use not only of the win/loss outcome of a game, but also the difference in VP at the game end. So in the example above, MCDomAI would think that buying the Province card will give a 66 VP - 20VP = 46 VP difference, while buying the Smithy card will lead to a 60VP - 20 VP = 40 VP difference. Since 46 VP ¿ 40 VP it will give the Province card a better score, and most likely buy that one instead of the Smithy card. Note that the difference can be negative too, in order to solve the situation for both winning and losing.

One could also modify the formula of the second scoring system, by dividing $Diff$ with something else than 100, in order to weigh more heavily upon the difference in score. However, it is important to not add in a too high value to the total score, as this will give a fluctuating score, thus making exploration and exploitation more difficult to balance. If a dynamic value for C was used, this could be possibly be tested as a different scoring system in the future.

In order to find the best scoring system, the experiment in Table 4 plays them against BMFSM, using the configurations in Table 5. The different scoring systems were tested only on UCT_{mod}, however the best scoring system should be applicable for all variants of MCDomAI, due to their similarities in selecting nodes.

Scoring	**C**	**Win Perc.**	**Wins**	**Losses**	**Ties**
Pure Win/Loss	0.3	76.7%	810	240	6
Pure Win/Loss	0.4	79.0%	779	202	5
Pure Win/Loss	0.5	81.9%	870	186	6
Pure Win/Loss	0.6	85.0%	899	146	13
Pure Win/Loss	0.7	80.9%	743	166	9
Win+Diff	0.3	78.2%	541	147	4
Win+Diff	0.4	85.7%	715	117	2
Win+Diff	0.5	86.9%	722	101	8
Win+Diff	0.6	87.5%	835	111	8
Win+Diff	0.7	**88.7%**	892	105	9
Win+Diff	0.8	87.4%	836	112	8

Table 4: Finding the best scoring system together with a value for C, when using UCT_{mod}. The configuration is found in Table 5. See Section 3 for table explanations.

Simulations	100 000
Scoring	Multiple
C	Multiple
Rollout	Heuristic Greedy
Epsilon	N/A
Min. Vis.	1
MPPAF	True
Thrs.	1
Its.	1

Table 5: Configuration for finding the best scoring system and value for C in Table 4, when using UCT_{mod}. See Section 3 for table explanations.

The best scoring system in Table 4 is Win+Diff, with a C value of 0.7. At the end of the work, it was discovered that if two or more nodes had the same value during selection, then the last node added would always be picked, thus not selecting at random. This could be of some importance to the Pure Win/Loss system, as nodes would more often end up with a similar score. However for Win+Diff, the chances are much lower that two nodes should share the same score, due to the addition of difference in VPs.

3.4.3 Rollout

The rollout policy, as described in Section 2.3.3, is used to simulate games has a big impact on how well MCDomAI performs. For all variants of MCDomAI, we have chosen to test three different rollout policies, which are fairly normal to use when experimenting in a new field. In the future it would also be interesting to test out more policies.

1. Random Rollout Policy
2. Heuristic Greedy Policy
3. Epsilon Heuristic Greedy Policy

The random rollout policy simulates a game where both players take their turns by picking a random option each time they are required to make a choice. This method is used in many other applications for UCT, but have been found to not perform well in Dominion, as shown in Table 6.

The second rollout policy is based on heuristics, and works like a greedy policy, hence *heuristic greedy*. In Dominion there are many ways to view how a 'greedy' policy should behave. Should it buy the cards that gives most VPs or that have the highest cost? Thus, it is important to clarify what greedy means in MCDomAI. There are two different situations when MCDomAI needs to make a choice, both when buying a card and when playing a card, and in both cases the most greedy choice is to pick the card with the highest cost.

When it comes to playing action cards, some more heuristics could have been used, but that is a very complicated process as the number of cards interacting with each other increase. Possibly could one put in a rule-based strategy as the rollout policy, but such strategies would then have to be made for each different set of cards, thus making MCDomAI less adaptive. However, a more general heuristic rule, 'Must Play Plus Actions First' (see Section 3.4.4), was applied, as this is easier to apply to other sets of cards as well. The pseudo-code in Algorithm 1 explains the rollout policy:

```
while game is not finished do
    while player has more buys or actions do
        if there are any action cards in player's hand then
            // The MPPAF enhancement
            if there is a plus actions card then
                play that card;
            else
                find action cards in hand with highest cost;
                play random card with the highest cost;
            end
        else
            money := hand.money();
            find cards with highest cost that is not higher than money and
            that is not Curse;
            if there are no cards then
                break;
            else
                buy a random card that has the highest cost;
            end
        end
    end
end
```

Algorithm 1: Heuristic Greedy rollout policy

The last rollout policy is a form of epsilon-greedy, where the greedy policy part is the heuristic greedy explained as policy 2. As described in Section 2, epsilon-greedy combines random rollouts with greedy rollouts, so that in $1 - \epsilon$ percent of the cases, a single move made during rollouts is done using greedy policy, while in ϵ percent of the cases, a single move is chosen at random. Note that it is not the whole simulation that is being decided by epsilon, only a single move at a time. Like the constant C, the value for ϵ must be found experimentally, but should generally have a low value.

The reasoning behind epsilon greedy is that sometimes in Dominion, the best choice is not to simply buy or play the highest value card available, mostly because of the wide variety of cards which can make other cards better when used in conjunction. So during rollout, the best move may actually be a random move, which is why epsilon greedy often performs better than pure greedy.

The following experiment tests different rollout policies used by UCT_{mod} against BMFSM over multiple games until convergence. The configurations used are displayed in Table 7. Note that the random policy was stopped after 121 games, since one win and 120 losses is sufficient information to say that this is not the best policy. Although the value for C in all experiments is 0.5, (0.7 was showed to be slightly better in Table 4), we argue that the rollout policy has an independent enough performance increase, to show that the best one will be better no matter which C is used, as long as the same C is used. The experiment only uses UCT_{mod} as a representative variant to find a rollout

policy for all variants.

Rollout	C	Epsilon	Win Perc.	Wins	Losses	Ties
Random	0.5	N/A	0.8%	1	120	0
Heuristic Greedy	0.5	N/A	86.9%	722	101	8
Heuristic Greedy	0.7	N/A	88.7%	892	105	9
Epsilon H. Greedy	0.5	1	89.5%	1038	114	8
Epsilon H. Greedy	0.5	5	91.8%	1009	80	10
Epsilon H. Greedy	0.5	10	93.0%	930	65	5
Epsilon H. Greedy	0.5	15	94.0%	942	56	4
Epsilon H. Greedy	0.5	20	93.0%	940	64	7

Table 6: Finding which rollout policy is best used to let UCT_{mod} win most games against BMFSM. The configuration used for the experiment is in Table 7. See Section 3 for table explanations.

As can be seen in Table 6, the epsilon heuristic greedy policy is the best one, with $\epsilon = 15$. Note that the heuristic greedy policy also includes one configuration running $C = 0.7$. Still, this configuration is worse than the epsilon rollouts, thus we can conclude that no matter which C is used, the best heuristic greedy policy is worse than the best epsilon heuristic greedy.

Simulations	100 000
Scoring	Win+Diff
C	0.5 (0.7)
Rollout	Multiple
Epsilon	Multiple
Min. Vis.	1
MPPAF	True
Thrs.	1
Its.	1

Table 7: Configuration for finding the best rollout policy using UCT_{mod} in Table 6. See Section 3 for table explanations.

3.4.4 Must Play Plus Actions First (MPPAF)

When watching the play of the UCT_{mod}, observations was made that it has some problems playing multiple actions cards in sequence, even if some of them give +X actions, thus enabling the play of more cards afterwards. For instance, if the AI has two actions cards in hand: A Smithy and a Village, the optimal play in almost all circumstances is to first play the Village card, then the Smithy card, since playing the Village will allow the Smithy to be played as well, in addition to another potential action card. However, the UCT_{mod} seems to rather play the Smithy card first, and thus cannot play the Village card. The reasoning behind this may be that in the search tree, the Smithy node has better children

below, whereas the Village node has worse, thus the Smithy node has a better average than the Village node. This is despite the fact that the Village node also has a Smithy node below.

To resolve this situation, we tried three different approaches:

1. Forcing playing of plus actions cards during rollout only.
2. Forcing playing of plus actions cards and also during rollout.
3. Changing propagation to best child, instead of averaging over all children.

The first solution enforces the two rollout policies described in Section 3.4.3, except Random Rollout Policy, to play +Actions cards first, while still performing UCT. Early tests showed that this solution alone was not enough for UCT_{mod} to play multiple action cards in sequence. The rollout policy still kept the enforcement as heuristics in the second solution, intuitively providing some benefit to the performance in general.

The second solution integrates the first solution in addition to forcing UCT_{mod} to first play action cards that give +Actions, overriding UCT. Using the specific set of cards in the supply, see Appendix C, there is only a neglectable amount of scenarios when it would be wise not to play any of the cards that give +Actions, so it would generally always be the best choice. Note that the sequence the +Actions cards are played in does not matter in this supply, so an order was not specified. The two main strengths of this approach are:

1. The solution saves time, by not always having to create a search tree before choosing an action.
2. The solution chooses the optimal move in almost all circumstances.

The weakness is however that it is not applicable in such a simple form to all cards that give +Actions, i.e Cellar and Spy from the original game (not to mention cards from expansions). However, should such cards also be added, one could always add more complex rules for these.

This solution seems to solve the problem well from the experiment in Table 8 using the configurations in Table 9.

Scoring	**MPPAF**	**Rollout**	**Win Perc.**	**Wins**	**Losses**	**Ties**
Pure W/L	False	Heuristic	44.0%	449	565	7
Pure W/L	True	Heuristic	82.6%	828	172	2
Win+Diff	False	Eps. H.	86.7%	716	105	5
Win+Diff	True	Eps. H.	93.0%	930	65	5

Table 8: Testing the effect of the MPPAF enhancement against BMFSM for UCT_{mod}. The configuration is in Table 9. See Section 3 for table explanations.

In both test cases, the MPPAF enhancement increases the performance significantly.

Simulations	100 000
Scoring	Pure Win/Loss, Win+Diff
C	0.5
Rollout	Heuristic Greedy, Epsilon H. Greedy
Epsilon	N/A, 10
Min. Vis.	1
MPPAF	True, False
Thrs.	1
Its.	1

Table 9: Configuration for testing the effect of the MPPAF enhancement against BMFSM for $\mathrm{UCT}_{\mathrm{mod}}$ in Table 8. See Section 3 for table explanations.

The third solution is more extensive, and is very similar to the Max strategy used for Go [9], based on a negamax approach [31]. Instead of using heuristics, the solution changes the propagation system to give each node the value of its best children. Since the source of the problem most likely lies in, that when MCDomAI has the choice between a card giving +Actions and a terminal one, for instance Village and Smithy respectively, Village becomes an inferior card, due to it having worse child nodes on average, than the Smithy node.

In order to fix this, the solution was to select the node that had the best children nodes, in cases were one can guarantee that the child node can be reached, thus not in cases involving the other player's turn or the previously used draw nodes from Section 3.2.

However, this solution was only tested through observations during using some of the obsolete configurations. Although it is possible that the solution may work if implemented in the future, for this thesis there was not enough time to test extensively.

3.4.5 Simulations

The performance of the UCT variants is directly connected to the amount of simulated games. A high simulation count lets the UCT variants create nodes deeper in the search tree, as well as explore many potential nodes. Thus a high simulation count allows the AI to see further ahead.

The experiment in Table 10 shows the difference in performance for $\mathrm{UCT}_{\mathrm{mod}}$ against BMFSM. The experiment configurations are found in Table 11.

Sims.	Win Perc.	Wins	Losses	Ties
100	0.1%	1	758	0
1 000	15.8%	164	866	6
5 000	70.4%	914	370	15
10 000	85.1%	1088	182	9
25 000	91.9%	1347	109	9
50 000	92.7%	879	59	10
100 000	94.4%	915	49	5
200 000	95.7%	878	34	5

Table 10: Finding out how simulations affect performance in skill level for UCT_{mod}. The configuration settings are found in Table 11. See Section 3 for table explanations.

Simulations	100 - 200 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	1
MPPAF	True
Thrs.	1
Its.	1

Table 11: Configuration for finding out how simulations affect performance in skill level for UCT_{mod} in Table 10. See Section 3 for table explanations.

The performance seems to increase throughout the whole table in Table 10, which means that the number of simulations can possibly be increased for further performance.

For variant UCT_{orig}, the number of simulations needed is much lower, due to the differences in the selection formula. The number of games run for this experiment was only 100 for each setting, due to time limits. Experiments in the future will increase this amount to achieve more confident values. The results for UCT_{orig} is found in Table 12, using the configuration in Table 13.

Sims.	Win Perc.	Wins	Losses	Ties
100	0%	0	100	0
1 000	62%	62	36	0
5 000	88%	88	9	3
10 000	94%	94	6	0
100 000	99%	99	1	0

Table 12: Finding out how simulations affect performance in skill level for UCT_{orig}. Configurations are found in Table 13. See Section 3 for table explanations.

Simulations	100 - 100 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	**100 sims**: 1, **The rest**: 10
MPPAF	True
Thrs.	1
Its.	1

Table 13: Configuration for finding out how simulations affect performance in skill level for UCT_{orig} in Table 12. The experiment also used 999 as maximum turns. See Section 3 for table explanations.

The results in Table 12 show that 10 000 simulations may be enough, as increasing the amount to 100 000 only increases the games won by 2. As already mentioned earlier, more games need to be run before the difference between 10 000 and 100 000 simulations can be concluded. Further experiments with UCT_{orig} use 10 000 simulations, as this seems sufficient.

3.4.6 Minimum Visits

When using UCT_{mod}, there is a case where one or more of the top nodes are only being visited exactly one time. The situation seems to occur mostly during the beginning of a game, when using the second scoring system mentioned in Section 3.4.2. This is probably because it receives a very low score during the rollout, thus is never visited again. This is not desirable, because the we wish to visit each of the top nodes at least a certain amount of times, to ensure that the score is relatively representative, and not a coincidence. This is also related to the scoring system and value for C from Equation 1, and could be possible to fix by adding a dynamic C, as described in Section 2.4.1. However, due to time limits, the only approach tested was to set a minimum amount of times each top node should be visited. The exact number needs to be experimented with.

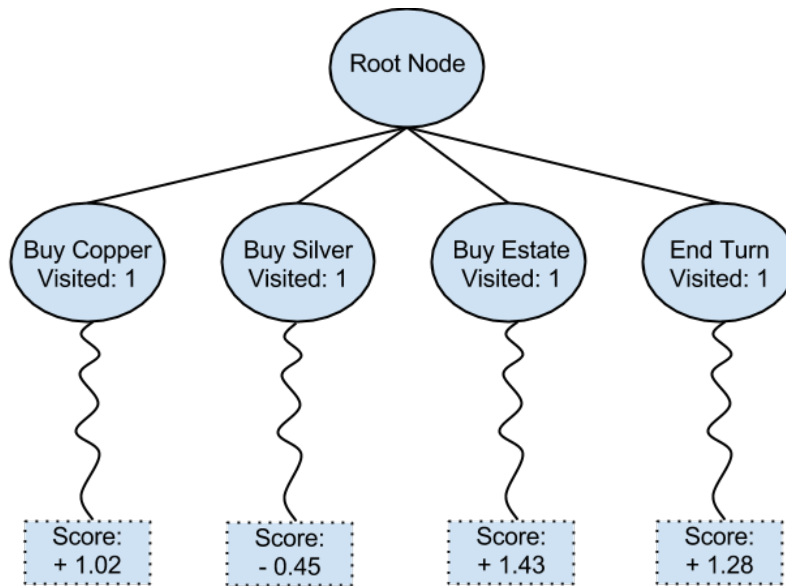

Figure 22: Showing an example over the first four simulations, where "Buy Silver" receives a low score during its first rollout. "Buy Silver" will not be picked by the UCB1 algorithm during selection.

In order to resolve the situation in Figure 22, we tried to force every top node to be visited at least a certain amount of times, before starting to exploit the results. In Table 14 this enforcement is applied to different setups, because the problem is likely related to the scoring system. The configuration used in the experiment is shown in Table 15.

Scoring	MPPAF	MV	Win Perc.	Wins	Losses	Ties
Pure Win/Loss	False	500	38.1%	317	506	10
Pure Win/Loss	False	100	38.2%	432	689	9
Pure Win/Loss	False	1	44.0%	449	565	7
Win+Diff	True	500	84.1%	739	136	4
Win+Diff	True	100	85.9%	826	122	14
Win+Diff	True	50	84.6%	862	141	16
Win+Diff	True	20	86.4%	551	82	5
Win+Diff	True	15	86.4%	554	78	9
Win+Diff	True	10	**89.2%**	837	89	12
Win+Diff	True	5	82.9%	518	102	5
Win+Diff	True	1	86.9%	722	101	8

Table 14: Testing whether the minimum visits enhancement can increase performance of UCT_{mod} against BMFSM. To save space in the table, Minimum Visits was abbreviated MV. The configuration is in Table 15. See Section 3 for table explanations.

Simulations	100 000
Scoring	Multiple
C	0.5
Rollout	Heuristic Greedy
Epsilon	N/A
Min. Vis.	Multiple
MPPAF	True, False
Thrs.	1
Its.	1

Table 15: Configuration for testing the minimum visits enhancement for UCT_{mod} in Table 14. See Section 3 for table explanations.

As can be seen, when setting MinVis to 10, there is a slight increase in performance. Since the Pure Win/Loss scoring system was showed to be inferior to Win+Diff in Table 4, only a few values was tested for Pure Win/Loss.

To test this enhancement with more optimal settings, another experiment was run using the configuration found in Table 17, where Epsilon H. Greedy is used as rollout policy. The results are found in Table 16

Scoring	MPPAF	Min. Vis.	Win Perc.	Wins	Losses	Ties
Win+Diff	True	1	94.4%	915	49	5
Win+Diff	True	10	94.6%	472	24	3

Table 16: Testing whether the minimum visits enhancement can increase win rate of UCT_{mod} against BMFSM, when using epsilon heuristic greedy. The configuration is found in Table 17. See Section 3 for table explanations.

Simulations	100 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	1, 10
MPPAF	True
Thrs.	1
Its.	1

Table 17: Configuration for the minimum visits enhancement for epsilon heuristic greedy rollout policy in Table 16. See Section 3 for table explanations.

While the results show that there may be an increase in performance also when using epsilon heuristic greedy rollout, the number of games run is not high enough to ensure this increase.

However, since the minimum visits enhancement seemingly increases performance, we chose to enhance both UCT variants with minimum visits, using 10 as parameter.

While minimum visits was not tested for UCT_{orig}, we chose to use it for UCT_{orig} too, because intuitively, worst-case scenarios should be rather harmless, because enforcing each node to be visited 10 times should not affect the end result. However, since this affects the order of how nodes are visited first, using this enhancement may also create side effects.

3.4.7 Parallelization

Another way to possibly increase performance is by applying parallelization and multi-threading. As mentioned in Section 2, there are different forms of parallelization, but MCDomAI uses root parallelization because of two reasons:

1. There is no need to lock the tree, thus eliminating the possibility of inconsistency.
2. It helps with the sampling process mentioned in Section 3.2, by adding more trees, thus more card draw samples.

The way parallelization is done for the UCT variants is by creating a search tree for each thread used during a run, and possibly creating more trees on each thread after the first is completed, as seen in Figure 23. When all trees are completed, the number of visits for the top child nodes are added together, and the one with the highest total is picked. It is important to note that root parallelization will not increase performance in terms of speed, unless a lower amount of simulations are used, but could rather increase playing strength.

When doing parallelization in this particular way, there is no difference in playing strength by either increasing the number of threads and or the number

of iterations, as increasing either option will increase the number of search trees made, which is equal to threads multiplied with iterations.

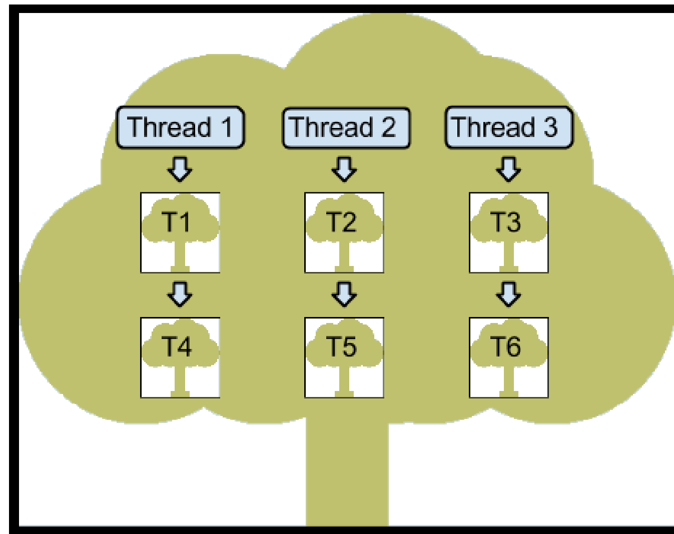

Figure 23: Illustrating how parallelization is done for the case of 3 threads and 2 iterations for each thread.

Table 18 shows the increase in playing strength only, when increasing the number of threads and iterations. For the impact of in time per move when using parallelization, see the experiment in Section 4.8.

Sims.	**Thrs.**	**Its.**	**Win Percentage**	**Wins**	**Losses**	**Draws**
100 000	1	1	93.6%	741	45	6
100 000	2	1	95.1%	993	44	7
100 000	3	1	93.8%	938	56	6
100 000	3	2	95.1%	548	26	2

Table 18: Employing root parallelization for UCT_{mod} shows an increase in playing strength when increasing the number of threads and iterations. The configuration is found in Table 19. See Section 3 for table explanations.

Simulations	100 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	10
MPPAF	True
Thrs.	1
Its.	1

Table 19: Configuration for the parallelization experiment in Table 18. See Section 3 for table explanations.

While the results are somewhat fluctuating, there seems to be a small increase in playing strength when using more iterations.

A likely reason for the increase in playing strength is that when using the second card drawing system, described in Section 3.2, creating more trees will also make more card draw samples, which could even out unlikely card draws in one tree, with more likely card draws in another.

Although the increase in strength seems to be minor, it is possible that the playing strength could increase even more against other opponents than BMFSM. Due to the time and capacity to run with extra threads and iterations, which is something that would have to be done in the future.

Also, it would be interesting to test the impact of root parallelization for UCT_{orig} too in the future.

3.5 UCB Variants: UCB_{orig} and UCB_{mod}

The approach for the flat UCB variants of MCDomAI is similar to the approach for the UCT variants described in Section 3.4, as well as the M:TG approach [41].

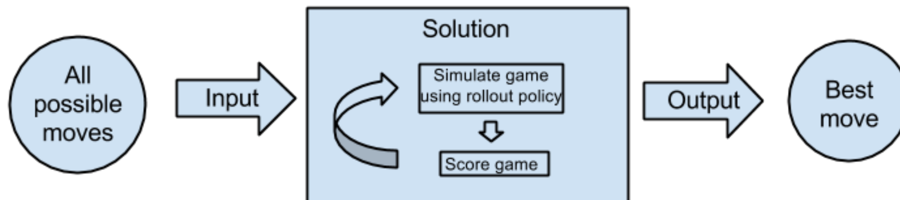

Figure 24: The approach to find the best move from the set of possible moves, by using flat UCB.

Figure 24 shows that the main difference from the UCT variants is the absence of a search tree to create and store child nodes. Instead, only the available options (top nodes in UCT search tree) are stored and updated with

values. The scoring system, rollout system used for UCT_{mod} and UCT_{orig}, as well as MPPAF, are also used for UCB_{mod} and UCB_{orig}, however there is no propagation for the flat UCB variants, due to the lack of a search tree.

UCB_{orig} uses the same selection formula in Equation 1 as UCT_{orig}, while UCB_{mod} uses the same modified UCB1 in Equation 8 as UCT_{mod}. In addition, the choice of selection formula also affects the number of simulations, thus the two main differences are:

1. UCB_{mod} uses the modified UCB1 selection formula in Equation 8.

2. UCB_{orig} uses a higher number of simulations, due to the modified selection formula. See Section 3.5.1.

3.5.1 Simulations

As with the UCT variants, the number of simulations is directly connected to the playing strength of the flat UCB variants as well. Both UCB_{orig} and UCB_{mod} have been tested against the BMFSM, in order to find how many simulations is needed before the results converge. Table 20 shows the results for UCB_{mod}, and Table 22 shows the results for UCB_{orig}.

Sims.	**Win Perc.**	**Wins**	**Losses**	**Ties**
100	0.9%	9	962	0
1 000	63.7%	893	499	10
5 000	95.2%	952	44	4
10 000	97.9%	979	20	1
25 000	98.6%	986	11	3
50 000	99.4%	1032	5	1
100 000	99.4%	994	6	0

Table 20: Experiment showing the performance of UCB_{mod} against BMFSM, with different numbers of simulations. The configuration is found in Table 21. See Section 3 for table explanations.

Simulations	Multiple
Scoring	Pure Win/Loss
C	0.5
Rollout	Epsilon H. Greedy
Epsilon	10
Min. Vis.	1
MPPAF	True
Thrs.	1
Its.	1

Table 21: Configuration for testing number of simulations against BMFSM in Table 20. See Section 3 for configuration explanation.

The experiment in Table 20, shows that 50 000 simulations is enough before the results start to converge. This could be because of the lack of a search tree, which might save some simulations compared to UCT_{mod}, another reason is that BMFSM is not able to provide a big enough challenge for UCB_{mod} to the point where we can see a difference in the results anymore.

While 50 000 simulations are enough against BMFSM, it is still possible that the number of simulations could be increased for further performance in other experiments, which is why UCT_{mod} uses 100 000 simulations in the rest of the experiments. The amount of simulations should however be further tested in the future against other opponents as well. The configuration used can be seen in Table 21.

The configurations shown in Table 21 are also not optimal, as this experiment was run before the rest of the settings were tweaked. In the future, one should also run the experiment for UCB_{mod} again with optimal settings, to see if it has any impact.

Sims.	**Win Perc.**	**Wins**	**Losses**	**Ties**
100	2%	2	98	0
1 000	88%	88	11	1
5 000	100%	100	0	0
10 000	99%	99	1	0
100 000	100%	100	0	0

Table 22: Experiment showing the performance of UCB_{orig} against BMFSM, with different numbers of simulations. The configuration is found in Table 23. See Section 3 for table explanations.

Simulations	Multiple
Scoring	Win + Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	10
Min. Vis.	**100 sims**: 1, **The rest:** 10
MPPAF	True
Thrs.	1
Its.	1

Table 23: Configuration for testing number of simulations against BMFSM in Table 22. See Section 3 for configuration explanation.

For UCB_{orig} the win percentage in Table 22 already starts converging at 5 000 simulations. Although only 100 games are used for each configuration, it is probably sufficient to say that either is 5 000 simulations the peak for UCB_{orig}, or BMFSM is not a good enough opponent for UCB_{orig}. In future experiments

the value of 10 000 simulations is mainly used, in the case that UCB_{orig} would perform better against better opponents.

In the future, more games should be run using different amounts of simulations against better opponents, in order to find how many simulations are needed for peak performance.

As mentioned in the start of Section 3.5, the number of simulations required to efficiently beat BMFSM is much lower for UCB_{orig} than for UCB_{mod}.

3.5.2 Minimum Visits

Since no observations were made of the flat UCB variants sometimes visiting the top child nodes only once, this enhancement could have been turned off. However, since it is a desired feature to visit each child node at least 10 times, the enhancement was left on for UCB_{orig} and UCB_{mod}.

As mentioned in Section 3.4.6, there is a possibility that this may have negative side effects for the flat UCB variants, which is something to test in the future.

3.5.3 Parallelization

Parallelization for UCB_{orig} and UCB_{mod} is done the same way as for the UCT variants. However, since there is no search tree with sampled card draws, it is as effective to simply increase the number of simulations, thus the flat UCB variants should intuitively be equally strong by running 10 000 simulations on two threads, or by running 20 000 on one thread. However, by running on multiple threads, one can spread out the number of simulations on different processing cores, thus increase the playing speed.

One could also increase the number of iterations, which will make each thread repeat the process of running the set amount of simulations, and then merge everything together at the end. This should also increase playing strength in the same way as with the number of threads. The actual playing strength could be measured using the formula in Equation 13.

$$TotalSimulations = NumberofSimulations \times Threads \times Iterations \quad (13)$$

where Number of Simulations is the number to be used by all threads and iterations, Threads is the number of Threads used and Iterations is the number of Iterations used for each thread.

Another important feature of root parallelization is that it prevents staying too long in a local optima, which could further increase playing strength beyond the total number of simulations [11].

However, since the parallelization for UCB_{orig} should be rather plain, excluding the feature about staying in local optima, no experiments were run on for testing the playing strength. If time had allowed, it would have been interesting to see how the playing strength actually increase when applying parallelization.

The speed performance should be proportional to the total number of simulations in Equation 13, divided by the number of threads. Time per move is measured for a few configurations of UCB_{orig} in Section 4.8.

3.6 Framework and Implementation

In order to test MCDomAI against other strategies and humans, a simple console framework was created using Python, but later, C++ was needed for memory conservation and speed. The framework was used to test MCDomAI against the other strategies.

The parameter setups were run using a maximum amount of turns, not allowing games to last any longer than 40 turns. The rollouts were also using this value, and would end unfinished games at turn 40. Usually games do not last as long as 40 turns, but this could have a small, but neglectable impact on the results, so the limit was set to 999 for the experiments in Section 4. The setting should have more impact on games with bad players, who avoid to buy VPs, thereby prolonging the game. However, since the parameter setup experiments all have the same maximum turn limit, this should not have any significant consequences, other than perhaps the time needed for each move.

In the future a different approach could have been to use a modifiable maximum turn limit for the rollout, while removing the maximum turn counter for the actual Dominion games.

4 Experiments

This section covers experiments on all variants of MCDomAI beyond parameter setup, and compares them to other novel strategies. Due to time limits, some of the experiments only test the performance of $\mathrm{UCT}_{\mathrm{mod}}$ and $\mathrm{UCB}_{\mathrm{mod}}$. More experiments will be done in the future, but due to similarities, it is reasonable to assume that the performance of $\mathrm{UCT}_{\mathrm{orig}}$ and $\mathrm{UCB}_{\mathrm{orig}}$ are similar to the ones tested.

First, we test MCDomAI's ability to recognize good and enormously good kingdom cards, which is important to know if one is to implement more cards later.

Secondly, we test MCDomAI in a scenario-like single-player situation, where both variants tested should be able to find an optimal single-player playstyle.

The third experiment plays some of the variants against each other, in order to compare their performance in a more direct manner.

The fourth experiment tests MCDomAI against a random playing AI. While a very weak opponent, it is important to have a very general strategy to compare MCDomAI against.

Experiment 5 and 6 test all four variants of MCDomAI against two successful Dominion strategies, by playing them against MCDomAI in two-player games.

The seventh experiment runs all four variants of MCDomAI against human players at different skill levels.

The last experiment measures how long it takes for the different variants of MCDomAI to make a move, when using different numbers of simulations and threads.

4.1 Unbalanced Kingdom Card

This experiment looks at whether $\mathrm{UCB}_{\mathrm{mod}}$ and $\mathrm{UCT}_{\mathrm{mod}}$ recognize extremely good cards or not. The reason we test this is to show that MCDomAI is capable of finding cards outside the range of test-bed cards, as well as finding cards that are far superior to the normal cards. We test this by modifying the Woodcutter card, making it provide more coins when played, to see how this affects what the variants will buy. Expected results is that as we increase the value of the Woodcutter card, it will be bought more often. Treasure cards should be less, when more Woodcutter cards are bought, since their purpose is the same (to give the player more buying power). The configuration for both variants is found in Table 24. The experiment is done over 10 games for each value of the Woodcutter card, which should be sufficient, as we are interested in watching play style, not competitive results.

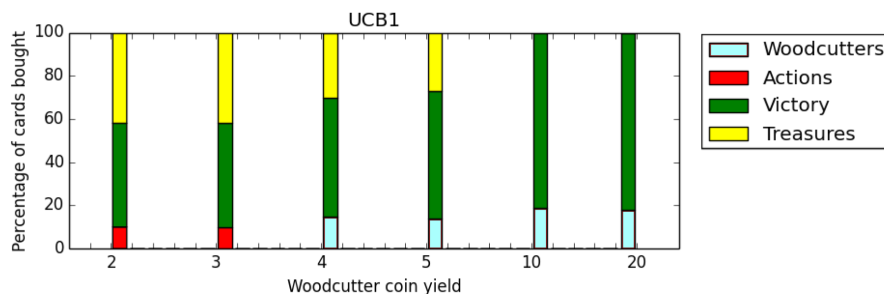

Figure 25: Cards bought as the coin value of the Woodcutter card increases for $\mathrm{UCB}_{\mathrm{mod}}$.

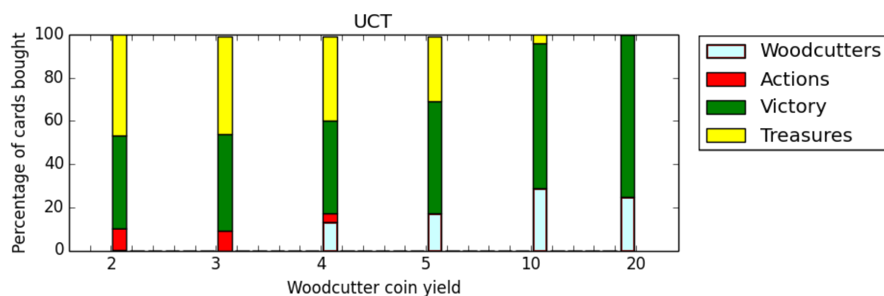

Figure 26: Cards bought as the coin value of the Woodcutter card increases for $\mathrm{UCT}_{\mathrm{mod}}$.

Simulations	100 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	1
MPPAF	True
Thrs.	1
Its.	1

Table 24: Configuration for the Unbalanced Card Experiment in Figure 25 and 26. See Section 3 for table explanations.

The graphs in Figure 25 and 26 show the distribution of cards bought as the coin yield of the Woodcutter card increases. For both variants the experiment goes as expected. Woodcutter cards are being bought when they increase in value, at least 4 coins, and is bought more often when the value further increases.

At the extreme values of 10 and 20 treasure cards are made obsolete. One could wonder as to why not treasures are being rendered obsolete at an earlier point, but the reason is probably that even though Woodcutter cards increase in value, only one Woodcutter card can be played each turn, but any number of treasures can be played, so in order to buy any Province cards, costing 8, one would need some additional coins.

Note that the number of Woodcutter cards actually decreases, when their value is increased from 10 to 20. This is most likely caused by the decreasing need for multiple Woodcutter cards as the value of every Woodcutter card is increased, thus allowing to buy more VPs instead.

4.2 Optimal Playstyle for Single-player

In this experiment the kingdom cards were removed, leaving only treasure, victory and Curse cards. The game is played with only one player in order to test whether an optimal solution can be found. The goal here is to achieve the maximum amount of VPs possible, and there is a set of optimal solutions here, all of them requiring not to buy empty any other supply pile than the three victory card piles, buying empty the Province cards last and not buy any Curse cards. If these requirements are met, one will end up with 83 VPs, which is the maximum amount of VPs possible to gain in single-player mode.

Because there is no kingdom cards, the amount of simulations needed to find an optimal solution is drastically reduced, as the branching factor is much smaller. The lack of an opponent should also approximately half the amount of simulations needed.

The experiment is done over 100 games on each configuration for both UCT_{mod} and UCB_{mod}. While it would be interesting to see the performance of the other two variants here as well, so those experiments will be run in the future. However, due to similarities with the other two variants, we argue that their performance should be about the same. The results are found in Table 25.

Sims.	**Variant**	**Average VPs**	**Optimal Games**
10	UCT_{mod}	79.8 ± 7.4	77%
20	UCT_{mod}	82.7 ± 1.6	96%
30	UCT_{mod}	83.0 ± 0.0	**100%**
10	UCB_{mod}	78.9 ± 10.5	83%
20	UCB_{mod}	82.7 ± 2.0	97%
30	UCB_{mod}	82.9 ± 0.6	99%

Table 25: Finding optimal plays in single-player games for UCT_{mod} and UCB_{mod}. Configuration is found in Table 26. Further table explanations can be found in Section 3.

The Average VPs column denotes how many points were achieved on average for all 100 games. Optimal games are the percentage of games that achieved 83 VPs. Std. dev. is the standard deviation of the average VPs achieved.

Simulations	10, 20, 30
Scoring	VPs
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	1
MPPAF	True
Thrs.	1
Its.	1

Table 26: Configuration for finding optimal plays single-player for UCT_{mod} and UCB_{mod} in Table 26. The scoring system, VPs, is only used for this experiment. Each rollout is scored equal to the amount of VPs achieved. See Section 3 for table explanations.

The results in Table 25 show that the number of simulations required to find an optimal playstyle for this setup is very low, compared to the number of simulations needed for a game using a full set and two players. UCT_{mod} achieves a slightly higher average VPs, while UCB_{mod} has more optimal games. Due to the low number of games run, neither of the variants can be concluded better on this particular task.

Both variants are able to achieve the highest score when using VPs as the scoring system, and a C that is not tuned, at a very small amount of simulations. This a very good display of how powerful Monte-Carlo methods can be, also for the game of Dominion.

4.3 Playing the Variants Against Each Other

A direct approach for testing which variant is better is to play them against each other. The experiment in Table 27 plays the variants using a modified UCB1 formula against each other, Table 29 plays the other two variants against each other, while Table 31 plays the best variants from Table 27 and 29 against each other.

Starting Player	Win Perc.	Wins	Losses	Ties
UCT_{mod}	26.8%	89	243	0
UCB_{mod}	87.0%	289	43	0
Total UCT_{mod}	19.9%	132	532	0
Total UCB_{mod}	**80.1%**	532	132	0

Table 27: 664 games between UCT_{mod} and UCB_{mod}. Starting position is important, so an equal amount of games are played for both possibilities. The configuration is found in Table 28. See Section 3 for table explanations.

Simulations	100 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	1
MPPAF	True
Thrs.	1
Its.	1

Table 28: Configuration for the games between UCB_{mod} and UCT_{mod} in Table 27. See Section 3 for table explanations.

The results from Table 27 show that UCB_{mod} is clearly better than UCT_{mod}, by maintaining 80.1% wins over 664 games.

One reason that UCB_{mod} performs better than UCT_{mod} can be the way they are dealing with the stochastic card draws, from Section 3.2 and 3.2. While UCT_{mod} uses proportional sampling to create picture of the card draws, UCB_{mod} simply avoids the problem by not creating a search tree at all.

Another possibility is that the game complexity of Dominion may not be as huge as expected, at least for the test-bed cards, see Appendix C, thus is a locally optimized (greedy) choice often the best choice. One strength of flat UCB is the ability to find this greedy choice, while UCT may be better at planning ahead.

Starting Player	**Win Perc.**	**Wins**	**Losses**	**Ties**
UCT_{orig}	35%	35	63	2
UCB_{orig}	86%	86	13	1
Total UCT_{orig}	24%	48	149	0
Total UCB_{orig}	**74.5%**	149	48	3

Table 29: 200 games between UCT_{orig} and UCB_{orig}. Starting position seems to be important, so an equal amount of games are played when each player starts. The configuration is found in Table 30. See Section 3 for table explanations.

Simulations	10 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	10
MPPAF	True
Thrs.	1
Its.	1

Table 30: Configuration for the showdown between UCT_{orig} and UCB_{orig} in Table 29. See Section 3 for table explanations.

The results in Table 29 still favors UCT over flat UCB. In addition to the different opponent for UCT_{orig}, another reason that UCT_{orig} is performing slightly better than UCT_{mod} from Table 27, could possibly be the change in the propagation system, described in Section 3.3.

Although the number of games played is a little low, the results from Table 27 and 29 show that flat UCB is likely better than UCT, when using the testbed cards in Appendix C setup of Dominion. Possibly could UCT outperform flat UCB when they do not play directly against each other, so both approaches are still tested the following experiments.

It was also observed that UCT_{orig} is capable of playing advanced combinations of cards, such as using the Remodel card to trash a Gold card to gain a Province card, and playing several action cards in a row. Since no such observations were made for the flat UCB variants, it is possible that UCT may perform better with a more complex card set. Expanding the card set is mentioned in Section 5.1 as something we would like to do in the future, especially to see if UCT will outperform flat UCB.

Also, since the UCT variants use proportional sampling, as described in Section 3.2, the performance of UCT could increase when more parallel search trees are created, by either increasing the number of threads or the number of iterations. Unfortunately, time limits made it difficult to test with these settings, so we would also like to test this in the future.

Starting Player	Win Perc.	Wins	Losses	Ties
UCB_{orig}	63%	63	37	0
UCB_{mod}	71%	71	29	0
Total UCB_{orig}	46%	92	108	0
Total UCB_{mod}	**54%**	108	92	0

Table 31: 200 games between UCB_{orig} and UCB_{mod}, which both outperformed their opponent variant in the previous experiments in Table 27 and 29. The configurations differ only in the number of simulations, as the minimum visits value is set to 10 for both variants. Full configuration is found in Table 32. See Section 3 for table explanations.

Simulations	$\mathbf{UCB_{orig}}$: 10 000, $\mathbf{UCB_{mod}}$: 100 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	10
MPPAF	True
Thrs.	1
Its.	1

Table 32: Configuration for the games between UCB_{orig} and UCB_{mod} in Table 31. See Section 3 for table explanations.

UCB_{mod} wins in 54% of the games played, which is a little unexpected compared to the later experiments against SingleWitch and DoubleWitch in Section 4.5 and 4.6. Although the tests in Section 3.5.1 found 10 000 simulations to be a peak for UCB_{orig}'s performance, it is still possible that increasing the number of simulations could affect the performance when playing against more advanced opponents.

Another reason that UCB_{orig} may be performing differently is because the minimum visits value is set to 10 for the experiment in Table 31. At the end of the DoubleWitch experiments in Section 4.6, it is speculated that this number may actually decrease the playing strength of UCB_{orig}. This is also something that should be tested further in the future.

The number of games played is not enough to conclude that UCB_{mod} is the best variant, and possibly could the other variants be better in other experiments and situations.

In the future we would like to test all variants against each other over a larger number of games, in order to verify which is the best when playing against each other.

Another interesting observation is that the starting player seems to have an advantage, which is further supported by the results in the SingleWitch and DoubleWitch experiments in Section 4.5 and 4.6.

4.4 MCDomAI versus Random AI

A simple benchmarking opponent is an AI that picks a random option whenever prompted to make a choice. MCDomAI should be able to maintain 99-100% win percentage against a random AI, as the nature of the game punishes bad choices, and rewards good ones. This is due to that bought cards are drawn later, thus bad buys early will leave poor choices later.

The experiment tests only the UCT_{mod} and UCB_{mod} variants in Table 33 and 34, because the experiment in Section 4.3 shows that the other two variants should outperform these, and it would be intuitively to say that the results for all variants should be approximately the same. However, we would also like to test this experiment for all variants to verify this statement in the future.

The configuration for both tested variants is found in Table 35. Note that the simulation number is set to as little as 50 simulations, which is very small compared to how many simulations normally run (100 000). This is because the random player has an extremely low skill level, so 50 simulations should be enough to beat it. Minimum visits for both variants is set to 1, due to the low number of simulations.

Starting Player	**Win Perc.**	**Wins**	**Losses**	**Ties**
UCT_{mod}	100%	100	0	0
Random Player	0%	0	100	0
Total UCT_{mod}	100%	200	0	0
Total Random Player	0%	0	200	0

Table 33: Experiment showing UCT_{mod} versus a computer player choosing random moves. UCT_{mod} wins in all 200 games. The configuration is found in Table 35. See Section 3 for table explanations.

Starting Player	**Win Perc.**	**Wins**	**Losses**	**Ties**
UCB_{mod}	100%	100	0	0
Random Player	0%	0	100	0
Total UCB_{mod}	100%	200	0	0
Total Random Player	0%	0	200	0

Table 34: Experiment showing UCB_{mod} versus a computer player choosing random moves. The configuration is found in Table 35. UCB_{mod} wins in all 200 games. See Section 3 for table explanations.

Simulations	50
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	1
MPPAF	True
Thrs.	1
Its.	1

Table 35: Configuration for UCT_{mod} and UCB_{mod} against random AI in Table 33 and 34. Note that the number of simulations is only set to 50, which is very low compared to the other experiments. See Section 3 for table explanations.

As expected, both variants are capable of beating a random computer player in all games played, even when using only 50 simulations per move. As mentioned in the start of the section, it would also be interesting to play the other

two variants against the random AI, but they would most likely win about the same amount of games as well.

This experiment also shows that Dominion is a complex game, which requires players to choose wisely between good and bad choices, since choosing random options will not lead to victory.

4.5 MCDomAI versus SingleWitch

There is a Dominion simulator available on a web page, where it is possible to test strategies against each other [37]. The simulator supports mainly rule-based strategies, so we could not directly add and test MCDomAI in the simulator. However, one of the strategies that seemed to perform very well is called SingleWitch, which is a tactic that buys one Witch card, plays a Witch card whenever possible and buys additional treasure and victory cards. The full algorithm is listed in Appendix B.

Running an experiment of 200 games using SingleWitch against BMFSM, SingleWitch won in all 200 games, where both strategies started in 100 games each.

All variants of MCDomAI were tested against SingleWitch over 200 games each, where SingleWitch and MCDomAI started an equal amount of times for each variant. The results are found in Table 36, 37, 39 and 40.

Starting Player	**Win Perc.**	**Wins**	**Losses**	**Ties**
UCT_{mod}	44%	44	50	6
SingleWitch	74%	74	26	0
Total $\mathbf{UCT_{mod}}$	**35%**	70	124	6
Total SingleWitch	62%	124	70	6

Table 36: UCT_{mod} plays against SingleWitch, but does not perform very well, only capable of winning 35% of the games. The configuration is found in Table 38. See Section 3 for table explanations.

Starting Player	**Win Perc.**	**Wins**	**Losses**	**Ties**
UCB_{mod}	86%	86	12	2
SingleWitch	48%	48	51	1
Total $\mathbf{UCB_{mod}}$	**68.5%**	137	60	3
Total SingleWitch	30%	60	137	3

Table 37: UCB_{mod} plays against SingleWitch, achieving a total of 68.5% win percentage. The configuration is found in Table 38. See Section 3 for table explanations.

Simulations	100 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	$\mathbf{UCT_{mod}}$: 10, $\mathbf{UCB_{mod}}$: 1
MPPAF	True
Thrs.	1
Its.	1

Table 38: Configuration for UCT_{mod} and UCB_{mod} against SingleWitch in Table 36 and 37. Note that the minimum visits were set to 10 for UCT_{mod}, and 1 for UCB_{mod}. As mentioned in Section 3.5.2, this should only have minor differences for UCB_{mod}, compared to using 10 as value. See Section 3 for table explanations.

Starting Player	**Win Perc.**	**Wins**	**Losses**	**Ties**
UCT_{orig}	62%	62	37	1
SingleWitch	54%	54	46	0
Total $\mathbf{UCT_{orig}}$	**54%**	108	91	1
Total SingleWitch	45.5%	91	108	1

Table 39: UCT_{orig} plays against SingleWitch, winning 54% of the games. The configuration is found in Table 41. See Section 3 for table explanations.

Starting Player	**Win Perc.**	**Wins**	**Losses**	**Ties**
UCB_{orig}	75%	75	25	0
SingleWitch	38%	38	62	0
Total $\mathbf{UCB_{orig}}$	**68.5%**	137	63	0
Total SingleWitch	31.5%	63	137	0

Table 40: UCB_{orig} plays against SingleWitch, winning 68.5% of the games. The configuration is found in Table 41. See Section 3 for table explanations.

Simulations	10 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	10
MPPAF	True
Thrs.	1
Its.	1

Table 41: Configuration for UCT_{orig} and UCB_{orig} against SingleWitch in Table 39 and 40. See Section 3 for table explanations.

An overview of all variants playing against SingleWitch in this experiment is provided in Table 42.

Variant	**Opponent**	**Win Percentage**
UCT_{mod}	SingleWitch	35%
UCB_{mod}	SingleWitch	68.5%
UCT_{orig}	SingleWitch	54%
UCB_{orig}	SingleWitch	68.5%

Table 42: An overview of the four experiments in Table 36, 37, 39 and 40. Opponent is the strategy used against the MCDomAI variant in the Variant column.

As in the experiments in Section 4.3, the UCT variants perform worse than the flat UCB variants. However, UCT_{orig} is performing better than UCT_{mod} this time, which could further support that the difference in the propagation system from Section 3.3 is an improvement.

UCB_{orig} and UCB_{mod} performs almost equally, the only difference being the number of losses instead of ties. But when running only 200 games against SingleWitch, it is difficult to tell which of the flat UCB variants that is the better one, possibly they perform equally well.

As mentioned in Section 4.3, in the future one could also test with 10 000 simulations for UCB_{orig} against SingleWitch, to see if the increased number of simulations will help increase performance.

4.6 MCDomAI versus DoubleWitch

DoubleWitch is a Dominion strategy from the Dominion Simulator, which is almost identical to SingleWitch [37], but buys two Witch cards instead of one. DoubleWitch is performing even better than SingleWitch, as shown in Table 43, and achieves a win percentage of 63.5% over SingleWitch, which should be sufficient to conclude that DoubleWitch is better than SingleWitch.

Additionally, we have attempted to find tactics which consistently beat DoubleWitch, but have been unable. This is indicates that DoubleWitch might be one of the strongest strategies for our setup.

Starting Player	**Win Perc.**	**Wins**	**Losses**	**Ties**
SingleWitch	44%	44	55	1
DoubleWitch	75%	75	24	1
Total SingleWitch	34%	68	130	2
Total DoubleWitch	**65%**	130	68	2

Table 43: Experiment showing SingleWitch versus DoubleWitch. DoubleWitch wins 65% games against SingleWitch. See Section 3 for table explanations.

All variants of MCDomAI were also tested against DoubleWitch over 200 games each, also equally dividing the starting position between the MCDomAI variants and DoubleWitch. The results for UCT_{mod} and UCB_{mod} are shown respectively in Table 44 and 45, using the configuration in Table 46, and the results for UCT_{orig} and UCB_{orig} are presented in Table 47 and 48, using the configuration in Table 49.

Starting Player	**Win Perc.**	**Wins**	**Losses**	**Ties**
UCT_{mod}	35%	35	62	3
DoubleWitch	72%	72	27	1
Total UCT_{mod}	**31%**	62	134	4
Total DoubleWitch	67%	134	62	4

Table 44: Experiment showing UCT_{mod} versus DoubleWitch, winning only 31% of the games. The configuration is found in Table 46. See Section 3 for table explanations.

Starting Player	**Win Perc.**	**Wins**	**Losses**	**Ties**
UCB_{mod}	71%	71	28	1
DoubleWitch	46%	46	54	0
Total UCB_{mod}	**62.5%**	125	74	1
Total DoubleWitch	37%	74	125	1

Table 45: Experiment showing UCB_{mod} versus DoubleWitch, where UCB_{mod} achieves 62.5% win percentage against DoubleWitch, which is a good result, considering that DoubleWitch may be one of the best possible strategies for our test-bed setup. The configuration is found in Table 46. See Section 3 for table explanations.

Simulations	100 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	$\mathbf{UCT_{mod}}$: 10, $\mathbf{UCB_{mod}}$: 1
MPPAF	True
Thrs.	1
Its.	1

Table 46: Configuration for UCT_{mod} and UCB_{mod} against DoubleWitch in Table 44 and 45. See Section 3 for table explanations.

Starting Player	**Win Perc.**	**Wins**	**Losses**	**Ties**
UCT_{orig}	49%	49	50	1
DoubleWitch	58%	58	42	0
Total $\mathbf{UCT_{orig}}$	**45.5%**	91	108	1
Total DoubleWitch	54%	108	91	1

Table 47: Experiment showing UCT_{orig} versus DoubleWitch, winning close to half the amount of the games with a win percentage of 45.5%. The configuration is found in Table 49. See Section 3 for table explanations.

Starting Player	**Win Perc.**	**Wins**	**Losses**	**Ties**
UCB_{orig}	78%	78	22	0
DoubleWitch	41%	41	59	0
Total $\mathbf{UCB_{orig}}$	**68.5%**	137	63	1
Total DoubleWitch	31.5%	63	137	1

Table 48: Experiment showing UCB_{orig} versus DoubleWitch, winning 68.5% of the games. This is a significant amount, even more than UCB_{mod} in Table 45. The configuration is found in Table 49. See Section 3 for table explanations.

Simulations	10 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	$\mathbf{UCT_{orig}}$: 10, $\mathbf{UCB_{orig}}$: 1
MPPAF	True
Thrs.	1
Its.	1

Table 49: Configuration for UCT_{orig} and UCB_{orig} against DoubleWitch in Table 47 and 48. The minimum visits for UCB_{orig} is set to 1, which may have been a reason for the good performance. See Section 3 for table explanations.

For clarity, an overview of the SingleWitch and DoubleWitch experiments is provided in Table 50.

Variant	**Opponent**	**Win Percentage**
UCT_{mod}	SingleWitch	35%
UCB_{mod}	SingleWitch	68.5%
UCT_{orig}	SingleWitch	54%
UCB_{orig}	SingleWitch	68.5%
UCT_{mod}	DoubleWitch	31%
UCB_{mod}	DoubleWitch	62.5%
UCT_{orig}	DoubleWitch	45.5%
UCB_{orig}	DoubleWitch	68.5%

Table 50: An overview of the four experiments in Table 44, 45, 47 and 48. Opponent is the strategy used against the MCDomAI variant in the Variant column.

As expected, since DoubleWitch outperforms SingleWitch in Table 43, all variants of MCDomAI perform slightly worse against DoubleWitch than against SingleWitch. However, this is not the case for UCB_{orig}, which achieves the same number of wins against both strategies.

One reason for this could be that the number of games run is not enough to show a difference between the experiments, however since all the other variants show decreased performance, there may be more important factors at work. While not affecting UCB_{mod} in as clearly, the configuration for UCB_{orig} in Table 41 and 49 shows that the minimum visits value is set to 10 against SingleWitch and 1 against DoubleWitch. Possibly is the minimum visits enhancement affecting the performance of UCB_{orig} in a negative manner. Although we argued in Section 3.5.2 that the difference should probably not affect the performance, no tests were run to verify the statement.

It is not possible to conclude how minimum visits affect the performance of UCB_{orig} and UCB_{mod}, so future work should test this enhancement for these

variants too, as well as run more games against both SingleWitch and DoubleWitch in case more games would yield different results.

4.7 MCDomAI versus Human Players

To test how UCB_{mod} and UCT_{mod} performs against human players, a group of four fairly experienced players, including ourselves, were put together to play against both variants. All games are played as two-player games, with MCDomAI against one human opponent. The amount of games played were however too small to say anything conclusive.

The results are showed in Table 51, using the configuration in Table 52.

Variant	**Win Perc.**	**Wins**	**Losses**	**Ties**
UCT_{mod}	87.5%	7	1	0
UCB_{mod}	25%	1	3	0

Table 51: UCT_{mod} and UCB_{mod} versus human players of different skill level. While UCB_{mod} achieves a very good win rate, wining 7 out of 8 games, the number of games played is not high enough to say anything conclusive. The Variant column denotes which variant is tested against the human players. The configuration is found in Table 52.

Simulations	100 000
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	10
MPPAF	True
Thrs.	1
Its.	1

Table 52: Configuration for UCT_{mod} and UCB_{mod} against human players in Table 51. See Section 3 for table explanations.

Winning 7 out of 8 games, UCB_{mod} shows very strong performance, even though the number of games is not high enough to be conclusive to which variant is better against human players. Since the earlier experiments are not conclusive to which variant is best, further experiments may find that any of the four variants is performing better against human players.

In the future, this is one the most interesting experiment to further test for all variants.

4.8 Speed: Time per Move

In this experiment we measure the time taken per each move, compared to the number of simulations and the number of threads used. The turn counter also impacts the time, because the closer a game is to ending, the fewer options are available, thus each rollout does not need to search as long to reach game end.

The game implementation requires MCDomAI to make a move whenever there is no more buys left, due to the possibility of choosing not to buy or play any card. However, since most of the 'end turn moves' are made much faster than the regular moves, these moves are left out to maintain a more reliable average.

This experiment tests each variant of MCDomAI over a number of different configurations. The opponent is BMFSM in all games. The results for each variant is shown in separate tables, where the most relevant configuration is marked in bold. Tot. Avg. is the average for all turns in the 10 games, while 1-10 Avg. denotes the average time for the 10 first turns, and 11+ Avg. refers to the turns coming after the 10th.

Sims.	**Thrs.**	**Its.**	**Tot. Avg.**	**1-10 Avg.**	**11+ Avg.**
50 000	1	1	4.8 ± 3.3 s	7.9 ± 2.9 s	2.6 ± 1.0 s
100 000	**1**	**1**	**9.1 ± 6.6 s**	**15.4 ± 5.7 s**	**4.8 ± 2.0 s**
100 000	2	1	10.2 ± 6.2 s	16.8 ± 4.6 s	6.1 ± 2.2 s
100 000	3	1	11.5 ± 6.3 s	17.5 ± 4.9 s	7.0 ± 2.0 s
100 000	3	2	21.6 ± 11.8 s	33.6 ± 8.9 s	13.6 ± 4.3 s

Table 53: Determining time per move for UCT_{mod}, using the configuration in Table 57. See Section 3 for table explanations.

Sims.	**Thrs.**	**Its.**	**Tot. Avg.**	**1-10 Avg.**	**11+ Avg.**
25 000	2	1	3.1 ± 1.5 s	4.5 ± 1.5 s	2.2 ± 0.2 s
50 000	1	1	4.6 ± 3.4 s	7.5 ± 3.1 s	2.2 ± 0.9 s
100 000	**1**	**1**	**8.4 ± 6.4 s**	**13.9 ± 5.7 s**	**3.9 ± 2.0 s**
100 000	2	1	9.3 ± 6.9 s	14.9 ± 6.6 s	4.7 ± 1.7 s

Table 54: Determining time per move for UCB_{mod}, using the configuration in Table 57. See Section 3 for table explanations.

Sims.	**Thrs.**	**Its.**	**Tot. Avg.**	**1-10 Avg.**	**11+ Avg.**
1 000	1	1	1.0 ± 0.0 s	1.0 ± 0.0 s	1.0 ± 0.0 s
5 000	1	1	1.1 ± 0.0 s	1.1 ± 0.1 s	1.0 ± 0.0 s
10 000	**1**	**1**	**1.2 ± 0.4 s**	**1.6 ± 0.5 s**	**1.0 ± 0.0 s**
100 000	1	1	8.2 ± 5.9 s	13.7 ± 5.4 s	4.6 ± 2.3 s

Table 55: Determining time per move for UCT_{orig}, using the configuration in Table 57. Note that the low amount of simulations lower the average time per move significantly. See Section 3 for table explanations.

Sims.	**Thrs.**	**Its.**	**Tot. Avg.**	**1-10 Avg.**	**11+ Avg.**
1 000	1	1	1.0 ± 0.0 s	1.0 ± 0.0 s	1.0 ± 0.0 s
5 000	1	1	1.0 ± 0.0 s	1.0 ± 0.1 s	1.0 ± 0.0 s
10 000	**1**	**1**	**1.2 ± 0.4 s**	**1.5 ± 0.5 s**	**1.0 ± 0.0 s**
100 000	1	1	8.6 ± 6.7 s	14.1 ± 6.0 s	3.9 ± 2.0 s

Table 56: Determining time per move for UCB_{orig}, using the configuration in Table 57. Note that the low amount of simulations lower the average time per move significantly. See Section 3 for table explanations.

Simulations	Multiple
Scoring	Win+Diff
C	0.7
Rollout	Epsilon H. Greedy
Epsilon	15
Min. Vis.	10
MPPAF	True
Thrs.	Multiple
Its.	Multiple

Table 57: Configuration for all variants when determining time per move in Table 53, 54, 55 and 56. See Section 3 for table explanations.

Table 53 and 54 show that the time used on average per move is proportional to the number of simulations multiplied with the number of iterations, with only a small overhead due to the number of extra threads. This seems to be the case until reaching about 10 000 simulations, there Table 55 and 56 show that the minimum amount of time per move is approximately 1 second.

The results in Table 55 and 56 have a much lower simulation number, which is why the measured time per move is significantly smaller than in Table 53 and 54. This is done because of those variants need a lower amount of simulations to perform at an acceptable level, as shown in Section 3.4.5 and 3.5.1.

The performance in speed seems to be about the same for all variants when using the same amount of simulations, but since UCT_{orig} and UCB_{orig} need

fewer simulations, they are actually much faster.

As mentioned in Section 3.6, the maximum allowed number of turns can also impact the speed, because a low number of maximum turns will let the rollout end before the game is actually finished. The impact of the maximum turns option is not tested, because all variants use a rollout policy which should simulate every game to the end, thus all variants use a value of 999 for maximum turns. In the future, one could implement this as a variable to be used with the rollout policy only, in order to allow simulations to stop at an earlier point, possibly adding in some extra points, based on the potential of each player's decks, to account for the early simulation finish.

5 Conclusion

During this thesis we have introduced four novel AIs for the game of Dominion using Monte-Carlo methods. The variants employ UCT or flat UCB, together with either the UCB1 selection formula or a slightly modified version. All variants were tested against two successful Dominion strategies for the chosen test-bed, as well as human players to some extent.

The two flat UCB-based AIs were capable of winning 68.5% games against the SingleWitch strategy (a finite-state machine based solution), while they won respectively 68.5% and 62.5% against the DoubleWitch strategy. The UCT-based solutions did not perform so well, winning respectively 35% and 54%, and 31% and 45.5% of the games played.

Testing two variants against human players, the best variant won 87.5% games, and the other 25%, against a group of experienced Dominion players (one player at a time), though the number of games played was too low to be conclusive.

This thesis shows that by increasing the value of selected cards, two variants were able to recognize the card by buying and playing it, this is likely applied to all variants, but only two variants were tested. This shows their potential to learn good cards, even with an incomplete overview of the complete game.

The speed of the main configuration for the variants using the UCB1 formula is approximately 1.2 seconds per move, while for variants using the modified version, the speed is about 8.4 and 9.1 seconds per move, without parallelization. Using parallelization, the time per move can approximately be divided by the number of threads used.

Finally, the variants were tested against each other, where the flat UCB variants performed the best, winning 74.5% and 80.1% games against their most similar UCT variants. The number of games played are however not enough to conclude the best variant.

A reason that flat UCB-based AIs outperform the UCT variants, could be that the greedy, or locally optimized choice, works well for the test-bed of Dominion cards used. This would lessen the need to plan far ahead, which is one of the strengths of UCT.

The UCT variants do however seem to be better at playing complex combinations of cards in sequence, and might perform differently when tested on other card setups.

The novel approaches for dealing with the stochastic element of drawing cards and interaction between players seem to work very well for the flat UCB variants, and fairly well for the UCT variants. The modified UCB1 formula seems to need more simulations before achieving the effect of UCB1, but can possibly outperform UCB1 when played directly against each other.

All in all, our solution is capable of competing with and outperform the top level strategies, and should be applicable to card setups extending the one used in this thesis. We believe that currently our solution is one of the best Dominion AIs.

5.1 Future Work

For future work, we would like to run some of the experiments with more games, in order to achieve more confident results. Playing more games with each variant against human players and other strategies would also be interesting to further benchmark performance. If possible, it would be interesting to test MCDomAI against the already existing Dominion AIs.

To test MCDomAI against the existing AIs, the rest of the cards in Dominion should also be added. This can possibly increase the need for more heuristics, especially during the rollout. If the extension of MCDomAI is successful, one could also extend the AI to for using the expansion packs of Dominion as well.

To improve MCDomAI in general, we would like to test out more UCT and UCB enhancements to possibly increase performance. For instance could such enhancements improve the rate at which MCDomAI plays multiple action cards are played in sequence.

References

[1] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine learning* 47.2-3 (2002), pp. 235–256.

[2] Peter Auer and Ronald Ortner. "UCB Revisited: Improved Regret Bounds for the Stochastic Multi-Armed Bandit Problem". In: *Periodica Mathematica Hungarica* 61.1-2 (2010), pp. 55–65.

[3] *Base Game Kingdom Cards*. 2010. URL: `http://dominioncg.wikia.com/wiki/Base_Game_Kingdom_Cards` (visited on 05/30/2014).

[4] Amine Bourki et al. "Scalability and Parallelization of Monte-Carlo Tree Search". In: *Computers and Games*. Springer, 2011, pp. 48–58.

[5] Cameron Browne. "The Dangers of Random Playouts". In: *ICGA Journal* 34.1 (2011), pp. 25–26.

[6] Cameron B Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *Computational Intelligence and AI in Games, IEEE Transactions on* 4.1 (2012), pp. 1–43.

[7] Tristan Cazenave and Nicolas Jouandeau. "A Parallel Monte-Carlo Tree Search Algorithm". In: *Computers and Games*. Springer, 2008, pp. 72–80.

[8] Tristan Cazenave and Nicolas Jouandeau. "On the Parallelization of UCT". In: *Proceedings of the Computer Games Workshop*. 2007, pp. 93–101.

[9] Guillaume Chaslot. "Monte-Carlo Tree Search". PhD thesis. Maastricht University, 2010.

[10] Guillaume M JB Chaslot et al. "Progressive Strategies for Monte-Carlo Tree Search". In: *New Mathematics and Natural Computation* 4.03 (2008), pp. 343–357.

[11] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. "Parallel Monte-Carlo Tree Search". In: *Computers and Games*. Springer, 2008, pp. 60–71.

[12] Wizards of the Coast. *Magic: The Gathering*. 2013. URL: `http://www.wizards.com/Magic/Summoner/` (visited on 05/30/2014).

[13] Pierre-Arnaud Coquelin and Rémi Munos. "Bandit Algorithms for Tree Search". In: *arXiv preprint cs/0703062* (2007).

[14] Peter I Cowling, Colin D Ward, and Edward J Powley. "Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering". In: *Computational Intelligence and AI in Games, IEEE Transactions on* 4.4 (2012), pp. 241–257.

[15] Mark Diehr. *Dominion: Base*. URL: `http://dominion.diehrstraits.com/?set=Base` (visited on 05/30/2014).

[16] *Dominion Online*. URL: `www.playdominion.com` (visited on 05/30/2014).

[17] Hilmar Finnsson. "CADIA-Player: A General Game Playing Agent". MA thesis. Reykjavik University, School of Computer Science, 2007.

[18] Hilmar Finnsson and Yngvi Björnsson. "Learning Simulation Control in General Game-Playing Agents." In: *AAAI*. Vol. 10. 2010, pp. 954–959.

[19] Hilmar Finnsson and Yngvi Björnsson. "Simulation-Based Approach to General Game Playing." In: *AAAI*. Vol. 8. 2008, pp. 259–264.

[20] Matthew Fisher. *Provincial: A Kingdom-Adaptive AI for Dominion*. 2014. URL: http://graphics.stanford.edu/~mdfisher/DominionAI.html (visited on 05/30/2014).

[21] Rasmus Bille Fynbo and CS Nellemann. "Developing an Agent for Dominion using Modern AI-Approaches". MA thesis. IT University of Copenhagen, 2010.

[22] Rio Grande Games. *Dominion*. 2014. URL: http://riograndegames.com/Game/278-Dominion (visited on 05/30/2014).

[23] Sylvain Gelly. "A Contribution to Reinforcement Learning; Application to Computer-Go". PhD thesis. Universite Paris-Sud, 2007.

[24] Sylvain Gelly and David Silver. "Combining Online and Offline Knowledge in UCT". In: *Proceedings of the 24th International Conference on Machine Learning*. ACM. 2007, pp. 273–280.

[25] Sylvain Gelly and Yizao Wang. "Exploration Exploitation in Go: UCT for Monte-Carlo go". In: (2006).

[26] Catan GmbH. *The Official Website for The Settlers of Catan*. 2014. URL: http://www.catan.com/ (visited on 06/01/2014).

[27] John Michael Hammersley and David Christopher Handscomb. *Monte carlo methods*. Vol. 1. Springer, 1964.

[28] Jing Huang et al. "Pruning in UCT Algorithm". In: *Technologies and Applications of Artificial Intelligence (TAAI), 2010 International Conference on*. IEEE. 2010, pp. 177–181.

[29] IBM. *Deep Blue*. URL: http://www-03.ibm.com/ibm/history/exhibits/vintage/vintage_4506VV1001.html (visited on 05/30/2014).

[30] M. H. Kalos. "Monte Carlo Methods". In: *Edward Teller Centennial Symposium - Modern Physics and the Scientific Legacy of Edward Teller*. World Scientific Publishing Co. Pte. Ltd., 2010, pp. 128–136.

[31] Donald E Knuth and Ronald W Moore. "An Analysis of Alpha-Beta Pruning". In: *Artificial intelligence* 6.4 (1976), pp. 293–326.

[32] Levente Kocsis and Csaba Szepesvári. "Bandit based monte-carlo planning". In: *Machine Learning: ECML 2006*. Springer, 2006, pp. 282–293.

[33] Tomas Kozelek. "Methods of MCTS and the Game Arimaa". MA thesis. Charles University, Prague, Faculty of Mathematics and Physics, 2009.

[34] Lars Schaefers, Marco Platzner, and Ulf Lorenz. "UCT-Treesplit-Parallel MCTS on Distributed Memory". In: *Proc. 21st Int. Conf. Automat. Plan. Sched., Freiburg, Germany.* 2011.

[35] Nicolae Sfetcu. *The Game of Chess*. Nicolae Sfetcu, 2014.

[36] Arthur Smith. *Game of Go, The*. Tuttle Publishing, 1956.

[37] Rob Speer. *Dominate - A Simulator for Dominion Strategies.* 2014. URL: `http://rspeer.github.io/dominiate/play.html` (visited on 05/30/2014).

[38] István Szita, Guillaume Chaslot, and Pieter Spronck. "Monte-Carlo Tree Search in Settlers of Catan". In: *Advances in Computer Games*. Springer, 2010, pp. 21–32.

[39] William R Thompson. "On the Likelihood that one Unknown Probability Exceeds Another in View of the Evidence of two Samples". In: *Biometrika* (1933), pp. 285–294.

[40] Donald X. Vaccarino. *Game Rules for Dominion.* 2008. URL: `http://riograndegames.com/uploads/Game/Game_278_gameRules.pdf` (visited on 05/30/2014).

[41] Colin D Ward and Peter I Cowling. "Monte Carlo Search Applied to Card Selection in Magic: The Gathering". In: *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE. 2009, pp. 9–16.

Appendices

A Big Money Finite-State Machine (BMFSM) Algorithm

The test opponent used to play against MCDomAI during parameter setups is a rule-based AI, consisting of a small number of conditional statements. BMFSM is not meant to compete at human-level, as it does not buy nor play any action cards. BMFSM is based upon the strategy called Big Money, which mostly consists of buying Province, Gold or Silver cards, in that order, depending on how much money is available. The general idea is to buy the best treasure cards available until turn 15, and then buying the best victory cards available. Note that the Gardens card is valued higher than the Duchy card, since BMFSM will often have time to buy 20 cards or more. Pseudo-code is given in Algorithm 2.

```
if Turn > 15 then
    if coins >= 8 then
        buy Province;
    else if coins >= 4 then
        buy Gardens;
    else if coins >= 5 then
        buy Duchy;
    else if coins >= 2 then
        buy Estate;
    else
        end turn;
    end
else
    if coins >= 6 then
        buy Gold;
    else if coins >= 3 then
        buy Silver;
    else
        end turn;
    end
end
```

Algorithm 2: BMFSM choosing a card during buy phase.

BMFSM is set to play for games with length about 30-40 turns, which is probably a little longer than what most games last. Being unable to determine game length itself, BMFSM is not capable of competing with humans and stronger AIs, but is a sufficient opponent for parameter setups.

B SingleWitch and DoubleWitch Algorithm

The algorithm for the successful Single- and DoubleWitch buy strategies is given in Algorithm 3. The algorithms do not buys Gardens cards, thus in some situations they could score a few points more if they did. During the action phase, they always play a Witch card if possible. Note that the only difference is whether to buy one or two Witch cards.

```
if coins >= 8 and totalDeck.goldCards > 0 then
  | buy Province;
else if coins >= 5 and totalDeck.witchCards < X then
  | buy Witch;
else if coins >= 5 and supply.provinceCards < 4 then
  | buy Duchy;
else if coins >= 2 and supply.provinceCards < 2 then
  | buy Estate;
else if coins >= 6 then
  | buy Gold;
else if coins >= 3 then
  | buy Silver;
else
  | end turn;
end
```

Algorithm 3: SingleWitch and DoubleWitch buy algorithm. X is respectively 1 and 2 for SingleWitch and DoubleWitch.

C Cards Chosen for the Test-Bed

The different cards in the base game of Dominion belongs to one of the following four groups:

1. Action cards
2. Treasure cards
3. Victory cards
4. Curse cards

C.1 Action Cards

Figure 27: Village: +1 Card, +2 Actions; Woodcutter: +1 Buy, +2 Coins.

Figure 28: Moneylender: Trash a Copper card from your hand. If you do: +3 Coins; Remodel: Trash a card from your hand. Gain a card costing up to 2 Coins more than the trashed card; Smithy: +3 Cards.

Figure 29: Festival: +2 Actions, +1 Buy, +2 Coins; Laboratory: +2 Cards, +1 Action; Market: +1 Card, +1 Action, +1 Buy, +1 Coin; Witch: +2 Cards. Each other player gains a Curse card.

C.2 Treasure Cards

Figure 30: Copper, Silver and Gold. Worth respectively 1, 2 and 3 coins.

C.3 Victory Cards

Figure 31: Estate, Duchy, Province and Gardens. Gardens is worth 1 VP for every 10 cards in your deck (rounded down).

C.4 Curse Cards

Figure 32: Curse is worth -1 VP.