Universiteit Leiden
The Netherlands

Opleiding Informatica

Strategies for the

Deck Building Game Dominion

Femke Slangen

Supervisors:
Walter Kosters & Luc Edixhoven

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

April 4, 2023

Abstract

DOMINION is a card game where during the game each player builds their own Deck. The goal is to have the most Victory Points at the end of the game. These points can be obtained by purchasing Victory Cards. But only purchasing these cards, and not others can decrease purchasing power. This makes the ratio of different types of cards important.

In this thesis strategies for DOMINION will be explored. Not only will simple strategies such as Random and a Smart player be implemented, but also several Monte Carlo Strategies, consisting of the standard strategy, a Double Monte Carlo strategy which has a small variation on the standard and finally Monte Carlo Tree Search. The last strategy which will be tested is a strategy using a Neural Network.

When testing these strategies, the Smart Strategy performed really well. Not only does it score well on its own, but when Monte Carlo Strategies use the Smart Strategy for their runs the resulting scores are even higher.

Contents

1 Introduction

DOMINION was created by Donald X. Vaccarino in 2008 [Gam23] and is the first Deck Building Game. Deck Building Games are card games where each player has their own Deck. These Decks start out the same, consisting of the weakest cards. Throughout the game, these Decks will change based on the player's decisions.

Each turn consists of three different phases. In order, they are: Action Phase, Buy Phase and Clean-up Phase. Any cards played during the Action and Buy Phases, and all cards remaining in the Hand are moved to the Discard Pile during the Clean-up Phase. After this five new cards are drawn into the player's Hand.

At the end of the game, the player with the most Victory Points wins. These Victory Points are obtained at the end of the game from Victory Cards. Victory Cards, and other kinds of cards, can be purchased during the game. Only cards in the player's Hand can be used to help make such purchases.

Figure 1: Pictured are a Discard Pile (stack bottom left), Deck (stack bottom right) and Table after the Action Phase [Mar23].

There are different types of cards, in Figure 1 they are shown from top to bottom. They are Treasure Cards, Action Cards, and Victory Cards. Treasure Cards are used to make purchases. During the Buy Phase, any Treasure Cards in the player's Hand can be used to generate Coins. These Coins can then be used to make a purchase.

Another type of card is an Action Card, which can only be used during the Action Phase. These cards have a wide variety of uses. Typical functions are drawing more cards or increasing the number of Coins, Actions, or Purchases which can be made.

Victory Cards however do not have an active use during the game. This means only buying Victory Cards does not lead to the best result, since this will shift the card ratio towards Victory Cards, resulting in fewer Treasure and Action Cards on average for each turn. This in turn decreases the potential to buy more Victory Cards. At the same time, if no Victory Cards are purchased, there is no way to win.

To explore this card ratio and strategies which are focused on Victory Point collection, this thesis will limit the scope of DOMINION to a single-player version. Action Cards with effects for other players have been removed. The end of the game will be determined by a turn limit, at which point the score will be calculated.

In this thesis, different strategies will be explored to see which strategy works best, and what effect Action Cards have on the performance of different strategies. Not only will simple strategies be used, like Random and Smart, but also multiple Monte Carlo variants and a Neural Network have been implemented.

In the following section, the game components, structures and rules are defined. In Section 3 the strategies used in this paper are explained. Next, in Section 4 these strategies are tested using different variables and compared to each other. Finally, Section 5 concludes this thesis.

This bachelor thesis has been created under the supervision of Walter Kosters and Luc Edixhoven at the Leiden Institute of Advanced Computer Science of Leiden University.

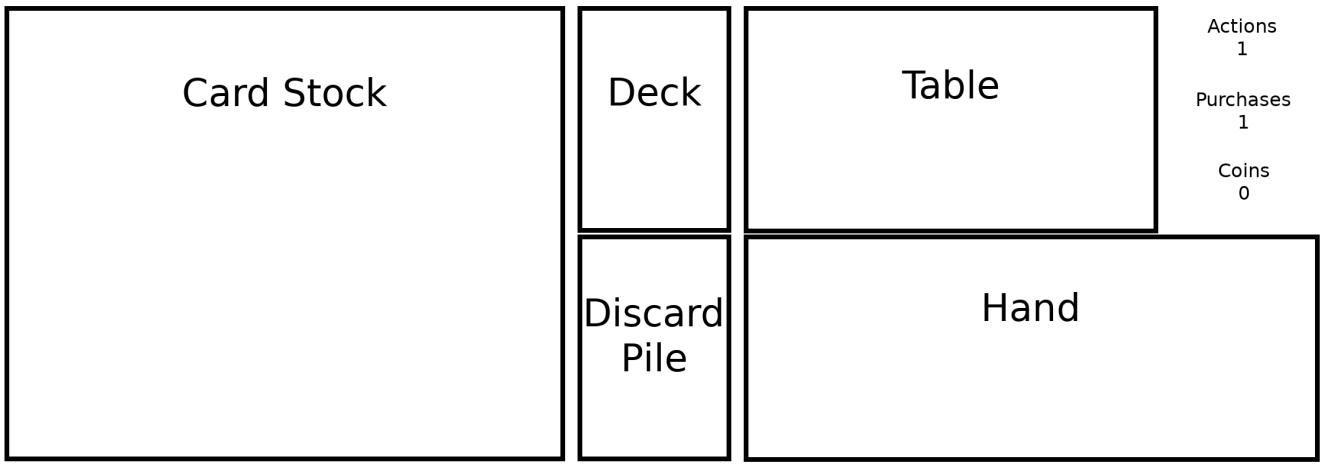


Figure 2: Structure of DOMINION visualized.

2 Definitions

In this section, the components, structures and rules of the game will be defined.

2.1 Game Structures

This section defines the structures used to play DOMINION, see also Figure 2.

Deck

The Deck is an ordered multiset of cards. At the beginning of the game, the Deck consists of $t = 7$ of the cheapest Treasure Cards and $v = 3$ of the cheapest Victory Cards, in random order. The order of the set of cards is not known to the player.

If a card needs to be taken from the Deck and it is empty, all cards in the Discard Pile are shuffled and moved to the bottom of the Deck. After this, the card can be taken from the Deck. If the Discard Pile is also empty, nothing happens.

Hand

The Hand is an unordered multiset of cards which is known to the player. Cards in the Hand can be used during a turn. At the start of the turn, the Hand consist of $h = 5$ cards. This number can increase or decrease during the turn.

Discard Pile

The Discard Pile is an unordered multiset of cards, which consists of cards that have been discarded. The Discard Pile is initially empty.

Table

The Table is an unordered multiset of cards. At the beginning of each turn, the Table is empty. Any cards played from the Hand end up on the Table. At the end of the turn, all cards on the Table are moved to the Discard Pile.

Card Stock

The Card Stock is a multiset which contains all cards which can be bought. For each card in the Victory Card Set, the Treasure Card Set and the Action Card Set, there are respectively $a\_v = 8$, $a\_t = 16$, $a\_a = 5$ versions of it in the Card Stock.

Actions, Purchases and Coins

For *Actions*, *Purchases* and *Coins* integers are used. At the start of each turn *Actions* is set to $i\_a = 1$, *Purchases* is set to $i\_p = 1$, and *Coins* is set to $i\_c = 0$.

2.2 Game Components/Cards

There are three different kinds of cards in DOMINION: Victory Cards, Treasure Cards and Action Cards. This section contains their definitions and the exact cards used.

Victory Cards

Victory Cards have a Victory Point Value which at the end of the game together determines the score. During the game, these cards have no use. The set of Victory Cards VC is a non-empty set of cards. The elements of VC are of the form $(Cost, PointValue)$. In DOMINION the Victory Card set is $\{(2,1),(5,3),(8,6)\}$.

Treasure Cards

Treasure Cards have a Coin Value which can be used to buy cards. The set of Treasure Cards TC is a non-empty set of cards. The elements of TC are of the form $(Cost, CoinValue)$. In DOMINION the Treasure Card set is $\{(0,1),(3,2),(6,3)\}$.

Action Cards

There are different kinds of Action Cards, each kind with its own function. These functions can consist of different effects or have their own unique effect. Effects which occur on multiple Action Cards are:

- Increase the number of Actions, *Actions* $\leftarrow$ *Actions + increase*
- Increase the number of Purchases, *Purchases* $\leftarrow$ *Purchases + increase*
- Increase the number of Coins, *Coins* $\leftarrow$ *Coins + increase*
- Take the top card of the Deck and take it into your Hand, *Draw 1 card from Deck*
- Purchase a card for free, *Purchase 1 card worth at most cost Coins*

The set of Action Cards AC is a non-empty set of cards. The elements of AC are of the form (*Name*, *Cost*). In Table 1 all Action Cards are shown with their associated pseudo code. For each card, the name of the card and its cost are shown on the first line.

The selection of Action Cards only includes cards which do not influence other players, Action Cards which destroy cards have also been avoided. From the remaining Action Cards a balanced selection has been made.

2.3 Game Rules

After the game has been set up, each turn consists of three phases, the Action Phase, Buy Phase and Clean-up Phase. These phases are repeated until the end of the game.

Festival: 5 *Actions* ← *Actions* + 2 *Purchases* ← *Purchases* + 1 *Coins* ← *Coins* + 2	**Market: 5** Draw 1 card from Deck *Actions* ← *Actions* + 1 *Purchases* ← *Purchases* + 1 *Coins* ← *Coins* + 1
Village: 3 Draw 1 card from Deck *Actions* ← *Actions* + 2	**Laboratory: 5** Draw 2 cards from Deck *Actions* ← *Actions* + 1
Smithy: 4 Draw 3 cards from Deck	**Merchant: 3** Draw 1 card from Deck *Actions* ← *Actions* + 1 *ExtraCoinFirstSilver* ← True
Throne Room: 4 *PickedCard* ← input() //input Call function of *PickedCard* Call function of *PickedCard*	**Workshop: 3** Purchase 1 card worth at most 4 Coins //input Move purchased card to DiscardPile

Vassal: 3
Coins ← *Coins* + 2
Card ← top card of Deck
if *Card* is an Action_Card:
  use ← input() //input
  if *use*:
  Call function of *Card*
  Move 1 card from Deck to Table
  else:
  Move 1 card from Deck to DiscardPile
else:
  Move 1 card from Deck to DiscardPile
	Library: 5
	while sizeof(Hand) ≤ sizeof(Hand) + 2
	  Card ← top card of Deck
	  if *Card* is an Action_Card:
	  keep ← input() //input
	  if *keep*:
	  Move 1 card from Deck to Hand
	  else:
	  Move 1 card from Deck to Temp
	  else:
	  Move 1 card from Deck to Hand
	Move all cards from Temp to DiscardPile

Table 1: Action Cards and their functions.

Set-up

Before the game begins the Deck is initialised with its $t = 7$ Treasure Cards and $v = 3$ Victory Cards. The Deck is then shuffled and the top five cards are moved to the Hand.

Action Phase

During the Action Phase, an action can be used to play an Action Card. The turn is started with $i\_a = 1$ Action, *Actions* ← 1. When an action is used the number of actions decreases by one, *Actions* ← *Actions* − 1. If *Actions* = 0, no more actions can be performed. Some Action Cards can be used to increase *Actions*.

Buy Phase

During the Buy Phase Treasure Cards can be used to generate coins, $Coins \leftarrow Coins + CoinValue$. These coins can be used in combination with a purchase to buy a card. The turn starts with $i\_p = 1$ purchases, $Purchases \leftarrow 1$. When a purchase is used the number of purchases decreases by one, $Purchases \leftarrow Purchases - 1$. If $Purchases = 0$, no more cards can be bought. During the Action Phase, some Action Cards can be used to increase the number of purchases which can be made.

Clean-up Phase

During the Clean-up Phase, any cards used during the turn and any cards remaining in the Hand are moved to the Discard Pile. Any remaining actions, purchases or coins are discarded, as *Actions*, *Purchases* and *Coins* are reinitialised. Next $h = 5$ cards are drawn from the Deck. If there are fewer than h cards in the Deck, the Discard Pile is shuffled and moved to the bottom of the Deck.

End of game

The game ends when the *TurnLimit* is reached. At this point, the Victory Points given by Victory Cards owned by the player are added up to become the final score.

2.4 Parameters

Table 2 contains an overview of all parameters used.

type	name	description	standard value
int	h	The number of Cards initially in a Hand.	5
int	t	The number of Treasure Cards in the start Deck.	7
int	v	The number of Victory Cards in the start Deck.	3
int	a_v	Initial stack height of Victory Cards in Card Stock.	8
int	a_t	Initial stack height of Treasure Cards in Card Stock.	16
int	a_a	Initial stack height of Action Cards in Card Stock.	5
int	i_a	The starting number of Actions.	1
int	i_p	The starting number of Purchases.	1
int	i_c	The starting number of Coins.	0
int	*Turn Limit*	Number of turns the game lasts.	10
set	VC	Contains the Victory Cards available during a game.	$\{(2,1),(5,3),(8,6)\}$
set	TC	Contains the Treasure Cards available during a game.	$\{(0,1),(3,2),(6,3)\}$
set	AC	Contains the Action Cards available during a game.	$\{(5,0),(5,1),(3,2),(5,3),(4,4),(3,5),(5,6),(3,7),(4,8),(3,9)\}$

Table 2: Parameters DOMINION.

3 Strategies

In this section, the strategies used in this thesis are explained. These strategies are used when a selection needs to be made. There are three different types of selections:

- **Selecting a card to Purchase**. Not only does this selection occur during the Buy Phase, but it can also occur during the Action Phase if the Action Card Workshop is used. In addition to the option to purchase no card, the other options depend on the cards available in the Card Stock and the maximum *Cost* the card may have. During the Buy Phase, the maximum *Cost* is the available Coins, otherwise, the maximum is decided by the Action Card.
- **Selecting an Action**. Apart from being used to select an Action Card to play using an Action, the Action Card Throne Room also requires such a selection. Apart from the option to use no Action Card, the other available options are the Action Cards in the Hand.
- **Selecting between two choices**. This occurs when the Vassal or Library card is used. The exact options depend on the card used.

3.1 Simple Strategies

Simple strategies use a small amount of computing power. This thesis uses two simple strategies, Random and Smart. In this section, these strategies are explained.

Random

For each type of selection, the Random strategy randomly selects an available option, see also Algorithm 1. Each option has the same odds of being selected, whether it is selecting a specific card or choosing none at all. This results in an increased chance of using an Action Card or purchasing a card if there are more cards available.

Smart

For a Smart Simple Strategy, a strategy has been created which has an improved performance and properly uses the Action Cards. As can be seen in Algorithm 2, the Smart strategy differs for each selection type:

- **Selecting a card to Purchase**. The *Cost* of all available cards are compared, and the most expensive card is purchased. If multiple cards are the most expensive, one of them is randomly selected.
- **Selecting an Action**. First, any Action Cards which increase the available Actions are used. This ensures the largest number of Action Cards possible are used. Next, the most expensive card is used, if there are multiple cards with the same *Cost*, one is randomly chosen.
- **Selecting between two choices**. The first option is always chosen. For the Vassal and Library, this means the card is respectively used or kept, instead of being discarded.

Algorithm 1 Pseudo-code Random

```
function SELECT(options[0 ... n − 1])
    k ← random(n)                ▷ random(n) returns a random integer smaller than n and ≥ 0.
    selection ← options[k]
    return selection
end function
```

Algorithm 2 Pseudo-code Smart

```
function SELECT_HIGHEST_COST(options[0 ... n − 1])
    best ← {}
    price ← 0
    for k ← 0 to n − 1 do
        if options[k].cost = price then
            best.add(options[k])
        else if options[k].cost > price then
            best ← {options[k]}
            price ← options[k].cost
        end if
    end for
    k ← random(sizeof(best))  ▷ random(n) returns a random integer smaller than n and ≥ 0.
    return best[k]
end function

function SELECT_PURCHASE(options[0 ... n − 1])
    selection ← SELECT_HIGHEST_COST(options)
    return selection
end function

function SELECT_ACTION(options[0 ... n − 1])
    Increases ← list of Action Cards which increase Actions
    for k ← 0 to n − 1 do
        if options[k] ∈ Increases then
            selection ← options[k]
            return selection
        end if
    end for
    selection ← SELECT_HIGHEST_COST(options)
    return selection
end function

function SELECT_BETWEEN(options[0, 1])
    selection ← options[0]
    return selection
end function
```

3.2 Monte Carlo Strategies

In Monte Carlo strategies, the game is played to completion using a simple strategy a predetermined number of times. This results in more computing power being used for a better result. In this section, the Monte Carlo strategies used are explained.

Standard Monte Carlo

When a selection needs to be made using the Monte Carlo strategy, each option gets evaluated. To determine which option will result in the best score, each option is sampled m_r times [Fis99]. This is achieved by playing the game till completion using a simple strategy. The resulting scores of these games added up give an indication of which moves result in better scores. The option with the best score is selected. In the rare case that multiple options have the same total score, the first option found is selected. Algorithm 3 contains the complete algorithm.

Algorithm 3 Pseudo-code Standard Monte Carlo

```
1: function SELECT(options[0 ... n − 1])
2:     best ← 0
3:     choice ← 0
4:     for k ← 0 to n − 1 do
5:         counter ← 0
6:         for i ← 0 to m_r do
7:             copy ← copy of current state of game
8:             copy ← IMPLEMENT_CHOICE(copy, options[k])
9:             score ← CompleteGameRandom(copy)
10:            counter ← counter + score
11:        end for
12:        if counter > best then
13:            best ← counter
14:            choice ← k
15:        end if
16:    end for
17:    selection ← options[choice]
18:    return selection
19: end function
```

Double Monte Carlo

The strategy for Double Monte Carlo is slightly different from Standard Monte Carlo. For selections with a small number of options, ≤ 2, Standard Monte Carlo is used. Otherwise, half the available Monte Carlo runs, m_r, are used to determine the better half of the options. These are then further explored with the remaining Monte Carlo runs.

Like the Standard Monte Carlo Algorithm, for the Double Monte Carlo Algorithm, which can be seen in Algorithm 4, there is also a preference towards earlier options. This preference is shown

in line 13, while determining the better half, and in line 27, when the best option is selected. On another note, this algorithm has been designed for the standard DOMINION setup. It functions on the assumption the *score* is always higher than 0, because all Decks start with three Victory Cards and there is no Action Card which destroys these cards.

Monte Carlo Tree Search

Like the other Monte Carlo algorithms the Monte Carlo Tree Search algorithm plays the game until completion a predetermined number of times. However, the runs are distributed in a completely different way. And as an extension, the game is played to completion in a slightly different way.

In the Monte Carlo Tree Search algorithm, a Tree is created to represent the current and future choices for the game. In this tree, the connections are the choices made, and the nodes are the resulting game states. Each node contains how often it has been visited, and the cumulative score. This allows choices with better scores to be explored more often. At the end, the most visited choice is selected.

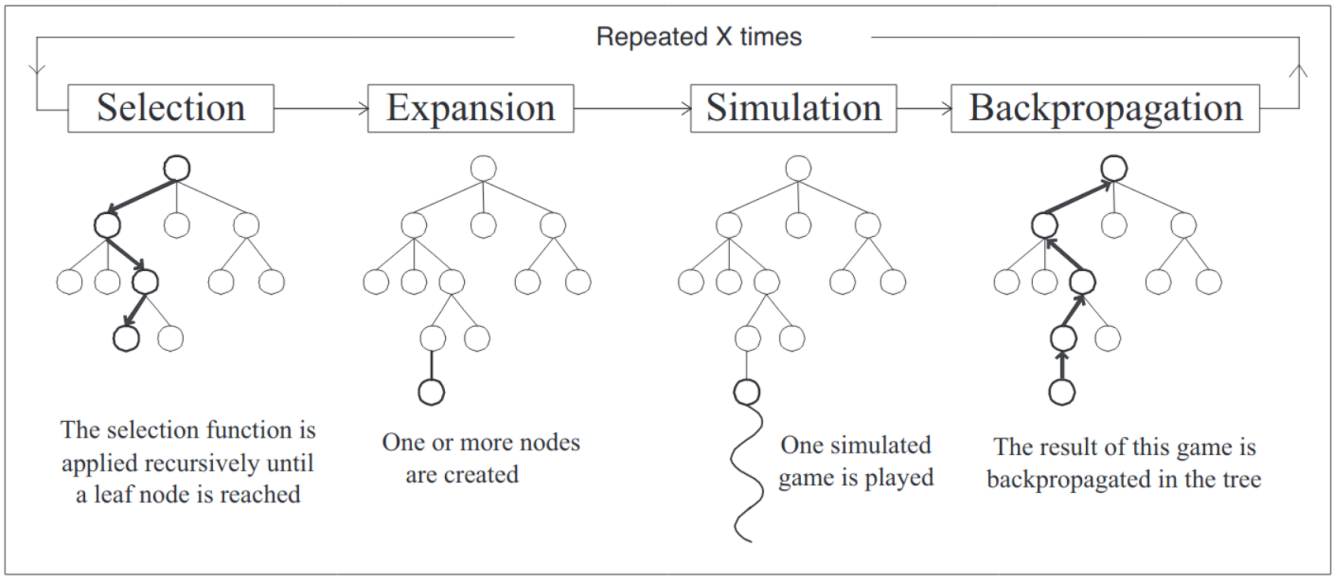


Figure 3: Outline of Monte-Carlo Tree Search [CBSS08].

The Monte Carlo Tree Search algorithm consists of four steps [RS19]:

- **Selection** During this step a leaf node is selected. A leaf node is a node with unexplored nodes.
- **Expansion** The selected node is expanded by creating a child node for one unexplored option.
- **Simulation** From the new child node, the game is played to completion using a simple strategy.
- **Back-propagation** The result of the game is back-propagated to all nodes on the taken path. The accumulated score gets updated, and the number of times the nodes have been visited.

Algorithm 4 Pseudo-code Double Monte Carlo

```
1: function SELECT(options[0 ... n − 1])
2:     counters[0 ... n − 1] ← {0, ..., 0}
3:     skip[0 ... n − 1] ← {1, ..., 1}
4:     for k ← 0 to n − 1 do
5:         for i ← 0 to m_r/2 do
6:             copy ← copy of current state of game
7:             copy ← IMPLEMENT_CHOICE(copy, options[k])
8:             score ← CompleteGameRandom(copy)
9:             counters[k] ← counters[k] + score
10:        end for
11:    end for
12:    for k ← 0 to (n + 1)/2 do
13:        best ← 0
14:        for i ← 0 to n − 1 do
15:            if skip[i] = 1 ∧ counters[i] > best then
16:                best ← counters[i]
17:                choice ← i
18:            end if
19:        end for
20:        skip[choice] ← 0
21:    end for
22:    best ← 0
23:    for k ← 0 to n − 1 do
24:        if skip ≠ 1 then
25:            for i ← 0 to m_r/2 do
26:                copy ← copy of current state of game
27:                copy ← IMPLEMENT_CHOICE(copy, options[k])
28:                score ← CompleteGameRandom(copy)
29:                counters[k] ← counters[k] + score
30:            end for
31:            if counters[k] > best then
32:                best ← counters[k]
33:                choice ← k
34:            end if
35:        end if
36:    end for
37:    selection ← options[choice]
38:    return selection
39: end function
```

The most important step is the selection step. During this step the balance between exploration and exploitation is important. Starting from the root of the tree, a node is selected. If it is a leaf node, the selection step is finished, otherwise, a new node is selected, in the same way. This selection is based on a formula which weighs exploration vs exploitation. Different formulas can be used to calculate the upper confidence index, the formula used in this thesis is UCB1 [ACBF02]:

$$value = wins/visits + 2 * \sqrt{\ln(total\_visits)/visits}$$

To use Monte Carlo Tree Search with DOMINION, the algorithm needs a slight adjustment to accommodate the non-determinism of DOMINION. During the expansion of a node, for each potential option, a child node is created. But when nodes are selected and a copy of the current game is played out, the nodes available are determined by the options that are actually possible. This means the cards drawn from the deck determine which options are available for selection. The final result can be seen in Algorithm 5.

Algorithm 5 Pseudo-code Monte Carlo Tree Search

```
1: function SelectUsingMCTS(options[0...n − 1])
2:     root ← New_Node() ▷ Contains cumulative score, number of visits and possible options.
3:     for k ← 1 to m_r * n do
4:         selected ← root
5:         copy ← copy of current state of game
6:         while Not_Leaf(selected) do
7:             selected ← Select_Child_Node(selected) ▷ Selects node with highest UCB value.
8:             copy ← Implement_Choice(copy, selected)  ▷ Implement choice in game copy.
9:         end while
10:        Expand_Node(selected)  ▷ Create child nodes for all options for selected node.
11:        selected = selected.children[0]  ▷ Select first child.
12:        copy ← Implement_Choice(copy, selected)
13:        if m_s = 0 then
14:            score ← CompleteGameRandom(copy)  ▷ Complete Random game from copy.
15:        else
16:            score ← CompleteGameSmart(copy)  ▷ Complete Smart game from copy.
17:        end if
18:        Back-propagate(selected, score)  ▷ Result is propagated to all nodes on taken path.
19:    end for
20:    selection ← 0
21:    for k ← 1 to n − 1 do
22:        if root.child[k].visits > root.child[selection].visits then
23:            selection ← k
24:        end if
25:    end for
26:    return selection
27: end function
```

3.3 Neural Network

The final strategy used in this thesis is to make selections using a Neural Network. A Neural Network is a form of Deep Learning since such networks consist of many layers [GBC16]. Neural networks have the ability to learn from input data. This allows the network to find connections which have not been specifically programmed for.

For better results, Neural Networks need to be trained to solve a single problem. Each output needs to be evaluated and punished or rewarded based on its desirability as a form of Reinforcement Learning [SB18]. The Neural Network will only be used to select a card to Purchase. For the other selections, a different algorithm can be used. Unless stated otherwise, the Smart algorithm is used.

The key element of a Neural Network is its structure. Neural Networks are composed of nodes connected by links. These links propagate the value of each node to the next. Each link has a weight which determines the strength and sign of the connection [RN20]. This weight is adjusted based on examples obtained from a data set.

In Figure 4 an example of a Neural Network is shown. As can be seen in the figure, Neural Networks consist of multiple layers:

- An input layer, which receives information fed into the network.
- Hidden layer(s), of which there can be many.
- And lastly the output layer, which gives the result of the Neural Network.

Each node in the network is connected to all nodes in the next layer.

To train the Neural Network we need a data set with input values and the correct output values. For each line of data, the input values are received by the Neural Network, and the output values are calculated. Next, these output values are compared to the correct output values, and the connections between nodes are adjusted based on the difference between those values.

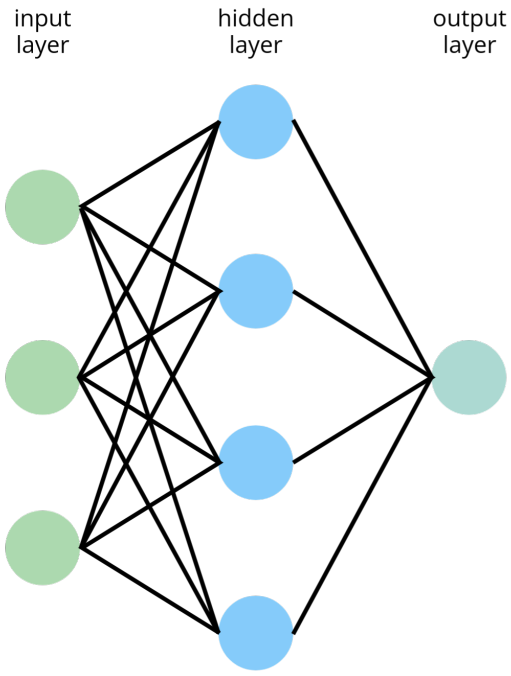


Figure 4: A simple Neural Network.

Before the Neural Network can be used to play Dominion, the Neural Network needs to be trained. The Neural Network will be trained to estimate the final score, based on a given state. The training data set is created by playing a number of games using either the Random or Smart strategy, since they can create a large data set in a small amount of time. For each game played the game states of each turn are stored in the data set with the final score. Next, the Neural Network is trained using this data set.

After the Neural Network has been trained, it can be used to play DOMINION. As seen in Algorithm 6, each time a selection needs to be made to Purchase a card, the Neural Network will first estimate the final score for each possible option. Next, the option with the best estimated score will be selected. The Neural Network only needs to be trained once, afterwards it can be used again and again. This means while using a Neural Network for the first time is time-consuming, each time afterwards it is as fast as a simple strategy.

For optimal results some changes, like a different *TurnLimit*, require the Neural Network to be trained anew. While some might not influence the network enough to be noticed. If a Network has been trained with an initial stack height of a hundred cards, one more or less will have little influence if it was impossible to buy all these cards anyway.

A change to the card sets TC, VC and AC requires a completely new Neural Network. If the length of one of these card sets is changed, the input generated by the game would not match the length required by the Neural Network. This can lead to crashes or undefined behaviour.

Keras had been used to train the Neural Network [Ker23]. Next frugally-deep [fde23] is used to evaluate the different options using the Neural Network SELECT_PURCHASE. One hidden layer is used, with two nodes.

Algorithm 6 Pseudo-code Neural Network

```
1: function SELECT_PURCHASE(options[0 ... n − 1])
2:     best ← 0
3:     choice ← 0
4:     for k ← 0 to n − 1 do
5:         state ← current game state
6:         expected_score ← NEURAL_NETWORK(state)
7:         if expected_score > best then
8:             best ← expected_score
9:             choice ← k
10:        end if
11:    end for
12:    selection ← options[choice]
13:    return selection
14: end function
```

4 Experiments

To discover which of the previously explained strategies performs the best, they will need to be compared to each other. Not only will they be compared with the standard parameters, but they will also be compared under other parameters. The card ratios will also be compared, to see if anything can be learned from these.

For more precise results each data point will represent the average result of at least $exp = 100$ games. This not only decreases the influence of outliers but also ensures these experiments will not take too long. In these experiments, unless stated otherwise, the standard parameters are used. For each experiment, the set of parameters will be tested with different values for the *TurnLimit*, ranging from 1 to 100.

4.1 Simple Strategies

First, the simple strategies will be tested. For extra precision, each data point is the average result of a thousand games, $exp = 1000$. We can use this many games because these simple strategies can be run quickly. To test the simple strategies Random and Smart, these strategies will be tested in three different scenarios:

- **Standard Parameters.** No changes have been made to the parameters.
- **Unlimited Stacks.** The stack height of Victory, Treasure, and Action Cards in the Card Stock are changed to a thousand, $i\_v = 1000$, $i\_t = 1000$, and $i\_a = 1000$. This makes it highly improbable that the end of a stack is reached within the given number of turns.
- **No Action Cards.** Instead of the standard set of AC, an empty set is used. $AC = \{\}$.

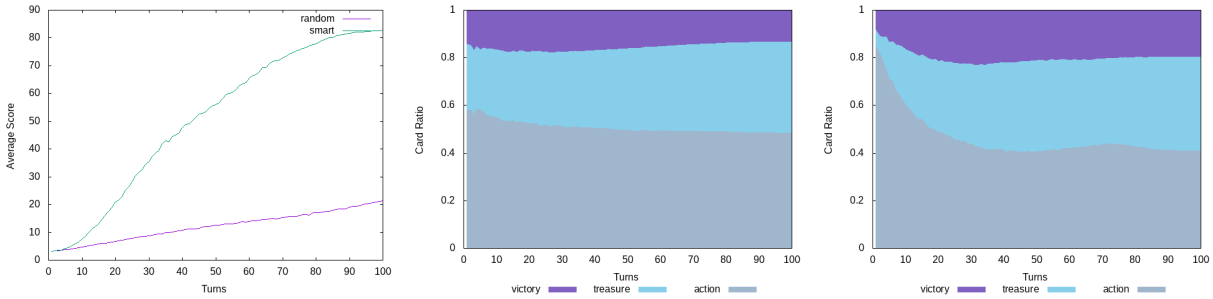
In Figures 5, 6 and 7 it can be seen that the Smart Strategy outperforms Random under all tested circumstances, these figures also show the card ratios. For the Standard Parameters and No Action Card Parameters, the maximum score which can be achieved is 83, this is the sum of all available Victory Cards. The algorithms perform better using the No Action Card Parameters, than the Standard Parameters. The number of options without Action Cards decreases significantly, from a maximum of seventeen to just seven. This means there is a larger chance a Victory Card will be bought, which improves the scores.

The card ratios of the Random Strategy show in what ratio cards are bought in purely random games. An interesting difference can be seen between the standard parameters and the unlimited stacks. This shows the influence of the size of the card stock. Interestingly for the Smart Strategy, during short games, $TurnLimit < 15$, there is a clear preference for Action Cards. For longer games, however, fewer Action Cards are bought, fewer than during the Random Strategy.

4.2 Monte Carlo

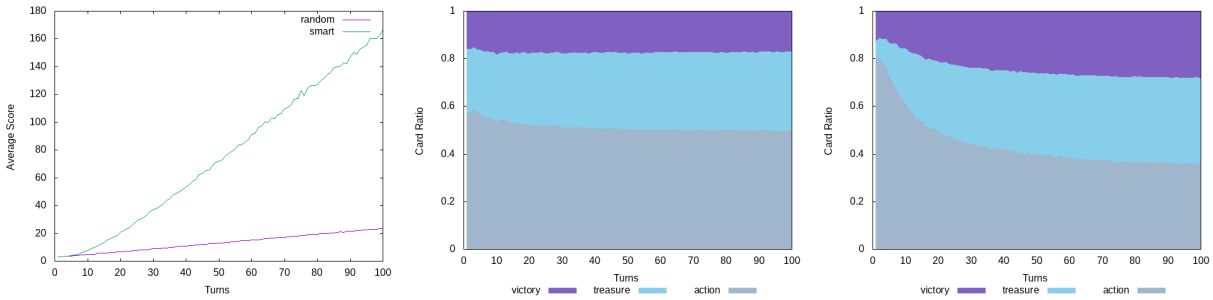
Before comparing Monte Carlo with the other strategies, it is important to discover which settings work best. Specifically how many Monte Carlo runs are needed, and which Simple Strategy works best. This will be tested with the standard parameters.

Based on the results shown in Table 3, not all parameters need to be as high as possible. When comparing the influence of the Monte Carlo Runs on the average score, after a hundred runs, any more has a minimal influence. In the following experiments, $m\_r = 100$ is used. Even this results in a program which takes significantly longer to run, since each turn the game is played till completion a hundred times per possible option. The Smart strategy also greatly increases the average score.



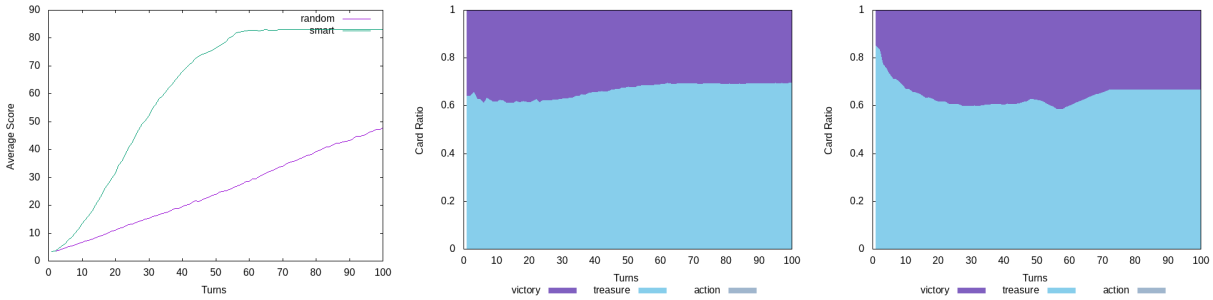
(a) Score (b) Ratio — Random (c) Ratio — Smart

Figure 5: Simple Strategies — Standard Parameters.



(a) Score (b) Ratio Random (c) Ratio Smart

Figure 6: Simple Strategies — Unlimited Stacks.



(a) Score (b) Ratio Random (c) Ratio Smart

Figure 7: Simple Strategies — No Action Cards.

		Random				Smart			
	$m\_r \setminus TL$	10	20	40	80	10	20	40	80
Monte Carlo Standard	10	11.00	21.80	30.20	62.30	15.70	30.70	45.80	63.75
	100	11.00	13.40	36.95	77.90	18.70	46.50	70.20	82.40
	500	11.40	14.15	38.00	83.00	18.65	47.90	75.80	83.00
	1000	11.25	13.85	39.35	83.00	19.05	49.35	75.25	83.00
Monte Carlo Double	10	10.05	15.20	29.15	64.10	15.60	34.15	50.45	68.60
	100	11.15	14.00	36.05	77.60	19.45	47.95	71.60	82.85
	500	11.55	15.20	39.05	83.00	18.80	47.90	75.65	83.00
	1000	11.35	14.90	37.10	83.00	18.30	47.90	74.75	83.00
Monte Carlo Tree Search	10	11.15	14.30	27.50	50.30	10.75	30.40	45.85	61.50
	100	11.10	12.95	33.35	58.10	17.45	42.05	71.95	81.20
	500	11.45	13.25	34.70	67.10	15.95	46.40	74.00	82.85
	1000	11.85	12.35	37.85	69.20	15.70	45.05	73.50	81.95

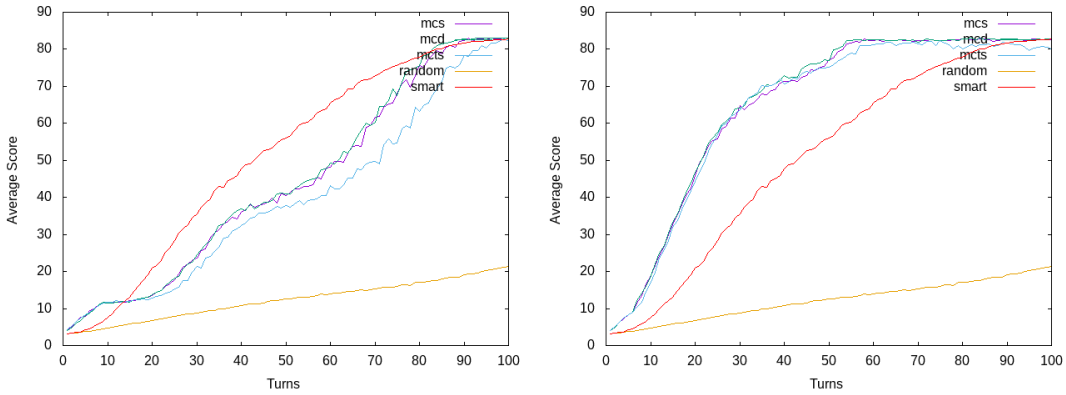
Table 3: Result using different Parameters, $TL = TurnLimit$

Next, the performance of the different Monte Carlo Strategies will be compared with each other. These strategies will be tested using the same scenarios as the Simple Strategies: **Standard Parameters**, **Unlimited Stacks**, and **No Action Cards**. The results of the Simple Strategies are also shown in the graphs showing average scores to allow for easier comparison. The graphs showing the average scores can be seen in Figures 8, 9 and 10, the card ratios can be seen in Figures 11 to 16.

For short games, $TurnLimit < 15$, the Monte Carlo Strategies outperform both Simple Strategies. The reason why can be seen in the card ratios. Here it can be seen that in such short games mostly Victory Cards are bought. Comparatively for simple strategies only about 20-40% of cards bought are Victory Cards.

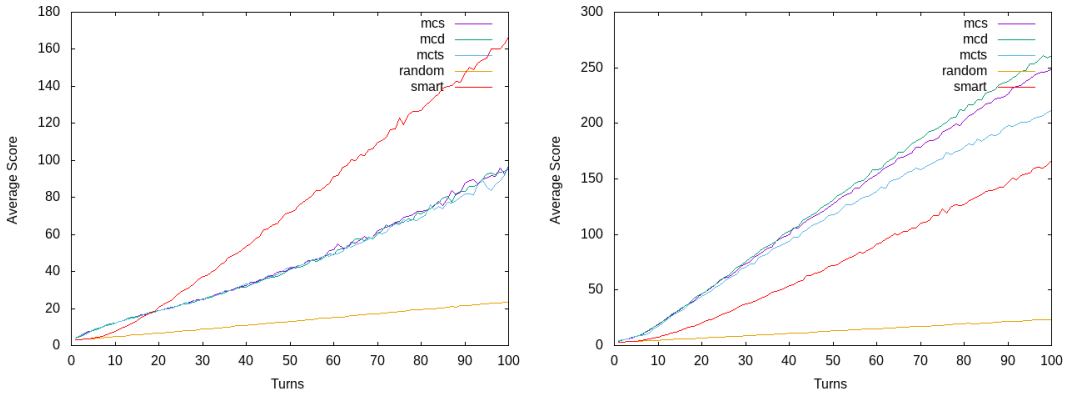
For longer games, the Simple Strategy used vastly influences the performance. While the Random Monte Carlo Strategies vastly outperform the Random Strategy, it falls short compared to the Smart Strategy. The Smart Monte Carlo Strategies vastly outperform both strategies. Only when the game contains no Action Cards does the performance of the Simple Strategy come close, the card ratios for these are also very similar for longer games.

Another interesting observation can be made when looking at the card ratios of the Smart Monte Carlo Strategies. For longer games, each of these strategies has around the same ratio of Victory Cards, roughly around 40%, this only changes when they have all been bought. This also creates an interesting comparison between the Victory Card ratios of the Smart and Random Monte Carlo Strategies, even though the Random Monte Carlo Strategies have a higher Victory Card Ratio, the average score is lower. This means more cards have been bought per turn by the Smart Monte Carlo Strategies.



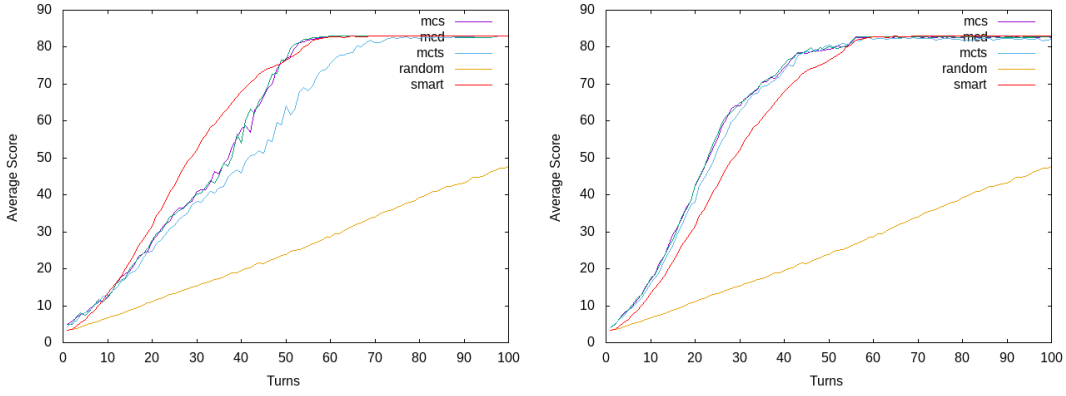
(a) Random (b) Smart

Figure 8: Average Score — Monte Carlo Strategies — Standard Parameters.



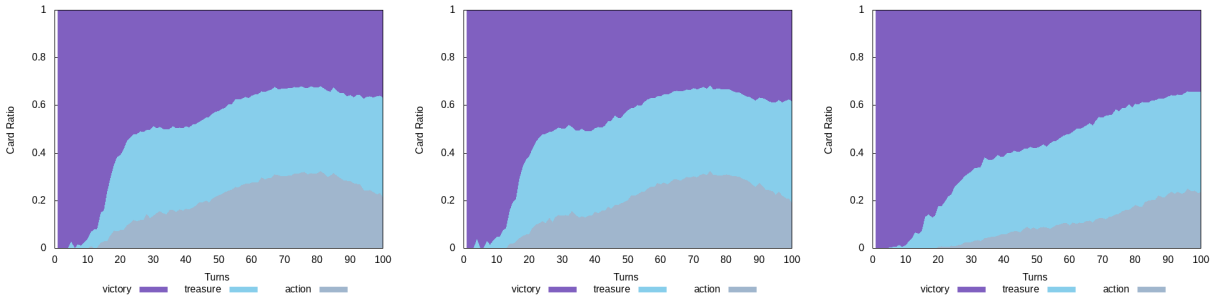
(a) Random (b) Smart

Figure 9: Average Score — Monte Carlo Strategies — Unlimited Stacks.



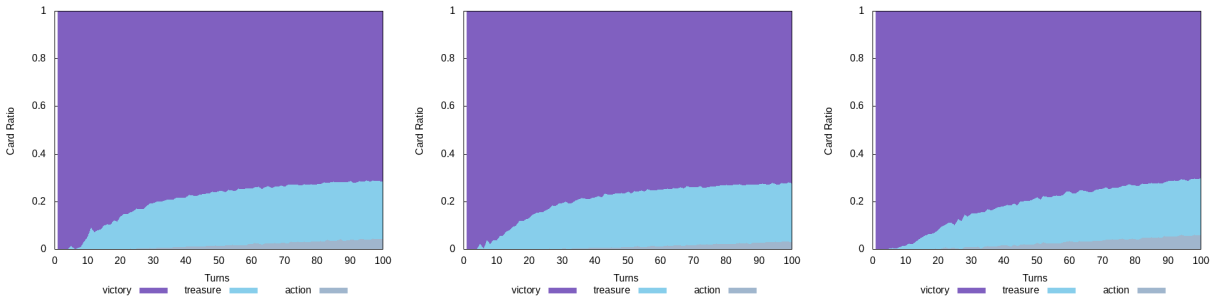
(a) Random (b) Smart

Figure 10: Average Score — Monte Carlo Strategies — No Action Cards.



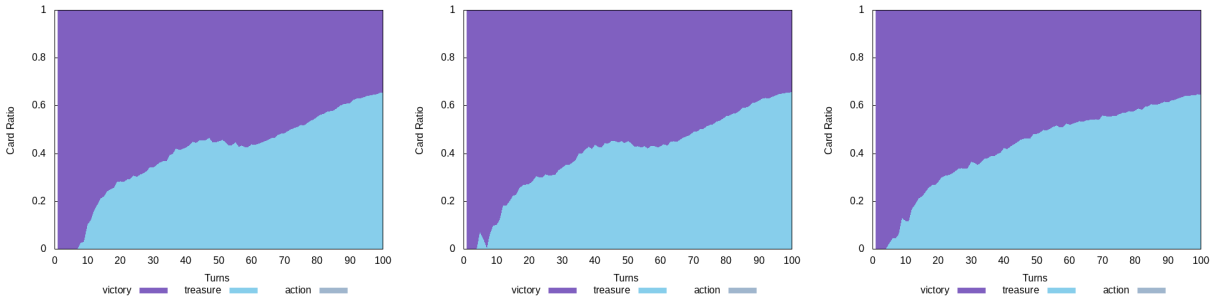
(a) Standard (b) Double (c) Tree Search

Figure 11: Card Ratio — Monte Carlo Strategies — Random — Standard Parameters.



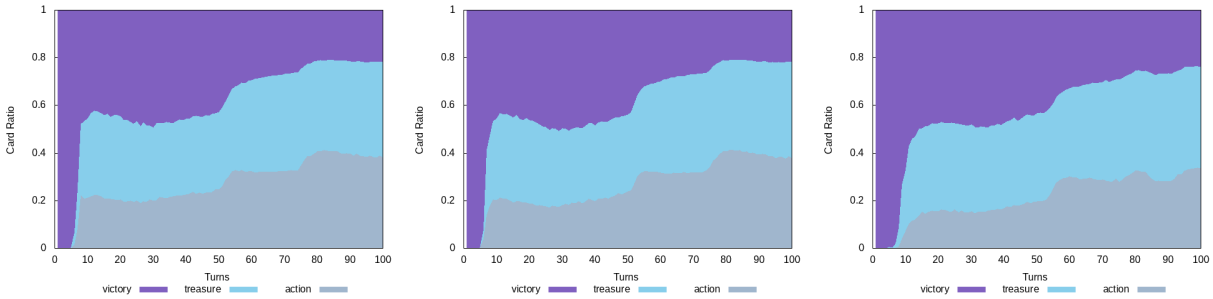
(a) Standard (b) Double (c) Tree Search

Figure 12: Card Ratio — Monte Carlo Strategies — Random — Unlimited Stacks.



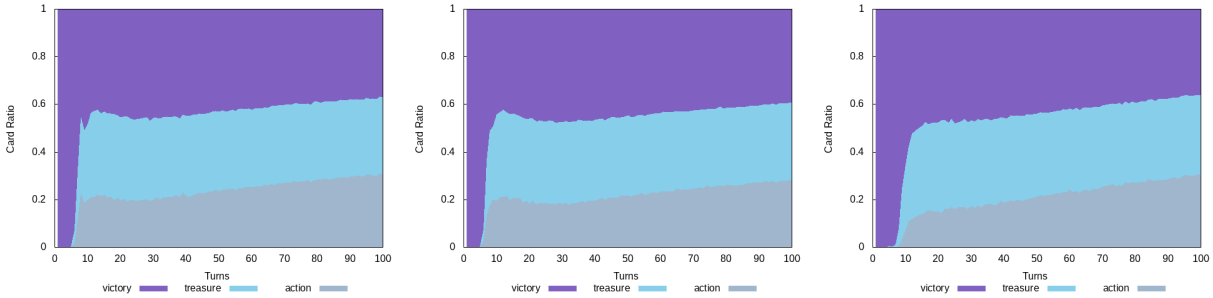
(a) Standard (b) Double (c) Tree Search

Figure 13: Card Ratio — Monte Carlo Strategies — Random — No Action Cards.



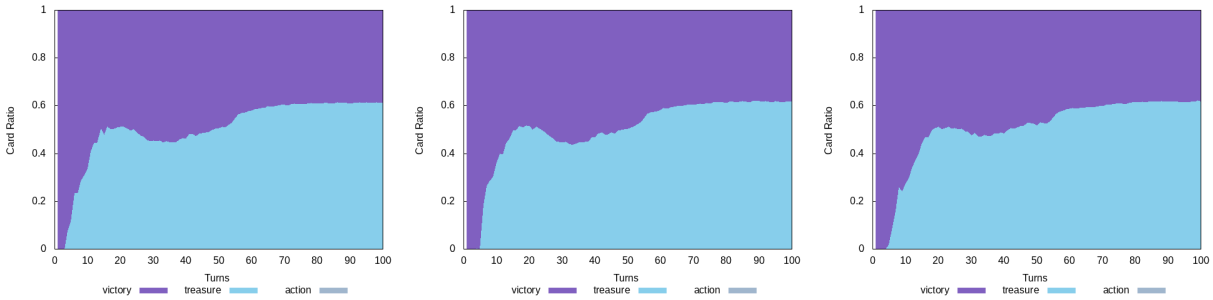
(a) Standard (b) Double (c) Tree Search

Figure 14: Card Ratio — Monte Carlo Strategies — Smart — Standard Parameters.



(a) Standard (b) Double (c) Tree Search

Figure 15: Card Ratio — Monte Carlo Strategies — Smart — Unlimited Stacks.



(a) Standard (b) Double (c) Tree Search

Figure 16: Card Ratio — Monte Carlo Strategies — Smart — No Action Cards.

4.3 Neural Network

Neural Networks can be trained on data sets generated by different strategies, namely Random, Simple or a Neural Network. To explore the difference in performance, these different strategies will be tested and compared with each other. The Neural Network strategy used to generate data sets will be trained using a Random Strategy. During testing for each game, a new data set is created, and a new Neural Network will be trained. To generate the data set, the game is played to completion $exp = 100$ times. To show the influence of the size of the data set, graphs showing the average score of the neural networks using $exp = 1000$ will also be created.

The Neural Network strategies will be tested using the same scenarios as the Simple Strategies: **Standard Parameters**, **Unlimited Stacks**, and **No Action Cards**. The results of the Simple Strategies are also shown in the graphs showing average scores to allow easier comparison. In Figures 17, 18 and 19 the performance of the Neural Networks can be seen, Figures 20, 21 and 22 show the card ratios.

When these scores obtained using $exp = 1000$ are compared to the results from $exp = 100$ an interesting difference can be seen. For the Neural Networks created using $exp = 100$ the performance during the Unlimited and No Action Cards scenarios is as good as random. For the Neural Networks using $exp = 1000$ the performance during the No Action Card scenario is better than random. This shows a larger data set improves the performance of the Neural Network.

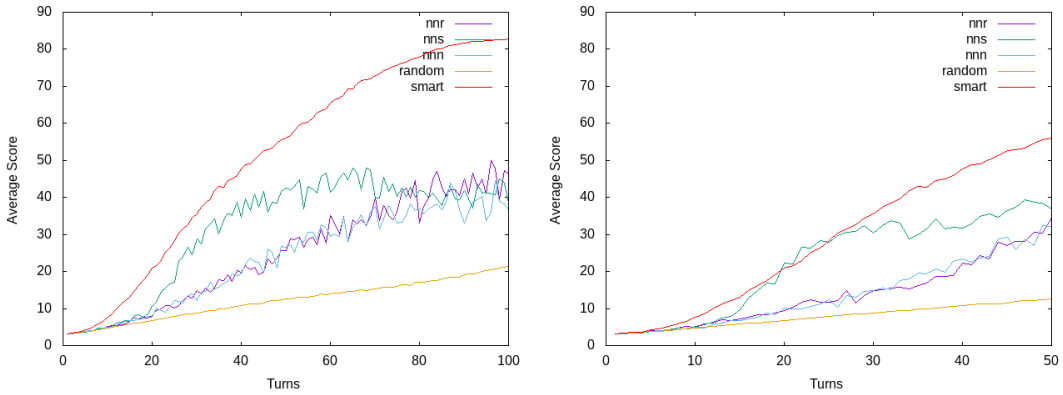
The drawback however is longer training times. In the time required to train the Neural Networks with the smaller data sets to $TurnLimit = 65$, the Neural Networks using the larger data sets only reached $TurnLimit = 33$.

While the card ratios of the Neural Networks are quite similar, their results are not. This can be explained by how the cards are used, when cards are bought and what value they have.

5 Conclusions and Further Research

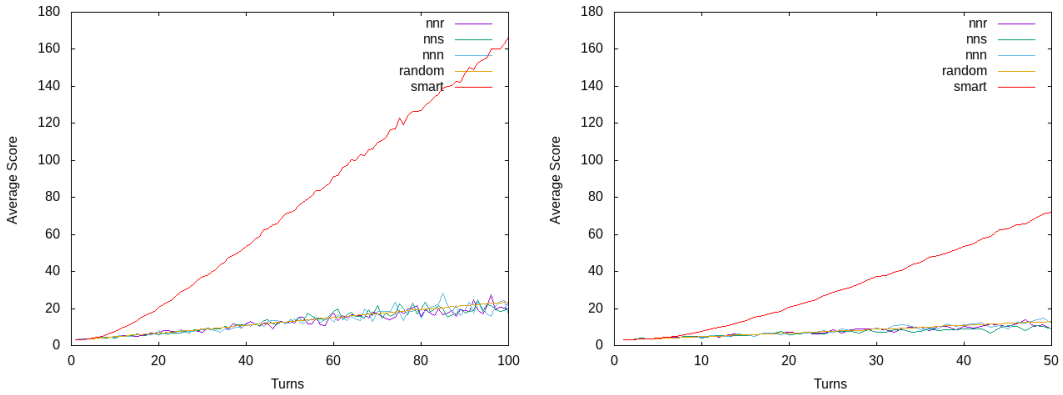
To discover which strategies work best for Deck Building Game Dominion, many algorithms have been tested. Not only does this include Simple Strategies, such as Random and Smart, but also Monte Carlo strategies were tested. This includes the Standard Monte Carlo Strategy, but also a Double Monte Carlo Strategy and Monte Carlo Tree Search. These Monte Carlo Strategies were run using Random or Smart. Finally, Neural Networks were also tested. These Neural Networks were trained using the Simple Strategies Random and Smart or using the output of a Neural Network trained using Random.

After comparing the results of these strategies in different scenarios, the best results are achieved by the Smart Monte Carlo Strategies. These strategies use the Smart Strategy to complete the game $m\_r = 100$ times for each option available, choosing the option which performs best. This allows the algorithm to choose the best most expensive card.



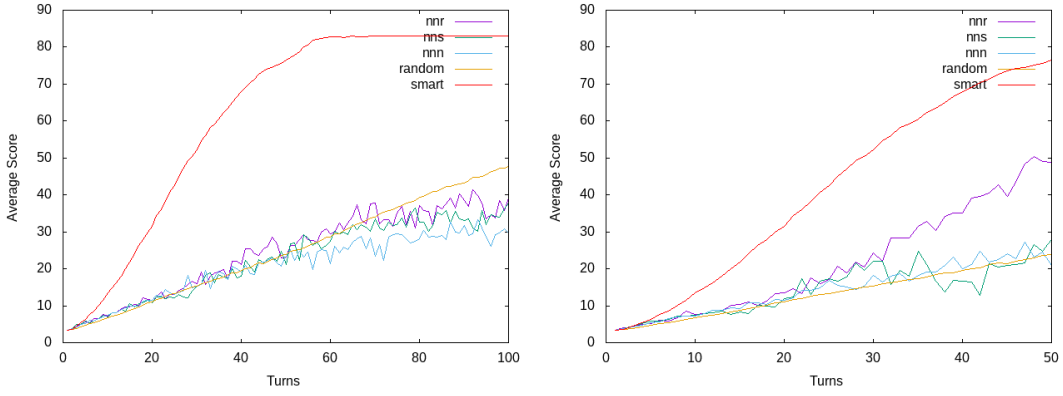
(a) $exp = 100$ (b) $exp = 1000$

Figure 17: Average Score — Neural Network Strategies — Standard Parameters.



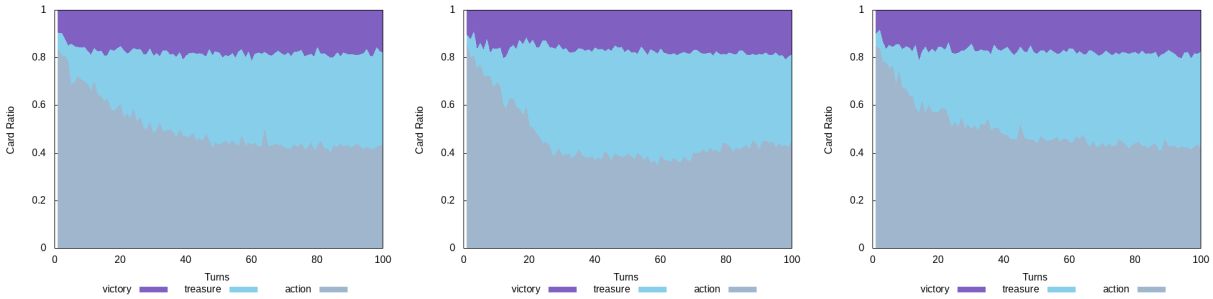
(a) $exp = 100$ (b) $exp = 1000$

Figure 18: Average Score — Neural Network Strategies — Unlimited Stacks.



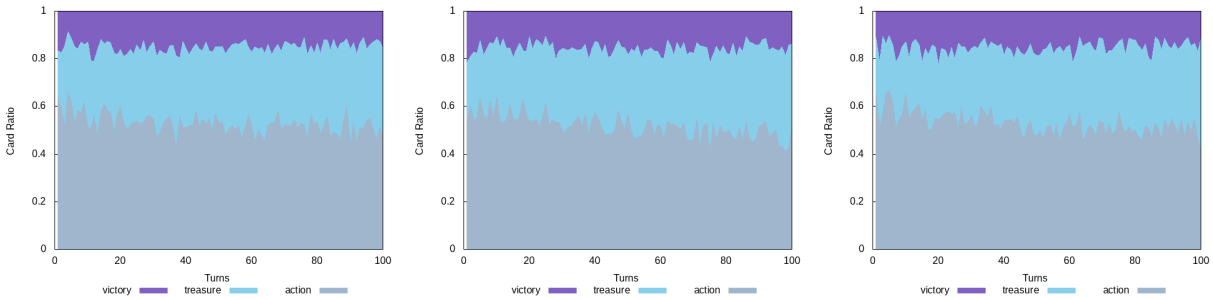
(a) $exp = 100$ (b) $exp = 1000$

Figure 19: Average Score — Neural Network Strategies — No Action Cards.



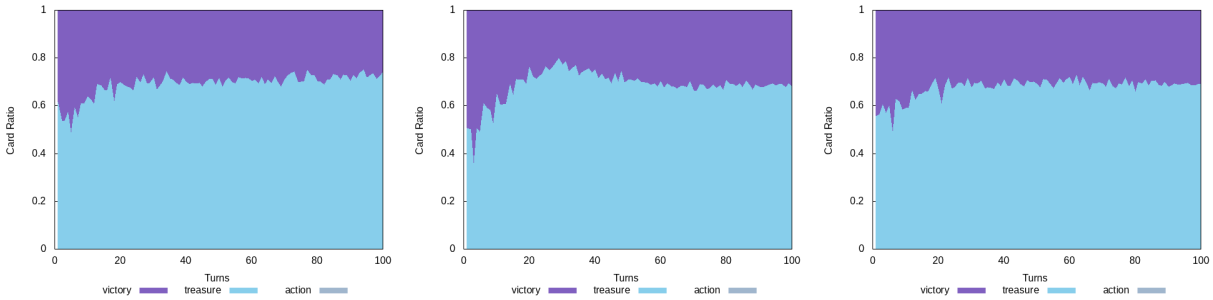
(a) Random (b) Smart (c) Neural Network

Figure 20: Card Ratio — Neural Network Strategies — Standard Parameters.



(a) Random (b) Smart (c) Neural Network

Figure 21: Card Ratio — Neural Network Strategies — Unlimited Stacks.



(a) Random (b) Smart (c) Neural Network

Figure 22: Card Ratio — Neural Network Strategies — No Action Cards.

If a strategy is needed with less computing power, or one which can be easily used when playing the game yourself, the Smart Strategy scores the best, where the most expensive card is bought, and during the Action Phase as many Action Cards are played as possible. This does depend on the set of Action Cards used, a different set of card could perform differently. If Action Cards have been chosen which do not increase *Actions*, it could result in many unplayable Action Cards.

Further Research

The Neural Networks can be improved. Not only can more resources improve its performance, a different type of Neural Network could also prove useful, such as a Deep Q-Network, where each card receives its own value to show how good a card is given the current state.

For the Monte Carlo Strategies, different strategies which are more chance-based can be tested. Would the Smart Monte Carlo Strategies perform better if it does not select from just the most expensive cards purchasable, but also considers slightly cheaper cards?

Other aspects of Dominion can also be explored. Such as the influence of card scrapping and exploring the multiplayer aspect of Dominion. What influence does the number of players have? Which strategies would perform best, focusing on your own growth or hindering your opponent's?

References

[ACBF02] Peter Auer, Nicoló Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multi-armed Bandit Problem. *Machine Learning*, 47:235–256, 2002.

[CBSS08] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 216–217, 2008.

[fde23] frugally-deep. https://github.com/Dobiasd/frugally-deep, 2023. Last accessed 27 March 2023.

[Fis99] George Fishman. *Monte Carlo: Concepts, Algorithms and Applications*. Springer, 3rd edition, 1999.

[Gam23] Rio Grande Games. Dominion. https://www.riograndegames.com/games/dominion/, 2023. Last accessed 19 January 2023.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[Ker23] Keras. https://keras.io, 2023. Last accessed 27 March 2023.

[Mar23] Mario. [Dutch] Review van Dominion (Tweede Editie). https://spellenbunker.nl/bordspellen/dominion-tweede-editie/, 2023. Last accessed 19 January 2023.

[RN20] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 4th edition, 2020.

[RS19] Christian Roberson and Katarina Sperduto. A Monte Carlo Tree Search Player for Birds of a Feather Solitaire. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 2019.

[SB18] Richard Sutton and Andrew Barto. *Reinforcement Learning*. The MIT Press, 2nd edition, 2018.