Proceedings of the Nineteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2023)

Playing Various Strategies in Dominion with Deep Reinforcement Learning

Jasper Gerigk, Steve Engels
University of Toronto
jasper.gerigk@mail.utoronto.ca, sengels@cs.utoronto.ca

Abstract

Deck-building games, like Dominion, present an unsolved challenge for game AI research. The complexity arising from card interactions and the relative strength of strategies depending on the game configuration result in computer agents being limited to simple strategies. This paper describes the first application of recent advances in Geometric Deep Learning to deck-building games. We utilize a comprehensive multiset-based game representation and train the policy using a Soft Actor-Critic algorithm adapted to support variable-size sets of actions. The proposed model is the first successful learning-based agent that makes all decisions without relying on heuristics and supports a broader set of game configurations. It exceeds the performance of all previous learning-based approaches and is only outperformed by search-based approaches in certain game configurations. In addition, the paper presents modifications that induce agents to exhibit novel human-like play strategies. Finally, we show that learning strong strategies based on card combinations requires a reinforcement learning algorithm capable of discovering and executing a precise strategy while ignoring simpler suboptimal policies with higher immediate rewards.

Introduction

Reinforcement learning has been successfully applied to both classical board games, such as checkers (Samuel 1967), backgammon (Tesauro et al. 1995), Chess and Go (Silver et al. 2018), and complex video games, from early Atari games (Mnih et al. 2015) to modern games such as Dota 2 (OpenAI et al. 2019). Less attention has been given to more modern tabletop games including Eurogames with rule sets which allow for far more complex and diverse strategies than those found in classical games. For these games, most computer agents rely on a mixture of heuristics and search algorithms, which do not capitalize on the breadth of strategies these games offer and often fail to provide a challenge to experienced players.

For classical board games, the game state can often be represented as either a vector or as a grid, and for video games, the rendered image is a natural model input. Modern tabletop games do not allow for such a simple representation as they often contain many parts including multiple card piles, counters, and complicated maps in addition to a large variety of different cards. The field of Geometric Deep Learning deals with such non-Euclidean data, including graphs and sets, and allows more natural representations of such game states (Bronstein et al. 2017).

A subgenre of Eurogames is deck-building games, which require players to build and manage a deck of cards with the goal to score the most points. So far, no human-level computer agent has been developed for such games. Dominion[1] is the first game defined by its use of a deck-building mechanic (Furino 2019). Individual games of Dominion differ significantly, with ten cards randomly chosen from the hundreds of available kingdom cards. Together with the seven basic supply cards, they make up the set of all cards used for that game, called the kingdom. Cards generally belong to one of three categories: Treasure Cards, which provide coins to buy new cards; Action Cards, which can be played for a variety of effects; and Victory Cards, which give Victory Points. The basic supply cards include three Treasure (Copper, Silver, Gold) and three Victory (Estate, Duchy, Province) cards, where the latter cards in each group cost more to buy and have a disproportionately stronger effect. All cards in the supply are available in limited quantities. When all Provinces have been bought, or any three card piles are depleted, the player with the most Victory Points wins.

Players start with a weak deck of cards and must improve it before focusing on buying Victory Cards. Winning strategies can roughly be classified into three categories: Big Money, Rush and Engine. Big Money strategies focus on buying Treasure Cards until the player can afford Provinces. Rush strategies aim to accumulate cheap Victory Cards as fast as possible, often resulting in large, inefficient decks. They are competitive only in a subset of kingdoms, for instance, when the Gardens Card is available. Engine strategies take the opposite approach. They are often the strongest strategy but require skillfully chaining together Action Cards with various effects, aiming to draw the entire deck each turn, thus using the owned Treasure Cards to maximum

[1] https://www.riograndegames.com/games/dominion/

effect. We consider any strategy an Engine strategy if it buys and plays a significant amount of Action Cards.

Up to now agent development for Dominion has largely focused on refining Big Money strategies, while ignoring alternative strategies due to their higher complexity and tuning requirements. The configuration of cards included in the kingdom can result in any of the three strategies being optimal, and the players must decide which strategy to select. Mastering individual mechanics such as trashing is often a challenge, with new players failing to understand the importance of deck-control, which includes trashing sub-par cards to remove them from the deck, thus increasing its overall quality. Developing computer agents capable of playing these strategies is necessary if they are to be competitive with human players and is the focus of this work.

In this paper, we develop a novel, more expressive, multiset-based representation of the game state of Dominion utilizing advancements in Geometric Deep Learning. Together with the Soft Actor-Critic (SAC) (Haarnoja et al. 2018) reinforcement learning algorithm, this is the foundation for building agents capable of learning the three main strategies for Dominion. Without relying on domain-specific modifications, the model learns a competitive Big Money strategy that utilizes Action Cards. Through modifications to the reward function, the training setup, and usage of a heuristic, we train agents to utilize the trashing mechanic and to play Rush or Engine strategies. We highlight how these results require a reinforcement learning approach and discuss the difficulties of learning Engine strategies.

Related Work

A range of techniques have been used to develop computer agents for Dominion. Generic heuristics for simple Big Money strategies have been known to the community since the game's publishing (DominionStrategy Wiki 2021). As part of "Geronimoo's challenges - First Game" (Geronimoo 2012), players hand-crafted heuristics for a specific kingdom which were able to play Engine strategies and win 89% of the time against the Smithy Bot, which augments a Big Money strategy by using the Smithy Action Card to draw additional cards. Scaling these heuristics to support multiple kingdoms while maintaining the same strength of play is considered unfeasible.

Fisher (2014) developed Provincial AI which uses an evolution algorithm to learn the optimal card buying heuristic for an individual kingdom while relying on hand-designed general heuristics and a simple look-ahead model for all other decisions, including the playing of Action Cards. Provincial AI is claimed to be challenging for experienced players, but no formal evaluation has been made.

Jansen and Tollisen (2014) proposed a Monte Carlo Tree Search (MCTS) based approach using either Upper Confidence Bounds (UCB) or UCB applied to trees, and a novel method for dealing with stochastic card drawing and player interaction. The algorithms were strong enough to achieve a win rate of 68.5% against augmented Big Money heuristics using the Witch Card, which draws cards and makes opponents gain Curses, but were unable to use Action Card combinations even after adding heuristics.

Angelopoulos and Metafas (2021) apply Q-Learning and achieve a win rate of 57.44% against three bot opponents. They only consider Chapel and Smithy Action Cards and limit the agent to buy only the latter. The complexity of playing Chapels and the necessity of a smaller state space for Q-Learning are the cause for this simplification.

Techniques combining reinforcement learning with neural networks have had limited success and have only resulted in agents playing Big Money strategies. Winder (2014) trains neural networks to make all decisions in a game for one kingdom using temporal-difference learning and back-propagation, hill-climbing, or a genetic algorithm. A genetic algorithm using two separate neural networks for the early and late parts of the game achieves the best performance and wins 74.7% of the games against a Big Money bot. The model plays an augmented Big Money strategy but never uses the trashing mechanic. Fynbo and Nellemann (2010) combine competitive co-evolution and Neuro Evolution of Augmented Topologies to develop three models with combine to form an agent. The first is tasked with predicting how far the game has progressed, the second learns to evaluate the value of different cards, and the third determines in which order Action Cards should be played using MCTS. The model input is a designed feature vector. They successfully train the model for card evaluation but find that a heuristic can outperform the third network. Playing against three Big Money Bots in a four-player game, the learned agent has a win rate of 54.33%.

The two commercial Dominion clients offer the opportunity to play against a computer agent. The agent provided by Shuffle IT[2] is based on heuristics and is generally regarded as weaker than the one developed by Temple Gates Games (Duringer 2022). It is based on the techniques introduced by AlphaZero. Their key innovation is that rather than representing the cards as one-hot encoded variables, they learn card embeddings allowing the agent to play with a large variety of Action cards. No rigorous evaluation of the play strength has been performed.

The only formal description of Dominion and deck-building games, in general, was made by Heijden (2014) who defines the game as a tuple containing the set of cards used, functions for determining the end of the game and various properties of cards, and multisets to track cards in starting

[2] https://dominion.games/

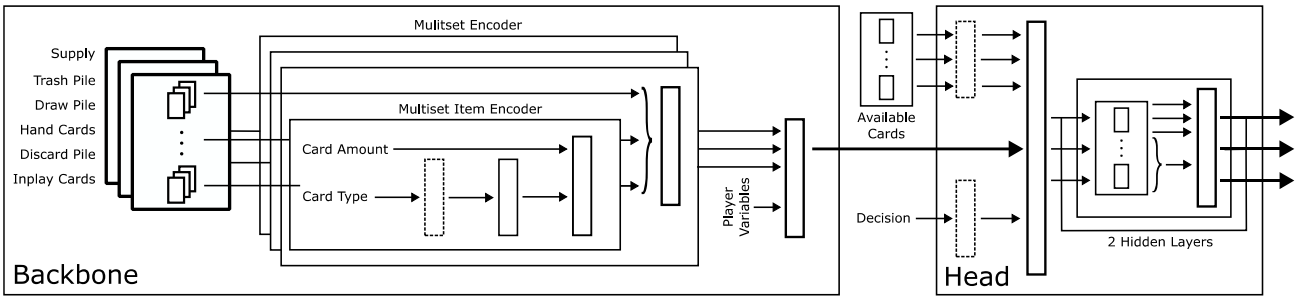

Figure 1: Overview of the model structure. Dashed rectangles represent embedding layers, solid rectangles represent Multi-Layer Perceptron (MLP) layers and curly parenthesis elementwise max aggregations.

deck, current deck, and the hand cards of a player. This formulation ignores the finite number of cards available to buy and the existence of Action Cards, which limits its relevance to Dominion. Heijden (2014) develops heuristic, MCTS, and dynamic programming-based agents for the simplified game and shows that, in this case, a Big Money heuristic strategy is close to optimal.

Methodology

Dominion can be described as an episodic Markov-Decision Process $(S, A, \mathrm{p}, \mathrm{r})$, where S is the state space, A the discrete action space, $\mathrm{p}(s'|s,a)$ the transition probability and $\mathrm{r}(s,a)$ the reward.

Each state is made up of the common supply, containing the cards which can be gained, the shared trash pile, which stores trashed cards, and the status of each player's deck. The deck consists of the cards in the players' draw and discard pile, their hand, and the Action Cards the player has played this turn. Since the draw pile is hidden and the order of the cards in the other locations does not matter, we can model all of them as multisets, which we call *piles*. The state also contains the current player's number of Action Points, Buys, and Coins.

The action space includes all possible decisions required during a game of Dominion and will be elaborated upon in the Decision and Model Structure section. Each action is modeled as choosing one card from a variable-size set of possible cards.

The transition probability generally follows the rules of the game. While some actions have a deterministic effect, such as trashing a card, others, like drawing a card, are stochastic since the order of cards in the draw pile is unknown. As stated previously, the game terminates, when any three supply card piles are empty or all Provinces have been bought. The only addition we make is to terminate the game after 40 moves to limit game lengths when the agents are playing badly. For stronger agents, this limit has no effect on the strategy, as games usually end within 15-30 turns.

A terminal reward was used, consisting of the difference between the player's and the opponent's scored victory points. A simpler reward, which would only depend on if the player won, was not used, as it gives the model no feedback on important marginal improvements during training.

Each player starts with a deck of 7 Copper Treasure Cards and 3 Estate Victory Cards, which are shuffled.

Using this formulation, we can apply a standard model-free reinforcement learning algorithm to Dominion. Soft Actor-Critic (SAC) was chosen due to its good performance on many control tasks and the inclusion of a temperature term, which encourages more exploration (Haarnoja et al. 2018). While SAC was originally developed for continuous actions, Christodoulou (2019) introduced a discrete formulation. SAC consists of an actor and a critic. The actor, given a set of actions, returns the probability with which each action should be played. The critic, given a set of actions, determines the expected time-discounted reward for choosing each action. The terminal reward was scaled by a factor of 20 to balance it with the entropy loss introduced by SAC.

In our agents, the actor and critic models use the same backbone and head structure to process the game state, although the heads make different predictions. The backbone is responsible for converting the game state into a vector representation, which the head then uses to evaluate the options. Both models are trained end-to-end with their own backbone. Figure 1 shows the structure of our model structure.

The Dominion base game contains 32 unique kingdom cards, and 15 expansions have added over 400 more. We consider 26 of the base game cards, as they are enough to allow Big Money, Rush and Engine strategies, and to prevent the agent from overfitting on one kingdom.

Game State

We augment the state by providing the model with each player's current number of Victory Points. As a simplification, we do not include the opponent's total deck composition in the state representation since the information can be derived from the remaining cards in the supply and the cards in the player's own deck. For this paper, we focus on the

two-player case, but the representation can easily be extended to include information for each opposing player.

Backbone

The large number of different cards in Dominion makes it unrealistic and inefficient to treat each card as a unique class (Duringer 2022). Cards often share variations of the same basic effects. Therefore, we represent each card as a learnable 16-dimensional embedding. This allows the model to learn about multiple cards at the same time.

To convert any pile, which is made up of a multiset of cards, to a vector representation, we first map each card type to its embedding and apply a pile specific Multi-Layer Perceptron (MLP). Next, we concatenate the multiplicity of the card, and then use another MLP to get a new vector representation. Note that this transformation can be done in parallel to all cards in the pile, as the operations are permutation equivariant, which means permuting the order of cards in the input is equivalent to permuting the output (Zaheer et al. 2018). To aggregate this set of representations into a single vector representation of the entire pile, we follow Zaheer et al. (2018) and use an element-wise maximum as a permutation invariant aggregation function, which means that a permutation of the input has no effect on the output. This is followed by a final MLP. To allow aggregation with hidden layers, we concatenate the set representation to each individual element representation in the set and repeat the steps.

This process, without any hidden layers, is applied to each of the piles, and the results are concatenated along with the status variables. We then apply a final MLP to get a 32-dimensional representation of the game state.

Decision and Model Structure

While prior work has focused primarily on optimizing what the agent should buy, Dominion requires the player to make a diverse set of decisions. Besides choosing which Action Card to play, some Action Cards require the player to make further decisions when played. While other decision types exist in the game, the cards considered in this paper only require choosing one or more cards from a multiset of possible cards. To simplify the decision-making process, we model decisions in which more than one card must be chosen as iterated single card decisions, where the set of options decreases as decisions are made. While this may make these decisions more difficult to learn, almost all decisions the agent must make involve only choosing a single card.

Since this set of actions is unordered but not limited in size, we use a set-based representation. This means that both the actor and critic models must be permutation equivariant with regard to the actions available and deal with sets as both an input and an output. As described for the backbone, we use the same Set-Aggregation structure with two hidden layers but do not aggregate the final layer since both actor and critic networks require a result per option.

The output of the actor is a probability distribution over the possible actions. Using softmax as the final activation function prevented training due to vanishing gradients. We therefore linearly rescaled the values to the range $[0,1]$. When sampling to choose the action from this categorial distribution, the values were treated as the relative probability of sampling that class. Further, we use the Tanh activation function for all layers in the actor, while we are using ReLU in the critic due to the different ranges of outputs.

To allow a single neural network to make choices for all decisions, the actor and critic models of the SAC algorithm are passed a decision type, along with the state representation and the set of options. Similar to the card embedding layer, we use a decision embedding layer to get a learnable representation of the decision rather than using a one-hot encoded vector. The rationale for this choice is that many decisions are very similar. For instance, playing either Mine, Chapel, or Remodel all requires choosing card/s to remove from the deck. In the case of Mine and Remodel, the decision is followed by a gain decision, whose options are dependent on the previous trash decision.

Soft Actor-Critic

Temperature is a hyperparameter of the original SAC algorithm, which controls how relevant the uncertainty of the actor's output is to the loss. In their follow-up work, Haarnoja et al. (2019) propose automatically optimizing temperature by using the target entropy as the constraint. While the proposed value, $-\log(\dim(\text{Action Space}))$, works well with continuous action spaces, it does not work in practice with the discrete formulation, and considerable effort was spent tuning the target entropy.

Using a variable-size set of options rather than a fixed-size action space further complicates the tuning since it is unclear how the entropy should depend on the number of options available. Additionally, decisions are of varying difficulty and, therefore, should be associated with differing levels of certainty. For example, choosing which Action Card to play is often easier than determining which card to buy. We ignore the difference in entropy for different decisions and use $-c\log(\dim(\text{Available Actions}) - 1)$ for the target entropy to address the changing dimension, where c is a tunable hyperparameter, which we set to 0.5. This means that the agent should become more uncertain when more actions are available and decisions with only one or two actions can be made with very high certainty.

To further stabilize the temperature during training, we used regularization as introduced by Zhou et al. (2022), with a scaling value of 0.7. Additionally, we clamped the alpha value between 0 and 4. For additional exploration, we take a random action 10% of the time. All other hyperparameters are listed in the Appendix.

Training

This paper aims to develop agents capable of playing the various strategies used by humans, rather than achieving optimal performance in a specific kingdom. The agent structure described above is versatile enough to support this, and none of the four distinct agents we trained required any changes to this structure, besides increasing the size of the hidden dimensions for the Engine Strategy. The difference in strategy was achieved primarily through changing the training setup and interface of the model with the game, thus allowing us to keep the model structure the same.

Like Angelopoulos and Metafas (2020), training was done by playing against bots and for some agents by self-play, which consisted of playing against a second version of the agent, like in Winder (2014). We trained all agents on kingdoms generated from the 19 kingdom cards listed in the Appendix. In the following subsections, we will describe the agent and training setup required for each of the strategies.

Big Money Strategy

We used a learning curriculum composed of two bots with different strengths to ensure that the agent was able to learn from the opponent and did not get stuck (Pang et al. 2019). The two bots implemented are the Random Bot, which uses a uniform distribution to choose the action, and the Big Money Bot (DominionStrategy Wiki 2021), which follows a big money strategy and is competitive with inexperienced human players. The first 100 games were played against the Random Bot. Afterwards, we randomly chose which bot to play against. For each bot, we tracked the win rate of the agent over the last 20 games. After clipping the win rate between 0.1 and 0.8, the probability of playing against a certain bot was proportional to the entropy of treating the clipped win rate as a Bernoulli distribution.

Rush Strategy

The only viable rush strategy, given our selection of cards, requires the Gardens card, as it gives Victory Points proportional to the size of the deck. To bias the agent from learning a Big Money strategy to learning a Rush strategy, we required the kingdom to always include the Gardens card.

Training Using Self-Play

Developing agents which utilize trashing required adjusting the terminal reward function and introducing a term that does not depend on the agents score. During training, this should prevent agents from focusing only on improving their immediate score. However, when training against the Big Money Bot, the agents lost consistently and badly, which led them to neglect the second objective and only focus on the score, thus resulting in Big Money strategies. Switching to primarily self-play for the remaining agents, allowed the agents to play against an opponent, who was also pursuing this non-score related goal and followed the same development, resulting in a similar performance level.

Self-play consists of two copies of the agent being trained simultaneously and playing against each other. Whenever the running score between the agents differs by more than 40, a copy of the stronger agent replaces the weaker one. To encourage the agents to also develop a competitive strategy, 20% of the games were played by one of the agents against a Big Money Bot.

Big Money with Trashing Strategy

Trashing can help any non-Rush strategy by removing weak cards from the deck. Before developing a complete Engine strategy, we trained an agent capable of playing Big Money while trashing. We introduced a heuristic to make the trashing decision for the Chapel Action Card by trashing all Curses, Estates and Coppers in that order. Additionally, we adjusted the terminal reward to include -60 points per Estate, Copper or Curse in the deck. Finally, we required the Chapel to be part of any kingdom played.

Engine Strategy

The key part of any Engine Strategy is the usage of Action Cards to draw many cards per turn. This requires the agent to buy and play Action Cards, either allowing it to play further Action Cards or draw additional cards. Trashing can support this goal by removing unwanted cards from the deck. Preliminary experiments showed that the agent was able to correctly utilize Action Cards once it had them in its deck but failed to buy them in sufficient quantities.

To motivate the agent to play an Engine strategy, we changed the starting composition of each player's deck. We defined a deck of 12 cards that can be played as a strong Engine capable of consistently buying a Province per turn. See the Appendix for the deck composition. The cards were set to always be part of the kingdom. For any individual game, we sample a random probability p_{engine} and iterate 12 times, each time taking a card from the standard starting card list or the engine card list whenever a new random number is larger than p_{engine}. 75% of the games starting configurations were generated this way, while 25% used the standard setup. We combined this with the modification with the changes made to encourage trashing.

Training Configuration

Hyperparameters were tuned manually while learning a Big Money strategy and were not changed for other agents. All agents were trained for 300,000 steps. Training was completed on a single NVIDIA RTX A4000 and took approximately one day per agent.

Results

Agents successfully learning their respective target strategies is reflected in a different deck composition, which can be seen in Figure 2 and is discussed in more detail in the sections below. For all evaluations in the section, we assessed the best model we trained. Further, we evaluated the

consistency of the training setups by running each training five times on different seeds. As shown in the Training section of the Appendix, besides the Engine agent training, all trainings were very stable.

To evaluate the performance of our agents, we compared them to the Big Money Bot, which was used during training, and Big Money variants, which were modified to utilize either the Chapel or the Witch card (Winder 2014; Jansen and Tollisen 2014). The Big Money Bot, we used, follows the algorithm described on the DominionStrategy Wiki (2021), rather than from prior work (Winder 2014; Angelopoulos and Metafas 2020; Jansen and Tollisen 2014), as it outperforms these implementations. The Witch Action Card is bought and played by the Witch and the Double Witch Bot.

Big Money Strategy

Trained on kingdoms using any of the nineteen kingdom cards, our agent won 73% of its games against the Big Money Bot, drawing an additional 8.5% of the games. Compared to the pure Big Money heuristic, the learned strategy used various Action Cards when available. It primarily used the Witch and Militia Cards, two cards that decrease the quality of the opponent's current hand or deck. On average, the agent's deck contained 15% Action Cards and 62% Treasure Cards at the end of the game. Games take 25.4 turns, and agents score 30.3 points.

While our agent was trained for two-player games, it can play against more opponents without modification. When playing against three Big Money Bots, it achieved a win rate of 63.5% and outperforms Fynbo and Nellemann (2010) and Angelopoulos and Metafas (2020), although they used slightly different sets of kingdoms.

When the pure Big Money Bot is augmented to use the Witch Card, it performs considerably better, but adding the Chapel did not significantly improve the strength with the current heuristic. This is reflected in the same agent only winning 54% (4% drawn) of the games against the Single Witch Bot, 42% (7.5%) against the Double Witch Bot, and 76.5% (8.5%) against the Chapel Bot. While performance against the Big Money strategy is comparable to Winder's (2014), our agent performs significantly better against the augmented Chapel Bot. The main difference between the Big Money strategies used by the agent and the Witch Bots is that the agent buys too many Action Cards, often never drawing the ones bought late into the game or drawing more Action Cards per turn than it can play.

If the relevant Witch Bot was included in training, the win rates increased to 59.5% (3.5%) and 57% (5%). Jansen and Tollisen (2014) achieve a performance of 68.5% against the two Witch bots using MCTS on a single kingdom. To train against these bots, the kingdom always included the Witch Action Card. This causes the agent to see these cards significantly more often, and its strategy changed to only using the Witch Card. The fundamental issue of overbuying Action Cards remained.

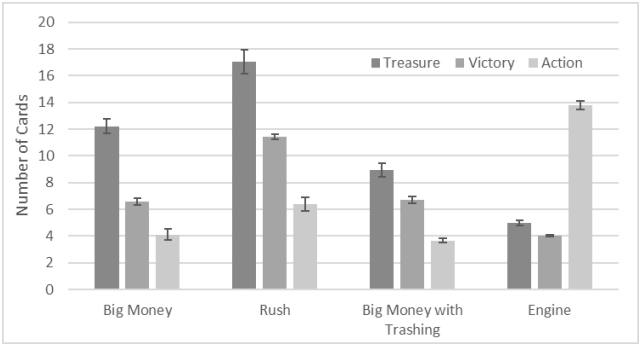

Figure 2: Average deck composition by card type of each of the four agents developed.

Rush Strategy

The agent converged on a strategy of playing the Woodcutter, Workshop, or Bandit Cards, when available, to gain an additional card per turn and the Militia Cards to slow down the opponent. It relied on emptying one Action Card, the Gardens, and the Estates piles to finish the game as soon as possible. This results in the agent's deck containing 18% Action Cards and 49% Treasure Cards at the end. When playing against itself, games on average took 28.76 turns and agents scored 38.0 points.

The agent achieved a win rate of 74% (0.5%) against the Big Money Bot. There are no prior results published on Rush strategies, so we developed our own baseline based on the DominionStrategy Wiki (2022), as included in the Appendix. If the model only encountered Big Money strategies in training and did not encounter a Rush strategy, the agent was unable to respond to an opponent who also plays a Rush strategy, winning only 16.5% (2%) of the games against the Gardens Bot. If additionally trained against the Gardens Bot, the agent achieved a win rate of 80.5% (12.5%) against the Garden Bot, while still winning 49.5% (1.5%) against the Big Money Bot.

Big Money with Trashing Strategy

The trained agent was able to use the Chapel to reduce the number of Coppers and Estates in its deck. Similar to human players, the agent bought a single Chapel at the beginning of the game. The agent then removed all Coppers and Estates from its deck and used this as a basis to play a Big Money Strategy. The agent learned to use Militia, Bandit, Witch, Merchant and Market Action Cards. At the end of the game, the deck contains 19% Action Cards and 46% Treasure Cards, with an average of 0.84 Coppers, 0.34 Estates, and 0.36 Curses cards remaining. This resulted in an average game length of 21.0 turns with 31.0 points. Against the Big Money Bot, it won 78% (6.0%) of games, and even with trashing to counter the Curses, it was only able to win

39% (2.5%) against Single Witch and 32% (3.5%) against Double Witch bots. This performance is comparable to that of the Big Money agent, and in a match between the two it achieved a 60% (4.0%) win rate.

Engine Strategy

The agent developed an Engine strategy that used a combination of Laboratory and Smithy Cards to draw cards, Village, Throne Room and Festival Cards to play additional Action Cards, and Chapels to trash Copper and Curses. For coins, it used a combination of Festival, Silver, and Gold, where some of the Gold was gained via the Bandit Action Card. At the end of the game, the deck consisted of, on average, 60% Action Cards and 22% Treasure Cards. The agent trashed somewhat successfully and ended the game with 2.57 Coppers, 1.05 Estates and 0.05 Curses. Game finished on average in 21.8 turns with 24.7 points.

When playing against the Gardens Bot, the agent achieved a win rate of 52% (0.5%), showing that even a weak Engine strategy can compete with suboptimal strategies. While the current agent was weaker than the much simpler Big Money Bot, winning only 14% (2%) of the games, it is the first agent capable of playing such a strategy.

The agent sometimes trashed too aggressively, resulting in it not having enough money to buy strong cards. As it was unwilling to gain Coppers, the agent ended the game with 0 points. Therefore, the agent will currently lose about 6% of games against the Random Bot. Illustrating that while the trashing heuristic was required to achieve the current level of play, it limits the agent's control and may hinder the final performance.

Discussion

Similar to previous work, developing an agent for Dominion using reinforcement learning to optimize the win rate leads to it learning a Big Money strategy. Due to using a more complex neural network, unlike Winder (2014), we did not need two separate models for the game stages to outperform the baselines. While this required no Dominion specific modifications, making an agent learn use trashing or how to play an Engine strategy required significant modifications, as the complexity of these strategies makes them far more difficult to discover and thus learn. For example, for trashing to be a net positive for the agent, it must buy the Chapel very early in the game, play it when there are cards on the hand that should be trashed and then choose the right cards to trash. Discovering this sequence by chance is highly unlikely. The agent requires samples to learn how to trash but initially, playing Chapel causes random cards to be trashed which is detrimental to the performance. Therefore, the agent will learn not to play or buy Chapels. On the other hand, Big Money strategies only require the agent to learn to buy cards and any error in the sequence will only slow down the agent by a turn or two.

This rise in complexity and the required precision explains why search-based approaches, like Jansen and Tollisen (2014), also find Big Money strategies rather than Engines. There are significantly more sequences of decisions that lead to a strong Big Money solution, and these sequences are also considerably shorter than Engine strategies. Accordingly, they conclude influencing MCTS to discover Engines is difficult since its position evaluation depends entirely on the outcome of the game. Influencing the learned policy to be an Engine strategy, on the other hand, is considerably easier when using a reinforcement learning approach as it requires an easily adjustable reward function.

Due to the simplicity of the Big Money strategy an agent will naturally learn such a strategy first. In most cases buying and playing any individual Action Card on its own will not improve the performance of the Big Money strategy and will therefore be judged as a mistake by the reinforcement learning algorithm. The only way to learn an Engine strategy is to avoid ever learning a Big Money strategy, as SAC's exploration efforts fail to break away from the local maximum.

As shown in this paper, we can nonetheless "trick" the agent into learning an Engine strategy. This is achieved when the algorithm starts playing from a range of starting positions: having all necessary cards for an Engine to having none. Since playing the Engine is optimal and quicker than Big Money in many of these positions, the agent will learn such a strategy for those positions and then apply it to the others.

SAC also faces exploration issues when trying to finetune its policy, as it fails to explore enough details to find the optimal solution. For example, when learning to play a Big Money strategy, it consistently runs into issues of overbuying Action Cards and does not learn to correct this during training. It is most likely caused by a combination of the small impact of decisions on the overall result and a lack of targeted exploration by the agent, as the stochastic exploration used by SAC is not temporally dependent. Even when the agent does not overbuy an Action Card due to a random exploration decision, it will do so at the next opportunity, as the random action is unlikely to repeat. So, the randomness of any exploration action is usually smoothed over.

However, the ineffectiveness of the exploration can be exploited, to make the agent learn a Rush strategy. Once the agent discovers the cheap Gardens cards and is then rewarded for getting a large deck, it fails to explore alternative strategies like Big Money. Note that this only works if Gardens is always available in all kingdoms the agent plays.

Future Work

While this paper shows that playing more complicated strategies with higher potential in Dominion is possible, the current agents' performances does not yet reach human levels. Moving from an approach that considers only a single action to one which operates on sequences of actions promises to improve the precision of play. Alternatively, algorithms that use search, such as AlphaZero (Silver et al. 2018), should be explored. Further, new exploration methods should be developed which allow the agent to learn trashing in any setup as a foundation for playing strong Engine strategies.

Conclusion

In this paper, we develop a new model structure to broaden the strategies supported by computer agents in Dominion beyond Big Money with the goal of playing more human-like. The model utilizes a multiset-based representation of the game state in Dominion, which, compared to prior approaches allows for better learning of various kingdom cards effects and interactions. We adapt the SAC algorithm to choose actions from sets of options with variable sizes. This allows the agent to make all decisions in Dominion using a single model, giving it more flexibility and removing the need to rely on heuristics.

Without any Dominion-specific modifications, our model learns to play a Big Money strategy. It is able to beat standard heuristics on a wide variety of kingdoms but falls short of Big Money strategies augmented to use the Witch Action Card. We introduce modifications to the agent and training process to develop the first agents which use trashing and play Rush or Engine strategies.

Engine strategies are the most difficult to learn due to their reliance on the interaction between various Action Cards with limited payoff until fully mastered. Current agents are limited by having to discover strategies through learning individual decisions independently. The next level of performance will be achieved by agents aware of the sequence in which decisions are made.

Hyperparameters

Actor, Critic, Alpha Learning Rate = 0.003

Discount rate = 0.99

Gradient clipping = 5 Replay Buffer Size = 100000

Batch Size = 256

Decision embedding dimension = 4

The replay buffer was initially filled with 1000 random actions.

All hidden dimensions are 32 except for the Engine Agent which uses 64 hidden dimensions and a 128-dimensional game state representation. This means the Engine Agent had 117,489 parameters with the other agents having 32,273 parameters. All weights were initialised using the standard PyTorch initialisation.

Kingdom Cards Used

Village, Throne Room, Militia, Witch, Bandit, Smithy, Laboratory, Council Room, Festival, Woodcutter, Workshop, Market, Chapel, Moneylender, Remodel, Merchant, Harem, Artisan, Gardens

Gardens Bot

The Gardens Bot requires both Workshops and Gardens to be in the kingdom. The following cards are bought, when possible, from highest to lowest priority: Workshops, Gardens, Estates, any Action Card with a cost below 4, which has already been bought, Copper. It will play a Workshop, whenever it can, to gain a Gardens Victory Card.

Engine Starting Deck

2x Village, 2x Smithy, 1x Festival, 1x Chapel, 3x Laboratory, 1x Throne Room, 2x Gold

Training

Figure 3 shows the Victory Points gained by the agent per round per game, calculated as total points scored divided by number of turns played. Each agent was trained for five different seeds. The training of the Engine strategy failed to learn any good strategy in one run, which was excluded from the figure.

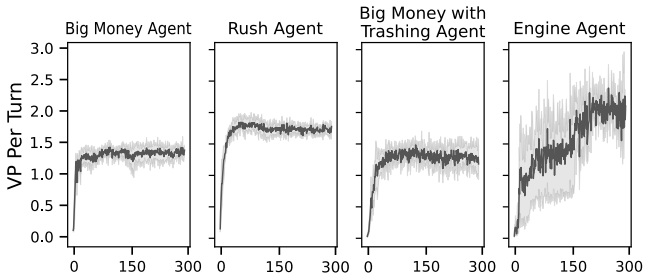

Figure 3: Victory Points gained per turn per agent during training over 300 thousand steps. The values are not comparable between the agents due to different strategies. For example, the Rush agent focused on buying Gardens while the opponent was buying Provinces, increasing maximum points available. Further the Engine agent uses different starting states, which allow for a lot quicker scoring.

References

Angelopoulos, G.; and Metafas, D., 2021, November. Q Learning applied on the Board Game Dominion. In *24th Pan-Hellenic Conference on Informatics* (pp. 34-37).

Christodoulou, P., 2019. Soft actor-critic for discrete action settings. arXiv:1910.07207.

DominionStrategy Wiki. 2021. Money Strategies. https://wiki.dominionstrategy.com/index.php?title=Money_strategies&oldid=53660. Accessed: 2023-05-26.

DominionStrategy Wiki. 2022. Combo: Workshop and Gardens. https://wiki.dominionstrategy.com/index.php?title=Combo:_Workshop_and_Gardens&oldid=67141. Accessed: 2023-05-26.

Duringer, T. 2022. Dominion AI. *Temple Gates*. 25 April 2022. https://www.templegatesgames.com/dominion-ai/. Accessed: 2023-05-26.

Fisher, M. 2014. Provincial: A Kingdom-Adaptive AI for Dominion. https://graphics.stanford.edu/~mdfisher/DominionAI.html. Accessed: 2023-05-26.

Furino, G. 2019. Dominion Review - The Game That Launched A Genre. TechRaptor. 28 March 2019. https://techraptor.net/tabletop/reviews/dominion-review-game-that-launched-genre. Accessed: 2023-05-26.

Fynbo, R. B.; and Nellemann, C. S. 2010. *Developing an Agent for Dominion Using Modern Ai-Approaches*. M. Sc. IT, IT University of Copenhagen.

Geronimoo. 2012. Geronimoo's Challenges - First Game. http://forum.dominionstrategy.com/index.php?topic=3779.msg101489#msg101489. Accessed: 2023-05-26.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S., 2018. Soft actor-critic: Off-policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, 1861–1870. PMLR

Haarnoja, T.; Zhou, A.; Hartikainen, K.; Tucker, G.; Ha, S.; Tan, J.; Kumar, V.; Zhu, H.; Gupta, A., Abbeel, P.; and Levine, S. 2019. Soft Actor-Critic Algorithms and Applications. arXiv:1812.05905.

Heijden, R. 2014. *An Analysis of Dominion*. Thesis Bachelor Informatica, Leiden Institute of Advanced Computer Science, Leiden University.

Jansen, J. V.; and Tollisen, R. 2014. *An AI for dominion based on Monte-Carlo methods*. Master's thesis, University of Agder.

Mastrangeli, T. 2014. Top 10 Deck Building Games. Board Game Quest. www.boardgamequest.com/top-10-deck-building-games. Accessed: 2023-07-27.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M., Fidjeland, A.K.; Ostrovski, G.; and Petersen, S. 2015. Human-level control through deep reinforcement learning. *Nature*, *518*(7540): 529-533. Number: 7540 Publisher: Nature Publishing Group.

OpenAI; Berner, C.; Brockman, G.; Chan, B.; Cheung, V.; Dębiak, P.; Dennison, C.; Farhi, D.; Fischer, Q.; Hashme, S.; Hesse, C.; Józefowicz, R.; Gray, S.; Olsson C.; Pachocki, J.; Petrov, M.; d. O. Pinto, H. P.; Raiman, J.; Salismans, T.; Schlatter, J.; Schneider, J.; Sidor, S.; Sutskever, I.; Tang, J.; Wolski, F.; and Zhang S. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. arXiv:1912.06680.

Pang, Z.-J.; Liu, R.-Z.; Meng, Z.-Y.; Zhang, Y.; Yu, Y.; and Lu, T. 2019. On Reinforcement Learning for Full-Length Game of Starcraft. In *Proceedings of the AAAI Conference on Artificial Intelligence*. volume 33: 4691-4698. Issue: 01

Samuel, A. L. 1967. Some studies in machine learning using the game of checkers. II—Recent progress. *IBM Journal of research and development*, 11(6): 601–617.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140-1144.

Tesauro, G.; et al. 1995. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3): 58–68.

Winder, R. K. 2014. Methods for approximating value functions for the Dominion card game. *Evolutionary Intelligence* 6 (4): 195–204.

Zaheer, M.; Kottur, S.; Ravanbakhsh, S.; Poczos, B.; Salakhutdinov, R. R.; and Smola, A. J. 2017. Deep sets. *Advances in neural information processing systems,* 30.

Zhou, H.; Zichuan L.; Junyou L.; Deheng Y.; Qiang F.; and Yang, W. 2022. Revisiting Discrete Soft Actor-Critic. arXiv:2209.10081.