DRAFT VERSION DECEMBER 9, 2019
Preprint typeset using LaTeX style emulateapj v. 01/23/15

AN AI FOR DOMINION USING DEEP REINFORCEMENT LEARNING

HUNG-I YANG, YU-CHI KUO

Draft version December 9, 2019

ABSTRACT

We explored 3 different model-free deep reinforcement learning algorithms, Monte Carlo Learning, SARSA, and Deep Q-Learning, to build an AI for the boardgame Dominion. 7 different hyperparameters are studied, corresponding to neural network structure, regularization, reinforcement learning exploration, and reinforcement learning reward design. With the Monte Carlo Learning algorithm and the right hyperparameters, we were able to achieve >85% win rate against a heuristic-based AI close to the level of human experts.

1. INTRODUCTION

Dominion(Wikipedia contributors 2019) is a deck building card game for 2-4 players. While there are some previous research on building an AI for this game(Fynbo 2010)(Tollisen et al. 2015), none of those uses deep reinforcement learning as the approach. This problem is interesting for deep reinforcement learning in many aspects, including:

- This game is suitable for reinforcement learning because it involves sequential decision making and long term planning. For human experts, a typical game length is 10-20 turns, with 2-5 sequential decisions to make each turn.
- Deep learning is useful to build a Q-function for a given game state and action. The game state is complicated, with 17 types of cards, 10 cards of each type, and each card of a type can be in one of 9 different piles. This results in approximately $(C_9^{19})^{17}$ game states plus a few in-game actions.
- This is one of the favorite board games of both authors.

A few challenges of this project includes:

- Stochastic nature of this game. This card game involves shuffling cards, so outcomes of a given action is not deterministic.
- Dynamic action set. At the buy phase of different rounds, the available cards are different depending on the number of coins that the player has.

2. DATASET

For this project, we decide to limit the scope of cards to 12 action cards that are chosen from the base set plus the intrigue expansion and to fix the number of players to 2. Our dataset comes from only AI generated match history. The 12 action cards we chose are: *village, cellar, smithy, festival, market, laboratory, chapel, warehouse, council room, militia, moat*, and *witch*. The effects of the cards can be found on *DominionStrategy Wiki* (Wiki contributors 2019).

There is an available python implementation of the game (rsp 2019) available on github. We designed a deep reinforcement learning framework and integrated it into the python implementation.

Each turn of the game Dominion can be separated into two phases:

- Action phase:
  Play action cards that has various effect. Ex. draw cards, trash cards, +actions, +buys, etc.
- Buy phase
  Buy cards from supply piles with the available coins.

In the Action phase, the state of the game can be hard to define for many of the action cards. Take the action card "Library" as an example: Keep drawing until you have 7 cards in hand, skipping any action cards you choose to. It is hard to define a game state that is compatible with each step of the decision process of "Library". Therefore, we decide to apply reinforcement learning only on the Buy phase of the game. For the Action phase, we design some simple heuristics to determine the order that each action card is played and what to do for each action card. Since the 12 cards we choose to include are relatively simple, it is straightforward to hand design heuristics for the Action phase, and it would not be too different from a good human game play.

For the Buy phase, the game state can be reduced to the number of actions, buys, coins, and the number of cards of each kind in the different piles. The categories of piles include supply pile, hand, draw pile, discard pile, in-play pile, and the opponent's full deck. Combining all the information above, in RL terms, each buy decision can be described as a 117-dimension vector s (denoting game state) and a 19-dimension vector a (denoting which card was bought), as shown below:

$$s = \begin{pmatrix} \#actions \\ \#buys \\ \#coins \\ \mathbf{n}_{supply} \\ \mathbf{n}_{hand} \\ \mathbf{n}_{draw} \\ \mathbf{n}_{discard} \\ \mathbf{n}_{in-play} \\ \mathbf{n}_{opponent} \end{pmatrix}, a = \mathbf{n}_{bought} \quad (1)$$

where $\mathbf{n}_p$ is a 19-dimension vector denoting number of each 19 cards in pile p.

The dataset is generated entirely with AI game play. In addition to the RL agents, we also implemented a few AIs using hard-coded heuristics. The **BigMoneyBot** buys treasure cards such as *Silver* and *Gold* and then buys victory cards *Province*, *Duchy*, and *Estate* depending on the progress of the game state. The **SmithyBot** buys action card *Smithy* (effect: draw three cards) on top of the **BigMoneyBot** policies. The **Random** bot buys cards randomly. The **RandomActionBot**

buys action cards randomly but otherwise buy treasure cards and victory cards according to the **BigMoneyBot** heuristics. All the heuristics bots and RL agents play action cards the same way according to some simple coded rules. The **BigMoneyBot** is a good baseline, the **RandomActionBot** performs at similar level to **BigMoneyBot**, whereas **SmithyBot** is very strong and has a respectable win rate even against good human players.

We let the RL agents play against itself and heuristic AIs to generate match data in tuples of $(s_t, a_t, r_t, s_{t+1}, a_{opt,t+1})$, where s_t is the game state (in Buy phase), a_t is the card bought, r_t is the reward of buying this card, s_{t+1} is the next state at buy phase, and $a_{opt,t+1}$ is the card available to be bought at the $t+1$ state. We will describe how the reward is designed and how to train the reinforcement learning model in section 5.1.

We mainly trained the RL bot in a environment of a mix of **SmithyBot/RandomBot/RL agent**. Many RL agents trained with this data achieved high win rate against **SmithyBot** but had poor performance when the opponent buys a wider variety of cards other than *Smithy*. Therefore, we also tried training with a mix of **SmithyBot/RandomActionBot/RL agent** where it will face opponents that play a wider variety of action cards.

3. REINFORCEMENT LEARNING ALGORITHMS

We tried three different model-free reinforcement learning algorithms: Monte Carlo, SARSA, and a modified deep Q learning. The high level approach of all three algorithms use a neural network to predict the state-action value function $Q(s,a)$. Given a state s and action a, $Q(s,a)$ evaluates the value of the state-action pair (s,a), and can be understood as the expected total reward by doing action a at state s.

3.1. *Monte Carlo Reinforcement Learning*

Monte Carlo learns from complete episodes of games. First, we define the total discounted reward $G(s,a)$.

$$G(s_t,a_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{T-t} r_T \tag{2}$$

where γ is the discount factor for future rewards, T is the terminal step of the episode, and r_t is the reward at step t.

Monte Carlo aims to learn the Q-function $Q_\pi(s,a)$ given a state and an action, which is the expected discounted reward if the agent take action a at state s, given its policy π.

$$Q_\pi(s,a) = E_\pi[G_t|(s_t,a_t) = (s,a)] \tag{3}$$

We generate several episodes of game plays, recorded as (s_t, a_t, r_t). For each (s_t,a_t) pair, we calculate $G(s_t,a_t)$ based on the entire game episode. The update rule is simply:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(G(s,a) - Q(s,a)) \tag{4}$$

where α is the learning rate. Monte Carlo learning is an on-policy reinforcement learning algorithms, so it is necessary to update the training data frequently.

The policy π during evaluation is simply:

$$\pi(a|s) = \begin{cases} 1, & \text{if } a = \arg\max_{a \in A} Q(s,a) \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

For training, we applied ϵ-greedy on top of the Q policy for RL exploration (Eq.6). We tried both a constant ϵ over all iterations and a decaying ϵ which approaches 0 over number of iterations (Eq.7):

$$\pi(a|s) = \begin{cases} (1-\epsilon) + \epsilon/|A|, & \text{if } a = \arg\max_{a \in A} Q(s,a) \\ \epsilon/|A|, & \text{otherwise} \end{cases} \tag{6}$$

, where $|A|$ is the number of available actions and

$$\epsilon = \begin{cases} \epsilon_0, & \text{if } \epsilon\text{-decay} = \text{False} \\ 10 \times \epsilon_0 / N_{iter}, & \text{if } \epsilon\text{-decay} = \text{True} \end{cases} \tag{7}$$

3.2. *SARSA*

The name SARSA stands for state-action-reward-state-action. SARSA is also an on-policy learning algorithm. However, different from Monte Carlo, it uses bootstrapping to fit for $Q(s,a)$. With the training data $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, we generate the update target as follow.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)) \tag{8}$$

$r + \gamma Q(s_{t+1}, a_{t+1})$ is the target bootstrapped from the current value function that we want to fit for, and α is the learning rate.

To avoid the problem of a constantly changing target, we have two different networks, the target network that generates the bootstrapped target $r + \gamma Q(s_{t+1}, a_{t+1})$, and the prediction network that tries to fit it. After some number of iterations, we then set the weights of the target network to be the same as the prediction network.

The policy $\pi(a|s)$ is the same as the one described in the Monte Carlo algorithm.

3.3. *Deep Q Learning*

Deep Q learning also learns from bootstrapping the current estimate of the $Q(s,a)$ similar to SARSA. However, instead of simply using the $r + \gamma Q(s_{t+1}, a_{t+1})$ as the target, it does a greedy search and use $r + \gamma \max_{a' \in a'_{opt}} Q(s', a')$ as the target. Different from the traditional Q learning algorithm, we don't evaluate the max over all possible actions, but just the actions that are available in that step (the cards that are affordable and available at that buy phase). This turns out to be essential for the algorithm.

The update rule is the follow:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a' \in a'_{opt}} Q(s',a') - Q(s,a)) \tag{9}$$

where α is the learning rate, s is the current state, a is the action, r is the reward, s' is the next state, and a'_{opt} is the set of available actions in state s'. The policy $\pi(a|s)$ is the same as the one described in the Monte Carlo algorithm.

The advantage of this algorithm over the other two is that it can potentially make use of slightly off-policy data and be more data efficient.

3.4. *Comparison of the three algorithms*

We trained RL agents using the three algorithms with multiple combinations of hyperparameters. The win rate of the various RL agents vs. **SmithyBot** are shown in Fig.1. We can see that Monte Carlo Learning achieves the best performance in both the maximum win rate and the average performance of all agents. SARSA is able to achieve good performance with the right hyperparameters, whereas DQL never manage to beat **SmithyBot** reliably. Monte Carlo agents also converges faster compared to SARSA and DQL. In the following

sections where we investigate hyperparameters and win reward design, we will focus on the results from Monte Carlo agents.

The reason that Monte Carlo Learning and SARSA greatly outperforms DQL might be because we mostly train on on-policy data. It is relatively fast to generate large amount of game data, so we are able to update the data buffer frequently. Therefore, the advantage of data efficiency for DQL does not matter in our case.

It seems that for Dominion two player games, each episode of game play is not very long (20-30 turns), so the high variance of Monte Carlo Methods is not an issue. Bootstrapping algorithms are more sensitive to different hyperparameters, and $Q(s,a)$ predictions can diverge if the wrong ones are chosen. For the game Dominion where on-policy data are easily generated and where game episodes do not last very long, the simple Monte Carlo Learning is the best option.

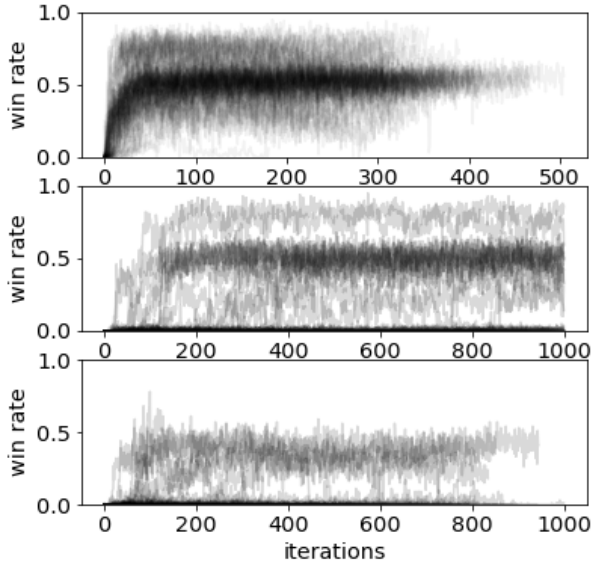

FIG. 1.— Top: Monte Carlo, Middle: SARSA, Bottom: DQL. Evolution of the RL agents with different hyperparameters for the three algorithms. The y axis is the win rate against **SmithyBot**. We can see that Monte Carlo achieves much better performance compared to the other two bootstrapping algorithms. All the agents of Monte Carlo achieves larger than 5% win rate, whereas SARSA and DQL only works with some sets of hyperparameters and has a lower best performance.

4. NEURAL NETWORK DESIGN

The Q-function $Q(s,a)$ takes $117+19=136$-dimension input vector and outputs one real number. We use a neural network (referred to as Q-network) to represent this function.

We use the Adam optimization to train the Q-network, and added dropout for regularization. The number of layers used and the dropout percentage are both hyperparameters that we explored. The structure of Q-network is shown in in Fig.2. Based on our results (Fig.4), the function is smooth enough that a small neural network with few layers is likely enough.

5. REWARDS AND HYPERPARAMETERS

There are different possibilities for designing the reward $r(s,a)$ in RL algorithm, and many different hyperparameters to explore.

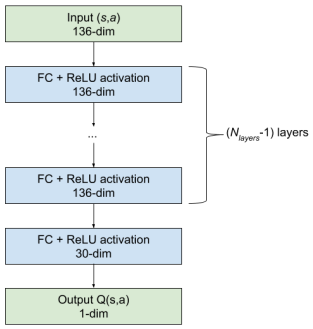

FIG. 2.— Structure of the Q-network. Input is a 136-dimensional vector $x=(s,a)$ and output is a real number $y=Q(s,a)$. There are N_{layers} hidden layers with ReLU activation, where the first $N_{layers}-1$ are 136-dimensional and the last hidden layer is 30-dimensional. The same dropout probability is applied to all hidden layers during training.

5.1. *Reward Design*

One possibility is to use the final game outcome as the reward, which can be considered as the true reward of this problem. However, this reward alone is very sparse and the agents might not be able to learn well. The other simple option is to use the victory points as the reward, since at the end of the game, the player with the most victory points wins. The advantage of using victory points for reward is that even when the win rate is 0% or 100%, the RL agents can still learn to get more points.

In addition to the victory points or the game outcome, we also tried a terminal reward that is proportional to the victory points gained per turn. The motivation for this is that when RL agents trained against **RandomBot**, it optimizes for getting every possible victory points, which is not a good strategy when playing against stronger opponents. Therefore, the points per turn terminal reward can motivate a shorter game and hopefully a stronger RL agent. Therefore, for the reward, we have:

$$r(a_t,s_t)=\begin{cases}\Delta P_t, & \text{if } s_t\neq s_T\\ \Delta P_t+R_w+R_p\times P_{\mathrm{T}}/N_{\mathrm{turn}}, & \text{if } s_t=s_T \ \&\text{win}\\ \Delta P_t-R_w+R_p\times P_{\mathrm{T}}/N_{\mathrm{turn}}, & \text{if } s_t=s_T \ \&\text{lose}\end{cases} \tag{10}$$

where ΔP_t is the victory points gained from buying card a_t, P_T is the total victory points at the end of the game, N_{turn} is the total number of turns, and R_w (win reward) and R_p (reward points per turn) are hyperparameters.

5.2. *Hyperparameters*

We randomly sampled 500 combinations of hyperparameter values from the distributions listed in Tab.1.

hyperparam	**category**	**min**	**max**	**distribution**
N_{layers}	network structure (Fig.2)	2	5	uniform (discrete)
Dropout	regularization (Fig.2)	0.0	0.5	uniform
ϵ_0	exploration (Eq.7)	0.1	0.01	log uniform
ϵ-decay	exploration (Eq.7)	False	True	Bernoulli $p=0.5$
γ	reward calculation (Eq.2)	0.9	0.99	(1- γ) log uniform
R_w	reward design (Eq.10)	0.01	100	log uniform
R_p	reward design (Eq.10)	0.01	10	log uniform

TABLE 1
FOR EACH ALGORITHM, WE PICK 100-500 RANDOM COMBINATIONS OF HYPERPARAMETERS FROM THE DISTRIBUTION.

To estimate the effect of random seeds and random game plays on the RL agents performance, we run 100 RL agents with one set of particular hyperparameters. In Fig.3, we can see that agents with the same set of hyperparameters converge around the 100th iteration. Therefore, we run at least 200 iterations when evaluating each set of hyperparameters.

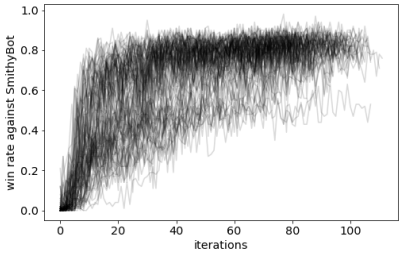

FIG. 3.— 100 agents with the same set of hyperparameters for Monte Carlo algorithm. We see that after 100 iterations, most agents have converged.

6. EVALUATING RESULTS

We evaluate the RL agents performance by the win rate of the agents against baseline heuristics AI **SmithyBot**. Win rate is evaluated by running 100 Dominion games and recording the number of games the RL agent wins. We do this evaluation at every iteration.

6.1. *Investigate hyperparameters*

We investigate the hyperparameters for the Monte Carlo algorithm that achieved the best performance. 500 random combinations of hyperparameters from Tab.1 are sampled according to the listed distributions.

We analyzed the win rate against each hyperparameter, as shown in Fig.4 (N_{layers} and ϵ-decay) and Fig.5 (all other hyperparameters).

As shown in Fig.4, $N_{layers} = 2$ in Q-network seems to perform best, although all $N_{layers} \in [1,5]$ performs reasonably well.

Dropout, ϵ_0, ϵ-decay, and γ does not seem to affect performance much, as shown in Fig.4 and Fig.5.

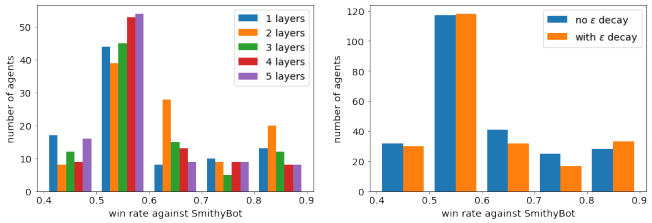

FIG. 4.— Left: Histogram of number of agents binned by win rate against **SmithyBot** for number of neural network hidden layers N_{layers}. Right: histogram of number of agents for with and without ϵ-decay. We can see that there are no strong correlation between the performance of agents with N_{layers} or ϵ-decay.

Hyperparameters related to reward design (R_w and R_p) are more correlated to performance. As shown in Fig.6, it seems that a higher R_w and a lower R_p results in best performance in win rate. This is because a higher R_w motivates the RL agents to win the game, whereas a higher R_p motivates the RL agents to end the game earlier (but not necessarily winning). However, with a small win reward R_w, R_p does seem to help the agents learn consistently a strategy that beats **SmithyBot** around 60% of the time. It seems that R_p helps the agents learn a good strategy (60% win rate), but prohibits the agents from learning the best strategy (80% win rate).

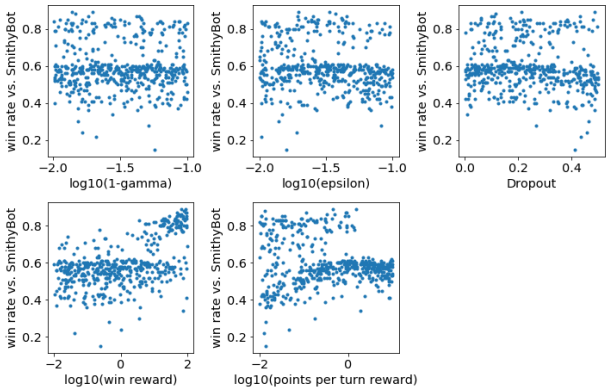

FIG. 5.— x: hyperparameters listed in Tab.1. y: win rate vs. **SmithyBot**. We can see that there are no clear correlation of γ, ϵ, and Dropout with the training performance. For win reward R_w and points per turn R_p, we can see that there are some interesting correlations. The best performing RL agents have larger win reward, while the agents with large R_p does not achieve high win rate.

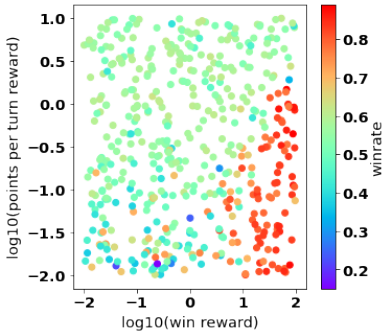

FIG. 6.— x: log of the win reward R_w , y: log of the points per turn reward R_p. color: Best win rate against **SmithyBot** for agents.
We can see that all the agents that achieve the highest win rate have a high win reward ($R_w \gtrapprox 10$), and the agents with high points per turn $R_p \gtrapprox 2$ fails to achieve win rate larger than 70%. However, when R_w is very small, higher R_p does help the agents to consistently achieve a slightly higher than 50% winrate.

6.2. *Training environment*

For the hyperparameter and reward studies, we generate data in an environment (env_1) with equal mix of **SmithyBot/RandomBot/RL agent**. As mentioned in section 2, when played against human players who usually buy a wide variety of cards, the RL agents does not react well. This is likely because it only faced strong opponents who buys *Smithy* or buys the same cards as itself. Therefore, we also tried training RL agents with env_2 **=SmithyBot/RandomActionBot/RL agent** in the hope that the RL agents trained this way will be more robust when exposed to different play styles.

Many RL agents trained in env_2 also managed to achieve a $> 80\%$ win rate against the baseline **SmithyBot**. We picked one agent RL_{Random} trained in env_1 and one agent $RL_{RandomAct}$ from env_2 that both achieved very high win rate against **SmithyBot**. Tab.2 shows the win rate of the two agents agains **SmithyBot** and **SmithyWitchBot**. (**SmithyWitchBot** is an

	RL_{Random}	$RL_{RandomAct}$	Smithy	SmithyWitch
RL_{Random}		0.45	0.83	0.71
$RL_{RandomAct}$	0.55		0.83	0.73
Smithy	0.17	0.17		0.39
SmithyWitch	0.29	0.27	0.61	

TABLE 2
THE LEFTMOST COLUMN IS THE PLAYER, AND THE TOP ROW IS THE OPPONENTS. THE ENTRIES SHOW THE WIN RATE OF THE PLAYER AGAINST THE DIFFERENT OPPONENTS. MORE ROBUST STRATEGIES CAN BE LEARNED BY FACING MORE DIVERSE OPPONENTS.

heuristic AI that buys the cards *Smithy* and *Witch* on top of the **BigMoneyBot** strategy). Both agents achieve the same win rate (83%) against the baseline **SmithyBot**. However, when played against each other, $RL_{RandomAct}$ has a 55% win rate against RL_{Random}. $RL_{RandomAct}$ also has a higher win rate against **SmithyWitchBot** compared to RL_{Random}. By training against more diverse opponents, the RL agents are able to learn more robust Q-functions while maintaining the same high win rate against the baseline.

6.3. *Human evaluation*

Both authors played a few games against the best version of AI trained with Monte Carlo Reinforcement Learning algorithm. While both authors were able to beat the RL agent, we find the AI pretty solid and would be very challenging for less experienced human players. If limited to not buying the card *Chapel* (effect: trash up to 4 cards from your hand), the best RL agent is arguable better than the authors. It is likely that the hard coded rules for playing *Chapel* is not optimal, therefore the agents failed to learn to buy it.

In Fig.5, we see that there seems to be a barrier at win rate 60% that are only surpassed by few RL agents. From playing against some of the RL agents with 80% or 60% win rate, it seems that the difference is whether the RL agents learn to use the card *Witch*.

7. CONCLUSIONS AND FUTURE WORK

We tried 3 different RL algorithms, namely Monte Carlo Reinforcement Learning, SARSA, Deep Q Learning. Monte Carlo Reinforcement Learning works best, achieving >80% win rate against baseline **SmithyBot** AI in many hyperparameter combinations. SARSA and Deep Q Learning hardly achieve >50% win rate against baseline **SmithyBot** AI.

Among all 7 hyperparameters we explored, those related to reward design affects the result the most. The true reward R_w (terminal win reward) helps achieve the best performance, while the points per turn reward R_p helps the agents consistently reach a 60% win rate but prohibits them from reaching the top performance.

The opponent environment where we generated the data also matter. Training against more diverse opponents help RL agents learn a more robust strategy that react well to opponents with more diverse play styles.

For the scope of this project, we limited the game to a predefined set of 12 action cards, fixed to 2 player games, and use heuristic-based logic for playing action cards. Possible future work includes: 1) include more action cards in AI training 2) expand beyond 2 player game into multiplayer games 3) also use Deep Reinforcement Learning for playing action cards.

Another interesting idea is to explore the possibility of building an AI that can handle different sets of cards. While this work limits the cards to 12 action cards plus the basic victory and treasure cards, in a real dominion game setup, a set of 10 different cards are picked randomly from a large card pool in every game. By training the RL agents with many different combination of 10 cards, it is possible that they can generalize and play well with a new combination of 10 cards that they have never seen before.

REFERENCES

2019, https://github.com/rspeer/dominiate-python, [Online; accessed 4-November-2019]

Fynbo, R. B. 2010, Developing an agent for Dominion using modern AI-approaches, http://gameprogrammer.dk/data/Developing_an_Agent_for_Dominion_using_modern_AI-approaches.pdf, [Online; accessed 10-October-2019]

Tollisen, R., Jansen, J. V., Goodwin, M., & Glimsdal, S. 2015, in Current Approaches in Applied Artificial Intelligence, ed. M. Ali, Y. S. Kwon, C.-H. Lee, J. Kim, & Y. Kim (Cham: Springer International Publishing), 43

Wiki contributors. 2019, Dominion Strategy Wiki, http://wiki.dominionstrategy.com/index.php/Main_Page, [Online; accessed 5-December-2019]

Wikipedia contributors. 2019, Dominion (card game) — Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Dominion_(card_game)&oldid=918940837, [Online; accessed 10-October-2019]