Automatic Dominion-Playing Agent

CS221 Project Final Report

Monisha White (mewhite) and Nolan Walsh (njwalsh)
Stanford University, Fall 2014

Introduction

Dominion is a member of the rapidly shrinking class of games where the best AI players cannot consistently win against the best human players. This makes it a great candidate for testing the limits of Artificial Intelligence game playing techniques.

In Dominion, players take turns playing and purchasing cards in an effort to improve their deck and amass more victory points than their opponent. The game ends when all Province cards are gone, or when three other piles of cards have been bought out. The game contains a lot of inherent randomness, which means that a large sample size is required to accurately rate any single strategy. On any given game, if a mediocre player gets lucky he/she could beat a significantly better player.

One quality of Dominion that makes the game interesting from an AI perspective is the existence of different Kingdoms. A kingdom in Dominion is the set of cards available to purchase. Each game of Dominion is usually played with a new kingdom that neither player has played before. This gives the game a great deal of variety, since different types of play are rewarded depending on the cards available. This quality of the game creates an interesting problem for AI players. Unlike in games like Chess or Go, an artificial Dominion player must be trained for each kingdom separately, rather than once for the entire game.

For more information on Dominion, please visit: http://wiki.dominionstrategy.com/.

Task Definition

To measure the our players progress, we played it against a simple Big Money strategy. The Big Money strategy plays by the following rules:

Baseline

To measure the progress of our player, we played it against another player employing a simple Big Money strategy. The Big Money player played by the following rules:

1. If possible, purchase a Province card
2. Otherwise, purchase the highest valued treasure card

Big money strategies are often effective in Dominion, and can actually win against inexperienced players. However, they are rarely the best strategy available, and a better player will easily defeat such simple play. The Big Money strategy is simple to implement and quick to play, making it a good opponent for testing our player.

Success for our project meant building a player for Dominion that could beat the Big Money strategy on games with simple kingdoms.

Oracle

The Oracle for our project was a reasonably skilled human player. The most advanced AI players still cannot beat the best human Dominion players. When starting our project we hoped that our player would learn to perform well on simplified versions of the game with smaller kingdoms. We anticipated that a fully expanded game of actual Dominion would be beyond the scope of our project, and that if we implemented and attempted to train our player on a full game, we would be able to consistently beat it.

Related Work

Not many people have worked on creating high level AI players of Dominion. Of the few that exist, the most successful and best documented approach comes from Matthew Fisher, a recent PhD student at Stanford[1]. Fishers Dominion player uses lists of cards to purchase (Buy Lists) to parameterize an overall strategy for a dominion kingdom. This strategy works well, and Fisher claims that his player is competitive with experienced Dominion players.

The Buy List approach allows a player to train quickly on a given kingdom, but also limits the types of play available. Because of the randomness inherent in a game of Dominion, different strategies may work better across different games played with the same kingdom. A Buy List player will not adapt to these types of situations and will continue to play the same strategy regardless of how the opponent is playing.

In our work, we wanted to explore options for playing Dominion that did not have these types of limitations. Our player explores and rates every possible move each game which means it is not limited to any one strategy. This flexibility gives our player a much higher ceiling when it comes to advanced play.

Approach

Modelling Dominion

We modelled the game of Dominion as a Markov Decision Process. To create a new Dominion MDP instance we set the starting kingdom, starting deck, and starting hand; this allowed us to expand or restrict the game as much as we wanted. Initially, we included only a few cards in order to simplify the game while we tried out different learning algorithms. As we honed in on better algorithms, we expanded the game, using full hands, decks, and kingdoms.

States

Each state in our MDP contains all relevant information about a players current situation in the game:

[1] http://graphics.stanford.edu/ mdfisher/DominionAI.html

- the kingdom
- the players deck, hand, draw pile, and discard pile
- the turn number and the current phase (action phase or buy phase)
- the number of buys, actions, and money the player has left to use in their current turn
- the states of all opponents in the game

Reward

In Dominion, the winner is the player who has the most Victory Points in his deck when the game ends. The reward is defined as:

$$Reward = \begin{cases} 100 & \text{if } win \\ 0 & \text{if } tie \\ -100 & \text{if } loss \end{cases}$$

However, for training purposes, we wanted to give higher a bigger reward if a player wins by a large margin, and a more negative reward if it loses by a large margin. This helped our player improve even if facing a weak opponent. To gain higher rewards it would have to learn to play smarter and win by larger margins. We trained with the following reward function:

$$Reward = \begin{cases} 10 + 10(v1 - v2) & \text{if } win \\ 0 & \text{if } tie \\ -10 + 10(v1 - v2) & \text{if } loss \end{cases}$$

`where v1 is the number of victory points of the player, and v2 is the number of victory points of the highest scoring opponent.`

Cards

The real-world version of Dominion contains hundreds of types of cards, some with extremely complicated effects. Implementing the more elaborate Dominion cards takes a lot of time away from the real task at hand: creating an automatic player. To keep focus on our goal, we implemented a single kingdom of ten real Dominion cards, along with the treasure cards and victory point cards available in the Base Set of Dominion. These ten kingdom cards are the simplest cards in the game and are all some variation on +buys +money +cards +actions. The cards we implemented, along with their effects are listed below.

Playing Dominion

We split each turn of Dominion into two phases and used different algorithms to determine how to play in each phase.

Action Phase

The first phase in a turn of Dominion is the Action Phase, where a player must choose which action cards to play from his/her hand, and what order to play them in. Strategies for playing action cards generalize well across games and kingdoms. This allowed

us to implement a single Action Player that we used across many different kingdoms and iterations of our player.

Expectimax The branching factor during the action phase is much smaller than during the game as a whole. Each hand starts with only five cards, and often only one or two of those cards are playable as actions. However, because of the complex randomness inherent in drawing multiple cards, even with this small branching factor, calculating the resulting states from a handful of actions can still be computationally expensive. We decided to explore to a limited depth and use heuristics for evaluating action play.

Action Play Heuristics The general goal during the action phase is to maximize the amount of money that a player has, and the number of Buys that a player has to use that money. With that in mind, we created a simple scoring method that our action player tries to maximize:

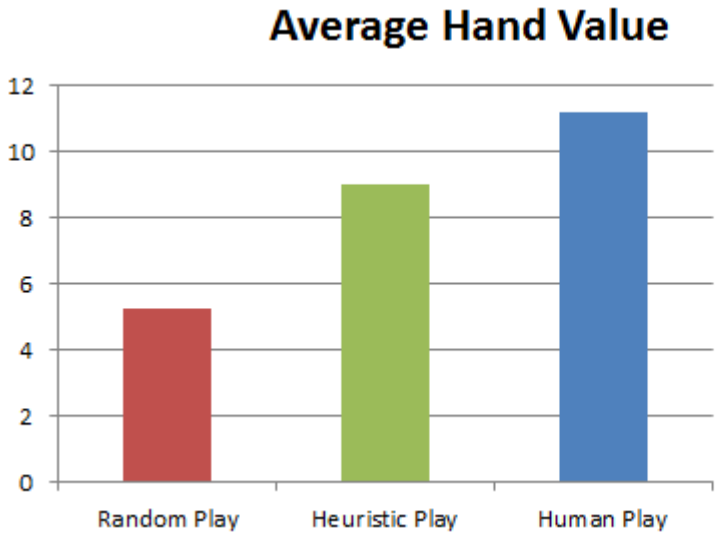
```
StateScore = money + ((buys - 1) * money / 4)
```

In Dominion, players must make choices about whether they want more buys or more money at the end of the turn. Buys become more important to obtain as the player amasses more money. For example, if a player has 2 buys but only 1 treasure, obtaining another buy is rather pointless. However, if the player has 2 buys and 24 treasure, an extra buy would give him/her much more flexibility.

The above heuristic works well for evaluating states at the end of the action phase, but it does not capture an important part of playing during the action phase: having more actions left to play. For this reason, we used a second heuristic to evaluate states that did not result in the end of the action phase:

```
StateScore = money + actions
```

This extremely simple heuristic gives a higher score to states that have more actions left, while still maximizing the money obtained.



Random Play	5.23
Heuristic Play	8.97
Human Play	11.18

The above table and chart was generated from the results of playing 100 random Dominion hands, each accompanied with a random deck. Using the heuristics and only exploring to a depth of two does not play as well as a human, but it does do significantly better than playing cards randomly. This is perfect for training our player, where speed is of utmost importance. When testing, or playing against human players, we can increase the exploration depth to closer approximate perfect play.

Buy Phase

During the buy phase, Dominion players make the most crucial decisions of the game. Choosing what cards to buy, and when, is what makes Dominion such a fun and complex game. The cards that a player buys form the basis of his/her strategy for that game. This is the phase of the game where our player was tasked with making decisions that would affect its success a long ways down the road.

Value Iteration On our very first version of Dominion we limited the game to three turns and only a few possible cards in the kingdom. At this stage we excluded any action cards, so the only relevant phase was the buy phase. This created a small enough game tree that we were able to run value iteration and solve the entire simplified game. This determined the best play to make at every state, giving us a good comparison for learning algorithms we were hoping to implement for our final player.

Q Learning/TD Learning We began by implementing two players, one that used Q Learning, and one that used TD Learning. Then we trained them and measured their performance in multiple simple scenarios.

One player, one Province, no actions:
Initially, we played a modified one player version of Dominion, with a goal to end the game as quickly as possible. The kingdom contained fewer cards, no action cards, limited numbers of each card, and only one Province card. In this version, Big Money is actually the ideal strategy, making it our Oracle. Big Money takes an average of 9.5 turns to buy the one Province and end the game. A random player takes an average of 80 turns. After training, both Q Learning and TD Learning averaged 11 turns, which was a promising initial result.

One player, four Provinces, no actions:
After adding more Provinces into the game, the Big Money strategy averaged 18.4 turns, while a random player averaged around 145 turns. The TD and Q Learning players were slow to converge. After thousands of iterations of training, they averaged 45 turns. Their improvement rate was slowing down, and though playing much better than random, their play was far from skillful.

The slow convergence rate we saw in TD and Q Learning after expanding the game led us to explore a new algorithm that we hoped would learn more effectively.

Average Reward Evaluation The Average Reward Evaluation algorithm has some similarities to TD Learning. It is also a reinforcement learning strategy that chooses actions based on the value of a state as computed by feature weights. However, the feature weights and the state values are computed in different ways. The weight of a feature is the average reward received from games that activated that feature.

```
weight[feature] = averageGameReward
```

The value of a state is the the mean of the average game rewards for all features of that state.

Our algorithm keeps track of every state it reaches throughout a game. At the end of the game, when it gets a reward, it incorporates that end reward (*gameReward*) into the weights of all features for each state along the way. The weight for each feature (activated *numVisits* times) in each state is updated as follows:

```
weight[feature] = (numVisits/(numVisits + 1)) * weight[feature] +
(1/(numVisits + 1)) * gameReward
```

To determine which action to take in a given state, it first calls on the MDPs succAndProbs function to return all possible successor states and the probability of getting into that state. The player scores each possible action by valuing each potential successor state, and weighting that value by the probability of the given action reaching that state. The value of a state represents the expected end of game reward given the player is in that state. An action score is the players expected end of game reward if a player takes a given action in its current state. It takes the action with the highest action score.

Using average reward evaluation converged much more quickly. The end reward of each game was factored into all states, even those that are only seen early on in games, so good choices early on in games is immediately reinforced.

One player, one Province, no actions: Similarly to TD Learning and Q Learning, it converged to around 11 turns to finish the game (in comparison with Big Moneys 9.5 turns).

One player, four Provinces, no actions: However, with four provinces, the average reward evaluation technique converged after 1,000 iterations to around 21 turns in comparison with Big Moneys 19 turns. (Note that TD Learning and Q Learning trained for much longer, still had not converged, and were averaging around 44 turns. Furthermore, the average reward evaluation player was using the same feature extractor as the TD Player.)

Features

We used many different features throughout the process, including features for the various different players we were initially using, features for our one player version, and features for the full Dominion that we focused on for our final player.

We will focus on the different features we tried out for our final Dominion player that used Average Reward Evaluation.

Average Money Value and Turn Number

One feature we use is an indicator on the average value of treasure cards in the players deck, along with the turn number divided by three (and rounded).

An essential part of Dominion strategy is having enough money in your deck that you can buy good cards and eventually buy Provinces, as they are worth the most Victory Points. Initially, the only money cards players have are Coppers, worth one coin each. In order to build your deck in a way that allows you to consistently buy Provinces, you need higher valued money cards. Buying Silver (worth 2 coins) and Gold (worth 3 coins) increases the average value of treasure cards in the deck.

We incorporated the players turn because at different stages of the game, different average money values are achievable and desirable. Late in the game, low average money is a negative attribute. However, after the first turn or two, the best play will still result in a low average money value. It was important to give our player the opportunity to recognize whether buying a card would lead to a good money average at its current point in the game. To generalize the feature a little more, we used turn/3, so that deck value in the first three turns, next three turns, etc, would each fall under the same feature.

Number of Each Card Type and Turn Number

For each card type, we added an indicator feature of the number of that card and the players turn/3. This way our player could learn how many of each card is ideal at every state of the game.

Victory Point Difference and Number of Provinces Left

We knew that we needed a feature to try to increase the number of victory points, since the player with the most victory points at the end wins. We also knew it had to relate to how far in the game the player was, since at the beginning of the game, building a strong deck is more important than buying victory points. We started with just the number of victory points our player had. We also started with the turn number, and changed it to the turn/3. However, we made two changes that allowed it to be much more successful at beating opponents as opposed to accumulating lots of Victory Points.

Our player takes into account the other players deck (good human players also track what cards their opponents buy), and calculates the difference between its own victory points and the highest number of victory points of any of the other players. It includes the number of Provinces left in the game as this is an indicator of how close the game is to the end.

Experiments and Results

As we progressed in creating our Dominion player, we developed many versions of the game, and trained different players at different stages. We progressed from one player to multiplayer. We started with only one Province, build up to four Provinces, and eventually trained and played with eight Provinces. We initially limited the kingdom to non-action cards, then we added in just a few simple action cards, and finally, we implemented and began training our players on a full Dominion kingdom.

At each stage of the game, we trained, experimented on, and developed our player in slightly different ways. The following training methods, experiments, and results pertain to our final Dominion player in two player games. They outline the progression to how we now, ultimately, train our player to prepare for play against any Dominion opponent.

Vs Big Money

Initially we attempted simply training against a Big Money Player. However, after thousands of games of training, our player had yet to win a single time. After observing our weights we speculated that since our weightless player started out playing randomly, it wasnt playing effectively enough to compare with Big Money. All rewards were so negative that better play was not sufficiently encouraged. We decided to start training against an easier opponent.

Vs Random Player

Training against a random player improved our player, and it quickly overcame the random player. But this did not prepare it for a Big Money matchup.

Vs Decreasingly Random Big Money

We decided to train our player against more and more advanced opponents as it improved. Instead of training against a purely random or purely Big Money strategy, we played it against a partial Big Money player that plays a random move with some probability (p). Once our player started beating a player that played randomly with probability 1, we lowered the randomness to .7. When our player began winning at that, we lowered the opponents randomness to .5, then .3, and finally we eliminated the opponents randomness entirely. We found that overtraining on a more random Big Money player led to worse play when we decreased its randomness, so as soon as our player began doing well on a partially random opponent, we decreased the randomness.

This method of training led to our first set of promising results.

Initially, the scope of our project was to play in simplified kingdoms. To create these kingdoms, we included all non-action cards and all eight Provinces, but added in limited extra action cards. After training on a kingdom, we tested our player in 500 games against a Big Money player.

Table 1: Results On Simplified Kingdoms

Extra Kingdom cards	Wins	Losses	Ties
Smithy	224	175	101
Laboratory	335	112	53
Smithy, Workshop, Village	405	112	53

With this level of success in the simplified kingdoms, we decided to expand our scope to a full Dominion kingdom. With the full game, we didnt expect to do as well against a Big Money player, so after training, we tested our player with 150 games against Big Money players of various randomness levels (1, .5, .3, and 0).

Table 2: Results on Full Kingdom Against Various Opponents

Opponent	Wins	Losses	Ties
Big Money	73 (48%)	11	66
BM (.3 Random)	123 (82%)	7	20
BM (.5 Random)	141 (94%)	1	8
Random	150 (100%)	0	0

Vs Self

Early on, we tried training our player against itself and didnt have much success. After changing the training reward to the variable reward (noted above) and adjusting our features, we attempted this method of training again. However, this time we trained our player for significantly longer than before.

Training our player against itself led to enormous improvements. We started with two identical players and 0 weights for every feature. The two players played hundreds of thousands of games against each other, each one storing and updating their own weights individually. After more than 24 hours of training, they were both ending the game with significantly more Victory Points in fewer turns than our player had been getting after training against the decreasingly random Big Money player. We ran a test of 1,000 games against Big Money with the new learned weights, and the win rates were even higher than before:

```
Win:  849/1,000
Lose:  134/1,000
Tie:  17/1,000
```

Cap Feature Weights Number of Times Visited

Finally, we realized that as we trained for longer, our weights updated at a decreased rate; the longer we trained, the slower we learned. This is because our player updates the average reward of features using the number of times each feature has been visited. If a feature has been visited 100,000 times, a games end reward barely affects the

weight. The new game reward is weighted at only 1/100,001 while the old average is weighted at 100,000/100,001 . We wanted our player to learn quickly even after training for thousands of iterations. To achieve this, we set a max value k on the number of times visited for each feature. We tested several values of k, and 250 worked well. After a feature had been seen 250 times, we stopped incrementing the number of times it was visited. Thus from then on, a games reward is (1/251) * (game reward) plus (250/251) * (previous weight).

Capping the number of times visited for all features had a huge impact on the learning speed, allowing our player to arrive at better weights within a few hours of training. After testing 1,000 games against Big Money using these weights, the results were as follows:

```
Win:  946/1,000
Lose:  47/1,000
Tie:  7/1,000
```

Note: After running this test we played a few games against the player ourselves. At this point our player won against us consistently, and we had to think hard to defeat it even once.

Conclusions and Future Work

We are very happy with our results. We expanded the scope of our project because we initially expected that it would be too difficult to implement and effectively train a player on a full version/kingdom of Dominion. However, after success on simplified kingdoms, we decided to progress to a full kingdom in Dominion. Our player plays significantly better than our baseline, beating Big Money 95% of the time. We were both surprised and thrilled to discover that our player was a formidable opponent even for us. In fact, its play seems to be more effective against our play style than against Big Money, possibly because training against itself was more similar to playing against a human than playing against the Big Money strategy.

Error Analysis

While we are very proud of our creation, there are still several areas where our play falters. Currently, our features do not capture possible three pile endings of the game (as mentioned earlier, Dominion can also end when three piles of cards are depleted). In order to anticipate these kinds of endings and to play well when they approach, we would have to add features that relate to how many piles are empty, or how many cards are left in the current game.

Near the end of a game of Dominion, our player does not play well from behind. For example, if our player has a choice to end the game by buying the final Province, it will often take that choice rather than buying a Duchy and hoping to be able to win on the next turn. This is a result of the reward function which we used to train our

player. Using a non-binary reward function made our player try to lose by as little as possible instead of trying to play for the greatest chance of a win. Solving this problem could be difficult, since using this reward function allowed our player to converge to better play much faster than using a simple win/lose reward system.

Future Work

The game of Dominion poses an interesting challenge in the field of AI. The existence of so many different kingdoms means that any automatic player must be able to quickly learn weights that lead it to play well on that kingdom. We only trained our player on one full kingdom of Dominion. It takes our player a couple of hours training against itself to play well on that kingdom. Speeding up this training process and implementing more cards would allow us to create a player that can learn to play many different kingdoms.

Matthew Fisher had good success parameterizing Dominion strategies as lists of cards to buy and using genetic algorithms to train these strategies against each other. This allows for very quick training, but not as much flexibility as our player. It would be interesting to explore a compound approach. Since our player learns from the players it plays against, we could first create a few buy lists for a given kingdom and then train our player against these strategies. This could combine the speed of genetic algorithms with the flexibility afforded by scoring every possible action.

Appendix

Cards Implemented

Card Name	Effect	Cost
Smithy	+3 cards	4
Village	+1 card +2 actions	3
Woodcutter	+2 money +1 buy	4
Laboratory	+2 cards +1 action	5
Market	+1 card +1 action +1 buy +1 money	5
Festival	+2 money +2 actions +1 buy	5
Bazaar	+1 card +2 actions +1 money	5
Worker's Village	+1 card +2 actions +1 buy	4
Great Hall	Victory Point Card: worth 1 Action Card: +1 card +1 action	3
Harem	Victory Point Card: worth 2 Treasure Card: worth 2	6
Copper	Treasure Card: worth 1	0
Silver	Treasure Card: worth 2	3
Gold	Treasure Card: worth 3	6
Estate	Victory Point Card: worth 1	2
Duchy	Victory Point Card: worth 3	5
Province	Victory Point Card: worth 6	8