Genetic Algorithm - Dominion Game Strategy

ME 575 Design Project

Jing-Song Huang & Ryan Larson

Project Summary

A genetic algorithm was used to optimize gameplay strategy for Dominion, a deck-building card game. Strategies were defined as a set of priorities for what actions to take, and were tested against an AI player using a known strategy that is fairly successful. The optimization was able to produce a strategy that had an average score margin of 27.72, and won 99.7% of games in a run of 1000 simulations against the known strategy.

Procedure

Model development:

Dominion is a complex deck-building card game that has been around since 2008. In this game, players try to buy victory cards (the only way to score at the end of the game) by incrementally building a custom deck of cards that will let them afford the victory cards. Besides victory cards, there are treasure cards, which are the currency with which players buy other cards, and action cards, which give increased options for a player on their turn. On each turn, a player may play one card and buy one card, using the treasure in their hand. Action cards may be played to modify the amount of treasure in a player's hand, get more cards in hand, or increase the number of allowable actions or buying opportunities on a given turn. Then the newly-bought cards and the just-played hand are discarded in an individual discard pile, and new cards are drawn from the player's individual drawing deck. When the drawing deck runs out of cards, the discard pile is shuffled and becomes the new drawing deck. In this way, a player can choose cards to build a deck that will frequently give them cards of their choice (i.e. buying more treasure cards will increase the likelihood that a player's next hand will contain treasure cards).

There are a myriad of strategies for playing Dominion, but in general, players are more successful if they stick to a strategy; the reason being that a deck with small numbers of lots of different cards will have low probability of giving a player what they need on any given turn. For this reason, Dominion players are very interested in determining which strategies will be the most profitable. We sought to use an optimization routine that would calculate the best strategy to counter a known opponent strategy.

We weren't aware of any analytical models for Dominion or other card games, so we constructed a model that was entirely code-based. We found a GitHub repository that uses Python to simulate the relative effectiveness of two Dominion strategies over a set number of games, but expresses it in terms of win percentage (https://github.com/rspeer/dominiate-python). Using this repository as a starting point, we constructed a Matlab model of Dominion gameplay that was compatible with the requirements of a real-value genetic algorithm.

The primary structure of the model uses custom classes to define important elements of gameplay, including cards, players, strategies, and games. These classes also include functions for interacting with each other and running a chosen number of games. To limit the complexity of the simulations, we chose to implement cards from the base game of Dominion only (there are many popular expansions), and of those cards we only chose to use the action cards with the simplest effects. Thus, the optimization that we performed can only be extended to a game that is set up with the cards we used.

For purposes of this project, we only implemented a single opponent, although the code is capable of handling up to three opponents (with corresponding increases in computational expense). We chose to have this single opponent use a strategy known as Big Money. This strategy is generally successful in real gameplay, and consists of only buying treasure cards and victory cards, rather than using actions. In this way, the player increases the likelihood that on any given hand they will have enough treasure to buy the victory cards they want. However, it also has some limitations, and some advanced strategies can beat it. We decided to make the opponent in our simulations use this strategy as a good baseline because it can be hard to beat for less experienced players.

For a fitness function, we chose to calculate the score margin from each game, meaning the difference between the optimized player's score and the highest-scoring other player, which is simply the only opponent in a 2-player game. The score margin was used as a more descriptive indicator of strategy

effectiveness, since it measures how *much* better one strategy was versus another, rather than simply indicating a win or loss. Each strategy was pitted against the opponent in 50 simulated games, and the average score margin across those 50 games was calculated and fed back into the optimization routine.

Validation of the model was performed by choosing known strategies and pitting them against randomly-generated strategies, then using text readouts to verify that the simulated gameplay was working according to the real game rules. Histograms of score margins were also generated to validate the model's ability to produce consistent results, as well as to evaluate the variance of actual game results using the chosen strategy.

Design variables:

On any given turn, a player has to make decisions about what cards to buy, and those decisions are affected by the cards they have in their hand at the time. Therefore, we decided to define a player's strategy as a list of priorities for card buying, actions, and trashing (permanent removal of a card from a player's deck, as opposed to letting a card cycle back through after being discarded). The priorities were implemented as integers that corresponded to an index in a list of possible cards, or continuous variables that corresponded to a condition to help determine whether to buy a certain card or not (for example, only buying a card if there aren't too many of that card in the player's deck already).

The design variables are grouped into four matrices, which are gain priority, gain cutoffs, play priority, and trash priority, and are altered in the genetic algorithm by switch, uniform crossover (U), and blend crossover (B) methods (see Fig. 1). For example, gain priority is a 2 x 17 matrix, where the first row is made up of non-repeated variables from 1 to 17. The integer in a given position within the first row of the gain priority matrix indicates the index of a preferred card that the player should buy, out of a known list. For example, if the third index in gain priority's first row is 1, then that indicates to the simulation that the player should first try to buy the third card in the card list. The card list was the same between all simulations in order to make this method possible. The second row in gain priority is a list of binary variables that determine whether the cards specified in the first row should be bought at all. If an index in this row has a value of 0, then it means the card corresponding to the same index in the first row shouldn't be bought. If the value is 1, then the card should be bought. In this way, strategies can be simplified to only include a few cards (which is generally more successful, from our experience playing the game). This method does have a drawback in that there are many ways to represent a given effective strategy. This means it doesn't really matter what the actual preference values are, as long as the relative order of the chosen cards stays the same; if your preferred cards are in indices 1, 2, and 3, it's no different from having them in 4, 7, and 11, as long as all the other cards are "switched off" and the relative order stays the same.

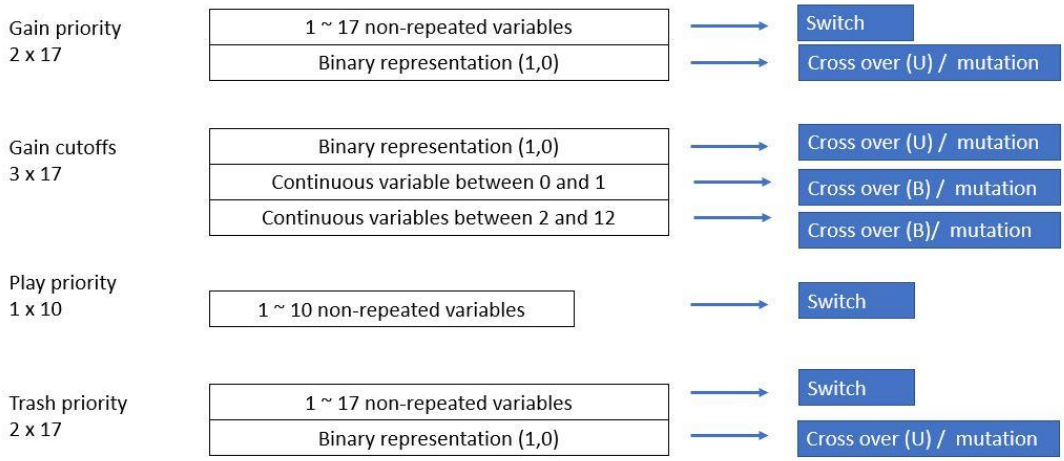


Figure 1: Details of each priority matrix are represented as rows of data, shown as white boxes. The blue boxes show the corresponding methods that alter the member variables.

In the variable matrix gain cutoffs, the first row is also binary, but in this case it acts as a switch to determine the type of condition that should be used on the gain priority. The second row of gain cutoffs is a continuous variable between 0 and 1 that corresponds to a percentage that should be maintained. For example, a value of 0.1 would indicate that for the corresponding card, the player should always prefer to buy it, but only if the percentage of that card in their deck is less than 10% at the time of buying. This is a useful constraint in normal gameplay because it can prevent a player from inflating their deck with cards that aren't useful in large quantities. The third row in gain cutoffs is a continuous variable with values between 2 and 12, and is used to check against the number of victory cards that are left at the time of a player's turn. Since the game end is determined by one of the victory card piles being completely drawn, this is a useful way to tell how close the game might be to ending. This can help a player buy the most important cards early in the game, and then focus on racking up points as the game comes to a close. The switching variable in the first row of gain cutoffs tells the simulation which condition to use for a given card index. For example, if the first index has 0 in the first row, 0.25 in the second row, and 5.5 in the third row, the player will use a percentage constraint of 25% on buying the first card in the card list. If the second index in gain cutoffs has 1 in the first row and the same values for the second and third rows, then the player will use a "cards left" constraint on the second card in the card list.

Play priority and trash priority both behave similarly to gain priority, with the only difference being that play priority only has a single row, and corresponds only to action cards. We set up play priority this way because we couldn't think of a situation where a player would have an action card, but not want to play it if they could.

There are many additional conditions or variables that could have been used, but we settled on the aforementioned variables because we felt they were a sufficient representation of complex strategy for our purposes. In reality, a player might use many other indicators to determine their strategy, including evaluating the choices made by their opponent, which our simulation doesn't do (it optimizes blindly, only knowing by how much it wins or loses).

Genetic algorithm implementation:

The genetic algorithm implementation is separated into five parts, which are selection, crossover, mutation, switch, and elitism.

The selection is implemented with a tournament. A number of random designs are chosen and they are fed into the model to get the fitness value. The design with the highest fitness becomes a parent for the next generation. After a desired number of parents are selected, the program enters the next stage.

Crossover is only applicable to rows of data that aren't non-repeated, so the first row of gain-priority, play-priority, and the first row of trash-priority do not use this method. If a random value is smaller than our threshold, we cross between two random parents. This checking process is done for each participating row. Crossover process is done on each cell by using the formula (see Table 1), binary variables are implemented with uniform crossover, while continuous variables are implemented with blend crossover.

The mutation is also divided in two methods corresponding to binary variables and continuous variables. Similar to the crossover, a random number is drawn for each row of the data. If certain rows' random number is less than the threshold, the mutation is performed on that row. For continuous variables, we use dynamic mutation (see Table 2 for equations). We choose beta to be 1, so alpha value will continuously decrease as the number of generations increases. Thus, towards the end of generations, the better solution will not likely be altered. For binary variables, we simply use a uniform random generator that produces 0 or 1 in an equal probability.

Switch method is special to certain rows of data as mentioned earlier . A random number is drawn for each row to check if that row is eligible for switching. If it is, the following procedure is done on each current cell. A random cell in the same row is chosen and switches the value with the current cell.

Finally, elitism is performed on all designs of the current generation. The number of the best designs is equal to the number of parents. However, we choose another three designs that have the top score to be preserved, which means they participate in mutation, switch, crossover, but their design variables won't be changed.

if $r \leq 0.5$ $y_1 = x_2$ $y_2 = x_1$ if $r > 0.5$ $y_1 = x_1$ $y_2 = x_2$	$if\ r \leq threshold \quad y_1 = (r)x_1 + (1-r)x_2$ $y_2 = (1-r)x_1 + (r)x_2$

Table 1: (left) equations for uniform crossover. (right) equations for blend crossover. r is a uniform random value. y1, y2 are children and x1, x2 are parents.

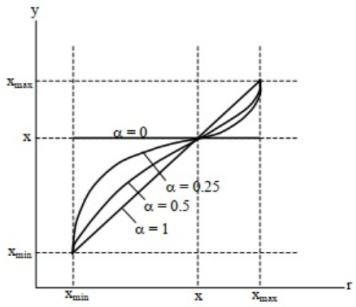
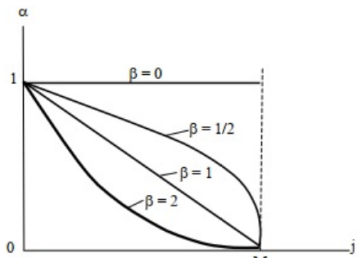
if $r \le x$ $\quad y = x_{min} + (r - x_{min})^{\alpha}(x - x_{min})^{1-\alpha}$ if $r > x$ $\quad y = x_{max} - (x_{max} - r)^{\alpha}(x_{max} - x)^{1-\alpha}$	$\alpha = \left(1 - \frac{j-1}{M}\right)^{\beta}$
 Fig. 6.4: Dynamic Mutation	 Fig. 6.5: Uniformity Exponent α

Table 2: (top-left) equations for dynamic mutation. (top-right) equation to determine alpha value. r is a uniform random value. y is child and x is parent

Tuning:

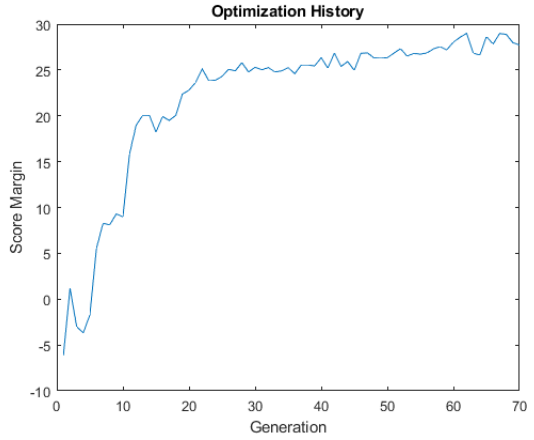
With our algorithm, it has a better result with larger population and larger generation, because the diversity of the designs can provide different combinations of the variables in our model. Another important factor is the number of population for the tournament. If larger, it has a higher chance to get a better design to be a parent. In our case, the fitness function, which is the average score over 50 simulated games, gives a pressure to our design genes. The design with lower score won't be selected at the tournament and elitism process. The genes with the better design variables in the remaining population get preserved for the next generations to do mutation, crossover and switch. Thus, besides large populations and generation size, we need to have a decent chance for each variable to perform crossover, mutation, and switch among themselves.

As the result of testing different thresholds for crossover, mutation, switch, we find it is better to keep the mutation rate to be around 20 percent, crossover rate to be around 40 percent, crossover rate to be around 60 percent. This is because we don't want too much random change. Also, mutation on the binary rows of the data can result in very large changes, since binary data control the on and off of playing certain cards or decisions. The higher crossover rate can be desirable because different combinations of the good design may produce other good effects. Overall, after running multiple times, the threshold for switching and mutation is considered good if they are around the number mentioned before. The most important factors, though, are the larger population size, generation and tournament size.

Results and Discussion

The system we chose has a huge amount of randomization, which makes it very difficult to optimize. This is an inherent part of a card game, since a player draws their hand randomly. For practicality, we chose to simulate only 50 games per fitness function call, which gives us a good population of testing with a particular strategy (checked with histograms of game results). Due to the inherent randomization, the same strategy might not produce the same fitness value each time 50 games are simulated. However, the fluctuation of the fitness for the same design between generations doesn't affect the optimization too much (see Figure 2 left), because 50 games give us a good normally distributed curve of fitness values.

Figure 2 shows that the genetic algorithm behaves as intended, because the score margin grows and becomes relatively steady after about 20 generations (although it's clear that it is still growing slowly). In order to confirm that this was the best strategy, we used the optimized strategy in our model for 1000 games (see Fig. 2 right). If the score margin is greater than zero, the optimized strategy won for a given game. Thus, our optimized strategy has a 99.7% win rate against one opponent that is using the "Big Money" strategy. This shows the effectiveness of choosing the average score margin over 50 games as our fitness function. It is not necessarily that we want the highest score margin; however, a higher average score margin means the bell-shaped curve will shift away from zero, meaning that the strategy is more likely to win even with the randomness of the game.



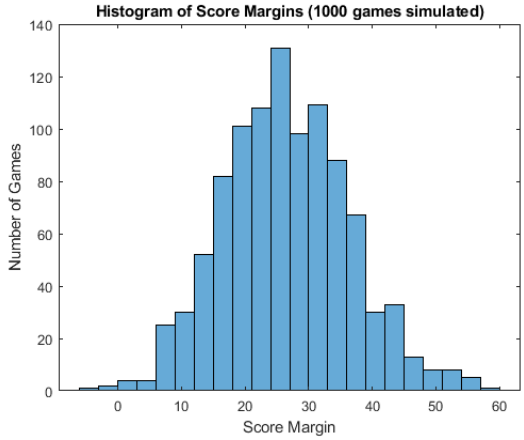
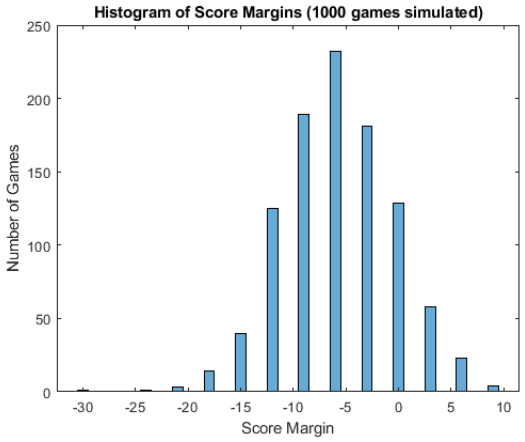


Figure 2 (left) Optimization history with population size of 40 and tournament size of 4. After 70 generations, the score is approaching 27, which is the best optimum we found. The crossover rate is 0.6, the switch rate is 0.4, and the mutation rate is 0.2. *(right)* With the optimized design from the left, it shows that we get a 99.7% win rate over 1000 games.



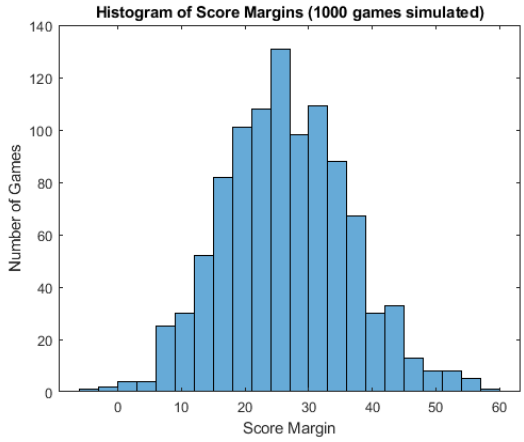
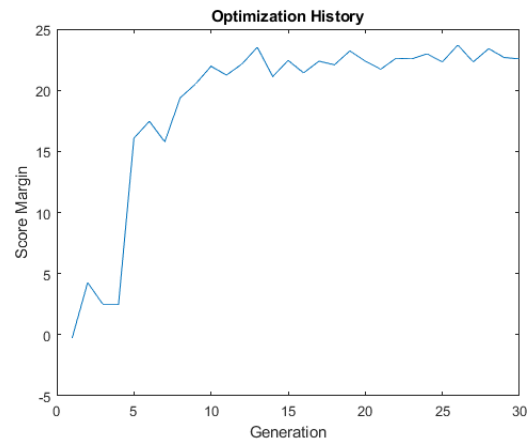


Figure 3 Comparison of effectiveness for the best strategy generated during the first generation (left) and the best strategy generated after 70 generations (right). These correspond to the optimization process in Figure 2.



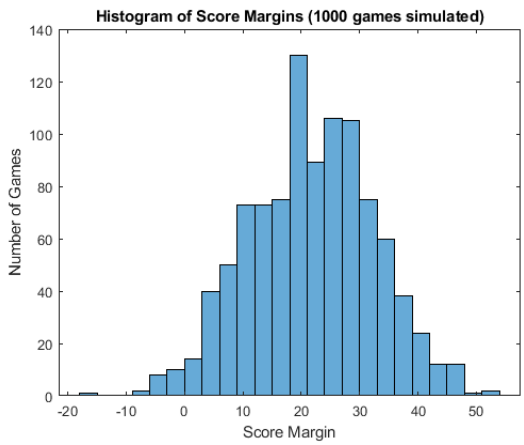
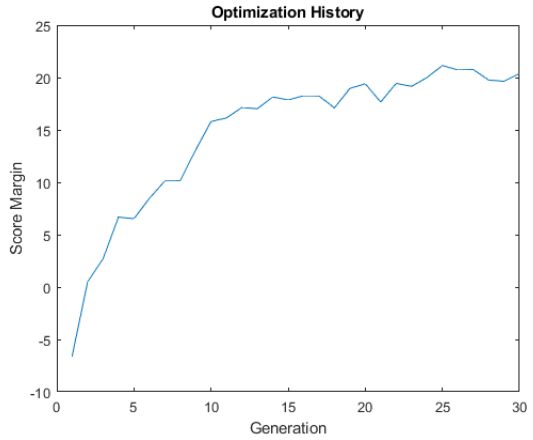


Figure 4 (left) Optimization history with population size of 30 and tournament size of 3. After 15 generations, the score approaches 22, which is a reasonable optimum. The crossover rate is 0.6, the switch rate is 0.3, and the mutation rate is 0.2. *(right)* With the optimized design from the left, it shows that this strategy wins 98% of the time over 1000 games.

In order to show the consistency of the resulting design, we simulated it. Another result is shown in Figure 4 with slightly different crossover, mutation, and switch rate. Each of the simulation takes about 2 hours to complete (although our best design took over 6 hours to run), so it is not feasible to try all combinations of different mutation, switch, and crossover thresholds. Thus, we tested some extreme cases to demonstrate low population size is bad in our system (see Figure 4). With the low population, the genes are less diverse, causing it to have a hard time growing. Another extreme case is to use a low switch rate (see Figure 5), the algorithm has a hard time giving an average margin score above zero. There are other cases with extreme high mutation, extreme low crossover that didn't produce good results.



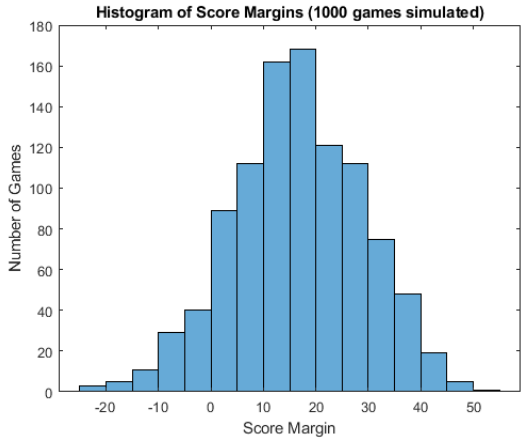
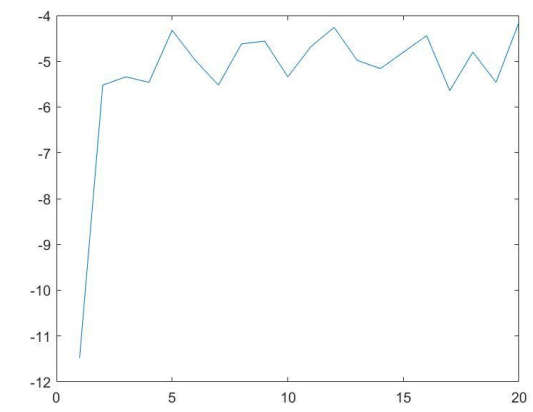


Figure 5 (left) Optimization history with population size of 30 and tournament size of 3. After 30 generations, the optimum is at 21. The crossover rate is 0.6, the switch rate is 0.4, and the mutation rate is 0.1. *(right)* With the optimized design from the left, it shows that this strategy wins 95% of the time over 1000 games.



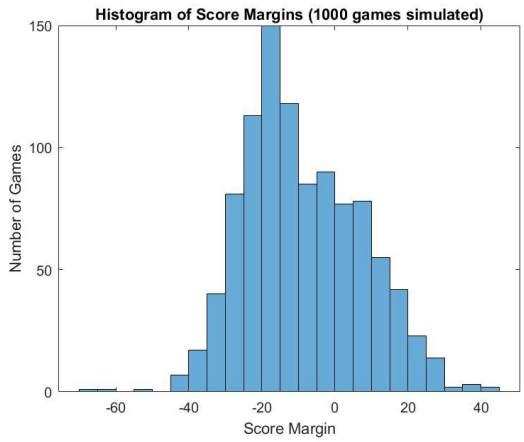
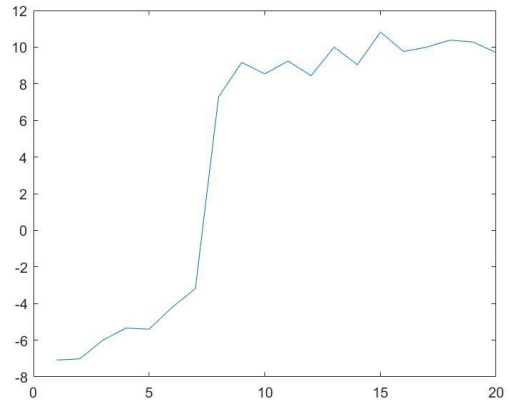


Figure 6 (left) Optimization history with population size of 10 and tournament size of 3. After 20 generations, the best design still hasn't gotten above an average score margin of 0. The crossover rate is 0.6, the switch rate is 0.4, and the mutation rate is 0.2. *(right)* With the optimized design from the left, it shows that this strategy loses more than 50% of the time over 1000 games.



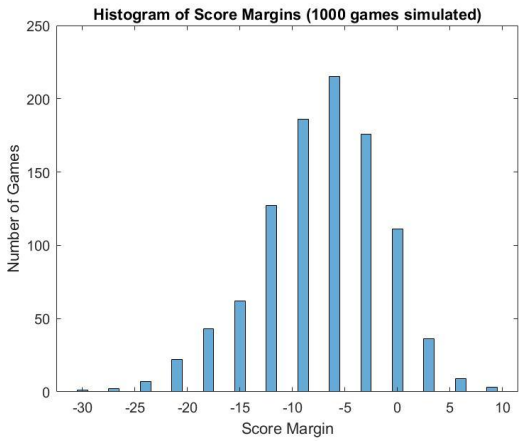


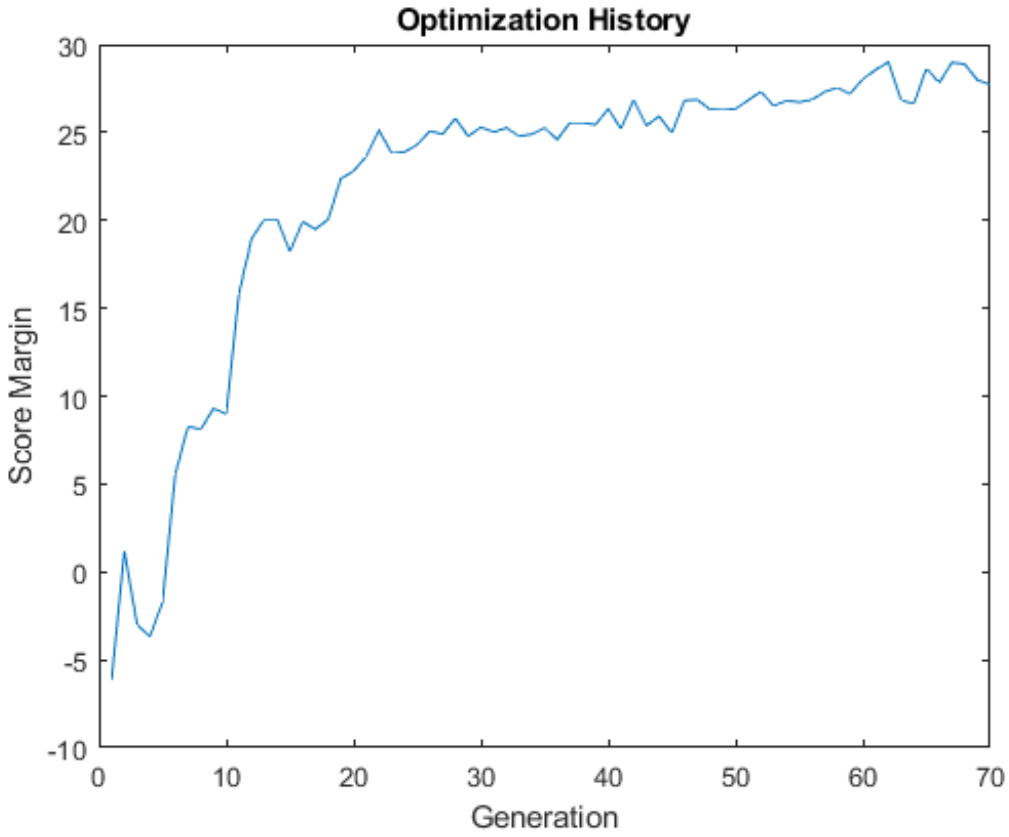
Figure 7 (left) Optimization history with population size of 20 and tournament size of 3. After 20 generations, the optimum is at 9.7. The crossover rate is 0.7, the switch rate is 0.2, and the mutation rate is 0.2. *(right)* With the optimized design from the left, it shows that this strategy loses more than 50% of the time over 1000 games.

Conclusion

It is necessary to have a binary variable as an on and off option to control whether our design should play a certain card at all. This may make some of the variable changes have no effect on the strategy, thus, the best result may produce many designs that have the same result. Also, the genetic algorithm doesn't observe opponents and change the strategy during games, there might be other areas beyond this project that we can further implement. Finally, as shown in figure 2, the genetic algorithm was able to produce a result that has a really high rate of winning against one opponent's consistent strategy.

Appendix

Detail of best optimization figure (seen in Figure 2):



Readable version of the best optimized strategy:

```
Player Strategy:
Play action cards in this order:
     Festival
     Woodcutter
     Market
Buy these cards in this order of preference, under conditions:
     Province if percentage of Province in deck is less than 0.86
     Duchy if a single victory pile or three action piles have 7 or less cards left
     Market if a single victory pile or three action piles have 5.102966e+00 or less cards left
     Estate if a single victory pile or three action piles have 4.457555e+00 or less cards left
     Gold if percentage of Gold in deck is less than 0.58
     Copper if percentage of Copper in deck is less than 0.37
     Festival if percentage of Festival in deck is less than 0.38
     Woodcutter if a single victory pile or three action piles have 12 or less cards left
Trash these cards in this order:
     Estate
     Province
     Village
     Council Room
     Copper
     Curse
```

Readable version of the worst design from the optimization run in Figure 2:

```
Player Strategy:
Play action cards in this order:
     Bureaucrat
     Council Room
     Village
     Witch
     Market
     Festival
     Smithy
     Woodcutter
Buy these cards in this order of preference, under conditions:
     Estate if percentage of Estate in deck is less than 0.97
     Council Room if a single victory pile or three action piles have 2 or less cards left
     Village if a single victory pile or three action piles have 2 or less cards left
     Festival if a single victory pile or three action piles have 7 or less cards left
     Bureaucrat if a single victory pile or three action piles have 2 or less cards left
     Witch if a single victory pile or three action piles have 8 or less cards left
     Silver if percentage of Silver in deck is less than 0.58
     Duchy if percentage of Duchy in deck is less than 0.28
     Smithy if percentage of Smithy in deck is less than 0.63
     Woodcutter if percentage of Woodcutter in deck is less than 0.88
     Gold if a single victory pile or three action piles have 8 or less cards left
     Market if a single victory pile or three action piles have 4 or less cards left
     Copper if percentage of Copper in deck is less than 0.87
Trash these cards in this order:
     Festival
     Curse
     Chapel
     Silver
     Province
     Woodcutter
     Witch
     Smithy
     Duchy
     Village
     Estate
     Copper
     Council Room
```

Cards used for simulation, in order:


1
COPPER
1
1
0
TREASURE


SMITHY
+3 Cards
4
ACTION


WITCH
+2 Cards
Each other player gains a Curse.
5
ACTION - ATTACK


VILLAGE
+1 Card
+2 Actions
3
ACTION


WOODCUTTER
+1 Buy
+ 2
3
ACTION


FESTIVAL
+2 Actions
+1 Buy
+ 2
5
ACTION

Market

+1 Card
+1 Action
+1 Buy
+ 1

5 Action

Illustration: Matthias Catrein © 2016 Rio Grande Games

Bureaucrat

Gain a Silver onto your deck. Each other player reveals a Victory card from their hand and puts it onto their deck (or reveals a hand with no Victory cards).

4 Action - Attack

Illustration: Matthias Catrein © 2016 Rio Grande Games

Council Room

+4 Cards
+1 Buy

Each other player draws a card.

5 Action

Illustration: Matthias Catrein © 2016 Rio Grande Games

Moat

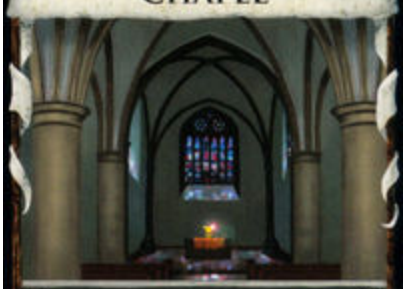
+2 Cards

When another player plays an Attack card, you may first reveal this from your hand, to be unaffected by it.

2 Action - Reaction

Illustration: Matthias Catrein © 2016 Rio Grande Games

Chapel



Trash up to 4 cards from your hand.

2 Action

Illustration: Matthias Catrein © 2016 Rio Grande Games

Code:

See code repository at https://github.com/ryantheengineer/dominiopt for details.

Explanations:

Bold names indicate the code that will most likely be of interest.

- **GeneticAlgo.m:** Main genetic algorithm script. Calls everything else to run an optimization routine. Number of generations, population size, and tournament size are selected here, as well as which cards to include in the optimization.
- **crossOver.m:** Crossover threshold can be altered here
- **mutate.m:** Switch and mutate thresholds can be altered here. The switch method is implemented in this file with mutation.
- **tournament.m:** Tournament selection function
- **Dominiopt.m:** This is the function that is called by GeneticAlgo.m to get the fitness function value for a given design. The number of games simulated can be altered here, as well as the number of opponents and strategies used.
- Card.m: Card class definition
- cardlist.m: Script that initializes all the different cards that can be used in the game
- chooseOpponent.m: Function for choosing one of the known strategies (in the end, we only used one of these for time's sake)
- **Dominion.m:** This is essentially the same as Dominiopt.m, except it's done directly in terms of custom classes instead of straight variable values. Dominiopt.m drives this function in GeneticAlgo.m, and we use this directly in testdrive.m to get histograms and more detailed test data for a given strategy.
- Game.m: Game class definition
- gene.m: gene class definition, used in GeneticAlgo.m
- getScore.m: Function for calculating scores
- interpret_gene.m: Takes an optimized gene object and interprets the values so that the strategy can be read easily
- Player.m: Player class definition
- random_strategy.m: Function used during validation testing of the model
- showcards.m: Function used during validation testing of the model
- shuffle.m: Function used during validation testing of the model
- Strategy.m: Strategy class definition
- **testdrive.m:** Script used for validation testing during model development, as well as more detailed simulation of optimized strategies
- testing.m: Early validation of methods to be used in the model