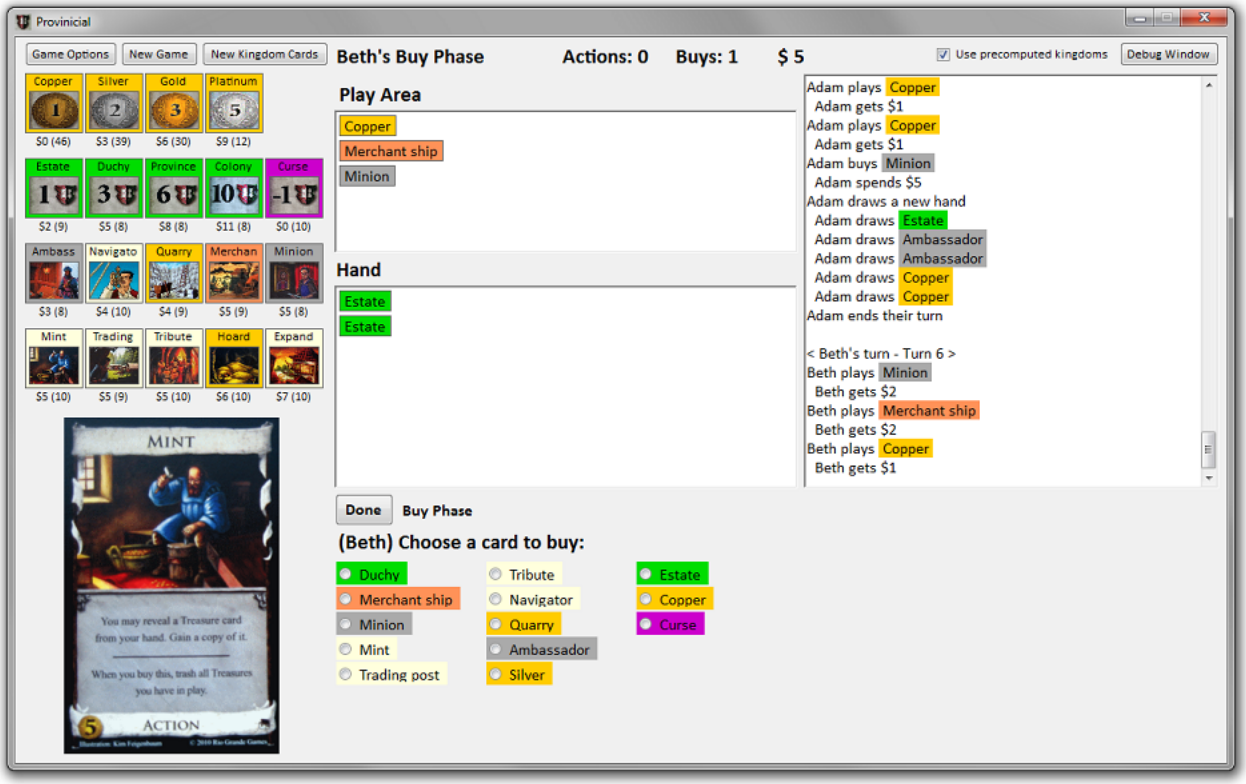
Matt's Webcorner

Stanford 2014

Provincial: A kingdom-adaptive AI for Dominion



Download link

This AI is easily competitive with experienced Dominion players. If you are not interested in the theory of writing an AI for Dominion, skip to the Bu Strategies section or play around with the application linked above.

Contents

- 1. Designing an Artificial Intelligence for Dominion
- 2. Parameterizing a Dominion Strategy
- 3. Buy Strategies
- 4. Simple and Complex Kingdoms
- 5. Evaluating New Cards
- 6. Algorithm Details
- 7. Future Work
- 8. Code

Dominion is a deck-building card game designed by Donald X. Vaccarino and published by Rio Grande Games. It is a very popular card game wit card interactions and a significant online following on sites like Goko and Isotropic, although at present you will have trouble finding an AI opponer compete with skilled players. One of the main attractions of Dominion is that the game begins by choosing ten cards at random to form an initial s cards (referred to as "the kingdom cards" or just "the kingdom" for the game). This makes the game very challenging to play well because every g will have to make decisions about how to play a kingdom, despite never having played with this exact set of cards before. Experienced players wil decisions based on prior games with similar, but still quite different, kingdoms. While this makes it possible to transfer knowledge about combinati

ratios of cards that work well together, there are still likely to be aspects of the new kingdom that are very challenging to understand without exten with those cards. The new kingdom might have novel combinations of cards that work extremely well together, or it might have cards that are effe disrupting otherwise powerful strategies. Expert players must make difficult judgments regarding what the dominant strategies for a new kingdom how effective these strategies will be against each other. To make things even more challenging, Dominion is a game with a considerable amount behavior. Since a very effective strategy has a chance of losing to a mediocre one, players must average their experiences over a huge number o games to learn how to make good decisions.

---

1. Designing an Artificial Intelligence for Dominion

The same properties that make Dominion both interesting and challenging for humans extend to designing effective AIs for Dominion. There are n different objectives and constraints an AI for Dominion might have, and a wide range of approaches that could be taken in designing one. At one e spectrum, the focus might be on the academic value of the project: for example, avoiding any form of heuristic play and comparing against known techniques such as competitive co-evolution. These projects are very interesting, but unfortunately such academic generality often sacrifices both effectiveness (resulting in weaker AIs) and computational efficiency (resulting in extremely long training times). One existing academic-centered a a Dominion AI can be found here. Although interesting, this work is limited to the base game, which has comparatively simple strategic behavior, a trained on specific kingdoms, which makes it very difficult to play effectively.

At the other end of the spectrum, there is the approached used by Goko where the AI is trained using a lengthy list of specific rules about what co of cards to purchase and in what ratios, and precise techniques for how to play each card that attempt to account for every possible scenario. In p such an AI is quite fragile, cannot easily adapt to new cards, and Dominion is sufficiently complex that it is likely going to miss important cases, re overall usefulness and effectiveness.

Provincial is a Dominion AI I wrote that attempts to strike a balance between academic value, effective play, and general usability. When designing primary goal was to produce results that are useful to experienced Dominion players. To accomplish this, the AI needs to be parameterized in a w be easily understood by humans — techniques such as neural networks and support vector machines, although they can be very effective at mac learning tasks, are effectively black boxes whose internal behavior are challenging for even machine learning experts to understand. To be useful players, the AI also needs to be able to compete at a high level of play, and produce results in a reasonable amount of time. The core of Provincia aggressive strategiziation framework based on competitive coevolution that takes a given kingdom and runs millions of games against itself, produ evolving list of dominant strategies. A tournament is then run between all the leading strategies and the results visualized to the user (see below f examples). Here is a summary of some of Provincial's key features:

- **Kingdom-specific training —** The AI in Provincial studies a given kingdom and develops a set of kingdom-specific strategies. This mimics t experienced humans play the game and results in a very effective player, but is also a drawback because it means the AI must be retrained a new kingdom. If the AI plays against a kingdom set it has not been trained on, it will make essentially random play decisions resulting i performance.
- **Generational co-evolution —** To determine which strategies are the best, the AI starts with a random pool of candidate strategies, and a m set of dominant strategies. All strategies in the candidate pool are then tested against the leading strategies, and those that do the best leaders in the next generation. The new candidate pool is formed as mutations of the new leaders, and the cycle repeats.
- **Human readability —** The AI uses a fairly simple parameterization over the space of Dominion strategies, explained in greater detail below. A more restricted parameterization does not exploit some of the power offered by more complex machine learning techniques, the ability to b humans makes the AI considerably more interesting. In the Future Work section, I'll talk about what advantages can be gained by us sophisticated but opaque model.
- **Performance —** The AI runs games extremely fast. On an eight core machine, a typical rate is 40,000 games per second. This is partially choice of language for the backend (C++), partially due to multithreading, and partially due to the simplicity of Dominion. A typical Dominion somewhere between 20 and 30 turns, and the majority of turns involve processing only a few simple behaviors, such as putting treasure deciding what to buy. Nevertheless, training on a new kingdom still takes several minutes, since hundreds of thousands of games must be r generation.

---

2. Parameterizing a Dominion Strategy

In its most general form, the problem statement for a Dominion AI presented with a specific kingdom to play on goes something like this:

- Search over all possible strategies to find those that win the highest percentage of games when played against all other possible strategies.

This statement overlooks many important subtleties, such as strategic dominance and Nash equilibrium. These interesting topics aside, however, question remains: how can we quantatively define a strategy in Dominion?

In artificial intelligence literature, one classic way to approach this problem is to model Dominion as a partially observable Markov decision proces paradigm, a strategy in Dominion is a map from the current game state (which encodes all information observable to the current player), to the act player controlling the current decision should make. Unfortunately, such a general method of defining a strategy is impractical for our purposes, si

game of Dominion can be in an effectively unlimited and non-continuous number of states and we require that the map from states to actions be h readable.

Provincial breaks up a Dominion strategy for a kingdom up into two largely disconnected components: a **buy strategy** that controls what cards sh bought during the buy phase or in response to a gain event, and a **play strategy** that controls how to play cards or otherwise respond to decision this decomposition, the **buy strategy** encodes the long-term goals of the AI — what combinations and ratios of cards will result in the highest vict total by game end, while the **play strategy** encodes the immediate goals of the AI — what is the best way to play the cards currently in hand. Det good **buy strategies** is by far the more challenging of these two components and the one most specialized to the kingdom; almost all of the AIs c time is spent working out which combinations of cards are most effective. On the other hand, good **play strategies** tend to generalize well across kingdoms and rarely requiring understanding the long-term behavior of the AI.

---

3. Buy Strategies

Provincial uses a simple model to represent what cards a given AI should buy. The core component of a buy strategy is an ordered list of card-inte which we'll refer to as a *buy menu*. The card represents what card to purchase and the integer is how many of that card should be bought. For rea will become apparent, the same card can appear multiple times as different entries in the menu. The buy rule is straightforward: buy the left-most can afford. Once a card is purchased, its corresponding menu entry is decremented. Naturally, if you cannot afford a card or there are none left in you skip over it to the next card, or buy nothing if you run out of menu entries. This menu is most often used for the Buy phase, but is also invoke response to any event that causes you to gain a card such as Workshop and Saboteur. The second component of a buy strategy controls when to estates, duchies, and provinces as a function of how close the game is to ending. For now we use a simple model for this: the AI is parameterized integers (3 for kingdoms with Prosperity cards) that control how many provinces (or colonies) must be left before the AI will override its buy menu purchase victory cards instead. To make the visualizations as short as possible, the AI is always greedy in buying the most expensive victory card this requirement is easy to remove (see the Future Work section).

Provincial trains good buy strategies for a kingdom, and the resulting visualizations are one of its most powerful features. To make this buy proces concrete, let's look at a sample visualization:

We refer to this type of visualization as a *leaderboard*. It encodes the current kingdom, the five dominant strategies at the current generation, and expected win ratio when the strategies are used against each other.

- **Kingdom cards —** Shows the ten supply cards, the starting hand condition (3-4 versus 2-5 copper split), and the presence of expans modifications to the supply such as platinum and colonies from Prosperity or Shelters from Dark Ages.

- **Dominant strategies —** A visual description of the buy menu for each of five strategies, and the conditions under which each strategy w estates, duchies, and provinces. These follow the buy menu algorithm described above. For example, if strategy A has 3 coins, it will first buy a second time it has 3 coins, it will buy a Chapel, and then it will transition to purchasing Shanty Towns. On the other hand, strategy B will Chapel the first time it has 3 coins. The strategy also indicates when the strategy switches to purchasing victory cards. For example, strateg estates when there is only one colony left, and will buy duchies and provinces when there are two or fewer colonies left. The number shown r strategy is the expected win ratio of this strategy when played against the other four leading strategies.
- **Tournament —** This table shows the expected win ratio when each of the five strategies plays against each other strategy, averaged over 10, The value shown is the percentage of games the row player can expect to win compared against the column player. Concretely, if the row play of the games and the column player wins Y% of the games, the corresponding entry will be (X - Y)%. A value of +100% means the row player a a value of -100% means the column player always wins, and a value of +50% corresponds to a win ratio of 75% to 25%. Values shown in g statistically significant and indicate that the two players are essentially tied. Statistical anomalies can still occur even when running 10,000 example, strategy E playing strategy E should have an expected win ratio of 0%, but shows up as -2.3%.

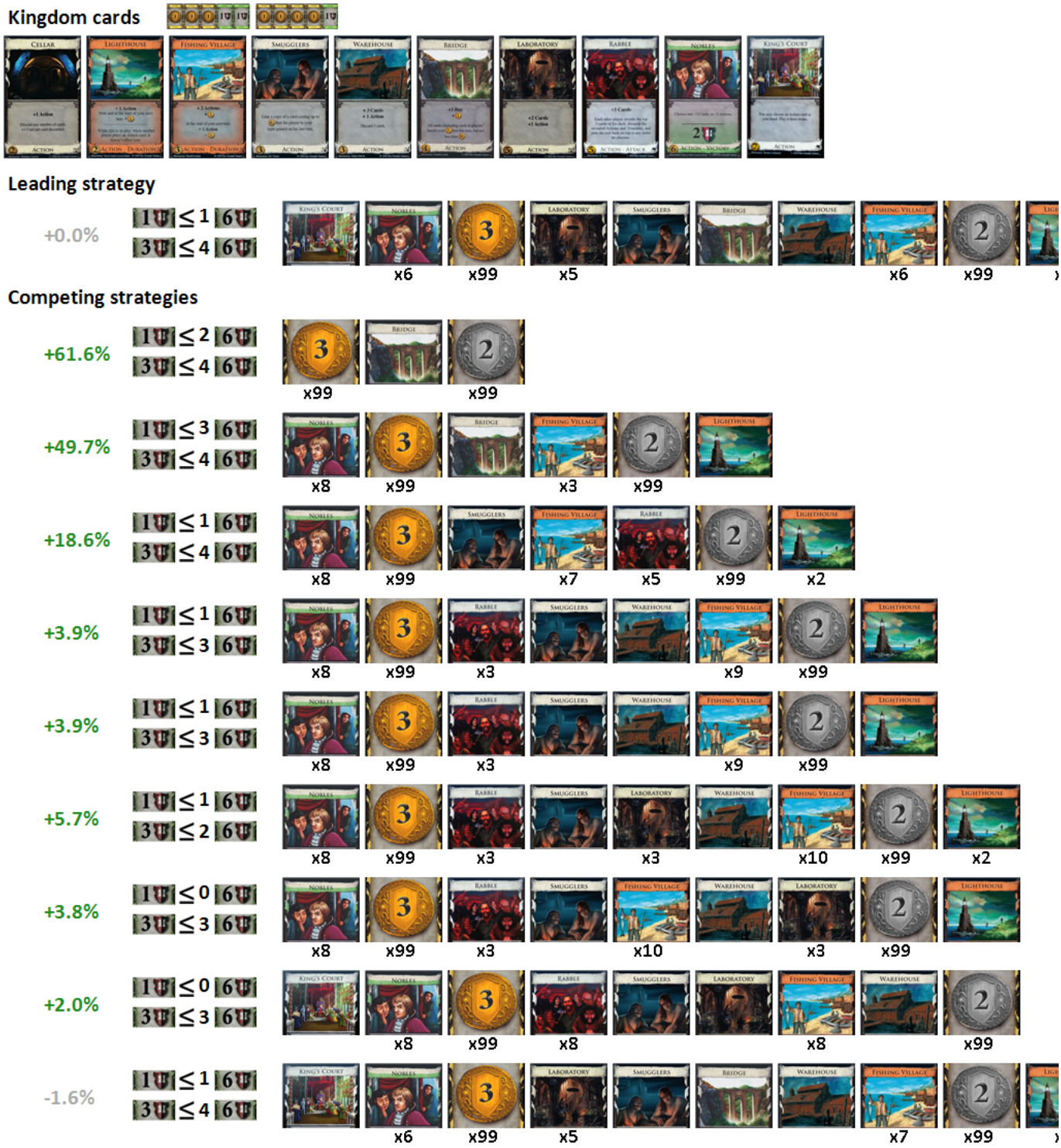
It's important to note that these visualizations only represent the leading strategies at a specific stage in the evolutionary process. Strategies from generations might be very poor quality as the AI is still working out which combinations of cards work well together. Even strategies from later gen might still be undergoing important evolutionary changes, because some parts of the strategy space might be subtle or hard to explore. For exampl purchasing Conspirator might be very detrimental to a strategy until a sufficient number of other "+action" cards are added and found viable.

For many kingdoms, the AI will have no trouble capturing "classic" strategies in a reasonable number of generations. For example, the canonical " + bridge" game:

4. Simple and Complex Kingdoms

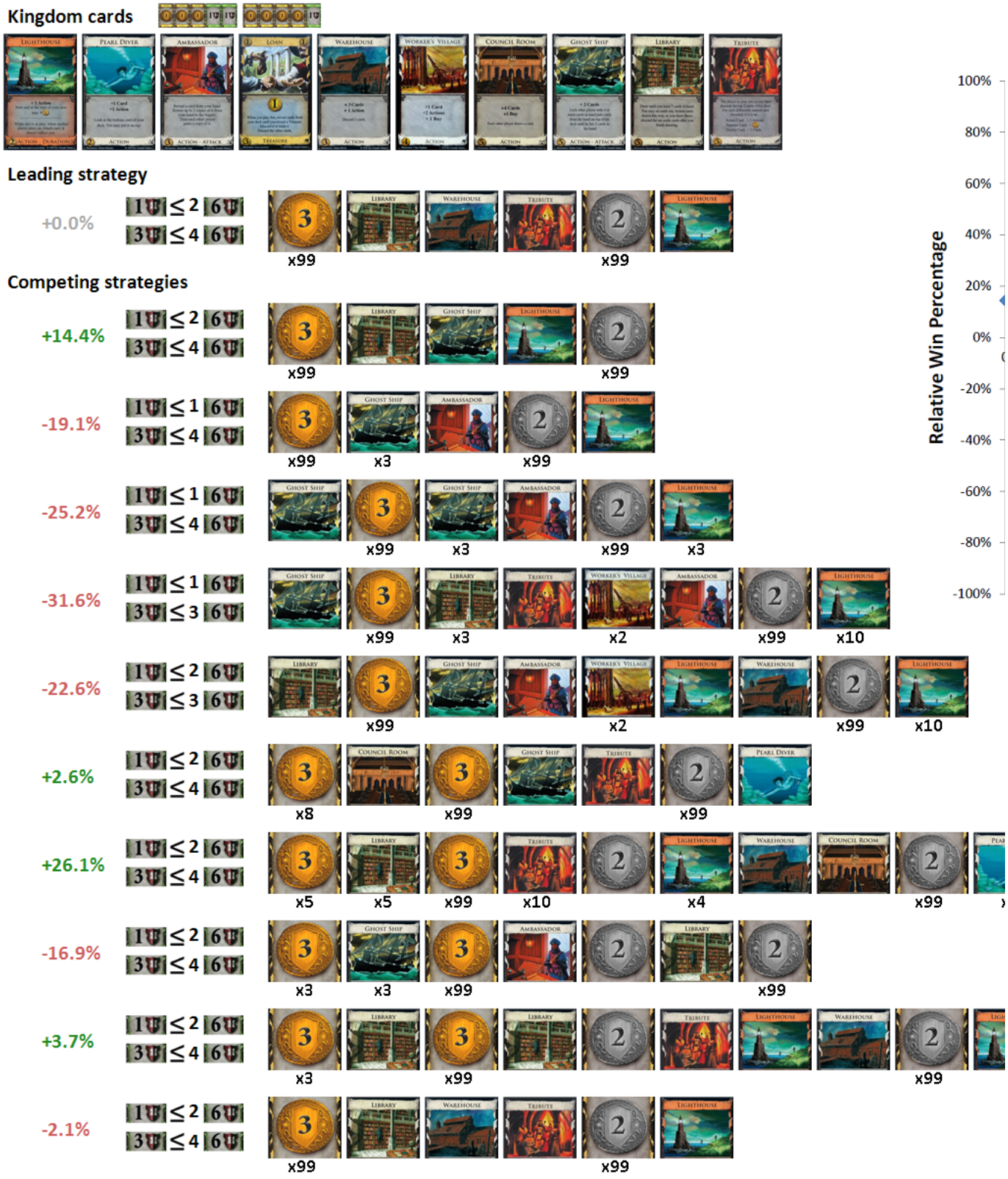
In some kingdoms, the dominant strategies will evolve in a very clear way: leaders at generation N will, averaged over thousands of games, either the leaders from generation 1 to N; we refer to such kingdoms as *simple kingdoms*. On the other hand, in *complex kingdoms*, the strategies might considerably more volatile, as some strategies move in and out of favor in response to the card choices made by competing strategies. This beha common in the presence of kingdoms with strong attack and defense cards, or cards with complex victory goals (Gardens, Duke), and is referred strategic metagaming. In *simple kingdoms*, it is typically the case that one strategy dominates by always matching or beating every other strategy, clear which strategy to choose. In the more interesting case of *complex kingdoms* where no strategy strongly dominates, an optimal player should randomly between the candidate strategies to achieve an equilibrium strategy.

We use a *progression visualization* to view the evolution of the metagame. This visualization shows how the leading strategy at generation N com against the leading strategy at generation 1 through N. Below is a progression visualization for a *simple kingdom*:

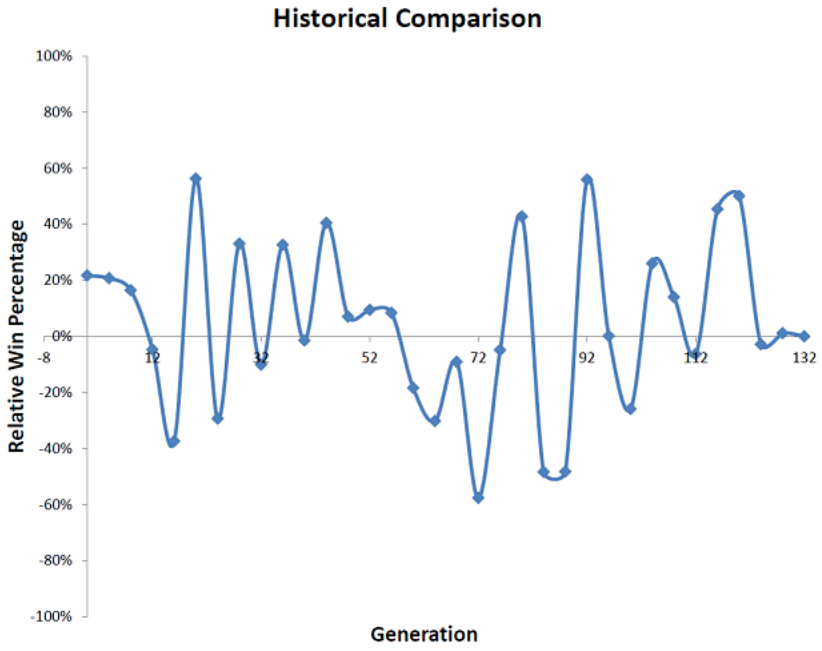


"Leading strategy" shows the dominant strategy at the current generation. "Competing strategies" shows the leading strategy from the previous N generations and the relative win ratio averaged over 10,000 games against the leading strategy (to save space, only every 4th generation is show right, we plot this win ratio as a function of generation. In this simple kingdom, the leading strategy is at least as good as all prior generations, indi slowly improving evolutionary behavior. Unless the AI happens to mutate some significantly better strategy, the future generations will approximat converge to the strategy shown, although small fluctuations will still occur because of the statistical variance inherent in comparing two strategies.

This steady improvement is not the behavior for all kingdoms. Here we show a progression visualization for a *complex kingdom*:



Here, the behavior is much more volatile than the slow and steady improvement seen in a simple kingdom. The leading strategy at generation 40 substantially better than some prior leaders, but substantially worse than others. These kingdoms are often more interesting to play because the b to choose is much less clear. Even if the AI is allowed to train for many generations, this periodic behavior continues. Here is the same plot, comp leader at generation 132 against all prior generations:



All visualizations for this kingdom

A natural question to ask is what is the ratio of complex to simple kingdoms. It turns out that complex kingdoms are definitely the exception and th majority of kingdoms are simple, converging directly to a very good strategy in a small number of iterations. Out of a thousand kingdoms, I was on find about five with non-dominating behavior. Note that just because the progression curve typically has simple behavior does not mean that only strategy is viable. It's sometimes the case that there are multiple leading strategies that use very different cards but are approximately evenly mat against each other. In this case, a player picking any one of the leading strategies is likely to do well.

---

5. Evaluating New Cards

One interesting use of Provincial is to try and quantify design decisions. Is Feast too weak? Is Wharf too powerful? Is Chancellor any good? If Wh would it still be good? Because Provincial runs games so quickly and produces competitive strategies, we can try to answer these questions. First quantify what constitutes a well-designed card.

A card is interesting if upon seeing it in a kingdom, a good player has to make non-trivial decisions regarding it. This might be because the card its decisions as part of its play, such as Steward or Apprentice. The most common decision, however, is whether to purchase a card or not, and if so A general rule of design is that a card is likely to be too powerful if an experienced player always purchases it, and likely to be too weak if an expe player never purchases it.

Provincial can be used to perform experiments that directly estimate this function. We'll look at two concrete examples, Wharf and Plunder:

We'll start with Wharf. We pick 30 random kingdoms, all of which are constrained to use Wharf. We then run Provincial on these kingdoms to estir constitutes a good strategy for each kingdom, and compute the percentage of kingdoms where Wharf was in one of the top three strategies. In thi

experiment, Wharf was chosen in 28 out of 30 kingdoms. This suggests that while Wharf is certainly powerful, very rarely the presence of other pc cards have the potential to force a skilled player to make a decision regarding whether or not to buy Wharf. We can then reprice Wharf to cost 6, a the same experiment. In this case, 18 kingdoms contain a strategy where a top player purchases wharf while 12 do not. While noticeably weaker, still something a player is likely to consider. Fortuantely, Dominion is a symmetric game so retains fairness even in the presence of very strong or cards.

We can also use Provincial to estimate parameters for cards that have not yet been printed, and can look at modifying any parameter, not just a c Consider the (hypothetical) Plunder card shown above. We perform the same experiment as with Wharf, and find that as depicted above Plunder chosen in 3 out of 30 kingdoms. After playing around with the card, I decided that the main reason for this weakness was that it provides 0 money detrimental when trying to purchase Victory cards and requires very lucky hands to help buy Treasures, since it must be accompanied by other hi treasures. To strengthen the card, I modified it to more closely mimic Quarry and provide 1 coin when played instead of 0 coins. Rerunning the ex Plunder was chosen in 10 out of 30 kingdoms. While still not nearly as strong as Wharf, it is a considerably more viable card and competitive with 5-cost cards. Plunder was found to be especially powerful when other good treasures were present, such as Venture:

6. Algorithm Details

Here we cover a few of the major components of Provincial's underlying AI training algorithm.

Evolution Process

At each iteration of the algorithm, there are two important entities involved: the general pool of strategies that are being evaluated, and a much sm leading strategies. The leaders and general pool start off as random strategies, and then each iteration proceeds as follows:

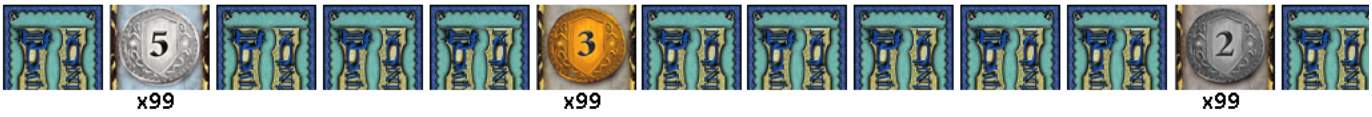
- $p_{n,i}$: the i^{th} strategy in the general pool at generation n
- $l_{n,i}$: the i^{th} leading strategy at generation n
- $r(a,b)$: the expected win ratio of strategy a fighting strategy b
- $s(p_{n,i}) = \sum_{l_n} r(p_{n,i}, l_{n,i})$: the score of $p_{n,i}$
- $l_{n+1,i}$ is chosen as the $|l|$-highest scoring strategies in p_n
- $p_{n+1,i}$ is built by choosing mutations of $l_{n+1,i}$

There are three very important parameters: the number of strategies in the general pool, the number of leaders, and the number of generations th is run for. After a bit of experimentation, I found between 5 and 10 leaders works well, and for the sake of efficiency I tend to go with 5 leaders. Fo general pool, I typically use 100 strategies, which provides a good amount of variety each generation. Increasing the size of the pool increases the takes to run a generation, but might also reduce the number of generations that need to be run. As shown above, the number of generations neec depends greatly on the kingdom. Some converge in as few as 20 generations, while some are still evolving in good ways at 100 or more generatic default number of generations used by Provincial is 32 which is sufficient for most kingdoms, but I have seen plenty of situations where significant improvements have been made after this period.

Menu Templates

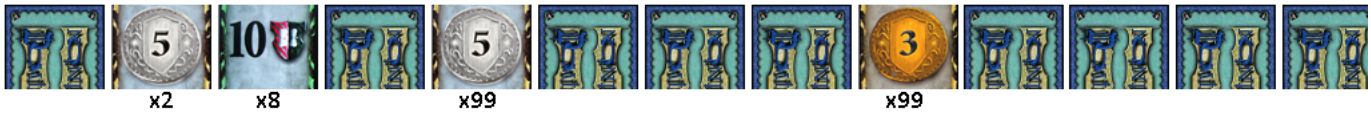
If the strategies are allowed to mutate without any form of structure, the algorithm will still work well but lots of computation will be wasted because majority of mutations are clearly not sensible strategies. For example, there are almost no strategies that would favor purchasing a Silver over a P a Smithy over a Nobles, or purchasing a Curse or Ruins under any circumstances. Likewise, some cards such as Platinum and Gold are universa and should almost always be considered as fallback purchases. To avoid wasting lots of time on strategies that are not likely to be viable, Provinci

strategies using a prior distribution over the space of possible buy menus. This *menu template* uses the Big Money strategy as a scaffolding. For l games, it looks like this:



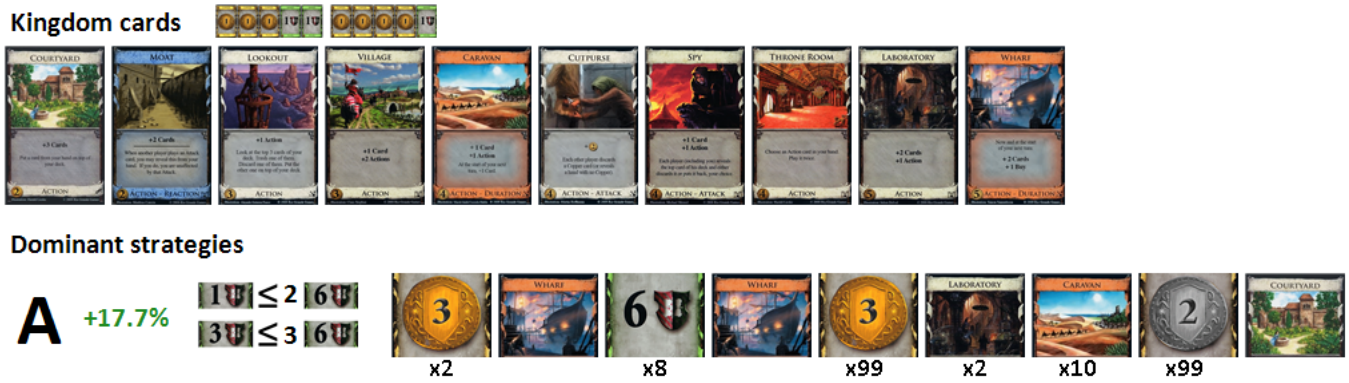
This strategy will buy Silver, Gold, and Platinum until the piles are depleted (indicated by the x99). The other cards can be any supply cards from t although each slot has a restriction over the range of prices allowed. For example, the last menu slot cannot cost more than 2, because it will alwa ignored in favor of Silver.

Victory cards that are part of the core game (Estates, Duchies, Provinces, and Colonies) are not allowed to be chosen as cards in the variable ent selection follows a game-stage-dependent purchasing model. Still, it is easy to use this menu structure to allow for more complex buy strategies. I example, we could easily allow the AI to decide when to buy Colonies (instead of greedily purchasing them) by using the following template:

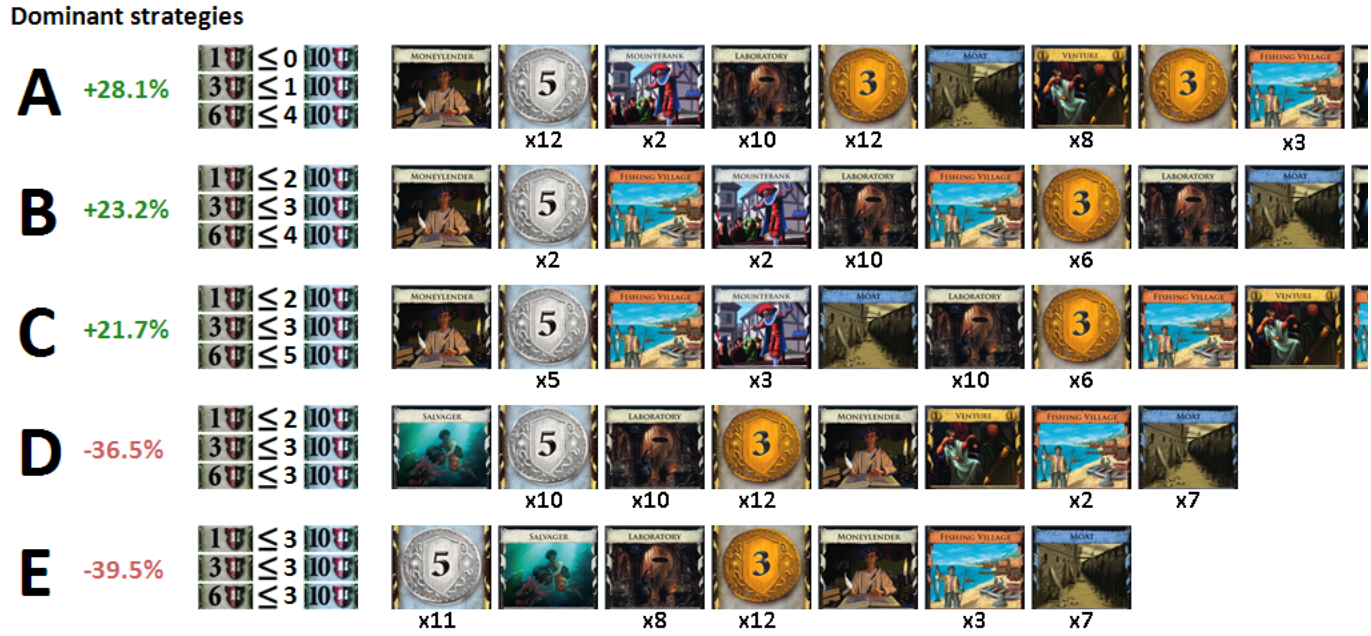


Here the AI will purchase one card and two Platinum before purchasing its first Colony, and then be greedy in Colonies ever-after. Naturally, the m process might change or remove these cards and revert to being greedy in Colonies, if that is found to be a superior buy strategy for the given kin only reason I don't default to this more expressive model is because the buy strategy visualizations already consume quite a lot of screen space. (this behavior is straightforward, see BuyAgenda.cpp.

Here an AI trained using this more flexible model chooses to buy one Wharf and two Gold before its first Province:



A natural question to ask is whether this bias towards Big Money strategies affects the final results. I ran several experiments where strategies tha unconstrained menus were compared against those based on Big Money. Although the unconstrained strategies tend to look a bit odd due to the the cards, ultimately the strategies have almost identical performance when compared against strategies that started with Big Money. Here is a lea trained using an unconstrained menu:



Mutations

At each generation, the new strategy pool is formed as mutations of the leaders of the previous generation. Here are the types of buy menu mutat considered:

- **Replace supply card —** Replace a supply card in the menu with another card randomly chosen from the supply.
- **Modify purchase count —** Change the number of cards purchased by a random amount.
- **Swap supply cards —** Swap the order of two menu entries.
- **Change victory card purchase thresholds —** Change the thresholds that control the purchasing of Estates, Duchies, and Provinces.
- **Change card-specific play parameter —** Change a "play parameter" attached to a specific kingdom card. For example, the AI might shift th controlling the threshold under which the AI will Remodel a gold into a Province (see the Play Agenda section).

To improve the algorithm's rate of convergence, each mutation occurs with a fixed probability proportional to how often mutations of that kind turn useful. If only one mutation is performed, it is also possible that important parts of the strategy space will be inaccessible because the intermediat might have very low score. To overcome this problem, a Gaussian number of mutations are performed, so there is always a chance that the low-s areas of strategy space can be skipped over. Nevertheless, there is still the possibility that interesting strategies will be skipped unless the algorith a very large number of generations. To see how pervasive this problem is, Provincial supports running multiple independent simulations in paralle referred to as the number of "test chambers" used. For simple kingdoms, typically a single chamber is sufficient. For complex kingdoms, however, interesting behavior can be achieved by using multiple test chambers. Provincial will automatically run all of the leaders of each chamber against and visualize the resulting leaderboard.

Play Agenda

The vast majority of the skill in Dominion is in determining what to purchase and when. Actually playing cards, while also important, generally has less impact on overall play. For the majority of cards, there are simply no options available: Laboratory, Mountebank, Venture, and so on are all ca "play themselves". Still, some strategies use cards that require more complex play styles. For example, while generally it is wise to reveal a Moat, a library in your hand and your opponent plays Militia, revealing your Moat is usually not the optimal play. The AI uses a simple look-ahead play m can be found in PlayerHeuristic.cpp. For many cards, this amounts to simple heuristics. For example, when playing Spy the AI will always discard opponent's card, unless it is a Curse, Copper, Ruins, or a pure victory card. I have run a few experiments with more complex models (discussed ir Work section), but at present I have not found them to significantly change the strategies that the AI converges to. For almost all kingdoms, the AI behavior has dramatically more impact than the choices made while actually playing the cards.

A few cards do admit the need for more complex card play behavior. Good examples are Salvager and Remodel, whose play properties might cha dramatically throughout the game. For example, if you are currently in the lead and want to deplete the Province pile faster, you might Salvage a and likewise if you are very late in the game you might remodel a Gold into a Province. When present in the kingdom, such cards emit per-card pa

that are added to the set of parameters available to the AI. They are then mutated in the same fashion as the Buy menu parameters, allowing the good play strategies for these types of cards.

---

7. Future Work

Provincial is still a work in progress. Here are some thoughts on the current implementation and some next steps:

- **Card implementations —** The majority of cards from Base, Prosperity, Intrigue, and Seaside have been implemented as well as several custc friends designed. While this leaves quite a lot of cards remaining, it still covers more than enough cards to encompass kingdoms with ve strategies. Implementing new cards is also fairly straightforward, as they tend to reuse common functionality from prior sets such as "discarc cards" or "put N cards on top of your deck".
- **Multiple opponents —** The game of Dominion changes dramatically when multiple opponents are present. Currently, Provincial can play aga opponents but simply uses strategies it learns from simulations against a single player. Moving forward, it would be interesting to look at buildir for multiple opponents, although the space of viable strategies is considerably larger and harder to search.
- **Dealing with novel kingdoms faster —** At present Provincial has to train a different strategy for each new kingdom. Although it can constru strategies in a matter of minutes, ideally it would do so even faster by taking advantage of kingdoms it has previously developed strategies fc its strategy database. Once a critical mass of kingdoms has been studied, it should be possible to seed the initial pool with strategies fr kingdoms" (for example, those with at least 4 overlapping supply cards). This should result in much faster convergence and demonstrate transfer from prior games similar to the process used by skilled human players.
- **Online accessibility —** Because Provincial's core buy strategy is just an ordered list of card-integer pairs, it is fairly easy to hook up tc systems. On Isotropic, Provincial is already very competitive and wins well over half its games even at a high level of play, although it does su initial latency of needing to pause for two minutes while it runs simulations. Unsurprisingly, Provincial can also (trivially) destroy the AI curre Goko's dominion implementation, and it would be fairly straightforward to have Provincial's buy strategies replace the ad-hoc buy strategy use AI.
- **Superior buy strategies —** Provincial intentionally avoids using complex machine learning techniques so that its strategies can be vis understood by human players. By removing this constraint, Provincial's AI could do noticeably better. For example, by playing many games, use a multi-class classifier to accurately guess how many more shuffles through the deck the AI is likely to receive. This is a much bette determining when the game is going to end compared to the number of Provinces left in the supply, and could result in much more elegant be deciding when it should transition into purchasing Provinces, Duchies and Estates.
- **More advanced play strategies —** Although Provincial uses a sophisticated buy strategy, its algorithm for playing cards is comparatively kingdoms where this is important, a classic AI approach can be adopted. In this model, we first define a state value function that indicates hc like or do not like a game state. The AI then uses Minimax or Monte-Carlo tree search to consider taking many different branches, and choose that gives the highest expected score. While not difficult to implement, Provincial does not use this approach because it would dramatica training time. However, when playing against a human after its buy strategy has been developed, this method is very attractive because th factor for Dominion is very low for the majority of kingdoms. This will allow the AI to capture many tactics used by skilled players such as The Province Rule and many other strategies often discussed on dominionstrategy.com.

Overall Provincial is quite a powerful AI that develops strategies fairly similar to those used by very experienced players. That said, it is important remember that the AI can still make mistakes when playing its cards, and Dominion is a high variance game — hundreds of games are needed to evaluate a player's skill with a high degree of confidence.

---

8. Code

The user interface for this project is all written in C#, while the backend is written in C++. Since the C++ backend uses no external libraries (it is ju Dominion game engine), it is actually not too difficult to get the application running under Mono when running on Linux or Mac operating systems. also has a GitHub repository and in the future I plan to add makefiles for UNIX-based systems.

The code is structured using a straightforward object-oriented event stack model. The card interface is invoked whenever a card is played or anot specific event triggers. If a card has an effect that might require a response from the user or other game entities (such as attacks that might trigge it pushes an Event object onto the stack. When the event stack is empty, the game advances by using the basic rules of the current phase of the g When the event stack is not empty, the game advances by processing the event at the top of the stack.

Provincial.zip (just the executable and data files)

ProvincialCode.zip (executable + data + code + project files)

README.txt

SampleLeaderboardVisualization.png

DominionAI Code Listing

DominionApp

- CardDatabase.cs, Web Version
- ConfigWindow.cs, Web Version
- ConfigWindow.Designer.cs, Web Version
- Constants.cs, Web Version
- DebugWindow.cs, Web Version
- DebugWindow.Designer.cs, Web Version
- DLLInterface.cs, Web Version
- DominionHelper.cs, Web Version
- DominionVisualization.cs, Web Version
- MainWindow.cs, Web Version
- MainWindow.Designer.cs, Web Version
- Program.cs, Web Version

DominionDLL

- AIUtility.cpp, Web Version
- AIUtility.h, Web Version
- App.cpp, Web Version
- App.h, Web Version
- AppParameters.cpp, Web Version
- AppParameters.h, Web Version
- BaseCodeDLL.cpp, Web Version
- BaseCodeDLL.h, Web Version
- BuyAgenda.cpp, Web Version
- BuyAgenda.h, Web Version
- CardDatabase.cpp, Web Version
- CardDatabase.h, Web Version
- CardEffect.h, Web Version
- CardEffectAlchemy.h, Web Version
- CardEffectBase.h, Web Version
- CardEffectCustom.h, Web Version
- CardEffectIntrigue.h, Web Version
- CardEffectProsperity.h, Web Version
- CardEffectSeaside.h, Web Version
- Config.h, Web Version
- Constants.h, Web Version
- DLLMain.cpp, Web Version
- DominionGame.cpp, Web Version
- DominionGame.h, Web Version
- Engine.cpp, Web Version
- Engine.h, Web Version
- Enums.h, Web Version
- Event.cpp, Web Version
- Event.h, Web Version
- EventAlchemy.h, Web Version
- EventBase.h, Web Version
- EventCustom.h, Web Version
- EventIntrigue.h, Web Version
- EventProsperity.h, Web Version
- EventSeaside.h, Web Version
- GameData.cpp, Web Version
- GameData.h, Web Version
- Main.cpp, Web Version
- Main.h, Web Version
- MetaTestChamber.cpp, Web Version
- MetaTestChamber.h, Web Version
- Player.h, Web Version
- PlayerHeuristic.cpp, Web Version
- PlayerRandom.cpp, Web Version
- State.cpp, Web Version
- State.h, Web Version
- TestChamber.cpp, Web Version
- TestChamber.h, Web Version

TestingGrounds

- MainWindow.cs, Web Version

MainWindow.Designer.cs, Web Version
Program.cs, Web Version
Total lines of code: **13730**

NodeTh