Distributed Processing of a Round-Robin Dominion AI Simulation

Jaineel Upadhyay, Warren Cobb, and Robert Roibu

University of Maryland Baltimore County

CMSC 483

05/08/2021

Abstract - Obtaining meaningfully accurate results from Dominion AI simulations is difficult because of the variance from random factors(such as shuffling the deck), the variance in starting game states(random kingdom each game), and the sheer number of computations AI models perform in a given game. For this reason, a large number of games must be played in each matchup to account for the aforementioned variance. In order to accommodate the large number of computations that must be performed in this many games, distributed processing is a necessary component to an efficient solution. Distributed processing also opens the door to more complex AI agents, such as Monte Carlo and other computationally-intensive models. Utilizing a distributed processing approach, it is possible to create a Dominion AI round-robin simulation that provides meaningful strategical results while staying within reasonable time and resource constraints. For the purposes of this project, there was a focus on a few common gameplay strategies(rush, big money, engine, etc.) and a few different AI agents.

Keywords-Distributed processing, parallel processing, Dominion, artificial intelligence, multithreading, game theory, strategy, Monte Carlo

I. INTRODUCTION

A. Background

Dominion is a deck-building card game deep with intense strategy. In Dominion, players compete to earn the most victory points before the game ends. There are 3 main card types that players will gain to their decks: action cards, treasure cards, and victory cards. Each player starts the game with 3 Estates(victory cards) and 7 Coppers(treasure cards). Each turn, players complete a play phase(in which action and treasure cards are played) and a buy phase(in which cards are bought and added to the deck for use in future turns). Play continues until either all Provinces(most expensive victory card) are bought or 3 supply piles are emptied.

Dominion differs from most games that AI is popularly utilized in, because the starting game state is different in every game. While a game like chess or checkers always starts with the same board state, Dominion may have completely different kingdoms from one game to the next.

Buyable cards in any game of Dominion are part of the "supply," which contains cards from 2 different classifications: kingdom cards and basic supply cards. Basic supply cards(Province, Duchy, Estate, Gold, Silver, Copper, Curse) are present in every game. However, the kingdom(a collection of 10 sets of kingdom cards) is randomly chosen before each game. As there are 25 possible kingdom cards, this means that the number of possible kingdoms(and thus, starting game states) is: $\frac{25!}{15!} = 11,861,676,000,000$. In addition, different types of gameplay strategies are stronger or weaker depending on the starting game state.

B. Basic Strategy

Although there are a plethora of complex strategies possible to employ in any game of Dominion, most deck-building strategies fall into one of a few basic categories: engine decks focus on actions cards that grant cards and further actions, big money decks focus on buying treasure cards, rush decks buy cheap cards in order to end the game early, and attack decks focus on buying attack cards to hinder opposing players. Furthermore, most successful gameplay strategies combine elements of multiple strategies.

II. MOTIVATION

The motivation for this project is three-fold. First, Dominion is an incredibly strategy-deep game, and analyzing thousands of games with physical cards is not feasible because of the time it would take. In order to obtain results to examine, simulated games are an efficient solution that take only a fraction of the time. Second, Dominion is a fantastic problem for AI models, because of the depth of strategy and the relatively unsolved nature of Dominion. Finally, AI models analyzing thousands of simulations of a game with a massive gamestate space like Dominion can cost a hefty amount of time and computational resources. Hence, Dominion AI round-robin simulations are a prime candidate for parallel processing because of the huge number of computations and games required.

III. METHODS

A. Object-Oriented Approach

After analysis of the nature of the project, it was decided to follow an object-oriented approach to the project as it allowed for great modularity and efficiency.

B. Round-Robin Simulation

In order to examine the success of various strategies and AI techniques against each other, a round-robin tournament is necessary, where each player plays each opponent a set number of times. Through experimentation, matchups typically converged to an accurate win rate between 700 and 1300 games. Thus 1000 games was chosen as the standard for each matchup. In addition to win percentage, average run time per matchup is important in order to evaluate player performance by both success and efficiency.

C. AI Players

In its simplest form a player is an entity who makes decisions based on various factors. This simulation narrowed down player decision-making to 14 primary functions. However, ultimately the function that separates the strategy of different players is choosing which cards to buy in the buy phase. This is the most important distinction between players, so it will garner the most focus.

1) Basic Bot

Basic Bot is a relatively simple player that frequently makes decisions randomly, rather than based on any particular strategy. Consequently, this player is the one with the most variance in the project. During the buy phase, basic bot compiles a list of the most expensive cards in the supply that it can buy and randomly chooses one. This player is a building block for many of the other players, either for training data or for other functionality.

2) Attack Bot

Attack Bot is a player that heavily prioritizes attack cards and focuses on disrupting other players through clogging opponents’ decks with Curse cards through Witch, forcing them to discard from their hand through Militia, etc.

3) Money Bot

The Money Bot player focuses almost exclusively on treasure and victory cards. For this reason, Money Bot is one of the most consistent players, because its strategy is rarely altered by what cards are chosen for the kingdom. Money Bot’s priority for cards is calculated based on the cost and attributes of each card. In general, the priority is as follows: Province, action cards with most improvements per each coin it costs to buy, Gold, Silver, less valuable action cards, and so on.

4) Rush Bot

Rush Bot is a player that attempts to buy as many cheap cards as it can as early as possible in order to end the game(on 3 empty supply piles) before its opponent can fully bring their deck online. Rush bot is arguably the most dependent on the cards chosen for the kingdom, as it struggles in games with expensive kingdoms and kingdoms without Workshop or Gardens.

5) Gini Bot

Gini Bot is a player that utilizes training data to analyze which cards it deems the most important. Named after the gini index from decision trees, this player uses gini index values in its calculations for card priority. To begin with, a specified number of simulated games are run in the trainer. Then, the cards in the decks of the winning and losing players are examined. First, the gini index with regards to the percentage of games a card was in the kingdom and appeared in the winner or loser decks is calculated. However, this could be highly inaccurate, as this gini index is only a measure of how stark a split there is between a card appearing in winning and losing decks, not necessarily the success of a card. Therefore, if the card appears in a higher percentage of loser decks than winner decks and has a card priority greater than 0.5, the value is set to (1 - value) to essentially invert its priority. That is:

$$Card\ Priority\ =\ 1.0\ -\ (W^2 + L^2);$$

$$if\ L\ >\ W\ AND\ CardPriority\ >\ 0.5:$$

$$CardPriority\ =\ 1.0\ -\ CardPriority$$

where *W* equals the number of games a card appears in the winner's deck divided by the number of games the card appears in the kingdom and *L* equals the number of games a card appears in the loser's deck divided by the number of games the card appears in the kingdom.

Furthermore, Gini Bot breaks down the game into three phases: early game, mid game, and late game. A card priority "modified gini index" list is calculated from the training data for each of these 3 game phases.

During its buy phase, the Gini Bot player buys the highest priority card for the number of coins available; if there are no cards for that exact number of coins, it recurses with (coins -1) for the cost (base case: coins < 1).

6) Monte Carlo Bot

Monte Carlo Bot is a player that relies on finding the best possible card through performing a large number of simulations. It performs a specified number of game simulations with the purchase of the card to determine the average win rate and score possible when that card is bought at that turn in the game. This method should converge to the best possible card affordable as it performs the computations. On average, the Dominion game was found to have ~40 turns in each game. Hence for each turn with each buy action in mind (including the option to not buy a card),

$$Num.\ game\ simulations\ /\ buy\ action\ =\ 30\ Simulations$$

$$Num.\ Monte\ Carlo\ AI\ turns\ =\ Num.\ turns\ of\ game$$

$$Num.\ simulations\ in\ each\ Monte\ Carlo\ AI\ game:$$

$$\Rightarrow Num.\ Monte\ Carlo\ AI\ turns\ *\ Num.\ Actions\ *\ Num.\ simulations$$

$$=\ 40\ *\ (32\ Cards\ +\ 1\ NA)\ *\ 30\ =\ 39,600\ Simulations$$

$$Num.\ Monte\ -\ Carlo\ Simulations\ in\ each\ matchup:$$

$$\Rightarrow Num.\ simulations\ in\ each\ Monte\ Carlo\ AI\ game\ *\ Num.\ games$$

$$=\ 39,600\ *\ 1,000\ =\ 39,600,000\ Simulations$$

The number of simulations to perform if no reductions are done is 39,600,000 for each matchup, which can not be performed due to the sheer number of simulations executed on a simple processor. While such simulations would provide us with potentially good buy actions based on the randomness of each simulation, based on the Law of Large Numbers, there needs to be optimization for the bot's buying strategy.

The Law of Large Numbers is a principle the Monte Carlo Bot is based on. The idea is that through running a number of sample simulations, independent of each other, the bot can find the expected win rate and average score by buying the card at that game state, which is provided by the sample of the simulation run. The sample score and win rate converge to their expected values. As such, more simulations are better but one has to keep in mind the computational resources and time it takes to run many simulations.

To reduce the number of simulations, the cards purchased are limited mainly to two types of cards - Victory and Treasure cards, with the option of not buying a card and skipping a turn, inspired from an analysis of Dominion which states that for every strategy, the best action in each state is to buy the most expensive treasure or victory card, or not buy a card, in an efficient game [1]. With this in mind, the number of turns in which Monte-Carlo takes place is reduced to a gap of four. Every fourth turn would be when the Monte-Carlo AI algorithm is used, otherwise the Basic Bot is used for the other turns. This provides the opportunity to build a deck with some variety, along with the reduction in simulations. Each action is a Node object that stores the card and the information required and updated for the Monte-Carlo Bot.

Hence, after increasing the gap between Monte Carlo and reducing the action space to three buy actions, the total number of simulations per match up is :

Total Number of Simulations for each Monte Carlo Bot turn :

$3 \; * \; 30 \; = \; 90 \, simulations$

Total Number of Simulations per game with Monte Carlo Bot:

$3 \; * \; 30 \; * \; (number\ of\ turns\ /\ 4)$

$= \; 3 \; * \; 30 \; * \; (40\ /\ 4)$

$= \; 3 \; * \; 30 \; * \; 10 \; = \; 900 \, simulations$

$Total\ Number\ of\ Simulations\ per\ each\ matchup:$

$1000 \; * \; 900 \; = \; 900,000 \, simulations$

Each game simulation involves the card to be bought and a custom version of Basic Bot with the buy action taken played against the Money Bot. The reasoning behind Money Bot is if the Basic Bot's performance improves with an action taken against the currently best AI in terms of win rate and average score, then it is likely to perform better against other AI bots. The best card to buy is chosen based on which action results in maximum value for the following formula:

$Best\ Card \; = \; max(wins/num.\ simulations \; * \; avgScore/num.\ simulations)$

IV. RESULTS

A. Overall Results

The round-robin simulation includes 6 players: Attack Bot, Basic Bot, Gini Bot, Money Bot, Monte Carlo Bot, and Rush Bot. For each player in the round-robin simulation, several important data features are calculated. Win rate versus each opponent, average deck composition, and runtime are calculated for each player.

B. Individual Player Results

For the following data, the round-robin simulation was run multiple times and mean values were taken of each result.

For all win rate radar charts, the red benchmark guideline represents a 50% win rate, and each white benchmark represents a 10% step.

As shown by the number line, for all average deck analysis radar charts, the red benchmark guideline represents 1.0 card in the average deck, while each white benchmark represents a 1.0 card step. Deck analysis visualizations can be a useful tool in examining player tendencies [2].

1) Basic Bot

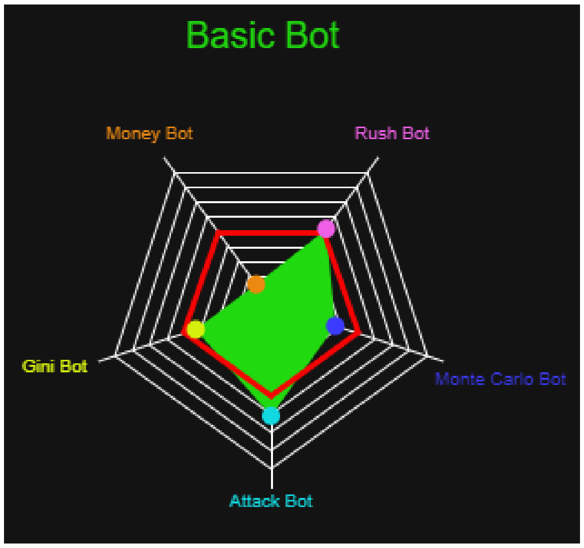

Fig. 4.1.1. Basic Bot win rate vs. each opponent over 1000 games in each matchup of the round-robin simulation. (a) Red benchmark represents 50% win rate. (b) Colored vertex represents a matchup with a player of corresponding color.

The Basic Bot player performed well against Rush Bot and Attack Bot players, but struggled against Gini Bot and Monte Carlo Bot players, and heavily struggled against Money Bot.

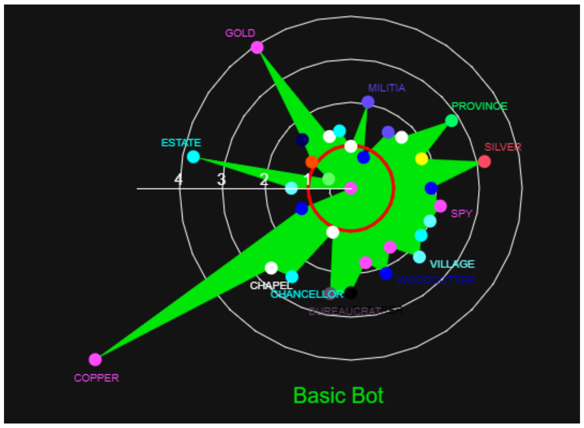

Fig. 4.1.2. Basic Bot average deck composition. (a) Red benchmark represents 1.0 card per deck. (b) Each colored vertex represents a specific card. (c) Only cards averaging more than 2 occurrences per deck are named in the diagram.

As previously mentioned, Basic Bot has the most variance in gameplay because it has the most varied deck. Part of the reason why it struggles so mightily against the Money Bot is because BasicBot’s random choices often force the deck to take too long to develop, whereas the MoneyBot is consistent in its deck progression.

2) Attack Bot

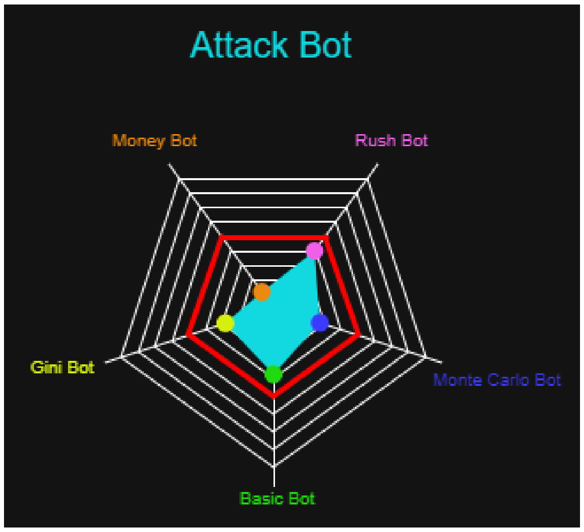

Fig. 4.2.1. Attack Bot win rate vs. each opponent over 1000 games in each matchup of the round-robin simulation. (a) Red benchmark represents 50% win rate. (b) Colored vertex represents a matchup with a player of corresponding color.

Attack Bot struggled slightly against Rush Bot, slightly more against Basic Bot, and struggled heavily against Gini Bot, Money Bot, and Monte Carlo Bot. All in all, it is pretty clear that prioritizing attack cards as the primary focus is not a viable strategy. All of the other players consistently beat Attack Bot, and often quite handily. Part of the problem with Attack Bot’s strategy is that having too many attack cards can result in cards that cannot be played due to limited actions. When Attack Bot targets Witch, but no action adders, it is bound to repeatedly run into “drawing dead” scenarios, where action cards are drawn with 0 actions remaining. In these situations, the action cards are worse than meaningless; they take up valuable hand space that could have been treasure. There are two directions that Attack Bot could be taken to improve performance. The first option is to limit the attack cards and focus more heavily on utility and treasure cards for the deck. This would allow for a still somewhat attack-heavy deck, but wouldn't bog the deck down with unusable and pointless attack cards. The other option is to continue the focus on attack cards, but supplement it with action adders, card adders, and cantrip cards so that a solid engine is developed for the attack deck. Either of these changes would likely improve the Attack Bot's success significantly.

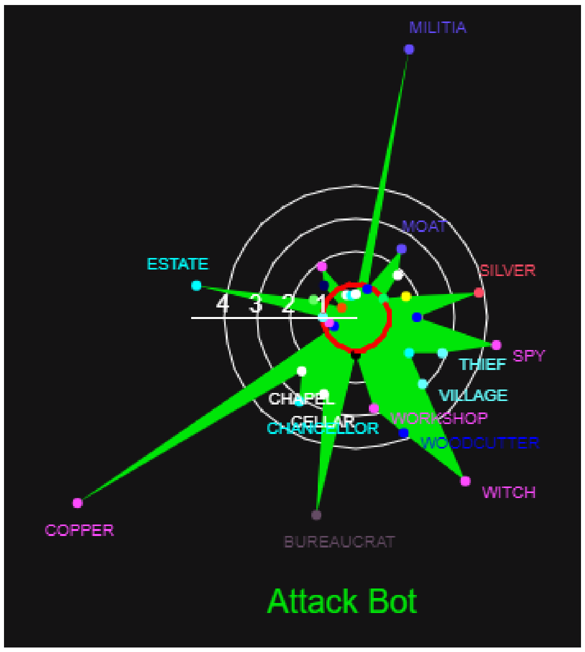

Fig. 4.2.2. Attack Bot average deck composition. (a) Red benchmark represents 1.0 card per deck. (b) Each colored vertex represents a specific card. (c) Only cards averaging more than 2 occurrences per deck are named in the diagram.

As previously mentioned, Attack Bot heavily prioritizes attack cards(Bureaucrat, Militia, Thief, Spy, Witch).

3) Money Bot

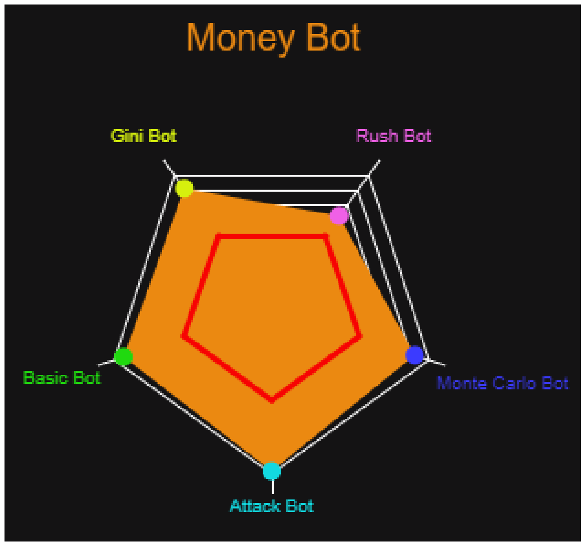

Fig. 4.3.1. Money Bot win rate vs. each opponent over 1000 games in each matchup of the round-robin simulation. (a) Red benchmark represents 50% win rate. (b) Colored vertex represents a matchup with a player of corresponding color.

Money Bot was extremely successful against Attack Bot, Basic Bot, Gini Bot, and Monte Carlo Bot. It was slightly above even against Rush Bot. The lower win rate against Rush Bot is very surprising considering how well it performs against players that perform well against Rush Bot.

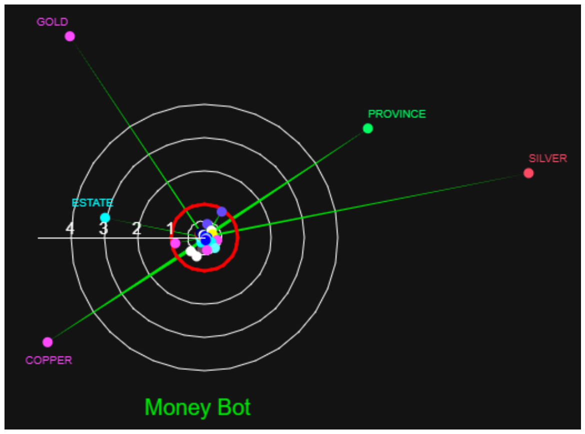

Fig. 4.3.2. Money Bot average deck composition. (a) Red benchmark represents 1.0 card per deck. (b) Each colored vertex represents a specific card. (c) Only cards averaging more than 2 occurrences per deck are named in the diagram.

Money Bot heavily prioritizes treasure and victory cards(Gold, Silver, Copper, Province, Estate, Duchy).

4) Rush Bot

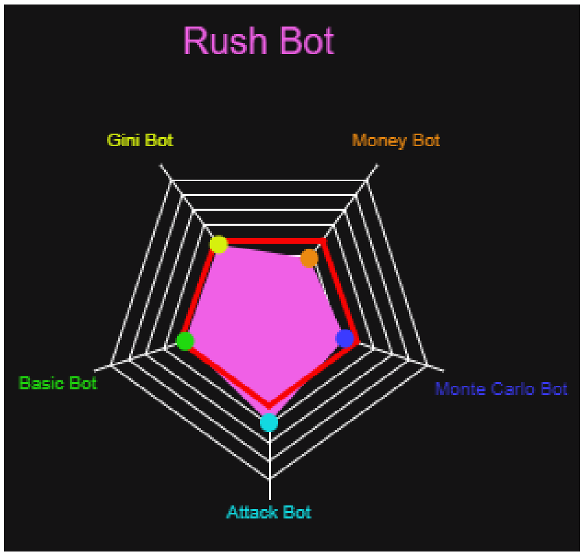

Fig. 4.4.1. Rush Bot win rate vs. each opponent over 1000 games in each matchup of the round-robin simulation. (a) Red benchmark represents 50% win rate. (b) Colored vertex represents a matchup with a player of corresponding color.

Rush Bot was close to evenly matched against all players. Surprisingly, even though Rush Bot was fairly dependent on the Kingdom each game, it was the most consistent player, maintaining close to 50% win rates against each other player.

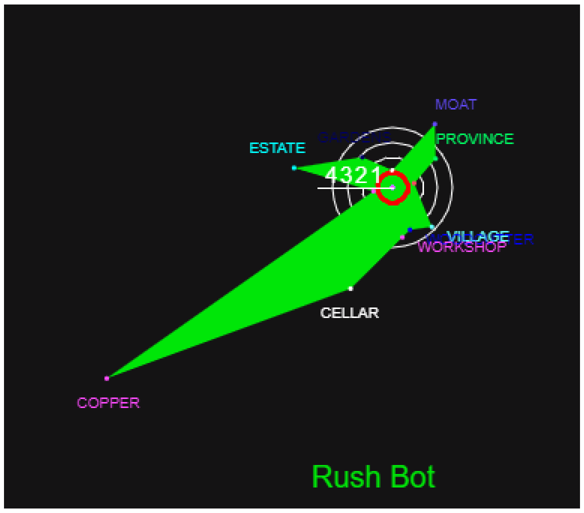

Fig. 4.4.2. Rush Bot average deck composition. (a) Red benchmark represents 1.0 card per deck. (b) Each colored vertex represents a specific card. (c) Only cards averaging more than 2 occurrences per deck are named in the diagram.

Rush Bot heavily prioritizes Copper, low cost cards(such as Estate, Cellar, Moat, Chapel, etc.), Workshop, and Gardens.

5) Gini Bot

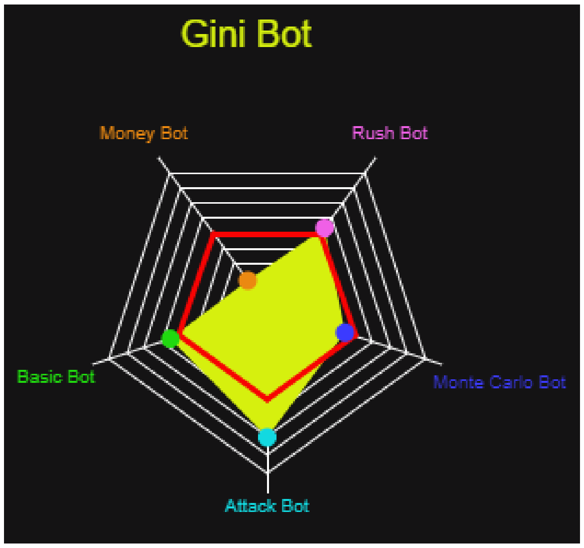

Fig. 4.5.1. Gini Bot win rate vs. each opponent over 1000 games in each matchup of the round-robin simulation. (a) Red benchmark represents 50% win rate. (b) Colored vertex represents a matchup with a player of corresponding color.

Gini Bot was fairly successful against Attack Bot, Basic Bot, and Rush Bot and close to even against Monte Carlo Bot, but performed very poorly against Money Bot.

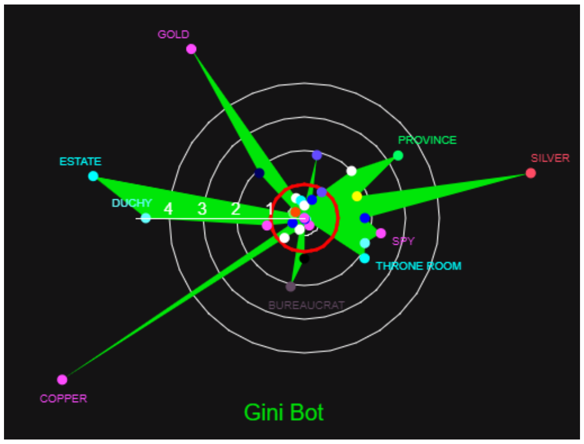

Fig. 4.5.2. Gini Bot average deck composition. (a) Red benchmark represents 1.0 card per deck. (b) Each colored vertex represents a specific card. (c) Only cards averaging more than 2 occurrences per deck are named in the diagram.

Gini Bot's deck is heavily weighted with whatever cards its training algorithm prioritizes, but typically prioritizes expensive game altering cards, such as Province, Witch, Throne Room, etc.

6) Monte Carlo Bot

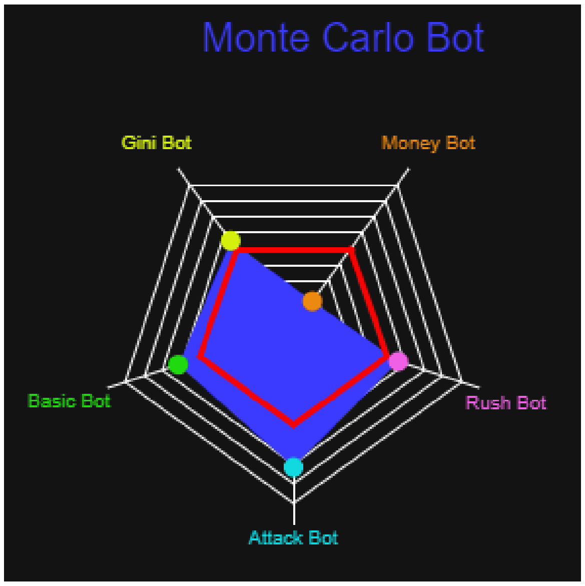

Fig. 4.6.1. Monte Carlo Bot win rate vs. each opponent over 1000 games in each matchup of the round-robin simulation. (a) Red benchmark represents 50% win rate. (b) Colored vertex represents a matchup with a player of corresponding color.

Monte Carlo Bot was very successful against Attack Bot, fairly successful against Basic Bot, Gini Bot, and Rush Bot, while really struggling against Money Bot.

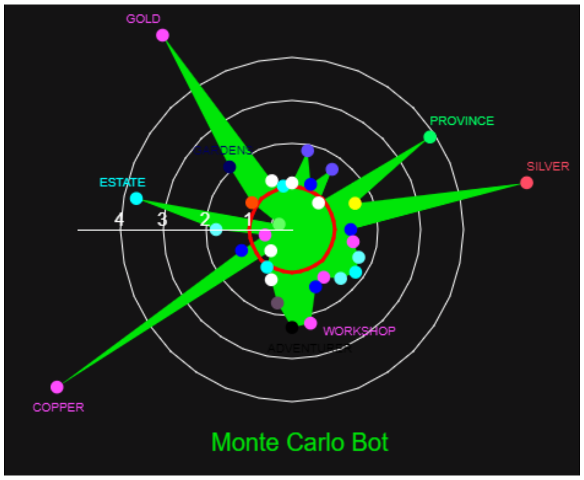

Fig. 4.5.2. Monte Carlo Bot average deck composition. (a) Red benchmark represents 1.0 card per deck. (b) Each colored vertex represents a specific card. (c) Only cards averaging more than 2 occurrences per deck are named in the diagram.

Monte Carlo Bot has a large amount of variance in cards bought once or twice per game, but for cards that appear more than twice per deck, it closely resembles the Money Bot because of its priority on treasure and victory cards.

V. DISCUSSION

A. Strategy Implications

Although Money Bot is the most successful of all simulated players, it surprisingly struggles (comparatively) with Rush Bot. This leads to an interesting situation where Gini Bot, Monte Carlo Bot, and even Basic Bot perform well against Rush Bot, but struggle against Money Bot, who in turn struggles against Rush Bot. This highlights the situational strategy inherent in Dominion. Presumably Rush Bot ending games early is not much of a hindrance to Basic Bot and Gini Bot, but is a significant hindrance to Money Bot's big money strategy. Translating this to Dominion gameplay implications, this suggests that when facing players who prioritize action cards and engine decks, big money is a very successful strategy, but when facing a player utilizing some form of rush strategy, big money is not nearly as successful. On the other hand, if the opponent is playing a big money deck, a rush strategy may be the best strategy to counter them.

B. Serial Versus Parallel

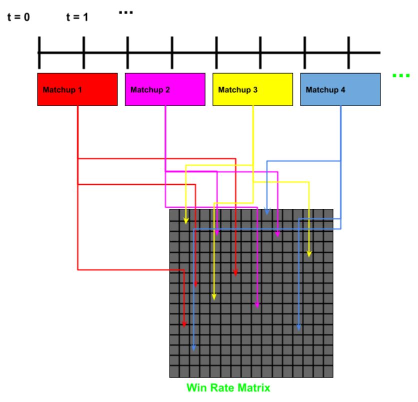

Fig. 5.1.1. Serial design of the round-robin simulation. Matchups are run sequentially, and after each game in a matchup, the result is saved to the win rate matrix.

The time complexity of multiple head-to-head matchups of AI agents playing out a thousand games is relatively high, thus before any statistical analysis on strategies could be attempted the driver simulating the games first needed to be parallelized.

1) Parallelization

Each matchup of agents is assigned to a thread; multiple threads, and thus, matchups, can then run simultaneously. Each thread handles its own matchup of 1000 games. Results from individual games in a matchup are stored locally in the thread. Once the thread finishes running all 1000 games for the matchup, the resulting statistics are saved to the global win rate matrix. Doing this incurs the risk of race conditions, which is addressed by adding synchronization through a mutex lock in the method that updates the result matrix.

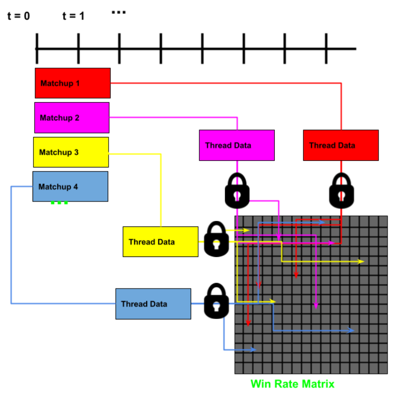

Fig. 5.1.2. Parallel design of the round-robin simulation. Matchups are run simultaneously, and after each game in a matchup, the results are saved locally in the thread. Once all 1000 games in the thread are complete, the data saved locally in the thread is saved to the win rate matrix. In order to prevent simultaneous writes to the matrix, the matrix has a mutex lock that only allows 1 thread at a time to write to it.

Once parallelization of the game simulation driver was done, parallelization of the different agents followed. As each agent focused on different aspects of AI creation, i.e decision trees, simple heuristic approaches, et al, the impact of parallelizing the agents varied.

While the average runtime for each agent worsened slightly, the overall runtime to execute all the matchups decreased. As can be seen in the graphs below.

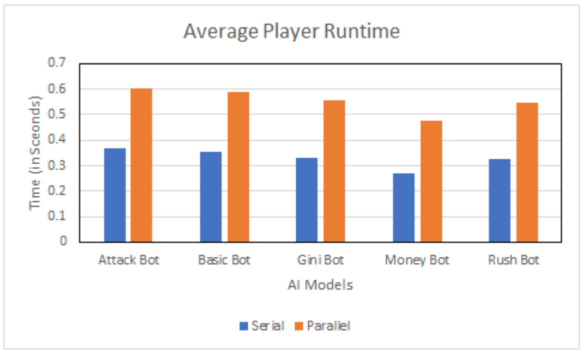

Fig. B.1.1. Average Run Times for each Player. (a) Orange bars indicate the average parallel runtimes for each AI bot. (b) Blue bars indicate the average serial runtime of all

matchups involving a bot. Average from 70 simulations of combined matchups, serial and parallel both.

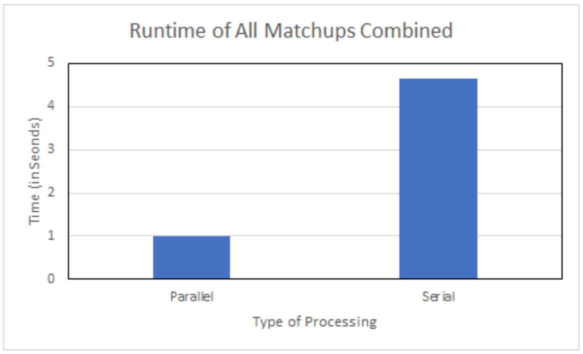

Fig. B.1.2. Overall Run Time of executing all matchups for all AI models. As can be seen, the multi-threading and distributed processing reduces the runtime as serial (sequential) processing waits for one matchup to complete before another can begin. Average from 70 simulations of combined matchups, serial and parallel both.

As Figures B.1.1 and B.1.2 show, when simulating individual matchups, it is faster to run these games serially, due to the lack of a need for synchronization and thread overhead. However, when looking at the overall time it takes to complete all matchups, parallel processing drastically reduces the runtimes by an average of ~80%, from serial runtime of 4.65 seconds to 0.98 seconds in parallel.

Aside from reduction in game time, parallelization allowed for the playing of more cards per second which allowed for more complex games. The following figure (Fig B.1.3) shows this:

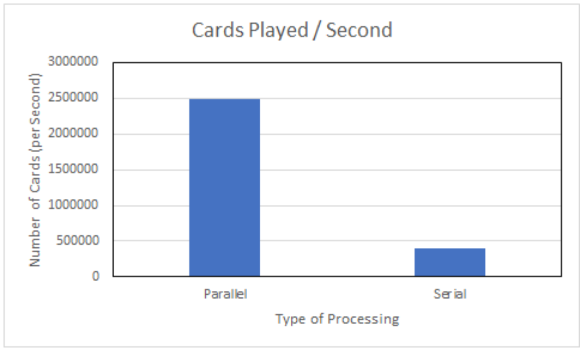

Fig. B.1.3. Number of cards played per second. (a) Top bar shows serial simulation. (b) Bottom bar shows parallel simulation. Average from 70 simulations of combined matchups, serial and parallel both.

The improvement in cards per second can be explained due to the reduction in the overall runtime due to parallelizing the matchups, so that there can be a higher execution rate of matchups simultaneously. Instead of sequential execution, where each game needs to be completed before the next game can start, parallel execution allows multiple games to be running simultaneously, so many more cards can be played per second.

2) Monte Carlo Bot

Sequential Monte Carlo performs each block of 10 game simulations of a potential action taken before moving onto the next action's simulations.

Gardens	Silver	NA (Nothing)

Fig. 3.6.1. A sequential processing of Monte Carlo simulations. If Gardens, Silver, and NA (Nothing to Buy) are the potential actions one can take and Gardens is the first action to simulate, then 3 blocks of Gardens would run 10 simulations each (sequentially it can be grouped as 30 simulations), before moving onto Silver and NA.

A parallel version of Monte Carlo would create 3 threads for each action, which would run 10 simulations on their own, while synchronized with ReadWrite locks to ensure no race conditions and proper writing of data. These specifications were decided after considerable experimentations with multi-threading. After the threads complete simulations, the best action (buying the best card) is taken based on the best card affordable.

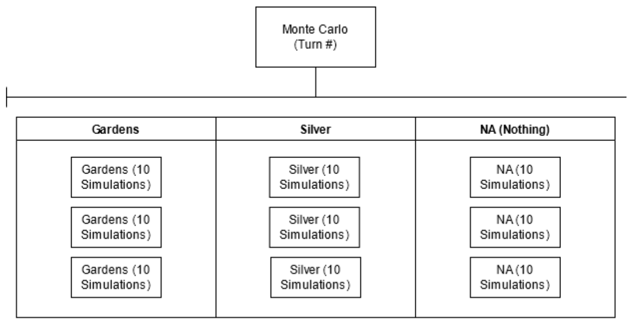

Fig. 3.6.1. A parallel processing of Monte Carlo simulations. If Gardens, Silver, and NA (Nothing to Buy) are the potential actions one can take then each thread runs 10 simulations concurrently, while keeping in mind the need to synchronize.

Monte Carlo Bot is parallelized with a ReadWrite lock which allows for future possibilities of each thread running a game as it can allow multiple threads to read through the data only when no one is writing to it. Here, a read write lock so that no other thread can read/write to the information being updated.

When the runtimes of the matchups of serial and parallelized Monte Carlo Bot with the benchmark AI Basic Bot are measured, there is a reduction of 85.17% in the runtimes when Monte Carlo is executed serially and when a parallelized version of Monte Carlo is used, as shown in Figure B.2.1. Monte Carlo matchup runtimes with other AIs have a deviation relatively smaller compared to the average Basic Bot matchup runtime, hence Monte Carlo bot is not compared with other AIs like Figure B1.1.1. As seen with the runtimes of Monte Carlo vs. Basic Bot below, the sequential version of Monte Carlo takes ~290 seconds to complete 1,000 games on average, whereas a parallel version of the AI player completes in ~43 seconds.

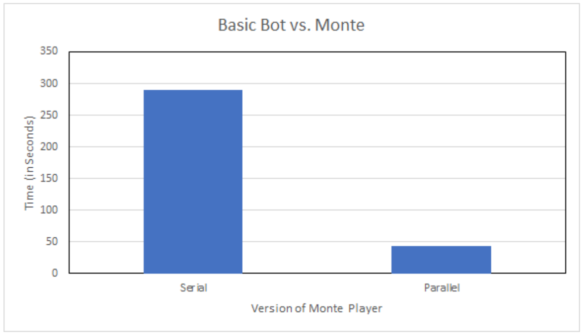

Fig. B.1.1. Average Run Times for Monte Player against Basic Bot in a single matchup.. (a) On the left is the matchup of Basic Bot with the sequential version of Monte Player. (b) On the right is the matchup of Basic Bot with the parallel version. Average taken from 20 simulations. Note : This figure talks about the processing types of Monte Carlo, not the execution of matchups which happened sequentially for each game.

Another reason why Monte Carlo's runtime is not compared with other AI matchups above is the fact that on average, a serial matchup execution of all the AIs, including Monte Carlo Bot[1], takes ~205 seconds to perform, whereas the parallel execution of all matchups including Monte Carlo Bot takes on average, ~145 seconds to complete. This is far from the overall runtimes for all matchups not including Monte Carlo Bot, as seen in Figure B.1.2., with a 3580% increase in runtimes for serial matchups, along with a 14,848% increase in parallel runtime. There is still a reduction in runtime with Monte Carlo included as can be seen below.

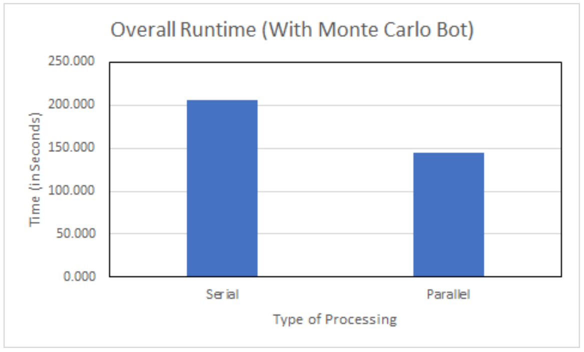

Fig. B.1.1. Average overall runtime of all matchups executed, sequentially vs. in parallel. (a) On the

[1] Monte Carlo, when mentioned, refers to the parallelized version, unless explicitly stated.

left is the runtime for sequential execution for all matchups. (b) On the right is the average runtime of performing all matchups in parallel. Average taken from 20 simulations with each type of processing.

Running individual games in each thread is a possibility but comes with multiple issues. First, the thread overhead due to 1,000 threads creation for 1 lightweight task (1 game) would lead to time wastage. Second, even if Monte Carlo Bot is considered a heavy weight task, synchronization to avoid race conditions can lead to deadlocks as multiple threads can wait for one thread to write data. Using a lock provides read access for multiple threads, but deadlocks can occur due to the synchronization. Similarly, Thread Pooling causes the same issues and context switching would be costly. The potential to be gained from such multithreading are not worth the mentioned risks. The gains acquired from such parallelization are minimal, so the task granularity was chosen accordingly.

C. Limitations

Although the attempt was to create simulations as accurate and efficient as possible, there are a few limitations that require discussion. First, any runtimes discussed and depicted here are the result of running on the machines of the authors. Obviously runtime is machine dependent, but the numbers shown give an indication of the performance improvement possible through parallel computing.

Another limitation of this project is that it only uses cards from the original game (the original Base Set). To date there have been more than 10 expansions released, each of which with the potential to change gameplay strategy significantly.

Finally, there was a limitation on the artificial intelligence approaches used due to the computing power of the machines available. If access to a supercomputer was possible, then there could have been a heavier focus on more complex decision trees and Monte Carlo players, but this was not possible due to resource limitations. As such, AI approaches that were less resource and computation intensive were used

D. Future Directions

In the future there is a plan to expand simulations with more complex AI players and strategies, implement some or all Dominion expansions, and analyze more statistical data from simulated games. Another feature to be implemented is to perform kingdom analysis before each game [3].

While Monte Carlo was optimized to be more resource and time efficient, there is room for even more improvement. With more time and better optimization with programming concepts such as dynamic programming, along with faster equipment, it is possible to further reduce the runtime for the Monte Carlo Bot, while also increasing the simulations for each action, leading to better buy actions.

Furthermore, in the future, further analysis into thread pooling and thread overhead costs can be performed.

VI. ACKNOWLEDGEMENTS

We would like to thank UMBC and Professor Tyler Simon for imparting us with the knowledge necessary for us to accomplish this project.

VII. REFERENCES

[1] R.R. van der Heijden, “An Analysis of Dominion”, Bachelor Thesis, Mathematical Institute, Leiden University, Leiden, Netherlands, 2014.

[2] H. Bendekgey, “Clustering Player Strategies from Variable-Length Game Logs in Dominion”, Pomona College, Claremont, California, 2018.

[3] M. Fisher, “Provincial: A kingdom-adaptive AI for Dominion”, https://graphics.stanford.edu/~mdfisher/DominionAI.html (accessed Mar. 9, 2021).

VIII. APPENDICES

Source code is published at: https://github.com/Wcobb1/Dominion483.
Visualizations used to create figures are published at
https://www.khanacademy.org/computer-programming/radar-chart-v10/4803410506235904?width=1400
and
https://www.khanacademy.org/computer-programming/radar-chart-v11/6440293879463936?width=1400.