Abstract

This paper simulates draws and mulligans in a trading card game deck, with different hands and draws resulting in different "turns to wins," based on a limited number of cards and interactions between them. The mulligan in this sense is when the first hand of cards drawn is replaced in the deck and a new hand is drawn with one fewer card. The simulation is executed in both Python and Go to evaluate execution speed between the two languages. The results find that Go performs 10 times faster than Python at 1000 iterations, and 15 times faster at 100,000 iterations.

Introduction

Magic: the Gathering (MTG) is a trading card game owned and published by Wizards of the Coast (WotC), a subsidiary of Hasbro, Inc, a multinational toy company. In 2022, Hasbro's annual financial report identified MTG as their first ever billion-dollar brand. Since its first published edition more than 30 years ago, the complex, ever-changing strategic content of the cards themselves and several different rulesets has drawn strategically minded players to ever improved analytical techniques to try to maximize their game. The game is almost indescribably complex, with tens of thousands of unique cards, the majority of which have unique rules associated with them that interact with various other cards, and new cards being published by WotC multiple times every year. In such a complex game, simplifications and simulations can provide a proof of concept for more exhaustive analysis in the future.

This study seeks to simulate a single, somewhat simplified permutation of an infamous deck in Magic the Gathering called Mono-Red Aggro (aka Red Deck Wins, "Sligh" (pronounced "Sly"), or various permutations of those) which has been studied somewhat extensively due to

its relative ease of access and simple-but-effective ethos of winning the game before other competing decks can interact with it effectively. Succinctly put, it is a set of 60 cards which are intended to, as much as possible, remove variables from the game through incredibly aggressive play, winning (or, frequently, losing) the game before the other players' complicated strategies can unfold.

Literature

This deck's origin is mathematics in a very literal sense. In 1994, during an algebra class at Georgia State University, Paul Sligh conceived of an approach to MTG in which an idealized distribution of cards was determined in such a way that the cards drawn to the initial hand could be played quickly and consistently. Termed the "Mana Curve," ("mana" being a resource built over turns and spent to play cards from the hand) this concept is now fundamental to how experienced players distribute the cards in their decks, having been adapted over the succeeding decades as more complex strategies are developed and the number of interactions between cards continues to explode exponentially. Originally, the Sligh deck was an Aggressive, Red-only deck with low-mana-cost, direct damage one-time-effects and literally "hasty" creature cards that saw significant success in the early competitive circuit.

More recently, this hyper-aggressive playstyle has resulted in an outright ban of a card deemed to facilitate these decks to an uncompetitive degree. The card, "Leyline of Resonance," allowed players to play it before the first turn of the game if it was drawn to the starting hand, incrementally increasing the damage of all successive plays in a way that particularly benefited numerous small-impact attacks characteristic of Mono-Red Aggro. While the governing body of MTG has access to colossal datasets provided by the online virtual play platform MTG: Arena to determine the impact of this card in the current metagame, such observability-based conclusions can frequently benefit from simulations which don't rely on player behavior but

instead lean on the mathematical probabilities and complexities that may not yet have broad adoption among the player base. The other advantage of a simulation is that, while pre-release play-testing is a common technique to evaluate new cards in upcoming releases, a simulation can provide significantly more data to potentially inform WotC of an potentially problematic card prior to its disruptive release.

Of course, card-game analysis and simulation is nothing new. Much more frequently applied to games with traditional playing-card decks like Poker, Blackjack, or Texas-Hold'em, these games which have existed in much the same state for more than a century have been analyzed to the point where the mathematics of the games have been "solved." Now, the statistical underpinnings of these games are frequently expected to be understood even by experienced amateur players, let alone professionals. Now that statistical simulation methods are becoming more accessible, more complex card games can begin to be analyzed with the same methods that solved the original subjects. In some ways, this resembles the logical advancement of optimization seen in board games from Checkers to Chess to Go.

Methods

Speaking of Go, simulations for such a complex game are going to require exceptional processing power and efficiency to be sufficiently thorough to describe a meaningful proportion of the game. Our method here is to run comparable simulations for the extremely limited portion of the much broader play behavior (Mono-Red Aggro decks) in both Python and Go to be able to speak to the relative efficiencies of the two languages. A key component of such a benchmark is to keep the simulation methodology as closely similar between the two coding languages as possible.

In both Go and Python, we use the standard library, with packages for time and randomization and begin by creating a representation of the deck and creating variables for the deck, hand, and play state of the table. Then, we shuffle the deck by randomizing the sequence of the cards, draw cards into the hand by removing from the deck sequence and adding cards of the same value to the hand, and then playing simulated “turns” based on the cards available in hand. Once the shuffling occurs, all other actions are procedural, not randomized. We run various numbers of trials and record the benchmark values of how long the simulation took to run in total.

Importantly, while we are simulating a much simplified depiction of MTG, this method doesn’t only work for our simplified deck but could be adapted to other sets of cards or cards. However, there’s still a significant degree of detail that would likely be needed for this proof of concept to be adopted to a more accurate model of MTG behavior. That being said, this limited simulation was relatively easy to implement and could be repeated and improved upon not only by professionals employed by WotC and other card game companies, but also enthusiasts with limited understanding of simulation techniques.

Results

Running the benchmarks between Go and Python took approximately a week. Difficulty of execution of the simulations between the two languages is subjective, but as Python is relatively much more popular, implementation of Go code can, on average, be expected to take longer. The workgroup experience with the languages mirrored that of the real world, in which certain members had more working knowledge of Go compared to Python.

	Python	**Go**

1,000 Runs		
Result (# of turns)	3.28	3.31
Execution Time (ms)	30	3.3594
10,000 Runs		
Result (# of turns)	3.62	3.60
Execution Time (ms)	280	19.2989
100,000 Runs		
Result (# of turns)	3.64	3.64
Execution Time (ms)	2710	176.0068

Conclusions

As the goal of our simulation was to benchmark performances of Go and Python, and since we were interested in only a single, specific card deck, our code was run with the data included within. We ran multiple trials with different magnitudes of runs to see the effectiveness of the simulation within the language. As seen in the results table as we increase the number of runs for both languages, their results begin converging, leading us to conclude that their accuracy is very similar, if not identical. We do see a difference in the execution time of the code. Within the trials that we have performed, scalability efficiency starts to become a little more apparent. Go's execution time increases at about a rate of 6x for every magnitude increase of runs, whereas Python's execution time increases by about 9x. As we would further expand the use of our code to multiple other decks, and realistically start running exponentially more runs, Go's efficiency within this use case becomes much more apparent.

While Python holds a much larger market share within the industry, Go is not necessarily a niche language and there is extensive documentation available for both languages. Thus in

the context of simulating this problem, it is recommended that Go be used as the difficulty of languages used is relatively similar and Go offers more efficient processing and associated costs. Loading data into our system is a little beyond the scope of this assignment, as it would require access to other decks that are already constructed and differing play conditions, but this simulation could further be expanded in this manner. Use cases of this would be to discover card decks that minimize the number of turns to win. Ultimately, these simulations would also go beyond simulating isolated decks, and simulate 2 decks interacting with one another.

References

“Hasbro Reports Fourth Quarter and Full-Year 2022 Financial Results” published February 16th, 2023. https://hasbro.gcs-web.com/news-releases/news-release-details/hasbro-reports-fourth-quarter-and-full-year-2022-financial retrieved November 17th, 2024

Rhystic Studies, “RED DECK WINS”, published May 6th, 2022. https://www.youtube.com/watch?v=P5oc_9ObMzc retrieved November 17th, 2024.

Wizards of the Coast, “MTG Arena Banned and Restricted Announcement – October 22, 2024” https://magic.wizards.com/en/news/announcements/mtg-arena-banned-and-restricted-announcement-october-22-2024 retrieved November 17th, 2024