Alex Wiest

CS 20 Final Project Report

(Please note that in this paper, I use a few Magic: The Gathering terms that you might be unfamiliar with, so if you haven't already, you may want to read the document "crash_course_magic.md" first)

For my CS 20 final project, I've created a program that simulates playing a Magic: The Gathering game against an empty opponent. The program plays cards and wins the game as quickly as possible. This is repeated a user-specified number of times, logging what cards are played and all relevant info, for the user to look at and determine what deck generally beats the opponent fastest. This is something that people do in real life; a practice known as "goldfishing." However, it's boring and tedious and takes way too long to collect an adequate sample set for it to be worth it, when doing it yourself with paper cards.

The tediousness of it is why I wanted to make a program to simulate it for me. It's an idea I'd had a few months ago, but didn't have the time to spare to get started on it. When I read the description of the final project, I realized that this idea fit the requirements almost perfectly.

I say "almost perfectly" for a reason. The first obstacle I hit was when I was writing up the project description. In Magic: The Gathering, you search your library a *lot*, because there are many cards that let you. So, it was easy for me to know what to implement for the search algorithm, just by adding a fetch land, which is something I had been planning on doing anyways.

However, in real-life MTG, there's no situation where you sort anything at all. The closest I find myself coming is when I sort the cards in my hand, with the cards I want to play next turn in front, and the cards I don't need immediately behind them. I considered trying to make the Hand object (the data structure used to represent my hand) a priority queue, because that would kind of fulfill the sorting requirement. However, I realized that the act of sorting your hand all the time would probably take longer than just iterating through it, since it's only going to contain around a maximum of 7 object at any given time.

With that idea scrapped, I found myself trying to wedge a sorting algorithm into places it didn't belong. I tried putting a sorting algorithm into the Library (draw pile) object, keeping it sorted all the time and when the program would draw a card, it'd randomly select one to provide. I tried various ideas of that, such as using a dictionary with a key: pair of {Cardname : number of cards of this name in the deck}, but also scrapped that idea. My next iteration was to sort it by just lands and nonlands. With that idea, I tried figuring out which sort would be best for just two different elements, as all nonlands were sorted like they were the same element. I ended up making a cool custom sorting algorithm that works great for datasets with only two elements.

I belatedly realized the practice of keeping your library always sorted was far more trouble than it was worth. Even though it didn't work out, I'm pretty proud of my sorting algorithm, enough so that I kept it in commented out, in case I want to use it later. I tried again, this time with my current method of a randomized stack, just like a Magic library is in real life.

After having spent several days trying to shoehorn a sorting algorithm where it shouldn't go, I had a realization moment of "wait, this probably isn't the best approach to this." I thought about all the places where I could put a sorting algorithm in a way where it'd actually fit and be useful, and almost immediately realized that the best place would just be to sort the results of all the games. By default, it displays game 1, then game 2, then game 3 and so on, but you could sort it by how many turns you won in, and other things. Once I implemented that, it didn't obstruct me in other areas of my program, because the game results are very self-contained with nothing really depending on them, but more importantly, it's a natural place to put a sorting algorithm.

That whole situation taught me one of the most important lessons I've learned in CS, which is to let the program flow how it should. My brain intuitively has a very good understanding of logic flow and the different paths a program can take, but I hamstrung myself by placing an obstacle in my way that didn't need to be there. If I had let the program logic tell me where to put it, so to speak, I would have had a much easier time, rather than trying to wedge something where it didn't belong.

This project has easily been the single most complicated, intricate, complex program I've worked on in my entire life. I've learned so much from making this, from simple stuff like how Linux encodes their text files differently than Windows, to more complex stuff like how to break up code across several dozen functions, and have the code placed where it makes sense.

To elaborate on the latter part, a struggle I hit in many cases was how to properly break up large chunks of logic into smaller, more easily accessible functions. This was especially

evident in both the logic for knowing which land to play, and for knowing which lands I should tap to get the mana I want.

I was astonished at how difficult it was to get those functions to work the way I wanted them to, because of how simple it is in real life. I figured I'd just write out how my logic in real life works, and then code a program to do that. Turns out, there's a lot more logic that goes into those decision-making processes than I had originally accounted for. I wrote both of those functions using nothing but conditionals, and in hindsight, that's most likely not the most effective way to program an AI. However, I believe that given my relatively inexperienced knowledge of advanced CS concepts, especially when it comes to decision-making processes and AI, using conditionals was still the best route I could have gone right now. I don't know how long it would take to teach myself proper AI programming, but just from the surface-level skimming I've done on the subject, it's way more involved than I could realistically teach myself in a month, and still have the time to work on the other ninety-five percent of the project.

I don't regret that I used conditionals instead of a game tree and proper AI. However, I wish I had spent less time trying to perfect it. As a coder, I want all my code to be perfect. It doesn't bother me too much to strip away layers of unnecessary complexity for a game such as Magic, which depending on how far down the rabbit hole you want to take it, can be *extremely* complex. However, if I find myself writing a function, I want that function's logic to be as perfect as possible. That's not usually a problem, and it usually lets me get things done faster, because if I find a new way of understanding the logic, I'm not afraid to cut and paste some things around and change some loops and make the logic flow better. It becomes far easier from there to expand upon it and finish the function, as well as easier to understand when

revising it later. However, with an extremely logic-heavy, complex function, which in hindsight, I was already going about in the wrong way, I should have settled for “not perfect but good enough.” I did get to that point eventually, but only after spending a long time working on just the logic for mana costs and land plays.

Another struggle I had with those two logic chunks (knowing which land to play, and how to tap the lands in the correct priority) was that the logic involved is rather vague and not clearly defined, and it took a lot of trial and error to figure out the correct place to put each block of logic. For example, I quickly realized that I needed a “draw a card” function. That was easy to isolate: just take the top card off your library, put it in your hand, and return. However, with playing cards, it became a lot harder. Do I pay for the mana here, or put that in a separate function? If I pass a card name to the “play card” function, should it be assumed I already have the required card in my hand, or should I check for the card before I play it and return false if it’s not there? Those were but two of the many details that I wasn’t sure to how best approach. In almost every scenario, I tried it one way and found it didn’t work very well, so I restructured the logic and combined or split up functions appropriately to make it easier to follow the logic flow. This also made the individual functions more versatile and usable in different places.

All in all, I’m proud of my work in this project, which honestly is a very unusual feeling for me. I tend to be very self-critical but this is coming together more cleanly than I expected. There are still many things I want to continue adding in even after this has been submitted and graded, and I plan on actually using to this to figure out what decks I want to play in real-life Magic: The Gathering games.