Cardio Move Recommendation Bot:
AI Undergraduate Final Project Report

Table of Contents

Team Members

Everette David Webber | edwebber@csu.fullerton.edu | 886894732

Abstract

In this project report, we will start by introducing the problem statement we were trying to solve. Next, we will briefly explain the game we were trying to get our AI bot to suggest moves for as well as briefly explain some of the terms the readers might not be familiar with. After that, we will talk about the approach we took for this project. Next, we will give a breakdown of the software we built. This breakdown will include all the components we used, the limited UI for this project, the heuristic formula we ended up using, and all the languages and libraries that we used. After all that we will go over our evaluation of the software that we made, focusing on how it feels to use as well as how well it did. Finally, we will give our conclusion for what we made, discussing the lessons we learned and some ideas we have for future improvement.

Problem Statement

In this project, our group was attempting to make a bot for recommending the best moves for a given turn in the open-sourced card game Cardio[2].

What is Cardio?

Cardio[2] is a free open-source card game based on other popular card games like Inscryption[4]. In Cardio[2], the goal is to take out your opponent's health first by attacking them with creature cards that you play from your hand. What separates Cardio[2] from other card games like Magic The Gather[5] or Yu-Gi-Oh[6] is that the player must sacrifice weaker cards to get their stronger cards out to win. This means the player, and by extension, our AI must think through all the possible combinations of moves they can play to win.

Players begin the game with three cards in hand and no cards on the board. At the start of the player's turn, they must choose to draw from one of two decks, their own deck or a ten-card deck of all hamsters. Hamsters have only one health and do no damage, but they are the only card that can be played for free without needing to sacrifice another card. Once players have drawn a card they may play as many cards from their hand as they can. At the end of the turn, the player's cards fight the enemy cards across from them. If any card attacks an empty spot, the person who owns that spot (player or enemy) takes that much damage. If any card dies and there is still leftover damage that damage carries through to the player or enemy. After this, the player takes their next turn starting from choosing what deck to draw from. Turns repeat until someone runs out of health, at that point, they lose and their opponent is the winner.

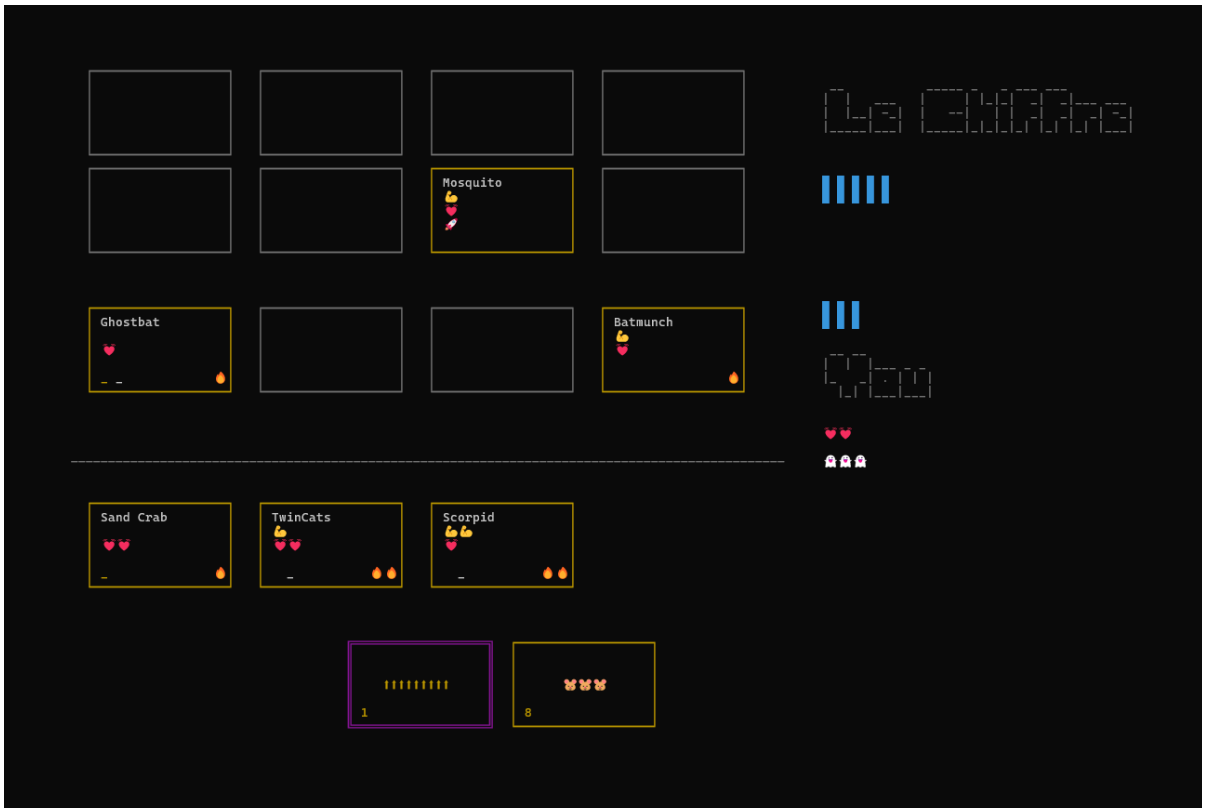

Figure 1: Cardio Gameplay

Explanation of Terms

In this section, we will briefly explain some terms that might not be known to those outside the gaming scene. Specifically, both terms are game genre titles.

Deck-Building

Deck-building [7] or deck-building[7] games refer to games where the player must slowly put together a deck of cards that have various effects. Most cards of unique or differing effects that may synergize in unique or interesting ways. The goal of these games is for the player to discover these synergies and put together the best possible deck for a given game. This will cause challenges for AI specifically because long-term planning with many unknowns is a given for this genre.

Rogue-Like

Rogue-like [8] (or roguelike) games are games that tend to revolve around randomized elements, permadeath, and run-based gameplay.

Randomized Elements

Throughout the course of the game, the player will gain the options to choose from a random selection of elements, or upgrades that will affect the rest of the run. These elements or upgrades may or may not have synergies with each other for the player to discover.

Permadeath

If the player dies or loses they go back to the beginning of the game, they don't get any checkpoints or saves. Once the player is dead that is it.

Run-Based Gameplay

The game revolves around single runs where the player can choose from a selection of randomized upgrades to try to last as long as possible. However, if the player loses they lose all their upgrades and go back to the start. To try it all over again.

Approach

Our approach used a modified greedy search algorithm using a custom heuristics function to evaluate a given board position based on things like damage taken, damage given, whether the enemy is dead or not, etc.

The AI holds a simple state of the game that is nothing more than a wrapper class with a bunch of lists for the spots for cards. The AI can then alter this state through a variety of actions identically to the actions the player can take in the game.

The AI will run through its modified greed search algorithm multiple times, each time with a lower heuristics threshold, until it finds a state that meets or exceeds this threshold. Once it has found something it will print out the actions needed to take to reach that state. Letting the user input those actions.

Software Description

In this section, we will go in-depth about the software that we made. We will start by looking at the components, then focus briefly on the UI, followed by looking at the heuristic formula we ended up using, and ending by going through the languages and libraries that we utilized.

Components

In this section we will go through each of the main files for our program and discuss the major functions or classes in each and what they do. You can also see a diagram below showing all the major components:

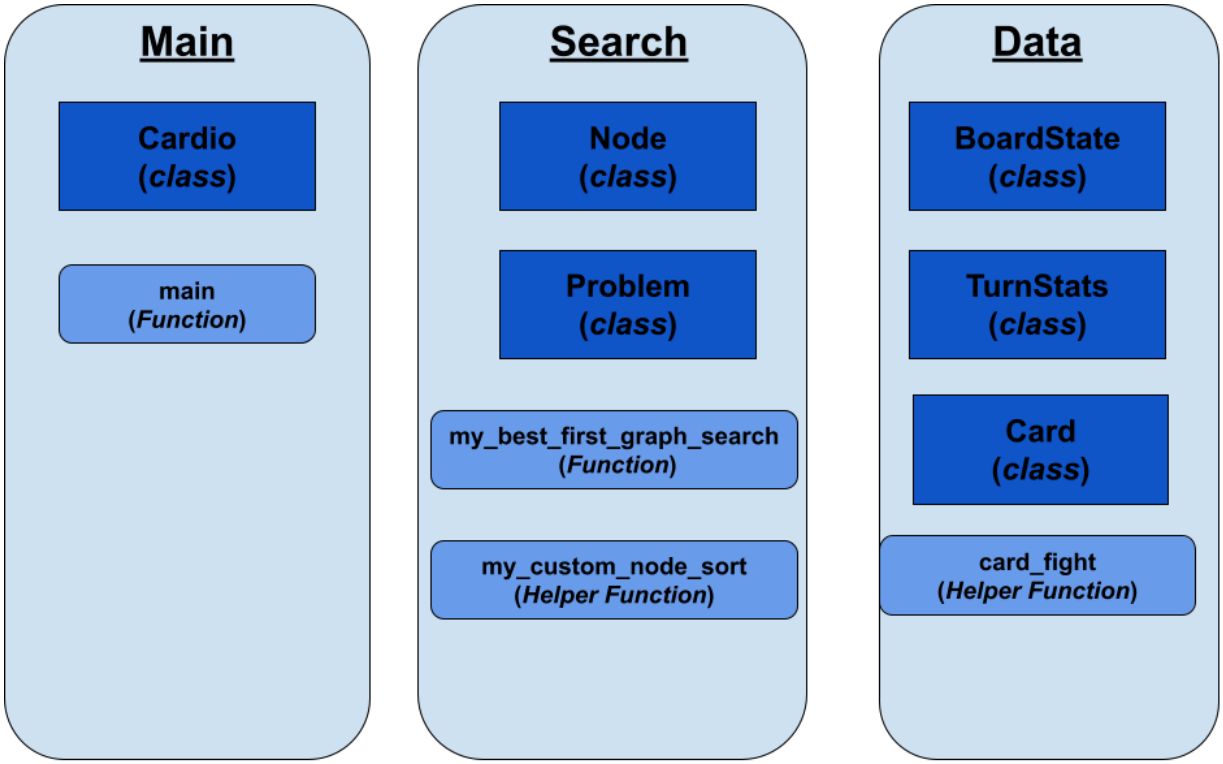

Figure 2: Software Components

Main

The main file holds the following relevant pieces:

- Main (Function)
- Cardio (Class)

Main (Function)

The main function sets up the initial state and initializes the Cardio class. The main function will continuously run the Cardio class through the my_best_first_graph_search function with a lower and lower goal threshold until a valid state is found. At that point, the main function will print out the list of actions needed to reach that state.

Cardio (Class)

The Cardio class is a subclass of the Problem class from the Search file[1]. The Problem class contains a framework for problems that can be solved by the various search functions found on the Search file[1]. In our implementation of the Cardio class, it is instantiated with an initial state and a goal threshold (an arbitrary number the given state must beat using its heuristics formula to find a valid goal state). The Cardio class also contains methods for getting all possible actions for any given state, what state you get for using any given action in a given state, and a method to run the given state through the heuristics test.

Search

The Search file[1] originally comes from the aima-python library[1] but has since been modified. For this project it contains the following relevant parts:

- my_best_first_graph_search (function)
- my_custom_node_sort (helper function)
- Node (class)
- Problem (class)

The search file[1] contains many other functions and classes. However, these are there since they were part of the original file in the aima-python library[1] and are not used or touched for the sake of this project.

my_best_first_graph_search (function)

This function started life as the best_first_graph_search function, which can still be found in the Search file[1]. However, given the nature of the project we needed to modify the function to meet a few different needs. One key need was to adjust it so the goal testing function only runs on states that were reached by using the Fight action. This is because in Cardio[2] all turns end with the player fighting. Without this adjustment, the AI would end in states that are not the fight action, which isn't helpful to the user..

The function starts with the initial state, it checks to see if this state is better than the goal threshold by looking at the output of the heuristic function. Next, it looks at all possible states that are one action away and adds those to the frontier (a list of future states to evaluate). Next, it sorts all the states in the frontier to check relevant nodes first. The relevant states are any states that come from fight actions because these actions are required to end a turn. After this, it chooses the first state in the frontier and repeats the whole process with it.

Once it finds a state that meets or exceeds the goal threshold in the heuristic function it returns that state and the actions need to reach that state.

my_custom_node_sort (helper function)

This is a helper function that is used by the my_best_first_graph_search to sort the states it has in its frontier (the states it still has to check). It sorts these based on the action taken to reach this state. The states are sorted in the following order: fight, drawing a hamster, drawing a card, followed by any remaining actions.

Node (class)

The Node class was a function included in the original Search file[1] and has not been changed or modified by me. It is still utilized by all other search functions on this file, including my_best_graph_search. A node contains a single state of a problem class, the parent node(the previous state), the action required to reach its state from its parent, all the child nodes, and the actions required to arrive at those nodes. Lastly it includes methods to return the path required to arrive from the root node at that given node.

Problem (class)

This is an abstract class that was included in the original Search file[1] and has not been changed or modified by me. It is utilized by all other search functions on this file[1], including the my_best_graph_search. It serves as the parent class for My_best_first_graph_search function. It gets initiated with an initial state and a goal state. It includes methods to get all possible actions in any given state, get the new state by using any possible action, and test if a state is the goal state.

Data

The Data file contains various helper or wrapper classes to compartmentalize the AI's understanding of the game cardio[2]. None of these functions are used in the game itself (none of the code in this project talks to or touches the game) but these are used in the AI's model of the game. The following relevant classes or functions can be found in this file:

- card_fight (helper function)
- BoardState (class)
- TurnStats (class)
- Card (class)

card_fight (helper function)

This is a static helper function that handles what happens when two cards fight. This is not used by the Cardio game[2] but is instead used by the model the AI uses to recommend moves to the player. This function will return the damage done to the defender and will delete the defending card if their health goes below zero.

BoardState (class)

This class is a massive wrapper class to encapsulate all the info about the board state in the model the AI uses. It contains lists for all the possible locations cards could be and stores Card classes in those locations. It contains various setter methods to set up the player's cards,

and the enemy's cards, and setters for the health of both the player and the enemy. It also contains methods for how the board state should change if the player draws a card, draws a hamster, or starts a fight. Lastly, it includes a method for printing out info about it for debugging purposes. Lastly, it includes a TurnStats class to encapsulate various stats about a given turn.

TurnStats (class)

This is a helper class that encapsulates info about various stats for a given state of a game. This class is mainly used when running the heuristic function on the Main file. The stats that are tracked include but are not limited to: damage the player and enemy took, max power on either side of the board, number of cards on either side of the board, and number of cards in the player's hand or deck.

Card (class)

This is an object class that is used to store all the info about a given card in play. This class gets created for each card and is passed around as that card moves between various locations on the board.

UI

The UI for this software can be broken up into the UI the user uses for inputting the boardstate and the output UI. However the first is the user directly editing the main function and the second is printed out directly to the terminal.

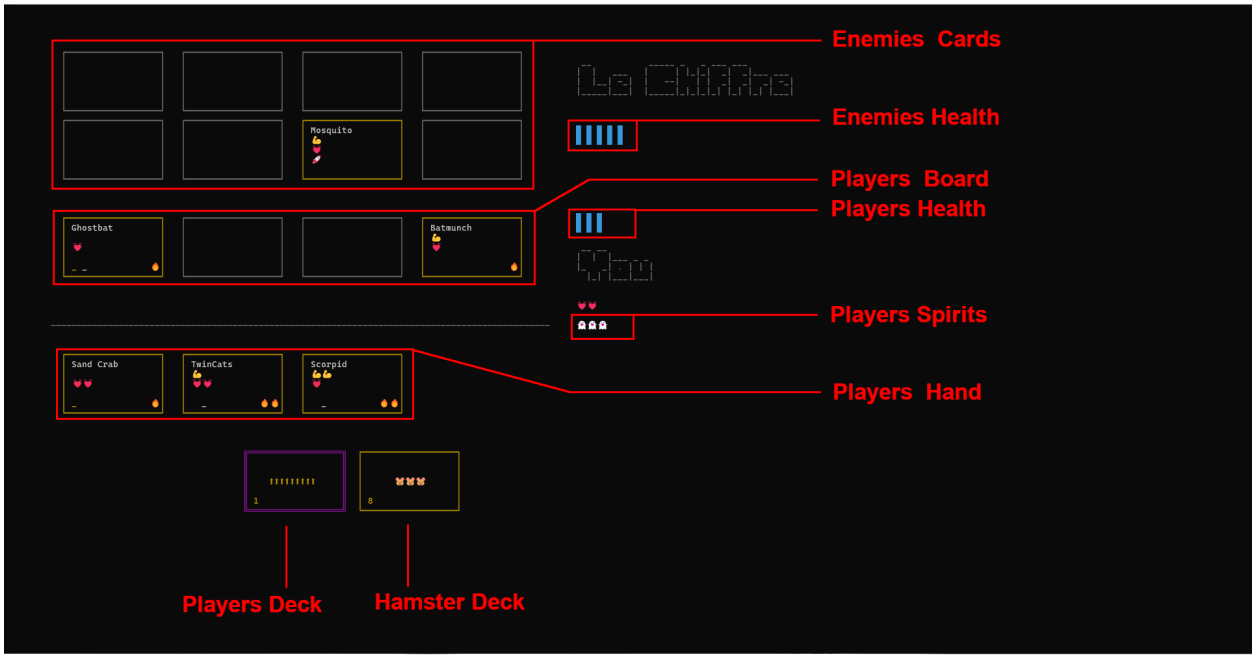

Figure 3: Cardio Game UI

Entering the Board State

Before running the AI the board state for the given turn must be manually entered on the main function on the Main file between lines 202 and lines 226. The player must enter data for the enemy cards, player cards (on the board, in their hand, and deck), the number of spirits the player has, the number of hamsters left in the hamster deck, and the health of both the player and the enemy.

```
201
202 #   Set enemy cards
203     enemy_cards = [Card(name="Sunfish", power=0, health=1), None, None, None,
204                    None, None, Card(name="Firepup", power=1, health=1),  None]
205     boardState.set_enemy_cards(enemy_cards)
206
207 #   Set player cards
208     player_cards_board = [None, None, None, None]
209
210     player_cards_hand = [Card(name="Smokepuff", power=0, health=2, fire_cost=2),]
211
212     player_cards_deck = [Card(name="Ghostshrew", power=2, health=2, fire_cost=2),
213                          Card(name="Batmunch", power=2, health=1, fire_cost=1),
214                          Card(name="Wisp Gazelle", power=0, health=1, fire_cost=0, spirit_cost=1),]
215
216     dead_cards = [Card(name="Weasel", power=1, health=1, fire_cost=0, spirit_cost=2),
217                   Card(name="Scorpid", power=2, health=1, fire_cost=2, has_spirit=1),]
218
219     spirits = 17
220     hamsters_remaining = 8
221     has_drawn = False
222
223     boardState.set_player_cards(player_cards_board, player_cards_hand, player_cards_deck, spirits, hamsters_remaining, has_drawn)
224
225 #   Set health
226     boardState.set_health(player_health=5, enemy_health=3)
```

Figure 4: Where the user enters the board state

Enemy Cards

Location is important as it should mirror the exact location of the enemy cards in the game. Any position not filled with a card should be filled with None. Lastly, the rows are reversed, so the bottom row in the game is the top row in the code (line 203)

Player Cards

Board

Location is important as it should mirror the exact location of the player cards in the game. Any position not filled with a card should be filled with None.

Hand

Location is important as it should mirror the exact location of the player cards in the game. There is no need to fill empty slots with None, just delete those slots.

Deck

Put any remaining cards here.

Dead Cards

This is where any cards that have died go. These include cards that have been sacrificed or died due to combat.

Spirits, Hamsters, and Health

Simply change these values to represent given values in the Cardio game[2].

Reading the Output

The output is printed to the terminal in two parts. The first part is the search and the second part is the result once it finds one.

```
Starting Search
  1 paths have been expanded and 2 paths remain in the frontierNone
Searching for goal with a threshold greater than or equal to:  6


Starting Search
  1 paths have been expanded and 2 paths remain in the frontierNone
Searching for goal with a threshold greater than or equal to:  4


Starting Search
  1 paths have been expanded and 2 paths remain in the frontierNone
Searching for goal with a threshold greater than or equal to:  2


Starting Search
  1 paths have been expanded and 2 paths remain in the frontierNone
Searching for goal with a threshold greater than or equal to:  0


Starting Search
  1 paths have been expanded and 2 paths remain in the frontierNone
Searching for goal with a threshold greater than or equal to:  -2


Starting Search
  1 paths have been expanded and 2 paths remain in the frontierGOAL FOUND
9 paths have been expanded and 3 paths remain in the frontier

Steps to get to goal of threshold -2
None
DRAW_HAMSTER
PLAYCARD_1_AT_2
FIGHT

Process finished with exit code 0
```

Figure 5: Example Output

The Search

This shows both what threshold it is using as well as how many nodes it has explored. For more complicated turns these may need to search upwards of 1k nodes before it finds a result. If no result is found it tries again with a lower threshold.

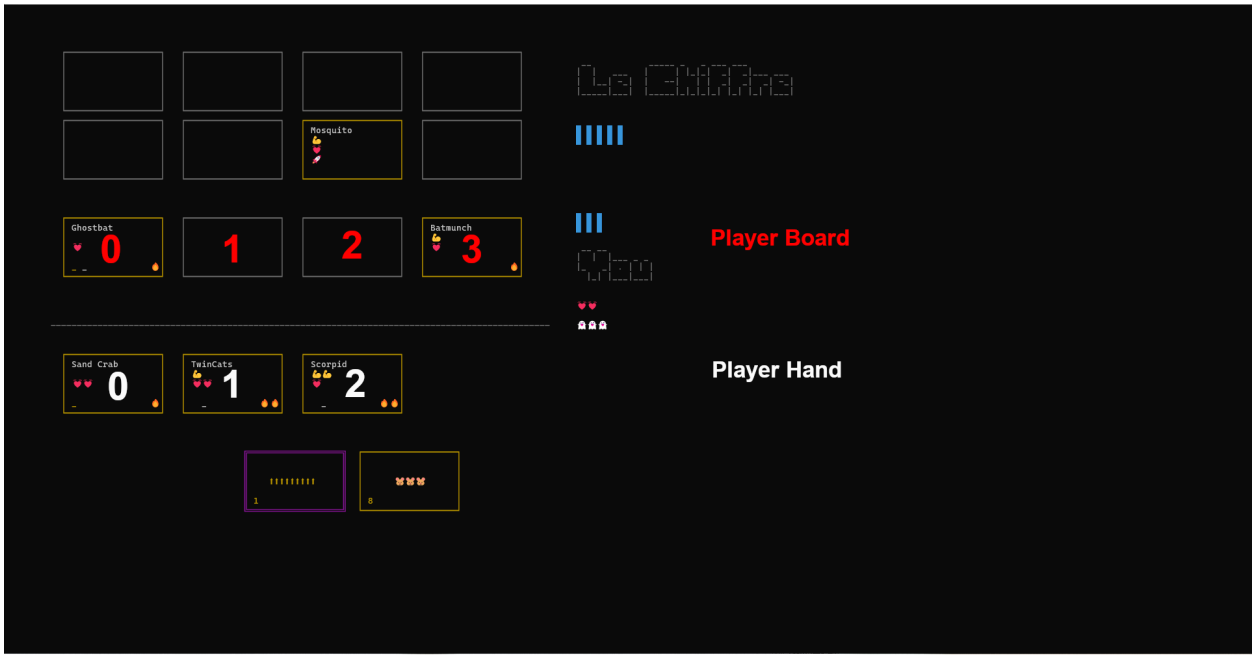

Figure 6: Where the numbers in the outputs are referring to

The Result

The result lists out the actions needed to get to that goal. For actions like PLAYCARD see the above image for what slots the numbers mean.

Heuristic Formula

The formula we settled on was as follows:

(2 * damage_to_enemy) + power_advantage - (1.5 * damage_taken) + enemy_is_dead - player_is_dead - is_stalemate

Where power_advantage is the difference in power (damage output) between the player's and enemy's cards on the board. Enemy_is_dead, player_is_dead, and is_stalemate are either 0 if false or massive numbers if true (ranging from 100 to 999) to heavily influence the heuristic result.

Languages and Libraries used

This program was built using the Python coding language. The main Integrated Development Environment was PyCharm 2022.2.1 (Community Edition). The main libraries

used included: itertools, ast, copy, collections, queue, time, and search from aimacode's aima-python repo on Github [1]

Evaluation

In this section, we will evaluate the software we made by going over how it feels to use as well as how well it did.

How it feels to use

The software is admittedly extremely clunky to use. The user must manually enter all the info about the boardstate of the game at the start of each turn before being able to see the recommended moves. The user must also enter this info into the source code itself since building a dedicated front-end UI fell out of the scope of this project. However, despite some slow calculation times on some turns due to hardware limitations, the AI is generally pretty quick in returning some solid moves for the player to use. The recommended moves are printed out in the terminal in relatively plain text allowing the user to easily input them into the game. Overall the front-end user experience is the area we would want to improve the most, but under the hood, the AI runs solidly.

How well it did

The AI does well in most cases recommending optimal moves for the player. In several test runs it provided the user with viable moves for any given turn. In one test run it was able to make it through five fights and only lost to an incredibly tricky fight that we doubt a human player could have found a way out of.

The biggest edge case it suffers from is that it can't look multiple turns ahead. This means it may suggest moves that will result in a deadlock (causing the player to lose) or moves that are optimal on a single turn but will not work out in the long term. One major strategy in the game is playing cards on one turn that can be used as a resource on the next turn. Since it can't look multiple turns ahead it does not recognize this strategy. However, on a single turn-by-turn basis, it tends to recommend highly optimal moves that even a human player would see and go for.

Conclusion

After everything we've gone over about the software we made, we will take some time to reflect on it. Going over the lessons we learned well making it and the ideas we have for possibly improving it in the future.

Lessons Learned

There were many lessons learned from this project. But a few of the major lessons were: good search algorithms for unknown answers, the computational complexity of normal AI (as opposed to small assignments), and the importance of developing a decent front end or UI.

Good Search Algorithms for Unknown Answers

Search algorithms are incredibly important in many AI projects. From finding the best video to recommend, to the right move in a card game, some AIs are nothing but search algorithms. That was incredibly true with this project as it was nothing but a search algorithm to find and recommend the best move. And well we are indeed taught many search algorithms in school. We found ourselves having a hard time applying any of them to this project. That was mostly because most search algorithms are taught with a concrete or known start node and end node in mind. For this software, we didn't know what the end node was. We were just trying to find the best possible set of moves. And for that matter how to rank the best possible state to end a turn-on was tricky enough to solve.

Much of the time we are taught AI or any software to solve known problems with known solutions. And many of the real-world applications for that AI or software are throwing them at vague problems with unknown solutions.

For our software, we used a greedy best-first algorithm that used a thrown-together heuristics formula we more or less half-guessed at. We did a little tweaking to it but we didn't get a chance to do a full analysis on it.

Working on this project has gotten us curious about what algorithms other companies use when trying to solve problems with unknown answers.

The Computational Complexity of Normal AI

When the AI is running it has a little output that shows how many nodes it has explored and how many it has its banks yet to explore (its frontier). On some turns that output would start to read upwards of 10k nodes having been explored and nearly 100 left to explore. Some of those were on runs where it never found a path that met the goal threshold, so it would start all over looking for a path that met a new lower threshold. We have always heard that AI can get complex and it can require dedicated hardware, servers, software, etc. But we never got a scale of that until running this little project of ours.

The Importance of a Decent Front End

A good UI can make terrible software feel amazing to use and a bad one can make great software feel terrible. We have no front end to our software. On every turn, the user must manually enter all the data about the board state into the source code itself, all the cards, their effects, and locations, and must not make any mistakes. Next, the user has to compile and run

the code in their terminal. After that, the software prints some info showing that it is searching before finally printing out the best results that it found.

This process of manually entering the board state on every turn heavily slowed down the testing process. Well it does work, it is for this reason alone that we weren't able to do a thorough in-depth performance test of the AI. Good UI will be completely invisible, but bad UI is so clear the user will want to look away, turn off the program, and never touch it again.

Ideas for Improvement

Art is never truly finished, it is only abandoned. And well the jury is still out on whether or not we will improve on this project here are a few things we would improve if we did. Firstly we'd improve the UI, at least making it so the user does not have to touch the source code to run it. Secondly, we'd fine-tune the AI to avoid stalemates or boring plays. In some edge cases, the AI will recommend moves that get the user stuck in a stalemate or deadlock. Finally, one thing we'd like to improve is working up to the point of having the AI play the game. At the moment the AI only recommends moves to the player. However, we love the idea of sitting back and being able to watch the AI run through the game all on its own. This last one is admittedly more of a stretch goal, but it is one thing we would love to improve.

References

1. aimacode / aima-python (Github Repo)
2. ymyke / cardio (Github Repo)
3. My Final Project (Github Repo)
4. Inscryption (Web Page)
5. Magic The Gathering (Web Page)
6. Yu-Gi-Oh (Web Page)
7. Deck-Building (Web Page)
8. Rogue-Like (Web Page)