# Implementation of Collectible Card Game AI with Opponent Prediction

(Implementacja agenta do kolekcjonerskiej gry karcianej
z predykcją ruchów przeciwnika)

Łukasz Klasiński Wojciech Meller Marcin Witkowski

Praca inżynierska

**Promotor:** dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

15 lutego 2020

**Abstract**

In the last decade, creation of digital versions of collectable card games (CCGs) caused a rise in those games' popularity. Due to specificity of this genre, CCGs are an excellent research environment for game agents with imperfect information. In this thesis, we develop an agent with multiple strategies for Legends of Code and Magic, a programming game created for AI research. This game is a simplified variation of popular collectable card games like The Elder Scrolls: Legends and Hearthstone.

We examine an idea of deck prediction, with the main strategy based on the game simulation. Presented options, based on Monte Carlo Tree Search, are compared against agents presented at competitions hosted by IEEE CEC and IEEE COG conferences.

Although tests of our approach did not yield definitive evidence to support this idea, we think this concept is worth further research. Experimental data also show that off-line drafting method outperforms other considered drafting agents.

---

W ostatniej dekadzie, kolekcjonerskie gry karciane powróciły do swojej dużej popularności przez pojawienie się tego gatunku na rynku gier komputerowych. Dzięki swej specyfice, gry te są dobrym środowiskiem do prowadzania badań nad agentami do gier z niepełną informacją. W naszej pracy skupiliśmy się na stworzeniu kilku strategii w ramach agenta do Legends of Code and Magic (LoCM), programistycznej wariacji strategicznych gier karcianych, uproszczonej na potrzeby badań. LoCM w dużej mierze był wzorowany na popularnych grach takich jak The Elder Scrolls: Legends oraz Hearthstone.

W pracy rozważamy ideę przewidywania kart przeciwnika, przy czym główna strategia polega na symulacji gry. Prezentowane implementacje bazują na metodzie Monte Carlo Tree Search. Podczas analizy porównujemy je z agentami zaprezentowanymi na turniejach organizowanych w ramach konferencji IEEE CEC i IEEE COG.

Pomimo, że przeprowadzone eksperymenty nie przyniosły ostatecznych dowodów potwierdzających skuteczność tego pomysłu, uważamy, że ta idea jest warta dalszych badań. Dane eksperymentalne pokazują również, że metoda dobierania kart off-line jest lepsza of innych rozważanych propozycji.

# Contents

# Chapter 1

# Introduction

Over the years, the AI development for collectable card games (CCGs) became very popular with research being mainly focused on Hearthstone and Magic: The Gathering. There are many tournaments such as Hearthstone AI Competition being held. We have taken on similar task and developed agent for Legends of Code and Magic [1], hereinafter referred to as LoCM. This game was created by Radosław Miernik and Jakub Kowalski with AI competition in mind – two contests in version 1.2 of rules [2, 3] and one in 1.0 [4] were conducted to date of writing this thesis.

## Collectable card games

Collectable card game is a board game for multiple players, usually two, where cards are collected and composed into decks. Those are then used to perform certain actions that change current state of the game. Popularity of CCGs is often attributed to Magic: The Gathering that was created in the last decade of 20th century. With creation of digital versions of CCGs, due to huge amount of possible board states and incomplete information about opponent this genre became popular in AI research.

The mechanics of each CCG can differ, but typically it requires players to own their decks. Matches are played in turns, which consist of drawing cards, playing them from hand and performing actions with those already on board. Actions modify players' and board's state.

Most notable about CCGs is the amount of playable cards. For example, Magic: The Gathering has almost 20,000 cards, Hearthstone with over 3,000 (around 1,000 of them unobtainable by players) and The Elder Scrolls: Legends with over 1,000 cards. This characteristic increases information imperfection and randomness as the deck can be any of the cards' subset with specified size and is often shuffled at the beginning of a game. This often introduces a *metagame* as deck building becomes

important part of the game as a whole.

## Problem description

LoCM consists of two parts, that are inspired by the Arena mode of other games.

- Draft Phase

  During this part of the game, agent has to choose one out of three cards in 30 turns creating its own deck. Opponent is given the same choices, but its actions are hidden.

- Battle Phase

  The goal of this part is to reduce opponent's health to zero points or below using cards obtained during draft phase.

Cards are split into two types. Creature cards can be summoned to board, which itself is split into two lanes. Summoned creatures can attack opponent or opponent's creatures on the same lane. Each creature has certain statistics, including attack, defence, cost points, and passive abilities. Those abilities alter default behaviour, i.e. *breakthrough* applies excess damage dealt to opponent, *charge* creature can attack in the same turn it was summoned, *drain* heals player for amount of damage dealt, *guard* has to be killed before player or other creature in the same lane, *lethal* kills target if any damage is dealt, *ward* protects creature from the next attack with non-zero damage.

Item cards can modify attack or defence points of their targets and alter their abilities. Summoning creatures or using items can also modify each player's health points and add extra cards to their hand.

As LoCM consists of two logical parts, we can create separate agents for both of them:

- Drafting agent

  Since number of cards is limited to a relatively small number (160), we can differentiate two strategies – either calculating cards' values off-line and picking them through resulting order or choosing cards on-line by evaluating their worth given current state of the deck.

- Playing agent

  Keeping in mind drafting mechanics, we wanted to take advantage of information we are provided to predict opponent's deck. As such, we tried to determinize opponent's state, and use perfect-information state algorithm like Monte Carlo Tree Search.

Furthermore, agents have limited time to perform their moves. First turn of draft and battle phase has one second before timeout and the remaining turns have $200ms$ each. If agent does not make a move on time, it loses by default. Due to this constraint, agents have to perform more aggressive searching for optimal action. It is also one of the reasons for us to pick faster language such as Rust.

For further information and more detailed rules description in version 1.2, please refer to the LoCM project page [5].

# Chapter 2

# Analysis of existing agents

In an attempt to prepare for development, we have analysed few existing agents for this game genre. We focused mainly on AI implementations showcased in contests and journals.

## 2.1 LoCM

Since release of Legends of Code and Magic, two competitions were held and multiple agents were presented, most notably Coac that won both of them.

The base algorithm of this agent is min-max's variation of depth three with alpha-beta pruning. For every action in current state, min-max is evaluated for up to three actions per chain. Then, for each sub-state, min-max of depth one is executed for enemy player. First action with maximum score is then added to current chain of actions. This process is repeated for as long as possible without time-outing whilst there are no more valid moves. If there are any unused cards on board, all of them will attack opponent. In the draft phase, Coac uses hard-coded tables of best cards, constrained by minimal amount of each type. As second player has always one mana more, there are two sets of weights exploiting this fact.

Another agent with completely different approach is UJIAgent3. Instead of some kind of game-tree traversal, an on-line evolutionary algorithm was used. Firstly, all of possible actions are generated. Then, until the end of the round, random permutations of possible actions are taken under consideration, for one with the highest score being performed by the AI. For draft phase, authors used Coac's drafting order with extended constraints.

Most of the other agents are either using min-max with depth of two or a greedy algorithm with hand-crafted heuristic function. Almost all of them are using static drafting strategy with presorted cards, based on play-outs between top agents with the highest win-ratio. Majority of them have implemented routine checking

if they can kill an opponent in current round, fall-backing to their default search algorithm.

As drafting phase of Legends of Code and Magic presents unique challenge compared to other CCGs, this subject is further explored in LoCM creators' work [6]. In this paper the authors propose genetic algorithm that uses a concept of an active gene to adjust choices to partially created deck. In their tests, three variants of this method outperformed non-evolutionary approaches and evolutionary baseline.

## 2.2 Hearthstone

AI development in Hearthstone is pretty advanced at the moment. Since it is well-known in the Hearthstone community that the in-game AI is pretty bad (no look-ahead; kills its own creatures, sometimes even itself) the desire for better agents is pretty high. Considering its complexity and available tools [7], this game is great for research about games with partial information [8]. As deck-building is hard and human players rarely use custom-made decks, most Hearthstone agents are assuming there is some set of pre-made decks.

As a result, and the fact that drafting mode in game is not for free, gaming agents are made and trained to play with single deck. It makes a big difference in comparison to LoCM, because agent is much less dependent on drafting and can easily guess what deck the opponent is using just by seeing first few cards. It makes predictions and creating good neural networks for AI more viable than other approaches.

Since drafting is limited to chosen set of cards (only from rotating formats), players use win-ratio statistics from the web of given cards to calculate their chances of winning. After that, mana curve, synergies and number of copies in deck are taken into account to decide what card is the best. Many players use this whilst playing Arena mode as an in-game add-on, that shows them calculated scores for each card. This helps them to decide best pick from three cards they are offered.

In case of AI's implementation, we found out that in almost every more relevant paper, agents use some variant of MCTS in conjunction with chosen reinforcement learning technique [9]. For deck-building, evolutionary algorithms [10] were most prominent.

## 2.3 Magic: The Gathering

Another interesting card game we stumbled upon whilst doing our research was Magic: The Gathering. It proved to be huge challenge to write good agent for this game, because of massive number of playable cards, as well as big variety of their

types and how they influence state of the game. For example, there are possible recursive actions, ways to play infinite number of cards and it was even proven that this game is Turing Complete [11]. Because of that, creating agent proved to be impossible using standard methods and workarounds were needed.

In C.D. Ward and Peter Cowling's work about card selection using Monte Carlo search [12], authors had to limit agent to play only single, pre-constructed deck from cards permitted by standard format [13], which allows only $\approx 2,000$ cards. To restrict huge branching factor of the game, they split agent into two parts. First part consists of MCTS algorithm and is used to decide what cards, and in which order, are played from hand, which is limited by simple rules (e.g. if you have given card on hand, play it first). Second part is used in attack phase of the game, which in contrast to LoCM allows only damaging an opponent. After declaration on which creatures will attack, opponent has to decide how to block them. Authors used simple greedy approach that generates all possible moves and picks one which would leave board state to its favour. This combination gained the best results in comparison to the other agents they have implemented, but they believed there is still room for improvement.

# Chapter 3

# Our solution

Firstly, we have implemented a basic game engine, along with multiple routines allowing us to communicate with referee. Then, we started working on abstraction that would later allow us to write agents, ignoring any I/O or other things that are not directly related to AI. Having that done, we moved to creating simple agents that would allow us to test created system. Lastly, we wrote unit tests for already existing code. From this point, we were ready to start thinking about more advanced solution.

## 3.1 Drafting algorithms

### Random draft

This was the simplest drafting agent, used mostly as a reference whenever other drafting methods are effective.

### Endgame Evaluator (EEval) draft

The idea behind this algorithm is based on MCTS. It provides heuristic that calculates value of the cards by simply summing its health, attack, card draw, inflicted damage and in case of creatures – abilities with weights (e.g. *charge* has priority 1 and *guard* 3). At first, we decide 3 cards via the heuristic function to provide base deck for simulations. After that, we pick cards by playing random game endings with each of them in deck. Number of games is limited by provided timeout limit.

Each card has $(timeout/3) - delta$ time to play games which is enough to make about $2,000$ simulations. The game endings are prepared by randomising all states of board – number of cards in both players hands, random number of creatures from deck on the board and random health. Other than that, we create random deck for opponent from cards that were seen in draft at given iteration. Actions

are played randomly and then the result is selected based on the score. Score is calculated by increasing it in case of win and decreasing after lose, and then normalised between 0 and 1. This approach lets us to try each card in many possible board states and draw the one that fits best.

### Off-line draft

Since current number of cards is rather low (160), it is possible to quite easily calculate value for each of them. Then, during the draft phase, the card is chosen based on their weights. At the beginning we have used ClosetAI's weights to test this kind of drafting method. Later, we used harmony search [14] to improve sets of weights provided in main LoCM repository, as they were trained and optimised for older version of rules with single lane. Search started with four Coac's sets, one ClosetAI's and 10 random ones, for which $1,500$ *improvisations* were run. Harmonic memory considering rate was set to 0.75, pitch adjusting rate to 0.5 and maximum pitch adjustment index to 10. We later refer to those weights as **HS Weights**.

## 3.2 Battle algorithms

### Aggressive agent

This algorithm is pretty simple: consider all currently possible actions and pick one that would deal the most damage, simulate the action and repeat this process whilst an action can be made. Particularly, it means that agent tries to attack the opponent, eliminate creatures with *guard* ability and use or summon cards as to optimise potential damage per mana point.

### MCTS agent

Our next approach was to write an agent using MCTS algorithm with UCT [15] tree policy. We represented single actions as edges and resulting states as nodes in our graph.

Firstly, we implemented simple heuristic that was looking only on difference between players' health and amount of cards on the board. After some trail and error with modifying heuristic, we were able to win about 40% of games against Baseline, which was not that satisfying. Whilst taking closer look at AI decisions, we have discovered that in situations when agent could clearly win the game by simply attacking opponent with all of its creatures, it was failing to do so. Our first attempt at addressing that issue was increasing amount of fully opened branches in tree. We did this by modifying MCTS algorithm to always expand selected node, even if it was not visited before. That yielded a score increase of 10 *p.p.*, but did not actually

resolve the issue. Then, we tried to penalise agent in situations where it could kill the opponent, but that solution was not enough in all the cases. Ultimately, we created routine that manually checks if we can kill enemy player with our accessible resources. In conjunction with altered heuristic we were able to beat Baseline 80% of the times.

### Weighted MCTS agent

As hand-crafted heuristic functions are in most cases sub-optimal, we tried to avoid this by making use of evolutionary programming. We used our basic MCTS agent as a base, and then started thinking about better board state evaluation. First thing that came to our mind was creating a function that could better describe usefulness of cards that are already on the board. For this we have created a matrix $\mathbf{W}$ of size $8 \times 8$, which was representing weights for card's attack, defence and its abilities. Then, the score for single card is $\sum_j \sum_i W_{ji} c_i$, where $\mathbf{c}$ is a vector consisting of card's attack, defence and presence or absence of possible abilities. Finally, we used that function to evaluate every card on the board by adding result if card belonged to AI and subtracting otherwise. We chose matrix instead of simple vector, because we wanted to have a way to express relationships between different attributes of card, e.g. we do not need to have high amount of damage if we already have *lethal* – in this scenario *ward* or high amount of defence would be much more valuable.

After we had working agent, we have begun the training. We implemented genetic algorithm and run it with population size 150 and mutation chance of 2%. After each generation, we were crossing 10 best individuals, then adding 5 new random phenotypes. As for mutation, we were altering randomly chosen gene with new random value. For fitness function we have used percentage of how many games given individual won against specified AI. For this we used AntiSquid, Coac and self-play with currently the best individual. After all, we did not get any good results with this method, mainly due to limited amount of generations (100) we were able to evaluate. Because of that and high amount of parameters, this method gained no performance improvement of our agent.

### Predict MCTS agent

After being done with the plain MCTS, we started to think about how to predict opponent's moves, and thus, to be able to simulate more than only actions performed by already summoned creatures. By exploring couple of ideas, we implemented simple routine which filters out cards that agent already seen from all 3-tuples known after *Draft phase*. Then, during MCTS play-out we are sampling with uniform distribution subset of cards that lasted after filtration. The main idea behind this stochastic approach is that if we run it enough times per single state evaluation, we will get a more genuine description of how much impact given action will have. This

method proved to be somewhat effective as it generally increased win-rate of our agent by about 10 *p.p.* To tune heuristic, we again used genetic algorithm – this time with fewer parameters and hand-crafted relationships between abilities. Using this method, we were able to have about 70% win-rate against AntiSquid and 35% against Coac. Overall, it increased win-ratio of our agent by another $\approx 10$ *p.p.*

#### R.A.V.E MCTS agent

This agent works similarly to previous method, but this time with implemented AMAF heuristic [16]. For $\beta$ factor we used simple linear smoothing, that is $\max(0, \frac{smoothing - visits}{smoothing})$, with *smoothing* set to 15. Like in previous approach, we fine-tuned before obtained weights using genetic algorithm. This little upgrade in conjunction with weighted predictor yielded 85% win-ratio against AntiSquid and 45% in opposition to Coac.

### 3.3 Prediction methods

#### Empty

As we wanted to test if our agents work, we created predictor which always returns empty deck.

#### Simple

Since we did not want to take too many resources from MCTS, we created a simple deck predictor that keeps cards already played by an opponent. When asked for prediction, it checks what cards must have been picked by iterating over all tuples, then chooses rest at random.

#### Weighted Simple

This approach is similar to the previous one: we are still sampling cards from remaining tuples, but instead of all cards having exactly same chance for being chosen, their probability is based on weights from ClosetAI.

#### Same-as-ours

Another approach is based on `Simple` method, but it assumes that our opponent made the same choices as we did whilst drafting. Firstly, we pick cards that were seen and then choose the rest using same method as our agent did.

### ClosetAI

Because most of the other AIs in LoCM competition are using some kind of off-line, weights based drafting method, we tried out an idea to base our prediction for MCTS using unchanged ClosetAI's draft order.

# Chapter 4

# Results

All statistical data presented below is a result of 500 games per pair of our agent and one out of four others – **AntiSquid**, **Baseline1**, **Baseline2**, **Coac**. We tried them using all possible drafting methods and in case of **Predict MCTS** agent, using different prediction functions. The best result from each batch is **bolded**.

| ***Drafting method*** | **AntiSquid** | **Baseline1** | **Baseline2** | **Coac** |
|---|---|---|---|---|
| Random | 25.20% | 53.40% | 39.40% | 4.80% |
| EEval | 26.00% | 55.20% | 41.80% | 6.43% |
| ClosetAI | **31.60%** | 58.80% | 36.20% | 4.06% |
| HS Weights | 29.40% | **67.00%** | **59.40%** | **8.00%** |

Table 4.1: Battle agent: **Random**

| ***Drafting method*** | **AntiSquid** | **Baseline1** | **Baseline2** | **Coac** |
|---|---|---|---|---|
| Random | 39.20% | 63.40% | 51.20% | 4.80% |
| EEval | 46.69% | 66.40% | 54.20% | 7.61% |
| ClosetAI | **51.40%** | 70.80% | 55.00% | 7.69% |
| HS Weights | 48.70% | **73.80%** | **61.80%** | **9.80%** |

Table 4.2: Battle agent: **Aggressive**

| ***Drafting method*** | **AntiSquid** | **Baseline1** | **Baseline2** | **Coac** |
|---|---|---|---|---|
| Random | 67.93% | 86.44% | 70.53% | 12.47% |
| EEval | 62.39% | 89.46% | 76.51% | 15.23% |
| ClosetAI | 70.80% | 86.45% | 68.74% | 15.27% |
| HS Weights | **73.09%** | **91.64%** | **91.64%** | **23.12%** |

Table 4.3: Battle agent: **Basic MCTS**

| ***Drafting method*** | **AntiSquid** | **Baseline1** | **Baseline2** | **Coac** |
|---|---|---|---|---|
| Random | 82.44% | 97.73% | 92.24 % | 37.99% |
| EEval | 78.17% | 96.42% | 90.85% | 38.47% |
| ClosetAI | **88.26%** | 97.77% | 96.76% | 41.67% |
| HS Weights | 86.78% | **98.16%** | **96.92%** | **52.68%** |

Table 4.4: Battle agent: **Predict MCTS**
Prediction method: **Empty**

| ***Drafting method*** | **AntiSquid** | **Baseline1** | **Baseline2** | **Coac** |
|---|---|---|---|---|
| Random | 75.93% | 95.29% | 90.73% | 30.24% |
| EEval | 78.54% | 94.88% | 91.51% | 32.17% |
| ClosetAI | **86.56%** | **96.98%** | 94.97% | 37.00% |
| HS Weights | 82.57% | 96.65% | **97.13%** | **50.72%** |

Table 4.5: Battle agent: **Predict MCTS**
Prediction method: **Simple**

| ***Drafting method*** | **AntiSquid** | **Baseline1** | **Baseline2** | **Coac** |
|---|---|---|---|---|
| Random | 75.20% | 94.03% | 91.96% | 34.22% |
| EEval | 82.70% | 93.80% | 90.55% | 35.39% |
| ClosetAI | **86.64%** | **97.78%** | 96.75% | 36.66% |
| HS Weights | 83.50% | 97.11% | **97.77%** | **47.22%** |

Table 4.6: Battle agent: **Predict MCTS**
Prediction method: **Same**

| ***Drafting method*** | **AntiSquid** | **Baseline1** | **Baseline2** | **Coac** |
|---|---|---|---|---|
| Random | 78.73% | 95.66% | 94.66% | 34.49% |
| EEval | 78.54% | 93.15% | 89.51% | 34.69% |
| ClosetAI | **84.85%** | 96.96% | 94.93% | 35.54% |
| HS Weights | 83.97% | **97.15%** | **96.16%** | **49.06%** |

Table 4.7: Battle agent: **Predict MCTS**
Prediction method: **ClosetAI**

| ***Drafting method*** | **AntiSquid** | **Baseline1** | **Baseline2** | **Coac** |
|---|---|---|---|---|
| Random | 77.00% | 94.42% | 90.48% | 33.53% |
| EEval | 76.92% | 93.30% | 90.14% | 37.16% |
| ClosetAI | **86.23%** | **96.37%** | 95.34% | 35.90% |
| HS Weights | 82.05% | 95.98% | **95.95%** | **49.49%** |

Table 4.8: Battle agent: **Predict MCTS**
Prediction method: **Weighted Simple**

## 4.1 Conclusions

Analysing the above data, we concluded that our prediction methods seem to make smaller improvements than we expected, but we are not excluding that usage of more sophisticated algorithms to determine opponent's state may provide better results. Since agent with prediction was not that far off from best results, we believe they introduce too much noise to the decision-making. Possibly, some combination of `empty` during the early and the mid-game with `simple` at the late-game could be an improvement, since predictors become better with each turn as we see more cards played by an opponent.

Off-line drafting, using our improved set of weights yielded much better results than other methods. On average, it increased win-ratio by about 6.5 *p.p.* This method proved to be superior to on-line drafting solutions. With greater randomisation parameter and more time for training, we think it could be further improved. Our best agent turned out to be Predict MCTS using Empty predictor, as we can see in Table 4.4.

We believe that implementations based on MCTS could perform better with less conservative time constraints. Overall, we are pleased with the results.

# Appendix A

# User manual

This program is meant to be run by the LoCM referee with another agent as an opponent, but may also rival itself. It will expect to read input and print output as described by LoCM rules [5]. Detailed usage of referee can be found on the LoCM project website [17].

Installation requires Rust tool-chain at least in version `1.42.0-nightly`, it can be installed by following steps listed on rust-lang website [18].

To build agent, go to `agent` directory, run `cargo +nightly build --release` and then executable file will be available in `agent/target/release` named as `agent`.

Our agent provides command line interface that follows GNU guidelines. All available options are listed with usage of `--help` argument. The main functionality is the ability to choose the approach for draft and battle phase. Default options are random approaches for each phase and time constraints as described in the rules.

## Examples

```
agent --draft=eeval --battle=predict_mcts --first-draft-response=500 \
      --first-battle-response=500 --turn-response=100
```

This will pick cards based on randomised simulations and choose actions using Monte Carlo Tree Search with prediction of opponent's state and behaviour under given time constraints in milliseconds.

```
agent --draft=random --battle=aggressive
```

This will pick random cards and try to deal most damage in greedy fashion during battle phase under default time constraints.

```
agent -d external_weights -b external_predict_mcts -p empty
```

Given `DRAFT_WEIGHTS` and `WEIGHTS` environmental variables contain paths to files with weights, this will choose cards and evaluate card's value based on provided weights.

```
java -jar LoCM.jar \
     -p1 "agent -b aggressive" \
     -p2 "agent -d eeval -b predict_mcts" \
     -s
```

Provided LoCM's referee written in java, this will create service to watch the game in a web browser accessible under `http://localhost:8888/`.

# Appendix B

# Development overview

Agent is written in Rust language using nightly features. Rust was chosen because of its performance, reliability and productivity. Due to concept of ownership, Rust requires no run-time environment or garbage collection. Its ownership and type system guarantee memory safety and prevent occurrence of some types of bugs. Provided tool-chain and unofficial tools increases productivity for example by auto formatting code or providing hints to make code more idiomatic. Usage of external code is simplified by requiring to just add packages, called crates, to the configuration file. Those crates, in most cases, come with well-made documentation in uniform style.

## B.1 Project structure

Program is split mainly into seven modules and the main file.

- `algorithms`

  In our agents we use algorithms implemented in this module. These implementations are generic and therefore are not tied to any particular agent or game. Available algorithm implementations are `AlphaBeta` and two variations of `MCTS`.

- `battle`

  All of our battle agents and some helping functions are part of `battle` module. Six approaches (`aggressive`, `basic mcts`, `minmax`, `external predict mcts` and `predict mcts`, `random`, `weighted mcts`) are separated into sub-modules and implement generic trait that makes them interchangeable. Agents can hold state and each turn they create chain of actions given game state in a form of `Player` and `Opponent` structures, predictor and available time after which occurs timeout.

- `draft`

  This module mainly contains `eeval`, `closetai`, `hs` and `external weights` agents for drafting phase, but also includes `random` agent. Similarly to solution in `battle` module, drafting agents implement genetic trait that allows inner state and provides method to pick card given game state and available time for decision.

- `predictors`

  In more advanced battle agents, we use prediction of opponent's deck. For this purpose module `predictors` provides five approaches (`closetai`, `empty`, `same`, `simple`, `weighted simple`) and similarly to agents in `battle` and `draft` modules, predictors implement trait that allows updating information about opponent's actions and method that returns predicted deck.

- `main`

  This file contains `main` function that parses command line arguments with help of `options` module. Then based on those options runs loops for draft and battle phases, which are detached to separate generic functions.

- `model`

  This module contains all the structures needed to store state and simulate game. Game state is represented by `Player` and `Opponent` structures which contain specific information known about that side and `BasePlayer` structure that represents player more uniformly. Action simulation and listing of valid actions are performed on two instances of `BasePlayer` representing players. It is useful in exploring in-depth possible action of both players.

- `options`

  In this file we took advantage of `structopt` crate and defined options accessible through command line interface. Moreover, this struct provides methods that wrap functions from `main` file to provide types and values as specified in arguments.

- `parser`

  Parsing input is quite trivial as a result of simple scheme described in LoCM rules. This module packs input into structures that will be used in loop of each phase to update agent's knowledge of the game state.

## B.2 Course of development

Development started with Marcin creating initial project structure with beginning of the parsing and game state struct (i.a. `Board`, `Card`, `Player`). He also managed git repository, created scripts to force proper formatting (with `rustfmt`) and ensure that tests passed.

Wojciech finished the parsing module and added unit tests to cover parsing. Next step was to create random drafting agent with simple main loop to test already created code with referee.

Before creating any battle agent, Łukasz created function to list all legal actions given game state. This function was helpful writing battle agents and was covered by unit tests.

Our first battle agent was made by Wojciech, it simply returned shuffled vector of valid actions at the begging of the turn. At this point he also added structure to represent opponent's state and methods to update game state every turn.

For current representation of the game state Łukasz created simulation, given an action the game state is updated. Then Marcin implemented generic Monte Carlo Tree Search. That implementation and game simulation were first steps to create more advanced agents.

Using existing simulation, Łukasz rewrote random battle agent to pick one action at random and update game state in a loop until no more actions were possible. This helped test updating and simulating game state.

Wojciech refactored random battle agent and with some testing we spotted some inaccuracies in our implementation of the game, so he rewrote simulation of actions following source code of LoCM. As we needed to simulate actions in both ways he took `Board` and common values form `Player` and `Opponent`, then created `BasePlayer` struct, from this point player and opponent's structures hold specific values to them. After that, Wojciech implemented simple greedy agent to perform aggressive chains of actions. As we had multiple agents, we needed a way to choose which one should run, so he added command line interface.

As MCTS was our main idea from the beginning, Łukasz implemented draft agent based on similar principle that we called `eeval` (Endgame Evaluator). He also added methods to perform game logic between turns, as we would have needed that for simulating games.

Once our project was ready to simulate game, Marcin implemented `basic mcts` battle agent and two agents to generate opponent's cards on hand (`empty` and `simple` predictors).

Having our first non-trivial agent, Łukasz worked on improving heuristic and Wojciech added support for time constraints as MCTS yields better results

the longer it runs, but also has to stop before timeout.

After creation of basic agent with MCTS, Marcin implemented `predict mcts` battle agent and generic AlphaBeta algorithm. He also added auto-generation for cards list from file.

Due to addition of new agents, Łukasz reworked heuristics. As we had multiple predictors, Wojciech added support to choose them with command line interface. Wojciech also added function that tries in greedy fashion generate winning action chain, which is used to check if simple solution exists before running MCTS.

Marcin created `same` predictor, draft agent mimicking ClosetAI and two battle agents based on mcts (`weighted mcts` and `rave mcts`). He also implemented another generic algorithm, MinMax. Based on `closetai` draft agent, Łukasz created predictor.

At this point, we started writing this paper and generating proper statistical data. Marcin rewrote heuristics for basic and prediction based MCTS battle agents and created weighted predictor. Around that time, Marcin and Łukasz started training weights, so Marcin added version of `predict mcts` and off-line drafting that takes trained sets from a file. He also created `hs weights` drafting agent with our final set.

## Example

Example usage with LoCM Nim based referee [19]:

```
Tester --referee="java -jar LoCM.jar"                \
       --player1="agent -d closetai -b predict_mcts" \
       --player2="agent -d eeval -b basic_mcts"      \
       --games=2048                                  \
       --threads=16                                  \
       --plain=true
```

Keep in mind that if the given amount of threads is too big, it is highly possible that our agent will start to lose games by default because of timeouts. We implemented our more demanding evaluation function in such a way, that they run till certain amount of time passes, and they have only few *ms* of room. When threads are overloaded, they often fail to finish in that short time window. This is why it is recommended to run at most the same amount of threads as number of cores on testing machine. Since our algorithm evaluates moves for as long as possible, single game will run for about three to five seconds on any configuration.

# Bibliography


[1] Jakub Kowalski and Radosław Miernik. Legends of Code and Magic, 2020. https://jakubkowalski.tech/Projects/LOCM/.

[2] Legends of Code and Magic CEC June 2019. https://github.com/acatai/Strategy-Card-Game-AI-Competition/tree/master/contest-2019-06-CEC.

[3] Legends of Code and Magic COG August 2019. https://github.com/acatai/Strategy-Card-Game-AI-Competition/tree/master/contest-2019-08-COG.

[4] CodinGame Legends of Code and Magic contest. https://www.codingame.com/leaderboards/contests/legends-of-code-and-magic-marathon/global.

[5] Rules of Legends of Code and Magic. https://github.com/acatai/Strategy-Card-Game-AI-Competition/blob/master/GAME-RULES.md.

[6] Jakub Kowalski and Radosław Miernik. Evolutionary Approach to Collectible Card Game Arena Deckbuilding using Active Genes, 2020.

[7] Dockhorn, Alexander and Mostaghim, Sanaz. Introducing the Hearthstone-AI Competition, 05 2019.

[8] Amy K. Hoover and Julian Togelius and Scott Lee and Fernando de Mesentier Silva. The Many AI Challenges of Hearthstone, 2019.

[9] Maciej Świechowski, Tomasz Tajmajer, and Andrzej Janusz. Improving hearthstone ai by combining mcts and supervised learning algorithms, 2018.

[10] García-Sánchez, Pablo and Tonda, Alberto and Squillero, Giovanni and Mora, Antonio and Merelo Guervós, Juan. Evolutionary Deckbuilding in HearthStone. 09 2016.

[11] Alex Churchill, Stella Biderman, and Austin Herrick. Magic: The Gathering is Turing Complete, 2019.

[12] C.D. Ward and Peter Cowling. Monte Carlo search applied to card selection in Magic: The Gathering, 10 2009.

[13] Magic the Gathering Standard game rulings. `https://magic.wizards.com/en/content/standard-formats-magic-gathering`.

[14] Zong Woo Geem, Joong Kim, and G.V. Loganathan. A New Heuristic Optimization Algorithm: Harmony Search. *Simulation*, 76:60–68, 02 2001.

[15] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, page 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[16] Sylvain Gelly and David Silver. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 175(11):1856 – 1875, 2011.

[17] Repository of Legends of Code and Magic. `https://github.com/acatai/Strategy-Card-Game-AI-Competition`.

[18] Rust programming language - recommended installation steps. `https://www.rust-lang.org/tools/install`.

[19] Nim based Legends of Code and Magic referee. `https://github.com/acatai/Strategy-Card-Game-AI-Competition/tree/master/referee-nim`.