# Artificial Intelligence for Strategy Card Games: Search and Content Generation

(Sztuczna inteligencja w strategicznych grach karcianych:
przeszukiwanie i generowanie treści)

Radosław Miernik

Praca magisterska

**Promotor:** dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

19 marca 2021 r.

## Abstract

In this thesis, we presented a study regarding two important aspects of evolving feature-based game evaluation functions: the choice of genome representation and the choice of opponent used to test the model.

We compared three representations. One simpler and more limited, based on a vector of weights that are used in a linear combination of predefined game features. And two more complex, based on binary and n-ary trees.

On top of this experiment, we also investigated the influence of fitness defined as a simulation-based function that plays against the best individual from the previous generation and against a fixed opponent – either weak or strong.

For a testbed, we have chosen a recently popular domain of digital collectible card games. We encoded our experiments in a programming game, Legends of Code and Magic, used in Strategy Card Game AI Competition. However, as the problems stated are of general nature we are convinced that our observations are applicable in the other domains as well.

---

Przedmiotem tej pracy jest zbadanie dwóch ważnych aspektów ewolucji funkcji ewaluacyjnych, bazujących na kluczowych cechach stanu gry: wyboru reprezentacji genomu oraz wyboru przeciwnika do sprawdzenia samego modelu.

Porównaliśmy trzy reprezentacje. Jedną, prostą i ograniczoną, opartą na wektorze wag, które reprezentują kombinację liniową cech. Oraz dwie, bardziej złożone, oparte na drzewach binarnych i n-arnych.

Dodatkowo, sprawdziliśmy także wpływ funkcji oceniającej osobnika bazującej na symulacji, która gra przeciwko najlepszemu osobnikowi z poprzedniej generacji i takiej, która gra przeciwko ustalonemu przeciwnikowi – słabemu lub silnemu.

Jako środowisko do eksperymentów wybraliśmy popularną ostatnio dziedzinę cyfrowych kolekcjonerskich gier karcianych. Zakodowaliśmy nasze eksperymenty w grze programistycznej Legends of Code and Magic, używanej w Strategy Card Game AI Competition. Jednakże, przedstawione w tej pracy problemy mają ogólną naturę i jesteśmy przekonani, że nasze obserwacje mają przełożenie na inne dziedziny.

# Contents

# Chapter 1

# Introduction

The vast majority of game playing algorithms require some form of game state evaluation – usually in the form of a heuristic function that estimates the value or goodness of a position in a game tree. This applies not only to classic min-max based methods, as used for early Chess champions [1], but also in modern approaches, based on deep neural networks combined with reinforcement learning [2]. The idea of estimating the quality of a game state using a linear combination of human-defined features was proven effective a long time ago [3], and is still popular, even when applied to modern video games [4, 5].

One of the recently popular game-related AI testbeds are Collectible Card Games (e.g., *The Elder Scrolls: Legends* [6], *Hearthstone* [7]). Because they combine imperfect information, randomness, long-term planning, and massive action space, they impose many interesting challenges [8]. Also, the specific form of game state (combining global board features with features of individual cards) makes such games particularly suited for various forms of feature-based state evaluation utilizing human player knowledge.

Because of their complex characteristics, we have chosen the domain of digital collectible card games as a testbed. We encoded our models in a programming game, designed for AI research, Legends of Code and Magic. The game is used in various competitions, e.g., contest on the CodinGame platform and *Strategy Card Game AI Competition*, co-organized with IEEE CEC and IEEE COG conferences.

The motivation for our research was to compare two models for evolving game state evaluation functions: a simpler and more limited, based on a vector of weights that are used in a linear combination of predefined game features; a more complex and non-linear, based on a tree representation with feature values in leaves and mathematical operations in nodes. Our expectation is that the different capabilities of these models will significantly impact the evolution process and the overall individual performance.

During the initial experiments, we observed some interesting behaviors of both representations regarding “forgetting” previously learned knowledge. As these observations were related to the chosen goal of evolution, we performed additional tests comparing three fitness functions: playing against a fixed weak opponent, playing against a fixed strong opponent, and playing against the best individual from the previous population.

This thesis is structured as follows. In the next chapter, we present the related work describing usages of evolutionary algorithms for game state evaluation and the domain of AI for collectible card games. The following chapter is dedicated to Legends of Code and Magic – game description, its motivation, related work, and the state of agents development. In the following chapter, we provide the details of our model describing the representations and fitness functions. The following two chapters present our experiments and discuss the results in the above-mentioned topics: comparing tree versus vector representations and fixed opponent versus progressive fitness calculation. In the last chapter, we conclude our research and give perspectives for future work.

# Chapter 2

# Background

## 2.1 Evolving evaluation functions

Evolutionary algorithms are used for game playing in two main contexts. One, based on the Rolling Horizon algorithm (RHEA) [9], utilizes evolution as an open-loop game tree search algorithm. The other, a more classic approach, employs evolution offline, to learn parameters of some model, usually a game state evaluation function. Such function can be further used as a heuristic in alpha-beta pruning, RHEA, some variants of MCTS [10], and other algorithms. The quality of the evaluation function directly translates into the agent's power.

The common approach is to define a list of game state features and evolve a vector of associated weights, so that they closely approximate the probability of winning the game from the given state. This type of parameter-learning evolution has been applied to numerous board games, including Chess [11] and Checkers [12] as well as digital games, like Hearthstone [13] or TORCS [14].

Although treating parameters as vectors and evolving their values using genetic algorithms is more straightforward, some research uses tree structures and genetic programming instead for this purpose. The idea behind it is that the capacity of a linear (vector) representation may be too limited to encode certain problems.

The majority of genetic programming applications in games are an evolution of standalone agents instead of evaluation functions only. It was successfully applied in various board games, e.g., Chess [15, 16] and Reversi [17].

It is also possible to combine genetic programming with other algorithms and techniques. One example is to combine it with neural networks to evolve Checkers agents [18]. Another is to evolve the evaluation function alone and combine it with an existing algorithm, e.g., beam search, minmax or MCTS, instead of evolving fully-featured agents. Such an evolution was successfully applied in games of varying complexity, e.g., Checkers [19] and Chess [20].

Another aspect is measuring the quality of an evaluation function. Because it has to compare the strength of the agents, usually a simulation-based approach is used, combined with a large number of repetitions to ensure the stability of obtained results. Thus, such an evolution scheme is very computationally expensive.

## 2.2 AI for collectible card games

*Collectible Card Game* (CCG) is a broad genre of both board and digital games. Although the mechanics differ between games, basic rules are usually similar. First, two players with their *decks* draw an initial set of cards into their *hands*. Then, the main game starts in a turn-based manner. A single *turn* consists of a few *actions*, like playing a card from the hand or using an onboard card. The game ends as soon as one of the players wins, most often by getting his opponent's health to zero.

Recently the domain has become popular as an AI testbed, resulting in a number of competitions [21, 22, 23], and publications focusing on agent development, deckbuilding, and game balancing.

Usually, agents are based on the Monte Carlo Tree Search algorithm [10] (as it is known to perform well in noisy environments with imperfect information), combined with some form of state evaluation either based on expert knowledge and heuristics as in [13], or neural networks [24]. An interesting approach combining MCTS with supervised learning of neural networks to learn the game state representation based on the word embeddings [25] of the actual card descriptions is described in [26].

The deckbuilding task for the *constructed* game mode has been tackled in several works. The most common approach is to use evolution combined with testing against a small number of predefined human-made opponent decks, often ones designed by experienced players. It was successfully applied in Magic: The Gathering [27] and Hearthstone [28, 29]. Alternatively, a neural network-based approach for Hearthstone has been presented in [30].

Balancing, in the context of collectible card games, usually means slight modifications of card statistics (e.g., attack, health, cost) to prevent overpowered decks. MAP-Elites with Sliding Boundaries algorithm has been proposed for this task [31]. The study in [32] proposes multi-objective evolution, trying to minimize the magnitude of changes.

Other attempts focus on understanding the game while it is played. One can try to predict cards that the opponent is likely to play during a game [33], or predict a probability of winning as in [34].

The biggest problem while developing new tools for CCGs is their scale. While there are *only* 160 playable cards in Legends of Code and Magic, real CCGs can have far more: over 1200 in The Elder Scrolls: Legends and over 3600 in Hearthstone. Of course, both games are still adding more cards, pushing the problem even further.

## 2.3 Deck archetypes

There are numerous websites with example decks and according tactics, often maintained by the community[1][2][3]. While most of the tactics emerge directly from the mechanics, the card cost – present in all CCGs – is the most influential factor.

Of course, differences in the average card cost imply completely different tactics, as the number of cards playable within one turn changes as well. Such a distinction leads to a concept of *deck archetypes*:

- *Aggro*, composed of cheap cards, mostly creatures. The main objective is to win as quick as possible. The tactic is to fill the board with creatures and attack before the opponent will be able to build up the advantage.
- *Control*, composed of items (or spells) and expensive but strong creatures. The goal is to endure as long as possible while constantly cleaning the board by removing enemy creatures. This tactic may not lead to a standard victory but rather to exhaust the opponent instead.
- *Midrange*, composed of average-cost cards, a balanced mix of creatures and items. It is somewhere in between the previous two – does not rush as much as *Aggro* and is not as durable as *Control*.

A rough approximation of the deck archetype can be made by looking at the *mana curve*. The mana curve is a histogram of the costs of all cards in a deck. Example mana curves, presenting all three archetypes are presented in Fig. 2.1.

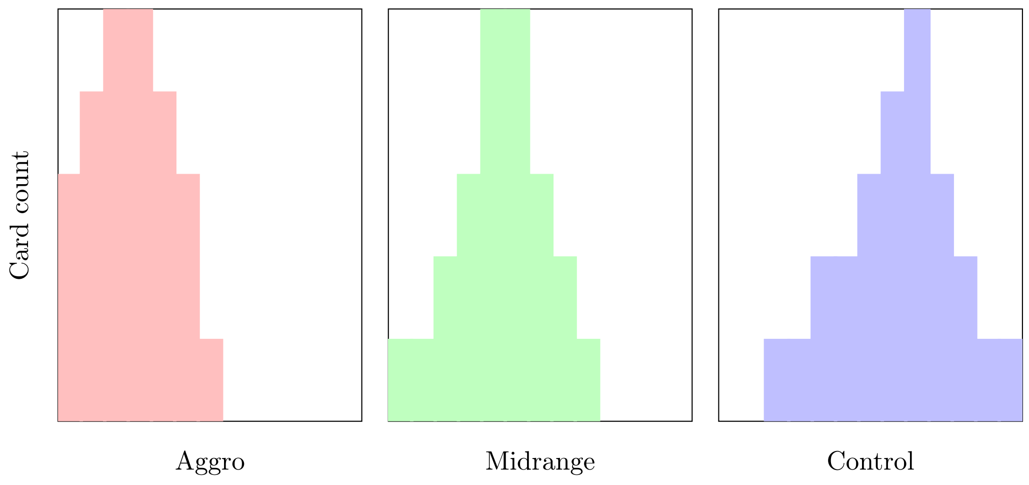

Figure 2.1: Example mana curves of all three deck archetypes. With the card cost on x-axis and card count on y-axis we plot a *curve* of the card cost. The mass of this curve suggest, whether this deck is focusing on early-, mid-, or late-game.

[1]`https://hearthstonetopdecks.com`
[2]`https://hsreplay.net/decks`
[3]`https://teslegends.pro`

# Chapter 3

# Legends of Code and Magic

## 3.1 Motivation

*Legends of Code and Magic* (LoCM) [23] is a CCG designed with AI research in mind, co-authored by the author of this thesis. In comparison to human-playable CCGs, like *The Elder Scrolls: Legends* [6] or *Hearthstone* [7], it is much simpler, and all player actions are fully deterministic. Such a limitation makes the implementation of AI agents easier, allowing the researchers to focus on experimenting.

A natural process present in all games, especially multi player ones, is the formulation of a so-called *meta* of the game or just *metagame*. In CCGs context, most often it is a couple of deck templates with an accompanying tactic, designed to exploit a certain mechanic in order to achieve high win rates. Less impactful but also present is a list of objectively good and bad cards, usually along with their cheaper or better counterparts. While the metagame itself is not inherently bad, it directly impacts the community and leads to sterilisation of the in-game content.

The majority of real-world CCGs have multiple game modes. However, all of them boil down to either a *constructed deck* or *arena* scheme. In the former, players construct their decks in between the games, using all of the cards they have collected so far. In the latter, the deck is constructed once per game or a series of games, using a predefined subset of cards – often by picking one of three cards $N$ times.

As expected, being able to play with an ad-hoc deck is a strictly harder task. On one hand, it implies greatly reduced impact of existing meta, as the player does not know, what cards he will be playing with. On the other, it forces him to be able to play with all of the available cards – even those outside of meta.

For these two reasons, LoCM relies on a novelty variant of the arena game mode, called *fair arena*. In contrast to arena modes in real CCGs, both players compose their decks once per game, choosing cards from the same options. This eliminates the problem where one player got strictly better choices during deck composition. Additionally, the simultaneous choices imply equal knowledge about the draft.

## 3.2 Gameplay

As most CCGs, LoCM is a turn-based, multi-action game. All in-game mechanics as well as the cards themselves are heavily inspired by *The Elder Scrolls: Legends*. Every game consists of two phases – 30 turns of the *draft* phase, followed by the *battle* phase. The game ends as soon as any player's health reaches zero or less. If both players' health reaches zero or less at the same time, the active player wins.

### 3.2.1 Draft phase

During a draft phase turn, players are presented with a choice of three cards. They can perform only one action: `PICK N`, $N \in \{0, 1, 2\}$. Additionally, there is the `PASS` action – an alias for `PICK 1`. The game itself does not provide any information about the already selected cards – if the agent would like to analyze its own choices, it has to do the bookkeeping by itself.

As the card choices are the same for both players, once the battle phase starts, one could track the statistics of possibilities of the opponent's cards. For example, if cards $A$ and $B$ appeared within one choice and only once, as soon as the opponent played card $A$, we can be sure, that he does not own card $B$.

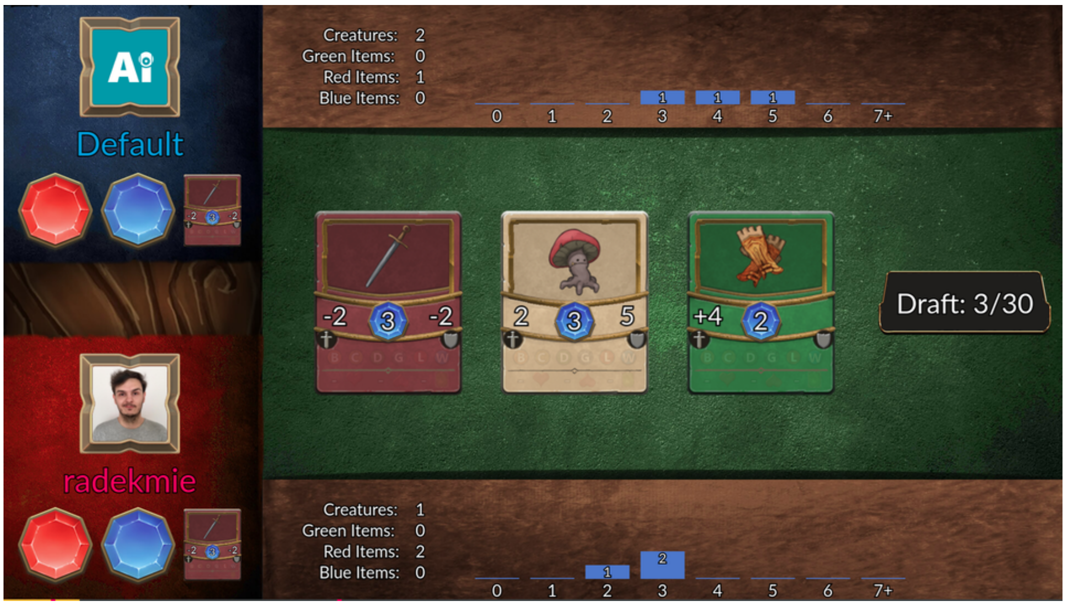

Figure 3.1: In-game visualization of the draft phase.

The in-game UI of the draft phase, presented in Fig. 3.1, looks as follows. Basic players' information, namely avatar, name and their selected card, are on the left sidebar. Available card choices and the turn indicator are in the center. Finally, next to the player's avatar there is the number of cards grouped by type and a so-called *mana curve* – a histogram of cards' costs.

### 3.2.2 Battle phase

Both players start with 30 health, 1 max. mana, 4 cards in hand, and 5 runes. As in most CCGs, the second player gets a bonus to minimize the impact of being second. He draws one additional card at the beginning and has a +1 max. mana bonus until he spends all of his mana within one turn.

The runes correspond to 25, 20, 15, 10, and 5 health thresholds respectively. The first time a player's health drops below a threshold, the rune breaks and results in one additional card draw during the next turn. This mechanism helps the player in a difficult situation by giving more cards and hence more possibilities.

Every turn starts with an increase of the maximum mana (up to 12) and a card draw. The current player draws one card and one more for each *rune* lost during the previous turn. If the player already has 8 cards in hand, the draw is cancelled. If there are no cards to draw, the player looses a rune instead, and his health is reduced to the rune threshold. With no runes left, player's health is reduced to zero. After 50 turns both decks are considered empty.

The player can perform multiple actions during their turn. It is worth noting, that the determinism of actions implies no interactions between players within one turn. (It is not true in real CCGs, e.g., *Prophecy* in TES:L.) Possible actions are:

- `SUMMON ID LANE` summons the creature with `instanceId` of `ID` into `LANE` lane.
- `ATTACK ID1 -1` attacks the opponent with creature `ID1`.
- `ATTACK ID1 ID2` attacks creature `ID2` with creature `ID1`.
- `USE ID1 -1` uses item `ID1` on self.
- `USE ID1 ID2` uses item `ID1` on creature `ID2`.
- `PASS` does nothing.

The `ATTACK` action happens simultaneously, i.e., both creatures deal and receive damage at once. When a creature's health reaches zero or less, it is removed from the board. Creature keywords may affect the attack resolution (see 3.3).

In order to ease the implementation of agents for new players, all invalid actions are ignored, as long as they are correctly formatted. Additionally, player may append a custom message at the end of action – it will be displayed in the UI and has no other meaning. Actions can be invalid for several reasons:

- There is no card with `instanceId` of `ID`.
- `LANE` is not a valid lane.
- `LANE` lane is full (at most 3 creatures).
- Player has not enough mana to play a card.
- Creature already attacked.
- Creature is in a different lane.
- Creature with a `G` keyword is blocking the attack (see 3.3).
- Creature was summoned this turn and has no `C` keyword (see 3.3).

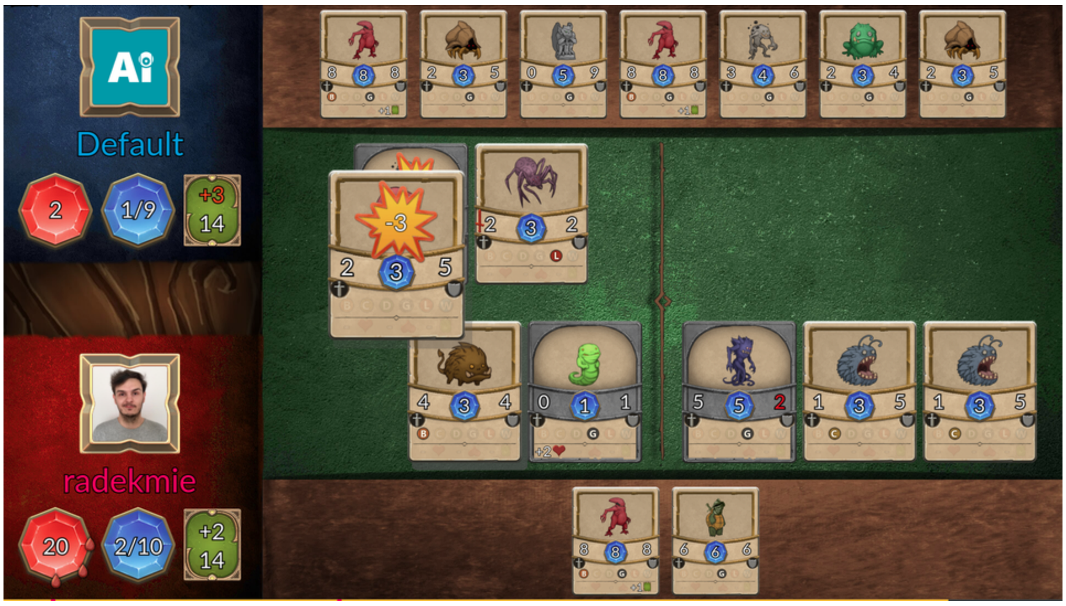

Figure 3.2: In-game visualization of the battle phase.

The in-game UI of the battle phase, presented in Fig. 3.2, looks as follows. Complete players' information, namely avatar, name, health, remaining runes, current and maximum mana, next turn draft, and deck size, are on the left sidebar. Summoned creatures are in the center, split into two lanes. Finally, next to the player's avatar is the player's hand.

## 3.3 Cards

There are two card types – *creatures* and *items*. All cards have three primary stats, three secondary stats, and a number of *keywords*. Primary stats are attack, mana cost, and defense. Secondary stats are player's health gain, damage dealt to the opponent, and additional draw for the next turn. While the secondary stats are resolved when the card is played, primary stats' meaning differs, based on card type.

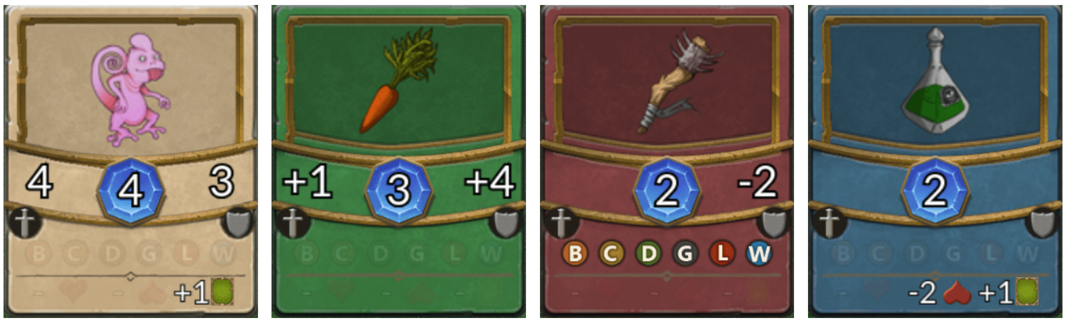

Figure 3.3: In-game cards visualization. From left: *Chameleskulk* (creature), *Healthy Veggies* (green item), *Helm Crusher* (red item), and *Poison* (blue item).

Creature cards can be placed on board ( `SUMMON` ) and attack the opponent as well as his creatures once per turn ( `ATTACK` ). Each creature on board can have any number of keywords, changing their attack resolution. Available keywords are:

- `B` reakthrough. Creatures with this keyword deal the excessive damage to the opponent, i.e., if their attack was higher than the target creature defense.
- `C` harge. Creatures with this keyword can attack in the same turn they were summoned. By default, creatures can attack only in the following turns.
- `D` rain. Creatures with this keyword heal the player when attacking for the amount of damage they deal (not their attack).
- `G` uard. Enemy creatures in the same lane must attack these creatures first.
- `L` ethal. Creatures with this keyword kill target creatures they deal damage to.
- `W` ard. Creatures with this keyword ignore the first damage they would receive. After receiving any damage, the keyword is lost. It does block `B` , `D` , and `L` .

Item cards can be used on creatures on board ( `USE` ). Positive effects increase target's – creature or player – stats and give them keywords, whereas negative effects decrease target's stats and remove their keywords. There are three item types:

- *Green*, targeting player's creatures with a positive effect.
- *Red*, targeting opponent's creatures with a negative effect.
- *Blue*, targeting opponent or his creatures with a negative effect.

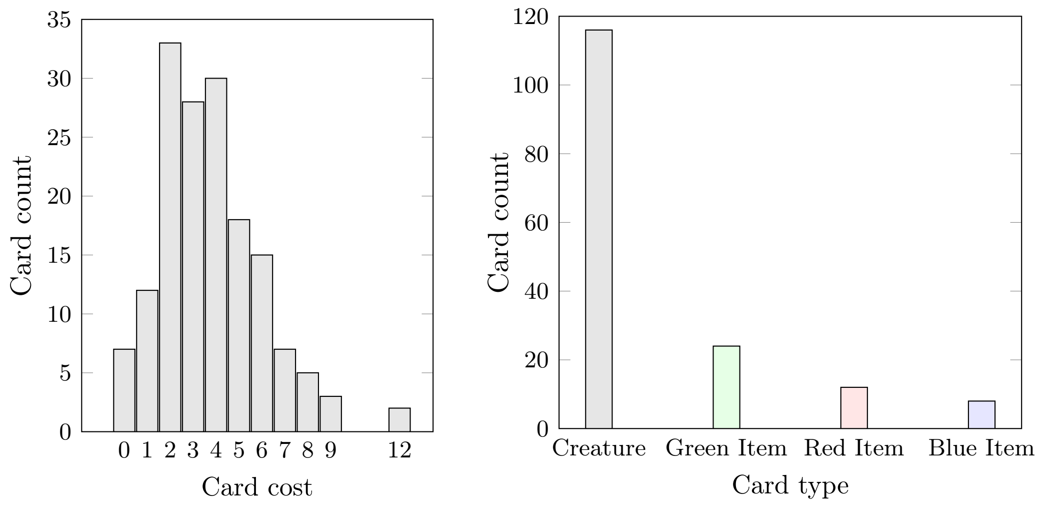

Figure 3.4: Card statistics. Most cards cost between 2 and 4 mana, with an average of 3.76. Out of 160 cards, there are 116 creatures, 24 green, 12 red, and 8 blue items.

## 3.4 Related work

As for today, LoCM was used as a testbed for a couple of research papers. These include agents utilizing reinforced learning [35], an evolutionary approach to deckbuilding [36] and tuning parameters of heuristic functions [37].

Separately, LoCM is used in the *Strategy Card Game AI Competition*. The competition is co-organized with IEEE Congress of Evolutionary Computation and IEEE Conference on Games conferences since 2019. Both competitions received 9 submissions (agents) in 2019, and 4 new or upgraded in 2020. The winner of the 2020 IEEE COG competition (`Chad`) has been described in [38].

Interestingly, there is no *standard* approach for a LoCM agent yet. The best agent in 2019 (`Coac`) used a handcrafted draft evaluation function and a minmax-like search with alpha pruning of depth 3 for the battle phase. The best agent in 2020 (`Chad`) used weights computed using harmony search for the draft and a MCTS with prediction of the opponent's hand for the battle phase. Other agents most commonly involved handcrafted heuristics but there were submissions utilizing neural networks (`ReinforcedGreediness`) or *almost* full one-turn deep search (`OneLaneIsEnough`).

## 3.5 Version 1.0

The above description refers to the current version (1.2) of LoCM. However, the first publicly available version of the game was slightly different. The first version (1.0) was used in August 2018 as a CodinGame[1] platform contest, attracting more than 2,000 players (or rather AI programmers) across the world [39].

The 1.0 version had only one lane of size 6 instead of two lanes of size 3. This not only changes the size and shape of the game tree but also the impact of all keywords. For example, the `G` keyword is now less valuable, as a single card no longer protects the player fully. Similarly, cards with the `L` keyword have fewer targets.

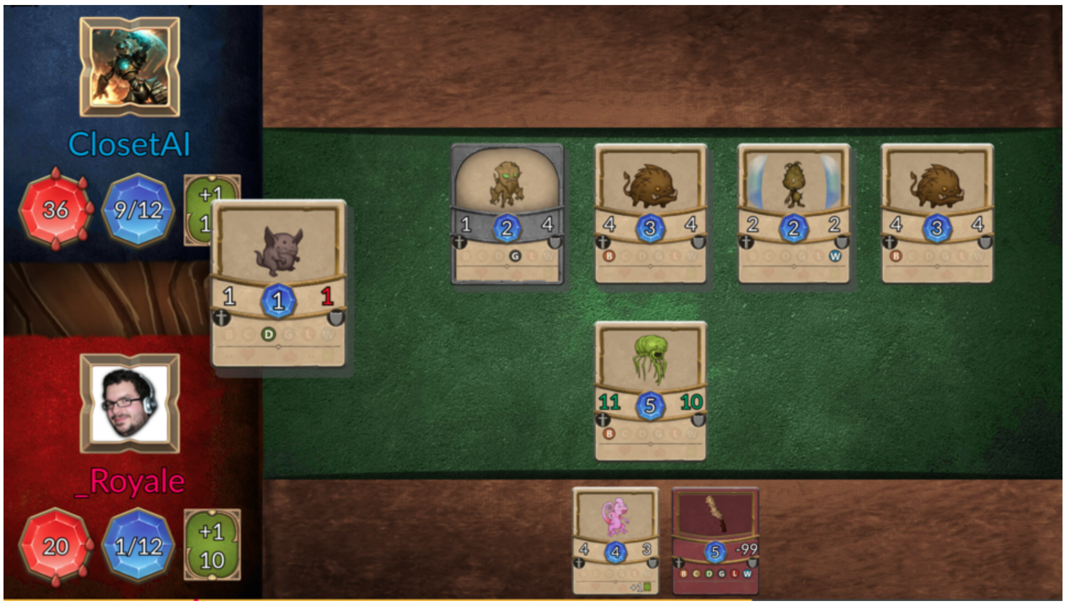

Figure 3.5: In-game visualization of the battle phase, LoCM 1.0.

[1] `https://codingame.com`

# Chapter 4

# Evaluating Card Games

## 4.1 Representation

We have developed three distinct representations: `Linear`, `BinaryTree`, and `Tree`. Each one implemented two operations: `evalCard` (used for the draft phase and as a part of the state evaluation) and `evalState` (used for the battle phase).

- `Linear`, is a constant-size vector of doubles. Each gene (from 1 to 20) encodes a weight of the corresponding feature. The first 12 are game state features, 6 for each of two players: current mana, deck size, health, max. mana, next turn draw, and next rune (an indicator for an additional draw as in [6]). The other 8 are card features: attack, defense, and a flag for each of the keywords, encoded as 1.0 when set and 0.0 when not. The final evaluation is a sum of features multiplied by their corresponding weights.
- `BinaryTree`, is a pair of binary trees, encoding state and card evaluation respectively. The leaf nodes are either constants (singular double) or features, same as in `Linear`. Both trees have the same set of binary operators (nodes): addition ($l + r$, where $l$ and $r$ are the values of left and right subtree respectively), multiplication ($l * r$), subtraction ($l - r$), maximum ($max(l, r)$), and minimum ($min(l, r)$). The final state evaluation is calculated recursively accumulating the tree.
- `Tree`, is a pair of n-ary trees, encoding state and card evaluation respectively. The leaf nodes are identical to the ones in `BinaryTree`. Operators are no longer binary, but n-ary instead – each operator stores a vector of subtrees. Available operators are addition ($\sum$), multiplication ($\prod$), maximum ($max$), and minimum ($min$). Additionally, to ensure that every operation stays well defined, all subtree vectors are guaranteed to be nonempty. To make subtraction possible, there is one additional, unary operator: negation ($-x$, where $x$ is the value of its subtree). The final evaluation is calculated recursively accumulating the tree.

To limit the vast space of possible evaluation functions, the final state evaluation is a sum of `evalState` and `evalCard` for each own card on the board, minus `evalCard` for each opponent card. It is a common simplification, used in e.g., [13].

Note that the expressiveness of the tree-based representations is greater, thus it is possible to map individuals encoded as `Linear` to trees, but not vice versa. This property is used in one of our experiments.

## 4.2 Opponent estimation

Every evolution scheme evaluates individuals either by comparing how well they deal with a specified task, without a normalized score, or by using an external, predefined goal. Both approaches have natural interpretations for CCGs – a win rate against each other for the former and a win rate against a fixed opponent for the latter.

As the course of evolution using a predefined opponent will be heavily impacted by the opponent itself, two nontrivial questions arise. What is the difference between using only the in-population evaluation from the one using a predefined opponent? And what is the difference between using a weak and a strong opponent? We have conducted two experiments to answer both of these questions.

As both experiments required different evolution schemes, in total, three groups of individuals were evolved. First one, called `progressive`, using the in-population evaluation (see 5.1). Second, called `weak-op`, using the existing LoCM baseline agent `Baseline2` (`WeakOp`). And third, called `strong-op`, using one of the pre-evolved `Tree-from-Linear` agents (`StrongOp`). The last two are described in 6.1.

Chapter 5

# Representation Study: Trees Versus Vectors

To measure the impact of different representations on the evolution, we ran the same experiment multiple times, each time substituting the underlying genome structure to one of described in 4.1. The metrics we find crucial are how well the agent performs in a real-life scenario, that is, in a proper tournament with other agents, and whether it progresses, i.e., plays better against own previous generations.

## 5.1 Experiment setup

We have evolved twelve copies of `progressive` agents, four for each representation. Every run used the same parameters, that is 50 generations with a population of size 50 (*population* parameter), elitism of 5 individuals, and the mutation rate of 5%. During the evaluation each two individuals played *rounds* times on each of *drafts* drafts on each side, which makes $2 \times (\textit{population} - 1) \times \textit{drafts} \times \textit{rounds}$ games in total. In our experiments, $\textit{drafts} = 10$, and $\textit{rounds} = 10$.

In order to compare the agents in a standardized real-life scenario, we ran a tournament, replicating the Strategy Card Game AI Competition. In addition to our evolved agents, we used two LoCM baseline agents – `Baseline1` and `Baseline2` – and four contestants of 2020 IEEE COG LoCM contest[1] – `Chad`, `Coac`, `OneLaneIsEnough`, and `ReinforcedGreediness`.

[1]`https://legendsofcodeandmagic.com/COG20/`

## 5.2 Learning comparison

As visible at the bottom row of Fig. 5.1, all representations successfully managed to converge into a green triangle at the bottom left of the heatmap.

Such shape means that the following generations were not only preserving the already gained knowledge but also slightly improving on each step. Therefore, every representation can be evolved, playing against own previous generations. Moreover, the progress of the `Linear` representation seems to be more stable, almost constant, while tree-based representations tend to improve by making larger but sporadic leaps.

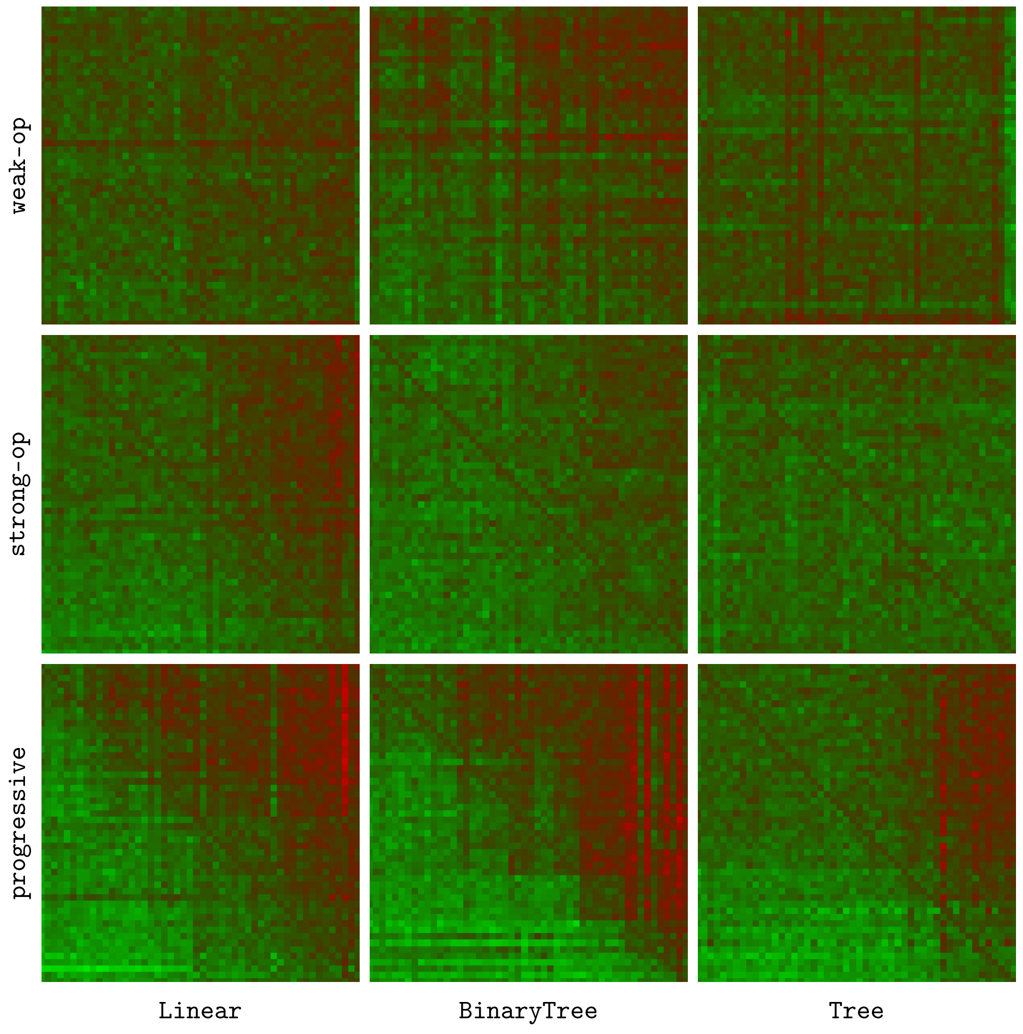

Figure 5.1: Example self-play win rate heatmaps (other runs show similar properties). Each cell represents how well the best individual of generation on the y-axis plays against the best individual of generation on the x-axis. A bottom left green triangle, present in all three `progressive` agents, proves that as the evolution progresses, so all individuals are getting constantly better at self-play. Other heatmaps seem to be more random, indicating no clear progression in self-play.

This is not the case for the top and middle row, representing evolution with a predefined opponent (described in detail in the next section). While the `Linear` representation manages to preserve hardly visible progress, both tree representations are more random, indicating no clear improvements in self-play. Furthermore, the top row contains a few red stripes, indicating an exceptionally weak agent. It is possible, as the evolution goal does not take self-play into consideration at all.

To measure how significant the learning progress is, we compare the win rate of the best individual of the first and the last generation against the best individuals of all generations.

With such a metric in mind, the `Linear` representation stands out again. As presented in Fig. 5.2, both `BinaryTree` and `Tree` result in a smaller difference of about 20% whereas the `Linear` representation achieves over 31% difference on average, across all the evolution schemes.

## 5.3 Tournament comparison

When evolved using a fixed, weak opponent (`weak-op`), the difference between the representations is significant. There is a huge, almost 10% wide, gap between `Linear` (42.9% average win rate) and `BinaryTree` (33.8%). The `Tree` representation is in between, performing slightly better than its binary counterpart and achieving 36.2%.

Results are similar when evolved using self-play evaluation and a randomly initialized population (`progressive`). Again, the gradually improving `Linear` representation yields strictly better results in the tournament (54.8%) than both `BinaryTree` and `Tree` (45.6% and 45.7% respectively).

However, using a better opponent (`strong-op`) yields completely different results. In this scenario, the differences between agents' performance are far less significant, around 4%. To be precise, `Linear` achieved almost 49%, `BinaryTree` slightly over 45%, and `Tree` nearly 45% average win rate.

This matches the results of playing against own previous generations, described in the previous section. We conclude that it is an implication of the capacity of the representation. It also matches the general expectations, that playing against stronger opponents leads to stronger agents.

While a more limited `Linear` representation finds a decent solution sooner and steadily improves it, more capable tree-based representations regularly leap towards the goal. More detailed tournament results are presented in Table 5.1.

Table 5.1: The tournament results. Every score is the win rate (%) of the agent on the left against the agent on top.

| | `Baseline1` | `Baseline2` | `Chad` | `Coac` | `OneLaneIsEnough` | `ReinforcedGreediness` | `BinaryTree-baseline` | `BinaryTree-frombest` | `BinaryTree-standard` | `Linear-baseline` | `Linear-frombest` | `Linear-standard` | `Tree-baseline` | `Tree-frombest` | `Tree-standard` | `Linear-from-Linear-standard` | `Tree-from-Linear-standard` | **Global average** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `Baseline1` | – | 40.28 | 2.78 | 0.00 | 6.94 | 2.78 | 43.40 | 32.50 | 38.33 | 45.83 | 29.44 | 16.11 | 36.97 | 34.37 | 37.32 | 16.99 | 21.69 | 28.89 |
| `Baseline2` | 59.72 | – | 4.17 | 0.00 | 0.00 | 2.78 | 34.03 | 45.56 | 57.50 | 17.01 | 48.61 | 35.56 | 43.98 | 53.24 | 63.53 | 39.89 | 51.47 | 42.73 |
| `Chad` | 97.22 | 95.83 | – | 31.94 | 59.72 | 100.00 | 83.33 | 86.11 | 89.44 | 79.86 | 86.11 | 74.17 | 82.35 | 88.17 | 87.18 | 81.75 | 83.09 | 83.18 |
| `Coac` | 100.00 | 100.00 | 68.06 | – | 55.56 | 69.44 | 89.93 | 91.39 | 92.22 | 84.03 | 90.28 | 90.83 | 93.28 | 95.21 | 92.31 | 90.88 | 93.20 | 90.42 |
| `OneLaneIsEnough` | 93.06 | 100.00 | 40.28 | 44.44 | – | 59.72 | 53.82 | 64.17 | 65.83 | 55.21 | 60.00 | 60.56 | 57.42 | 66.20 | 67.52 | 60.64 | 63.24 | 62.07 |
| `ReinforcedGreediness` | 97.22 | 97.22 | 0.00 | 30.56 | 40.28 | – | 88.89 | 94.17 | 88.89 | 89.24 | 89.17 | 84.72 | 89.08 | 89.01 | 90.60 | 82.83 | 85.85 | 85.23 |
| `BinaryTree-baseline` | 56.60 | 65.97 | 16.67 | 10.07 | 46.18 | 11.11 | – | 35.07 | 38.89 | 43.06 | 36.81 | 26.53 | 49.30 | 36.69 | 33.19 | 23.43 | 27.11 | 33.94 |
| `BinaryTree-frombest` | 67.50 | 54.44 | 13.89 | 8.61 | 35.83 | 5.83 | 64.93 | – | 47.00 | 57.57 | 47.83 | 40.39 | 56.97 | 49.92 | 46.04 | 38.57 | 35.22 | 45.20 |
| `BinaryTree-standard` | 61.67 | 42.50 | 10.56 | 7.78 | 34.17 | 11.11 | 61.11 | 53.00 | – | 48.26 | 45.39 | 40.28 | 61.85 | 50.93 | 50.11 | 40.00 | 37.68 | 45.63 |
| `Linear-baseline` | 54.17 | 82.99 | 20.14 | 15.97 | 44.79 | 10.76 | 56.94 | 42.43 | 51.74 | – | 51.46 | 33.26 | 47.55 | 49.30 | 44.71 | 31.14 | 39.06 | 43.02 |
| `Linear-frombest` | 70.56 | 51.39 | 13.89 | 9.72 | 40.00 | 10.83 | 63.19 | 52.17 | 54.61 | 48.54 | – | 46.50 | 62.77 | 52.90 | 56.06 | 43.15 | 40.29 | 48.99 |
| `Linear-standard` | 83.89 | 64.44 | 25.83 | 9.17 | 39.44 | 15.28 | 73.47 | 59.61 | 59.72 | 66.74 | 53.50 | – | 66.69 | 61.69 | 61.26 | 43.58 | 41.07 | 54.92 |
| `Tree-baseline` | 63.03 | 56.02 | 17.65 | 6.72 | 42.58 | 10.92 | 50.70 | 43.03 | 38.15 | 52.45 | 37.23 | 33.31 | – | 42.17 | 36.88 | 28.28 | 23.29 | 36.40 |
| `Tree-frombest` | 65.63 | 46.76 | 11.83 | 4.79 | 33.80 | 10.99 | 63.31 | 50.08 | 49.07 | 50.70 | 47.10 | 38.31 | 57.83 | – | 51.25 | 38.04 | 36.38 | 44.96 |
| `Tree-standard` | 62.68 | 36.47 | 12.82 | 7.69 | 32.48 | 9.40 | 66.81 | 53.96 | 49.89 | 55.29 | 43.94 | 38.74 | 63.12 | 48.75 | – | 39.56 | 35.27 | 45.76 |
| `Linear-from-Linear-standard` | 83.01 | 60.11 | 18.25 | 9.12 | 39.36 | 17.17 | 76.57 | 61.43 | 60.00 | 68.86 | 56.85 | 56.42 | 71.72 | 61.96 | 60.44 | – | 44.36 | 58.12 |
| `Tree-from-Linear-standard` | 78.31 | 48.53 | 16.91 | 6.80 | 36.76 | 14.15 | 72.89 | 64.78 | 62.32 | 60.94 | 59.71 | 58.93 | 76.71 | 63.62 | 64.73 | 55.64 | – | 60.27 |

## 5.4 Tuning good solutions

Knowing that the tree-based model is more general but also harder to learn, the natural question is if we can use it to improve the solutions, instead of generating them from scratch. The ideal scenario will be to reach a limit of optimization based on the linear representation, encode obtained solutions into the tree format, and continue evolution using this stronger model.

To verify this hypothesis, we have evolved two more agents: `Linear-from-Linear` and `Tree-from-Linear`. Each of the four previously evolved `Linear-progressive` agents was used as a base for a second evolution process.

Now, rather than randomly, the population was initialized with copies of the base agent, each one mutated $n = 5$ times; hence the `from-Linear` suffix in their name.

However, translation of a `Linear` representation into a `Tree` representation is ambiguous. We have used what we believe is the most straightforward one – an `Add` operator in the root with a list of `Mul` + `Literal` + `Feature` subtrees, one for each of the available features.

As expected, the tournament results for such pre-evolved agents are entirely different. For the first time, the `Linear` representation is not the top one. The `Tree` agent performs better, ending up with an average win rate of over 60%, whereas the `Linear` agent finished with 58%. This is a relatively minor but consistent improvement that may further improve with a longer evolution.

A similar difference is visible in the process of evolution. Both representations perform similarly, but `Tree` is on average above the `Linear` almost constantly. Comparison of the best individuals across the generations is presented in Fig. 5.3.

Our conclusion is that both results are implications of the representation. While the more restricted `Linear` is not able to progress after a certain point, more expressive `Tree` benefits from the bootstrap and keeps improving. Once again, more detailed results are presented in Table 5.1.

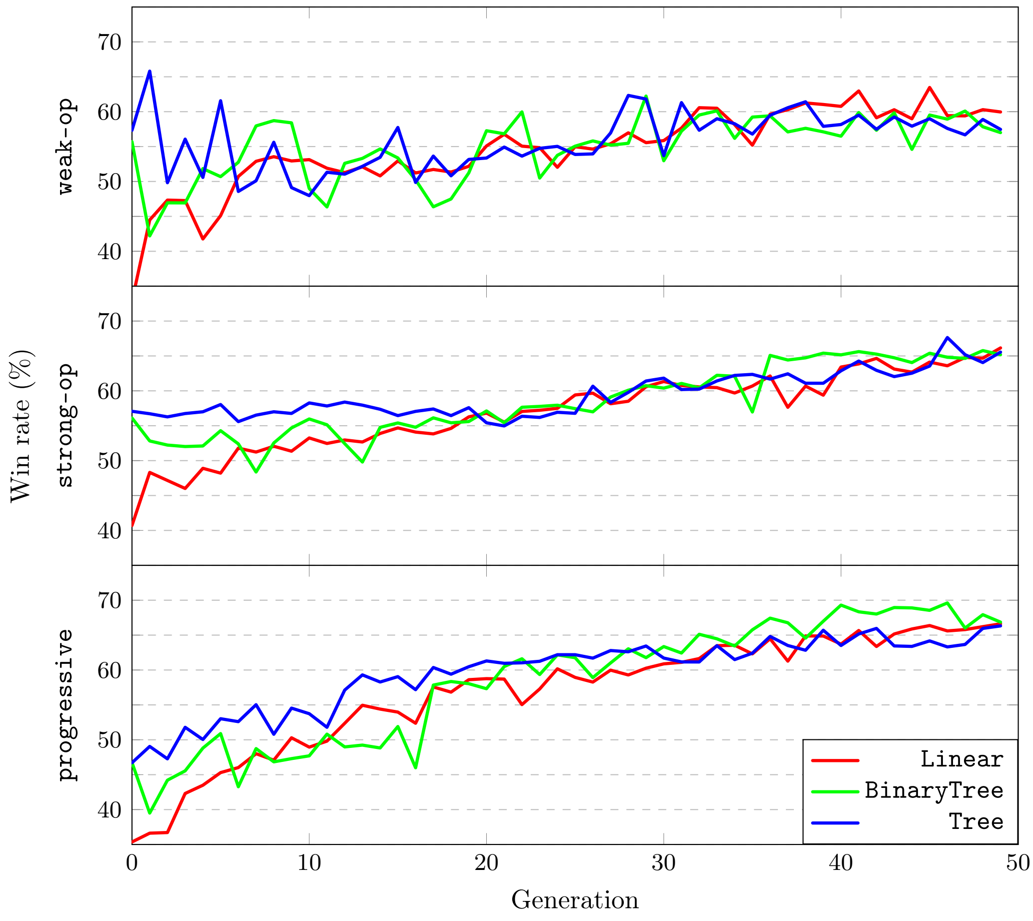

Figure 5.2: Evolution progress of all the agents. Best individuals from a generation (x-axis) fought against the top individuals of all own generations, yielding an average win rate (y-axis).

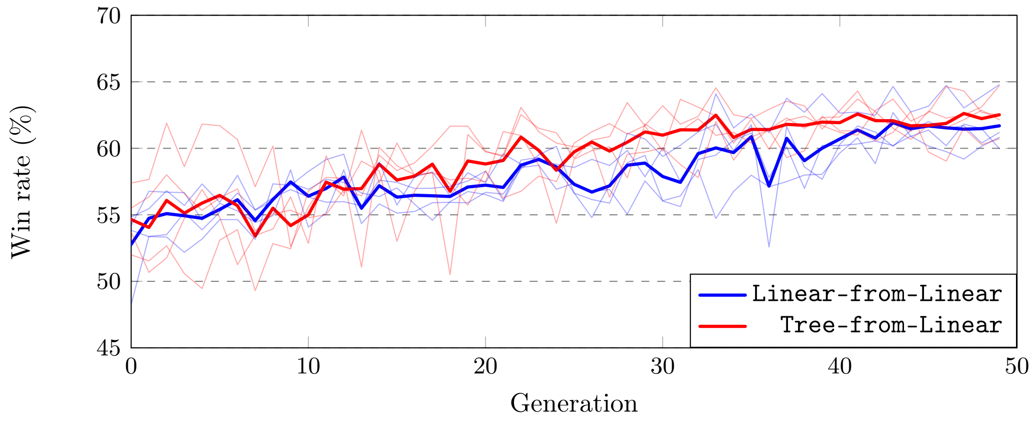

Figure 5.3: Evolution progress of the `from-Linear` agents. Best individuals from a generation (x-axis) fought against the top individuals of all own generations, yielding an average win rate (y-axis). Each of the four `Linear` agents was used as a base for the initial population twice. The two bold lines average the thin, semi-transparent lines that are the averaged results of agents with the same base.

Chapter 6

# Opponent Estimation Study: Progressive Versus Fixed

On one hand, a decent agent – better than a fully random one – usually emerges from the task definition itself. It is often heuristic, filled with expert knowledge about the problem. On the other, a standard in-population evaluation is proved to yield a good solution, e.g., using a linear combination of some expert-based features and a basic evolution scheme. Our question is whether one of these approaches is superior to the other in terms of real-world evaluation.

Additionally, we are interested in what the difference is between using a weak and a strong opponent as a measure. This is often an interesting dilemma, as both approaches may be potentially vulnerable to fast stagnation. A weak opponent may be too easy to beat, so the win rate quickly caps at large values, while with a strong one it may struggle to achieve any victories thus, there may be no progress at all.

One may try evaluations based on a portfolio of agents, i.e., a couple of agents of different power. This is usually a good idea, as evaluations involving versatile opponents make each one more valuable. However, it is computationally much more expensive and may lead to other potential problems, like overfitting toward a local optimum, being able to beat only weaker of the opponents in portfolio.

## 6.1 Experiment setup

To properly compare the two evolution schemes, one needs to compare not only their outcomes but also the costs. In our case, the overhead of a given scheme is negligible in comparison to the cost of simulations. Thus, we use the number of simulated games as a metric of evolution cost. For the sake of simplicity, we assume that every game takes the same amount of time. That is not true, as better agents tend to play longer. In practice, the average time of a single game across the whole evolution run is comparable between schemes.

In `progressive` scenario, each two individuals played *rounds* games on each of *drafts* drafts. In `weak-op` and `strong-op` scenarios, every individual played *population* $\times$ *rounds* games on each of *drafts* drafts for each side. Overall, all three evaluation schemes use the same number of games for each individual. In our experiments, $drafts = 10$, $population = 50$, and $rounds = 10$.

## 6.2 Learning comparison

To verify whether the evolution progresses, we compare the best individuals of all generations after the evolution finishes. Such progress, visualized in Fig. 5.1, is definite in all of the `progressive` individuals and less significant for the opponent-based ones. It is clear that the `progressive` scheme is better at this task – the elitism in combination with the in-population evaluation implies it.

Additionally, all three heatmaps of `weak-op` agents have some bold red stripes. Every stripe represents a few consecutive agents that were significantly weaker than the local average. The same happens in other evolution schemes, but it is rather exceptional. It is understandable, as only the `progressive` evolution takes in-population evaluation into account. But also, it shows that learning against the stronger opponent is more resilient to the forgetting issue.

## 6.3 Tournament comparison

The lack of progress visible in heatmaps does not imply a lack of general improvement. As seen in Table 5.1, the two tree-based representations achieve comparable results for both `progressive` and `strong-op` – around 45%.

However, this does not hold for the most straightforward `Linear` representation. The differences between `weak-op`, `strong-op`, and `progressive` are large, around 6% each.

When we compare how well both fixed-opponent agents play against their evolution goals, we see that their scores do not correlate with the representation. The `BinaryTree` representation performs best in the `weak-op` variant and achieved 82.9% wins against the `WeakOp`, while the other representations were much weaker – 65.9% and 56% for `Linear` and `Tree` respectively. At the same time, `BinaryTree` is the worst in the `strong-op` variant, achieving only 34.1% wins against `StrongOp`, whereas `Linear` and `Tree` achieved 38.8% and 36.7% wins respectively.

As presented in Fig. 6.1, all `weak-op` agents are strictly worse than the `strong-op` since almost the beginning of the tournament. It is not the case for all of the `progressive` and `strong-op` agents – every representation is indeed stronger in the former variant, but the `Linear-strong-op` agent is superior to both tree-like representations using `progressive` evolution.

To summarize, evolution via self-play yields better agents than evolution towards a fixed opponent. However, it is not true if the representation is not fixed, e.g., `Tree-progressive` is inferior to `Linear-strong-op`.

Additionally, evolution using a fixed opponent has a slightly different cost characteristics. While evolution using self-play simulates longer games gradually, using a fixed opponent starts with much shorter ones but ends with longer ones – already trained agent quickly deals with initial, almost random agents and holds better against the evolved ones. As expected, the `weak-op` evolution is faster than the `strong-op`.

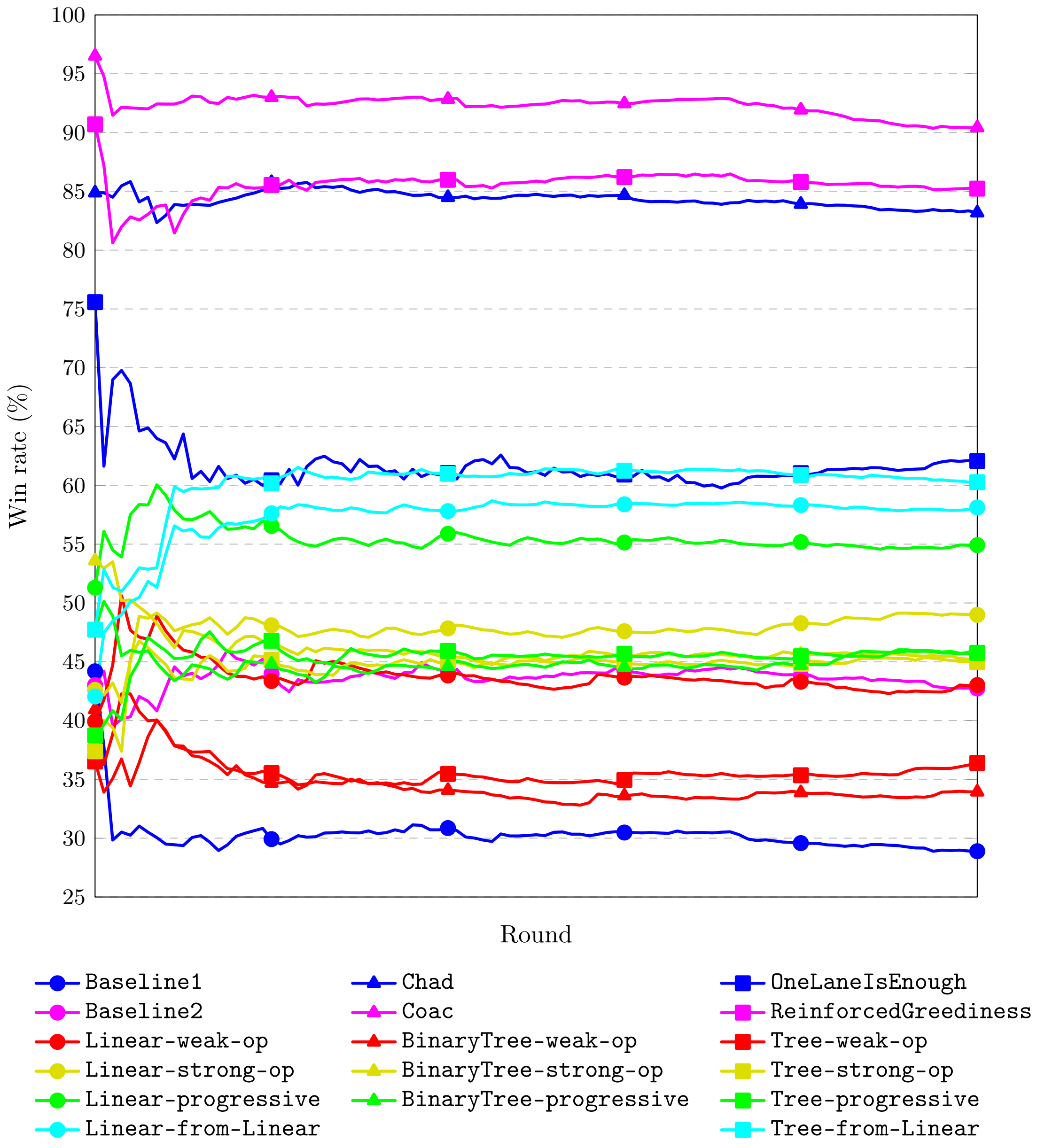

Figure 6.1: The tournament results. All scores (y-axis) stabilize towards their true values as the number of rounds (x-axis) increases.

# Chapter 7

# Conclusion

In this work, we presented a study regarding two important aspects of evolving feature-based game evaluation functions: the choice of genome representation (implying the algorithm used) and the choice of opponent used to test the model.

Although our research was focused on the domain of collectible card games, the problems stated are of general nature, and we are convinced that our observations are applicable in the other domains as well.

The key takeaway is that having limited computational resources, it is probably better to stick with a simpler linear genome representation. Based on our research (which also supports intuition), they are more reliable to produce good solutions fast.

However, with a large computational budget, we recommend applying a two-step approach. After evolving vector-based solutions, transform them into equivalent trees and continue learning to take advantage of a more general model.

Another important observation is that self-improvement is potentially a better strategy than a predefined opponent when used as a goal of evolution. Definitely, there is no point in learning against a weak opponent. Learning against a strong opponent may be profitable but does not guarantee good performance in a broader context (e.g., tournament). And although progressive learning may also stagnate into some niche meta, it still seems to be more flexible in this aspect.

For future work, we plan to investigate a generalized bootstrapping-like scheme, that would switch between the representations automatically, as soon as the evolution progress drops below a certain threshold. Separately, we would like to apply our approach to other tasks as well as evaluate different models, e.g., additional expert-knowledge features or trees with more operators. When it comes to the different evolution schemes, some kind of ensemblement of both in-population evaluation and an external goal would be interesting. Also, we can consider the extension of using a portfolio of agents as the opponents, with some dynamic additions and removals, based on the win rates against the particular opponents.

# Bibliography

[1] M. Campbell, A. J. Hoane, and F. Hsu. Deep Blue. *Artificial intelligence*, 134(1):57–83, 2002.

[2] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[3] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[4] Pablo García-Sánchez, Alberto Tonda, Antonio J Fernández-Leiva, and Carlos Cotta. Optimizing hearthstone agents using an evolutionary algorithm. *Knowledge-Based Systems*, 188:105032, 2020.

[5] Niels Justesen, Tobias Mahlmann, and Julian Togelius. Online evolution for multi-action adversarial games. In *European Conference on the Applications of Evolutionary Computation*, pages 590–603. Springer, 2016.

[6] Dire Wolf Digital and Sparkypants Studios. *The Elder Scrolls: Legends.* Bethesda Softworks, 2017.

[7] Blizzard Entertainment. *Hearthstone.* Blizzard Entertainment, 2004.

[8] Amy K Hoover, Julian Togelius, Scott Lee, and Fernando de Mesentier Silva. The Many AI Challenges of Hearthstone. *KI-Künstliche Intelligenz*, pages 1–11, 2019.

[9] Diego Perez, Spyridon Samothrakis, Simon Lucas, and Philipp Rohlfshagen. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *GECCO*, pages 351–358, 2013.

[10] C. B. Browne, E Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[11] Omid E David, H Jaap van den Herik, Moshe Koppel, and Nathan S Netanyahu. Genetic algorithms for evolving computer chess programs. *IEEE transactions on evolutionary computation*, 18(5):779–789, 2013.

[12] Magdalena Kusiak, Karol Walędzik, and Jacek Mańdziuk. Evolutionary approach to the game of checkers. In *International Conference on Adaptive and Natural Computing Algorithms*, pages 432–440. Springer, 2007.

[13] André Santos, Pedro A Santos, and Francisco S Melo. Monte carlo tree search experiments in hearthstone. In *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, pages 272–279. IEEE, 2017.

[14] Mohammed Salem, Antonio Miguel Mora, Juan Julian Merelo, and Pablo García-Sánchez. Evolving a torcs modular fuzzy driver using genetic algorithms. In *International Conference on the Applications of Evolutionary Computation*, pages 342–357, 2018.

[15] Roderich Groß, Keno Albrecht, Wolfgang Kantschik, and Wolfgang Banzhaf. Evolving chess playing programs. 12 2002.

[16] Ami Hauptman and Moshe Sipper. Gp-endchess: Using genetic programming to evolve chess endgame players. volume 3447, pages 120–131, 03 2005.

[17] Amit Benbassat and Moshe Sipper. Evolving both search and strategy for reversi players using genetic programming. pages 47–54, 09 2012.

[18] Gul Muhammad Khan, Julian Miller, and David Halliday. Developing neural structure of two agents that play checkers using cartesian genetic programming. pages 2169–2174, 01 2008.

[19] Amit Benbassat and Moshe Sipper. Evolving board-game players with genetic programming. pages 739–742, 01 2011.

[20] Gabriel Ferrer and W. Martin. Using genetic programming to evolve board evaluation functions. 11 1995.

[21] Alexander Dockhorn and Sanaz Mostaghim. Hearthstone AI Competition. `https://dockhorn.antares.uberspace.de/wordpress/`, 2018.

[22] Andrzej Janusz, Tomasz Tajmajer, and Maciej Świechowski. Helping AI to Play Hearthstone: AAIA'17 Data Mining Challenge. In *2017 Federated Conference on Computer Science and Information Systems*, pages 121–125. IEEE, 2017.

[23] Jakub Kowalski and Radosław Miernik. Legends of Code and Magic. `http://legendsofcodeandmagic.com`, 2018.

[24] Shuyi Zhang and Michael Buro. Improving hearthstone ai by learning high-level rollout policies and bucketing chance node events. In *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, pages 309–316. IEEE, 2017.

[25] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[26] Maciej Świechowski, Tomasz Tajmajer, and Andrzej Janusz. Improving Hearthstone AI by Combining MCTS and Supervised Learning Algorithms. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

[27] Sverre Johann Bjørke and Knut Aron Fludal. Deckbuilding in magic: The gathering using a genetic algorithm. Master's thesis, NTNU, 2017.

[28] Pablo García-Sánchez, Alberto Tonda, Giovanni Squillero, Antonio Mora, and Juan J Merelo. Evolutionary deckbuilding in Hearthstone. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–8, 2016.

[29] Aditya Bhatt, Scott Lee, Fernando de Mesentier Silva, Connor W Watson, Julian Togelius, and Amy K Hoover. Exploring the Hearthstone deck space. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, pages 1–10, 2018.

[30] Zhengxing Chen, Christopher Amato, Truong-Huy D Nguyen, Seth Cooper, Yizhou Sun, and Magy Seif El-Nasr. Q-deckrec: A fast deck recommendation system for collectible card games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

[31] Matthew C. Fontaine, Scott Lee, L. B. Soros, Fernando De Mesentier Silva, Julian Togelius, and Amy K. Hoover. Mapping Hearthstone Deck Spaces Through MAP-elites with Sliding Boundaries. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 161–169, 2019.

[32] Fernando de Mesentier Silva, Rodrigo Canaan, Scott Lee, Matthew C Fontaine, Julian Togelius, and Amy K Hoover. Evolving the hearthstone meta. In *IEEE Conference on Games*, pages 1–8. IEEE, 2019.

[33] Elie Bursztein. I am a legend: Hacking hearthstone using statistical learning methods. In *CIG*, pages 1–8, 2016.

[34] Łukasz Grad. Helping ai to play hearthstone using neural networks. In *2017 federated conference on computer science and information systems (FedCSIS)*, pages 131–134. IEEE, 2017.

[35] Ronaldo Vieira, Luiz Chaimowicz, and Anderson Rocha Tavares. Reinforcement learning in collectible card games: Preliminary results on legends of code and magic. In *18th Brazilian Symposium on Computer Games and Digital Entertainment, SBGames*, pages 611–614, 2019.

[36] J. Kowalski and R. Miernik. Evolutionary Approach to Collectible Card Game Arena Deckbuilding using Active Genes. In *IEEE Congress on Evolutionary Computation*, 2020.

[37] Raúl Montoliu, Raluca Gaina, Diego Perez Liebana, Daniel Delgado, and Simon Lucas. *Efficient Heuristic Policy Optimisation for a Challenging Strategic Card Game*, pages 403–418. 04 2020.

[38] Marcin Witkowski, Łukasz Klasiński, and Wojciech Meller. *Implementation of collectible card Game AI with opponent prediction.* Engineer's thesis, University of Wrocław, 2020.

[39] CodinGame. Legends of Code and Magic – Multiplayer Game. `https://www.codingame.com/multiplayer/bot-programming/legends-of-code-magic`, 2018.