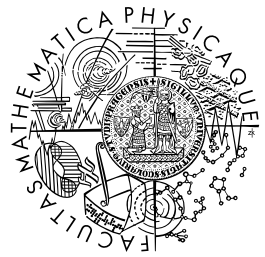

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Michal Kodad

Artificial Intelligence for the Unstable Unicorns Game

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis:	Mgr. Martin Pilát, Ph.D.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Dedication. I would like to thank my supervisor, Mgr. Martin Pilát, Ph.D., for his guidance, advice and never-ending patience during the writing of this thesis. I would also like to thank my family and friends for their support and encouragement.

Title: Artificial Intelligence for the Unstable Unicorns Game

Author: Michal Kodad

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: This work explores artificial intelligence for the game Unstable Unicorns. This game started on Kickstarter and over the years, the game creators released several expansions. This work aims to implement a game simulator for this game, analyze the game, and design the artificial intelligence for this game. First, we will analyze the game rules, game mechanics, and artificial intelligence in similar games. We implemented the game simulator as close as possible to the original rules. Afterward, we developed three different artificial intelligence algorithms. These are rule-based agents, Monte Carlo agents and evolutionary agents. Finally, we ran the experiments and comparison tests with the implemented agents. The best-performing agent is the evolutionary agent. It is quick with the best win rate.

Keywords: card game artificial intelligence multiplayer game

Contents

Introduction

Unstable Unicorns is a game from 2017 that started on Kickstarter. When the campaign on Kickstarter ended 33,720 backers pledged $1,865,140[1].

Kickstarter is a crowdfunding platform where people present their ideas or work and they are finding funds to start, finish or make more copies of the project. Backers are persons who donate to specific projects and typically they get a copy of the product or a reward for their donation (for example, a t-shirt, tin figurine, etc.). The campaign typically has more tiers of rewards based on the donation amounts.

The game became popular and the creators of the game released several expansions. Unfortunately, the expansions are not translated into the Czech language.

The game is a card game for 2-8 players. Each player has a stable of unicorns and the goal is to be the first player to have 6 (for a game with 6-8 players) or 7 unicorns (for a game with 2-5 players) in their stable after all effects are resolved.

If more players have the same number of unicorns in their stable, the player with the greatest sum of letters in the names of their unicorns wins. If more players have the same sum of letters, then everyone loses.

In this thesis, all games will be played with 6 players and the win condition is to have 6 or more unicorns in the player's stable. The game implementation itself can handle a different number of players, but the AI agents would need to play different strategies based on the number of players.

All players start with 5 cards in their hand and 1 unicorn called "Baby Unicorn" in their stable. The game is turn-based and each turn has 4 phases:

- Beginning of Turn phase
- Draw phase
- Action phase
- End of Turn phase

At the beginning of the turn, all effects that are triggered at the beginning

[1] `http://unstablegameswiki.com/index.php?title=2017_Kickstarter`

of the turn are resolved. Then the player draws a card from the deck and can typically play one card from their hand or draw a card.

After the player announces he wants to play a given card, other players can play cards as a reaction. In the base game, there are only 2 cards that can be played as reaction cards: "Neigh" and "Super Neigh". In the expansions, there are a lot more cards that can be played as reaction cards. Played cards are stacked on top of each other. When nobody wants to play a card, the top card is resolved. Then anybody can play a card as a reaction to the card which is on the top of the stack. This process is repeated until the stack is empty.

After the action phase, all effects triggered at the turn's end are resolved.

This is a brief overview of the game. The game has a lot of rules, cards, and effects, which will be described in the next chapter.

This thesis has several goals: The first goal of this thesis is to implement the game itself because there is no implementation of the game except the implementation in the Tabletop Simulator game engine, which is unsuitable for this work. The second goal is to implement AI agents that can play the game. The third goal is to compare the performance of the implemented AI agents.

Chapter 1

Game description

The following chapter describes the mechanics of the game in detail.

First, we will describe the terminology used in the game. It has a lot of keywords so the first section can be used as a glossary. Then we will look at the effects in more detail. After that, we will describe how the effects of cards are resolved. And finally, we will talk about impossible actions.

1.1 Terminology

Terminology of places (card **sources**) where cards can be stored are:

- **Stable** - The area where the player plays Unicorn cards, Upgrades cards and Downgrades cards.
- **Nursery** - The area where baby unicorns are stored when no player owns them.
- **Deck** - The face-down pile of cards from which the players draw cards.
- **Discard pile** - The face-up pile of cards where the players put cards when they are "destroyed" or "played".

The cards can have various keywords, and most of them are used in this thesis. Many keywords can be similar to those from other card games, but sometimes the meaning can be slightly different.

- **Sacrifice** - Send a card from **your** stable to the discard pile.
- **Destroy** - Send a card from **any other** player's stable to the discard pile.
- **Steal** - Move a card from **any other** player's stable into **your** stable.
- **Discard** - Send a card from **your** hand to the discard pile.
- **Pull** - The player gets a random card from the source and puts it into his hand.

- **Choose** - The player chooses a card from the source and puts it into his hand.
- **Search** - The effect is similar to choose card effect. The player selects a card from the source. This card is **revealed** to all players and put into his hand.
- **Draw** - Pull a card from the deck (Draw a card from the top of the pile).
- **Play** - It refers to playing a card from your hand during the Action Phase or playing an Instant card. When the card is being played, any other player can react to the card by instant card.
- **Bring** - Move a card to the stable. The brought card cannot be Neigh'd (it is impossible to disallow putting the card to the stable).

Each card has one of the following types:

- **Unicorn** card - The card is a unicorn and can be put into **any** stable. The unicorn type has one of these subtypes:
  - **Baby unicorn** card - The baby unicorn can occur only in the nursery or in the player's stable.
  - **Basic unicorn** card - The unicorn without any special effects.
  - **Magic unicorn** card - The unicorn that has some effect.
- **Upgrade** card - The card grants some positive effects. The upgrade card can be played into **any** stable.
- **Downgrade** card - The card grants some negative effects. The downgrade card can be played into **any** stable.
- **Magic** card - The card with a one-time effect (spell). It takes effect immediately when it is played and then is discarded.
- **Instant** card - The card that can be played at any time when any other player plays a card (unless the card says otherwise). If the instant card is played, then you can react to that card by playing the instant card (even though you already played the card this turn). Any number of instant cards can be chained during a single turn.

Some effects target one or more players, but sometimes it can be unclear whether the statement includes you. Therefore, there is precise wording in the effect description. The following terminology for player targeting is used:

- **Any player** - Refers to any player in the game, **including** you.
- **Any other player** - Refers to any player in the game, **excluding** you.
- **Each player** - Refers to each player in the game, **including** you.
- **Each other player** - Refers to each player in the game, **excluding** you.
- **Any number of players** - Refers to any number of players you choose, **excluding** you.

1.2 Types of effects

The game has three types of effects: one-time effect, continuous effect, and trigger effect. The one-time effect has an immediate disposable impact on the game. For example, *Steal a unicorn card.* The continuous effect is active as long as the card is in your stable. For instance, *All your unicorns are pandas* (Pandamonium). This effect can be beneficial because all effects that target unicorns cannot target the player's pandas, but the player cannot win because the player has no unicorns in his stable! The trigger effect is activated when some event occurs. For example, *If this card is in your Stable at the beginning of your turn, you may draw an extra card.* (Extra Tail). An extra card means that the player may draw a card in the draw phase if the player has drawn a card. If some effect disallows the player to draw a card or even skip the draw phase, the player cannot draw an extra card.

For some effects like **draw an extra card**, it is good to consult more detailed rules on the official wiki. Sometimes there are described some special cases.

Effects can be combined in different ways. The first combination is simple "and". Perform the first and the second effect simultaneously. The second combination is "then". The first effect must be resolved, then the second effect. The last combination is conditional "if you do". The effect after "if you do" is performed only if the player performed the first effect. For example, *If this card is in your stable at the beginning of your turn, you may sacrifice a card. If you do, destroy a card.* (Glitter bomb).

1.3 Resolving of cards effects

In this section, we will describe the core of the game mechanics – how the effects of the cards are resolved.

At the same time, one or more effects must be resolved. Typically more effects must be resolved at the beginning of the turn or the end of the turn. Effects that must be resolved at the same time must be resolved simultaneously and independently. These effects are in the first chain link.

The term chain link comes from the fact that the effects make a chain in resolution. The first trigger effect is triggered by some effect and then the second trigger effect is triggered by the first effect and so on. These effects make a chain and the chain link is one whole resolution step.

After the first chain link is resolved simultaneously and independently, some effects may be triggered as a reaction to the first chain link. All these triggered effects form the second chain link. This is repeated until the next chain link is empty. The card is resolved when the whole effect chain is resolved (all chain links are resolved).

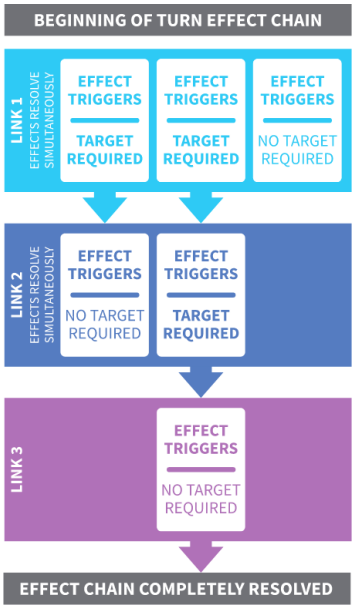

Figure 1.1 The whole chain effect with individual chain links. Source: the official game rules.

This approach to how the effects are resolved can be a little bit confusing because even people typically think in sequential order. Additionally, these have some consequences. For instance, when two effects can be activated at the beginning of the turn, then the player must decide which effects will be activated and resolve them simultaneously. The player cannot activate the first effect, see the result, and then decides to activate the second effect.

1.4 Impossible actions

Effects can be mandatory or optional. Mandatory effects must be resolved. All **magic** cards have mandatory effects. **Unicorn** cards can have mandatory or optional effects. Optional effects use the "may" in the effect description.

In general, players should not play cards with mandatory effects that cannot be resolved during unloading the card or when a player says he wants to play a given card. For instance, the player plays a card with the effect discard a card, but the player has no other cards in his hand. But it is fine if the player has only a "Neigh" card in his hand and the player plays this card during the reaction phase. In this case, the player does not discard any card (his hand is empty) and does not violate the rules.

Sometimes another player plays an effect, or some card in the player stable

says that some player must do some impossible action. In this case, this effect is ignored. There are some examples of this situation:

- **Discard** a card from the player's hand when the player has no card in his hand.
- **Sacrifice** a card from the player's stable when the player has no card in his stable or the player has a card in his stable, but cannot be sacrificed.

Chapter 2

Related work

2.1 Properties of task environments

Before we can compare our game with other similar games we must analyze the properties of the game environment. The properties of the environment are well described in this book [1] in the second chapter. In the following section, I will briefly describe the properties of the Unstable Unicorns game.

The game is partially observable. Players can see their cards, the discard pile and the board state (unicorns/upgrades/downgrades in each stable). They normally cannot see other players' cards in hand and the order of cards in the draw pile.

The game is a multi-agent environment. The game can be played with 2-8 players.

The game is stochastic. The cards themselves are deterministic – all effects have no random components. However, the order of cards in the draw pile is random.

The game is sequential. The next game state is determined by the previous game state and the action of the players.

The game is dynamic. When a player announces he wants to play a given card other players can react to this card. Each player independently decides whether to react or not.

The game is discrete. In the game, there is a finite number of cards and each time an integer number of cards can be played. It is not possible to play a fraction of a card. This is also true for the effects of cards.

The game theoretically can be infinite, but in practice, the game ends after at most hundreds of turns.

2.2 Artificial intelligence in similar games

A very popular and similar game is Magic: The Gathering[2]. Magic: The Gathering is a collectible card game where players buy booster packs, build their decks and play against each other. The game rules are not too hard but the game complexity grows with the different effects of cards and the combinations of cards. After years of existence, there are a lot of decks that break the game balance (for instance infinite combo decks[1]). Every year, the game developers release new sets of cards, and it is impossible to balance the game. Therefore, there exists a lot of formats that allow using only selected sets of cards. Sometimes, some cards are too strong and are banned for the format. The game has a lot of competitive tournaments for big prizes.

In 2009, Monte Carlo Tree Search was applied to use to play the game [3]. Results showed that the algorithm can beat the strong rule-based agent that was designed by an expert player of Magic: The Gathering.

Another well-known card game is Poker. Poker is a game for multiple players where each player knows their two cards and the cards on the table. The goal is to have the best combination of cards. This game is based on evaluating the probability of the opponent's card strength combination and the probability of the player's card strength combination. Another aspect of the game is psychological. The players can bluff and bet to confuse the opponent. Some people can think that the game is just gambling and the game is based on luck. However, a strategy was developed that wins the game on average [4] (not all games can be won). This is a proof that the game is not just gambling.

The last game with slightly different environmental properties is Go game. Go is a game for two players, turn-based, deterministic with perfect information. The game is challenging because of the huge search space. In 2016, the DeepMind team developed an algorithm[5] based on Monte Carlo Tree Search with deep neural networks, which became very successful in Go. The algorithm was able to beat the best human players.

2.3 Evolutionary algorithms

Evolutionary algorithms[6] are nature-inspired algorithms. The idea is to mimic Darwin's theory of evolution. The individuals compete to reproduce their genes for the next generation.

The whole algorithm is based on individuals in the population. The individual is the solution to the problem. In this section, the individual is a bit string. Other representations of the individual will be used later.

[1] `https://tappedout.net/mtg-decks/selesnya-infinite-elf-tokens/`

Algorithm 1 Pseudocode of the evolutionary algorithm

```
function EvolutionaryAlgorithm
    Initialize the population
    Evaluate the fitness of each individual
    while Until the stopping criterion is not met do
        Select the parents
        Crossover
        Mutation
        Evaluate the fitness of each individual
        Offspring is the new population
    end while
end function
```

Most of the time, the initial population is randomly generated. It is possible to use some smart initialization but we must do it very carefully, otherwise, we can get stuck in a local optimum.

The next step is to evaluate the fitness of each individual. The fitness function evaluates how good is the individual as a solution to the problem. The fitness function depends on the problem and the computing of the fitness function should be as quick as possible. Most of the time, the bottleneck of the algorithm is the fitness function.

The next step is selecting the parents. The selection can be done in many ways. The most common approaches are roulette wheel selection and tournament selection. In the roulette wheel, selection each individual is assigned an arc on the wheel. The arc is proportional to the fitness of the individual in the population. Then we are just spinning the wheel and selecting the individual. The probability of the individual x is:

$$\frac{f(x)}{\sum_i f(i)}$$

where sum iterates over whole population and f is fitness function. In the tournament selection, we select randomly k individuals and the best individual is selected as a parent. Both approaches have their advantages and disadvantages. The roulette wheel can have stronger selection pressure when the fitnesses are more different but when the fitnesses are similar, the roulette wheel performs as random selection. The tournament has the same selection pressure during the whole evolution. Another benefit is that the tournament selection can be easily switched to minimization instead maximization of the problem.

Another step is crossover. The crossover combines the genes of the parents to create a new individual. The crossover can be done in many ways too. The most common crossovers are one-point crossover and uniform crossover. The

one-point crossover combines two parents and produces two offspring. The crossover randomly selects a point and the first offspring gets the genes from the first parent up to the point and the genes from the second parent after the point. The second offspring gets the other part of the genes. The uniform crossover randomly chooses for every gene from which parent it will be taken.

The last step is mutation. The mutation randomly changes offspring's genes. The mutation is mostly for the exploration of the search space and escaping from local optima. The most common mutation is bit-flip and uniform mutation. The bit-flip mutation randomly decides whether the gene will be flipped. The uniform mutation is mostly same as bit-flip. The only difference is that the uniform mutation randomly generates values from the allowed subset or interval of the gene.

Then offspring are evaluated and the new population replaces the old population. This process is repeated until we reach the stopping criterion. For instance: the maximum number of generations reached, the best individual has lower fitness than some constant, or the fitnesses did not improve after some number of generations.

2.4 Monte Carlo Tree Search

Monte Carlo Tree Search[7] is a search technique that combines a tree search algorithm and machine learning principles (the algorithm tries to exploit information that it detained so far). It solves the problem of exploration and exploitation. The algorithm expands a node that is most promising (best-first search).

The algorithm itself is quite simple and has four main steps: Selection, Expansion, Simulation and Backpropagation.

- Selection: The algorithm starts from the root node and traverses the tree until it reaches a leaf node. A leaf node is a node that has a child node that is not expanded yet. The selection of the next child node (successor) is done by UCT formula (Upper Confidence Bound applied to Trees) which is based on the UCB1 formula. The UCT formula is following:

$$\text{successor} = \arg\max_{s \in \text{successors}} \left(\frac{w_s}{n_s} + c \cdot \sqrt{\frac{\log n_{\text{parent}}}{n_s}} \right)$$

  where w_s is the number of wins after the successor state, n_s is the number of playouts after the successor state, n_{parent} is the number of playouts after the parent node of successors and c is a constant that controls the exploration-exploitation trade-off. The constant c is usually set to $\sqrt{2} \approx 1.41$.

- Expansion: The algorithm randomly selects an unexpanded successor of a selected leaf node and expands it.
- Simulation (Playout): The algorithm plays one game from the expanded node state until the end of the game. Moves in simulation are chosen by a given base strategy (random, greedy, etc.).
- Backpropagation: The algorithm updates the statistics of the expanded node and its ancestors.

The algorithm is repeated for a given number of playouts. The next move (the successor of the root node) is the one with the highest number of playouts (not the most win rate).

Chapter 3

Rule and game modifications

We have implemented the core game to be as close as possible to the original rules. One rule is a little bit strange. The rule says that *If multiple triggered effects require a target, the same card may not be selected twice in the same link.*[8]. This rule is clear for one player, but how should it work for two or more players targeting simultaneously? It is practically impossible to obey this rule because targeting must be independent and simultaneous. We implemented this rule in a way that the same card could be selected at most once in the same link but is not independent (if the first effect targets card A, then the second effect knows that card A is already targeted). We must implement this rule in some way because, without it, we must solve which effect is performed and which is not. For instance, when the same card is targeted by the steal effect and by the destroy effect, which effect should "win"? Additionally, the rule says that the effects are resolved simultaneously.

The solution is the effects are not resolved independently. The effects are resolved in the order when they are added to the chain link. But it still mimics that effects are simultaneously resolved. All effects see the same state, and they are independent except for the targeting.

This rule has a continuation: *If there are not enough cards to target, you may only trigger as many effects as you have options for targets. You must select targets for mandatory effects before selecting targets for optional effects.*[8]. The rule is not implemented at all. In the implementation, we do not distinguish between mandatory and optional effects during the resolution of the chain link. However, we still check a card's requirements when a player plays the card. For instance, if a player plays a spell card with the discard effect, they must hold at least one other card in their hand. In the implementation, I do not trigger as many effects as a player has options for targets because all triggered effects are added to the chain link. As we discussed in the previous paragraph, the effects are resolved in the order in which they are added to the chain link.

Then we do not implement one card. The card is “Nanny cam” with effect *Your hand must be visible to all player at all times.* This card was not implemented because no agent can use this effect. However, the simulator tracks this knowledge for every player because it is necessary information for copying the state. The copy function only copies the visible part of the state for a player. If a player does not know the given card in the player’s hand, then this card is exchanged for some other card.

Chapter 4

Game implementation

4.1 Existing implementations

There are multiple implementations of the game in the Tabletop Simulator. These implementations are unsuitable for this work because the game is not simulated. All effects are resolved by players. It is only a board game in digital form.

4.2 Implementation requirements

- The game must be reproducible if all players play the same cards. Initial randomness is a game seed.
- The game simulator must be extendable for new decks as a plugin.
- The game is playable by a user and other players are random agents.
- The game simulator implements cards from the base game set.

4.3 Used frameworks

The game simulator was implemented in C# .NET 6. We used one of the C# features, Nullable reference types[1]. This feature semantically changes that all reference objects are disallowed by default to store null values. It additionally warns about assigning a null value to a not-null value type. Programming with this feature is a little bit different, but it is a good feature that prevents a lot of bugs.

The game is implemented as a console application. For unit testing, we used the xUnit framework, as we were the most familiar with it.

[1] `https://learn.microsoft.com/en-us/dotnet/csharp/nullable-references`

For the evolutionary algorithm, we used the GenericSharp library. We did not find other usable libraries in C#.

Finally, for the command line parser, we used the System.CommandLine library. It is a new library for parsing command-line arguments. We are using the preview version but it works well. Other libraries have problems with subcommands or parameter restrictions on values. Additionally, the usage is very similar to `argparse` library in Python.

4.4 Core game implementation

The main problem the simulator must solve is the resolution of effects. Most of the game is about a card being played and the card has some effects. In the game, there are three types of effects: one-time effects, triggered effects, and continuous effects. The one-time effects are quite simple. They are added to the effect chain link and are resolved at the right time. The trigger effects have a condition and when the condition is met, the effect is added to the effect chain link. From the implementation point of view, the trigger effects can be easily implemented as a subscriber to some event, the trigger condition and the one-time effect that can be added to the effect chain link. The continuous effects are the most complicated because they are always active. Some components must remember the continuous effects and change the behavior of the game according to the continuous effects.

For the implementation, we have one constraint: We want to be able to add new cards and effects to the game. In C#, a library can be loaded and used by reflection. This way, it is very easy to make a plugin to the existing application (for example, a new card). Most card effects can be implemented in this way. However, some may require changes in the simulator itself. The most restricted type of effect is the continuous effect because the components in the simulator must be able to handle the continuous effects. I have hardcoded the continuous effects in the simulator because implementing, for instance, *Triggered effects of your Unicorn cards do not activate* (Blinding Light) is not possible without editing the simulator.

The constraint shows that the one-time and trigger effects cannot be implemented in the core simulator (GameController). Most of the functionality must be implemented in the effect itself and the GameController just resolves the effects (the chain link). For the effect resolution, the best option is to use event-driven programming. Event-driven programming is a pattern where the flow of the program is determined by events such as mouse clicks or key presses. For the simulator, the events are the effects. The cards register the effects (events) and during the resolution of the chain link, all the effects will be resolved. Event-driven programming is a good choice because each effect can override the methods

which are called by GameController to implement the effect at the right time. The GameController does not need to know about the effect implementation.

For the trigger effects, we used the observer pattern. The observer pattern is a special type of event-driven programming based on the simple idea where the subject (GameController) notifies the observers (trigger effects) when something happens, for example, a card left stable.

4.5 Effect implementation

The effects have these methods, which are used later in this chapter:

- `ChooseTargets` - This method sets the targets of the effect. When this effect targets some card, this targeting must be done in this method.
- `InvokeEffect` - This method should execute the effect.
- `InvokeReactionEffect` - This method should be implemented only by reaction effects. This method must do the effect preparation. For example, if this effect retargets the target, this method must remove the old target and add a new one.

4.6 Steps of the effect resolution

The resolution of one chain link is divided into seven steps:

1. Choosing targets of the effect. During this step, the `ChooseTargets` method is called on every effect.
2. Triggering the effects that may change the targets of the effects. For example, *If 1 of your Unicorn cards would be destroyed, you may SACRIFICE this card instead.* (Black Knight Unicorn). During this step, the `InvokeReactionEffect` method is called on every triggered effect (this is done in `TriggerEffect` class).
3. Triggering the effects that may change the card location. For example, *If this card would be sacrificed, destroyed, or returned to your hand, return it to the Nursery instead.* (Baby Unicorn). During this step, same as in the previous step, the `InvokeReactionEffect` method is called on every triggered effect.
4. Triggering the effects on the event `PreCardLeftStable`. It is all effects that must be resolved before the "owning" card left the stable. The term "owning" means that the effect is written on the card that is leaving the stable. For instance, *If this card is sacrificed or destroyed, you may DESTROY a Unicorn card.* (Stabby The Unicorn). Triggered effects are normally added

to the next chain link but can be set to be resolved in the current chain link. However, the `InvokeReactionEffect` **is not** called on the triggered effect.
5. Unregistering all the effects belonging to the card that are targets of some effect in the chain link and they are on the table – unicorn in stable, upgrade, or downgrade. Cards in hand have no effect on the game, therefore, their effects did not need to be unregistered.
6. Performing the effects in the chain link. During this step, the `InvokeEffect` method is called on every effect.
7. Registering all the effects of the cards that were targets of some effect and they are on the table.

The first three steps are not very interesting, but the steps must be in this order because some effect can theoretically save a card from destroy effect like Black Knight Unicorn, but then this card can go back to the hand of the stable owner. The second reason is that it makes sense first to find out that the card is still a target of the effect and afterward to find out where the card should be located after the effect.

Now is the right place to introduce reactive trigger effects. It is a special type of trigger effect that is triggered during chain link resolution and this trigger effect creates an effect that is added to the **current** chain link. It works as a normal trigger effect, but a normal trigger effect adds effect to the **next** chain link. Another difference is that a reactive trigger effect is added during resolution – during step 2 or step 3. This added effect must bear in mind that the `ChooseTargets` function will not be called for this effect. If the effect needs to set up the card target, remove targets from other effects or change the location of the card, it must be done in the `InvokeReactionEffect` function. On the other hand, the non-reactive effect should never implement the `InvokeReactionEffect` function.

In the fourth step, only the effects of the owning cards that are targeted by the effect should be added to the **next** chain link. The reason is that this effect cannot be registered during `CardLeftStable` event because the effects of owning a card will be already unregistered (step 5). For the effects that **check** if some card has left the stable use `CardLeftStable` event. The reason comes from the game rules – effects must be resolved simultaneously. If card A and card B are targeted by the same or different effect and card B should be triggered when card A left the stable then it does not make sense to trigger this effect because, at the same time, card B is not already in the stable.

The rest of the steps are quite straightforward. Step 5 unregister the effects of the cards that are targeted by the effect. Step 6 performs the effect. If the card has left or entered the stable, the appropriate event is published (`CardLeftStable`, `CardEnteredStable`). Step 7 registers the effects of the cards that have been

targeted by the effect.

4.7 Resolving instant cards

Whenever a card is played, other players can react to this card by playing instant cards. In the normal game, the order of the instant cards in the stack is determined by who played quicker the card. In the simulator, we cannot use this because the `GameController` asks each player if they want to play an instant card. Then it has a list of reactions of players that want to react and we cannot say this player played the card quicker than the other player. Therefore, we pick randomly the player's reaction and this player was the "quickest". Afterward, we ask all players if they want to react to the reaction on the card on the top of the stack. This is repeated until there are no cards on the stack.

4.8 State machine

Some agents want to copy the state of the game and then run simulations of the game. Getting the state of the game is not free. If some agent copies the current state then when the game started to simulate from that point the simulation must continue from the point where it was copied (even from the middle of the chain link resolution). This is done by two conditions. The `GameController` must be the state machine and all effects must be the state machine too. If the effect does at most one prompt to any player then it is a state machine already otherwise it must remember if some prompt was already asked.

4.9 Copy game state

Coping of the game state is implemented in the `GameController` class by the `Clone` function, which makes a deep copy of all objects (cards, effects, game state). This function has two parameters. The first parameter says which player is making a copy of the game state. It is important because this game is partially observable and the players did not know the same information about other players cards in hand. The second parameter is the mapping function from the original game players/agents to new ones. This makes sense because when the player decides which action should be played, the player is not 100% sure what the opponent will do. If we copy the game state with actual agents, then it will be cheating. Secondly, the agents **are** part of the game state, therefore, they **cannot be reused** in the copied game state. Lastly, if we copy the agent representing a

real player, then this player will not play hundreds of games for a single decision in the game.

During cloning, we must convert objects of the original game state to new objects. In all base class effects, this is done automatically, but if we create a new effect where we store some additional information then we must convert it too. When we don't convert, for example, a card, then we want to move the card to a different location then the card will be moved in the **original** game state. This can easily occur in trigger effects when some lambda function stores something from the original game state. The way to solve this problem is to store only indexes of the objects. The `GameController` has an `_allCards` property where all cards in the game are stored. In the `Players` property, all players in the game are stored.

4.10 Random actions

Some effects have random elements, such as the pull card effect or some agents want to make random decisions. If every component creates its random number generator, then the game will not be reproducible. This is bad for several reasons. The debugging of the game simulator is much harder. When we do experiments, then we cannot rerun the experiment with the same random seed. For that reason, the `GameController` has a `Random` property that provides a global random number generator for the game. We use this random number generator whenever we need to generate a random number.

4.11 Creating a new card

Creating a new card is quite simple. Create a new class that inherits from `CardTemplateSource`. This class must implement one method which returns a `CardTemplate` object. The `CardTemplateSource` has one field called `EmptyCard` that returns a new empty object of `CardTemplate`. Then sets the properties of the card by the fluent syntax. At least `Name` and `CardType` should be set.

It is possible to add one or more effects to the card. We can add a one-time effect by the `Cast` method. Other method names in the `CardTemplateSource` class are self-explanatory. The methods can be called multiple times to add more effects of the same type. The methods accept effect factories (`FactoryEffect` or `ContinuousFactoryEffect`) because the effect on the card can be resolved multiple times during one game. The easiest way to solve this problem is to create a new effect whenever the effect must be added to the effect chain link or

whenever a new continuous effect is registered.

All implemented cards are in the `BaseSet` directory. Good examples of implemented classes are `Glitter Tornado`, `BabyUnicorn` and `QueenBeeUnicorn` cards, which are simple examples of how to use one-time, trigger and continuous effects.

4.12 Creating a new effect

When we want to create a new one-time effect, we inherit the `AEffect` base class. When we make a more specific effect, for example, the destroy effect, then we inherit one of the destroy effects that are already implemented. The reason is that some effects are activated on destroy effect, and a check, if the effect inherits from this class, is done.

The base class implements a helper function on validation selection, maintaining the cards that are already selected in the current chain link and removing the card from that list.

Some effects can give us (as a player) additional information, for instance, which cards the other player has in his hand. For this situation, there is the `CardVisibilityTracker` component in `GameController` that maintains this information. On the other hand, effects can remove this information, for example, the choose effect. Player A see player B's cards and took one of them. Player A knows all cards of player B but other players know nothing about player B's cards. Even if before the choose effect, player C knew all cards of player B then after the choose effect, player C did not know which card was took by player A. We must somehow maintain this knowledge for player C and the easy solution is just to remove all information about player B's cards.

The continuous effects must inherit the `AContinuousEffect` base class. The continuous effect implementation is much simpler than the one-time effect. Only override the method which you want to implement. The trigger effect is only a composition of the trigger predicate, the list of events to listen to, and the effect factory for the one-time effect.

All implemented one-time effects are in the `BasicEffects` directory. Good examples of implemented classes are `BringCardFromSourceOnTable`, which is a pretty simple effect, and a more complicated effect is `DiscardEffect`.

All implemented continuous effects are in the `BasicContinuousEffects` directory. An example of implemented class can be `PlayerCantPlayUpgrades`.

4.13 Creating a new agent

Creating a new agent is not complicated. We need to create a new class that inherits from the `APlayer` base class and implements all methods. Each method has a short description of when this method is used. There is one condition, the agent **should not cheat**. Making an agent that cheats is easy because it can look at other player cards – all information is accessible through `GameController`. We can add some cards to the agent's hand by `GameController`. We can find out who other players are (for instance, for cooperation) and many other things.

Additionally, the agent should not modify the list of available actions. The list can be a list of effects, cards or players. Do not modify it because effects do not make a copy of this list before the agent is called. It is not hard to fix this problem, but this solution was chosen primarily due to performance reasons (not to make lots of list copies).

All implemented agents are in the `Agent` directory. Good examples of implemented classes are `RandomPlayer` and `RuleBasedAgent`.

Chapter 5

Artificial agents

In this thesis, we implemented four types of artificial agents for the game Unstable Unicorns. We will start with standard agent types such as random or rule-based. The random agent is a very nice baseline for other agents and comparison. It is also very helpful for catching bugs during the development of the game simulator.

5.1 Random agent

As the name suggests, the random agent plays random actions from the set of possible actions. It is very easy to implement because nearly every method provides a list of possible actions, and we can pick one of them randomly.

5.2 Rule-based agent

The rule-based agent is a bit more complicated than the random agent. The reason is that there are a lot of different actions that we must implement. Which card should be selected to discard, and which one to destroy? Should we play a card, or should we activate an effect?

We developed a rule-based agent based on a card tier list. Nearly every card game has a tier list of cards to measure a card's power. The used tier list can be found on the Reddit website[9]. We interpret values in the S tier as zero, in the A+ tier as one and so no. Action selection is done by sorting cards by this tier list with additional changes.

For **sacrifice** effect, we want to select one of the downgrade cards first. This is easily done by changing the value of the card retrieved from the tier list. In the implementation, we used subtraction by 1000. On the other hand, we do not want to sacrifice upgrade cards, so we add 1000 to the card's value.

When we need to select targets for **destroy** or **return** effect, we use the tier list and set our cards to lower priority by adding a value 100. However, we subtract 1000 for our downgrade cards (we do not want to have downgrade cards). On the other hand, for the opponent's downgrade cards, we add 1000.

For **discard** effect, the card selection is based only on the tier list. We choose cards with the lowest tier. On the other hand, for **choose**, **pull**, **sacrifice**, save (the effect which saves some unicorn like Black knight unicorn card) and **steal** effects, the card order is reversed.

When we decide if we should use the optional effect of the card, the method always returns true because there are no bad optional effects. Some optional effects are not good to activate every time, but the decision of when the effect is bad or good is not so trivial.

For the decision of which effect should be activated (the effects can have variants), we randomly select one. When we move cards between the stables, random selection is also used.

When we decide which card should be played, we use the tier list, but instead of playing the best card, it plays the worst card. The reason is that at the start of the game, other players will likely have some instant cards to disallow playing a good card.

For choosing card location of agent cards, we play all cards except downgrades to our stable. The downgrade card will be played to the opponent's stable with the most unicorns.

When we deciding when we play an instant card, then we do not "Neigh" our cards. Otherwise, if the card on the bottom of the stack is tier A+ or better, we play an instant card.

Finally, when choosing the player for the effect, we select our agent if we can. Then we order the players by the number of unicorns in stable and select players with the most unicorns.

All values are implemented as a float because the evolutionary agent will reuse the method implementation.

5.3 Monte Carlo Tree Search (MCTS) agent

The Monte Carlo Tree Search (MCTS) agent is an implementation of the MCTS algorithm described earlier in this thesis. The algorithm is not complicated to implement, but integrating it into the game simulator is a little bit tricky. The main problem is making a specific single action and then storing the state after this action. The reason is that we cannot tell who will be the next player taking an action. Some agents can be skipped by the simulator because they cannot play any action. The solution is to make a helper agent to play the action and store

the next state. This agent will be set to play for every player, and these agents will share information if they played the selected action and if they should store the state. There is one additional problem to mention. We cannot store directly the action that we want to play. The reason is that after the copy is made, all references in the state will be different. The solution is to store the index of the action from the list of possible actions.

The last problem was how to evaluate agents during the backpropagation step. We used the simple formula: the first player in the final game results (the best player) gets $n - 1$ points where n is the number of players. The second player gets $n - 2$ points and so on.

Now, we show the performance of the MCTS agents that use the rule-based agents as base playout strategy with different number of playouts. In the game, there were two MCTS agents and four random agents. The win of MCTS agents was counted if one of the MCTS agents won. Each of the MCTS agents played 100 games. The results are shown in the table below.

Table 5.1 Win rate of MCTS agents with a different number of playouts.

Number of playouts	Win rate of MCTS agents
100	69 %
200	75 %
400	76 %
800	84 %

As the table shows, the MCTS agent does better with increasing number of playouts, which is an expected behavior.

5.4 Evolutionary agent

The last implemented agent is the evolutionary one. The core behavior of this agent is the same as the rule-based agent. The difference is that the values of the cards are not hardcoded based on some tier list. Instead, the values will be chosen by the evolutionary algorithm.

We choose the float representation of individuals. Each gene of an individual represents one card as a number between 0 and 1 (how good the specific card is). The fitness function will be evaluated this way: Each individual will play a specific number of games and the fitness of the individual is computed by this formula:

$$\sum_{i=1}^{\#\text{games}} 2^{n-p+1}$$

where n is the number of players and p is the final position of the player in the game. The formula has this form because we want to prefer the higher positions to the lower ones. This formula says that the first place is twice as good as the second place and so on.

The selection is done by the tournament selection. The crossover is done by an arithmetic crossover. The arithmetic crossover is a simple crossover that takes the average of the parents. Finally, the mutation is done by the uniform mutation. The uniform mutation simply generates a new value for the gene.

I used the GeneticSharp library for the implementation of the evolutionary algorithm and there is a functionality that is a bit strange. If the crossover is not performed, both individuals are thrown away and they do not create children individuals. This means that the population size would be decreasing over time. However, there is a phase called `reinsertion` after a new population is finished. This reinsertion can be used to implement elitism. This reinsertion will take the best individuals from the old population and add them to the new population. This means if the chance for the crossover is 85% on average, there is 15% of elitism.

The GeneticSharp library does not evaluate the already evaluated individuals' fitness. In a typical case, this is good because it saves time but in our case, we want to evaluate the fitness of the individuals again and again. Some individuals can be lucky during the fitness evaluation and they get a good fitness. For instance, the individual can luckily win all games and then this individual will be copied to the next generation. If we evaluate all individuals no matter if they were evaluated before then the evolution will be slower but it will be more accurate for individual measurements. Theoretically, we will throw away the bad individuals and we will keep the good ones.

Then there are additional questions. The first one is how many games should be played by each individual during the fitness evaluation. This is an important question because the fitness function evaluation is the most time-consuming part of the algorithm and the more games are played the longer the algorithm will take. To find it out I ran the benchmark with the different number of games and initial seeds. In the games, there were three random agents and three rule-based agents. The results are shown in the table below.

Table 5.2 The win rate and variance of the fitness evaluation with a different number of games and initial seeds.

Number of games	Initial seed	Win rate	Variance
3	0	1	0
	4000	1	0
	6000	1	0
	9000	0.6667	0.2222
5	0	1	0
	4000	1	0
	6000	1	0
	9000	0.8	0.16
10	0	0.9	0.09
	4000	1	0
	6000	1	0
	9000	0.9	0.09
20	0	0.95	0.0475
	4000	1	0
	6000	1	0
	9000	0.95	0.0475
100	0	0.99	0.0099
	4000	0.98	0.0196
	6000	0.99	0.0099
	9000	0.99	0.0099
200	0	0.985	0.0148
	4000	0.965	0.0338
	6000	0.975	0.0244
	9000	0.99	0.0099
500	0	0.972	0.0272
	4000	0.98	0.0196
	6000	0.978	0.0215
	9000	0.98	0.0196

It shows that with the increasing number of games, the win rate is more accurate but around 100 games, the win rate accuracy is good enough. The win rate after 100,000 games is around 98.3%. Unfortunately, this number of games will take ages with more complex agents as the MCTS agents with a lot of playouts. For this reason, I choose only ten games for the fitness evaluation with MCTS agents.

The second question is if it is better to have a smaller population and more

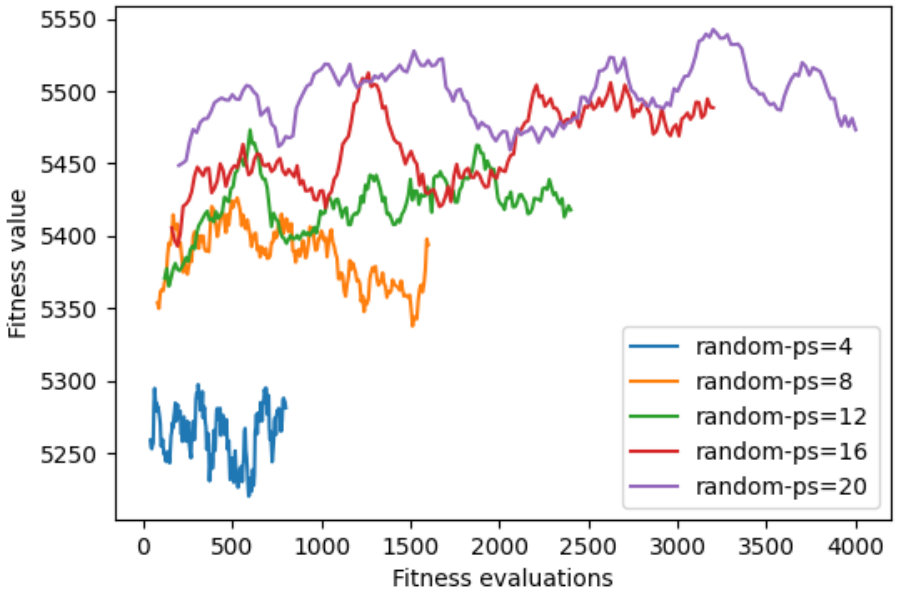

Figure 5.1 The performance of the different population sizes. "ps" means the population size. The single line in the figure is the mean of the 10 experiments and lines are smooth by moving average over 10 values. Values are noisy but we can see the increasing performance. In all experiments, the evaluated individual played 200 games with five random agents.

generations or a larger population and fewer generations. The answer depends on the problem. I made tests with different population sizes. The results are shown in the figure below.

The figure shows that the bigger the population size and the fewer generations are better. The figure also shows that the population relatively quickly goes to the local optimum and then it does not change much.

Chapter 6

Comparing AI agents

First, we will compare the rule-based agent with the MCTS agents with a different number of playouts. Each game will have four random agents and two agents of the selected type (for instance, two MCTS with 100 playouts, two rule-based etc.). The reason is that the random agents play totally randomly and can end the game quickly because they can play to a specific stable and meet the win condition. This will not be frequent in the first rounds, but later in the game, when some players have 3-4 unicorns in their stable, this chance is much higher. To eliminate this effect, we changed the ratio of agent types. The win of a specific agent type is counted if either one of the agents of this type wins. All MCTS agents use the rule-based agent as a default policy. For each agent type, we played 100 games. It is a little bit small number for an accurate comparison, but more games with MCTS agents will take a lot more time. The results are shown in the table below.

Table 6.1 Win rate of agents. MCTS *n* means MCTS agent with *n* playouts.

Agent type	Win rate
Rule-based	93 %
MCTS 100	69 %
MCTS 200	75 %
MCTS 400	76 %

Interestingly, the rule-based agent does quite well. It seems to be a good opponent for testing how well the different evolutionary agents perform in the game.

Next, we will train the different evolutionary agents with different parameters. From the previous chapter, we know it is better to have a bigger population size and a smaller number of generations. We will use the population size 20 and 200 generations if the type of agent is not MCTS. For the MCTS agents, we use only

100 generations. The number of generations should be enough, according to the results from the previous chapter. The first parameter in evolution is the agent opponent type used during the fitness evaluation. The second parameter is the number of games during the fitness evaluation. This number is set to 100 for all agents except MCTS. For MCTS agents, we use ten games because the MCTS agent is much slower in evolution. We will make these evolutions with these agent types:

- five random agents vs. one evolutionary agent (evo_random)
- five rule-based agents vs. one evolutionary agent (evo_rule_based)
- five MCTS agents with 100 playout and random agent as default policy vs. one evolutionary agent (evo_mcts_random)
- five MCTS agents with 100 playout and rule-based agent as default policy vs. one evolutionary agent (evo_mcts_rule_based)
- five best agents from the previous generation (for evaluation of initial population rule-based agent is used) vs. one evolutionary agent (evo_best_last_gen)

All these evolution runs will be repeated ten times. For comparison, we will evaluate individuals from the last generation. We will not compare individuals by their fitnesses because it highly depends on the opponents (a random agent is a much easier opponent). For that reason, each individual will play 1000 games against the rule-based and random agent. In each game, there will be two same evolutionary individuals, and the rest will be random or rule-based agents. The reference value for two rule-based agents and four random agents is 90.9%. The win rate in this test will be the mean of the best individuals in the population for each agent type. The results are shown in the table below.

Table 6.2 Win rate of the agents. The second column shows the mean of the best individual for a given agent type. The third column shows the best-performing individual from all runs.

Evolution type	Win rate vs. random agent	Best individual win rate
evo_rule_based	93.73 %	94.8 %
evo_random	93.59 %	94.7 %
evo_best_last_gen	93.23 %	94.2 %
evo_mcts_rule_based	92.87 %	93.9 %
evo_mcts_random	92.33 %	94.1 %

Table 6.3 Win rate of the agents. The second column shows the mean of the best individual for a given agent type. The third column shows the best-performing individual from all runs.

Evolution type	Win rate vs. rule-based agent	Best individual win rate
evo_rule_based	40.62 %	42.8 %
evo_best_last_gen	38.84 %	46.0 %
evo_random	34.14 %	36.8 %
evo_mcts_rule_based	32.81 %	35.2 %
evo_mcts_random	30.18 %	38.0 %

In the tables, the first column is the type of agent used in evolution. The second column shows the mean of the best individual for a given agent type against the random agent and the rule-based agent. The third column shows the best-performing individual from all runs against the random agent and the rule-based agent.

From the tables above, we can see that the evolutionary agents are better than the rule-based agent. From the first table, we can see that the evolutionary agents are 2.7% better on average. This is no huge improvement, but the second table shows the improvement much better. The best evolutionary agent got a 40.62% win rate, and there were two individuals against four rule-based agents in the game. If the agents are equally strong, the win rate will be around 33%. The best individual got even a 46% win rate against rule-based agents.

The final evolution takes the best 100 agents from previous experiments and sets them as an initial population. We will use only the last type of evolution with the five best agents from the previous generation. The population size is 100, and the number of generations is 200.

Results against random agents: the mean win rate of best individuals is 95.05%, which is 1.3% better. The best individual got 95.7%.

Results against rule-based: the mean win rate of best individuals is 45.3%, which is 4.4% better. The best individual got 46.8%.

6.1 Conclusion

The evolutionary agents improved performance by changing the tier list of the rule-based agents. Additionally, the run speed is as quick as the rule-based agent, which is a significant advantage over MCTS agents.

The second place has the rule-based agent. It is incredible that using a card tier list performs so well. Even MCTS agents have a much lower win rate. The

increasing number of playouts helps, but the run time dramatically slows down. Even during the evolution, the MCTS agent does not help to train good individuals. Maybe we would get better results with a higher playout number, but the run time will be much slower.

Chapter 7

Conclusion

In this work, we have analyzed the Unstable Unicorns game. This includes game rules, game mechanics and description of successful AI agents in similar games. We have developed a game simulator that can be easily extended with new effects, cards and deck. We have implemented and tested several AI agents. We have developed a successful rule-based agent, which was further improved by an evolutionary algorithm. The huge advantage of the evolutionary agent is run time. The MCTS agent probably can be the same or better than the evolutionary agent with a higher number of playouts, but the run time is much higher for better comparison. Combining the win rate and run time, the evolutionary agent is an absolute winner.

7.1 Future work

One way of extending this work is to implement additional AI agents. No agents use the information about known cards in opponents hand and this could improve the win rate. This card tracking is already done by the simulator.

The game has a lot of expansions. It could be interesting to analyze how the agents will perform with the new cards from expansions. Some effects of new cards are unique. It changes how the game is played.

The last one, but most interesting for human players, is implementing a graphical interface for the game. The game can be played from a console but it is not user-friendly. The game control from the console is not bad but it is hard to track the game. I think a user interface in the console for this game is hard to make. One reason is that more effects perform simultaneously, which is hard to understand. The graphical interface helps a lot to solve this problem. Additionally, it improves the game experience and understanding of the game's progress.

Bibliography

[1] Stuart J Russell and Peter Norvig. *Artificial Intelligence A Modern Approach Third Edition*. by Pearson Education, Inc., 2010. ISBN: 978-0136042594.

[2] Wizards of the Coast. *Magic: The Gathering*. Accessed: 2. 5. 2023. URL: https://magic.wizards.com/en.

[3] C. D. Ward and P. I. Cowling. "Monte Carlo search applied to card selection in Magic: The Gathering". In: *2009 IEEE Symposium on Computational Intelligence and Games*. IEEE, 2009, pp. 9–16. DOI: 10.1109/CIG.2009.5286501.

[4] Matej Moravčík et al. "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker". In: *Science* 356.6337 (2017), pp. 508–513.

[5] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[6] A.E Eiben and J.E Smith. *Introduction to evolutionary computing*. Springer, 2015. ISBN: 978-3-662-44873-1.

[7] Guillaume Maurice Jean-Bernard Chaslot. *Monte-carlo tree search*. Doctoral thesis. Maastricht University, 2010.

[8] Unstable Games Wiki. *UU game terminology - effects*. Accessed: 2. 5. 2023. URL: http://unstablegameswiki.com/index.php?title=UU_Game_Terminology_-_Effects&oldid=16340.

[9] DirtyBlueJJeans. *My Personal Tier List for the Base Deck (First Edition)*. Accessed: 2. 5. 2023. URL: https://www.reddit.com/r/UnstableUnicorns/comments/himbmz/my_personal_tier_list_for_the_base_deck_first/.

Appendix A

Usage

When we run the application, we must specify at least one command. Some commands require additional arguments or subcommands. Of course, there are some optional arguments. The application has a help command.

A.1 Commands

- `game` - run the game. The game can be played by AI agents and one agent can be controlled by a human player.
- `variance-benchmark` - run the variance benchmark shown in chapter 5
- `mcts-agent-tests` - run the MCTS agent tests shown in chapter 5
  - all games (in json format) are stored in the `mcts_agent_tests` directory
- `evolution` or `evo` - run the evolution. This command has a lot of subcommands. All commands are described help of this command. The evolution stores results in the `eva_logs` directory. **This directory is not created automatically.** For every run, three files will be stored in the directory:
  - file starting with `stats` - contains statistics about the evolution
  - file starting with `last` - contains the last population
  - file that contains the best agent
- `evo-evaluate` - evaluate the evolution results and print sorted results

A.2 Plotting evolution results

For plotting evolution results, we use the `eva_stats_plotting.py` script. This script requires the `matplotlib`, `numpy` and `pandas` packages. The script has two

arguments:

- `directory` - directory with evolution results
- `exp_ids` - list of experiments to plot, a file name prefix can be specified

A.3 View game log

For viewing the game log (json file), we use the `logs_viewer.html` html file. This log file is created, for example, when we run `mcts-agent-tests`. Then we select a log file and the log is shown in the browser. By the left and right arrow keys, we can move between turns. By the enter, we can show/hide the game summary.

Appendix B

Attachment

The attachment of the electronic version contains the following folders:

- `build` - contains the built application
- `LogsViewer` - contains the html file for viewing game logs
- `UnstableUnicorn`
  - `TestPluginDeck` - contains sources of test deck for the plugin
  - `UnstableUnicornCore` - contains sources of the implementation of the game
  - `UnstableUnicornCoreTest` - contains unit tests for the game implementation
- `eva_stats_plotting.py` - script for plotting evolution results