Developing an AI Agent for the Urban Rivals Card Game

(Opracowanie agenta AI dla gry karcianej Urban Rivals)

Marcelina Oset

Praca inżynierska

Promotor: dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

3 września 2024

Abstract

The goal of the thesis was to develop an artificial intelligence agent capable of effectively competing against human players in the Urban Rivals collectible multiplayer card game. In subsequent chapters, I will describe a detailed overview of the game rules, the idea of the algorithmic approach I used for AI, and other implementation aspects. Finally, I will show the results of the experiments on their efficiency and effectiveness, conclusions, and the opportunities for further improvements.

Celem mojej pracy było opracowanie agenta AI, będącego w stanie efektywnie konkurować innymi graczami w grze karcianej grze multiplayer Urban Rivals. W kolejnych rozdziałach opiszę szczegółowo zasady tej gry, podejście algorytmiczne użyte do przygotowania AI, oraz pozostałe aspekty implementacyjne. Na koniec przedstawiłam także wyniki eksperymentów dotyczących wydajności i efektywności mojego programu oraz podsumowanie i możliwości dalszego rozwoju tego projektu.

Contents

Chapter 1

Introduction

1.1 The goal of the project

Urban Rivals is a collectible turn-based card game with incomplete information designed for a human player. Implementing an AI agent constitutes a particularly intriguing research field for two main reasons - the multitude of cards containing a large number of information is an exceptional implementation challenge due to the need for time optimization and the choice of algorithm for searching the game tree is not an obvious assignment. This work aims to examine these issues and present potential solutions.

1.2 The thesis outline

Chapter 2 offers an in-depth analysis of the game and its governing rules. Chapter 3 delves into the design and implementation of the AI agent, including a description of Python and C++ implementations, inter-component communication, and the underlying algorithmic ideas. Chapter 4 presents the results of the conducted experiments, while Chapter 5 provides a summary and general conclusions about the project.

Chapter 2

Urban Rivals

Urban Rivals is an online massively multiplayer card game created by Acute Games in 2006 [1]. An important part of this game is the market allowing trading cards between players. The game currently contains a collection of over 2,300 cards. Each card embodies a character, frequently inspired by popular culture or a real-world person, and is affiliated with one of the game's clans, of which are currently 31. The game allows the user to play in various modes, e.g. tutorials, games against an AI opponent, and with other players.

2.1 Game Rules

Figure 2.1: The first turn of the game. The opponent's cards are always in the upper part of the board, and the player's cards are in the lower part.

In each game, the player is given a starting hand of four cards, randomly drawn

from a previously prepared deck consisting of 8 cards. Additionally, they get fourteen life points and twelve pillz – a resource, which mechanics will be detailed further. Each card possesses two statistics: power and damage. Moreover, every card has at most one bonus and one ability. Bonus is only activated if the player's hand contains at least one other card belonging to the same clan.

Figure 2.2: An example card from the game. In the upper left corner of each card, is an icon of the clan to which it belongs, and right next to it is the card's name. At the bottom of the card, there is a number representing power, and below it a number representing damage. Next to them are skill (higher) and bonus (lower). Amount of stars represents the level of the card.

The game consists of four rounds, with players alternating turns. If a player starts a given round, they will move second in the following round. During each turn, players select a card from their hand to play and then assign pillz (which later will be used for calculating the attack) and fury (for the cost of three pillz, the player can increase the card's damage by two points) to the card.

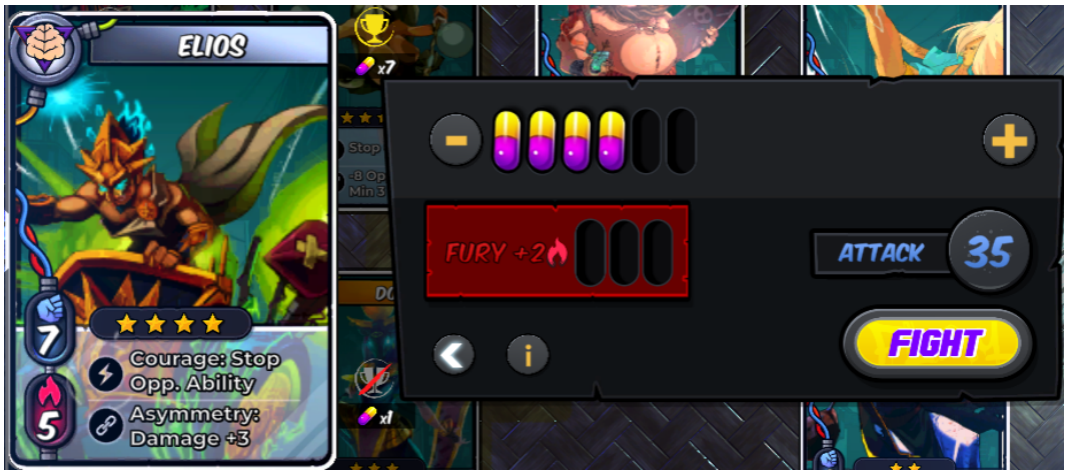

Figure 2.3: Assigning pillz to the card.

The second player knows only which card the first player has chosen. Once a card is selected it cannot be used again. When both players have made their

moves, the round is resolved. The attack is calculated (each player's card's power is multiplied by the number of pillz assigned to this card). In addition, all applicable abilities and bonuses are activated (as detailed in the following subsection). The player with the higher attack points wins the round, in case of a tie, the round is won by the card with the lower level (stars). If the levels are also identical, the player who started the round is declared the winner.

Figure 2.4: Cards' battle view.

The game ends after all cards have been played or one player's health points (HP) reach 0. The winner is the player, with the higher HP. If both players have the same HP, the game is a draw[2].

2.2 Cards keywords

As mentioned in the previous section, every card has at most one ability and one bonus. These skills are defined by a set of key terms. I will explain the meaning of the most important of them below. Keywords can be split into some categories.

1. Straightforward abilities manipulating statistics such as power, attack, damage, life, and pillz. They can contain two of them at once. Sometimes they have a minimum or a maximum value. We can distinguish several basic targets:

   - `Life` – e.g. *+2 Life*
   - `Opp. Life` – e.g. *-3 Opp. Life, Min. 2*
   - `Pillz` – e.g. *+3 Pillz, Max. 9*
   - `Opp. Pillz` – e.g. *-2 Opp. Pillz And Life, Min 4*

Figure 2.5: The second turn of the game. Darkened cards mean that they were used in previous turns. Each card shows how many pillz were used on it and the victory icon if the card won its turn. The opponent started this turn – the card they selected is marked with yellow triangles.

- `Power` – e.g. *Power +1*
- `Opp. Power` – e.g. *-1 Opp. Power, Min. 1*
- `Attack` – e.g. *Attack +16*
- `Opp. Attack` – e.g. *-13 Opp. Attack, Min. 3*
- `Damage` – e.g. *Damage +6*
- `Opp. Damage` – e.g. *-2 Opp. Damage, Min. 3*

2. Long-lasting abilities that modify certain statistics for each subsequent round following their activation. They also can have a minimum and maximum value of player or opponent life, when they are not triggered. This ability triggers if the player wins the round, as long as there are no other conditions. Examples:

   - `Poison/Toxin` – opponent loses X life at the end of every following round. The `Toxin` is applied immediately, whereas the `Poison`'s effect begins in the next turn.
   - `Heal/Regen` – player regenerates X life at the end of every following round. The `Regen` is applied immediately, whereas the `Heal`'s effect begins in the next turn.
   - `Dope` – player regenerates X pillz at the end of every following round.
   - `Consume` – opponent loses X pillz at the end of every following round.
   - `Corrosion` – opponent loses X life points multiplied by round number, starting at 1 at the end of every following round.

Abilities within this category that reduce an opponent's life points cannot be stacked. For instance, if a new poison effect is applied, it will replace any existing poison. `Toxin`, `Corrosion`, and similar effects are also classified as types of poison in that context. The same is applied to abilities that increase player life points or player pillz. But e.g. `Heal` and `Poison` can be used on the same player simultaneously.

3. Conditions. This category contains conditions other than victory of a card, that enable certain skills to be activated. They can be categorized into those that apply at the start and the end of a round, which is beneficial observation for implementing the AI agent's game engine.

   Preconditions:

   - `Courage` – activated if the player is first this round.
   - `Reprisal` – activated if the player is second this round.
   - `Symmetry` – activated if the card chosen by the opponent is directly above this player's card.
   - `Asymmetry` – the opposite of symmetry, activated only if the opponent card isn't directly above this player's card.

Figure 2.6: Example card that contains two different conditions.

   - `Confidence` – activated if the player won the previous round
   - `Revenge` – activated if the player lost the previous round
   - `Disunion` – activated when at least one card in the player's hand is from a different clan than the selected one.
   - `Unison` – activated when all cards in the player's hand are from the same clan.
   - `Stop` – activated if opponent's card has the ability *Stop Opp. Ability.*

- `Day/Night` – some cards have different abilities depending on the time in the game – they change every 4 hours in real life.

Oculus is one of the clans in the game having particular abilities and bonuses. This clan can "infiltrate" others and be considered a card from this clan (and activate its bonus). If all three other cards in the player's hand belong to the same clan, the infiltrating card merges with that clan. If there are two cards from one clan and one card from another, the infiltrating card joins the clan represented by the solitary card. If every card is from a different clan or the player's hand contains more than one Oculus card, none are infiltrated.

Oculus-related preconditions:

- `Clan:` activated if the infiltrated clan is among those listed on the card.
- `Versus clan:` activated when the card is played against an opponent's card from one of the specified clans.

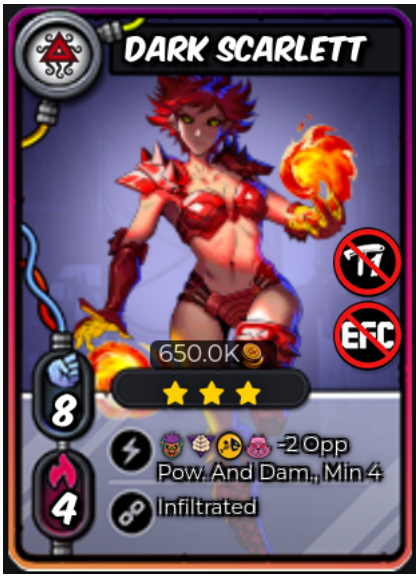

Figure 2.7: Example card from the Oculus clan. If *Dark Scarlett* infiltrates one of the listed clans, and ability *-2 Opp. Power And Damage, Min. 4* is applied.

Postconditions:

- `Backlash` – if the card with this condition wins the round, that card negatively affects the owner. These cards tend to be exceptionally powerful, and this ability serves as a counterbalance.
- `Defeat` – activated if the player lost the round.
- `Victory Or Defeat` – activated both upon winning and losing the round.
- `Killshot` – activated when the player wins the round with an attack at least twice as high as their opponent's.
- `Reanimate` – activated when player health reaches zero, preventing losing the game.

- `Perfect` – activated if the number of pillz is exactly as high as needed to win this round.

4. Skills that allow to copy or modify another player's statistics, abilities, and bonuses.

   - `Cancel Opp. X Modif` – prevents changing a given target `X` such as an attack, damage, pillz, etc. by the opponent's skills for both players.
   - `Protection X` – similar to `Cancel Opp. X Modif`, but stronger. Prevents the opponent from modifying player statistics or stopping abilities and bonuses, depending on `X`.
   - `Stop Opp. Ability/Stop Opp. Bonus` – the opponent's card ability or bonus is canceled. There is no set order for the application of these. To deactivate an opponent's skill (ability or bonus), verifying if another skill is preventing its activation recursively is necessary.
   - `Copy X` – allows duplicate characteristics like an opponent's ability and bonus, but also attack, power, or damage, depending on what is `X`.
   - `Power/Damage Impose` – after the card with this ability is played, set the opponent's power or damage as the same as the player who owns this card.
   - `Power/Damage Exchange` – after the card is played, swap the player and opponent cards' power or damage.

5. Multipliers – abilities that change their value depending on other statistics. Most of them have condition-like structures.

   - `Growth` – the card becomes stronger with each successive turn, its affected statistic increasing multiplicatively with the turn number. For instance, the *Growth: Pillz +1* mechanic provides 1 extra pillz in the first round, scaling up to 4 extra pillz in the fourth round.
   - `Degrowth` – the card is strongest in the first round, and then becomes weaker with each successive turn. The affected statistic decreases proportionally to the remaining number of turns. For example, the *Degrowth: Pillz +1* mechanism grants 4 pillz in the first round, gradually decreasing to 1 pillz in the fourth round.
   - `Equalizer` – the card effect is multiplied by the level (number of stars) that the opponent's card has.
   - `Support` – the card effect is multiplied by the number of cards from the same clan as this card in the player's hand.
   - `Brawl` – the card effect is multiplied by the number of cards from the same clan as this card in the opponent's hand.
   - `X Per Y` – increases statistic `X` based on statistic `Y`. For example *+2 Attack Per Opp. Damage*, or *+1 Power Per Life Left Max. 7*.

- `Recover X pillz out of Y` – returns the player X/Y multiplied by the number of pillz used in this round at the end of it. This number is rounded to the nearest integer less than or equal to itself. For example, if the ability is *Recover 1 Pillz Out Of 3* and the player used 6 pillz, they have 2 pillz returned.

6. Other skills that are limited to one clan or character. A distinctive representative is the leader clan, which features numerous cards possessing unique abilities that defy the conventional rules of the game. An interesting example is the *Genesis* card from the Oculus clan with the `Beyond` skill, which allows you to play the fifth turn with cards randomly selected from the remaining cards in the players' decks if the number of players' life points has not dropped to 0 after the fourth turn. However, these special abilities will not be the focus of our discussion within the context of the AI agent. Below are some of the other most interesting examples:

   - `Tune-out` – the attack value is ignored and the winner of the round is the player who used more pillz this round.
   - `Team` – every card on the player's hand also has this enhancement. This ability is immune to *Stop Opp. Ability*.

2.3 Urban Rivals API

Urban Rivals provides an API[3] (Application Programming Interface), a set of requests it can handle. Anyone can apply for a consumer key granting access to the public API (all character details, market history) and user API (fight logs, collection).

Chapter 3

Implementation of the agent

3.1 Getting information about current game

As mentioned in the previous chapter, most information is provided from the Urban Rivals API. The card data (level, power, damage, abilities, bonuses, and clan) was downloaded from the public API as a `.csv` file and loaded at the beginning of the Python script responsible for communication with the game.

Moreover, this Python script uses elements of screen recognition (Python `cv2`[4] module) to detect if it is a player turn or if it can play again after a game is finished.

The ability parser can be found in the `AbilityParser.py` file. The ability parser uses the idea from the previous chapter that we can group the keywords into categories. This allowed it to recursively analyze and parse each skill piece by piece using regular expressions and convert the abilities description to the `JSON` format (using Python `regEx` module).

The program does not send back any information via API. Instead, it simulates user input by clicking on specific coordinates to select cards and apply pillz or fury (using `pyautogui` module to operate the mouse).

3.2 Python communication with C++

The core component of the project, responsible for determining the optimal player moves was implemented in C++. This language was chosen considering its advantages such as speed and efficiency, which are crucial for real-time calculations.

The development of programs in two distinct languages created the need to use something to connect them. For this purpose, the Python `subprocess` module was selected, which, through the `subprocess.Popen()` constructor and its associated methods, enabled the execution of the C++ program from within Python and the

exchange of data between the two programs via standard input, output, and error streams. This enabled the transmission of the battle data in `JSON` format to the C++ component.

The only additional library used for the C++ program was `nlohmann-json`[5], which enabled the convenient and efficient parsing of data transmitted by the Python script.

3.3 Implementation of the engine

The project uses the `Bazel`[6] build tool. The project contains the configuration files for this tool and the `src` directory, which includes C++ files with classes and functions with the AI agent logic.

The most important files that `src` directory contains:

- `abilities.cpp` and `abilities.hpp` – The files containing the `Ability` structure and various enums associated with it, as well as functions for converting them to string and string to enum.

```
struct Abilities {
    std::bitset<17> conditions;
    uint64 targets;
    uint64 cancel_targets;
    std::array<int8, 3> amount;
    Stop stop_target;
    std::vector<uint16> oculus_ability_clans;
}
```

Abilities using `std::bitset` for conditions, and `uint64` for both simple targets and the targets for canceling modifiers to use bitwise operations. Both targets and conditions have enums associated with them. Here are the examples:

```
enum Conditions : uint8 {
    COURAGE,
    REPRISAL,
    ASYMMETRY,
    SYMMETRY,
    CONFIDENCE,
    REVENGE,
    DISUNION,
    UNISON,
    CLAN,
    VERSUS,
    BACKLASH,
    DEFEAT,
    VICTORY_OR_DEFEAT,
    KILLSHOT,
    REANIMATE,
    PERFECT,
    STOP,
    DAY
    NIGHT
};
```

```
enum Targets : uint8 {
    LIFE,
    OPPLIFE,
    PILLZ,
    OPPPILLZ,
    POWER,
    OPPPOWER,
    ATTACK,
    OPPATTACK,
    DAMAGE,
    OPPDAMAGE,
};
```

This enables efficient bitwise operations, resulting in substantial performance gains for the program. Some skills have an associated value by which they modify a given statistic and a minimum and maximum to determine how much this statistic can be modified. They are kept as `uint8` in a three-element `std::array`.

- `card.hpp` – the file containing the implementation of the `Card` structure and its constructors.
- `player.hpp` – the file containing the implementation of the `Player` structure – variables containing hp, pillz, cards, and information about a card was already used, and one method `do_move` that, based on the player's move, modifies variables containing details whether a given card has been used or how many pillz the player has left to use.

```
void do_move(Move move) {
    assert(used[move.card] == 0);
    used[move.card] = 1;

    assert(pillz >= move.pillz);
    pillz -= move.pillz;

    if (move.fury) {
      assert(pillz >= 3);
      pillz -= 3;
    }
}
```

- `move.hpp` – the file containing the implementation of the `Move` structure and its constructors.
- `main.cpp` and `main-test.cpp` – files with the game's main loop with two distinct purposes. The first is designed to load and write data in bytes to communicate with a Python script. The second file is utilized for local testing of the AI agent, accepting string data in JSON format via standard input and returning the calculated move and the value assigned to it by the search algorithm to standard output.
- `search.hpp` and `expectimax.hpp` – The files containing implementations of search algorithms, that are elaborated upon in the following section.
- `state.cpp` and `state.hpp` – files that contain `State` structure with information such as instances of player structure for the player and the opponent. The most important methods of this class are:
  - the function that reads and parses input in JSON format
  - the function calculating the legal moves for the player in a current state

- the function for evaluating the current game state
- the function checking if the state is terminal

```
bool State::is_terminal() {
  return (player.used.all() && opponent.used.all()) ||
         (player.hp == 0 || opponent.hp == 0);
}
```

- a function that allows updating the state and calculating the outcome of the round and its auxiliary functions such as checking the conditions, calculating player attack, damage, life, and pillz, applying *Stop Opp. Ability* resolving *Stop* order and loops and *Cancel Opp. X Modif.*

For example, this function uses bitwise operations effectively and can modify specific statistics based on function arguments in a generalized way.

```
uint8 State::activate_ability_for_target(Abilities &ab, Targets target, uint8 val) {
  if (ab.targets & (1 << target)) {
    val = std::clamp((int8)(val + ab.amount[Amount::VALUE]), ab.amount[Amount::MIN],
                     ab.amount[Amount::MAX]);
  }
  return val;
}
```

Here is the part of the code used for canceling opponent modifiers for a given statistic and it uses bitwise operations.

```
if (player_ability_active && player_card.abilities.cancel_targets > 0) {
    opponent_card.abilities.targets &= ~player_card.abilities.cancel_targets;
    opponent_card.bonus.targets &= ~player_card.abilities.cancel_targets;
}

if (player_bonus_active && player_card.bonus.cancel_targets > 0) {
    opponent_card.abilities.targets &= ~player_card.bonus.cancel_targets;
    opponent_card.bonus.targets &= ~player_card.bonus.cancel_targets;
}

if (op_ability_active && opponent_card.abilities.cancel_targets > 0) {
    player_card.abilities.targets &= ~opponent_card.abilities.cancel_targets;
    player_card.bonus.targets &= ~opponent_card.abilities.cancel_targets;
}

if (op_bonus_active && opponent_card.bonus.cancel_targets > 0) {
    player_card.abilities.targets &= ~player_card.bonus.cancel_targets;
    player_card.bonus.targets &= ~opponent_card.bonus.cancel_targets;
}
```

The function below checks if the player's abilities and bonuses are active and applies the previously mentioned function to calculate the attack. Functions for calculating power, damage, life points, and pillz are implemented analogically to this function.

```
uint8 State::calc_attack(Card &player_card, Card &op_card, uint8 player_power,
                         uint8 player_pillz) {

    uint8 attack = player_power * (1 + player_pillz);

    if (player_bonus_active) {
        attack =
          activate_ability_for_target(player_card.bonus, Targets::ATTACK, attack);
    }
    if (player_ability_active) {
        attack =
          activate_ability_for_target(player_card.abilities, Targets::ATTACK, attack);
    }
    if (op_bonus_active) {
        attack = a
          ctivate_ability_for_target(op_card.bonus, Targets::OPPATTACK, attack);
    }
    If (op_ability_active) {
        attack =
          activate_ability_for_target(op_card.abilities, Targets::OPPATTACK, attack);
    }
    return attack;
}
```

3.4 Playing algorithm

The program employs two distinct search algorithms based on the round number. Both searches are based on the min-max algorithm augmented with alpha-beta pruning, but they differ in underlying assumptions and the methodology employed for calculating the result.

1. An algorithm with the assumption that the opponent possesses knowledge of the selected pillz count, while the player remains unaware of the opponent's choice.

   If the player starts the turn the search algorithm goes through two phases:

   - Phase 1: Player choosing card, pillz, and fury.
   - Phase 2: Opponent choosing card, pillz, and fury, fully knowing the player's move.

   If the opponent starts the turn the search algorithm goes through three phases:

   - Phase 1: Opponent choosing the card.
   - Phase 2: Player choosing card, pillz, and fury, knowing only the opponent's card.
   - Phase 3: Opponent choosing the pillz and fury, knowing the player's move all details.

The algorithm looks for the best result for the player by recursively calling functions corresponding to subsequent phases. This strategy benefits from a relatively low branching factor, enabling the algorithm to efficiently calculate the solution even at a depth of 4 (each round is considered one depth more in this approach). However, the drawback of this solution is that it only defends itself against the most dangerous opponent moves (according to the evaluation function), which is why it is not always the best choice. That is why the AI agent uses this algorithm only in the first round.

2. An algorithm that computes the opponent's average score for playing a specific card, considering all potential pillz configurations. This approach is a modification of `expectiminimax` algorithm. Searching the game tree in each round consists of two phases:

   - If a player is first, chooses a card and pillz, and recursively calculates the value for this move
   - If a player is second, a card is chosen, and the average of the results for all possibilities of assigning pills to this card is recursively calculated

   Implementation is the same for the player and the opponent. This algorithm exhibits slower performance due to a larger branching factor. However, it considers more responses to a given player's move than the worst-case scenario, potentially yielding a closer to the optimal solution than the previous one.

3.5 Game state evaluation function

The evaluation function employs a straightforward approach, beginning with calculating the difference between the player's and the opponent's health points.

```
float State::evaluate() {
  float value = player.hp - opponent.hp;
  if (is_terminal()) {
    if (player.hp > opponent.hp) {
      return 1e6 - 1000 * round + value;
    } else if (player.hp < opponent.hp) {
      return -1e6 + 1000 * round + value;
    } else {
      return 0.f + 0.1f * player.hp;
    }
  }
  return value;
}
```

If the game state is terminal, the following cases are considered:

- The player won, which is determined by having a higher number of life points compared to the opponent, the following calculation is performed – a victorious player receives a base reward of one million points, adjusted by subtracting 1000 points multiplied by the turn number, and adding the value of health point difference. This approach is motivated by the desire of the player to achieve victory in the earliest possible round while maintaining the largest possible advantage in terms of health points.

- The player lost, which is determined by having a higher number of life points compared to the player, the following calculation is performed – a losing player receives a base penalty of minus one million points, adjusted by adding 1000 points multiplied by the turn number and the value of health point difference. This approach is motivated by the desire to, in a losing position, delay defeat as long as possible and minimize the difference in health points, increasing the probability of opponent mistakes and changing the outcome of the game.

- If the game is a draw, which is determined by both players having the same number of life points, the following calculation is performed – the initial number of 0 is adjusted by adding 0.1 times the number of player life points, which is motivated by the fact that it is safer to stay on more life points, just in case the opponent notices a chance of victory that the AI agent did not take into account in the calculations.

The algorithm always calculates results for the maximum depth for a given game state, so the other case was not the subject of consideration. Although the current function structure can readily incorporate such evaluations if required.

Chapter 4

Experiments

In this chapter, I will present the results of experiments using different decks. I will test each deck in two ways. In the first case, only the usual minmax will be used in each turn, in the second case, expectimax will also be used, as described in the section about the playing algorithm.

4.1 Calculation time

In real-time calculations, the algorithm's runtime must be short enough. The table below shows the average computation time for each game turn.

Round	I		II		III		IV	
First player	Me	Opp.	Me	Opp.	Me	Opp.	Me	Opp.
Minmax	23.3 s	10.85 s	69.7 ms	48 ms	0 ms	0 ms	0 ms	0 ms
Expectimax	–	–	1.69 s	1.84 s	0.8 ms	0 ms	0 ms	0 ms

Table 4.1: The average calculation times of both algorithms depending on the turn number and the player who picked the card first. The table does not provide a time for the expecitmax in the first round due to the extensive computational time required (5-20 minutes), so its calculation was not feasible within the given timeframe during a game.

4.2 Tests against the other AI

The first way to test the program was to play games against another AI supplied by my supervisor - the simple rule-based bot, with non-deterministic choices. This AI did not use any search and keyword meaning; only probing self and opponent health points and pillz. More details about this algorithm are described in Appendix A. The tables below show the results of testing bots against each other, using decks consisting only of characters from one clan. Deck details can be found in Appendix B.

For historical reasons AI agents was tested using the decks:

- consisting of less than 25 stars in total.
- consisting of more than 25 stars in total.

The first type of deck will be called limited and the second unlimited in the rest of this work.

4.2.1 Tests with unlimited deck

Results of battles between minmax and rule-based AI:

	Ulu Watu clan	Pussycats clan
Ulu Watu clan	36.7%	43.3%
Pussycats clan	76.7%	56.7%

Table 4.2: Win ratios of minmax-based bot against the rule-based bot. Columns contain clans used by a rule-based bot and rows contain names of clans used by minmax. Each win ratio is calculated for the minmax-based bot, based on a sample of 15, and draws were counted as half-wins.

Results of battles between expectimax and rule-based AI:

	Ulu Watu clan	Pussycats clan
Ulu Watu clan	6.7%	20%
Pussycats clan	70%	53.3%

Table 4.3: Win ratios of the bot using expectimax against the rule-based bot. Columns contain clans used by the rule-based bot and rows contain names of clans used by expectimax. Each win ratio is calculated for the expectimax-based bot, based on a sample of 15 games, and draws were counted as half-wins.

The results of an agent using minmax are similar to a rule-based bot. Conversely, the expectimax algorithm demonstrates unsatisfactory results, particularly when applied to certain deck configurations.

4.2.2 Tests with limited deck.

Results of battles between minmax and rule-based AI:

	Komboka clan	Montana clan
Komboka clan	40%	56.7%
Montana clan	56.7%	40.0%

Table 4.4: Win ratios of minmax-based bot against the rule-based bot. Columns contain clans used by the rule-based bot and rows contain names of clans used by minmax. Each win ratio is calculated for the minmax-based bot, based on a sample of 15 games, and draws were counted as half-wins.

Results of battles between expectimax and rule-based AI:

	Komboka clan	Montana clan
Komboka clan	40%	50%
Montana clan	36.7%	13.3%

Table 4.5: Win ratios of the bot using expectimax against the rule-based bot. Columns contain clans used by the rule-based bot and rows contain names of clans used by expectimax. Each win ratio is calculated for the expectimax-based bot, based on a sample of 15 games, and draws were counted as half-wins.

4.3 Test against the human players

The program was also tested in the Daily Tournaments (Tourney) mode, in which human players compete with each other. Tournaments last half an hour and during that time each player aims to score as many points as possible. The deck consists of 8 cards and the sum of their levels cannot exceed 32. The player receives 50 points for a win, 10 points for a draw, and 6 points for a loss. Players earn additional points for defeating higher-level cards with lower-level cards, executing moves quickly, and if the opponent gives up or timeouts.

The decks' details used for testing the algorithm with human players are shown in Appendix C.

Algorithm used	placement/participants	battlepoints	matches	win ratio
minmax	9/34	592	17	47%
minmax	6/28	445	12	33.3%
minmax	8/40	438	10	55%
expectimax	9/32	504	14	37.5%
expectimax	7/23	456	9	44.4%
expectimax	7/35	457	13	15.3%

Table 4.6: Results for deck with cards including Gheist, Pussycats, and Oculus clan

Algorithm used	placement/participants	battlepoints	matches	win ratio
minmax	7/29	475	11	40%
minmax	10/26	370	11	22.7 %
minmax	7/33	500	12	33.3%
expectimax	8/25	399	10	20%
expectimax	10/31	454	11	13.6%
expectimax	9/29	453	11	27.3%

Table 4.7: Results for deck with cards including La Junta and Berzerk clan

Chapter 5

Conclusions

Even though the program does not consider all possible skills from the cards, a version of the algorithm with a more straightforward min-max approach assuming the worst possibility can compete with a human player and achieve optimistic results. Regrettably, the expectimax methodology turned out to be not useful in the context of this game. Not only does this algorithm exhibit significantly slower execution, but it also yields inferior outcomes.

The expectimax strategy assumes that the player, who is playing the second in a given turn, knows the first player's pillz, and for every card calculates the average of scores for every possibility of assigning pillz to this card. Although, the best move appears to be not as strongly connected to the choice of card, and is much more dependent on the amount of pillz.

The project has three primary avenues for advancement, which are anticipated to improve its future performance significantly. These areas include integrating the remaining keywords associated with abilities and bonuses in the AI agent's game engine, implementing a more nuanced evaluation function tailored to specific game modes, and researching suitable algorithms for traversing the game tree, possibly using elements of game theory.

Bibliography

[1] Urban Rivals Main Page https://www.urban-rivals.com/

[2] Urban Rivals Rules https://www.urban-rivals.com/game/rules/

[3] Urban Rivals API https://www.urban-rivals.com/api/developer/

[4] Open Computer Vision by Intel https://opencv.org/

[5] Niels Lohmann's JSON library nlohmann/json for C++ https://github.com/nlohmann/json

[6] Bazel by Google https://bazel.build/

Appendix A

Rule-based bot description

The algorithm takes as input a list of values denoting the rules for a given round respectively 1, 2, 3, 4. These values can be of the following types:

- float number from 0 to 1 – multiplies the available number of pillz by this value. Additionally, if this value is more than zero calculates this number `-1`, and if it is less than the number of available pillz this number `+1`. After that select randomly one of these, usually three values.
- `None` – meaning that the algorithm should use all pillz that are left to use.
- `(X, Y)` – meaning that the algorithm is supposed to randomly select the amount of pillz from the range `X` to `Y`.
- Dictionary with a key of `'2KO':` the algorithm checks three cases:
  1. If the player's health points value is less than 7 use all of the available pillz.
  2. If opponent health points value is less than 7 health points, with a probability of $\frac{2}{3}$ use all of the pillz; with a probability of $\frac{1}{3}$ select 0 pillz.
  3. Otherwise, use the strategy from the value in the dictionary.

Also, there are the strategies for clans used for testing:

- Komboka clan: `[(3,7), (3,7), None, None]`
- Montana clan: `[(3,6), (3,6), 0.8, None]`
- Pussycats clan: `[(3,7), {'2KO': (3.7)}, {'2KO': 0.5}, None]`
- Ulu Watu clan: `[(3,7), (3,7), None, None]`

Appendix B

Testing decks used with the rule-based bot

Figure B.1: A deck using cards from the Pussycats clan.

Figure B.2: A deck using cards from the Ulu Watu clan.

Figure B.3: A deck using cards from the Komboka clan.

Figure B.4: A deck using cards from the Montana clan.

Appendix C

Testing decks for the Daily Tournaments

Figure C.1: A deck using cards from the clans: Gheist, Pussycats, and Oculus.

Figure C.2: A deck using cards from the clans: La Junta and Berzerk.