Implementing Tales of Tribute as a Programming Game

(Implementacja Tales of Tribute
jako gry programistycznej)

Dominik Budzki Damian Kowalik Katarzyna Polak

Praca inżynierska

Promotor: dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

24 lutego 2023

Abstract

The goal of our project was to create a tool that allows users to write AI bots for „Tales of Tribute" – a card game that is part of the massively popular multiplayer online role-playing video game „The Elder Scrolls Online" (ESO). To the best of our knowledge, this is the first approach to recreate this game outside of the original implementation.

We developed an engine that simulates a game environment and a graphical user interface that allows playing against AI. Such an environment opens up new opportunities not only for casual players (that would like to test their skills outside of ESO) but also for AI developers. In particular, we aim for our project to be a basis for future AI competitions (similar to terminated Hearthstone AI Competition at IEEE CoG conference).

In this thesis, we described the implementation of the engine, UI in Unity, and how to implement bots. We also describe our implementations of a few basic AI algorithms (including MCTS and Beam Search), and run experiments testing their performance.

Celem naszego projektu było stworzenie narzędzia, które pozwoli użytkownikom na programowanie botów do „Tales of Tribute" – karcianej gry, która jest częścią bardzo popularnej wieloosobowej gry fabularnej „The Elder Scrolls Online" (ESO). Według naszej wiedzy, jest to pierwsza próba odtworzenia gry poza oryginalną implementacją.

Napisaliśmy silnik symulujący środowisko gry oraz graficzny interfejs użytkownika, który pozwala na grę przeciwko sztucznej inteligencji. Takie środowisko otwiera nowe możliwości nie tylko przed zwykłymi graczami (którzy mogą przetestować swoje umiejętności poza ESO), ale także przed programistami SI. Chcielibyśmy, żeby nasz projekt był podstawą przyszłych zawodów w pisaniu botów (podobnych do nieistniejącego już Hearthstone AI Competition na konferencji IEEE CoG).

W pracy opisaliśmy implementację silnika, interfejsu w Unity oraz jak zaimplementować agentów przy użyciu udostępnionych przez nas narzędzi. Opisaliśmy także implementację kliku podstawowych algorytmów sztucznej inteligencji (na przykład beam search'a oraz MCTS-a) i przeprowadziliśmy testy mierzące ich efektywność.

Contents

Chapter 1

Introduction

The goal of our project was to create a tool that allows users to write bots for Tales of Tribute – a card game which is part of the massively popular multiplayer online role-playing video game „The Elder Scrolls Online". Our work provides future users with an engine that simulates a game environment and a user interface that allows testing their AI. With these features, users can quickly improve bots' performance in the game. We also created sample bots to test our tool and see how basic AI techniques work in this game.

Games are commonly used as a testbed for artificial intelligence algorithms, improving the process of learning and research. Because of that, some scientific conferences hold official AI competitions (for example, IEEE Conference on Games). In the future, we also want to organize an event based on our project. Our project is unique because it connects the programming community with regular ESO players, so our new potential users can be people without solid computer science knowledge. They can use our tool to improve their skills and test strategies, which is an additional feature inaccessible in the original game.

Writing bots for games can be an exciting and fun experience, even if we ignore the possibility of competition. This is a great opportunity to deepen our understanding of the game, its mechanism and system. It can also improve programming and problem-solving skills. Often, approaches polished on games are then transferred, for example, to business. All these things together make our project needed and useful.

Chapter 2

AI Programming Competitions

AI contests in writing bots for games have become increasingly popular in recent years. These contests provide a platform for a community of players and computer scientists to showcase their skills in programming and AI by creating bots that can play a given game. There are both hobby-level solutions and reserach competitions that take place regularly.

Probably the best platform for AI contests is CodinGame[1], which regularly holds competitions for various games. These contests are open to users of all skill levels and allow competing against each other. In contrast to other platforms, the CodinGame community perfectly balances being competitive and helpful for novice persons. These contests are not only about writing bots because, in the past, some problems were also about optimizations or algorithms so everyone could find something interesting. CodinGame also has many problems that can be solved between contests, which makes people attached to this platform.

In addition to the CodinGame platform, AI games contests are held at academic conferences such as the Conference on Games (COG)[2], Congress on Evolutionary Computations (CEC)[3], Genetic and Evolutionary Computation Conference (GECCO) [4]. These conferences connect researchers, students and AI professionals to showcase their work and share knowledge. The contests at these conferences are often more complex and require a higher skill level. However, they also allow researchers to impact the field significantly and are often base for publications.

[1] `https://www.codingame.com/`
[2] https://ieee-cog.org/
[3] http://www.wikicfp.com/cfp/program?id=411
[4] https://sig.sigevo.org/index.html/tiki-index.php?page=GECCOs

2.1 Collectible Card Games

Collectible Card Games (CCGs) are card games where players collect and build decks representing various characters, creatures, spells, and abilities. The first CCG was „Magic: The Gathering", which was created in 1993 and is still very popular today.

CCGs have a few things in common; the most important ones are:

- A large number of cards available and each card has its unique abilities and attributes.
- Significant randomness in game because of large number of cards. As players construct their decks, they must take into account the likelihood of drawing a specific card at a specific time.
- Deckbuilding - players must carefully choose which cards to include in their decks, considering each card's strengths and weaknesses and how they will interact with the other cards in the deck. Deckbuilding requires a lot of strategic planning and decision-making. Tales Of Tribute represents the deckbuilding subgenre, so the deck is built during the game not before as in e.g., Magic: The Gathering, or Hearthstone.

2.2 Dominion

Dominion is considered to be the first representative of the deckbuilding subgenre where the player's deck is build during game. The game was designed by Donald X. Vaccarino and was first published in 2008.

Dominion is similar to Tales of Tribute in many aspects:

- Players start with a small, basic deck of cards and by them, they can acquire more powerful cards to add to their deck.
- There exist *supply* of cards (in Tales of Tribute tavern), which represents the cards that are available for players to acquire during the game. These cards are laid out on the table at the start of the game and can be depleted as players acquire them.
- The win condition is to get a specific amount of victory points before the opponent (prestige in Tales Of Tribute).

This game appears in several academic papers and theses, e.g., [1, 2, 3, 4].

2.3 Hearthstone

Hearthstone is a popular digital CCG which has a large competitive scene. A prime example would be the Hearthstone AI Competition[5], which has been held three times (2018-2020). The competition is designed to challenge developers to develop stepwise the best AI agent for the game. It has been well-received by the AI and gaming communities, especially if we look at the number of participants (about 50 every year). Participants could submit their work in one of the two tracks:

- „Premade Deck Playin'–track: write an agent that will have the best performance on known and unknown decks.
- „User Created Deck Playin" –track: the task is to create a deck that will beat other decks.

Thanks to this contest, several scientific papers were created. [6, 7, 8, 9] This contest would not be possible without the SabberStone[5] engine, written in C# and mainly used to simulate games. In our project, we tried to include the best of SabberStone, which served as our inspiration. Some similarities: the bot must inherit from the *AbstractAgent* class, the ability to simulate a game, a similar Game class that represents a single game, etc.

2.4 Legends of Code and Magic

The game "Legend of Code and Magic" (LOCM) is an example of (CCG) that has gained popularity since its first contest in 2018 on the Codin Game platform. Since then, over 2,500 people have participated in that contest. Since 2019 the game was also featured at the Congress on Evolutionary Computations(CEC) and Conference On Games(COG), where participants could showcase their bot-writing skills.

LOCM was inspired by „The Elder Scrolls: Legends" and there are also slight similarities to the „Hearthstone". Players use a deck of cards representing characters, spells, buffs/debuffs. There are two main phases in this game: deck building where around 30 cards are selected and battle phase where players can summon monsters, use spells and attack.

Based on this game several publication have been published.[10, 11, 12, 13]

[5] `https://hearthsim.info/sabberstone/`

Chapter 3

Tales of Tribute

Tales of Tribute is a two-player deck-building card game released in June 2022 with an expansion „High isle" to the popular MMO RPG game „The Elder Scrolls Online". It features several decks of cards that differ in playing style. Each deck is represented by a *Patron*. These decks can be acquired during ESO gameplay.

3.1 Overview of the Game rules

At the beginning of the game, players choose a total of four decks of cards. The starting player chooses first and fourth, while the second player chooses second and third. Then the main part of the game begins. The *Starter* cards go to the players, while the rest are shuffled and found in the Tavern. The player moving second receives one Coin at the start. Each turn player draws five cards from his draw pile. Used cards go to the so-called *Played pile*, with an exception of Agent cards. Once the turn is over Cards that were played go to the *Cooldown pile*. When the *Draw pile* gets empty, and a player needs to draw a card, the *Cooldown pile* is shuffled and put in place of the *Draw pile*.

In Tales of Tribute, there are three main resources: Coins, Prestige, and Power. Coins are used to buy cards from the Tavern or to buy *Patron's* abilities. At the end of the turn unused Coins are lost. Power can be spent to attack the opponent's Agents or, just like Coin, to buy some of the Patron's favor. Unspent Power transitions to Prestige at the end of the turn. Prestige acts as the scoring in Tales of Tribute and one of the win conditions is to amass the most Prestige.

3.1.1 Tavern

A tavern is a place where the player currently playing can use Coins to purchase cards. There are always five cards available. After buying one card, another one appears in its place, drawn from a shuffled *Tavern* deck (initially consisting of all

cards belonging to the decks chosen by the players).

3.1.2 Cards

The cards we can play are divided into types:

- Starter cards: Cards that are in the player's pool at the beginning of the game. Each *Patron* deck has one starter card with an exception of the neutral „Treasury" deck, which gives the Player six *Gold* cards with a „Coin +1" effect.
- Action cards: When played, they will carry out the relevant action before entering player's *Cooldown Pile*.
- Agent cards: When played, they will be placed on the board and can be activated every turn to use their effects. They are distinguished by their health, which if reduced to zero will move Agent to the *Cooldown pile*.
- Contract Action cards: Similar to Action cards are put into play as soon as they are acquired but go into the *Tavern pile* after enacting its effects.
- Contract Agent cards: Immediately placed on the board upon buying. When knocked out either by effect or by health equal to 0, they go back to the *Tavern*.

The cards have effects that are activated when played and extended effects that will be triggered when cards from the same deck are combined in the same turn.
There are more than a dozen different effects in the game, some of them very simple, such as „*Coin +n*" which increases the number of Player's coins by *n*. There are also more complicated effects that change the situation on the board, but require the player to make a choice, e.g. „*Knockout agents*", which allows the player to get rid of the opponent's agents, „*Replace n cards in Tavern*" - allows you to replace n cards in the tavern, or „*Destroy cards*" which is a well-known effect in this type of card games, the player has the opportunity to remove some of his cards in order to shrink his deck.

3.1.3 Patrons

Each deck in Tales of Tribute is represented through a *Patron*. Each *Patron* has a special skill that has a cost, but in return provides various benefits. *Patrons*, except for the neutral Treasury, have a "status" of endorsement. They can support any of the players or be neutral. Purchasing a *Patron's* skill results in the support indicator shifting toward the player. Every turn Player can use one *Patron* skill unless he gets an effect that increases the amount of available *Patron* calls.
Patrons available in TalesOfTribute and their skills:

Patron	Activation cost	Effect gain
Ansei	2 Power	1 Coin in this and every turn Ansei favors you
Crows	All Coins, needs atleast 1	Power equal to #Coins -1
Pelin	2 Power	Return an Agent from your Cooldown pile to the top of your deck
Rajhin	3 Coins	place 'Bewilderment' Card in opponent's Cooldown pile
Psijic	4 Coins	Knock out one opponent's active Agent
Hlaalu	Sacrifice one of your cards	Prestige equal to its Coin cost - 1
Orgnum	3 Coin	Gain Power equivalent to number of your cards, depending on favor level. If Favors you already, get Maormer Boarding Patry card in your Cooldown pile
Red Eagle	2 Power	Draw a card

3.1.4 Win Conditions

In Tales of Tribute, you can win the game in several ways. When one player gains at least 40 prestige, the opponent in his turn will have to gain more to stay in the game. If he fails, then he loses. If he succeeds then the game goes into a "Sudden death" situation. At this stage of the game, the player who fails to top his opponent's score in his turn loses. The upper limit is 80 prestige. The player who reaches this limit wins automatically.

A player can also win if he gains the favor of all four patrons in his turn.

Chapter 4

Engine Implementation

The engine is the heart of the project. It is implemented in C#, as a library, and contains all logic related to the game rules, packaged into one interface – `ScriptsOfTributeAPI` – that is used by `ScriptsOfTributeUI` and `GameRunner`.

4.1 Technology

The engine is written in C#. We chose this language because it is modern, fast, easy to pick up and use, and also cross-platform. Furthermore, C# is the native language of the `Unity` game engine. This was also very important to us, because one of the main goals of the project was to provide a convenient UI for the user to test and play against his bots. Due to the limitations of `Unity`, we target .NET Standard 2.1 framework version, which is not the newest and lacks a few features, but it is not deprecated - it is still supported and secure.

We use several libraries in conjunction with the language. `Newtonsoft.Json` [14] library is used to parse JSON. The newest versions of .NET Framework have this capability built-in, however, this is one of the things missing in .NET Standard. For testing, we use `Xunit` [15]. `CLIGameRunner` uses `System.CommandLine` [16] library to parse arguments. This package is still in preview, however, it has many functions and works well for our use case. Furthermore, it is developed by Microsoft, and it will most likely be the standard way to parse arguments in C# command line apps in the future if it is not already.

4.2 Project Structure

In C#, applications are contained inside `projects`. These can be executable files, or libraries. `Projects` live inside a `solution`. A `solution` can have multiple `projects` [17].

In our case, we have a single `solution` that contains tests, `Engine`, `CLI Game Runner`, and our bots in separate `projects`. `ScriptsOfTributeUI` is contained in a completely separate `solution`, because this makes working with `Unity` much easier.

4.2.1 Engine Structure

`BoardManager` is the heart of the engine. It holds all objects related to the behaviour of the game, and acts as a proxy between them and the `ScriptsOfTributeAPI`. For example, when the user asks the `API` to play a card, `BoardManager` makes sure that the `CurrentPlayer` has this card from its hand removed, and `CardActionManager` executes its effects with all pending combos. It uses following objects for executing most of the logic of the game:

- `CardActionManager` – This class is responsible for executing effects, taking into accounts combos, and handling choices.
- `List<Patron>` – `Patrons` that are in the current game. They can be called to perform effects associated with them, and contain information such as which player they favour.
- `Tavern` – Representation of `Tavern`, it holds currently available cards, and a queue of upcoming cards that will replace the card that is bought. It also exposes some helper functions for things such as replacing a card.
- `PlayerContext` – Holds both current and enemy player and makes sure to swap them at the end of each turn. `Player` objects contains fields such as the list of cards representing the player's hand or his agents, and also exposes some functions for things such as discarding cards.

`CardActionManager` is the most complex out of all of these classes. It also uses several internal classes to handle things such as combos and to decide what to do as a follow up to a resolved choice.

Most of the classes that hold information, such as `Tavern`, `Player`, or `Choice` have corresponding read-only objects that expose copies of relevant fields, such as the list of available cards in case of `Tavern`. These objects are then used as part of game state representation.

The `API` is not only a proxy for `BoardManager`. It is responsible for handling and gracefully representing errors, and also has some functions of its own, like generating list of possible moves.

Other than that, the `AI` namespace contains the interface for bots to use, and also the `ScriptsOfTribute` class. This is the object used to run games between the bots with features like logging or timeouts.

Later sections of this chapter go into more detail about many of these classes.

4.3 Overview of Important Objects

This section introduces some objects that are used in the engine and knowing them is important for understanding other sections.

- `EndGameState` – contains information about how the game ended. It has a `Reason` field that indicates why the game ended, which can for example be `TURN_TIMEOUT` or `INCORRECT_MOVE`. It also contains `ID` of the winning player (unless the game ended to reason such as an internal failure) and a `string` containing additional context about why the game ended, for example, in case of incorrect move it contains the move and a list of all other correct moves that were possible and should have been played instead. `API` functions often return `EndGameState?` – in most cases, it is null, but in case the player makes a mistake, or his move ends the game, this object is returned to indicate this.
- `Move` – represents a move that a player can make. It contains a `Type` field, which can be for example `PLAY_CARD`, `END_TURN` or `ACTIVATE_PATRON`. Depending on the type, it also contains additional information, such as the card that is to be played in case of `PLAY_CARD` move. Moves can be created using static methods in `Move` class, for example: `Move.PlayCard(card)`.
- `Choice` – represents a `choice` that the player has to make. It can be, for example, a choice of which card to discard. Either cards or effects can be chosen, depending on the card played. This object also contains some information about the choice, including all possible items to choose, how many items need to be chosen, or what the effect of the choice will trigger (for example: destroy the chosen cards).
- `ChoiceContext` – this object lives inside a `Choice` and holds additional context, including the `card` and `effect`, or the `patron` that triggered the choice.

4.4 Combos and Choices

Combos and choices were probably the two most complex mechanics to implement in the engine. They are problematic, because they can interact in many complex ways:

- Almost all actions performed by the user can end in either a failure (for example: he plays a card that he doesn't have in hand), a simple success (for example: a card is played and the player gains 1 coin), or a choice. How to represent all these possible results? How to handle the choices?
- Playing cards can trigger combos from previously played cards. That means that playing a single card can result in multiple successes, a mix of successes

and a failure, or, the worst of all: playing a `single` card can result in `multiple` choices.

- All effects triggered by playing a card need to be executed in order, because previous effects can affect the possibilities the in upcoming effects. Consider this scenario: the player plays a card that has an effect that requires him to make a choice to remove a card from `tavern`, but this play also triggers an effect that requires the player to acquire a card in `tavern`. The second effect has to execute after the first one had already completed, because it cannot present the discarded card as one of the possibilities to acquire. And both of these choices can have more simple effects interwoven between them.
- A choice can result in another choice. For example: consider a choice between two effects: `Coin 1` and `Acquire 5`. If the user chooses `Acquire 5`, another choice needs to be resolved immediately - which card to acquire?

In order to solve this, we created an `ExecutionChain`. This class holds a queue of effects that need to be executed after performing an action. This chain is consumed internally in the engine. It enacts all effects in order, and does different things based on the result of that execution:

- If it ends in a success, the loop continues.
- If it ends in a failure, the game ends, because either one of the bots made a mistake, or an internal engine failure occured.
- If it ends in a choice, the execution is paused. This choice is set as a currently pending choice in the engine, and the board state changes to `CHOICE_PENDING`. In that state, the API does not allow any other moves other actions to be performed other than completing the choice. If using `ScriptsOfTribute`, the control is immediately returned to the bot, and he has to make a move of type `Choice`. Even when the player's choice is provided, the execution of this effect is not over – the engine cannot immediately move to the next effect. It needs to first provide the objects that player chose to `ComplexEffectExecutor` class, which, based on the choice's follow-up, will enact appropriate actions. This can also result in another choice, as described above, which would need to be resolved before moving to the next effect.

The implementation of combos is less complicated. Each player has a `ComboContext` object assigned, that resets upon the end of turn. When a card is played, this class is responsible for providing the effects that this move should induce. For each patron, it holds an array of lists of effects. Each item of the array, from one to four, represents activation, or one of the combos, respectively. When the card is played, all its effects are added to appropriate lists - activation gets added to the first list,

combo 3 gets added to the third list. After that, based on current combo counter that increments with each card of the same type played, all the effects from the arrays – up to that current combo counter – are collected and returned. Next, they are added to the `ExecutionChain` and begin executing.

4.5 Game State Representation

The engine needs to be able to represent game state in a concise manner, so that it can be passed to bots so they have knowledge of what is happening, or, in case of `Unity`, it can be used to populate main board view. There are several objects representing the state, but the main one is `FullGameState`.

4.5.1 Full Game State

```
public class FullGameState
{
    public readonly SerializedPlayer CurrentPlayer;
    public readonly SerializedPlayer EnemyPlayer;
    public readonly PatronStates PatronStates;
    public readonly List<PatronId> Patrons;
    public readonly List<UniqueCard> TavernAvailableCards;
    public readonly List<UniqueCard> TavernCards;
    public readonly BoardState BoardState;
    public readonly SerializedChoice? PendingChoice;
    public readonly ComboStates ComboStates;
    public readonly List<UniqueBaseEffect> UpcomingEffects;
    public readonly List<UniqueBaseEffect> StartOfNextTurnEffects;
    public readonly List<CompletedAction> CompletedActions;
    public readonly EndGameState? GameEndState;
    public readonly ulong InitialSeed;
    public readonly ulong CurrentSeed;
    public readonly bool Cheats;
}
```

This object contains all the information needed to recreate a game from a given point in time, including all the cards in players' `hands`, `draw piles`, `cooldown piles`, and `played piles`. It also contains all the `cards` in `tavern`, both available to buy and upcoming. The cards are ordered in a way that they are going to be drawn next, and also current `seed` is stored, so that future card shuffling can be deterministic.

- `PatronStates` field contains information about `patron favour`.
- `BoardState` contains information about what state the board is in – it can be `NORMAL`, or `CHOICE_PENDING` if there is a pending choice that needs to be resolved before making another move.

- `PendingChoice` – if it is not null, that means the next move that player makes should be of type `MAKE_CHOICE`. It contains all information neede to make that choice, including the `effect/patron/card` that caused it, list of possible choices, and the minimum and maximum amounts of choices.
- `ComboStates` maps all patrons to the combo state that they are in. It contains information about how many cards from that patron were already played, and what combo effects are in queue to be trigered by playing next card of that type.
- `UpcomingEffects` is a queue of effects that will be triggered after completing pending choice - these effects usually appear when a `card` with an `effect` that requires a choice was played, and that card also triggered a combo - in that case, the `effects` in queue cannot be executed until the choice is completed.
- `StartOfNextTurnEffects` contains effects that the enemy player must complete before he can make any other moves at the start of his turn. Currently, there is only one such effect in the game, that requires the enemy to choose a card to discard.
- `CompletedActions` contains all actions that were completed previously in the game. This contains information about what `cards` were played, but also what exact `effects` they triggered, which `agents` were attacked, which died and similar. This is useful for bots to quickly be able to check what the opponent did in his turn, but also for human players in `Unity` to quickly see a history what happened in previous turns.
- `InitialSeed` is the seed that was generated at the beginning of the game or was provided in the configuration.
- `CurrentSeed` is the seed that will be used to generate the next pseudo-random number.
- `Cheats` – if this is enabled, simulating moves will not hide cards that the bots should have no knowledge about (see next section).

4.5.2 Limited Game State

`FullGameState` presents a problem. Since it contains so much information, it also contains things that should not be available for a traditional player, or a bot, to see, like the order of upcoming `cards` in `draw pile` and `tavern`. That is why `LimitedGameState` object is introduced. It is designed to be provided to bots so that they only have information available for players in the original game. In this object, sensitive information such as order of `cards` in `draw pile` and `tavern` is is not displayed. Instead, all cards in these piles are sorted lexicographically, so that bots know which cards can be drawn, but have no information about which specific

card will be drawn, which is consistent with the base game. Similarly, enemy player is also modified, since player should not be able to see the hand of his opponent. Instead, `draw pile` and `hand` are combined to a single list, so that the bot cannot see which `cards` exactly the enemy has, but he knows which he might have. Some fields, such as `InitialSeed` and `CurrentSeed` are also hidden, since they could be used to abuse seeded simulations, which are explained in the next subsection. Internally, this class holds the `FullGameState` object, and just exposes some of its fields - directly or altered as explained above.

4.5.3 Simulating Moves

The engine exposes the ability to simulate moves given a `FullBoardState`. This can be done in two ways: a `ScriptsOfTributeAPI` object can be created from the `FullBoardState` and used freely, or, alternatively, the `FullBoardState.ApplyState(Move)` function can be used, that takes a `Move` as a paremeter and returns `FullBoardState` that is the result of applying this move, and a list of possible moves that can be applied to that new state.

Bots can use `LimitedGameState.ApplyMove` function, that internally calls the equivalent function in `FullGameState`, and converts its results back to `LimitedGameState`. However, this poses some challenges.

Usually, bots should not be allowed to see into the future, that is, see what cards they will draw next, or what cards will next appear in the `tavern`. But what if a bot decides to simulate buying something from the `tavern`? The anser is: he will receive a new game state object with that card in the `tavern` bought and already replaced by the next card in the tavern cards queue. This way, the bot could outsmart the system and be able to reverse engineer part of the `tavern` or, similarly, part of his `draw pile`.

In order to solve this problem, we decided to introduce an `Unknown card`. If a bot simulates a move that would give him a knowledge that he should not have, like drawing a card, the card will be replaced with an `Unknown card`. This is a card of unknown type, with unknown `effects`, and such it cannot be played, or bought in a tavern. That way, the bot can still simulate problematic moves, but he cannot gain any additional information from them. That is also why `Cheats` fields exists on the `FullGameState` object – if it is set to true, then cards are not replaced with `Unknown card`.

Because of this limited information policy, the bot also cannot do anything after ending the turn, because he would need to simulate playing as the enemy, which would require information like contents of his `hand`, which should not be available.

4.5.4 Seeded Game State

The limitations when simulating moves are useful for ensuring that the rules and information available to bots are consistent with the base game, however, they also prevent bots from utilising some useful techniques which would require full knowledge of all cards in hand, or the ability to play as the enemy player. This is why we decided to introduce way of simulating games for bots. They can use an overloaded version of the `ApplyState` function, that also accepts a seed. This function shuffles the `tavern` and `draw pile` according to the seed provided, and then applies the move as normal. Since the card order is no longer the same as in the „real" game currently being played (well, there is an unlikely chance that the order is the same, but the bot cannot know this anyway), there is no reason to hide the cards anymore. This function also returns an object of type `SeededGameState` that can be used to continue simulating more moves for the game state originating from the given seed. This way, the bot can play drawn cards and even play as the enemy.

This feature is also the reason why `LimitedGameState` cannot have information about current or initial seed. If it did, the bot could use the current seed to simulate the game from current state (thus, simulating the exactly same game with exact same card order). Even if the bot only had the initial seed, he could try to reverse-engineer the current seed based on the amount of cards drawn so far. This is why that information is hidden from the bot until the game ends.

`SeededGameState` works similarly to `LimitedGameState` - internally, it holds `FullGameState`, uses it to simulate moves and exposes some of its fields, though it does not sort the card lexicographically – it can expose the exact order, since this is not the real game anyway.

4.6 API

This class can be constructed with an array of `Patrons` that should be used for the game and, optionally, with a seed to make the card shuffling deterministic. It contains functions for all actions that can be performed in the game. The interface is as follows:

```
1 public interface IScriptsOfTributeApi
2 {
3     int TurnCount { get; }
4     int TurnMoveCount { get; }
5     BoardState BoardState { get; }
6     SerializedChoice? PendingChoice { get; }
7     Logger Logger { get; }
8     EndGameState? MakeChoice(List<UniqueCard> choices);
9     EndGameState? MakeChoice(UniqueEffect choice);
10    EndGameState? ActivateAgent(UniqueCard agent);
11    EndGameState? AttackAgent(UniqueCard agent);
```

```
12      EndGameState? PatronActivation(PatronId patronId);
13      EndGameState? BuyCard(UniqueCard card);
14      EndGameState? PlayCard(UniqueCard card);
15      EndGameState? EndTurn();
16      List<Move> GetListOfPossibleMoves();
17      FullGameState GetGameState();
18      bool IsMoveLegal(Move playerMove);
19  }
```

Most functions, such as `PlayCard` or `AttackAgent` return an object of type `EndGameState?`. In case the function is called with incorrect input (presumably: the player currently playing made a mistake, like trying to play a card he does not have in hand) current player loses the game and this object represents that. Usually, this object is null - this indicates that everything completed successfully.

`GetGameState` is a crucial function. It returns an object that contains all information required to restart the game from a given point, including: draw piles, cooldown piles, played piles and hands of both players, cards currently available in the tavern, currently pending choice, and more. This object is the basis of the state object provided for bots, but that one is a cut down version of this - for example, bots should not have information about upcoming draws, so draw piles for them are sorted lexicographically. `FullGameState` also allows the `ScriptsOfTributeAPI` object to be recreated from it, which allows starting games from a provided state.

The `API` object was crucial in implementing multiple ways for users to interact with the engine. The logic for calling `AI` to make moves or select `Patrons` could not be reused between `Unity` and the `GameRunner`, so the `API` was created so that game rules can be implemented in one place and reused.

4.7 Cards and Effects

The cards are stored in a `JSON` format, as a list of objects. This JSON was created based on Unofficial Elder Scrolls Pages [18] - a website that contains a lot of unofficial information abot "The Elder Scrolls" game series, including a section about `Tales of Tribute`. An example object looks like this:

```
1  {
2    "id":5,
3    "Name":"Hlaalu Kinsman",
4    "Deck":"Hlaalu",
5    "Cost":10,
6    "Type":"Agent",
7    "HP":1,
8    "Activation":"Acquire 9",
9    "Combo 2":"Remove 1",
10   "Combo 3":null,
11   "Combo 4":null,
```

```
12   "Family":5,
13   "Copies":2,
14   "PostUpgradeCopies":1
15 }
```

The objects contain some trivial information about the card, like its `ID`, `name`, the `deck` it belongs to, or the `cost`. However, some fields are more interesting.

`Activation` and `Combo{2,3,4}` fields contain information about what `effects` this card can induce. The `effects` are parsed to an `Effect` class that contains information about its type and „amount". Since all effects contain a number (that can represent different things like amount of cards drawn or maximum cost) it is very elegant and easy to parse. For example: `Acquire 9` gets parsed to an effect of type `EffectType.ACQUIRE` with `Amount` field set to 9. The effect executor can then look at this information and knows that he should present the player a choice with all `cards` with cost up to 9, so he can acquire one of them. `Cards` can sometimes also have multiple effects, or the ability to choose one of two effects - in that case, we insert `OR` or `AND` between the `effects`. For example: `Acquire 9 OR Coin 8`. Addition of new effect keywords is also easy: we only need to define an enum, and implement the logic in the effect executor.

The `Family` field is used for cards that have upgraded versions. `Family` is the `ID` of the base card – the upgraded card also contains this field. Base cards also have a `PostUpgradeCopies` field. Since upgrades replace a set amount of base cards (usually two of four, or one of two) we need this information to know how many of pre-upgrade cards should be left in the deck. We chose to always include upgraded cards, even though they are something that need to be unlocked in the base game, since they are always strictly better, however introducing an ability to configure this behavior in the future is not out of the question.

The implementation of card definitions as a `JSON` file has many advantages. For example, players can easily introduce their own cards, or modify existing cards. This also makes it easy for us to make small adjustements to cards as balance patches to `ToT` are released, and even more: in the future, we plan to give users the ability to choose on which version of balance patch of `ToT` they want to use in the engine, so that older bots that have no knowledge of newer cards can still play against each other.

4.7.1 Card's Common ID

Each card from `Tales of Tribute` that is implemented in our engine has its own ID. These IDs are stored in an enum. For example: the card "Hlaalu Kinsman" can be referred to as `CardId.HLAALU_KINSMAN`. Thanks to this, in order to create a card, we do not need to do anything like parsing a string with the card's name, since we have enums for each card, which makes for much more readable and less error-prone

code. However, there is a caveat: since there are so many cards, manually creating the enum with IDs would be tedious and it would be easy to make mistakes, so we created a Python script that can be run to generate the enum based on the JSON file with cards.

4.7.2 Unique Cards and Unique Effects

The `Card` and `Effect` objects introduced above are really just blueprints for real cards and effects. Since there can be multiple copies of the same card in the game, we need a way to differentiate between them - especially since these copies could be in different places (one in `tavern`, the other one in `hand`) - so we cannot refer to them just by their common ID. Similarly, there are some effects that need to know the card they belong to (like the `Heal` effect).

To solve this, we introduced `UniqueCard` and `UniqueEffect` classes. Objects of this type are created from their blueprints, and contain all the same information except for one addition: unique cards contain a field with a `UniqueId`, which is different for each card. In case of unique effects, they also contain this field, but the unique ID is the same as their parent card's.

All card blueprints are stored in a singleton called `GlobalCardDatabase`. This object can be used to obtain unique cards based on a card ID provided. This object is not useful for bots though - it always generates a new unique copy of a requested card, so it will always generate a card that does not exist in the game, so if a bot tries to play it, he will lose.

4.8 Patrons

`Patrons` often have more complicated activation conditions and effects, so they are not defined in a `JSON` file but as separate classes extending `abstract class Patron`. Currently, eight `Patrons` are implemented, so all with the exception of the newest one (`The Druid King`), added after we had already started working on the engine.

Adding a new `Patron` is not difficult, but more complex than adding new effects, and it can depend on the complexity of the Patron's powers. Following functions need to be implemented:

- `GetStarterCards` – each `Patron` provides a card that gets added to each player's decks at the beginning of the game.
- `PatronActivation` – the most important function, this is the main effect of the `Patron`, so what happens when user decides to interact with it.
- `PatronPower` – some `Patrons` have effects that are executed every turn – they can be implemented here.

- `CanPatronBeActivated` – checks if the player has resources required to activate the `Patron`.

When creating the `API` object, four patrons need to be provided that the game should use. `Treasury` is an exception – this `Patron` is always present in every game. When using the in-engine game runner, it automatically calls bots to choose `Patrons`.

4.9 Running Games Directly With Engine

We provide two convenient tools to allow users to run games with their bots - `CLIGameRunner` and `ScriptsOfTributeUI`. However, what if the user just wants to include the `engine` in his project, create bots, and run them in the Main function?

`ScriptsOfTribute` is a class that allows the user to conveniently do just that. It can be created with two bots (that is, two objects extending the `AI` abstract class), and then the `Play` function can be used to run a single game. The class also exposes public fields, such as `Seed`, that can be used for configuration. This object is used internally by `CLIGameRunner`, so it contains all the same functionality that the application does.

Additionally, for convenience, `GameEndStatsCounter` is also exposed for the user. It is a utility class that can collect information (in the form of `GameEndState` objects) from multiple played games, compile them into useful fields with statistics, and also provide an elegant way to convert these statistics to a single string that can later be printed. This class is also used internally by the `CLIGameRunner`.

4.10 CLI Game Runner

`CLI Game Runner` (where CLI stands for `Command Line Interface`) is a command line application that allows the user to load bots from DLLs and run games between them.

4.10.1 Usage

In order to be able to use this tool, the user needs to provide a DLL with bots that he wants to use. To create this DLL, the user should create a `library project`, implement the bots there, and then just compile it. The DLL will be available in the build output directory.

The DLL with bots should be moved to the same directory as the `CLI Game Runner` executable. Upon launch, the application will scan the folder, find all valid DLLs,

and the extract all bots from them. The runner requires only two arguments - names of the first and second bot. For example: `GameRunner Bot1 Bot2`. This will launch a single game between the two bots (Bot1 goes first in this case), report the seed used for the game and the result. If the bots were not found in the DLLs, an error will be shown.

The game runner supports several options:

- `--runs <numberOfRuns>` – this option specifies how many games should be played between the two bots. In case this number is greater than one, the seed reported in results will be the seed generated for the first game. Each subsequent game is played with this seed incremented by one. For example, if the user specified `--runs 3` option and the seed `123` was reported, that means that the games were played with seeds `{123, 124, 125}`.
- `-n` – short syntax for `--runs`.
- `--seed <seed>` – specifies what `seed` to use to play the games. If this option is created, no seed will be generated, instead, this one will be used. In case `--runs` option is greater than one, each subsequent game will be played with the seed incremented by one.
- `-s` – short syntax for `--seed`.
- `--threads <threadAmount>` – number of threads to use to run the games. When using this option, if `<threadAmount>` is greater than logical CPU core count, a warning will be shown. Using this option does not change the behaviour of reporting or using `seed`. Logging to `stdout` is not available in this mode.
- `-t` – short syntax for `--threads`.
- `--enable-logs <p1|p2|both|none>` – controls whether logs should be enable. Depending on the option chosen, this allows for the ability to enable the log for one of the players, for both, or to disable them completely. Logs are disabled by default.
- `-l` – short syntax for `--enable-logs`.
- `--log-destination <destination>` - -if this option is set, logs will be saved to files and saved in `destination` directory, instead of sent to `stdout`. This option must be used to support logging in multiple threads. All logs will be saved to files with names conforming to the blueprint: `<seed>_p1.log`, where `seed` is the seed that was used in that game, and `p1` signifies which player generated these logs (it can also be `p2` in case of the second bot).
- `-d` – short syntax for `--log-destination`.

4.10.2 Examples

- Run a single game between two bots.

  Command:

```
GameRunner RandomBot RandomBot
```

  Output:

```
Running 1 games - RandomBot vs RandomBot

Initial seed used: 408779109572556930
Total time taken: 145ms
Average time per game: 145ms

Stats from the games played:
Final amount of draws: 0/1 (0%)
Final amount of P1 wins: 1/1 (100%)
Final amount of P2 wins: 0/1 (0%)
Ends due to Prestige>40: 0/1 (0%)
Ends due to Prestige>80: 0/1 (0%)
Ends due to Patron Favor: 1/1 (100%)
Ends due to Turn Limit: 0/1 (0%)
Ends due to other factors: 0/1 (0%)
```

- Run 1000 games in two threads.

  Command:

```
GameRunner RandomBot RandomBot -n 1000 -t 2
```

  Output:

```
Playing 500 games in thread #0
Playing 500 games in thread #1
Thread #0 finished. Total: 3124ms, average: 6.248ms.
Thread #1 finished. Total: 3168ms, average: 6.336ms.

Initial seed used: 8101434962557867673
Total time taken: 3425ms

Stats from the games played:
Final amount of draws: 1/1000 (0.1%)
Final amount of P1 wins: 502/1000 (50.2%)
Final amount of P2 wins: 497/1000 (49.7%)
Ends due to Prestige>40: 956/1000 (95.6%)
Ends due to Prestige>80: 0/1000 (0%)
Ends due to Patron Favor: 43/1000 (4.3%)
Ends due to Turn Limit: 1/1000 (0.1%)
Ends due to other factors: 0/1000 (0%)
```

- Run 200 games with 12345 seed, and output logs from the first bot to `logs` directory.

```
GameRunner RandomBot RandomBot -n 200 -s 12345 -l p1 -d
    logs
```

4.11 Other Features

4.11.1 Deterministic Games

The API can be created with a seed, which makes the games completely deterministic, as long as the bots are deterministic. `Tales of Tribute` does not have much randomness – in reality, RNG is only used when shuffling the tavern at the beginning of the game, or when shuffling new cards into the player's drawpile. Simulating moves is also deterministic – the bot can either provide his own seed for the simulation, or current seed will be used, but in that case – the cards will be hidden. Since randomness is not of critical importance in the game (that is: we do not need a cryptographically safe random function), we can get away with using a simple RNG function. Thanks to that, we only neeed to remember two numbers when keeping state - current seed, used for generating the next number, and the initial seed.

Even random bots can be deterministic, because they are provided with a `seed` based on the current game seed. If they use this seed for all their random decisions, then games can be 100% deterministic. However, the `seed` provided to bots is not the same as the game seed, in fact it is hidden until the end of the game. The reason for that is that we allow bots to simulate moves with their own seed, and these simulations do not conceal any cards – so, if the bots had access to the initial seed, they could try to reverse-engineer the current seed and simulate some moves, which could give them knowledge that they should not have. To prevent this, the initial seed is hashed using `SHA256`, and the first 64 bits of the hash are used as the new seed provided for the bots.

4.11.2 Logging

The engine provides a `Logger`. The logs produced by the logger contain the name of the player, a timestamp when the log was created, and also current turn and move number within that turn. It can be configured to enable or disable logs for one of the players, or for both. Additionally, the logger accepts an object of type `TextWriter` for each player. This object is an object from C# standard library. It represents a text stream that can be written to. This makes the logger highly configurable - a `TextWriter` can easily be created to point to standard output, or to a file – which is probably what most of the users want – but there are many more possibilities for more flexibility for the end user.

4.12 Testing

We use `Xunit` [15] as the framework for tests. We chose this framework, because it is modern, simple to use and covers all of our use cases. The tests are contained in separate projects in our main solution. In order to create a simple test, the only thing you need to do is add the `[Fact]` attribute to a function that performs some asserts about the code. This framework is also the most popular and recommended out of all competitors, which made our decision even easier.

4.12.1 Testing with FullGameState

`FullGameState` was initially created as an object that should represent the game state with all the information that would be useful to bots to help make them decisions. However, later on, when we were implementing the move simulation feature for bots, this object became capable of playing a game that began in a given state. Then we thought that we could use this object to create convenient module tests. We added a constructor to the class that allows to set all the fields such as the players' hands, or the tavern to the desired values.

That way, whenever we encounter a bug in the engine, we can recreate `FullGameState` object with simplified state of the game that was causing the problem. For example: if a specific combination of cards played in a specific order caused a problem, we could recreate the state with just these cards in a player's hand. Then we could simulate the problematic moves and debug the issue in a controlled environment, with the smallest game state possible, to reduce complexity when diagnosing. When the bug gets fixed, the test can stay to ensure that it does not happen again. Thanks to this invention, we can make sure each bug fix is associated with a corresponding module test. This gives the engine more stability and greatly reduces the chance of accidentally breaking some complex behaviour when making changes.

4.12.2 Testing with Bots

Simple bots, such as the bot that always makes a random move or always performs all possible actions before ending its turn, can be useful not only for demonstrating the engine, but also for testing. As a part of our testing, we run hundreds of games between these bots and catch any games that end due to internal engine failure. Since these bots always choose a move from the list of possible moves and never generate any on their own, the game ending due to incorrect move being made also means that there is an engine problem.

This is a great way of testing, because just the sheer amount of moves the bots can make in a single second makes it very likely to encounter a bug, if it exists. The bots can make some obscure moves that a human player would likely never perform,

such as discarding their decks and all cards in the tavern, which can also lead to edge cases and hidden problems. However, this approach is limited in what types of issues it can find. If there is a bug in the algorithm for generating the list of possible moves that adds some moves that are not legal, we are almost guaranteed to catch it: at some point the bot will try to do something he should not be able to, like trying to play a card he does not have, so the game will end due to incorrect move. On the other hand, if the bug is a subtle difference in behaviour – for example, combos not triggering properly – it can not be caught this way.

Chapter 5

Bot interface

5.1 Game Management

Bot-versus-bot gameplay is controlled in the `ScriptsOfTribute.AI` namespace. The entire game setup starts in the `ScriptsOfTribute` class, to which we pass instances of bots. User is able to set the seed of the game and the output for logs in this class (by default it is the standard output, but the user can set the output to file). User can also define the time within which a bot has to execute its turn, by default we set this value to thirty seconds. The turn limit is an additional form of protection against endless games. In the case of random bots or bots that are not trying to win, we have set a turn limit so that games can end and the user receives information that the bots playing were unable to complete the game – a special enumeration type is used for this. This limit should be set high enough to ensure that we only interrupt the game if the bots truly do not want to win. The default turn limit is 500, after which the game is considered a draw.

After successful configuration, including the selection of patrons, the gameplay moves to the `ScriptsOfTributeGame` class, in which methods that return moves are called on objects representing bots. This class uses the `ScriptsOfTributeAPI` to control gameplay.

5.2 Bot Implementation

To implement your bot playing *Tales of Tribute* on *ScriptsOfTribute*, you need to write a C# class that inherits from the abstract `AI` class from the `ScriptsOfTribute.AI` namespace. In the `AI` class there are three methods that should be overwritten:

- `SelectPatron`: This method is called during the game exactly two times at the stage of selecting *Patrons*. The first argument it receives is the list of *Patrons* available for selection, that is, *Patrons* selected earlier and *Treasury*,

which is a neutral deck and is always in the game, are not included. A second argument is a number that specifies which choice in turn is the current function call. The method returns the ID of the *Patron* the bot wants to select.

- `Play`: It receives a `GameState` and a list of possible moves, that is, all legal moves that can be made. This method returns an object of type `Move`.
- `GameEnd`: This method is called after the game has ended. The purpose of this function is to allow the programmer to analyze the data from the `EndGameState` object as they wish.

5.3 Tools Available to Bots

5.3.1 Simulations

Many algorithms used to create AI for games use some form of search during which they perform simulations and choose their next move based on them. To enable such simulations, the `GameState` object, which is one of the `Play` function arguments arguments, has an `ApplyState` method to which we can pass a `Move` object and get the state of the board after applying this move. This board is independent of the current board on which the game takes place. We have two possibilities to call such a method:

- Without seed: Every draw, either from *the Draw pile* or *Tavern*, is replaced with a special card with ID `Unknown`. This card cannot be played or bought, because the state of the board after performing an action with this card is unknown. The reason for this solution is to hide information that the player should not have access to if he performs simulations in the same seed.
- With seed: Bot sets the seed of `ulong` type for the simulation. Cards in the *Draw pile* and *Tavern* are shuffled with this seed, with an exception of known draws from *the Draw pile*, such knowledge can be gained by using a card with the effect of `Toss` or `Refresh`.

5.3.2 Seeding

For bots that use randomness, we provide a `seed` based on the seed of the current game, and a `SeededRandom` object, based on this seed. If a bot wants to use randomness, but also stay deterministic, he can use these tools. Playing two games between two such bots will always give the same results. Game simulations are also deterministic.

5.3.3 Logs

During the execution of the `Play` method, the user is able to use our logs system to debug his code. The class `AI` has a method `Log` that accepts text of type `string`, which is stored in a variable in the same class of type `List<(DateTime, string)>`, where the first item is the real time at which this method was executed and the second item is the text passed to the method. Then, in the gameplay supervisor class, after the execution of the `Play` method is completed, the contents of the list are written out to the appropriate output and cleaned up to control memory. The logs contain additional useful information, such as a timestamp of when the log was created, and current turn and move number, for more convenient reading and debugging.

5.3.4 Completed Actions

The `GameState` object contains a `List` of previous actions performed in this game. This might be useful if the user would like calculate next draw of the opponent based on the order of cards previously played.

Chapter 6

ScriptsOfTribute User Interface

As one of the agent-testing tools, we created an application in the `Unity` framework [1] that allows the user to conveniently play against bots. This application uses `ScriptsOfTributeAPI` class from our game engine for all calculations. At the beginning of the game and after each action performed by the user, we make a query for the new state of the board and then refresh the entire interface based on the object we received.

The interface was modeled after the original game to ensure intuitiveness for Tales of Tribute players. However, we opted for the simplicity of graphics to make the interface clear so that the user can focus on testing their agent. For example, the cards do not have graphics from the original game, in their place is text describing the effects of the card. We used the space the original game does not use for functional things like tools to help debug bots.

6.1 Presentation

The game was created in 2D with a simple card design so that it has all the information needed (Fig. 6.1). As this game is designed to test and debug our agent, the user has full access to theirs and their opponent's cards. In addition to the visible cards in the bot's hand, user can thoroughly check the *Draw Pile*, *CooldownPile* and *Played Pile*.

[1] https://unity.com

Figure 6.1: The board at the beginning of the game

The choice screens have a simple design. It has a description of what type it is and an upper and lower limit on the number of cards to select (Fig. 6.1). This screen can be minimized if the user wants to check the board state before making a choice.

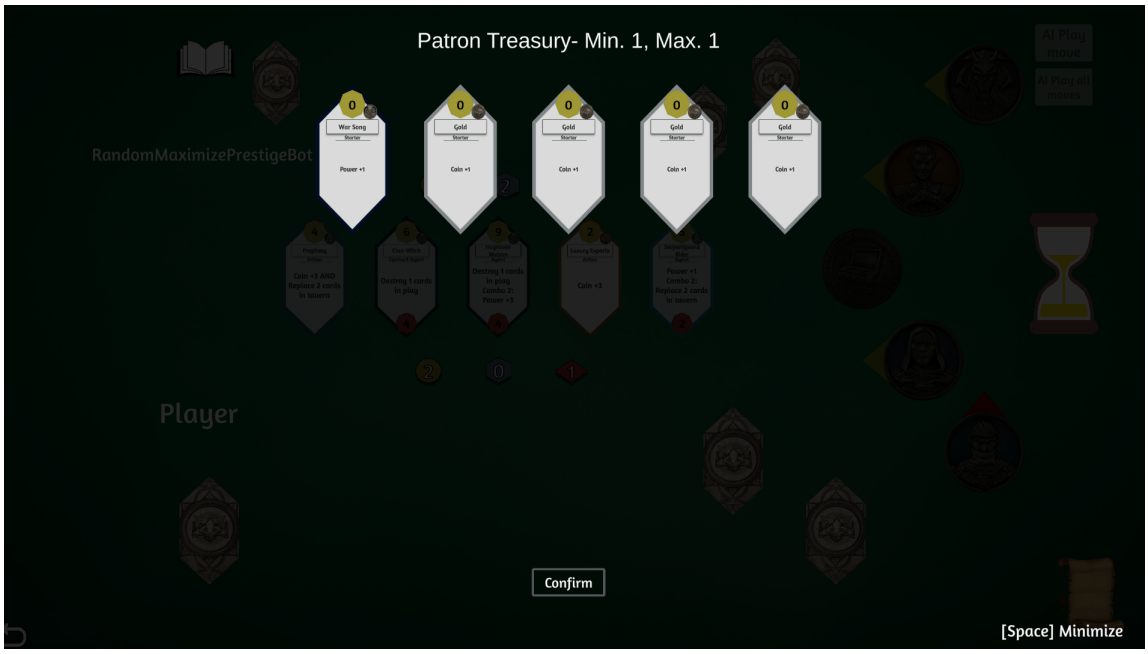

Figure 6.2: Choice screen triggered by *Treasury Patron*

All Patrons have tooltips describing their abilities, and an arrow (gold-colored by default) indicating the player it favors, similarly to Tales of Tribute. If a player in play is unable to activate a given patron, that patron takes on a darker color.

Figure 6.3: *Hlaalu* is in neutral state and can be activated by current player, *Red Eagle* can't be activated and favors us.

6.2 Usage

6.2.1 Adding Bots

To add our bot to the GUI application to play with it, we must meet two requirements:

- Class that represents the bot must inherit from the abstract class „AI" from the `ScriptsOfTribute.AI` namespace.

- We must compile our code into the Class Library (DLL file). We put such a file with the extension `.dll` into the directory `ScriptsOfTribute_Data\StreamingAssets`. This is a special directory from which Unity can conveniently and safely download files. Once this is done, the agent selection screen should show all the bots contained in the DLL file.

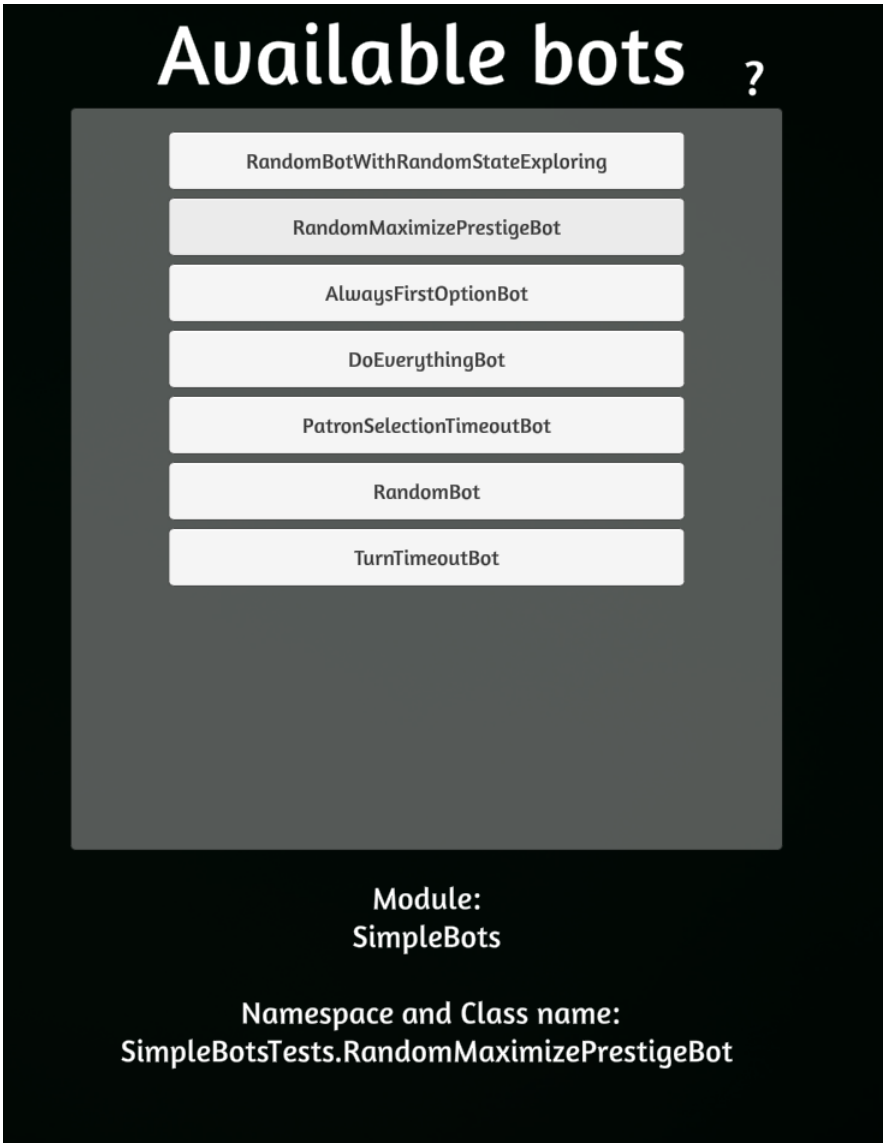

Figure 6.4: List of available bots with additional information.

6.2.2 Game Settings

The application allows the user to set the seed value of the game. In case this field is left blank, the seed will be selected randomly. However, if user would like to play again this random game, on the end game screen user can copy the seed and use it further. Another variable the user can set is the amount of time we grant the agent to complete his turn. The default value is 30000ms. The variable affecting the game is which player starts first. The first player gains access to the tavern, To level the advantage of the first player, the one playing second gets one free *Coin*. The user can decide whether or not to allow the agent to start the game first.

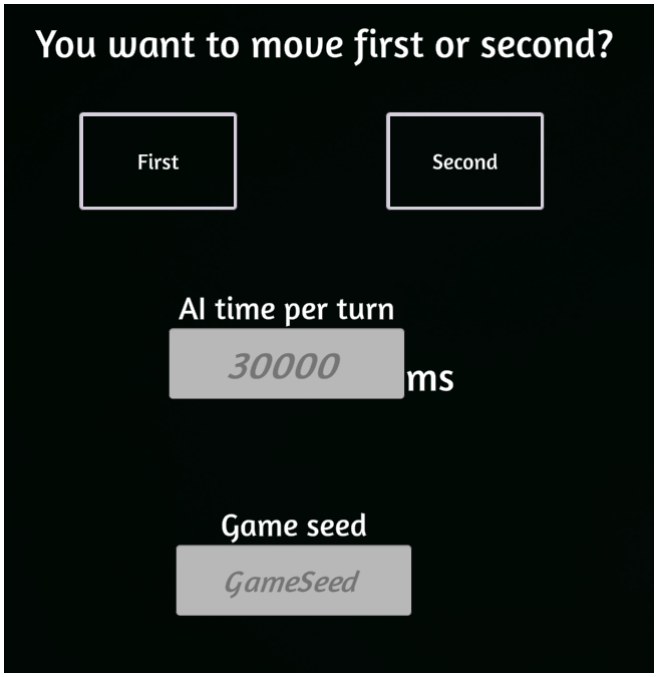

Figure 6.5: Available settings

6.2.3 Bot Moves

During gameplay, when the current turn belongs to the bot, the user is provided with two buttons that call the bot's actions, specifically its `Play` method. The first one „AI Play move" calls the `Play` method of the agent once in another thread. On successful execution, a brief description of what action the bot performed will appear under the button and the board will refresh. If the player wants to proceed all bot moves at once, he can click the „AI Play all moves" button, the program will call the `Play` method in a loop until the returned move does not mean the end of the turn.

Figure 6.6: Buttons and text representing the action performed by the bot

6.3 Analysis Tools

6.3.1 Logs

In order to view the logs that our bot writes out when performing an action, we can use a special panel that can be launched with the book icon button. This panel is scrollable, so it can hold any number of logs. If the user would like to clear the list, he can use the button with the trash can icon.

Figure 6.7: Logs of MaxPrestigeBot – Move object converted to string

6.3.2 Moves History

Another useful tool for analyzing the game is the history of the moves made by both players. It allows us to fully comprehend the events on the board and to track every action and its results. To launch this screen, click the parchment icon in the lower right corner of the application. To close it just click on the background.

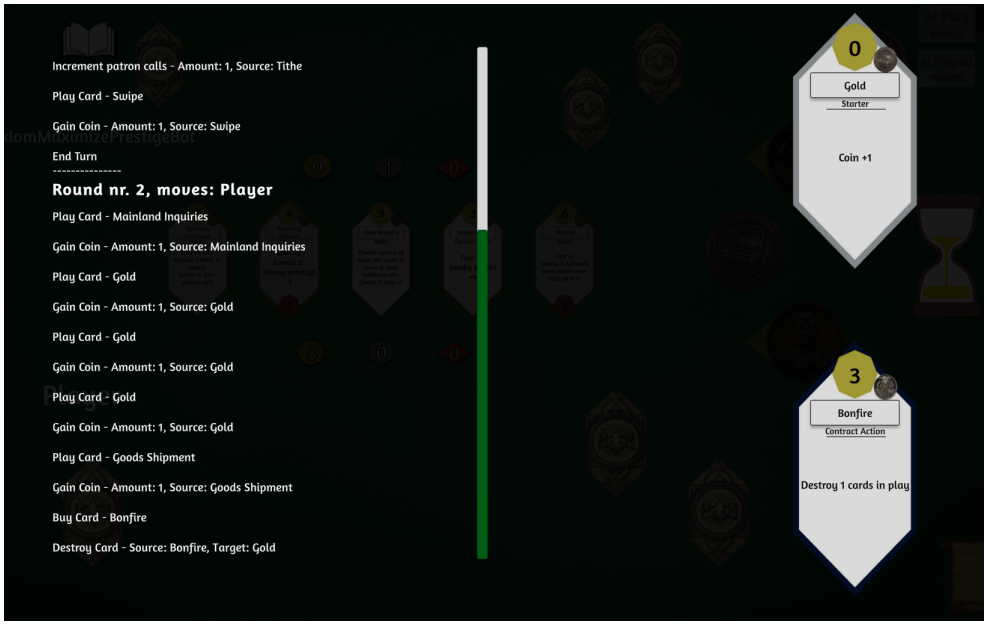

Figure 6.8: Moves history UI – when hovered over action UI displays cards this action relates to.

6.4 Implementation

6.4.1 Game Manager

In `Unity`, the main class that manages all the code is the `GameManager` class. In this class there are methods that change what the user sees on the screen and hears, methods that handle the bot's moves and the user's moves. In the `Update` method from the `MonoBehaviour` class, only the capture of keyboard buttons pressed by the user is implemented. The rest of the code works on the basis of specific actions such as pressing a button. This makes the application lightweight so we do not have to worry about the end user's hardware requirements. After each action, e.g. the user plays a card from his hand, we call the `RefreshBoard` method in a `Coroutine` [2], which is responsible for refreshing the board. This method updates the cards in hand, in the *Tavern*, and on the board in the case of agents, it also checks the status of patrons.

6.4.2 Communication with Engine

`GameManager` holds `ScriptsOfTributeAPI` object to communicate with engine. To make this publicly accessible this field is set to `static`. After the „selecting patrons" phase `GameManager` object creates a new `ScriptsOfTributeAPI` object that will coordinate that current game.

[2] https://docs.unity3d.com/Manual/Coroutines.html

The most important queries to the API are made when refreshing the board and making a move. The human-user movement is parsed based on the tags of the object the user clicked on, e.g. if he clicked on a card from the *Tavern* it is obvious that the player wants to buy this card, for this reason such a card has the tag „TavernCard". Bot movements, on the other hand, are parsed based on type. The corresponding subclass of the `Move` class is parsed and then we execute the corresponding method from the API.

6.4.3 Handling Bot Objects

Since during the game there is only one AI instance we decided to use the *Singleton* pattern to manage bot behaviour. `ScriptsOfTributeAI` class in `Assets\Scripts\AI` contains several fields:

- AI instance which is an instance of the class selected earlier by the user
- Information about timeouts
- Seed
- Move field

This class also contains equivalents of the methods from the AI class from our ScriptsOfTribute engine: responsible for selecting patron and the one making a move. However, we cannot just call these methods every time we need the bot to make a move for two reasons: we have to keep an eye on the time the agent has for a turn, and calling such a method can take a long time, even a few seconds, so we cannot let this call stop the Unity engine from running.

The solution to these problems is to run these methods in other threads. The C# `Task` [19] class allows to run methods in threads, count the execution time and will not freeze the user interface, because we can use `Coroutine` to wait for thread to finish task. Every frame engine will check if task has finished and move is picked. If not, the engine will check again in the next frame. If the task has finished, then we will proceed with the selected move which will be the value of the `Move` field in `ScriptsOfTributeAI` singleton object.

6.5 Building Project

To build our project in Unity, user ought to build project using default settings for his operating system. Since this project works for `.NET 2.1 Standard` library files (`DLL`) have to be compiled for the same .NET version. Pick the destination directory and let Unity do the rest. The most important thing is to add `cards.json` file to the directory with `ScriptsOfTribute.exe` file to let engine read and parse cards. This

`JSON` file can be found inside the project in the root folder or in the GameEngine repository.

Chapter 7

Bot implementation

7.1 Similar Games and Their AI

Designing a bot for playing digital card games is an interesting task. From one perspective, these games are relatively easy to write an engine because they are turn-based and similar to board games. But, on the other hand, because of high randomness, quite big replayable, and a lot of variants in such games, they are demanding and most of the simple algorithms will give a relatively low performance. Moreover, even classical algorithms such as MCTS are not easy to write for such games, and they often need to be changed in some ways to approach this problem.

As Tales Of Tribute is a new game, there is no study research about creating bots or strategies in this game. On the other hand, Hearthstone has recently become a testbed for several AI algorithms. Many problems in that game can be an interesting starting point for AI research. Some of the research mainly focuses on balancing the game (or even metagame – game outside of the game – some decks are strong and because of that, they become popular, it's good to have tactics against them). Balancing often means how nerf or buff cards to make them more equally good. This task is usually approached by some variants of evolutionary algorithms or neural networks [20]. This problem may also be interesting in Tales of Tribute. The other two areas of Hearthstone research are programming a gameplaying bot and deckbuilding. In the case of deckbuilding, these works are state-of-the-art [21, 22, 23, 24]. But because the deck-building phase is a separate and independent phase in Hearthstone (opposite to Tales Of Tribute), we cannot use knowledge from these papers to create our bots. The third field focuses on creating an agent for the game. There are two main approaches: one based on structures like tier list, heuristics, classical algorithms, and human expert rules [25], and the other one try to use machine learning solution [26, 27, 8, 28].In recent years we can observe that these two techniques are combined with good results [29].

7.2 Basic Bots

We started the bot implementations with a few simple ideas that we wanted to compare with each other and choose those that could be useful when it comes to writing more advanced bots. (All the bots described below select patrons randomly from the list of available ones.)

7.2.1 Random Bot

Plays actions by uniformly choosing a random one from the pool of available legal moves. The main problem of this bot is ending the turn prematurely – often the whole turn is skipped as the first move drawn was the `END_TURN`.

7.2.2 Random* Bot

The answer to the main problems of the `Random` bot – plays random moves from the pool of moves minus `END_TURN`. Ends his turn only when this pool is empty. In most cases, playing all the cards from your hand is a safe move. Only in some cases it is better to end the turn without using all cards (e.g., empty *Draw pile*, in hand really good card and card with `TOSS`). I think this bot corresponds to how NPCs play on the Novice level in The Elder Scrolls Online.

7.2.3 MaxPrestige Bot

Probably the first strategy that comes to mind is to try to maximize your prestige (and power, since it has a chance to be converted to prestige at the end of the turn). This bot, while simulating moves, checks all paths of lenghts up to two, and chooses the move for which the sum of prestige and power is the greatest. Unless he finds a move that gives it a win, then the bot chooses that move.

7.2.4 PatronFavors Bot

Most games end with 40+ prestige and it is easy to forget about the other way to win, which is favoring a player by all four patrons. Usually, this win condition is hard to achieve during the game. That is why we wanted to try a bot that mainly focuses on winning by favoring patrons.

In each move, the bot checks whether there is a possible move that activates the patron, which does not favor him yet – if there is such a move, he selects it. Otherwise, the bot checks if it has not used up all possible activations in a turn – in this case, it checks whether it is possible to buy a „Tithe” card, allowing for one more activation of the patron in a turn. If so, the purchase action is simulated and

the new state of the game is checked for the possibility of another activation of the patron. When this is possible, the bot buys a card and in the next move, it should play a patron activation move. If none of the conditions are met, the bot plays a random action (except `END_TURN`).

7.2.5 MaxAgent Bot

Agents seem to be relatively strong cards in this game, especially if the opponent has trouble removing them. You can think of them as additional cards in your hand that you can play, increasing the likelihood that you will be able to play some combos. Activation effects are also usually quite good – so why not focus on maximizing the agents you have in your deck?

This bot is also relatively simple. First, he randomly uses all the `PLAY_CARD` or `ACTIVATE_AGENT` actions – this is how he gets the coins (and other stats) needed to phase two. In the second phase, he checks whether he can buy any agents. The bot favors regular agents over contract ones because it will be possible to play them again. Agents are sorted by their Tier (more information about that in the next section) and the best one is selected for purchase. If the buy agent action is not available, random action is played (without `END_TURN`).

7.3 Improving Basic Bots

In this section, we briefly describe some techniques and algorithms which we used during the implementation of more advanced bots.

- Choice of patrons
  The first choice the bot has to face. Since 50% of the patrons are picked by the opponent, a good bot should be able to handle any combination of patrons. While playing Tales of Tribute, we can notice some synergies between the decks of patrons. One possible technique could be to create general rules for selecting the next patron. However, it should be remembered that generally assigned rules serve not only our bot but also the opponent. It is impossible to create a perfect bot and every bot will have decks on which it performs better than others.

  So we decided to use the apriori algorithm to select patrons. After each played game that the bot won, it saves a set of patrons to the list, assuming it is a promising selection of patrons. In the next game, the bot selects the next patrons based on the apriori algorithm, which allows us to believe that the final selection will be statistically the best for him.

- Tier card list
  A card tier list is a ranking system that helps categorizing cards based on

their power, versatility, and synergies between them. This handy tool can help players during the decision-making process, and both beginners and experienced players use it. A community of players often creates such tiers; a good example is youtuber *Pink Apple*[1], who created a couple of films where he made tier lists for some decks from Tales Of Tribute. Over time, when the game evolves, new decks are added, and some cards are nerfed/buffed, so it needs to be remembered to update the tier lists so they can still be useful.

Such tier lists can be made by using algorithms as some researchers did for Hearthstone, but we, in our project, created a static one by ourselves. Our tier list rank cards from S to D. The list is subjective, but it makes it easier to choose cards in different situations. Each bot uses the same list (if uses any), which will give a fair indication, which methods are better.

- Selection of heuristic values
  Some of the bots described later use different heuristics. Some of them were written by us, but in some cases, we used simple evolutionary algorithms to select and properly scale the values. Bots have `SetGenotype` and `GetGenotype` methods that allow them to set and extract heuristic values. We usually set the number of generations to 1000 and the size of population to 100. In each generation, we randomly matched individuals from the population into pairs, and they played a match in these pairs. By this, we obtained 50 winners in every generation. These winners became parents who gave 50 children. To get them, we combined parents in pairs to create two children by randomly giving them genes from their parents. Each child's gene could be mutated with a low probability (mutation ratio is 1%-2%). Then the parents and children were shuffled and the next generation began.

7.4 Advanced Bots

By analyzing the mechanism of the game, it can be spotted that there are three ways by which a bot can influence the opponent – by patrons, by tavern (buying cards which may suit the opponent's deck), and by agents (knocking out or destroying opponent's best ones). There is also discard action, but it is situational and does not often occur in the game to have the same impact as the three things mentioned before. The rest is optimizing our deck to beat the threshold of 40 prestige before the opponent. The way combos work makes it a good idea to play cards in any order most of the time. All these things make that the most important ascpect of game for a bot should be getting high-tier cards and maximizing the number of cards from the patrons' decks. The rest of bot's decisions should use these three ways of influencing the opponent's gameplan to make his situation more complicated.

[1] `https://www.youtube.com/@PinkAppleYT/featured`

7.4.1 Decision Tree Bot

Makes decisions about move based on the actual board state. Main rules that guide this bot:

- Playing cards – it plays all the cards, beginning with these which come from *Treasury* or *Psijic* (potential TOSS effect of moving weak cards to cooldown pile and leaving only good ones – high tier or combo effect). Usually, the bot will play all the cards and then decide to make other moves.
- Buying cards – bot prioritizes buying cards that make his deck more powerful: high-tier cards or cards from a patron's deck that he already has a lot of cards from, or combining these two things. Bot also has rules to buy cards from the Treasury deck – they are highly situational, but some of them are good in the specific moment – for example, knock out two opponent's agents, get power at the end of the game, or get additional activations of a patron. When these conditions are not fulfilled, the bot will buy cards that suit the opponent the most to thwart his plans.
- Patron activation – bot checks if the opponent is close to getting favors from all four patrons – in this case, he will try to activate a patron who favors the enemy player. Also, if the bot is close to winning by patrons, he will try to find a way to activate additional patrons (if needed). In other cases, he will try to use patrons rationally, for example: not activating *Duke of Crows* if he does not have a relatively large amount of coins, activating *Hlaalu* on expensive and not needed anymore cards during end phase of the game, etc.
- Choices – depend on the type of choice – if it is card effect, it will decide if the effect is bad or good for the bot and selects the minimum amount of bad cards or maximum amount of good cards (based on tier and other factors). When it comes to the choice of effect, the bot will try to select an option that is better for the bot now (for example, does it need more power or gold in this situation).

7.4.2 Heuristic

Every bot that used simulation was scored based on the heuristic described below. Each component has its weight which was defined by us or tuned by evolution. The main things this heuristic looks at:

- amount of power and prestige,
- patrons' level of favoritism (also a penalty if two or more patrons favor the opponent).

Until the bot's prestige is smaller than 30 – subjective threshold between the middle game (deck building phase) and end game – heuristic also takes into consideration the following things:

- tier of bot's agents on the board,
- penalty for opponent agents on board (increasing by their tier),
- tier of cards in the bot's deck,
- amount of cards from different patrons – the bigger the number is, the more likely it is to activate different combos,
- penalty for high-tier cards left in the tavern,
- penalty for leaving in a tavern card that may suit to opponent's deck.

We don't use these things after getting 30 prestige because in end game bots need to focus mainly on ending game criteria - to find win for themselves and prevent opponent wins opportunities.

7.4.3 Random Simulation Bot

The first bot that uses the ability to simulate the game that the engine provides. In a given time interval, it simulates random playouts of bot's turn and after all simulations selects the best one based on the heuristic score. The selected playout is remembered, and until something random happens, the bot uses it for the next move selection. A random event is defined as: drawing a card, a move that requires some choice, or buying a card (a new card appears in the tavern) – so every event that changes the situation on board in an unpredictable way.

The idea of this bot is similar to most people's way of playing in this game. They try to figure out the best playouts with all information available right now. In the first implementation, we encountered the same problem as in `Random` bot – even though we simulated a significant number of playouts, a large number of them were relatively short because the `END_TURN` action was selected too often, which impacted the bot's performance. So the final implementation allows to the selection of `END_TURN`, when other moves are possible, with only a slight chance (0.1%).

7.4.4 MCTS Bot

This classic AI algorithm needed a few adjustments to work in our case. Every part of that algorithm: selection, expansion, simulation, and backpropagations, has to be changed. First, the engine does not allow the simulation of the opponent's turn. Even if we could simulate that, the number of possible playouts would be too

big and simulating playout could take too much time (especially at the beginning) to make it worth. That means there is no win/lose situation after the simulation. Because of that, we decided to simulate only our turn and based on the situation on board after the playout, gave a heuristic score to this simulation. This score was normalized to make it more similar to the classic MCTS algorithm and also to make UCT metric still works.

As we simulate only our turn, we have total control over playout – the opponent cannot disturb our best possible scenario. For that reason, our bot selects the maximum between the previous and current value. This promises to select the subtree with the best situation on board at the end of the turn because it will not be covered up by not so high values from other nodes in that subtree. The selection of the best child node is also a little different – a node with the highest score is chosen, not the one with the largest number of visits. UCT score is also modified – unlike the original, we do not take the average but the maximum normalized value of the heuristic in the current node. The biggest advantage of this bot is the fact that it uses `SeededGameState`, a powerful feature provided by the engine that allows simulating moves without `Unknown` cards. By this and a big number of simulation, bot based on statistics, can decide for example whether to activate *Red Eagle* to draw a card or how high is a chance to buy a card that makes the bot's deck more powerful. Based on that argument, in UTC, we should choose mean, not max score, because there is a chance that we choose a worse path, counting on a rare event on this path (for example drawing the best card in our deck). However, during tests, it turned out that the maximum performs better – probably because drawing a card is not so often action.

7.4.5 Beam Search Bot (with Simulated Annealing)

Another classic algorithm. As the bot simulates only his turn, the tree of all playouts is not so big. By using a large k (beam width) in that algorithm, it can easily cover most of the tree. Also, because the problem is relatively small, the bot can do all computations during every move selection. The way of working is simple – from a root, bot simulates all possible moves – all paths of length one and based on board state after that simulation, assign a score to these moves (based on heuristic described above) and selects best k paths – k new nodes. In every child of that node recursively, we compute this algorithm until reaching `END_TURN` action. All that bot remembers is a list of tuples with possible moves from the root and a heuristic value to which this move can lead. To avoid getting stuck in the local maximum, the bot uses simulated annealing to select, with a small probability, paths with lower heuristic scores.

7.5 Comparisions and Conclusions

To test our bots, we wrote the `ClashBots` class, which worked on round-robin series. The first player in each match was drawn and all combinations of bots playing with each other was consider as one tournament. We saved the win ratio of each bot after every tournament to a file (to protect ourselves from possible errors which can delete all acquired data) and repeated the tournament several hundred times. By these methods, we created plot 7.1 and the table below 7.1.

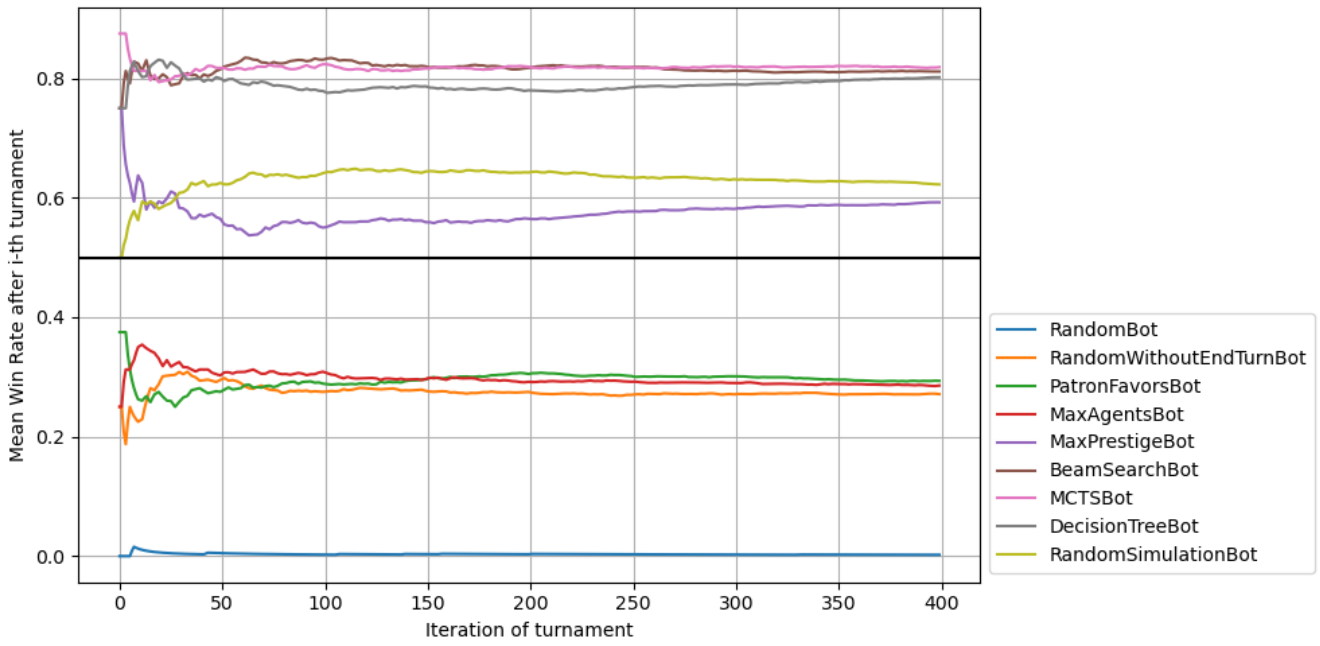

Figure 7.1: Mean Win Rate during testing

For the first 200 tournaments, bots chose randomly and then they switch to use the Apriori algorithm. As we can see, this did not significantly improve the results of most bots; the exception here is the `MaxPrestige` bot. The most favorable list of patrons for this bot turned out to be: *Duke of Crows*, *Red Eagle*, *Ansei*, and *Psijic*. While the first three are not surprising, the last one is quite unexpected (mainly `TOSS` action in his deck). Also the lack of *Pelin* in the list is interesting (because cards in his deck are mainly focused on getting power). The results shown in Figure 7.1 are no surprise. The Beam search is widely used on CodinGames for turn-based games, with a good score. MCTS, another classic algorithm, also performs good as well. The fact that the `DecisionTree` bot also scored high is satisfactory because it allows us to believe that knowledge of algorithms is not necessary for this game. This is important because the future contest can target not only programmers but also regular Tales Of Tribute players. This may also indicate that a large pool of different techniques can produce similarly good results in this game, so the variety

Table 7.1: Win rate of improved bots (after 400 round-robin games)

Win Rate with	MaxPrestigeBot	BeamSearchBot	MCTSBot	DecisionTreeBot	RandomSimBot
MaxPrestigeBot	-	0.7675	0.7325	0.8725	0.6275
BeamSearchBot	0.2325	-	0.4850	0.5650	0.1925
MCTSBot	0.2675	0.5150	-	0.4375	0.1525
DecisionTreeBot	0.1275	0.4350	0.5625	-	0.3175
RandomSimBot	0.3725	0.8075	0.8475	0.6825	0.0

of effective agents can be large. As it was predicted, `Random` bot performance is bad due to ending turns before playing all the cards. Focusing only on the thing does not seem to be beneficial; the exception here is the `MaxPrestige` bot. Even though the idea is simple, he performs quite well. The good score of this bot is less surprising if we look at the winners of the Hearthstone AI Competition in 2020 – first and second-placed bots used a similar idea of a dynamic lookahead.

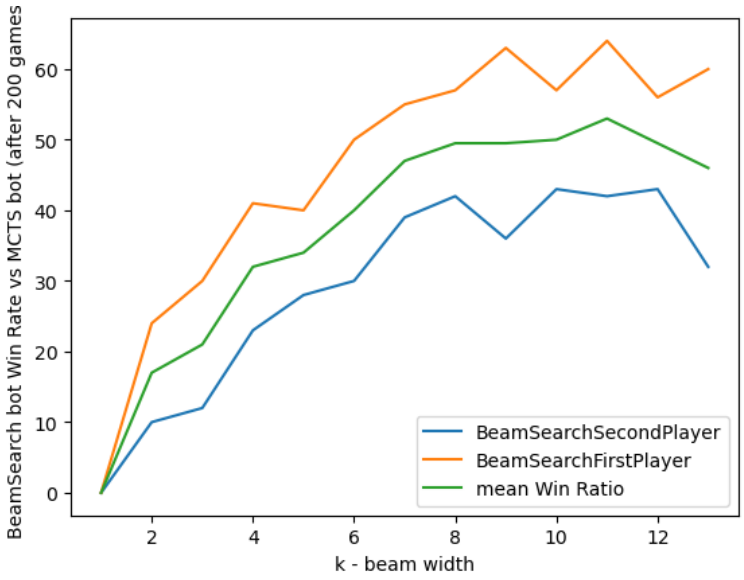

Figure 7.2: Win rate BeamSearch bot vs MCTS bot versus k in beam search algorithm

Figure 7.2 shows the win rate of `BeamSearch` bot against `MCTS` bot depending on the selected k. Special attention deserves the difference between the win rate depending on whether BeamSearch starts the game or is the second player. This is not unique to this bot only; repeating tests on other bots, win rate is usually a few percentage points higher if the bot is the first player.

Chapter 8

Conclusions

In this work, we presented the implementation of the game engine from scratch, the graphical interface in Unity, and developed several bots using the features provided by the engine. We believe that the current state of our work allows us to conduct an AI contest based on our tool in the future (e.g., at IEEE Conference on Games).

However, we hope that by then, we will be able to improve some things. The main goal for the UI is to reduce the amount of text on the cards and represent the effects of the cards using pictograms. It will also be important to add the mode which allows to bot vs bot game. It would also be useful to have two modes: with the bot's hand cards face down and up. In the future, an interesting feature in the application would be to create a server where players could share their bots and other players could play against them. Another less urgent improvement would be the moves history searchbar. When it comes to the engine, a big advantage would be the ability to play on predefined decks and logging full game state after every move.

In the case of bots implementation, we highly count on the future users of our solution. Our bots currently serve as a solid foundation for creating advanced agents as well as a practice tool for beginners who want to learn new Tales of Tribute strategies. However, we can already point out some things that can be considered in the bot projects – for example in bots that useses simulation it is often to have duplicate cards in hand – bot can effectively reduce the search tree by ignoring playing a duplicate as a unique action.

Bibliography

[1] Eric Yang and Yu-Chi Kuo. An AI for Dominion using Deep Reinforcement Learning.

[2] Jon Vegard Jansen and Robin Tollisen. An AI for dominion based on Monte-Carlo methods. Master's thesis, University of Agder, 2014.

[3] Tobias Mahlmann, Julian Togelius, and Georgios N Yannakakis. Evolving card sets towards balancing dominion. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2012.

[4] Mok Ming Foong. Creating a Dominion AI Using Genetic Algorithms. 2016.

[5] Alexander Dockhorn and Sanaz Mostaghim. Introducing the Hearthstone-AI Competition. pages 1–4, may 2019.

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

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

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

[9] Alexander Dockhorn and Sanaz Mostaghim. Introducing the Hearthstone-AI Competition. *CoRR*, abs/1906.04238, 2019.

[10] Jakub Kowalski and Radoslaw Miernik. Evolutionary Approach to Collectible Card Game Arena Deckbuilding using Active Genes. *CoRR*, abs/2001.01326, 2020.

[11] Ronaldo E Silva Vieira, Anderson Tavares, and Luiz Chaimowicz. Drafting in Collectible Card Games via Reinforcement Learning. pages 54–61, 11 2020.

[12] Ya-Ju Yang, Tsung-Su Yeh, and Tsung-Che Chiang. Deck Building in Collectible Card Games using Genetic Algorithms: A Case Study of Legends of Code and Magic. pages 01–07, 12 2021.

[13] Ronaldo Vieira, Anderson Tavares, and Luiz Chaimowicz. Drafting in Collectible Card Games via Reinforcement Learning. pages 895–898, 10 2021.

[14] Newtonsoft.Json documentation. `https://www.newtonsoft.com/json/help/html/Introduction.htm`.

[15] Xunit documentation. `https://xunit.net/#documentation`.

[16] System.CommandLine overview. `https://learn.microsoft.com/en-us/dotnet/standard/commandline/`.

[17] Solutions and projects explained. `https://learn.microsoft.com/en-us/visualstudio/ide/solutions-and-projects-in-visual-studio?view=vs-2022`.

[18] Unofficial Elder Scrolls Pages – Tales Of Tribute. `https://en.uesp.net/wiki/Online:Tales_of_Tribute`.

[19] Task class. `https://learn.microsoft.com/pl-pl/dotnet/api/system.threading.tasks.task?view=netstandard-2.1`.

[20] Fernando De Mesentier Silva, Rodrigo Canaan, Matthew Fontaine, Julian Togelius, and Amy Hoover. Evolving the Hearthstone Meta. 08 2019.

[21] Sverre Johann Bjørke and Knut Aron Fludal. Deckbuilding in Magic: The Gathering Using a Genetic Algorithm. Master's thesis, NTNU, 2017.

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

[23] Aditya Bhatt, Fernando De Mesentier Silva, Connor Watson, Julian Togelius, and Amy Hoover. Exploring the hearthstone deck space. pages 1–10, 08 2018.

[24] Andreas Stiegler, Claudius Messerschmidt, Johannes Maucher, and Keshav Dahal. Hearthstone deck-construction with a utility system. pages 21–28, 01 2016.

[25] Andre Santos. *Monte Carlo Tree Search experiments in Hearthstone*. PhD thesis, 06 2017.

[26] Alysson Silva and Fabrício Góes. HearthBot: An Autonomous Agent Based on Fuzzy ART Adaptive Neural Networks for the Digital Collectible Card Game HearthStone. *IEEE Transactions on Computational Intelligence and AI in Games*, PP:1–1, 08 2017.

[27] Pablo García-Sánchez, Alberto Tonda, Antonio Fernández-Leiva, and Carlos Cotta. Optimizing Hearthstone agents using an evolutionary algorithm. *Knowledge-Based Systems*, 188:105032, 09 2019.

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

[29] Maciej Świechowski, Tomasz Tajmajer, and Andrzej Janusz. Improving Hearthstone AI by Combining MCTS and Supervised Learning Algorithms. pages 1–8, 08 2018.