Developing Card Playing Agent for Tales of Tribute AI Competition

(Tworzenie agenta grającego w grę
Tales of Tribute)

Adam Ciężkowski Artur Krzyżyński

Praca licencjacka

Promotor: dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

30 czerwca 2023

Abstract

The main goal of this thesis is to implement the agent playing the deck-building card game Tales of Tribute. It is one of the games that will appear as a competition at the IEEE Conference on Games 2023 and in which we plan to compete. In subsequent chapters, we presented game principles, the Scripts of Tribute engine that simulates the game environment, our ideas to optimise Monte Carlo Tree Search, and an evolutionary approach to finding the proper parameter values. We tested our agent against the agents provided by the competition organisers, and in 81% of the tests, our agent outperformed the strongest of them.

Głównym celem pracy jest implementacja agenta grającego w grę karcianą Tales of Tribute. Jest to jedna z gier, która pojawi się jako zawody na IEEE Conference on Games 2023 i w których to zawodach planujemy rywalizować. W kolejnych rozdziałach przedstawiliśmy zasady gry, silnik Scripts of Tribute symulujący środowisko gry, nasze pomysły na optymalizację algorytmu Monte Carlo Tree Search oraz użycie algorytmu ewolucyjnego w celu znalezienia odpowiednich wartości parametrów. Testowaliśmy naszego agenta przeciwko agentom dostarczonym przez organizatorów konkursu, wygrywając z najsilniejszym z nich w 81%.

Contents

Chapter 1

Introduction

The annual IEEE Conference on Games (CoG) is one of the largest conferences on the scientific approach to games, and AI agent competitions are an important part of this event. They are motivation to develop new methods of competitive artificial intelligence and an opportunity to showcase the current achievements in this area. One of the games appearing at this year's CoG edition is Tales of Tribute (ToT). In this thesis, we describe our agent for this game. It begins with a description of the game, followed by a description of Scripts of Tribute (SoT) – the engine allowing the implementation of agents. In the following chapters, we focused on the challenges we encountered and how we solved them. We described the optimisations used in the implementation of the Monte Carlo Tree Search (MCTS) algorithm and the details of the evolutionary algorithm used to improve MCTS parameters. At the end of the thesis, we presented the results of experiments testing multiple variants of our algorithm and comparing our agent with the baseline bots provided by the competition's organisers.

Chapter 2

Tales of Tribute

Tales of Tribute is a two-player deck-building card game inside the popular game The Elder Scrolls Online, introduced with the High Isle expansion. In this thesis, we have not described the entire rules but only a subset implemented in the SoT engine on which the competition takes place.

2.1 Game rules

There are seven card decks. Each deck is represented by a different patron. At the beginning of the game, players choose four patrons 2.1 in order 1221 (where "1" is the first player and "2" is the second player) from six out of seven decks (the remaining Treasury patron with its deck is present in all matches). Then the main part of the game begins. Players start with ten starter cards in their decks and play in turns. At the beginning of the turn, the current player draws five cards from their *draw pile*. There is no cost to playing cards. After a card is played, it goes to the *played pile*. At the end of the turn, all cards from the *played pile* go to the *cooldown pile*. When the *draw pile* is empty, the *cooldown pile* is shuffled into the *draw pile*. Cards from chosen patrons' decks and from the Treasury deck go to the tavern, where they can be purchased by players. All the time, there are five available cards to buy. When the player decides to get the card, it goes to their *cooldown pile*, and the empty space in the tavern is filled by the next card. There are four main types of cards:

1. Action card – immediately applies certain effects and goes to the *cooldown pile*.
2. Agent card – immediately applies certain effects. After the activation, it is placed on the board and can be used every turn to apply the same effect until killed by the opponent; in this case, it goes to the *cooldown pile*. Agents can be destroyed by power (1 power = 1 HP).

3. Contract action card – works like an action card but is destroyed after use.
4. Contract agent card – works like an agent card but is destroyed when killed.

Except for playing and buying cards, players can also call on a patron (from the four chosen ones and the Treasury). Patrons have their costs and have different effects. They can have different statuses during the game. They can favour us, the opponent, or be neutral. Interaction with a patron changes its status. We can only call on one patron per turn. There are seven patrons:

Patron name	Cost	Effect
Saint Pelin	2 Power	Return up to one agent card from your cooldown pile to the top of your draw pile.
Grandmaster Delmene Hlaalu	Sacrifice a card with a cost > 0	Gain prestige equal to the card's cost minus 1.
Duke of Crows	All coins (cannot favour the current player)	Gain power equal to paid coins minus 1.
Ansei Frandar Hunding	2 Power (cannot favour the current player)	Gain 1 coin. When being favoured, gain 1 coin at the start of the turn.
Red Eagle, King of the Reach	2 Power	Draw 1 card.
Rajhin, the Purring Liar	3 Coins	Place 1 Bewilderment (a card with no effects) in your opponent's cooldown pile.
Treasury	2 Coin	Turn one of the played cards into a Writ of Coin (card with effect: Coin +2).

The Treasury is always neutral. After calling on Ansei, he always changes his status to favour the current player. The rest of the patrons change status by one step, that is, from favouring the enemy to neutral and from neutral to favouring the current player.

Figure 2.1: Patron selection

2.1.1 Cards effects

Cards have different effects. Below, we presented all the keywords:

1. Gain n coins.
2. Gain n power.
3. Draw n cards.
4. Call on one additional patron.
5. Destroy up to n cards in play.
6. Destroy up to n opponent's active agents.
7. Lower the opponent's prestige by n.
8. Return up to n cards from *cooldown pile* to the top of the *draw pile*.
9. Replace up to n cards from the tavern.
10. Acquire one card from the tavern with a cost up to n.
11. Heal agent by n health points (only agents have it).
12. The opponent discards n cards from their hand at the start of their turn.

Cards can have combo effects. A card combo is activated after playing a certain number of cards from the same deck in one turn. Combos cause effects from the same pool described above.

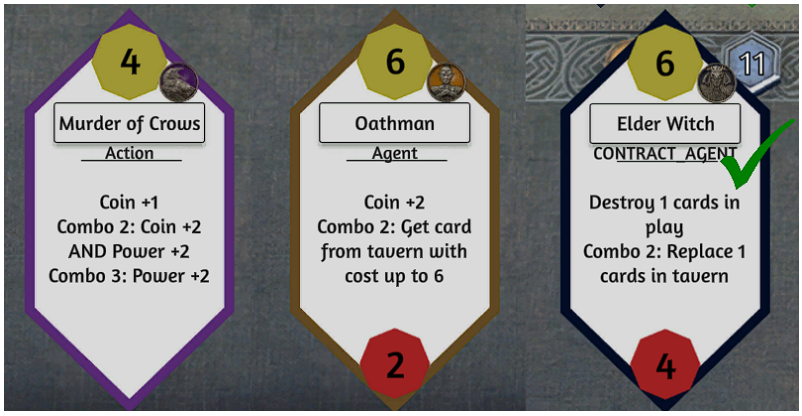

Figure 2.2: Example cards from SoT GUI

2.1.2 Win conditions

There are several win conditions:

- Gain at least 40 prestige and hold the lead (or tie) after your opponent's round.
- Be favoured by all patrons.
- Gain at least 80 prestige.

Chapter 3

Scripts of Tribute

3.1 Implementing a bot

Since the SoT engine[1] [1, 2] is written in C#, the agents should also be written in this language. The bot class should inherit from the abstract AI class and override the following three methods:

- `PatronId SelectPatron(List<PatronId> availablePatrons, int round)`

  The `SelectPatron` method is called when players select patrons before a game. We are given a list of patrons that are still available and the number of the selection turn. The method should select a patron.

- `Move Play(GameState gameState, List<Move> possibleMoves)`

  The `Play` method is called each time our agent makes a move. It receives a current `GameState` object and a list of possible moves. The method should return one of the listed moves.

- `void GameEnd(EndGameState state, FullGameState? finalBoardState)`

  The `GameEnd` method is called after a game. It allows for the analysis of the match data. Players are provided with `EndGameState` and `finalBoardState` objects that contain information about the winner and the final board state.

A `Log` method is available to save information during the game, which will be displayed by the GUI or redirected to a file by a Game Runner.

The `GameState` object provides the `ToSeededGameState` method, which takes a seed as an argument and creates a `SeededGameState` object. This object allows to simulate further gameplay using the specified seed. The `SeededGameState` has an `ApplyMove` method that takes one of the possible moves and returns the `SeededGameState` object after applying it and a new list of possible moves.

[1] https://github.com/ScriptsOfTribute

After creating the bot, it should be compiled into a library. The resulting DLL file must be placed in the right folder for the GUI and Console Runner to load the agent.

3.2 Game Runner

The SoT engine includes a console program called Game Runner 3.1. It enables to load agents from a DLL file and run games between them. It is also possible to choose the number of matches to be played, the seeds for these matches, the settings for logging into files, and the number of threads on which matches will be played in the game runner.

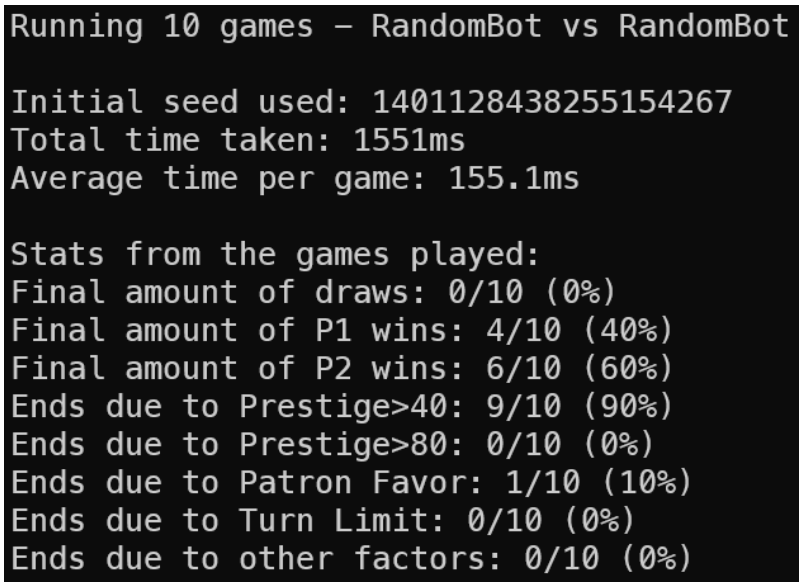

Figure 3.1: Game Runner

3.3 GUI

It is possible to play the game using a GUI written in Unity (human vs. AI agent) 3.2. It allows to learn the rules faster. Above all, it can be used for efficient debugging of the written bot and easier observation of its game play. At the beginning, the player needs to select an AI opponent, choose who starts first, set a time limit per turn for the AI agent, and optionally add a seed value. During the game, the player can see the AI agent's cards, moves, and logs.

Figure 3.2: GUI

Chapter 4

Core Idea of the Agent

4.1 Focusing on our turn

It is hard to predict what cards we will draw in the next round due to the number of cards in our deck, especially in the later stages of the game. It is even more difficult to predict the opponent's cards and moves for the next turn. Taking all this into account, it is very difficult to prepare any plan for our next turn. We decided to focus on playing the current turn optimally and maintaining a good deck.

4.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [3, 4] is a tree search-based algorithm used in situations where a complete search of the game tree is too costly. It builds only part of the tree, focusing its expansion on more promising parts. The size of the final tree depends on the time given to the algorithm. The algorithm performs the following four steps until it runs out of time:

- Selection – starting from the root, go to the leaf, selecting children along the way using the given policy (in our case, the UCB1 formula to balance exploration and exploitation).
- Expansion – add a new node to the tree.
- Simulation -– evaluate the state. We decided to finish the turn by performing random moves and evaluating the game state after our turn using a heuristic evaluation function.
- Backpropagation – update scores along the chosen in-tree path.

4.3 Board evaluation

Since we used MCTS to select moves in our turn, the evaluation function was needed to score the position at the end of our turn. There were many things to consider: prestige, patrons' favour, our deck, the opponent's deck, and the board. When evaluating a deck, we took into account the quality of the cards we had and the potential for combos. When evaluating the board, we took into account the agents on board and cards in the tavern (punishing when they are good for the opponent).

Creating a good evaluation function is a key to developing a good agent. Even if MCTS allows us to perform a complete search and calculate an accurate solution, a bad evaluation function would rate poor states better than good ones. More details related to this topic are described in the next chapter.

Chapter 5

Main Challenges

5.1 Branching factor

The branching factor is the average number of children of a node in the game tree. If we want to search the tree at a depth equal to k and our branching factor is d, we would have to search over d^k nodes. Minimising the branching factor is critical for efficient tree search.

5.1.1 Problem

This game, especially in the later rounds, has a high branching factor due to the large number of moves we can make, such as selecting which card to play, which patron to call on, or which card to buy. Cards like "Return 3 cards of any type from your *cooldown pile* to the top of your *draw pile*" or "Destroy up to 2 of your cards that are in play or in your hand from the game" result in the addition of dozens of children.

5.1.2 Solution

Instant card play. The first improvement is that there are many cards that can be played immediately at the beginning of a turn, for example "Gain 1 coin". Implementing instant moves shrinks search space massively. It removes all paths where this card is played after some complicated moves. Let us assume, for example, that at the beginning of a turn we have 3 cards that can be played immediately and 2 cards with non-trivial effects. Then, instead of considering all 5! possibilities to play them, we consider only 2 (at the beginning, we play these 3 cards, and then the remaining two in 2 ways).

Treasury. The second improvement comes with the Treasury (which replaces one of the played cards with a Writ of Coin (WoC)). The only cards that we would like to exchange are those that are worse than WoC, of course. Luckily for us, there are not many cards like that, and moreover, we do not want to consider all repetitions of the same card (for the engine, these are different actions because every card has its `UniqueId`), so when calling on Treasury, we consider only starter cards without repetitions. It helps to shrink the search space to 1–2 possibilities from 5+.

Effects, which manage our deck. The third improvement is when it comes to the effect "Return up to k cards from the *cooldown pile* to the *draw pile*". Cards with this effect have k up to 3, but the size of the *cooldown pile* can be big. Suppose we have n cards in our *cooldown pile*. There are $\binom{n}{k}$ possibilities (picking order does not matter). We decided to choose the best $k+2$ cards according to our card evaluating function (more details in the next sections) and only from those cards add all $\binom{k+2}{k}$ possibilities. When we take some real values like $n = 10$ and $k = 3$, we reduce from $\binom{10}{3} = 120$ to $\binom{5}{3} = 10$ possibilities. When it comes to the effect "Destroy up to n cards in play", we do the same, but instead of picking the best ones, we pick the worst ones.

MCTS Solver. This improvement [5] is not necessarily about branching factor by definition, but it greatly improves search efficiency by reducing redundant computations. We remember in every node if its subtree is entirely calculated, and if so, we do not take this node into account in the MCTS selection process.

5.2 Non-deterministic game state transitions

Playing a move that results in drawing cards or obtaining a card from the tavern creates a random event (e.g., what cards are drawn or what card is inserted in place of the previous one in the tavern).

5.2.1 Problem

The problem is that our MCTS works on one seed, so we are analysing only one possible outcome of a random event. It can lead to disaster because MCTS may assume that the outcome of a random event is very promising, but in reality it may not be true, or the other way around.

5.2.2 Solution

We had two approaches to this problem:

Open-loop MCTS. Open-loop MCTS [6] differs from the standard one (closed-loop) in the way it looks at nodes. In an open-loop MCTS, nodes are represented by the sequence of moves that led to them, not their current state. The upside of this approach is that in every run we simulate the game from root (and because of that, we can create a new `SeededGameState` with a new random seed). The downside is that it has less accuracy due to keeping a sequence of moves instead of states. Scoring nodes is harder and also less accurate because one node in open-loop MCTS can refer to very distinct states (with a different real score).

Parallel MCTS. Instead of one MCTS, we proposed to run a few of them, like in root parallelization [7], but in our case with different seeds. When selecting a single move, we choose the one for which the average result is the highest. This does not completely eliminate the problem, but it reduces the probability of making a huge mistake. With several MCTS trees, we need to manage the time between them (not necessarily equally; the result then should be equal to the weighted average). Individual MCTS calculations will be less accurate, but overall, this change allows for a more accurate estimation of the results of random moves. One more possible improvement to this approach is to reject around 20% of extreme values. It will eliminate super-lucky or super-unlucky events that influence our average move score. We reuse these trees unless reality does not match our predictions.

The second approach performed significantly better in our tests, and because of that, we decided to continue with it (however, we believe in the open-loop approach and will try to investigate it further before the competition). One reason why parallel MCTS trees performed better is that we could use the MCTS Solver (because single MCTS has established seed and is deterministic).

5.3 Time management

In AI agent competitions time management is very important, especially when it comes to the simulation-based algorithms.

5.3.1 Problem

The game has a limit of 10 seconds per turn. Each turn consists of several of our moves. Due to random events, we would like to recalculate the MCTS after each such move, as there is a high probability that we have not guessed the random move

correctly. The question is how to manage the given time so that we have enough until the end of the turn without wasting it.

5.3.2 Solution

We created a function that predicts how many moves will be played in our turn. It simulates a few runs of our turn and takes the maximum number of played moves. Before running the MCTS, we calculate how much time we should run it according to the given formula: $\frac{time\ left}{number\ of\ predicted\ moves+1}$. When we are low on time, we switch from a few MCTS trees to just one.

5.4 Creating evaluation function

As mentioned above, a good evaluation function is a key part of our agent's performance. We set the interval for our evaluating function for $[-10\,000, 10\,000]$ and, for MCTS purposes, scale it to $[0, 1]$. There are a few things we considered. The exact values of constants and tier lists used in the next paragraphs are listed in the appendix. Their names are written in camel case.

Patrons. We assigned different weights to different patrons depending on their status. The exact values of weights are in the appendix, but below we presented the main ideas behind them.

The Duke of Crows is a powerful game-ending patron. It converts coins into power, so we can gain a lot of prestige in one turn. The activation of this patron is only available when it does not favour us. Because of that, we assigned him negative weight when favouring us and positive weight when favouring the enemy.

Ansei gives passive income when it favours a player, so we obviously give a positive score when favouring us and a negative score when favouring the enemy.

For other patrons, we decided not to score them because their effects do not depend on their status. Activating these patrons influence on the evaluation function by applying their effects, as, for example, we reward putting good agents on the top of the *draw pile* (which is Saint Pelin's effect).

Deck. We use a tier list of cards to evaluate our deck and the opponent's deck. The tier list assigns each card a certain weight depending on the current phase of the game. Since combos provide many useful effects and resources, we also added bonuses for the number of cards from the same patron's deck according to the formula: $(number\ of\ patron\ cards)^{ComboPower}$. We also penalise when the deck consists of too many cards; as the number of cards exceeds the card limit for the current game phase, bonuses from single cards are multiplied by $\frac{CardLimit}{number\ of\ cards}$, and

bonuses from combos are calculated using the following formula: $(\frac{CardLimit}{number\ of\ cards} \times number\ of\ patron\ cards)^{ComboPower}$.

We use another tier list to evaluate cards we choose after playing a card with the effect "Return up to n cards from *cooldown pile* to the top of the *draw pile*". The reason we used that tier list is because it evaluates the pure strength of cards, not their cost or game phase, as in the first tier list.

Prestige. To evaluate the prestige difference, we use the formula: $(our\ prestige - enemy\ prestige) \times PrestigeValue$.

Board. To evaluate the board, we use the agent tier list, which, like the card tier list, gives us information about the strength of the agents depending on the phase of the game. To evaluate enemy agents, we used the formula: $Tier[agent, game\ phase] \times EnemyAgentValue \times \frac{agent's\ current\ HP+2}{agent's\ max\ HP+2}$. We wanted to take into account the agent's current HP, but remembering at the same time that an agent even with 1 HP is still a strong card, that is what these +2 stand for (for example, an agent with 1 HP and a maximum of 6 is multiplied by $\frac{1+2}{4+2} = \frac{1}{2}$ not by $\frac{1}{4}$).

As for our agent, we calculate their value the same, but then minimise it with $(current\ HP) \times PrestigeValue \times OurAgentValue$, because the cost to destroy our agent for the opponent is our agent's current HP in power (which turns into prestige at the end of the turn). $OurAgentValue$ is a constant slightly greater than 1, because it is, of course, an inconvenience for the enemy to take care of our agents.

Tavern. To evaluate cards in the tavern, we calculate which cards are good for the opponent's deck and penalise leaving them in the tavern. To calculate the strength of the card, we calculate enemy deck power with and without the given card, and its value is the difference. Then we calculate the penalty with the following formula: $Tier[card, game\ phase] \times TavernPenalty$.

We also penalise leaving in tavern contract action, which allows to knock out enemy agents when we have agents on the table. Suppose on the table there is a card to knock out two enemy agents. Then we calculate scores for our two best agents and only count 25% of their original value, as the opponent can easily kill them.

5.4.1 Game phases

We observed that the optimal weights change noticeably depending on the current phase of the game. This also applies to the tier lists, as some cards are good in the early game but useless later on. Others, on the other hand, are not worth buying too early.

We decided to choose the phase of the game at the start of each turn based on our and the enemy's prestige:

- If our prestige is bigger than 26 or the enemy's prestige is bigger than 29, it is late game.
- If our prestige is less than 11 and the enemy's prestige is less than 14, it is early game.
- In other cases, it is mid game.

Early game. In the early game, we almost do not care about prestige. On the other hand, cards bought in this phase will be with us all along, so their value is high as they will be drawn several times. Mainly, cards with big resource income effects are good to buy in order to afford more expensive cards in upcoming turns.

Mid game. In the middle phase of the game, when our deck is already quite developed, we can switch a bit from building the economy to buying cards with more interesting effects to try to get the upper hand over the opponent in terms of economy and prestige (but it is still not the biggest priority).

Late game. In the late game, our main priority is to gain prestige to win the game. Our deck is developed, and we usually do not want to buy new cards because, due to the short time until the end of the game, they probably will not be used anyway.

5.4.2 Adjusting weights using evolutionary algorithm

Coming up with well-adjusted weights on your own is hard, so we created some starting weights using our knowledge of the game and ran an evolutionary algorithm to obtain better ones. An evolutionary algorithm consists of an initial population choice and a main loop with three steps:

- Evaluate fitness – judge how good individuals are in the population by comparing them against each other.
- Reproduction – select a set of individuals to be the parents of the next generation (often the fittest ones).
- Crossover and mutation – parents create children by combining their properties. Children can mutate by changing weights from crossover a little bit.

We performed these steps as follows:

First generation. The first generation was created by mutating the initial weights we assigned. Each parameter was multiplied by a random value from the interval [0.5, 1.5] with a probability 70%.

Evaluate fitness. The number of games each individual won against the best one from the previous generation determined its rating. The second generation fought the bot with initial parameter values.

Reproduction. The parents of the next generation consisted mainly of the fittest individuals. To avoid creating generations with similar parameter values, we also added some weaker individuals (around 15% of the parents).

Crossover and mutation. 70% of the new generation was created by randomly choosing two parents and combining them using the standard one-point crossover. Then, we multiplied each of its parameters by a value from the range [0.95, 1.05] with 40% probability. The two best individuals from the current generation are also added to the new generation, and the rest of the next generation was created by mutating randomly chosen parents (each parameter was multiplied by a value from the range [0.8, 1.2] with 30% probability).

Chapter 6

Experiments

6.1 Multiple MCTS trees

We run fights between agents using a different number of MCTS trees; their results can be seen in Table 6.1. As the table shows, increasing the number of MCTS trees helps to increase the average number of wins. However, more parallel trees result in a shallower search, as the turn's time constraint does not allow for adequate expansion.

versus	1 MCTS	3 MCTS	5 MCTS	10 MCTS
1 MCTS	-	$51 \pm 4.11\%$	$55.5 \pm 4.09\%$	$57.75 \pm 4.06\%$
3 MCTS	$49 \pm 4.11\%$	-	$54 \pm 4.10\%$	$54 \pm 4.10\%$
5 MCTS	$44.5 \pm 4.09\%$	$46 \pm 4.10\%$	-	$47 \pm 4.1\%$
10 MCTS	$42.25 \pm 4.06\%$	$46 \pm 4.10\%$	$53 \pm 4.1\%$	-
Average	$45.25 \pm 2.36\%$	$47.67 \pm 2.37\%$	$54.17 \pm 2.37\%$	$52.92 \pm 2.37\%$

Table 6.1: Results against agents using different numbers of MCTS after 400 games, with their 90% confidence intervals

6.2 Evolving parameter weights

The Figure 6.1 shows the results of our agent using the parameters calculated by the evolutionary algorithm. It was tested every 20 generations against the best baseline bot – `MCTSBot`. The evolutionary algorithm has done its job, increasing the win ratio from below 70% to above 80%.

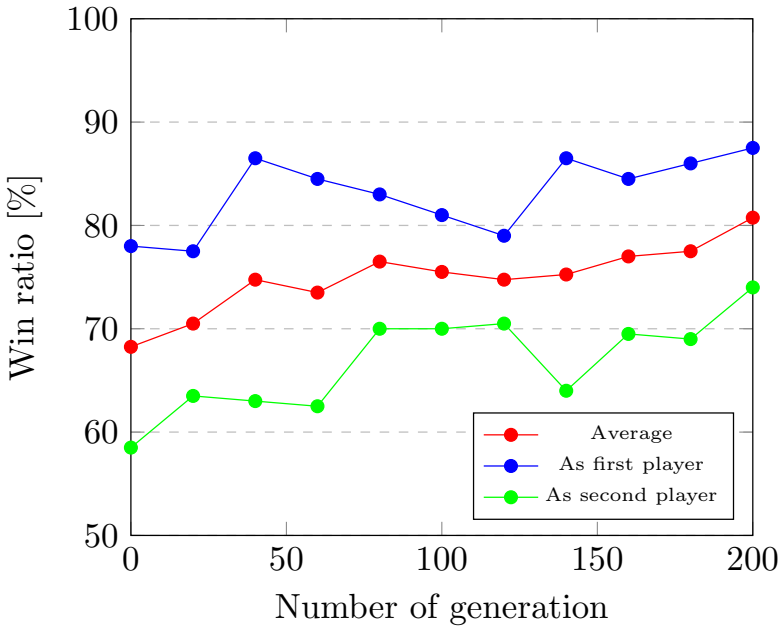

Figure 6.1: Win ratio depending on generation number after 200 games on each side

6.3 Results against agents from the SoT engine

Finally, we tested our final bot against the baseline bots provided by the SoT authors. As shown in Table 6.2, our agent has no problem defeating the strongest agents included in the SoT engine. Despite the fact that agents `DecisionTreeBot`, `MCTSBot`, `BeamSearchBot` in battle against each other performed similarly [1], `MCTSBot` occurred to be the toughest rival for our agent.

Our agent	Decision Tree	MCTS	Beam Search
First player	$96 \pm 1.61\%$	$87.5 \pm 2.72\%$	$95.5 \pm 1.70\%$
Second player	$90 \pm 2.47\%$	$74 \pm 3.61\%$	$81.5 \pm 3.19\%$
Average	$93 \pm 1.48\%$	$80.75 \pm 2.29\%$	$88.5 \pm 1.86\%$

Table 6.2: Results against the best agents provided in SoT after 400 games with their 90% confidence intervals

Chapter 7

Conclusions

The main goal of our thesis was to create an AI agent playing Tales of Tribute. In the thesis, we described how the AI agent was created and what techniques we used. We also described how Monte Carlo Tree Search can be used to search through the game tree and how we improved its performance. Furthermore, in order to develop a decent evaluation function, we used an evolutionary method to improve its parameters. The thesis also discusses the key difficulties we encountered and how we overcame them. Finally, we created the AI agent that is ready to compete in the first Tales of Tribute AI Competition at the IEEE Conference on Games 2023, and we are looking forward to good results.

Bibliography

[1] Dominik Budzki, Damian Kowalik, and Katarzyna Polak. Implementing Tales of Tribute as a Programming Game. Engineer's thesis, University of Wrocław, 2023.

[2] Jakub Kowalski, Radosław Miernik, Katarzyna Polak, Dominik Budzki, and Damian Kowalik. Introducing Tales of Tribute AI Competition. *arXiv preprint arXiv:2305.08234*, 2023.

[3] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[4] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte Carlo Tree Search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2022.

[5] Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-Carlo Tree Search Solver. In *Computers and Games*, pages 25–36, 2008.

[6] Erwan Lecarpentier, Guillaume Infantes, Charles Lesire, and Emmanuel Rachelson. Open Loop Execution of Tree-Search Algorithms, extended version. *27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, 2019.

[7] Guillaume M. J. B. Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel Monte-Carlo Tree Search. In *Computers and Games*, pages 60–71, 2008.

[8] Jakub Kowalski and Radosław Miernik. Evolutionary Approach to Collectible Card Game Arena Deckbuilding using Active Genes. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2020.

[9] Rules of Tales of Tribute. https://www.elderscrollsonline.com/en-us/news/post/62081. [access: 18.06.2023].

[10] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996.

[11] Maciej Świechowski and Jacek Mańdziuk. A Hybrid Approach to Parallelization of Monte Carlo Tree Search in General Game Playing. In *Challenging Problems and Solutions in Intelligent Systems*, pages 199–215. 2016.

Appendix A

Constants

In Table A.1, we listed the constant values used by our agent.

Constant name	Early game	Mid game	Late game
Crow	357.165	401.261	439.462
Ansei	69.727	65.175	51.416
PrestigeValue	14.407	115.438	244.106
CardLimit	3.584	9.803	6.175
ComboPower	1.453	2.018	1.765
OurAgentValue	2.407	1.205	0.628
EnemyAgentValue	-36.382	-31.121	-165.216
UpcomingCard	14.453	10.997	96.387
TierMultiplier	32.456	15.362	17.865
TavernPenatly	-1.019	-1.624	-1.843
KnowingCardCombo	0.986	1.268	1.880
After40Bonus	469.629	562.802	436.450

Table A.1: The values of constants

The constant *Crow* is multiplied by -1 if the patron favours us (because we cannot call on it until the opponent changes its status), by 0 if it is neutral, and by 1 if the patron favours the opponent (because they cannot call on it until we change its status). The constant *Ansei* is multiplied by -1 if the patron favours the enemy (because the opponent has passive income), by 0 if it is neutral, and by 1 if the patron favours us (because we have passive income).

The description of constants unmentioned before:

- UpcomingCard – bonus for cards on the top of the *draw pile* (if they were placed there using a card with the effect "Return up to n cards from the

cooldown pile to the top of the *draw pile*" or by calling on Saint Pelin). Cards are evaluated using tiers listed in Table B.1 in the column "From cooldown" and multiplied by UpcomingCard.

- TierMultiplier – multiplier of card tiers from the card tier list (in the card tier list, cards have different tiers B.1 and tiers have different values B.2). Its goal was to improve the ratio of card values to other factors.
- KnowingCardCombo – combo potential bonus for cards on top of the *draw pile*. For each of these cards, we add the number of cards we own from the same deck and multiply by this constant.
- After40Bonus – additional bonus for each prestige point over 40.

Appendix B

Tier lists

B.1 Card tier list

Card name	Early game	Mid game	Late game	From cooldown
Currency Exchange	S	S	A	HB
Luxury Exports	S	S	C	HD
Oathman	A	A	B	HD
Hlaalu Councilor	A	A	C	HS
Hlaalu Kinsman	A	A	C	HS
House Embassy	A	A	C	HS
House Marketplace	B	A	C	HA
Hireling	C	C	D	HD
Hostile Takeover	B	C	D	HD
Customs Seizure	D	D	D	HD
Goods Shipment	D	D	D	HF
Midnight Raid	S	S	S	HB
Hagraven	D	B	D	HB
Hagraven Matron	D	A	C	HA
War Song	D	D	D	HF
Blackfeather Knave	S	S	A	HB
Plunder	S	S	S	HS
Toll of Flesh	S	S	A	HC
Toll of Silver	S	S	A	HC
Murder of Crows	S	S	A	HA
Pilfer	A	S	A	HB
Squawking Oratory	A	S	A	HA
Pool of Shadow	B	A	B	HC
Scratch	A	A	B	HD

Blackfeather Brigand	C	C	C	HD
Blackfeather Knight	A	B	C	HB
Peck	C	C	C	HF
Conquest	A	A	B	HC
Hira's End	S	S	S	HS
Hel Shira Herald	B	A	B	HS
March on Hattu	A	A	A	HS
Shehai Summoning	B	B	B	HS
Warrior Wave	S	A	B	HB
Ansei Assault	B	A	B	HA
Ansei's Victory	B	A	B	HS
No Shira Poet	C	C	C	HC
Way of the Sword	D	D	D	HF
Rally	A	S	A	HA
Siege Weapon Volley	A	S	B	HC
The Armory	A	S	A	HB
Banneret	A	S	A	HS
Knight Commander	S	S	A	HS
Reinforcements	S	A	B	HD
Archers' Volley	B	A	B	HD
Legion's Arrival	A	A	B	HD
Bangkorai Sentries	C	B	C	HA
Knights of Saint Pelin	C	A	C	HA
The Portcullis	C	C	D	HD
Fortify	D	D	D	HF
Bewilderment	F	F	F	HF
Grand Larceny	A	A	B	HC
Jarring Lullaby	B	A	B	HC
Jeering Shadow	C	C	C	HE
Pounce and Profit	S	S	B	HC
Prowling Shadow	B	C	C	HB
Shadow's Slumber	A	A	B	HB
Slight of Hand	A	B	D	HF
Stubborn Shadow	D	C	D	HD
Twilight Revelry	B	A	B	HC
Swipe	D	D	D	HF
Gold	E	F	F	HF
Writ of Coin	WOC	B	E	HF

Table B.1: Card tier list

Card tier	Value
S	50
A	30
B	15
WOC	10
C	3
D	1
E	0
F	-3
HS	5
HA	4
HB	3
HC	2
HD	1
HE	0
HF	-1

Table B.2: Values of card tiers

B.2 Agent tier list

Agent name	Value
Oathman	C
Hlaalu Councilor	S
Hlaalu Kinsman	S
Hireling	C
Clan-Witch	A
Elder Witch	A
Hagraven	A
Hagraven Matron	A
Karth Man-Hunter	B
Blackfeather Knave	B
Blackfeather Brigand	B
Blackfeather Knight	B
Hel Shira Herald	B
No Shira Poet	C

Banneret	S
Knight Commander	S
Shield Bearer	B
Bangkorai Sentries	B
Knights of Saint Pelin	A
Jeering Shadow	C
Prowling Shadow	B
Stubborn Shadow	C

Table B.3: Agent tier list

Agent tier	Value
S	5
A	4
B	3
C	2

Table B.4: Values of agent tiers