Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Monte Carlo tree search in card and other games

Bachelor Thesis

2023

Jakub Kaššák

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Monte Carlo tree search in card and other games

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	doc. RNDr. Ján Mazák, PhD.

Bratislava, 2023
Jakub Kaššák

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Jakub Kaššák
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Monte Carlo tree search in card and other games
Stromové prehľadávanie Monte Carlo v kartových a iných hrách

Anotácia:

Cieľ:

Vedúci: doc. RNDr. Ján Mazák, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 18.10.2022

Dátum schválenia: 18.10.2022 doc. RNDr. Dana Pardubská, CSc.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Jakub Kaššák
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Monte Carlo tree search in card and other games

Annotation: Implement variants of Monte Carlo tree search for several chosen games, investigate their performance and compare it to other game-playing algorithms.

Aim:
1. Study and create an overview of current research on Monte Carlo tree search and its applicability to playing various games.
2. Implement variants of Monte Carlo tree search for a couple of chosen games and investigate if they are viable for automated game play. Compare them with other algorithms if any are available.
3. Investigate how heuristics affect the performance of Monte Carlo tree search for specific games.

Supervisor:	doc. RNDr. Ján Mazák, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.

Assigned: 18.10.2022

Approved: 18.10.2022 doc. RNDr. Dana Pardubská, CSc.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgements: I would like to thank a number of people who have had a substantial influence on my work and I would like to mention them in this acknowledgements section.

First of all, I would like to express my gratitude to Associate Professor Mazák for his valuable comments, professional help, guidance, and encouragement throughout my bachelor thesis. His wisdom, experience, and support were very helpful in helping me achieve my goals and improve my bachelor's thesis.

I would also like to thank Jozef Melicher for his feedback and support when I did not feel like working.

Last but not least, I would also like to thank my family and friends who have provided me with support and encouragement throughout the course of this thesis.

To all these persons I would like to express my sincere gratitude for their contribution and support which enabled me to successfully complete this bachelor thesis.

Abstrakt

Hlavným cieľom tejto bakalárskej práce bolo preskúmať a analyzovať aplikáciu algoritmu Monte Carlo Tree Search (MCTS) na novú kartovú hru. Najprv sme uviedli komplexné definície kartových hier vo všeobecnosti a potom sme sa zamerali konkrétne na hru Faraón. Táto hra má zvyčajne malý počet dostupných ťahov, ale je ťažké heuristicky vyhodnotiť aktuálny stav (na rozdiel napr. od šachu). Hru Faraón sme potom implementovali spolu s algoritmom MCTS vrátane dvoch zaujímavých variantov odporúčaných existujúcim výskumom. Starostlivými manuálnymi úpravami sme doladili parametre algoritmu MCTS.

Na vyhodnotenie vplyvu našich úprav sme uskutočnili rozsiahly turnaj s použitím rôznych variantov MCTS a alternatívnych algoritmov. Výsledky sme zaznamenali a analyzovali s cieľom posúdiť výkonnosť týchto algoritmov. Zaznamenali sme výrazné zlepšenie výkonu pri rovnakom výpočtovom rozpočte, ale zároveň zavedenie Beam Search ako jedného z vylepšení neočakávane viedlo k poklesu výkonu.

Okrem toho bolo kľúčovým aspektom našej práce skúmanie evolučných zákonitostí stromovej štruktúry vytvorenej algoritmom MCTS. Zistilo sa, že strom vykazuje pozoruhodnú vyváženosť, čo naznačuje, že v hre Faraón neexistujú žiadne super výhodné ťahy. Zavedenie heuristiky orezávania ťahov, druhého z dvoch zlepšujúcich variantov, však spôsobilo jav, ktorý sme nazvali neočakávaná prehra, hoci celkový výkon sa zlepšil.

Naše výsledky naznačujú, že algoritmus MCTS má pozoruhodný potenciál na riešenie úloh hier, ako je napríklad hra Faraón, pričom neustále prekonáva ostatné algoritmy. Tento výskum poskytuje cenné poznatky o použití MCTS v kontexte hry Faraón a ponúka zaujímavé vyhliadky na budúci pokrok v tejto oblasti.

Kľúčové slová: MCTS, kartové hry, Faraón, analýza, algoritmus

Abstract

The primary objective of this bachelor thesis was to explore and analyse the application of the Monte Carlo Tree Search (MCTS) algorithm to a novel card game. First, we provided comprehensive definitions of card games in general and focused specifically on the game Pharaoh. The game typically has a small number of available moves, but it is difficult to heuristically evaluate the current state (unlike e.g. chess). We then implemented the Pharaoh game together with the MCTS algorithm, including two interesting variants recommended by existing research. Through careful manual adjustments, we fine-tuned the parameters of the MCTS algorithm.

To evaluate the impact of our modifications, we conducted an extensive tournament using different MCTS variants and alternative algorithms. The results were recorded and analysed to assess the performance of these algorithms. We observed significant performance improvements for the same computational budget, but at the same time the introduction of Beam Search as one of the improvements unexpectedly led to a decrease in performance.

Furthermore, a crucial aspect of our work was to investigate the evolutionary patterns of the tree structure constructed by the MCTS algorithm. It was found that the tree exhibited remarkable balance, suggesting that there were no super-advantageous moves in the Pharaoh game. However, the introduction of a move-pruning heuristic, the second of the two improvement variants, caused a phenomenon we named unexpected loss, although overall performance was improved.

Our results indicate that the MCTS algorithm has remarkable potential for game solving tasks such as Pharaoh, consistently outperforming other algorithms. This research provides valuable insights into the use of MCTS in the context of Pharaoh, and offers exciting prospects for future advances in the field.

Keywords: MCTS, card games, Pharaoh, analysis, algorithm

Contents

List of Figures

List of Tables

Introduction

The Monte Carlo Tree Search (MCTS) was introduced by Kocsis and Szepesvári as suitable for playing the game of Go in 2006. This game was until then very problematic mainly because of the high degree of branching in each move. The algorithm quickly proved to be effective when Gelly et al. [8] constructed a computer Go player who was at an advanced level (5 dan). Later, the MCTS algorithm was also behind the success of DeepMind's AlphaGo [15], which was the first to defeat a professional human player, and a follower of AlphaGo eventually defeated the best human player in Go.

This bachelor's thesis focuses on the application of the Monte Carlo Tree Search (MCTS) algorithm in the context of the card game Pharaoh.

Card games have long been a source of entertainment, challenge, and strategic thinking. The card game Pharaoh, with its unique mechanics and gameplay elements, provides an interesting domain for studying the application of artificial intelligence algorithms. By applying the MCTS algorithm to Pharaoh, we aim to improve our understanding of how this algorithm can be used.

The structure of this work is as follows. First, we aim to define a card game in general, which will help us to explain the MCTS algorithm and the game of Pharaoh.

Second, we will explore the fundamentals of the MCTS algorithm, explaining its underlying principles such as selection, expansion, simulation and backpropagation. This exploration will provide a solid foundation for understanding the mechanics and functionality of the algorithm and its potential applications in Pharaoh.

Third, we provide a comprehensive analysis of the game Pharaoh, including its rules, mechanics and winning conditions. By gaining a clear understanding of the game's characteristics, we can effectively adapt and apply the MCTS algorithm to this specific domain.

Fourth, we implement the game Pharaoh and MCTS algorithm.

Fifth, we will conduct experiments and evaluate the performance of the MCTS algorithm on the game of Pharaoh. By implementing and fine-tuning the parameters of the algorithm, we aim to optimise its performance, and then we will conduct a tournament using different variants of the MCTS algorithm as well as other algorithms in the game of Pharaoh to compare the results and assess the performance of these algorithms in the game environment.

Finally, we will analyse the evolution of the tree constructed by the MCTS algorithm.

The results of this research will contribute to the field of artificial intelligence in games, especially in the context of card games. The knowledge gained from applying the MCTS algorithm to Pharaoh can potentially inform the development of intelligent agents capable of competing with human players. Furthermore, the results of this research may have implications beyond the game of Pharaoh. The principles and techniques explored in this thesis can be extended and applied to other card games, thereby increasing our understanding of the capabilities and limitations of the MCTS algorithm in the broader domain of gaming.

Chapter 1

Definition of card game

In this chapter we explain what we mean by a card game. We hope that these definitions will help us to better explain the MCTS algorithm later in this paper.

Definition 1. Let there be a set of $SUITS$ and a set of $VALUES$. Deck D is a Cartesian product $D = SUITS \times VALUES$. Card C is a pair ($suit$, $value$) $\in D$. [1]

Example 2. Let the deck be a Cartesian product $SUITS \times VALUES$, where $SUITS = \{leaf, heart, bell, acorns\}$ and $VALUES = \{VII, VIII, IX, X, under, over, king, ace\}$. What we have just defined is a standard deck of german-suited playing cards.

Definition 3. Discard pile DP is a list of cards that are face-up. The top card in the discard pile is called TOP. Stock ST is a list of cards where the cards are face-down.

Definition 4. Game state is a tuple $GS = (DP, ST, LH, LP\_MC, n, i, mc)$, where DP is a discard pile, ST is a stock, LH is a list of hands, n, i and mc are integers, LP_MC is a n-tuple of integers. The integer n denotes the number of players. We call the integer i an index, because it marks whose player's turn it is (turn is described in definition 9). We call the integer mc a counter, because it counts the number of turns. Sign $deck(GS) = DP \cup ST \bigcup_{H \in LH} H$ marks the deck of the game GS.

Definition 5. Rule R is an implication of the form "if antecedent, then consequent". The instance of a rule is called a move. Antecedents are conditions on a game state. A move derived from a rule can only be applied to a game state if the conditions are met. Consequents define the changes to the game state that occur when the move is applied.

Example 6. Let R be a rule: If there is a card in the hand (of the player whose turn it is) of the same value as the TOP, then that card may be played into the discard pile.

[1] In the real world, a card has two faces - the front and the back. The value and suit are marked on the front. On the contrary, cards are indistinguishable by their backs.

Definition 7. Game is a pair $G = (GS, RS)$ of an initial game state GS and a set of rules RS. We require that if n is the number of players, then $0 \leq G.i < n$ and $G.LP\_MC$ has exactly n entries, where $\forall j \in \mathbb{N}, 0 \leq j < n : G.LP\_MC[j] \leq mc$.

Remark. To distinguish between a game as described in the above definition and the activity of playing a game, we will call the latter a match.

Definition 8. Player is an entity that can change the game state by making moves. Game play is pair $GP = (G, LP)$, where G is a game and LP is a list of players. A player is uniquely identified by its position in the list of players.

Definition 9. Turn is player's opportunity to change the game state by using a move. Players take turns in a fixed round-robin order.

Definition 10. The game state is terminal, if there is no move left.

Definition 11. Game state tree of a game $G = (GS, RS)$ is an oriented rooted tree, where each node stores a game state and the root stores the initial state GS. Each edge in the tree corresponds to a move derived from rule $R \in RS$ (i.e. for an edge from node $N1$ to node $N2$, there exists a move M, so that when the move M is applied to the state stored in $N1$ we get the state stored in $N2$), and for each move M from the state $S1$ stored in a node $N1$, there exists an edge to a node $N2$ storing the state reached from $S1$ by applying the move M.[2] Note that each leaf stores a terminal state. We will call a branch of the game state tree a play, and we will call a path from a vertex to one of its descending leaves a playout.

[2]Note that it is also possible for a single state to be stored in multiple nodes if the state can be reached by different sequences of moves.

Chapter 2

Monte Carlo tree search

In this chapter we explain how the Monte Carlo Tree Search (MCTS) algorithm works. In the first subsection we describe the phases of the baseline MCTS, and in the second subsection we take a closer look at the selection policy, which we also refer to as the tree policy.

2.1 Four phases

MCTS is an algorithm that searches the state space represented as a tree - in our case, the game state tree. The goal of the algorithm is to find the next promising move. MCTS builds a subgraph of the game state tree and also stores statistical data about possible decisions in the explored nodes.

Definition 12. To distinguish between the game state tree and the tree thath the algorithm builds (which is a subgraph of the game state tree but also stores additional data), we will call the latter a Monte Carlo tree or MC tree.

MCTS builds a MC tree by iterating these four phases:

1. Selection
2. Expansion
3. Simulation
4. Backpropagation

The first phase is called **selection**. The algorithm starts at the root and proceeds down the already constructed MC tree, selecting the next node using the tree policy, which will be described in the next section 2.2. The first phase ends when the algorithm reaches a leaf.

The second phase is **expansion.** Let $l_1$be a leaf. If leaf l_1 does not represent a terminal state, the algorithm constructs all states that are reachable from the state in leaf l_1 and hangs them as children under leaf l_1. Otherwise, the algorithm proceeds directly to the next phase.

Then the **simulation** phase follows. The algorithm chooses a child c of the leaf l_1 (or if $l_1$is terminal, then the l_1 itself) and simulates a playout from it up to the terminal state. If the playout is random, we call it a light playout. This is the Monte Carlo part of the algorithm.

It is worth noting that that some variants use heavy playouts instead of light playout. Heavy playout involves enhancing the randomness of the playouts by introducing more sophisticated and complex strategies. Instead of relying solely on purely random moves, heavy playout incorporates additional heuristics, domain-specific knowledge, or advanced game-playing techniques to guide the simulated play.

In the final stage - **backpropagation -** the algorithm propagates the result of the playout to all nodes on the path to node c. The algorithm records the number of simulated playouts and the number of wins. In the backpropagation phase at each node on the branch with c, the algorithm increments the total number of simulated playouts and, if the move to that node was chosen by the player who won in the current playout, it increments the number of wins by one.

The algorithm iterates through these four phases until it spends all of its computational budget. The algorithm then returns the most promising move.

The computational budget can be based on several metrics:

1. Time
2. Number of iterations
3. Number of explored moves/states
4. Some combination of previous

The time-based approach allocates a specific amount of time for MCTS. This method is useful for real-time decision making. The iteration-based approach specifies a number of iterations of the four phases. It is useful for predictable and comparable performance across different platforms. This is why we used this approach. The explored moves/states approach specifies the number of moves/states explored rather than the number of iterations. Some approaches combine a time-based budget with a maximum iteration limit. For example, the algorithm may run for a fixed time but terminate early if the maximum iteration limit is reached.

2.2 Tree policy

Since the MCTS algorithm does not search the entire game state tree, we must ensure that it searches at least some important part of it. To do this, we use a tree policy that selects the next child in the selection. Thus, the goal of the tree policy is to balance the discovery of new moves and the exploitation of already discovered advantageous moves. In our work, we use the UCT (Upper Confidence bound applied to Trees) function proposed by Kocsis and Szepesvári [11]. We give its prescription in the following equation:

$$i^* = \arg\max_{i \in children(v)} \left\{ \frac{w_i}{n_i} + \sqrt{c \cdot \frac{\log(N)}{n_i}} \right\}$$

where v is the current vertex reached in the selection phase, i is one of the children of the vertex v, w_i is the total number of wins in child i, n_i is the total number of visits to child i, N is the number of visits to vertex v, and c is the exploration constant. We can see that the first component $\left(\frac{w_i}{n_i}\right)$ is large if there are many wins in child i, and in turn the second component $\left(\sqrt{c \cdot \frac{\log(N)}{n_i}}\right)$ is large if child i is under-explored relative to its siblings. Thus, the first component ensures exploitation of discovered advantageous moves and the second ensures discovery of new ones. The exploration constant c is used to tune the balance between exploration and exploitation.

The UCT function is clearly the most widely used tree policy, but there are several alternatives. One of them is called UCB1-TUNED and has been studied by Gelly and Wang [8]. This method bears a strong resemblance to UCT, but additionally adds an variance of the score. Gelly and Wang empirically showed that this function gives better results than UCT in their game of Go. Examples of other tree policies are Exploration-Exploitation with Exponential weights (Auer et al. [3]) or Thomson sampling (Bai et al. [4]).

2.3 MCTS variants

In this chapter we provide short summery of other works on MCTS especially those which focus on some kind of MCTS variant. It is often the case that the base version of MCTS underperforms on certain problems and so there are modifications and enhancements that attempt to improve efficiency on various problems. In our work, we are currently concerned with complete information games, i.e., games where the player knows the entire state of the game (e.g., chess), so we summarize at least a few interesting approaches to improve the efficiency of the algorithm suitable for a complete information game.

Connecting of transpositions By transposition, we call the same state that can be achieved by playing different sequences of moves. Thus, let S_1, S_2 be the states, let $\boldsymbol{a} = (a_1, \ldots, a_n)$ and $\boldsymbol{b} = (b_1, \ldots, b_m)$ be two different sequences of moves that can be played to the state S_1. If both $\boldsymbol{a}$ and $\boldsymbol{b}$ transform the state S_1 to S_2, then S_2 is a transposition. The standard MCTS searches a regular tree, which results in a different branches for each transposition and therefore each transposition has to be searched separately, which is inefficient. However, it significantly eases the implementation because if we link the transpositions we get an oriented acyclic graph instead of a tree, which causes new non-trivial problems (e.g. how to prune this tree/graph after playing a move). However, solving them successfully leads to a more efficient use of the computational budget. This idea has been used, e.g., by Kishimoto and Schaeffer [10] (but in their case it was not MCTS, but MiniMax).

Heuristic move-pruning The state tree tends to be very large, which causes the MC tree to grow in width instead of depth. One possible solution is to prune some moves using heuristics based on domain knowledge. Move-pruning heuristics in Monte Carlo Tree Search (MCTS) aim to improve the efficiency of the search process by selectively expanding or considering only a subset of the available moves at each simulation step. The goal is to reduce the number of exploratory simulations required while maintaining the effectiveness of the search. This approach was used by Sephton et al. [13] for the game Lords of War.

Beam Search Another interesting variant is called Beam Search. The basic idea behind Beam Search is to maintain a fixed-sized set of partial solutions called the beam throughout the search process. Beam Search with MCTS (BMCTS) uses two additional parameters L and W. W specifies the beam width, i.e the width of the MC tree at depth k, and L the number of simulations after which low-success moves are pruned. Thus, the algorithm performs $k \cdot L$ simulations and then prunes the number of vertices at depth k to at most W (keeping the most promising moves) for each integer k greater then one. This can lead to faster convergence towards high-quality solutions compared to traditional search. However, it's important to note that beam search is not guaranteed to find the optimal solution. With this technique, the spatial (and hence temporal) complexity is reduced to linear. BMCTS was used by Baier and Winands [5] and achieved a 2-4 fold improvement in efficiency with the same computational budget.

Chapter 3

Card game of Pharaoh

In our work we focus on the card game of Pharaoh. So in this chapter we describe the rules of our version of Pharaoh and why we chose those rules. We also explain what problems arise when playing game of Pharaoh.

3.1 Rules and initial state

Definition 13. Now we define the initial state of the game of Pharaoh for $2 \leq n \leq 6$ players. The game starts in the initial state $GS = (DP, ST, LH, LP\_MC, n, i, mc)$, where D is a subset of the german-suited playing cards, $LH = [H_1, \ldots, H_n]$ is list of hands, with each hand symbolising one player, $LP\_MC = [-1, \ldots, -1]$ of length n and $i = mc = 0$. Next, the deck of cards $deck(GS)$ is randomly shuffled, then each player is dealt a specified number of cards into his hand (e.g., if playing with all 32 cards, 5 cards are usually dealt), one card is placed in the discard pile DP as the TOP and the remaining cards are placed in the stock ST.

Definition 14. Now let us define the rule set RS of the game of Pharaoh. Let $\bar{C}$ be a k-tuple of cards of the same value, that the player, whose turn it is, has in his hand. If the TOP has same value as the cards in $\bar{C}$, then the player may play this k-tuple $\bar{C}$ into the discard pile. The last card he puts into the discard pile will be the TOP next turn, so it depends on the order in which he discards the cards. We call this rule **match value**. The second rule is similar, but the first card from the k-tuple $\bar{C}$ must match the suit of the TOP, so we call this rule **match suit**. There are two possibilities regarding the last rule: if there is at least one card in the stock, a player can draw that card into his hand, and we call it **draw a card**. Or there may be no card in the stock, in which case the player whose turn it is may do nothing. We call it **skip a turn**.

After one of these rules has been played, it is the next player's turn. If the player has no cards left after his turn, he is finished in this match and the current value of the move counter mc is stored in his part of LP_MC. At the end of the match, the

first player to finish (player with the lowest number in LP_MC) is the winner.

Definition 15. The game of pharaoh $P = (GS, RS)$ is defined by initial state GS described in **definition 13** and rule set RS described in **definition 14**.

Remark. There are several other rules in the standard game of Pharaoh, for example when a player plays a card with value *over*, he can choose the suit that other players must match if they want to play a card into the discard pile. However, for reasons of performance and ease of implementation, we have decided not to implement such rules. We have chosen the three or four rules described above, because we believe they represent the cornerstones of the game of Pharaoh.

3.2 The intricacies

The difficulties in implementing a computer opponent for the game of Pharaoh arise from the fact that it is not clear what the most effective strategy is and we do not know of a good heuristic functions for game state evaluation. Also, Pharaoh is a game, where sometimes choosing a particular move at the beginning of the play has little effect on the outcome. For these reasons we have chosen to implement MCTS for the game of Pharaoh and describe the performance and progress of the algorithm throughout the match. It is worth noting that Pharaoh is typically a game with incomplete information[1], but our implementation of MCTS uses complete information.

[1]In games with incomplete information, players only have access to part of the game state. For example, in Pharaoh, each player sees the cards in his hand and the top card in the discard pile. The opposite example is chess, where each player knows the entire game state.

Chapter 4

Implementation

This chapter describes the implementation process, starting with a description of the tool stack we used, followed by a list of the most interesting problems and decisions we made.

4.1 Toolstacks

This bachelor thesis is a follow up for the year project and it carries some decisions since then. One of these decisions is the choice of the programming language to be used. We chose Python because we liked its simplicity and we had some experience with it from other assignments. It is somewhat questionable whether this was a good choice, as the MCTS algorithm would benefit greatly from a more performance-oriented language, as we found out later. Python is a dynamically typed language, but we decided to use type annotations and mypy to check for type correctness. This was a great decision as it helped us find many errors that naturally occur in a project of this size.

We believe that one of the many aims of an undergraduate degree is to learn new things. That is why we decided to use Docker to host and run our code.

4.2 Pharaoh business logic

In our work, we implement the business logic of the game of Pharaoh using the object-oriented paradigm. Initially, we thought that an immutable game state would suit our needs, since the MCTS algorithm stores different versions of the game state in the vertices of the MC tree, and we could examine each macth in more detail. We used pyrsistent's PClass as the base class for our GameState class, which made implementation easy.

However, when we later implemented the first version of MCTS, we encountered rather unimpressive performance. As a result, we had to limit ourselves to a small

number of rules, as mentioned in Chapter 3 about the game of Pharaoh. We suspected that part of the problem might be caused by the fact that the game state is immutable and the other part might be caused by the choice of language. The main disadvantage of an immutable game state is that a new state has to be constructed after every change, which we believe is a big problem for MCTS, since it relies heavily on fast simulation of random playouts.

We measured the time the algorithm spent simulating random playouts and found that it used about 96-98% of the allocated computational budget there. We decided to shift the paradigm to a mutable game state, which cost us a lot of work because the paradigm shift was significant. However, we saw about a twofold improvement in performance, and the time spent simulating random outcomes dropped to about 90%.

Move and Rule

In this section we explain why we distinguish between the notion of a rule and the notion of a move. When we explain a game to someone, we usually describe it in terms of rules. Rules describe what a player can do in a macth. Usually rules do not tell us exactly what a player can do, but they do give several ways of interpreting and using them. However, as we will explain in the subsection on implementing the MCTS algorithm, one of the requirements is the ability to list all legal moves/transitions from a given game state. The problem we faced was how to define a game without manually listing all these moves/transitions to subsequent states, but when needed, how to generate them all.

We decided to separate rules and moves, as we did in our definition, and we implemented the rule as a generator of moves that can be derived from the rule. So a rule generates moves according to a version of Pharaoh that we want to play (more precisely, it generates moves according to the number of players that will be playing the game and the deck that will be used). Now a move class is an object composed of conditions and actions, and it has two member functions - test and apply. The test function returns true if the conditions are met and false if they are not. The apply function changes the state of the game. Before a match starts, a list of moves is generated from the rules. Thanks to this distinction we can easily generate all possible moves if the number of players or the deck changes. If we want to enumerate only moves that are actually applicable to a particular state (which is a requirement of MCTS, as we mentioned earlier), we filter the moves that return true for that state.

Ties

A play of Pharaoh is potentially infinite. The problems arise from the skip a turn rule and from the fact that we do not shuffle the discard pile back into the stock. When

the stock is exhausted, it can happen that no player can play a card into the discard pile (because no one has a card that matches the top card in the discard pile), or that a player can play a card into the discard pile but does not want to. If this happens, everyone gets stuck in an endless skip-a-turn loop.

To prevent this unwanted behaviour, in our implementation of game state we have introduced two counters - the lock counter and the draw counter. The lock counter addresses the first problem. If all players have only one move left (skip a turn), the match ends. The draw counter works similarly: if all players skip a turn for three full rounds, it arbitrarily ends the match, even though the match is not yet stuck in the infinite skip a turn loop.

But if a match ends early, some players will still have cards in hand and there will be a tie. We do not want ties because a tie gives us no additional information about the performance of the MCTS algorithm. And we have seen that the algorithm sometimes prefers a tie when it would probably have lost on any other move. So we decided to remove ties altogether. If a match ends early, the players who have already finished (have no cards left) will be assigned positions normally (in to the order in which they finished). Those who still have cards are ordered in descending order according to the number of cards they have left. And those who have the same number of cards are placed in the reverse order in which they have played, i.e. the player who started each round with cards left is placed last, and the player who played last in each round is placed immediately after the players with no cards left.

4.3 Computer opponents

In order to measure the performance of our MCTS algorithm, we needed to compare it with some alternative algorithms. In this subsection we explain other algorithms/-players that we have created to compare with the MCTS. We also give reasons why we think our solutions should be effective.

We have implemented the following algorithms/players:

1. Random player
2. Biggest tuple player
3. Smallest tuple player
4. Annoying player
5. Back tracking player

The simplest is the **random player**. It does exactly what the name suggests - it randomly chooses a move from the legal moves. As we will see in a later section, this is the worst performing strategy compared to the others.

Biggest tuple player, smallest tuple player and annoying player are inspired by strategies commonly used by human players in the game Pharaoh. However, we have not found any research that has looked at them more closely.

The **biggest tuple player** works like this - from all the legal moves available from a given state, it chooses the one that plays the largest number of cards. For example, if it can play one X, two X, one VII, or draw a card, it will choose to play the two X. The reason this should be a good strategy is that it can quickly play a lot of cards into the discard pile, which should hopefully help it get rid of all of its cards. It is also usually better to play all cards of the same value in one turn.

The **smallest tuple player** is in a way an extension and opposite of the biggest tuple player. It builds on the idea that it is usually better to play cards of the same value together in one turn. But it introduces a new idea that if a player has several cards of the same value, they have a much higher chance of matching the top card in the discard pile. This may sound a little counterintuitive, because if a player has multiple cards of the same value, the likelihood of matching the top card's value is lower (or impossible if the player has all the cards of the same value). On the other hand, the likelihood of a match in suit increases because each card of the same value has a different suit (in the German suited deck). So if a player has two cards of the same value, he can match them with card of two suits, which is on average half the deck. And if he has four cards of the same value, he can always match them with the top card, because he can match cards of any suit. Therefore, it makes sense to play the tuple of cards with the lowest probability of matching the top card in the discard pile before playing a tuple with a higher probability. So the smallest tuple player plays the smallest meaningful tuple of cards of the same value. For example, if it can play one X, two X, one VII, or draw a card, he will play that one VII.

The **annoying player** uses strategy that prevents the next player from playing a card. This strategy is based on the fact that we play Pharaoh with full information, i.e. we play with open cards. In the case of multiple options that result in the following player not being able to play a card, the annoying player chooses according to one of the above strategies.

All of these heuristic players are based on our experience with the game, because we did not find any useful research about strategies for the game of Pharaoh.

Finally, the **back tracking player** search the game state tree for strategy of moves / sequence of moves that always lead to win. It is based on algorithm from the university textbook on computability and complexity written by Dana Pardubská. However this approach is too slow even for the small size of game (which we will define in chapter

5.1). Therefore we add in heuristic move-pruning similar to the one used in MCTS algorithm (mentioned in chapter 2.3 and we will also explain it in more detail in the following subsection 4.4) and we limit the search depth to some arbitrary limit. This makes a reasonably powerful opponent as will be shown later.

4.4 MCTS algorithm

The MCTS algorithm is not difficult to implement once understood. The reason why we implemented the MCTS algorithm ourselves comes from the year project from which our bachelor thesis follows. Back then we wanted to implement the algorithm so that more than two players could play, and we found no implementation of MCTS for multiple players. However, our design was influenced by articles on baeldung.com [2] and analyticsvidhya.com [1].

Our task was to understand the algorithm properly and then implement it so that it would work for multiple players. We were not sure how to keep track of the number of wins in a particular node, because in a multiplayer game the result is not binary win or loss. In our implementation, the node in the MC tree stores the sum of the results for the player who had the previous turn (because his move brought the game to the state in that node).

We were wondering whether we should recycle part of the MC tree after a full round (when the MCTS player is on the turn again), and if so, how we should do it. We implemented this recycling using BFS – the algorithm searches the MC tree until it finds a node with the current state and the roots the MC tree there.

In the explanation of MCTS we write that in the expansion phase the algorithm constructs all states that are reachable from the state in the leaf. But this is not the most efficient way. A better option seems to be lazy expansion, i.e adding only that child from which the playout follows and so on until all legal moves are covered. Otherwise a lot of constructed leaves will never be visited, wasting time and space.

Improving variant

We implement two enhancements to the baseline MCTS. The first enhancement uses move-pruning with domain knowledge-based heuristics. We mentioned this enhancement in Section 2.3. We implemented this by passing a simple heuristic function to the MCTS's object constructor. This function filters out moves that are unlikely to be good options and then returns a list of available moves sorted according to domain knowledge. We have implemented the following three heuristic functions:

1. my_shuffle - this function simply shuffles the available legal moves.

2. biggest_tuple - works similarly to biggest tuple player, i.e. it only returns moves that play all cards of the same value and prefers to play the biggest number of cards.

3. smallest_tuple - acts in the opposite way to biggest_tuple, i.e. it only returns moves that play all cards of the same value and prefers to play the smallest number of cards.

The second improvement uses the Beam Search method, which was also mentioned in chapter 2.3. It takes two parameters limit L and width W. When nodes in the pruning depth have been visited L times, the procedure *prune_tree* is called. The procedure will keep only W most visited nodes in the pruning depth. The remaining nodes in the prune depth are removed along with the subtrees rooted in them. Then the pruning depth is incremented by one so that the children of the nodes that were in the pruning depth are now in the pruning depth. If there are at most W nodes in the prune depth, no nodes are pruned, only the prune depth is incremented.

Note that when W equals 1, the search behaves like a step by step search. And when W approaches infinity, it behaves like a basic MCTS.

Chapter 5

Performance analysis

In this chapter, we present our methodology and the performance results of the MCTS algorithm compared to other algorithms in the game of Pharaoh. We first explain our testing environments, then describe the process of optimising the algorithm parameters, and finally provide a large-scale comparison of the different computer players we have implemented in our work, focusing on MCTS.

5.1 Card decks

In this subsection we explain our test environments, which are characterised by the number of players and the number of cards used in all the simulations described in this chapter.

We only simulate two-player matches, although our implementation of MCTS is capable of playing a game with more than two players. The reason for this is that we feel that the performance comparison can be best seen in a two player game. However, it would be interesting to see how the algorithm would perform in a game with more than two players.

We decided to test our algorithms in two different environments. First, with a small number of cards. This makes the search space smaller, which is more suitable for the backtracking algorithm, and it is also less time consuming to simulate. To achieve this, we define a small deck of cards with only three different suits and five different values, giving a total of 15 cards. Both players are initially dealt 6 cards. From our experience, this seems to be the smallest deck that gives us interesting results (with smaller decks, the results of a match depend too much on the initial state of the game). It may also seem that giving each player 6 cards at the start leaves only 3 cards in the deck to be drawn, and this is true. However, this is intentional, as giving players fewer cards makes the results less dependent on the algorithm used, making it difficult to compare the performance of different algorithms.

We also wanted to see how the algorithm performed in a large game. Therefore, we also test with the full set of German-suited playing cards, i.e. four different suits and eight different values, giving a total of 32 cards. Both players start with 10 cards, which is more than usual in Pharaoh, but still leaves 12 cards in the deck.

5.2 Parameter optimization

The MCTS algorithm has multiple parameters, which we can set:

1. Iteration count[1]
2. Exploration constant[2]
3. Move pruning heuristic [3]
4. Beam Width [4]
5. Beam Limit

In this subsection we describe the process by which we set these parameters. Optimising multivariable models can be challenging, as the number of variables and interactions can quickly become overwhelming. However, there are several optimisation techniques that can be used to efficiently optimise multivariable models.

Gradient based optimisation is a popular technique for optimising multivariable models. This technique involves calculating the gradient of the objective function with respect to each of the model's parameters, and adjusting the parameters in the direction of the gradient to minimise the objective function. The most commonly used gradient-based optimisation algorithm is gradient descent. For more information, see the book on deep learning by Ian Goodfellow and Yoshua Bengio and Aaron Courville [9].

Genetic algorithms are another powerful optimisation technique for multivariable models, introduced by M. Mitchell [12]. This technique involves evolving a population of candidate solutions over time, using selection, crossover and mutation operations to produce better and better solutions. The most commonly used genetic algorithm is the simple genetic algorithm (SGA).

Bayesian optimisation is a relatively new optimisation technique for multivariable models. This technique involves modelling the objective function as a probability distribution and using Bayesian inference to guide the search for the optimal solution. The most commonly used algorithm for Bayesian optimisation is the algorithm based

[1] explained in Section 2.1
[2] explained in Section 2.2
[3] explained in the paragraph about heuristic move-pruning in Chapter 2.3
[4] Width and limit are part of Beam Search, which was explained in paragraph about BMCTS in Chapter 2.3

on the Gaussian process. Bayesian optimisation has been studied by B. Shahriari et al. [14].

As it is beyond the scope of our work to investigate and use these techniques, we have decided not to use them. After a thorough literature review, we implemented several algorithm variants that differed in minor details or parameter configurations. We then ran different versions of these algorithms against each other many times, which allowed us to gain rich insights. Based on this experience, we identified parameters that we considered worthy of further quantitative evaluation, which we focus on in this chapter.

Iteration count

One of the main strengths of the MCTS algorithm is that it can be easily adapted to deal with time constraints. In the case of games, the algorithm can be used to find the next promising move to make in a given amount of time.

Reduction of the time available for the algorithm to search the game tree may, however, reduce the quality of the results produced by the algorithm. This is because the algorithm may not be able to explore enough of the game state tree and may miss important moves.

However, in our work we have chosen to limit the computational budget by the number of iterations the algorithm runs. One reason is that we cannot compare different time constraints between small and large decks. The second is that the time constraint is very sensitive to the performance of the CPU - we got different results depending on which performance mode we used on our test computer. We therefore decided that limiting the number of iterations was better for our purposes, even though it has less real-world relevance.

In general, increasing the number of iterations that the MCTS algorithm is allowed to run will lead to better results because the algorithm has more opportunities to explore the space of possible moves. However, the trade-off is that more iterations also require more computing resources, which may not always be available in practical applications. Therefore, finding the optimal balance between computational resources and number of iterations is an important consideration when using the MCTS algorithm.

To evaluate the performance of our MCTS algorithm, we compared its win rate against a backtracking player in multiple matches. We simulated 400 matches for the small deck and 200 for the large deck for each number of iterations. We also used the same number of simulated matches for each combination described in the following subsections of this chapter. We varied the number of iterations given to the MCTS algorithm and measured its win rate against the backtracking opponent.

The MCTS algorithm was configured with the following settings: exploration con-

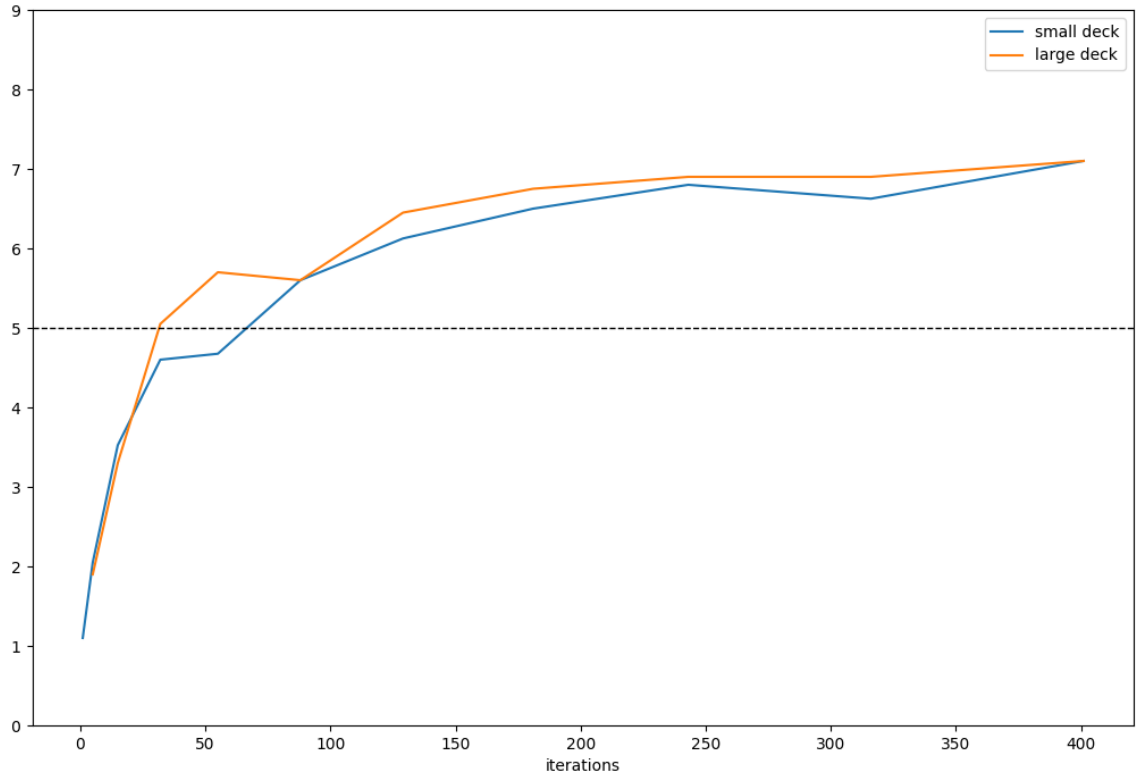


Figure 5.1: Win rate and iterations

stant of 2, heuristic my_shuffle and a limit of 100 000 000 and a width of 100 000 000. The exploration constant value is a recommended baseline setting, as we will explain in the following section, and the my_shuffle heuristic also simulates the behaviour of the baseline MCTS. We used such a high number for limit and width because Beam Search behaves like standard search with these settings. Meanwhile, the backtracking player was configured with the smallest_tuple heuristic and a depth limit of 13 for the small deck and a depth limit of 9 for the large deck.

The results of our experiments were plotted to visually compare the win ratios of the MCTS algorithm and the backtracking opponent over different numbers of iterations. Figure 5.1 shows the win rate of MCTS player as the number of iterations for the MCTS algorithm increases. The results show that the MCTS algorithm outperforms the backtracking player as the number of iterations increases. The results of our experiments showed that the win rate of the algorithm, plotted against the number of iterations given to it, has a shape similar to logarithm, although it is obvious that the relationship cannot be logarithmic because it is not possible to win more than 100% of the matches.

Furthermore, our experiments showed that this logarithmic trend in win rate was consistent across matches with both small and large decks. This suggests that the apparent logarithmic relationship between iteration count and performance is independent of the size of the game tree.

To explore the behaviour of the MCTS algorithm in the following sections in this chapter over a range of iteration counts, we focused our analysis on two different iteration counts: 32 and 243. These iteration counts were strategically chosen to provide insight into the performance of the algorithm at different ends of the spectrum.

The choice of 243 as the higher iteration count was deliberate in order to assess the behaviour of the MCTS algorithm when provided with a reasonably high number of iterations. This number of iterations was chosen to strike a balance between gaining meaningful insights and the practical consideration of simulating multiple matches within a manageable time frame.

On the other hand, the choice of 32 as a lower number of iterations was aimed at investigating the performance of the algorithm with a significantly reduced number of iterations. This choice allowed us to investigate how low we could reduce the number of iterations and still obtain meaningful results.

Exploration constant

The exploration constant in the MCTS algorithm defines the balance between exploration of new moves and exploitation of already discovered and promising moves, as mentioned in Section 2.2.

A higher exploration constant encourages the algorithm to prioritise exploration, resulting in a more extensive search of the game tree. This can help discover new, potentially promising paths in the search space. However, a higher exploration constant can also lead to increased computational complexity and longer search times as the tree grows.

Conversely, a lower exploration constant encourages exploitation by prioritising the use of already known, promising paths. This can lead to faster convergence, but may cause the algorithm to miss potentially valuable but unexplored areas of the search space as the tree grows deeper.

The choice of exploration constant depends on the specific problem, the characteristics of the game or domain being explored, and the desired trade-off between exploration and exploitation. According to Brownley et. al. *Many MCTS enhancements require the optimisation of some parameter, for example the UCT exploration constant ... These values may need adjustment depending on the domain and the enhancements used. They are typically adjusted manually, although some approaches to automated parameter tuning have been attempted [6, 5.2.9 Parameter Tuning].*

In the case of our research, we initially set the exploration constant to 2, as this is a commonly used and recommended value[5]. According to wikipedia value 2 is derived

[5]It should be noted that the exploration constant is often excluded before the square root. Therefore, the recommended value is $\sqrt{2}$

from theoretical analysis, but in practice is chosen empirically, therefore we decided to start with the value 2.

In order to assess the impact of different exploration constants on the performance of the MCTS algorithm, we ran experiments comparing it to another MCTS variant (which we call MCTS_VANILLA) instead of backtracking player. We were concerned about mis-tuning MCTS by comparing it to backtracking player, since it does not search through all possible moves (as it uses the my_shuffle heuristic for move-pruning). The MCTS_VANILLA configuration has these settings: exploration constant of 2, heuristic my_shuffle, 243 iterations, limit and width of 100000000.

The other MCTS player used exactly the same settings except for the exploration constant and iteration count. We varied the exploration constant from zero to approximately three, and we did this for both 32 and 243 iteration counts. By analysing the results at different iteration counts, we aimed to gain insight into the effect of the exploration constant on the performance of the algorithm at both ends. Keeping other parameters constant allowed us to isolate the effect of different exploration strategies on the performance of the algorithm.

During our experiments, we found that a lower value for the exploitation constant gave better performance for both the small and large decks, and for both the 32 and 243 iteration counts. The difference in performance was measurable, especially for the higher iteration count. It resulted in up to 12.5% improvement in win rate compared to the default setting of 2 in the case of 243 iterations and the large deck, as can be seen in Figures 5.2 and 5.3, where we plot the wins with different settings of the exploration constant.

The best value for the small deck matches seems to be equal to 0.4 for both iteration counts. For the large deck matches, we observed the best performance with an exploration constant of 0.29 for 32 iterations and 0.57 for 243 iterations.

In conclusion, we believe that the low value of the exploration constant is advantageous because the game of Pharaoh, as we have implemented it, is deterministic. In deterministic games, where the outcome is determined solely by the actions of the players and there is no random element involved, a low exploration constant can be advantageous in certain cases. The rationale behind this is that a low exploration constant encourages the exploitation of current knowledge and known good moves rather than the exploration of unknown paths. By focusing on exploitation, the algorithm aims to make the most immediate gains based on the available information. One of the notable references that discusses the impact of the exploration constant in MCTS is the paper A Survey of Monte Carlo Tree Search Methods by Cameron Browne et al [6].

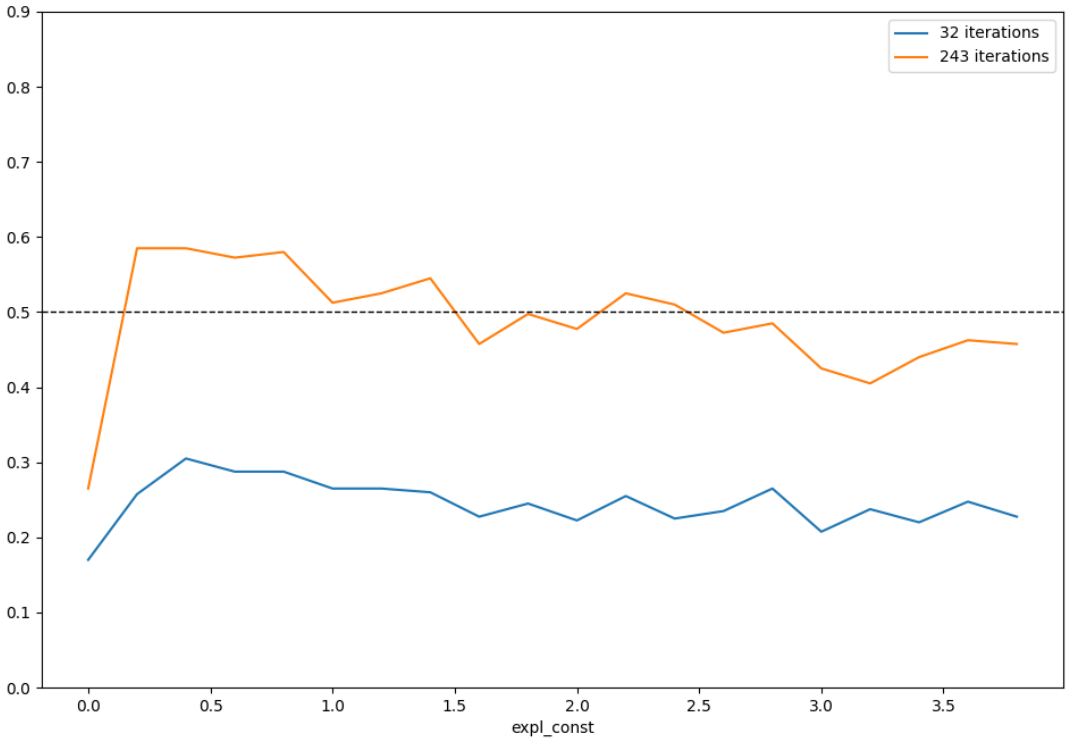


Figure 5.2: Wins with different exploration constant in a game with small deck

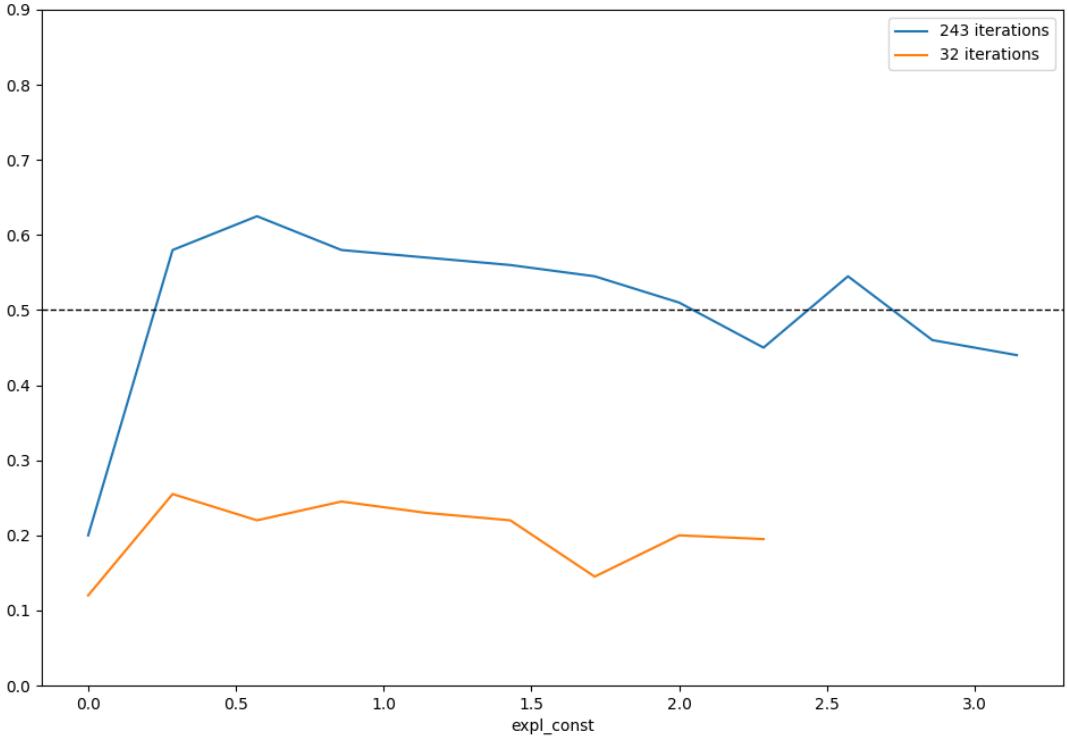


Figure 5.3: Wins with different exploration constant in a game with large deck

Comparison of heuristics

In this section we investigate the impact of incorporating domain knowledge-based move-pruning heuristics, specifically in our case in the context of the game Pharaoh. The move-pruning heuristic is a technique used in both the backtracking player and the MCTS algorithm to optimise the search process by reducing the number of moves considered. A good heuristic should allow the algorithm to prioritise and explore the more promising moves, while excluding less favourable or less relevant moves from consideration.

In our research we defined three heuristics: my_shuffle, biggest_tuple and smallest_tuple. However, only the latter two, biggest_tuple and smallest_tuple, are true move-pruning heuristics. The my_shuffle heuristic simply randomises the available moves, while the biggest_tuple and smallest_tuple heuristics selectively filter out moves based on criteria, that most of the time it is bad not to use all cards of the same value. The biggest_tuple heuristic also favours playing as many cards as possible, while the smallest_tuple heuristic favours playing the smallest sensible amount. For more details go to Section 4.4.

We incorporated the smallest_tuple heuristic into the backtracking player, which significantly improved the efficiency of the search process. By selectively pruning moves that are unlikely to lead to desirable outcomes, we were able to increase the search depth while still maintaining an acceptable search time. This move-pruning mechanism allowed the backtracking player to focus on more promising branches of the game tree, resulting in reduced computational resources and faster decision making.

However, our main focus in this paper is on the MCTS algorithm. We performed a comparison between different variants of the MCTS algorithm that use the biggest_tuple and smallest_tuple heuristics, and a baseline MCTS that uses the my_shuffle heuristic. For the baseline MCTS with the my_shuffle heuristic, we used the best setting of the exploration constant determined in the previous section (we call this variant EXPL_S for small deck and EXPL_L for large deck) .

Due to practical constraints, it was not possible for us to repeat the entire process of optimising the exploration constant for each new heuristic. Therefore, we decided to use three different settings for the exploration constant: the one found in the previous section, a much smaller one (half of the previous best), and a much larger one (four times larger). This allowed us to study the effect of different exploration constants on the performance of the MCTS algorithm with different heuristics.

The other two parameters, Width and Limit, were kept constant for all MCTS variants. They were set to 100 000 000, similar to the settings used in our previous test, to isolate the effect of the heuristics and exploration constants on the performance of the MCTS algorithm.

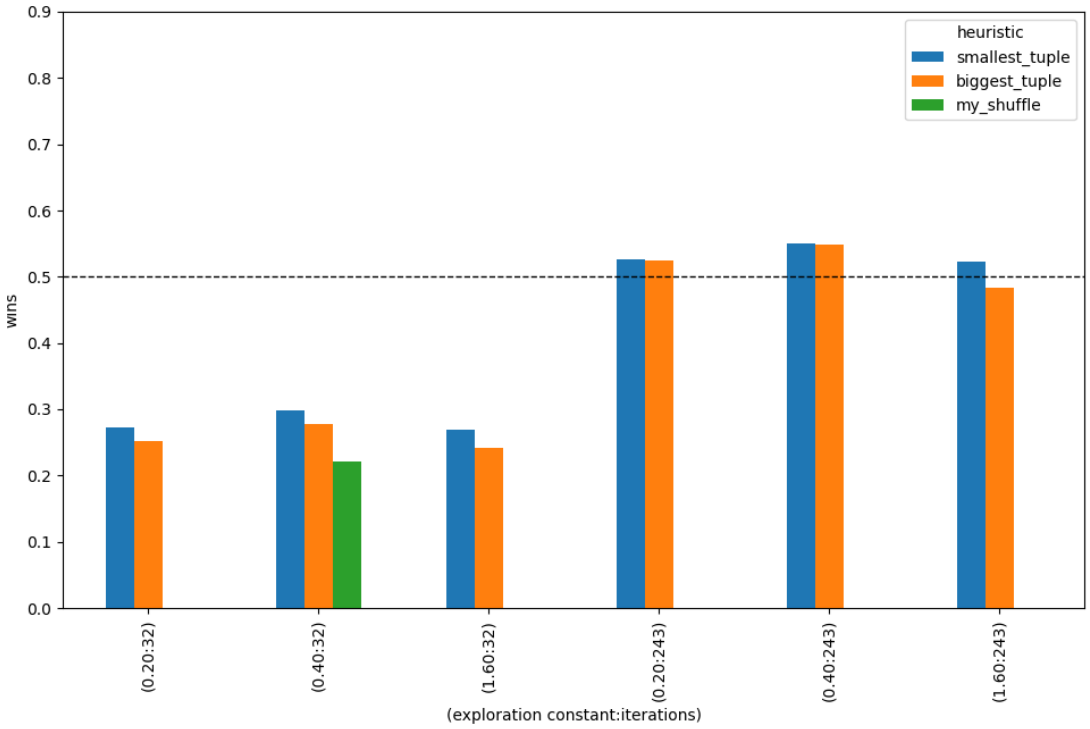


Figure 5.4: Wins with different heuristics in a match with small deck

Through this comparison, we aimed to evaluate how the different heuristics and exploration constants affect the efficiency and effectiveness of the MCTS algorithm in the context of our specific problem domain.

The results differ between large and small deck and between 32 and 243 iterations, so we will describe them separately.

Small deck

First, we describe a small deck and 32 iterations, which can be seen in Figure 5.4 on page 23. In our experiments, the MCTS variant with smallest_tuple and exploration constant of 0.4 performed best. Compared to the my_shuffle heuristic with an exploration constant of 0.4, we achieved an 8% increase in win rate. This suggests that the smallest_tuple heuristic effectively pruned moves that were less likely to lead to desirable outcomes, resulting in a more focused and efficient search.

The results from the small deck and 243 iterations are plotted in the same Figure 5.4 on page 23. In our experiment with the small deck and 243 iterations, we observed that incorporating the smallest_tuple heuristic, along with the previous best setting of the exploration constant equal to 0.4, led to an improvement of up to 5% in the win rate of the MCTS algorithm.

On the other hand, the biggest_tuple heuristic performed slightly worse than the

smallest_tuple heuristic. While it still provided the same move-pruning benefits, the different search order caused it to perform slightly worse at the small and medium exploration constant settings. However, it actually performed significantly worse than the MCTS player with the my_shuffle heuristic at the highest exploration constant setting.

The smallest_tuple outperformed the biggest_tuple at every exploration constant and iteration count setting, suggesting that the order in which moves are explored is important. However, the difference was more pronounced at the lower iteration count, showing that order is more important with fewer iterations.

Overall, in the small deck matches the previous best setting of exploration constant equal to 0.4 seems to be the best for all tested move-pruning heuristics and the best heuristic seems to be smallest_tuple.

Large deck

The results from the large deck are quite different for the biggest_tuple heuristic, as can be seen in Figure 5.5 on page 25. The previous best setting of the exploration constant of 0.29 is actually the worst setting for it. The MCTS variant with the biggest_tuple heuristic actually outperforms smallest_tuple with the small exploration constant setting. Nevertheless, the smallest_tuple heuristic with exploration constant 0.29 has the best win rate of all variants with 32 iterations, and it is a 6% improvement in win rate over the variant with my_shuffle and exploration constant of 0.29.

The smallest_tuple heuristic is also the best performing heuristic with 243 iterations, but the best exploration constant setting is only 0.14, which is only a quarter of the value that was best for the my_shuffle heuristic. It achieves an additional 16% improvement in win rate. Initially, we only ran matches with exploration constants of 0.29, 0.57 and 2.29, but after seeing the negative inverse trend of the smallest_tuple heuristic variant, we decided to run additional matches for it with exploration constant of 0.07 and 0.14. The biggest_tuple heuristic performs similarly to the small deck, with the best exploration constant setting being 0.57. Surprisingly, both smallest_tuple and biggest_tuple perform worse than the my_shuffle variant when the exploration constant is high.

To sum up, the best move-pruning heuristic seems to be smallest_tuple. In our tests the best value of the exploration constant for smallest_tuple heuristic was 0.29 for small number of iterations and 0.14 for large number of iterations.

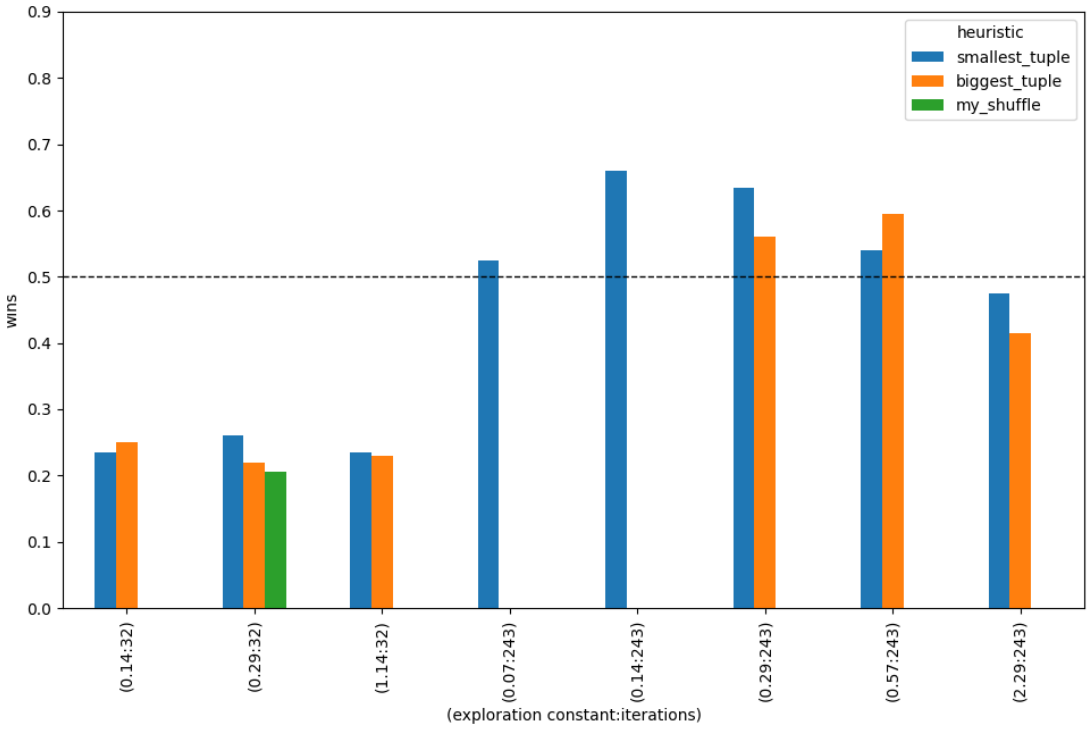


Figure 5.5: Wins with different heuristics in a game with large deck

Width and limit

The results of our experiments with BMCTS (Beam Search Monte Carlo Tree Search) were disappointing. Despite our initial expectations and efforts to optimise the width and limit parameters, the inclusion of BMCTS did not improve the performance of the algorithm. The BMCTS variant consistently underperformed compared to the baseline algorithm without BMCTS.

We first tried BMCTS in a small deck game. We tested against the variant used in the previous phase - MCTS_EXPL_S. The BMCTS variants used the smallest_tuple heuristic and an exploration constant of 0.4, which gave the best results in the previous phase. We tried widths of 1, 2, 4 and 8. We combined them with limits of 4, 8, 16. The values for limit were chosen strategically, because with a limit of 4 the search tree should be pruned 8 times, while with a limit of 16 the tree should be pruned only once. We only tested combinations where the limit was greater than the width, otherwise the prune tree method would be called earlier than all the nodes at prune depth were visited.

As can be seen in Figure 5.6, the algorithm performs better with the largest values of both width and limit. The best win rate is only 0.265 and was achieved with a width of 8 and a limit of 16. It seems that it would be best not to prune the tree at all, as the variant from the previous section (without Beam Search) with the smallest_tuple

heuristic and an exploration constant of 0.4 achieved a win rate of 0.298.

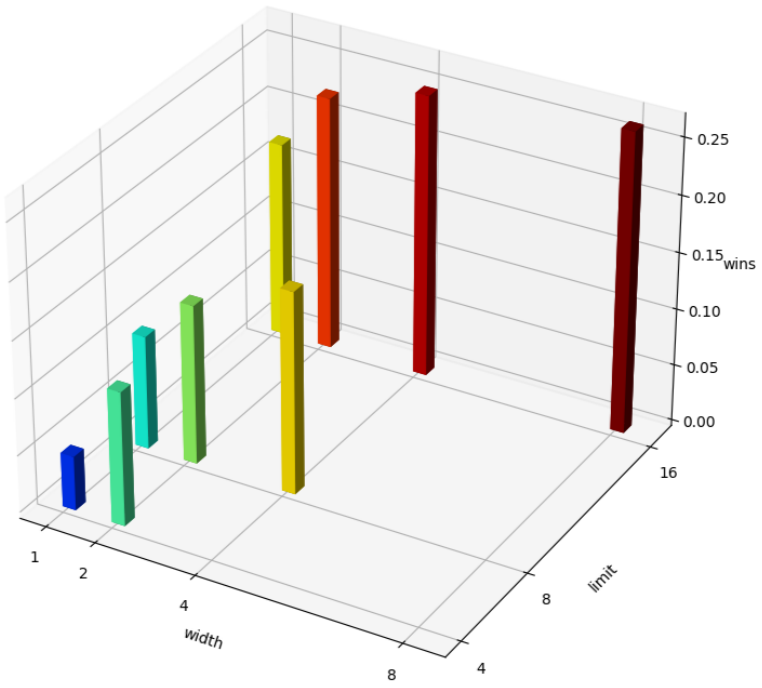


Figure 5.6: Wins with different for different values of width and limit in a game with small deck using 32 iterations

For 243 iterations we used 2,4,8,16 for width and 4,8,16,32,64,128 for limit. The performance is even worse with 243 iterations, as can be seen in Figure 5.7. None of the tested combinations achieved a win rate above 0.5. The best result - a win rate of 0.4975 - was achieved with a combination of width 4 and limit 128. This variant pruned the search tree twice. In comparison, the best variant from the previous phase, which did not use beam search, achieved a win rate of 0.55 (using the my_shuffle heuristic and an exploration constant of 0.4).

In the matches with the large deck, we compared the BMCTS variants with the MCTS_EXPL_L variant. We tried width and limit values similar to the small deck and the performance was similarly disappointing. The results with 32 can be seen in Figure 5.8. The win rates are more chaotic compared to small deck, which could be partly caused by simulating on 200 matches with large deck and 400 matches with small deck. Still, the best win rate of 0.235 was achieved by the variant with the largest width and limit of 8 and 16 respectively, which is not as good as the win rate of 0.26 achieved in the previous phase under the same conditions without using the BMCTS improvement.

Finally, the best result of 0.495 in a game with a big deck using 243 iterations was achieved by the variant with a width of 8 and a limit of 128. In comparison, the best variant with 243 iterations from the previous phase achieved a win rate of 0.66 using the my_shuffle heuristic and an exploration constant of 0.14. The results are plotted

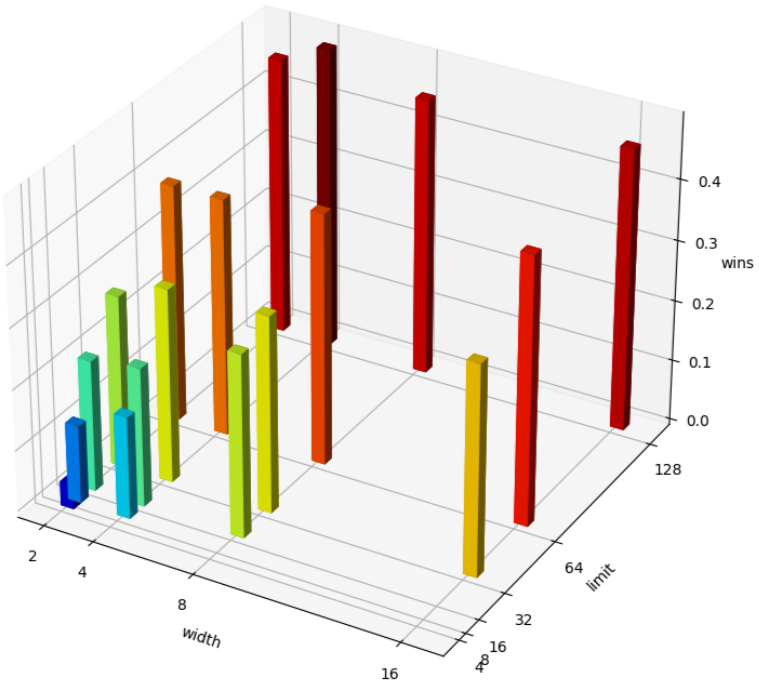


Figure 5.7: Wins with different for different values of width and limit in a game with small deck using 243 iterations

in Figure 5.9

Further analysis and investigation is required to understand the underlying factors contributing to the poor performance of BMCTS in our specific scenario. This result highlights the complexity and variability of algorithmic performance, and the need for careful evaluation and comparison of different approaches.

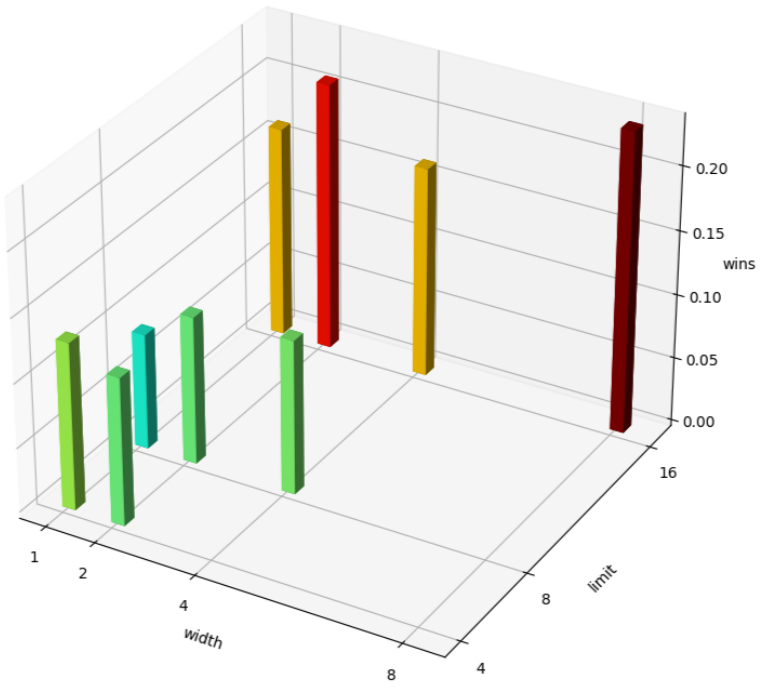


Figure 5.8: Wins with different for different values of width and limit in a game with large deck using 32 iterations

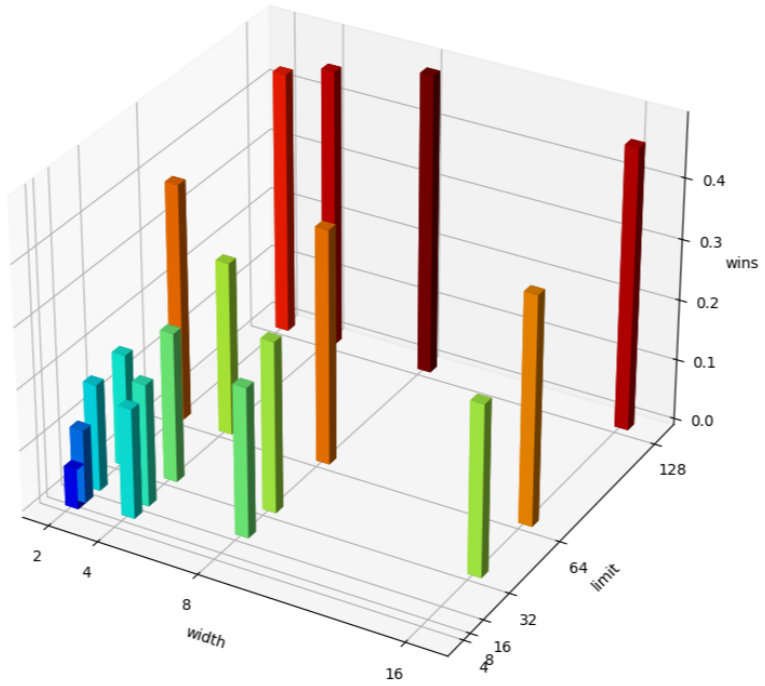


Figure 5.9: Wins with different for different values of width and limit in a game with large deck using 243 iterations

5.3 The great tournament

In this section we make a big comparison of the players and variants we implemented in our work. We also analyse and compare actual times which each algorithm used in simulated matches.

List of Players

In this subsection we list and explain all the players which we used in the tournament.

The simplest players are random player, biggest tuple player, smallest tuple player and annoying player which were already mentioned in Section 4.3 about computer opponents. In our tournament the random player is called RANDOM, the biggest tuple player is called BIGGEST, the smallest tuple player is called SMALLEST and the annoying player is called ANNOYING. Apart from annoying player there are no parameters to set to change the behaviour of these players and the same players are used for both small and large deck games.

However, the behaviour of the annoying player can be further specified by what it chooses to do if there are multiple annoying moves (moves that prevent the other player from playing a card). In this case we chose a behaviour similar to the smallest tuple player.

We compare two slightly different backtracking players for each size - the BT_SMALL and BTSH_S are used in matches with small deck and BT_LARGE and BTSH_L are their counterparts in large deck matches. Backtracking players with the BTSH prefix use my_shuffle heuristic, the other variants use the smallest_tuple heuristic. Each player has a different search depth limit, because we tried to even out the search times by adjusting the search depth limit. BT_SMALL has a limit 13, BTSH_S has a limit of 9, BT_LARGE has a limit of 9 and BTSH_S has a limit of 7.

We have four variants of MCTS for each size - VANILLA, EXPL_S, HEUR_S and HEUR_S32 are used in matches with small deck and MCTS_VA-NILLA, MCTS_EXPL_L, HEUR_L and HEUR_L32 are compared in matches with large deck.

The VANILLA variant has the same settings for both sizes, as we used it as a starting point, when we were optimising the parameters. Then each variant should represent further optimised settings from the previous chapter. The only two variants that use 32 iterations are HEUR_S32 and HEUR_L32, as we wanted to see how much we achieved by tuning MCTS with this lower number of iterations. We decided not to use any variants with Beam Search, because this improvement did not improve the performance.

For the sake of clarity, we decided to list the parameter values of each MCTS variant in the following Table 5.1.

	iterations	width	limit	heuristic	expl_const
VANILLA	243	100000000	100000000	my_shuffle	2.0
EXPL_S	243	100000000	100000000	my_shuffle	0.4
EXPL_L	243	100000000	100000000	my_shuffle	0.571429
HEUR_S32	32	100000000	100000000	smallest_tuple	0.4
HEUR_S	243	100000000	100000000	smallest_tuple	0.4
HEUR_L32	32	100000000	100000000	smallest_tuple	0.285714
HEUR_L	243	100000000	100000000	smallest_tuple	0.142857

Table 5.1: Overview of MCTS variants

Small deck

We simulated 400 matches for each pair of players and recorded the results in table 5.2. Each row represents one player/algorithm participating in the tournament. The values represent the win rate of the player/algorithm in the row against the player/algorithm in the column (e.g. if in the row ANNOYING and in the column RANDOM the value is 0.94, this means that ANNOYING won 94% of the matches against RANDOM). The diagonal of the table (values where the row and column names are the same) represents the performance of each player/algorithm against themselves, resulting in " - " values since it doesn't make sense to compare a player/algorithm against itself.

name	ANNOYING	BIGGEST	BT_SMALL	BTSH_S	**EXPL_S**	**HEUR_S**	**HEUR_S32**	**VANILLA**	RANDOM	SMALLEST	mean
ANNOYING	-	0.70	0.44	0.58	0.18	0.15	0.37	0.22	0.94	0.55	0.46
BIGGEST	0.30	-	0.35	0.53	0.04	0.03	0.12	0.04	0.86	0.40	0.30
BT_SMALL	0.56	0.65	-	0.64	0.32	0.37	0.54	0.32	0.88	0.66	0.55
BTSH_S	0.42	0.47	0.36	-	0.22	0.19	0.36	0.27	0.72	0.41	0.38
EXPL_S	0.82	0.96	0.68	0.78	-	0.40	0.66	0.59	0.98	0.80	0.74
HEUR_S	0.85	0.97	0.63	0.81	0.60	-	0.78	0.60	0.99	0.84	0.78
HEUR_S32	0.63	0.88	0.46	0.64	0.34	0.22	-	0.30	0.96	0.65	0.56
VANILLA	0.78	0.96	0.68	0.73	0.41	0.40	0.70	-	0.99	0.77	0.71
RANDOM	0.06	0.14	0.12	0.28	0.02	0.01	0.04	0.01	-	0.11	0.09
SMALLEST	0.45	0.60	0.34	0.59	0.20	0.16	0.35	0.23	0.89	-	0.42

Table 5.2: Win rates in matches with small deck

If we were to award one point for each win rate above 50%, we would get the following leaderboard:

Position	Name	Points
1	**HEUR_S**	9
2	**EXPL_S**	8
3	**VANILLA**	7
4	BT_SMALL	6
5	**HEUR_S32**	5
6	ANNOYING	4
7	SMALLEST	3
8	BIGGEST	2
9	BTSH_S	1
10	RANDOM	0

We can see that the best variant was HEUR_S, as we expected, as it was the best tuned variant. What is surprising, however, is the low ranking of BTSH_S, which came second to last. The only MCTS variant with 32 iterations - HEUR_S32 - managed to win 5 matches, enough for 5th place. It did not manage to beat any other MCTS variant, nor the backtracking player BT_SMALL.

If we compare the means, we get slightly different order - HEUR_S32 and BT_SMALL swap positions. The HEUR_S32 has an mean win rate of 0.56, which puts it just ahead of BT_SMALL, which has a mean win rate of 0.55. These two players are in fact very close to each other. In their duel, BT_SMALL wins only 54% of the matches. However, this is quite a disappointing result as we achieved the same win rate in Section 5.2 for the baseline MCTS variant with 32 iterations.

Similarly, BTSH_S and BIGGEST swap places when comparing means. This brings us to the comparison of the backtracking players. As can be seen in table 5.3, the shuffle player uses more time on average, but its mean win rate is much worse than that of the smallest_tuple variant. In head-to-head matches, the shuffle variant won only 36% of the matches, while spending on average 1.26s per move, which is 37% more than the smallest_tuple variant.

It is also worth noting that although HEUR_S managed to beat all opponents (i.e. win more than 50% of matches against them), including other MCTS variants, it did not have the best win rate against BT_SMALL player. Both VANILLA and EXPL_S won 68% of the matches, while HEUR_S won only 63%.

Regarding the time needed to calculate the next move, the vanilla variant takes the most time of all variants, the second is EXPL_S with slightly less time, followed by HEUR_S which is 38% more efficient than VANILLA, and finally the fastest variant is HEUR_S32 as it needs only 32 iterations.

name	ANNOYING	BIGGEST	BT_SMALL	BTSH_S	**EXPL_S**	**HEUR_S**	**HEUR_S32**	**VANILLA**	RANDOM	SMALLEST	mean
ANNOYING	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
BIGGEST	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
BT_SMALL	0.73	0.73	-	0.92	0.66	0.74	0.68	0.77	0.89	0.83	0.77
BTSH_S	1.26	0.96	1.26	-	1.11	1.25	1.15	1.18	1.10	1.37	1.18
EXPL_S	0.76	0.73	0.70	0.74	-	0.73	0.72	0.66	0.82	0.84	0.74
HEUR_S	0.53	0.53	0.54	0.56	0.49	-	0.51	0.48	0.62	0.58	0.54
HEUR_S32	0.13	0.13	0.12	0.12	0.13	0.12	-	0.13	0.13	0.14	0.13
VANILLA	0.87	0.89	0.80	0.88	0.80	0.81	0.83	-	0.95	0.97	0.87
RANDOM	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00
SMALLEST	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00

Table 5.3: Average time per move in matches with small deck (in seconds)

The MCTS variants with 243 iterations are very close to BT_SMALL in terms of time. BT_SMALL is on average faster than the vanilla variant, but slower than the other two variants.

A negligible amount of time, rounded to zero, is used by RANDOM, BIGGEST, SMALLEST and ANNOYING.

Large deck

In the large deck tournament we played 100 matches for each pair of algorithms and we recorded the results in table 5.4. In the large deck matches we see two interesting trends.

Firstly, backtracking players perform much worse compared to MCTS variants and simple heuristic players (e.g. ANNOYING). The most likely explanation is that lowering their search depth limit made them less effective. However, it was necessary to keep their time within practical limits. It should be noted, however, that we did not manage to balance this as well as we did balance the small deck matches. The reason for this imbalance is that we wanted to use same iteration count for both small and large deck games. On average, VANILLA spent 2.3 times more time per move than BT_LARGE. In comparison, VANILLA takes on average only 1.13 times more time per move than BT_SMALL (in small deck matches). The average time per move for each algorithm is recorded in table 5.5.

Secondly, the MCTS variants perform better overall against backtracking players

name	ANNOYING	BIGGEST	BT_LARGE	BTSH_L	**EXPL_L**	**HEUR_L**	**HEUR_L32**	**VANILLA**	RANDOM	SMALLEST	mean
ANNOYING	-	0.71	0.50	0.76	0.06	0.03	0.15	0.14	0.97	0.59	0.43
BIGGEST	0.29	-	0.45	0.71	0.00	0.00	0.03	0.01	0.80	0.38	0.30
BT_LARGE	0.50	0.55	-	0.72	0.25	0.26	0.31	0.32	0.86	0.60	0.49
BTSH_L	0.24	0.29	0.28	-	0.14	0.14	0.21	0.13	0.57	0.26	0.25
EXPL_L	0.94	1.00	0.75	0.86	-	0.40	0.75	0.64	0.99	0.83	0.80
HEUR_L	0.97	1.00	0.74	0.86	0.60	-	0.86	0.66	0.97	0.95	0.85
HEUR_L32	0.85	0.97	0.69	0.79	0.25	0.14	-	0.36	0.94	0.84	0.65
VANILLA	0.86	0.99	0.68	0.87	0.36	0.34	0.64	-	0.99	0.80	0.73
RANDOM	0.03	0.20	0.14	0.43	0.01	0.03	0.06	0.01	-	0.07	0.11
SMALLEST	0.41	0.62	0.40	0.74	0.17	0.05	0.16	0.20	0.93	-	0.41

Table 5.4: Win rates in matches with large deck

as well as simple heuristic players. We think that the main reason for the better performance against backtracking players is the imbalance in time spent (compared to the small deck tournament). We also think that the larger game gives the better algorithm a greater advantage, which explains the improvement in performance against simple heuristic players.

If we were to award one point for each win rate above 50%, as in the small deck matches, we would get the following leaderboard:

Position	Name	Points
1	**HEUR_L**	9
2	**EXPL_L**	8
3	**VANILLA**	7
4	**HEUR_L32**	6
5	BT_LARGE	4
	ANNOYING	4
7	SMALLEST	3
8	BIGGEST	2
9	BTSH_L	1
10	RANDOM	0

The leaderboard is basically the same as in the small deck matches, except BT_LARGE goes backwards. While BT_SMALL managed to place 4th, BT_LARGE was addition-

name	ANNOYING	BIGGEST	BT_LARGE	BTSH_L	**EXPL_L**	**HEUR_L**	**HEUR_L32**	**VANILLA**	RANDOM	SMALLEST	mean
ANNOYING	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
BIGGEST	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
BT_LARGE	6.08	7.86	-	7.51	6.80	6.61	6.69	6.51	7.59	6.47	6.90
BTSH_L	8.16	7.92	6.23	-	5.95	6.49	7.20	6.52	6.09	6.58	6.79
EXPL_L	13.76	15.14	15.00	16.10	-	12.42	13.40	13.89	16.11	15.98	14.64
HEUR_L	10.20	10.94	11.24	12.39	9.49	-	10.18	10.34	13.73	11.67	11.13
HEUR_L32	1.95	2.08	1.97	2.15	1.86	1.94	-	1.84	2.11	1.99	1.99
VANILLA	15.49	15.88	15.52	16.65	14.37	14.11	14.27	-	17.06	17.58	15.66
RANDOM	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00
SMALLEST	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00

Table 5.5: Average time per move in matches with large deck (in seconds)

ally beaten by HEUR_L32 and placed 5th next to ANNOYING. However, BT_LARGE beats ANNOYING when comparing mean win rates. BT_LARGE has a mean win rate of 0.49 and ANNOYING has a mean win rate of 0.43. Unlike in the small deck matches, HEUR_L32 is quite an improvement over the baseline MCTS with 32 iterations from subsection 5.2.In that subsection, the MCTS variant only managed to win 51% of the matches, whereas the HEUR_L32 variant wins 69% of the matches against the same opponent - BT_LARGE.

In the battle of the backtracking players it is clear that the smallest_tuple heuristic helps BT_LARGE a lot. While BTSH_L could only beat RANDOM and had an average win rate of only 0.25, BT_LARGE could also beat BIGGEST, SMALLEST and came close to beating ANNOYING. In their duel, BT_LARGE won 72% of the matches, using on average an extra second per move. In conclusion, backtracking players struggle with the more possibilities offered by the larger deck, but their performance in head-to-head matches is only slightly in favour of the smallest_tuple variant compared to small deck.

Similar to the small deck tournament, HEUR_L beats all other players, but does not achieve the best score against all of them. The gap to the best score is in the range of 1% to 3%, with the largest gap being against RANDOM, which is probably just a matter of chance. This suggests that we managed to tune HEUR_L better (i.e. less specifically) than HEUR_S, as HEUR_S had a 5% deficit to best win rate against BT_SMALL in small deck matches.

The comparison of the times of the different MCTS variants is basically the same

as in the small deck matches. The vanilla variant takes the most time of all variants, second is EXPL_L, followed by HEUR_L which is 29% more efficient, and finally the fastest variant is HEUR_L32 because it takes only 32 iterations and only 13% of the time of VANILLA.

Chapter 6

Evolution of a play

In this chapter we describe and analyse how the MC tree evolves during a game. First we define and explain the metrics we used, and then we present the results.

6.1 Metrics

We wanted to better understand how the algorithm works. To gain more insight into how it works, we thought it might be useful to observe and describe the shape, balance and other properties of the MC tree. We let the algorithm find the next move in a given computational budget, and then we analyse the MC tree built during the search process. We call this type of data a tree metric.

Definition 16. Maximum depth is a tree metric that records the length of the longest path from the root to any leaf in the MC tree.

Definition 17. Average depth is a tree metric that records the average length of all paths from root to leaves in the MC tree.

Definition 18. Maximum degree is a tree metric that records the maximum degree (or the maximum number of children of a node) in the MC tree.

Definition 19. Average degree is a tree metric, that records the average degree in the MC tree.

When the MCTS algorithm runs, it records in each node how many playouts have been simulated in the subtree of each node and how many of them have been won by the player who was on the move in the parent of the given node. If we record the number of visits in the root after each move, we get the following tree metric.

Definition 20. Root visits is a tree metric that records how many playouts have been simulated in the MC tree, i.e. how many iterations the algorithm has run.

Definition 21. Size is a tree metric that measures the size of the MC tree, i.e. the number of nodes in it.

Definition 22. Number of terminal leaf nodes is a tree metric that records how many leaves in the MC tree are terminal, i.e. leaves that store terminal game state and therefore cannot have children in the future.

We also wanted to analyse and describe the balance of the MC tree. In our work we use imbalance indices taken from the paper on tree balance indices [7]. However, these indices are not defined for trees that have nodes with only one child, which naturally occurs in the MC tree when there is only one move available. To overcome this discrepancy, we decided to ignore such nodes in the MC tree. This is equivalent to 'smoothing' paths in the MC tree that have all inner vertices with only one child, i.e. the child c of a node n with parent p is hung directly under the parent p if n has only one child c and so on recursively, or if root has only one child it is replaced by its child. We believe that calculating the value of the index for this altered tree gives us a good insight into the balance of the tree, as the MCTS algorithm does not need to branch when there is only one move available and therefore does not make any interesting changes to the shape of the MC tree.

Definition 23. (Imbalance index). A tree shape statistic t is called an *imbalance index* if and only if:

1. the caterpillar tree with n leaves is the unique tree maximizing t on any tree with n leaves for all $n \geq 1$
2. the fully balanced tree (a rooted binary tree with n leaves in which all inner vertices are balanced) is the unique tree minimizing t on binary trees with n leaves for all $n = 2h$ with $h \in N \geq 0$.

Definition 24. Sackin index is sum of the depths of the leaves of MC tree. Sackin index is an imbalance index.

Definition 25. Cophenetic index is sum of the cophenetic values of all different pairs of leaves of MC tree. It adds up the depths of the lowest common ancestor of every pair of different leaves. Cophenetic index is an imbalance index.

Let t be a balance or imbalance index, let T be a tree with n leaves, let $min(t, n)$ be the minimum value of index t on any tree with n leaves and $max(t, n)$ the maximum value. Then we define

$$\tilde{t}(T) = \frac{t(T) - min(t, n)}{max(t, n) - min(t, n)}$$

and we call $\tilde{t}$ normalised t.

As the MC trees are not the same size, but we wanted to compare them, we decided to normalise them using the usual affine transformation, as suggested in the paper on tree balance indices. We therefore use the normalised Sackin index and the normalised Cophenetic index, but we will simply refer to them as the Sackin or Cophenetic index.

Definition 26. Win expectation is a tree metric that records the ratio of won to total playouts and should therefore describe how likely the MCTS algorithm is to win.

6.2 Results

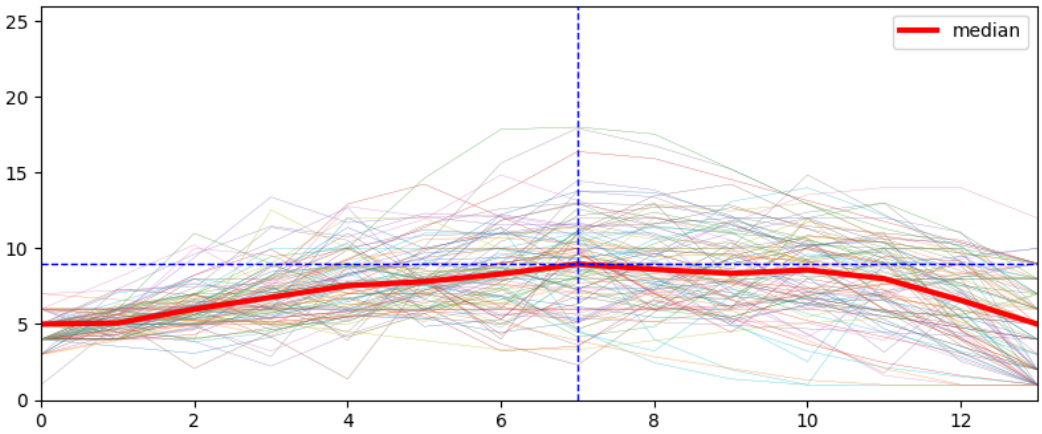
We played 100 matches with a large deck of the three variants - VANILLA, EXPL_L, HEUR - against another VANILLA. Because in this chapter there is no need to differentiate between small and large variant (e.g HEUR_L vs HEUR_S) we will call them simply EXPL and HEUR. We chose these variants because they represent different steps in the tuning process. We decided not to include variants with 32 iterations, as the MC tree of such variants is small and often produces very spiky tree metrics.

Due to the different lengths of the matches, we decided to normalise the data by interpolating it to an average length (which was 13.13) rounded up. This allows us to plot and compare the tree metrics together between matches of different lengths.

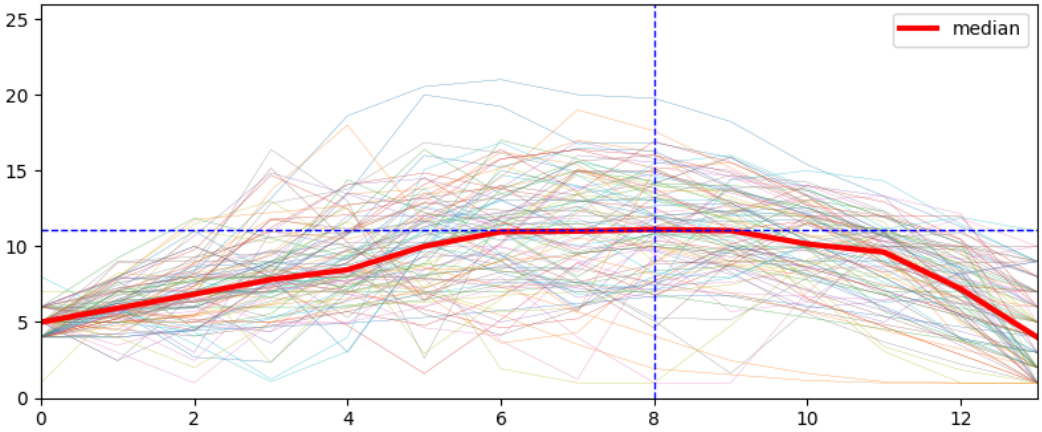
The first metric we describe is the maximum depth. We have plotted the maximum depths from each variant in Figure 6.1. In each plot there are lines for each match and the median with the bold red line. The blue dashed cross indicates the maximum of the median. We observe that all variants reach the highest value of maximum depth in the middle of the match. The VANILLA variant has the smallest median with a peak just below 10. The EXPL variant has a higher median of maximum depth - slightly above 10 - reflecting the lower value of the exploration constant. The move-pruning heuristic of HEUR allows it to search the game tree much deeper, resulting in a peak median of over 15. We can also see that the maximum depth of the MC tree of HEUR has higher extrema and reaches peaks earlier than the other two.

The plots of average depth have a very similar shape to maximum depth, so we decided not to include them. The maximum of medians of average depth for VANILLA is 5.85 and occurs after the 9th move, the maximum of medians for HEUR is 6.81 and the maximum of medians for HEUR is 8.95 and occurs after the 7th move.

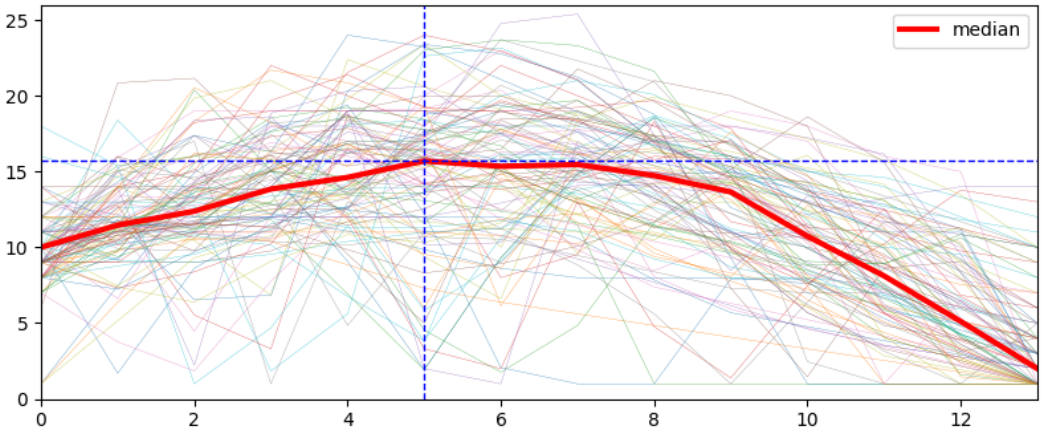
Next we compare the metric maximum degree of the MC tree. The results are plotted in Figure 6.2, where the bold red line is the median and the blue dashed cross marks the median of the maximum degree. All plots show a downward trend. While the plots of VANILLA and EXPL are very similar, the plot of HEUR is different in that it has no match with a maximum degree above 15 and the median also has a slightly smaller



(a) VANILLA



(b) EXPL_L



(c) HEUR

Figure 6.1: Maximum depth

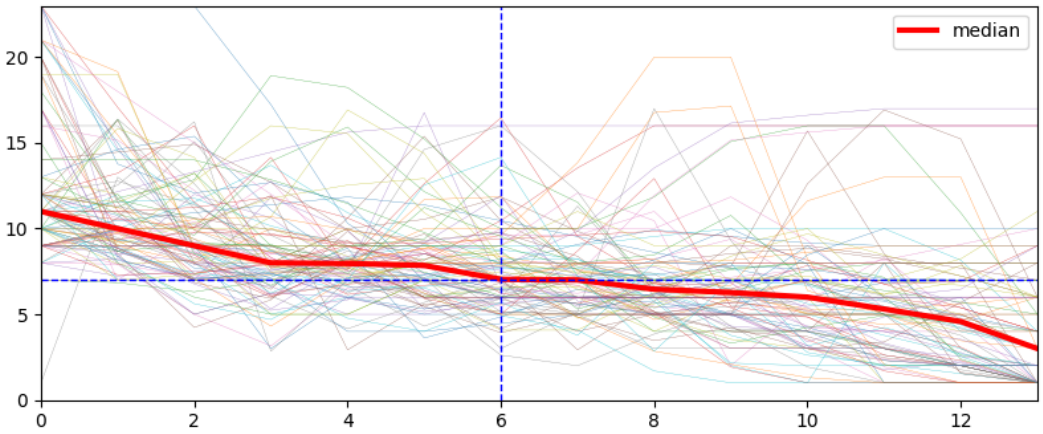
value. It seems that the move-pruning heuristic is more effective at the beginning of the match, as the slope is less steep for HEUR than for the other two.

The situation with the average degree is similar to that with the maximum degree and is shown in Figure 6.3. The only interesting thing is the flat part or even the upward trend of the average degree a few moves before the end, especially visible in the plot of HEUR, but also for the other two. Our hypothesis is that this behaviour occurs because the losing player has to draw cards, which increases the number of moves available to him and thus the average degree of the MC tree.

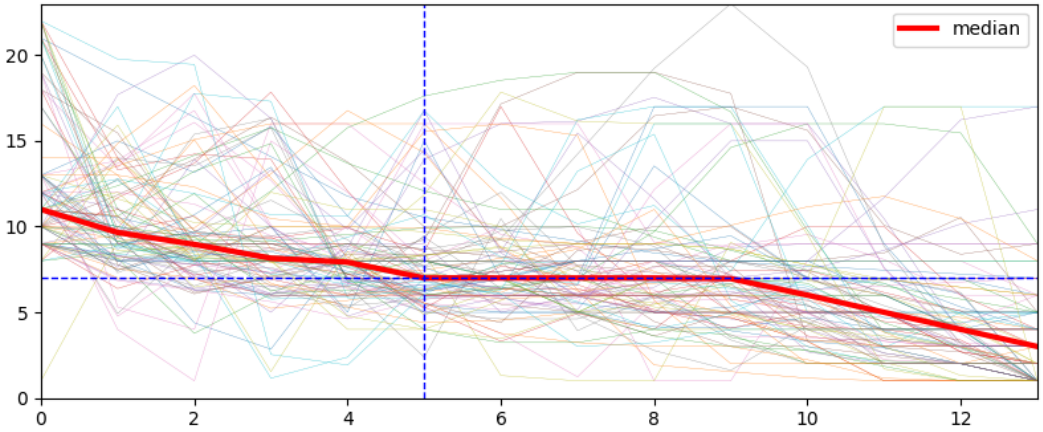
The root visits tree metric is interesting because we naturally expect the root visits to be the number of iterations we set for each variant - in our case 243 - or be very slightly above that, but as part of the MC tree is often recycled this is not always the case as can be seen in Figure 6.4. All variants start at 243, but EXPL and HEUR have an upward sloping median. So the slopes are different and even more different are the high extrema. After a few moves, the line of medians of the VANILLA variant is flat and its maximum is 273.73. The EXPL variant has a plot with an upward slope and a higher maximum of medians at 386.19. Finally, the HEUR variant has a similar upward slope and its maximum median is 425.5. It is also interesting to look at specific matches. While VANILLA has no match with more than 1000 root visits, EXPL has three and HEUR several. It is very interesting that such an event occurs at all, because the algorithm is only allowed to make 243 iterations per move, and having so many root visits means that it must have acquired them in previous moves. If all the playouts were simulated under a node (which becomes a root in the future), it would take more than 6 matches to get 1600 visits in such a node. We think that the algorithm gains an additional advantage as it builds up more root visits. It is also interesting that EXPL is closer to HEUR in root visits, but closer to VANILLA in maximum depth. We therefore think that it becomes progressively more difficult to recycle more of the tree as it becomes necessary to correctly guess the future moves.

Unlike the root visits, the size of the MCTS tree does not continue to grow throughout the match, as shown in Figure 6.5. Our explanation is as follows: As the match progresses towards the end, the paths in the MCTS tree reach terminal states and the tree stops growing. It seems that this happens earlier for the HEUR variant and for EXPL.

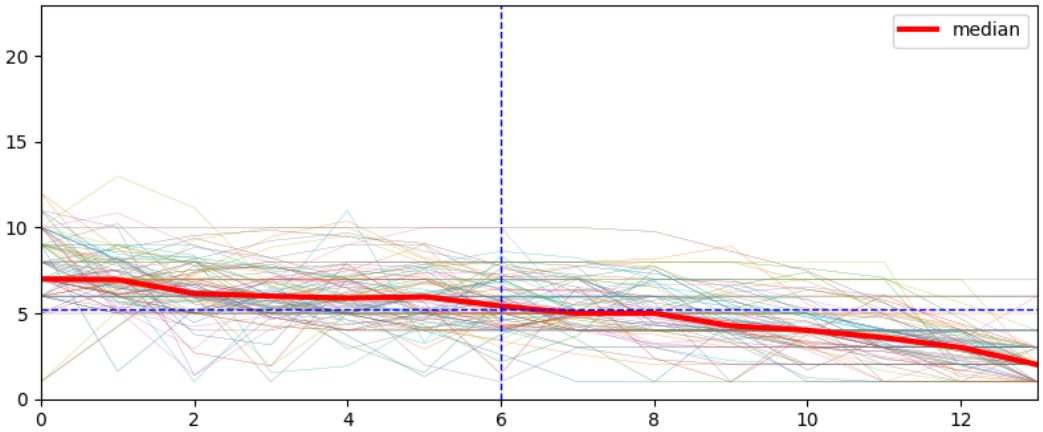
To confirm the hypothesis from the previous paragraph, we analysed the number of terminal leaf nodes. We plotted the medians of the number of terminal leaves in Figure 6.6. We decided not to plot all the matches as we did with the previous metrics because the medians became too flat (the number of terminal leaves goes up to 90). HEUR has the most terminal leaves in the beginning and middle of the match out of all variants. At the end of the matches, EXPL overtakes HEUR in the number of terminal leaves. Except for VANILLA, it seems that the decrease in size starts a bit before the



(a) VANILLA

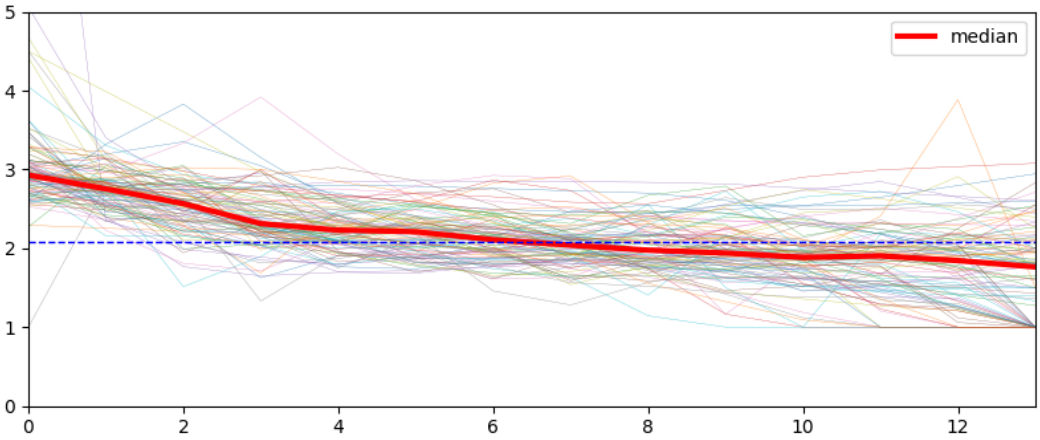


(b) EXPL_L

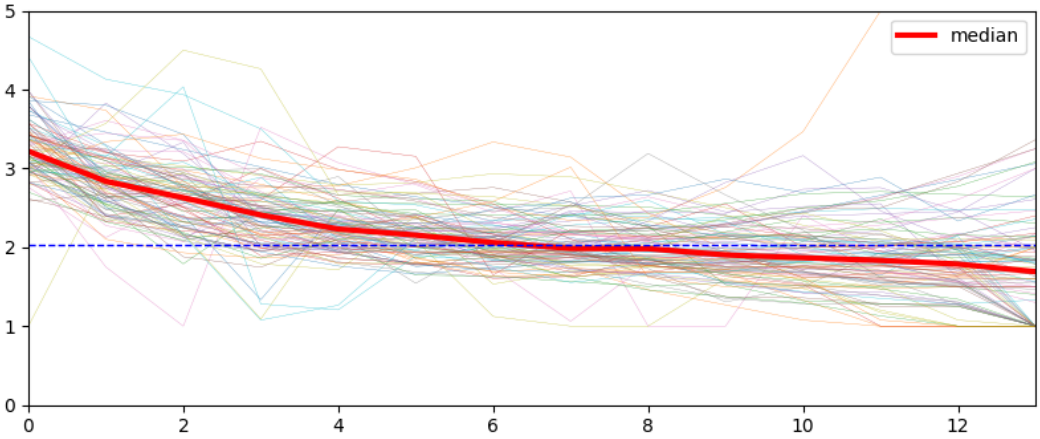


(c) HEUR

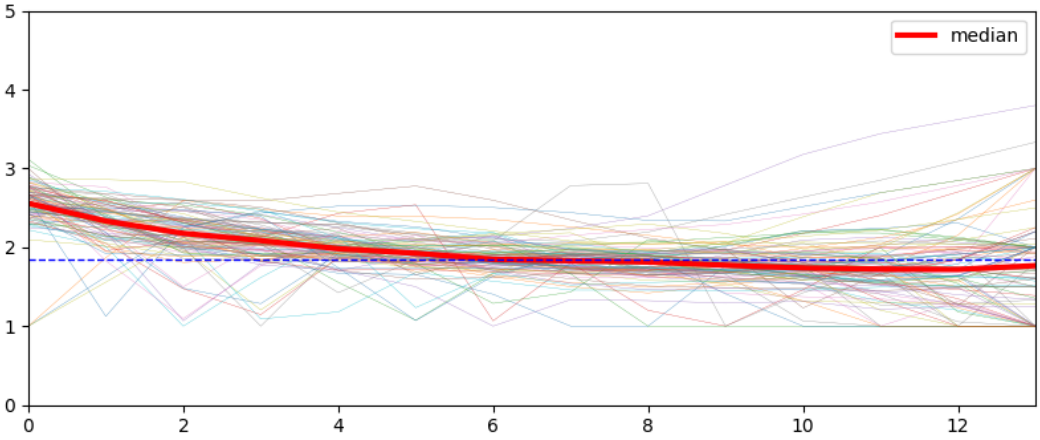
Figure 6.2: Maximum degree



(a) VANILLA

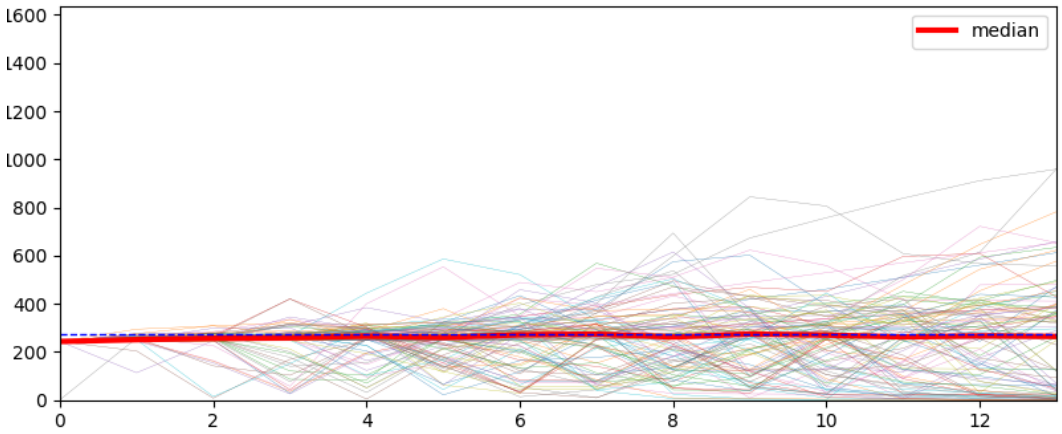


(b) EXPL_L

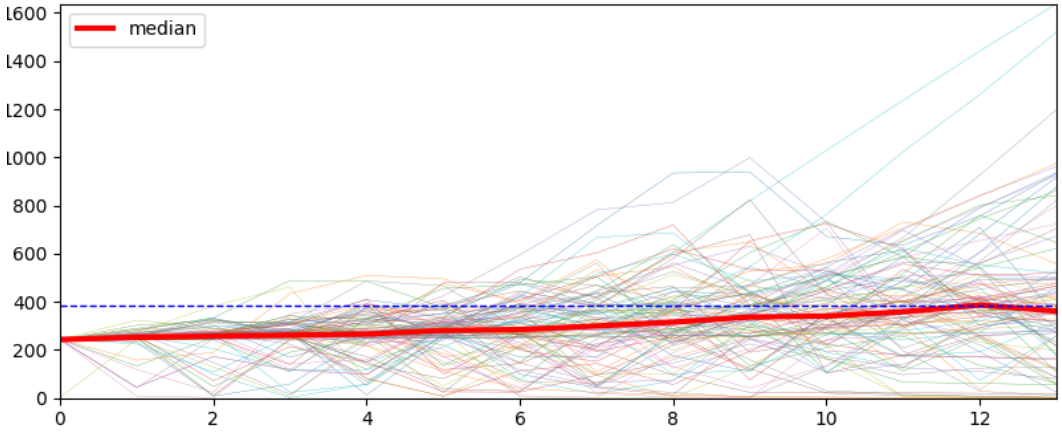


(c) HEUR

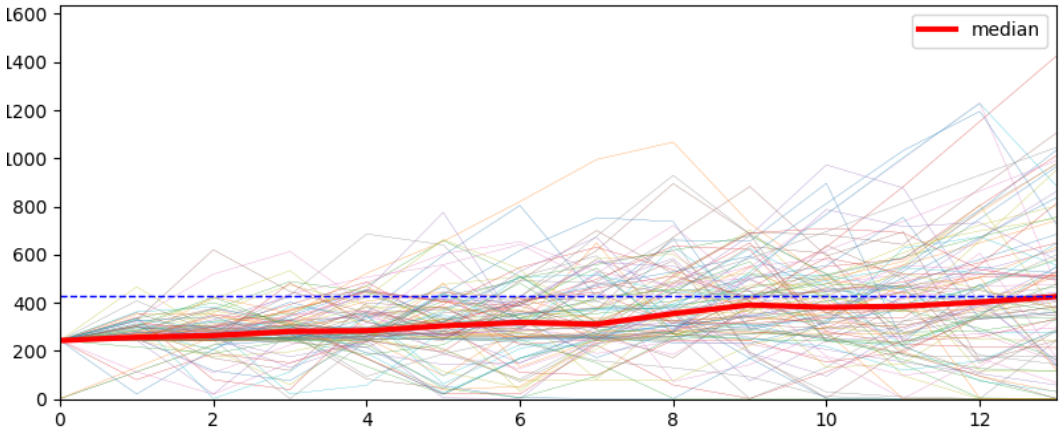
Figure 6.3: Average degree



(a) VANILLA

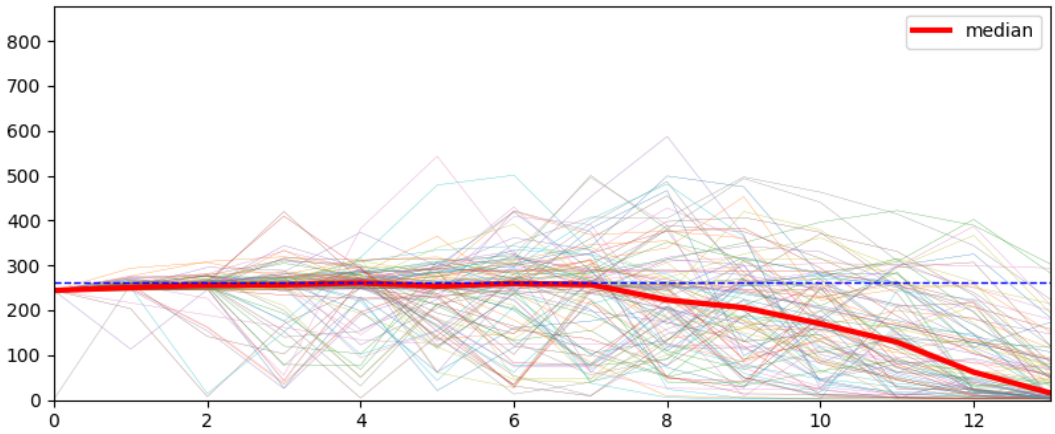


(b) EXPL_L

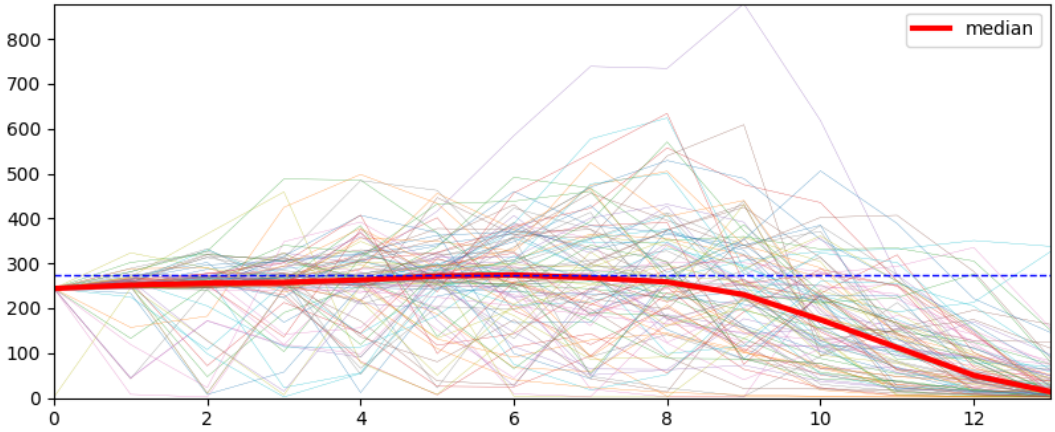


(c) HEUR

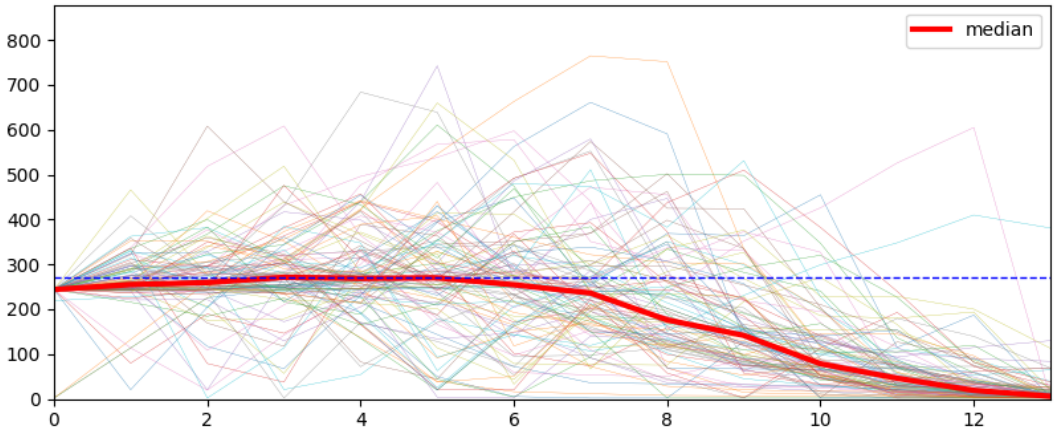
Figure 6.4: Root visits



(a) VANILLA



(b) EXPL_L



(c) HEUR

Figure 6.5: Tree size

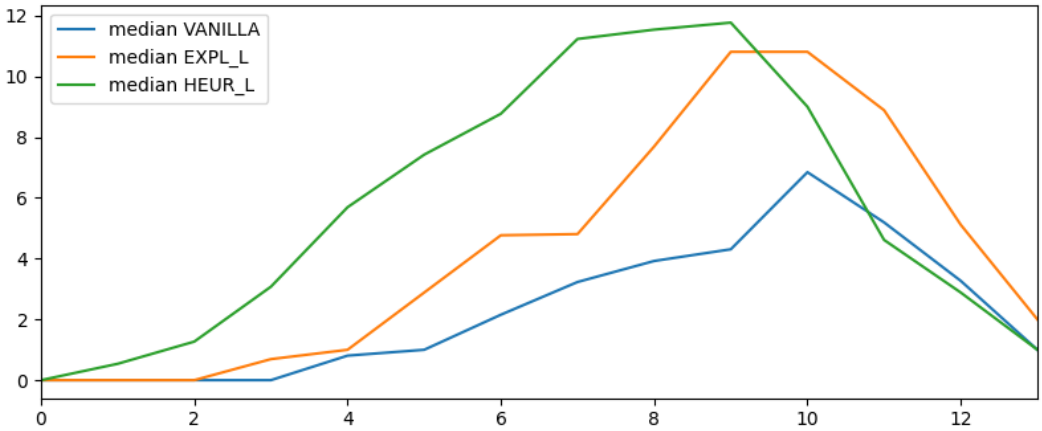


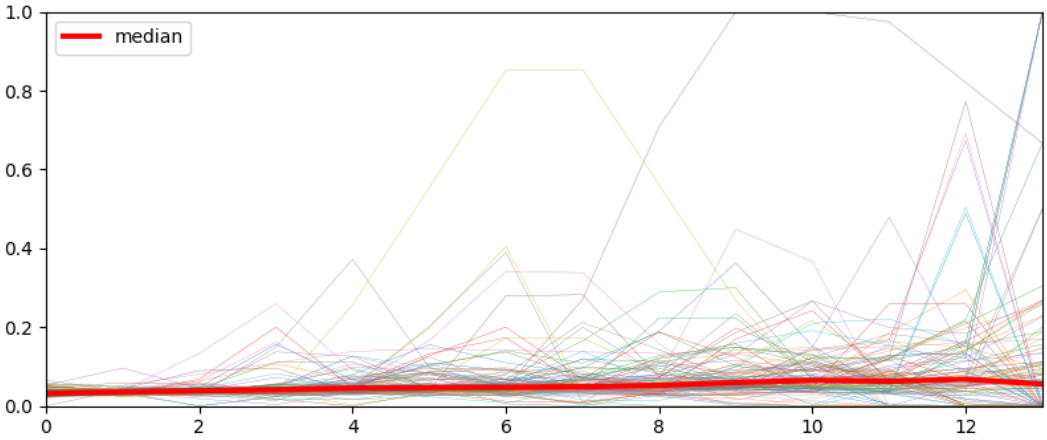
Figure 6.6: Number of terminal leaves

number of terminal leaves peaks. Even though the number of terminal leaves does not keep increasing as the size decreases, we still think that our hypothesis - that the size decreases as the match reaches the last few moves because the game tree is smaller - holds, but it also affects the number of terminal leaves

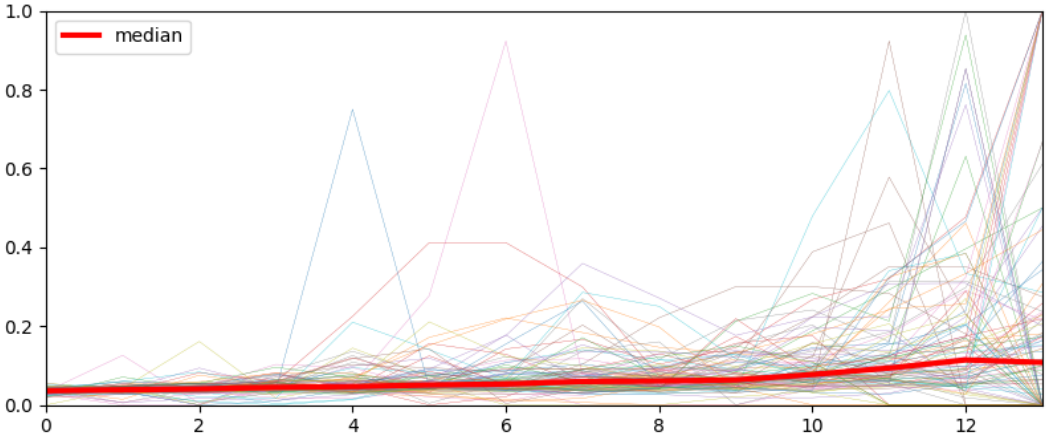
The MC tree is usually very well balanced, as can be seen in Figure 6.7 and Figure 6.8 which record Sackin and Cophenetic indices. Both indices are normalised, so their values range from 0 to 1, where 0 means a perfectly balanced tree and 1 means a completely imbalanced one. We decided to plot only medians for the Cophenetic index, as opposed to plotting all matches for the Sackin index, because the values are mostly very small. For both indices the imbalance increases towards the end of the match. However, it is interesting to note that the imbalance of HEUR actually decreases in the last move. We can also observe that HEUR (and to some extent EXPL) has a higher imbalance throughout the match until the final dip according to the Sackin index. But if we zoom in on the first part of the plot of Cophenetic index, this behaviour is not replicated . The final observation is that the variance for both indices increases towards the end. We conclude that it is not obvious whether the move-pruning heuristic increases or decreases the balance of the MC tree and that the tree is relatively well balanced.

The last tree metric that we monitored is the win expectancy, which can be seen in Figure 6.9a. We can see that the win expectancy starts at around 0.5 and then has either an upward trajectory, suggesting that the algorithm is winning, or a downward trajectory, suggesting that the algorithm is losing. To highlight this behaviour, we also present the plot 6.9b where we have replaced values below 10 percentile or above 90 with their interpolation.

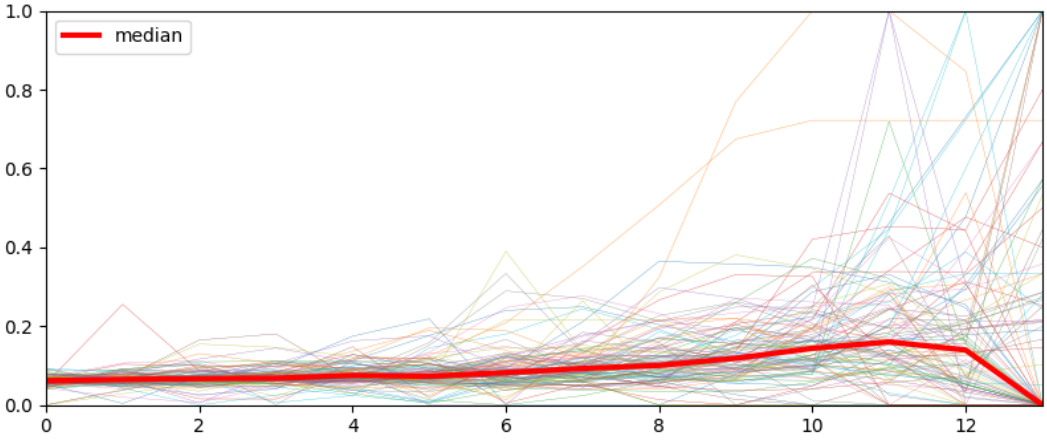
But what we really wanted to analyse, and unfortunately it is not quite clear from the plots shown, is whether the move-pruning heuristic causes surprising or unexpected losses - losses where the algorithm has a high win expectation but ends up losing. We



(a) VANILLA

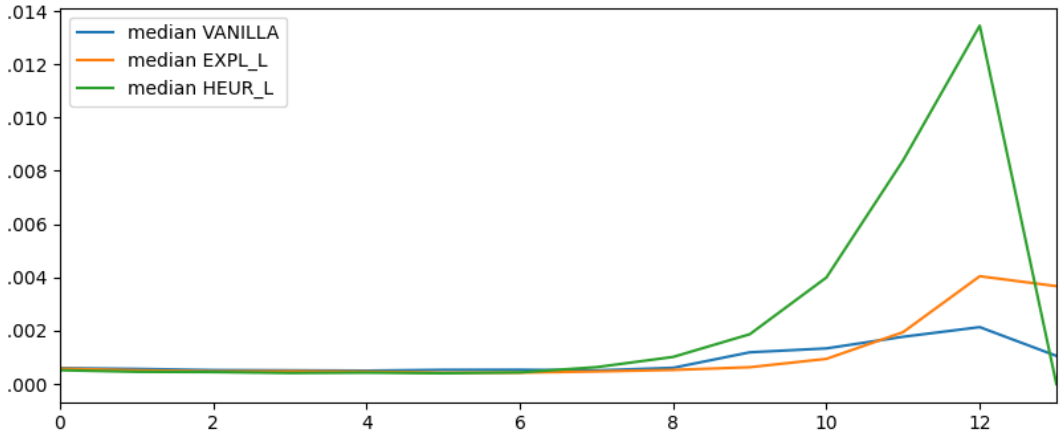


(b) EXPL_L

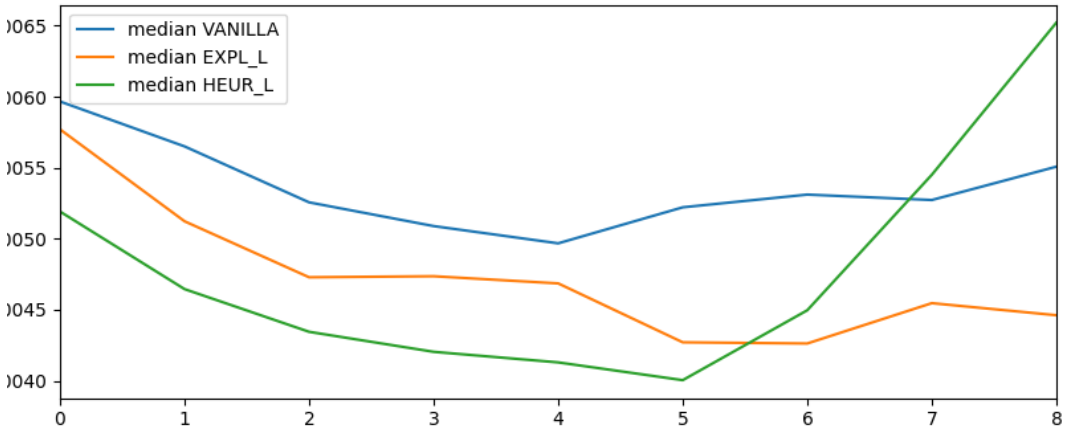


(c) HEUR

Figure 6.7: Sackin index



(a) Entire length



(b) First two halves

Figure 6.8: Cophenetic index

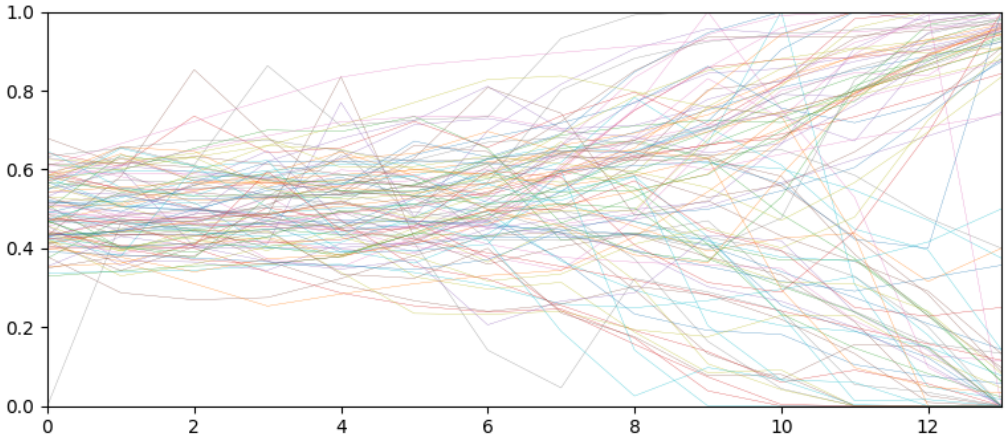
		1/2		2/3		3/4	
name	losses	surprising	ratio	surprising	ratio	surprising	ratio
VANILLA	50	1	2%	3	6%	4	8%
EXPL	31	1	3.23%	0	-	2	6.45%
HEUR	29	4	13.79%	8	27.59%	5	17.24%

Table 6.1: Comparison of surprising losses

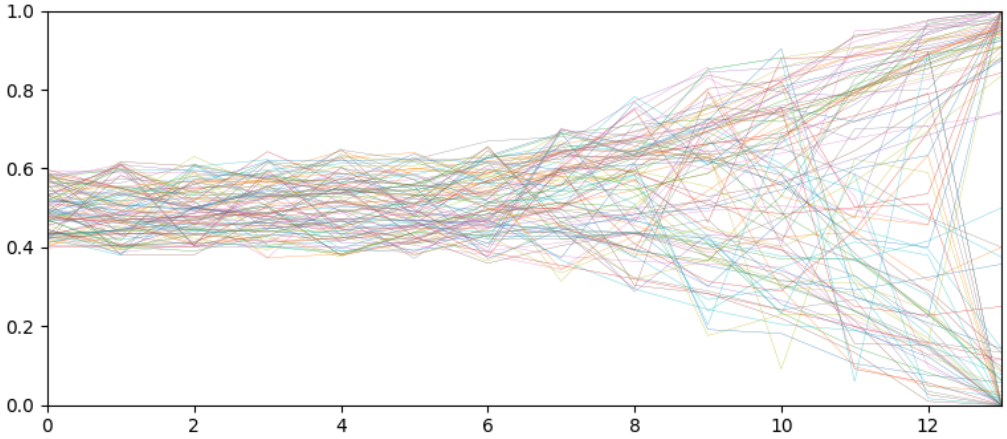
therefore analysed how many matches had a high win expectancy (above 0.8), but the algorithm ended up losing. We measured the win expectancy at half, two-thirds and three-quarters of the match. To be able to compare the variants with different numbers of wins and losses, we divided the number of surprising losses by all losses and recorded the data in table 6.1. It is clearly visible that the HEUR variant has the highest proportion of surprising losses, especially when we measure the win expectancy in two thirds of the match. It is as high as 27.59% of all losses. On the other hand, when we measure in two thirds and three quarters, EXPL has the lowest number of surprising losses and it has only a slightly higher ratio than VANILLA when we measure in half the match.

We also counted the number of surprising wins, but they are rather rare - no more than one in 100 matches - so we do not present them in any table.

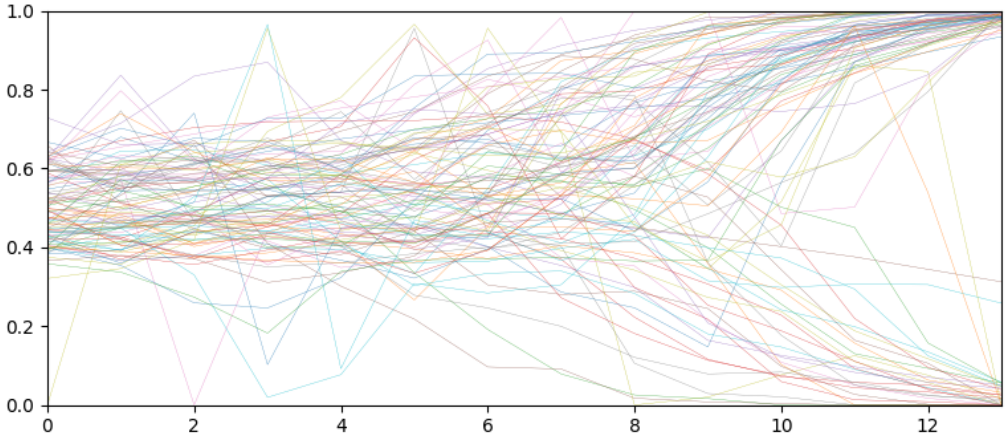
We conclude that a move-pruning heuristic can increase the number of surprising losses. This is the case in our example. However, in the case of the smallest_tuple heuristic in the Pharaoh game, the total number of surprising losses is still low and therefore worth using.



(a) VANILLA



(b) VANILLA - filtered data



(c) HEUR

Figure 6.9: Win expectancy

Conclusions and future work

This work focused on exploring and analysing the application of the Monte Carlo Tree Search (MCTS) algorithm in the game Pharaoh.

We started by including a formal definition of a card game, followed by a specific definition of the Pharaoh game. By establishing a clear understanding of the characteristics and rules of card games in general, we were able to delve into the unique aspects and mechanics of the Pharaoh game.

The formal definition of a card game of Pharaoh served as a foundation which ensured a clear understanding of the game's mechanics. This definition encompassed essential elements such as the deck of cards, rules for dealing and playing cards, win conditions, and any specific mechanics or variations relevant to card games.

Afterwards we provided a description of the MCTS (Monte Carlo Tree Search) algorithm and its variants, namely heuristic move-pruning and beam search.

The MCTS algorithm, known for its ability to make informed decisions in uncertain and complex environments, was thoroughly explained in this work. Its underlying principles, such as selection, expansion, simulation, and backpropagation, were elaborated upon to provide a clear understanding of its mechanics and functionality. Additionally, the advantages and limitations of the MCTS algorithm were discussed, highlighting its effectiveness in situations where extensive exploration and exploitation are required.

Furthermore, two specific variants of the MCTS algorithm were explored: heuristic move-pruning and beam search. Heuristic move-pruning involves the use of domain-specific heuristics to guide the tree traversal process, aiming to prioritize more promising moves and reduce computational overhead. On the other hand, beam search focuses on restricting the number of considered paths during simulation, allowing for a more focused exploration and potentially improving the efficiency of the MCTS algorithm.

Furthermore, an important aspect of this work was the optimization of the algorithm's parameters. The performance of the MCTS algorithm and its variants heavily relies on carefully selecting and fine-tuning these parameters to achieve optimal results.

Through systematic experimentation and analysis, various parameter configurations were tested and evaluated. Metrics such as win rates or computational efficiency were

considered to assess the effectiveness of different parameter settings. By conducting rigorous experiments and analysing the obtained results, the optimal parameter values for the MCTS algorithm and its variants in the context of the game of Pharaoh were determined.

Throughout the research, we conducted a tournament in which different variants of the MCTS algorithm, as well as other algorithms (for example, backtracking or simple heuristic players inspired by strategies commonly used by human players but not studied in other research), played the game Pharaoh. The objective was to compare the results to assess the performance of these algorithms in the gaming environment. Additionally, a significant aspect of this work involved analyzing the development of the tree constructed by the MCTS algorithm.

Based on the findings, we observed that the MCTS algorithm demonstrates significant potential in solving games like Pharaoh and achieves superior results compared to other algorithms. The research provided valuable insights into the application of MCTS in the game Pharaoh, opening up new perspectives for further development in this field.

Throughout the project, we utilized various techniques and tools, including data processing, visualization, and analysis. The use of Python libraries such as NumPy, Pandas, and Matplotlib facilitated the efficient handling and visualization of the data.

There are a number of areas that need to be explored further. First, there is a need for a deeper understanding of why our experiments with Beam Search have not yielded significant improvements. An examination of the specific challenges and limitations of beam search in the context of the game Pharaoh would be a valuable source of insight.

Secondly, it would be beneficial to explore the use of comprehensive backtracking techniques to determine the theoretical maximum win rate achievable in the game. This analysis could serve as a benchmark for evaluating the performance of the implemented algorithms.

Thirdly, extending the study to evaluate the performance of the algorithm in multiplayer games would be an exciting direction for future research. Investigating the dynamics, strategies and potential challenges that arise in games with more than two players could provide valuable insights. Furthermore, broadening the scope of the study by implementing and evaluating the algorithm in other card games would provide a broader perspective on its effectiveness. We have considered games such as President, Sedma or Pharaoh with some modifications of the rules, but have not explored them for various reasons (e.g. because there are many variants of the rules, some of them are difficult to implement, many are not suitable for analysis due to limited computational resources, because backtracking is too slow, etc.). Comparing its performance in different game domains would provide valuable insights into the versatility and gen-

eralisability of the algorithm.

By addressing these areas in future work, we can improve our understanding of the algorithm's performance, explore its applicability in different game scenarios, and pave the way for advances in the field of game playing algorithms.

Bibliography

[1] Introduction to Monte Carlo Tree Search: The Game-Changing Algorithm behind DeepMind's AlphaGo. `https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/`. Accessed: 2023-03-13. 4.4

[2] Monte Carlo Tree Search for tic-tac-toe game in java. `https://www.baeldung.com/java-monte-carlo-tree-search`. Accessed: 2023-03-13. 4.4

[3] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, May 2002. 2.2

[4] Aijun Bai, Feng Wu, and Xiaoping Chen. Bayesian mixture modelling and inference based thompson sampling in monte-carlo tree search. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. 2.2

[5] Hendrik Baier and Mark Winands. Beam monte-carlo tree search. pages 227–233, 09 2012. 2.3

[6] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Steve 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. 5.2

[7] Mareike Fischer, Lina Herbst, Sophie Kersting, Luise Kühn, and Kristina Wicke. Tree balance indices: a comprehensive survey, 2021. 6.1

[8] Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, Canada, December 2006. (document), 2.2

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. 5.2

[10] Akihiro Kishimoto and Jonathan Schaeffer. Distributed game-tree search using transposition table driven work scheduling. pages 323–330, 05 2002. 2.3

[11] Kocsis L and Szepesvári C. Bandit based monte carlo planning. In *Proceedings of the 17th European conference on machine learning*, page 282–293. ECML'06. Springer, Berlin, 2006. 2.2

[12] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998. 5.2

[13] Sephton N, Cowling PI, Powley E, and Slaven NH. Heuristic move pruning in monte carlo tree search for the strategic card game lords of war. In *IEEE conference on computational intelligence and games*, pages 1–7, 2014. 2.3

[14] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016. 5.2

[15] David Silver, Richard S. Sutton, and Martin Müller. Temporal-difference search in computer go. *Machine Learning*, 87(2):183–219, May 2012. (document)