Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

The Surprising Effectivity of Monte Carlo Tree Search

Josef Vonášek

Supervisor: RNDr. Vojtěch Kovařík, PhD
Field of study: Open Informatics
Subfield: Artificial Intelligence
January 2021

Poděkování

I would like to thank everyone that gave me the support in order to complete the thesis. I want to especially thank my supervisor Vojtěch Kovařík, as his guidence, expertise and patience were extremely important for me and this work. Finally, my family also deserves many thanks for their endless support thorough the studies.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 1. ledna 2022

Abstract

The worst-case performance of the Monte Carlo tree search (MCTS) algorithm is orders of magnitude worse than that of naive brute-force methods and not many domain-specific bounds known for it. Nonetheless, its practical performance is outstanding, leading to its widespread adoption in game solvers. As a result of this gap in our understanding, state-of-the-art algorithms such as AlphaZero generally do not (yet) have meaningful performance guarantees. To partially address this gap, we investigate the practical performance of MCTS in Tic Tac Toe, Hex and in additional artificial games, and demonstrate its fast convergence to optimal policy. Additionally, we show that when combined with Alpha-Beta pruning, MCTS outperforms the traditional Alpha-Beta pruning minimax algorithm even in offline mode. We show the uniform policy value to be the major culprit behind the observed performance, demonstrating that a) the value estimates are most often correctly ordered, b) incorrectly ordered values reduce performance noticeably. We propose *advantage accumulation* as the driving force behind the high quality of uniform policy estimates. We show in artificial games that higher advantage accumulation improves the UCT performance and describe some of the properties found in Tic Tac Toe and Hex with positive effect on said accumulation.

Keywords: Game theory, Monte Carlo tree search, UCT

Supervisor: RNDr. Vojtěch Kovařík, PhD

Abstrakt

Efektivita algoritmu prohledávání stromu Monte Carlo (MCTS) je v nejhorším případě řádově horší, než u naivních metod hrubé síly, přičemž je známo jen málo zárukách pro specifické domény. Přesto je jeho praktická efektivita vynikající, což vedlo k jeho širokému uplatnění v teorii her. V důsledku této mezery v našich znalostech nemají (zatím) nejmodernější algoritmy, jako je AlphaZero, smysluplné výkonnostní záruky. Tato práce proto zkoumá praktickou efektivitu tohoto algoritmu v hrách Tic Tac Toe, Hex a v dalších umělých hrách a prokazujeme jeho rychlou konvergenci k optimální strategii. Navíc ukazuje, že UCT v kombinaci s Alpha-Beta překonává tradiční Alpha-Beta algoritmus i v offline režimu. Zkoumáme toto chování a zjišťujeme, že hlavním viníkem pozorované efektivity je dobrá hodnota uniformní strategie, přičemž se ukazuje, že a) odhady hodnot jsou nejčastěji správně uspořádané, b) nesprávně uspořádané hodnoty znatelně snižují efektivitu. Zdůvodňujeme, proč tomu tak může být, a navrhujeme *akumulaci výhody* jako hnací sílu stojící za vynikajícími hodnotami uniformní strategie. Na umělé hře ukazujeme, že vyšší akumulace výhody zlepšuje výkon UCT, a popisujeme některé vlastnosti zjištěné v hrách Tic Tac Toe a Hex s pozitivním vlivem na tuto akumulaci.

Klíčová slova: Teorie her, Monte Carlo tree search, UCT

Překlad názvu: Překvapivá Efektivita Metody Monte Carlo Tree Search

Contents

Figures

Tables

Chapter 1

Introduction

In 1996 the IBM chess computer named Deep Blue became historically the first artificial player to defeat the world chess champion – Garry Kasparov – under standard tournament rules [Campbell et al., 2002]. It relied on extensive domain knowledge, a large Grandmaster game database, and a massively parallel alpha-beta tree search strategy. It was a big success for artificial intelligence as chess was historically one of the most popular yet highly complex board games.

Nevertheless, it took another 20 years of engineering to create an artificial player that would beat the world champion of another widely popular, ancient board game – Go [Silver et al., 2017]. One of the reasons was the enormous state space of Go. While a typical professional chess match counts around 80 moves, a typical Go match can count up to 150 successive moves. Moreover, the average branching factor (number of legal actions) is much higher in Go – approximately 250 compared to 35 in chess.

Nevertheless, another reason made the algorithm from Deep Blue unsuitable for Go. The performance of the Alpha-Beta algorithm relies heavily on a good evaluation function, which was not available for Go. Then, the idea to use random play (i.e., Monte Carlo simulation) as an evaluation function emerged. This technique got later extended into a brand new tree search technique – Monte Carlo tree search [Ginsberg, 2002]. It quickly became the basis of the most successful Go algorithms – Crazy Stone, MoGo [Gelly and Silver, 2008], and, ultimately, AlphaGo Silver et al. [2017]. In the end, it was shown to work across a wide variety of games (including chess) by the algorithm AlphaZero [Silver et al., 2018].

However, there is one issue with the commonly used variant of Monte Carlo tree search – UCT (upper confidence bound for trees) [Kocsis and Szepesvári, 2006]. Although it converges to optimal play in the limit, the bounds on its performance are rather weak [Audibert et al., 2009]. Despite that, the algorithm performs surprisingly well in practice, as shown by the success of its applications.

Investigating this surprising phenomenon is the primary motivation behind this thesis. The original idea behind UCT was to use the UCB1 algorithm [Kocsis and Szepesvári, 2006] – initially designed for a Multi-Armed Bandit problem (MAB) – in a Monte Carlo tree search to carefully balance between exploitation of best-known moves and exploration of the less visited subtrees. However, Multi-Armed Bandit problem is conceptually much simpler than a typical game, where the UCB1 algorithm assumptions become violated. So even though increasing the UCB1 computation budget in classical MAB settings leads to more precise solution approximation, there is no similar guarantee when it becomes recursively applied to arbitrary game trees – as is done in UCT.

In practice, however, the UCT algorithm performs exceptionally well. This perhaps points to specific characteristics shared among the games humans seem to enjoy. This thesis aims to explore the existence of such characteristics, describe them formally, and finally, measure their prevalence in various games and their effect on the performance of MCTS.

First, we confirm the UCT performance in the games Tic Tac Toe and Hex, showing that UCT is able to solve the end states of each game under 50 simulations. On top of that, UCT combined with the Alpha-Beta algorithm is also shown to consistently outperform the default Alpha-Beta algorithm. Finally, it is shown that turning off the exploration factor does not have significant effect on the performance, suggesting that the initial state values (used by the UCT algorithm) obtained by the uniformly random policy are mostly accurate.

We then show on synthetic benchmarks that the initial estimates indeed play a huge role in the UCT algorithm performance. Our explanation for this phenomenon is some form of accumulation of advantage present in each game, as advantage accumulation would lead to similar rewards between sibling moves - a property that we suspect to improve the MCTS performance. We go on to show on synthetic benchmarks that this assumption indeed holds. Finally, we provide a description of some of the properties common to both Hex and Tic Tac Toe that would support this conclusion.

Thesis Overview

The content of the thesis is following.

- Section 2 presents the MCTS problematic in more detail.
- Section 3 reviews existing work on the performance of Monte Carlo tree search.
- Section 4 describes the rules of the traditional games used in this thesis.
- Section 5 presents the technical background behind the algorithms used in this work.
- Section 6 explores the effectivity of the UCT, Alpha-Beta and Alpha-Beta UCT on traditional games benchmarks.
- Section 7 investigates the properties of traditional and artificial games that make UCT more effective.
- Section 8 draws the final conclusion and suggests further research directions.

Chapter 2

Problem Statement

Although the UCT algorithm is guaranteed to eventually find an optimal policy in any perfect information game, it might take extremely long to discover it, as the theoretical bounds of the UCT algorithm are very weak [Coquelin and Munos, 2007]. Consider a simple one-player game with a single optimal outcome at the maximum depth, where stepping of the optimal path gives progressively lower rewards the further from the root the player is. In such a case, the UCT algorithm is going to spend most of its exploration budget on visiting the nodes with the highest known reward, which are located at the top of the tree.

An example of such a game tree is presented in Figure 2.1. Here, the non-optimal actions at a depth d give a reward equal to $\frac{D-d}{D}$ where D is the total depth of the tree. Such game is going to require more than $\Omega(\exp(\exp(..\exp(2)..)))$ (where the exp is repeated $D-1$ times) simulations in order to reach the terminal node with highest reward as shown by Coquelin and Munos [2007]. They also show the bound is improved to $\Omega(\exp(\exp(D))$, when the usual logarithmic confidence bound gets replaced by a square root. However, while this bound is accurate for adversarially designed games such as the one presented in Figure 2.1, it appears to be overly pessimistic on practically relevant games such as chess or Go as suggested by the success of AlphaGo [Silver et al., 2017] and AlphaZero [Silver et al., 2018].

One of the main reasons behind this phenomena is most likely the fact that many real-world games possess, quite a different structure than the worst-case example we have shown in Figure 2.1. First, the rewards are usually discrete coming from the set $\{1, 0, -1\}$, as for most games we only have 3 different

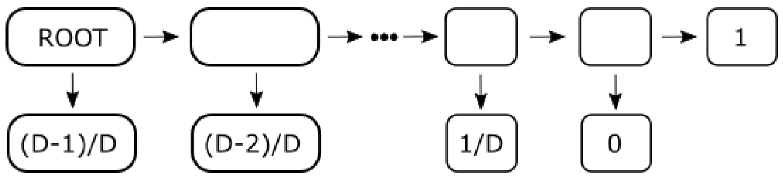

Figure 2.1: An example of difficult to solve 1-player game tree. The action values on the path to the optimal reward decrease linearly with the formula $\frac{D-n}{D}$. This makes UCT focus its exploration at the top of the tree instead of the bottom. Requiring a composition of $D-1$ exponential function $\Omega(\exp(\exp(..\exp(2)..)))$ simulations in order to get to the optimal reward.

outcomes: victory, loss and a draw. This makes it harder to craft adversarial games such as the one presented in Figure 2.1, as it relied on fine grained real-valued rewards. The value of node can still appear to the UCT to have a non-discrete value, as the real values are approximated by uniform policy. However, for the value to stay the same for long enough time (i.e., no converge to the real reward), the tree must be sufficiently deep and/or wide. Second, the games usually include two players, which could have a positive effect on the approximation of the true reward of a given move. Finally, in most games, players try to build up some sort of advantageous position over the course of the game. This could have a significant effect on the smoothness of the reward distributions in the tree, which should, in general, improve the performance of the MCTS algorithm.

Our goal is to explore and measure the prevalence of these phenomena on some of the popular 2-player, deterministic, perfect information games.

Chapter 3

Related Work

3.1 Multi Armed Bandit Problem And Its Generalization To Trees

Monte Carlo methods were traditionally used in statistics for numerical approximations of otherwise intractable problems [Metropolis and Ulam, 1949]. The idea is to use cheap, repeated random sampling to build an approximately correct solution. Monte Carlo tree search is used to approximate the optimal strategy of games where other methods cannot find an optimal solution in reasonable amount of time – usually because the game state space is too large. One of the first application of the Monte Carlo methods to game theory was done by Abramson [1990] who combined linear regression with random sampling to build an efficient and model-independent expected outcome estimator. Additionally, the world's strongest, at that time, artificial players of Scrabble and Bridge (games of imperfect information) also combined game tree search strategy with a Monte Carlo uniform action selection mechanism [Ginsberg, 2002].

Kocsis and Szepesvári [2006] introduced a significant improvement to the vanilla Monte Carlo planning algorithm, which up to that point either sampled actions uniformly or used a domain-specific heuristic with little to no convergence guarantees. They correctly assessed that to improve the performance of Monte Carlo, the estimation error of state-values must decay fast. Therefore, an efficient algorithm must balance its search budget optimally between the best-looking actions and their seemingly suboptimal alternatives.

This is a well-studied problem of the trade-off between exploration and exploitation, and its simplest form shows up in the multi-armed bandit problem (MAB). Their idea was to apply the UCB1 (Upper Confidence Bounds) algorithm from MAB to tree search. They named the algorithm UCT (Upper Confidence Bounds for Trees) and showed that the probability of selecting the optimal action converges to 100% as the number of samples approaches infinity.

3.2 UCT Regret Bounds

However, Audibert et al. [2009] showed that the convergence guarantees that hold for MAB do not straightforwardly generalize to problems with several nested sequences of bandits. They studied an abstract class of UCB-type algorithms that use variance estimates during action selection. It has been shown by Auer et al. [2002] that the expected regret of UCB1 decreases logarithmically in N, where N is the number of samples. Audibert et al. [2009] confirmed and generalized this result. However, they also demonstrated that the *cumulative* expected regret (which appears in anytime tree algorithms based on MAB) could decrease at most polynomially in N. Finally, they also constructed a variant of UCB for a given finite time horizon N, named PAC-UCB. Since the time horizon is known in advance, the algorithm achieves a logarithmic bound on the *cumulative* expected regret and is, therefore, more efficient than a traditional anytime UCB variant.

The work of Coquelin and Munos [2007] introduces an improvement on the worst-case regret bounds of UCT. They show that the bounds are e^{e^D} in an unbalanced tree with decreasing rewards where the maximum reward happens to be at maximum depth D. However, a modified UCT with knowledge of the depth of the tree can improve on the bounds significantly – they designed an algorithm with regret bounds of $2^D/\sqrt{N}$ where N was the number of samples. In order to take a local smoothness into account (reward similarity between neighboring states), they inspected the performance of flat-UCB that used UCB only in tree leaves and found its regret to be very similar to the modified UCB. This allowed them to create the BAST (bandit algorithm for smooth trees) algorithm that is able exploit smooth reward distributions.

3.3 Game Pathologies

A game pathology occurs when a deeper search in the game tree results in a less accurate prediction of the value of a given state [Nau et al., 2010]. Although it is possible to design a pathological game artificially, it has been of great interest how this problem will manifest in real human games. Nau et al. [2010] suggests that it is more common than thought. He found that the African game of Kalah with sufficient branching factor contains pathologies at almost every granularity. However, even other, mostly non-pathological games exhibited some pathologies. For example, in the 8-puzzle, approximately 20% board configurations happened to be pathological, while in the chess championship matches, pathologies accounted for up to 10% board configurations. Wilson et al. [2009] showed that a majority of interesting zero-sum game contain local pathologies. They then designed a more robust minimax variant named EMM (error minimizing minimax) by tracking the error associated with the heuristic at each level. They experimentally demonstrated that EMM outperforms minimax in most situations.

Long et al. [2010] measured the effect of elementary game properties in imperfect information games on the performance of Perfect Information Monte Carlo (PIMC) tree search methods. The use of Monte Carlo methods has been criticized by Frank and Basin [1998] who showed that it is prone to two types of errors – strategy fusion and non-locality. Both errors come from assumptions that hold in perfect information but not in imperfect information settings. However, there was no straightforward way to quantify these errors. They, therefore, crafted three types of game properties – leaf correlation, bias, and disambiguation factor. They then showed how to measure these properties and experimentally confirmed that they were a good indicator of the game difficulty.

3.4 Application To Traditional Games

Gelly and Silver [2008] attempted to improve the standard UCT algorithm by combining it with additional value functions. First, they used a linear combination of binary features as a simple state-value heuristic learned by self-play, each weight corresponding to the influence of the feature on the expected value. This heuristic was used as the default policy in each new state. Second, they used a rapid action-value estimation to reduce the high variance for a few samples. This was done by using any sequence containing a given action to approximate the action value. Finally, they used prior knowledge to initialize node values in the UCT tree. They combined all of these techniques in the (at the time) world strongest 9x9 Go program MoGo, significantly improving its strength.

Recently, a new approach appeared in the literature that combined deep neural networks with the Monte Carlo algorithm. The main idea was to get a better approximation of good moves by using a neural network to evaluate board positions and improve action selection. Silver et al. [2017] demonstrated that such approach is capable of super-human play in the game of 19x19 Go despite its enormous branching factor and game length – 250 and 150, respectively.

Their program AlphaGo used recorded human professional games combined with self-play to train deep convolutional networks as a heuristic function (called policy and value networks), obtaining estimates on par with state-of-the-art Monte Carlo tree search algorithms. In combination with MCTS, they managed to defeat the human European Go champion Fan Hui and the Korean champion Lee Seldol. Their follow-up work improved upon the idea by not depending on any domain knowledge, mastering the game entirely by reinforcement learning from self-play. The algorithm has also been simplified. Firstly, by using a single neural network instead of two separate policy and value networks. Secondly, by replacing the Monte Carlo rollouts with the value estimates obtained by the above-mentioned deep neural network. The final program named AlphaGoZero defeated AlphaGo 100 to 0 and required significantly less time to train.

Chapter 4

Games

This section describes the setup of traditional zero-sum, deterministic games used in this work – Tic Tac Toe and Hex. We use them to study the practical UCT performance. These games are provided in the collection of games of Deepmind's open-source library Openspiel Lanctot et al. [2019].

Next, we will describe the rules and setup of the chosen board games. Each one has a relatively simple set of rules and a useful property that makes the number of moves decrease by one after each turn. This is important as it makes the computational analysis of end games much more efficient. Despite this similarity, the games have wildly different levels of complexity. Tic Tac Toe is the simplest as it allows for no more than 9! unique move sequences. On the other hand, a typical Hex match on an 11x11 board is much more complex, as the upper bound on possible unique play-throughs reaches 121!. As a consequence, it is computationally intractable to run Monte Carlo tree search on the whole game tree, which limits us to either reducing the size of the board or carrying out the analysis only on a subset of end games. The rules of each game will be described in more detail in the following sections.

4.0.1 Tic Tac Toe

Tic Ta Toe, or Noughts and Crosses, is a zero-sum paper-and-pen game played on a 3x3 grid. Each player takes turns, marking one of the fields with X or O, respectively. The first player that makes 3 of their symbols align horizontally, vertically or diagonally, wins the game. The game contains only

765 unique positions after accounting for rotation and reflection. Due to its simplicity, it is mostly used for teaching purposes. However, the game can be generalized to any grid size and even dimensionality, which increases the difficulty considerably.

We will stick with the original definition. In such a case, it is known that the game will always end up in a draw if played perfectly . However, the legal outcomes consist of 91, 44, and 3 distinct winning positions for the X, O, and neither player, respectively, assuming that X is the starting player and that a draw counts as a win for no player. Such imbalance is of our interest, as it affects the quality of the common random-rollout MCTS heuristic.

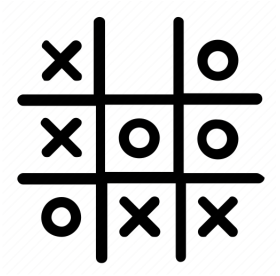

Figure 4.1: An example of Tic Tac Toe game, with the second player (circle) winning by connecting 3 circles diagonally.

4.0.2 Hex

Hex is a positional, perfect information board game invented in the 20th century by the mathematician Piet Hein. It is typically played on 11x11, 13x13, and 19x19 hexagonal boards. Each player takes a turn, placing stones of his color on empty spaces, trying to connect opposing sides of the board. The first one to do so wins. In order to win, players focus on building short, safely connected chains of stones where two unconnected chains are said to be safely connected, if and only if the opposing player cannot prevent their connection by any sequence of moves.

Hex itself has several notable properties. First, it has been shown that the beginning player can never lose (similar to Tic Tac Toe). The proof relies on the standard strategy-stealing argument discovered by John Nash. In short, if there would be a winning strategy for the second player for any board setup, the first player could place his first stone randomly and then use the opponent strategy against him. As both players have the same set of moves and because extra stone of player's color is never a disadvantage, this will inevitably lead to a contradiction, as both players should win. This is, of course, impossible, and therefore such a second-player winning strategy

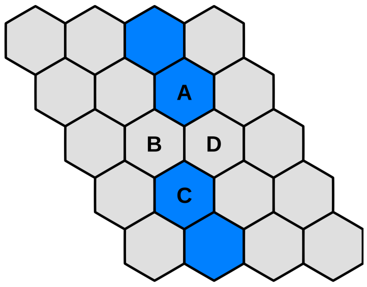

Figure 4.2: An example of Hex **bridge**. The blue player can always connect stones A and B, no matter what opponent does. Such setup is called to be safely connected and serves as the basic strategic element.

cannot exist.

The second property, called the Hex theorem, states that a draw is impossible. This too was discovered by John Nash in the 1950s, and it has been shown to be equivalent to the Brouwer fixed-point theorem. As a consequence, the first player will always win if playing perfectly. Despite that, a perfect strategy has never been shown or computed for boards larger than 9x9, and it is generally very difficult to compute such strategy for boards of arbitrary size, as the problem has been shown to be PSPACE-complete.

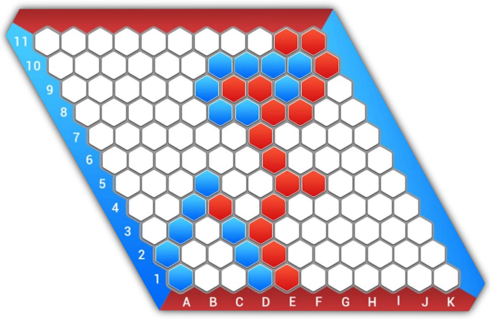

Figure 4.3: An example of 11x11 Hex game, with the red player winning by connecting top and bottom edges of the game board

Chapter 5

Technical Background

In this chapter, we provide a formal description of the types of games we are going to study and the search techniques being used to solve them. We focus on deterministic, perfect information games as they are conceptually simple yet give rise to complex games (like chess, Go or Hex) challenging to both human and artificial players.

5.1 Extensive Form Games

Deterministic, perfect information, zero-sum, extensive-form games (EFGs) can be informally described as trees with actions representing edges, inner nodes representing game states where one of the players takes action, and leaf nodes representing terminal states with a payoff for every player.

Formally, they are defined as a tuple $(P, \mathcal{P}, A, \mathcal{A}, H, Z, h_R, \rho, r)$ where

- $P = \{P_1, P_2\}$ is a set of players
- A is a finite, non-empty set of actions
- H is a finite, non-empty set of choice nodes
- $h_R \in H$ is the root node

- Z is a finite, non-empty set of terminal nodes
- $\mathcal{P} : H \to P$ defines which player gets to act at given node
- $\mathcal{A} : H \to 2^A$ defines available function at given node
- $\rho : (h \in H) \times \mathcal{A}(h) \to H \cup Z$ is a successor function where
  - $\rho(h, a) = \rho(h', a') \implies h = h' \,\&\, a = a'$
- $r : Z \to \mathbb{R}^2$ is the reward function where
  - the rewards sum to zero: $r(h)_1 = -r(h)_2$

Additionally, for the analysis we carry out in this work, we need to be able to describe arbitrary player strategies, expected utilities and optimal policies. These are formally defined as:

- $u(\pi_1, \pi_2) \in \mathbb{R}^2$ is the expected utility of players policies
- $\pi : (h \in H) \times \mathcal{A}(h) \to \Theta(\mathcal{A}(h))$ is a player's policy function where
  - $\Theta(X)$ is the set of all probability distributions over X

We say that a policy π is a **Nash equilibrium** if neither of the players can improve their utility by unilaterally deviating from π, that is, if the following holds for both $i \in \{1, 2\}$ (where $-i$ denotes the opponent of i):

$$\forall \pi_i' \ : \ u_i(\pi_i, \pi_{-i}) \geq u_i(\pi_i', \pi_{-i}). \tag{5.1}$$

We typically denote Nash equilibria as π^*.

The goal of each player is to find his Nash equilibrium policy π_i^*. We will therefore call it an *optimal* policy. When such policy is known for each player, the game is said to be *solved*. The traditional technique to find the optimal policy in zero-sum games is the Minimax algorithm and its extensions.

5.2 Minimax algorithm

The minimax algorithm is a recursive algorithm that finds the set of Nash equilibrium policies in a given deterministic, two-player, zero-sum game. It does so by recursively searching for the actions that give each player the maximum utility value across all actions available in a given game state assuming the opponent policy is chosen such that it minimizes the value of this maximum utility. The pseudocode of the algorithm for two player game is shown in the Algorithm 1.

However, obtaining the optimal policy by the Minimax algorithm is not computationally feasible for many relevant games. Therefore, the search is usually limited to a certain depth, and the utility values are replaced by a positional evaluation function, specifically designed for each game in advance. This function computes a heuristic value for choice nodes. Its value can represent an approximation of the theoretically optimal utility or an estimate of the probability to win.

Algorithm 1 The minimax algorithm

```
function minimax(node, depth)
  if depth == 0 or node is a terminal node
    return the heuristic value of node
  if isMaximizingPlayer(node)
    value = -infinity
    for each child of node
      value = max(value, minimax(child, depth - 1))
    return value
  else # minimizing player
    value = +infinity
    for each child of node
      value = min(value, minimax(child, depth - 1))
    return value
```

A common extension to the Minimax algorithm is the Alpha-Beta algorithm, which significantly speeds up its runtime. The minimax does so by skipping an evaluation of any node once its utility is guaranteed to be worse than its alternative. The α and β represent the minimum and maximum bounds on the utilities. The pseudocode of the algorithm for two player game is given in the Algorithm 2.

However, even with these additional extensions, the algorithm did not work very well in games like Go, where no good positional evaluation was available. The Monte Carlo tree search appeared as an excellent alternative, as it does not suffer from the same problem. Its most prominent variant called UCT finds its origins in the multi-armed bandit problem, which we now describe.

Algorithm 2 The Alpha-Beta Algorithm

```
function alphabeta(node, depth, a, b)
  if depth == 0 or node is a terminal node
    return the heuristic value of node
  if isMaximizingPlayer(node)
    value = -infinity
    for each child of node
      value = max(value, alphabeta(child, depth-1, a, b))
      if value >= b
        break # beta cutoff
      a = max(a, value)
      return value
  else
    value = +infinity
    for each child of node
      value = min(value, alphabeta(child, depth-1, a, b))
      if value <= a
        break # alpha cutoff
      b = min(b, value)
    return value
```

5.3 Multi-Armed Bandit Problem

The stochastic multi-armed bandit problem (MAB), sometimes called the N-armed bandit problem, is a classic reinforcement learning problem in which a fixed limited set of resources must be allocated between competing choices in a way that maximizes their expected gain. The stochastic outcome of each choice is not known upfront but may become better understood as time goes on.

Formally, the multi-armed bandit is defined as a tuple (A, T, r), where:

- A is a fine, non-empty set of actions
- $T \in \mathbb{N}$ is the number of turns
- $\mu : A \to \Theta$ is the utility function where
  - Θ is a set of real valued probability distributions

The game is played in the following way:

- On turn $t \in \{1, .., T\}$ the player chooses action $a_t \in A$

- The player receives a reward r_t independently sampled from $\mu(a_t)$
- The player may update their assumption on the reward distributions.
- The game moves on to the turn $t+1$.

The goal of the player is to maximize the reward accumulated over T turns: $\sum_{t=0}^{T} r_t$. The problem the player faces is how to effectively balance between playing the node with maximum known reward (exploitation) and playing other nodes to account for early bad luck caused by the stochastic nature of rewards (exploration). This problem is called the *exploration-exploitation dilemma*. The common techniques to deal for dealing with it are:

- ϵ-greedy strategy
- Thompson sampling
- Upper confidence bounds (UCB)

However, we will be interested only in UCB, as it is used in the UCT variant of the Monte Carlo tree search that we are about to study.

5.3.1 Upper Confidence Bounds

The upper confidence bounds algorithm solution to the exploration-exploitation dilemma is to estimate the error of each action's estimated expected reward. The next action is then selected based on the estimated reward and its possible error (upper confidence bound). Formally, the action a_t at the time t is selected so that:

$$a_t = \text{argmax}_a(Q_t(a) + U_t(a)). \tag{5.2}$$

Where

- $Q_t : A \to \mathbb{R}$ is the average of sampled rewards
    - $Q_t(a) = \frac{1}{t} \sum_{\substack{i=0 \\ a_i=a}}^{t} r_i$
- $U_t : A \to \mathbb{R}$ is the upper confidence bound
    - Defined in the Equation 5.3

A common way to estimate the upper confidence bound is by using Hoeffding's inequality Hoeffding [1994] - a theorem applicable to any bounded distribution. This variant of the algorithm is called UCB1. Let $C_t : A \to \mathbb{Z}$ be the amount of times the action a was selected up to the time t. The bound then becomes equal to

$$U_t(a) = \sqrt{\frac{2\log t}{C_t(a)}}. \tag{5.3}$$

5.4 Monte Carlo tree search

Monte Carlo tree search is a stochastic heuristic for efficiently exploring the agent's space of possible actions. Compared to algorithms like minimax, MCTS trades off optimality for speed and a possibility to stop the search at any time.

The focus of MCTS is on the analysis of the most promising moves, expanding the search tree based on a random sampling of the search space. The application of Monte Carlo tree search in games is based on playouts, also called rollouts. In each playout, the game is played out to the very end by selecting moves at random. The final game result of each playout is then used to weigh the nodes in the game tree so that better nodes are more likely to be chosen in future playouts.

Each round of Monte Carlo tree search consists of four steps as depicted in the Figure 5.1:

- Selection: From the root node, a selection strategy is applied recursively until a position is reached that is not a part of the tree yet. The selection strategy controls the balance between exploitation and exploration. The variant of MCTS that uses UCB1 as the selection strategy is called UCT (Upper confidence bounds for trees).
- Expansion: The expansion step happens when a new node n is visited for the first time at the end of selection. Expanding this node refers to storing it inside a list of expanded nodes for future use.
- Simulation: Simulation (also called playout or rollout) is the step that selects moves in self-play until the end of the game, starting at node n. This task usually consists of playing uniform random moves or pseudo-random moves until terminal node is reached. The reward given by the terminal node is taken is if it was given by the starting node n.

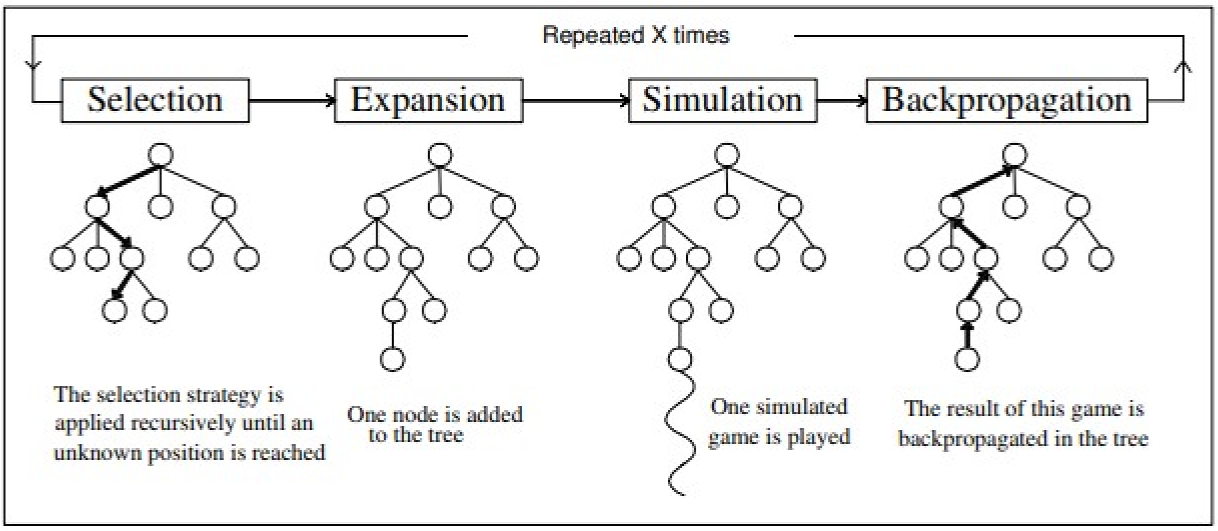
Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection strategy is applied recursively until an unknown position is reached
One node is added to the tree
One simulated game is played
The result of this game is backpropagated in the tree

Figure 5.1: The visualization of the 4 phases of the Monte Carlo tree search algorithm.

- Backpropagation: Backpropagation is the step that propagates the result of a simulated game backward from a leaf node to the nodes it had to traverse in order to reach that leaf node by updating the accumulated reward in each node.

The pseudocode of the algorithm is given in the Algorithm 3.

5.4.1 Final policy selection

At the end of the simulation, the actual move played by the algorithm is the action with the highest score available at the root node. The most common possibilities to determine the score are:

- The score is how many times the action was used.
- The score is the lower confidence bound on the action's reward.

However, it can happen that two actions with completely different outcomes will both get a very high score. This is not going to get properly reflected in the above policy. This might become an issue when a lot of randomness

Algorithm 3 The MCTS algorithm

```
function mcts(rootNode, simulations)
  expandedNodes = {}
  for s in {1,2..,simulations}
    node = rootNode

    # selection
    while node in expandedNodes
      node = select(node)

    # expansion
    expandedNodes.insert(node)

    # simulation
    reward = simulateGame(node)

    # backpropagation
    while hasParent(node)
      node.reward += reward
      node.visits += 1
      node = node.parent

  return rootNode action to node with maximum visits
```

is involved, for example when using the random rollout estimation with low number of rollouts. In order to limit the variance of the final policy under these circumstances, the final action can be played stochastically.

Again, there are different ways to compute the probabilities for each move according to its score, each with its own trade-off. We want our policy to quickly converge to the optimal move and therefore we will ignore all moves with score bellow 90% the maximum achieved score at given state. To get a smooth probability distribution we will rescale the remaining values with the Softmax transformation:

$$\sigma(x_1, ..., x_n)_i = \frac{e^{x_i}}{\sum_{j=0}^{n} e^{x_j}}. \tag{5.4}$$

5.5 Alpha-Beta UCT

The original UCT algorithm has two shortcomings:

- It requires an undefined amount of exploration.
- It needlessly revisits states like the terminal nodes.

This is because the algorithm is only guaranteed to find the optimal strategy in a limit, and therefore the actual state rewards r^* are unknown during the run. Both of these problems can be addressed by combining the propagation of the minimum and maximum reward bounds from the Alpha-Beta algorithm with the action selection and reward approximation from the UCT algorithm. The result is an algorithm that explores the game tree in a similar manner to UCT but prunes parts of the search space as the Alpha-Beta algorithm. This allows us to more effectively measure the impact of the key component of Monte Carlo tree search (that is, the action selection phase) on the overall algorithm performance.

We are going to refer to this modification as the offline Alpha-Beta UCT algorithm. All benchmarks in this thesis that show the performance of the offline UCT are going to use this variant unless stated otherwise. Its pseudocode is shown in the Algorithm 4.

Algorithm 4 The Alpha-Beta UCT Algorithm

```
function AlphaBetaUCT(rootNode)
  expandedNodes = {}
  while not(rootNode.solved)
    node = rootNode
    # selection
    while node in expandedNodes
      node = selectUnsolved(node)
    # expansion
    expand(node, expandedNodes)
    # simulation
    reward = simulateGame(node)
    # backpropagation
    while hasParent(node)
      node.reward += reward
      parent = node.parent
      # alpha beta bounds propagation
      if parent.player == maximizingPlayer
        if all children of node are node are solved
          node.solved = True
          parent.value = min(parent.value, node.value)
        if node.value > parent.beta
          node.solved = True # cut-off
        node.beta = min(node.value, node.beta, parent.beta)
      else
        if all children of node are node are solved
          node.solved = True
          parent.value = max(parent.value, node.value)
        if node.value <= parent.alpha
          node.solved = True # cut-off
        node.alpha = max(node.value, node.alpha, parent.alpha)

      node = parent

  return rootNode action with maximum value
```

Chapter 6

Search Efficiency Analysis

In this chapter, we explore the efficiency of Monte Carlo tree search in traditional games. Particularly, we investigate the number of explored positions required to find a winning strategy in Tic Tac Toe and Hex. Our goal is to confirm the assumption that the MCTS algorithm is effective at this task. We use the UCT algorithm with default and zero exploration along with the Alpha-Beta UCT and Alpha-Beta algorithms to get a better insight into which part of UCT is the most responsible for its performance. The experimental results are presented and discussed in Section 6.2.

6.1 The efficiency metric

To estimate how effective is an algorithm at utilizing new information (i.e., information that is obtained from expanding nodes in the game tree) we need to define an efficiency metric. A straightforward way to measure this is by using the size of the sub-tree explored by the algorithm, i.e., the number of simulations passed to the UCT algorithm 3. The UCT policy is calculated based to number of times V_{ha} each action $a \in A$ was used at given choice node $h \in H$ as discussed in Section 5.4.1:

$$\pi(h, a) = \frac{\exp(V'_{ha})}{\sum_a \exp(V'_{ha})} \tag{6.1}$$

Where V'_{ha} treats low number of visits as zero:

$$V'_{ha} = \begin{cases} 0 & \text{if } V_{ha} < 0.9 * \max_{a'}(V_{ha}) \\ V_{ha} & \text{otherwise.} \end{cases} \tag{6.2}$$

However, since it is not known how many simulations are needed to find the optimal policy pi^*, we are going to run the UCT algorithm with increasing number of simulations until π^* is found. In the case of Alpha-Beta and Alpha-Beta UCT, the calculation is simpler, as the number of explored states simply corresponds to the number of recursive calls of the Alpha-Beta function 2 and the final size of the *exploredStates* variable in the Alpha-Beta UCT algorithm 4, respectively.

6.2 Experimental results

In this section, we qualitatively measure the number of explored positions required to find a winning strategy with the MCTS and Alpha-Beta algorithm in Tic Tac Toe and Hex. The primary goal is to confirm the assumption that MCTS method is very effective at finding optimal policies in traditional games. However, due to the enormously large state space of each game, we will limit our analysis to only a subset of end-games. The procedure we use is the following. First, we randomly sample 1 000 subtrees in both games after 3 and 100 random moves, respectively. Furthermore, games solved by Alpha-Beta under 50 expansions are ignored, as we consider their game trees too small to be useful for our analysis. Finally, we measure the number of explored states of the online UCT algorithm 3 and the offline Alpha-Beta UCT algorithm 4. These numbers are then compared to the Alpha-Beta as a percentage $100\frac{E_U}{E_M}$ where E_U and E_M are the numbers of states explored by UCT (or Alpha-Beta UCT) and Alpha-Beta, respectively.

We measure the UCT performance in two settings. First, with the exploration constant C set to the standard value $C = \sqrt{2}$. Second, with the same constant set to $C = 0$, making the action selection is greedy. This allows us to highlight the role of exploration in the benchmarked games. Additionally, we use the average of 3 random rollouts as the heuristic function (a higher amount did not seem to have a significant effect on the performance). Furthermore, as computing the expected reward of a UCT based policy is costly (it amounts to running UCT search in every state reachable by the policy), we only do it when the number of simulations reaches a predefined number from the set $\{25, 50, 100, 150, 200, 300, 400, 500\}$.

The results are presented in Table 6.1 as a median along with mean and

Game	MCTS	C	median (eff. %)	mean (eff. %)
TicTacToe	online	$\sqrt{2}$	19%	23% ±15%
TicTacToe	online	0	19%	23% ±16%
TicTacToe	offline	$\sqrt{2}$	59%	69% ±44%
TicTacToe	offline	0	51%	58% ±30%
Hex	online	$\sqrt{2}$	3%	6% ±5%
Hex	online	0	7%	8% ±4%
Hex	offline	$\sqrt{2}$	25%	29% ±21%
Hex	offline	0	14%	20% ±20%

Table 6.1: A median and mean with standard deviation of explored states by UCT (online) and Alpha-Beta UCT (offline) relative to the Alpha-Beta algorithm on 1 000 randomly sampled games. The UCT is run with either default or no exploration when the parameter C is $\sqrt{2}$ or 0, respectively.

standard deviation. Figure 6.1 then depicts the distributions in the game of Tic Tac Toe. As the game is rather small, we were able to sample all subgames after 3 moves. The results in Table 6.1 show that the UCT algorithm dominates Alpha-Beta in both offline and online settings, performing only about 20% and 50% explorations done by Alpha-Beta, respectively. In Hex, the UCT is even more effective than in Tic Tac Toe. This is most likely caused by the shallow depth of the Tic Tac Toe game tree. Furthermore, the data also shows that limiting exploration in the algorithm (i.e., setting the constant C to 0) does not have a significant impact on its efficiency. On the contrary, in the offline mode, the efficiency is even improved! In other words, the reward estimates themselves must be largely responsible for the success of the Monte Carlo method. However, it is unclear why the estimates should be correct in the first place.

The experiment results confirm that UCT is, on average, more efficient than Alpha-Beta. This also holds for the offline Alpha-Beta UCT algorithm, which suggests that the algorithm efficiency cannot be attributed only to a complex heuristic function (i.e., trained neural network in the case of AlphaGo [Silver et al., 2017]) or its ability to work in an online manner. Furthermore, turning off the exploration by using setting $C = 0$ did not have significant effect on the number of explored states. This suggest the structure of the explored games must have some favorable properties that make the search more efficient.

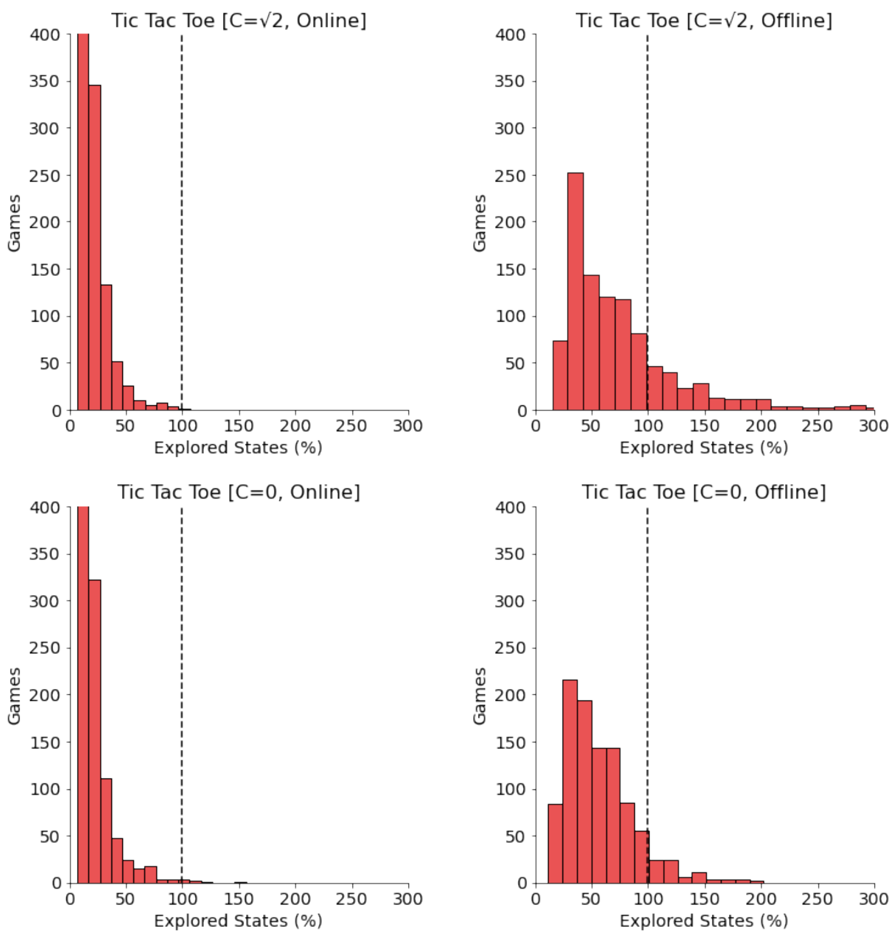

Figure 6.1: The distribution of the number of states explored by UCT as a fraction of the states explored by the Alpha-Beta algorithm (x-axis). The top row compares the standard UCT algorithm with $C = \sqrt{2}$ evaluated separately (online mode) in each state with the Alpha-Beta UCT, that is evaluated only once (offline mode). The bottom row mimics this setting, but with the parameter C set to 0. This disables exploration and makes UCT always expand state with the highest average reward.

Chapter 7

Game Properties

In this chapter, we explore several properties that we believe play significant role in MCTS performance. As highlighted in the Chapter 6, the standard UCT algorithm rarely requires more than 50 simulations to find the optimal policy and the offline Alpha-Beta UCT consistently outperforms the Alpha-Beta algorithm in both Hex and Tic Tac Toe. Moreover, removal of the exploration factor did not seem to have a significant effect on the algorithm performance 6.1. This can be only explained by the fact that such games possess favorable properties aligned with the inherent bias of the Monte Carlo tree search. It is unclear, however, what they are and why they exist. Here, we try to answer both of these questions. In the following text, we are going to propose several such properties, examine their effect on UCT, and attempt to explain their presence in each game.

7.1 Strong Uniform Policy

We believe that correct action reward estimates by the uniform policy is one of the key ingredients behind the success of Monte Carlo tree search. The uniform policy shows up in the MCTS algorithm precisely twice. First, as the heuristic function, i.e., the random rollout evaluation. This is not a uniform policy per se, but rather its approximation through random sampling. Second, in the UCT algorithm inside the UCB1 formula as the value estimate. Technically, this is true only in the early stages of the search, as the algorithm gradually selects optimal actions more and more often. However, this could be understood as a specific form of the weighted uniform policy, where the weights are initially set to $\frac{1}{N}$ where N is the number of moves, and later on, gradually increased for actions with better outcomes.

We have explained why we believe that uniform policy is the key ingredient in the UCT value estimates. Therefore, it should be the first property to be investigated since the effectivity of UCT is directly related to the accuracy of its value estimates. Specifically, it is very likely, that games where UCT performs well are also games where a uniform random policy gives good reward estimates. Imagine an extreme situation where the estimates become identical to true rewards. Evidently, in such a case, the algorithm will arrive at the optimal strategy immediately. In the general case, the more accurate the random estimates are, the fewer states should the UCT need to explore.

However, an accurate value estimate should not be strictly needed for good performance, as moving or multiplying all estimates by a positive constant does not change their maximum. Such transformation with a sufficiently small constant should have no effect on the selection process. As a consequence, we will be more interested in the correct ordering of moves than their exact estimates. Specifically, we want the expected value of a uniformly random strategy to be highest for the action that also achieves the highest reward with the optimal policy π^*.

7.1.1 Artificial Games

Artificial games allow us to precisely measure the role of the initial uniform policy value on the amount of explored states by the UCT algorithm, as they give us fine control over all parameters of the game tree.

We are going to use randomly generated, balanced-tree games with fixed branching factor and depth – 3 and 6, respectively. The terminal values are chosen from the set $\{-1, 1\}$ for both players. However, the vast majority of such games would be trivial for UCT to solve, i.e., require less than 50 simulations. In order to increase the portion of challenging games, we are going to choose only such games which contain precisely one unique sequence of moves guaranteeing a victory for the starting player. Therefore, we are going to refer to these games as the 1-Path Games.

The results are presented in Figure 7.1. It shows that games which on average, contain correctly ranked actions (when ordered by uniform policy value), also require a smaller amount of exploration done both by the UCT and Alpha-Beta UCT algorithms.

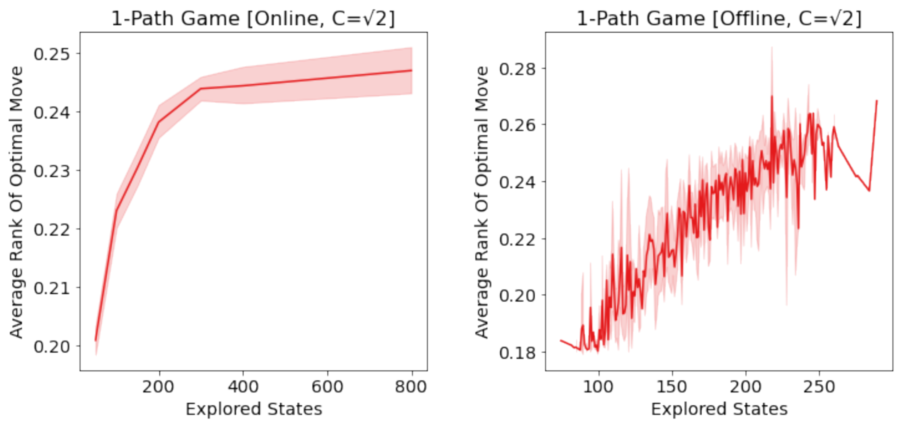

Figure 7.1: The explored states by UCT (x-axis) when compared to the optimal move rank (when ordered by uniformly random policy value) averaged across all states of each game (y-axis). Rank 0 is the best, while 3 is the worst.

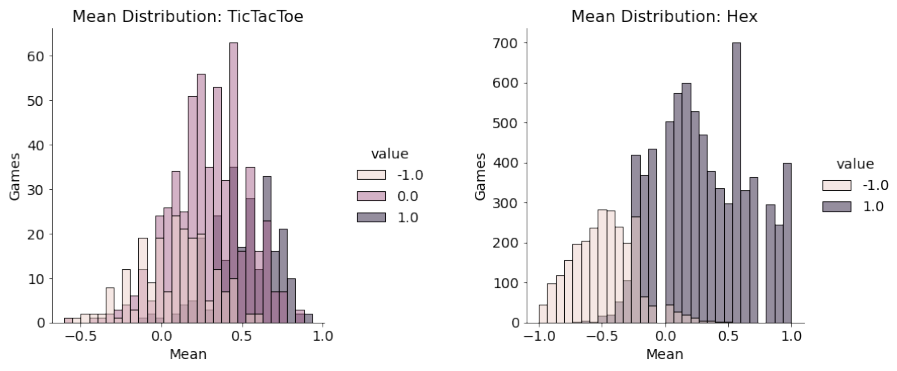

Figure 7.2: The estimated values (x-axis) of uniform strategy from 1 000 random-sampled games in Tic Tac Toe and Hex. The color of each bin refers to the minimax value r^* of each game.

7.1.2 Traditional Games

In this section, we investigate the role of uniform policy value in the game of Tic Tac Toe and Hex 11x11. We randomly sample 1 000 and 10 000 games at a depth of 3 and 110, respectively. Then we compute the value of optimal and uniform strategy (approximated by 30 random samples) in each of those games. The estimated value distributions are presented in Figure 7.2.

The charts show that uniform policy is not a very good estimate of the exact true values. However, the values can be used effectively for move ordering, as each distribution has a distinct mean. This is most visible in Hex, where the distributions also have almost no overlapping regions. In Tic Tac Toe, the overlap is more significant, mainly for draw-scoring moves. However,

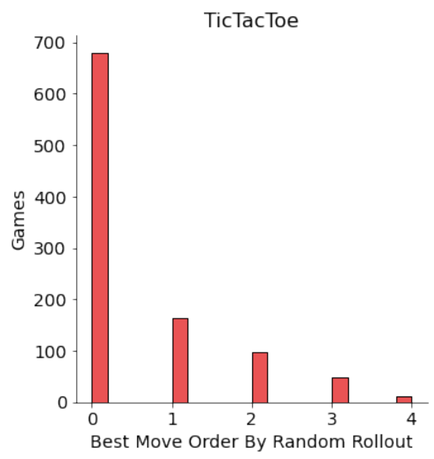

Figure 7.3: The distribution of the optimal move ranks (when ordered by uniformly random policy value) averaged across all states of each game (y-axis). Rank 0 is the best, while 4 is the worst.

this still makes the estimates relevant for differentiating between winning and losing actions. Most importantly, the distributions are also correctly arranged - the losing and winning moves have the lowest and highest mean, respectively. An opposite order would most likely have a negative effect on MCTS performance, as it would increase the probability of receiving uniform policy value estimates that do not reflect the true value of each move.

Most importantly, the best actions in Tic Tac Toe also have the highest uniform strategy value in approximately 70% states as depicted in Figure 7.3. This means the UCT will have a 70% chance to explore the correct (optimal) move right away. The last 10 moves of Hex are even more favorable, as the percentage of optimal moves with the highest uniform strategy value reaches a staggering 95%.

7.2 Advantage Accumulation

The term “advantage” is often used when talking about the strength of a particular game position, even though it has no standardized definition in game theory. One could call positions with a guaranteed win as advantageous. Yet, that would not align with its informal description, as some positions are described to be more advantageous than others.

Informally, possessing an advantage should guarantee better odds of winning a game. The traditional strategy common to most games is then to accumulate such an advantage in order to improve on those odds, for example, by capturing additional enemy pieces in chess. Moreover, it is also usually the case that such accumulation favors the player with the most advantage - capturing an additional piece in chess is way easier when the opponent has fewer pieces to start with.

This suggests that traditional games have a very specific structure. They are, in essence, positional. That is, strong positions lead to positions of similar strength; the winning positions are in a close neighborhood (in terms of actions) to other winning positions. In other words, the change in position strength by a single move should be, on average, very small. Furthermore, it is reasonable to assume that the strength should, in some sense, refer to the total number of winning states accessible from a given position, as humans rarely play perfectly.

Combining these two facts together could explain the surprising strength of uniform policy. First, if the optimal winning move is guaranteed to be in close proximity of other winning moves, then the best chance finding such a move is in the region of highest winning-move density. Second, if adjacent positions have similar strength, then the initial value of a state should be assumed to be of its neighbor. Both of these assertions favor the uniform policy, as it averages the heuristic value (strength) of neighboring states and directly corresponds to winning-move density in the case of $\{1, -1\}$ terminal rewards.

7.2.1 Artificial Games

In this section, we explore the impact of positional structure on MCTS efficiency in synthetic benchmarks. Using synthetic trees has two advantages. First, we know precisely the amount of advantage accumulation that happens in a given game. Second, we can generate games with different accumulation rates and measure their influence on MCTS performance. In order to achieve that, we design a simple resource-stealing game, where both players try to steal money from each other. The one with a positive balance wins. This is meant to imitate the exchange of pieces in games like Chess or Go. We give it the name “Robbers”.

The game is defined by a tuple (D, B, R, S, K) where

- $D \in \mathbb{Z}$ is a the depth of the game tree (i.e. the number of turns).
- $B \in \mathbb{Z}$ is the branching factor (i.e. the number of ations).
- $R \in \mathbb{R}^{D^B}$ are the rewards for each action.
- $S \in \mathbb{R}^2$ is the sum of money both players posses (their balance).
- $K \in \mathbb{R}$ is an advantage accumulation parameter.

The game is played in turns $d \in 0, 1, .., D$:

- Player $i \in 0, 1$ chooses one of $0, 1, .., B$ actions.
- This action steals $R_{d,b} - K \cdot S_i$ from the opponent balance.
- Now new turn begins, second player $(1 - i)$ gets to act.
- This repeats for D steps. Then, the player i with highest balance S_i wins.

The rewards R are known by the players ahead of time and don't change during the game. The "Karma" parameter is a simple tool to control the amount of advantage accumulation in the game. It makes it either easy or hard to accumulate additional money when $K < 0$ or $K > 0$ respectively.

7.2.2 Experiments

Here, we measure the effect of advantage accumulation on MCTS performance in the game of "Robbers" as defined in 7.2.1. We use a fixed depth and branching factor $D = 10, B = 4$. The "Karma" factor and action rewards $R_{d,b}$ are randomly uniformly sampled from the interval $[-1, 1]$. In total, we generate 10 000 games with this setup.

As can be seen from Figure 7.4, games with higher advantage accumulation (low Karma) tend to require an increased amount of exploration. The only exception to this is the UCT algorithm in offline mode with parameter $C = \sqrt{2}$. This is because high advantage accumulation makes sibling states very likely to have an identical expected reward - both for an optimal and random uniform policy. As a consequence, the estimated rewards by UCT will also be almost identical, making MCTS behave like a breadth-first-search

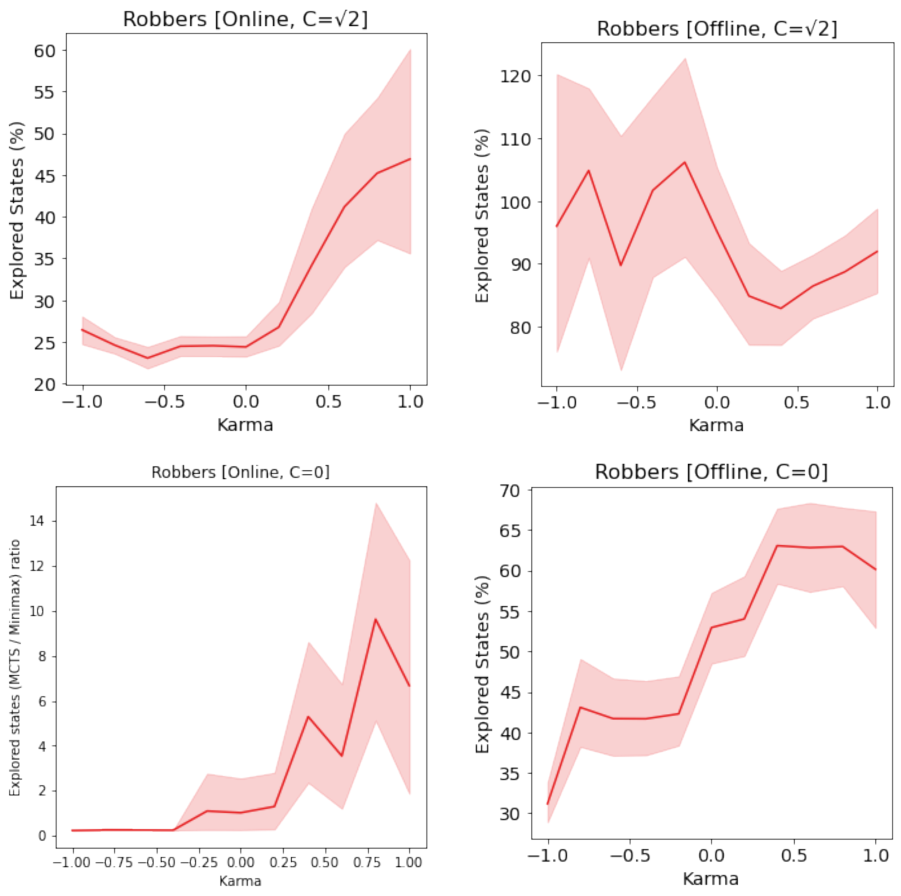

Figure 7.4: The amount of explored by UCT relative to Alpha-Beta (y-axis) in the game of Robbers 7.2.1 with varying Karma parameter (x-axis). The Karma parameter is inversely correlated to the amount of accumulation in given game. Similar to 6.1, both online and offline mode of UCT is presented, with the standard $\sqrt{2}$ and zero exploration parameter C.

algorithm. Therefore, the algorithm will explore many more states than if it simply greedily searched the first state it came across, as depth-first-search algorithms (like Minimax) do. However, as visible in the graph, this issue disappears once the algorithm is made greedier, for example, by turning off exploration (setting $C = 0$).

7.2.3 Traditional Games

Both Tic Tac Toe and Hex are subject to advantage accumulation, as we will show in this section. Informally, in Tic Tac Toe, each move is guaranteed to have only a slight impact on the total amount of winning positions in the game. In Hex, not taking an important action does not remove a significant amount of winning position from the game. The formal description are explained in more detail in the following subsections.

7.2.4 Tic Tac Toe

The rules of Tic Tac Toe declare the player who connects N symbols of the same type in the line as the winner. This means that a space on the board can show up only in limited win-giving configurations. The space can be at $\{1, 2, .., N\}$ positions of a left diagonal, right diagonal, vertical, and horizontal line. Thus, the total amount of winning positions s that include a given space must be:

$$s \leq 4N. \tag{7.1}$$

This bound suggests that a single move should not significantly alter the value of the uniform policy unless there are only a few additional moves left. In such case, the proportion of game-winning configurations should remain fairly stable across states sharing common predecessor, as expected by our advantage accumulation formulation.

7.2.5 Hex

In Hex, on the other hand, the bound (7.1) does not hold. Consider a placement of stones that guarantees a win for one of the players, i.e., connects the opposite sides of the board. Such sequence can consists of up to half of all available spaces on the board. However, since this sequence must connect opposing sides of the board, it isn't ever possible for both players to have a winning path at the same time in a single board configuration. This allows us to calculate the probability (in a simple manner) of obtaining a sequence of stones when playing by the uniform policy, as we can treat each space to be filled with with a stone of one of the players. This is because placing stones after one of the players won is not going to change the outcome of the game.

The rules of Hex however have other interesting favourable properties. Consider a game-winning move a^* available in a game state h. The probability of playing such a move is equal to $\frac{1}{D}$ where $D \in \mathbb{N}$ is the number of available actions. Without prior knowledge of the game rules, the probability of victory $w(h)$ with a uniformly random policy would be $w(h) \geq \frac{1}{D}$. However, in Hex the bound is stronger and does not depend on D: $w(h) \geq \frac{1}{2}$. This is because the uniform policy gives both players the same probability to place their stone on a given space (both choose from the same amount of spaces). As a consequence, in a completely filled hex board, the starting player will have at least 50% probability to control the game-winning space.

We can generalize this bound for more than a single action. Imagine a Hex board configuration h where the current player can win by placing $m \in \mathbb{N} \leq D$ stones at $2D \in \mathbb{N}$ available spaces. For simplicity assume that $2D$ is even. Then the chance to win for the current player following the uniformly random policy $w(h)$ is bounded according to the Equation 7.2, and can be further generalized for multiple independent game-winning stone placements.

$$w(h) \geq \frac{(2D-m)!D!}{(2D)!(D-m)!}. \tag{7.2}$$

Because Hex has only two rewards $\{1, -1\}$, the value of uniform policy $u(h)$ is uniquely determined by $u(h) = 2w(h) - 1$. As a consequence, the bound 7.2 guarantees the uniform policy value will give (in states with many available actions) significantly more accurate prediction of the optimal move sequence than one could see in adversarially designed games similar to the example given by the Figure 2.1.

Another property that seems to have positive effect on the UCT performance as shown on the synthethic benchmarks in the Figure 7.4 is a small relative change in the uniform policy value between neighbor states. We will show how the value of uniform policy depends for a given state depends on the uniform policy value of some of its neighbors.

Consider a board configuration h with the player with stone symbol O at turn. The second player uses the stone symbol X. Let $w(h)$ be the probability of victory with uniformly random policy and h_i^y be a board configuration same as h with the addition of stone $y \in \{O, X\}$ being placed at space s_i. Similarly, let h_{ij}^{xy} be the board configuration h with the additional stones x, y placed at the positions i, j, respectively. Finally, let $c(h)$ denote the total number of board configurations available from h by any sequence of actions, and let $q = \frac{1}{2} - q' = \frac{c_{12}^{OX}}{c(h)}$. Then it must hold that

$$w(h_1^O) = 2w(h) - w(h_1^X) = 2w(h) - 2q' \cdot w(h_{12}^{XO}) - 2q \cdot w(h_{12}^{XX}) \tag{7.3}$$

$$w(h_2^X) = 2w(h) - w(h_2^O) = 2w(h) - 2q' \cdot w(h_{12}^{OX}) - 2q \cdot w(h_{12}^{OO}) \tag{7.4}$$

$$w(h_{12}^{OX}) = \frac{1}{q}(w(h) - q' \cdot w(h_{12}^{OO}) - q' \cdot w(h_{12}^{XX}) - q \cdot w(h_{12}^{XO})) \tag{7.5}$$

By substituting the first two equations into the third, we also get:

$$w(h_{12}^{OX}) = \frac{1}{q}(-w(h) - \frac{1}{2}w(h_1^O) - \frac{1}{2}w(h_2^X) + q \cdot w(h_{12}^{XO})) \tag{7.6}$$

This gives us an insight how are winning and losing configuration accumulated across subsequent moves. Clearly, the change in uniform policy value of after performing an action depends on the uniform policy value of the other locally available actions. In other words, the advantage accumulation condition seems to hold, i.e., strong moves with high uniform policy value are indeed in close proximity of other strong moves.

Chapter 8

Conclusion

We have described the origin of the Monte Carlo tree search and its popular variant UCT that adapts the UCB1 algorithm from the Multi-Armed Bandit Problem to MCTS. We examined the theoretical bounds of this algorithm and pointed out their shortcoming when applied to traditional board games like Go or chess. Next, we evaluated the performance of the vanilla UCT algorithm on the two popular games, Tic Tac Toe and Hex, and showed the bounds to be overly pessimistic. We have also introduced a UCT modification that is guaranteed to arrive at optimal strategy after expanding all nodes in the game tree by combining the state pruning from Alpha-Beta algorithm with the MCTS node selection and backpropagation phase. We showed that this variant in offline mode significantly outperforms the traditional Alpha-Beta algorithm, exploring on average at least 50% states less in the games of Tic Tac Toe and Hex.

We have discussed why this might be the case and noted that the initial value estimates done by random rollouts are surprisingly accurate in both of these games. We have shown on synthetic benchmarks that the initial estimates indeed play a huge role in the UCT algorithm performance: The closer the random rollout values of the optimal moves were to their true value (when compared to non-optimal moves), the fewer states required UCT to explore.

Our explanation for this phenomenon was an accumulation of advantage during the course of each game, as accumulation advantage would lead to similar rewards between sibling moves - a property that we suspected to improve the MCTS performance. We have shown on synthetic benchmarks that this assumption indeed holds: A decrease in our accumulation parameter leads to a decrease in value similarity and an increase in explored states. Finally, we have shown that the special structure of both Hex and Tic Tac Toe gives rise to a specific type of advantage accumulation, as it limits the amount by which a sibling move's random rollout value can differ.

8.1 Future Work

This thesis explored the performance of the UCT algorithm on Hex and Tic Tac Toe. However, both of these games share a similar structure - i.e., the board is filled with pieces that cannot be removed, each player shares the same set of moves and wins by making a connected pattern of symbols or stones. However, games like Go or chess that do not share some of these properties. This makes both of these games much harder to study. It also means some of the insights we presented might not apply to these games. The next logical step would therefore be to check whether how these properties manifest in different games, or come up with other explanation for the success of the Monte Carlo tree search method in those domains.

Source Code

The source code is available with this thesis. It contains the definition of the algorithms used in this work, the artificial games used in synthetic benchmarks and also the some useful tooling and statistics to make their exploration easier. Most of the experiments and visualizations are shown in a Python3 notebook file. In order to run the experiments, you will need the OpenSpiel library [Lanctot et al., 2019] installed (follow the instructions on their github page). To run the experiments, using the jupyter notebook is recommended.

The project structure is the following:

- *!graphics.ipynb* contains the visualizations and benchmarks
- *mcts2.py* contains the UCT definition
- *mcts.py* contains some useful UCT utilities
- *minimax.py* contains the minimax algorithm
- *stats.py* contains statistics definitions
- *game.py* contains game statistics aggregators
- *games.py* contains the artificial games
- *rollout.py* contains random rollout simulation logic

Bibliography

Bruce Abramson. Expected-outcome: A general model of static evaluation. *IEEE transactions on pattern analysis and machine intelligence*, 12(2): 182–193, 1990.

Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. Exploration–exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science*, 410(19):1876–1902, 2009.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. *arXiv preprint cs/0703062*, 2007.

Ian Frank and David Basin. Search in games with incomplete information: A case study using bridge card play. *Artificial Intelligence*, 100(1-2):87–123, 1998.

Sylvain Gelly and David Silver. Achieving master level play in 9 x 9 computer go. In *AAAI*, volume 8, pages 1537–1540, 2008.

Matthew L Ginsberg. Gib: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, 14:313–368, 2002.

Wassily Hoeffding. Probability inequalities for sums of bounded random variables. In *The collected works of Wassily Hoeffding*, pages 409–426. Springer, 1994.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. OpenSpiel: A framework for reinforcement learning in games. *CoRR*, abs/1908.09453, 2019. URL `http://arxiv.org/abs/1908.09453`.

Jeffrey Richard Long, Nathan R Sturtevant, Michael Buro, and Timothy Furtak. Understanding the success of perfect information monte carlo sampling in game tree search. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

Nicholas Metropolis and Stanislaw Ulam. The monte carlo method. *Journal of the American statistical association*, 44(247):335–341, 1949.

Dana S Nau, Mitja Luštrek, Austin Parker, Ivan Bratko, and Matjaž Gams. When is it better not to look ahead? *Artificial Intelligence*, 174(16-17): 1323–1338, 2010.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419): 1140–1144, 2018.

Brandon Wilson, Austin Parker, and DS Nau. Error minimizing minimax: Avoiding search pathology in game trees. In *International Symposium on Combinatorial Search (SoCS-09)*, 2009.

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Vonášek** Jméno: **Josef** Osobní číslo: **456932**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Umělá inteligence**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Překvapivá účinnost algoritmu Monte Carlo tree search

Název diplomové práce anglicky:

The Surprising Effectivity of Monte Carlo Tree Search

Pokyny pro vypracování:

Seznam doporučené literatury:

[1] Long, Jeffrey Richard, et al. "Understanding the success of perfect information monte carlo sampling in game tree search." Twenty-Fourth AAAI Conference on Artificial Intelligence. 2010.
[2] Brown, Noam, and Tuomas Sandholm. "Simultaneous abstraction and equilibrium finding in games." Twenty-Fourth International Joint Conference on Artificial Intelligence. 2015.
[3] Moravčík, Matej, et al. "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker." Science 356.6337 (2017): 508-513.
[4] Gelly, Sylvain, and David Silver. "Combining online and offline knowledge in UCT." Proceedings of the 24th international conference on Machine learning. ACM, 2007.
[5] Rosin, Christopher D. "Multi-armed bandits with episode context." Annals of Mathematics and Artificial Intelligence 61.3 (2011): 203-230.
[6] Wilson, Brandon, et al. "Improving Local Decisions in Adversarial Search." ECAI. 2012.

Jméno a pracoviště vedoucí(ho) diplomové práce:

RNDr. Vojtěch Kovařík, Ph.D., centrum umělé inteligence FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **11.02.2020** Termín odevzdání diplomové práce: **04.01.2022**

Platnost zadání diplomové práce: **30.09.2021**

RNDr. Vojtěch Kovařík, Ph.D.	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce		podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání Podpis studenta