Preference-based Monte Carlo Tree Search for Multiplayer Domains

Master-Thesis von Julius Stecher
Tag der Einreichung:

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Tobias Joppen

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Knowledge Engineering

Preference-based Monte Carlo Tree Search for Multiplayer Domains

Vorgelegte Master-Thesis von Julius Stecher

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Tobias Joppen

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den October 26, 2017

(Julius Stecher)

Abstract

Monte Carlo Tree Search is a common approach for finding approximate solutions in discrete problem spaces. The basic variant of this algorithm does not require domain knowledge and has a number of desirable properties that make it well suited for realtime application, such as the ability to terminate early and still provide useful results even if an optimal solution was not yet found. MCTS can work both with terminal rollouts as well as heuristic evaluations of non-terminal states (limited rollout length), the latter generally making use of numeric feedback. Preference based MCTS aims to not use these values directly, but rather always in context of a comparison with another feedback value to form a preference. Applying this to a multiplayer context will result in a preference among available actions in a game state based on the interests of the particular player conducting the action. The experiments show that the preference-based approach exhibits inferior performance on the Connect Four domain under the conditions tested.

Contents

List of Figures

List of Tables

1 Introduction

A problem requiring successive choices in different *states* is called a *sequential decision problem*. Games that operate in discrete state spaces are a subset of this type of problem that requires a solution in the form of information about which decision to take in each state. For a computer agent designed for game playing, these decisions impact the playing strength of the agent and as such, a well-informed choice is necessary in each single state.

1.1 Motivation

Artificial Intelligence in games has seen a significant boost in playing strength in domains previously unfeasible for computer play since the description of Monte Carlo Tree Search (MCTS) in 2006 [3]. In particular, it contributes a means of handling games with high branching factors, deep trees and game states with no easy way to assign heuristic values to (in its generalized, basic form). As such, MCTS can avoid some of the performance and memory issues that arise with other forms of tree search on these domains and has a number of interesting properties that make it feasible for realtime use. Among these is the general lack of requirement for state evaluation beyond applying the ruleset for win conditions (if so desired), the tree search approach closely resembling a best-first method (resulting in asymmetric tree growth favoring promising areas of the game tree), and possibly most importantly, the ability to terminate at any time and return an educated guess of the optimal action in the root node. It is because of the latter that MCTS can provide easily scalable playing strength and make good use of a given computational or time budget, since unlike an exact solver, it neither terminates on its own nor does it have to in order to return *any* usable result at all.

Given two states s_1 and s_2, a preference $P(s_1, s_2)$ makes a statement about whether s_1 is more desirable than s_2 or vice versa, and as such provides the pure result of a direct comparison [5]. This does not necessarily require numerical information about the quality of either state - in fact, it may be hard to find an absolute quality metric for states of a given domain, but still viable to generally form a preference given two states. For example, even a domain expert may have a hard time to judge the exact value of a car in terms of an amount of currency, but it may be easier to state with some confidence that a particular car is in better condition, safer, or attributed more resale value than another. A preference does not necessarily imply the existence of a numerical rating method, but the latter can be used to obtain preferences. In this case, the information about the value difference is lost and only the comparison result is preserved. It can be argued that a loss of information is not always undesirable - in fact, it may be helpful in avoiding local optimal if the numerical reward function is generally reliable, but plagued by local noise or other inaccuracies.

1.2 Thesis Structure

A selection of definitions from game theory will be introduced in 2, focusing on multiplayer game domains. Then, the basic idea behind MCTS in general as well as an existing preference-based approach to MCTS will be summarized, including the chosen method of adapting both a

realtime UCT-based MCTS configuration as well as the particular preference-based approach to multiplayer. The multiplayer domain of Connect Four will be formally introduced in 3. Because both algorithms require parameter tuning, this will be conducted in a fair and balanced way in 4 before pitting them against each other in a series of experiments. The results on this particular domain do not show an advantage of the preference based approach, with several possible reasons given in 5, ranging from domain-specific reasons to generalized statements about domain properties that may be disadvantageous for a preference-based approach in the case of MCTS.

2 State of the art: Artificial Intelligence in Game Playing

The following chapter will largely concern itself with a short non-exhaustive summary of relevant existing approaches to artificial intelligence based on stochastic methods and tree search on discrete domains, based on material from several sources [3] [2] [7] [6].

2.1 Game Theory

Complex games such as Go have been around since ancient times. However, formal research with a mathematical basis was not widely applied to strategic games before 1944 , when it effectively first received notable attention with the publication *Theory of Games and Economic Behavior* [8]. *Game theory* is an extension of decision theory with the aim of formalizing the interaction of multiple agents according to a set of estabilished rules, and as such, a game can be described by a set of components as follows [3].

2.1.1 Formal model for games

- S: The set of all states. Frequently, the cardinality of this set makes enumeration impractical.
- $s_0 \in S$: The initial state, before any actions have been executed by any agent.
- $n \in N$: The number of players. Of specific interest in this thesis is the case $n = 2$
- A: The set of all actions.
- $A(s) \subseteq A$: The set of actions available to an agent in a state $s \in S$. Note that some states s may only allow for a strict subset $A_s \subset A$ of actions, for example a movement action in a grid world setting rendered illegal due to the destination space being obstructed by a static obstacle (such as the border of the playing field). Note that only illegal actions are excluded; an action that would lead to the immediate loss of the active agent may well be (and usually is) a legal move, as such no information about the expected reward can be derived from the (non-) availability of an action in a specific state. However, we set $A(s) = \emptyset$ for a game-over state (this will be defined later).
- $transition : SxA \rightarrow S$: The ruleset of the game, formalized by a *state transition function* that maps state-action pairs to successor states.
- $r : S \rightarrow [0-1]^n$: The reward function. In case of $k = 1$, a reward is given as a real number. Within the scope of this thesis, all reward values - both atomar ones as well as individual components of reward vectors - are assumed to be within the range of 0 to 1.
- $P : p_x$ with $x \in (0, 1, ..., n-1)$: The agents (players) involved in the game. In addition, we reserve $index : P \rightarrow \{0, 1, ..., n-1\}$ as a helper function to map each player to an index number.
- $player(s) : S \rightarrow P$: The player whose turn it is in a particular state.

It should be noted that this is essentially a specialization of a generic sequential decision process - more precisely, a Markov decision process (MDP) with the additional constraints that the transition from a state s_1 via an action a will always result in the same successor state s_2

(*determinism*). In addition, the algorithms used to handle games specified as such in this thesis will not provide feedback for a transition unless $|A(s_2)| = 0$ (the successor state is a *terminal state*, ensuring that the game ends in a victory for at least one player or a draw for all players), or the algorithm forces a *heuristic evaluation* of s_2 due to specific parameter settings (more detailed description to follow in later sections), assigning an estimated reward vector to a state that does not yet imply the termination of the game.

2.1.2 Properties of Multi-player games

- **Competition:** A game is called *zero-sum* if the sum of all components of the reward vector is constant - usually zero (if the rewards range from -1 to 1) or one (if the rewards range from 0 to 1). In a zero-sum game, every victory or rise in reward for one player equals a loss or lowered reward for at least one other player; specifically, in a two-player game, zero-sum implies strict competition (as the concept of *coalitions* can be disregarded).

- **Information:** Can be either perfect (each player knows everything about the game and can make informed decisions without accounting for unknown variables) or incomplete (each player may at one or more points during the game be forced to make decisions that are partially based on assumptions).

- **Determinism:** If the game solely depends on the actions chosen by the players, it is deterministic. This is not the case if any randomness (such as dice, ordering of cards in a stack or actions conducted by a third party) has any influence over the successor state of a particular state.

- **Action order:** Sequential games will feature actions being applied in a specified order. This can take the shape of alternating (or circular in case of $n > 2$ turn order for the players involved, including reversal of turn order (*uno* card game) or any arbitrary rules, as long as the sequence of all actions executed during the flow of the game adheres to a *strict total ordering*, e.g. for any two actions a_1 and a_2, either the consequences of a_1 have already materialized before a_2 is executed, or vice versa. A game where this is not the case would have actions be executed simultaneously.

- **Discreteness:** A game is discrete if execution of actions is not done in real-time, but instead clearly separated into intervals.

2.2 Bandit-based methods

2.2.1 From one-armed to k-armed bandits

The underlying problem definition of a bandit problem explains its namesake: The classic casino slot machine *one-armed bandit* allows merely a single action choice (pulling the arm) which triggers an event of entirely random reward, without specifically accounting for any additional properties like the speed of pulling the arm. As such it is a purely random experiment, with

the defining difference from a (fair) dice roll or coin toss that the player is not told about the odds, and while they may have been told about the possible reward brackets, this information is meaningless without knowledge of the underlying probabilities, which would depend on internal specs of the *black-box* machine that takes a binary input as trigger (e.g. arm pull or button press = *applying an action*) and converts it into a reward feedback.

All by itself, this does not offer the player any choice at all except to eventually stop playing. However, if there are more than one machine available, each with their own individual odds, the player is suddenly faced with an interesting problem: any action can now be seen as a choice between the k available arms. Because the player still does not have any prior information about the underlying odds of *any* involved machine, this does not initially make a difference - the decision of which arm to first pull must still be made on a purely random basis. However, as the player continues to play the slot machines, they will eventually accumulate experience as to which machine tends to pay better.

Of course, this is highly unreliable information, but as time passes and a specific arm has been pulled many times, chances are that some machines turn out to be worse choices compared to others, and the player will develop a preference for one or more machines (*exploitation* of gained knowledge). Of course, this could also be a trap: if a slot machine by chance nets a high reward once or twice, it may still feature objectively worse odds under the hood, and if the player narrows down their preference too quickly, they would continue to play that machine believing this to be in their own best interest, while their expected net reward sum might end up higher if they continued to occasionally test other machines (*exploration*). The decision which bandit to play next is determined by the player's *policy*.

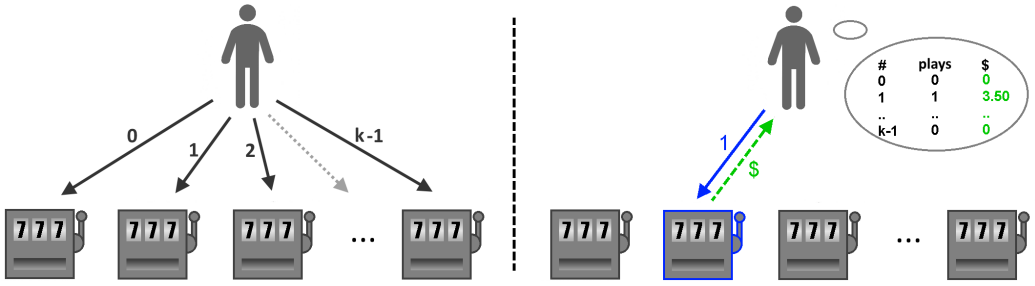

Figure 2.1.: Visualization of the k-armed bandit problem

Finding the optimal arm in a k-armed bandit problem

Staying with the previous example, it is easy to see that the optimal expected reward would be attainable by continuously playing only one machine - the one with the best expected reward per pull. Naturally, this is a unrealistic, especially considering that the player will begin their imaginary casino trip with no prior knowledge of the individual odds, and even after playing a while, will only have a rough, iteratively refined idea of how advantageous a particular machine is compared to another. The *regret* is thus the difference of the reward sum obtained by consistently playing the theoretical optimum n times, minus the reward sum actually obtained by following the player's policy n times and summing up the rewards. A good policy should never

assign a probability of zero to play any given arm [3], as no individual machine can at any point be *proven* to not be the optimal choice, and completely abandoning it for the rest of the casino session would let regret grow indefinitely and without an upper bound with increasing number of plays.

Regret growth cannot be ensured to be stopped entirely by any given policy - however, it is theoretically possible to limit its growth to a constant factor of $O(ln(n))$ [3], which has spawned several policies for the solution of the *exploration vs exploitation* tradeoff problem. For computational purposes, it is preferable to only rely on the reward history of the individual available actions, leading to the definition of the *Upper Confidence Bound* (UCB), more specifically the *UCB1* variant, which is going to be briefly summarized as follows.

Upper confidence bounds

A simple measure for the quality of a choice is the summed-up reward, divided by the number of times n_i this particular choice has been empirically tested, resulting in the average reward $\overline{X}_i$ of action i, with $X_{i,j} \in [0,1]$ the result of the experiment number j of action choice i:

$$\overline{X}_i = \sum_{j=0}^{n_i} \frac{X_{i,j}}{n_i} \tag{2.1}$$

This of course could result in a specific choice irreversibly losing competitiveness, even though it might have turned out to be the optimal one. However, this exploitation trap can be avoided by artificially boosting the value of choices with less available data, resulting in the *upper confidence bound* that choice i will be optimal [2] [7] [3]:

$$UCB1 = \overline{X}_i + \sqrt{\frac{2 \cdot ln(n)}{n_i}} = \sum_{j=0}^{n_i} \frac{X_{i,j}}{n_i} + \sqrt{\frac{2 \cdot ln(n)}{n_i}} \tag{2.2}$$

which is just the average reward (2.1) with an added bonus term that grows logarithmically while action i is not being tested in favor of other available actions in that state. Of course, the value obtained for $\overline{X}_i$ is still merely an estimate. Using Hoeffding's inequation, it was shown that the probability of our estimate being off by more than $|\sqrt{\frac{2 \cdot ln(n)}{n_i}}|$ compared to the actual (unknown) value μ_i is lower than or equal to $2n_i^{-4}$ [7] [2], thus decreasing every time action i is being tested. Applying this formula to a single k-armed bandit problem to select the next arm being pulled and repeating this process for a sufficiently large number of iterations will thus make it more likely that the action with the highest assigned value is the objectively best choice, even though this can never be guaranteed in a finite number of iterations.

Upper confidence bounds for trees

The guarantee of convergence UCB1 can provide for a single k-armed bandit problem only holds if testing of actions is done in an entirely random, evenly-distributed fashion. As we

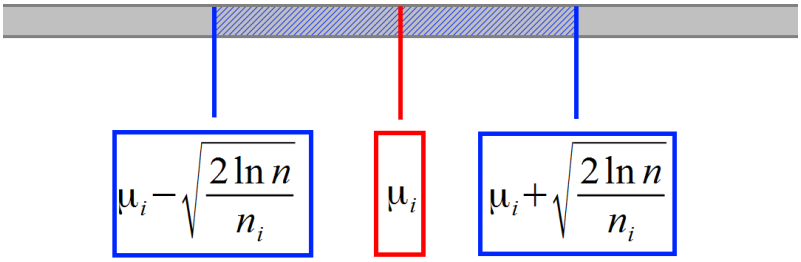

Figure 2.2.: The probability that the true value lies within the confidence interval grows over time

will see in section 2.3, Monte Carlo Tree Search spans a search tree where each node itself can be regarded as a k-armed bandit problem, with the resulting tree being a tree of k-armed bandits. Starting from the root node of that tree, the first actions are thus not selected at random. Instead, the action selection is initially controlled by a *policy* meant to handle the exploration/exploitation tradeoff as described in subsection 2.2.1, with the advantage that the first couple of actions (as opposed to just the first) are chosen according to the k-armed bandits of the respective nodes. In the case of MCTS, we will introduce the *tree policy* in subsection 2.3.2 for this purpose. Depending on the size of the tree, several actions in sequence could be chosen by this non-random policy. As such, altering the UCB1 formula (2.2) is necessary to preserve the convergence property; it was shown that adjusting the exploration term by a factor C achieves that effect, resulting in the UCT formula (*Upper confidence bounds for trees* [7]):

$$UCT_i = \overline{X}_i + 2C\sqrt{\frac{ln(n)}{n_i}} = \sum_{j=0}^{n_i} \frac{X_{i,j}}{n_i} + 2C\sqrt{\frac{ln(n)}{n_i}} \tag{2.3}$$

Where for $C = \frac{1}{\sqrt{(2)}}$, same-strength convergence was shown [7], although the value for C used in practice is usually determined empirically. We will later see that a smaller value for C is better suited for the particular domain and the experiment conditions handled in this thesis.

2.2.2 The dueling k-armed bandit problem

The player in the above example always conducts their experiments in the same fashion: A single action is selected for testing, which will return a (numerical) reward that can then be used to update and refine the knowledge base one step at a time. Then, the experiment is repeated, this time making an informed choice based on the new information (which may lead to the same, previously favorable action being tested (*exploitation*) or a different action being selected instead (*exploration*). However, obtaining such a numerical reward, while trivial in the case of a slot machine payout, is not guaranteed to be a trivial task for all problems, as such a reward implies a total ordering among evaluated actions. In many cases, it is easier to evaluate which one of two available outcomes is preferable. Note that such a preference can also be established based on a numerical reward; in this case, the higher score is going to be seen as preferable compared to the lower score, but if only this preference is being retained, the information about the *difference* between both scores is lost.

This does not necessarily have to be a disadvantage, and can in some cases even be seen as desirable, because inaccuracies in the score returned by an evaluation only matter if they would be

capable of flipping the preference, and most importantly, the effect of outliers in the evaluation can be reduced. In the casino example, obtaining a high payout at a specific slot machine once would most likely lead to the player implicitly or explicitly considering it to be the best choice for several following iterations, even if this was merely the result of one lucky pull. In contrast, if merely forming a *preference* among actions, this payout would merely be seen as preferable to another.

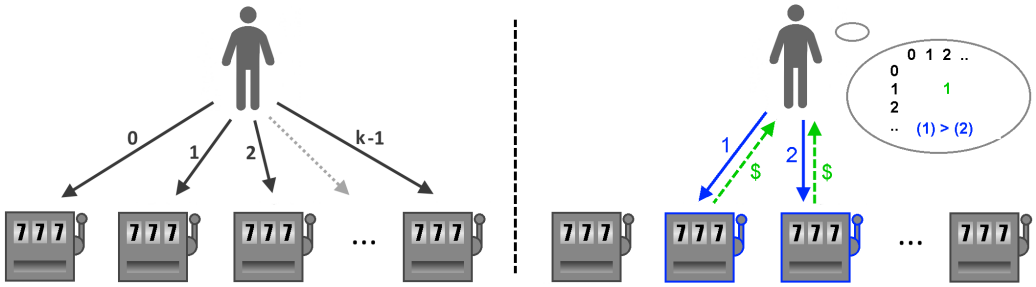

Figure 2.3.: Visualization of the k-armed dueling bandit problem

Preference-based machine learning aims to capitalize on this by shifting the focus from evaluating a single action to evaluating two actions a_1 and a_2 at once and regardless of whether this evaluation interally uses a numerical score, retaining only information about which of both actions has turned out to be preferable [5]. In our case, we will define a preference of a_1 over a_2 given availability of a heuristic h as $h(a_1) > h(a_2)$, a preference of a_2 over a_1 as $h(a_1) < h(a_2)$ and finally, an equality of a_1 and a_2 if $h(a_1) = h(a_2)$. Note that the heuristic in theory applies to *transitions*, not states; however, given that we operate in a deterministic setting, the same action executed in the same state will always yield the same successor state and as such, it is possible to evaluate a state and assign the knowledge obtained to the action whose corresponding transition resulted in that state.

Unlike the ordering implied by a memorized score, a *preference* does not make any statements about the quality of either a_1 or a_2 compared to any other available action - if there was a third available action a_3, it could be better than both, better than either a_1 or a_2 or worse than both, but to actually assess this, two additional preferences would have to be formed, comparing a_3 to a_1 and then to a_2 separately. As such, the number of preferences to be formed rises quadratically with the number of possible actions, which can be considered a drawback of the preference based method as more memory has to be allocated to keeping track of the results, and a single update will carry less information than with UCT (see chapter 5 for a more in-depth look at this drawback).

Relative UCB

The result of opting for a preference-based approach in a *bandit* scenario is the *dueling k-armed bandit problem*, where two actions are being tested in a single iteration. *Relative UCB* (RUCB) [10] is a technique to apply the approach of upper confidence bounds to the dueling banding problem making use of preference based feedback. For every action, we now no longer store a score, but instead the results of direct comparison with any other action in the form of number

of direct *victories*. As expected, this results in a quadratic matrix $W = [w_{ij}] \in \mathbb{Q}^{k,k}$ with a number of rows and columns equal to the number of actions available, and an entry w_{ij} denoting the number of direct comparison victories of action i against action j, with draws (comparison between action i and action j returned equality) handled by adding 0.5 to both w_{ij} and w_{ji}.

This data can then be used to calculate upper confidence bounds in the same fashion as the UCT formula does for numeric feedback. Again, balancing exploration against exploitation of accumulated knowledge is desirable; the RUCB formula, much like UCT, is based on a sum of two terms, respectively accounting for exploitation and exploration. Where the influence of the exploration term is balanced by parameter C for the UCT formula (2.3), RUCB modifies the influence of the exploration term with a parameter α. It should be noted that adaption of RUCB for trees (as UCT (2.3) does for UCB1 (2.2)) can be done simply by changing the range of possible values of the already existing parameter α from $\alpha > 0.5$ to $\alpha > 0$ [6]. While structurally similar, one defining difference of RUCB compared to UCB/UCT is that the value obtained by the formula is associated with a pair of non-identical actions as opposed to a single action. This means that an ordering of actions according to their value is not feasible; instead, we are interested in the *set of condorcet winners* $\mathscr{C}$ defined as the set of all actions that show more than or equal victories against all other actions individually compared to losses. While the set of condorcet winners is defined on actual wins and losses, we still need to account for uncertainty in the form of confidence intervals and as such, in practice define $\mathscr{C}$ based on a matrix of values $u_{i,j}$ calculated as per the RUCB formula below (2.4):

$$u_{ij} = \frac{w_{ij}}{w_{ij} + w_{ji}} + \sqrt{\frac{\alpha \cdot ln(t)}{w_{ij} + w_{ji}}}, \tag{2.4}$$

where t is the number of experiments conducted on the dueling bandit problem at hand. This then results in the derived definition of the set of condorcet winners $\mathscr{C} = \{a_i \in A | \forall j \in \{0, 1, ..., k-1\} : u_{ij} \geq 0.5\}$ [6]. The process to select two actions to play the bandit problem with begins with attempting to select the first action based on the above set $\mathscr{C}$ of condorcet winners. If the set contains only one element, that element is chosen, while an empty set means no action has shown dominance based on the currently available data and any random action is chosen instead. Lastly, if $|\mathscr{C}| > 1$, an action is chosen at random from $\mathscr{C}$, with the caveat that an action that has once been the sole member of the set and subsequently appeared in all later iterations is given half the total weight, with the rest being distributed equally among the remaining actions.

Having obtained a first action for the selection process, the challenge is to select a good *second* action without relying on an ordering of the individual actions. As the process of selecting two actions aims to specifically compare how well one fares against the other, a promising challenger is chosen, again based on the RUCB formula, but this time selecting a_j so that $argmax_j(u_{ij})$, representing the action that scored best in direct comparison with the first selected action. The procedure as described in [6] is summarized in algorithm 1.

Algorithm 1: Procedure RUCB-SELECT

Data: Node n with associated state s and matrix $W = [w_{ij}] \in \mathbb{Q}^{k,k}$
Result: Pair of actions (a_i, a_j)

Calculate values $u_{i,j}$ based on formula 2.4 with the information from W
$\mathscr{C} := \{a_i \in A(s) | \forall j \in \{0, 1, ..., k-1\} : u_{ij} \geq 0.5\}$
if $|\mathscr{C}| = 0$ **then**
    select a_i as random action $a \in A(s)$
else if $|\mathscr{C}| = 1$ **then**
    select a_i by choosing the only element $c \in \mathscr{C}$
else
    if *Action b once was single element of $\mathscr{C}$ in previous iteration and $b \in \mathscr{C}$ now* **then**
        adjust weight of $b \in \mathscr{C}$ to be half of total
        adjust weight of all other actions $d \in \mathscr{C}, d \neq b$ to share remaining half of total
    select a_i as random action $c \in \mathscr{C}$ using above weights if applicable
select a_j using $argmax_j(u_{ij})$
return (a_i, a_j)

2.3 Monte Carlo Tree Search (MCTS)

The following subsections will provide a short description of basic MCTS and highlight the differences between the two variants of MCTS that will be used for the experiments in chapter 4.

2.3.1 Why Monte Carlo?

In many practical applications, approximations of theoretical values are sufficiently precise and can be obtained with relative ease compared to exact solutions. In some cases, such as irrational numbers, it would not be possible to store the theoretical value in a binary data format at all anyways. Monte-Carlo methods apply in situations where an approximation can be continually refined by providing more time and processing power (and thus, iterations), with each step gradually improving the approximation and as such allowing termination at any time to obtain a value that approaches the theoretical value, with the error diminishing in magnitude over time. A frequent application for this method in mathematics is approximating integrals [4].

Extending this to game tree search and thus AI has the benefit of allowing to obtain a decent estimate in reasonable time that is likely to either be optimal or at least usable, as opposed to having to rely on exact solvers that may not find a solution in the external constraints given, such as available time or memory. in Comparison, an exact solver, like for instance the A*-algorithm [9] for solving pathfinding problems, must run to completion before any assumption can be made about the result, and after completion, optimality is guaranteed. For many practical applications such as those cited above as well as artificial intelligence in games, an optimal result is neither realistically obtainable nor always required over a decent approximation while external constraints (such as computational budget) may be strict. The defining property of a

Monte Carlo method is random sampling (thus simulation).

2.3.2 The MCTS algorithm

A game as defined in the beginning of chapter 2 can be seen as a space of possible states, obtained by executing legal actions within a state to arrive at a successor state. In a deterministic scenario, we can rely on a transition function $f : SxA \rightarrow S$ to generate the single successor state corresponding to a state-action pair. Because the player agent will at any point in time know the current state of the game board, this state can be seen as a root state from which a tree of successor states can be spanned. Notably, each state on its own constitutes a k-armed bandit problem; as such, the resulting tree is a tree of k-armed bandit problems.

Monte Carlo Tree Search (MCTS) is a method to obtain a single promising action given a state; note that this requires the algorithm to be ran again everytime an agent relying on MCTS is prompted to execute an action on the game board. This action is obtained by solving the k-armed bandit problem in the root state of the tree spanned by the algorithm; the rest of the tree contributes to this informed choice of action by providing increasingly solid guidance before having to rely on the name-giving Monte Carlo random simulation. Note that the tree spanned during successive iterations of one single MCTS run usually makes up only a small part of the entire theoretical game tree spanning from the root state. Within the context of MCTS, root state denotes the state that the MCTS algorithm was started with; this does not equal the root state of the game itself except for the very first execution of MCTS over the course of a game. The following sections will provide a non-exhaustive summary of MCTS where relevant for the scope of this thesis and as presented in [3] and [6].

MCTS can be divided into iterations, which in turn consist of four phases (selection, expansion, simulation, backpropagation). Iterations are repeated until the algorithm is terminated, for example due to running out of resources such as time. Within the scope of this thesis, we assume this resource to be *state advances*, e.g. operations of copying a game state and progressing it by executing one action. An iteration will always be completed in full, e.g. the four phases listed above will always be executed consecutively. It should be noted that for deterministic domains such as the one used in later experiments, all *game states* associated with a single *tree node* are semantically equivalent.

Selection: Making use of the tree policy

The *tree policy* essentially sums up the information about the estimated quality of actions gained in all previous iterations since the algorithm has been freshly started. Initially, no knowledge is available, and as such, the selection step simply returns the root node. If there is information available from previous iterations, however, MCTS is designed to make use of it, while at the same time not relying too much on this very information - this is where the *exploration vs exploitation* problem must be taken care of. As we have seen in chapter 2, both UCT and RUCB offer solutions for this problem for the *k-armed bandit problem* and the *dueling k-armed bandit problem*, respectively.

The selection step aims to select one or more actions (depending on the algorithm variant, with basic MCTS selecting just one action) within a node, starting at the root node of the tree. The tree is then traversed by following these actions to their respective successor states, and if the resulting nodes are already expanded in full (which means all their successor nodes are already part of the known tree described by the tree policy), the selection continues to step further down the tree. For basic MCTS, since exactly one action is chosen in each node, the selection step will form a path within the tree, starting from the root node.

Expansion: Growing the tree

After finishing the selection step, we have obtained one or more (depending on the algorithm) nodes that have successor states not yet part of our tree. We then add one or more nodes (again, depending on the variant) to our known tree, with the basic variant of MCTS adding exactly one node to the tree during this step.

Simulation: The Monte Carlo method

The simulation step is where the "Monte Carlo" part of the name of MCTS comes to bear. We are interested in obtaining a quality estimate of the node(s) we have arrived at, but in the most basic form of the algorithm, we only know about the rules of the game, but have no way of evaluating an intermediate game state that does not directly constitute a victory or loss for one player (or a draw). A single simulation starting from a specific node is executing by continuously selecting actions according to the *rollout policy*, which may be entirely random (and is used as such within this thesis), but does not necessarily have to be. A simulation stops either when a game state offers no further actions to be taken (game over), or in the case of the algorithm variants discussed later, a maximum number of actions have been chosen. In the latter case, a *heuristic function* is necessary to evaluate these intermediate states. Simulations are started from newly expanded nodes.

Backpropagation: Updating the tree policy

After conducting the simulation(s), the information is passed back through the tree until it eventually arrives at the root node. All nodes traversed in this process are then being updated according to the simulation results. This constitutes an update of the *k-armed bandit* (or alternatively *dueling k-armed bandit*) associated with each concerned node. As such, the *tree policy* is updated with new knowledge, which successive iterations (starting at the selection step again) can then make use of.

2.3.3 Algorithm Properties

- **Anytime**: After each iteration consisting of above four steps, MCTS can either commit to another (full) iteration, or terminate according to a given criterion. If the selection step makes use of techniques known to converge, such as UCT [7], a larger number of iterations will result in a better quality estimate and as such, a higher likelihood that the

action chosen in the root node is the *best* choice. In contrast, an *exact solver* will demand as much processing power or memory as it needs, but will guarantee an optimal solution when it terminates (and returns no solution at all, not even an estimate, when forcefully terminated). In realtime scenarios, the *anytime* property is valuable because it can be determined up-front when a solution is needed at the latest, as long as that solution is not necessarily required to be optimal.

- **Convergence**: As listed above, if the tree policy is chosen according to certain criteria, the tree will converge towards a minimax tree **mcts-survey**, which knows the optimal choice in each node of the tree, if the algorithm is given infinite resources. Making use of UCT or RUCB as tree policy will fulfill this criterion.

- **Asymmetric tree growth**: Generally speaking, it is worth exploring more promising sections of the game tree more thoroughly. Simply conducting a breath-first search would not achieve this goal and may not be feasible for any sufficiently complex domain. Depth-first search on the other hand may not be viable either and usually is not. MCTS will gather knowledge (described by the *tree policy*) according to the *exploration vs exploitation* trade-off and as such, grow a game tree that is more likely to carry meaningful semantic value.

- **Domain independence**: If the simulation step is implemented without a limiter of the maximum number of actions taken in a single simluation rollout, the algorithm only needs to evaluate states that do not offer any available actions anymore, and thus only needs to know whether a terminal state is (one of the) goal states or not. This property is forfeited by both MCTS variants discussed later, but nevertheless the basic MCTS algorithm can be applied to any domain where only the essential rules are known beforehand.

Realtime MCTS (RT-MCTS)

The first variant which is going to be applied to the testing domain is *realtime MCTS*, which equals the basic version of the algorithm with the addition of allowing the termination of rollouts (simulations) after a set number of actions have been executed, thus requiring a domain-specific heuristic to evaluate these states. It should be noted that this already negates the previously listed domain independence property.
The selection step will select one action in each layer of the tree according to the UCT formula, playing the k-armed bandit in that node. As such, it can be defined as selecting action a_i with $argmax_i(UCT_i) = argmax_i(\overline{X}_i + 2C\sqrt{\frac{ln(n)}{n_i}})$ in each node. After expansion, a single simulation will occur in each iteration, and the result used to update the tree policy in every node on the path back to the root node as with standard MCTS.

Relative MCTS with one-back propagation (ROB-MCTS)

Relative MCTS has a couple of defining differences compared to realtime MCTS, although both share the possibility of terminating simulations before a terminal state has been reached and as such, it requires the use of a heuristic function in some cases, too. The first major difference occurs in the selection step already: Because each node is now treated as a *dueling k-armed*

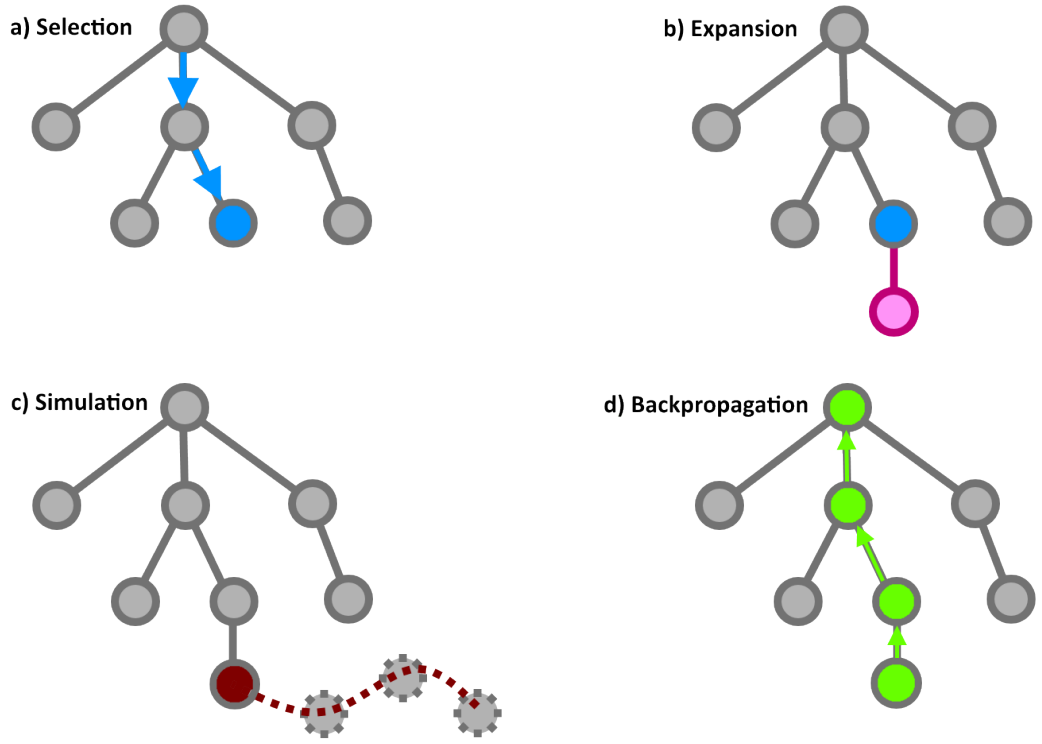

Figure 2.4.: Visualization of realtime MCTS

bandit, the tree policy now relies on RUCB instead of UCT, selecting two actions in each node as described in chapter 2. The result of the selection phase is thus a subtree of the tree defined by the tree policy.

Note that in this variant of MCTS, if a node has only one unexpanded child, we have to select that node (to expand the last missing child and not miss out on an unexplored but potentially good action as per exploration vs exploitation tradeoff), but also provide an action to compare the action that leads to the newly expanded node to, which requires the selection step to additionally go further down the tree.

A major part of relative MCTS is that multiple simulations occur in a single iteration, one starting from each newly added node. These simulations do not have to begin at the same depth level and generally will not due to asymmetric tree growth as is innate to MCTS. Once this is done, results are being propagated back through to the root node, but in the case of filtered one-back propagation as used here, with the additional twist that each node, after receiving two simulation results, will only propagate *one* of them according to its *return policy*. A return policy can be seen as a filter being applied in each node to determine which information is allowed to affect parent nodes; if the return policy is not defined as allowing all results to pass, parent nodes will thus receive less information, but potentially of higher quality. The return policy used for the algorithm in later experiments is to simply filter out the loser of the direct comparison after playing the dueling k-armed bandit and updating matrix W.

Relative MCTS in its basic form would not necessarily have to filter any simulation results, propagating all of them and thus allowing for more and more comparisons to take place (always comparing each element coming from one child to each element propagated by the second child). While in theory, this allows for more information to be collected during a single itera-

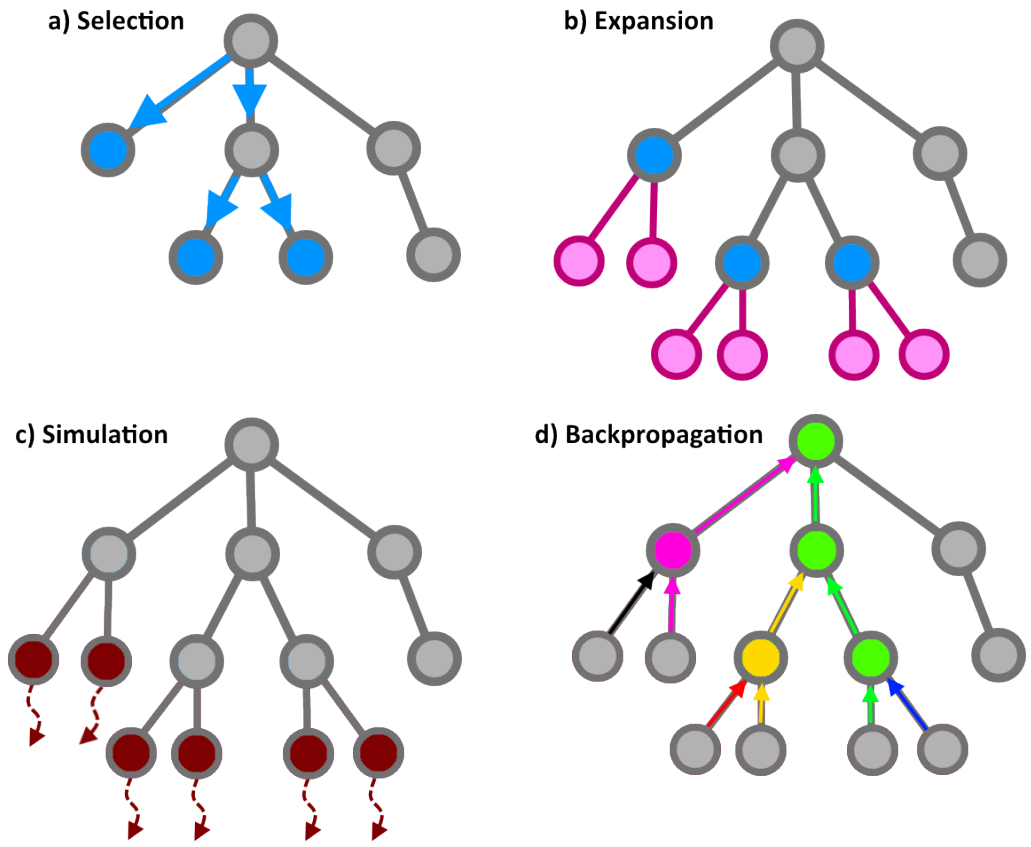

Figure 2.5.: Visualization of relative MCTS with one-back propagation

tion, this does not necessarily have to be advantageous in practice. In fact, filtering simulation results during backpropagation will become important when adapting the algorithm variants to multiplayer, for reasons that will be explained later.

Multiplayer game trees

One distinct feature of multiplayer games over generalized tree search problems is that there are multiple interests involved - in a zero-sum game, these interests will aim to actively work against each other. In this context, it is helpful to talk of *plies* as opposed to *turns* or *moves* - a *ply* in a combinatorial two-player game is a turn taken by one of the players, while *move* is sometimes ambiguously used to describe an entire round of actions, sometimes using *half-move* to refer to a ply (as is the case in chess).

A *game tree* for a *combinatorial* game is in essence a directed graph with each node representing a game state and each edge connecting two nodes n_1 and n_2 in one direction representing an action $a \in A(n_1)$ so that $transition(s_1, a) = s_2$ (in this case, a node is associated with one state). The root of a game tree is the initial state $s_0 \in S$ of the game, whereas the total number of leaves (nodes n so that $deg^+(n) = 0$) in a combinatorial game amounts to the total possible unique sequences of actions that can be conducted. It should be noted that game trees quickly grow to prohibitive sizes even for games of medium complexity, which is one of the motivations for using MCTS in game AI. As described earlier, MCTS does not commit to a full breath-first

search, but rather grows a tree asymmetrically, thus spanning a sub-tree or *partial game tree*.

Such a game tree can be seen as split into layers, where all nodes sharing a specific integer distance to the root node (measured in the number of actions necessary to each this state from the root node) belong to the same layer. Each layer corresponds to a ply - whose turn it is in a specific node is the same for all nodes of that layer if the action order of the game is sequential. In the two-player domain referenced to later, the turn order is alternating, which is a special case of a game with any number of players where the turn order is circular. For such a game with circular turn order, we can build on the definition in subsection 2.1.1 to define $player - index(s_2)) = (player - index(s_1) + 1) mod(n)$ for a successor state s_2 of s_1, with n total players and $p_0 \in P$ (player with index 0) to begin the game. Figure 2.6 shows the first four plies and associated players for an example game tree with $A(s) = 2$ for every $s \in S$, specifically for $n = 2$ and $n = 3$ number of players.

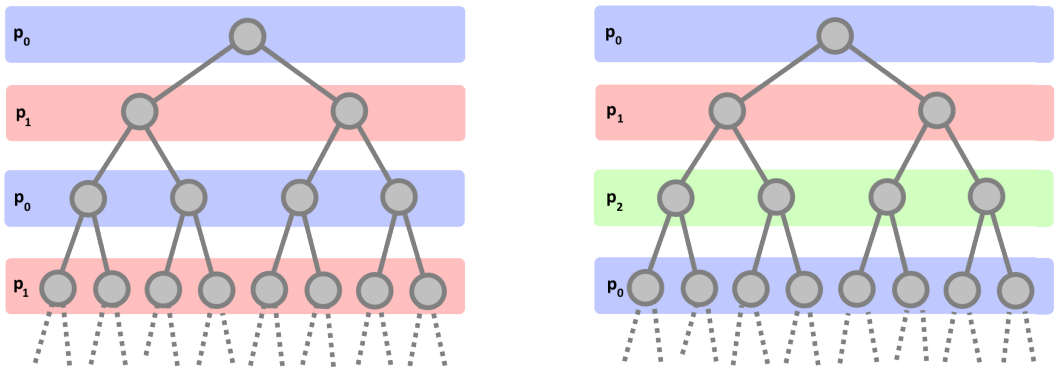

Figure 2.6.: The first four game tree layers for an example game with 2 (left) and 3 (right) players (circular turn order). The experiments in later chapters will be based on a two-player setting.

Adapting MCTS for multiplayer

While the domain highlighted in the next chapter is a two-player combinatorial game, we would want the approach to be more general and allow any number of players as well as dropping the zero-sum requirement. It is suggested to adopt the general idea of the max^n approach [3]: the numerical reward value is replaced by an n-dimensional score vector, with n being the number of players. This corresponds to the vector $\mathbf{r}^n$ defined in subsection 2.1.1. How the individual components are calculated is a degree of freedom; however, it makes sense to pack all information a specific player is supposed to base their decisions on into the single value that corresponds to their associated component of the reward vector. In the two-player zero-sum case using UCT, this can be achieved by enforcing a range of $[0, 1]$ for all components as well as $\sum_{i=0}^{n-1} r_i = 1$ (because the individual components range between 0 and 1 instead of -1 and 1, *zero-sum* is not to be taken literally in this case).

Adapting UCT can then be done by calculating the upper confidence bound based only on the corresponding component of the reward vector [3], which can be determined based on the player p whose turn it is in that node. Building on the defintion of UCT (see subsection 2.2.1)

and assuming that the result of the play j of action i is still encoded as $X_{i,j}$, but now delivered in vector form, this results in the multiplayer UCT formula as follows:

$$UCT_i = \sum_{j=0}^{n_i} \frac{(\mathbf{X}_{i,j})_p}{n_i} + 2C\sqrt{\frac{ln(n)}{n_i}} \tag{2.5}$$

$(\mathbf{X}_{i,j})_p$ refers to the component of the vector corresponding to the particular player in question as described above. In a similar fashion, when relying the preference-based approach of RUCB, we can conduct our comparisons based only on the corresponding component of both result vectors. Assuming that two reward vectors $\mathbf{p}$ and $\mathbf{r}$ are to be compared in a node, associated with actions a and b respectively, and s is the unique state corresponding to the node, we first retrieve the corresponding index i with $player-index(s)$ and then form a preference $P(a,b) = a$ if $p_i > r_i$, $P(a,b) = b$ if $p_i < r_i$ and a non-preference otherwise. The result of that comparison is then used as before to update matrix W (see subsection 2.2.2).

An interesting aspect of relative MCTS is that the *return policy* is a degree of freedom in the configuration of the algorithm. The basic approach is a return policy of propagating all simulation results unfiltered, thus increasing the number of comparisons as the backpropagation phase of relative MCTS approaches the root node and in every node, comparing each member of the set of results propagated by the first action with each member of the set of results propagated by the second action. The implementation from [6] which the multiplayer environment used for the later experiments is built upon already includes other return policies which have a common property of only allowing one simulation result to be propagated further back. This means that each node will receive two results, compare them, and propagate only one according to its return policy. As such, the number of comparisons in each node does not increase and is always one. While this drastically reduces the information gained in each iteration of relative MCTS, the assumption is that for some domains, it can be useful to limit information this way.

In fact, as preliminary experiments have shown, this is the case for two-player Connect Four, and it is possible that this would hold true for other multiplayer zero-sum games as well. The reasoning is that if the *return policy* is chosen so that only the *winner* of the direct comparison in a node is propagated further, this results in a backpropagation process that is semantically sound. The simulation itself, in this case based on a random rollout policy, is not guaranteed or even likely to represent a semantically interesting game (this is the case for both single- and multiplayer problems, as random choice of action is rarely the best possible strategy). However, when propagating results back through the known tree where the selection phase is governed by the non-random *tree policy*, it may make sense to cut backpropagation paths to the root node that are unlikely to correspond to an actual game being played. When a node filters the result that it would deem unfavorable in direct comparison, then its parent node cannot be misled by involving that specific result in its own comparison.

For instance, consider a crafted scenario where in each state, a player can either move left or right ($A = left, right$ and $A(s) = A \forall s \in S$). Additionally assume that p_0 will always wish to go left, while p_1 will always prefer to go right, and that the choice of action in the first few states is of great significance for the evaluation of any state further down. In that case, it would be unlikely to observe a game where left or right were chosen twice in a row (in two

successive *plies*), assuming that both players aim to play according to their own best interest. A backpropagation path that takes such a route back to the root node would then potentially confuse the decision making process in that node. Essentially, a return policy such as only allowing the winner of the comparison to proceed in the backpropagation step of relative MCTS has the potential to make the propagated results more relevant, at the cost of reducing the number of comparisons in each node to one.

3 The Connect Four Domain

Connect Four is a *combinatorial* game and as such, *discrete*, *deterministic*, *zero-sum*, *sequential* and *perfect information*. These properties make it lend itself well to serving as a multiplayer domain for AI research in general and MCTS methods specifically, as there a manageable yet nontrivial number of possible actions in each state (at most seven, minus one for each filled column) and a simple ruleset, but the resulting game tree can still be rather complex. In accordance with the formal definition found in [3] and as summarized in chapter 2 for a game with these properties, *Connect Four* can be described as follows:

- **Game board and states:** The board is a 7-column, 6-row matrix $B = T^{7x6}$, where $T = \{empty\} \cup \{token_p : p \in P\}$ denotes the possible values (player tokens or empty space) a single entry may assume, effectively giving each player a set of tokens sharing a color or similar identifier that does not discriminate between tokens of the same player, but does identify them belonging to one unique player. S is the set of all possible board configurations that can be reached from the initial state; for every possible $s \in S$, there is exactly one corresponding game board configuration, which serves as an exhaustive description of that state instance (as all necessary other information, such as which player has last executed an action and thus who is next in line can be determined from the board alone in a two-player scenario). Each *instance* of a game board represents a unique *state*. An upper bound for the number of possible states in a two-player game of *Connect Four* is thus $|S| \leq |T|^{7 \cdot 6} = 3^{42}$, however this number is not accurate as it contains invalid states, such as those with empty fields below one or more player tokens in the same column.

- **Initial state $s_0 \in S$:** Before the first action is executed, the game board will only consist of empty fields, and as such is represented by $empty^{7x6}$.

- **Actions**: An agent may choose between one of up to seven columns, resulting in an action set $A = a_0, ..., a_6$, with each element corresponding to the respective column index $i \in \{0, ..., 6\}$. Actions can be assumed to be mapped to columns via a function $target - column : A \rightarrow \{0, ..., 6\}$ that maps any action a_i to column i for formal modeling purposes.

- **State transition function:** $transition : SxA \rightarrow S$: Assume the board is represented by the state $s_{current} = b_{i,j}$ with $i \in \{0, ..., 6\}, j \in \{0, ..., 5\}$ and the agent $p = p(s_{current}) \in P$ chooses an action $a_i \in A$. In addition, for $i \in \{0, ..., 6\}$ and $j, k, l \in \{0, ..., 5\}$ we define

$$target - depth(i) = \begin{cases} j, & \text{if } \forall k > j : b_{i,k} \neq empty \text{ and } \forall l <= j : b_{i,l} = empty \\ -1, & \text{otherwise} \end{cases}$$

Then the game board $c_{i,j}$ with $i \in 0,...,6$ and $j \in 0,...,6$ of the corresponding successor state $s_{next} = transition(s,a)$ following an execution of action $a \in A(s)$ with the game board of s stored in $b_{i,j}$ is determined unambiguously by

$$c_{i,j} = \begin{cases} token_{player(s_{next})}, & \text{if } i = target-column(a) \text{ and } j = target-depth(i) \\ b_{i,j}, & \text{otherwise} \end{cases}$$

.

- **Possible actions** $A(s) \subseteq A$ in a state $s \in S$: Any incomplete column can be played. A column i is *incomplete* if and only if $target-depth(i) \neq -1$, effectively requiring the column to still have at least one free space.

3.1 Terminal states and win conditions

Above definition shows that a game is completed (it has arrived at a *terminal state*) after 42 turns at the latest, because at that point all columns are stacked (*complete*) and no further free space is available, thus the game state cannot advance anymore. This alone would not make for a very interesting game yet, as connect four is supposed to provide players with a means of achieving victory (and victory can be achieved earlier than 42 turns into the game). Namely, the win condition $victory_p(s)$ for any specific player $p \in P$ in a state $s \in S$ is to have an uninterrupted vertical, horizontal or diagonal group of four tokens (subsequently referred to as series of four) belonging to that player on the field.

It is easy to see that either exactly one player emerges as victorious within 42 turns, or the game ends in a draw after exactly 42 turns; in both cases, the game being played can be described as a series of states and actions $(s_0,a_0),(s_1,a_1),...,(s_t)$ where $\forall i > 0 : s_i = transition(s_{i-1},a_{i-1})$, $terminal(s_t)$ and $\forall i < t : \neg terminal(s_i)$. In the latter case, no player has managed to meet the win condition before all columns are complete and thus offer no further possibility of generating a series of four tokens associated with any player. In the former case, victory must have been achieved by the last action executed on the game board filling a previously empty space with a token that connects to three other tokens of the same player allegiance, forming a series of four. In this case, a victor has been determined, and the game ends regardless of whether all columns are *complete* or not.

However, since for any given state $s \in S$, each action $a \in A(s)$ will only add one token to the game board, and executing an action $a \in A(s)$ in a state $s \in S$ implies that s itself did not constitute a *terminal state* (thus all token groups amount to a size of three or less and there is at least one *incomplete* column), it is not possible for *more* than one player p to meet the win condition $win_p(s_{next})$ with $s_{next} = transition(s,a)$ for any $a \in A(s)$ and $p \in P$. To summarize, each state $s \in S$ is either a *nonterminal state* with $\forall p \in P : \neg victory_p(s)$ and $|(A(s)| > 0$, or a terminal state if the previous conditions are not met. A *terminal state* will be associated with either exactly one winner or none at all (ending the game in a draw, in which case $|(\mathrm{A(s)}| = 0$).

3.2 Evaluating terminal and non-terminal states

Plain Monte Carlo Tree Search does not require anything other than evaluating terminal states, given that a rollout will only end when no further action can be executed. However, both major MCTS variants in this thesis make use of numerical feedback in one way or the other - either for direct use as a numerical reward in the UCT formula in the case of realtime MCTS, or to form a preference (and retain only the preference) in the case of preference-based MCTS. In general, it would be sufficient for the evaluation of terminal states to assign (from the perspective of a specific player) a value of one to a victory, zero to a loss, and 0.5 to a game-over state that ends in a draw.

However, for preference-based MCTS in particular, it can be advantageous to form a preference concerning two different game-over states that both carry the same end result for a particular player: usually, a more immediate, less distant victory is better than a victory far into the future (especially considering that when using a random rollout policy as is the case here, a long simulation trajectory will inherently be less reliable and is less likely to represent a realistic game). Similarily, a loss in the distant future is less grave than an immediately threatening loss. A draw will always only occur when the entire game board is filled with tokens, and as such, can always be treated the same way. For all other (non-terminal) cases, which may occur when limiting the simulation length, a heuristic must then be used. As such, we can estabilish this basic evaluation function for a specific player $p \in P$ given a heuristic h_p and for all states $s \in S$, where $d(s)$ is the distance in game turns from the initial game state, and $\varepsilon > 0$:

$$evaluate_p(s) = \begin{cases} 1.0 - \varepsilon \cdot d(s), & \text{if } victory_p(s) \\ 0.0 + \varepsilon \cdot d(s), & \text{if } victory_q(s) \text{ for any player } q \in P, q \neq p \\ 0.5, & \text{if } terminal(s) \text{ and } \neg victory_q(s) \text{ for all players } q \in P \\ h_p(s), & \text{otherwise} \end{cases}$$

.
The value of $\varepsilon > 0$ should be chosen small enough so as to only tie-break preferences, but not alter the preferences imposed by the heuristic function where it is required.

3.2.1 Heuristics for non-terminal states

As noted in chapter 2, the difference between realtime MCTS and plain MCTS is that simulations can be stopped once the chain of actions taken according to the rollout policy hits a maximum value, even if a terminal state has not yet been reached at that point. This property of realtime MCTS also applies to the variant of relative MCTS used for later experiments. As such, heuristics for intermediate states of the *Connect Four* are required. It should be noted that it is generally nontrivial to find good or even just usable heuristics for some domains, including most multi-player games. We will later highlight this problem in the specific case of *Connect Four* in chapter 5.

For the creation of heuristics for the experiments, two components have been drafted, based on relatively simple ideas that supposedly carry semantic value for this particular domain. Because

UCT requires values to be within the $[0, 1]$ range, we make liberal use of generous upper bounds to ensure this, which will however lead to values averaging closely around 0.5. While this does not endanger the convergence property of UCT, it reduces convergence speed. This problem will be tackled later on in chapter 4. Because the domain at hand is a two-player combinatorial game, we also want to make use of the zero-sum property, even though the multiplayer MCTS implementation would not necessarily require this to be the case.

- **Component A:** Victory in Connect Four requires a horizontal, vertical or diagonal line of four tokens belonging to the same player. As such, assuming a $7*6$ game board, there are a total of 69 theoretically possible victory combinations for each player: $6 \cdot 4$ horizontal rows, $7 \cdot 3$ vertical rows, and $4 \cdot 3 \cdot 2$ diagonal rows. A value between 0 and 1, centered around 0.5, can then be created by summing up all *remaining* theoretical victory combinations for the player in question and subtracting the same value calculated for the other player, then dividing that value by $(69 \cdot 2)$ and adding 0.5. A *remaining theoretical victory combination* is a horizontal, vertical or diagonal series of four game board tiles that all either contain a token belonging to that player, or are empty. Component A is stand-alone and as such can be used to form a heuristic on its own.
- **Component B:** We now traverse all 42 tiles of the game board. If a specific tile is empty, we check if placing a *virtual* token at this position would form a series of four for either player (disregarding if that would correspond to a valid game move), which means that a series of three with at least one free end position was already in place. Such a series of three is known as a threat; because generally, the player that begins the game will prefer *even threats* (with a free end point in the lowest row or going up multiples of two from there) and the other player preferring *odd threats* even threats are valued double for the first player (and odd threats double for the second player). A generous upper bound (that will never be reached) is thus 84 for the threat score of a specific player (42 game board tiles, times two). Component b is then formed by subtracting the opponent threat score from the player's threat score, dividing the result by 84 and adding 0.5, in a similar fashion as this was done for component a. This component however evaluates all states that do not have a threat for either player (which is the case for most early states) as 0.5 and as such, it should only be used as an additional component a composite heuristic.

Because of the generous upper bounds used in above calculations, the values will close to 0.5 in both cases. To better visualize this and help find a solution, the values obtained in practice by using component A and B as isolated heuristics have been logged over the course of 100 games, where the algorithms in play were not of particular importance but instead the focus was on obtaining data for the values of components A and B as they occur in practice (every state was evaluated on creation). These values (ranged between 0 and 1) have then been binned into 101 bins of width 0.01 and plotted in figure 3.1 for component A and figure 3.2 for component B. It is immediately apparent that the value distribution is closely centered around the neutral midpoint at 0.5 as predicted.

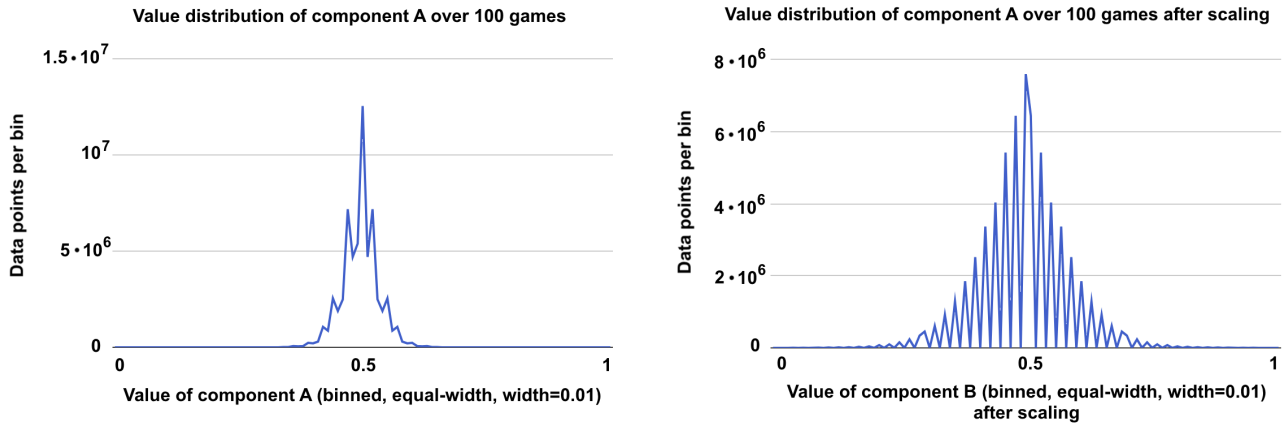

Figure 3.1.: Value distribution over 101 equal-width bins for component A before (left) and after (right) scaling

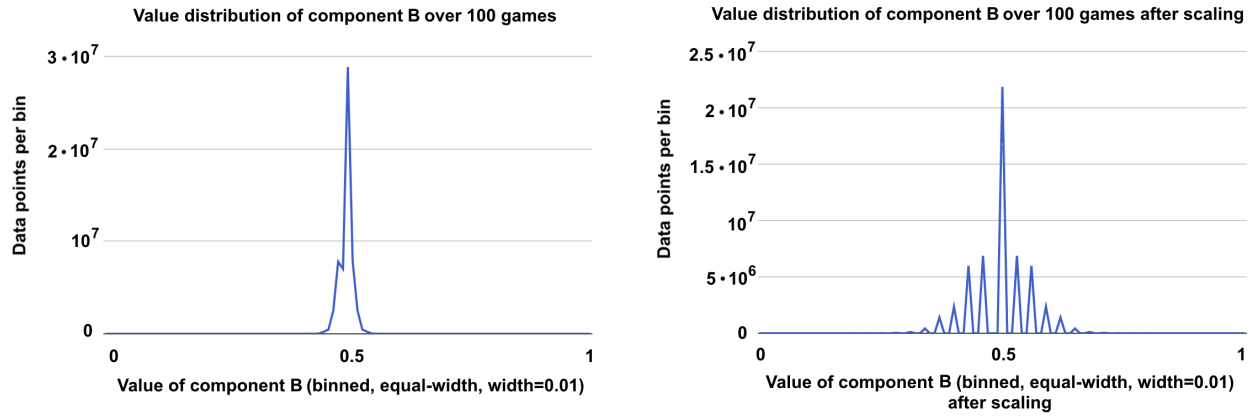

Figure 3.2.: Value distribution over 101 equal-width bins for component B before (left) and after (right) scaling

The process to increase the spread across the permissible range (and thus help UCT converge faster) is as follows: From the above data and for each of these components, the minimum value val_{min} and maximum value val_{max} are extracted. Recall that in chapter 3 we have defined terminal loss states to at maximum evaluate at $42 \cdot \epsilon$ and win states to at minimum evaluate to $1 - 42 \cdot \epsilon$. Given that terminal states are supposed to always be valued lower (in the case of losing terminals in the perspective of a specific player) or higher (for winning terminals) than intermediate states, the values for components A and B are ultimately supposed to be within the range of $(42 \cdot \epsilon,\ 1 - 42 \cdot \epsilon)$. We can then calculate a *scaling factor* $x_{scaling}$ for components A and B in the following way: First, the larger difference from 0.5 given both extreme obtained values is calculated ($diff = max(|val_{max} - 0.5|, |val_{min} - 0.5|)$. Then we determine $x_{scaling}$ so that $diff * x = 0.5 - 42 \cdot \epsilon$. The resulting values based on above simulation are $x_a = 2.63$ for component a and $x_b = 5.2$ for component b. From there, we obtain $a_{scaled} = (a - 0.5) \cdot x_b + 0.5$ and $b_{scaled} = (a - 0.5) \cdot x_a + 0.5$. Lastly, in the (unlikely) case that these values exceed our desired range of $(42 \cdot \epsilon, 1 - 42 \cdot \epsilon)$ ($\epsilon = 0.001$ was used in this case), the values are capped, however this rarely occurs in practice at all.

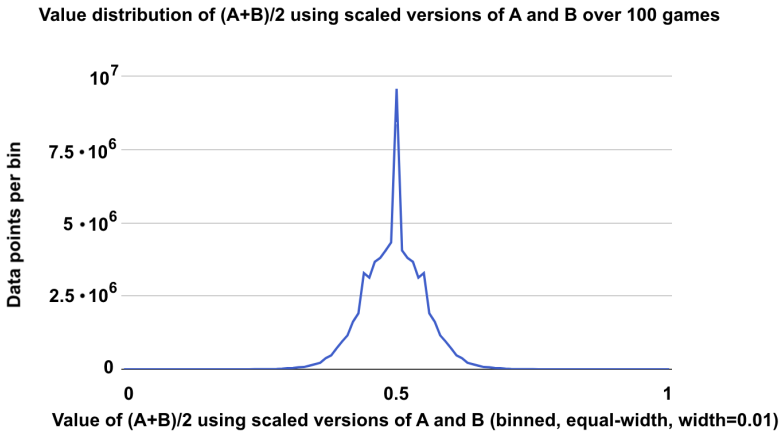

Figure 3.3.: Value distribution over 101 equal-width bins for the composite heuristic (A+B)/2, using the scaled components

We then define the heuristic h_{simple} to merely use the value of component A, and $h_{composite}$ as the arithmetic mean of components A and B. A reward vector is then formed by calculating the heuristic value from the perspective of each player and assigning the i-th vector component to the value obtained for player p_i. Note that other ways of combining the two basic components (whereas component B should not be used on its own) are possible, however the arithmetic mean without additional weighting of either component featured the least empty bins and was thus chosen. The value distribution of $h_{composite}$ as seen in figure 3.3 covers a wide range of non-empty bins and thus is more likely to not funnel most values into a few particular values, which was originally a problem given the discreteness and granularity of the domain (see figures 3.1 and 3.2, where the upscaling amplifies the effect).

4 Experiment setup

Recall that Connect Four is not necessarily a fair game (3); as such, for all following experiments, whenever two algorithm configurations play a series of games against each other, it must be ensured that the same number of games is then played with reversed colors (and as such, each algorithm begins the game exactly as often as it is being assigned the second move). Two computational resource budgets (10000 and $5 \cdot 10^4$ state advances) and two heuristics (h_{simple} and $h_{composite}$)have been tested; recall that a state advance is spent everytime a successor state is calculated, and that rollouts (during the simulation phase) thus make up for the majority of computational resource usage. Two algorithms were considered, namely realtime MCTS with UCT as well as relative MCTS using RUCB and backpropagation with a *return policy* that only permits the winning trajectory to be propagated further. Before these are pitted against each other in a multiplayer scenario, we first want to empirically determine optimal values for their parameters C and α, respectively.

4.1 Domain specific parameter tuning

In practice, the value chosen for the UCT parameter C depends on the domain at hand and is frequently determined and adjusted manually [3]. In this section, an optimal parameter for both UCT C (out of 10 equidistant values from 0.2 to 2, including) and RUCB α (out of 10 equidistant values from 0.1 to 1.0, including) on the domain at hand will be selected. Because a higher computational budget may mean that assigning a higher weight to exploration has beneficial effects, the optimal parameter will be determined independently for the two examined budgets of $1 \cdot 10^4$ and $5 \cdot 10^4$ state advances. In addition, fairness is ensured by testing each combination of two parameter values twice, with the algorithm configuration using the smaller value beginning the game 100 times, and configuration the algorithm using the larger value beginning the game for a second set of 100 games. The optimal parameter for one budget class is then determined by summing up the first and second series of 100 games for each combination of both parameters and selecting the parameter value that, all things considered, won the most games against all other parameter values. The rollout length across all tuning experiments was chosen to be 5, in order to approximate a setting with realistic numbers of terminal rollouts vs nonterminal rollouts.
The format of the following result tables is always the same: on the left, a table is shown with the entry (x,y) corresponding to how many times x won against y, and (y,x) showing the number of victories of y against x over the course of 100 games. In cases where these two values do not sum up to 100, the remainder were draws. The rows are then summed up to obtain the number of total victories a specific parameter value has scored against all others. This is then repeated by reversing the player index (after always allowing the smaller value to conduct the first move, the larger value gets the first move afterwards) and formatting the results in a new figure for the second series of 100 games between any two values. The total results are then summed up and plotted in a third figure, with the parameter value scoring the highest total number of victories being selected for the subsequent experiments.

v P0 | P1 >	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
0.2		72	81	71	76	79	70	82	84	85
0.4	21		74	71	72	76	82	77	81	84
0.6	16	12		82	71	80	80	77	82	90
0.8	24	10	10		67	60	66	68	71	76
1.0	21	12	17	20		46	60	68	68	50
1.2	13	10	16	27	32		48	53	57	61
1.4	20	6	7	23	28	37		52	58	49
1.6	12	8	16	19	26	36	34		41	47
1.8	11	5	9	12	26	29	34	45		48
2.0	10	8	7	13	35	31	41	40	42	

C	Wins
0.2	700
0.4	638
0.6	590
0.8	452
1.0	362
1.2	317
1.4	280
1.6	239
1.8	219
2.0	227

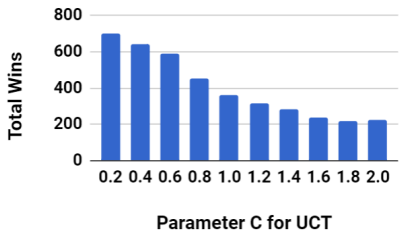

Figure 4.1.: 50k advances, values given are for UCT C. Configuration with smaller value begins the game.

v P0 | P1 >	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
0.2		21	34	27	31	37	38	38	46	38
0.4	42		15	33	31	50	48	50	63	63
0.6	24	60		15	41	43	62	67	61	75
0.8	31	47	68		31	47	44	61	54	57
1.0	27	39	38	55		46	47	49	58	64
1.2	20	25	31	33	35		43	51	55	53
1.4	17	18	18	33	31	37		50	51	64
1.6	12	21	18	25	35	25	33		54	55
1.8	14	20	24	26	26	30	31	34		49
2.0	12	20	19	28	24	25	25	32	34	

C	Wins
0.2	310
0.4	395
0.6	448
0.8	440
1.0	423
1.2	346
1.4	319
1.6	278
1.8	254
2.0	219

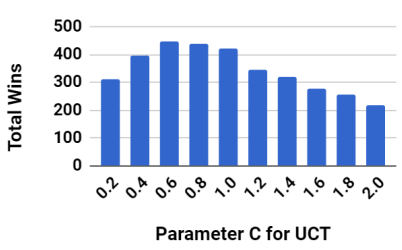

Figure 4.2.: 50k advances, values given are for UCT C. Configuration with larger value begins the game.

C	Wins
0.2	1010
0.4	1033
0.6	1038
0.8	892
1.0	785
1.2	663
1.4	599
1.6	517
1.8	473
2.0	446

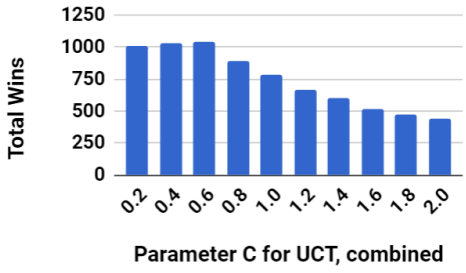

Figure 4.3.: 50k advances, combined wins for values of UCT C. The optimal value for C is 0.6

The results of parameter testing for UCT C and a budget of $5 \cdot 10^4$ advances is displayed in figures 4.1 and 4.2, the former containing the results for the algorithm configuration making use of the smaller parameter value beginning each game, and the latter for the reverse case of the larger of both values in a matchup being assigned the first ply. There is a visible difference between both approaches, showing that at a budget of $5 \cdot 10^4$ advances, beginning the game does indeed constitute a small advantage, given that the larger parameter values score more total wins if each series of 100 games between two UCT based MCTS variants has the variant with the larger value of C begin the game compared to vice versa. Where in the case of allowing the algorithm associated with the smaller value to begin the game, $C = 0.2$ scores the highest number of total wins, the best parameter values lie in the range of $C = 0.6$ to $C = 1.0$ for the reverse case. Summed up in figure 4.3 are the combined results of both tests, with a parameter value of $C = 0.6$ scoring the most wins and thus being selected for subsequent experiments for the UCT- based MCTS variant in the case of a budget of $5 \cdot 10^4$ advances.

v P0 | P1 >	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
0.2		47	51	58	56	67	68	67	73	70
0.4	47		50	65	63	61	70	70	67	86
0.6	35	42		57	48	54	66	66	70	62
0.8	32	22	34		47	54	51	59	55	61
1.0	38	31	35	36		47	44	48	41	48
1.2	26	32	33	37	38		51	41	49	45
1.4	25	21	24	40	45	41		48	50	50
1.6	28	24	23	30	39	39	38		42	49
1.8	22	26	10	30	46	36	31	46		43
2.0	21	9	25	27	35	37	41	36	43	

C	Wins
0.2	557
0.4	579
0.6	500
0.8	415
1.0	368
1.2	352
1.4	344
1.6	312
1.8	290
2.0	274

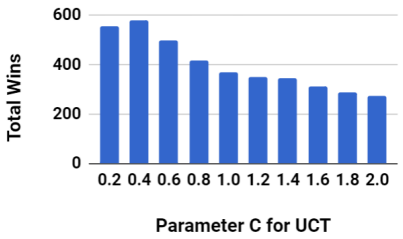

Figure 4.4.: 10k advances, values given are for UCT C. Configuration with smaller value begins the game.

v P0 | P1 >	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
0.2		38	43	46	54	55	60	54	66	55
0.4	51		33	52	58	54	48	78	72	63
0.6	42	52		43	51	55	63	64	65	71
0.8	31	35	45		52	58	52	65	59	61
1.0	29	30	37	37		65	60	61	65	56
1.2	30	41	32	27	35		48	51	56	50
1.4	23	32	29	37	30	41		46	44	50
1.6	26	16	27	27	29	33	40		46	38
1.8	16	19	26	32	27	39	44	41		48
2.0	26	26	20	27	24	32	40	40	39	

C	Wins
0.2	471
0.4	509
0.6	506
0.8	458
1.0	440
1.2	370
1.4	332
1.6	282
1.8	292
2.0	274

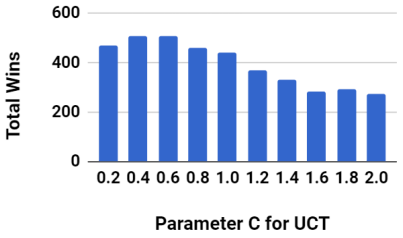

Figure 4.5.: 10k advances, values given are for UCT C. Configuration with larger value begins the game.

C	Wins
0.2	1028
0.4	1088
0.6	1006
0.8	873
1.0	808
1.2	722
1.4	676
1.6	594
1.8	582
2.0	548

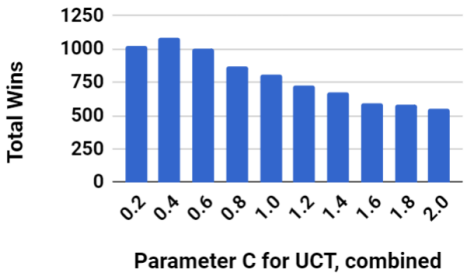

Figure 4.6.: 10k advances, combined wins for values of UCT C. The optimal value for C is 0.4

Figures 4.4 and 4.5 show the results of parameter testing for a budget of $1 \cdot 10^4$ advances with the smaller value beginning all matchups (4.4) and the same repeated, but with the algorithm configuration using the larger of two values in a matchup being assigned the first ply (4.5). The combined results are summarized in figure 4.6 for the C parameter of UCT and a budget of $1 \cdot 10^4$ advances. The optimal parameter within this setting is found to be $C = 0.4$, which is a comparatively small weight for the exploration term. However, the parameter appears to be relatively stable, given that the (discrete) results approximate a curve with a single global maximum. It is also notable that the difference between the two setups with assignment of the first ply to the smaller respectively larger value is not as visible as in the case of $5 \cdot 10^4$ advances. This may be because of generally more noise in the results as the computational budget of $1 \cdot 10^4$ advances is comparatively small. In addition, it should be noted that the optimal value for C is larger (0.6 vs 0.4) if the computational budget rises. This can be explained by a greater viability of exploration if the computational budget is more forgiving.

v P0 | P1 >	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.1		33	35	33	36	34	49	40	38	39
0.2	38		48	52	47	43	48	51	51	46
0.3	44	23		40	50	46	52	51	46	49
0.4	46	31	30		40	43	44	42	40	53
0.5	35	30	32	35		45	41	44	42	47
0.6	44	29	39	31	36		39	45	51	40
0.7	29	27	25	29	40	39		43	44	46
0.8	44	30	25	40	36	31	31		38	49
0.9	40	27	34	40	39	29	32	36		47
1.0	43	30	37	28	31	41	34	28	35	

α	Wins
0.1	337
0.2	424
0.3	401
0.4	369
0.5	351
0.6	354
0.7	322
0.8	324
0.9	324
1.0	307

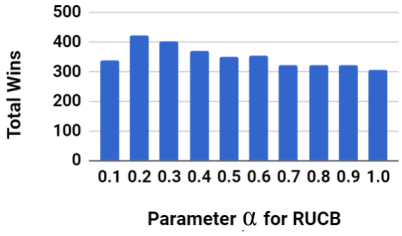

Figure 4.7.: 50k advances, values given are for RUCB α. Configuration with smaller value begins the game.

v P0 | P1 >	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.1		26	30	31	39	30	33	30	24	32
0.2	52		36	29	29	30	37	45	37	42
0.3	50	39		37	41	39	36	42	43	49
0.4	44	44	39		36	39	51	36	42	47
0.5	37	43	38	39		38	28	44	37	37
0.6	45	46	35	43	40		36	40	32	36
0.7	37	37	36	38	46	42		31	34	41
0.8	41	33	39	40	35	34	44		36	42
0.9	39	42	35	38	33	51	41	49		41
1.0	51	32	32	28	37	48	40	38	46	

α	Wins
0.1	275
0.2	337
0.3	376
0.4	378
0.5	341
0.6	353
0.7	342
0.8	344
0.9	369
1.0	352

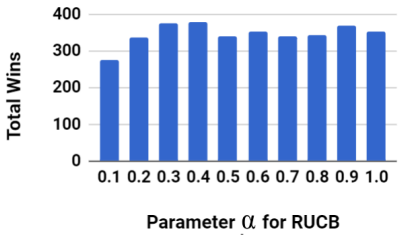

Figure 4.8.: 50k advances, values given are for RUCB α. Configuration with larger value begins the game.

α	Wins
0.1	612
0.2	761
0.3	777
0.4	747
0.5	692
0.6	707
0.7	664
0.8	668
0.9	693
1.0	659

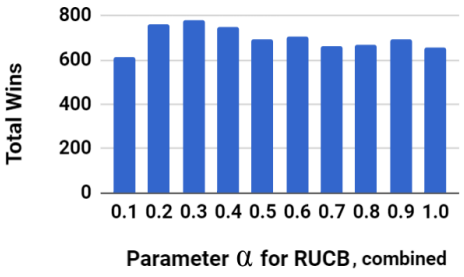

Figure 4.9.: 50k advances, combined wins for values of RUCB α. The optimal value for α is 0.3

After covering all cases for both budget sizes for the C parameter of UCT, the same must be repeated to determine an optimal value for the α parameter used by RUCB. The only constraint is $\alpha > 0$ [6]. For $5 \cdot 10^4$ advances, the results are displayed in figure 4.7 for allowing the configuration associated with the smaller of two values in a matchup to make the first move in the same fashion as it was done for UCT. Figure 4.8 shows the same results for the reverse case, with the combined results displayed in figure 4.9. Compared to the C parameter of UCT, the decision in favor of a specific parameter value is much less apparent; this is likely explained by the generally higher budget requirements of RUCB (see later explanations in chapter 5) and thus a more noisy outcome compared to UCT within the same budget (which would apply even moreso for a smaller computational budget). The optimal value of RUCB α was ultimately selected as $\alpha = 0.3$ for a budget of $5 \cdot 10^4$ advances.

v P0 | P1 >	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.1		36	43	51	45	47	47	48	44	37
0.2	46		41	45	55	49	51	46	45	48
0.3	45	38		40	43	45	53	52	50	47
0.4	34	38	35		50	39	49	40	50	49
0.5	33	34	40	31		43	43	48	41	40
0.6	33	41	41	42	43		43	45	40	41
0.7	43	37	29	29	48	48		50	43	53
0.8	34	35	32	40	37	34	40		45	46
0.9	46	35	35	38	42	41	42	34		43
1.0	38	31	39	31	37	37	37	37	33	

α	Wins
0.1	398
0.2	426
0.3	413
0.4	384
0.5	353
0.6	369
0.7	380
0.8	343
0.9	356
1.0	320

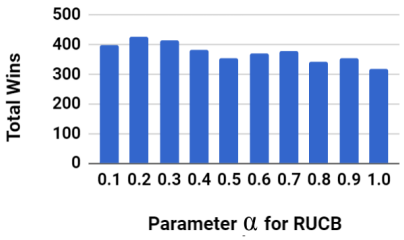

Figure 4.10.: 10k advances, values given are for RUCB α. Configuration with smaller value begins the game.

v P0 | P1 >	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.1		39	40	45	38	37	39	43	46	38
0.2	40		44	49	42	48	46	54	53	41
0.3	40	40		53	46	45	51	48	54	46
0.4	33	35	31		42	40	46	46	39	58
0.5	41	40	40	39		33	55	52	47	40
0.6	45	36	35	42	49		44	48	37	49
0.7	45	36	26	36	29	40		45	47	41
0.8	40	30	38	41	29	34	40		41	38
0.9	36	29	33	45	42	44	42	44		50
1.0	39	37	37	31	45	33	38	44	32	

α	Wins
0.1	365
0.2	417
0.3	423
0.4	370
0.5	387
0.6	385
0.7	345
0.8	331
0.9	365
1.0	336

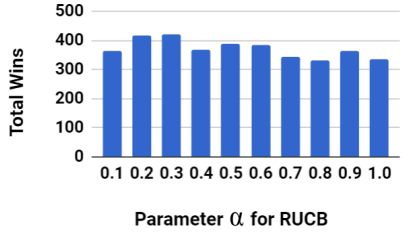

Figure 4.11.: 10k advances, values given are for RUCB α. Configuration with larger value begins the game.

α	Wins
0.1	763
0.2	843
0.3	836
0.4	754
0.5	740
0.6	754
0.7	725
0.8	674
0.9	721
1.0	656

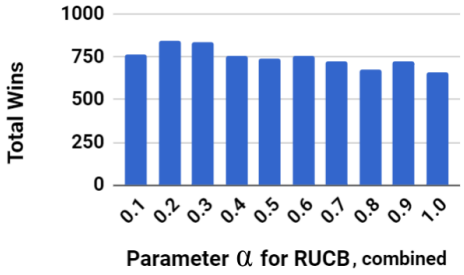

Figure 4.12.: 10k advances, combined wins for values of RUCB α. The optimal value for α is 0.2

The final step is to also determine α for the smaller budget of $1 \cdot 10^4$ advances, with the results displayed in figure 4.10 (smaller value begins game) and figure 4.11, again summing up the total number of victories for all considered values in figure 4.12, revealing the optimal parameter setting for the budget of $1 \cdot 10^4$ advances to be $\alpha = 0.2$.

4.2 Multiplayer experiments

Two algorithms are being tested against each other; realtime MCTS using UCT (RT) and relative MCTS using RUCB and one-back propagation (ROB). The experiments conducted are split in eight major batches, based on all eight possible combinations of the following three binary frame choices:

- **Computational budget:** $1 \cdot 10^4$ and $5 \cdot 10^4$ state advances per turn
- **Heuristic used for terminal states:** h_{simple} vs $h_{composite}$
- **Assignment of the first ply in all games:** RT begins vs. ROB begins

Each of these eight batches consists of a total of 2100 games, letting the two algorithms play 100 games of *Connect Four* against each other over 21 different settings for the maximum simulation length, with the maximum rollout length $RL_{max} \in \{2, 4, 6, ..., 42\}$. Because even the initial state of a fresh game of Connect Four is bound to reach a conclusion over the course of 42 plies, this is a sensible choice for maximum value, given that at that point simulations are effectively unlimited. In addition, the parameter values as empirically determined in the previous section are being used. This means that for all batches using $1 \cdot 10^4$ advances as computational budget, $C = 0.4$ for UCT and $\alpha = 0.2$ for RUCB, and for $5 \cdot 10^4$ advances, $C = 0.6$ and $\alpha = 0.3$. This section will display the results in graphical form, with the corresponding result tables listed in the appendix. The following statistics have been gathered during the course of the experiments:

- **Game results**: The number of wins for either agent, as well as the number of draws, over 100 games.
- **Terminal rate** TR_{avg}: The macro average terminal rate over 100 games, and as such the ratio of terminal simulations vs. non-terminal ones. For each game out of the series of 100 games, a terminal rate is calculated; the macro average for the entire series is then formed by averaging these 100 per-game averages.
- **Rollout length** RL_{avg}: The macro average rollout length that occurred over all 100 games, calculated in a similar fashion as the terminal rate.
- **Game length** GL_{avg}: The average game length across 100 games.

Figure 4.13 shows the results of all 2100 games played using $h_{comnposite}$, a budget of $5 \cdot 10^4$ advances and the preference-based MCTS variant (ROB) beginning each game. At first glance it is easy to see that regardless of simulation length setting (horizontal axis), the UCT-based approach wins the majority of games. However, the difference is generally smaller if lower simulation lengths are being chosen. The performance of ROB peaks at a simulation length limit of 8, with 34 wins (and 61 losses as well as 5 draws). Especially when reducing the simulation length further, the variance appears to increase, which is shown by the peak occurence of total draws at a simulation length limit of 4 and 24 draws. Interestingly, the win rate of ROB remains in line with the neighbouring limit settings of two and 6 at 24 wins, while the extra draws appear to be at the cost of the win rate of RT-MCTS. This may well be due to generally unreliable games at such short limits, but nevertheless can be observed, also explaining the peak in average game length at 37.5 plies (right vertical axis). After the peak at a limit of 8, the performance of ROB then quickly deteriorates both in win count as well as forced draws, with RT beginning to dominate more clearly until a rollout limit of 20 and more is reached, after which RT wins the

vast majorit of games.

The linear trends for the win rates of RT and ROB further show that a rising rollout length limit benefits RT and in this case both diminishes the wins scored by ROB as well as the number of draws, which begin to approach zero. The average rollout length slowly rises up to that same point and then stays more or less constant at a value of 3.9. This rollout length limit of 20 also marks a point where most rollouts end in terminal states. However, consider that due to the structure of the domain at hand and the MCTS process, both the macro average rollout lengths as well as the macro average terminal rates are not to be taken too literally - while the plotted curve may suggest that almost all simulations end in terminal states, this is not universally true, because both late iterations and (most importantly) late calls to MCTS when the game has progressed very far distort both of these numbers considerably (see also the more detailed explanation in chapter 5 concerning these two values), which applies to all following batches.

The results of the experiment batch of 2100 games with identical settings, but reversed assignment of the first move in every game is shown in figure 4.14. Interestingly, ROB appears to be starting stronger with 45 wins (vs 51 losses and 4 draws) at a simulation length of two. However, it is again important to point out that the reliability of the results is likely to be especially low with such a low simulation horizon. This theory is supported by the high average game length of 36.2 plies for this series of 100 games, suggesting that both algorithms are more concerned with preventing immediate losses, thus inflating the game length, while unable to plan ahead much. Once the simulation length limit is set to be larger than four, similar results to the reverse batch shown in figure 4.13 occur, with the average game length quickly stabilizing and RT taking the dominant lead. Where in the previous batch, ROB managed to peak at 34 wins over 100 games at a simulation length of 8, this is no longer the case here, scoring 20 wins in the otherwise same setting and then losing performance faster, again stabilizing at a low win ratio of about 10 to 15 wins per 100 games. This is likely due to RT being assigned the first move in this batch and thus shows that Connect Four may offer an advantage to the beginning player. In fact, assuming perfect play on both sides, it was proven [1] by Victor Allis that the first player can force a win in at most 41 plies. This of course is not entirely relevant to MCTS, given that both algorithms are probably far from perfect play regardless of setting, but nevertheless the observation that the performance of ROB drops faster compared to the previous batch may be due to RT making the first move in all games.
Figure 4.15 displays results for the same computational budget, assigning the first move to ROB again, but this time using the simple heuristic h_{simple} using only component A. Of particular interest is that the results (again with the exception of the potentially noisy first series of 100 games) show less of a trend plotted over the simulation length limit (with no visible performance trend from a lenght limit of 4 and above). The slight improvement in winrate ROB was able to show when using the composite heuristic $h_{composite}$ is not visible anymore. This does not match the intuition that RT-MCTS using UCT would benefit more from a complex heuristic than ROB, as in this case it estabilishes dominance almost immediately. However, this may well be due to the complex heuristic not necessarily being *better* (see chapter 5 for some observations concerning the heuristic functions used). The average game length only stabilizes after a rollout length limit of 12 and above, with the other results otherwise being comparable to the previous batches. The win rate of RT still shows a slight upwards linear trend with increasing rollout length, however this (as well as the minimal downwards trend of ROB) is likely due to the out-

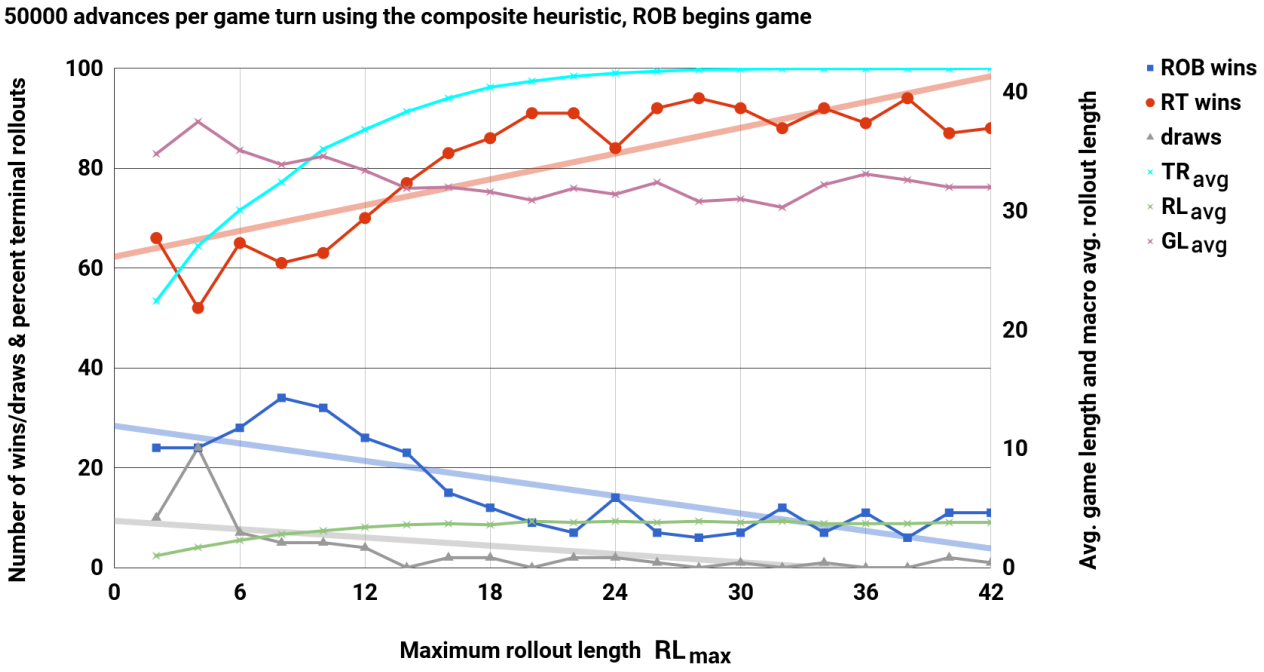

Figure 4.13.: Results of playing 100 games per rollout limit setting using the composite heuristic. ROB begins the game. Budget of $5 \cdot 10^4$ advances. Based on table A.1 (see appendix)

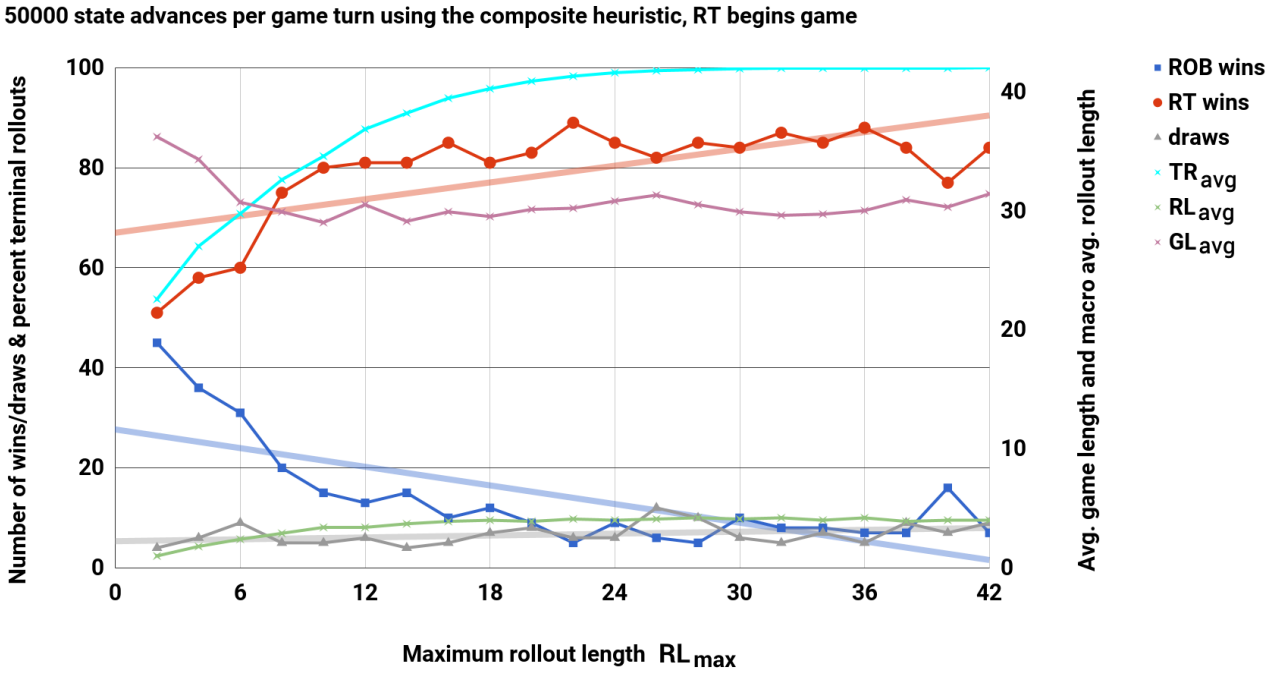

Figure 4.14.: Results of playing 100 games per rollout limit setting using the composite heuristic. RT begins the game. Budget of $5 \cdot 10^4$ advances. Based on table A.2 (see appendix)

lier result for a rollout length limit of 2.

The next batch of experiments is also the last batch to make use of the computational budget of 50000, now assigning the first move to RT in each game again and using the simple heuristic, with the results plotted in figure 4.16. It is unclear whether the lack of deviation from the rest of the results at a rollout length limit of two is due to RT being assigned the first move in every game, allowing it to score 71 wins out of 100 games at that setting, or whether this is again due to variance. Ultimately though, there is barely any visible linear trend left, with RT outperforming ROB with at minimum 71 wins (at rollout length 2) to at maximum a whole 91 wins at rollout length 24. Similarities to previous batches arise with the average game length being notably higher at lower simulation length limits.

The following four batches mirror the same structure as the first four, except setting the computational budget to $1 \cdot 10^4$ advances in all cases. The results of the first batch using the lower budget is shown in figure 4.17, using $h_{composite}$ and allowing ROB to make the first move. A notable difference to the corresponding batch with the same settings, but a higher computational budget, is more variation in the cumulative outcome of each series of 100 games. For instance, the number of wins for RT not only exhibits variance for lower rollout length limits (from 51 wins at rollout length limit 2 to 71 wins at rollout length 4 and again back to 62 wins at rollout length 6), but this behavior continues for the rest of the individual series of 100 games. For a simulation length limit of 32, RT wins a whole 97 games out of 100, however neither of the neighbouring rollout settings mirrors this (87 wins at a limit of 30 as well as 89 wins at a limit of 34). This higher variation of outcome is the case across the entire batch, suggesting that $1 \cdot 10^4$ advances are a comparatively small budget that does not allow either variant of MCTS to stabilize and assigns a lot of weight to the random factor native to the algorithm. However, the general superiority of RT is still mirrored clearly in this batch, as well as the linear trend favoring RT even more with rising simulation length limit.

Figure 4.18 shows the results for the 2100 games played with the same settings as the previous batch at $1 \cdot 10^4$ advances and using $h_{composite}$, but allowing RT to move first in every game. Again, the lower stability of the results across multiple rollout limit settings compared to $5 \cdot 10^4$ advances becomes apparent. Most notably, at a rollout length limit of two, RT scores 82 wins out of 100 games, whereas it merely won 51 out of 100 games with reversed turn order priority. This could be assumed to be due to the first-ply advantage now being assigned to RT, but comparing the results of the series of 100 games at a rollout length limit of 4 (the next step up) has RT winning 71 out of 100 games when it is *not* being assigned the first move in every game, compared to only 59 wins with the supposed first-move advantage. As such, these results cannot be used to show a first-move advantage. An unique result of this batch is that for a rollout length of two, the average game length is very low (23.5 plies). This is in fact the lowest average across 100 games in the entire set of experiments and shows that many of these games likely have been lost due to grave mistakes on the part of either algorithm during the early part of several games. Closer inspection reveals that indeed, a few games in this series of 100 have ended less than 10 plies into the game, with ROB entirely failing to block its opponent's first attempt at building a winning combination. ROB appears to be more prone to such error at very low budget settings; the interpretation in chapter 5 offers several explanations for why ROB appears to be hit even stronger by a lack of computational budget on this domain, resulting in

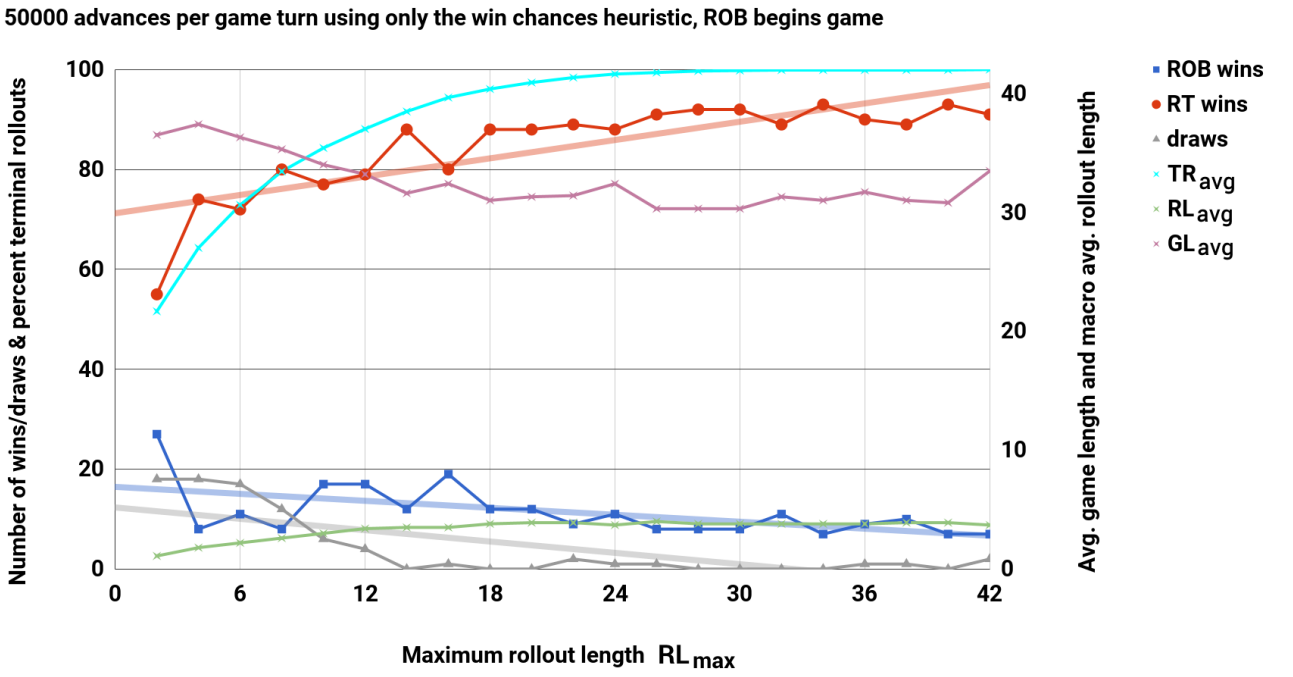

Figure 4.15.: Results of playing 100 games per rollout limit setting using the simple heuristic. ROB begins the game. Budget of $5 \cdot 10^4$ advances. Based on table A.3 (see appendix)

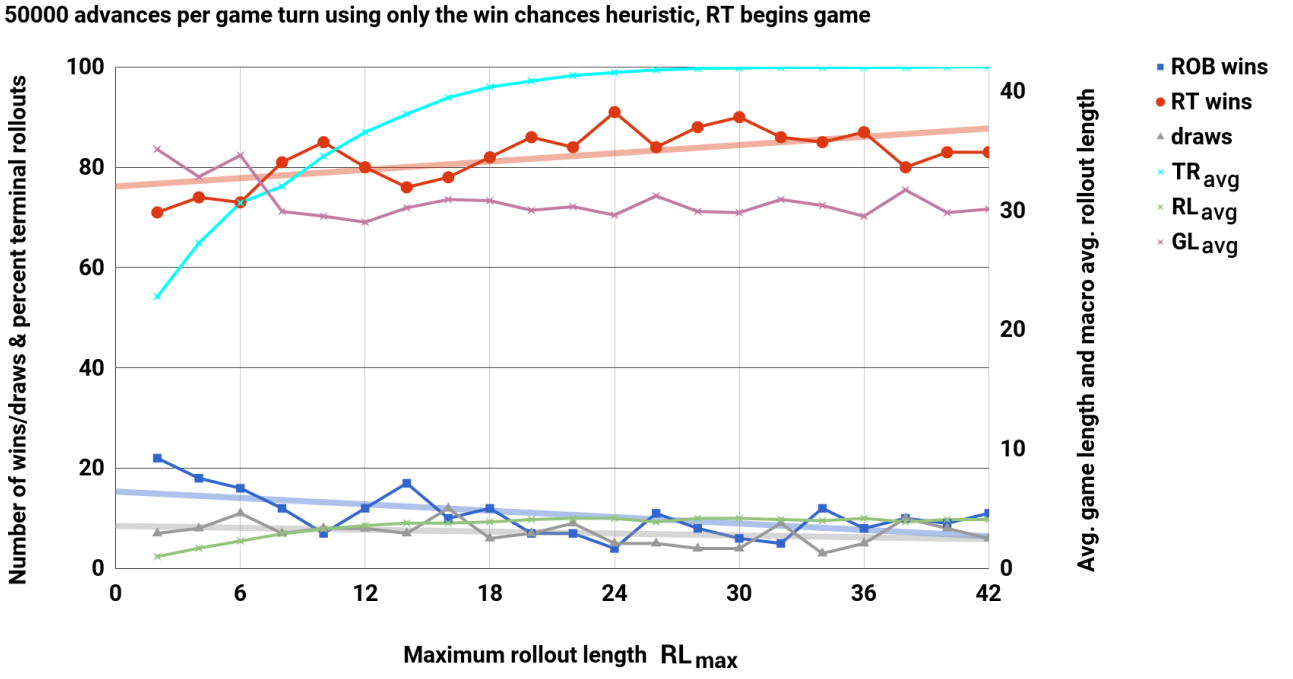

Figure 4.16.: Results of playing 100 games per rollout limit setting using the simple heuristic. RT begins the game. Budget of $5 \cdot 10^4$ advances. Based on table A.4 (see appendix)

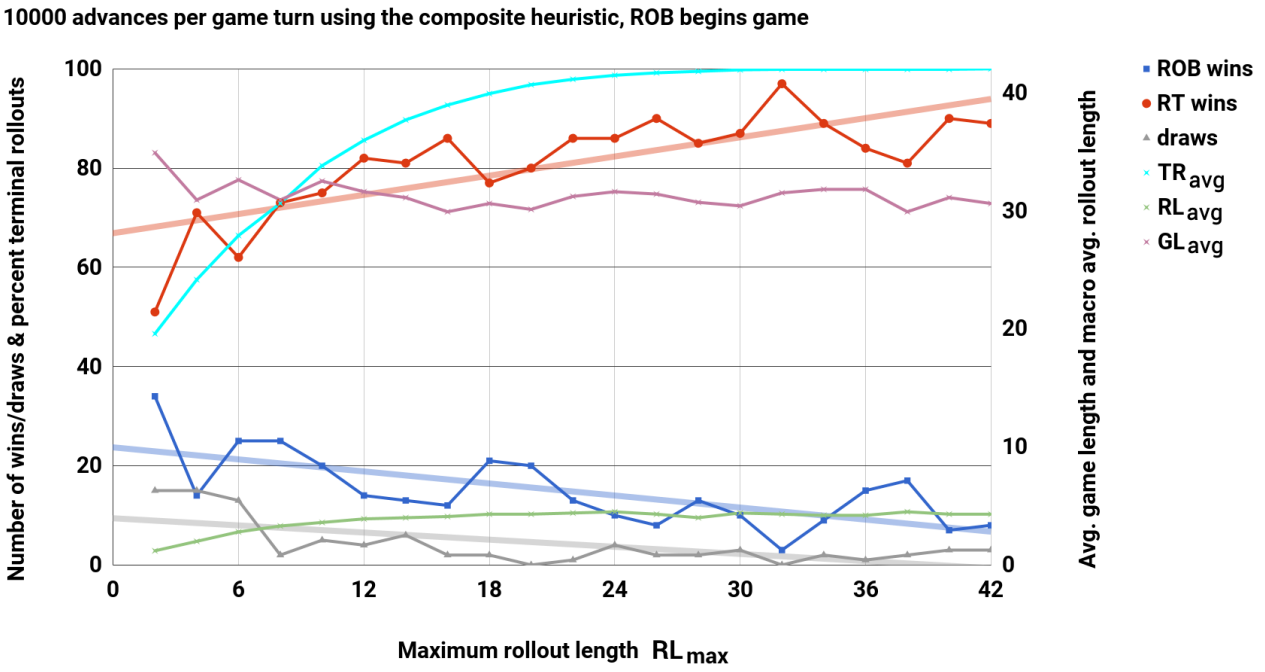

Figure 4.17.: Results of playing 100 games per rollout limit setting using the composite heuristic. ROB begins the game. Budget of $1 \cdot 10^4$ advances. Based on table A.5 (see appendix)

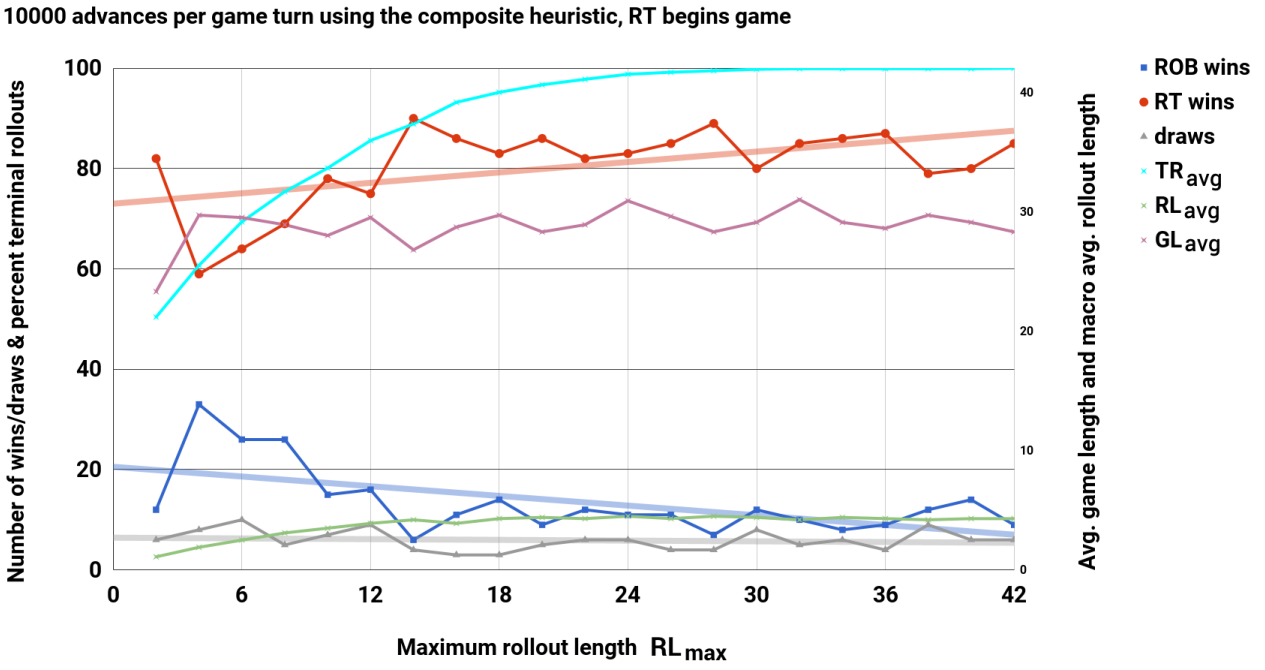

Figure 4.18.: Results of playing 100 games per rollout limit setting using the composite heuristic. RT begins the game. Budget of $1 \cdot 10^4$ advances. Based on table A.6 (see appendix)

an outlier value for average game length, especially considering that a rollout length limit of two was unreliable in all previous experiments just as well.

The last two batches cover the case of using h_{simple} with a budget of $1 \cdot 10^4$ advances. The results of assigning the first move in every game to ROB under these conditions are shown in figure 4.19. Similar to the results using the same heuristic, but in a context of $5 \cdot 10^4$ advances, the performance of ROB is low across the entire batch, suggesting that even though h_{simple} purposefully omits the element of threats from its score, UCT may generally favor purely considering the remaining win chances on this domain compared to the composite heuristic.

The last batch reverses the turn order one last time while keeping the settings of using h_{simple} with a budget of $1 \cdot 10^4$ advances, showing the results in figure 4.20. Most notably, the average game length at a rollout length limit of 2 is particularily high, which interestingly is also the case of a limit of 4. However, as the previous three batches have already shown, not too much weight should be assigned to the batches using the lower of both budgets. Nevertheless, this contrasts the results with the same settings, but using $h_{composite}$, as shown in figure 4.18, where the rollout limit of two has shown a much lower average game length (23.5 vs 36.5). It is unclear whether this is due to the different heuristics being used, or whether both outliers merely mark two extreme ends of essentially the same interpretation - a high randomness during gameplay, with the extreme low result due to games lost very early and the extreme high result due to games played without much foresight. The low number of draws (9) suggests that the latter may not be the case, but even a series of 100 games may not be enough to arrive at a solid conclusion given that the computational budget of $1 \cdot 10^4$ advances appears to simply be too low for ROB in particular.

A summary of peak results for the combined two batches using $5 \cdot 10^4$ advances and $h_{composite}$ (figures 4.13 and 4.14) is shown in table 4.1. Note that for this table as well as the following ones, neither terminal rate nor rollout length are included, as the former always peaks at maximum rollout length limit settings and the latter generally only exhibits significant growth up to a simulation length limit of 20, after which it fluctuates within a small margin. RT scores the highest number of victories in configurations with longer maximum rollouts, namely 28 and 38 (with 94 wins out of 100 each). The generous limit goes hand in hand with high terminal rates (0.997 and 0.999, but consider the explanation for the seemingly high values and rapid growth given in chapter 5) and comparatively long rollout lengths (3.9 and 3.7), with the average game length (30.8 and 32.6) hinting at games not being won purely out of chance as the game board begins to fill up completely. The opposite is true for ROB, which performs best at a minimal rollout length limit of two, also leading to long games (36.2 plies on average) as both algorithms are unable to simulate ahead much and rely on shorter-sighted action choices. The longest games (37.5) are found in the same configuration (rollout length limit 4, ROB is given the first plies in each game) as the highest number of draws (24), hinting at a subset of games playing out without much foresight.

The two batches (based on figures 4.15 and 4.16) using $5 \cdot 10^4$ advances combined with h_{simple} are being summarized in terms of peak wins for either player, peak draws and longest average game length in table 4.2. Unlike in the case of $1 \cdot 10^4$ advances shown later, there are no real differences based on the heuristic function used as far as peak results are concerned: RT wins

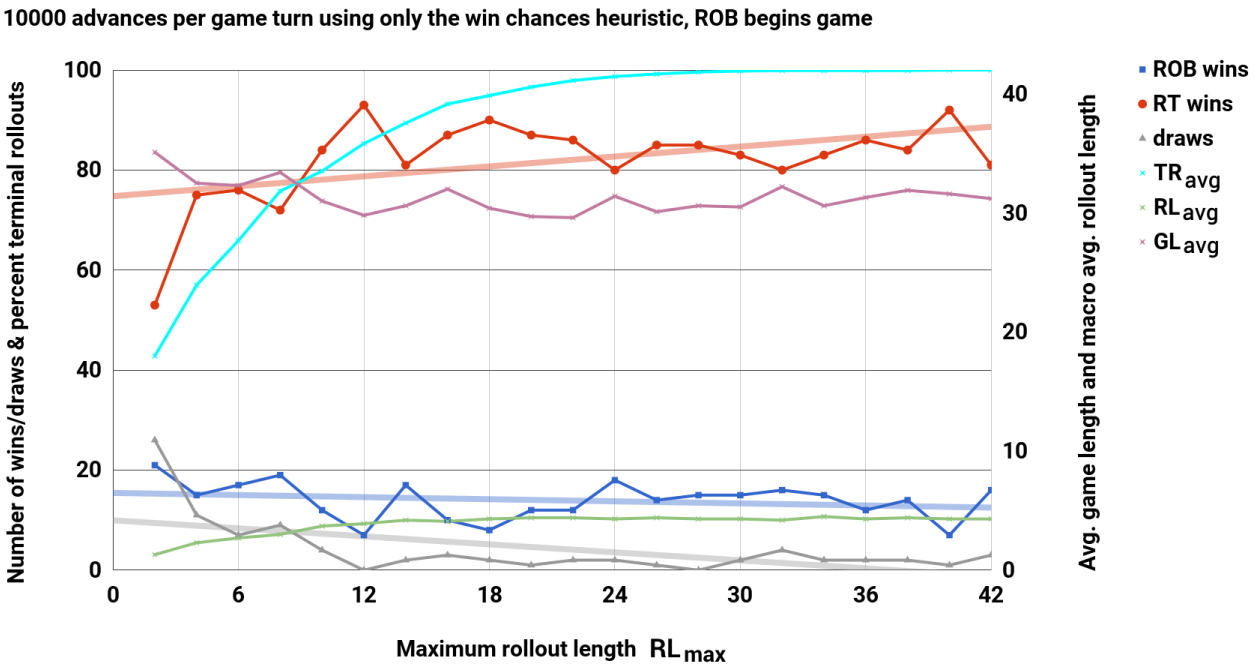

Figure 4.19.: Results of playing 100 games per rollout limit setting using the simple heuristic. ROB begins the game. Budget of $1 \cdot 10^4$ advances. Based on table A.7 (see appendix)

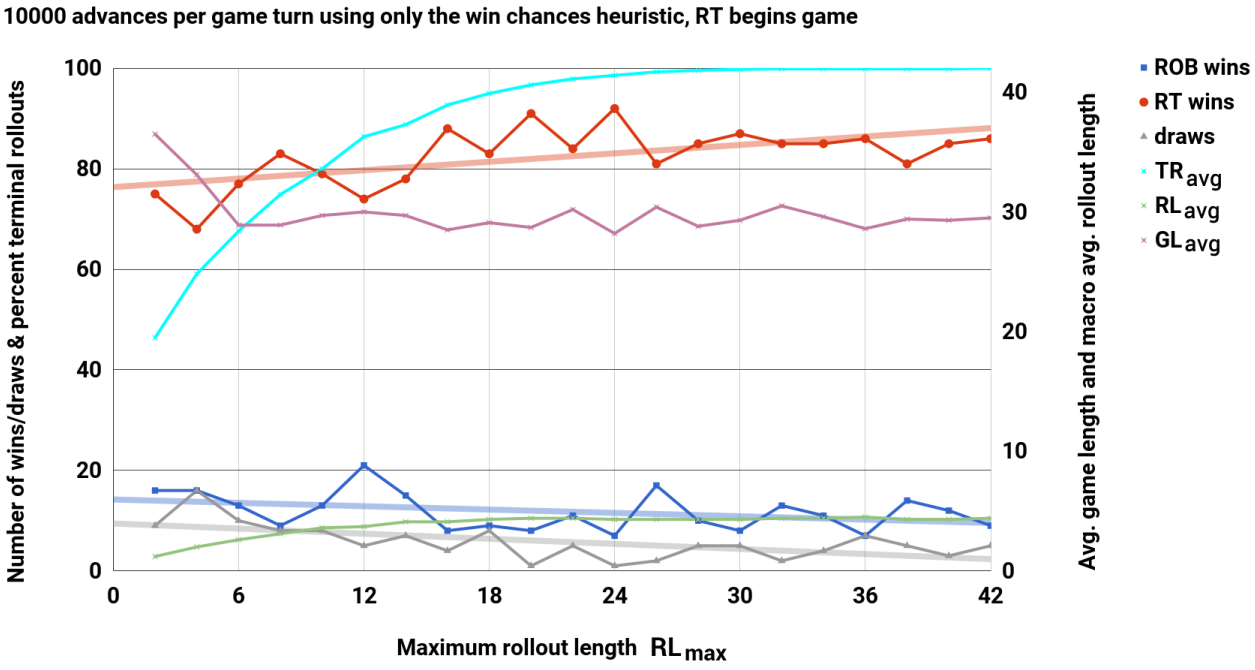

Figure 4.20.: Results of playing 100 games per rollout limit setting using the simple heuristic. RT begins the game. Budget of $1 \cdot 10^4$ advances. Based on table A.8 (see appendix)

Table 4.1.: Peak results for $5 \cdot 10^4$ advances and $h_{composite}$. Both batches meeting these criteria are being considered (covering both cases of either algorithm making the first move in all games)

	RL_{max}	first ply	ROB wins	RT wins	draws	TR_{avg}	RL_{avg}	GL_{avg}
Max. RT wins	**28**	ROB	6	94	0	99.7	3.9	30.8
	38	ROB	6	94	0	99.9	3.7	32.6
Max. ROB wins	**2**	RT	45	51	4	53.7	1.0	36.2
Max. draws	**4**	ROB	24	52	24	64.4	1.7	37.5
Max. GL_{avg}	**4**	ROB	24	52	24	64.4	1.7	37.5

Table 4.2.: Peak results for $5 \cdot 10^4$ advances and h_{simple}. Both batches meeting these criteria are being considered (covering both cases of either algorithm making the first move in all games)

	RL_{max}	first ply	ROB wins	RT wins	draws	TR_{avg}	RL_{avg}	GL_{avg}
Max. RT wins	**34**	ROB	7	93	0	99.9	3.8	31.0
Max. ROB wins	**2**	ROB	27	55	18	51.6	1.1	36.5
Max. draws	**2**	ROB	27	55	18	51.6	1.1	36.5
	4	ROB	8	74	18	64.3	1.8	37.4
Max. GL_{avg}	**4**	ROB	8	74	18	64.3	1.8	37.4

the most games at a high rollout length limit of 34, with most simulations ending in terminals (0.999) and the game length being unremarkably average (31.0), whereas ROB favors a rollout length limit (2) for both players that is too low to allow for meaningful simulation. Again, there are notable similarities between the configurations with peak ROB wins and those with peak draws, with the top configuration for ROB wins doubling as one of two top configurations as far as draws are concerned, and all of these showing very long average game lengths (36.5 and 37.4; the latter also being the configuration with highest average game length, at a rollout length limit of 4).

The top results for either player's win count, the draw count and the average game length for $1 \cdot 10^4$ advances and $h_{composite}$, taking into account both concerned batches (which again differ only in assignment of the first play) are summarized in table 4.3; as such, this refers to figures 4.17 and 4.18 both. As expected, the configuration yielding the highest number of RT wins across these two batches allows for much longer simulations ($RL_{max} = 32$) than the configuration allowing for relative peak performance of relative MCTS with filtered backpropagation (ROB), which performs best at a very small limit of two. This observation also holds for the terminal rate (TR_{avg}) where the configuration with the most RT wins exhibits a much higher rate (0.999) than the best configuration for ROB (0.466) across these two batches. All of the peak results are part of the experiment batch allowing ROB to draw first. The configuration with the most ROB wins is also one of two with the highest number of draws (15) and a low averaged rollout length (1.2) as well as showing the longest average game length at 34.9; another configuration with the same number of draws occurs at a rollout length limit of 4. These limits are comparatively small and will not allow MCTS to plan ahead much, which suggests that a relatively good (albeit still inferior) performance of ROB relies on unfavorable conditions for both algorithms and thus more on chance. The long average game length and ROB never actually scoring more wins than RT in any configuration appear to support this observation, whereas RT manages to express a solid performance with the best performing configuration allowing it to

Table 4.3.: Peak results for $1 \cdot 10^4$ advances and $h_{composite}$. Both batches meeting these criteria are being considered (covering both cases of either algorithm making the first move in all games)

	RL_{max}	first ply	ROB wins	RT wins	draws	TR_{avg}	RL_{avg}	GL_{avg}
Max. RT wins	**32**	ROB	3	97	0	99.9	4.3	31.5
Max. ROB wins	**2**	ROB	34	51	15	46.6	1.2	34.9
Max. draws	**2**	ROB	34	51	15	46.6	1.2	34.9
	4	ROB	14	71	15	57.5	2.0	30.9
Max. GL_{avg}	**2**	ROB	34	51	15	46.6	1.2	34.9

Table 4.4.: Peak results for $1 \cdot 10^4$ advances and h_{simple}. Both batches meeting these criteria are being considered (covering both cases of either algorithm making the first move in all games)

	RL_{max}	first ply	ROB wins	RT wins	draws	TR_{avg}	RL_{avg}	GL_{avg}
Max. RT wins	**12**	ROB	7	93	0	85.3	3.9	29.8
Max. ROB wins	**2**	ROB	21	53	26	42.8	1.3	35.1
	12	RT	21	74	5	86.4	3.7	30.0
Max. draws	**2**	ROB	21	53	26	42.8	1.3	35.1
Max. GL_{avg}	**2**	RT	16	75	9	46.4	1.2	36.5

score 97 wins out of 100 games.

Similar results occur when looking at peak results (shown in table 4.4) for the two batches (figures 4.19 and 4.20) swapping $h_{composite}$ for h_{simple} under otherwise same budget conditions (10000 advances), although in this case the top performance of RT occurs at a rollout length limit of only 12, like one of the two configurations with peak ROB victories (although interestingly, the top performance of ROB occurs when RT begins each game, and vice versa). ROB also scores the same number of wins (21) in a second configuration with a rollout length limit of two, as in the previous two batches. Again, the best configuration in terms of RT wins shows a high terminal rate (0.853), but one of the two configurations with relative peak performance for ROB has a slightly higher terminal rate (0.864), whereas the other configuration (with a limit of two) clocks in at 0.428. Again, the maximum number of draws (26) are achieved in one of the two configurations that also shows peak ROB performance at a rollout length limit of two and ROB being assigned the first ply in each game. Again at a limit of only two, but with RT beginning each game, the longest average game length (36.5 plies) occurs.

Combining the results of tables 4.1, 4.2, 4.3 and 4.4 in table 4.5, the most surprising result is that the globally best performing configuration in terms of RT wins at 97 out of 100 (10000 advances, $h_{composite}$, ROB begins) has ROB begin each game, while the reverse is true for the global peak result for ROB wins (45 wins, using $5 \cdot 10^4$ advances, $h_{composite}$, and allowing RT to move first). It is unclear whether this is simply due to the relative instability of the results, or whether in this experiment setup, the first ply can also be seen as a disadvantage, although the generally fluctuating number of wins, losses and draws would suggest the former. Other observations are less surprising: RT plays its best series of 100 games in a setting with generous rollout length limits (32), but limited budget (10000 advances), while the reverse is true for ROB (limit of 2 and $5 \cdot 10^4$ advances). Possible reasons for this behavior based on the differences of both

Table 4.5.: Peak results over all eight batches of 2100 games each

	RL_{max}	first ply	ROB wins	RT wins	draws	GL_{avg}	heuristic	budget
Max. RT wins	**32**	ROB	3	97	0	31.5	$h_{composite}$	10k
Max. ROB wins	**2**	RT	45	51	4	36.2	$h_{composite}$	50k
Max. draws	**2**	ROB	21	53	26	35.1	h_{simple}	10k
Max. GL_{avg}	**4**	ROB	24	52	24	37.5	$h_{composite}$	50k

algorithms are explained in chapter 5. The highest number of draws (26) to ever occur in a series of 100 uses h_{simple} and a budget of $1 \cdot 10^4$ advances as well as a low rollout lenght limit of only two, which is not surprising given that these combined settings are unlikely to allow for meaningful execution of either MCTS variant, thus letting more games end in a draw by filling up the game board with semi-random moves, although it should be noted that the longest average game length is achieved in a configuration with $5 \cdot 10^4$ advances and $h_{composite}$, but again a low rollout length limit (4).

5 Interpretation

Summing up the results of the previous chapter, the non-relative UCT based MCTS variant won the majority of games playing *Connect Four* against the preference-based MCTS variant making use of RUCB. The following sections aim to provide an array of possible explanations for this behavior by highlighting domain specific properties and their effects on the playing strength of the algorithms. Although the listing order of these possible explanations should not be seen as order of importance, the first two sections depend only on the (average) number of actions k for any particular domain, and as such are particularily important for domains with a high branching factor. Because one of the original motivations for MCTS specifically was handling such domains [3] where exact solvers or other forms of tree search are not feasible, this can be seen as a potential weakness of the preference based approach in the context of MCTS in general.

5.1 Number of possible actions in each state

The domain Connect Four allows anywhere between one to seven legal actions in each game state. It should be noted that for most states, especially those within the first half of the game, most of these seven theoretically available actions are also available in practice due to an action only becoming unavailable whenever a column is filled. An UCT based method will aim to sort the actions available in each state by an (absolute) quality metric - in this case, the upper confidence bound. Each update of a node during the process of MCTS backpropagation will update the quality estimate of one action out of k total available actions in that state. It is easy to see that this scales linearily with number of actions, and domains with large numbers of actions will not result in an unwieldy tree policy.

A preference-based approach to action selection does not offer the same scalability. It was shown [6] that preference-based MCTS yields better results than all UCT based variants on the (singleplayer) 8-puzzle domain. The 8-puzzle consists of nine tiles, eight of the occupied by numbered tokens, and each state being defined by the position of these tokens, with the goal of reaching a previously defined state. A legal action in the 8-puzzle domain moves one of the up to four tokens that are (non-diagonally) adjacent to the empty tile into said tile. If the empty tile is in the middle of the 3x3 board (1 out of 9 cases on average), there are $k = 4$ tokens that can be moved; if the empty tile is in one of the four corners of the board, only $k = 2$ actions are possible, otherwise $k = 3$ (4 out of 9 cases on average both). This results in an average of $(4 * 2 + 4 * 3 + 1 * 4)/9 \approx 2.67$ actions available in a single state of the 8-puzzle.

If $k = 1$, no preference can be formed. For $k = 2$, exactly one comparison between actions is possible; for $k = 3$ and $k = 4$, 3 respectively 6 comparisons can be made (the maximum number of comparisons possible in a single state of the 8-puzzle). For Connect Four, this number is much higher - in the worst case of $k = 7$, the total number of comparisons (and thus preferences) is 21. In the general case, the number of preferences in the context of k actions ($k > 1$) is $(k^2 - k)/2$ or alternatively $\sum_{i=1}^{k-1} k$, resulting in quadratic growth. Figure 5.1 visualizes the rapid growth

of possible comparisons for cases $k = 2$ through $k = 4$ (minimum and maximum number of actions for an 8-puzzle state, respectively) as well as $k = 7$ (maximum number of actions in Connect Four). Note that when filtering trajectories to only allow the winner of a comparison to be propagated back, each RUCB node will make exactly one such comparison per update. As such, the information gain per single update of a node rapidly diminishes with greater k.

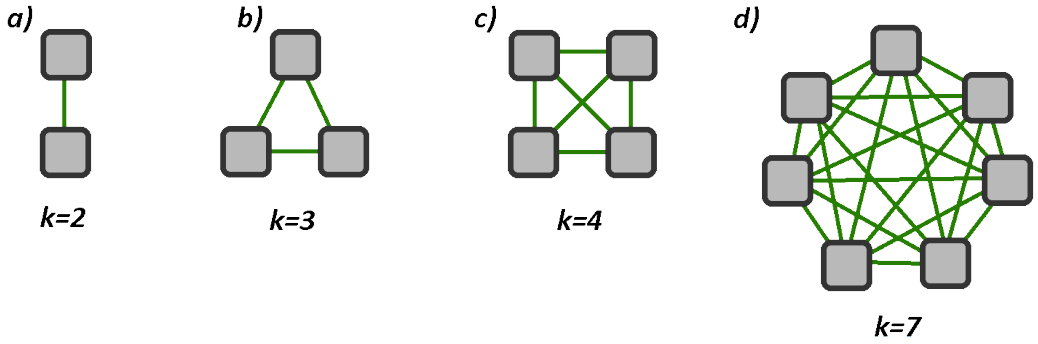

Figure 5.1.: Visualization of possible comparisons of two actions in the case of k=2 (a), k=3 (b), k=4 (c) and k=7 (d) available actions

5.2 Number of node updates per single rollout

As seen in chapter 2, a defining feature of preference based MCTS is the spanning of a *selection subtree* instead of a *selection path*, as well as the simultaneous addition of two new nodes to the list of known children of one selected node (= a leaf in the *selection subtree*). This will result in multiple rollouts per single iteration of MCTS; in the first iteration, this number is two, but subsequent iterations will on most domains begin a rapidly growing number of rollouts. Recall that we are using *state advances* as a measure of "cost" (as opposed to time) and as such, conducting rollouts uses up the majority of computational resources given to the algorithm before it stops initiating another iteration.

MCTS with UCT will make use of a selection path during the selection phase, and only add a single new node to the known tree (as defined by the tree policy) per iteration. This also results in a single rollout, which is then used to provide a node update to each node along the selection path. As such, for the cost of one rollout, multiple nodes receive a node update. The important root node is updated once per rollout. Relative MCTS, after committing to multiple rollouts per iteration, will also update multiple nodes during the backpropagation process (the nodes that are part of the selection subtree). The selection subtree of relative MCTS will generally contain more nodes than the selection path of MCTS with UCT. However, the number of nodes updated on average per iteration does not grow as fast as the *number of rollouts* per single iteration. Assuming that relative MCTS has, in one particular iteration, conducted m rollouts, and that the length (and thus cost in state advances) of a single rollout is on average comparable, it can be assumed that UCT-based MCTS would on average spread these m rollouts across roughly $i \approx m$ iterations with exactly one rollout each. However, in each iteration, the selection path is formed anew; this means that over the course of these i iterations, a node could be part of the selection path more than once. In fact, the root node - arguably the most important node, given that the final action selection after termination of the algorithm will be based on the tree policy in the root node - is updated every single time. In contrast, relative MCTS will use up the computational resources required by these m rollouts within a single iteration, but a node can only be part of the selection subtree or not in that particular iteration, and will thus only be updated once at maximum assuming a return policy that only allows a single trajectory to be propagated further after each comparison. This also applies to the root node, which only receives a single node update per iteration, regardless of the number of rollouts.

Each node update in relative MCTS with one-back propagation compares two actions. Given that the number of possible comparisons grows quadratically with number of actions (see above subsections) and that *on average*, fewer nodes receive a node update *per single rollout*, the algorithm thus requires a considerably higher number of state advances to make up for the reduced information gain compared to non-relative realtime MCTS on domains with a high number of actions. Combining the above two points can be seen as a structural domain-dependent weakness.

5.3 High availability of terminal states

In theory, a simulation ending in a terminal state benefits UCT over RUCB, because the reward value (or the components of the reward vector in the case of multiplayer) is located at the

lower and upper extremes of the permitted value range of 0 to 1. This helps the UCT formula converge faster, and given that no heuristic insecurity is involved in the evaluation of such a terminal state, the value is highly reliable. In contrast, one of the theoretical advantages of RUCB is that inaccuracies in the heuristic evaluation of a state do not matter as long as the preference formed based on the heuristic values is likely to be correct and as such, a domain where terminal states are not commonly available would potentially be more suited for the preference-based approach. It can be argued that *Connect Four* has a high availability of such states, given that the game tree has a maximum depth of 42 plies, and that terminal states can be found in the tree as early as 7 plies into the game (assuming the player who began the game was able to construct a row or column of four without being interrupted).

Recall that in chapter 4, a high raw percentage of simulation results were based on terminal states, with even short simulation lengths resulting in more than half of all simulation results being of terminal nature. However, it must be noted that these raw numbers likely overstate the influence of terminal states (this also applies to the average rollout length). This is particularily because of two reasons:

- **Late iterations within one run of MCTS:** With successive iterations, the tree as known by the tree policy continually grows asymetrically. This of course means that even when the simulation length is limited, the tree will eventually grow into regions that feature predominantly terminal nodes within reach of short simulations. Because such late iterations still have some number of state advances left if the limit has been set high enough in the beginning, and simulations will terminate much earlier at this point, this results in a high raw number of total simulations, thus overrepresenting the terminal situations in the calculation of the average per whole game. However, the tree growth in earlier simulations is more likely to have been steered by nonterminal states.
- **MCTS being called on late game stages:** MCTS will be called anew, with the full computational resource budget and an empty tree, everytime an agent is prompted to decide a move on the actual game board. However, the tree will be spanned from the game state currently present on the board, and associate this state with its root node. Because the full state advance budget is available (like with any call to MCTS), but the tree is spanned from a state that may already be only a few plies away from game over in all cases, this again results in a large number of very short terminal situations that distort the calculation of the average number of terminal states across the entire game. However, the game is either usually already decided at this point, or if it is not, then one of the agents will have estabilished a lead either by chance or by high playing strength in earlier plies. As such, the simulations that occur at this point, while greater in number than their predecessors due to reasons mentioned above, will have a lower influence on the outcome of the actual game.

In summary, the ratio of terminal states should not be taken at face value, but rather as a general view into how higher simulation limits will naturally increase this ratio (up to 100 percent if the simulation length equals the maximum game length in plies, since then every single simulation is guaranteed to at least end in a draw, if not a victory for either player). However, it must nevertheless be stated that terminal states are much more readily available than for example in the singleplayer domain of the 8-puzzle, which does not structurally force terminal states at all

(and typically artificially forces a loss if the win condition has not been met within a budget of 100 moves).

5.4 Quality of the heuristic functions

The theoretical advantage of preference-based methods builds upon the property of reducing the influence of noise in the heuristic evaluations by only memorizing the results of a direct comparison rather than the values involved (or the absolute difference between two values). However, this builds upon the core assumption that if $h(a) < h(b)$, it is reasonable to assume $P(a, b) = b$ and vice versa. While this does not have to universally hold true, it may be sufficient if this is generally the case, or if $P(a, b) = b$ is a reasonable assumption at the very least when $h(a) \ll h(b)$. Generally speaking, preference based methods are better suited for domains where the ordering implied by a heuristic function is somewhat likely to be reliable even if the actual values are not.

An example for domains that exhibit above property are single-agent pathfinding problems in an n-dimensional discrete space, assuming that local optima do not constitute high spikes or peaks in the problem landscape. For such problems, a heuristic can take the form of air distance or a similar metric that will exhibit some variance based on the topography of the problem, but probably tend to produce better values for states (nodes) that are a low number of discrete moves away from one or more target states. It should be noted that such problems usually can be solved with the help of A* [9] or similar exact solvers making use of the same heuristic because it is *admissible* and as such never overestimates the distance to the target state.

Multiplayer domains usually do not have the luxury of high quality heuristic functions, unless making use of extensive lookup tables using expert evaluation of particular states to generate their values. In fact, in the case of Connect Four, several examples have come up during the course of the experiments where a state has been found to be rated as highly undesirable for a particular player while in practice, that player not only was only a few plies away from a victory state, but they had already ensured their victory at that point (assuming no gross misplays). Figure 5.2 shows an example of such a state. Because the red player has blocked all eight theoretical rows of four in the upper two rows of the game board (visualized by the green lines), and the number of threats is otherwise mostly balanced (3 and 2 threats for blue and red respectively), the reward vector returned for this state by the composite heuristic is $(0.41, 0.59)$, with the first component being associated with the blue player. However, at this point, blue would have already won the game. By playing a token in the second column from the left (id 1), victory in the following turn would be guaranteed regardless of whether red would attempt to block the first threat.

As such, it cannot be assumed that even if $h(a) \ll h(b)$, $P(a, b) = b$ would be a reasonable assumption, much less so if the value difference is smaller. Thus, the preference based approach is deprived of one of its major selling points as the noise in the heuristic evaluations often extends beyond localized problematic areas of the search space. Finding working heuristics for multiplayer games that take into account all of the intricacies is a non-trivial task and likely to require either a high investment in memory or computational resources (or both), thus voiding

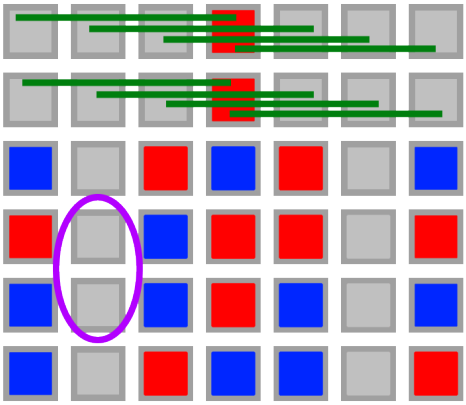

Figure 5.2.: Example of a Connect Four state that both heuristics used in this thesis wrongly consider as bad for the blue player, in particular due to the theoretical victory combinations blocked by red in the upper two rows. Blue has already won at this point due to the double threat (circled). This is not hinted at in the heuristic evaluation.

the motivation for limiting rollouts, as seeking out terminal states would then possibly be both faster and more reliable.

6 Conclusion

In the Connect Four multiplayer domain for two players, the preference-based approach of relative MCTS with one-back propagation has not been able to secure an advantage over regular UCT. A variety of possible reasons for this outcome have been highlighted. Namely, the quadratic growth of possible comparisons (and thus preferences) with rising number of actions available within a single state poses a problem for relative MCTS, which makes use of the RUCB formula. This causes the information gain from a single node update to be less significant than in the case of UCT, which at all times will update the absolute quality estimate for a single action once per update, thus maintaining a list with linear growth. In addition, UCT- based MCTS will on average update more nodes per unit of computational resource than relative MCTS. The difference is particularily drastic in the case of the important root node, which after termination will be consulted to determine the final action choice for the agent. Because the root node only updates once per iteration (generally for UCT, as well as for RUCB if using a return policy to only propagate one trajectory per node), and relative MCTS will expend far more computational resources within a single iteration (requiring a theoretically unlimited number of simulatenous rollouts compared to a single rollout in the case of UCT), this results in less node updates in addition to lower information gain per update for any node. Both of these issues are of structural nature and a result of the domain (and similarily structured other domains) at hand. In addition, the problem of finding heuristics that provide a semantically solid ordering of states even if they are allowed to have localized noise is non-trivial for the *Connect Four* domain. The results suggest that the preference-based approach is not well suited for domains with a large number of actions (high branching factor) but shallow (game-) trees (terminal state availability), as well as domains where the unreliability of heuristic evaluations extends beyond local optima, although this would be too broad of a statement to prove based on the available data.

Bibliography

[1] Victor Allis. A knowledge-based approach to connect-four. the game is solved: White wins. Master's thesis, Vrije Universiteit, 1988.

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

[3] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 2012.

[4] Michael Evans and Timothy Swartz. *Approximating Integrals via Monte Carlo and Deterministic Methods*. Oxford University Press, 2000.

[5] Johannes Fürnkranz and Eyke Hüllermeier. *Preference Learning*. Springer-Verlag New York, Inc., 1st edition, 2010.

[6] Tobias Joppen. Präferenzbasierte monte carlo baumsuche. Master's thesis, Knowledge Engineering Group, TU Darmstadt, 2016.

[7] Levente Kocsis and Csaba Szepesvári. Bandit-based Monte-Carlo planning. In *European Conference on Machine Learning*, 2006.

[8] John Von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[9] Stuart Jonathan Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.

[10] Masrour Zoghi, Shimon Whiteson, Rémi Munos, and Maarten de Rijke. Relative upper confidence bound for the k-armed dueling bandit problem. *CoRR*, abs/1312.3393, 2013.

A Experiment results

Table A.1.: Results for 50000 advances using $h_{composite}$ with ROB beginning each game

RL_{max}	2	4	6	8	10	12	14	16	18	20	22
ROB wins	24	24	28	34	32	26	23	15	12	9	7
RT wins	66	52	65	61	63	70	77	83	86	91	91
draws	10	24	7	5	5	4	0	2	2	0	2
TR_{avg}	53.4	64.4	71.6	77.2	83.8	87.7	91.3	94	96.2	97.4	98.4
RL_{avg}	1	1.7	2.3	2.8	3.1	3.4	3.6	3.7	3.6	3.9	3.8
GL_{avg}	34.8	37.5	35.1	33.9	34.6	33.4	31.9	32	31.6	30.9	31.9

RL_{max}	24	26	28	30	32	34	36	38	40	42
ROB wins	14	7	6	7	12	7	11	6	11	11
RT wins	84	92	94	92	88	92	89	94	87	88
draws	2	1	0	1	0	1	0	0	2	1
TR_{avg}	99	99.4	99.7	99.8	99.9	99.9	99.9	99.9	99.9	100
RL_{avg}	3.9	3.8	3.9	3.8	3.9	3.7	3.7	3.7	3.8	3.8
GL_{avg}	31.4	32.4	30.8	31	30.3	32.2	33.1	32.6	32	32

Table A.2.: Results for 50000 advances using $h_{composite}$ with RT beginning each game

RL_{max}	2	4	6	8	10	12	14	16	18	20	22
ROB wins	45	36	31	20	15	13	15	10	12	9	5
RT wins	51	58	60	75	80	81	81	85	81	83	89
draws	4	6	9	5	5	6	4	5	7	8	6
TR_{avg}	53.7	64.3	70.8	77.6	82.3	87.7	90.9	93.9	95.8	97.3	98.3
RL_{avg}	1	1.8	2.4	2.9	3.4	3.4	3.7	3.9	4	3.9	4.1
GL_{avg}	36.2	34.3	30.7	29.9	29	30.5	29.1	29.9	29.5	30.1	30.2

RL_{max}	24	26	28	30	32	34	36	38	40	42
ROB wins	9	6	5	10	8	8	7	7	16	7
RT wins	85	82	85	84	87	85	88	84	77	84
draws	6	12	10	6	5	7	5	9	7	9
TR_{avg}	99	99.4	99.6	99.8	99.9	99.9	99.9	99.9	99.9	100
RL_{avg}	4	4.1	4.2	4.1	4.2	4	4.2	3.9	4	4
GL_{avg}	30.8	31.3	30.5	29.9	29.6	29.7	30	30.9	30.3	31.4

Table A.3.: Results for 50000 advances using h_{simple} with ROB beginning each game

RL_{max}	2	4	6	8	10	12	14	16	18	20	22
ROB wins	27	8	11	8	17	17	12	19	12	12	9
RT wins	55	74	72	80	77	79	88	80	88	88	89
draws	18	18	17	12	6	4	0	1	0	0	2
TR_{avg}	51.6	64.3	72.9	79.6	84.3	88.1	91.6	94.4	96.1	97.4	98.4
RL_{avg}	1.1	1.8	2.2	2.6	3	3.4	3.5	3.5	3.8	3.9	3.9
GL_{avg}	36.5	37.4	36.3	35.3	34	33.2	31.6	32.4	31	31.3	31.4

RL_{max}	24	26	28	30	32	34	36	38	40	42
ROB wins	11	8	8	8	11	7	9	10	7	7
RT wins	88	91	92	92	89	93	90	89	93	91
draws	1	1	0	0	0	0	1	1	0	2
TR_{avg}	99.1	99.4	99.7	99.8	99.9	99.9	99.9	99.9	99.9	100
RL_{avg}	3.7	4	3.8	3.8	3.8	3.8	3.8	3.9	3.9	3.7
GL_{avg}	32.4	30.3	30.3	30.3	31.3	31	31.7	31	30.8	33.47

Table A.4.: Results for 50000 advances using h_{simple} with RT beginning each game

RL_{max}	2	4	6	8	10	12	14	16	18	20	22
ROB wins	22	18	16	12	7	12	17	10	12	7	7
RT wins	71	74	73	81	85	80	76	78	82	86	84
draws	7	8	11	7	8	8	7	12	6	7	9
TR_{avg}	54.2	64.9	72.9	76.2	82.2	87	90.6	93.9	96	97.2	98.3
RL_{avg}	1	1.7	2.3	2.9	3.3	3.6	3.8	3.8	3.9	4.1	4.2
GL_{avg}	35.1	32.8	34.6	29.9	29.5	29	30.2	30.9	30.8	30	30.3

RL_{max}	24	26	28	30	32	34	36	38	40	42
ROB wins	4	11	8	6	5	12	8	10	9	11
RT wins	91	84	88	90	86	85	87	80	83	83
draws	5	5	4	4	9	3	5	10	8	6
TR_{avg}	98.9	99.4	99.7	99.8	99.9	99.9	99.9	99.9	100	100
RL_{avg}	4.2	3.9	4.2	4.2	4.1	4	4.2	3.9	4.1	4.1
GL_{avg}	29.6	31.2	29.9	29.8	30.9	30.4	29.5	31.7	29.8	30.1

Table A.5.: Results for 10000 advances using $h_{composite}$ with ROB beginning each game

RL_{max}	2	4	6	8	10	12	14	16	18	20	22
ROB wins	34	14	25	25	20	14	13	12	21	20	13
RT wins	51	71	62	73	75	82	81	86	77	80	86
draws	15	15	13	2	5	4	6	2	2	0	1
TR_{avg}	46.6	57.5	66.4	72.9	80.5	85.6	89.7	92.7	95	96.8	97.9
RL_{avg}	1.2	2	2.8	3.3	3.6	3.9	4	4.1	4.3	4.3	4.4
GL_{avg}	34.9	30.9	32.6	30.9	32.5	31.6	31.1	29.9	30.6	30.1	31.2

RL_{max}	24	26	28	30	32	34	36	38	40	42
ROB wins	10	8	13	10	3	9	15	17	7	8
RT wins	86	90	85	87	97	89	84	81	90	89
draws	4	2	2	3	0	2	1	2	3	3
TR_{avg}	98.7	99.2	99.5	99.8	99.9	99.9	99.9	99.9	99.9	100
RL_{avg}	4.5	4.3	4	4.4	4.3	4.2	4.2	4.5	4.3	4.3
GL_{avg}	31.6	31.4	30.7	30.4	31.5	31.8	31.8	29.9	31.1	30.6

Table A.6.: Results for 10000 advances using $h_{composite}$ with RT beginning each game

RL_{max}	2	4	6	8	10	12	14	16	18	20	22
ROB wins	12	33	26	26	15	16	6	11	14	9	12
RT wins	82	59	64	69	78	75	90	86	83	86	82
draws	6	8	10	5	7	9	4	3	3	5	6
TR_{avg}	50.4	60.7	69.4	75.4	80.1	85.6	88.9	93.2	95.2	96.7	97.8
RL_{avg}	1.1	1.9	2.5	3.1	3.5	3.9	4.2	3.9	4.3	4.4	4.3
GL_{avg}	23.3	29.7	29.5	28.9	28	29.5	26.8	28.7	29.7	28.3	28.9

RL_{max}	24	26	28	30	32	34	36	38	40	42
ROB wins	11	11	7	12	10	8	9	12	14	9
RT wins	83	85	89	80	85	86	87	79	80	85
draws	6	4	4	8	5	6	4	9	6	6
TR_{avg}	98.8	99.2	99.5	99.8	99.9	99.9	99.9	99.9	99.9	100
RL_{avg}	4.5	4.3	4.5	4.4	4.2	4.4	4.3	4.2	4.3	4.3
GL_{avg}	30.9	29.6	28.3	29.1	31	29.1	28.6	29.7	29.1	28.3

Table A.7.: Results for 10000 advances using h_{simple} with ROB beginning each game

RL_{max}	2	4	6	8	10	12	14	16	18	20	22
ROB wins	21	15	17	19	12	7	17	10	8	12	12
RT wins	53	75	76	72	84	93	81	87	90	87	86
draws	26	11	7	9	4	0	2	3	2	1	2
TR_{avg}	42.8	57	65.9	75.8	79.8	85.3	89.4	93.2	94.9	96.6	97.9
RL_{avg}	1.3	2.3	2.7	3	3.7	3.9	4.2	4.1	4.3	4.4	4.4
GL_{avg}	35.1	32.5	32.3	33.4	31	29.8	30.6	32	30.4	29.7	29.6

RL_{max}	24	26	28	30	32	34	36	38	40	42
ROB wins	18	14	15	15	16	15	12	14	7	16
RT wins	80	85	85	83	80	83	86	84	92	81
draws	2	1	0	2	4	2	2	2	1	3
TR_{avg}	98.7	99.2	99.6	99.8	99.9	99.9	99.9	99.9	100	100
RL_{avg}	4.3	4.4	4.3	4.3	4.2	4.5	4.3	4.4	4.3	4.3
GL_{avg}	31.4	30.1	30.6	30.5	32.2	30.6	31.3	31.9	31.6	31.2

Table A.8.: Results for 10000 advances using h_{simple} with RT beginning each game

RL_{max}	2	4	6	8	10	12	14	16	18	20	22
ROB wins	16	16	13	9	13	21	15	8	9	8	11
RT wins	75	68	77	83	79	74	78	88	83	91	84
draws	9	16	10	8	8	5	7	4	8	1	5
TR_{avg}	46.4	59.1	67.6	74.9	80	86.4	88.8	92.7	95	96.7	97.9
RL_{avg}	1.2	2	2.6	3.1	3.6	3.7	4.1	4.1	4.3	4.4	4.4
GL_{avg}	36.5	33.1	28.9	28.9	29.7	30	29.7	28.5	29.1	28.7	30.2

RL_{max}	24	26	28	30	32	34	36	38	40	42
ROB wins	7	17	10	8	13	11	7	14	12	9
RT wins	92	81	85	87	85	85	86	81	85	86
draws	1	2	5	5	2	4	7	5	3	5
TR_{avg}	98.6	99.3	99.6	99.8	99.9	99.9	99.9	99.9	99.9	100
RL_{avg}	4.3	4.3	4.3	4.3	4.4	4.4	4.5	4.3	4.3	4.4
GL_{avg}	28.2	30.4	28.8	29.3	30.5	29.6	28.6	29.4	29.3	29.5