Entwicklung eines dynamischen Abbruchkriteriums für UCT-Spielbaum-Suche

Development of a dynamic termination criterion for UCT-Gametree-Search

Bachelor-Thesis von Christoph Tilman Strübig aus Heppenheim (Bergstraße)

März 2018

TECHNISCHE UNIVERSITÄT DARMSTADT

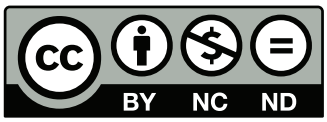
Fachbereich Informatik
Knowledge Engineering

Entwicklung eines dynamischen Abbruchkriteriums für UCT-Spielbaum-Suche
Development of a dynamic termination criterion for UCT-Gametree-Search

Vorgelegte Bachelor-Thesis von Christoph Tilman Strübig aus Heppenheim (Bergstraße)

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

Tag der Einreichung:



Erklärung zur Abschlussarbeit gemäß §23 Abs. 7 APB der TU Darmstadt

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

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Datum / Date: Unterschrift / Signature:

_____ _____

Contents

1 Introduction

Upper Confidence Bound 1 applied to trees (UCT), as a representative of Monte Carlo tree search (MCTS), is one of the most important algorithms for domain-independent and random based tree search. The basic idea is to play a huge number of random games, to identify the best possible action, without the need of expert knowledge. UCT combines this with the upper bound of confidence intervals, to direct the search and focus on promising paths. UCT converges to an optimal solution, although it doesn't have an utility to recognize it, if it eventually find it. Additionally, this kind of algorithm is rarely used in an environment where optimal solutions are required (deterministic algorithms like the α-β algorithm are more suited for these situations). The task is rather to detect a good action, respectively the best action with the current knowledge.
This thesis' subject is to develop a criterion for dynamic termination in UCT. In other words: At which point can we say, that we found a solution, that is good enough, without adding any expert knowledge? To accomplish this, we combine UCT with a deterministic interval based algorithm, called B*. B* assigns an interval to each possible action, which always has to contain its true value. With this condition and narrowing intervals , B* has a strong termination criterion: Whenever one interval is above all other intervals, the according action has to be the best possible, no matter how good it is in the end.
This thesis focuses on two-player, zero-sum games, but the findings are not necessarily restricted to this domain. The result is a variation of UCT, that uses the interval-termination of B*. We gave this new algorithm the name UCT*.

1.1 Structure

This thesis is divided into four parts. First, some foundations like game-tree-search and the used algorithms are described, followed by a section, in which the different approaches for early termination and pruning in UCT are presented. After that, the results are evaluated and discussed, before the final part addresses the conclusions and possible future work.

2 Foundations

The following section addresses the main foundations for this thesis. Next to general game tree search, this would be UCT and B*, as well as a short introduction to the three games, used for evaluation (Connect4, Othello and Breakthrough).

2.1 Game Tree Search

A common way to approach artificial intelligence in two-player zero-sum games is the game tree search. Zero-sum means, that one player's profit is equal to the other player's loss and vice versa. The basic idea is to build a tree, containing all possible board states and search for a path, that guarantees a victory or at least improves the chances to win. Building the tree is quite simple: First, expand the root node (create a child for every possible move), then expand every child and so on, until a terminal state (e.g. win, loss, draw) is reached or no possible moves are left (see Algorithm 1). Each layer of the tree represents choices of one specific player, e.g. every node in the second layer represents a turn of the second player. The player with the turn on the root node is referred to as the *max* player, since he tries to maximize the outcome of the game. The other player is called the *min* player and tries to minimize the profit of the *max* player. A simple example can be seen in Figure 1. Each leaf node gets a value (1 if the *max* player wins, 0 if the *min* player wins). These values are then propagated upwards, while the *min* player always chooses the option with the lowest value, and the *max* player chooses the one with the highest value. In the presented scenario, the *max* player has a guaranteed win if he takes the second option on the root node.
In practice, building a full game tree is often neither recommendable, nor realizable, due to the huge amount of possible board states. Most algorithms solve this problem by building the tree on the run. This way the search depth can be locally variable and already visited partial trees without a promising path can be pruned, which can greatly reduce calculation time and required memory.

Algorithm 1 Building a full game tree

```
1:  BuildTree(root)
2:
3:  procedure BuildTree(Node node)
4:      Expand(node)
5:      for all Node child in node.getChildren() do
6:          BuildTree(child)
7:      end for
8:  end procedure
9:
10: procedure Expand(Node node)
11:     for all move in node.getPossibleMoves() do
12:         Board copyOfBoard = node.getCopyOfBoard()
13:         copyOfBoard.performMove(move)
14:         Node child = new Node(copyOfBoard)
15:         node.appendChild(child)
16:     end for
17: end procedure
```

Game tree search algorithms can be divided into two classes: Deterministic approaches try to traverse the tree, respectively relevant sub trees, to a certain depth and evaluate all nodes on the deepest layer with some sort of heuristic. Since most game trees are too large to be fully traversed, these heuristics have a great impact on the quality of the result and usually have to be domain specific (e.g. number of pieces in chess).The most famous algorithm of this kind would be the α-β algorithm [1]. Random-based algorithms on the other hand try to approach a node's/move's value by playing multiple random

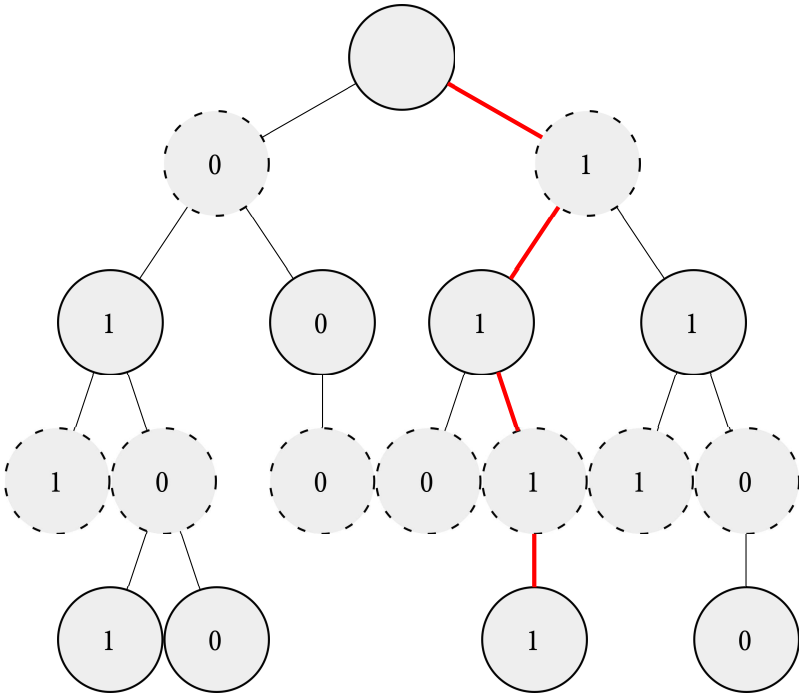


Figure 1: A simple example of a full game-tree. The red lines indicate one rollout with perfect play of both players.

games. This kind of algorithm does not necessarily need any kind of domain-specific knowledge, except the basic rules, what makes them domain-independent, even though the search can be improved by adding expert-knowledge. Expert-knowledge could come, for example, in form of a database with pre-evaluated favorable positions. Another advantage of random-based algorithms is convergence. They can often be interrupted at any point of the calculation, while more time should roughly equal a better move selection. Deterministic algorithms on the other hand have to traverse the whole tree to a certain depth before any expressive decision can be made. This problem can be lessened by using iterative deepening techniques, where the search depth is increased step by step, but the main problem remains, if considering the exponential growth of the tree. On top of that, deterministic approaches struggle often with the so called “horizon effect”: Since the search depth is limited, game-deciding situations below this depth are not taken into account, a problem, that random-based tree search is not affected by. Local increase of the search depth can be a solution to this problem, but recognizing situations, that require a deeper look is a problem on its own.
In other words, random-based algorithms have the advantage of usually utilizing complete rollouts, which prevents the horizon effect, while they can work domain-independent. This thesis’ focus lies on one of the most popular random based algorithm, called UCT, which is described in the following section.

2.2 UCT

UCT (Upper Confidence Bound 1 Applied to Trees) is a variant of the Monte Carlo Tree Search algorithm (MCTS) [2] and based on the Upper Confidence Bound 1 (UCB1) formula [3]. It was developed by Koscis and Szepesvári in 2006 [4]. UCT, as a MCTS algorithm, is domain independent and uses random sampling to approximate each moves value for the player. Provided with the basic game rules (possible moves and termination), it generates random rollouts and returns the move with the best expected profit. With an increasing number of iterations, the approximation for each move should converge to its true value, the value that it would have in a full game tree. In other words: UCT converges to an optimal solution. Due to its random based character, UCT can only provide information based on probabilities, hence it has no utility to identify the optimal move with absolute certainty as the final solution.

MCTS in general can be divided into 4 steps: Selection, expansion, simulation and back-propagation. In the selection phase, starting from the root node, a child is selected by a given selection function, until a leaf is reached (this would be the root node itself on the first iteration, since MCTS usually starts with a single node). Next, this leaf gets expanded. Now a random rollout from this node is generated/simulated (random moves until the game reaches a terminal state or a predefined maximum rollout-depth). The result of the rollout is then stored in each parent node, up to the root node (back-propagation). Algorithm 2 shows the four phases with random selection. These four steps are executed repeatedly, until a given time limit or number of iterations is reached. UCT can also be interrupted at any time, returning the currently best solution (this is a huge advantage versus deterministic algorithms like the α-β-algorithm, which requires a full search to a specific depth, before providing any usable output).

Algorithm 2 Phases of MCTS with random selection

```
 1: procedure SELECT(Node node)
 2:     Node currentNode = node
 3:     while currentNode.isExpanded() and currentNode.hasChildren() do
 4:         currentNode = currentNode.getRandomChild()
 5:     end while
 6:     return currentNode
 7: end procedure
 8:
 9: procedure EXPAND(Node node)
10:     for all move in node.getPossibleMoves() do
11:         Board copyOfBoard = node.getCopyOfBoard()
12:         copyOfBoard.performMove(move)
13:         Node child = new Node(copyOfBoard)
14:         node.appendChild(child)
15:     end for
16: end procedure
17:
18: procedure SIMULATE(Node node)
19:     Board board = node.getBoard()
20:     while not board.isGameOver() do
21:         board = board.performRandomMove()
22:     end while
23:     return board.getScore()
24: end procedure
25:
26: procedure BACKPROPAGTE(Node node, Float score)
27:     node.addScore(score)
28:     if node.hasParent() then
29:         Backpropagate(node.getParent(), -score)
30:     end if
31: end procedure
```

UCT improves the random selection from above by replacing it with an approach that takes previous rollouts into account. Its formula is derived from UCB1, which provides a formula for minimal regret for the multi-armed bandit problem. The strength of the UCT selection is, that it *exploits* known good moves, while *exploring* moves with low expectations from time to time to reduce the probability of a overlooked profitable path. The child, for which the following formula 1 has the highest value U_i, is

selected on every layer of the tree (see Algorithm 3). The first part (see formula 2) is the current payout P_i of the move: The higher the win rate in this sub tree is, the higher this value gets. The second part V_i (see formula 3) grows, whenever the according move is not selected and guarantees, that all moves are taken into account eventually. V_i can also be considered as the variance.

$$U_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}} \tag{1}$$

$$P_i = \frac{w_i}{n_i} \tag{2}$$

$$V_i = c\sqrt{\frac{\ln N_i}{n_i}} \tag{3}$$

Algorithm 3 Selection in UCT

```
1: procedure SELECT(Node node)
2:     Node currentNode = node
3:     while currentNode.isExpanded() and currentNode.hasChildren() do
4:         currentNode = currentNode.getChildWithHighestUCTValue()  ▷ Only difference to MCTS with random selection
5:     end while
6:     return currentNode
7: end procedure
```

w_i is the number of wins for the move, n_i represents the number of times, this move was performed. N_i stands for the total number of simulations for the current node ($N_i = \sum n_i$). c is called exploration rate and can be chosen empirically. If c gets lowered, the algorithm will focus more on exploiting, if it gets higher, the algorithm will focus on exploration.
UCT outclasses MCTS with random selection by a large margin, due to its ability to focus on profitable-paths, while causing only minimal overhead, and is one of the most common MCTS algorithms. The objective of this thesis is to equip UCT with a utility to terminate on its own, if one evaluated move is clearly better than the other moves, while retaining the benefit of its exploitation-exploration-balance.

2.3 B*

B* was first mentioned by Berliner in 1979[5] and is based on the α-β-algorithm. Rather than assigning fixed values to evaluated nodes, it provides intervals which has to contain the true value of the node (see Figure 3). These intervals can then be shrinked by evaluating more children to get a more exact guess. The borders of each node are also called pessimistic (lower bound) and optimistic (upper bound). The great advantage of this approach is the possibility to terminate, if (from the root-node's perspective) one interval (and hence one move) is clearly better than all other intervals. An interval is considered clearly better when its lower bound is higher than the upper bounds of all siblings) (see Figure 2.

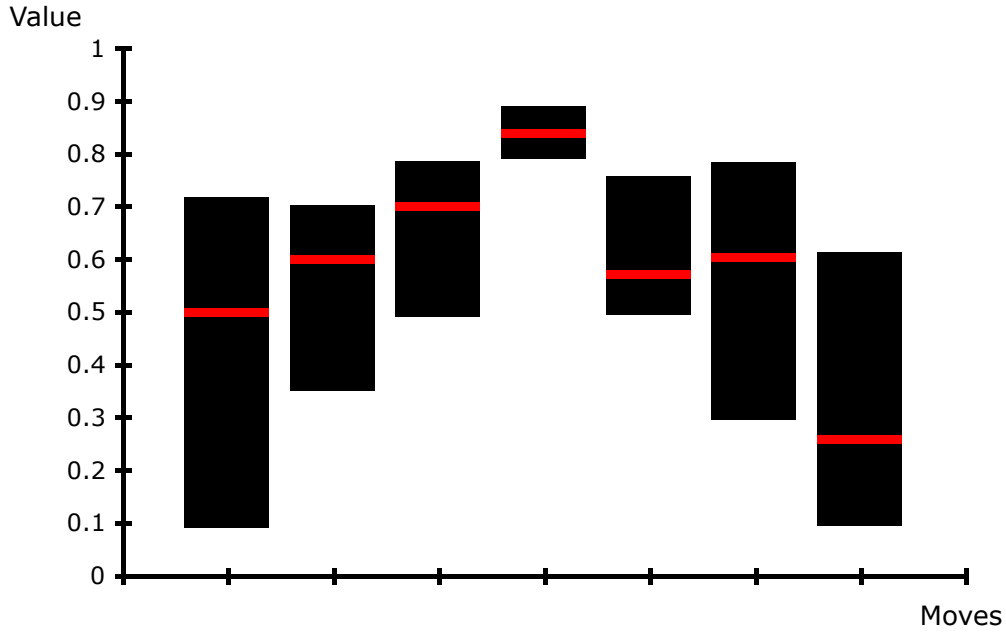


Figure 2: An idealized example of B*. The red bars indicate the unknown true value of each move, while the black bars indicate the assigned intervals.

Instead of traversing the whole tree (respectively relevant sub trees) like the α-β-algorithm, the B*-algorithm applies two main strategies: Prove-best and disprove-rest (see Figure 4). Prove-best tries to raise the lower bound of the best child over the upper bounds of all siblings, while disprove-rest tries to lower the upper bounds of all siblings under the best child's lower bound. Which strategy should be applied in which situation on the root-node is the topic of additional papers, like [6]. On every following layer, the prove-best strategy is used. Whenever a leaf-node is reached, it is expanded and the new nodes are evaluated by a provided function. After that, the parent's intervals are updated: The max-player chooses the highest lower and upper bound of its children, while the min-player chooses the lowest. It is important, that upper bounds should only be lowered, while lower bounds should only be raised. If this requirement isn't satisfied, the algorithm might terminate with a move that isn't optimal. Whenever the evaluation function returns a fixed value instead of an interval (e.g. due to a terminal position), this value can be handled like an interval, with the fixed value operating as the lower and upper bound.
Despite there are scenarios where this approach works quite well, in most cases choices are not clearly separable without traversing partial trees to the maximum depth. If symmetry isn't detected by the implementation, two or more nodes can even have the exact same intervals/values in each step. This leads to the necessity of another termination criterion, like a maximum number of iterations, after which the currently best move, e.g. the move with highest optimistic value, is returned. Besides similar moves and symmetry, the increased complexity of the evaluation function is another problem. Not only does it have to evaluate an interval instead of one guessed value, it also has to guarantee that this interval contains all following intervals and values.
For this thesis, the idea of observing intervals rather than values, was used to prune branches with low expectations and (in some cases) terminate the search in UCT.

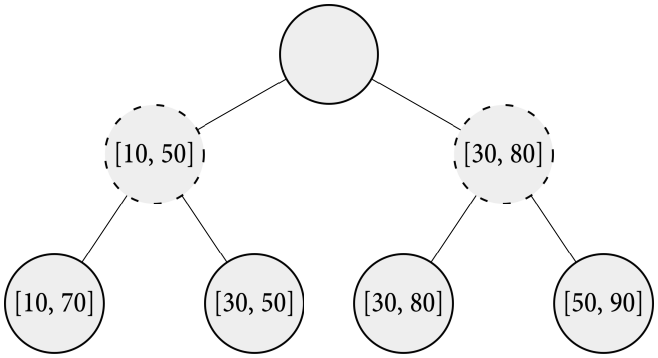
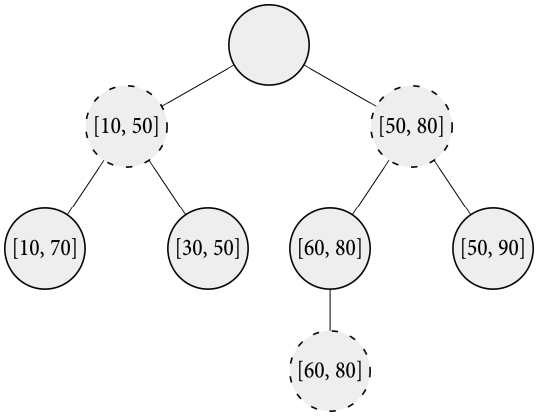
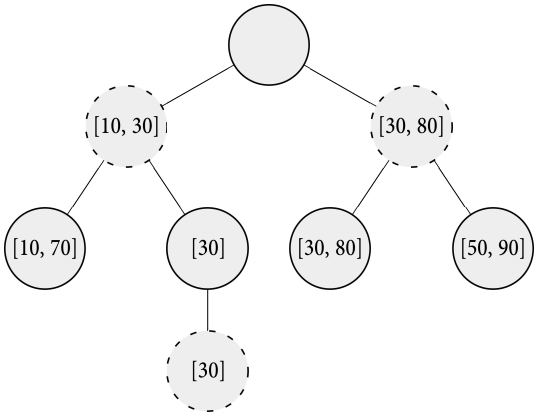


Figure 3: A simple example of a B*-tree. Each leaf-node has an assigned interval, each intermediate's node interval consists of the highest lower and upper bounds of it's children for the max player, while the min player chooses the lowest upper and lower bounds.



(a) Prove-best: From the root-node's view, we try to raise the highest lower bound over the upper bounds of all siblings.



(b) Disprove-rest: From the root-node's view, we try to lower all upper bounds below the highest lower bound.

Figure 4: Prove-best (a) and disprove-rest (b) in B*.

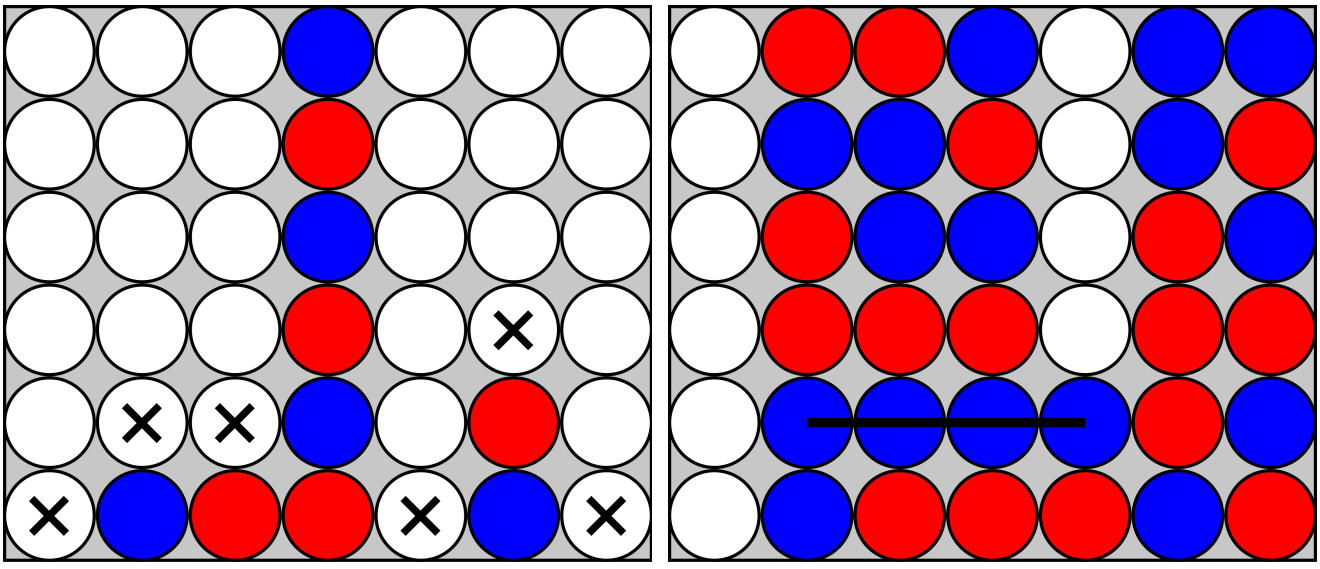
2.4 Games for Evaluation

Each developed algorithm was tested by competing in three different games. The selected games (Connect4, Othello and Breakthrough) are deterministic two-player-games, with perfect information (all available and already played moves are known by all players) and a fixed maximum number of turns. Working with Monte-Carlo-based algorithms, this provides the advantage of getting complete rollouts in every simulation step. If there was no upper bound for the number of moves, the simulation step would have had to stop after a certain depth, before a leaf is reached, to prevent infinite rollouts. In these situations, no statement could be made without adding domain-specific knowledge and the back-propagated result would be a draw.
Regarding the theoretical full game tree, the three games mainly differ in two properties: Average branching factor (how many child-nodes does the average node have) and maximum tree-depth (how long is the longest possible rollout). The specific games will be presented in the following sections.

2.4.1 Connect4

Branching factor: 4
Maximum depth: 42

Connect4 is played on a vertical board (all placed stones will drop to the lowest free space in their column) with a width of 7 and a height of 6. In alternating turns and starting with an empty board, players drop stones of their own color in a free column (see Figure 5(a)). The player that manages to build a row of 4 stones of his color, either horizontal, vertical or diagonal, wins the game (see Figure 5(b)). If no empty spaces are left, the game ends with a draw. The difficulty is to force the opponent into some sort of dilemma.



(a) Possible moves, marked with X

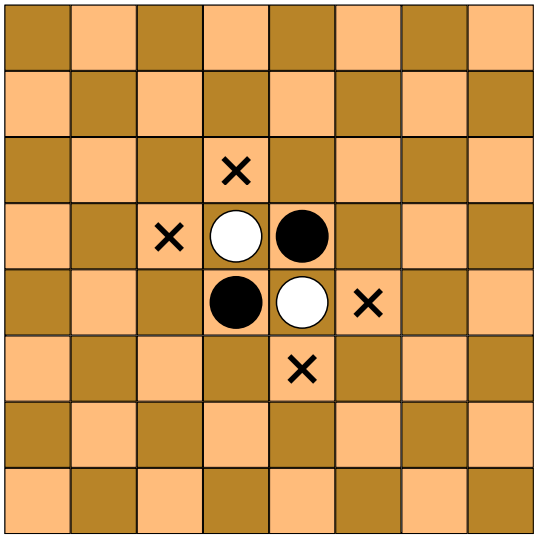
(b) Terminal state, winning row indicated by the black line

Figure 5: A typical board of Connect4

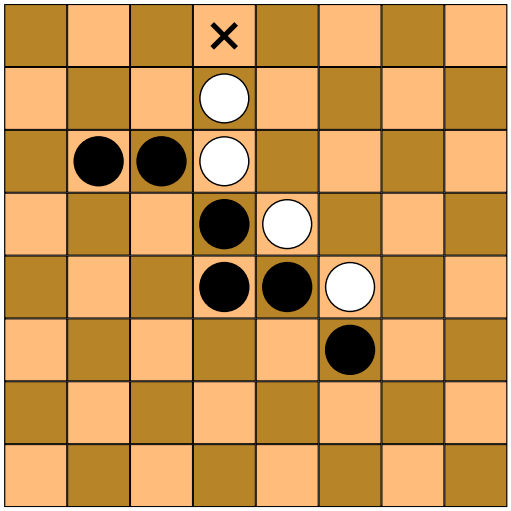
2.4.2 Othello

Branching factor: 10
Maximum depth: 60

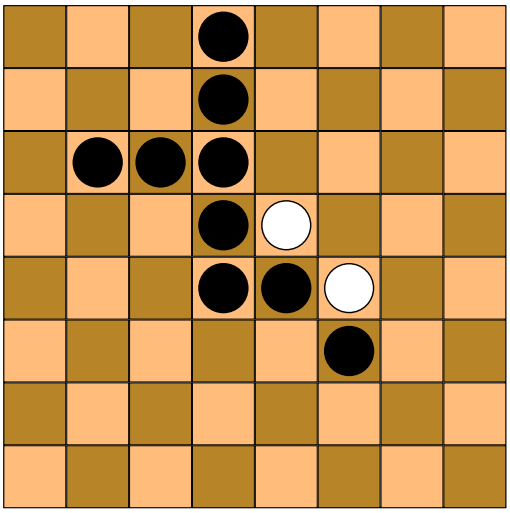
Othello is played on an 8 by 8 chess-like board. Starting with 4 placed stones in the middle, 2 of each color, placed criss-cross (see Figure 6(a)), each player tries to "capture" enemy stones by trapping them between stones of his own color (see Figure 6(b)). Every captured stone is turned and becomes one of the capturing player's color (see Figure 6(c)). In each move, the active player has to place a stone adjacent to an already placed stone and capture at least one enemy stone. If there are no moves available for a player, his turn is skipped. If both players have no possible moves left or the board is full, the game is over and the player with the most stones on the board wins. The difficulty is, to get so called "safe" stones (e.g. in the corners). These stones can't be trapped and adjacent stones of the same color are safe as well.



(a) Starting setup, possible moves for black are marked with X



(b) A possible move for black, marked with X



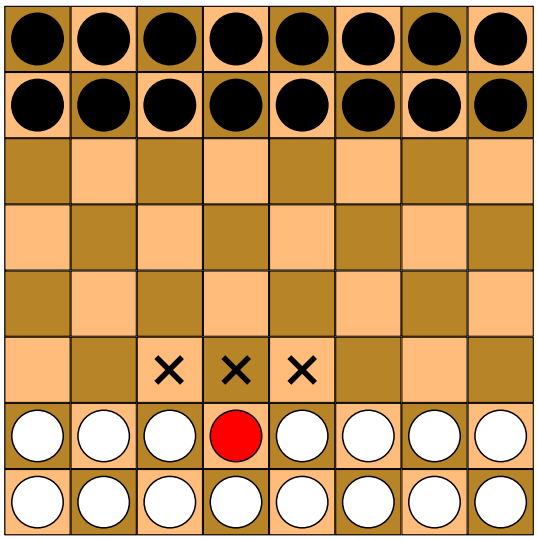
(c) Board after the capture from (b) was played

Figure 6: A typical board of Othello

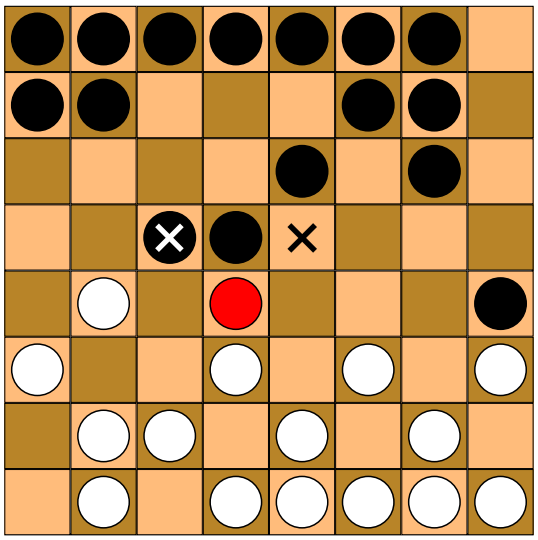
2.4.3 Breakthrough

Branching factor: 20
Maximum depth: 178

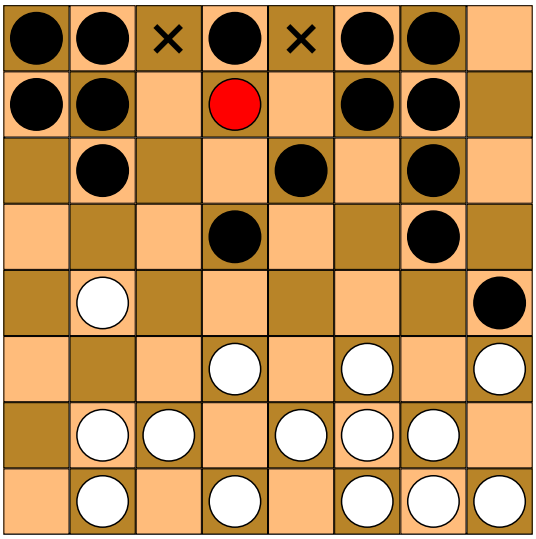
Breakthrough is played on a 8 by 8 chess-like board. Both players start with 16 stones on their closest two rows (see Figure 7(a)). The goal is to break through to the opponents side (see Figure 7(c)), either by eliminating opponent stones or by finding a path between them (see Figure 7(b)) without getting captured. Stones can only be moved forward and one step per turn: straight, if the next place is free and diagonal if the next diagonal place is free or occupied by an opponent stone. In the last case, the opponent's stone is captured (removed from the game). Even though these rules are quite simple, the amount of possible moves leads to very complex board situations.



(a) Starting setup, a set of possible moves for white is marked with X



(b) A possible move for white. Capture to the left and simple move to the right

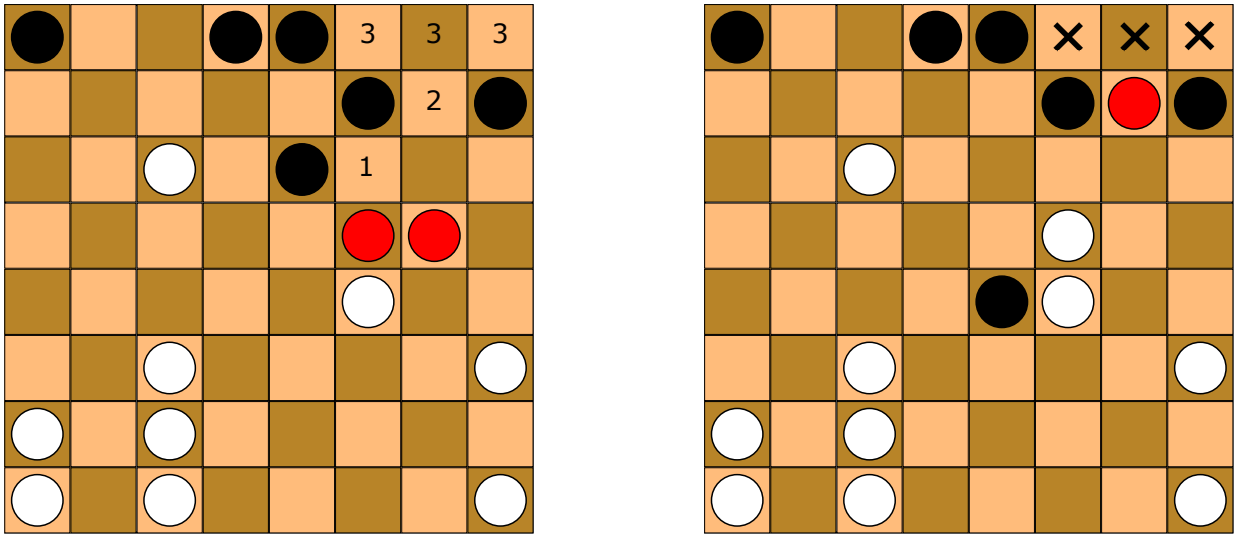


(c) Possible win for white, marked with X

Figure 7: A typical board of Breakthrough

A very important characteristic of Breakthrough, is the high local symmetry: There are often different moves, that achieve the same position, or at least positions, that are similar enough to state, that the

moves are essentially equal. Figure 8 shows two examples. In the first situation, both red marked stones of white can move to position 1, which leads to a guaranteed victory, following the steps 2 and 3. The second situation is from the same game, but four turns later. White has now three opportunities to win the game instantly.
In general, a winning moves in Breakthrough can always be executed on at least two ways. Only exception is a stone, placed on the most left, respectively the most right column, with an opponent stone right in front of it (see Figure 9).



(a) Two stones for white (marked red) can achieve a guaranteed win (following moves marked by numbers)

(b) White can win with 3 different moves (stone marked red, winning positions marked with X)

Figure 8: Two examples for local symmetry in Breakthrough in a single game.

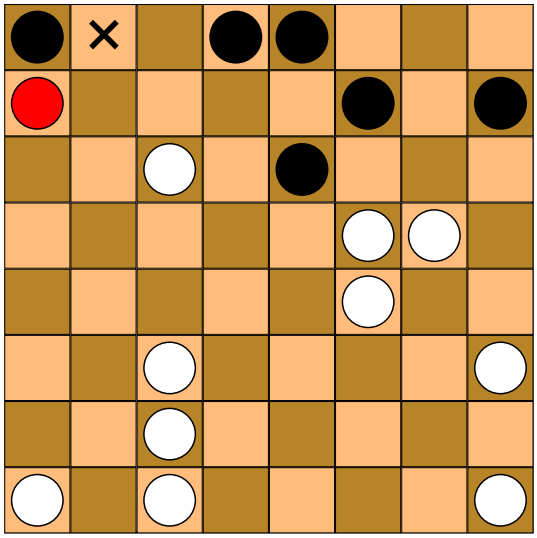


Figure 9: An example for a single winning move without direct alternatives. White has only one option (stone marked red, target position marked with X), to win the game on this turn.

3 Methods

The following section describes our main approaches to combine UCT with the pruning criterion of B*. Although all presented algorithms are developed for two-player, zero-sum games, there should be no reason, why the concepts could not be applied to different scenarios.

3.1 Applying B* Pruning to Basic UCT

The idea of B*, to view the node's values as intervals rather than fixed numbers, seems to fit the general idea of UCT quite well, since UCT works with confidence intervals (even though only the upper bound is used normally). The lower bound of each interval is given by replacing the "+" between the exploitation and the exploration part of the UCT formula (see formula 1) with a "—". The full interval I_{UCT} is then given by the following formula 4 and lies symmetrically around the payout.

$$I_{UCT} = \left[\frac{w_i}{n_i} - c\sqrt{\frac{\ln N_i}{n_i}}, \frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}} \right] \tag{4}$$

Before our different approaches for combining UCT with B* pruning are discussed, it has to be mentioned, that UCT and B* are technically in contradiction. B* specifically prohibits growing intervals, because an already pruned branch could become viable again otherwise, while UCT is based on the idea of growing and shrinking intervals, to balance exploitation and exploration. Regarding the fact, that random-based algorithms usually don't really claim to be optimal (UCT for example only claims to converge to an optimal solution, but has no evidence on whether the current solution is already the optimal one or not), we decided to ignore this problem. Hence, every following approach is not optimal and due to pruning of sub trees that could possibly contain the optimal solution, the algorithms won't even necessarily converge to an optimal solution. Still, our tests show a playing strength comparable to UCT.
In a first attempt to use the UCT intervals with the B* rules, we simply added an additional component to the selection phase of UCT, in which we first identify the highest lower bound and afterwards check all other upper bounds against it. Every upper bound below the highest lower bound is then considered a "hopeless" branch and the according sub tree is pruned (see Algorithm 4).

Algorithm 4 Selection in Basic UCT*

```
1:  procedure SELECT(Node node)
2:      Node currentNode = node
3:      while currentNode.isExpanded() and currentNode.hasChildren() do
4:          Float highestLowerBound = currentNode.getHighestLowerBoundInChildren()
5:          for all Node child in currentNode.getChildren() do
6:              if child.getUpperBound() < highestLowerBound then
7:                  currentNode.removeChild(child)                ▷ Actual pruning
8:              end if
9:          end for
10:         currentNode = currentNode.getChildWithHighestUCTValue()
11:     end while
12:     return currentNode
13: end procedure
```

If only one move is left, due to no other possible moves or pruning of all alternatives, the algorithm can stop and return the remaining one. Theoretically, this approach already fulfills the requirements, set for this thesis, since we added a domain-independent and dynamic termination criterion to UCT.
Experiments with this approach, however, provide very bad results (see section 4.2): Not only does the general iteration-per-second ratio go down due to the additional overhead per step, but the algorithm

prunes very rarely. The reason for this becomes clear, after a closer look on the upper bounds in UCT: The payout should converge against some true value, while the exploration part will continually increase, until the according move is considered again, hence the UCT value of each node will increase over time and eventually rise above the best (from a payout perspective) node's value. Since the algorithm always picks the node with the highest UCT value, the upper bounds harmonize and usually no upper bound gets below the highest lower bound (see Figure 10).

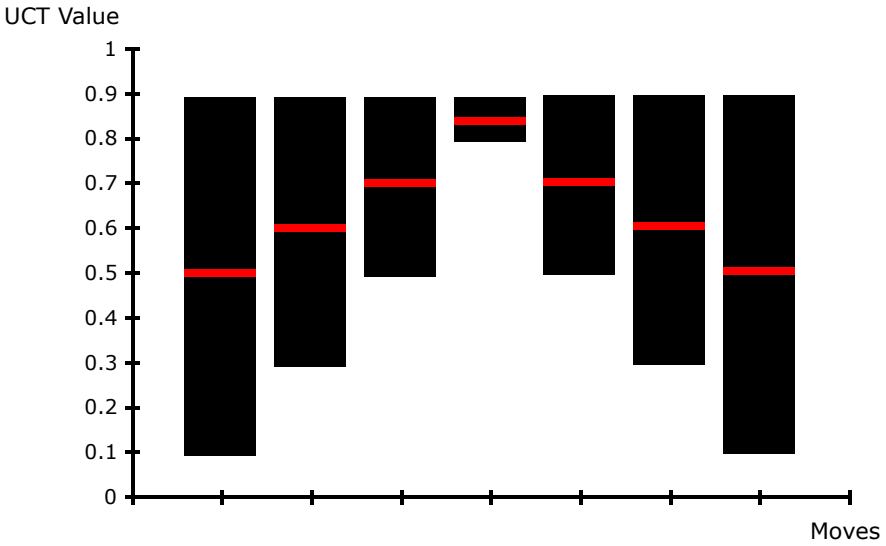


Figure 10: An idealized example for the harmonization of the upper bounds in UCT. The red bar indicates the current payout, the black bar shows the full interval.

3.2 Introducing a Pruning-Rate

The problem of harmonized upper bounds led to the idea, to separate the general UCT algorithm from the pruning by introducing a pruning-rate p. While the selection of the next child still uses the exploration rate c, the pruning step uses p to calculate upper and lower bounds (see Algorithm 5). This way, the harmonization of the upper bounds doesn't prevent separable intervals for pruning. The according pruning interval I_{prune} would be given by formula 5.

$$I_{prune} = \left[\frac{w_i}{n_i} - p\sqrt{\frac{\ln N_i}{n_i}}, \frac{w_i}{n_i} + p\sqrt{\frac{\ln N_i}{n_i}} \right] \tag{5}$$

Figure 11 shows an idealized example for this approach: Although the upper bounds of the UCT intervals still harmonize, the pruning intervals are separated much better. In the presented case, the search could even be terminated, since the pruning interval of the forth move is clearly above all other pruning intervals.

Since larger intervals, than the ones used for selection, would suffer even more from the harmonization, the pruning rate p should always be equal or smaller than the exploration rate c. With a decreasing p, the pruning gets more aggressive and possibly leads to termination much earlier, while raising the risk of non-ideal choices. The extreme cases would be $p = 0$, where the first move with a winning rollout or the last with a loosing rollout (which ever comes first) would instantly be picked, and $p = c$, in which case we would essentially get the basic UCT*, described above (see section 3.1).

This approach appears to be some sort of competitive, depending on chosen parameters. It still comes short to UCT without pruning, but the appearance of actual pruning steps is increased drastically. The reason why this version of UCT* can not challenge UCT seems to be the main characteristic of UCT: It generally does a good job at ignoring branches with low expectations, and even if a branch could be

Algorithm 5 Selection in UCT*

```
1: procedure SELECT(Node node)
2:     Node currentNode = node
3:     while currentNode.isExpanded() and currentNode.hasChildren() do
4:         Float highestLowerBound = currentNode.getHighestLowerBoundInChildren()   ▷ now
   using p
5:         for all Node child in currentNode.getChildren() do
6:             if child.getUpperBound() < highestLowerBound then                    ▷ now using p
7:                 currentNode.removeChild(child)                                   ▷ Actual pruning
8:             end if
9:         end for
10:        currentNode = currentNode.getChildWithHighestUCTValue()
11:    end while
12:    return currentNode
13: end procedure
```

pruned, the upper bound of this branch (and its payout) would be so far below the other upper bounds, that it wouldn't be considered nearly often enough in the future to justify the increased overhead to prune it in the first place and since the tree grows highly asymmetrically, the reduced memory is rather insignificant. At the same time branches can be pruned premature, if p is chosen too low, compared to c. The full evaluation can be found in section 4.4.

3.3 Increasing the Time between Prunings

To reduce the mentioned overhead, we decided to increase the time between pruning attempts. Since UCT usually runs with multiple thousands of iterations, but significantly less possible moves, it might not be necessary to look for possible prunings in every iteration. In Connect4 for example, it should be sufficient to look for prunable sub trees every 100 iterations, if we have an iteration-limit of 100,000.
The next approach takes this into account by using a new parameter $s > 0$, that determines the step-width, in which each node checks for possible prunings (see Algorithm 6). This parameter should be set dependent on the expected total number of iterations. With few iterations (like 2000), a step-width of 1000 would be extremely high (only one relevant pruning step), while the same step-width would be just fine with 100,000 iterations. If chosen too low, unnecessary calculation time would be spent (with $s = 1$ this would be UCT* as described in section 3.2).

This version of UCT* shows an improved performance compared to the first two. Increased time between pruning attempts solve the problem of increased calculation time pretty convincingly (compare section 4.3 for evaluation), without giving up the developed approach for pruning.

3.4 Restricting Pruning to the Root Node

Parameter tuning for UCT* with increased time between prunings shows a clear tendency to equal parameters c and p (see section 4.4). The closer p gets to c, the better the algorithm performs, which led to the assumption, that pruning doesn't really help UCT to select a better move or focus on better sub trees. From another perspective, these results show, that UCT* performs best, if it is reduced to plain UCT, with as few prunings as possible.
If pruning doesn't help the quality of UCT, its only purpose has to be early termination. Termination however is only achieved by pruning all but one child on the first layer, while pruning on all other layers would only hinder UCT, without contributing to early termination. By restricting the pruning to the root node, the algorithm would essentially be plain UCT with the possibility of early termination and a minimal overhead. Lowering p would lead to earlier termination, at the cost of possible premature prunings and weaker decisions.

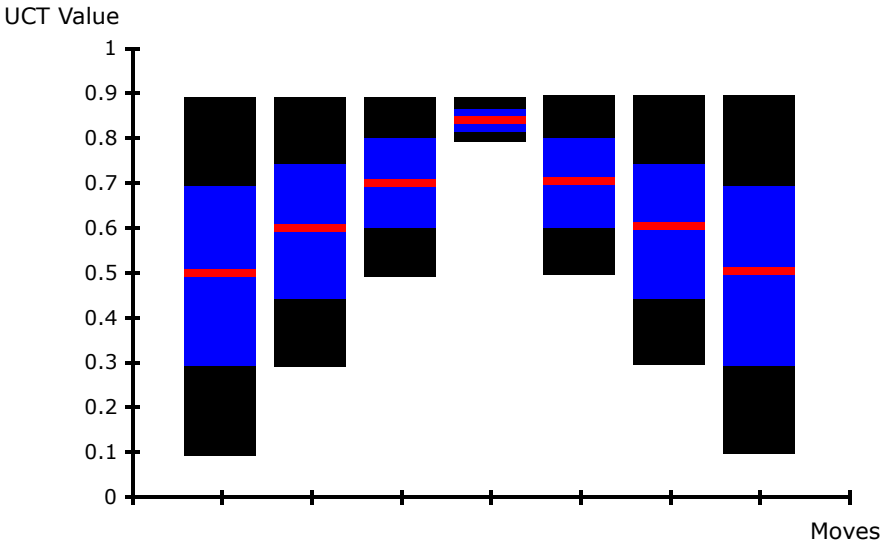


Figure 11: An idealized example for possible pruning in UCT* despite harmonization of the upper bounds. The red bar indicates the current payout, the black bar shows the full UCT interval with the exploration rate c, the blue bar represents the pruning intervals, given by the pruning rate p.

The final version of UCT* (see Algorithm 7) restricts pruning to the root node and should theoretically perform similar to UCT if $p = c$.

3.5 Applying B* Pruning to MCTS with Random Selection

After applying B* pruning to UCT, it seemed obvious, to do the same with an MCTS algorithm with random selection. The concept is close to the one UCT uses, since we try to separate profitable from non-profitable sub trees by using the UCT formula. The main difference is, that MCTS* doesn't accomplish this by focusing on any particular path, but simply prunes non-profitable ones (see Algorithm 8). While UCT* only prunes on the root node, MCTS* has to prune on all layers, due to the random selection and missing focus on "good" sub trees. For the same reason, the step-width s between prunings should be chosen smaller than in our final version of UCT*.
The advantage of this approach would be the termination criterion that comes with B* pruning, while having a possibly rapid selection phase (dependent on the step-width s).

Algorithm 6 Selection in UCT* with increased time between prunings

```
1:  procedure SELECT(Node node)
2:      Node currentNode = node
3:      while currentNode.isExpanded() and currentNode.hasChildren() do
4:          if node.getVisits() mod s == s - 1 then          ▷ Only prune after s iterations
5:              Float highestLowerBound = currentNode.getHighestLowerBoundInChildren()
6:              for all Node child in currentNode.getChildren() do
7:                  if child.getUpperBound() < highestLowerBound then
8:                      currentNode.removeChild(child)
9:                  end if
10:             end for
11:         end if
12:         currentNode = currentNode.getChildWithHighestUCTValue()
13:     end while
14:     return currentNode
15: end procedure
```

Algorithm 7 Selection in UCT* root-node-exclusive pruning

```
1:  procedure SELECT(Node node)
2:      Node currentNode = node
3:      while currentNode.isExpanded() and currentNode.hasChildren() do
4:          if node.getParent() == null and node.getVisits() mod s == s - 1 then   ▷ Prune only at the
    root node
5:              Float highestLowerBound = currentNode.getHighestLowerBoundInChildren()
6:              for all Node child in currentNode.getChildren() do
7:                  if child.getUpperBound() < highestLowerBound then
8:                      currentNode.removeChild(child)
9:                  end if
10:             end for
11:         end if
12:         currentNode = currentNode.getChildWithHighestUCTValue()
13:     end while
14:     return currentNode
15: end procedure
```

Algorithm 8 Selection in MCTS* with increased time between prunings

```
1:  procedure SELECT(Node node)
2:      Node currentNode = node
3:      while currentNode.isExpanded() and currentNode.hasChildren() do
4:          if node.getVisits() mod s == s - 1 then
5:              Float highestLowerBound = currentNode.getHighestLowerBoundInChildren()
6:              for all Node child in currentNode.getChildren() do
7:                  if child.getUpperBound() < highestLowerBound then
8:                      currentNode.removeChild(child)
9:                  end if
10:             end for
11:         end if
12:         currentNode = currentNode.getRandomChild()
13:     end while
14:     return currentNode
15: end procedure
```

4 Evaluation

The following section discusses parameter-tuning and evaluation of UCT and UCT*. Due to an early implemenation error, that was noticed in the last days of this thesis, we use a slightly different UCT formula (see formula 6), respectively a different formula for calculating the pruning intervals I_{prune} (see formula 7). The only difference is, that the exploration rate c (respectively the pruning rate p) moved inside the square root. This doesn't affect the behavior of UCT and UCT* at all, but it is important to note, that all tuned and evaluated values for c and p are basically c^2, respectively p^2.

$$U_i = \frac{w_i}{n_i} + \sqrt{c * \frac{\ln N_i}{n_i}} \tag{6}$$

$$I_{prune} = \left[\frac{w_i}{n_i} - \sqrt{p * \frac{\ln N_i}{n_i}}, \frac{w_i}{n_i} + \sqrt{p * \frac{\ln N_i}{n_i}} \right] \tag{7}$$

4.1 Parameter-Tuning

UCT is highly dependent on well chosen parameters and passes this characteristic to UCT* and MCTS*. For example, UCT* with an exploration rate of $c = 0.5$ and a pruning rate of $p = 0.25$ outclassed UCT* with an exploration rate of $c = 4$ and a pruning rate of $p = 0.5$ in 50 matches of Connect4 with a score of 47 to 3. Parameters are dependent on the game and the maximum number of iterations, so each algorithm has to be tuned for each game specifically. The maximum number of iterations is set to a fixed value for each game, regardless of the tuned algorithm (100,000 for Connect4, 25,000 for Othello and 10,000 for Breakthrough).
Since tuning multiple algorithms - one with quadratic complexity - is extremely time consuming, it was executed on a cluster system, provided by the Knowledge Engineering group in Darmstadt. The implementation was done in plain, single threaded java and up to 13 games could be played simultaneously on the three available cluster-nodes.
The first tuning results for UCT are shown in Table 1. While the best c for Connect4 is inside the testing-frame, Othello and Breakthrough show a clear tendency towards the lowest exploration rate. For these two games, a second round-robin-tournament with lower exploration rates was played (see Table 2).

Based on the win rates, the exploration rates shown in Table 3 are used for UCT in the following evaluation.

Connect4	
c	Win rate
0.125	0.52
0.25	0.624
0.5	0.708
1	0.622
2	0.374
4	0.152

Othello	
c	Win rate
0.125	0.804
0.25	0.718
0.5	0.538
1	0.398
2	0.284
4	0.258

Breakthrough	
c	Win rate
0.125	0.716
0.25	0.572
0.5	0.544
1	0.44
2	0.332
4	0.396

Table 1: Parameter-tuning of the exploration rate c for UCT from 0.125 to 4. 50 games for each fixture were played in a round-robin-tournament.

Othello	
c	Win rate
0.005	0.378
0.01	0.488
0.02	0.592
0.04	0.534
0.08	0.56
0.125	0.448

Breakthrough	
c	Win rate
0.005	0.248
0.01	0.308
0.02	0.384
0.04	0.564
0.08	0.712
0.125	0.784

Table 2: Parameter-tuning of the exploration rate c for UCT from 0.005 to 0.125. 50 games were played for each fixture in a round-robin-tournament.

Game	c
Connect4	0.5
Othello	0.02
Breakthrough	0.125

Table 3: Tunes exploration rates c for all three mentioned games.

Due to the fixed rates, the chosen parameters are not necessarily optimal, but since the further tuning is based on these values, non-optimal choices are inherited, hence no algorithm has an unfair advantage.

4.2 Basic UCT*

This section discusses the quality of basic UCT*, mentioned in section 3.1. In a quick performance-comparison in Connect4 against UCT, UCT* accomplishes a score of 22 to 28 in 50 games. While UCT averages about 1,613,208 iterations per game, UCT* only needs 1,330,741. This, however, comes at the cost of calculation-time: UCT takes only 11.56 seconds per game, while UCT* has an average time of 13.31 seconds. Since time is the more important measurement and the overall score is in favorite of UCT, basic UCT* doesn't provide any advantages.
The full results of the competition are shown in Table 4. In Breakthrough, the average time and number of iterations is slightly in favor of UCT*, but the win rate is far from even. In Othello, the results are not really expressive: Even though the time and iteration save is very significant, UCT* comes short to UCT by a huge margin.

Connect4 ($c = 0.5$)

Algorithm	Win rate	Avg. iterations	Avg. time / s
UCT	0.56	1,613,208	11.56
Basic UCT*	0.44	1,330,741	13.31

Othello ($c = 0.02$)

Algorithm	Win rate	Avg. iterations	Avg. time / s
UCT	0.89	731,501	77.41
Basic UCT*	0.11	81,101	4.20

Breakthrough ($c = 0.125$)

Algorithm	Win rate	Avg. iterations	Avg. time / s
UCT	0.7	297,400	47,87
Basic UCT*	0.3	272,489	45,16

Table 4: Results and performance measurements for basic UCT* against UCT in all three mentioned games. Iterations and time are averaged over 50 played games.

In a first conclusion it can be stated, that the main concept works: UCT can be terminated early, using intervals. This, however, comes with a loss in playing strength and an increased seconds-per-iteration ratio.
By taking a closer look at single iterations, two main problems can be identified: Harmonization of upper bounds in UCT (see Figure 12) and an undersized parameter c (see Figure 13(a)). While harmonization only leads to unsuccessful pruning attempts, the undersized exploration rate c leads to extremely random pruning early in the game. Since this problem lessens with additional iterations, it leads to a higher chance of pruning for the first explored moves. Especially the first move in Othello (Figure 13(b)) shows this problem: All 4 moves should be exactly equal, but in our example basic UCT* prunes the second move already at the start of the 3rd iteration, since the rollout for the first move returned a victory, while the rollout for the second move returned a loss. Move 3 and 4 still have an "infinite" UCT interval. The premature pruning explains the massive time save for UCT* against UCT, as well as the clear victory of UCT.

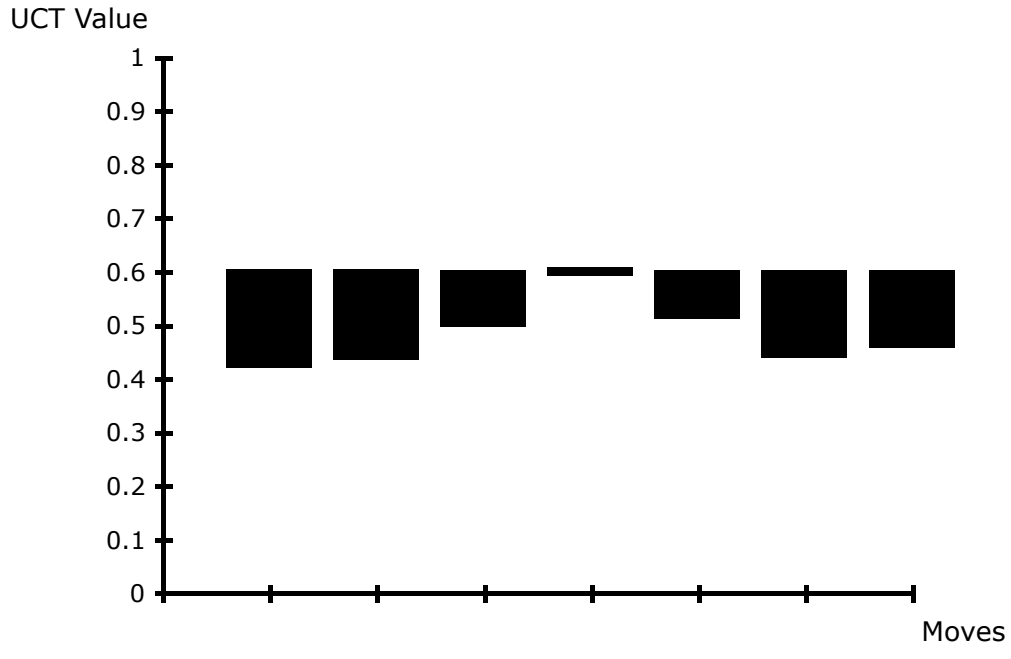
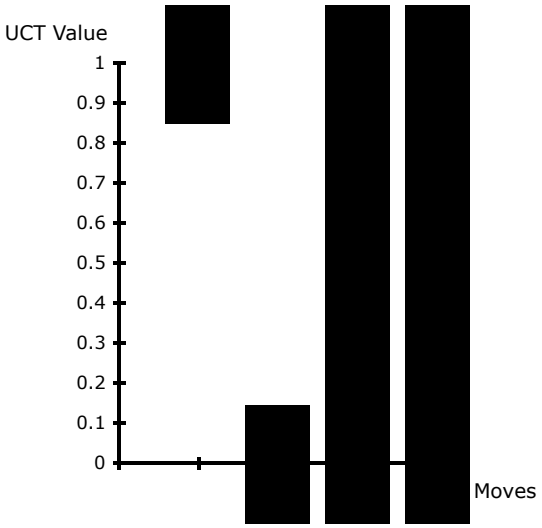
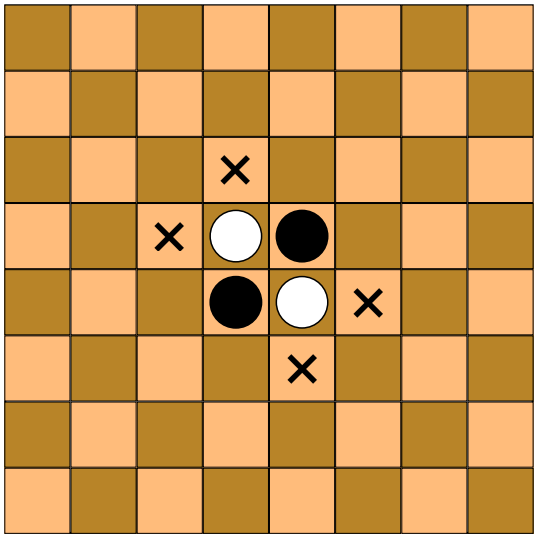


Figure 12: UCT intervals for the first move in Connect4 after 100,000 iterations.



(a) Intervals for Othello, first move, with an exploration rate $c = 0.02$ (after second iteration)



(b) Initial board in Othello. Possible moves for black are marked with X.

Figure 13: Example for premature pruning in basic UCT*

Looking at the first two iterations and assuming a positive rollout for the first and a negative rollout for the second, basic UCT* would prune the second move in the third iteration, if the following formula (8) holds true.

$$1 - c * \sqrt{\frac{\ln 2}{1}} > 0 + c * \sqrt{\frac{\ln 2}{1}} \tag{8}$$

This formula can be rearranged, to get a minimum pruning rate of 0.60 for the UCT formula (9). For the UCT formula, used in this evaluation, this would be $0.60^2 = 0.36$.

$$c < \frac{0.5}{\sqrt{\ln 2}} = 0.60 \tag{9}$$

Both Othello and Breakthrough suffer from an undersized exploration rate c, according to the limit $c = 0.36$ and the bad results, compared to UCT, are most likely a consequence of this.

4.3 Increased Time between Prunings

Although adding increased time between prunings was the third step of our development, its value will be discussed before actually evaluating different pruning rates. The reason for this is the increased (quadratic) tuning effort for different parameters c and p.
To evaluate the increased step-width s, 500 games were played for each of the three mentioned games with the UCT-tuned basic UCT* versus the same algorithm, but with an increased step-width of $s = 100$. The results (see Table 5) for Connect4 are pretty much as expected, since the iteration-per-game ratio goes up very slightly, while the average time per game decreases noticeably, when an increased step-width s is used. The win rate is even.
In Othello, the results show, that the problem with undersized values of c is implicitly solved or lessened. With an increased time before the first pruning attempt, UCT has time to play a lot more roll-outs and with that reduce the strong influence of variance. With an increased value of s, the problem should have less and less impact on the performance. Since basic UCT* with $c = 0.02$ and $s = 1$ is so far from being

competitive, it can be stated, that basic UCT* with very small exploration rates should never be executed without an increased step-width.
Breakthrough combines both results: The win rate is clearly in favorite of $s = 100$, while the average number of iterations and time only increased slightly. The distinct win rates indicate less premature prunings, although the difference is not as high as in Othello.

Connect4 ($c = 0.5$)			
s	Win rate	Avg. iterations	Avg. time / s
1	0.498	1,396,378	22.17
100	0.502	1,397,267	16.70
Othello ($c = 0.02$)			
s	Win rate	Avg. iterations	Avg. time / s
1	0.165	79,020	4.80
100	0.835	226,174	13.30
Breakthrough ($c = 0.125$)			
s	Win rate	Avg. iterations	Avg. time / s
1	0.24	256,835	78.80
100	0.76	268,123	81,27

Table 5: Results and performance measurements for basic UCT* with $s = 1$ against basic UCT* with $s = 100$ in all three mentioned games. Iterations and time are averaged over 500 played games.

Looking at all three results, an increased step-width appears to help UCT* in two ways: it reduces overhead and lessens the problem with undersized parameters. Hence, in the following evaluation steps, a step-width of $s = 100$ is used. This value is not tuned, so it is most likely not optimal for any of the three games, but it is good enough to show and evaluate the basic concept.

4.4 UCT*

This section is about the parameter-tuning and evaluation of UCT* as described in section 3.3. Figure 14 shows the exploration and pruning intervals for the first move in Connect4 after 1398 iterations with an exploration rate of $c = 0.5$ and a pruning rate of $p = 0.25$. In the next iteration, the last move will be pruned, since its upper pruning bound is below the lower pruning bound of the center move and since it is the nect iteration is the 1399th, UCT* would attempt pruning with a step-width of $s = 100$. This example shows, that the concept works as predicted, and pruning is possible, despite fairly harmonized upper bounds in UCT.

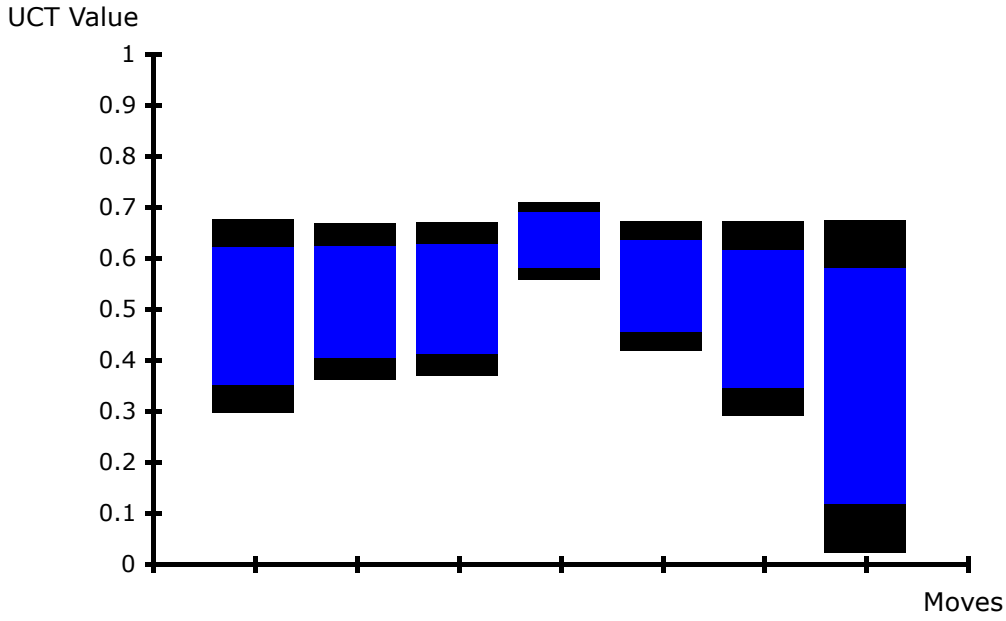


Figure 14: An example for pruning in UCT*. The graph shows the exploration interval (black) and the pruining interval (blue) for UCT* with $c = 0.5$ and $p = 0.25$ for the first move of Connect4 after 1399 iterations.

In a first attempt to tune c and p, a round-robin-tournament with all parameter pairs (c, p), that satisfy the following condition 10, was played.

$$(c, p) \in \{0.25, 0.5, 1, 2, 4\} \times \{0.125, 0.25, 0.5, 1, 2\}, \text{ with } p < c \tag{10}$$

After playing 50 games for each fixture, with a step-width of $s = 100$ (see Table 6), two observations can be made: Connect4 tends towards $p = c$, with $c = 0.5$ yielding the best result. (Note, that $c = 0.5$ is the same value, that the parameter-tuning for UCT in section 4.1 showed). Othello and Breakthrough on the other hand suffer from a wrong testing-frame (also similar to UCT-tuning).

Connect4 ($s = 100$)

c	0.25	0.5		1			2				4				
p	0.125	0.125	0.25	0.125	0.25	0.5	0.125	0.25	0.5	1	0.125	0.25	0.5	1	2
WR	0.49	0.52	0.78	0.48	0.70	0.76	0.34	0.56	0.57	0.57	0.25	0.35	0.39	0.37	0.37

Othello ($s = 100$)

c	0.25	0.5		1			2				4				
p	0.125	0.125	0.25	0.125	0.25	0.5	0.125	0.25	0.5	1	0.125	0.25	0.5	1	2
WR	0.76	0.72	0.53	0.67	0.53	0.39	0.65	0.53	0.4	0.31	0.64	0.51	0.36	0.28	0.23

Breakthrough ($s = 100$)

c	0.25	0.5		1			2				4				
p	0.125	0.125	0.25	0.125	0.25	0.5	0.125	0.25	0.5	1	0.125	0.25	0.5	1	2
WR	0.69	0.56	0.59	0.49	0.52	0.5	0.45	0.52	0.48	0.48	0.43	0.44	0.43	0.46	0.46

Table 6: Results for UCT* with different parameters c and p and a step-width of $s = 100$ for all three mentioned games. 50 games were played for each fixture in a round-robin-tournament.

To confirm the assumption, that UCT* performs best with a UCT-tuned c and $p \to c$, another tournament of Connect4, with $c \in \{0.5, 1\}$ and converging values of p, was played (see Table 7). First, it can be seen, that $c = 0.5$ yields the best result, which matches the outcome of Table 6. Additionally, $p = c$ shows

the best performance. To substantiate the last observation, the UCT-tuned c parameters with converging values of p were tested in Othello and Breakthrough (see Table 8), with similar results.

Connect4 ($s = 100$)

c	0.5			1		
p	0.25	0.375	0.5	0.5	0.75	1
Win rate	0.48	0.54	0.63	0.42	0.47	0.46

Table 7: Second tuning-attempt for UCT* with different parameters c and p and a step-width of $s = 100$ in Connect4.

Othello ($c = 0.02$)

p	Win rate
0.005	0.37
0.01	0.42
0.015	0.55
0.02	0.65

Breakthrough ($c = 0.125$)

p	Win rate
0.005	0.18
0.01	0.27
0.02	0.37
0.04	0.55
0.08	0.77
0.125	0.85

Table 8: Second tuning-attempt for UCT* with an UCT-tuned exploration rate c and different pruning rates p in Othello and Breakthrough.

Summarizing it can be stated, that UCT* shows best results with an UCT-tuned parameter c, and that the playing strength decreases, with a decreasing p. For the highest playing strength, $p = c$ appears to be appropriate.

4.5 Final UCT*

In this final part of the evaluation for UCT*, we discuss the performance of UCT* with a step-width of 100 and root-node-only pruning, by looking at the results of UCT* versus the original UCT algorithm. Besides the win rate, we also observe the average number of iterations and time.
First, 50 games of Connect4 for each pruning-rate $p \in \{0.3, 0.35, 0.4, 0.45, 0.5\}$ and a step-width of $s = 100$ were played, and 500 games for the same fixtures, but with an increased step-width of $s = 1000$ (see Table 9). Both tests show a trade-off between playing strength and average number of iterations. The peaks at $p = 0.4$ are most likely caused by variance, since $p = 0.4$ fell short to $p = 0.5$ in a direct face-off with 1000 played games.

Connect4 ($c = 0.5, s = 100$, 50 games)

p	Win rate	Avg. iterations
0.3	0.47	1,108,438
0.35	0.49	1,222,564
0.4	0.52	1,245,594
0.45	0.51	1,319,550
0.5	0.59	1,369,552

Connect4 ($c = 0.5, s = 1000$, 500 games)

p	Win rate	Avg. iterations
0.3	0.511	1,114,518
0.35	0.485	1,205,296
0.4	0.54	1,240,894
0.45	0.517	1,306,282
0.5	0.529	1,390,550

Table 9: Results for UCT* with different pruning rates p versus UCT in Connect4.

Next, 1000 games of Connect4, Othello and Breakthrough between UCT* with $p = c$ and UCT were played (see Table 10), to evaluate the performance of UCT* compared to UCT with the win-rate-wise

strongest settings. The step-width was set back to 100. Connect4 and Breakthrough show a pretty even win rate with time and iterations saves for UCT*. The reason for the better results in Connect4 are most likely the complexity and local symmetry of Breakthrough. UCT* still has a very bad win rate in Othello, but compared to the results of basic UCT* (see Table 4 in section 4.2), a clear improvement can be seen. Even with an increased step-width of 100, premature pruning is still a problem, for an exploration/pruning rate as low as 0.02.

Connect4 ($c = 0.5$)

Algorithm	Win rate	Avg. iterations	Avg. time / s
UCT	0.47	1,730,000	19.22
UCT*	0.53	1,398,791	16.21

Othello ($c = 0.02$)

Algorithm	Win rate	Avg. iterations	Avg. time / s
UCT	0.70	731,676	106.96
UCT*	0.30	96,110	10.74

Breakthrough ($c = 0.125$)

Algorithm	Win rate	Avg. iterations	Avg. time / s
UCT	0.506	283,530	84.60
UCT*	0.494	269,830	81.88

Table 10: Results and performance measures for UCT* versus UCT with a step-width of $s = 100$. Iterations and time are averaged over 1000 played games.

Overall, the final version of UCT* shows a comparable strength to UCT, while accomplishing noticeable time/iteration saves. In the case of Connect4, UCT* even outperforms UCT in several test runs. The reason has to be a slight advantage due to pruned non-profitable branches. Even though this effect is very small, it is most likely not caused by variance, given the amount of played games. In Breakthrough, the local symmetry prevents early termination (and with it time/iteration saves) in most cases, but the win rate is still even. In all three games we can observe a massive improvement compared to the first, basic UCT* attempt.

4.6 MCTS*

Since MCTS* is nothing more than a byproduct of UCT*, only a single evaluation round, with UCT-tuned parameters (Table 11) was run. The results show, that MCTS* isn't able to challenge UCT, although early termination and pruning is possible.
Othello shows similar results to UCT*. Due to the low pruning rate, a massive time and iteration save can be seen, while UCT dominated win-rate-wise. Breakthrough shows only few successful early termination attempts, also similar to UCT*. The iteration-per-second ratio is slightly higher in MCTS*, but the algorithm doesn't benefit from it.

Connect4 ($c = 0.5$)

Algorithm	Win rate	Avg. iterations	Avg. time / s
UCT	0.907	1,455,200	16.67
MCTS*	0.093	862,210	8.03

Othello ($c = 0.02$)

Algorithm	Win rate	Avg. iterations	Avg. time / s
UCT	0.838	730,251	106.32
MCTS*	0.162	77,330	8.60

Breakthrough ($c = 0.125$)

Algorithm	Win rate	Avg. iterations	Avg. time / s
UCT	0.81	280,360	87.74
MCTS*	0.19	245,573	76.75

Table 11: Results and performance measures for MCTS* versus UCT with a step-width of 100 (500 games). Iterations and time are averaged over 500 played games.

MCTS* in the presented form is not competitive at all. However, variations could yield much better results. We will discuss some of these in the Future Work section (see section 5.2.4).

5 Conclusions and Future Work

5.1 Conclusions

Summarizing, it can be stated, that it is possible, to terminate UCT early, using its confidence intervals and applying the B* pruning rules. The exploration-rate for UCT* was in the three evalutated games almost the same as in plain UCT, while the pruning-rate controlled the aggressiveness of the termination. If $p = c$, the resulting algorithm's strength is comparable to UCT, while the benefit of early termination and pruning is relatively small.
When working with exploration/pruning rates < 0.60, an increased step-width is strongly advised, to prevent pruning in the third iteration. An increased step-width however, doesn't seem to hinder UCT* at all, so it should be always used and its extent should be dependent on the expected maximum number of iterations. Pruning on its own doesn't seem to help UCT, so it should be restricted to the root-node. In this case, the additional overhead, compared to UCT, is minimal, while early termination is still possible. UCT* doesn't terminate early if confronted with symmetry/similar moves, however, due to the minimal overhead, it still performs on the same level as UCT. The main problem of UCT*, compared to UCT and other MCTS algorithms, is the lost convergence to an optimal solution.
MCTS*, on the other hand, doesn't yield any value in its current form.

5.2 Future Work

In the following section, some problems of the current concepts/implementation will be addressed, as well as possible solutions for some of them. Additionally, extensions and variations for UCT* and MCTS*, that could yield improved performance, will be discussed.

5.2.1 Implementation

UCT* appears to suffer from an increased overhead. Overhead, however, is not necessarily caused by the algorithms, but also lies in the implementation. The implementation, used for this thesis, focuses on observablity, rather than performance, which leads to possibly needless calculations. A better, computation-sensitive approach could show even better results for UCT*.

5.2.2 Additional Evaluations

In this thesis, we focused on the general playing-strength of UCT* as well as its problems. But we didn't exactly search for situations, in which UCT* might outperform UCT consistently. That these situation exist is indicated by the positive win rate of UCT* in Connect4 (see section 4.5).
Another possibly interesting experiment would be to test UCT* in games without a maximum number of moves and with a limited roll-out depth. These games should result in a lot of simulated draws in the early stages, which could hinder pruning later on. Games with random elements or non-perfect information could be worthwhile as well.

5.2.3 Extensions for UCT*

UCT* showed problems regarding symmetry/similar moves. In some cases - like the first move in Othello - multiple moves are converging to the exact same value and it shouldn't matter, which of the presented actions is chosen. If it is possible, to identify similar moves, considering only the UCT intervals, no expert/domain specific knowledge would be necessary, to prune/terminate in these situations, since one of the equal moves could be randomly picked. This, however, has to be done very carefully, since premature prunings of possibly high- but still underrated moves, could be disastrous regarding the playing strength. This approach could solve the obvious problem in Breakthrough.
Another possible extension would be back propagation of already solved partial trees. If all children of

a are terminal nodes, or at least one node guarantees a win for the according player, no roll-outs have to be generated anymore, and the result, can be back-propagated instantly. The node itself can then be seen as a terminal node, which leads to solved partial trees, growing from the bottom. Next to time saves, this could also help UCT* separating moves even clearer, which than would possibly lead to earlier termination.
The next variation addresses the formula, used for pruning. The UCT formula doesn't take already received results into account, calculating the confidence interval. If a move is evaluated 10 times with a win, it's interval still has the same dimensions as if it lost 10 times. The only difference is the pivot point, the current payout. With another formula, that takes previous roll-outs into account, UCT* could possibly prune even more accurate.

5.2.4 Variation of MCTS*

The current version of MCTS* shows really bad results, if compared to UCT. The reason is very likely the completely random selection. If we replace this selection with a new approach, that uses the two strategies of B* (prove-best and disprove-rest), while still pruning with the UCT formula, we would get a directed search, with active disproving of non-promising sub trees.This new algorithm might have potential on the same level as UCT with better termination than UCT*. The UCT formula could also be replaced by any other formula for variance, as discussed earlier.

5.2.5 Résumé

This thesis shows, that the main concept of pruning in MCTS/UCT is not only possible, but can even lead to improved performance, compared to UCT. If UCT*-favorable situations can be identified and reinforced with suitable extensions, UCT* and even MCTS* could have potential, especially in scenarios, where time is a critical factor.

References

[1] Ronald W. Moore Donald E. Knuth. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence 6 (4)*, pages 293–326, 1975.

[2] Bruce Abramson. The Expected-Outcome Model of Two-Player Games. 1987.

[3] Paul Fischer Peter Auer, Nicolò Cesa-Bianchi. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning 47*, pages 235 – 256, 2002.

[4] Csaba Szepesvári Levente Kocsis. Bandit based Monte-Carlo Planning. *Machine Learning: ECML 2006: 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, pages 282–293, 2006.

[5] Hans Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence 12 (1)*, pages 23 – 40, 1979.

[6] Andrew J. Palay. An experimental analysis of the B* tree search algorithm. 1980.

List of Figures

List of Tables

List of Algorithms