
PROJECT REPORT

# Monte Carlo Tree Search for playing Go

ausgeführt am

Institut für
Analysis und Scientific Computing
TU Wien

unter der Anleitung von

**Assoz. Prof. Dipl.-Ing. Dr. techn. Clemens Heitzinger**

durch

**Sebastian Ertel**
Matrikelnummer: 1328851
Pfalzauerstraße 1a/3/16
3021, Pressbaum

Wien, am 10.09.2018 x

# Contents

# 1 Introduction

Go is an ancient Chinese board game, in which two players alternately place stones on a board, in order to surround as much territory as possible.
Following the dominance of computer programs over human players in chess during the late 1990s to the early 2000s, it was seen as the grand challenge of AI research. This status remained until 2015 to 2017, when a series of programs by the Google subsidiary Deepmind beat several professional human players.

In this project, which is to a large degree inspired by the book 'Deep Learning and the Game of Go' by Pumperla and Ferguson [DeepGo], we aim to build a Go framework and several Go playing agents.
Its main
In the second chapter we discuss our implementation of a Go framework and its improvements compared to the one found in [DeepGo].

The third chapter is dedicated to random bots and how they can be used for move value estimation via Monte Carlo Simulation.

In the fourth chapter we introduce Monte Carlo Tree Search, a very general concept of move selection in board games. We will see how its performance can be improved and ways to parallelize it.

Finally in the fifth chapter we explore the improvement of Monte Carlo Tree Search through Rapid Action Value Estimation and the usage of heuristics for node initialization.

The code described in this report can be found in the github repository MCTS for Go .
The Go framework (everything necessary to play a game of Go) is found in the module fast_goboard.jl. The different agents we implemented and their auxiliary functions found in fast_agent.jl.
The module play_go.jl simulates a game between two bots and graphically displays it.Furthermore the repository includes all setups for our experiments and some earlier versions of our implementation (the one that is used for speed comparisons to the current version is goboard.jl)

# 2 The Go framework

## 2.1 Rules

Go is a turn based board game, in which two players, black and white, alternately place a stone of their color on the intersections/nodes of a $19 \times 19$ grid. Of course stones can only be placed on intersections that are not occupied by other stones and they can also not be moved once they were placed on the board.

**Remark 1.** *The board does not necessarily have to be a* $19 \times 19$ *grid. A gridsize of* $13 \times 13$ *or* $9 \times 9$ *is also very common as they allow quick games for beginners.*
*Infact the game can be played on any sufficiently large grid, without any adaptations of the rules. We will make use of this, as we will often test our algorithms on a* $7 \times 7$ *grid.*

Two stones of the same color that were placed on adjacent nodes of the grid are connected. A set of connected stones is called a string or a chain. I.e. a string is a maximal set of stones of the same color such that for all stones of that string, there is a series of adjacent stones in the string, that connects these two stones.
Strings are practically treated as one unit. This is manifested in the liberties of a string, which are those empty nodes of the grid, that are adjacent to one of the stones of the string. Once a string has no liberties left, which means that all adjacent nodes are occupied by stones of the other color, it is removed from the board. The players can now again place stones on those gridnodes that were occupied by the removed string.
The act of removing a string of stones of the opposite color by placing a stone on its last liberty is called capturing those stones.
Of course single stones, that are not connected to any other stone of the same color are treated as strings.
Empty nodes that are completely surrounded by stones of a single string, i.e. that all adjacent nodes are occupied by stones of that string, are called eyes of the string.

**Remark 2.** *The board configuration can only change over time by stones being placed and set of stones being removed.*

The game is always started by the black player. In every turn, the current game state consists of the current board configuration and the player whose turn it is. This player can now either resign, pass the turn or place a stone on the board.

Players are free to place their stones on any node of the grid, that is not occupied by another stone, as long as the placed stone will have at least one liberty after the end of this

turn and a previous game state is not repeated. The last restriction is called Ko rule.
Thus a stone can be placed on a node, that is completely surrounded by stones of the opposite color and therefore will initially have zero liberties, as long as one of those neighbors will in turn be captured.
Another interesting consequence of this rule is, that strings with more than one eye can never be captured.

The game ends once either player resigns, or both players have passed their turn consecutively. Note that a player has to pass his/her turn if there are nodes left, where he/she can legally place a stone.

At the end of the game both players have to agree, which stones, that are still on the board could not avoid capture, if the play continued. These stones are called dead stones and they are removed from the board, before the game is scored. If there is disagreement between players, which stones are dead, the game usually continues, until it ends again.

Finally there are two major scoring systems, area scoring and territory scoring. In most cases the winner of a game will be the same under both methods. We will use area scoring throughout this project, as it is easier to implement. Hereby the score of a player is equal to the number of his/her stones that are on the board plus the number of nodes that are completely surrounded by his/her stones.
Futhermore the white player receives a point compensation, for playing second, called Komi. The Komi usually amounts to an additional 6.5 points for white.

**Remark 3.** *The exact Komi can also vary between rulesets. Note that as long as the compensation amounts to $x + 0.5$ points, ties can not happen. This will turn out specifically useful for the design of agents, as the outcome of each game can thus be seen as a random variable with a Bernoulli distribution.*

## 2.2 Implementation

In this section we want to briefly go through our implementation of a Go framework. It is to a very large degree based on the implementation described in [DeepGo]. As many well performing Go playing agents rely on a large number of simulations to choose their moves, a fast framework is of utmost importance.
We therefore chose to implement our framework, and thus also the agents described later, in Julia. To ensure good performance, we put special focus on the type stability of all variables and we kept all composite types (structs in Julia) immutable. Furthermore we tried to avoid creating deepcopies of structures.
This resulted in a significant speed up, compared to the direct translation of the fastest implementation found in [DeepGo]. In tests, random games were simulated roughly five times faster.

Unlike in [DeepGo], we chose to represent players by symbols and points by tuples instead of using composite types, which turned out to improve performance in tests.

Moves are represented by the composite type *Move*, that consists of three boolean fields that describe the action of the move (play a stone, pass the turn or resign) and a field *point* that stores the node at which the stone is placed. Note that we set point to $(-1, -1)$ when no stone is placed (instead of setting *point* to nothing), or $(-100, -100)$ when there is no move at all (instead of setting the *Move* variable to nothing). This is done to enforce type stability and by that speed up the implementation.

Like the name says, the structure *GoString* is used to represent strings of stones. It consists of the fields *color*, *stones* and *liberties*. The latter two are represented by sets of tuples. Thus the stones and liberties of a *GoString* can be changed although its fields are immutable. Furthermore we have implemented functions to add and remove liberties and to merge two strings. Note that the liberties of the merged string are defined by the formula

$$\text{merged(str1, str2).liberties} = \left(\text{str1.liberties} \bigcup \text{str2.liberties}\right) \setminus \left(\text{str1.stones} \bigcup \text{str2.stones}\right).$$

The board is represented by the composite type *GoBoard*, which consists of fields to store the number of rows and columns, the field *_grid*, which is a dictionary, that stores the string that each stone, present on the board, is a part of, and the $1 \times 1$ vector *_hash*, that stores the hash value of the current board configuration. *_hash* is a vector so that the hash value of the board can be changed, while keeping the structure *GoBoard* immutable.

The hash value of the board configuration is attained through Zobrist hashing. This is a special kind of hash function generation, that is often used for abstract board games such as chess and Go.
First a random hash value is generated for every possible move, i.e. for every point on the grid two values, one for a black stone and one for a white stone, are generated. The hash values are always unsigned 64 bit integers (type *UInt64*).
The empty board has the hash value zero. Each time a stone is placed on the board, the hash value of the corresponding move is applied to the hash value of the board, through the xor operator, i.e. the uptdate formula is

$$\text{xor}(\text{hash}(move), \_hash) \longrightarrow \_hash.$$

When a stone is removed from the board, the hash value of the move that put the stone on its position is again applied to the hash value of the board. This is reasonable as the xor operator is commutative, reflexive and has a zero diagonal, i.e. $\text{xor}(x, x) = 0$.

**Remark 4.** *We never check whether different board configurations have different hash values, we just assume that this is true, as the contrary is very unlikely.*

The structure *GoBoard* allows to quickly access the string, that a certain stone on the board is a part of, by simply entering the dictionary *_grid* with the respective node coordinates

$(x, y)$, by *_grid*$[(x, y)]$, or by using the get function on *_grid*.
When we want to place a stone on a specific node, we use the function *place_stone*. This function first loops through all the neighbors of the node and collects those that are occupied by stones of the other color, those that are occupied by stones of the same color and those that are empty in different sets.
Then it creates a new string only consisting of the placed stone and merges this string with all the strings that belong to the neighbors. The newly created string will now be the dictionary entry for each stone that belongs to it. Finally each string belonging to a neighboring stone of the opposite color has one liberty removed and will itself be removed form the board, if no liberties are left.

The last type that we created is the structure *GameState*. It consists of a field, that represents the board, a set *previous_state*, which store the hash values of all previous states, an array that stores the last two moves and an array, which stores the next player as a symbol. Again the field *next_player* is an array to keep the field mutable, while the gamestate remains immutable.

The reason why the type *GameState* stores the last two moves, is because the game is over if the last two moves were passes by the respective players.

We have implemented two versions of the function *apply_move*, which as the name says, changes the current gamestate according to the specified move. The first version is directly taken from [DeepGo], and creates a new variable of type *GameState* from the input gamestate, the second version just changes the fields of the input gamestate according to the move. The latter is much more efficient. This results in another major speedup compared to the implementation in [DeepGo].

We have furthermore implemented two functions that check the legality of moves. First the function *is_self_capture* checks, whether we attempt to place a stone on a point where it would have no liberties left at the end of the turn. The second function checks, whether a move would violate the Ko rule by comparing the hash value,the board would have afterwards, with those in the set of previous hash values.

Finally we have implemented two functions for determining the winner of a game. The first function is *area_scoring*, which simply returns the winner and the scores according to area scoring. Another version (called *area_scoring_graphical*) furthermore prints the board and marks, which nodes were scored for which player. Both functions simply count all stones of all strings in the string dictionary *_grid* (field of the *GameState* class) and determine all territories of empty stones (and the color by which they are surrounded) via Breadth First Search.

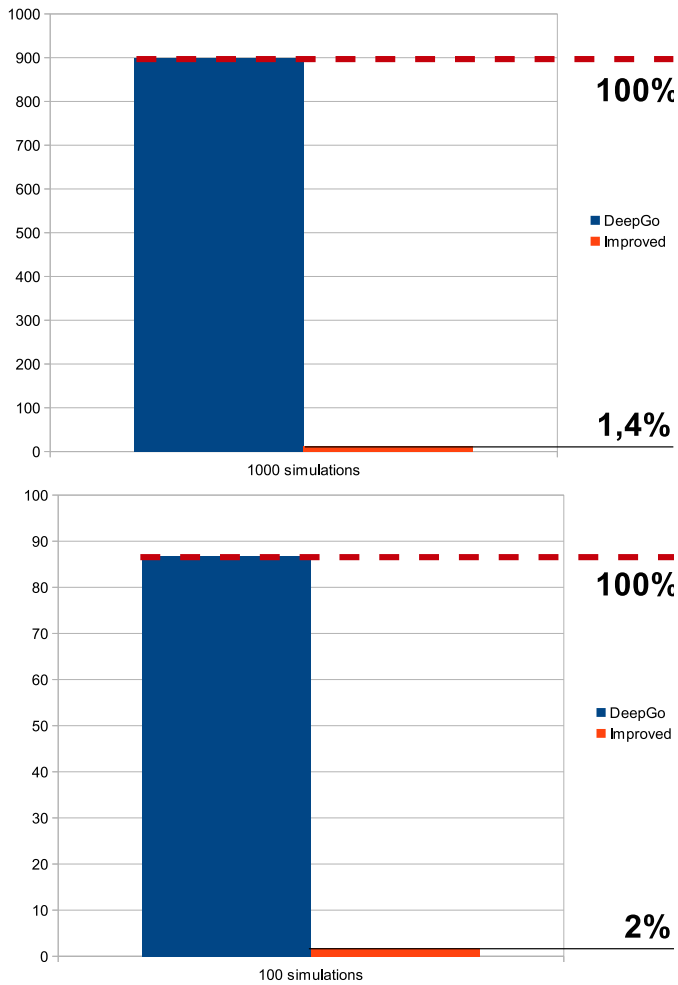
The second function is called *get_winner* and it determines the winner of a game, where the whole board is filled with stones and eyes, i.e. there are no empty nodes that are not eyes of a string. This situation is typical for the end of randomly simulated games and it allows us to determine the winner in a quicker and easier fashion. We will therefore use

*get_winner* to rapidly determine the winner of Monte Carlo simulations. And by that speed up the gameplay of many of our bots.

**Remark 5.** *Oddly there is no code for area scoring in current version of [DeepGo], that inspired this project.*

Finally we are left to show, that our changes to the implementation found in [DeepGo] beared fruit and significantly increased speed.

Figure 2.1: Speed comparison between the fastest implementation of [DeepGo] and our improved implementation




Here we compare the time (in seconds) required for 100/1000 game initializations and simulations of random games.

# 3 Random bots and Monte Carlo simulations

## 3.1 RandomBot

The first and easiest iteration in the design of Go playing agents is random play. Our random bot (simply called *RandomBot*) chooses its moves randomly among all possible legal moves, that do not fill out an eye of the same color. If there is no such move, our bot will pass its turn.

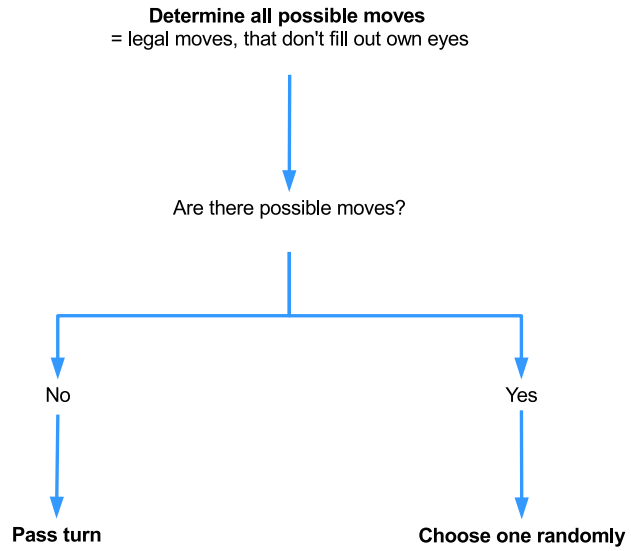



Figure 3.1: Decision diagram for *RandomBot*

**Remark 6.** *The reason why we only look for possible moves among those, that do not fill out any eyes, is that there is no possible scenario, where filling out ones own eyes would make sense, and that it significantly reduces the length of games between random bots. This will improve the performance of bots that use Monte Carlo simulation to evaluate moves.*

One possibilty to improve random bots would be to use special distributions for selecting a move (instead of the uniform distribution used in our implementation *RandomBot*). Another possibilty is to split up the selection process into several phases and in each only certain moves are considered.

## 3.2 RandomBot_with_coded_patterns

This is what we have implemented in a more advanced version of our random bot, called *RandomBot_with_coded_patterns*. It is an implementation of a random bot described in [ModUCT].

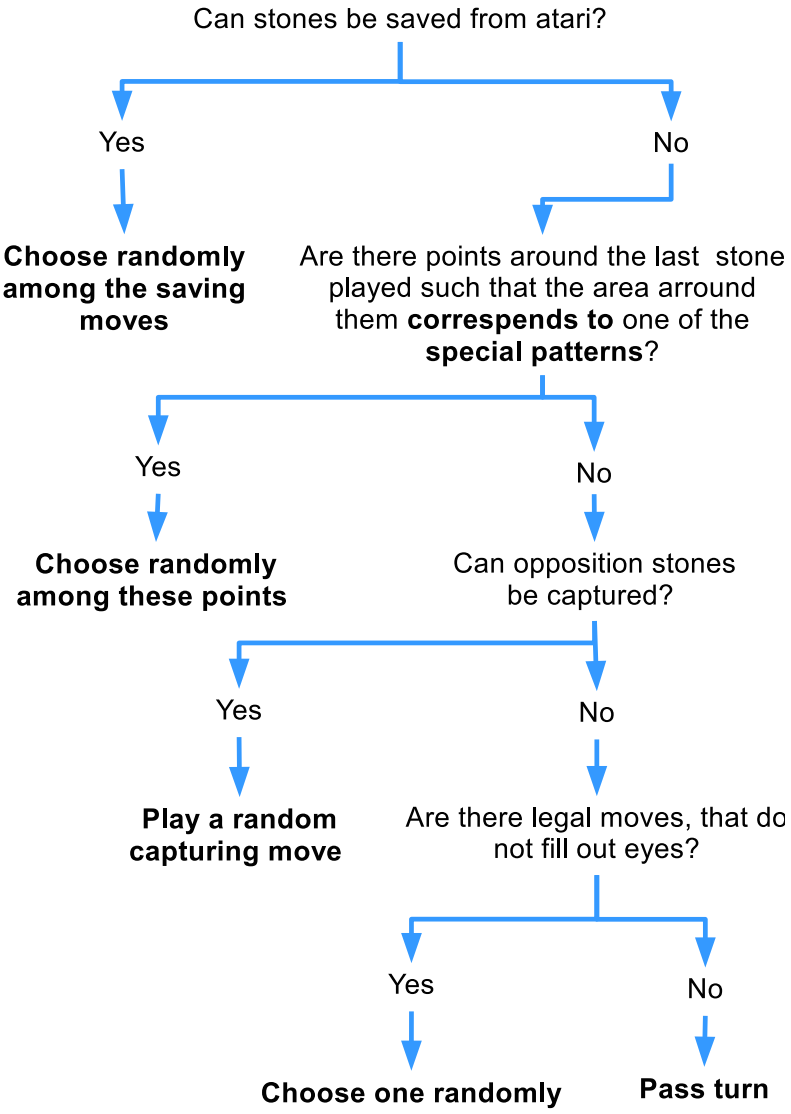



Figure 3.2: Decision diagram for *RandomBot_with_coded_patterns*

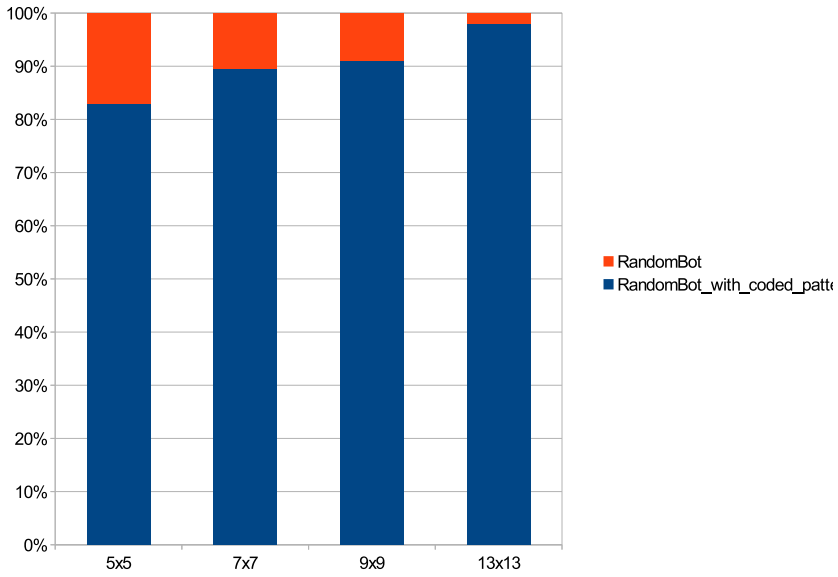
When selecting a move, *RandomBot_with_coded_patterns* will first only consider those, that could possibly save stones from an atari and select one of them randomly (again using a uniform distribution). Hereby a stone, or rather a full string is said to be in atari, if its

liberties have decreased to one and is thus in acute danger of being captured.
A saving move places a stone on this last liberty in order to increase the liberties. Of course this is only a viable move as long as this last liberty is not completely surrounded by stones of the opposite color.

If there are no moves that could possibly save a string from an atari, the bot will then look for special patterns in the area around the last stone played by the opponent. These patterns (pattern for Hane, Cut1, Cut2 and for moves on the boundary) are taken from [ModUCT] and were inspired by the program IndiGo [IndiGo].
If one of the (at most eigth) nodes surrounding the last played node is the center of one of these $3 \times 3$ patterns, it is added to a list of potential moves. The bot will then choose one of these potential moves at random.

If none of the patterns could be found around the last placed stone, the bot will determine whether there are opposition stones that can be captured, i.e. it determines the set of stones of the opposite color, that only have a single liberty left. If the set is nonempty, it will select one of these capturing moves at random. Otherwise it will select a move just as *RandomBot* would.

Figure 3.3: *RandomBot* vs *RandomBot_with_coded_patterns*



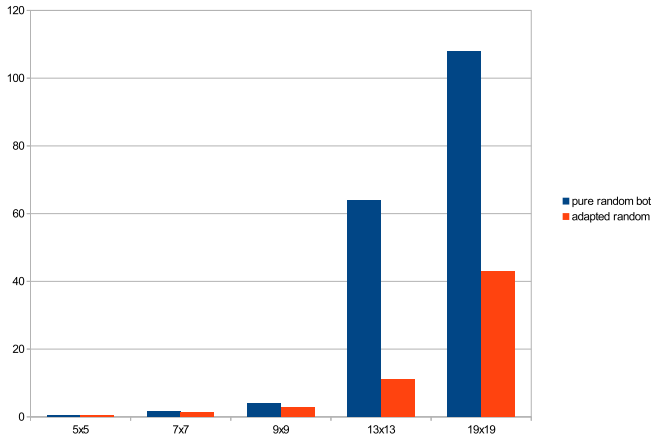

This chart shows the percentage of wins of each bot in 200 games in relation to the board size. Each bot played 100 games as white and 100 games as black.

Although this adapted random bot is much stronger than the original (see the statistics above), it still does not play very well. But the main reason behind the changes to our original random bot was to make random simulations more useful for move evaluation, as games

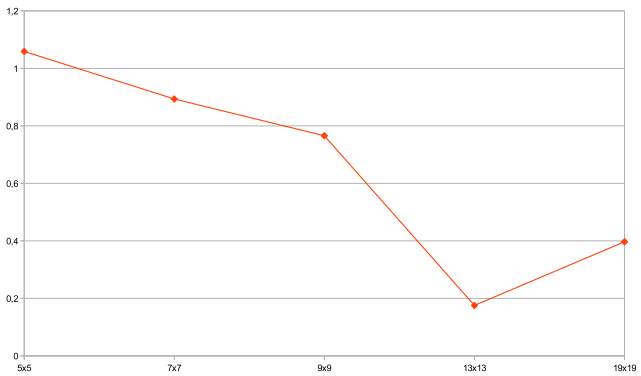
between two adapted random bots tend to generate more natural move sequences, since they are much more likely to select moves that are a direct reaction to the last stone played.

**Remark 7.** *The implementation of both bots can be found in the document 'fast_agent.jl'. The functions that determine the legality of moves and the functions that look for the four patterns mentioned above are implemented in 'fast_goboard.jl'. As seen in the chart below, RandomBot_with_coded_patterns is not only better than the original bot, but also produces much faster simulations on larger boards.*

Figure 3.4: speed comparison of *RandomBot* and *RandomBot_with_coded_patterns*




The first graph shows a comparison of the average time (in seconds) needed to complete 100 simulated games for both types of random bots, plotted against the gridsize.




The second graph shows the quotient between the red bar divided by the blue bar in the graph above. Thus it shows the the time it took our adapted random bot to simulate 100 games as fraction of the time it took the original bot to do the same.

## 3.3 Monte Carlo Simulations for move evaluation

We can use our random bots to create a much stronger Go playing agent, by using simulations of random games for move evaluation. To estimate the how likely a certain move will lead to a win, we could play this move (on a deepcopy of the Go board) and then simulate many games between random bots. The percentage of games that are won in these random simulations could then by used as an estimate for the real likelihood, that this move will lead to a win.
In otherwords we could try to sample subtree of the game tree, that represents all possible continuations of the game after the move is played.

Our agent will repeat this procedure for every possible moves and then play the move with the highest estimated winning probability. If the samples gained by these random simulations are good (i.e. large) enough, the agent will play almost optimally.

# 4 Monte Carlo Tree Search

A drawback of plain Monte Carlo Simulations is that a lot of computational effort might be wasted on the evaluation of moves, that are clearly not good, as a simulation starting from a move, that in most previous simulations lead to defeat, is equally as likely as one starting from a move that previously has shown to lead to a win.

Monte Carlo Tree Search (MCTS) overcomes this problem, by mainly investigating moves that have not been explored thoroughly and those that have so far often lead to wins. This is achieved by building a tree step by step. Each building step is called a rollout and adds one new node to the tree.

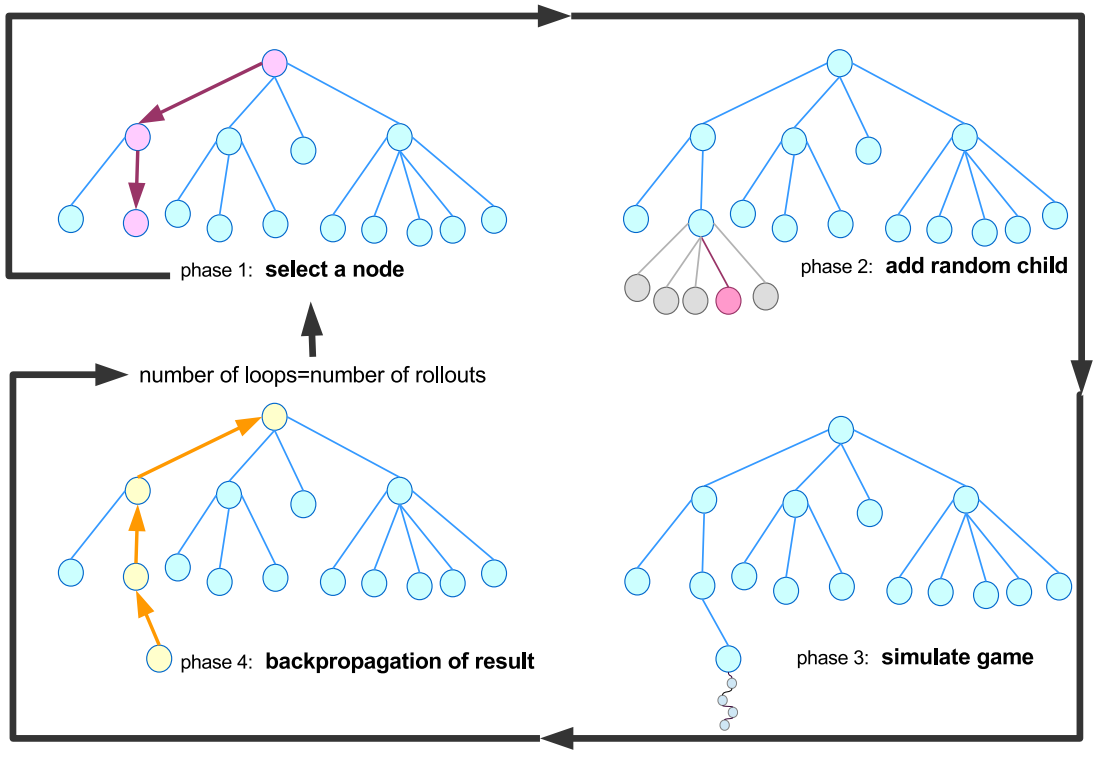



Figure 4.1: The four phases of a rollout

Every node of the tree will represent a future gamestate and by that also a move by one of the two players. The root of the tree represents the current gamestate.

In our implementation the nodes are represented by the struct *MCTSNode*, which consists of the represented gamestate and move, the nodes parent and its children in the tree. Furthermore each node stores the number of rollouts in which it was visited, the set of possible moves, that so far were not visited from this node (i.e. the potential children of the node) and the gamestatistics for all visits (i.e. the number of wins for each player).

Each rollout consists of four phases (see Figure 4 above).

First a node is selected (see Figure 4 below) at which the tree will be expanded. Starting at the root, the tree is recursively traversed until a node is reached, that represents the end of a game (terminal gamestate), or onto which new nodes can be added (i.e. the set of unvisited moves is nonempty).
Everytime a node is encountered that can not add any new nodes (and thus its children represent all possible moves from its stored gamestate), the child with the highest UCT-score (see equation (4.1)) is chosen as the next step of the tree traversal.

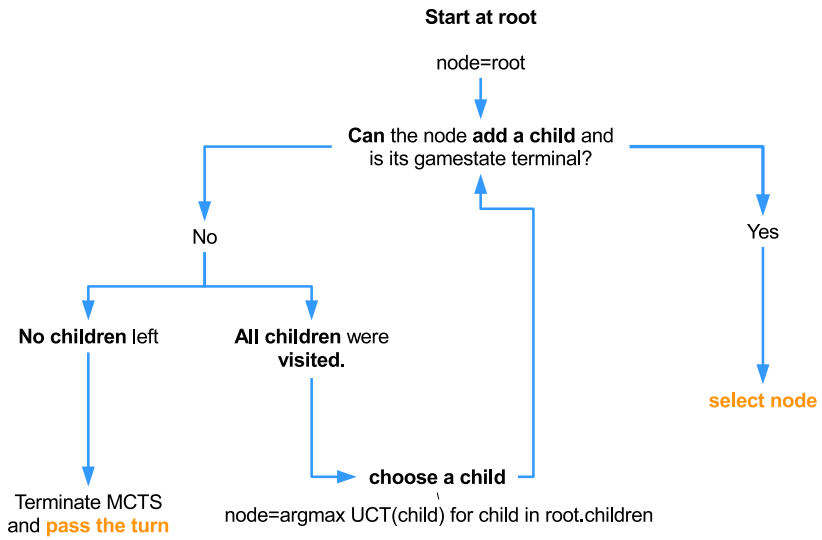



Figure 4.2: The process of selecting a child

The UCT-score (standing for Upper Confidence Bound 1 applied for trees) of a node is the sum of two terms and balances the exploration of rarely encountered nodes with the exploration of nodes, that were shown to be successful (=exploitation). The exploitation term is the percentage of wins, that were encountered during rollouts, that visited the node.

The explorational term is given by the square root of the logarithm of the number of visits of the parent node, divided by the times the node itself has been visited during the previous rollouts. It is then scaled by the so called temperature (the variable $\tau$ in formula

(4.1)). Although the temperature should be theoretically be set to $\sqrt{2}$, it is often chosen empirically. We will always use temperature $\sqrt{2}$ in our experiments.

$$\mathrm{UCT}(\mathrm{node}) = \underbrace{\mathrm{winning\_percentage}(\mathrm{node})}_{\text{exploitation}} + \tau \underbrace{\sqrt{\frac{\log(\mathrm{parent}(\mathrm{node}).\mathrm{num\_rollouts})}{\mathrm{node.num\_rollouts}}}}_{\text{exploration}} \tag{4.1}$$

In the second phase of every rollout, a possible move is picked at random from the set of unvisited moves of the previously selected node. This move is then used to create a new *MCTSNode* that is added to the tree as a child of the selected node. If the state is terminal and the node does not have any children left, this phase is skipped.

In the third phase a game is simulated from the gamestate of the node that was added in the second step. This is also called playout. Although in theory any bot (even another MCTS based one) could be used for this simulation, it makes sense to only consider simple reflex bots, i.e. bots that choose their moves only based on the current board position, without looking too far ahead, regarding the further development of the board. Otherwise MCTS might be slowed down drastically. In this chapter we will only look at MCTS algorithms, that use the two random bots discussed in the previous chapter.

In the fourth and final phase, the result of the simulation is reported and backpropagated through the tree. Thus every predecessor of the node that was added in phase 2 will adapt its gamestatistics according to the outcome of the simulation.

After all rollouts are completed, the bot will compare the children of the root of the generated tree. They form the first layer of the tree. The node with the highest percentage of wins is selected and the corresponding move will be played by the bot.

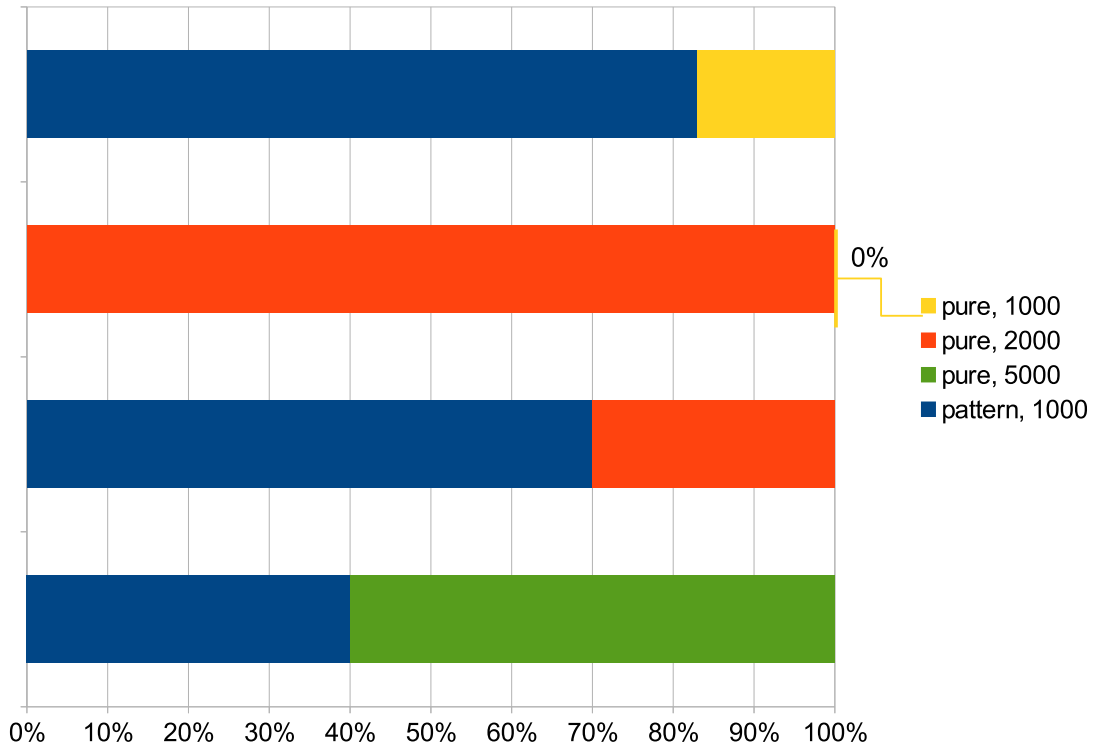
Note that the bot will not only select a move that is to be played next, it also estimates the associated probability of winning the game. Thus we can easily make the bot resign, if this probability is deemed too low.

There are two main components of MCTS, that determine the strength of the agent: the number of rollouts and the method chosen for the simulation of games
The higher the number of rollouts and the better the quality of the information that can be drawn from the from the playouts, the better the bot will play (see Figure 4).
As we have already seen an improvement regarding the simulation of games in the previuous chapter, in the form of our adaptions to our original random bot, we will turn our focus on increasing the speed of MCTS and by that the number of feasible rollouts.

**Remark 8.** *Note that the adaptions to the random bot not only improved the quality of the information that can be retrieved from a playout, but also did significantly speed up the*

*simulation of games on larger board sizes and by that also help to increase the number of rollouts.*

Figure 4.3: results of series of games between different MCTS bots




This graph depicts the results (by the percentage of games won) in a series (30 or 20 games) of games between different versions of MCTS. Hereby pure means that a normal random bot was used for playouts, whereas pattern means that *RandomBot_with_coded_patterns* was used. The number besides the type of playouts is the number of rollouts that were used.
As we can see, both increasing the number of rollouts and improving the quality of simulations improves MCTS drastically. Using *RandomBot_with_coded_patterns* seems to be worth almost as much, as tripling or quadrupling the number of rollouts (at least for 1000 rollouts).

## 4.1 Improving MCTS

A major drawback of our MCTS implementation is the large memory required to store the tree, as at the end of the last rollout, it will consist of $N$ nodes, where $N$ is the number of rollouts.

In each node the gamestate is the field that takes up most memory. Fortunately for us this field is acutally not necessary for MCTS, as the required information is already stored in the *move*-field.

## 4.2 Improving the performance

Instead of storing the gamestate in the nodes, we will seperately keep track of it. Therefore, at the beginning of each rollout, we create a deepcopy of the current gamestate (i.e. the current situation found on the board). During the tree traversal, we apply each move that is stored in one of the encounterd nodes to the board.
After a random leaf is added to the search tree, we apply the corresponding move to the gamestate and start a simulation from this gamestate.

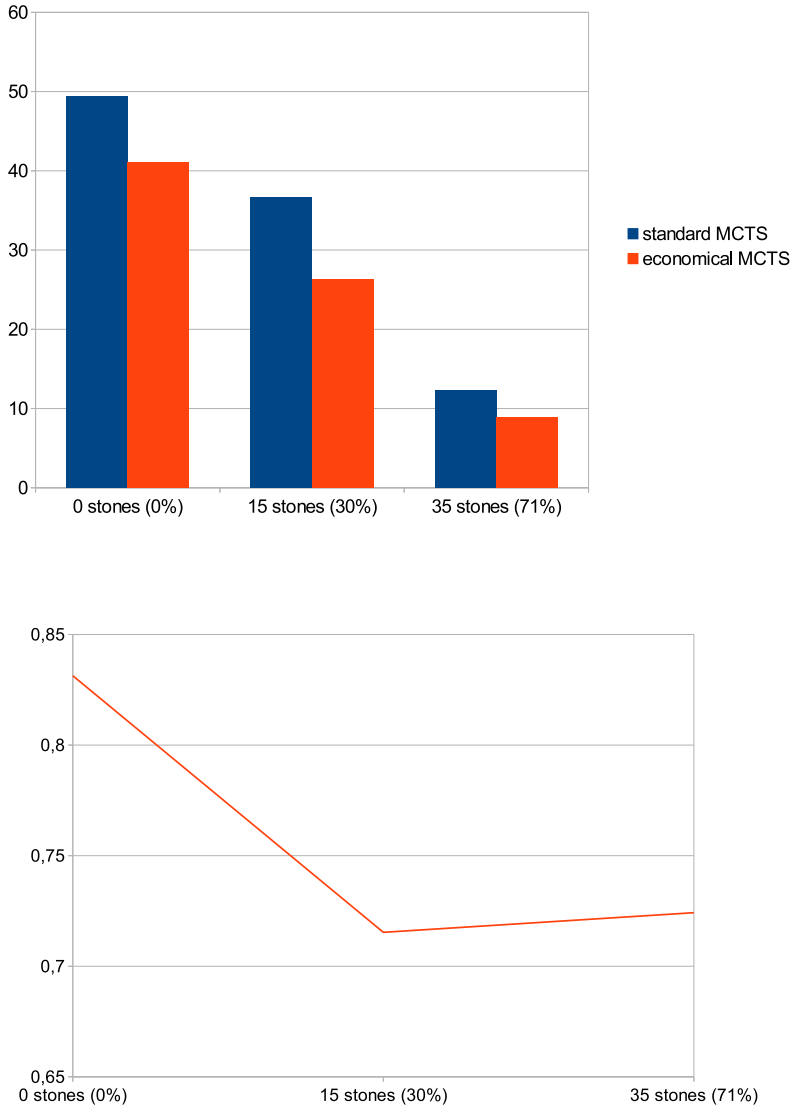
We implemented this sligthly adapted MCTS nodes in the structure *spareMCTSNode*. It contains all fields, that make up our original nodes structure *MCTSNode*, bar the field *gamestate*. The class of bots carrying out a MCTS, that utilise these new node structure is called *EconomicMCTSBot*.

Note that with the adaptation made, only two gamestates are stored at the same time during the MCTS: the current gamestate and its deepcopy, on which we apply the moves given by the search tree. This is a drastic improvement compared to the $N$ gamestates that occupy memory in our first implementation.

At first glance the changes made, seem to trade off decreased memory consumption for additional actions, slowing down the MCTS (namely applying moves to the Go board during tree traversal/node selection). But actually they also decrease the number of deepcopies made of different gamestates.
In our original implementation, each time a new node was added to the tree, a deepcopy of its parents gamestate had to be made. Furthermore the playout from a leaf node required an additional deepcopy of its gamestate, in order to not change the gamestate of the leaf. Thus $2N$ deepcopies had to be made in total.
In contrast the adapted MCTS only requires a deepcopy of the current gamestate at the start of each rollout, thus making $N$ deepcopies in total.

In our experiments (see Figure 4.2 below) this trade off (additional applications of moves to the Go board for a decreased number of deepcopies) seems to pay off, resulting in a faster move selection via MCTS. As we can see the difference in performance seems to get bigger for more filled boards. This is probably a consequence of the higher memory consumption of filled boards, which slows down the process of making their deepcopies.

Figure 4.4: A comparison of the drawtime of *StandardMCTSBot* and *EconomicMCTSBot*




In the first graph we plotted the absolute drawtime (in seconds) on a $7 \times 7$ board of the two implemented versions of MCTS (both 3000 rollouts) against the number of stone on the board (percentage of nonempty board positions in brackets).
The second graph puts the performances of both bots in relation and shows the time it took the improved/economical MCTS to select a move as a fraction of the time it took the original MCTS.

## 4.3 Virtual MCTS

Another way to decrease the memory consumption of MCTS is to make the search tree virtual. Hereby all nodes of the tree are represented by their gamestate (internally represented by the symbol of the next player and the hash value of the board position) and their stored move. The connections between nodes are not stored explicitly.

The statistics of each tree node (number of visits and the winning percentage) are stored in a dictionary (the keys are the next player the hash value of the board configuration and the move stored in the node). Another dictionary stores the number of times each state in the search tree was visited.

Without the explicitly implemented connections between the tree nodes, the tree traversal consists of choosing moves according to the statistics stored in the dictionaries and applying them to the current gamestate. The algorithm will store all visited states and moves in arrays and then adapt the MCTS statistics of the nodes by changing the dictionary entries accordingly.

The main difference to our original implementation of MCTS is, that the children and unvisited nodes of each node do not have to be stored explicitly, but are rather calculated during each visit.
This means that only a couple of unvisited moves are stored at the same time. Unfortunately it also results in a slightly slower execution of the tree search, compared to *EconomicMCTSBot*.
Another major advantage of this version of MCTS is that since (the only virtually existant) nodes are accessed through their gamestate, different branches of the (only virtually) built tree are automatically 'merged', as soon as two nodes represent the same gamestate. Thus information (about the statistics of nodes) is shared among all branches of the tree, resulting in a better exploration of the search tree.

**Remark 9.** *The version of MCTS discussed above is implemented in the composite type VirtualMCTSBot. The corresponding move selection function is based on the UCT-algorithm (Algorithm 1) found in [MCTSrave], with adaptions made to the tree traversal process.*

This version of MCTS will prove particularly usefull in the last chapter, as the devolpment of MC RAVE is based on it.

## 4.4 Parallelization

Another way to increase the speed of MCTS is to split up the workload among several processors. Luckily Julia offers many methods to parallelize programs easily. Specifically we made use of the function *pmap*, that allows us to excute a function on various different inputs in parallel without having to manage the control flow manually. It executes a function call for an input array in parallel on all available workers (not the master process).

**Remark 10.** *Note that in order to execute a function of a certain module in one thread, the module has to be loaded on that thread. Thus we have to load our Go modules (fast_goboard.jl and fast_agent.jl) on every process, we intend to use for the parallel MCTS. This has to be done before the start of the game, as Julia does not allow to load modules via using within a function call.*

We examine two possible parallelizations: root parallel and leaf parallel MCTS (see Figure 4.4 below).

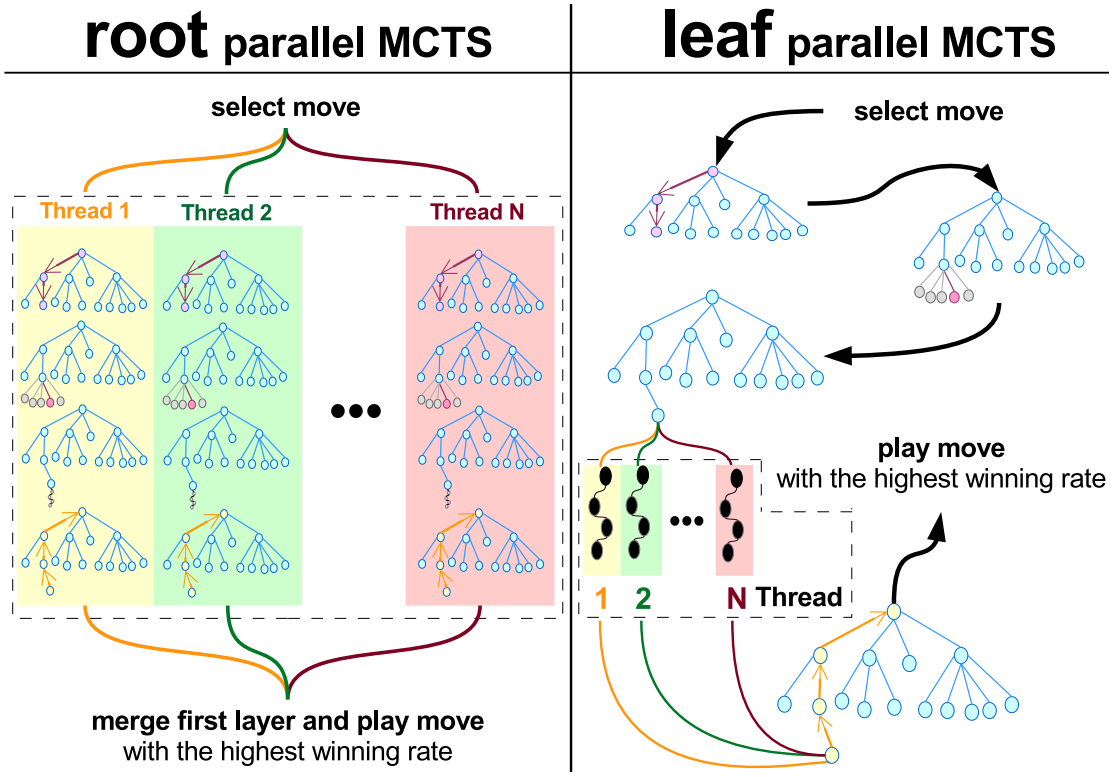



Figure 4.5: A comparison of root and leaf parallelisation

Leaf parallelization builds a single tree on the master process and only uses the workers (other processes) for the playouts. Thus three of the four phases of each rollout are carried

out exactly as in the standard version of MCTS, but instead of only simulating one game from the selected node, N simulations are simultaneously executed on the N threads.
Note that this form of MCTS parallelization does not directly increase the speed of MCTS, it just enables the bot to explore future gamestates more thoroughly without the need of additional rollouts.

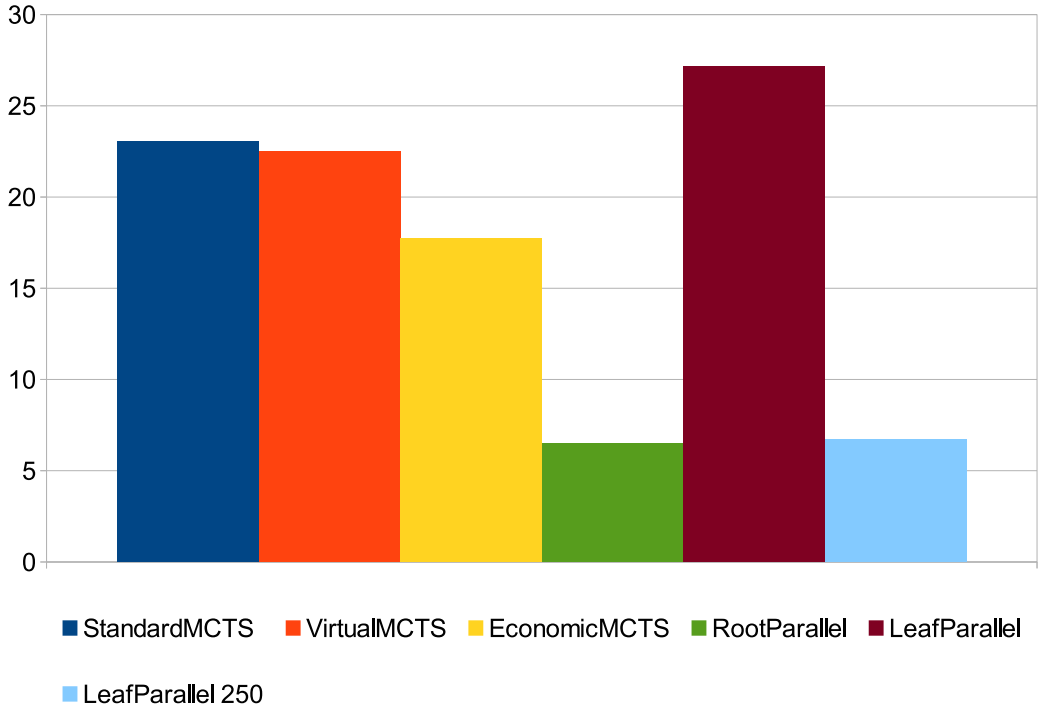
Root parallelization builds N (smaller) trees seperately on N threads. After all these trees are built, their first layers are 'merged' and the best move (highest average winning percentage) is played.
In our implementation, we defined the function *rate_moves*, that just works as the move selection function of our MCTS bots, but instead of returning a single move, it will return an array of tuples of moves and winning percentages that correspond to the first layer of the built MCTS tree.
Our root parallel MCTS bot then just calls this function *rate_moves* via *pmap*. Afterwards the different MCTS trees that were built are 'merged', by simply averaging the winning percentage of every move that was returned by *pmap*. Finally the move with the highest average winning percentage is selected to be played.
Note that in our implementation each worker executes a MCTS with $\lceil \frac{\text{number of rollouts}}{\text{N}} \rceil$ rollouts.

**Remark 11.** *In [ParMCTS] a third parallelization method, called tree parallelization, was proposed. Hereby the search tree is shared among all processes , and traversed in parallel. All workers nodes can access the tree and change the information in it. Thus this form of parallelization requires special attention for the communication between processes, as to avoid simultaneous writes.*
*We will not discuss this form of parallelization any further and refer the interested reader to the article mentioned above.*

Both parallelization methods discussed significantly increase the speed of exploration of MCTS (see Figure 4.4 below). Whereas root parallelization decreases the time consumed by MCTS, by directly distributing the workload of all rollouts among the workers, leaf parallelization just allows us to simulate more games at the same time.

**Remark 12.** *To compare the (computational) performance of leaf parallel MCTS to any other form of MCTS, one should set the number of rollouts of the leaf parallel algorithm to be equal to the number of rollouts of the other MCTS algorithm, divided by the number of workers used. This ensures that both algorithms will carry out exactly the same number of simulations.*
*Using this guideline, our leaf parallel algorithm seems to be approximately as fast as the root parallel one.*

Figure 4.6: A comparison of the performance of all MCTS algorithms




This chart show the time (in seconds) it took each algorithm to select a move on an empty $7 \times 7$ board, using 1000 rollouts (except for LeafParallel 250 which used 250 rollouts). Both parallelized algorithms used/carried out EconomicMCTS.

# 5 Rapid Action Value Estimation and heuristics

In this final chapter we want to discuss some adaptations to the standard MCTS algorithm, that lead to drastic improvements in Go playing engines and made the program Zen the first Go program to achieve a dan (professional) rank on full $19 \times 19$ boards.

The first alteration we want to discuss, is the use of heuristic winning estimates when adding new nodes to the tree.
Instead of only adding one single node to the tree in the second phase of each rollout, all possible children are initialized with an estimated winning percentage according to a heuristic function and a number of visits that expresses the confidence in this winning estimate, and added to the tree.
The node, from which the playout will be made, can then be chosen according to the highest estimated winning percentage, or randomly.
The main advantage of this method is, that it makes the constructed search tree larger, as the tree traversal does not have to be stopped, everytime a node is encountered from which not all children have been visited at least once. Furthermore it lets the bot minimize the time spent for the exploration of moves that are obviously bad.

Popular heuristics are:

- Even game heuristics, i.e. each node is initialized with a winning percentage of $\frac{1}{2}$. This very simple heuristic turns out to be quite effective (see the statistics in [MCTSrave]).
- The grandfather heuristic initializes every node with the values of the parent of its parent.
- Heuristics that make use of local shapes, either learned (e.g. through Reinforcement Learning) or coded by hand.

The confidence in the heuristic values is often a constant that is chosen empirically. In our experiments we used the values in [MCTSrave] as a guideline.

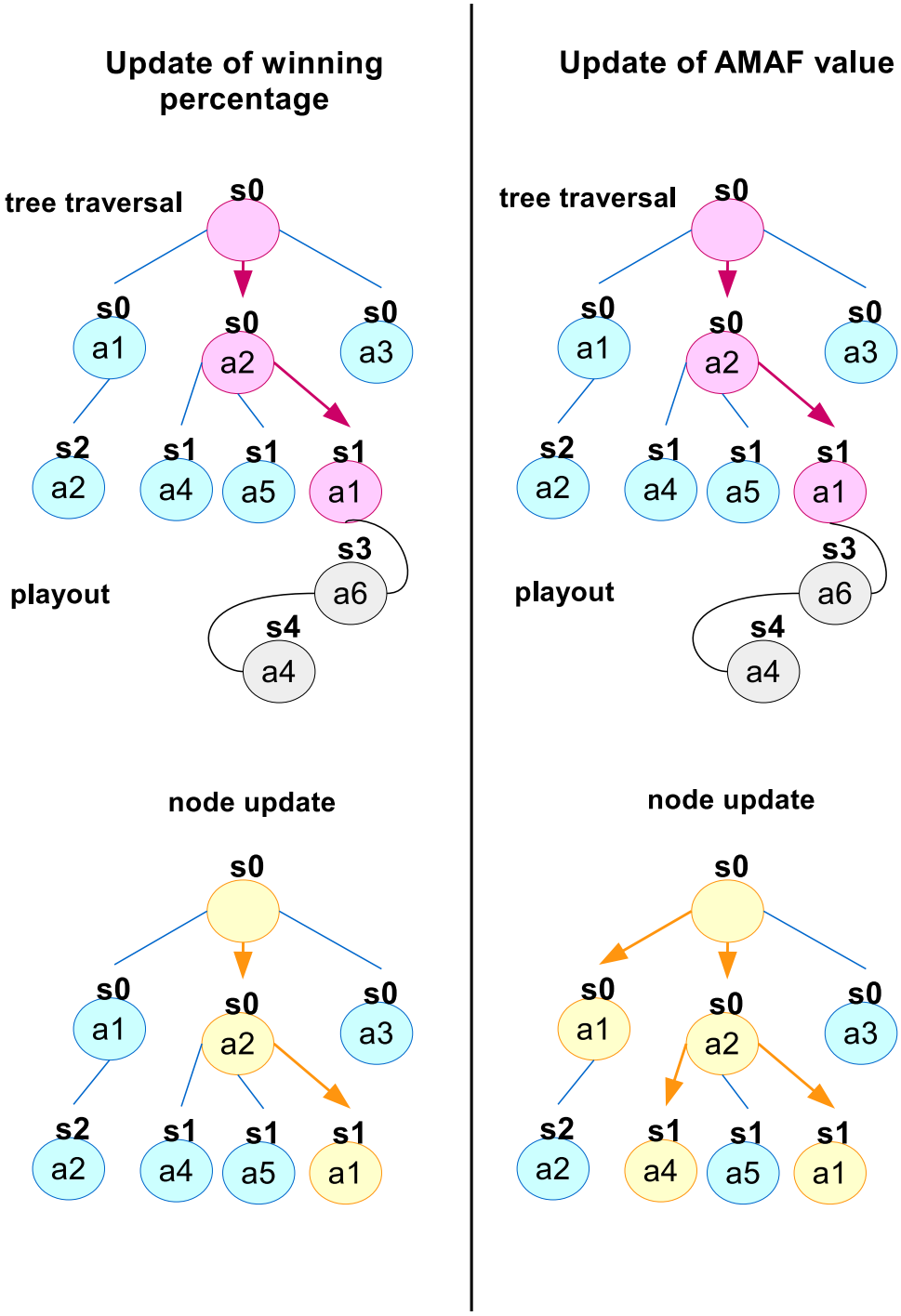



Figure 5.1: A comparison between the updating of the winning percentage and the AMAF value

Another adaption to MCTS is the so called Rapid Action Value Estimation (RAVE). Hereby the tree keeps track of a second statistic of each node, called the AMAF value. AMAF stands for all moves as first and makes the assumption, that the value of moves is not influenced by other moves and no matter at what point of the game a certain move is played, it will always have the same value to the player.
While this notion is obviously not true for Go (at least in a strict sense), we can still use it to improve our MCTS bot by enabling it to quickly estimate the value of moves.
Whereas the winning percentage of a node will only be updated if it was encountered during tree traversal prior to the playout, the AMAF value is be updated if the corresponding move was played at any later stage of the rollout (including the playout) (see Figure 5).

This means that the AMAF value will be updated far more often then the winning percentage and thus it gives the agent a quick but likely skewed estimate of the true value of a move.

The combination of RAVE and MCTS is often referred to as Monte Carlo RAVE (MC RAVE). A key factor to its success is the way how the winning percentage, stemming from classical MCTS, and AMAF value are combined for the node selection. A popular choice, that was introduced in [MCTSrave] is to select

$$\text{argmax}_{x\in\text{children(node)}}\ \beta(x)\text{AMAF}(x) + (1-\beta(x))\text{winnning\_pct}(x) + \tau\sqrt{\frac{\log(n(\text{node}))}{n(x)}},$$

with

$$\begin{aligned}
\beta(x) &:= \frac{\tilde{n}(x)}{n(x)+\tilde{n}(x)+4b^2n(x)\tilde{n}(x)},\\
n(x) &:= \text{number of times the winning percentage of } x \text{ was updated},\\
\tilde{n}(x) &:= \text{number of times the AMAF value of } x \text{ was updated}.
\end{aligned}$$

Hereby $b$ is a bias that should be chosen empirically. Note that the last term is the explorational term introduced in UCT. Often the temperature $\tau$ is set to zero, as the introduction of AMAF takes care of rarely visited nodes. Furthermore note that $0 < \beta \leq 1$. The formular of the combination term $\beta$ guarantees that, as a node is visited more often, the influence of the AMAF value on its value decreases.

The combination of MC RAVE and heuristics to initialize nodes is called Heuristic MC RAVE. We have implemented it in the structure *HeuristicMCRaveBot*. Hereby the process of move selection works just as with virtual MCTS. The only difference is, that new moves are initialized according to the heuristics and that each simulation also returns the array of moves that were played.

**Remark 13.** *Our implementation was inspired by the pseudo code found in [MCTSrave]. Unfortunately it was substantially slower and weaker than standard MCTS in our experiments, in which we used even game heuristics. The former was expected as the agent has to create and handle much larger move arrays (since all moves of a playout have to be*

*available for the AMAF value update).*
*The latter is strange considering that even with even game heuristics the agent, described in [MCTSrave] faired quite well. The author would guess that this could be a result of wrongly chosen parameters (confidence in heuristics and the bias term b in $\beta$).*
*Unfortunately our computational resources did not allow for extensive parameter tuning.*

# 6 Conclusion

In this project we have implemented a fast Go environment and several Go playing agents in Julia. Hereby we specifically focused on Monte Carlo Tree Search and related algorithms that were state of the art for Go playing agents until the early 2010s. The next logical step is to use these algorithms as a foundation and to implement Deep Learning algorithms on top of them (which is the fundamental concept behind the Go playing algorithms that are currently the best performing ones).

The main achievements of this project is the fast Go framework and the fast MCTS algorithms (Economical MCTS) that significantly outperform the algorithms of the source material used in terms of speed and the use of computational resources. The author believes that they can form a good foundation for similar projects.

Areas onto which the work presented here could be expanded (which will certainly be done by the author) are the improvement of the RAVE algorithm and the usage of more modern AI models like Deep Learning.

# Bibliography


[DeepGo] Max Pumperla, Kevin Ferguson
*Deep Learning and the Game of Go*,
Manning Early Access Program

[ModUCT] Sylvain Gelly, Yizao Wang, Remi Munos, Olivier Teytaud
*Modification of UCT with Patterns in Monte-Carlo Go*,
[Research Report] RR-6062, INRIA. 2006. <inria-00117266v3>

[IndiGo] Bruno Bouzy
*Associating domain-dependent knowledge and monte carlo approaches within a go program*,
Information Sciences, Heuristic Search and Computer Game Playing IV,
Edited by K. Chen, (4):247–257, 2005.

[ParMCTS] Guillaume M.J-B. Chaslot, Mark H.M. Winands,
H. Jaap van den Herik
*Parallel Monte Carlo Tree Search*,
Proc. Computers and Games 2008 (CG 2008), 60-71, 2008
Springer

[MCTSrave] Sylvain Gelly, David Silver
*Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go*,
Artificial Intelligence, Volume 175 Issue 11, July 2011, Pages 1856-1875