# Monte-Carlo tree search

Robert Klassert

June 4, 2019

**Abstract**

This report explains a class of algorithms known as Monte-Carlo tree search (MCTS). The concept of upper-confidence bound based action selection is explained in the context of UCT search and a simple Python implementation is presented. Finally, the algorithm's behaviour is studied in a number of examples.

## 1 Introduction

In this report the idea of Monte-Carlo based tree search is reviewed in the context of game AI. MCTS was officially introduced as a class of algorithms in Chaslot et al., 2008 [1]. The method especially took off due its effectiveness in computer Go and is still being used in DeepMind's *AlphaGoZero* the most advanced Go AI to date.
Back then people essentially still tried to conquer Go by game tree search with fancy heuristics. But due to the relatively large branching factor ($\approx 250$) of Go ($19 \times 19$) in comparison to chess ($\approx 35$), the maximum tree depth that can be analyzed by $\alpha - \beta$ search is very limited (One would need about $10^{d/2}$-times as many evaluations to reach the same tree depth in Go as in Chess). This huge complexity encountered in Go could - at the time - not be captured in an evaluation function good enough for producing respectable play.
MCTS skips these problems by

1. avoiding the need of an evaluation function all together (although it can help to have one)
2. being an anytime algorithm, meaning that it can be stopped at any point without wasting computing time

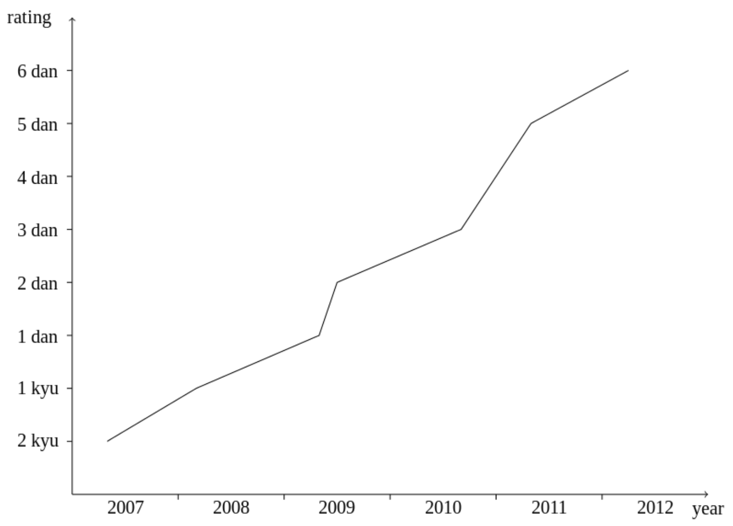

Figure 1: Increase in Go playing strength since the onset of MCTS [2]

3. growing the search tree dynamically and asymmetrically, thereby focusing on the most promising moves

One might now ask: How does MCTS know about the quality of moves without an evaluation function? - This is where the "Monte-Carlo" part comes in: By random rollouts of the game starting with the variation to be evaluated. Each time the game has been randomly played out in such a way, the result (current player won/lost) is used to update the weights of each node that was involved in the variation. Iteration of this process yields an estimation of the win rate for each node which is always a good heuristic for good moves. In Go, for example, a move with high win rate can be interpreted as a stone with large influence, because many random variations were not able to "disturb" the position afterwards.
MCTS is not limited to games. It can also be employed for decision making in the general framework of Markov decision processes. In this context the win rate in games corresponds to the so-called expected reward.

## 2 Vanilla MCTS

In this section the most basic version of MCTS is presented. Starting from the current game position it consists of 4 steps (see fig. 2) that are repeated until a computational or time constraint is met.

1. Selection: The selection phase traverses the tree level by level, each time selecting a node based on stored statistics like "number of visits" or "total reward". The rule by which the algorithm selects is called the *tree policy*. Selection stops when a node is reached that is not fully explored yet, i.e. not all possible moves have been expanded to new nodes yet.

2. Expansion: The expansion step consists of adding one or multiple new child nodes to the final selected node.

3. Rollout: A rollout is initiated from each of these added nodes. Depending on the mean depth of the tree, rollout is done until the end of the game or as deep as possible before performing an evaluation. When dealing with terminal states in games the evaluation is done based on the win condition and usually yields 0 (loss), 0.5 (draw) or 1 (win).

   In the original algorithm the rollout is performed at random, but in general the so-called *default policy* can be enhanced by heuristics. For example, in the case of *AlphaGoZero* the rollout step was completely replaced by a neural network evaluation function.

4. Backup: The outcome of the rollout step is backed up to the nodes involved in the selection phase by updating their respective statistics.

After the algorithm has stopped, the final move selection is done based on the statistics gathered in the resulting tree.
The two key items of MCTS are the tree and rollout policies. Together they determine how the tree is built and thus what moves the algorithm focuses its efforts on.
As an example: a bad tree policy would be one that solely selects the node with the highest win rate. This would lead to a lock in of a particular move that by chance happened to end in a few wins while others did not. This is referred to as a greedy policy in the sense that it only cares about the reward estimate but does not account for the uncertainty involved in the estimates.
The desired behaviour is one, where it balances between gathering knowledge about the different possible actions (exploration) and, if sufficient, acts based on it (exploitation).

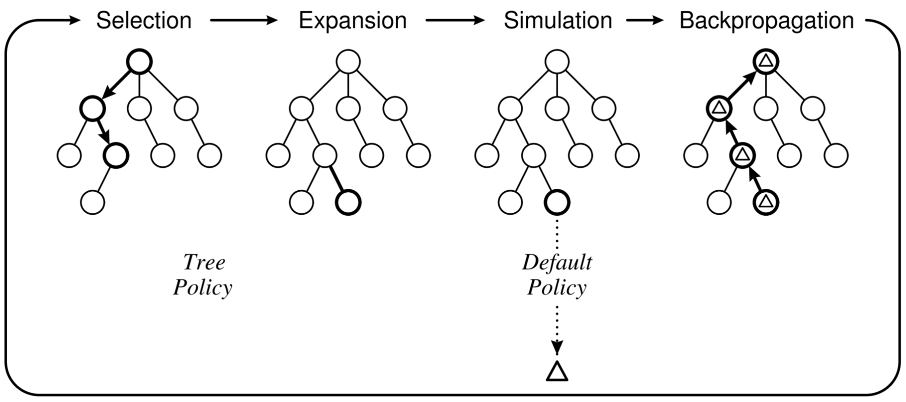

Figure 2: 4 steps of the vanilla MCTS algorithm

## 3 Exploration & Exploitation

To find a tree policy that can solve the exploration/exploitation dilemma it is instructive to look at the simplest example where it occurs: the multi-armed bandit problem.
In this problem a slot-machine similar to those found in real casinos is imagined. It offers $k$ actions at every step, that yield payoffs $X_a$ , $a \in 1, ..., k$. The payoffs are random variables and their underlying distributions unknown. The task is to maximize the payoff within some number of steps using an appropriate policy. The easiest way to maximize the reward while incorporating the uncertainty of the payoff estimates is to use an upper confidence bound for the expected payoff (UCB action selection).
Let $\mu_a$ and $\sigma_a^2$ be the mean and variance of the reward distributions. Then a reasonable policy would be

$$\hat{a} = \text{argmax}_a(\mu_i + \sigma_i) \tag{1}$$

On the one hand actions with large variance would be selected to improve the mean estimate, hence exploring them. On the other hand, actions with small uncertainty and large means would be selected to obtain the payoff, hence exploiting them.
It seems the dilemma has been resolved, however, the standard UCB only offers an *arbitrary* balance between exploring and exploiting. Given enough time the

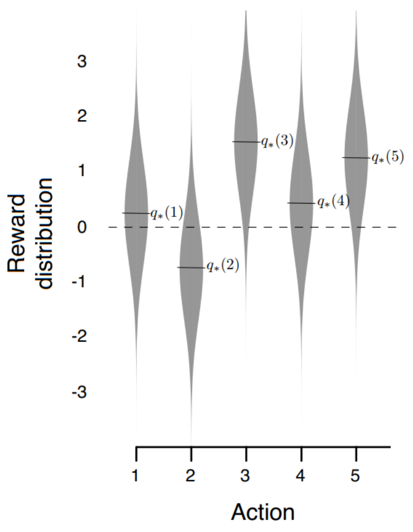

Figure 3: Reward distributions of a 5-armed bandit, taken from Sutton [3]

policy should always end up selecting the optimal action in the majority of times. In other words the difference between the optimal payoff and the expected payoff due to our UCB should be minimal. This difference is referred to as the *regret* of the policy.
Auer et al. [4] found different UCB's which are within a constant factor of the optimal bound thus minimizing the regret. The most popular bound is UCB1:

$$\text{UCB1} = \mu_a + \sqrt{\frac{2 \ln N}{N_a}} \tag{2}$$

where $N$ is the total number of steps and $N_a$ is the number of times action arm $a$ has been selected. The first term is identical to eq. (1) and the second term is also proportional to $\sigma_a \propto 1/N_a$. The important difference to the non-optimal policy is the term $\propto \sqrt{\ln N}$.

## 4 Implementation of the UCT algorithm

With the UCB1 solution of the multi-armed bandit problem the tree policy in MCTS can also be made asymptotically optimal. This means, that with growing number of iterations the game tree built by the algorithm will approach the "true" minimax tree.

Each arm is identified with a possible move and the expected payoffs correspond to the win rates $Q_a/N_a$. Since UCB is only within a constant factor $C$ of the optimal bound, it is added as a domain-dependent parameter that can be optimized:

$$\hat{a} = \text{argmax}_a \left( \frac{Q_a}{N_a} + C \cdot \sqrt{\frac{2 \ln N}{N_a}} \right) \tag{3}$$

The UCT algorithm (upper confidence bounds for trees) is thus nothing but MCTS with a tree policy according to UCB1.
The Python implementation below allows the execution of the UCT algorithm for arbitrary games and decision environments by passing a set of rules to the MCTS object. Although, performance-wise it is far from optimized, the algorithm can be easily tested on simple games and the impact of different enhancements can be studied.

```python
import numpy as np
from random import randint

# Game tree node
class Node:
    def __init__(self, state, parent=None, statistics={}):
        self.state = state
        self.parent = parent
        self.children = {}
        self.statistics = statistics

    def expand(self, action, next_state):
        child = Node(next_state, parent=self)
        self.children[action] = child
        return child

class MCTS_with_UCT:
    def __init__(self, state, gamerules, C=1):
        self.game = gamerules
        self.C = C
        self.root = Node(state, statistics={"visits": 0,
            "reward": np.zeros(self.game.num_players())})

    def is_fully_expanded(self, node):
        return len(self.game.get_actions(node.state))
            == len(list(node.children))


    def best_action(self, node):
        children = list(node.children.values())
        visits = np.array(([child.statistics["visits"]
            for child in children]))
        rewards = np.array(([child.statistics["reward"]
            for child in children]))
        total_rollouts = node.statistics["visits"]
        pid = self.game.get_current_player_id(node.state)

        # calculate UCB1 value for all child nodes
        ucb = (rewards[:,pid]/visits +
            self.C*np.sqrt(2*np.log(total_rollouts)/visits))

        best_ucb_ind = np.random.choice(
            np.flatnonzero(ucb == ucb.max()))
        return list(node.children.keys())[best_ucb_ind]

    def tree_policy(self, node):
        while not self.game.is_terminal(node.state):
            if not self.is_fully_expanded(node):
                act_set = np.setdiff1d(
                    self.game.get_actions(node.state),
                    list(node.children.keys()))
                action = act_set[randint(0, len(act_set) - 1)]
                newstate = self.game.perform_action(
                    action, node.state)
                childnode = node.expand(action, newstate) # <- expansion
                childnode.statistics = {"visits": 0,
                    "reward": np.zeros(self.game.num_players())}
                return childnode
            else:
                node = node.children[self.best_action(node)] # <- selection
        return node


    def rollout(self, node):
        roll_state = node.state.copy()
        while not self.game.is_terminal(roll_state):
            act_set = self.game.get_actions(roll_state)
            action = act_set[randint(0, len(act_set) - 1)]
            roll_state = self.game.perform_action(action, roll_state)
        return self.game.reward(roll_state)

    def backup(self, node, reward):
        while not node is None:
            node.statistics["visits"] += 1
            node.statistics["reward"] += reward
            node = node.parent

    def run_iter(self, iterations):
        # Vanilla MCTS loop
        for i in range(iterations):
            selected_node = self.tree_policy(self.root) # selection + expansion
            reward = self.rollout(selected_node) # rollout
            self.backup(selected_node, reward) # backup
        return self.best_action(self.root)
```

## 5 Examples

I applied the above implementation of MCTS to explore some basic problems: A grid world, where a single player has to find the exit square, classical Tic-Tac-Toe and Go on a smaller board.

### 5.1 Grid world

The player is trapped in a two-dimensional square grid world with periodic boundaries. A square can either be empty, a hole or the magical key to a higher dimension. For testing 3 holes, 1 magic key and the player were randomly placed into a $4 \times 4$ grid at the start of the game.
At each time step the player can move in the four cardinal directions. The goal of the game is to leave as fast as possible by moving to the key. When moving to a hole the game ends with a loss for the player.
Fig. 4 shows two histograms of a series of 100 games each. While for the left histogram the number of MCTS iterations was 10, the second series shown on the right had 150 iterations per move. Consequently, of the first series the agent won 79 games, while it won all games in the second series.
From the histograms it is also clear that the MCTS player got a lot better in finding the shortest routes to the key when comparing 10 to 150 iterations. Due to the geometry of the grid world the distribution of path lengths is peaked at 3 moves.

### 5.2 Tic-Tac-Toe

Here, I wanted to test if my implementation worked correctly by playing Tic-Tac-Toe against itself on the usual $3 \times 3$ board. Since it is a theoretical draw, games should also end in a draw with increasing frequency when increasing the playing strength (iterations) of the MCTS agents.

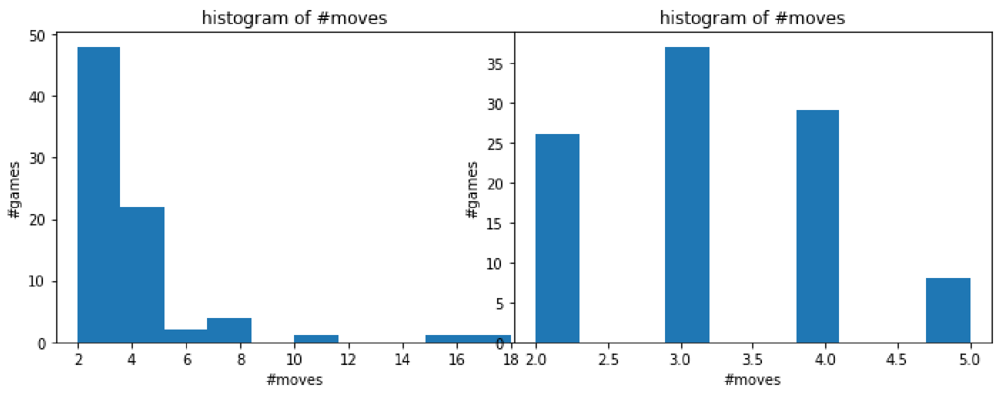

Figure 4: Histogram of move number of won grid world games

As shown in table 1 this is indeed what is observed. It can also be seen that player 1 always wields an advantage over player 2, as expected.
In generalized Tic-Tac-Toe, MCTS could actually be used as a tool for analyzing if a certain geometry and win condition would be a theoretical draw or not.

| iterations | 100 | 1000 | 10000 |
|---|---|---|---|
| 1st player win | 4 | 2 | 0 |
| 2nd player win | 2 | 0 | 0 |
| draw | 4 | 8 | 10 |

Table 1: 10 match series between two MCTS agents

## 5.3 Go

Since MCTS is supposed to be good at Go I had to also test my MCTS implementation on it. Also I was curious if the algorithm would actually be able to beat me, as I have no experience in Go besides knowing the rules.
For dealing with the Go positions and rules I used a publicly available Go implementation in Python: `https://github.com/brilee/go_implementation` [5]. However, I quickly found out, that my MCTS code was too slow for more than 1000 iterations on the full $19 \times 19$ board. So I scaled down to a mini-Go with $5 \times 5$.
Initial experiments between two MCTS agents showed that black is winning more often than white as is to be expected. In tournament Go rules this disad-

vantage is actually compensated by granting white a fixed value of points for at the end of the game (*komi*).
Testing agents with increasing iteration number against myself showed that a basic knowledge of the rules was sufficient to easily defeat the program until about 5000 iterations. Of course, I kept on increasing the number of iterations and... I actually lost a game at 9000 iterations! (Though I was quite tired at that point :))

## 6 Conclusion & Outlook

In this report the basic idea of Monte-Carlo tree search was presented. The UCB solution to the exploration/exploitation dilemma was discussed and applied to MCTS by means of the UCT algorithm.
A simple Python implementation of MCTS served as the test bed for a number of simple tasks. Although the functionality of the algorithm could be verified, it was found to be lacking performance-wise in more complex scenarios like Go.
Future work might include improving the performance by switching completely to *numpy* based computation and parallel execution on GPUs.

## References

[1] Guillaume Chaslot, Sander Bakkes, István Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. 2008.

[2] Mciura [CC BY-SA 3.0 (https://creativecommons.org/licenses/by sa/3.0)]. Historical kgs ratings of the best computer go programs, 2012.

[3] Richard S. Sutton, Andrew Barto, and Andrew G. Barto. *Reinforcement Learning*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass. [u.a.], 3. printing edition, 2000.

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

[5] Brian Lee. Implementing the game of go, 2016.

[6] C. B Browne, E Powley, D Whitehouse, S. M Lucas, P. I Cowling, P Rohlfshagen, S Tavener, D Perez, S Samothrakis, and S Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[7] Guillaume Chaslot, Jahn takeshi Saito, Jos W. H. M. Uiterwijk, Bruno Bouzy, and H. Jaap Herik. Monte-Carlo Strategies for Computer Go.

[8] Istvan Szita, Guillaume Chaslot, and Pieter Spronck. Monte-Carlo Tree Search in Settlers of Catan. *Ethical Theory and Moral Practice*, 6048:21–32, 05 2009.

[9] Levente Kocsis and Csaba Szepesvri. Bandit based Monte-Carlo Planning. pages 282–293, 2006.