A Survey on Monte Carlo Tree Search Methods

Simon Schwarz*

Chair For Pervasive Computing Systems – TECO,
Karlsruhe Institute of Technology

Abstract. Monte Carlo Tree Search (MCTS) is a sampling-based, iterative search algorithm that is well known to address the exploration-exploitation tradeoff and was made famous in its application to the game Go. Its basic formulation works well in small or simple domains. However, applying it to specific use cases which have unique requirements (like real-time or cooperative scenarios) necessitates additional engineering. Consequently there has been a lot of research into extending MCTS to these new domains. This mainly concerns performance improvements, modifying the algorithm's components to leverage domain knowledge or through the use of heuristics. This survey presents notable examples of this research.

Keywords: Monte Carlo Tree Search · MCTS · UCB

1 Introduction

Monte Carlo Tree search (MCTS) is a sampling-based search algorithm for finding optimal decisions in a given domain by iteratively building a tree. It was first proposed by Chang et al. [8] and consecutively refined by Coloum [10] and Chaslot et al. [9]. MCTS has since distinguished itself by addressing the tradeoff known as the exploration-exploitation dilemma, offering a good balance between the exploration of new paths and the exploitation of known paths deemed to be promising. MCTS has become widely used in the domain of machine learning and especially in General Game Playing (GGP) because it performs well even in large domains and when no domain knowledge is available [32]. The most famous application of MCTS in this context is its usage as part of *AlphaGo*: A neural network trained using MCTS that beat some of the worlds greatest human players in the game of Go – something deemed impossible because of the sheer number of possible game states and the resulting size of the search space making the problem seem intractable with current methods [31].

The basic structure of MCTS is very simple. It is run in iterations until some condition is no longer satisfied, usually until a computational budget is exhausted. Each iteration consists of only four steps. Starting from the root node, i.e. the node representing the current state of the domain, the *Tree Policy* is used to select a single new node and expand the existing tree. Now the *default policy* is

* Advisor: Yiran Huang

used to execute a simulation until a terminal state is reached. This state is now evaluated with respect to its reward. This reward is then backpropagated and used to update any statistics that are kept about the nodes involved. Details concerning the steps above are left up to the concrete implementation, the most popular being *UCT* which models the node selection in step one as a *Multi-Armed Bandit Problem* (MAB). This in turn allows for the utilization (and indeed reuse) of a variety of methods and analyses defined for MABs.

The algorithm can additionally be augmented by heuristics. A notable example of such an heuristic is *All Moves as First* (AMAF) which was first proposed in the context of (again) Go and later combined with UCT by Gelly et al. [12]. AMAF modifies which nodes are updated in each iteration by updating not only the selected node but also all nodes that are not chosen but used during the simulation. An idea that has been further extended by Gelly and Silver [13] and Cazenave [7] among others.

This paper is a survey of recent developments of modifications of the basic search algorithm and its use in domains inside as well as outside of computer science. It is structured as follows: Section 2 gives an overview of the concepts used and extended by MCTS as well as a detailed explanation of the algorithm itself and its most important use case in trees. Section 3 presents variations of MCTS in five select categories. Then Section 4 discusses the application of MCTS in various use cases. Finally, Section 5 gives a summary of the paper and examines possible future research opportunities.

Due to the limited extend of this paper the overview is by no means exhaustive, there are scenarios and whole fields of modifications that are not mentioned at all. However, in the fields that are discussed an attempt was made to present work that is both prominent as well as representative of trends in research from 2013 to 2020. This paper can further be seen as an update of a 2012 survey by Browne et al. [6].

2 Background

This section gives a brief overview of the theoretical considerations and concepts related to Monte-Carlo Tree Search. These ideas will be extended in Section 3 and Section 4.

2.1 Game Theory

Following the conventions used by Browne et al.[6] a game is formally modeled using the following components:

- $n \in \mathbb{N}$ players $k_1, \ldots, k_n$
- A set of states S with an initial state s_0 and terminal states $S_T \subseteq S$
- A set of possible actions A
- A state transition function $f : S \times A \to S$
- A reward function $r : S \to \mathbb{R}^k$

A game starts in state s_0 and progresses according to the state transition function f which at point t gives us the next state s_{t+1}. Transitions happen through the action a taken by a player k_i. Usually it is only one players' turn at any given point t. The reward function r determines points gained or lost by players through these actions. The game ends when a terminal state $s_j \in S_T$ is reached. The probability of a given player choosing any action a is determined by its *policy* and may be constrained by the current state s via the rules of the game. Games can be classified using the following criteria:

a) *zero sum*: Do the rewards given to all players add up to zero?
b) *information*: Is the complete state of the game known to the players?
c) *determinism*: Does change play a part?
d) *sequential*: Can multiple players choose an action simultaneously?
e) *discreteness*: Are actions atomic or do they take time?

An important category of games are so-called *combinatorial games*. They are zero-sum, perfect information, deterministic, sequential and discrete. Examples are *Go*, *Chess* and *Tic-Tac-Toe*.

2.2 Monte Carlo Methods and Multi Armed Bandits

The basic idea of Monte-Carlo based methods is that of *sampling*. While operating on a very large domain it is not feasible to simply do calculations which consider all possible elements of said domain. We therefore have to select only a (usually much smaller) subset of these elements to represent the domain somewhat accurately and to gain as much information as possible. In the context of games the aim of such methods is to find good moves while only analyzing some parts (states, actions and rewards) of the game. This can be done by starting at an initial state (either the actual first state or just the current game state) and repeatedly estimating the value of possible actions. The idea is that these estimations will converge and allow us to make an informed choice, that is, to choose a good action at a rate much higher than chance. One widely used concept when applying Monte Carlo methods to games is the so-called Q-value. At an action a at a state s it is defined as the expected reward of this action:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a) \cdot Q^i(s)$$

$Q^i(s)$ is the result of the i-th simulation started from s, $N(s)$ denotes how often s has been visited, $N(s,a)$ denotes how often action a has been selected when s has been visited and $\mathbb{I}_i(s,a)$ is one if a has been selected when in state s during the i-th simulation from s and zero otherwise. Simulation in this context refers to a play-out of the game from the selected state, a more precise definition is given in later sections.

In a *Multi-Armed-Bandit* problem (see e.g. [22] for an introduction) a player has to choose one of K actions ("arms") out of the set $\mathcal{K} = \{1, \ldots, K\}$ at each

one of T rounds. The number of rounds T is called the *horizon* of the game and the decision points $t_1, \ldots, t_T$ are called the decision epochs. Each arm has a value associated with it. These rewards X_t^k of arm k at decision epoch t can be modeled as a random variable with values $X_k^t \in [0, 1]$ and expected value $\mu_t^k = \mathbb{E}[X_t^k]$. The best possible reward is therefore given via

$$\mu_t^\star = \max_{k \in \mathcal{K}} \{\mu_t^k\}$$

In the *non-stationary* problem formulation the value of each arm may change over the course of the game. The extent of this change is called *temporal variation*. The performance of a given policy (arm selection) can be measured relative to an oracle which has perfect information and picks the best possible choice. The difference in rewards between the policy and the oracle is called *regret*. After n epochs it is given as:

$$R_n = \mu^\star n - \sum_{j=1}^{K} \mu^j \cdot \mathbb{E}[T_j(n)]$$

$\mu^\star$ is the maximum reward mean, μ^j is the reward mean of arm j and $\mathbb{E}[T_j(n)]$ is the expected number of times arm j has been played in the first n epochs. The regret can be seen as the difference between the performance of an oracle with complete information and an algorithm with incomplete information. Any policy aims to minimize this value. As shown by Lai and Robbins [21] there exist no policy with a growth of the regret slower than $O(\ln n)$ for most reward distributions (i.e., distributions of the rewards X_t^k). For bandit problems it is desirable to know the *Upper Confidence Bound* (UCB) that any given arm will be optimal in the game-theoretic sense described in Section 2.1. In the formulation of Sironi and Winands [32] (called UCB1) it is given by:

$$\text{UCB1} = \underbrace{Q(s,a)}_{(*)} + \underbrace{\sqrt{\frac{2 \ln N(s)}{N(s,a)}}}_{(**)}$$

Since this term is maximized by any reasonable policy (i.e., one that aims to win) its parts can be analyzed quite easily: The first term $(*)$ facilitates the exploration of higher reward choices while the second term $(**)$ does the same for less-visited, currently deemed less promising choices. This term is prominently used as a sampling strategy in the UCT-MCTS variant described below.

2.3 Monte Carlo Tree Search

All paths through a game, in the above definition, can be exhaustively described in the so-called *game tree*. This (conceptual, theoretical) data structure is a tree that contains all possible states a game could be in. Its nodes represent the states and its edges (s_i, s_j) represent the action leading from state s_i to state s_j. In practice however, this tree is too large to actually be represented. For

example, in chess there are about 35^{80} possible playthroughs and consequently as many paths through the game tree. Monte Carlo Tree Search (Algorithm 1) is an algorithm for traversing game trees utilizing the idea of sampling. MCTS iteratively builds a partial game tree (the *search tree*) while a certain condition holds, usually referred to as the computational budget. In each iteration this current "view" of the game tree is refined by estimating and refining the value of the states contained. Consecutive iterations of MCTS usually represents a turn of a game-playing agent. That means the search is used to find the most valuable state. When the budget is exhausted and no more iterations are possible the action leading to this state (or, generally to a subtree containing it) is returned and executed by the agent. This will be the setting for the remainder of this paper but since MCTS is a general-purpose search algorithm for any problem that can be modeled as a game it may not be the specific purpose of every use case. This will become evident in Section 4. The algorithm consists of four basic steps displayed in Figure 1:

1) *Selection*: Beginning from the root node t_0 the tree is traversed using a child selection policy until an expandable node t_n (representing state s) is reached: that is, a node representing a non-terminal state that has unvisited children.
2) *Expansion*: One or more child nodes are added to the tree if the actions leading to them are possible. Here, t_l representing state s' is added by selecting action a.
3) *Simulation*: From the state of the new node a simulation of the game is run according to the *Default Policy*. Its results are taken to be the value of the node.
4) *Backpropagation*: The new values are backed up through the tree which may result in updates to the statistics of the nodes. This means updating the Q-values.

How the game states are represented, if/how games need to be discretized and how the outcome of terminal states is evaluated is not part of the algorithm.

Algorithm 1 Basic Monte Carlo Tree Search function.

```
function MCTS(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TreePolicy(v_0)                          ▷ v_l is the last node reached
        Δ ← DefaultPolicy(s(v_l))      ▷ simulate from state node v_l with state s_l
        Backup(v_l, Δ)
    return a(bestChild(v_0)))                          ▷ return most valuable action a
```

Browne et al. [6] identify three characteristics of Monte Carlo Tree Search: a) *Aheuristic*: There is no concrete domain knowledge required. MCTS can be applied to any domain that can be modeled as a tree. However, any available

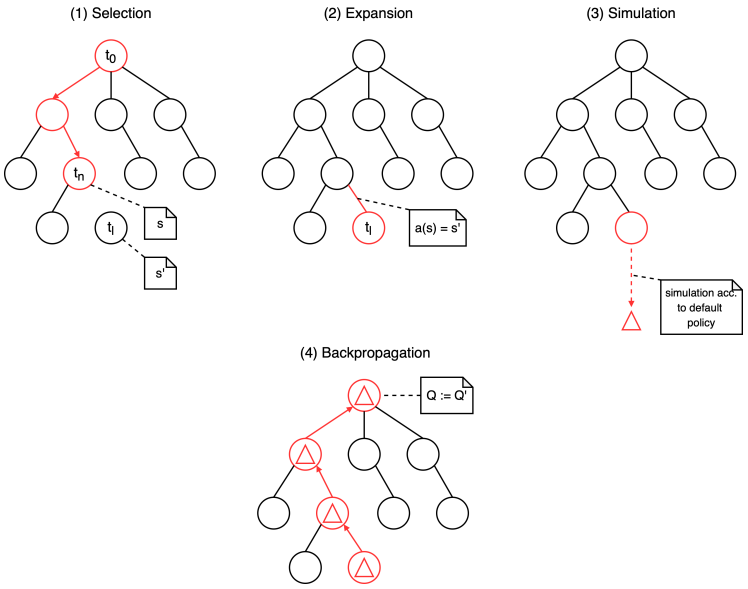

Fig. 1. The four basic MCTS steps executed in each iteration. [modified image from [6]]

knowledge can greatly improve the performance of the search, whether this manifests in modeling design decision of the problem itself or heuristics for use in the tree and/or simulation policy. b) *Anytime*: Due to the immediate backpropagation of each game outcome the values of all nodes are always up-to-date. This makes it possible to return (i.e., end the search and return the root action) at any time. c) *Asymmetric*: Since more promising nodes are favored by the search the tree does not grow at the same rate (of iterations) everywhere. The tree can gain an irregular, asymmetric shape.

2.4 Upper Confidence Bound For Trees

Upper Confidence Bound for Trees (UCT) is an important family of MCTS-Algorithms that was first proposed by Kocsis and Szespevari [20]. They follow the basic MCTS-schema shown in Algorithm 1 but define their own default- and tree-policies. In the following we discuss an implementation of UCT in pseudocode for games (except for some changes in the nomenclature, UCT is of course applicable to all generic search problems). All possible states of the game are described using nodes, each node v has four members:

- associated state $s(v)$
- incoming action $a(v)$
- total simulation reward $Q(v)$
- visit count $N(v)$

Initially all nodes except the initial state of the game are not part of the search tree but can be added ("chosen") during the search. More precisely: While the budget is not exhausted a child node is selected according to the tree policy, which is discussed in detail below. Then the simulation is executed according to the default policy of choosing actions (and thus child nodes) uniformly at random. When a terminal state is reached the results are backpropagated and the four entries in the nodes are updated accordingly. When the budget is used up the best node is returned as a result. Relating to games, the result of the search is the best action possible from the root node (the "root action").

A key concept of UCT is the treatment of child-node selection as a MAB-problem. With this abstraction we can use the UCB1 equation from above in the tree policy (see Algorithm 2):

$$UCT(v) = \frac{Q(v')}{N(v')} + C_p \cdot \sqrt{\frac{2 \ln N(v)}{N(v')}}$$

The node v' is a child of v. There are two differences to default UCB1. One is the treatment of a and s as nodes to fit the pseudocode. In the next sections we omit the treatment of $a(v)$ and $s(v)$ as members of a node for convenience and just write a and s. Read "state s represented by a node" and "action a which leads to the state s represented by a node". More consequential however is the parameter $C_p > 0$ that mediates between exploration and exploitation. Usually we just set $C_p = \frac{1}{2}$ but this can of course be tuned. Now the child selection just translates to maximizing this function. UCT always chooses the child node $v^\star$ (or, more precisely, the action $a^\star$ leading to $v^\star$) with

$$v^\star = \underset{v' \in v.children}{\arg\max} \; UCT(v)$$

2.5 Comparison to Other Algorithms

The two most notable search search algorithms that can be viewed as direct alternatives to MCTS are *Minimax/Expectimax* and *Alpha-Beta-Pruning*. While their relation to each other is clear (Alpha-Beta-Pruning is strictly better than Minimax since it is a generalization of the latter [19]) an apples-to-apples comparison of each to MCTS is difficult. Ramanujan et al. [30] show that there are both search spaces where MCTS significantly outperforms Minimax and ones where the exact opposite is true. They argue that this is due to the existence of trap states. That is, game states which are only a small number of moves away from defeat. In the real world this can be seen in the fact that while for playing Go (few traps) MCTS is a key success factor, in Chess (many traps) Minimax and its derivations are usually the better choice. On the other hand Kato et al. [18] show that when it comes to other games like Amazons Alpha-Beta-Pruning yields better results. Additionally, Baier and Winands [2] argue that when comparing search algorithms the heuristics and evaluation functions used can massively influence the results and thus the choice depends on existing domain knowledge and not primarily on the basic algorithmic framework.

Algorithm 2 The tree policy of UCT.

function TreePolicy(v)
  while v is nonterminal **do**
    if v is not fully expanded **then**
      return Expand(v)
    else
      $v \leftarrow$ BestChild(v, C_p)
  return v

function Expand(v)
  choose untried $a \in A(s(v))$
  add new child v' to $v.children$ with $s(v') = f(s(v), a)$ and $a(v') = a$
  return v'

function BestChild(v, c)
  return $\underset{v' \in v.children}{\arg\max} \frac{Q(v')}{N(v')} + C_p \cdot \sqrt{\frac{2 \ln N(v)}{N(v')}}$

Algorithm 3 The default policy of UCT.

function DefaultPolicy(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for s

function Backup(v, Δ) ▷ no strictly speaking part of the default policy
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow v.parent$
  $N(v) \leftarrow N(v) + 1$
  $Q(v) \leftarrow Q(v) + \Delta$
  $\Delta \leftarrow -\Delta$
  $v \leftarrow v.parent$

3 Variations

Many variations of the basic MCTS approach have been proposed by the research community. This section presents some notable developments in recent years. Section 3.1 considers new results with respect to Multi-Armed Bandits and methods of determining upper confidence bounds. Section 3.2 describes heuristics to improve general MCTS algorithms while Sections 3.3 and 3.4 lay out more narrow approaches to apply MCTS in certain domains, namely real-time scenarios and situations where multiple instances of the search algorithm are executed in agents that need to cooperate. Finally, Section 3.5 presents a novel way of using the framework of evolutionary algorithms in conjunction with MCTS.

3.1 MAB and UCB-Tuning

Fixed Budgets The *fixed budgets* setting (as opposed to *fixed confidence* [16]) describes the constraint of a MAB-problem that the best possible arm must be identified using no more than m "pulls". Karnin et al. [17] provide (what they call) an almost optimal exploration algorithm in this setting called *Sequential Halving* given in Algorithm 4. Each arm of the MAB problem is associated with a random variable yielding values in the interval $[0, 1]$. W.l.o.g the k different arms are ordered according to their expected value $p_i \in [0, 1]$ at every decision epoch t: $p_1, \geq p_2 \geq \ldots \geq p_k$. $\Delta_i \coloneqq p_1 - p_i$ denotes the suboptimality gap of arm i. Consequently, Δ_2 is the smallest of all these gaps. It can be shown that for the required budget T for identifying the best arm with a probability of at least $(1 - \delta)$ we have $T \in \Omega(H \log(\frac{1}{\delta}))$ where $H \coloneqq \sum_{i=2}^{k} \frac{1}{\Delta_i^2}$. H is a measure of the complexity of the problem. Another related measure is given via: $H_2 \coloneqq \max_{i \neq 1} \frac{i}{\Delta_i^2}$. With these considerations we can now analyze the algorithm: Given a budget of T pulls it identifies the best arm with a probability of at least $1 - 3 \log_2 k \cdot \exp\left(-\frac{T}{8 H_2 \log_2 k}\right)$.

Algorithm 4 Sequential Halving.

function SEQUENTIALHALVING(T)
  $S_0 \leftarrow [k]$
  for $r = 0$ to $\lceil \log_2 k \rceil - 1$ **do**
    $t_r = \left\lfloor \frac{T}{|S_r| \lceil log_2 k \rceil} \right\rfloor$
    sample every arm $i \in S_r$ t_r times
    calculate average reward $\hat{p}_i^r$
    $S_{r+1} \leftarrow \left\lceil \frac{|S_r|}{2} \right\rceil$ arms in S_r with largest average reward
  return arm in $S_{\lceil \log_2 k \rceil}$

Fixed Confidence Jamieson et al. [15] devise an optimal exploration algorithm called LIL'UCB for stochastic MAB-problems with a *fixed confidence*. More precisely they define a procedure with a single input $\delta > 0$ that solves the best arm problem with a confidence of δ. That is, it identifies the arm with the largest mean with a probability of at least $1 - \delta$ irrespective of the actual mean payoff of the arms $p_1, \ldots, p_K \in [0, 1]$. A key concept here is the *sampling* of arms, that is, the realization of a Gaussian random variable with mean p_i for arm i. Using the *law of iterated logarithms* (LIL) they prove that this procedure cannot be improved by more than a constant factor.

Non-Stationary For non-stationary MAB-problems Auer et al. [1] prove a lower bound for policy performance. Let $(V_t)_{t=1}^{T}$ be a non-decreasing sequence of numbers greater zero with $V_1 = 0$ and $KV_t \leq t$. V_T is called the *variation*

budget. If at each decision epoch t the rewards X_t^k are distributed via a Bernoulli distribution with mean μ_t^k we have for any policy π:

$$\mathcal{R}^\pi(V_T, T) \geq CK^{\frac{1}{3}}V_T^{\frac{1}{3}}T^{\frac{2}{3}}$$

where C is a constant that is independent of T and V_T. Besbes et al. [4] provide a policy with this optimal bound called *Rexp3*. Concretely, it defines a distribution $\{p_t^k\}_{k=1}^K$ over the K arms according to which the arms are drawn. The values of the distribution are weighted with values w_k^t that are updated each turn. How much is influenced by a parameter γ which itself is updated every $\Delta = \left\lceil (K \log K)^{\frac{1}{3}} (\frac{T}{V_T})^{\frac{2}{3}} \right\rceil$ epochs. Intuitively γ represent the exploration rate, Δ is the batch size and p_t^k represents the certainty of the policy that k is optimal.

Similarity Information There are a lot of MAB-problems which are, due to the large number of possible choices (i.e, arms), not computationally tractable without additional information. Slivkins [33] provides a method for using such information, namely the similarity between arms. This is done in the framework of *contextual bandits* where each round the algorithm is given a *context* containing information about rewards. Formally, in round t a context $x_t \in X$ is given and the algorithm chooses an arm $y_t \in Y$ leading to a reward $\pi_t \in [0, 1]$. We write $\mathcal{P} \subset X \times Y$ as the set of possible context-arm pairs. For each $(x, y) \in P$ there exists a distribution with expectation $\mu(x, y)$. We now define the *similarity space* as a metric space $(\mathcal{P}, \mathcal{D})$ that satisfies the following (Lipschitz-)condition for the metric $\mathcal{D}$.

$$|\mu(x, y) - \mu(x', y')| \leq \mathcal{D}((x, y), (x', y'))$$

In other words, the expectation function μ is Lipschitz-continuous on (X, P) with a Lipschitz constant of one. An r-covering of a space S is a set of subsets of S with a diameter less than r that cover P, the minimal r for which this is possible is the r-covering number $N_r(S)$. The covering dimension d is the smallest d such that $N_r(S) \leq cr^{-d}$ with a constant c. Now we (somehow) choose partitions S_X and S_Y of the context and arm space and approximate x_t and y_t via their nearest points in these partitions x_t' and y_t'. By considering these two partitions jointly, i.e., by creating a partition of the similarity space we can leverage similarity information. Omitting a lot of details, the approach is as follows: In each round the algorithm maintains a set of balls in (P, D) which, in their union, cover the whole similarity space. At the beginning of round t the context x_t is revealed whereupon the algorithm selects a ball B (B is *activated*) with $(x_t, y_t) \in B$ and arm y_t is played. Which ball B is activated is determined similarly to UCB by maximizing the expected reward (here called *confidence radius*). Once a ball is activated the value of its confidence radius may be used to modify the set of balls for the next round. For example by pruning (i.e., replacing) balls with a confidence radius that is below a certain fraction of the activated ball. This is also called "zooming". The regret of this algorithm is:

$$R(T) \leq O(T^{1-1/(2+d_X+d_Y)})(\ln T)$$

where d_X and d_X are the covering dimensions of X and Y respectively.

3.2 General Heuristics

AMAF and RAVE There exist many heuristics for MCTS. That is use-case specific enhancements and modifications of the basic MCTS steps (most importantly tree and default policies). In the context of MAB and UCT child selection one important heuristic is *All Moves As First* (*AMAF*) [12] and its enhancement *Rapid Action Value Estimation* (*RAVE*) [13]. AMAF treats every action used during the simulation (i.e., chosen by the default policy) as if it was chosen by the tree policy. This is illustrated in Figure 2 in the context of a game similar to Tic-Tac-Toe. Beginning from the given state UCT selects $(C, 2)$ as the next black position and $(A, 1)$ for the next white one. Then the progress of the game is simulated with $(B, 1)$ black, $(A, 3)$ white and $(C, 3)$ black, leading to the terminal state displayed and resulting in a win for black. During the backpropagation the values for the nodes visited by UCT are updated as before. But due to AMAF more nodes are updated: Since $(B, 1)$ (as well as the others) could have been chosen by UCT and was used during the simulation its statistics are updated as well. The value generated this way is called *AMAF score* and is generally separate from the UCT value.

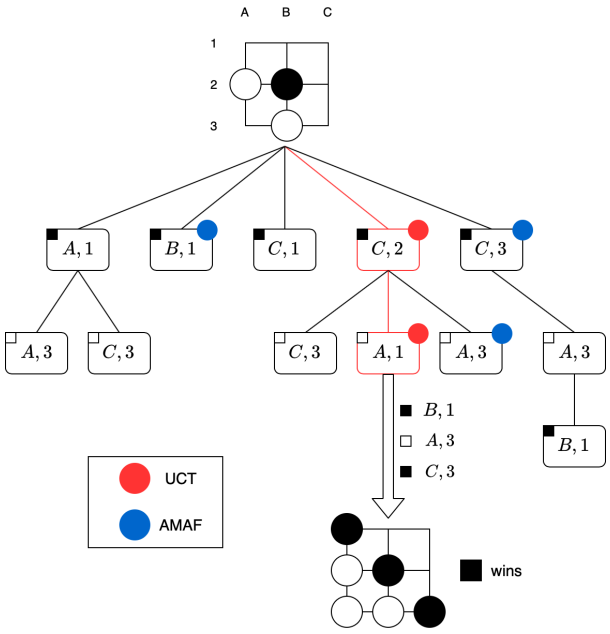

Fig. 2. The AMAF-heuristic updates all nodes that are used in the simulation as if they were selected by UCT directly. [modified image from [6]]

RAVE is a way of combining the UCT value $Q(\cdot)$ and AMAF score $AMAF(\cdot)$ for an action a at a state s via the equation:

$$RAVE(v) = (1 - \beta(v)) \cdot Q(v) + \beta(v) \cdot AMAF(v')$$

where

$$\beta(v) = \sqrt{\frac{K}{3 \cdot N(v) + K}}$$

$N(v)$ denotes the number of visits to node v. The term $AMAF(v')$ is the average result of all simulations (i.e., the backpropagated value Δ) in which v was selected after v was visited. v' is an ancestor of v. Which ancestor is dependent on the actual formulation, in its basic form *RAVE* always selects the direct predecessor of v' on the path through the search tree. *GRAVE*, described below, modifies this selection procedure. K is the so-called *equivalence parameter* which determines the number of simulations where UCT and AMAF are both considered with equal weight. Intuitively, this combination of a node and its predecessors results in generalizing simulation results to the whole subtree of the node.

GRAVE RAVE and its generaliztion GRAVE [7] are both answers to the following AMAF-tradeoff: Consider a node representing state s at any point in the search tree. If we want to combine its UCT value and the AMAF-value of its predecessors it is not obvious which predecessor we should choose: If we choose the direct predecessor s' (as does RAVE) its AMAF-value is similar to that of s but less accurate overall since the number of times s' was used in a simulation or chosen directly decreases the deeper we traverse the search tree. If we go further up the tree, i.e., choose more distant predecessors, the relationship between s and s' becomes more distant which usually means that the associated AMAF-values are also less close. However, s' is part of simulations more often so all associated values are more precise in general. GRAVE addresses these problems by relying on an additional parameter *ref*. It picks s' as the closest predecessor that has been used at least *ref* times in simulations. It is thus a true generalization as setting *ref* to zero results in RAVE. In experiments with Go GRAVE performs better than either RAVE or UCT.

Parameter Randomization and GGP Sironi and Winands [32] focus on MCTS as a search strategy in *General Game Playing* (GGP), that is the creation of agents that learn to play a variety of games while being given only their rules. These agents usually have only a limited time to prepare and execute their moves. To this end they need good (i.e., fast and accurate) search strategies to predict the course of the game. When MCTS is used in this scenario the actual actions taken by the search are controlled by the user via a number of parameters. Which values of these parameters are optimal depends on the game but since this is unknown in advance parameter tuning is often simply done offline by taking some aggregate of parameters tested on a set of games. An alternative approach is online tuning of parameters, i.e., adapting the MCTS-variant used while actually

playing the game. However, since the time available for adjustments is so short randomizing parameters often results in similar performance. It should be noted however that this does not mean just assigning truly random numbers but rather the specific mapping of parameters to a set of possible values depending on the game and the role the agent plays in it. There are four general strategies for randomizing parameters:

- *Per run*: Update once before the game is started.
- *Per turn*: Update every time the agent has to take a turn.
- *Per simulation*: Update every time a new simulation is started (similar to online tuning).
- *Per state*: Update every time a state is visited during a simulation.

Experimentally they show that tuning the parameters once *per simulation* leads to the best performance.

3.3 Real-Time

In the context of developing agents to play the arcade game Ms Pac-Man Pepels et al. [28] propose a variety of modifications to MCTS that enable real-time search, that is, search within invariable time constraints. To this end they define an array of modifications that are specific to the game in their formulation but can be generalized. Two of them are described below. One aspect concerns the creation and use of the search tree. This tree discretizes the complex game state. As shown in figure 3 nodes represent junctions in the game maze. These nodes are connected by edges with an associated length representing the distance that the player has to travel to reach them. Within "corridors" there are no decisions to be made and moves that lead back to the parent are not considered and modeled. Opponents are not represented as nodes at all. Their movements are simulated while the player traverses the tree and are an additional factor in the game state given by the player's nodes, making these states themselves only approximations. Upon reaching a node (i.e., a junction) the children considered in the expansion step are those that are i) directly reachable (i.e., neighbors of the current junction) ii) have a distance that would make the length of the search path no longer than a parameter T_{path}. The last requirement limits the depth of the search tree.

The other aspect concerns the reuse of the search tree since the quality of the results of Monte Carlo methods depends on the number of simulations which is necessarily small in real-time scenarios. Additionally, Pac-Man follows a long-term reasoning heuristic (a *plan*). Starting the search over every turn makes such long term reasoning impossible. However, simply keeping an old plan may lead to bad decisions since the values change over time, making the information used outdated. Two techniques are used to mitigate this problem when reusing trees. One is *Rule-Based Reuse*: If one of a set of conditions is met the existing tree is discarded and a new one is built. While the exact conditions are game-specific, the death of the player is a prominent example. The second technique is *Continuous*

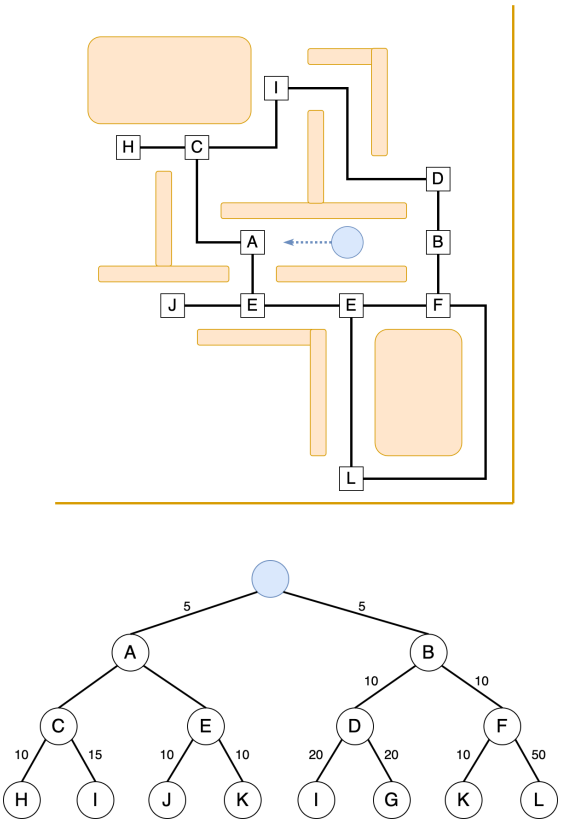

Fig. 3. A sample game state of Ms. Pac-Man and its corresponding search tree. [modified image from [28]]

Decay: The values stored at nodes describing the state of the game are multiplied by a parameter $\gamma \in [0, 1]$ at the beginning of each turn. Setting this parameter to zero means no reuse since all values are discarded and setting it to one means no decay at all.

Soemers et al. [34] present eight enhancements for real-time MCTS, again in the context of real-time games. Four examples are:

a) *Breadth-First Tree Initialization*: Sometimes the number of simulations that can be executed during a turn is smaller than the number of possible actions. This can lead to near-random behavior like selecting an action that leads to a direct loss. This problem is mitigated prior to the start of MCTS by executing a one-step Breadth-First Search from the root node generating all its successors. Then for every successor M simulations are executed and their results backpropagated. Now the actual search is started where there is now information available to avoid bad first moves. One modification is

to save the results of the M simulations directly in the successor nodes for reuse later and thereby reduce the overhead.

b) *Loss Avoidance*: If there are many loss states in a game tree the estimation of a node generated by MCTS can be overly negative. That is, if a node has many children leading to a loss state and only a few leading to a win state simulations will mostly encounter loss nodes and return a low value (for example when crossing rivers in Frogger). *Loss Avoidance* deals with this problem as follows: The first time a node is visited losses are ignored and alternatives are explored immediately. More precisely, any time a simulation involving an unexplored node ends in a loss the result is not backpropagated right away but instead the algorithm backtracks and explores the neighboring nodes. After all alternatives are exhausted only the result of the highest value is backpropagated back to the root.

c) *Novelty-Based Pruning*: Often there are many redundant paths through the game tree. The aim of *Novelty-Based Pruning* is to prune nodes in such redundant paths. To this end, a novelty measure $nov(s)$ is introduced which assigns each state (node) a score that is high if it is redundant by being very similar to other states in its neighborhood $N(s)$. This set consists of the union of four sets of states, namely
   i) the siblings on the "left" side of s
   ii) the parent of s, denoted as $p(s)$
   iii) the siblings of $p(s)$
   iv) the neighborhood of $p(s)$, i.e., $N(p(s))$

   Note that the first set introduces non-determinism since the states are generally not ordered but still only the left siblings are chosen. Now we can define $nov(s, N(s))$ as the size of the smallest tuple of features that is true in s and not true in $N(s)$. States with a high $nov(\cdot)$ value are pruned and the number of "unnecessary" paths reduced. This way we do not necessarily avoid bad paths but redundant ones.

d) *Knowledge-Based Evaluations*: Depending on the game it is often the case that simulations do not find a terminal state or one that changes the score. In such cases the same value is returned for most nodes and no meaningful information is gained. To distinguish between states that have the same evaluation a heuristic function is used. First any object in the game is assigned a type (say, "wall", "enemy", "friendly") and a weight $w_i > 0$ is computed for every type. Let Δ_{s_0} and Δ_{s_T} denote the evaluation of the current game state and that of the final state of a simulation. Let $d_0(i)$ and $d_T(i)$ denote the distance to the closest object of type i from the initial state and the terminal state, computed via the $A^\star$ pathfinding algorithm. Now a heuristic value given by the equation $\sum_i w_i \cdot (d_0(i) - d_T(i))$ is added to Δ_{s_T}. Intuitively this heuristic computes the distance to "interesting" objects of state s_T compared to s_0.

3.4 Multiple Agents

Multi-Agent MCTS Most MCTS research is focussed on competitive games, i.e., games where two or more players play against each other and each aim to win themselves. Zerbel and Ylinemi [39] focus instead on cooperation between multiple agents. That is games where multiple agents still have individual policies but have to coordinate them to achieve a common goal. The proposed algorithm *Multi-Agent MCTS* (MAMCTS) works in episodes (i.e., discrete turns). First each agent executes the four steps of MCTS Node Selection, Expansion, Simulation and Backpropagation on its own. After that each agent selects the best policy from its own tree and executes the actions contained in this policy. Now, crucially, the agents (or more precisely, their policies) are compared using difference evaluations. Using this calculated difference reward their search trees are updated. Finally the internal state of each agent is reset and the next episode is executed.

Decentralized, Multi-Agent Planning In the context of robot move planning Best et al. [5] propose a variation of MCTS that is both decentralized and online called *Dec-MCTS*. Each robot is part of a joint action space, it optimizes its own planning locally via a probabilistic distribution over all plans and periodically exchanges its search tree with other robots in a compressed version. This information is then used to update its own plans (i.e. the distribution thereof). To this end, they also define a new tree expansion policy called *D-UCT* which is, in turn, a generalization of *D-UCB* [11].

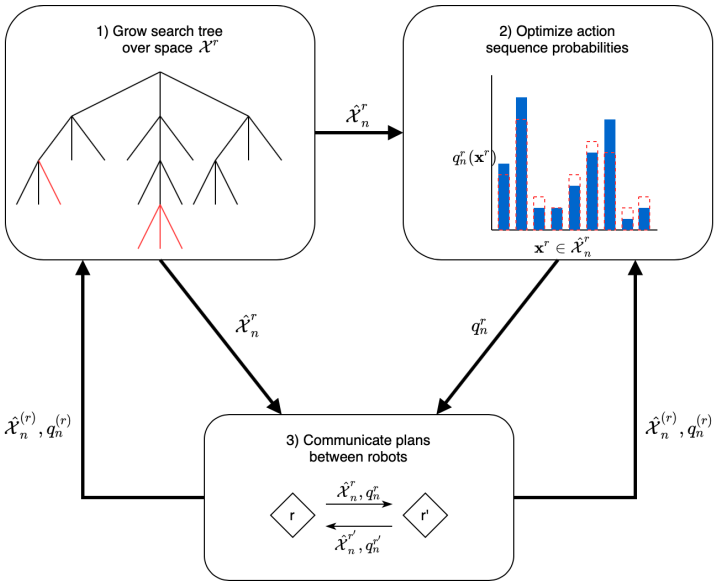

Fig. 4. The three basic steps of Dec-MCTS. [modified image from [5]]

The problem to be solved can be stated as follows: Each robot r in the set of robots $\{1, \ldots, R\}$ plans its personal sequence of future actions $\mathbf{x}^r \coloneqq (x_1^r, x_2^r, \ldots)$. An action x_j^r has a cost c_j^r and the cost of all actions must not exceed the robots budget B^r. Let $\mathcal{X}^r$ denote the set of all possible action sequences $\mathbf{x}^r$ for a robot r and let $\mathbf{x} \coloneqq \{\mathbf{x}^1, \ldots, \mathbf{x}^R\}$ denote this set for all robots. Now the aim is to maximize a global objective function $g(\mathbf{x})$. This function is known to each robot but not the action sequences selected by the others. This information must be exchanged in an asynchronous manner which usually is additionally restricted in other ways (e.g. limits on size, number and timing of messages). The proposed Algorithm Dec-MCTS is displayed in Figure 4. It consists of three phases and is running asynchronously on all robots until the computational budget is exhausted. In particular the algorithm is fault tolerant with respect to the success of the communication with other robots.

Consider robot r running the n-th iteration of the algorithm. Its current plan is defined as a probability distribution q_n^r over all action sequences $\mathcal{X}^r$. To make the problem tractable the actual domain is refined to $\hat{\mathcal{X}}_n^r \subset \mathcal{X}^r$. That is the subset of all action sequences where the probability is greater than zero. The search tree $\mathcal{T}^r$ contains only the actions of robot r, each edge representing one action and paths representing valid action sequences. In the first phase this tree is grown while considering the (set of) plans of the other robots $q_n^{(r)}$ over $\hat{\mathcal{X}}_n^{(r)}$ using the algorithm discussed below. In phase two (executed periodically) the domain and action sequences are optimized using a decentralized adaptation of gradient descent. Afterwards these changes are communicated to the other robots in phase three.

We now focus on step one: Both the tree and default policies of the standard MCTS-algorithm are adapted. To define the tree policy a so-called *discounted version* of UCT is introduced. Given a parameter $\gamma \in [0, 1]$ the definitions of $N(\cdot)$, $Q(\cdot)$ and UCT are modified:

$$N_\gamma(s) \coloneqq \sum_{u=1}^{t} \gamma^{t-u} \cdot \mathbb{I}_t(s)$$

The function $\mathbb{I}_t(s)$ is equal to one if s has been selected at turn t and zero otherwise.

$$\bar{Q}_\gamma(s) \coloneqq \frac{1}{N_\gamma(s)} \sum_{u=1}^{t} \gamma^{t-u} \cdot Q^u(s) \cdot \mathbb{I}_u(s)$$

Now the child selection function is given via:

$$\text{D-UCT}(\gamma, s) = \bar{Q}_\gamma(s) + C_p \cdot \sqrt{\frac{2 \ln N_\gamma(s)}{N_\gamma(s')}}$$

The default policy used for simulations is changed as follows. The reward predicted by a simulation can be seen as an approximation for $\mathbb{E}[g]$. Since g is a function of all plans we first need a sample of the other robots' plans $q_n^{(r)}$ before the simulation starts. However, since this sample may be inaccurate (especially compared to

our own plans $q_n^{(r)}$) we execute the simulation now with respect to g but to a local utility function f^r that is less sensitive to the possible high variance of $q_n^{(r)}$. Now the simulation is executed and the results are backpropagated.

3.5 Evolutionary Algorithms

Lucas et al. [23] describe a way of incorporating an evolutionary algorithms into MCTS in place of a heuristic. Multiple variations ("individuals") of the same MCTS-algorithm with different parameters are instantiated. Then their performance (their "fitness") is evaluated using a metric and only the best ones get to "reproduce" and/or are modified using "mutations". There are two general ways of evaluating fitness: Either the fitness of the algorithm is evaluated over a set of games and the actual effects of the algorithm (or MCTS-*agent*) are viewed as a black box whose parameters can be tuned. Or, and this is the approach taken here, the evaluation happens during the simulation (i.e., step 3 in the description given above). Each individual is defined via a parameter-vector and is evaluated a) on the same tree b) every time the default policy is executed. Consequently, there is much more information available and the evolution can happen faster. The executed steps are displayed in Algorithm 5. At first a parameter vector w is drawn and a new statistics-object S is initialized. Now the tree and default policies controlled by w are sampled K times and the resulting metrics (such as min/max, average reward) are used to evaluate the fitness of the parameter vector. After the budget (time or computational resources) is used up the best vector is chosen and returned.

Algorithm 5 Fast Evolutionary MCTS.

```
function FastEvoMCTS(K, v_0)
    while within computational budget do
        w ← evo.getNext()
        Initialize S
        for i := 1 to K do
            v_l ← TreePolicy(v_0, T(w))
            Δ ← DefaultPolicy(s(v_l), D(w))
            Backup(S, Δ)
            UpdateStats(S, Δ)
        evo.setFitness(w, S)
    w ← evo.getBest()
    a ← recommend(v_0)
    return (w, a)
```

Benbassat and Supper [3] present another evolutionary approach to MCTS that focuses on zero-sum, deterministic, full-knowledge board games while remaining scalable.

4 Use Cases

Due to its proximity to games and machine learning (MCTS can be viewed as a reinforcement learning algorithm) Monte Carlo Tree Search is widely applied to these two fields, some notable examples are detailed below as well as the use of MCTS in other domains.

4.1 Go

In the machine learning research community the board game Go has long been viewed as intractable with current methods. The number of possible moves and resulting game states is much larger than Chess for example, where the average number of legal moves is 35 with an average game length of 80 moves, resulting in a search space of 35^{80} sequences. For Go we have 250 moves, a length of 150 and thus 250^{150} sequences. In a breakthrough project called AlphaGo Silver et al. [31] combine two neural networks and MCTS to beat the European human Go champion, something that has never been done before.

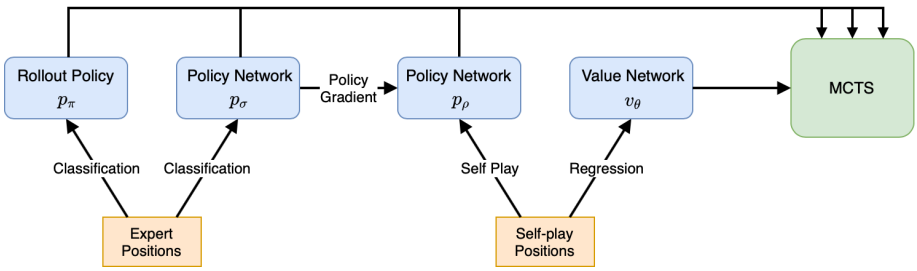

Fig. 5. The basic pipeline of AlphaGo. [modified image from [31]]

Figure 5 shows the basic steps of the approach: Initially, AlphaGo makes use of supervised learning. On the basis of input s which is a simple representation of the game state a policy network outputs a probability distribution over all legal moves. This network p_σ contains alternating convolutional layers with weights σ and rectifier nonlinearities ending in a softmax-output layer. The concrete network used is called *SL policy network* and consists of 13 layers. It was trained on 30 million positions and the associated moves of human experts, so-called state-action pairs (s, a). Concretely, stochastic gradient was used to maximize the likelihood of move a when in state s by optimizing the following equation:

$$\frac{\partial \log p_\sigma(a|s)}{\partial \sigma}$$

With a less elaborate approach (a linear softmax of fewer features) another policy $p_\pi(a|s)$ was learned. This policy is far less accurate but much faster in selecting

an action (2 μs compared to 3 ms). In a second stage reinforcement learning is applied to improve the learned policy. The *RL policy network* p_ρ is identical in structure and (initially) has the same weights $\rho = \sigma$. This network now plays against itself, that is against randomly selected previous iterations of itself. At time step t the following term is maximized (again using SGD).

$$\frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t$$

z_t is the reward $r(s_T) \in \{-1, +1\}$ at the terminal step T of the reward function $r(s)$ which is zero for any $t < T$.

In a third and final step a value network is used to estimate a value function $v^p(s)$ that predicts the outcome of games that (starting) from state s are played out by using policy p for both players, i.e., $v^p(s) = \mathbb{E}[z_t|s_t = s, a_t, \ldots, a_T \sim p]$. Concretely, the RP policy p_ρ is estimated using a value network v_θ whose architecture is similar to that of the policy networks with weights θ but returns a single value instead of a distribution. It is trained via regression on state-outcome pairs (s, z). That is the MSE between the predicted outcome $v_\theta(s)$ and the actual outcome z, $z - v_\theta(s)$ is minimized. MCTS is now used to combine the policy networks and the value network in a modification called *lookahead search* that is similar to UCT. Exactly like in UCT each node in the search tree represents a state s of the Go game and contains its incoming action a, its cumulative value $Q(s, a)$ and visit count $N(s, a)$. In addition it also contains the prior probability P as determined by the policy network $P(s, a) = p_\sigma(a|s)$. During the simulation at step t the next action is selected to maximize the term $Q(s_t, a) + u(s_t, a)$ where the latter term is a "bonus" with $u(s, a) \propto \frac{P(s,a)}{1+N(s,a)}$. When expanding a leaf node, s_L is evaluated by the value network with output $v_\theta(s_L)$ and by the result z_L of a simulation using p_π as the default policy. Both these values are combined (roughly similar to (G)RAVE) using a mixing parameter λ into a leaf evaluation $V(s_L) = (1 - \lambda)v_\theta(s_L) + \lambda z_L$. This value is then backpropagated. It is worth noting that this procedure uses the weaker policy p_σ instead of the stronger policy p_ρ because the former seems to be more similar to human Go-play. However, in direct competition p_ρ wins 80% of the time.

4.2 Neural Networks and Machine Learning

Video Games Guo et al. [14] apply an UCT-algorithm to generate training data for classifiers combining reinforcement learning and deep learning. These classifiers are then used in an AI system for playing Atari video games. For the seven games they evaluated MCTS beat the previous-state of the art AI for game playing: DQN [24]. Concretely, the emulator for each game is accessed at a certain state yielding a deterministic Markov Decision Problem. This is then solved by UCT with 3 parameters: the number of trajectories, the maximum depth and an exploration constant. With these parameters the trajectories are simulated. Consider trajectory k at state s and depth d, i.e., the search is at the state-action pair (s, d). For each possible action a a score is calculated and the action

with the biggest score is selected. The term to be maximized is (mostly) just a reformulation of UCT: It is the sum of the average reward of simulations in (s, d) in the previous $k-1$ trajectories and the exploration term $\sqrt{\log(n(s,d))/n(s,a,d)}$ where $n(s, a, d)$ is the number of times a has been selected at (s, d) and $n(s, d)$ is the total number of simulations for the previous trajectories at this pair.

To use the UCT agents in neural networks they propose three methods:

- *UCT to Regression*: For each game the UCT agents runs 800 times, the dataset to train a regression-based CNN is created by saving the last four actions selected by the agents for each state and each trajectory. The result can now be viewed as a table where the rows correspond to the last four frames for each state and trajectory and the single row contains the best action (as determined by UCT).
- *UCT to Classification*: Since the dataset above only contains a single action for each state/trajectory pair it can be easily used for classification as well with the action being the label. This is done via a CNN implementing multinomial classification.
- *UCT to Classification Interleaved*: This method alternates data collection and playing. First the agent is run 200 times. Now the classification is executed as above. Its results are then used to inform the agents choices in another 200 runs, potentially leading to different results.

Stanescu et al. [35] compare various MCTS-variations with each other as well as other search algorithms with respect to their performance in Real Time Strategy (RTS) Games and show that combining MCTS with convolutional neural networks beats many conventional state-of-the-art algorithms. To this end a simple game was designed where two players fight each other by gathering resources, creating buildings that produce units and controlling four unit types: workers, light, ranged and heavy units. Four strategies were implemented: *WorkerRush* where all workers are directly sent to fight and {Light,Ranged,Heavy}-Rush where a barracks are built and the corresponding units are produced and sent to fight. These strategies can be viewed as actions taken by a MCTS algorithm. They introduce and compare four different variations with increasing complexity and performance:

a) *Naive MCTS*: Simple MAB approach to select the action at the current game state.
b) *Greedy MCTS*: MCTS using a greedy sampling strategy (more exploitation, less exploration).
c) *AHTN-P*: An Adversarial Hierarchical Task Network using minimax gametree search as well as a RTS-variation of MCTS presented by Ontanon [27].
d) *AHTN-F*: A more elaborate NN using the approach above.

Deep Learning Architectures The problem of choosing the architecture of a neural network is critical as its performance is usually greatly affected by it. However, most research focuses on the tuning of hyperparameters, not the architecture itself resulting in a situation where the success or failure of a projects rests

on intuition. To improve this process, Negrinho and Gordon [25] introduce a framework for the automatic generation and testing of different architectures. It consists of two parts: First, the *model search specification language* allowing for the description of (complex) models, i.e., black boxes that make up parts of a neural network which are only defined by their interface (their hyperparameters) and may themselves be made up of components. Any search space defined this way will be a tree, a property used by the second component, the *model search algorithm*. This algorithm determines how (and which parts of) the search space are explored. Here, a variation of MCTS is used. Its most critical subroutine is the model evaluation algorithm which determines the performance of a fully specified architecture, that is a leaf in the tree. Once a search space is defined

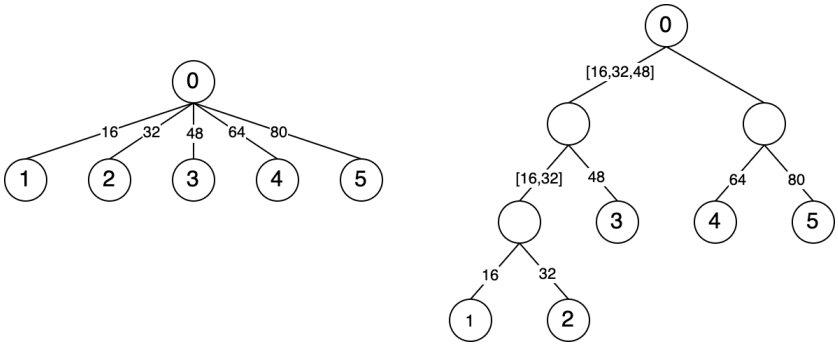

Fig. 6. Through bisection MCTS can evaltate a large number of possible values. [modified image from [25]]

(via a process not described here) it is explored by incrementally building a tree. The build process consists of assigning the hyperparameters of a model until it is completely specified whereby any assignment results in moving down the tree by traversing another edge. Thus, any leaf is a completely specified model and inner nodes still have unassigned parameters, the possible values of which determine the branching factor. Examples of possible assignments are integers for the number of filters of a convolutional module, or one of a relatively small set of activation functions, whether there should be a dropout layer etc. It it also important to note that (like in games) previous choices affect the number and kind of hyperparameters that are assignable later: a module that has not been selected does also need not be configured, making the path through the tree shorter.

The MCTS-algorithm used corresponds directly to UCT. The evaluation function is implemented as a machine learning prediction (i.e., as a prediction of a model trained on a training set). The default policy is simple as well, it just selects random nodes. One optimization is introduced, however. The possible values that a hyperparameter can be assigned are often numeric and thus have two properties: There are a large number of possible values but at the same

time similar values are expected to yield similar performance. We can therefore make use of a bisection of the tree. Assuming a natural ordering, rather than picking a single value at a decision we choose only in which half of the set of possible values the hyperparameter will lay. This process is displayed in Figure 6. One can also tune this process by experimenting with different thresholds which results in a tradeoff between breadth and depth. Experiments demonstrate that this method is superior to previous work in this area while only requiring very high-level knowledge.

Wang et al. [37] also use MCTS in the exploration of neural network architectures in their project *AlphaX*. In the benchmark suite NASBench-101 [38] their method provided a 3x speedup over simple random search. Similarly to the approach above they again define a search space of possible architectures that gets explored using Monte Carlo Tree Search. The elements of said search space are evaluated by a Meta-Deep Neural Network (DNN). While the search goes on AlphaX simultaneously generates (more) training data for the Neural Network. However, unlike in the other project there is explicit support for multiple kinds of work spaces. One of them is *NASNet* where neural networks are made up of cells consisting of the layers and connections of neural networks. Like modules, they can be built recursively. The other space is called *NasBench* which corresponds to a directed acyclic graph (DAG). Its nodes represent layers and its edges the connections between the layers. Since the search algorithm used is again UCB the remainder of this section focuses on the simulation step.

Generally, since neural network training is expensive (w.r.t. both time and computing resources) the actual number of times a simulation can be run is limited which decreases the accuracy of the search. To mitigate this a hybrid approach is used. The conventional training of the network is complemented by a performance prediction of the meta-DNN. This network aims to predict the performance of unseen architecture based on previously evaluated architectures as a regression task. Thus it receives a vector-representation of an architecture as input and outputs a performance metric. This encoding is rather involved but can be viewed as mapping blocks and their relations to numbers. Its training data is generated by traversing the search space. This way the Q-value of an action at a state s is approximated via the following equation:

$$Q(s,a) \approx \frac{1}{2}\left(Acc(sim_0(a(s))) + \frac{1}{k}\sum_{i=1}^{k} Pred(sim_i(a(s)))\right)$$

$sim_i(s)$ denotes the result of the i-th simulation starting from s, $Acc(\cdot)$ is the actual accuracy of a trained network and $Pred(\cdot)$ is the predicted accuracy of the meta-DNN.

4.3 Other Use Cases

Databases Omondi [26] applies UCT to automatically tune configuration parameters of databases. Previously these had to be manually analyzed and modified by system administrators as a reaction to environmental or runtime changes to avoid

bottlenecks. The state of the database system at a point in time is quantified by measures such as the number of concurrent users, transaction throughput and the measured response time for queries. The goal of the algorithm is to optimize two of these quantities: i) maximize the throughput, ii) minimize the response time. Concretely, optimizing for these goals means selecting from the actions available in the current state that lead to subsequent states which perform better with respect to said goals. As described below the proposed algorithm does not actually optimize both at the same time but only one depending on the kind of database. If the workload is classified as *OLTP* (Online Transaction Processing, many writes but limited reads) it is optimized w.r.t. throughput and if the workload is labeled as *OLAP* (Online Analytical Processing, many reads and few writes) the goal is to minimize the response time.

The MCTS algorithm used is a combination of UCT and a variation of a heuristic proposed by Stankiewicz [36] called *Last Good Reply with Forgetting* (LGRF). The variation is called *Lean LGRF*. It is used as an additional term in the UCT formula during the selection step:

$$\frac{Q(v')}{N(v')} + LGRF_{lean}(v, v') + C_p \cdot \sqrt{\frac{2 \ln N(v)}{N(v')}}$$

Graph Matching A (generalized) geometric graph is a tuple $G = (V, E)$ with vertices $V \subseteq \mathbb{R}^d$ and edges $E \subseteq V \times V$ where the edges are described as curves. More precisely $e \in E$ is given via a continuous function $\xi_e : [0, 1] \to \mathbb{R}^d$ by $e = (\xi_e(0), \xi_e(1))$. The curve itself is the image of this function: $\xi_e(I) = \{\xi_e(t) | t \in I\}$. Intuitively, geometric graphs are regular graphs but the "actual location" of edges is defined and matters. Pinheiro et al. [29] use MCTS to define an algorithm for the problem of graph matching in this general case. The matching problem is defined as follows: Given (w.l.o.g) two graphs, determine which parts (nodes and edges) are present in both graphs. Matching of arbitrary curves can be done computationally be calculating the distance in space using some metric (e.g. Euclidean) and checking if the result is below some threshold. Clearly this is a much harder problem than matching non-geometric grahps and the well-understood theory of graph matching is not directly applicable.

Since the relationship between curves of different graphs is not immediately clear (that is, two curves in one graph may correspond to only one in the other) the concept of superedges is used. A superedge $s = (e_1, e_k)$ consists of a sequence of k edges, its corresponding curve ξ_s is the concatenation of the curves $\xi_{e_1}, \ldots \xi_{e_k}$. For a graph G let S denote the set of all its superedges. Then, a matching M between two geometric graphs G_1, G_2 can be defined as a set of superedge pairs $M \subseteq S_1 \times S_2$. The quality Q of a matching M consisting of a node matching M^V and an superedge matching M^S between graphs G_1, G_2 can be evaluated using the equation $Q(M^V, M^S) = l(M^S) + \kappa \, \overline{l(S_1, S_2)} \, \left|M^V\right|$ where $l(M^S)$ is the sum of the length of the edges that are members of the superedges and $\overline{l(S_1, S_2)}$ is the average length of the superedges of both graphs and $\kappa > 0$ is a constant set to 0.8 in the algorithm below. The proposed MCTS algorithm is illustrated in Figure 7.

It starts with an empty matching and in each step adds a superedge pair. For the child selection a variant of UCT is used by maximizing the equation:

$$\frac{Q(v')}{Q_{norm}(v')} + C_p \cdot \sqrt{\frac{2 \ln N(v)}{N(v')}}$$

The denominator $Q_{norm}(\cdot)$ is a normalization factor and upper bound for the node reward $Q(v')$.

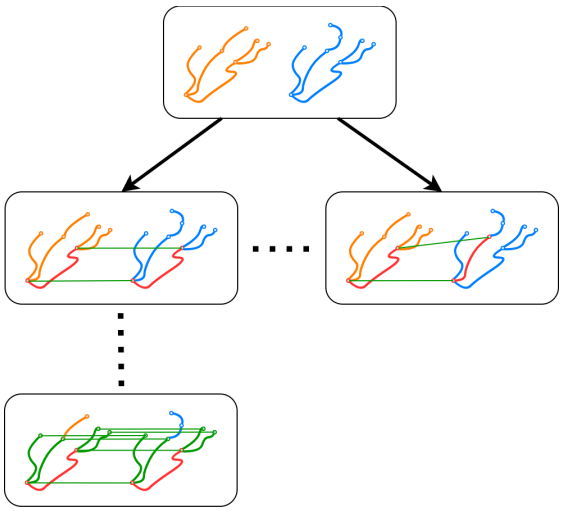

Fig. 7. The MCTS matching algorithm. Different actions lead to selecting different superedges. The simulation result displayed is the optimal matching for the given graphs. [modified image from [29]]

5 Conclusion

This paper gives an overview of research developments concerning the tree search algorithm Monte Carlo Tree Search. The algorithm iteratively executes four steps: In each iteration the current tree (usually a subset of the *game tree*) is traversed starting from the root until a (new or existing) node is selected and expanded. The exact details of this process are defined by the tree policy. Now the default policy is used to run a simulation whose results are then backpropagated up the tree. This way values are assigned to the states contained in the tree. When a computational budget is exhausted the action leading to the most valuable state from the root of the tree is returned as the result. The most important implementation of this basic algorithm is UCT which models the node selection in the first step as a Multi-armed Bandit problem and aims to maximize the

expected value of the selected node. There are multiple ways of achieving this. Section 3.1 gives algorithms that solve this problem in various settings (such as when only a limited number of turns is available or a dynamic change of rewards is possible). The final step of backpropagation and deciding which recorded statistics are updated is also open to modification. Section 3.2 (among more general aspects) presents a heuristic called AMAF which does exactly this and extends the nodes that are modified after each simulation.

Other variations of MCTS describe its application in different settings. Section 3.4 examines two modifications that allow for it to be executed on multiple agents that (unlike in the game-theoretic focus of most research) cooperate and communicate to achieve a common goal mostly related to planning. Section 3.3 is about the real-time setting where time is limited and special heuristics allow for the reuse of results from previous iterations or to aim to avoid running into dead-ends. Section 3.5 describes ways of combining MCTS with the paradigm of evolutionary algorithms which take inspiration from biology to evaluate the fitness (performance) of multiple, slightly different MCTS-instances and aim to increase this fitness both by randomly modifying and deterministically combining the best available instances.

MCTS was made famous by its application in the game of Go. Its most recent breakthrough as part of AlphaGo [31] (a neural network that was the first AI to beat a human champion) is described in Section 4.1. Further use cases such as in the design of neural network architectures, as database optimization or for (more theoretical) graph matching algorithms are discussed in Section 4.2 and Section 4.3 respectively. Such use cases (as well as others) have shown MCTS to be effective in large domains or when no additional domain knowledge is available. While MCTS does not always perform better than comparable search algorithms, variations like UCT generally address the exploration-exploitation tradeoff very effectively. At the same time they are open to a large number of modifications to fine-tune the performance if additional domain knowledge exists or specific properties are required. This results in a large body of current and possible future research as MCTS gets applied to an ever-increasing range of domains.

References

1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine learning **47**(2-3), 235–256 (2002)
2. Baier, H., Winands, M.H.: Monte-carlo tree search and minimax hybrids with heuristic evaluation functions. In: Workshop on Computer Games. pp. 45–63. Springer (2014)
3. Benbassat, A., Sipper, M.: Evomcts: A scalable approach for general game learning. IEEE Transactions on Computational Intelligence and AI in Games **6**(4), 382–394 (2014)
4. Besbes, O., Gur, Y., Zeevi, A.: Optimal exploration–exploitation in a multi-armed bandit problem with non-stationary rewards. Stochastic Systems **9**(4), 319–337 (2019)

5. Best, G., Cliff, O.M., Patten, T., Mettu, R.R., Fitch, R.: Dec-mcts: Decentralized planning for multi-robot active perception. The International Journal of Robotics Research **38**(2-3), 316–337 (2019)
6. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games **4**(1), 1–43 (2012)
7. Cazenave, T.: Generalized rapid action value estimation. In: Twenty-Fourth International Joint Conference on Artificial Intelligence (2015)
8. Chang, H.S., Fu, M.C., Hu, J., Marcus, S.I.: An adaptive sampling algorithm for solving markov decision processes. Operations Research **53**(1), 126–139 (2005)
9. Chaslot, G., Bakkes, S., Szita, I., Spronck, P.: Monte-carlo tree search: A new framework for game ai. In: AIIDE (2008)
10. Coulom, R.: Efficient selectivity and backup operators in monte-carlo tree search. In: International conference on computers and games. pp. 72–83. Springer (2006)
11. Garivier, A., Moulines, E.: On upper-confidence bound policies for switching bandit problems. In: International Conference on Algorithmic Learning Theory. pp. 174–188. Springer (2011)
12. Gelly, S., Silver, D.: Combining online and offline knowledge in uct. In: Proceedings of the 24th international conference on Machine learning. pp. 273–280 (2007)
13. Gelly, S., Silver, D.: Monte-carlo tree search and rapid action value estimation in computer go. Artificial Intelligence **175**(11), 1856–1875 (2011)
14. Guo, X., Singh, S., Lee, H., Lewis, R.L., Wang, X.: Deep learning for real-time atari game play using offline monte-carlo tree search planning. In: Advances in neural information processing systems. pp. 3338–3346 (2014)
15. Jamieson, K., Malloy, M., Nowak, R., Bubeck, S.: lil'ucb: An optimal exploration algorithm for multi-armed bandits. In: Conference on Learning Theory. pp. 423–439 (2014)
16. Jamieson, K., Nowak, R.: Best-arm identification algorithms for multi-armed bandits in the fixed confidence setting. In: 2014 48th Annual Conference on Information Sciences and Systems (CISS). pp. 1–6. IEEE (2014)
17. Karnin, Z., Koren, T., Somekh, O.: Almost optimal exploration in multi-armed bandits. In: International Conference on Machine Learning. pp. 1238–1246 (2013)
18. Kato, H., Fazekas, S.Z., Takaya, M., Yamamura, A.: Comparative study of monte-carlo tree search and alpha-beta pruning in amazons. In: Information and Communication Technology-EurAsia Conference. pp. 139–148. Springer (2015)
19. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial intelligence **6**(4), 293–326 (1975)
20. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: European conference on machine learning. pp. 282–293. Springer (2006)
21. Lai, T.L., Robbins, H.: Asymptotically efficient adaptive allocation rules. Advances in applied mathematics **6**(1), 4–22 (1985)
22. Lattimore, T., Szepesvári, C.: Bandit algorithms. preprint p. 28 (2018)
23. Lucas, S.M., Samothrakis, S., Perez, D.: Fast evolutionary adaptation for monte carlo tree search. In: European Conference on the Applications of Evolutionary Computation. pp. 349–360. Springer (2014)
24. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
25. Negrinho, R., Gordon, G.: Deeparchitect: Automatically designing and training deep architectures. arXiv preprint arXiv:1704.08792 (2017)

26. Omondi, A.O.: A Monte Carlo tree search algorithm for optimization of load scalability in database systems. Ph.D. thesis, Strathmore University (2019)
27. Ontanón, S.: The combinatorial multi-armed bandit problem and its application to real-time strategy games. In: Ninth Artificial Intelligence and Interactive Digital Entertainment Conference (2013)
28. Pepels, T., Winands, M.H., Lanctot, M.: Real-time monte carlo tree search in ms pac-man. IEEE Transactions on Computational Intelligence and AI in games **6**(3), 245–257 (2014)
29. Pinheiro, M.A., Kybic, J., Fua, P.: Geometric graph matching using monte carlo tree search. IEEE transactions on pattern analysis and machine intelligence **39**(11), 2171–2185 (2016)
30. Ramanujan, R., Sabharwal, A., Selman, B.: On the behavior of uct in synthetic search spaces. In: Proc. 21st Int. Conf. Automat. Plan. Sched., Freiburg, Germany (2011)
31. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of go without human knowledge. Nature **550**(7676), 354–359 (2017)
32. Sironi, C.F., Winands, M.H.: Comparing randomization strategies for search-control parameters in monte-carlo tree search. In: 2019 IEEE Conference on Games (CoG). pp. 1–8. IEEE (2019)
33. Slivkins, A.: Contextual bandits with similarity information. The Journal of Machine Learning Research **15**(1), 2533–2568 (2014)
34. Soemers, D.J., Sironi, C.F., Schuster, T., Winands, M.H.: Enhancements for real-time monte-carlo tree search in general video game playing. In: 2016 IEEE Conference on Computational Intelligence and Games (CIG). pp. 1–8. IEEE (2016)
35. Stanescu, M., Barriga, N.A., Hess, A., Buro, M.: Evaluating real-time strategy game states using convolutional neural networks. In: 2016 IEEE Conference on Computational Intelligence and Games (CIG). pp. 1–7. IEEE (2016)
36. Stankiewicz, J.A., Winands, M.H., Uiterwijk, J.W.: Monte-carlo tree search enhancements for havannah. In: Advances in Computer Games. pp. 60–71. Springer (2011)
37. Wang, L., Zhao, Y., Jinnai, Y., Tian, Y., Fonseca, R.: Alphax: exploring neural architectures with deep neural networks and monte carlo tree search. arXiv preprint arXiv:1903.11059 (2019)
38. Ying, C., Klein, A., Christiansen, E., Real, E., Murphy, K., Hutter, F.: Nas-bench-101: Towards reproducible neural architecture search. In: International Conference on Machine Learning. pp. 7105–7114 (2019)
39. Zerbel, N., Yliniemi, L.: Multiagent monte carlo tree search. In: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems. pp. 2309–2311. International Foundation for Autonomous Agents and Multiagent Systems (2019)