Monte Carlo Tableau Proof Search

Michael Färber[1(✉)], Cezary Kaliszyk[1], and Josef Urban[2]

[1] Universität Innsbruck, Innsbruck, Austria
{michael.faerber,cezary.kaliszyk}@uibk.ac.at
[2] Czech Technical University in Prague, Prague, Czech Republic
josef.urban@gmail.com

Abstract. We study Monte Carlo Tree Search to guide proof search in tableau calculi. This includes proposing a number of proof-state evaluation heuristics, some of which are learnt from previous proofs. We present an implementation based on the leanCoP prover. The system is trained and evaluated on a large suite of related problems coming from the Mizar proof assistant, showing that it is capable to find new and different proofs.

1 Introduction

Recent advances in Automated Reasoning include both theoretical improvements in the calculi, including combining superposition with SAT solving in recent versions of Vampire [4] and research on the InstGen calculus in iProver [16], but also more practical improvements, such as more efficient and precise term indexing techniques [24], efficient non-clausal tableau proof search [20], or the use of machine learning for problem size reduction [12]. Furthermore, many automated reasoning techniques have been extended to interesting theories beyond first-order logic, including the developments in SMT solving in CVC4 [3] or to higher-order logic [6,29]. Many of these developments have been of great value for interactive theorem provers, whose most powerful general purpose automation techniques today rely on automated reasoning tools [5].

However, current automated theorem provers are still quite weak in finding more complicated proofs, especially over large formal developments [27]. The search typically blows up after several seconds, making the chance of finding proofs in longer times exponentially decreasing [2]. This behaviour is reminiscent of poorly guided search in games such as chess and Go. The number of all possible variants there typically also grows exponentially, and intelligent guiding methods are needed to focus on exploring the most promising moves and positions.

The guiding method that has recently very significantly improved automatic game play is Monte Carlo Tree Search (MCTS), i.e., expanding the search based on its (variously guided) random sampling [7]. Recent developments in MCTS include combination of exploration and exploitation [15], combination of online and offline knowledge with the All-Moves-As-First (AMAF) heuristic [9], and adaptive tuning of rollout policies during search [21]. As shown for example in the AlphaGo system [25], machine learning can be used to train good

L. de Moura (Ed.): CADE 2017, LNAI 10395, pp. 563–579, 2017.
DOI: 10.1007/978-3-319-63046-5_34

position evaluation heuristics even in very complicated domains that were previously thought to be solely in the realm of "human intuition". From the point of game theory, automated theorem proving is a combinatorial single-player game. For some games in this category, including SameGame [22] and the NP-hard Morpion Solitaire [21], MCTS has produced state-of-the-art players. [11] shows that proof search can be positively guided by one-step lookahead, and MCTS allows approximation of multi-step lookaheads by use of random sampling. While "finishing the randomly sampled game" – as used in the most straightforward MCTS for games – is not always possible in ATP (it would mean finishing the proof), there is a chance of learning good *proof state evaluation heuristics* that will guide MCTS for ATPs in a similar way as e.g. in AlphaGo.

In this work, we study MCTS methods that can guide the search in automated theorem provers, and evaluate their impact on interactive theorem proving problems in first-order logic. We focus on the tableau calculus and on the leanCoP prover [18], which has a compact implementation that is easy to experiment with. We can also build on previous machine learning extensions of leanCoP [13,28]. To our knowledge, this is the first time MCTS has been applied to theorem proving.

Contributions

We introduce a set of MCTS heuristics tailored to proof search including two state transition probability heuristics, three state evaluation heuristics, and two tree expansion policies related to restricted backtracking (Sect. 4). Furthermore, we present an implementation interleaving a traditional proof search with MCTS (Sect. 5) and measure its performance on a set of Mizar Mathematical Library problems (Sect. 6).

2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a technique to guide search in a large decision space by taking random samples and evaluating their outcome. First, we will establish a format for problems tractable with MCTS. Then, we give a notation for Monte Carlo trees. Finally, we show how to create and evolve a Monte Carlo tree for given problems with problem-specific heuristics.

2.1 Problem Setting and Example

A tree search problem can be minimally characterised with:

- a set of states $\mathcal{S}$,
- an initial state $s_0 \in \mathcal{S}$, and
- a state transition function $\delta : \mathcal{S} \to 2^{\mathcal{S}}$.

As an example of a tree search problem, consider the travelling salesman problem: A salesman has to visit a set of cities C and wants to minimise the total distance travelled, where $d(c_1, c_2)$ is the distance between two cities.

A possible tree search characterisation of the travelling salesman problem is:

- The set of states $\mathcal{S}$ are the sequences of cities visited so far; for example [Prague], [Prague, Vienna, Bratislava], [Paris].
- The initial state s_0 is the empty sequence.
- The state transition function δ returns for a sequence of already visited cities the set of sequences where one previously unvisited city is added; i.e. $\delta(s) = \bigcup_{c \in C, c \notin s} [s, c]$. For example, δ([Prague, Vienna]) could contain [Prague, Vienna, Budapest] and [Prague, Vienna, Bratislava].

The number of states of the travelling salesman problem is exponential in the number of cities, therefore constructing a whole tree to obtain an optimal solution is not feasible. To bias the tree search towards more promising regions, we define two types of heuristics:

- the probability $P : \mathcal{S} \to [0, 1]$ of choosing a state and
- the reward $\rho : \mathcal{S} \to [0, 1]$ for a state.

$P(s' \mid s)$ is the probability for choosing state s' when being in its predecessor state s, where $s' \in \delta(s)$. In the travelling salesman example,

$$P(\text{[Prague, Vienna]} \mid \text{[Prague]}) > P(\text{[Prague, Paris]} \mid \text{[Prague]})$$

means that when the salesman is in Prague, Vienna should be chosen as next city with a higher probability than Paris.

$\rho(s)$ is the overall quality of a (final) state s. Due to the MCTS flavour we use [15], it has to be normed between [0, 1]. In the travelling salesman example, a sensible ρ should yield larger values for city sequences with smaller overall distance.

2.2 Trees

A Monte Carlo tree stores the states that have been expanded during a tree search, and keeps statistics about the states. We define the set $\mathcal{T}$ of Monte Carlo tree nodes.

Definition 1 (Monte Carlo tree node). *A* Monte Carlo tree node *is a 5-tuple* $(n, r, s, S, T) \in \mathcal{T}$*, where:*

- $n \in \mathbb{N}$ *is the number of times the node was visited,*
- $r \in \mathbb{R}$ *is the sum of the rewards of successors,*
- $s \in \mathcal{S}$ *is the state of the node,*
- $S \in 2^{\mathcal{S}}$ *are the unvisited successor states of* s*, and*
- $T \in 2^{\mathcal{T}}$ *are the child tree nodes.*

Monte Carlo Tree Search evolves a tree by repeatedly applying a step function until a certain criterion is fulfilled, e.g. a certain number of steps is performed, time has elapsed etc. Ideally, every step should refine the quality estimate of the states in the Monte Carlo tree. We show the step function in the next section.

2.3 Monte Carlo Step Function

The Monte Carlo step function performs the following: First, it selects a *node* in the Monte Carlo tree. From the state of *node*, it randomly samples a sequence of successor states (called *simulation*). It then creates a new $node'$ with some state from the simulation and makes it a child of the original *node*. Finally, a reward is calculated from the simulation and backpropagated to all ancestors of $node'$.

The idea is that rewards obtained from simulations starting from a certain node let us estimate the usefulness of the node itself. A description of the pseudocode in Algorithm 1 follows. For brevity, the pseudocode assumes that every state has at least one successor state (i.e., for every $s \in \mathcal{S}$, $|\delta(s)| > 0$).

Algorithm 1. Monte Carlo step function.

```
 1: procedure STEP(node)
 2:     if node.S = ∅ then
 3:         best ← arg max_{t∈node.T} UCT(node.n, t)                ▷ selection
 4:         reward ← STEP(best)
 5:     else
 6:         s' ← BIASEDDRAW(P, node.s, node.S)
 7:         sim ← SIMULATION(D, s')                                 ▷ simulation
 8:         (node', reward) ← EXPANSION(sim)                        ▷ expansion
 9:         node.S ← node.S \ {s'}
10:         node.T ← node.T ∪ {node'}
11:     node.n ← node.n + 1                                         ▷ backpropagation
12:     node.r ← node.r + reward
13:     return reward
```

In l. 3, the step function recursively selects the child node with the highest UCT (Upper Confidence Bounds for Trees) value [15]. UCT establishes an order on nodes, combining *exploration* and *exploitation*: *exploration* prefers less frequently visited nodes, whereas *exploitation* prefers nodes with higher average reward. The ratio between these two goals is determined by the exploration constant C_p, where higher values give more emphasis to exploration. The average reward of a node (n_j, r_j, s, S, T) is $\frac{r_j}{n_j}$. The UCT function takes the number of times n that a parent node was visited, as well as a child node:

$$\text{uct}(n, (n_j, r_j, s, S, T)) = \frac{r_j}{n_j} + C_p \sqrt{\frac{\ln n}{n_j}}.$$

As soon as a node with unvisited successor states is encountered (l. 5), an unvisited successor state s' is drawn (l. 6), where the probability of picking s' is proportional to $P(s' \mid s)$. From the chosen s', a simulation is performed up to a constant simulation depth D. A simulation starting from a state s_i draws a state s_{i+1} from $\delta(s_i)$, with probability $P(s_{i+1} \mid s_i)$. This is repeated for a certain

number of times, yielding a simulation $[s_1, \ldots, s_D]$, where D is the simulation depth and for every i, $s_{i+1} \in \delta(s_i)$.

From the simulation, the expansion operation yields a new $node'$ and a $reward$. The default expansion policy creates $node'$ from the first state s_1 of the simulation and calculates the reward from the last state, i.e. $reward = \rho(s_D)$. Therefore, the new expansion node is $node' = (1, \rho(s_D), s_1, \delta(s_1), \emptyset)$.

The expansion node $node'$ is added to the child nodes (l. 10) and the $reward$ is propagated back until the root (l. 11–13).

3 Tableau

In this section, we shortly recall some basics of tableau [10] and introduce notions specific to representing tableau as MCTS.

Tableau calculi are methods to prove the inconsistency of formulae. A tableau is a tree with formulae as nodes. The root is the formula whose inconsistency one attempts to show. All other nodes are produced by application of tableau rules to nodes above them; such rules include α-rules to treat conjunctions and β-rules to treat disjunctions. For example, when a branch contains a disjunction (called β-formula), then an application of a β-rule (parametrised by the disjunction) adds the disjuncts as children to some leaf of the branch.

The choice of β-formulae in tableau proof search is one of the main sources of nondeterminism and has a considerable impact on the length of the proof search. It corresponds to the choice of given clauses in saturation-based provers and extension clauses in the connection calculus. Therefore, in this work, we focus on influencing the proof search mostly by influencing the choice of β-formulae. We will abstract from the actual tableau steps, only assuming that the considered tableau calculus that is sound and complete.

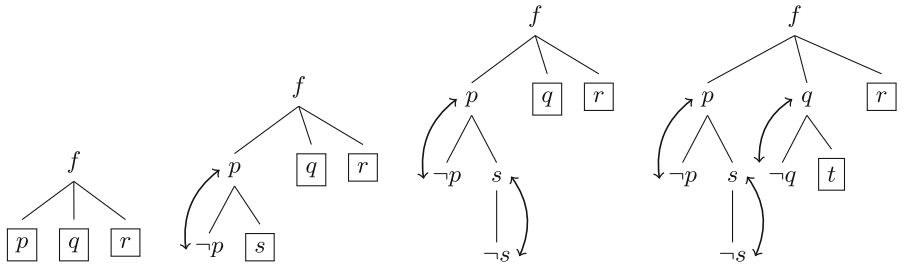
A branch of a tableau is closed iff it contains some formula and its negation. A tableau is closed iff all of its branches are closed. A formula is proven inconsistent when there is a closed tableau with the formula at the root. For the heuristics in Sect. 4, we define *β-children*, which correspond to *open subgoals* in interactive theorem provers and literals of *open branches* in the connection calculus.

Definition 2 (β-children). *Given a tableau t, we call direct children of branches* β-children *of t and denote them as $\beta(t)$.*

Open β-children *(denoted $\beta_o(t)$) are all β-children on open branches not having any branch as descendant.* Closed β-children *(denoted $\beta_c(t)$) are all β-children that are not open, i.e. $\beta_c(t) = \beta(t) \setminus \beta_o(t)$.*

Example 1. In state 4 in Fig. 1, closed β-children are p, $\neg p$, s, q, and $\neg q$. Open β-children are t and r.

The *successor tableaux* of a tableau are all tableaux that can be obtained from the original tableau by the application of some rule. We now give a description of tableau construction for a given formula f in the language of Sect. 2.1:



(a) State 1. (b) State 2. (c) State 3. (d) State 4.

Fig. 1. Proof search for formula $f = (p \vee q \vee r) \wedge (\neg p \vee s) \wedge (\neg p \vee t \vee u) \wedge \neg s \wedge (\neg q \vee t) \wedge (\neg q \vee s)$. Open β-children are surrounded by boxes.

- The set of states $\mathcal{S}$ is the set of tableaux.
- The initial state s_0 is a tableau containing only the formula f as root.
- The transition function $\delta(s)$ obtains all successor tableaux of s produced by applications of tableaux rules.

This characterisation in conjunction with the default expansion policy from Sect. 2.3 has the downside that its Monte Carlo trees are approximately as deep as the number of proof *steps*, whereas the corresponding tableaux are as deep as the maximal proof *depth*. For example, the TPTP [26] problem `PUZ035-1` permits a proof consisting of about 40 proof steps in a tableau of depth 6. The Monte Carlo tableau characterisation, however, requires building a Monte Carlo search tree with a depth close to 40, which is challenging even when using a good state reward ρ. The required tree depth can be often decreased with the tableau-specific expansion policies described in Sect. 4.3, but finding a characterisation that reliably reduces the depth of the search tree remains future work.

4 Tableau Heuristics

In Sect. 2.1, we defined two kinds of heuristics to guide Monte Carlo Tree Search, namely transition probability and state reward. In this section, we propose such heuristics, as well as a set of incomplete expansion policies.

4.1 Transition Probability

The transition probability $P(s' \mid s)$ is the probability of choosing state s' as successor state when in state s, where $s' \in \delta(s)$. P is used to bias the selection of a successor state in random simulations, as well as to determine the order of visiting previously unvisited successor states; see Algorithm 1.

When in some state s, different kinds of tableau rules might be applicable; for example α-rules and β-rules (similarly to extension and reduction rules in the connection calculus). In this work, we focus on influencing the probability of

β-rules depending on their used β-formulae, which corresponds to earlier work about choosing good extension clauses in the connection calculus [28]. Therefore, we only vary the probabilities of β-rules and attribute to all non-β-rules the same probabilities.

As transition probabilities are among of the most frequently calculated values in Monte Carlo Tree Search, the speed of this heuristic is important. The baseline heuristic is to give the same probability to all transitions, i.e. $P_1(s' \mid s) \propto 1$.

4.1.1 β-size

The β-size heuristic attributes a probability to a β-rule that is inversely proportional to the number of newly opened β-children:

$$P_\beta(s' \mid s) \propto (|\beta_o(s')| - |\beta_o(s)|)^{-1}.$$

Example 2. In state 1 of Fig. 1, it is possible to apply the β-rule to the leftmost branch with either $\neg p \vee s$ or $\neg p \vee t \vee u$. The first formula consists of two disjuncts and the second of three disjuncts, so the β-size heuristic attributes a probability proportional to $\frac{1}{2}$ to the first and $\frac{1}{3}$ to the second formula. The probabilities are normalized to sum to 1, obtaining the actual values $\frac{3}{5}$ and $\frac{2}{5}$ respectively.

4.1.2 Naive Bayesian Probability

Given the information about the formulae that were used in previous successful proofs at particular proof states, it is possible to calculate the likelihood that a given formula contributes to the current proof attempt in the current proof state. Naive Bayesian probability is used in [13] to order formulae by

$$P(l_i \mid \boldsymbol{f}) = \frac{P(l_i)P(\boldsymbol{f} \mid l_i)}{P(\boldsymbol{f})} \propto P(l_i) \prod_j P(f_j \mid l_i),$$

where l_i is a β-formula from a set $\boldsymbol{l}$ of applicable β-formulae, and $\boldsymbol{f}$ is a set of features that characterises the current tableau, such as its formulae symbols.

$P(l_i)$ and $P(f_j \mid l_i)$ as in [13] frequently yield values such that the probability of applying β-rules is magnitudes smaller than for non-β-rules, slowing down proof search. For that reason, we introduce normed probability estimates.

First, let us denote the knowledge about the usage of β-formulae in previous proofs by $F(l_i)$, which is the multiset of sets of features having occurred in conjunction with l_i when l_i was used in a proof. $|F(l_i)|$ is the total number of times that l_i was used in previous proofs.

Example 3. $F(l_1) = \{\{f_1, f_2\}, \{f_2, f_3\}\}$ means that the formula l_1 was used twice in previous proofs; once in a situation characterised by the features f_1 and f_2, and once when features f_2 and f_3 were present.

This allows us to write the normed formula probability as

$$P(l_i) = \frac{|F(l_i)|}{\max_{l_j \in \boldsymbol{l}} |F(l_j)|}.$$

Using max instead of $\sum$ yields larger probabilities, while still ensuring that the probabilities do not exceed 1.

To obtain the normed conditional feature probability, we distinguish whether the feature already appeared in conjunction with the formula. In case it did, its probability is

$$P(f_j \mid l_i, \exists \boldsymbol{f'} \in F(l_i).f_j \in \boldsymbol{f'}) = \frac{\sum_{\boldsymbol{f'} \in F(l_i)} 1_{\boldsymbol{f'}}(f_j)}{|F(l_i)|},$$

where $1_A(x)$ denotes the indicator function that returns 1 if $x \in A$ and 0 otherwise. In case the feature f_j has never appeared with the rule l_i before, we attribute it some minimal probability with respect to all current features $\boldsymbol{f}$ and all currently applicable rules $\boldsymbol{l}$:

$$P(f_j \mid l_i, \neg\exists \boldsymbol{f'} \in F(l_i).f_j \in \boldsymbol{f'}) = \min_{f_j \in \boldsymbol{f}, l_i \in \boldsymbol{l}, \exists \boldsymbol{f'} \in F(l_i).f_j \in \boldsymbol{f'}} P(f_j \mid l_i)$$

The two definitions form a complete description of the normed feature probability $P(f_j \mid l_i)$.

4.2 State Reward

The state reward $\rho(s)$ is evaluated for the final state s of a random simulation. It estimates the likelihood of finding a proof from any ancestor of the starting node of the random simulation. Therefore, the state reward influences which regions of the Monte Carlo tree are explored.

As the state reward is only calculated once per random simulation, it can in practice be a function that is more expensive to calculate than, say, the transition probability. A baseline state reward function ρ_r returns random values between 0 and 1.

To estimate the *discrimination* of a heuristic, i.e. its ability to distinguish nodes that lead to proofs from nodes that do not, we take the ratio of the average rewards on the Monte Carlo tree branch leading to a proof and the average rewards of all Monte Carlo tree nodes.

4.2.1 β-ratio

The β-ratio reward function considers the ratio of closed β-children and all β-children in the tableau:

$$\rho_\beta(s) = \frac{|\beta_c(s)|}{|\beta(s)|}.$$

This heuristic guarantees that for a closed tableau s, the reward $\rho_\beta(s)$ is 1.

Example 4. For state 4 in Fig. 1, there are five closed β-children and seven β-children in total. Therefore, the reward ρ_β is $\frac{5}{7}$.

4.2.2 Formula Weight Reward

The formula weight reward heuristic calculates the average inverse weight (i.e. formula size) of all open β-children, encouraging tableaux with smaller formulae. Furthermore, the heuristic gives higher impact to formulae closer to the root, because the closer to the root a formula is in the tableau, the more likely it is to be chosen in other random simulations from the same starting node, therefore it is more characteristic for the starting node. For that reason, the heuristic weighs every inverse formula weight with the *depth* of the formula in the tableau, where the depth of a formula f in a tableau is expressed as $d(f)$. However, because rewards need to be normed between 0 and 1, the depth needs to be normalised. For that purpose, we introduce the concept of a *normalisation function*.

Definition 3 (Normalisation function). *A* normalisation function $N_l^u : [0, \infty) \to [u, l)$ *with* $l < u$ *is strictly increasing and fulfils* $\lim_{x\to\infty} N_l^u(x) = u$ *and* $N_l^u(0) = l$.

We choose the normalisation function $N_l^u(x) = u - \left(x + (u - l)^{-1}\right)^{-1}$. This allows us to write the final formula weight function:

$$\rho_w(s) = \frac{1}{|\beta_o(s)|} \sum_{c \in \beta_o(s)} \frac{1}{|c|} N_l^1(d(c)),$$

where $l > 0$ is a constant that determines the impact of formula depth. For example, when $l = 1$, then depth has no influence whatsoever, whereas $l \approx 0$ gives hardly any weight to formulae close to the root. In this particular ρ_w, we use the arithmetic mean, but we have also experimented with geometric and harmonic means as well as the minimum.

Example 5. The open β-children r and t in state 4 of Fig. 1 are at depth 1 and 2, respectively. Therefore, the formula weight reward of the tableau is the mean of $\frac{1}{|r|} N_l^1(1)$ and $\frac{1}{|t|} N_l^1(2)$.

This heuristic is based on similar ideas as the *pick-given ratio* popularised by Otter [23].

4.2.3 Machine-Learnt Refutability Estimate

The *refutability* of a tableau s can be estimated with knowledge how often open β-children of s were successfully refuted in previous proofs.

We call a formula refuted when all branches on which it lies are closed. A formula is unsuccessfully refuted if it is present in the tableau, but lies on at least one open branch. Note that refuted β-children are always closed (as defined in Sect. 3), but closed β-children are not necessarily refuted.

Example 6. The β-child q in state 4 of Fig. 1 is closed, but not refuted.

When statistics about previous refutations of formulae are available, we use them to estimate the refutability of formulae in the current proof search, similarly to [8]. Let $p(f)$ be the number of successful and $n(f)$ the number of unsuccessful refutations of a formula f. Then the irrefutability ratio of f is $\frac{n(f)}{p(f)+n(f)}$.

We want the irrefutability ratio to have an effect proportional to the amount of information available about previous refutation attempts. Consider the case for a formula f where $p(f) = 0$ and $n(f) = 1$. The irrefutability ratio of f then is 100%, but because we have information about only a single refutation attempt, we want to attribute less meaning to it compared to, say, a formula where $p(f) = 0$ and $n(f) = 1000$. To achieve this, we weigh the irrefutability with $N_u^l(v(p(f)+n(f)))$, where $v \geq 0$, $u \geq 0$ and $l \leq 1$ are constants. This term reflects the *confidence* in the irrefutability ratio. v determines how fast we gain confidence, u is the minimal and l is the maximal confidence.

The estimated refutability of the formula f then is the opposite of its confidence-weighted irrefutability:

$$1 - N_u^l(v(p(f)+n(f)))\frac{n(f)}{p(f)+n(f)}$$

The machine-learnt refutability estimate of a whole tableau is the mean of estimated refutabilities of the tableau's open β-children.

Example 7. The open β-children in state 4 of Fig. 1 are t and r. Assume that $p(t) = 222$, $n(t) = 115$, $p(r) = 62$, and $n(r) = 553$. Then the machine-learnt refutability estimate of the tableau is the mean of $1 - N_u^l(v \cdot 337)\frac{115}{337}$ and $1 - N_u^l(v \cdot 615)\frac{553}{615}$. In case we have total confidence in the statistics (e.g. by setting $u = l = 1$) and use the arithmetic mean, the resulting refutability estimate is 0.38.

4.3 β-minimal Expansion Policies

The default expansion policy in Sect. 2.3 creates new nodes in the Monte Carlo tree from the first state s_1 of a random simulation $[s_1, \ldots, s_D]$. This can be counterproductive in cases where the random simulation closes a subtree, but fails to find a proof in the end. In that case, keeping the successful part of the proof attempt, i.e. the closed subtree, can accelerate proof search.

This motivates β-minimal expansion policies, where new nodes are created not from the first state of a simulation, but from some state minimising a function related to β-children.

The first policy is the β-child expansion policy, which chooses the state with fewest open β-children, i.e., $\min_i |\beta_o(s_i)|$.

The second policy is the β-parent expansion policy, which chooses the state with fewest parents of open β-children, i.e. $\min_i \left|\bigcup_{o \in \beta_o(s_i)} p(o)\right|$, where $p(s)$ denotes the parent of a node s.

Similarly to restricted backtracking [19], the β-minimal expansion policies lose completeness, but can in practice perform significantly better than complete strategies.

Example 8. In the proof search in Fig. 1, the proof attempt failed. We assume that the proof search started from a Monte Carlo node n containing state 1. The default expansion policy would add a new node corresponding to state 2 as child tree node of n to the Monte Carlo tree. However, this would discard the closed subtree found in state 3. In contrast, the β-child expansion policy compares the open β-children in all successor states of state 1: State 2 has three open β-children (s, q, and r), state 3 has two (q and r) and state 4 has two as well (t and r). State 3 and 4 are therefore minimal, in which case the first of them (i.e. state 3) is used as state for a new Monte Carlo leaf node that is added as child tree node of n to the Monte Carlo tree.

5 Implementation

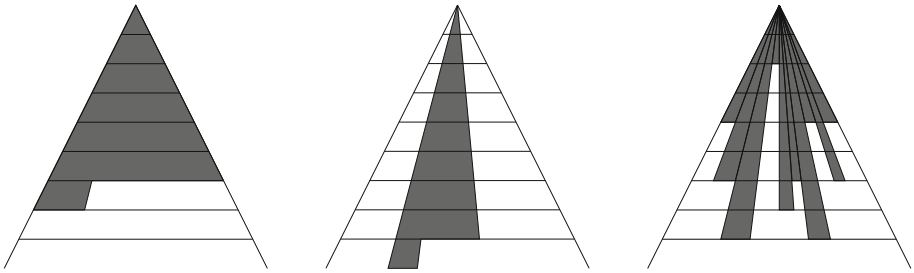
We implemented the proposed Monte Carlo Tableau calculus in the OCaml version [14] of leanCoP [18]. The implementation and experimental data are available at: http://cl-informatik.uibk.ac.at/users/mfaerber/cade-26.html. In the rest of this paper, we refer to the OCaml version of leanCoP as leanCoP.

Monte Carlo proof search can be used to *advise* a *base prover*: The proof search is conducted by a base prover such as leanCoP. When the base prover has a choice between different applicable proof rules, it starts the advisor, i.e. Monte Carlo proof search, which returns after a certain number of iterations an order on the proof rules to be tried by the base prover. This order is based on the average Monte Carlo rewards achieved for each rule. Furthermore, when Monte Carlo proof search finds proofs while establishing the proof rule order, the proofs are used directly by the base prover. In the extreme case, when setting the number of Monte Carlo iterations to ∞, the whole proof search is done by Monte Carlo proof search and the base prover is only responsible for starting it and printing the proof. We refer to our implementation of Monte Carlo proof search as advisor for leanCoP as *Monte Carlo Prover*.

In contrast to leanCoP, Monte Carlo proof search does not require iterative deepening. Instead, an important parameter is the simulation depth D as shown in Sect. 2.3, which determines the length of random simulations.

leanCoP is equipped with a set of strategies, where each strategy consists of a set of options, such as whether to use definitional clausal normal form. A strategy schedule tries different strategies for a defined amount of time until a strategy succeeds. One of the most influential developments in leanCoP was restricted backtracking [19], which discards other possibilities to close a subtree once it has been closed. See Fig. 2 for a comparison of the complete strategy with the restricted backtracking strategy, as well as an illustration of a Monte Carlo search.

In the next section, we evaluate how well our Monte Carlo prover performs in comparison to single leanCoP strategies.



(a) Iterative deepening without restricted backtracking.

(b) Iterative deepening with restricted backtracking.

(c) Monte Carlo.

Fig. 2. The two main leanCoP strategies compared with Monte Carlo proof search.

6 Evaluation

In this section, we evaluate the Monte Carlo prover described in Sect. 5. We first describe the dataset and the evaluation parameters. Then we evaluate the different heuristics given in Sect. 4, as well as the influence of several numeric parameters. Finally, we show our best obtained Monte Carlo configuration and compare it to leanCoP.

Experimental Setup. We used the bushy version of the MPTP2078 dataset [1], which is particularly valuable for our machine learning algorithms as it provides consistent symbols over all problems. To generate training data for the machine learning heuristics, we ran leanCoP for 60 s on all the MPTP2078 problems, using a strategy schedule with three strategies, including a restricted backtracking and a complete strategy. The outcome of the training runs were formula usability data for the Naive Bayes heuristic in Sect. 4.1.2 as well as formula refutability data for the heuristic in Sect. 4.2.3.

For the main evaluation, we used definitional classification and a timeout of 10 s per problem for both leanCoP and the Monte Carlo prover, where the 10 s timeout is also used for the MPTP2078 evaluation in [14]. In that setting, leanCoP solves 509 problems with restricted backtracking and 388 without, the union being 562 problems. In the remainder of this paper, leanCoP refers to the restricted backtracking strategy of leanCoP.

For the Monte Carlo prover, we used the following initial parameters:

- Maximal simulation depth D: 50
- Exploration constant C_p: 1 (see Sect. 2.3)
- Transition probability: β-size (see Sect. 4.1.1)
- State reward: β-ratio (see Sect. 4.2.1)
- Depth attenuation for formula weight reward: 0 (see Sect. 4.2.2)
- Refutability mean: min (see Sect. 4.2.3)
- Refutability confidence velocity: 1 (see Sect. 4.2.3)

– Minimal/maximal refutability confidence: 0/1 (see Sect. 4.2.3)
– Expansion policy: β-child expansion policy (see Sect. 4.3)

Heuristics Influence. We evaluated the Monte Carlo prover with a set of configurations where each configuration deviates by one heuristic from the initial parameters. For every configuration, we collected the set of solved problems. Furthermore, we collected the problems solved by all Monte Carlo configurations, amounting to 196 problems. On these problems, for all Monte Carlo configurations, we evaluated the average number of MCTS iterations and MCTS simulation steps, as well as the average reward discrimination; see Table 1.

The machine-learnt reward heuristic performs best, with a very good discrimination rate of 2.30. Surprisingly, the random reward heuristic solves only three problems less, despite its worse discrimination.

The Bayesian transition probability shows very poor performance. The β-size heuristic is the winner for transition probability.

The β-parent expansion policy outperforms the default expansion policy by 20 problems, i.e. 6%.

Table 1. Comparison of Monte Carlo heuristics. Iterations, simulation steps and discrimination ratio are averages on the 196 problems solved by all configurations.

Configuration	Iterations	Sim. steps	Discr.	Solved
Base	116.46	1389.82	1.37	332
Random reward	104.88	1167.98	1.19	364
Formula weight reward	108.13	1268.88	1.12	334
ML reward	108.52	1151.61	**2.30**	**367**
Bayes P	528.39	8014.03	1.35	248
Constant P	949.62	17539.59	1.31	237
β-parent exp.	224.72	2769.12	1.40	348
Default exp.	371.81	4793.58	1.38	328

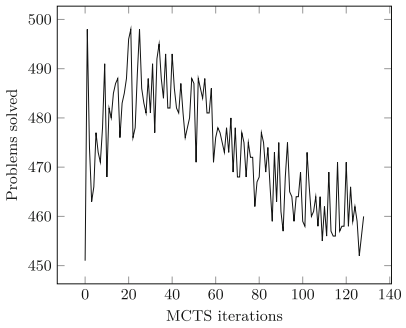
Parameter Influence. We identified three numeric parameters to be highly influential for proof search; namely the simulation depth D, the exploration constant C_p, and the maximal number of MCTS iterations per base prover step. We evaluated a large range of values for these parameters, keeping the remaining parameters fixed to the standard values. The results are shown in Fig. 3.

We achieve the highest performance of the Monte Carlo prover when using it as an advisor for a base prover. From Fig. 3a, it becomes clear that the Monte Carlo prover is most useful when given between 20 and 40 iterations per base prover step. Below that mark, the reward estimates are too imprecise, and above that mark, the reward precision increases only marginally, compared to the time spent in the MCTS prover.

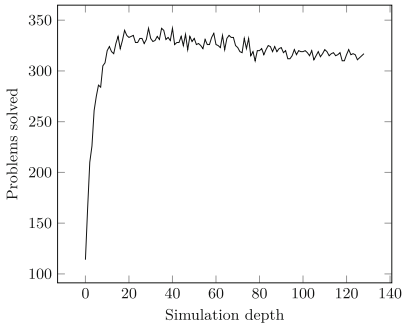
The higher the maximal simulation depth D (see Fig. 3b), the more time the prover spends looking for proofs at less promising higher depths. Figure 3c shows

that the average number of simulation steps decreases with increasing D. This indicates that at higher simulation depths, the computational effort to calculate the set of possible steps increases.

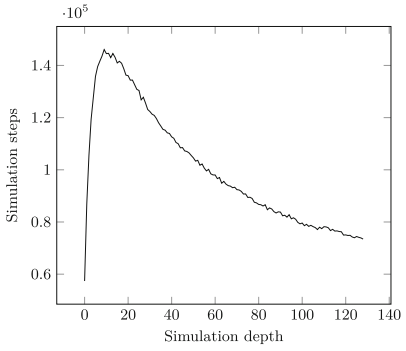
Figure 3d shows the number of solved problems for the β-ratio and the machine-learnt state evaluation heuristics as function of the exploration constant C_p. For a good state reward heuristic, one expects in such a graph a local optimum, where exploration and exploitation combine each other best. As one can see, this is given for the machine-learnt heuristic at $C_p \approx 0.75$, whereas the curve for the β-ratio heuristic does not expose such an optimum.



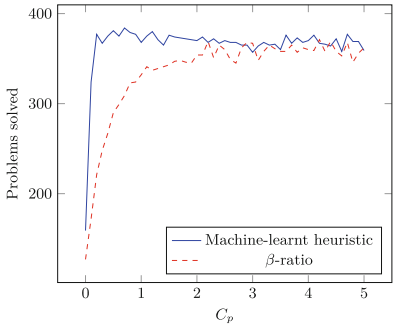
(a) Maximal number of MCTS iterations.



(b) Simulation depth D.



(c) Simulation steps / Simulation depth.



(d) Exploration constant C_p.

Fig. 3. Parameter influence.

Best Found Monte Carlo Configuration. Our best found configuration MC$^+$ for the Monte Carlo prover uses the arithmetic mean for the ML reward, a maximal number of 27 MCTS iterations and a simulation depth of 20. Interestingly, is has a discrimination ratio of only 1.07, which suggests that a high discrimination ratio indicates good performance, but is not absolutely necessary to achieve it.

MC$^+$ performs on average 902 times more inferences in MCTS than in the base prover. Furthermore, for the problems solved both by leanCoP and by MC$^+$, leanCoP takes on average 21698 inferences, while MC$^+$ takes 20243 inferences (sum of base prover + MCTS inferences).

MC$^+$ solves 538 problems, compared to 509 by leanCoP. Of the 538 problems, 90 problems were previously not solved by leanCoP. The union of MC$^+$ and leanCoP solves 599 problems, compared to 531 problems solved by leanCoP with a timeout of 20 s. That means that we solve 12.8% more problems. Furthermore, MC$^+$ proves more problems than leanCoP when given only half the time.

Prover	Timeout [s]	Solved problems
leanCoP	10	509
MC$^+$	10	538
leanCoP + MC$^+$	10 + 10	599
leanCoP	20	531

7 Conclusion

We have proposed a combination of Monte Carlo Tree Search and tableau automated theorem proving. MCTS provides a theoretically founded fine-grained mechanism to control the search space of tableau-based theorem provers based on random sampling and state evaluation heuristics, which might eventually even replace iterative deepening. We have shown that a fast rollout policy combined with a machine-learnt state evaluation heuristic and a custom expansion policy produce the best results. The strength of the current system has turned out to be its function as advisor for existing provers, demonstrated by our integration into leanCoP. This opens a wide space of future work, profiting from the ongoing research in MCTS; examples include self-updating reward heuristics, adaptive simulation depths, automatic parameter tuning, and different characterisations of tableau search or expansion policies such as AMAF to produce more shallow Monte Carlo trees. Furthermore, identifying controversial choices in the base prover would allow using the Monte Carlo prover as advisor more efficiently. Finally, neural networks could be used as state reward heuristics.

Acknowledgements. We thank the anonymous CPP and CADE referees for their valuable comments on previous versions of this paper. This work has been supported by the Austrian Science Fund (FWF) grant P26201 and the European Research Council (ERC) grants no. 649043 *AI4REASON* and no. 714034 *SMART*.

References

1. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., Urban, J.: Premise selection for mathematics by corpus analysis and kernel methods. J. Autom. Reasoning **52**(2), 191–213 (2014)
2. Alama, J., Kühlwein, D., Urban, J.: Automated and human proofs in general mathematics: an initial comparison. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 37–45. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28717-6_6
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_14
4. Biere, A., Dragan, I., Kovács, L., Voronkov, A.: Experimenting with SAT solvers in Vampire. In: Gelbukh, A., Espinoza, F.C., Galicia-Haro, S.N. (eds.) MICAI 2014. LNCS, vol. 8856, pp. 431–442. Springer, Cham (2014). doi:10.1007/978-3-319-13647-9_39
5. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formaliz. Reasoning **9**(1), 101–148 (2016)
6. Brown, C.E.: Satallax: an automatic higher-order prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 111–117. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31365-3_11
7. Browne, C., Powley, E.J., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Liebana, D.P., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **4**(1), 1–43 (2012)
8. Färber, M., Brown, C.E.: Internal guidance for Satallax. In: Olivetti and Tiwari [17], pp. 349–361
9. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Ghahramani, Z. (ed.) ICML, vol. 227, pp. 273–280. ACM, New York (2007)
10. Hähnle, R.: Tableaux and related methods. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 2, pp. 100–178. Elsevier and MIT Press, New York (2001)
11. Hoder, K., Reger, G., Suda, M., Voronkov, A.: Selecting the selection. In: Olivetti and Tiwari [17], pp. 313–329
12. Kaliszyk, C., Schulz, S., Urban, J., Vyskočil, J.: System description: E.T. 0.1. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 389–398. Springer, Cham (2015). doi:10.1007/978-3-319-21401-6_27
13. Kaliszyk, C., Urban, J.: FEMaLeCoP: fairly efficient machine learning connection prover. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 88–96. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48899-7_7
14. Kaliszyk, C., Urban, J., Vyskocil, J.: Certified connection tableaux proofs for HOL Light and TPTP. In: Leroy, X., Tiu, A. (eds.) CPP, pp. 59–66. ACM, New York (2015)
15. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006). doi:10.1007/11871842_29
16. Korovin, K.: Inst-Gen – a modular approach to instantiation-based automated reasoning. In: Voronkov, A., Weidenbach, C. (eds.) Ganzinger Festschrift. LNCS, vol. 7797, pp. 239–270. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37651-1_10

17. Olivetti, N., Tiwari, A. (eds.): IJCAR 2016. LNCS (LNAI), vol. 9706. Springer, Cham (2016). doi:10.1007/978-3-319-40229-1
18. Otten, J.: leanCoP 2.0 and ileanCoP 1.2: high performance lean theorem proving in classical and intuitionistic logic (system descriptions). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 283–291. Springer, Heidelberg (2008). doi:10.1007/978-3-540-71070-7_23
19. Otten, J.: Restricting backtracking in connection calculi. AI Commun. **23**(2–3), 159–182 (2010)
20. Otten, J.: nanoCoP: a non-clausal connection prover. In: Olivetti and Tiwari [17], pp. 302–312
21. Rosin, C.D.: Nested rollout policy adaptation for Monte Carlo tree search. In: Walsh, T. (ed.) IJCAI, pp. 649–654. IJCAI/AAAI, New York (2011)
22. Schadd, M.P.D., Winands, M.H.M., Tak, M.J.W., Uiterwijk, J.W.H.M.: Single-player Monte-Carlo tree search for SameGame. Knowl.-Based Syst. **34**, 3–11 (2012)
23. Schulz, S.: E - a brainiac theorem prover. AI Commun. **15**(2–3), 111–126 (2002)
24. Schulz, S.: System description: E 1.8. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 735–743. Springer, Heidelberg (2013). doi:10.1007/978-3-642-45221-5_49
25. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of Go with deep neural networks and tree search. Nature **529**, 484–503 (2016)
26. Sutcliffe, G.: The 6th IJCAR automated theorem proving system competition - CASC-J6. AI Commun. **26**(2), 211–223 (2013)
27. Urban, J., Hoder, K., Voronkov, A.: Evaluation of automated theorem proving on the Mizar Mathematical Library. In: Fukuda, K., van der Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 155–166. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15582-6_30
28. Urban, J., Vyskočil, J., Štěpánek, P.: MaLeCoP machine learning connection prover. In: Brünnler, K., Metcalfe, G. (eds.) TABLEAUX 2011. LNCS (LNAI), vol. 6793, pp. 263–277. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22119-4_21
29. Wisniewski, M., Steen, A., Benzmüller, C.: LEOPARD — a generic platform for the implementation of higher-order reasoners. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) CICM 2015. LNCS, vol. 9150, pp. 325–330. Springer, Cham (2015). doi:10.1007/978-3-319-20615-8_22