Prolog Technology Reinforcement Learning Prover (System Description)

Zsolt Zombori[1,2(✉)], Josef Urban[3], and Chad E. Brown[3]

[1] Alfréd Rényi Institute of Mathematics, Budapest, Hungary
zombori@renyi.hu
[2] Eötvös Loránd University, Budapest, Hungary
[3] Czech Technical University of Prague, Prague, Czechia

Abstract. We present a reinforcement learning toolkit for experiments with guiding automated theorem proving in the connection calculus. The core of the toolkit is a compact and easy to extend Prolog-based automated theorem prover called plCoP. plCoP builds on the leanCoP Prolog implementation and adds learning-guided Monte-Carlo Tree Search as done in the rlCoP system. Other components include a Python interface to plCoP and machine learners, and an external proof checker that verifies the validity of plCoP proofs. The toolkit is evaluated on two benchmarks and we demonstrate its extendability by two additions: (1) guidance is extended to reduction steps and (2) the standard leanCoP calculus is extended with rewrite steps and their learned guidance. We argue that the Prolog setting is suitable for combining statistical and symbolic learning methods. The complete toolkit is publicly released.

Keywords: Automated theorem proving · Reinforcement learning · Logic programming · Connection tableau calculus

1 Introduction

Reinforcement learning (RL) [36] is an area of Machine Learning (ML) that has been responsible for some of the largest recent AI breakthroughs [3,32–34]. RL develops methods that advise agents to choose from multiple actions in an environment with a delayed reward. This fits many settings in Automated Theorem Proving (ATP), where many inferences are often possible in a particular search state, but their relevance only becomes clear when a proof is found.

Several learning-guided ATP systems have been developed that interleave proving with supervised learning from proof searches [4,10–13,17,19,23,39]. In

ZZ was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002) and the Hungarian National Excellence Grant 2018-1.2.1-NKP-00008. JU and CB were funded by the *AI4REASON* ERC Consolidator grant nr. 649043, the Czech project AI&Reasoning CZ.02.1.01/0.0/0.0/15_003/0000466 and the European Regional Development Fund.

N. Peltier and V. Sofronie-Stokkermans (Eds.): IJCAR 2020, LNAI 12167, pp. 489–507, 2020.
https://doi.org/10.1007/978-3-030-51054-1_33

the saturation-style setting used by ATP systems like E [31] and Vampire [21], direct learning-based selection of the most promising given clauses leads already to large improvements [14], without other changes to the proof search procedure.

The situation is different in the connection tableau [22] setting, where choices of actions rarely commute, and backtracking is very common. This setting resembles games like Go, where Monte-Carlo tree search [20] with reinforcement learning used for action selection (*policy*) and state evaluation (*value*) has recently achieved superhuman performance. First experiments with the rlCoP system in this setting have been encouraging [19], achieving more than 40% improvement on a test set after training on a large corpus of Mizar problems.

The connection tableau setting is attractive also because of its simplicity, leading to very compact Prolog implementations such as leanCoP [29]. Such implementations are easy to modify and extend in various ways [27,28]. This is particularly interesting for machine learning research over reasoning corpora, where automated learning and addition of new prover actions (tactics, inferences, symbolic decision procedures) based on previous proof traces seems to be a large upcoming topic. Finally, the proofs obtained in this setting are easy to verify, which is important whenever automated self-improvement is involved.

The goal of the work described here is to develop a reinforcement learning toolkit for experiments with guiding automated theorem proving in the connection calculus. The core of the toolkit (Sect. 2) is a compact and easy to extend Prolog-based automated theorem prover called plCoP. plCoP builds on the leanCoP Prolog implementation and adds learning-guided Monte-Carlo Tree Search as done in the rlCoP [19] system. Other components include a Python interface to plCoP and state-of-the-art machine learners and an external proof checker that verifies the validity of the plCoP proofs. The proof checker has proven useful in discovering bugs during development. Furthermore, it is our long term goal to add new prover actions automatically, where proof checking becomes essential.

Prolog is traditionally associated with ATP research, and it has been used for a number of Prolog provers [5,24,29,35], as well as for rapid ATP prototyping, with core methods like unification for free. Also, Prolog is the basis for Inductive Logic Programming (ILP) [25] style systems and a natural choice for combining such symbolic learning methods with machine learning for ATP systems, which we are currently working on [41]. In more detail, the main contributions are:

1. We provide an open-source Prolog implementation of rlCoP, called plCoP, that uses the SWI-Prolog [40] environment.
2. We extend the guidance of leanCoP to reduction steps involving unification.
3. We extend leanCoP with rewrite steps while keeping the original equality axioms. This demonstrates the benefit of adding a useful but redundant inference rule, with its use controlled by the learned guidance.
4. We provide an external proof checker that certifies the validity of the proofs.
5. The policy model of rlCoP is trained using Monte Carlo search trees of all proof attempts. We show, however, that this introduces a lot of noise, and we get significant improvement by limiting policy training data to successful theorem proving attempts.

6. Policy and value models are trained in rlCoP using a subset of the Monte Carlo search nodes, called *bigstep nodes*. However, when a proof is found, not all nodes leading to the proof are necessarily bigstep nodes. We make training more efficient by explicitly ensuring that all nodes leading to proofs are included in the training dataset.
7. The system is evaluated in several iterations on two MPTP-based [37] benchmarks, showing large performance increases thanks to learning (Sect. 3). We also improve upon rlCoP with 12% and 7% on these benchmarks.

2 Prolog Technology Reinforcement Learning Prover

The toolkit is available at our repository.[1] Its core is our plCoP connection prover based on the leanCoP implementation and inspired by rlCoP. leanCoP [29] is a compact theorem prover for first-order logic, implementing connection tableau search. The proof search starts with a *start clause* as a *goal* and proceeds by building a connection tableau by applying *extension steps* and *reduction steps*. leanCoP uses iterative deepening to ensure completeness. This is removed in rlCoP and plCoP and learning-guided Monte-Carlo Tree Search (MCTS) [8] is used instead. Below, we explain the main ideas and parts of the system.

Monte Carlo Tree Search (MCTS) is a search algorithm for sequential decision processes. MCTS builds a tree whose nodes are states of the process, and edges represent sequential decisions. Each state (node) yields some reward. The aim of the search algorithm is to find trajectories (branches in the search tree) that yield high accumulated reward. The search starts from a single root node (*starting state*), and new nodes are added iteratively. In each node i, we maintain the number of visits n_i, the total reward r_i, and its prior probability p_i given by a learned *policy* function. Each iteration, also called *playout*, starts with the addition of a new leaf node. This is done by recursively selecting a child that maximizes the standard UCT [20] formula (1), until a leaf is reached. In (1), N is the number of visits of the parent, and cp is a parameter that determines the balance between nodes with high value (exploitation) and rarely visited nodes (exploration). Each leaf is given an initial value, which is typically provided by a learned *value* function. Next, ancestors are updated: visit counts are increased by 1 and value estimates are increased by the value of the new node. The value and policy functions are learned in AlphaGo/Zero, rlCoP and plCoP.

$$\mathrm{UCT}(i) = \frac{r_i}{n_i} + cp \cdot p_i \cdot \sqrt{\frac{lnN}{n_i}} \tag{1}$$

MCTS for Connection Tableau: Both rlCoP and plCoP use the DAgger [30] meta-learning algorithm to learn the policy and value functions. DAgger interleaves ATP runs based on the current policy and value (*data collection phase*) with a *training phase*, in which these functions are updated to fit the collected

[1] https://github.com/zsoltzombori/plcop.

data. Such iterative interleaving of proving and learning has also been used successfully in ATP systems such as MaLARea [38] and ENIGMA [14]. During the proof search plCoP builds a Monte Carlo tree for each training problem. Its nodes are the proof states (partial tableaux), and edges represent inferences. A branch leading to a node with a closed tableau is a valid proof. Initially, plCoP uses simple heuristic value and policy functions, later to be replaced with learned guidance. To enforce deeper exploration, rlCoP and plCoP perform a *bigstep* after a fixed number of playouts: the starting node of exploration is moved one level down towards the child with the highest value (called *bigstep node*). Later MCTS steps thus only extend the subtree under the bigstep node.

Training data for policy and value learning is extracted from the tableau states of the bigstep nodes. The value model gets features from the current goal and path, while the policy model also receives features of the given action.

We use term walks of length up to 3 as main features. Both rlCoP and plCoP add also several more specific features.[2] The resulting sparse feature vectors are compressed to a fixed size (see Appendix E). For learning value, each bigstep node is assigned a label of 1[3] if it leads to a proof and 0 otherwise. The policy model gets a target probability for each edge based on the relative frequency of the corresponding child. Both rlCoP and plCoP use gradient boosted trees (XGBoost [9]) for guidance. Training concludes one iteration of the DAgger method. See Appendix D for more details about policy and value functions.

Prolog Implementation of plCoP: To implement MCTS, we modify leanCoP so that the Prolog stack is explicitly maintained and saved in the Prolog database using assertions after each inference step. This is done in the `leancop_step.pl` submodule, described in Appendix C. This setup makes it possible to interrupt proof search and later continue at any previously visited state, required to interleave prover steps with Monte Carlo tree steps, also implemented in Prolog. The main MCTS code is explained in Appendix B. The MCTS search tree is stored in destructive Prolog hashtables.[4] These are necessary for efficient updates of the nodes statistics. The training data after each proof run is exported from the MCTS trees and saved for the XGBoost learning done in Python.

To guide the search, the trained XGBoost policy and value functions are accessed efficiently via the C foreign language interface of SWI-Prolog. This is done in 70 lines of C++ code, using the SWI C++ templates and the XGBoost C++ API. The main foreign predicate `xgb:predict` takes an XGBoost predictor and a feature list and returns the predicted value. A trained model performs 1000000 predictions in 19 s in SWI. To quantify the total slowdown due to the guidance, we ran plCoP with 200000 inference step limit and a large time limit (1000 s) on the M2k dataset (see Sect. 3) with and without guidance. The average execution time ratio on problems unsolved in both cases is 2.88, i.e., the XGBoost guidance roughly triples the execution time. Efficient feature collection in plCoP

[2] Number of open goals, number of symbols in them, their maximum size and depth, length of the current path, and two most frequent symbols in open goals.

[3] A discount factor of 0.99 is applied to positive rewards to favor shorter proofs.

[4] The `hashtbl` library of SWI by G. Barany – https://github.com/gergo-/hashtbl.

is implemented using declarative association lists (the `assoc` SWI library) implemented as AVL trees. Insertion, change, and retrieval is $O(log(N))$. We also use destructive hashtables for caching the features already computed. Compared to rlCoP, we can rely on SWI's fast internal hash function `term_hash/4` that only considers terms to a specified depth. This all contributes to plCoP's smaller size.

Guiding Reduction Steps: The connection tableau calculus has two steps: 1) extension that replaces the current goal with a new set of goals and 2) reduction that closes off the goal by unifying it with a literal on the path. rlCoP applies reduction steps eagerly, which can be harmful by triggering some unwanted unification. Instead, plCoP lets the guidance system learn when to apply reduction. Suppose the current goal is G. An input clause $\{H, B\}$, s.t. H is a literal that unifies with $\neg G$ and B is a clause, yields an extension step, represented as $ext(H, B)$, while a literal P on the path that unifies with $\neg G$ yields a reduction step, represented as $red(P)$. The symbols red and ext are then part of the standard feature representation.

Limited Policy Training: rlCoP extracts policy training data from the child visit frequencies of the bigstep nodes. We argue, however, that node visit frequencies may not be useful when no proof was found, i.e., when no real reward was observed. A frequently selected action that did not lead to proof should not be reinforced. Hence, plCoP only extracts policy training data when a proof was found. Note that the same is not true for value data. If MCTS was not successful, then bigstep nodes are given a value of 0, which encourages exploring elsewhere.

Training from all Proofsteps: Policy and value models are trained in rlCoP using bigstep nodes. However, when a proof is found, not all nodes leading to the proof are necessarily bigstep nodes. We make training more efficient by explicitly ensuring that all nodes leading to proofs are included in the training dataset.

(Conditional) Rewrite Steps: plCoP extends leanCoP with rewrite steps that can handle equality predicates more efficiently. Let $t|_p$ denote the subterm of t at position p and $t[u]_p$ denote the term obtained after replacing in t at position p by term u. Given a goal G and an input clause $\{X = Y, B\}$, s.t. for some position p there is a substitution σ such that $G|_p\sigma = X\sigma$, the rewrite step changes G to $\{G[Y]_p\sigma, \neg B\sigma\}$. Rewriting is allowed in both directions, i.e., the roles of X and Y can be switched.[5] This is a valid and well-known inference step, which can make proofs much shorter. On the other hand, rewriting can be simulated by a sequence of extension steps. We add rewriting without removing the original congruence axioms, making the calculus redundant. We find, however, that the increased branching in the search space is compensated by learning since we only explore branches that are deemed "reasonable" by the guidance.

Proof Checking: After plCoP generates a proof, the `leancheck` program included in the toolkit does an independent verification. Proofs are first

[5] The rewrite step could probably be made more powerful by ordering equalities via a term ordering. However, we wanted to use as little human heuristics as possible and let the guidance figure out how to use the rewrite steps.

translated into the standard leanCoP proof format. In case of rewriting steps, plCoP references the relevant input clause (with an equational literal), its substitution, the goal before and after rewriting, the equation used and the side literals of the instantiated input clause. Using this information, `leancheck` replaces the rewriting step with finitely many instances of equational axioms (reflexivity, symmetry, transitivity, and congruence) and proceeds as if there were no rewriting steps.

The converted output from plCoP includes the problem's input clauses and a list of clauses that contributed to the proof. The proof clauses are either input clauses, their instances, or extension step clauses. Proof clauses may have remaining uninstantiated (existential) variables. However, for proof checking, we can consider these to be new constants, and so we consider each proof clause to be ground. To confirm we have a proof, it suffices to verify two assertions:

1. The proof clause alleged to be an input clause, or its instance is subsumed by the corresponding input clause (as identified in the proof by a label).
2. The set of such proof clauses forms a propositionally unsatisfiable set.

Each proof clause alleged to be an instance of an input clause is reported as a clause B, a substitution θ and a reference to an input clause C. Optimally, the checker should verify that each literal in $\theta(C)$ is in B, so that $\theta(C)$ propositionally subsumes B. In many cases this is what the checker verifies. However, Prolog may rename variables so that the domain of θ no longer corresponds to the free variables of C. In this case, the checker computes a renaming ρ such that $\theta(\rho(C))$ propositionally subsumes B.[6] We could alternatively use first-order matching to check if C subsumes B. This would guarantee a correct proof exists, although it would accept proofs for which the reported θ gives incorrect information about the intended instantiation. For the second property, we verify the propositional unsatisfiability of this ground clause set using PicoSat [7]. While plCoP is proving a theorem given in disjunctive normal form PicoSat is refuting a set of clauses. Hence we swap polarities of literals when translating clauses to PicoSat. [7] An example is given in Appendix A.

3 Evaluation

We use two datasets for evaluation. The first is the *M2k* benchmark that was introduced in [19]. The M2K dataset is a selection of 2003 problems [15] from the larger Mizar40 dataset [16], which consists of 32524 problems from the Mizar Mathematical Library that have been proven by several state-of-the-art ATPs used with many strategies and high time limits in the Mizar40 experiments [18]. Based on the proofs, the axioms were ATP-minimized, i.e., only those axioms were kept that were needed in any of the ATP proofs found. This dataset is by

[6] Note that this ρ may not be unique. Consider $C = \{p(X), p(Y)\}$, $B = \{p(c), p(c)\}$ and $\theta = W \mapsto c, Z \mapsto c$.

[7] Technically swapping the polarities is not necessary since unsatisfiability is invariant under such a swap. However, there is no extra cost since the choice of polarity is made when translating from the plCoP proof to an input for PicoSat.

construction biased towards saturation-style ATP systems. To have an unbiased comparison with state-of-the-art saturation-style ATP systems such as E, we also evaluate the systems on the bushy (small) problems from the MPTP2078 benchmark [1], which contains just an article-based selection of Mizar problems, regardless of their solvability by a particular ATP system.

We report the number of proofs found using a 200000 step inference limit. Hyperparameters (described in Appendix E) were selected to be consistent with those of rlCoP. A lot of effort has already been invested in tuning rlCoP, furthermore, we wanted to make sure that the effects of our most important additions are not obfuscated by hyperparameter changes. As Tables 1 and 2 show, our baseline is weaker, due to several subtleties of rlCoP that are not reproduced. Nevertheless, since our main focus is to make learning more efficient, improvement with respect to the baseline can be used to evaluate the new features.

M2k Experiments: We first evaluate the features introduced in Sect. 2 on the M2k dataset. Table 1 shows that both limited policy training and training from all proofsteps yield significant performance increase: together, they improve upon the baseline with 31% and upon rlCoP with 3%. However, guided reduction does not help. We found that the proofs in this dataset tend to only use reduction on ground goals, i.e., that does not involve unification, which indeed can be applied eagerly. The rewrite step yields a 9% increase. Improved training and rewriting together improve upon the baseline by 42% and upon rlCoP by 12%. Overall, thanks to the changes in training data collection, plCoP shows greater improvement during training and finds more proofs than rlCoP, even without rewriting. Note that rlCoP has been developed and tuned on M2k. Adding ten more iterations to the best performing (**combined**) version of plCoP results in 1450 problems solved, which is 17.4% better than rlCoP in 20 iterations (1235).

Table 1. Performance on the M2k dataset: original **rlCoP**, plCoP (**baseline**), plCoP with **guided reduction**, plCoP with **limited policy** training, plCoP trained using **all proofsteps**, plCoP using the previous two **improved training** plCoP with **rewriting** and plCoP with rewriting and improved training **combined**. **incr** shows the performance increase in percentages from iteration 0 (unguided) to the best result.

Iteration	0	1	2	3	4	5	6	7	8	9	10	Incr
rlCoP	770	1037	1110	1166	1179	1182	1198	1196	1193	**1212**	1210	57%
Baseline	632	852	860	915	918	944	949	**959**	955	943	954	52%
Guided reduction	616	840	884	905	915	900	914	924	**942**	915	912	53%
Limited policy	632	988	1037	1071	1080	1094	1092	1101	1103	**1118**	1111	77%
All proofsteps	632	848	930	988	986	1018	1033	1039	**1053**	1043	1050	67%
Improved training	632	975	1100	1154	1180	1189	1209	1231	1238	1243	**1254**	**98%**
Rewriting	695	913	989	995	1003	1019	1030	1030	1033	1038	**1045**	50%
Combined	695	1070	1209	1253	1295	1309	1322	1335	1339	1346	**1359**	96%

MPTP2078: Using a limit of 200000 inferences, unmodified leanCoP solves 612 of the MPTP2078 bushy problems, while its OCaml version (mlcop), used as a basis for rlCoP solves 502. E solves 998, 505, 326, 319 in *auto*, *noauto*, *restrict*, *noorder* modes[8] plCoP and rlCoP results are summarized in Table 2. Improved training and rewriting together yield 63% improvement upon the baseline and 7% improvement upon rlCoP. Here, it is plCoP that starts better, while rlCoP shows stronger learning. Eventually, plCoP outperforms rlCoP, even without rewriting. Additional ten iterations of the **combined** version increase the performance to 854 problems. This is 12% more than rlCoP in 20 iterations (763) but still weaker than the strongest E configuration (998). However it performs better than E with the more limited heuristics, as well as leanCoP with its heuristics.

Table 2. Performance on the MPTP2078 bushy dataset: original **rlCoP**, **baseline** plCoP, plCoP using **improved training** and **combined** plCoP. **incr** shows the performance increase in percentages from iteration 0 (unguided) to the best result.

Iteration	0	1	2	3	4	5	6	7	8	9	10	Incr
rlCoP	198	300	489	605	668	701	720	737	736	732	**733**	**270%**
Baseline	287	363	413	420	429	441	454	464	465	**479**	469	67%
Improved training	287	449	544	611	640	674	692	704	720	731	**744**	140%
Combined	326	460	563	642	671	694	721	740	761	775	**782**	140%

4 Conclusion and Future Work

We have developed a reinforcement learning toolkit for experiments with guiding automated theorem proving in the connection calculus. Its core is the Prolog-based plCoP obtained by extending leanCoP with a number of features motivated by rlCoP. New features on top of rlCoP include guidance of reduction steps, the addition of the rewrite inference rule and its guidance, external proof checker, and improvements to the training data selection. Altogether, plCoP improves upon rlCoP on the M2K and the MPTP2078 datasets by 12% and 7% in ten iterations, and by 17.4% and 12% in twenty iterations. The system is publicly available to the ML and AI/TP communities for experiments and extensions.

One lesson learned is that due to the sparse rewards in theorem proving, care is needed when extracting training data from the prover's search traces. Another lesson is that new sound inference rules can be safely added to the underlying calculus. Thanks to the guidance, the system still learns to focus on promising actions, while the new actions may yield shorter search and proofs for some problems. An important part of such an extendability scheme is the independent proof checker provided.

[8] For a description of these E configurations, see Table 9 of [19].

Future work includes, e.g., the addition of neural learners, such as tree and graph neural networks [10,26]. An important motivation for choosing Prolog is our plan to employ Inductive Logic Programming to learn new prover actions (as Prolog programs) from the plCoP proof traces. As the manual addition of the rewrite step already shows, such new actions can be inserted into the proof search, and guidance can again be trained to use them efficiently. Future work, therefore, involves AI/TP research in combining statistical and symbolic learning in this framework, with the goal of automatically learning more and more complex actions similar to tactics in interactive theorem provers. We believe this may become a very interesting AI/TP research topic facilitated by the toolkit.

A Proof Checking Example

As a simple example suppose we are proving a proposition $q(a)$ under two assumptions $\forall x.p(x)$ and $\forall x.p(x) \Rightarrow q(a)$. plCoP will start with three input "clauses" $p(X)^-$, $p(Y) \vee q(a)^-$ and $q(a)$. The connection proof proceeds in the obvious way and yields three proof clauses that are either input clauses or instances of input clauses: $p(X)^-$, $p(X) \vee q(a)^-$ and $q(a)$. Unification during the search makes the two variables X and Y the same variable X. For proof checking, we now consider X to be constant (in the same sense as a). Switching from the point of view of proving a disjunction of conjuncts to the point of view of refuting a conjunction of disjuncts, we see these as three propositional clauses: P, $P^- \vee Q$ and Q^- where P stands for the atom $p(X)$ (now viewed as ground) and Q stands for $q(a)$ (also ground). The set $\{P, P^- \vee Q, Q^-\}$ is clearly unsatisfiable. In terms of the connection method, the unsatisfiability of this set guarantees every path [2,6] has a pair of complementary literals.

B Monte Carlo Tree Search in Prolog

We show the most important predicates that perform the MCTS. The code has been simplified for readability.

We repeatedly perform playouts, which consist of three steps: 1) find the next tree node to expand, 2) add a new child to this node and 3) update ancestor values and visit counts:

```
mc_playout(ChildHash,ParentHash,NodeHash,FHash):-

    % get the current bigstep node (root of exploration)
    rootnode(StartId), !,

    % find node to expand
    mc_find_unexpanded(StartId,ChildHash,NodeHash,
                       ExpandId,UnexpandedActionIds),
    nb_hashtbl_get(NodeHash,ExpandId,[State,_,_,_,ChProbs]),
```

```
    State=state(_,_,_,_,_,_,Result),
    (  Result ==  1 -> Reward = 1 % we found a proof
     ; Result == -1 -> Reward = 0 % proof failed
    ;  get_largest_index(UnexpandedActionIds, ChProbs,
                         ActionIndex),
       flag(inference_cnt, X, X+1), % increase inference count

       % we expand the child with the largest prior probability
       mc_expand_node(ExpandId,ChildHash,ParentHash,NodeHash,
                      FHash,ActionIndex,Reward)
    ),

    % update ancestor visit counts and values
    mc_backpropagate(ExpandId,Reward,ParentHash,NodeHash).
```

We search for the node to expand based on the standard UCT formula:

```
% +Id: current node id
% -Id2: next node id to expand
mc_find_unexpanded(Id,ChildHash,NodeHash,
                   Id2,UnexpandedActionIds):-
    mc_child_list(Id,NodeHash,ChildHash,ChildPairs),
    nb_hashtbl_get(NodeHash,Id,[State,_,VisitCount,_,_]),
    action_count(State,ActionCount),
    length(ChildPairs,L),
    (  ActionCount == 0 -> % no valid moves
       Id2=Id, UnexpandedActionIds=[]
    ;  mc_ucb_select_child(VisitCount,ChildPairs,NodeHash,
                           SelectedId,UCBScore),
       ( L < ActionCount,
         mc_ucb_score_unexplored(VisitCount,ActionCount,
                                 UCBUnexploredScore),
         UCBUnexploredScore > UCBScore -> %select current node
         Id2=Id,
         missing_actions(ActionCount,ChildPairs,
                         UnexpandedActionIds)
       ;
         % we move towards the child with the highest UCB score
         mc_find_unexpanded(SelectedId,ChildHash,NodeHash,
                            Id2,UnexpandedActionIds)
       ), !
    ;  % The current node is a leaf, so we select it
       Id2=Id,
       missing_actions(ActionCount,ChildPairs,
                       UnexpandedActionIds)
    ).
```

Once the node to expand has been selected, we pick the unexplored child that has the highest UCB score:

```
mc_expand_node(ParentId,ChildHash,ParentHash,NodeHash,FHash,
               ActionIndex,ChildValue):-
    nb_hashtbl_get(NodeHash,ParentId,[ParentState,_,_,_,ChProbs]),

    % we perform the inference step corresponding to the new child
    copy_term(ParentState,ParentState2),
    logic_step(ParentState2,ActionIndex,ChildState),

    % value estimate from the external (xgboost) model
    guidance_get_value(ChildState, FHash, ChildValue),

    % probability estimates for the children of the new node
    % from the external (xgboost) model
    guidance_action_probs(ChildState,FHash,ChChProbs),

    % store the new node in the hash tables
    nth0(ActionIndex, ChProbs, ChProb),
    flag(nodecount, ChildId, ChildId+1),
    nb_hashtbl_set(ChildHash,ParentId-ActionIndex,ChildId),
    nb_hashtbl_set(ParentHash,ChildId,ParentId),
    nb_hashtbl_set(NodeHash,ChildId,
                   [ChildState,ChProb,1,ChildValue,ChChProbs]).
```

Finally, we update ancestor nodes after the insertion of the new leaf:

```
mc_backpropagate(Id,Reward,ParentHash,NodeHash):-
    nb_hashtbl_get(NodeHash,Id,[State,Prob,VCnt,Value,ChProbs]),
    VCnt1 is VCnt + 1,
    Value1 is Value + Reward,
    nb_hashtbl_set(NodeHash,Id,[State,Prob,VCnt1,Value1,ChProbs]),
    ( nb_hashtbl_get(ParentHash, Id, ParentId) ->
      mc_backpropagate(ParentId,Reward,ParentHash,NodeHash)
    ; true
    ).
```

C `leancop_step.pl` Module

Below we provide the code for the most important predicates that handle leanCoP inference steps in such a way that the entire prover state is explicitly maintained. For better readability, we omit some details, mostly related to problem loading, logging, and proof reconstruction. The `nondet_step` predicate takes a prover state along with the index of an extension or reduction step and returns the subsequent state. Before returning, it repeatedly calls the `det_steps` predicate,

which performs optimization steps that do not involve choice (loop elimination, reduction without unification, lemma, single action).

```
% init_pure(+File,+Settings,-NewState)
init_pure(File,Settings,NewState):-
    NewState = state(Goal,Path,Lem,Actions,Todos,Proof,Result),

    % store options
    retractall(option(_)),
    findall(_, ( member(S,Settings), assert(option(S)) ), _ ),

    % load tptp file and store contrapositives
    {...}

    % perform any potential optimizations
    det_steps([-(#)],[],[],[],[init((-#)-(-#))],
              Goal,Path,Lem,Todos,Proof,Result),

    % collect valid moves from this state
    % valid_actions(Goal, Path, Actions).


:- dynamic(alt/6).
% step_pure(+ActionIndex,+State,-NewState,-SelectedAction))
step_pure(ActionIndex,State,NewState,Action0):-
    State = state(Goal0,Path0,Lem0,Actions0,Todos0,Proof0,_),
    NewState = state(Goal,Path,Lem,Actions,Todos,Proof,Result),

    nth0(ActionIndex,Actions0,Action0),

    % if there were other alternative actions, store them
    (option(backtrack), Actions0=[_,_|_] ->
         select_nounif(Action0, Actions0, RemActions0), !,
         asserta(alt(Goal0,Path0,Lem0,RemActions0,Todos0,Proof0))
     ; true
    ),

    % perform any potential optimizations
    nondet_step(Action0,Goal0,Path0,Lem0,Todos0,Proof0,
                Goal1,Path1,Lem1,Todos1,Proof1,Result1),

    % if proof search fails, pop an alternative
    ( Result1 == -1, option(backtrack),
      pop_alternative(Goal,Path,Lem,Actions,Todos,Proof) ->
          Result=0,
     ; [Goal,Path,Lem] = [Goal1,Path1,Lem1],
```

```
        [Todos,Proof,Result] = [Todos1,Proof1,Result1],
        valid_actions(Goal,Path,Actions)
    ).

%%% make a single proof step from a choice point
% nondet_step(Action,Goal,Path,Lem,Todos,Proof,
%             NewGoal,NewPath,NewLem,NewTodos,NewProof,Result)
% reduction step
nondet_step(red(NegL),[Lit|Cla],Path,Lem,Todos,Proof,
            NewGoal,NewPath,NewLem,NewTodos,NewProof,Result):-
    neg_lit(Lit,NegL),
    Proof2 = {...}
    det_steps(Cla,Path,Lem,Todos,Proof2,
              NewGoal,NewPath,NewLem,NewTodos,NewProof,Result).
% extension step
nondet_step(ext(NegLit,Cla1,_),[Lit|Cla],Path,Lem,Todos,Proof,
            NewGoal,NewPath,NewLem,NewTodos,NewProof,Result):-
    neg_lit(Lit, NegLit),
    ( Cla=[_|_] ->
      Todos2 = [[Cla,Path,[Lit|Lem]]|Todos]
     ; Todos2 = Todos
    ),
    Proof2= {...}
    det_steps(Cla1,[Lit|Path],Lem,Todos2,Proof2,
              NewGoal,NewPath,NewLem,NewTodos,NewProof,Result).


% perform steps until the next choice point (or end of proof)
det_steps([],_Path,_Lem,Todos,Proof,
          NewGoal,NewPath,NewLem,NewTodos,NewProof,Result):-
    !,
    ( Todos = [] -> % nothing to prove, nothing todo on the stack
           [NewGoal,NewPath,NewLem,NewTodos,NewProof,Result] =
           [[success],[],[],[],Proof,1]
     ; Todos = [[Goal2,Path2,Lem2]|Todos2] ->
           % nothing to prove, something on the stack
           det_steps(Goal2,Path2,Lem2,Todos2,Proof,NewGoal,
                     NewPath,NewLem,NewTodos,NewProof,Result)
    ).
det_steps([Lit|_Cla],Path,_Lem,_Todos,Proof,
          NewGoal,NewPath,NewLem,NewTodos,NewProof,Result):-
    member(P,Path), Lit == P, !, % loop elimination
    [NewGoal,NewPath,NewLem,NewTodos,NewProof,Result] =
    [[failure],[],[],[],Proof,-1].
det_steps([Lit|Cla],Path,Lem,Todos,Proof,
```

```
          NewGoal,NewPath,NewLem,NewTodos,NewProof,Result):-
    member(LitL,Lem), Lit==LitL, !, % perform lemma step
    Proof2 = [lem(Lit)|Proof],
    det_steps(Cla,Path,Lem,Todos,Proof2,
              NewGoal,NewPath,NewLem,NewTodos,NewProof,Result).
det_steps([Lit|Cla],Path,Lem,Todos,Proof,
          NewGoal,NewPath,NewLem,NewTodos,NewProof,Result):-
    neg_lit(Lit,NegLit),
    ( option(eager_reduction(1)) ->
      member(NegL,Path),
      unify_with_occurs_check(NegL, NegLit), ! % eager reduction
    ; member(NegL,Path),
      NegL == NegLit, ! % reduction without unification is safe
    ),
    Ext = [NegL, NegL],
    Proof2 = [red(Ext-Ext)|Proof],
    det_steps(Cla,Path,Lem,Todos,Proof2,
              NewGoal,NewPath,NewLem,NewTodos,NewProof,Result).
det_steps(Goal,Path,Lem,Todos,Proof,
          NewGoal,NewPath,NewLem,NewTodos,NewProof,Result):-
    valid_actions(Goal,Path,Actions),
    ( option(single_action_optim),  Actions==[A] ->
          % only a single action is available, so perform it
          nondet_step(A,Goal,Path,Lem,Todos,Proof,NewGoal,
                      NewPath,NewLem,NewTodos,NewProof,Result)
    ;Actions==[] ->             % proof failed
          [NewGoal,NewPath,NewLem,NewTodos,NewProof,Result] =
          [[failure],[],[],[],Proof,-1]
    ; option(comp(PathLim)), \+ ground(Goal), length(Path,PLen),
      PLen > PathLim -> % reached path limit
      [NewGoal,NewPath,NewLem,NewTodos,NewProof,Result] =
      [[failure],[],[],[],Proof,-1]
    ;[NewGoal,NewPath,NewLem,NewTodos,NewProof,Result] =
     [Goal,Path,Lem,Todos,Proof,0]
    ).
```

D Policy and Value Functions

Here we describe 1) the default value and policy functions used in the first iteration, 2) the training data extraction and 3) how the predicted model values are used in MCTS. All these formulae are taken directly from rlCoP [19] and have been highly hand-engineered. We currently use these solutions in plCoP without altering them; however, we believe some of these decisions are worth reconsidering.

D.1 Value Function

In the first iteration, the default value V_d is based on the total term size of all open goals. Given s with total term size of open goals t, its value is

$$V(s) = \frac{1}{1 + e^{-3.7 * e^{-0.05t} + 2.5}} \tag{2}$$

After the MCTS phase, training data is extracted from the states in the bigstep nodes. If a state s is k steps away from a success node, its target value is 0.99^k. If none of its descendants are success nodes, then its target value is 0. We can then build a model using logistic regression. However, the authors of rlCoP find that the xgboost model works better if standard regression is used, so the target value is first mapped into the range $[-3, 3]$. The value V_t used for model training is

$$V_t(s, k) = min(3, max(-3, log(0.99^k/(1 - 0.99^k))))$$

In subsequent iterations, the prediction V_p of the model is mapped back to the $[0, 1]$ range (V'_p):

$$V'_p(V_p) = \frac{1}{1 + e^{-V_p}}$$

This value is further adjusted to give an extra incentive towards states with few open goals. If the state has g open goals, then the final value (V_f) used in MCTS is

$$V_f(V'_p, g) = (\sqrt{V'_p})^g$$

D.2 Policy Function

The default policy P_d is simply the uniform distribution, i.e., if a state s has n valid inferences, then each action a has a prior probability of

$$P_d(n) = \frac{1}{n}$$

After the MCTS phase, training data is extracted from the (state, action) pairs in the bigstep nodes. Target probabilities are based on relative visit frequencies of child nodes. These frequencies are again mapped to a range where we can do standard regression. Given state s with n valid inferences, such that s was expanded N times and its jth child was visited N_j times, then the policy P_t used for model training is

$$P_t(s, n, N, N_j) = max(-6, log(\frac{N_j}{N} n))$$

The prediction P_p is mapped back to the $[0, 1]$ range and normalized across all actions using the softmax function $\text{softmax}(x)_i = \frac{e^{\frac{x_i}{T}}}{\sum_j e^{\frac{x_j}{T}}}$, where T is the

temperature parameter that was set to 2. The final prior probabilities used in MCTS are

$$P_f(P_t) = \text{softmax}(P_t)$$

E Experiment Hyperparameters

plCoP is parameterized with configuration files (see examples in the `ini` directory of the distributed code), so the key parameters can be easily modified. Here we list the most important hyperparameters used in our experiments.

Feature Extraction. Our main features are term walks of length up to 3. We also add several more specific features: number of open goals, number of symbols in them, their maximum size and depth, length of the current path, and two most frequent symbols in open goals. The resulting feature vectors are sparse and long, so they are first compressed to a fixed size d: vector f is compressed to f', such that $f'_i = \sum_{\{j|j \bmod d=i\}} f_j$.

One difference from rlCoP is that they hash features to a fixed 262139 dimensional vector while plCoP uses a 10000 dimensional feature vector for faster computation. Over 5 iterations of our baseline on the M2k dataset, this even yields a small improvement (928 vs. 940), likely due to less overfitting.

MCTS. MCTS has an inference limit of 200000 steps and an additional time limit of 200 s. Bigsteps are made after 2000 steps. The exploration constant (cp) is 3 in the first iterations and 2 in later iterations.

leanCoP *Parameters.* leanCoP usually employs an iteratively increasing path limit to ensure completeness. We set path limit to 1000, i.e., we practically remove it, in order to allow exploration at greater depth.

XGBoost Parameters. To train XGBoost models, we use a learning rate of 0.3, maximum tree depth of 9, a weight decay of 1.5, a limit of 400 training rounds with early stopping if no improvement takes place over 50 iterations. We use the built-in "scale_pos_weight" XGBoost training argument to ensure that our training data is sign-balanced.

Furthermore, there is an option to keep or filter duplicate inputs with different target values. Our experiments did not show the importance of this feature, and all results presented in this paper apply duplicate filtering.

References

1. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., Urban, J.: Premise selection for mathematics by corpus analysis and kernel methods. J. Autom. Reason. **52**(2), 191–213 (2013). https://doi.org/10.1007/s10817-013-9286-5
2. Andrews, P.B.: On connections and higher-order logic. J. Autom. Reason. **5**(3), 257–291 (1989)

3. Anthony, T., Tian, Z., Barber, D.: Thinking fast and slow with deep learning and tree search. arXiv preprint arXiv:1705.08439 (2017)
4. Bansal, K., Loos, S.M., Rabe, M.N., Szegedy, C., Wilcox, S.: HOList: an environment for machine learning of higher-order theorem proving (extended version). arXiv preprint arXiv:1904.03241 (2019)
5. Beckert, B., Posegga, J.: Leantap: lean tableau-based deduction. J. Autom. Reason. **15**, 339–358 (1995)
6. Bibel, W.: Automated Theorem Proving. Artificial Intelligence, 2nd edn. Vieweg, Braunschweig (1987)
7. Biere, A.: Picosat essentials. J. Satisf. Boolean Model. Comput. (JSAT) **4**, 75–97 (2008)
8. Browne, C., et al.: A survey of Monte Carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **4**, 1–43 (2012)
9. Chen, T., Guestrin, C.: XGBoost: a scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2016, pp. 785–794 (2016). https://doi.org/10.1145/2939672.2939785
10. Chvalovský, K., Jakubův, J., Suda, M., Urban, J.: ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 197–215. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_12
11. Gauthier, T., Kaliszyk, C., Urban, J., Kumar, R., Norrish, M.: Learning to prove with tactics. arXiv preprint arXiv:1804.00596 (2018)
12. Goertzel, Z., Jakubův, J., Urban, J.: ENIGMAWatch: proofwatch meets ENIGMA. In: Cerrito, S., Popescu, A. (eds.) TABLEAUX 2019. LNCS (LNAI), vol. 11714, pp. 374–388. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29026-9_21
13. Jakubův, J., Urban, J.: ENIGMA: efficient learning-based inference guiding machine. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) CICM 2017. LNCS (LNAI), vol. 10383, pp. 292–302. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62075-6_20
14. Jakubuv, J., Urban, J.: Hammering Mizar by learning clause guidance. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving, ITP 2019, Portland, OR, USA, 9–12 September 2019. LIPIcs, vol. 141, pp. 34:1–34:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.34
15. Kaliszyk, C., Urban, J.: M2K dataset. https://github.com/JUrban/deepmath/blob/master/M2k_list
16. Kaliszyk, C., Urban, J.: Mizar40 dataset. https://github.com/JUrban/deepmath
17. 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). https://doi.org/10.1007/978-3-662-48899-7_7
18. Kaliszyk, C., Urban, J.: MizAR 40 for Mizar 40. J. Autom. Reason. **55**(3), 245–256 (2015). https://doi.org/10.1007/s10817-015-9330-8
19. Kaliszyk, C., Urban, J., Michalewski, H., Olsák, M.: Reinforcement learning of theorem proving. In: NeurIPS 2018, pp. 8836–8847 (2018)
20. 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). https://doi.org/10.1007/11871842_29

21. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
22. Letz, R., Stenz, G.: Model elimination and connection tableau procedures. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 2, pp. 2015–2114. Elsevier and MIT Press (2001)
23. Loos, S.M., Irving, G., Szegedy, C., Kaliszyk, C.: Deep network guided proof search. In: Eiter, T., Sands, D. (eds.) 21st International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), vol. 46, pp. 85–105 (2017)
24. Lukácsy, G., Szeredi, P.: Efficient description logic reasoning in prolog: the DLog system. TPLP **9**(3), 343–414 (2009). https://doi.org/10.1017/S1471068409003792
25. Muggleton, S., Raedt, L.D.: Inductive logic programming: theory and methods. J. Log. Program. **19/20**, 629–679 (1994). https://doi.org/10.1016/0743-1066(94)90035-3
26. Olsák, M., Kaliszyk, C., Urban, J.: Property invariant embedding for automated reasoning. arXiv preprint arXiv:1911.12073 (2019)
27. 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). https://doi.org/10.1007/978-3-540-71070-7_23
28. Otten, J.: MleanCoP: a connection prover for first-order modal logic. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 269–276. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_20
29. Otten, J., Bibel, W.: leanCoP: lean connection-based theorem proving. J. Symb. Comput. **36**, 139–161 (2003)
30. Ross, S., Gordon, G., Bagnell, D.: A reduction of imitation learning and structured prediction to no-regret online learning. In: Gordon, G., Dunson, D., Dudík, M. (eds.) Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics. Proceedings of Machine Learning Research, 11–13 Apr 2011, vol. 15, pp. 627–635. PMLR, Fort Lauderdale (2011). http://proceedings.mlr.press/v15/ross11a.html
31. Schulz, S.: E - a brainiac theorem prover. AI Commun. **15**(2–3), 111–126 (2002)
32. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016). https://doi.org/10.1038/nature16961
33. Silver, D., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815 (2017)
34. Silver, D., et al.: Mastering the game of go without human knowledge. Nature **550**(7676), 354 (2017)
35. Stickel, M.E.: A prolog technology theorem prover: implementation by an extended prolog computer. J. Autom. Reason. **4**(4), 353–380 (1988). https://doi.org/10.1007/BF00297245
36. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, vol. 1. Cambridge University Press, Massachusetts (1998)
37. Urban, J.: MPTP 0.2: design, implementation, and initial experiments. J. Autom. Reason. **37**(1–2), 21–43 (2006)
38. Urban, J., Sutcliffe, G., Pudlák, P., Vyskočil, J.: MaLARea SG1 - machine learner for automated reasoning with semantic guidance. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 441–456. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_37

39. 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). https://doi.org/10.1007/978-3-642-22119-4_21
40. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory Pract. Log. Program. **12**(1–2), 67–96 (2012)
41. Zombori, Z., Urban, J.: Learning complex actions from proofs in theorem proving. Accepted to AITP 2020. http://aitp-conference.org/2020/abstract/paper_11.pdf