EasyChair Preprint

№ 4433

Automated Theorem Proving, Fast and Slow

Michael Rawson and Giles Reger

January 25, 2021

Automated Theorem Proving, Fast and Slow

Michael Rawson Giles Reger

January 25, 2021

Abstract

State-of-the-art automated theorem provers explore large search spaces with carefully-engineered routines, but do not learn from past experience as human mathematicians can. Unfortunately, machine-learned heuristics for theorem proving are typically either fast or accurate, not both. Therefore, systems must make a tradeoff between the quality of heuristic guidance and the reduction in inference rate required to use it. We present a system that is completely insulated from heuristic overhead, allowing the use of even deep neural networks with no measurable reduction in inference rate. Given 10 seconds to find proofs in a corpus of mathematics, the system improves from 64% to 70% when trained on its own proofs.

1 Introduction

The great majority of automatic theorem provers use some kind of heuristic search. This could be simple, such as the use of iterative deepening on a certain property to achieve completeness [21]; complex, as in hand-engineered schemes [6]; or even learned in some way [36]. Such heuristics are critical for system performance: an excellent heuristic could find a proof in linear time, while a poor heuristic increases search time exponentially. Historically these routines have been engineered, rather than learned, resulting in fast yet disproportionately-effective heuristics like the age/weight schemes [30] used in systems like Vampire [13].

Learning a good heuristic from previous proof attempts has become more popular recently, and can achieve good results [3]. Techniques from machine learning can approximate complex functions that are difficult to discover or write down, but this comes at computational cost. This cost can result in an unfortunate outcome where a learned heuristic that appears promising during testing actually *degrades* performance when included in a concrete system, due to reduced inference throughput.

Even assuming a heuristic is both fast and accurate, it is not always clear how to gainfully include predictions into existing target systems, particularly as a single wrong prediction can sometimes have disastrous results. Approaches are either ad-hoc or adapt existing techniques from other domains which are not necessarily well-suited to theorem proving.

2 Contributions

We construct a system from scratch specifically designed to avoid these issues. lazyCoP is an automatic theorem prover for first-order logic with equality in the connection tableaux family (Section 4). The system may use a policy learned end-to-end from previous proofs (Section 6) to bias a special-purpose backtracking search (Section 5.1) toward areas the policy considers promising. Performance penalties are eliminated by asynchronously evaluating the policy network on a coprocessor, such as commodity GPU hardware (Section 5.2).

The result is a system in which learned guidance has no measurable impact on inference rate (Section 7.1) and learns in a feedback loop from previous proofs on a set of training problems (Section 7.2). No manual features are used for learning, and the only manual heuristic used is "tableaux with fewer subgoals are more likely to lead to a proof". The system augmented with the final learned policy improves from 64% to 70% in real time under identical conditions.

3 Related Work

The rlCoP system introduced in "Reinforcement Learning of Theorem Proving" [12] is the inspiration for this work and is most similar in spirit. A connection tableaux system is guided by Monte-Carlo Tree Search (*MCTS* henceforth, as in work on two-player games [32]), learning both policy and value guidance with gradient-boosted trees from hand-engineered features. Learning from previous proofs or failures is a common approach for many different applications of machine learning to theorem proving, avoiding the need to generate data manually. For instance, all learned premise-selection systems we are aware of are trained using premises used by automated systems in existing proofs [37, 10]. rlCoP sets up a feedback loop in which new information automatically found by the system is added to the training set in order to guide future iterations, as here.

Connection tableaux and classical first-order logic are popular settings for other internal guidance experiments — notably monteCoP [5], rlCoP, MaLeCoP [36], FEMaLeCoP [11], and FLoP [38] — but internal guidance for other domains exist, including first-order saturation systems [3], SAT and QBF solvers [31, 14], and systems for higher-order logics [1, 4].

Performance is a recurring problem for systems with learned internal guidance. The authors of rlCoP exclude some kinds of learned models for performance reasons, and results are reported based on an inference, rather than time, limit. "Deep Network Guided Proof Search" [16] reports that the main bottleneck in the guided saturation-style system E [29] is the evaluation of inferences, and suggest a two-phase guided/unguided approach to theorem proving with learned guidance. Asynchronous evaluation was suggested in earlier work on the same problem [24].

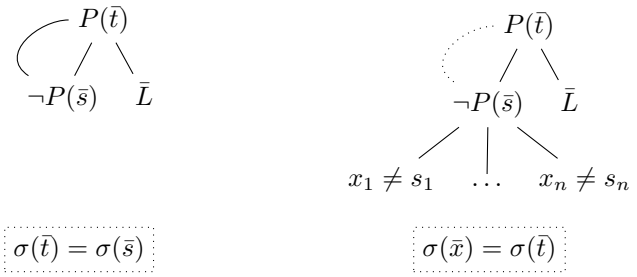

Figure 1: Adding $\neg P(\bar{s}) \vee \bar{L}$ to a tableau where $P(\bar{t})$ is the current goal. The left tableau shows conventional "strict" extension, the right LPCT "lazy" extension.

4 Unguided System

If an unguided system is completely hopeless, little progress can be made: very few positive training data can be generated from successful proofs, and the learned guidance must be better still in order to achieve reasonable performance. However, it is not as simple as selecting a state-of-the-art theorem prover, as some are more amenable to guidance than others. Instead, there is a spectrum of different possible research directions, from attempting to guide weaker-yet-amenable systems up to meet stronger unguided systems, to integrating learning into already-strong systems which are not so easily improved by guidance.

The guidance scheme suggested here is designed for backtracking search, such as that found in systems based on connection calculi. It is not clear how this could be adapted to a modern saturation theorem prover such as Vampire or E, which employ proof-confluent search with a time-sensitive choice point at the selection of a given clause. The basic system must therefore be as strong as possible while still allowing backtracking policy-guided search, and lazyCoP is purpose-built for this. A prototype version [26] entered the most recent CASC competition [34], and subsequent developments including a dedicated clausification routine have significantly improved performance.

4.1 Connection Tableaux

lazyCoP belongs to the connection-tableaux/model-elimination family [15] of theorem provers, which includes systems such as leanCoP and SETHEO. Such systems aim to *refute* a proposition by building a *closed tableau*: a tree of case-splits such that every path through the tree ends in a contradiction. *Connection* tableaux reduce the search space by constraining tableaux such that each addition to any given tableau must be *connected* in some way to the current leaf, as shown on the left-hand side of Figure 1 where $P(\bar{t})$ connects to $\neg P(\bar{s})$. To *prove* a conjecture, it suffices to begin with the negated conjecture and build a closed tableau refuting it.

Since there is often more than one possible next step in building a tableau, not all of which lead to a proof, it is necessary to backtrack if

a misstep is made. Typical connection systems often use some kind of iterative deepening to maintain completeness, but any fair scheme works: rlCoP uses MCTS for this purpose.

4.2 Lazy Paramodulation

Reasoning with equality has traditionally been a weak point of connection systems. The most widespread method for efficiently reasoning with equality, *paramodulation* [18], is incomplete in the obvious formulation for connection tableaux due to insufficient flexibility in the order of inferences. There have been various attempts to remedy this deficit, but as yet there is no conclusive solution.

lazyCoP uses the "lazy paramodulation" proof calculus LPCT [23], which relaxes some of the classical connection-tableaux rules in exchange for a paramodulation-like rule and some extra refinements. The basic idea is delaying unification to allow rewriting terms in the resulting disequations. For example, in the right-hand side of Figure 1, it is not required that $P(\bar{t})$ unify with $P(\bar{s})$ immediately as in the classical calculus, instead deducing that at least one of the terms must not be equal. Terms may still be unified with a reflexivity rule dispatching goals of the form $t \neq s$.

This implementation detail of lazyCoP is not the main focus of this work: the vital feature of the proof calculus is backtracking proof search.

4.3 Calculus Refinements

To improve performance against the pure calculus, lazyCoP implements a number of well-known refinements of the classical predicate calculus (which are lifted to equalities where appropriate), including tautology deletion, various regularity conditions, and *folding up*, a way of re-using proofs of literals. Additionally, it is frequently the case that a unification is "lazy" when it could have been "strict" — such as in the case with no equality. lazyCoP therefore implements "lazy" and "strict" versions of every relevant inference rule, which shortens some proofs considerably. The resulting duplication is eliminated by not permitting "lazy" rules to simulate their "strict" counterparts.

It is not clear whether some refinements help or hinder the learned-guidance scenario. Some are definite improvements: folding up and strict rules decrease proof lengths and therefore increase the potential benefit of learned guidance. However, others, such as the regularity condition or the term ordering constraints in LPCT, are not as clear-cut. In some cases such refinements lengthen proofs significantly, outweighing the pruning effect, and previous work shows that guidance can partially replace these pruning mechanisms [7]. We leave all refinements switched on for this approach, but allowing the learned policy a greater amount of freedom is an interesting future direction.

Some techniques such as *restricted backtracking* [22] sacrifice completeness for performance. lazyCoP does not implement any approach known to be incomplete: all problems attempted can be solved in principle.

5 Proof Search

Given a learned policy, we aim to use it to improve proof search outcomes. The *policy* $\pi(a \mid n)$ is a function from a tableau n and possible inferences a to a probability distribution. We work with an explicit search tree, each node of the tree representing an open tableau, although tableaux are not actually kept in memory for efficiency reasons. From each open tableau, there is a positive non-zero number of possible inferences (or *actions* in the reinforcement learning literature) which may be applied to generate a new child tableau. Nodes with zero possible inferences cannot be closed and are pruned from the tree. The root of the tree is an empty tableau, from which possible inferences are the *start clauses*, in this case clauses derived from the conjecture.

5.1 Policy-Guided Search

There are many possible tree search algorithms which can include some kind of learned heuristic. We experimented with the classical A^* informed-search procedure, although we found that it was difficult to learn a good heuristic function that was neither too conservative nor too aggressive. Other approaches might include the aforementioned MCTS, single-player adaptations of MCTS [28] single-agent approaches like that of LevinTS or LubyTS [20], or simply following a stochastic policy with restarts if no proof is found at some depth. While these approaches are no doubt interesting and provide theoretical guarantees, we did not find them to be necessary for our case.

Instead, we could simply employ best-first search, expanding the leaf node that the policy considers most likely first. If a leaf node n was obtained by taking actions a_i from parent nodes n_i, select

$$\operatorname*{argmax}_n \prod_i \pi(a_i \mid n_i)$$

Unfortunately, this simple scheme is not likely to recover if π makes a confident misprediction, and is even incomplete if any node has an infinite chain of single children beneath, where $\pi(a_j \mid n_j) = 1$ by definition. To correct this issue we take inspiration from rlCoP's initial value heuristic, where tableaux are exponentially less likely to be closed the more open branches they have. We model this idea as an exponential distribution

$$p(n) = \lambda e^{-\lambda g(n)}$$

where λ is a tunable parameter (set to 1 in our experiments here) and $g(n)$ is "number of open branches plus length of the active path". Including "length of the active path" in $g(n)$ makes little practical difference and makes the search procedure complete again. The two estimates are combined with a geometric mean so that nodes are selected by

$$\operatorname*{argmax}_n \sqrt{p(n) \prod_i \pi(a_i \mid n_i)}$$

In practice this expression is numerically difficult to evaluate, but in logarithmic space it is better-behaved, producing the final expansion criterion

$$\underset{n}{\operatorname{argmax}} \left[\left(\sum_i \ln \pi \left(a_i \mid n_i \right) \right) - \lambda g(n) \right]$$

5.2 Asynchronous Policy Evaluation

The proof search routine above assumes that the policy is evaluated synchronously for each expanded node. As discussed in the introductory sections, this has a significant impact on performance, particularly so for computationally-expensive policies. Instead, evaluation is deferred and a separate CPU thread continuously arranges for nodes to be processed on a GPU, selecting the first non-evaluated node on the path to the current best leaf node. $\pi \left(a \mid n \right)$ is set to 1 for nodes not yet evaluated: applying a uniform distribution does not work well in practice.

It does not appear to be particularly important that all nodes are evaluated for a learned policy to improve search, perhaps because guidance at the top of the search tree has a disproportionate effect. Asynchronous policy evaluation allows use of policies that are orders of magnitude slower than expansion steps without reduction in inference rate.

6 Learned Policy

Section 5.1 describes biasing proof search with a learned policy, directing node expansions toward areas the policy considers useful. lazyCoP's policy is trained from its own proofs: at each non-trivial step in proofs the tableau, all available actions and the action that lead to a proof is recorded. This procedure produces a training set of tableaux and actions which we use to train a neural-network based policy to predict the correct action. Learning from existing system proofs in this way has advantages and disadvantages: each example's label is guaranteed to lead to a proof, but it is not necessarily the shortest proof, nor can the training data express preference amongst other actions.

We train and evaluate using the same set of problems from the MPTP translation [35] of the Mizar Mathematical Library [8] into first-order logic with equality. There are 32,524 problems in total in the *M40k* set; we use the *M2k* subset of 2003 problems in order to iterate quickly. All problems have a labelled conjecture which lazyCoP is able to exploit so that search proceeds backward from the conjecture. Problems from the *M2k* set come from related articles in Mizar, suggesting a degree of similarity which may be exploited by learning.

6.1 Representing Tableaux with Actions

There are many possible ways to represent first-order logical data in neural networks. We selected directed graphs paired with residual graph convolutions, as introduced for other tasks on logical data [25]. This approach has significant advantages for a first-order tableau system such as lazyCoP

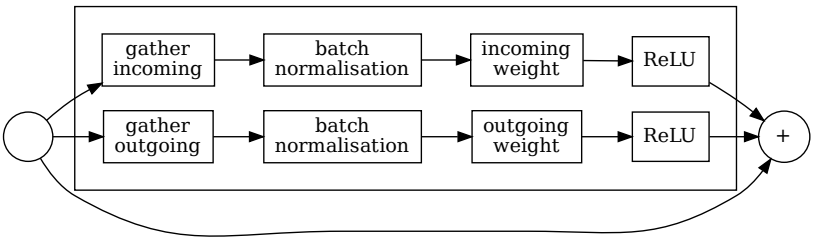

Figure 2: Residual block used in the network. Note disjoint parameters for incoming and outgoing edges, both linear and normalisation layers.

as it allows reconstructing an equivalent tableau (up to renaming) from a compact, pre-parsed representation invariant up to e.g. variable names.

Construction of directed graphs from tableaux is mostly typical for first-order representations [37], with a few problem-specific modifications. First, while occurrences of identical symbols and variables share nodes in the graph, identical compound terms do not: this is because they may be rewritten by equalities separately in LPCT. Additionally, variable binding is non-destructive in LPCT to implement a form of basic paramodulation. Bound variables therefore remain in place but have an outgoing edge attached to their binding.

Encoding actions is then straightforward. lazyCoP implements a small number of different types of inference, such as reductions, extensions, reflexivity and so on. Each inference is attached to some terms or literals in the tableau to form a concrete action: rewriting $t = s$ in $L[p]$, for example, is represented as a node connected to the graph with an incoming edge from t and outgoing edge from p, uniquely identifying the inference.

6.2 Network Architecture

We use a residual version of the directed graph networks introduced in previous work [25] which allow the network to distinguish incoming and outgoing edges. The core of the network is the residual block shown in Figure 2: this allows one round of message-passing from neighbouring nodes in the graph, treating incoming and outgoing edges separately before combining the results for the next layer. Batch normalisation [9] is inserted before the linear part of the convolution. The theoretical merits of this are unclear but it works well in practice. The complete network (Figure 3) is, in order:

Embedding. An embedding layer projects integer node labels into a real vector of the same size used in the convolutional layers.

Convolution layers. Several residual blocks combine and transform feature maps from neighbouring nodes, producing in particular a real

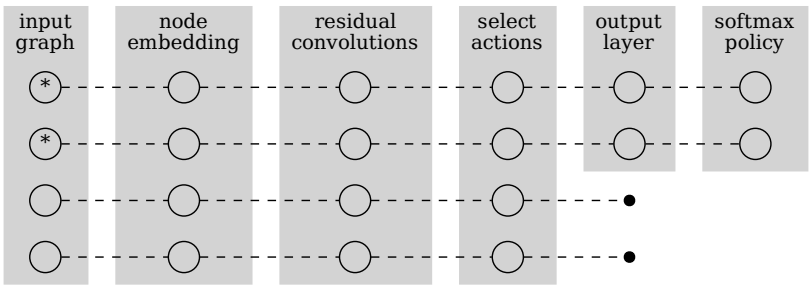

Figure 3: Network diagram. As there is no pooling of any kind, data is processed at the node level until action nodes marked (*) are projected out.

Table 1: Network and training hyper-parameters.

Parameter	Value	Parameter	Value
node dimension	64	initial learning rate	0.01
residual layers	24	cycle batches	2000
		batch size	64
		momentum	0.9
		weight decay	0.0001

vector for each action node.

Action projection. The vector for each action node is projected out, all other nodes are discarded at this point.

Output layer. Computes a single output value for each action.

Rectified linear units are used as non-linearities throughout.

6.3 Training

Training such a network on limited training examples from early iterations is challenging due to its tendency to memorise the training set if sufficient parameters are available and underfit drastically if they are not. This is perhaps a good argument for feature-based learning rather than the end-to-end approach we take here. However, the network can be made to train somewhat effectively by cosine annealing a high initial learning rate to 0 with "warm restarts" [17], repeating after a certain number of mini-batches. This has two benefits: the regularising effect of high learning rates somewhat reduces overfitting, and the network also trains faster.

Table 2: Results from iterative training of lazyCoP's policy on *M2k*.

#	Proved	Cumulative	Steps
0	1,289	1,289	16,880
1	1,390	1,406	19,394
2	1,402	1,419	19,700
3	1,403	1,426	19,881

6.4 Integration and Optimisation

After the network is trained, network weights are compiled into lazyCoP. The forward pass is re-implemented from scratch in CUDA [19], allowing a number of optimisations such as known array sizes, re-use of allocated buffers and the ability to profile for the specific workload. Additionally, batch normalisation layers' forward operation can be fused into the subsequent layer in this case, decreasing implementation complexity and increasing performance.

7 Experimental Results

We investigate two areas of practical interest: the effect of learned policy evaluations on inference rate, and whether this learning translates into improved performance on a training set of problems. Systems are only allowed 10 seconds of real time: this is relatively short, but a good approximation to real-world settings in which users of automatic "hammers" included in interactive theorem proving systems are unwilling to wait much longer than 30 seconds [2].

7.1 Inference Rates

There is no measurable decrease in inference rate when learned guidance is switched on. Occasionally the rate of inference even *improves*, perhaps due to guidance producing areas which are less productive or otherwise easier to explore. Running on `TOP001-1`, a non-theorem mid-sized topology problem from TPTP [33], unguided lazyCoP achieves around 62,000 expansions per second for 10 seconds at the time of writing on desktop hardware. Guided, the system evaluates around 200 policies per second and reaches inference speeds in excess of 70,000 expansions per second.

7.2 Effect of Guidance

We train lazyCoP iteratively on *M2k* as described in Section 6, training each iteration on the proofs produced by all previous iterations. Iteration 0 does not have access to a learned policy, iteration 1's policy is trained on iteration 0's proofs, iteration 2 on proofs from both iteration 0 and 1, etc. If there are two proofs for the same problem, the shorter proof is retained. The system is given 10 seconds of real time per problem, measured from

program startup to the point of discovering a proof (but before output begins), and an unlimited amount of memory. Table 2 shows the number of problems solved by that iteration, the number of problems proved by all previous iterations, and the total number of proof steps for training available after the iteration finishes.

8 Remarks

8.1 Future Work

There are several future directions we will consider pursuing:

Scaling network and problem sets. It is very possible that a larger/deeper policy network would allow learning even better policies. This requires either more careful tuning or a larger set of problems such as *M40k* to avoid overfitting excessively.

Parallelism. Implementing both parallel search and parallel evaluation on today's multicore machines would have a beneficial impact on performance. Parallel search allows exploiting remaining cores to search faster and is a clear win, the explicit search tree of lazyCoP allowing for several easy schemes to inject parallelism. Parallel evaluation does not inherently improve performance, but does ensure that the coprocessor is always kept busy: at present there are short pauses while the evaluation thread propagates the previous evaluation and prepares another input. Using multiple host threads also allows hiding latency from e.g. coprocessor cache misses, increasing overall throughput at the expense of the speed of single evaluation.

Incomplete modes. A system does not necessarily have to be complete to be useful. leanCoP includes a powerful but incomplete restricted-backtracking mode, for example. As well as e.g. restricted backtracking, lazyCoP could implement a strategy in which parts of the search tree are progressively discarded as resource limits draw nearer, in a similar way to Vampire's *limited resource strategy* [27]. We expect this to help with finding extremely long proofs.

References

[1] K. Bansal, S. Loos, M. Rabe, C. Szegedy, and S. Wilcox. HOList: An environment for machine learning of higher order logic theorem proving. In *International Conference on Machine Learning*, pages 454–463, 2019.

[2] S. Böhme and T. Nipkow. Sledgehammer: judgement day. In *International Joint Conference on Automated Reasoning*, pages 107–121. Springer, 2010.

[3] K. Chvalovský, J. Jakubův, M. Suda, and J. Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In *International Conference on Automated Deduction*, pages 197–215. Springer, 2019.

[4] M. Färber and C. Brown. Internal guidance for Satallax. In *International Joint Conference on Automated Reasoning*, pages 349–361. Springer, 2016.

[5] M. Färber, C. Kaliszyk, and J. Urban. Monte-Carlo connection prover. *Second Conference on Artificial Intelligence and Theorem Proving*, 2017.

[6] B. Gleiss and M. Suda. Layered clause selection for theory reasoning. In N. Peltier and V. Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 402–409, Cham, 2020. Springer International Publishing.

[7] Z. A. Goertzel. Make E smart again. In *International Joint Conference on Automated Reasoning*, pages 408–415. Springer, 2020.

[8] A. Grabowski, A. Kornilowicz, and A. Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010.

[9] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[10] G. Irving, C. Szegedy, A. A. Alemi, N. Eén, F. Chollet, and J. Urban. DeepMath — deep sequence models for premise selection. In *Advances in Neural Information Processing Systems*, pages 2235–2243, 2016.

[11] C. Kaliszyk and J. Urban. FEMaLeCoP: Fairly efficient machine learning connection prover. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 88–96. Springer, 2015.

[12] C. Kaliszyk, J. Urban, H. Michalewski, and M. Olšák. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems*, pages 8822–8833, 2018.

[13] L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[14] G. Lederman, M. Rabe, S. Seshia, and E. A. Lee. Learning heuristics for quantified boolean formulas through reinforcement learning. In *International Conference on Learning Representations*, 2020.

[15] R. Letz and G. Stenz. Model elimination and connection tableau procedures. In *Handbook of Automated Reasoning*, volume 2. MIT Press, 2001.

[16] S. Loos, G. Irving, C. Szegedy, and C. Kaliszyk. Deep network guided proof search. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, pages 85–105, 2017.

[17] I. Loshchilov and F. Hutter. SGDR: stochastic gradient descent with warm restarts. In *5th International Conference on Learning Representations*, 2017.

[18] R. Neuwenhuis and A. Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning*, volume 1. MIT Press, 2001.

[19] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[20] L. Orseau, L. Lelis, T. Lattimore, and T. Weber. Single-agent policy tree search with guarantees. In *Advances in Neural Information Processing Systems*, pages 3201–3211, 2018.

[21] J. Otten. leanCoP 2.0 and ileanCoP 1.2: High-performance lean theorem proving in classical and intuitionistic logic. In *International Joint Conference on Automated Reasoning*, pages 283–291. Springer, 2008.

[22] J. Otten. Restricting backtracking in connection calculi. *AI Communications*, 23(2-3):159–182, 2010.

[23] A. Paskevich. Connection tableaux with lazy paramodulation. *Journal of Automated Reasoning*, 40(2-3):179–194, 2008.

[24] M. Rawson and G. Reger. A neurally-guided, parallel theorem prover. In *International Symposium on Frontiers of Combining Systems*, pages 40–56. Springer, 2019.

[25] M. Rawson and G. Reger. Directed graph networks for logical reasoning. In *Practical Aspects of Automated Reasoning*, 2020.

[26] M. Rawson and G. Reger. lazyCoP 0.1. EasyChair Preprint no. 3926, EasyChair, 2020.

[27] A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, 36(1-2):101–115, 2003.

[28] M. P. Schadd, M. H. Winands, H. J. Van Den Herik, G. M.-B. Chaslot, and J. W. Uiterwijk. Single-player monte-carlo tree search. In *International Conference on Computers and Games*, pages 1–12. Springer, 2008.

[29] S. Schulz. E — a brainiac theorem prover. *AI Communications*, 15(2, 3):111–126, 2002.

[30] S. Schulz and M. Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In *International Joint Conference on Automated Reasoning*, pages 330–345. Springer, 2016.

[31] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill. Learning a SAT solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

[32] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[33] G. Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337, 2009.

[34] G. Sutcliffe. The CADE ATP system competition — CASC. *AI Magazine*, 37(2):99–101, 2016.

[35] J. Urban. MPTP 0.2: Design, implementation, and initial experiments. *Journal of Automated Reasoning*, 37(1-2):21–43, 2006.

[36] J. Urban, J. Vyskočil, and P. Štěpánek. MaLeCoP: machine learning connection prover. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 263–277. Springer, 2011.

[37] M. Wang, Y. Tang, J. Wang, and J. Deng. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems*, pages 2786–2796, 2017.

[38] Z. Zombori, A. Csiszárik, H. Michalewski, C. Kaliszyk, and J. Urban. Towards finding longer proofs. *arXiv preprint arXiv:1905.13100*, 2019.