CONNECT++

A fast, flexible and modifiable connection prover to support machine learning

Dr. Sean B. Holden[1], PhD (University Associate Professor)

[1]*University of Cambridge, Department of Computer Science and Technology, The Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK.*

Abstract

CONNECT++ is an automated theorem prover for first-order logic with equality, based on the clausal connection calculus and designed with three primary goals. The first was to produce a system that is *fast*, through careful coding in a compiled language (C++). The second was to allow the system to support the addition of *machine learning* to the maximum extent possible. The third, somewhat inspired by the success of the MINISAT solver for Boolean satisfiability, was to provide an implementation sufficiently modifiable as to provide a *common basis* for experiments by others. In addition to these aims I wanted to exploit the opportunities inherent in the connection calculus to explore the production of readable and certified proofs. This paper describes the system as it stands; development is ongoing and some plans for the future are also outlined.

Keywords

Connection Prover, C++, Machine Learning, Certified Proof, Clausal Connection Calculus

1. Introduction

CONNECT++ is a prover for first-order logic, implemented in C++ and based on clausal connection calculus.[1] The advantages of connection provers, with respect to their goal-oriented search and ability to produce readable proofs, are well-known, and have led to great interest in their use for applying *machine learning (ML)* to *automated theorem proving (ATP)*. This paper introduces CONNECT++, and gives a high-level description of the system; it also motivates its development. The paper falls a little outside the usual form of discourse for such papers, combining three related perspectives. First, it is a 'system description'. Second, it argues for design decisions based on practical experience in conducting large-scale experiments; a pursuit often mandating use of preferred technologies for these two, often disparate research areas. Third, it presents motivations underlying the system's design decisions, based purely on accommodating ML.

The initial motivation for CONNECT++ arose while writing a review of ML applied to *satisfiability (SAT) solvers* [1]. It was apparent that there are many SAT solvers, but an *enabler of research* over the last two decades was MINISAT [2], which had a transformative effect. MINISAT

AReCCA 2023: Automated Reasoning with Connection Calculi. International Workshop, TABLEAUX 2023. 18th September 2023. Prague, Czech Republic.

sbh11@cl.cam.ac.uk (S. B. Holden)

https://www.cl.cam.ac.uk/~sbh11 (S. B. Holden)

0000-0001-7979-1148 (S. B. Holden)

© 2023 Copyright for this paper by its author. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

[1]The system will be open-sourced via github.com later this year, with the current aim of making it available along with the full version of this paper.

was fast enough to be competitive,[2] while being sufficiently easy to modify that it provided a tool for other researchers. Much of the research leading to today's best solvers relied on MiniSAT. One aim of Connect++ is to provide a similar basis for experiments with connection provers. (And this is true regardless of any ML.)

At present, several connection calculus provers are available, implemented in Prolog [5, 6], C [7], OCaml [8, 9], Rust [10], Python [11] and C++ [12]. This has led to a difficulty which in the SAT world MiniSAT helped to defray: if two researchers conduct ML experiments using different solvers, how do we decouple the effect of the ML from the characteristics of the solvers? For one to re-implement the other's ML work on their own system would be a solution; but the use of a common solver, in the manner of MiniSAT, would benefit both sides.

Further motivation to develop Connect++ arose from two observations made while working on ML applied to ATP more generally:

1. The ATP world has two main currencies: proving *more things* and proving *known things faster*. This motivates the development of ATPs that are inherently *fast*. It is also in conflict with the use of *lean* ATPs for connection calculus, which tend to rely on languages known not to be the fastest.
2. The ML world is heavily invested in Python, with the most prevalent libraries using it to implement their API. To produce research in ML for ATP, one needs to run large numbers of experiments. My own groups' experience in continuing the work presented in [13] was that there are significant difficulties in combining Python (for ML), Prolog (for leanCop [5, 6]), and one or more compiled languages supporting the overall process, into an experiment on a high-performance computing facility.

These points suggested the development of a connection prover in a fast, complied langauge such as C++, or in Python. C++ emphasizes the speed requirement and is the approach taken here, addressing Point 1. Python is also the subject of current work [11]. Both approaches address Point 2, but in complementary ways.

Further motivations appeared while I worked on the development of Connect++, and these are some of the most central issues, both discussed further in Section 2:

1. The lean approach to ATP, exploiting the inherent strengths of Prolog, leads to beautiful, compact implementations. However Prolog's model of computation based on backtracking search is not always amenable to the use of ML. (See also Point 2 above.) There are two issues here: APIs allowing ML methods to interact with a running Prolog program are often inadequate, and Prolog's model of computation itself can present a barrier.
2. In particular, the Prolog `cut` is perfect for leanCop's backtracking restriction [6]. However it is a blunt instrument, and more subtle backtracking heuristics might profitably be explored, in turn providing new ways for ML to control the proof search. However the implementation of such alternatives in Prolog potentially becomes cumbersome.

While the realisation that the use of C++ supported these points was serendipitous, the overall design of Connect++ was informed by them as described in Section 4, and they support the overall aim of making a system that supports ML to the widest possible extent.

[2] I will not be claiming that Connect++ currently competes with the fastest solvers such as E [3] and Vampire [4]; the comparison is among connection solvers.

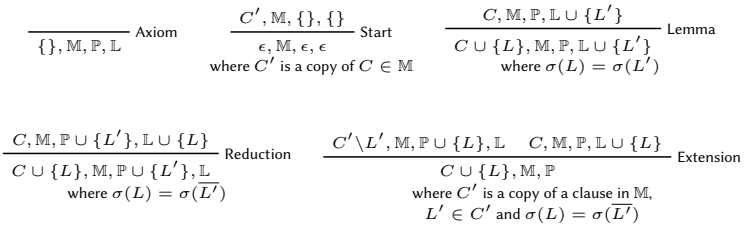


Figure 1: Rules for constructing a clausal connection calculus proof. See Figure 3 for an example proof.

A final motivation for Connect++ was to explore the production of readable proofs and proof certificates.

2. ML for connection calculus: why use C++?

Prolog provides compact implementations for connection provers, but to facilitate ML there are good reasons to consider alternatives. In this section I attempt to explain my reasoning.

Connect++ implements the clausal connection calculus with lemmata and regularity, as in Figure 1. The *matrix* $\mathbb{M}$ is the set of clauses for the problem. The *path* $\mathbb{P}$ is a set of literals, as is the set $\mathbb{L}$ of lemmata. A copy of an item has a fresh set of unused variables. An overline denotes the complement of a literal. Regularity adds the condition

$$\forall L' \in C \cup \{L\} \,.\, \sigma(L') \notin \sigma(\mathbb{P})$$

to the Reduction and Extension rules. When applying the Reduction and Extension rules the substitution is applied to the entire proof. In the case of Regularity and Lemmata we test for equality using the substitution as it stands, but no further unification is applied.

The rules in Figure 1 are the basis for a classical backtracking search problem. leanCop, for example, arranges this search using Prolog's underlying search algorithm. It is well-known that the use of heuristics to order such a search is critical for good performance. For the calculus described, there is ample scope for learning such heuristics; for example, at any point in the search there may be multiple ways to extend a partially complete proof, possibly including mutliple applicable uses of Reduction, Extension, or Lemmata rules, and we might aim to learn a good choice of the next to try. ML algorithms therefore need to be able to influence such choices, and the extent to which this is feasible will depend on the details of the implementation of a prover. This is a particular area where we need a solver implementation facilitating fine control over search heuristics by external ML code.

Some of the most successful heuristics for connection provers go further, relying on the restriction of backtracking during the proof search [6]. In the Reduction, Extension and Lemmata rules, L is called the *principal literal* when the rule is selected as a candidate for extending a proof. For the Reduction and Lemmata, L is considered *solved*; for an Extension it is considered solved if the left sub-tree of the Extension is completed. After L is solved, no other options for solving it will be considered on backtracking. This heuristic is remarkably effective and the Prolog `cut` makes its implementation extremely convenient. However, by moving beyond the

use of `cut` we potentially open a wide design space of learnable backtracking heuristics. For example, having solved a literal using Extension, should we backtrack within the left subtree or discard it completely? Should we stop backtracking for a principal literal after it has been solved once, or should we limit the *maximum* number of ways we try to solve it? Should we limit backtracking just for this principal literal, or remove *other* possibilities once some literal has been solved? There is considerable opportunity for new work here, and facilitating such research is a key motivation for Connect++. (See Section 5 and [14], which provide some results in this direction.)

Other successful heuristics include restriction of backtracking on start clauses, forms of reordering and randomization [15], and others. Ideally a solver should support all forms of heuristic, and allow ML algorithms to modify them. However we should also consider a further potential need: to modify the operation of the proof search *dynamically*. Much research on ML for ATP addresses the tuning of heuristics that are then *fixed* when attempting new proofs. There is ample evidence from work on SAT solvers that learning on a *per-proof* basis is extremely effective. For example, *variable selection heuristics* are light-weight learning algorithms adjusting variable choice using feedback obtained *while* the proof search progresses. This type of learning also deserves attention in ATP, and the need to modify heuristics during proof search is better served by a move away from Prolog.

A final consideration relates to the way in which the proof search is structured. Some systems search recursively, trying the left branches of Extensions first. There has been considerable interest in applying *reinforcement learning* [16] and *Monte-Carlo tree search* [17] to connection provers, and in these cases it makes sense to allow proofs to be constructed in a less constrained manner. This is the subject of complementary work on representing the process as a *Markov Decision Process* [11], but similar arguments apply regarding the need for a system developed specifically to support ML.

3. Connect++

Connect++ supports most of the functionality provided by leanCop, and has some additional facilities. It supports restricted backtracking, restriction of start clauses and reordering. It deals with equality by detecting its use in the input file and adding the necessary axioms; more sophisticated approaches, such as described in [18, 19], are a subject for future extension. It assumes negative (CNF) representation by default but can use positive (DNF) representation—this changes the equality axioms added, and the behaviour for some of the start clause selection options. It supports iterative deepening by path length or tree depth; switching to complete search after a given depth; specified start depth and depth increment; and can detect non-theorems if the search is exhausted when settings are for a complete search. It accepts input in the conjunctive normal form (CNF) format of the *Thousands of Problems for Theorem Provers (TPTP)* [20] library. Support for problems in first-order form is a work in progress, and subsequently it does not at present support *definitional clausal form (DCF)* [6]; this is perhaps the first priority for further development as leanCop's standard schedule employs three different settings related to DCF.

A lesson learned applying ML to SAT is that solvers should avoid hiding within their implementation, parameters that are potentially important for tuning performance. Instead, such

```
2  limitedstart complete 7 ;
60 conjecturestart nocomplete ;
20 limitedstart restrictstart nocomplete ;
2  conjecturestart reorder 23 nocomplete ;
2  limitedstart restrictstart reorder 29 nocomplete ;
2  conjecturestart reorder 37 nocomplete ;
2  limitedstart restrictstart reorder 41 nocomplete ;
2  conjecturestart reorder 47 nocomplete ;
0  limitedstart allbacktrack nocomplete ;
```

Figure 2: Default schedule file for CONNECT++.

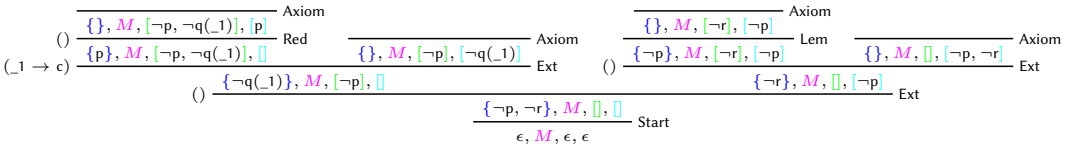


Figure 3: LaTeX output for Example 1. Variables of the form _1, _2, . . . are fresh variables introduced by the Start or Extension rules. Annotations to the left of a rule, such as $(\_1 \to c)$, denote substitutions applied to the entire proof.

parameters should be exposed, preferably at the command line, so that they can be adapted to suit a particular domain of application. Automated systems for this task, such as ParamILS [21], SMAC [22] and GGA [23] have been used with great success, as has Bayesian optimisation [24]. CONNECT++ exposes a large, and growing, number of such parameters.

Some connection provers run with a hard-coded schedule optimized through experiments. A schedule is a sequence of sets of parameter settings, each element of the sequence being assigned a percentage of the run time. The prover might start with a schedule line stating that it should run with full backtracking for 10% of the time, then switch to restricted backtracking for 5% of the time, then switch to a different configuration and so on. Figure 2 shows the current default schedule used by CONNECT++, which is similar to that of leanCop version 2.0. The differences are necessary as CONNECT++ does not at present implement definitional clausal form. This is clearly an effective method, but there is evidence that learning of schedules can be beneficial [25]. CONNECT++ supports the use of arbitrary schedules and can read these from a file in a simple format, making it easy to incorporate the results of ML applied to schedule choice, and potentially to support the ML process for learning schedules.

CONNECT++ can produce a readable proof via LaTeX. Example 1 is a problem from [6].

Example 1. $\mathbb{M} = \{\{\neg P, \neg R\}, \{P, Q(c)\}, \{P, \neg Q(x)\}, \{\neg P, R\}, \{\neg P, Q(a)\}\}$.

Figure 3 shows the typeset output for a proof of Example 1.

CONNECT++ can output a simply-formatted proof certificate. While there is currently no consensus on what format a certificate should take, suggestions have appeared [26, 8, 27] and a proposal for a standard is given in [28]. Figure 4 shows a certificate for the proof shown in Figure 3. It consists of Prolog-readable summaries of the matrix and a stack representation of the proof (see Section 4). Each element of the stack describes the proof rule employed,

```
matrix(0, [ -p, -r ]).
matrix(1, [ p, q(c) ]).
matrix(2, [ p, -q(X) ]).
matrix(3, [ -p, r ]).
matrix(4, [ -p, q(a) ]).
```

(a) Prolog-readable representation of the matrix.

```
proof\_stack([
start(0),
left\_branch(2, 0, 2),
left\_branch(1, 1, 3),
reduction(0),
right\_branch(3),
right\_branch(2),
left\_branch(3, 1, 3),
lemmata(),
right\_branch(3)
]).
```

(b) Stack representation of the proof.

Figure 4: Certificate for the proof of Example 1.

with `left_branch` and `right_branch` denoting the left and right premises of the Extensions. Numbers after an item identify the element(s) used in applying a rule; clauses are indexed from 0 in the matrix and literals are indexed from 0 within clauses. For example, `left_branch`$(2, 0, 2)$ denotes that there is an Extension rule with C' a copy of $\{P, \neg Q(x)\}$ and $L' = P$. The third number denotes the depth within the proof. A short Prolog program reads this certificate and verifies that the rule has been correctly applied at each stage, using the built-in unification mechanism to build the substitution.

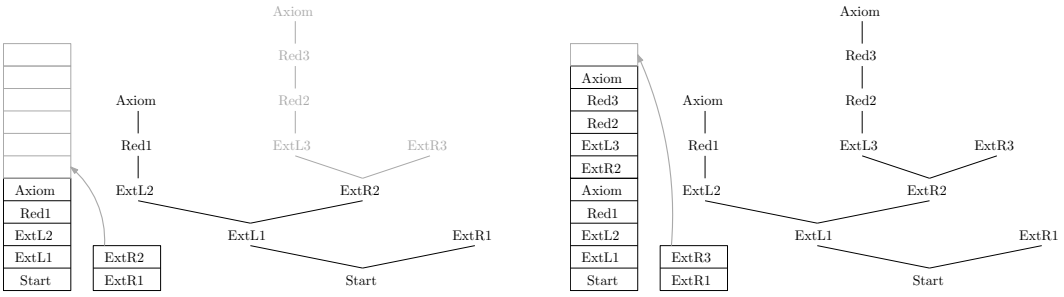
4. Implementation: some technical details

The requirements for compiling Connect++ are light. Aside from a C++ compiler, it uses the Boost libraries[3] for parsing TPTP and schedule files, and dealing with command-line parameters. LaTeX is needed for readable proofs and SWI Prolog [29] is needed for verifying proofs.

Connect++ exploits the structure of the connection calculus by using an optimized representation for variables and terms. As substitutions apply to an entire proof, any variable is only represented once, and terms using that variable do so via a pointer. Any substitution applied to a variable therefore takes effect everywhere the variable appears, in constant time, and backtracking (removing a substitution) is equally fast and straightforward. Fresh variables are recycled where possible. Terms are constructed using pointers to subterms; subterms are shared and no copy of an existing subterm is ever made. This is supported by an index: new (sub)terms are added to the index and only stored of not already present; if already present then a pointer to the existing copy is provided. Literals are straightforward identifiers paired with a list of pointers to terms. Clauses are lists of literals. The matrix is indexed to allow fast lookup of which clauses contain a literal, and the position of that literal in each relevant clause. This aids fast identification of possible Extensions.

As one aim of Connect++ is to provide flexibility in the proof search, it avoids the use of recursion in favour of an iterative approach using a pair of stacks. Figure 5 illustrates this. The proof is built in the left stack, and as left premises of Extensions are explored first, the right stack is used to store the currently outstanding right premises. Stack items store C, $\mathbb{P}$ and $\mathbb{L}$, a

[3] https://www.boost.org/



(a) Early in a proof search—the shaded area is not yet explored.

(b) After generating the shaded area, explore `ExtR3`.

Figure 5: Proof search arranged around two stacks. The proof itself is built on the left-hand stack, while the right-hand stack maintains details of the currently outstanding right premises for Extensions.

substitution, and a list of all actions (applications of the proof rules) that can be used to further extend the proof at this point. This structure is manipulated iteratively, and as at any point there is direct access to the list of possible actions stored in each stack item, there is great flexibility in directing the search—the action list in each stack item can arbitrarily be re-ordered, added to (increasing the degree of backtracking) or reduced (restricting the degree of backtracking) *while* a proof is in progress.

Planned developments to Connect++ include: (1) completion of clause translation, including definitional clausal form, from TPTP first-order format; (2) enumeration of different proofs for a single problem as a means of generating training data; (3) implementation of the leanCop backtracking heuristic; (4) addition of other heuristics for connection provers; (5) experiments with new heuristics; (6) addition of further command-line options, and addition of all options to the schedule language; (7) implementation of readable proofs similar to those produced by leanCop version 2.1; (8) implementation of any forthcoming standard for proof certificates; and (9) implementation of better approaches to equality.

5. Evaluation

To illustrate how Connect++ facilitates new experiments, we use it to provide a brief comparison with leanCop while employing a much more aggressive backtracking restriction heuristic. Recall that a principal literal is *solved* when used by a Reduction or Lemmata, or when used to start an Extension which then leads to a solved left premis. leanCop then reduces the backtracking employed in the proof search, at the point the rule was applied. What might happen if we removed *all* possibilities at those points? This is straightforward in Connect++ as for Reduction and Lemmata we have a pointer `si` to the stack item, which in turn contains a list `actions` of all the ways the proof can still be extended. Thus

```
if (params::limit_bt_reductions)
    si->actions.clear();
```

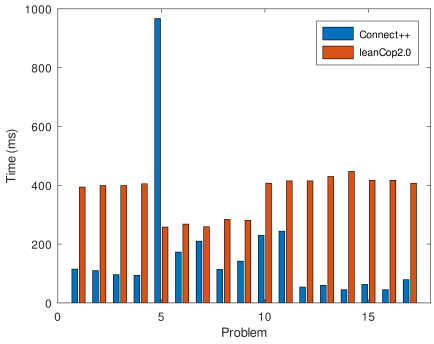


Figure 6: Running times for leanCop and Connect++ for the subset of AGT solved by both.

(and similarly for Lemmata) is all that is needed. Extensions are only slightly more involved:

```
if (params::limit_bt_extensions)
    stack[si->bt_restriction_index].actions.clear();
```

The AGT problems from TPTP version 8.0.0 were converted to formats readable by Connect++ and leanCop, with equality axioms added, using `tptp2X`. leanCop version 2.0 was used with ECLiPSe version 5.10 #147, and its schedule was edited to bring it closer to that for Connect++, by removing the variations involving definitional clausal form. Each prover was run on all problems using the University of Cambridge High-Performance Computing Facility.[4] Of the 52 AGT problems, both systems solve the same subset, with the exception of a single problem solved by leanCop and not by Connect++. Figure 6 shows the running times for the problems solved by both systems. Clearly Connect++ is considerably faster in all but 1 problem. While this is obviously not a fair or direct comparison, because we don't know whether the difference arises from the speed of the implementation or the different backtracking heuristic,[5] it seems at least of note that even with a hugely aggressive backtracking restriction Connect++ solves essentially the same problems. This also demonstrates that Connect++ can be used to support the kind of experimentation it was developed for.

Acknowledgments

This work was performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (www.csd3.cam.ac.uk), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/T022159/1), and DiRAC funding from the Science and Technology Facilities Council (www.dirac.ac.uk). For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission.

[4]This provided Ice Lake CPUs each with 3380MiB of memory, interconnected using Mellanox HDR200 Infiniband, and one CPU per problem.

[5]And note that Connect++ is still applying its default schedule.

References

[1] S. B. Holden, Machine Learning for Automated Theorem Proving: Learning to Solve SAT and QSAT, volume 14 of *Foundations and Trends® in Machine Learning*, now publishers, Boston, Delft, 2021. URL: https://www.nowpublishers.com/article/Details/MAL-081. doi:10.1561/2200000081.

[2] N. Eén, N. Sörensson, An extensible SAT-solver, in: E. Giunchiglia, A. Tacchella (Eds.), Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing, volume 2919 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 502–518.

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

[4] A. Riazanov, A. Voronkov, The design and implementation of VAMPIRE, AI Communications 15 (2002) 91–110.

[5] J. Otten, W. Bibel, leanCoP: lean connection-based theorem proving, Journal of Symbolic Computation 36 (2003) 139–161.

[6] J. Otten, Restricting backtracking in connection calculi, AI Communications 23 (2010) 159–182.

[7] C. Kaliszyk, Efficient low-level connection tableaux, in: H. D. Nivelle (Ed.), 24th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2015), volume 9323 of *Lecture Notes in Artificial Intelligence*, Springer, 2015, pp. 102–111.

[8] C. Kaliszyk, J. Urban, J. Vyskočil, Certified connection tableaux proofs for HOL Light and TPTP, in: Proceedings of the 2015 Conference on Certified Programs and Proofs (CPP), Association for Computing Machinery, New York, NY, United States, 2015, pp. 59–66.

[9] M. Färber, C. Kaliszyk, J. Urban, Machine learning guidance for connection tableax, Journal of Automated Reasoning 65 (2021) 287–320.

[10] M. Färber, cop: Highly efficient first-order connection proving, 2021. URL: https://lib.rs/crates/cop.

[11] F. Rømming, J. Otten, S. B. Holden, Markov decision processes for classical, intuitionistic, and modal connection calculi, 2023. Submitted to the International Workshop on Automated Reasoning with Connection Calculi (AReCCA) 2023.

[12] G. Prusak, C. Kaliszyk, Lazy paramodulation in practice, in: B. Konev, C. Schon, A. Steen (Eds.), Proceedings of the Workshop on Practical Aspects of Automated Reasoning (PAAR), volume 3201 of *CEUR Workshop Proceedings*, 2022.

[13] V. Toth, Reinforcement Learning for Automated Theorem Proving, Master's thesis, University of Cambridge, Department of Computer Science and Technology, The Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, 2021.

[14] M. Färber, A curiously effective backtracking strategy for connection tableaux, 2021. ArXiv:2106.13722v1 [cs.LO].

[15] T. Raths, J. Otten, randoCoP: Randomizing the proof search order in the connection calculus, in: B. Konev, R. A. Schmidt, S. Schulz (Eds.), Proceedings of the First International Workshop on Practical Aspects of Automated Reasoning (PAAR), volume 373 of *CEUR Workshop Proceedings*, 2008.

[16] R. S. Sutton, A. G. Barto, Reinforcement Learning: An Introduction, 2nd edition ed., MIT Press, 2018.

[17] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshag, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of Monte Carlo tree search methods, IEEE Transactions on Computational Intelligence and AI in Games 4 (2012) 1–43.

[18] B. E. Oliver, J. Otten, Equality preprocessing in connection calculi, in: P. Fontaine, K. Korovin, I. S. Kotsireas, P. Rümmer, S. Tourret (Eds.), Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning (PAAR) and the 5th Satisfiability Checking and Symbolic Computation Workshop (SC-Square) Workshop, volume 2752 of *CEUR Workshop Proceedings*, 2020.

[19] G. Prusak, C. Kaliszyk, Lazy paramodulation in practice, in: B. Konev, C. Schon, A. Steen (Eds.), Proceedings of the 8th Workshop on Practical Aspects of Automated Reasoning (PAAR), volume 3201 of *CEUR Workshop Proceedings*, 2022.

[20] G. Sutcliffe, C. Suttner, The TPTP problem library for automated theorem proving, 2023. URL: https://tptp.org/.

[21] F. Hutter, H. H. Hoos, K. Leyton-Brown, T. Stützle, Paramils: An automatic algorithm configuration framework, Journal of Artificial Intelligence Research 36 (2009) 267–306.

[22] F. Hutter, H. H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: C. A. C. Coello (Ed.), Proceedings of the 5th International Conference on Learning and Intelligent Optimization (LION), volume 6683 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 507–523.

[23] C. Ansótegui, M. Sellmann, K. Tierney, A gender-based genetic algorithm for the automatic configuration of algorithms, in: I. P. Gent (Ed.), Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP), volume 5732 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 142–157.

[24] C. Mangla, S. B. Holden, L. Paulson, Bayesian optimisation of solver parameters in CBMC, in: F. Bobot, T. Weber (Eds.), Proceedings of the 18th International Workshop on Satisfiability Modulo Theories (SMT), volume 2854, CEUR Workshop Proceedings, 2020, pp. 37–47. URL: http://ceur-ws.org/Vol-2854/.

[25] C. Mangla, S. B. Holden, L. Paulson, Bayesian ranking for strategy scheduling in automated theorem provers, in: J. Blanchette, L. Kovács, D. Pattinson (Eds.), Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR), volume 13385 of *Lecture Notes in Artificial Intelligence*, Springer, 2022, pp. 559–577. URL: https://link.springer.com/chapter/10.1007/978-3-031-10769-6_33. doi:https://doi.org/10.1007/978-3-031-10769-6.

[26] J. Otten, G. Sutcliffe, Using the TPTP language for representing derivations in tableau and connection calculus, volume 9 of *EPiC Series*, 2012, pp. 95–105.

[27] M. Färber, C. Kaliszyk, J. Urban, Machine learning guidance and proof certification for connection tableaux, 2018. ArXiv:1805.03107v3 [cs.LO].

[28] J. Otten, S. B. Holden, A standardized syntax for connection proofs, 2023. Submitted to the International Workshop on Automated Reasoning with Connection Calculi (AReCCA) 2023.

[29] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, SWI-Prolog, Theory and Practice of Logic Programming 12 (2011) 67–96.