Reset Early, Reset Often, Eliminate Models

Martin S. Fixman[1], Fredrik Rømming[2], and Sean B. Holden[3]

University of Cambridge,
United Kingdom

[1]msf43@cl.cam.ac.uk [2]fr409@cl.cam.ac.uk [3]sbh11@cl.cam.ac.uk

Abstract

Connection-tableau provers typically construct closed tableaux by backtracking search. This gives lightweight proof procedures, but their low-memory search state leaves little derived information to reuse after a failed attempt. SATCoP addresses this by retaining grounded tableau clauses in an incremental SAT solver, allowing the prover to refute the problem once the accumulated ground clauses become propositionally unsatisfiable. We introduce SATResetCoP, which relies more directly on this persistent SAT state by not backtracking on unproductive tableaux and instead starting new ones to generate further ground clauses for the SAT solver. On TPTP, SATResetCoP improves over our leanCoP-style baseline by 54% and our SATCoP-style baseline by 18%. On the bushy MPTP2078 benchmark, it solves 616 problems, a 49% improvement over our leanCoP-style baseline and a 12% improvement over our SATCoP-style baseline. These results suggest that changing tableau guidance to accumulate useful ground clauses faster improves the performance of the prover.

1 Introduction

Connection-tableau provers prove first-order problems by constructing closed tableaux from sound tableau-construction rules [8]. Implementations usually turn this construction into backtracking search over applications of those rules. This gives a lightweight and goal-directed proof procedure, but its small search state means that failed partial tableaux leave little derived information for later attempts to reuse.

Much practical work on connection-tableau proving has therefore focused on controlling backtracking, and learning how to restrict it. The line from semantic tableaux and model elimination to leanTAP and leanCoP showed that goal-directed tableau search could be implemented in small, effective proof procedures [4, 9, 3, 13].

leanCoP uses Prolog's SLD resolution for depth-first connection-tableau search with iterative deepening, and leanCoP 2.0 obtains substantial speedups by restricting this backtracking with Prolog cuts at the cost of completeness [11]. meanCoP refines this pruning strategy by separating cuts on reductions from cuts on extensions and by introducing exclusive cuts, which preserve more alternatives than the inclusive cuts used by leanCoP [7].

SATCoP addresses the same weakness from a different direction [18]. During tableau construction, it grounds encountered clauses and stores them in an incremental SAT solver. If the accumulated ground clauses become propositionally unsatisfiable, the original first-order problem can be proved before the active tableau has closed. We introduce SATResetCoP, which pushes this idea further: when the active tableau stops producing new ground clauses, it is reset, while the accumulated SAT solver state is kept. The tableau is therefore used as a generator of ground clauses, and the SAT solver becomes the persistent object from which a proof may ultimately be obtained.

We evaluate SATResetCoP on TPTP against leanCoP-style and SATCoP-style baselines implemented in a fully compatible backend. SATResetCoP solves 54% more problems than the

leanCoP-style baseline and 18% more than the SATCoP-style baseline, suggesting that once ground clauses are preserved in the SAT solver, starting a new tableau rather than continuing to backtrack on an unproductive one can be an effective way to generate the clauses needed for a proof.

2 Preliminaries

2.1 Connection Calculus and Tableau Search

Connection tableaux and model elimination give a goal-directed alternative to the saturation style used by provers such as Vampire [19]. Instead of deriving clauses from a global clause set until a contradiction is found [2, 22], connection calculi try to close a tableau by connecting goal literals to complementary literals, either on the current path or in a fresh copy of an input clause [8, 14].

The standard clausal connection tableau has three proof-building rules [8, 15].

Start Choose an input clause and make its literals open goals.

Extension Open a branch on a clause containing a complementary literal, and add the remaining literals as new subgoals. This may require a substitution of variables with terms.

Reduction Close the current goal against a complementary literal already on the current branch. This may also require a substitution.

The rules are sound, and with the possibility of backtracking over rule applications the calculus is complete. The calculus is not confluent: a locally sound connection can bind variables in a way that prevents later branches from closing [8]. Backtracking is therefore part of the operational search procedure, not a proof-building rule.

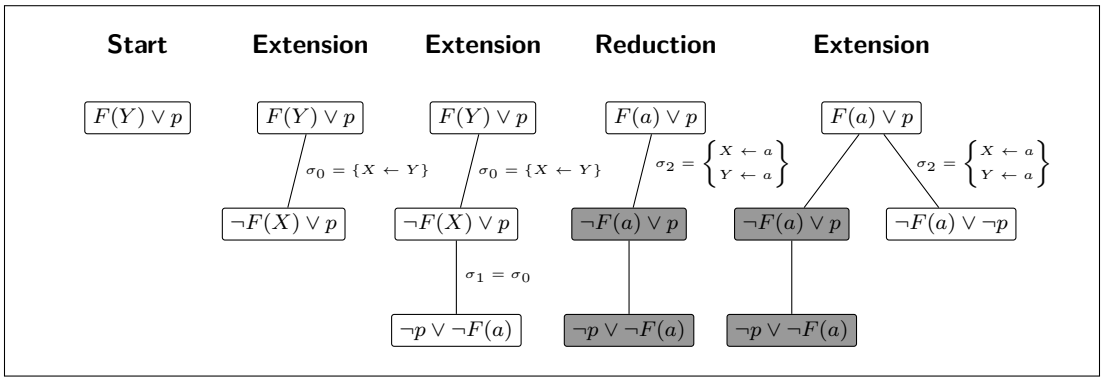


Figure 1: Drawing convention for the tableau diagrams, illustrated on steps from SYN052-1.p [16]. Each box is a clause instance in the tableau. An edge records the connection that introduced a child clause, with the edge label showing the resulting substitution. Grey boxes mark clauses whose selected open literal has been closed. A Reduction closes against a complementary literal already on the current path, so it adds no new child.

2.2 Backtracking Control

Backtracking lets a prover recover from locally sound choices that later prevent tableau closure. In leanCoP-style provers, this search is controlled mainly by iterative deepening and Prolog cuts.

Iterative deepening bounds the tableau height and raises the bound when no proof is found. Cuts prune alternatives after selected reductions or extensions, reducing search but making the cut phase incomplete unless the surrounding schedule later explores less restricted search [11].

meanCoP refines this pruning mechanism by separating cuts on reductions from cuts on extensions and by introducing exclusive cuts, which preserve more alternatives than the inclusive cuts used by leanCoP [7]. hopCoP takes a different route: instead of pruning alternatives directly, it learns constraints explaining why a search state is stuck, reducing repeated backtracking while preserving completeness at a fixed iterative-deepening bound [17].

2.3 SATCoP and Persistent Ground Clauses

SATCoP maintains an incremental SAT solver alongside tableau search [18]. During tableau search, clauses appearing in the tableau are heuristically grounded and added to the SAT solver. If the accumulated ground clauses are propositionally unsatisfiable, proof search can stop even if the active first-order tableau has not closed.

The soundness condition is that every clause sent to the SAT solver must be a ground instance of a clause generated by the tableau. Remaining tableau variables are therefore mapped to ground terms before a clause is added. Rawson and Reger use the grounding heuristic inspired by a similar suggestion by Schulz [21]: choose the most frequent constant in the conjecture, falling back to a fresh constant when no conjecture constant is available [18].

Later SATCoP variants also use the SAT solver to guide tableau search. Rawson and Reger describe a literal-selection refinement that prefers extension choices whose selected literal is true in the current SAT assignment [18, § 9.1].

3 Research Vehicle

3.1 SATResetCoP Specification

We propose a new tableau prover based on SATCoP, but guided by the goal of producing the ground clauses needed for a SAT refutation rather than using the leanCoP heuristics that attempt to close the current tableau.

The resulting strategy is simple: aggressively reset the tree when the SAT solver stops receiving new ground clauses. This adds a broader set of ground clauses to the solver in less time, allowing the SAT solver to reach unsatisfiability earlier.

SATResetCoP uses many of the properties of SATCoP described in the original paper and in its "Future Research" section [18, § 9.1], but changes the tableau guidance to prioritise aggressive resets.

Like SATCoP, the implementation keeps a persistent SAT solver alongside the active tableau search. Ground clauses accumulated in earlier tableau attempts remain available in later attempts, and new clauses are added incrementally to the same solver instance. This keeps the approach sound but incomplete. However, we expect it to certify unsatisfiability of problems faster and in fewer steps.

SATResetCoP randomises the Start and Extension steps in the same way as SATCoP. When starting a new tree, the implementation shuffles the possible starts and the literals inside each start clause. When several extensions are possible from the leftmost open node, it shuffles the extension choices and the literal order inside the copied extension clause.

A new virtual operation, Reset, clears the tableau, the current substitution, and the proof sequence, and starts a new tableau attempt with a specified depth bound. It does not clear the

persistent SAT solver: every reset starts from an empty tree, while the SAT solver keeps the ground information produced by earlier attempts. An example of how this operation works can be seen in Figure 2.

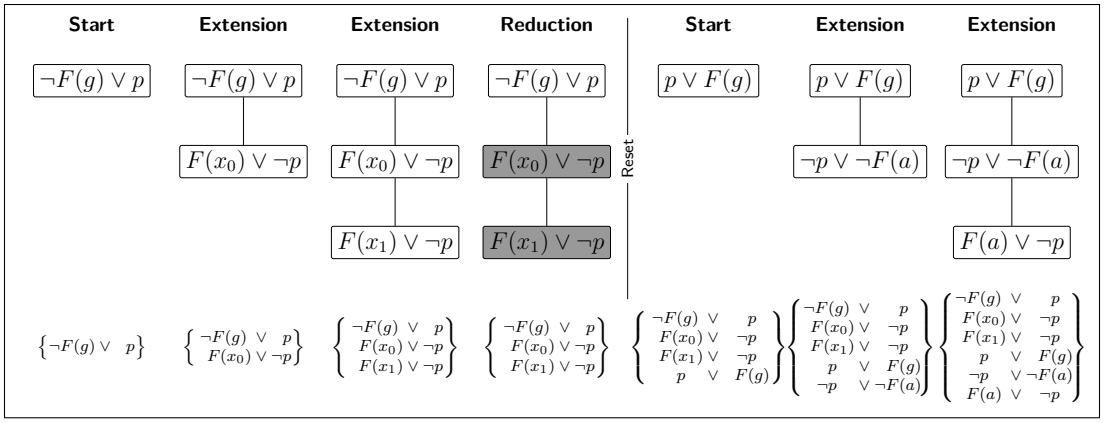


Figure 2: Example of SATResetCoP on SYN052-1.p. SATCoP and previous provers would backtrack after closing the branch with $F(x_1) \vee \neg p$; instead, SATResetCoP resets into a new tree. This accumulates the ground clauses that form the UNSAT core shown at the bottom faster than in SATCoP, allowing the SAT solver to reach unsatisfiability earlier. Substitutions are omitted for simplicity. At the end of the last extension, the accumulated ground clauses are unsatisfiable and the entire tableau prover can report UNSAT.

The grounded clauses used by SATResetCoP come from clauses that have appeared in the tableau, after applying the tableau substitution accumulated so far. Start clauses are added to the SAT state when the run is initialised. After successful Extension and Reduction steps, the implementation grounds the clauses currently present in the tableau and records the genuinely new ones, which are later added to the SAT solver.

The grounding scheme follows the soundness argument of Section 2.3: the SAT solver receives only ground instances of clauses generated by the tableau. Ground terms are kept as they are, and variables assigned by the current tableau substitution are replaced by their assigned terms. Only tableau variables that remain unassigned need to be replaced by a ground term. Our Python SATCoP-style variants use the simpler fallback-style choice throughout, mapping every remaining tableau variable to the same fresh artificial constant.

The reset policy does not change this soundness argument. Cuts and resets only affect which ground instances are discovered, and hence the completeness and performance of the search, not the soundness of a SAT refutation once found.

The aggressive reset-on-backtrack policy is the main implementation difference between SATResetCoP and SATCoP. At a dead end, the prover first asks whether the current tableau attempt produced any new ground clauses that have not yet been added to the SAT solver. If it did, those clauses are flushed to the SAT solver and the prover resets at the same depth. This gives the next tableau attempt the same depth bound, but a larger SAT state and a fresh set of grounded clauses. If the attempt did not produce any new ground clauses, the prover treats the current depth as exhausted and resets at the next depth instead of spending more steps on local backtracking. The process stops as soon as the accumulated ground clauses become propositionally unsatisfiable.

Since SATResetCoP resets the tree aggressively, it only increases the depth bound when the accumulated ground clauses have not been updated since the last Reset. This prevents trees from growing out of control, while still trying to maximise the set of accumulated ground clauses.

```
def solve_tableau(start_clauses):
  solver := incremental_sat_solver()
  solver.add(ground(start_clauses))
  pending := empty_set

  max_depth := 1
  tableau := empty_tableau(max_depth)

  while solve_sat(solver) != UNSAT:
    model := current_model(solver)
    actions := legal_actions(tableau)

    if any Start, Extension, Reduction in actions:
      model_rank(a) :=
        2 if a is Start
        0 if selected_literal(a) is true in model
        1 if selected_literal(a) is unknown in model
        3 if selected_literal(a) is false in model

      next_action = argmin(actions, key = model_rank)
      tableau_step(next_action)

      if next_action is Reduction or Extension:
        pending.add(ground(tableau))

    else:
      if pending is not empty:
        solver.add_all(pending)
        pending := empty_set
        tableau := empty_tableau(max_depth)
      else:
        max_depth := max_depth + 1
        tableau := empty_tableau(max_depth)
```

Listing 1: Pseudocode of the SATResetCoP control policy.

Pseudocode for this implementation can be found in Listing 1.

3.2 Implementation and benchmarking

We implement SATResetCoP using the Connections library [20], which provides primitives for implementing connection-tableau provers, together with the incremental SAT solver CaDiCaL [5]. We also implement the control strategies leanCoP^{Con}, SATCoP$_0{}^{\text{Con}}$, and SATCoP^{Con} using the same backend, as mirrors of LeanCoP, SATCoP$_0$, and SATCoP. Using a common backend makes the strategies easier to compare by keeping the tableau representation, rule applications, grounding interface, SAT interface, and step accounting fixed.

The source code for this implementation is available online,[1] and is released under the GNU Lesser General Public Licence [6].

We evaluate these 3 control strategies along with the test strategy SATResetCoP on 7815 CNF problems and 8779 FOF problems from the Thousands of Problems for Theorem Provers (TPTP) library [23]. The TPTP problems cover all 50 TPTP categories containing at least one CNF or FOF problem.

These benchmark sets provide a varied collection of problems for first-order provers. TPTP also provides declared statuses for its problems, which are a useful sanity check for false unsatisfiability results.

[1] https://github.com/mfixman/resetsatcop

We additionally test these strategies in the Bushy MPTP2078 library, which contains 2078 problems exported from the Mizar Mathematical library [1]. This benchmark is useful because its problems share a large formal-mathematical vocabulary and test whether the strategies remain effective on related large-theory problems rather than only on the more heterogeneous TPTP collection.

4 Experimental Evaluation

4.1 Experiment Design

As a leanCoP-style baseline we use leanCoP^{Con} [11]. This is a Python version of the leanCoP Prolog implementation, with the two recommended predicates used in the first step of the schedule of LeanCoP 2.0 [11, § 5.5].

`cut` Choose one extension or reduction and commit to it, similarly to Prolog's `!` operator.

`comp(7)` The prover increases the depth bound by 1 whenever the current tree saturates, until reaching maximum height 7. Afterwards, the height limit is removed.

These simple policies tend to be very effective in tableau provers [12].

We add Python implementations in the connection prover of the two versions of SATCoP presented by Rawson and Reger [18]: SATCoP$_0$^{Con}, corresponding to SATCoP$_0$, and SATCoP^{Con}, corresponding to the version of SATCoP with the improvements in the "Future Research" section that was submitted to CASC. These implementations use similar techniques to the original provers, and since they use the same backend as SATResetCoP allow for a fair comparison.

An overview of these strategies is shown in Table 1.

Strategy	Explanation	Cut	SAT Refutation	SAT Guidance	Reset on Backtrack
leanCoP^{Con}	leanCoP with initial set of predicates.	✓			
SATCoP$_0$^{Con}	Based on the initial implementation of SATCoP [18].	✓	✓		
SATCoP^{Con}	Improvements on SATCoP [18, § 9.1].	✓	✓	✓	
SATResetCoP	Novel strategy, specified in Section 3.1.	✓	✓	✓	✓

Table 1: Control features of the baseline strategies and SATResetCoP.

Each problem-strategy pair is run until the problem is solved, or until either 120 seconds or 1 million steps pass, on a single core of the University of Cambridge CSD3 Cluster with dual-socket Intel Xeon Platinum 8368Q CPUs at 2.60 GHz, with 38 cores per socket and one thread per core, and 3370 MiB RAM per core [24]. We thank the Computer Science Lab of the University of Cambridge for providing this computer.

4.2 Results

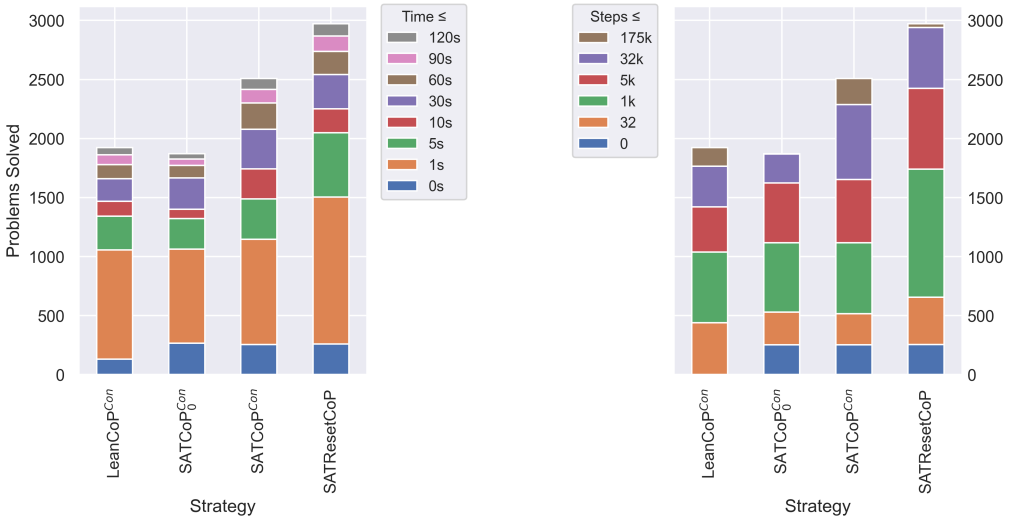
4.2.1 Individual Results

SATResetCoP considerably improves on the number of problems solved by both $\mathsf{SATCoP_0}^{\mathsf{Con}}$ and $\mathsf{SATCoP}^{\mathsf{Con}}$ under the 120-second time limit, as shown in Table 2. The results are consistent with the claim that the SAT solver reaches unsatisfiability in less time and fewer Start, Extension, and Reduction steps than in the previous strategies, as seen in Figures 3a and 3b.

The improvement is visible in both TPTP problem formats and on the MPTP2078-derived problems. SATResetCoP solves 529 more problems than $\mathsf{SATCoP}^{\mathsf{Con}}$, with a net gain of 206 TPTP CNF problems, 256 TPTP FOF problems, and 67 Mizar Bushy problems. Compared with $\mathsf{SATCoP_0}^{\mathsf{Con}}$, it solves 1395 more problems, an increase of 63.6%, and compared with $\mathsf{leanCoP}^{\mathsf{Con}}$ it solves 1251 more problems, an increase of 53.6%.

	TPTP CNF		TPTP FOF		MPTP2078		Total	
strategy	Number	%	Number	%	Number	%	Number	%
$\mathsf{leanCoP}^{\mathsf{Con}}$	723	9.3	1200	13.7	413	19.9	2336	12.5
$\mathsf{SATCoP_0}^{\mathsf{Con}}$	804	10.3	1067	12.2	321	15.4	2192	11.7
$\mathsf{SATCoP}^{\mathsf{Con}}$	1067	13.7	1442	16.4	549	26.4	3058	16.4
SATResetCoP	1273	16.3	1698	19.3	616	29.6	3587	19.2

Table 2: Number and percentage of TPTP CNF, TPTP FOF, and MPTP2078-derived problems solved within the time limit by each strategy. The total column combines all three benchmark sets.



(a) Problems solved, stacked by time in seconds.

(b) Problems solved, stacked by number of steps. SAT-based strategies can solve a problem in 0 steps if the initial ground SAT clause set is already unsatisfiable.

Figure 3: Results of running the strategies on all TPTP problems under a time limit of 120 seconds and step limit of 1 million steps. SATResetCoP solves more problems in fewer steps than in previous strategies, and also faster except for the niche category of $\leq 1s$ solves.

4.2.2 Comparative Results

The tableau guidance in SATResetCoP allows it to solve several problems left unsolved by strategies without the same mechanism for accumulating ground clauses, and even by $\mathsf{SATCoP_0}^{\mathsf{Con}}$ and $\mathsf{SATCoP}^{\mathsf{Con}}$, which use less aggressive guidance. A comparison of newly solved and newly unsolved problems can be seen in Table 3.

This comparison is useful because the total number of solved problems hides changes in which problems are solved. On TPTP, SATResetCoP solves 598 problems that $\mathsf{SATCoP}^{\mathsf{Con}}$ does not, but loses 136 problems solved by $\mathsf{SATCoP}^{\mathsf{Con}}$. The reset policy is therefore changing the search distribution rather than just speeding up the same proof attempts.

Strategy	Newly Solved	Newly Unsolved	Total Solved
$\mathsf{leanCoP}^{\mathsf{Con}}$	—	—	1923
$\mathsf{SATCoP_0}^{\mathsf{Con}}$	327	379	1871
$\mathsf{SATCoP}^{\mathsf{Con}}$	661	23	2509
SATResetCoP	598	136	2971

Table 3: Problems from the TPTP dataset newly solved and newly unsolved by each strategy compared with the previous strategy in the table, together with the total number of solved problems. The reset policy changes which parts of the search space are explored: SATResetCoP solves 598 TPTP problems missed by $\mathsf{SATCoP}^{\mathsf{Con}}$, but misses 136 problems solved by $\mathsf{SATCoP}^{\mathsf{Con}}$. This shows that aggressive resetting is not only a speedup of the same proof attempts, but a different search tradeoff.

The same tradeoff appears on the bushy MPTP2078 benchmark. SATResetCoP solves 111 MPTP2078 problems that $\mathsf{SATCoP}^{\mathsf{Con}}$ does not, while $\mathsf{SATCoP}^{\mathsf{Con}}$ solves 44 problems missed by SATResetCoP. The resulting net gain of 67 problems is smaller than on TPTP, but the pattern is similar. Resets help when useful ground instances appear quickly, and hurt when the useful branch requires longer local tableau search.

This is the expected behaviour for a policy that treats the tableau mainly as a source of ground clauses. When the early branches already expose the clauses needed by the SAT solver, resetting avoids spending many steps on similar local backtracking. When the useful clauses only appear after a longer sequence of tableau choices, the same reset policy can discard the partial search state too early.

Figure 4 shows the number of problems solved by one strategy but not by another.

The improvement between SATResetCoP and the best previous experimental strategy, our $\mathsf{SATCoP}^{\mathsf{Con}}$ implementation of SATCoP, is especially visible in TPTP categories where the accumulated ground clauses become propositionally unsatisfiable quickly, such as SYN, SWC, and SWV. This can be seen in Table 4.

As expected, this strategy also solves problems in fewer steps, as can be seen in Figure 5 for TPTP problems and in Figure 6 for the Mizar MPTP2078 dataset.

Strategy	SYN	SWC	SWV	GEO	NUM	Rest	Total
$\mathsf{leanCoP}^{\mathsf{Con}}$	486	14	163	216	86	958	1923
$\mathsf{SATCoP_0}^{\mathsf{Con}}$	459	2	161	177	90	982	1871
$\mathsf{SATCoP}^{\mathsf{Con}}$	549	11	173	260	151	1365	2509
SATResetCoP	672	64	215	293	179	1548	2971

Table 4: Number of problems solved, per category, for each strategy.

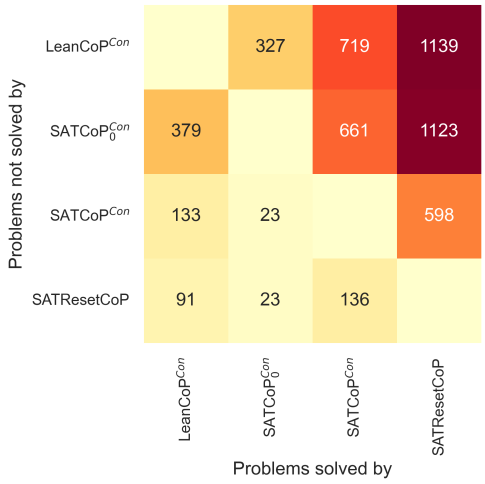


Figure 4: Pairwise comparison of strategy results: the cell in column y and row x shows the number of problems solved by strategy y, but not by strategy x. For example, SATResetCoP solved 1123 TPTP problems that $\mathsf{SATCoP_0}^{\mathsf{Con}}$ did not.

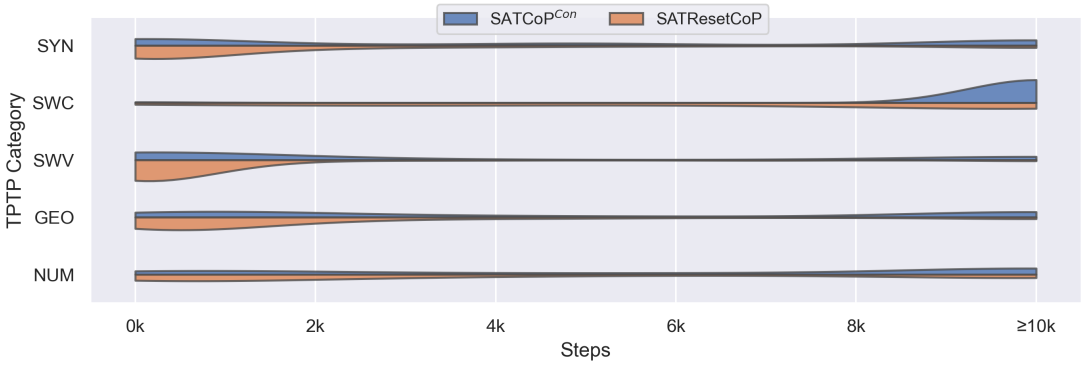


Figure 5: Density of problems solved at each step count by $\mathsf{SATCoP}^{\mathsf{Con}}$ and SATResetCoP on the TPTP categories SYN, SWC, SWV, GEO, and NUM. The area of each violinplot is the same; besides solving more problems in each category, SATResetCoP solves many problems in fewer steps. Unsolved problems are collected in the $\geq 10k$ category on the right.

Figure 6: Density of problems solved at each step count by $\mathsf{SATCoP}^{\mathsf{Con}}$ and SATResetCoP on the Mizar Bushy MPTP2078 dataset. Just like in TPTP, SATResetCoP solves more problems in fewer steps. Unsolved problems are collected in the $\geq 10k$ category on the right.

4.2.3 Satisfiability Sanity Check

The TPTP library includes declared statuses for its problems [23]. While this does not demonstrate soundness, it provides a useful check that the strategy has not become unsound by mistake.

As a sanity check, we group the SATResetCoP results by TPTP declared status in Table 5. The results are consistent with what we would expect if the prover were sound: it does not find unsatisfiability ("Solved") in any problem marked satisfiable or countersatisfiable. It also does not solve any unknown or open problems, which would require solutions to be checked closely to ensure that they were not caused by an incorrect prover.

	FOF		CNF	
Declared status	Unsolved	Solved	Unsolved	Solved
Satisfiable	358	0	1095	0
CounterSatisfiable	746	0	0	0
Unknown	139	0	745	0
Open	4	0	28	0
Unsatisfiable	110	66	4674	1273
ContradictoryAxioms	493	36	0	0
Theorem	5231	1596	0	0

Table 5: Number of problems solved and unsolved by SATResetCoP, grouped by TPTP declared status. No satisfiable or countersatisfiable problem was incorrectly proven to be unsatisfiable.

5 Conclusion

The experiments on SATResetCoP show that replacing local backtracking with resets can improve tableau performance when ground clauses are retained.

Section 4.2.2 shows that, on a 1 million step and 120 second time limit, SATResetCoP solves 2971 problems. This is 462 more problems than SATCoP^{Con}, an 18% increase, and 1048 more problems than leanCoP^{Con}, a 54% increase. Pairwise, SATResetCoP solves 598 problems that SATCoP^{Con} does not, and 1139 problems that leanCoP^{Con} does not.

The improvement is not uniform across TPTP. It is strongest in categories where useful ground clauses accumulate quickly: SATResetCoP solves 672 SYN problems, 64 SWC problems, and 215 SWV problems, compared with 549, 11, and 173 for SATCoP^{Con}. It also improves on GEO and NUM, solving 293 and 179 problems respectively.

The newly unsolved problems show the other side of this tradeoff. These are cases where the accumulated ground clauses do not become propositionally unsatisfiable quickly enough, and the aggressive reset policy can discard tableau branches that a less aggressive prover eventually closes.

This suggests that accumulating ground clauses is a useful objective for connection tableaux, but not a replacement for all tableau control. The most promising direction is therefore not a single fixed reset policy, but a schedule or controller that can use aggressive resets when ground instances are informative and fall back to ordinary connection search when they are not.

6 Future Work

6.1 Schedule with different provers

The reset policy is deliberately incomplete, and the newly unsolved problems show cases where less aggressive tableau search finds proofs that SATResetCoP misses. A natural next step is to use SATResetCoP as one component of a schedule, following leanCoP 2.1 and the strategy-scheduling work of Mangla [11, 10]. leanCoP 2.1, for example, obtains stronger performance by scheduling several strategy variants with different options and resource limits, rather than relying on a single fixed configuration [12].

The simplest schedule would run SATResetCoP first as a fast ground-clause generator, then switch to a more conventional leanCoP- or SATCoP-style strategy after a fixed time or step budget. More adaptive schedules could use the SAT state itself, for example whether new ground clauses are still being produced or recent resets have stopped changing the accumulated clause set.

6.2 Further Experiments

The current implementation uses CaDiCaL as its incremental SAT solver. Future experiments should test whether the results are stable under different SAT solvers, solver restart policies, and incremental interfaces. This matters because SATResetCoP relies on the solver not only for the final unsatisfiability result, but also for the model information used to guide tableau extensions.

It would also be useful to vary the reset policy itself. The present policy resets aggressively when the current tableau stops producing new ground clauses, but the newly unsolved problems suggest that some domains need longer tableau branches before the useful ground instances appear. A broader experiment could compare fixed reset budgets, category-specific schedules, and policies that delay resets when recent SAT models still give useful literal guidance.

6.3 CASC Submission

SATResetCoP will also be packaged as a CASC entrant. This will test whether the improvements seen here survive the standard competition setting, including the official hardware, time limits, problem selection, and input requirements. It will also make the comparison with earlier connection-tableau systems more direct, since both leanCoP and SATCoP have been evaluated through CASC.

Preparing a CASC version requires more than submitting the current research prototype. The prover should expose a stable command-line interface, handle TPTP input robustly, report SZS statuses consistently, and include a reproducible installation path for the SAT backend. This would turn the current experimental result into a system that can be compared against the broader ATP ecosystem.

7 Acknowledgements

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).

References

[1] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *Journal of Automated Reasoning*, 52(2):191–213, 2014.

[2] Leo Bachmair, Harald Ganzinger, David McAllester, and Christopher Lynch. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, pages 19–99. Elsevier, 2001.

[3] Bernhard Beckert and Joachim Posegga. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[4] Evert Willem Beth. *Semantic Entailment and Formal Derivability.* Mededelingen der Koninklijke Nederlandse Akademie van Wetenschappen, Afd. Letterkunde, Nieuwe Reeks. Noord-Hollandsche Uitgevers Maatschappij, 1955. Deel 18, no. 13.

[5] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In *Computer Aided Verification*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024.

[6] Free Software Foundation. GNU lesser general public license, version 3, 2007.

[7] Michael Färber. A Curiously Effective Backtracking Strategy for Connection Tableaux. In Jens Otten and Wolfgang Bibel, editors, *Automated Reasoning with Connection Calculi*, volume 3613 of *CEUR Workshop Proceedings*, pages 23–40. CEUR-WS.org, 2023.

[8] Reinhold Letz and Gernot Stenz. Model Elimination and Connection Tableau Procedures. In *Handbook of Automated Reasoning*, pages 2015–2114. Elsevier, 2001.

[9] Donald W. Loveland. Mechanical theorem-proving by model elimination. *Journal of the ACM*, 15(2):236–251, 1968.

[10] C. Mangla, S.B. Holden, and L.C. Paulson. Bayesian ranking for strategy scheduling in automated theorem provers. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 559–577, 2022.

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

[12] Jens Otten. 20 years of leanCoP — an overview of the provers. In *Proceedings of the 1st International Workshop on Automated Reasoning with Connection Calculi (AReCCa@TABLEAUX 2023)*, volume 3613 of *CEUR Workshop Proceedings*, 2023.

[13] Jens Otten and Wolfgang Bibel. leanCoP: Lean connection-based theorem proving. *Journal of Symbolic Computation*, 36(1–2):139–161, 2003.

[14] Jens Otten and Wolfgang Bibel. Advances in Connection-Based Automated Theorem Proving. In Mike Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog, editors, *Provably Correct Systems*, pages 211–241. Springer International Publishing, 2017.

[15] Jens Otten and Sean Holden. A Syntax for Connection Proofs. In Jens Otten and Wolfgang Bibel, editors, *Automated Reasoning with Connection Calculi*, volume 3613 of *CEUR Workshop Proceedings*, pages 84–94. CEUR-WS.org, 2023.

[16] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986.

[17] Michael Rawson, Clemens Eisenhofer, and Laura Kovács. Constraint Learning for Non-confluent Proof Search. In Gian Luca Pozzato and Tarmo Uustalu, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 15980, pages 103–119. Springer Nature Switzerland, 2026.

[18] Michael Rawson and Giles Reger. Eliminating Models During Model Elimination. In Anupam Das and Sara Negri, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 12842, pages 250–265. Springer International Publishing, 2021.

[19] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2–3):91–110, 2002.

[20] Fredrik Rømming, Jens Otten, and Sean B. Holden. Connections: Markov Decision Processes for Classical, Intuitionistic and Modal Connection Calculi. In *Proceedings of the 1st International Workshop on Automated Reasoning with Connection Calculi*, volume 3613 of *CEUR Workshop Proceedings*, pages 107–118, 2023.

[21] Stephan Schulz. Light-weight integration of SAT solving into first-order reasoners: First experiments. In Laura Kovács and Andrei Voronkov, editors, *Vampire 2017. Proceedings of the 4th Vampire Workshop*, volume 53 of *EPiC Series in Computing*, pages 9–19. EasyChair, 2018.

[22] Stephan Schulz and Martin Möhrmann. Performance of Clause Selection Heuristics for Saturation-Based Theorem Proving. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning*, volume 9706, pages 330–345. Springer International Publishing, 2016.

[23] Geoff Sutcliffe. The TPTP problem library and associated infrastructure: From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[24] University of Cambridge Research Computing Services. Ice Lake Nodes. https://docs.hpc.cam.ac.uk/hpc/user-guide/icelake.html, 2026. Accessed 13 May 2026.