




Constraint Learning for Non-Confluent Proof Search

Michael Rawson¹ , Clemens Eisenhofer² , and Laura Kovács² 

¹ University of Southampton, Southampton, UK
`michael@rawsons.uk`

² TU Wien, Vienna, Austria
`{clemens.eisenhofer,laura.kovacs}@tuwien.ac.at`

Abstract. Proof search in non-confluent tableau calculi, such as the connection tableau calculus, suffers from excess backtracking, but simple restrictions on backtracking are incomplete. We adopt *constraint learning* to reduce backtracking in the classical first-order connection calculus, while retaining completeness. An initial constraint learning language for connection-driven search is iteratively refined to greatly reduce backtracking in practice. The approach may be useful for proof search in other non-confluent tableau calculi.

Keywords: Constraint Learning · Connection Tableaux · Backjumping

1 Introduction

State-of-the-art methods for automated theorem proving are based on exhaustive search, using a *proof calculus* to explore the space of possible proofs. The search for proofs can be either backtracking or non-backtracking in nature. Backtracking search is required when the underlying calculus allows search to become “stuck” because of choices made previously in the search. These previous choices must be undone, and an alternative choice made, in order for the search to continue, which we call *backtracking*.

Some calculi do not require backtracking, such as confluent tableau calculi. Calculi like superposition and instance generation also fall into this category. Backtrack-free calculi are sometimes preferred and often enjoy theoretical advantages. However, in some cases, a non-confluent calculus is more practically effective, or is preferred for some other reason. We are therefore interested in *improving the behaviour of backtracking proof search*.

Backtracking search is not a problem inherently: state-of-the-art SAT solvers are uniformly based on backtracking procedures. Problems arise when the backtracking behaviour is pathological, backtracking too little, and trying to close the same goals again when the root cause of the dead end has not changed. We note in passing that backtracking *too much* would also cause problems. In SMT solving, for example, adding or removing theory literals to or from their respective decision procedures is a relatively expensive operation that should be avoided if possible, which is why smaller backtracking steps are preferred.

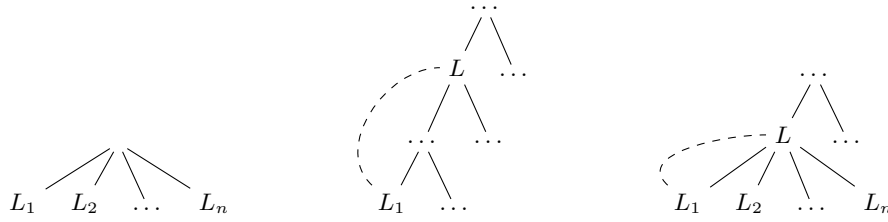


Fig. 1. The three inference rules of the clausal connection tableau calculus: *start*, *reduction*, and *extension*. In the *start* and *extension* rules, $C = L_1 \vee L_2 \vee \dots \vee L_n$ is a clause from the input set, with its variables renamed apart from the tableau. In the *reduction* and *extension* rules we require that $\sigma(\neg L) = \sigma(L_1)$, i.e. L and L_1 are *connected* (shown with dashed lines).

We address this here by adapting a technique called *constraint learning*³ from the constraint satisfaction community. During the search for a closed tableau, we sometimes arrive at a dead end where no further inferences are applicable in the tableau calculus. At this point, we analyse the *reason* that no inference is applicable and *learn* a constraint clause that prevents us from arriving at a similar tableau that is stuck for the same reason. The accumulating constraint database helps to guide the search towards more promising areas, or eventually shows that no closed tableau exists.

A potential source of confusion. Readers familiar with SAT solving, instance generation and/or refutational theorem proving may suspect that we are learning consequences of the input problem, and that if we derive an obviously unsatisfiable constraint like the empty clause or $0 = 1$, the input problem is unsatisfiable. This is *not* the case: we are learning constraints about the search space, and such constraints show that there is no closed tableau to be found (at a particular resource bound).

2 Background and Motivation

We assume familiarity with tableau methods [10] and classical first-order logic. The connection method [6], connection tableau calculus [21], model elimination [22], and the method of matings [2] are closely related proof search methods. We will present our work in the language of connection *tableaux*, given the audience. The connection refinement demands that any addition to an open branch is *connected* (contains a literal of opposite polarity) to the leaf literal, which produces a very strong goal-directed effect. This comes at the expense of confluence, even for propositional logic: choosing the wrong extension of the tableau can prevent closing the tableau.

³ better known as *clause learning* in the context of Boolean satisfiability

By lifting propositional connection tableau to a free-variable tableau calculus with a global substitution σ , one obtains a complete calculus for classical first-order logic [21]. We show the three inference rules of the clausal connection tableau calculus in Figure 1: *start* adds a clause to an empty tableau, *reduction* closes a branch by connecting a leaf literal to a literal on its path, and *extension* adds a clause that is connected to an open branch. With minor modifications, the connection method can be adapted to other logics such as intuitionistic [26] or modal [28] logics, or to non-clausal proof search [29].

The simplicity of the calculus admits very compact theorem provers, often making use of Prolog and related technology [26,31,38]. While tableau methods are no longer the state of the art in classical first-order theorem proving, they are still competitive for proving conjectures in the presence of large numbers of irrelevant axioms (a key application for interactive theorem provers [23]) or in specialised settings [41]. They have also found a new home in experiments applying machine learning to theorem proving [20,36,42], where their simplicity — and to some extent their backtracking — makes them an attractive choice. We assume here that equality has been preprocessed away from the input [25,8,33], although it is possible to extend connection calculi with support for equational reasoning [6,32,4].

2.1 Excess Backtracking in Connection-Driven Search

In order to remain complete, propositional connection systems must consider alternative additions to the tableau, but once a branch has been closed, it can remain so. At the first-order level, *alternative ways to close the same branch* must also be considered: this is because closing a branch may bind variables in the global substitution in a way that prevents closing a different branch later.

Ottén noticed that this requirement produced an enormous amount of backtracking in some cases [27]. He introduced a Prolog *cut* into *leanCoP*’s search routine, rendering it incomplete but significantly reducing backtracking and increasing performance in many cases. Later, Färber studied the behaviour of these cuts extensively and developed several variants [14]. All of Färber’s variants are incomplete, but some are considerably more effective in practice than others. We will use his *meanCoP* system as a point of comparison in Section 6.

2.2 Terminology and Convention

We use some terms informally, which we hope will aid understanding rather than cause confusion. As tableau-based first-order theorem provers typically manipulate a tableau and a substitution together, we will refer to both of them simply as “the *tableau*” where it is not confusing. When an inference of the calculus is attempted but cannot be successfully applied, we say it has *failed*. Moreover, if we can detect that a tableau can never be closed, we say it is *stuck*. Both failed inferences and stuck tableaux may be *explained* in terms of a constraint, which we call a *reason*.

We will need to refer explicitly to particular positions in a tableau. We use the obvious scheme where for any position p , $p.i$ is the position at the i^{th} branch below p . The empty position stands for the root of the tableau. First-order variables in a tableau are named according to the *position* below which their clause is attached, and are hence *consistent* across backtracking. We simply use u, v, w, x, y and z for first-order variables, c and d for constants, f for a function, and P, Q, R , and S for predicates. Finally, we elide parentheses in terms and literals and consider $\neg\neg L$ identical to literal L .

2.3 Constraint Learning

Constraint learning [11] is a well-known but somewhat vaguely defined approach in constraint satisfaction and artificial intelligence. It is not necessary for our purposes to formally define constraint learning nor explore all of its developments, but the core of the idea is as follows. In a backtracking search for a solution to a set of constraints, we may encounter a dead end, where making a step in any available direction violates some constraint. A subset of the search's previous decisions may be blamed for this situation by a justification extraction process. A constraint enforcing that not all of the elements within the justification may be selected simultaneously is added to the constraint set, which prevents search from running into a similar unfortunate situation again. This *learned* constraint (usually in the form of conflict clauses) can also be used to do *backjumping*: backtracking by more than one level.

Constraint learning was particularly effective for Boolean satisfiability [37] (SAT) solving and is the basis for modern *conflict-driven clause learning* (CDCL) SAT solvers and therefore satisfiability modulo theory (SMT) solvers [5].

2.4 Running Example

We use a particular example throughout the paper. Consider the following set of first-order clauses:

$$\forall xyz. Px \vee Qy \vee Rxy \vee Pz \quad (1)$$

$$\forall x. \neg Px \vee S \quad (2) \qquad \neg Pfc \quad (5)$$

$$\neg S \vee \neg Pc \quad (3) \qquad \forall x. \neg Rxc \quad (6)$$

$$\neg Qd \quad (4) \qquad \forall x. \neg Rdx \quad (7)$$

Figure 2 shows a connection tableau built from this set of clauses. It has a single open branch Rxy , shown boxed at position 3. The current substitution is $\{ x \mapsto c, y \mapsto d, z \mapsto fc, w \mapsto x \}$. The only available extension steps for Rxy are $\neg Rdv$ or $\neg Rvc$. The tableau is stuck, as neither are possible. At an earlier stage of construction, before $x \mapsto c$ and $y \mapsto d$, both extensions would have been possible. Note that the way in which Pz at position 4 is closed is irrelevant, while the sub-tableaux at 1 and 2 contribute to the dead end.

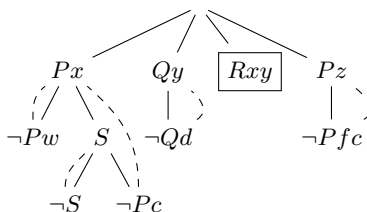


Fig. 2. Running example: a connection tableau built from clauses 1–7.

3 Learning Constraints

We propose learning and storing constraints during proof search in the connection tableau calculus in order to prevent us from repeatedly reaching dead ends for similar reasons. We will now define a constraint language to explain why no inference step is possible in a given situation, which will allow us to design an improved search procedure in Section 4. Suppose that a particular inference step would normally be applicable to an open branch in the tableau. If this step is not applicable, it must be that some rule applications elsewhere in the tableau prevented it. We therefore define our constraint language to be based on the inference rules of the connection tableau calculus. We will refine this language in Section 5, but for now, consider the following definition.

Definition 1 (Simplified Constraint Language). Constraints are sets of atoms. Each atom is either:

1. S_C , representing starting the tableau with clause C ;
2. \mathcal{R}_p^q , representing a reduction from position p to an ancestor q in the tableau;
3. $\mathcal{E}_{C/i}^p$, representing extending position p by a connection to the i^{th} literal of clause C .

Note that each atom includes the open goal (or root) to which the step is applied. This language is sufficient to explain why an inference j that would be possible otherwise is currently not possible within the tableau and describes this situation in a way to cover a whole class of similarly affected tableaux. This is done by finding a subset of the inference steps already applied to the tableau that prevent the application of inference j .

Definition 2 (Reasons for failed inferences). Take an open branch B in a tableau T constructed by a series of inferences I . We construct a sub-tableau T' by applying only those inferences $I' \subseteq I$ which are necessary to produce B , i.e. the start clause and a series of extensions along the path to B . Suppose that there is an inference j that can be applied to B in T' but not in T . A reason for failing to apply j in T is a minimal set $E \subseteq I \setminus I'$ which, if applied additionally to T' , prevents applying j at B in the resulting tableau.

Example 1 (Reasons for inference failure 1). Consider the tableau in Figure 2. To close it, the remaining open branch Rxy must be extended. Suppose that we wish to extend it with clause 7. This would have been initially possible, but by now the global substitution contains $x \mapsto c$ and the extension is impossible.

We can explain this in our language by noticing that the minimal set of previous inferences required to make the extension impossible are those that close the branch at position 1, Px . The other two branches at 2 and 3 are irrelevant, even though those branches are also closed and affect the global substitution. If we take the minimal set of previous inferences, we obtain $\{ \mathcal{S}_1, \mathcal{E}_{2/1}^1, \mathcal{E}_{3/1}^{1,2}, \mathcal{R}_{1.2.2}^1 \}$

Example 2 (Reasons for inference failure 2). We return to the tableau in Figure 2, but now consider extending the open branch with the unit clause 6. Again, we notice that only the branch Qy at position 2 is relevant for explaining why this extension fails, and produce the reason $\{ \mathcal{S}_1, \mathcal{E}_{4/1}^2 \}$.

Now we turn our attention to explaining why a tableau is stuck. This has two parts: stating that there is an open branch B , and showing that no inference can be applied to B .

Definition 3 (Reasons for stuck tableaux). Take T , B , I , T' and I' from Definition 2, and take J to be the set of possible inferences from B in T' . Suppose that no $j \in J$ can be applied to B in T . We compute the set of reasons R_j for each inference j as follows:

1. If the calculus prevents j in T , we compute an explanation R_j by Definition 2.
2. If j is applicable but leads to a tableau that is also stuck, we compute a reason R' for that tableau recursively and set $R_j = R' \setminus \{ j \}$.

I' describes B as an open branch, and the union of all inference failure reasons R_j shows that no inference can be applied to B . We define

$$\bigcup_j R_j \cup I'$$

to be a reason that T is stuck.

Example 3 (Reasons for stuck tableaux). Consider once again the tableau in Figure 2 and its open branch. If the only possible extensions are with clauses 6 and 7, the tableau is stuck as neither can be applied here. Using the reason set from Examples 1 and 2 and noting that I' is simply \mathcal{S}_1 , we obtain

$$\{ \mathcal{S}_1, \mathcal{E}_{2/1}^1, \mathcal{E}_{3/1}^{1,2}, \mathcal{R}_{1.2.2}^1, \mathcal{E}_{4/1}^2 \}$$

as a reason for T being stuck.

In general, reasons are not unique for any given stuck tableau, both because there could be more than one open branch, and because there could be more than one reason for a failed inference. Choosing one reason suffices, but some are likely to be stronger than others.

Algorithm 1 An iterative search routine for finding closed tableaux.

```

 $T \leftarrow$  empty tableau
constraints  $\leftarrow \emptyset$ 
trail  $\leftarrow$  nil
repeat
    success  $\leftarrow$  false
     $B \leftarrow$  select_open_branch( $T$ )
    learn  $\leftarrow$  explain( $T, B$ )
    for all possible inferences  $j$  at  $B$  in  $T$  do
        if not apply( $T, j$ ) then
            learn  $\leftarrow$  learn  $\cup$  compute_reason( $T, j$ )
        else if there is a conflict clause  $C \cup \{j\}$  violated by  $j$  and the trail then
            learn  $\leftarrow$  learn  $\cup C$ 
        else
            trail  $\leftarrow j ::$  trail
            success  $\leftarrow$  true
            break
        end if
    end for
    if not success then
        while learn is violated do
             $i \leftarrow$  pop(trail)
            undo( $i, T$ )
        end while
        record_learned_clause(learn)
    end if
until  $T$  is closed or learn is empty
    
```

4 Search with Learned Constraints

In the previous section, we defined the constraint language so that stuck tableaux can be adequately explained. The search algorithm should now be redesigned to make use of these learned constraints. We implement something similar to that found in CDCL SAT solvers, SMT solvers, or constraint satisfaction systems. Alongside the current tableau, we maintain a *trail* of atoms that are true for the current tableau.

The search algorithm repeatedly applies rules, gets stuck, learns a constraint, and backtracks. It terminates when the tableau is closed or the empty constraint is learned. We maintain the invariant that no learned constraint is *violated* by the current trail: a constraint is violated if all of its atoms are contained in the trail. In case the tableau is empty, a start clause is chosen. Otherwise, at each iteration, an open branch of the tableau is selected for reduction or extension. If any such inference is possible, it is applied to the tableau, and the corresponding atom describing the result of the inference is added to the trail. On the other hand, an inference j may be impossible either:

1. because the calculus does not permit it, or

2. because adding its corresponding atom would violate a learned clause.

In the first case, a reason for the failed inference is computed as in Definition 2. Otherwise, the reason for its failure is the learned constraint it would violate, *minus* the atom corresponding to j (Definition 3). In this way, when all possible inferences at an open branch have failed, a constraint against the stuck tableau is learned such that all the atoms in the constraint are on the trail. To restore the invariant, the system backtracks until at least one violated atom is no longer on the trail.

The overall procedure is shown in Algorithm 1: `compute_reason` computes a reason in the sense of Definition 2, and `explain` computes I' for B as in Definition 3. It is considerably more complex than the usual procedures for finding closed connection tableaux, particularly those embedded in Prolog via the “lean” methodology [26]. However, it is simpler in one aspect: there is no need to remember alternative inferences at backtracking points, which can be quite involved if not implemented in terms of Prolog’s existing backtracking mechanism [19].

4.1 Resource Bounds

Connection systems typically search by iterative deepening on a particular metric, such as the length of the longest branch. A small limit is set initially, and then a system will begin search, bounded by the current limit. If no tableau exists at one iterative deepening level, the limit is increased and search tried again. We follow this approach here: our search algorithm looks for a tableau bounded by some maximum branch length, and if one does not exist, it will eventually terminate by learning the empty clause.

Constraints learned at one iterative deepening level cannot be reused for the next, as our approach would become incomplete. It is possible to alter the constraint language in order to express constraints that (do not) depend on the depth limit, but this is of limited practical purpose for at least two reasons. Firstly, because the search space at the next iterative deepening level tends to be much larger than the exhausted previous level, reusing constraints independent of the depth limit from the previous level does not help much. Second, in practice, there are few such constraints.

4.2 Soundness, Termination and Completeness

We show that Algorithm 1 terminates at any fixed depth limit and use this to show completeness. Soundness is trivial, as the routine searches within an existing sound calculus.

Lemma 1 (Termination). *Fix a depth limit. Algorithm 1 terminates.*

Proof. First, note that at any depth limit and for any finite set of input clauses, there is a finite number of possible tableaux, all of which are of finite size. Because all tableaux are of finite size, the routine will eventually either close

the tableau, terminating immediately, or become stuck. When stuck, the routine learns a constraint which, by construction, is violated by the current trail, and therefore prevents reaching at least this particular tableau again. As constraints are never forgotten within the same depth limit, and there are a finite number of possible tableaux, termination is guaranteed as the solver eventually learns constraints eliminating all possible tableaux. \square

Lemma 2. *Learned constraints are not violated by any closed tableau reachable in the proof calculus at a given depth limit.*

Proof. By induction on the derivation of learned constraints. Suppose a learned constraint C is violated by a closed tableau T^* . By definition, C is a subset of the rules required to construct T^* . Construct the intermediate tableau T generated by C . Take the next inference step j from T towards T^* . In Definition 3, C is justified on the basis that all inferences, including j , from T are impossible, either because the calculus prevents it (Definition 2), or because another constraint C' is violated. By the induction hypothesis, applying j does not violate any such C' , so j must not be legal in the calculus, and we have a contradiction. \square

Theorem 1 (Completeness). *If a closed tableau exists at a depth limit, it will be found by Algorithm 1.*

Proof. By termination and Lemma 2. \square

5 Refining the Constraint Language

The simple constraint language introduced in Section 3 is sufficiently expressive to block classes of similar tableaux, but is quite specific to a particular tableau and fails to block all the similar tableaux we might like. It is also quite clunky to work with and would be difficult to compute inference failure reasons efficiently in practice: see Section 6.2.

We therefore decompose each atom into multiple smaller atoms of two kinds: placing literals at positions in a tableau, and binding variables to terms. However, *mutatis mutandis* the search procedure remains the same, pushing one or more such atoms onto the trail for any one inference. As well as being simpler to implement, constraints can be much stronger as they do not block only a particular derivation of a tableau, but any tableau having particular literals and variable bindings.

Definition 4 (Refined Constraint Language). *A constraint remains a set of atoms. However, each atom is either*

1. $L@p$, a literal L being placed at position p
2. $x \mapsto t$, a variable x is bound to a term t where t itself may be a variable.

Each inference of the connection tableau calculus can be expressed as some combination of these. Adding clauses in start and extension rules is done by placing their literals at the corresponding positions. Connections of literals in extension and reduction rules are applied by computing the required variable bindings.

Example 4 (Refined constraint learning). Take the tableau in Figure 2. We will explain why it is stuck in terms of the new constraint language. Extending Rxy with $\neg Rvc$ is not possible because $y \mapsto d$, which is on the trail because Qy was connected to $\neg Qd$. Similarly, extending Rxy with $\neg Rdv$ is not possible because $x \mapsto c$. To finish the explanation, we have to say why Rxy needs to be closed in the first place, but this is straightforward: $Rxy@3$. The final explanation is therefore

$$\{ Rxy@3, x \mapsto c, y \mapsto d \}.$$

5.1 No-Connection Atoms

In the proposed language, explaining why an open branch cannot be *reduced* can become overly specific.

Example 5 (Explaining reduction failure). Consider an open branch $\neg Pc$ at the depth limit, with path literals Px, Qc, Rcd, S and a substitution containing $x \mapsto d$. Suppose the positions from root to leaf are $p_1 \dots p_5$. Clearly $\neg Pc$ cannot be reduced. In the case of Px , the constraint contains somewhat useful information: if x were not bound to d , this branch could be reduced. However, for all other path literals, the only useful information is that they cannot be connected with $\neg Pc$, but this is not in the language, and we must learn

$$\{ \neg Pc@p_5, Px@p_1, x \mapsto d, Qc@p_2, Rcd@p_3, S@p_4 \}.$$

This kind of situation occurs often in practice and needlessly specialises the learned constraint to a particular sequence of path literals. To avoid this problem, a new kind of atom $p \not\sim q$ is introduced, representing that no connection can ever be made between the two positions p and q , regardless of the substitution. Whenever a literal is added to the tableau at position q , its path is checked to see, which literals at positions p it could be reduced with, and where this is impossible, $p \not\sim q$ is added to the trail. In the above example, the learned constraint would be

$$\{ \neg Pc@p_5, Px@p_1, x \mapsto d, p_2 \not\sim p_5, p_3 \not\sim p_5, p_4 \not\sim p_5 \}.$$

and more general, as it does not specify which literals are at p_2, p_3 , or p_4 .

5.2 Disequations

Classical refinements such as regularity and eliminating tautologies greatly improve the performance of connection systems [21]. These are classically implemented by means of *disequations*. We can support this naturally in the constraint language by adding disequation atoms of the form $s \neq t$. When a disequation is falsified, backtracking can be induced by giving the disequation and the variable bindings required to falsify it as a learned constraint.

6 Implementation and Experimental Validation

We implemented a prototype system `hopCoP`⁴ to experiment with constraint learning. So far, the implementation is imperative — although we suspect a lean Prolog implementation may be possible via `assert/1` [18] — and owes much to implementation techniques found in the Bare Metal Tableaux Prover [19] and `meanCoP` [14]. `hopCoP` implements the clausal connection tableau calculus, without cuts [27] and starting from clauses derived from the conjecture. With the obvious exception of constraint learning, `hopCoP`’s search routine resembles that of `meanCoP`, if the `meanCoP` flags `--conj` `--nopaths` are set⁵. `meanCoP` also implements a lemma mechanism [21] that `hopCoP` so far lacks. In Sections 6.1 and 6.2 we highlight two aspects of the implementation that may be of interest to implementers of similar systems.

6.1 Constraint Management and Detecting Conflicts

A large number of constraints are learned during search, millions with a long enough time limit. However, this seems to be less dramatic than in other settings such as SAT solving, and we did not find any benefit from attempting to garbage-collect old constraints, so `hopCoP` retains *all* constraints it learns.

It is still necessary to efficiently find conflicts among this large set when adding atoms to the trail. This is done by a 1-watched-literal scheme [24]: there is no need for the 2-watched-literal scheme popular in SAT solving, as all atoms have the same polarity and unit propagation is of little use. More than one conflict may be found when adding an atom to the trail: it is worth trying to choose conflicts that minimise the resulting learned constraint. `hopCoP` greedily chooses the conflict that adds the fewest atoms to the constraint learned so far.

6.2 Computing Explanations

To explain why an inference that connects two literals is not possible, a subset of the current substitution must be computed that prevents their unification. It is no doubt possible to construct a complex variable-tracking scheme to do this quickly, but for our application, the following procedure is acceptably fast.

1. Unify the two literals using a new scratch substitution τ . As the inference was possible with an empty substitution (but not with σ), this must succeed.
2. Record the current state of τ , say τ_0 .
3. For every binding $x \mapsto t$ in σ we:
 - (a) Try to unify x and t in τ . If this succeeds, continue the loop at step 3.
 - (b) Reset τ to τ_0 and try to unify x and t again. If successful, go to step 2.
 - (c) Exit the loop on failure, retaining $x \mapsto t$ in τ .

After this procedure, τ should contain the necessary subset of σ . A similar routine can be used to determine why a disequation is falsified.

⁴ <https://github.com/MichaelRawson/hopcop>, commit `a4a0f66`

⁵ starting with the annotated conjecture clauses (`-conj`) and preventing input clauses reordering (`-nopaths`)

6.3 Experiments

Table 1. The number of extension steps tried in order to determine that there is no closed tableau of a certain depth on PUZ005-1. A proof exists at depth 8.

depth	1	2	3	4	5	6	7
meanCoP	1	4	24	108	535	9,963	6,445,008
hopCoP	1	4	89	495	2,309	10,066	48,517

Having gone to some effort to reduce backtracking in theory, we wish to know whether this also helps in practice. We first manually inspected the behaviour of both meanCoP and hopCoP on problems taken from the PUZ domain of TPTP. meanCoP reports the number of successfully applied extension steps required to exhaust each iterative deepening level, so we instrumented hopCoP to do the same. Table 1 shows the number of steps required for PUZ005-1⁶. At lower iterative deepening levels, the result is mixed due to differences in search decisions and meanCoP’s lemma rule, but hopCoP typically extends a clear lead at higher levels. meanCoP maintains a much higher rate of inference: our implementation is not highly optimised, but we suspect that the overhead of maintaining the learned constraints would cause significant inferences-per-second overhead compared to meanCoP even if it were.

hopCoP also ran head-to-head against meanCoP on several popular first-order benchmark sets: FOF and CNF problems from TPTP version 9.0.0 [39], the MPTP challenge problems [1] in *bushy* and *chainy* variants, and the *Miz40* ATP-minimised set [20], of which *M2k* is a subset. Both systems were given a time limit of 10 seconds per problem and meanCoP was configured with `--conj --nopaths` (to better match hopCoP, see above). We also ran meanCoP with the additional `--cut` argument, which we call !meanCoP: this renders meanCoP incomplete in exchange for significantly reduced backtracking.

We do not wish to claim anything about the relative strength of the systems, only that the data are consistent with the hypothesis that the reduction in backtracking achieved overcomes the overhead in terms of inferences-per-second speed. Table 2 shows the number of solved problems. Readers may also be interested in the very thorough experimental data in Färber’s discussion of various backtracking schemes [14].

7 Related Work

The most directly related work is the various fixed restrictions on backtracking in connection tableau [27, 14]: these are by nature incomplete, but effective. Older techniques such as *failure caching* [3] also achieve a reduction in backtracking and

⁶ the first CNF problem of moderate difficulty in the PUZ domain

Table 2. Theorems proved in 10 seconds by `hopCoP`, `meanCoP`, and `!meanCoP` on various benchmark sets.

	M2k	Miz40	bushy	chainy	TPTP
<code>meanCoP</code>	795	7,592	480	157	3,578
<code>!meanCoP</code>	878	9,748	562	337	3,283
<code>hopCoP</code>	1,050	13,040	589	203	4,026

remain complete, but with different mechanisms. The Goéland tableau system exchanges substitution information [9] between concurrent branch explorations: this is likely to reduce backtracking, but again with a different mechanism. Encodings of connection-driven search into SAT or SMT, such as ChewTPTP [12], are complete and will also learn constraints during proof search, but behave very differently and are mostly encoded upfront.

We ourselves have tried various encodings [13] of connection methods into SAT and SMT via *user propagation* [7,15]. We found that SAT/SMT solvers, even with a lazy user-propagator encoding, are not a good match for this kind of proof search, as their internal heuristics have no knowledge of the current state of the tableau. The solver will, for instance, very happily decide or propagate variables that encode some sub-tableau completely disconnected from the current state. Refined encodings such as in Section 5 improve the encoding’s performance, but allow the solver to partially apply inferences: we are not sure of the performance merits of this, but it is highly confusing.

The first author has also investigated *generating* SAT clauses from instances of clauses found in connection tableau during search [34], as a kind of instance-based method [16]. When the set of SAT clauses becomes unsatisfiable, it shows that the input clause set was also unsatisfiable. While an extension of this instance-generation approach can be used to influence the connection tableau search, it is not the core of the method, unlike the constraint learning approach here. The two are largely orthogonal and could be combined profitably.

Other work that is highly related but may be confusing is the concept of *backjumping* in modal and other tableau [17]: the concept and its origins appears to be similar, but it is used to *avoid* logical conflicts on a branch when looking for a model, rather than to avoid getting stuck when looking for a closed tableau. The MeTTeL² tableau prover generator [40] has generic support for a similar kind of backjumping, which it calls *conflict-directed backjumping*.

8 Outlook

We have integrated a constraint learning approach to guide search and reduce backtracking into a prototype first-order connection theorem prover `hopCoP` and observe that it reduces the search space significantly, which translates into practical performance. A trade-off is memory use: constraints have to be kept somewhere. This was not excessive in our experience, but may prevent running `hopCoP` on your iPod® [30].

It is likely that other non-confluent tableau calculi may benefit from such an approach. We would also be interested in the intersection of this kind of learning with the *machine* kind of learning: constraint learning could reduce the options available to a learned heuristic, while a good learned heuristic might rapidly learn useful constraints.

There are some areas for further improvement to the technique. The most irritating are the explicit positions present in the constraint language, which limits the application of learned constraints. Eliminating this would require detecting conflicts modulo structurally equivalent positions, which we suspect may be difficult to do efficiently. A future *lean* implementation may help with this, perhaps based on the recent realisation that *backjumping is exception handling* [35].

Acknowledgments. We are grateful to Michael Färber in particular for his meanCoP tool and stimulating discussions on this and related topics. This research was funded in whole or in part by the ERC Consolidator Grant ARTIST 101002685, the ERC Proof of Concept Grant LEARN 101213411, the TU Wien Doctoral College SecInt, the FWF SpyCoDe Grant 10.55776/F85, the WWTF grant ForSmart 10.47379/ICT22007, and the Amazon Research Award 2023 QuAT.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Alama, J., Heskes, T., Kühlwein, D., Tsvitsivadze, E., Urban, J.: Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reason.* **52**(2), 191–213 (2014). <https://doi.org/10.1007/S10817-013-9286-5>
2. Andrews, P.B.: Theorem proving via general matings. *J. ACM* **28**(2), 193–214 (1981). <https://doi.org/10.1145/322248.322249>
3. Astrachan, O.L., Stickel, M.E.: Caching and lemmaizing in model elimination theorem provers. In: Kapur, D. (ed.) *CADE. LNCS*, vol. 607, pp. 224–238. Springer (1992). https://doi.org/10.1007/3-540-55602-8_168
4. Backeman, P., Rümmer, P.: Theorem proving with bounded rigid e-unification. In: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings. Lecture Notes in Computer Science*, vol. 9195, pp. 572–587. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_39, https://doi.org/10.1007/978-3-319-21401-6_39
5. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) *Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications*, vol. 336, pp. 1267–1329. IOS Press (2021). <https://doi.org/10.3233/FAIA201017>
6. Bibel, W.: *Automated theorem proving*, 2nd Edition. Artificial intelligence, Vieweg (1987), <https://www.worldcat.org/oclc/16641802>
7. Bjørner, N.S., Eisenhofer, C., Kovács, L.: Satisfiability modulo custom theories in Z3. In: Dragoi, C., Emmi, M., Wang, J. (eds.) *VMCAI. LNCS*, vol. 13881, pp. 91–105. Springer (2023). https://doi.org/10.1007/978-3-031-24950-1_5
8. Brand, D.: Proving theorems with the modification method. *SIAM J. Comput.* **4**(4), 412–430 (1975). <https://doi.org/10.1137/0204036>

9. Cailler, J., Rosain, J., Delahaye, D., Robillard, S., Bouziane, H.: Goéland: A concurrent tableau-based theorem prover (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR. LNCS, vol. 13385, pp. 359–368. Springer (2022). https://doi.org/10.1007/978-3-031-10769-6_22
10. D’Agostino, M., Gabbay, D.M., Hähnle, R., Posegga, J., (eds): Handbook of tableau methods. *J. Log. Lang. Inf.* **10**(4), 518–523 (2001). <https://doi.org/10.1023/A:1017520120752>
11. Dechter, R.: Learning while searching in constraint-satisfaction-problems. In: Proceedings of the 5th National Conference on Artificial Intelligence. pp. 178–185. Morgan Kaufmann (1986), <http://www.aaai.org/Library/AAAI/1986/aaai86-029.php>
12. Deshane, T., Hu, W., Jablonski, P., Lin, H., Lynch, C., McGregor, R.E.: Encoding first order proofs in SAT. In: CADE. LNCS, vol. 4603, pp. 476–491. Springer (2007). https://doi.org/10.1007/978-3-540-73595-3_35
13. Eisenhofer, C., Rawson, M., Kovács, L.: Spanning matrices via satisfiability solving (2025). <https://doi.org/to-appear>, appears in the same TABLEAUX’25 inproceeding
14. Färber, M.: A curiously effective backtracking strategy for connection tableaux. In: Otten, J., Bibel, W. (eds.) ARECCa. CEUR Workshop Proceedings, vol. 3613, pp. 23–40. CEUR-WS.org (2023), https://ceur-ws.org/Vol-3613/ARECCa2023_paper2.pdf
15. Fazekas, K., Niemetz, A., Preiner, M., Kirchweger, M., Szeider, S., Biere, A.: IPASIR-UP: user propagators for CDCL. In: Mahajan, M., Slivovsky, F. (eds.) SAT. LIPIcs, vol. 271, pp. 8:1–8:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). <https://doi.org/10.4230/LIPICS.SAT.2023.8>
16. Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. In: LICS. pp. 55–64. IEEE Computer Society (2003). <https://doi.org/10.1109/LICS.2003.1210045>
17. Hustadt, U., Schmidt, R.A.: Simplification and backjumping in modal tableau. In: de Swart, H.C.M. (ed.) TABLEAUX. LNCS, vol. 1397, pp. 187–201. Springer (1998). https://doi.org/10.1007/3-540-69778-0_22
18. ISO/IEC: Information technology — Programming languages — Prolog — Part 1: General Core. Standard, International Organization for Standardization (1995)
19. Kaliszyk, C.: Efficient low-level connection tableaux. In: de Nivelle, H. (ed.) TABLEAUX. LNCS, vol. 9323, pp. 102–111. Springer (2015). https://doi.org/10.1007/978-3-319-24312-2_8
20. Kaliszyk, C., Urban, J., Michalewski, H., Olsák, M.: Reinforcement learning of theorem proving. In: NeurIPS. pp. 8836–8847 (2018), <https://proceedings.neurips.cc/paper/2018/hash/55acf8539596d25624059980986aaa78-Abstract.html>
21. Letz, R., Stenz, G.: Model elimination and connection tableau procedures. In: Handbook of Automated Reasoning (in 2 volumes), pp. 2015–2114. Elsevier and MIT Press (2001). <https://doi.org/10.1016/B978-044450813-3/50030-8>
22. Loveland, D.W.: Mechanical theorem-proving by model elimination. *J. ACM* **15**(2), 236–251 (1968). <https://doi.org/10.1145/321450.321456>
23. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. *J. Autom. Reason.* **40**(1), 35–60 (2008). <https://doi.org/10.1007/S10817-007-9085-Y>
24. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC. pp. 530–535. ACM (2001). <https://doi.org/10.1145/378239.379017>
25. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Handbook of Automated Reasoning (in 2 volumes), pp. 371–443 (2001). <https://doi.org/10.1016/B978-044450813-3/50009-6>

26. Otten, J.: *leanCoP 2.0 and ileancop 1.2: High performance lean theorem proving in classical and intuitionistic logic (system descriptions)*. In: IJCAR. LNCS, vol. 5195, pp. 283–291. Springer (2008). https://doi.org/10.1007/978-3-540-71070-7_23
27. Otten, J.: Restricting backtracking in connection calculi. *AI Commun.* **23**(2-3), 159–182 (2010). <https://doi.org/10.3233/AIC-2010-0464>
28. Otten, J.: *MleanCoP: A connection prover for first-order modal logic*. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR. LNCS, vol. 8562, pp. 269–276. Springer (2014). https://doi.org/10.1007/978-3-319-08587-6_20
29. Otten, J.: *nanoCoP: A non-clausal connection prover*. In: IJCAR. LNCS, vol. 9706, pp. 302–312. Springer (2016). https://doi.org/10.1007/978-3-319-40229-1_21
30. Otten, J.: The pocket reasoner – automatic reasoning on small devices. In: NIK. Bibsys Open Journal Systems, Norway (2018), <https://ojs.bibsys.no/index.php/NIK/article/view/512>
31. Otten, J.: 20 years of *leanCoP* - an overview of the provers. In: AReCCa. CEUR Workshop Proceedings, vol. 3613, pp. 4–22. CEUR-WS.org (2023), https://ceur-ws.org/Vol-3613/AReCCa2023_paper1.pdf
32. Paskevich, A.: Connection tableaux with lazy paramodulation. *J. Autom. Reason.* **40**(2-3), 179–194 (2008). <https://doi.org/10.1007/S10817-007-9089-7>, <https://doi.org/10.1007/s10817-007-9089-7>
33. Prusak, G., Kaliszzyk, C.: Lazy paramodulation in practice. In: Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022), Haifa, Israel, August, 11 - 12, 2022. CEUR Workshop Proceedings, vol. 3201. CEUR-WS.org (2022), <https://ceur-ws.org/Vol-3201/paper3.pdf>
34. Rawson, M., Reger, G.: Eliminating models during model elimination. In: Das, A., Negri, S. (eds.) TABLEAUX. LNCS, vol. 12842, pp. 250–265. Springer (2021). https://doi.org/10.1007/978-3-030-86059-2_15
35. Robbins, E., King, A., Howe, J.M.: Backjumping is exception handling. *Theory Pract. Log. Program.* **21**(2), 125–144 (2021). <https://doi.org/10.1017/S1471068420000435>
36. Rømming, F., Otten, J., Holden, S.B.: Connections: Markov decision processes for classical, intuitionistic and modal connection calculi. In: AReCCa. CEUR Workshop Proceedings, vol. 3613, pp. 107–118. CEUR-WS.org (2023), https://ceur-ws.org/Vol-3613/AReCCa2023_paper8.pdf
37. Silva, J.P.M., Sakallah, K.A.: GRASP — a new search algorithm for satisfiability. In: ICCAD. pp. 220–227. IEEE Computer Society / ACM (1996). <https://doi.org/10.1109/ICCAD.1996.569607>
38. Stickel, M.E.: A Prolog technology theorem prover. In: 9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings. LNCS, vol. 310, pp. 752–753. Springer (1988). <https://doi.org/10.1007/BFB0012881>
39. Sutcliffe, G.: The TPTP problem library and associated infrastructure - from CNF to TH0, TPTP v6.4.0. *J. Autom. Reason.* **59**(4), 483–502 (2017). <https://doi.org/10.1007/S10817-017-9407-7>
40. Tishkovsky, D., Schmidt, R.A., Khodadadi, M.: The tableau prover generator *MetTeL²*. In: del Cerro, L.F., Herzig, A., Mengin, J. (eds.) JELIA. LNCS, vol. 7519, pp. 492–495. Springer (2012). https://doi.org/10.1007/978-3-642-33353-8_41
41. Wernhard, C., Bibel, W.: Investigations into proof structures. *J. Autom. Reason.* **68**(4), 24 (2024). <https://doi.org/10.1007/S10817-024-09711-8>

42. Zombori, Z., Urban, J., Brown, C.E.: Prolog technology reinforcement learning prover - (system description). In: IJCAR. LNCS, vol. 12167, pp. 489–507. Springer (2020). https://doi.org/10.1007/978-3-030-51054-1_33