Eliminating Models during Model Elimination

Michael Rawson and Giles Reger

University of Manchester, Manchester, UK
`michael@rawsons.uk, giles.reger@manchester.ac.uk`

Abstract. We investigate the integration of SAT technology into clausal connection-tableau systems for classical first-order logic. Clauses present in tableaux during backtracking search are heuristically grounded and added to an incremental SAT solver. If the solver reports an unsatisfiable set of ground clauses at any point, search may be halted and a proof reported. This technique alone is surprisingly effective, but also supports further refinements "for free". In particular we further investigate depth control of randomised search based on grounded clauses, and a kind of ground lemmata rule derived from the partial SAT model.

Keywords: Connection Tableaux · Boolean Satisfiability · Instantiation

1 Introduction

The style of heuristic search in backtracking/iterative-deepening theorem provers for first-order logic, often used in conjunction with connection tableaux, is very different from the search found in saturation-style systems, often used with superposition calculi. Both approaches have their strengths and weaknesses, and typically perform well on different kinds of domains and problems.

One possible weakness of backtracking systems is that very little search effort expended in failing to find a proof can be reused, and in fact many popular backtracking systems "learn" almost nothing as search progresses. Contrast this with saturation systems, where deduced formulae are typically retained indefinitely, and even formulae not used in the final proof can aid proof search via mechanisms such as subsumption. Fixing this defect in backtracking systems generally and efficiently is not easy, and if taken to extremes results in a saturation system.

However, *ground* reasoning is typically more efficient than full first-order reasoning. This suggests something of a compromise: first-order search remains backtracking in nature, but a ground approximation to first-order information is retained and used to aid future first-order search. More concretely, we heuristically ground the clauses that make up tableaux constructed during backtracking search, then insert these grounded clauses into an incremental SAT solver, where they stay for the entire duration of proof search.

This extra effort is compensated by the ability to report proofs found at the ground level (Section 4); a good heuristic for controlling a combination of restricted backtracking, randomisation and iterative deepening (Section 5); and a partial assignment of literals that can be skipped, focussing proof search (Section 6). We build a testbed system (Section 3) and experimentally evaluate our approach against a baseline and other systems (Section 8), showing that the overhead of grounding clauses pays off handsomely in practice.

2 Preliminaries

The following relates to fully-automatic theorem provers for classical first-order logic (with equality) with the usual syntax and semantics [46]. We focus particularly on systems implementing connection tableaux calculi and systems that use ground reasoning tools such as SAT or SMT solvers to accelerate or improve first-order search.

2.1 Connection Tableau Systems

The connection tableau[1] calculus [27] is chiefly a restriction on clausal free-variable tableaux requiring all additions to tableaux be *connected* to leaf literals: that is, extension clauses must contain a unifiable literal of opposite sign. This is an extremely strong restriction on general clause tableaux, which remains complete but loses proof confluence, necessitating backtracking search to build closed connection tableaux. Backtracking may be reduced in exchange for a loss of completeness with *restricted backtracking* schemes [32]. Figure 1 shows the basic rules of the calculus: tableaux begin with a *start* clause; a leaf may be closed by *reduction* if there is a unifiable literal of opposite sign in the current branch; and *extension* clauses may be added to the tableau if they are connected to the current leaf literal.

Competitive connection systems such as the SEquential THEOrem prover SETHEO [5] and, later, leanCoP [31] typically employ a number of optimising preprocessing steps, calculus refinements, search heuristics and efficient implementation techniques to improve performance on problems of interest. We note here a whole area of research designed to re-use work performed in other areas of backtracking search, such as failure caching [27], to which our work has similar aims but a different method.

As well as good performance in exchange for little complexity, such systems have a number of advantages: they are simple to implement (particularly in Prolog), often leading to a relatively small "trusted computing base" which can be easily certified [23, 51]; they cope well with a large number of axioms due to a goal-directed search style; and their memory use remains low, or even constant. They can often be adapted for other domains and research areas, such as intuitionistic logic [31], modal logic [33], non-clausal reasoning [34], machine learning for theorem proving [17, 22], and low-resource computing [35].

[1] also known as, or closely related to, the *connection method* [6], *model elimination* [28], and/or the method of *matings* [2]

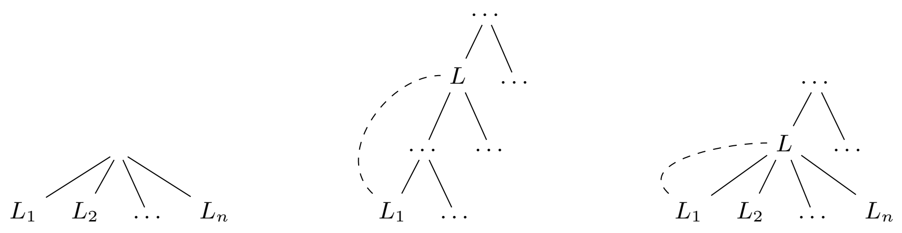

Fig. 1. The three inference rules of the clausal connection tableau calculus: *start*, *reduction*, and *extension*. In the *start* and *extension* rules, $C = L_1 \vee L_2 \vee \ldots \vee L_n$ is a fresh copy of a clause. In the *reduction* and *extension* rules, the global unifier is refined such that $\sigma(\neg L) = \sigma(L_1)$. i.e. L and L_1 are *connected* (illustrated by dashed lines).

2.2 Boolean Satisfiability

Boolean satisfiability (SAT) is a well-known NP-complete problem [10]. We will concisely phrase the problem as "given a set of propositional clauses, find an assignment of propositional variables such that each clause is satisfied, or report their unsatisfiability". Despite the computational difficulty, SAT solvers have improved rapidly [9] and can now quickly solve SAT instances previously considered impractically large or hard [3]. Arguably the major driving force behind this improvement is the realisation that most useful problems are not merely random, but contain structure that can be exploited by carefully-designed heuristics.

One such heuristic, *conflict-driven clause learning* (CDCL), in which new clauses are "learned" from a certain conflicting section of search space, is particularly effective [29]. It also allows for SAT solvers to become *incremental*, so that recomputing satisfiability as new clauses are added to the set is a cheaper operation. SAT is also often used as an "assembly language" for richer or harder problems. We discuss SAT for aiding first-order reasoning below, but Satisfiability Modulo Theories (SMT) [14] and bounded model checking [8] are two well-known applications from other domains.

2.3 Ground Support for First-Order Reasoning

The use of SAT solvers to provide ground support within first-order reasoning has been previously explored in various ways. In some approaches the main reasoning method is by reduction to SAT. For example, finite model finding methods [12, 40] iteratively ground a first-order problem with a growing set of domain constants in order to find a finite model. Or the Instance Generation calculus [24, 25], which approximates the unsatisfiability problem for sets of first-order clauses by a sequence of propositional problems: a propositional abstraction is iteratively refined by the addition of new instances. More naïvely, there are also cases where near-propositional problems can be decided directly via grounding [43]. Going beyond first-order reasoning, Satallax [11] is a a higher-order prover that reasons via reduction to a series of SAT problems.

The previous approaches use SAT solvers as black boxes, as opposed to the more fine-grained approach taken by the Model Evolution Calculus [4] which interleaves instance generation with DPLL-style reasoning. In an unusual twist, the CHEWTPTP system uses a clever incremental ground encoding [15] of connection tableaux *search* such that at some point the ground solver may return propositional assignments representing closed connection tableaux.

Other first-order reasoning methods utilise SAT solvers to aid a separate proof search method, as in this paper. The AVATAR [49, 37] framework implemented within Vampire [26, 38] uses a SAT or SMT solver to organise the process of clause splitting within saturation-based search. The global subsumption simplification technique [24, 39] uses a SAT solver to replace a clause by a subclause if the subclause holds globally, which can be under-approximated by propositional reasoning.

Finally, the saturation-based E theorem prover [45] has been extended with a lightweight technique that periodically grounds the search space and checks for propositional unsatisfiability [44]. This work is closest to what we propose in this work but in the context of saturation-based methods.

2.4 First-Order Benchmarks

We use several first-order benchmark problem sets to evaluate work experimentally. By "TPTP", we mean the provable FOF fragment (7,609 problems) of the Thousands of Problems for Theorem Provers set [47] 7.3.0. The MPTP2078 challenge [1] provides 2078 problems translated from the Mizar Mathematical Library [18] by the MPTP system [48], in two forms: "bushy", where problems are typically smaller and contain only relevant premises; and "chainy" where problems contain all preceding results. *M2k* is a slightly-easier set of 2003 related problems used for development [22], also originating from Mizar and MPTP.

3 Research Vehicle: SATCoP

We require a testbed for our experiments with the techniques outlined. In principle we could have modified e.g. leanCoP to take advantage of the "lean Prolog technology" approach (and we hope to explore this direction in future), but for these first experiments we found it easier to use an imperative language and our own system. We refer to the basic system described below as SATCoP_0, and to the system improved with additional SAT-based techniques as SATCoP.

SATCoP_0 implements the clausal connection tableau calculus. A simple clause normal form translation without definitions [32] translates general first-order formulae into clauses, and equality (if present) is then axiomatised in the usual way. No other preprocessing, such as reordering of clauses, takes place. Search starts with clauses derived from the conjecture[2], and proceeds by iterative deepening

[2] Unless there are no such clauses or *all* clauses stem from the conjecture, in which case positive clauses are used instead.

Algorithm 1: sketch of the basic SATCoP_0 search routine

```
$\sigma_U = \emptyset$; // global tableaux-level unifier, modified by unify()
limit = 0; // depth limit for iterative deepening

function start() : bool is
    loop
        foreach $C \in$ start clauses do
            if prove-all($\epsilon$, $C$) then return true ;
            $\sigma_U = \emptyset$; // reset $\sigma_U$ to try again
        limit = limit + 1;

function prove-all(path, clause) : bool is
    foreach literal $\in$ clause do
        if $\neg$prove(path, literal) then return false ;
    return true

function prove(path, goal) is
    // apply the reduction rule (restricted backtracking)
    foreach $L \in$ path do
        if $sign(goal) \neq sign(L)$ and unify(goal, $\neg L$) then return true ;
    // limit search depth
    if |path| $\geq$ limit then return false ;
    // apply the extension rule (restricted backtracking)
    $\sigma'_U = \sigma_U$;
    foreach fresh copy $C$ of a problem clause do
        foreach $L \in C$ do
            if $sign(goal) \neq sign(L)$ and unify(goal, $\neg L$) then
                if prove-all(append(path, goal), $C \setminus \{L\}$) then return true ;
                $\sigma_U = \sigma'_U$; // reset $\sigma_U$ to try again
                continue
```

on the length of the path. When trying to close a branch, reduction steps are tried before extension steps, and backtracking is restricted [32] in the style that Färber calls REI in his description of backtracking schemes [16]. The regularity condition [27] is enforced and some clause-level tautologies are eliminated. No intra-tableau mechanisms for re-use of intermediate results (such as *lemmata* or *folding up*) are implemented as this would overlap somewhat with Section 6, but in principle nothing prevents implementing this for further performance. For readers not familiar with connection systems and restricted backtracking, Algorithm 1 provides a sketch of the search routine.

The concrete system owes many implementation techniques to the Bare Metal Tableaux Prover [21]. In any case, the precise details of the basic system are not critically important here: we present the effect of each different techniques and final performance by experimental evaluation in Section 8. We expect these methods to be generally applicable to similar connection systems, at least for classical first-order logic, given a careful implementation.

4 Grounding Clausal Tableaux

A clausal tableaux (not necessarily closed) is built from instances of clauses derived from the negated input problem. In the first-order case, tableau variables represent a concrete ground term that is yet to be fully determined. As a result, any given tableaux represents a multiset of partially-instantiated clauses. Tableaux operations have pleasant interpretations in this setting: clauses added to tableaux are added to the set, and unifications within the tableau monotonically refine the instantiation of clauses in the set.

Backtracking search for closed clausal tableaux can therefore be seen as producing a *stream* of clauses with various instantiations: each inference rule produces a tableau built from a certain multiset of clauses, each of which can be fed into the stream. It is a sound deduction to apply any grounding substitution scheme to each clause, mapping remaining variables to ground terms.

To see this, consider a clause C in the input problem containing variables $\bar{x}$. During backtracking search, C is added to the tableau by applying a renaming substitution σ_R, mapping $\bar{x}$ to variables fresh for the tableau. Then, a number of unification steps results in a tableau-level unifier σ_U from tableau variables to arbitrary terms constructed over the signature and tableau variables. Finally, a grounding substitution σ_G maps tableau variables to arbitrary members of the Herbrand universe. Trivially, the composite substitution $\sigma = \sigma_G \circ \sigma_U \circ \sigma_R$ is a grounding substitution and

$$(\forall \bar{x}.C) \Rightarrow C\sigma$$

is a tautology, so $C\sigma$ is both a ground clause and a valid deduction from $\forall \bar{x}.C$.

Ground atoms can be bijectively mapped to propositional variables, obtaining a propositional approximation to the partially-instantiated clause present in the tableau. In this way, backtracking tableaux search over premises produces a stream of ground clauses such that if the ground approximation is unsatisfiable, so are the premises.

4.1 Reporting Unsatisfiability

This stream of ground clauses does not seem immediately useful. However, by inserting this stream of grounded clauses into a SAT solver, it can report when the clauses seen so far are unsatisfiable, witnessing a proof. Often this state occurs significantly before finding a closed connection tableau, which makes the technique potentially useful. We modify the basic system to perform an iterative deepening step, generating a large number of clauses from backtracking, and inserting clauses continuously. Before increasing the depth limit, we first query the SAT solver to check the current status. This appears to be a good tradeoff between reporting unsatisfiability early, and wastefully querying the solver.

4.2 Grounding Schemes

There are a large number of possible choices for the grounding scheme σ_G, and in fact using a whole family of grounding schemes to ground each clause multiply is sound, if potentially wasteful. The simplest scheme is to map every variable to a fresh constant, and in fact this works quite well immediately. Schulz [44] suggests choosing the most frequent constant from the conjecture, and we use this suggestion here, achieving a slight increase in performance over the simple scheme. If there is no constant in the conjecture, we fall back to the fresh constant.

4.3 SAT Solving

SAT *solving*, rather than the grounding procedure or backtracking search, is by far the biggest bottleneck in the resulting system. Additionally, the SAT instances generated by our approach are quite unusual: there are a large number of propositional variables, but conflicts are relatively rare until the clause set becomes unsatisfiable. Further, when new clauses are added, the existing model can often be extended to satisfy the new clauses without backtracking. When the clauses do become unsatisfiable, the unsatisfiable core is typically fairly small compared to the clause space.

After some initial experimentation with an off-the-shelf solver, PicoSAT [7], we found that *in this specific case* we can improve performance by implementing a custom SAT routine. We stress that we do not claim to improve on e.g. PicoSAT's general SAT performance or any similar claim. The custom routine is a more-or-less standard CDCL solver, with the following tweaks:

– The only possible mode is incremental.
– The next decision variable is always chosen as the unassigned variable first produced from proof search. This is both cheap to implement and difficult to beat with more sophisticated heuristics such as VSIDS, we hypothesise because variables introduced sooner are "closer to the conjecture".
– Conflict analysis backtracks through (and possibly resolves with) the entire trail, effectively restarting after every conflict. Since conflicts happen rarely, but it is critical that forced variables are assigned as soon as possible to avoid more conflicts later, this seems to be a good tradeoff in practice.
– The solver does not automatically restart on receiving new clauses. First, it tries to satisfy the new clauses by extending the current assignment, and only if a conflict is reached does it restart.
– Since conflicts are rare and the clause space is already huge, no effort is made to delete the relatively-small number of learned clauses.

4.4 A Note on Proofs

Connection tableau systems have access to an obvious and explicit proof object, the closed tableau. Typically this is also the smallest such with respect to the iterative deepening condition. Unfortunately, this is not the case here: to write a proof we must first obtain an unsatisfiable core (not necessarily minimal, but the smaller the better) from the SAT solver. By storing both the first-order atom that corresponds to a propositional variable, and the first-order premise that was instantiated to a propositional clause, an unsatisfiable set of ground instantiations of first-order clauses can be reported in exchange for a small amount of memory. These can be transformed into a proof by a ground reasoning system.

5 Randomisation and Depth Control

Randomisation of the search order is known to markedly increase performance of connection systems in the presence of restricted backtracking, exploited to great effect in the randoCoP system [36]. The idea here is roughly that if restricted backtracking renders a connection system unable to close a tableau, changing the order of clauses or the order of literals within those clauses may help as a different part of search space is explored. We found a modification of this idea particularly helpful for SATCoP and further allows a powerful depth-control heuristic.

randoCoP randomises both the order of premises and the order of literals within clauses, then runs the leanCoP-based core uninterrupted on the resulting problem, restarting from scratch frequently. Restarting from scratch is not so helpful in our case as we lose the propositional information we have worked so hard to achieve. It can also be wasteful with very large axiom sets as the entire set must be shuffled repeatedly, even though most will not be touched.

Instead, we take an ad-hoc randomisation approach: when there is a list of literals or clauses to be tried, we shuffle them[3]. We shuffle the order of literals in *start* or *extension* clauses, and also the order in which extensions are tried. The order in which the path is traversed looking for reductions is another possible shuffling area, but this does not seem to make much difference in practice.

Randomising search means that it is very likely that after an iterative deepening step generates some propositional clauses, running another iterative deepening step *at the same level* will still yield more propositional clauses from a different part of the search space found by randomisation. This feature of search suggests an optimisation: remain at the same iterative deepening level until no more new propositional clauses are found. As the next iterative deepening level has potentially exponentially many more states to explore, only increasing the search depth when absolutely necessary can be helpful.

6 Model-Based Lemmata

Our final technique is perhaps the most interesting, but easiest to explain. In order to reach an unsatisfiable set of ground clauses, the SAT solver's model must be forced to change until no more models are available. With this in mind, if we have a goal literal G at the leaf of a connection tableau, and its corresponding propositional literal is assigned false in the current model, refuting it will not change the model and is wasted effort from this perspective. To avoid this, we consider ground literals that are assigned false at the SAT level to be solved and skip them, in a similar way to the *lemmata* refinement for connection tableaux.

[3] pseudo-random shuffle such that results are reproducible

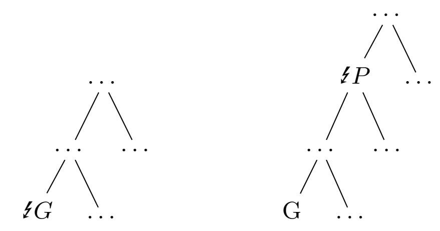

Fig. 2. If the abstraction of a ground goal G is assigned false in the current SAT model, refuting it can be skipped, as this will not force a change in the model. Generalising, if any ground path literal P is assigned false, the whole sub-tableau can be skipped.

We call this technique "model-based lemmata" due to this similarity, but the effect on proof search is not as clear. Literals may change assignment several times during proof search, although if refuted by a sub-tableau the literal will be *forced* false. Further, it is no longer sound to consider closed tableaux as a proof, as they may contain ground literals that have been skipped and therefore we can rely only on the SAT solver reporting unsatisfiability. An interesting side-effect is that iterative deepening steps do not take as long due to skipped literals: this may well have a positive effect on proof search by itself.

There is also a natural generalisation of this idea which we implement: if there are path literals $P_1, P_2, \ldots P_n$ available and the goal literal is G, we essentially try to refute the conjunction $P_1 \wedge P_2 \wedge \ldots \wedge P_n \wedge G$, or to show $P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow \neg G$ if you prefer. If *any* of the path literals P_i become ground and assigned false through unification, these can also be skipped, closing an entire sub-tableau. The general idea is illustrated in Figure 2.

7 First Impressions

Algorithm 2 extends that given in Algorithm 1 with the additions discussed in Sections 4–6. New lines are marked with a $\rightarrow$. The resulting system is implemented in Rust and is available online[4].

Initial impressions of the resulting system are positive. Compared to the baseline system the most obvious change is an increase in memory use (SAT data and mapping information has to be kept somewhere), but this is not typically excessive, and is comparable to saturation systems. The majority of problems that the baseline system solved can now be solved in fewer steps, which typically also results in a shorter time-to-proof.

Practical performance on other problems also appears improved, particularly in cases where the SAT approach is very helpful. `PUZ010-1`, "who owns the zebra?" from the TPTP library contains a large number of nearly-ground axioms and a completely ground conjecture formed from a large disjunction of literals.

[4] https://github.com/MichaelRawson/satcop
commit `65122a99e08648f5b2e331280d0a0011e73a0836` is discussed here

Algorithm 2: sketch of the exended SATCoP search routine

```
   $\sigma_U = \emptyset$; // global tableaux-level unifier, modified by unify()
   limit = 0; // depth limit for iterative deepening
→  ground = $\emptyset$; // set of propositional clauses produced so far
→  new = $\emptyset$; // new propositional clauses produced this iteration
→  model = $\emptyset$; // partial propositional model of ground
   **function** *start() : bool* **is**
       // add start clauses to the grounding
→      **foreach** *clause* $\in$ *start clauses* **do**
→          ground = ground $\cup$ $\{(clause)\sigma_G\}$;
       **loop**
           **foreach** $C \in$ *start clauses* **do**
→              shuffle $C$;
→              prove-all($\epsilon$, $C$);
               $\sigma_U = \emptyset$; // reset $\sigma_U$
→          **if** (new $\setminus$ ground) $\neq \emptyset$ **then**
→              ground = ground $\cup$ new;
→              new = $\emptyset$
           **else**
               limit = limit + 1; // only increase limit if no new clauses
→          **if** *there is a* model *satisfying* ground **then**
→              set model
           **else**
               // unsat propositional clauses: found a proof!
→              **return** *true*
   **function** *prove-all(path, clause) : bool* **is**
       **foreach** *literal* $\in$ *clause* **do**
           **if** $\neg$*prove(path, literal)* **then** **return** *false* ;
       **return** *true*
   **function** *prove(path, goal)* **is**
       // model-based lemmata
→      **foreach** $L \in$ path $\cup$ {goal} **do**
→          **if** $(L)\sigma_U$ *is ground and assigned false in* model **then return** *true*;
       **foreach** $L \in$ *path* **do**
           **if** *sign(goal)* $\neq$ *sign*($L$) **and** *unify(goal, $\neg L$)* **then**
→              ground all clauses in the tableau and add them to new;
               **return** *true*
       // limit search depth
       **if** |path| $\geq$ limit **then** **return** *false* ;
       $\sigma'_U = \sigma_U$;
→      **foreach** *fresh copy C of a problem clause in random order* **do**
→          shuffle $C$;
           **foreach** $L \in C$ **do**
               **if** *sign(goal)* $\neq$ *sign*($L$) **and** *unify(goal, $\neg L$)* **then**
→                  ground all clauses in the tableau and add them to new;
                   **if** *prove-all(append(path, goal), $C \setminus \{L\}$)* **then** **return** *true* ;
                   $\sigma_U = \sigma'_U$; // reset $\sigma_U$ to try again
                   **continue**
```

The unaided system cannot solve this problem in reasonable time[5], but the SAT-assisted system solves it near-instantaneously, producing a proof consisting of 322 grounded clauses.

It is not only problems tailor-made for SAT, either. `GRP001-2` is a unit equality version of a problem from group theory, "if the square of every element is identity, the system is commutative". This problem is much easier for rewriting systems that specially handle equality: Vampire solves this immediately, but the baseline system cannot solve it at all. However, with the enhancements described, SATCoP solves this in 4 seconds with no specialised equality handling.

8 Experimental Evaluation

We run two experiments to determine the practical effect of the preceding work. The first runs various configurations of SATCoP to evaluate different techniques from Sections 4–6 against each other. The second compares SATCoP against other systems. All experiments are run on a desktop machine clocked at 3.4GHz.

8.1 System Configurations

We run the state-of-the-art saturation system Vampire [26] 4.5.1, and the strong connection system leanCoP [31] 2.1[6] to provide a comparison. Both of these systems expose options which can drastically alter proof search, and further both provide *portfolio modes* in which a number of different option combinations are tried in sequence. Inventing and evaluating good portfolios is a hard problem in itself, which we avoid here by running all systems with a fixed set of options: we stress that the results presented here do not necessarily reflect the "competition strength" of a system. Vampire runs in its default mode, which entails a limited resource strategy [41], AVATAR [49], and a number of other search parameters. leanCoP was configured with `[cut,conj]` — that is, a restricted backtracking strategy, starting from clauses relating to conjectures — which more closely reflects SATCoP_0's strategy, but may not be the strongest available.

In the presence of large axiom sets containing extraneous axioms, saturation systems can sometimes choke. SInE [19] heuristically selects some subset of axioms that may be relevant for proving a conjecture, which can significantly accelerate proof search, provided that no necessary axiom is removed. Vampire (SInE) runs SInE-style axiom selection with an additional flag.

8.2 Results and Discussion

We use 1-second runs on the *M2k* set of 2003 problems throughout development to quickly gauge practical effectiveness. Table 1 shows the effect produced by

[5] It is interesting to note that the saturation-based Vampire theorem prover also fails to solve this problem in reasonable time without support from a SAT solver.

[6] run with SWI Prolog 7.6.4 [50]

Table 1. Problems from the *M2k* set solved in 1 second by all possible combinations of techniques. "grounding" is the method described in Section 4, "shuffle" the ad-hoc randomisation described in Section 5, "depth control" the modification of iterative deepening presented in the same section, and "model lemmata" the topic of Section 6.

grounding	shuffle	depth control	model lemmata	solved
				886
✓				998
	✓			957
✓	✓			1135
✓	✓	✓		1173
✓			✓	1061
✓	✓		✓	1189
✓	✓	✓	✓	1252

Table 2. Problems solved in 10 seconds by existing systems and SATCoP on a variety of first-order benchmark sets. SATCoP$_0$ is SATCoP without any of the techniques described — i.e. a more standard connection system — for direct comparison.

	TPTP		bushy		chainy	
	solved	unique	solved	unique	solved	unique
Vampire	3650	388	1162	132	402	6
Vampire (SInE)	3013	258	781	34	550	109
leanCoP	1946	22	648	18	272	7
SATCoP$_0$	1837	8	564	0	221	0
SATCoP	3049	282	953	52	505	101
benchmark size	7609		2078		2078	

different combinations of the techniques discussed here. Note that some combinations are omitted as nonsensical: for example, it is not possible to control iterative deepening as in Section 5 without grounding clauses, and without randomisation it is possible but provides no benefit.

We are pleased that the union of all techniques described performs the best, and that all produce some amount of benefit. It is interesting to note that some *combinations* are disproportionately effective, suggesting a synergising effect. Grounding clauses and randomisation gain 112 and 71 problems respectively over SATCoP$_0$, but combined gain 249. One might conjecture about *why* this happens — perhaps randomisation produces a larger number of ground clauses and thereby increases the likelihood of unsatisfiability — but in any event the outcome is encouraging.

We now compare our final system SATCoP against SATCoP$_0$ and other representative systems. We allow a 10-second time limit and evaluate the TPTP, "bushy" and "chainy" problem sets discussed in Section 2.4. Table 2 shows these data: the "solved" column is the number of problems solved for a given solver/set combination, while “unique” is the number of problems in a set *only that system and no other* solved.

9 Conclusions and Future Directions

We are pleasantly surprised at the improvement in performance achieved by very simple application of ground reasoning techniques to a connection tableau system. Further performance improvements can be obtained for relatively little effort using the existing ground information, which we demonstrate through final evaluation on a number of benchmark problem sets.

While the resulting system is not quite as concise as some of the beautiful systems achieved in Prolog, it is certainly effective and remains compact compared to state-of-the-art saturation systems. It is also possible that future investigations could make use of the “lean Prolog technology” approach, combined either with a Prolog implementation of CDCL [42], Prolog bindings to an existing SAT solver [13], or even (with some modification) constraint logic programming [20].

The SAT world also merits further investigation: SAT instances generated by our system are relatively unusual, and are mostly easily-satisfiable, until very suddenly they are not. A WalkSAT-like solver with some amount of clause learning [30] may improve SAT-level performance. SMT is another interesting direction, particularly for the theory of equality and uninterpreted functions. Application to other logics is a related topic we would like to investigate further: some seem quite achievable, such as some kind of support for arithmetic theories, but we acknowledge that intuitionistic logic may present a challenge.

9.1 A Note From the Future

Since submission, we have been busy preparing SATCoP for competition at CASC-28. Some ideas were found to further improve performance from that reported here. We report these modifications here both for interest and to document them in context for the competition.

- Our custom SAT routine is fast on the type of incremental SAT problems generated by SATCoP, but is not a good general SAT routine. We implement a new routine which first tries a few rounds of stochastic local search, then falls back to PicoSAT if we fail to find a satisfying assignment. This makes the common case very fast, allows solving the harder SAT problems quickly, and is much simpler than the approach described above.
- This improved routine allows us to continuously solve the SAT problem as clauses are added, rather than at each iterative deepening step.
- We restrict application of “model-based lemmata” to ground literals above. We can relax this restriction, allowing a sort of “literal selection” technique in which the first goal literal assigned true from a clause is attempted.
- Multiple CPU cores can be usefully occupied by launching multiple proof search attempts with different pseudo-random seeds.

References

1. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., Urban, J.: Premise selection for mathematics by corpus analysis and kernel methods. Journal of Automated Reasoning **52**(2), 191–213 (2014)
2. Andrews, P.B.: Theorem proving via general matings. Journal of the ACM (JACM) **28**(2), 193–214 (1981)
3. Balyo, T., Froleyks, N., Heule, M.J., Iser, M., Järvisalo, M., Suda, M.: Proceedings of SAT Competition 2020: Solver and benchmark descriptions (2020)
4. Baumgartner, P., Tinelli, C.: The model evolution calculus. In: International Conference on Automated Deduction. pp. 350–364. Springer (2003)
5. Bayerl, S., Letz, R.: SETHEO: A sequential theorem prover for first-order logic. Esprit'87-Achievements and Impacts, part **1**, 721–735 (1987)
6. Bibel, W.: Automated theorem proving. Springer Science & Business Media (2013)
7. Biere, A.: PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation **4**(2-4), 75–97 (2008)
8. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking (2003)
9. Biere, A., Ganesh, V., Grohe, M., Nordström, J., Williams, R.: Theory and practice of SAT solving (Dagstuhl Seminar 15171). In: Dagstuhl Reports. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
10. Biere, A., Heule, M., van Maaren, H.: Handbook of satisfiability, vol. 185. IOS press (2009)
11. Brown, C.E.: Reducing higher-order theorem proving to a sequence of SAT problems. Journal of Automated Reasoning **51**(1), 57–77 (2013)
12. Claessen, K., Sorensson, N.: New techniques that improve MACE-style model finding. In: Model Computation (2003)
13. CODISH, M., LAGOON, V., STUCKEY, P.J.: Logic programming with satisfiability. Theory and Practice of Logic Programming **8**(1), 121 (2008)
14. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Communications of the ACM **54**(9), 69–77 (2011)
15. Deshane, T., Hu, W., Jablonski, P., Lin, H., Lynch, C., McGregor, R.E.: Encoding first order proofs in SAT. In: International Conference on Automated Deduction. pp. 476–491. Springer (2007)
16. Färber, M.: A curiously effective backtracking strategy for connection tableaux. CoRR **abs/2106.13722** (2021), https://arxiv.org/abs/2106.13722
17. Färber, M., Kaliszyk, C., Urban, J.: Machine learning guidance for connection tableaux. Journal of Automated Reasoning **65**(2), 287–320 (2021)
18. Grabowski, A., Korniłowicz, A., Naumowicz, A.: Four decades of Mizar. Journal of Automated Reasoning **55**(3), 191–198 (2015)
19. Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: International Conference on Automated Deduction. pp. 299–314. Springer (2011)
20. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 111–119 (1987)
21. Kaliszyk, C.: Efficient low-level connection tableaux. In: International Conference on Automated Reasoning with Analytic Tableaux and Related Methods. pp. 102–111. Springer (2015)
22. Kaliszyk, C., Urban, J., Michalewski, H., Olšák, M.: Reinforcement learning of theorem proving. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems. pp. 8836–8847 (2018)

23. Kaliszyk, C., Urban, J., Vyskočil, J.: Certified connection tableaux proofs for HOL Light and TPTP. In: Proceedings of the 2015 Conference on Certified Programs and Proofs. pp. 59–66 (2015)
24. Korovin, K.: Instantiation-based automated reasoning: From theory to practice. In: International Conference on Automated Deduction. pp. 163–166. Springer (2009)
25. Korovin, K.: Inst-Gen — a modular approach to instantiation-based automated reasoning. In: Programming Logics, pp. 239–270. Springer (2013)
26. Kovács, L., Voronkov, A.: First-order theorem proving and VAMPIRE. In: International Conference on Computer Aided Verification. pp. 1–35. Springer (2013)
27. Letz, R., Stenz, G.: Model elimination and connection tableau procedures. In: Handbook of Automated Reasoning, pp. 2015–2114. Elsevier (2001)
28. Loveland, D.W.: Mechanical theorem-proving by model elimination. In: Automation of Reasoning, pp. 117–134. Springer (1968)
29. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5), 506–521 (1999)
30. McDonald, A., et al.: Parallel WalkSAT with clause learning. Data analysis project papers (2009)
31. Otten, J.: leanCoP 2.0 and ileanCoP 1.2: High performance lean theorem proving in classical and intuitionistic logic (system descriptions). In: International Joint Conference on Automated Reasoning. pp. 283–291. Springer (2008)
32. Otten, J.: Restricting backtracking in connection calculi. AI Communications **23**(2-3), 159–182 (2010)
33. Otten, J.: MleanCoP: A connection prover for first-order modal logic. In: International Joint Conference on Automated Reasoning. pp. 269–276. Springer (2014)
34. Otten, J.: nanoCoP: A non-clausal connection prover. In: International Joint Conference on Automated Reasoning. pp. 302–312. Springer (2016)
35. Otten, J.: The pocket reasoner — automatic reasoning on small devices. In: Norwegian Informatics Conference, NIK (2018)
36. Raths, T., Otten, J.: randoCoP: Randomizing the proof search order in the connection calculus. In: First International Workshop on Practical Aspects of Automated Reasoning. pp. 94–103 (2008), http://ceur-ws.org/Vol-373/
37. Reger, G., Bjorner, N., Suda, M., Voronkov, A.: AVATAR modulo theories. In: Benzmüller, C., Sutcliffe, G., Rojas, R. (eds.) GCAI 2016. 2nd Global Conference on Artificial Intelligence. EPiC Series in Computing, vol. 41, pp. 39–52. EasyChair (2016). https://doi.org/10.29007/k6tp, https://easychair.org/publications/paper/7
38. Reger, G., Suda, M.: The uses of SAT solvers in Vampire. In: Kovács, L., Voronkov, A. (eds.) Proceedings of the 1st and 2nd Vampire Workshops, Vampire@VSL 2014, Vienna, Austria, July 23, 2014 / Vampire@CADE 2015, Berlin, Germany, August 2, 2015. EPiC Series in Computing, vol. 38, pp. 63–69. EasyChair (2015), https://easychair.org/publications/paper/ZG9
39. Reger, G., Suda, M.: Global subsumption revisited (briefly). In: Kovacs, L., Voronkov, A. (eds.) Vampire 2016. Proceedings of the 3rd Vampire Workshop. EPiC Series in Computing, vol. 44, pp. 61–73. EasyChair (2017). https://doi.org/10.29007/qcd7, https://easychair.org/publications/paper/QDj
40. Reger, G., Suda, M., Voronkov, A.: Finding finite models in multi-sorted first-order logic. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9710, pp. 323–341. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_20, https://doi.org/10.1007/978-3-319-40970-2_20

41. Riazanov, A., Voronkov, A.: Limited resource strategy in resolution theorem proving. Journal of Symbolic Computation **36**(1-2), 101–115 (2003)
42. Robbins, E., King, A., Howe, J.M.: Backjumping is exception handling. Theory and Practice of Logic Programming pp. 1–20 (2020)
43. Schulz, S.: A comparison of different techniques for grounding near-propositional CNF formulae. In: FLAIRS Conference. pp. 72–76 (2002)
44. Schulz, S.: Light-weight integration of SAT solving into first-order reasoners — first experiments. Vampire pp. 9–19 (2017)
45. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: International Conference on Automated Deduction. pp. 495–507. Springer (2019)
46. Smullyan, R.M.: First-order logic. Courier Corporation (1995)
47. Sutcliffe, G.: The TPTP problem library and associated infrastructure. Journal of Automated Reasoning **43**(4), 337 (2009)
48. Urban, J.: MPTP 0.2: Design, implementation, and initial experiments. Journal of Automated Reasoning **37**(1-2), 21–43 (2006)
49. Voronkov, A.: AVATAR: The architecture for first-order theorem provers. In: International Conference on Computer Aided Verification. pp. 696–710. Springer (2014)
50. Wielemaker, J.: SWI-Prolog version 7 extensions. In: Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments. vol. 109. Citeseer (2014)
51. Zombori, Z., Urban, J., Brown, C.E.: Prolog technology reinforcement learning prover. In: International Joint Conference on Automated Reasoning. pp. 489–507. Springer (2020)