Twee: An Equational Theorem Prover (System Description)

Nicholas Smallbone[0000−0003−2880−6121]

Department of Computer Science and Engineering,
Chalmers University of Technology, Gothenburg, Sweden
nicsma@chalmers.se

Abstract. Twee is an automated theorem prover for equational logic. It implements unfailing Knuth-Bendix completion with ground joinability testing and a connectedness-based redundancy criterion. It came second in the UEQ division of CASC-J10, solving some problems that no other system solved. This paper describes Twee's design and implementation.

Keywords: Automated theorem proving · unit equality · completion

1 Introduction

Twee is an automated theorem prover for equational logic, available as open-source software [17]. It features good performance (coming second in the UEQ division of CASC-J10), low memory use, and human-readable proof output.

Twee's general architecture is quite traditional: it uses a DISCOUNT loop [7] implementing unfailing Knuth-Bendix completion [3]. However, it has a few characteristics which are unusual in a high-performance theorem prover:

Fixed heuristics. Twee does not adjust its strategy based on the input problem. It uses a fixed term order, a fixed critical pair scoring function, and so on. Rather than detecting the kind of problem, Twee uses general-purpose strategies that work for all sorts of problems (Section 2).

Strong redundancy tests. Rather than using special strategies for associative-commutative functions, Twee builds in strong redundancy tests, based on ground joinability and connectedness (Section 3). These handle not just AC functions but many kinds of unorientable equations, in particular permutative ones (where both sides are almost the same but with variables in a different order).

A high-level language. Twee consists of 5300 lines of Haskell code, whereas for example Waldmeister [12] is 65000 lines of C. As such, it is easy to experiment with. Despite the choice of programming language, Twee is quite fast at raw deduction steps, thanks to careful coding of low-level term operations (Section 4).

Despite the fixed heuristics and high-level language, Twee comes close in performance to E [14] and Waldmeister [12]. It is strong in many problem classes,

including `LAT` (lattices) and `REL` (relation algebra) from TPTP, which feature many commutative operators where Twee's redundancy tests shine, and on unusual problems, where no prover has special heuristics. Twee is however poor at `RNG` (rings), where it seems important to choose a good term order. The rest of the paper describes Twee's design in detail, focusing on the three aspects above.

Notation. We use $t \equiv u$ to mean that t and u are syntactically equal.

2 Architecture

Twee natively supports only unit equality problems with ground goals, but the frontend also supports arbitrary quantification, Horn formulas, and many-sorted logic. These features are eliminated using the external tool Jukebox [16], which:

- Clausifies the problem to eliminate conjunction and quantifiers.
- Encodes Horn clauses as equations [5].
- Encodes sorts using extra functions [4].

At this point, the goal can still contain existentially-quantified variables, which must be eliminated. To do so, we use an old trick, also used by Waldmeister: if the goal $t = u$ is non-ground, we add new function symbols *eq*, *true* and *false*, and two axioms $\forall X.\ \mathit{eq}(X, X) = \mathit{true}$ and $\mathit{eq}(t, u) = \mathit{false}$, and replace the goal with $\mathit{true} = \mathit{false}$. Now we have a unit equality problem with a ground goal.

The main proof loop is shown in Algorithm 1. It implements unfailing completion [3] using a DISCOUNT loop [7]. The state consists of R, a set of rewrite rules and unorientable equations (the *active set*, initially empty); Q, the set of unprocessed critical pairs formed from R (the *passive set*, initially containing all the axioms); J, a set of ground joinable equations used for subsumption checking (following [1]); and the goal. The main loop removes the best critical pair from Q (see below), and if it is not redundant, adds it to R (oriented if possible) and adds all its critical pairs to Q. Every so often, the rules in R are reduced with respect to one another and redundant rules are removed. The goal is kept normalised with respect to R and the prover succeeds if the goal becomes trivial.[1]

The passive set is normally quadratic in the size of the active set: typical numbers are $|R| \approx 10,000$ and $|Q| \approx 10,000,000$. Hence we must process each passive critical pair at high speed, but can spend time on each new rewrite rule.

Term ordering. We always use KBO, with all functions having weight 1, and ordered so that more frequently-occuring functions are smaller.

Critical pair selection. When a critical pair is added to Q, it is first normalised and then assigned a score; the proof loop selects the critical pair with the lowest score. The score function's job is to pick out promising critical pairs, and the choice of score function can make or break the prover. However, as it is applied to every critical pair, it also needs to be fast. We compute scores as follows:

[1] An equation is considered trivial if it is of the form $t = t$.

Algorithm 1 The main proof loop

```
(R, J, Q) = (∅, ∅, A)
while Q ≠ ∅ do
  P = remove lowest-scoring element of Q
  if P's parent rules are still present in R then
    normalise P using R to get t = u
    if t ≱ u and t = u is not connected and t = u is not subsumed by J then
      if t = u is ground joinable then
        add t = u to J
      else
        orient t = u and add it to R
        for all critical pairs cp of t = u and R do
          normalise cp using only the oriented rules in R
          if cp is non-trivial then add cp to Q end if
        end for
        normalise goal using R
        if goal is trivial then return "theorem" end if
        simplify rules in R wrt each other, but limit this step to 5% of total runtime
      end if
    end if
  end if
end while
return "countersatisfiable"
```

- We start with a weighted sum of the size of the two terms. By default we take $4\,\mathrm{weight}(t) + \mathrm{weight}(u)$, where t is the bigger term and u the smaller. In other words, the size of the bigger term is most important. Variables are weighted slightly less than function symbols, to encourage finding more general rules.
- To encourage Twee to use all the axioms, we add the critical pair's *depth*, where axioms have depth 0, critical pairs of the axioms have depth 1, etc.
- If a term contains the same subterm multiple times, only one occurrence of that subterm is counted; the other occurrences get a nominal weight of one symbol. In effect we measure the weight as if the term was a DAG rather than a tree. The idea is that identical subterms form the same critical pairs, and tend to get rewritten at the same time: they come and go together.
- Finally, any critical pair of the form $eq(v, w) = \mathit{false}$ (where eq is the function used to encode existential goals) with v and w unifiable is given a fixed cost of 1, because selecting it will immediately prove the goal. This trick is also used by Waldmeister, and is vital in practice for existential goals.

Proof production and checking. Twee uses an LCF-style kernel [9] to guarantee soundness. Every member of the active set comes with a proof object, which is verified by a trusted proof checker (consisting of about a page of code). The proofs are low-level and thus easy to check: the only proof steps allowed are reflexivity, symmetry, transitivity, congruence and applying an axiom or lemma. It is not possible to add a rule to the active set without supplying a proof, and

any invalid proof step causes a fatal runtime error. The key to making this fast is that only the active set, not the passive set, includes proof objects.

Once the goal is proved, we transform the proof object into a human-readable proof, consisting of a flat sequence of rewrite steps. We also introduce lemmas, to avoid exponentially-sized proofs: any active rewrite rule is a candidate lemma. Our approach is similar to [8], but simpler as our proof steps are smaller; but their lemma selection strategy is smarter than ours and produces fewer lemmas.

Goal transformation. Twee's frontend can optionally transform the problem to make the prover more goal-directed. The transformation is simple, but strange. For every function term $f(\ldots)$ appearing in the goal, we introduce a fresh constant symbol a and add the axiom $f(\ldots) = a$. For example, if the goal is $f(g(a), b) = h(c)$, we add the axioms $f(g(a), b) = d_1$, $g(a) = d_2$, and $h(c) = d_3$. Simplification will rewrite the first axiom to $f(d_2, b) = d_1$ and the goal to $d_1 = d_3$.

By doing this transformation, (1) any subterm of the goal gets normalised to a constant, so critical pairs containing goal terms get a lower score, and (2) new critical pairs involving these constants appear, which are likely to be relevant to the goal. We evaluate this transformation in Section 5.

Weak rewrite rules. Completion sometimes deduces equations where both sides have a variable not occurring on the other side, such as $f(x, y) = g(x, z)$. Such equations are awkward for rewriting: suppose we want to use this equation to rewrite the term $f(t, u)$—what value should we choose for z?

Twee splits this equation into nicely-behaved rewrite rules instead. To do so, we introduce the concept of a *weak rewrite rule*. A weak rewrite rule $t \rightsquigarrow u$ is like an ordinary rewrite rule, except that it only satisfies $t \succcurlyeq u$, not $t \succ u$.[2] Weak rewrite rules form critical pairs and participate in rewriting just like any other rewrite rule, except that to ensure termination, we may only perform the rewrite step $t\sigma \rightsquigarrow u\sigma$ if $t\sigma \not\equiv u\sigma$, i.e. $t\sigma$ and $u\sigma$ are syntactically different terms.[3]

Using weak rewrite rules, Twee splits $f(x, y) = g(x, z)$ into the two rules $f(x, y) \to g(x, \bot)$ and $g(x, z) \rightsquigarrow g(x, \bot)$, where $\bot$ is the *minimal* term in the term ordering. Note that $g(x, z) \rightsquigarrow g(x, \bot)$ is a valid weak rewrite rule because $g(x, z) \succcurlyeq g(x, \bot)$, with equality exactly when $z \equiv \bot$.

As another example, the equation $f(x, x, y, z) = g(x, y, y, w)$ is split into $f(x, x, y, \bot) = g(x, y, y, \bot)$, $f(x, x, y, z) \rightsquigarrow f(x, x, y, \bot)$ and $g(x, y, y, w) \rightsquigarrow g(x, y, y, \bot)$. In this case, we are still left with an unorientable rule afterwards, but since it has the same variables on both sides it is unproblematic for rewriting.

It is always possible and safe to split an equation into an equivalent set of:

- ordinary rewrite rules $t \to u$ with $t \succ u$,
- weak rewrite rules $t \rightsquigarrow u$ with $t \succcurlyeq u$, and
- unorientable equations $t = u$ where both sides have the same set of variables.

Twee does this whenever an equation is about to be added to R.

[2] $t \succcurlyeq u$ means: for all grounding substitutions σ, either $t\sigma \succ u\sigma$ or $t\sigma \equiv u\sigma$.

[3] This is different from e.g. constrained rewriting: we can perform the rewrite even if t and u are unifiable, as long as they are not the same term right now.

3 Redundancy Criteria

The basic redundancy criterion of Knuth-Bendix completion is *joinability*: a critical pair can be discarded if both sides normalise to the same term. Joinability runs into problems when we have unorientable equations. For example, consider a rewrite system for an associative-commutative operator "+":

$$x + y = y + x \tag{1}$$
$$(x + y) + z \to x + (y + z) \tag{2}$$
$$x + (y + z) = y + (x + z) \tag{3}$$

From (1) and (2) we get the critical pair $x+(y+z) \xleftarrow{(2)} (x+y)+z \xrightarrow{(1)} z+(x+y)$, which cannot be rewritten any further so it is not joinable. However, the critical pair is redundant, because the above rewrite system is ground confluent. We would like to detect redundant but non-joinable critical pairs.

This section presents the redundancy criteria that Twee uses to handle unorientable equations: our take on the well-known approach of ground joinability testing [6], and a novel (we believe) approach based on connectedness [2]. Unlike the standard techniques for associative-commutative functions [1], our criteria handle any kind of permutative equation; we evaluate our approach in Section 5.

3.1 Ground Joinability Testing

Although the critical pair $x + (y + z) \leftarrow (x + y) + z \to z + (x + y)$ is not joinable, all ground instances of it are joinable, and we say that the critical pair is ground joinable. For example, the instance $a + (b + c) \leftarrow (a + b) + c \to c + (a + b)$, with $a \prec b \prec c$, can be joined since $c + (a + b) \xrightarrow{(3)} a + (c + b) \xrightarrow{(1)} a + (b + c)$. Any ground joinable critical pair is redundant.

Martin and Nipkow [13] suggest an approach for checking ground joinability:

- Consider all possible orderings between the variables of the critical pair, such as $x \prec y \prec z$, $y \prec x \equiv z$, $x \equiv y \prec z$, and so on.
- For each ordering, show that the critical pair is joinable when the variables have that order. Formally, this means showing that all ground instances satisfying the ordering are joinable. For example, the rewrite proof above shows our critical pair joinable for any ground instance satisfying $x \prec y \prec z$.

Their algorithm effectively does a case analysis on all possible variable orderings, but it is inefficient because there are so many possible orderings.

Our algorithm is similar, but tries to minimise the number of cases it considers. It does so by allowing orderings that: (1) constrain only a subset of the variables, such as $x \prec y$, and (2) use $\preccurlyeq$, as in $x \preccurlyeq y \prec z$. It works as follows:

1. Choose a strict total order on *all* the variables, using *only* $\prec$; e.g., $x \prec y \prec z$.
2. Show that the critical pair is joinable under that ordering. Formally, we show that all *ground instances* satisfying the ordering are joinable.

3. We have now shown that the critical pair is joinable in one specific case. Now *generalise* that case, by: (1) removing variables from the ordering, and (2) replacing $\prec$ with $\preccurlyeq$ in the ordering, as long as the critical pair is joinable under the resulting ordering. (This may e.g. generalise $x \prec y \prec z$ to $x \preccurlyeq z$.)
4. Repeat, but pick an ordering that is not covered by any of the cases so far.
5. When all variable orderings involving only $\prec$ have been covered, all the ones that remain must involve $\equiv$. For each such ordering, take the critical pair, unify all equal variables, and recursively call the ground joinability check.

Example. Take the critical pair $x+(y+z) \leftarrow (x+y)+z \rightarrow z+(x+y)$ and suppose that we choose the ordering $x \prec y \prec z$. It can be joined when this order holds, as for any instance where $x \prec y \prec z$, we have $z+(x+y) \xrightarrow{(3)} x+(z+y) \xrightarrow{(1)} x+(y+z)$.

Having joined the critical pair in one case, we now generalise the case. We first try to remove each variable in turn, i.e. to join the critical pair in the three cases $x \prec y$, $y \prec z$, and $x \prec z$ in turn. None of these attempts succeeds.

Now we try replacing a $\prec$ with a $\preccurlyeq$, to get $x \prec y \preccurlyeq z$. We must check if all ground instances satisfying $x \prec y \preccurlyeq z$ are joinable, but how? We might think of splitting this into two cases $x \prec y \prec z$ and $x \prec y \equiv z$, but instead we are going to find *one rewrite proof* that works for both.

Consider the rewrite proof above. In it, the step $x + (z + y) \rightarrow x + (y + z)$ is fine if $y \prec z$, but does not seem to be allowed if $y \equiv z$. But in fact it is fine: if $y \equiv z$, the terms $x+(z+y)$ and $x+(y+z)$ are identical, so this rewrite step does nothing and can just be dropped. That is, the proof works both when $x \prec y \prec z$ and $x \prec y \equiv z$, and shows joinability for the case $x \prec y \preccurlyeq z$. We generalise the other $\prec$ similarly, showing that the critical pair is joinable in the case $x \preccurlyeq y \preccurlyeq z$.

Next, we pick another total order on the variables, but not one in which $x \preccurlyeq y \preccurlyeq z$. We might pick, for example, $z \prec y \prec x$. The process repeats: we show ground joinability under this ordering, and generalise it to $z \preccurlyeq y \preccurlyeq x$. We repeat until all cases are covered, and the ground joinability test succeeds.

Although our algorithm can be expensive in theory, in practice it needs to consider only a few orderings, and a small number of variables. Step (5) can occasionally be expensive, but by generalising $\prec$ to $\preccurlyeq$ we can usually avoid it.

The general case. Here is how we test joinability under a given variable ordering. First, we parameterise our term order. Given an ordering C, we define $t \succcurlyeq_C u$ to mean that, for all grounding substitutions σ, if σ satisfies C then $t\sigma \succcurlyeq u\sigma$.

In the example, we weakened a $\prec$ to a $\preccurlyeq$. To do so, we used a rewrite step that, in some ground instances, rewrote a term to *the same term*. To allow these kind of steps, we loosen our definition of rewriting: we may perform a rewrite $t \rightarrow u$ under C as long as $t \succcurlyeq_C u$ and $t \not\equiv u$. Rewriting terminates because given a rewrite proof $t \succcurlyeq_C u \succcurlyeq_C v \succcurlyeq_C \ldots$, there is always a ground instance where $t' \succ_C u' \succ_C v' \succ_C \ldots$, since C was constructed as a strict order in step (1).

With this definition, normalising $z+(x+y)$ using the ordering $C := x \preccurlyeq y \preccurlyeq z$ yields $z + (x + y) \rightarrow x + (z + y) \rightarrow x + (y + z)$, where e.g. the first step is allowed because $z + x \succcurlyeq_C x + z$ and $z + x \not\equiv x + z$. Thus we can join our example critical pair under a given variable ordering just by normalising both sides, as we want.

The last ingredient is to implement a test for $t \succcurlyeq_C u$, which we have done for KBO. The tricky part is checking whether $\mathrm{weight}(t) \geqslant \mathrm{weight}(u)$, which can be solved by taking the expression $\mathrm{weight}(t) - \mathrm{weight}(u)$, a linear combination of the weights of t's and u's variables, and computing its minimum possible value.

One nice property is that the rest of the ground joining code is independent of the term order. To support e.g. LPO, one just needs to implement $\succcurlyeq_C$ for it.

Why not allow arbitrary ordering constraints? Some critical pairs can only be ground joined by using ordering constraints on arbitrary terms (e.g. $x + y \prec z$). We do not support these, as they make everything enormously more complex:

- The number of possible orderings becomes infinite. You can get stuck enumerating more and more cases of a case split which never ends. In our design, there are finitely many orderings and the algorithm clearly terminates.
- Computing $\succcurlyeq_C$ for KBO becomes NP-complete [10]. In our setting, it takes polynomial time, and we expect it can be done in linear time following [11].

3.2 Connectedness

Ground joinability testing is rather heavyweight, constructing and analysing a sometimes large case split, and sometimes it fails because it only supports case splits on variables. Twee also supports a simpler, complementary method that works well when an unorientable equation is applied *under* another function.

The method makes use of *connectedness*. A critical pair $s \leftarrow t \rightarrow u$ is *connected* if there is a rewrite proof $s = t_1 = \ldots = t_n = u$ such that each t_i is strictly less than t [2]. In Knuth-Bendix completion, any connected critical pair is redundant. In other words, when joining $s \leftarrow t \rightarrow u$, we can do rewrite steps that *increase* the term, as long as the result is always strictly less than t.

Here is how we use connectedness. Let σ be a substitution that grounds s and u. When joining $s \leftarrow t \rightarrow u$, we may want to perform a rewrite step $v \rightarrow w$ using an unoriented equation, but we don't know if $v \succcurlyeq w$. We allow the rewrite step $v \rightarrow w$ as long as: (1) $w \prec t$, and (2) $v\sigma \succ w\sigma$. Condition (1) ensures connectedness, and condition (2) ensures that rewriting eventually terminates.

For example, suppose we take the earlier rules for "+" and add a function f:

$$f(x + y, z + w) \rightarrow f(x, f(z, f(y, w))) \tag{4}$$
$$f(x, f(y, z)) = f(y, f(x, z)) \tag{5}$$

Assume KBO with both f and $+$ having weight 1. One critical pair is $f(y, f(z, f(x, w))) \xleftarrow{(4)} f(y + x, z + w) \xleftarrow{(1)} f(x + y, z + w) \xrightarrow{(4)} f(x, f(z, f(y, w)))$. We can show this to be connected using $\sigma = \{x \mapsto a, y \mapsto b, z \mapsto c, w \mapsto d\}$, $a \prec b \prec c \prec d$. The left term $f(y, f(z, f(x, w)))$ rewrites to $f(y, f(x, f(z, w)))$ using (5), because $f(y, f(x, f(z, w))) \prec f(x + y, z + w)$ (connectedness) and $f(b, f(c, f(a, d))) \succ f(b, f(a, f(c, d)))$ (termination); and that rewrites to $f(x, f(y, f(z, w)))$ similarly. The right term $f(x, f(z, f(y, w)))$ also rewrites to $f(x, f(y, f(z, w)))$. Thus the critical pair is redundant.

In general we try two choices of σ: one where the first variable in $s = u$ is mapped to a_1, the second to a_2, and so on (with $a_1 \prec \ldots \prec a_n$); and another where the variables are mapped in reverse order. The critical pair is redundant if either choice of σ works. This is not a principled choice—most likely, some critical pairs need a different σ—but we do not know how to find the "best" σ.

4 Implementation

Twee consists of 5300 lines of Haskell code, comprising: terms, unification etc. (1150 lines); the frontend (850 lines); proof output (700 lines); general data structures (700 lines); the main proof loop (600 lines); joining, ground joining and connectedness (500 lines); critical pairs and the passive set (400 lines); term indexing (250 lines); and KBO (150 lines). This does not include TPTP parsing, clausification, etc., which are provided by the 4000-line Jukebox [16] program.

Most of Twee is written in a high-level, Haskell-idiomatic, somewhat inefficient style. Performance-critical parts (term manipulation, term indexing, and the passive set) are coded more carefully, and are described below. The bottleneck is usually normalising the many millions of critical pairs that are generated.

4.1 Terms

The simplest way to represent terms in Haskell, as trees, is not ideal: it creates pressure on the garbage collector, and core operations such as matching and unification become heavily recursive and needlessly slow.

Instead, we represent terms as *flatterms*—the term is flattened into a list of symbols and stored in an array. In order to preserve the structure of the term, each symbol is paired with a number giving the size of the subterm rooted at that symbol. For example, the term $f(x, g(x, y))$ is represented as:

$f:5$	$x:1$	$g:3$	$x:1$	$y:1$

where e.g. $g : 3$ indicates a subterm with root g that is 3 symbols long (g, x, y).

In addition, each function and variable has an *ID number*, and the term stores those ID numbers, rather than a pointer to the function or variable. So, in the array above, the "f" really means the ID number of f. Functions have positive ID numbers, and variables negative, so they can be easily told apart, and there is a separate global array which maps ID numbers to functions. This design allows us to represent a term as a simple array of integers, so that pressure on the garbage collector is reduced. Also, comparing two terms for equality just amounts to a bytewise comparison of the arrays (a C `memcmp`). What's more, by using array slicing, we can view a term's subterms as flatterms in their own right.

On top of this we build a higher-level API. There are two types, terms and termlists, both implemented as flatterms. With the help of Haskell's user-defined patterns, they are exposed to the user as ordinary algebraic datatypes. We can use normal pattern matching to e.g. check if a term is a function or variable, access its children (as a termlist), iterate through it a symbol or subterm at a

time, etc. All these operations turn into a few machine instructions. Matching and unification are implemented using this API as efficient tail-recursive loops.

4.2 Indexing

Rewriting uses a perfect discrimination tree [15], including Waldmeister's refinements [12]. The implementation takes care not to create backtracking points unless needed. There is no unification index, since this is not usually a bottleneck.

4.3 The Passive Set

Early versions of Twee often ran out of memory after about 30 minutes. The reason is the passive set—it grows quadratically in the number of active rules, because any pair of rules can have a critical pair. In typical prover runs it contains anywhere between a million and a hundred million critical pairs.

Twee now uses a space-efficient passive set representation adapted from Waldmeister [12]. The main idea is to throw away all terms involved in the critical pair, and only remember: (1) the ID numbers of the two rules involved, (2) the position of the overlap, and (3) the score of the critical pair. When a critical pair is selected, the ID numbers and position are used to reconstruct the critical pair. This design uses about 12 bytes of memory per critical pair, so Twee can run for many hours without running out of memory.

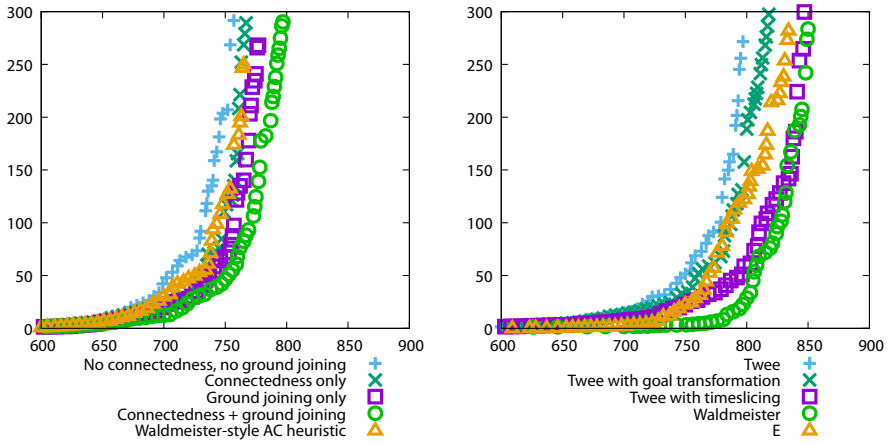
5 Evaluation

In this section we report on two evaluations: one investigating the effect of the different redundancy criteria of Section 3, and one comparing the performance of Twee against E 2.5 and Waldmeister. In both cases we ran Twee on all 981 unsatisfiable UEQ problems from TPTP 7.4.0, with a time limit of 5 minutes.

Redundancy criteria. Figure 1a shows how the performance of Twee varies depending on which redundancy criteria are enabled. The x-axis shows the number of problems solved (starting from problem 600) and the y-axis shows the runtime for that problem. The combination of ground joinability testing and connectedness is much stronger than either on their own—it seems that each catches cases that the other misses. It is clearly best to have both switched on.

The figure also includes a variant of Twee which implements the heuristic for AC functions described in [1] (and no other redundancy criterion), which solves fewer problems than our approach. This is perhaps not surprising, as our approach handles a wider class of functions.

Twee, E, Waldmeister. Figure 1b compares Twee's performance against E and Waldmeister. Twee is run in three variations: with and without the goal-directed transformation from Section 2, and as a timesliced version which runs the other two versions for 150s each. By far the best choice for Twee is to timeslice, when it comes close to Waldmeister's performance. This suggests that Twee with and without the goal transformation solve somewhat different sets of problems.



(a) Different redundancy criteria. (b) Compared against Waldmeister and E.

Fig. 1: Benchmarks.

6 Future Work

Knuth-Bendix completion pays little attention to the goal: it simply completes the rewrite system until the goal becomes trivial. We plan to search for ways to make Twee more goal-directed, for example by rewriting the goal backwards somewhat in the style of [18]. The success of the goal transformation shows that goal direction ought to be important.

Twee uses a fixed term ordering, which is clearly a weakness on certain problem kinds such as `RNG`. We do not want to choose a term order based on syntactic analysis of the problem, but would like to choose it dynamically based on the state of the proof, perhaps by incorporating ideas from MædMax [19].

7 Conclusion

Twee is a unit equality prover implemented in 5300 lines of Haskell code. Its performance is good, thanks to a careful implementation, strong redundancy criteria and a transformation to help goal-directness. It performs particularly strongly on problems involving permutative laws, such as those in `LAT` and `REL`. Its main weaknesses are that it always uses a fixed term order, and has only weak goal direction. We hope that a future version of Twee, with real goal direction and a smart choice of term order, will be even stronger.

Acknowledgements. This work was supported by the Swedish Research Council (VR) grant 2016-06204, *Systematic Testing of Cyber-Physical Systems (SyTeC)*.

We thank the reviewers for their many helpful comments.

References

1. Avenhaus, J., Hillenbrand, T., Löchner, B.: On using ground joinable equations in equational theorem proving. Journal of Symbolic Computation 36(1), 217–233 (2003), https://doi.org/10.1016/S0747-7171(03)00024-5
2. Bachmair, L., Dershowitz, N.: Critical pair criteria for completion. Journal of Symbolic Computation 6(1), 1–18 (1988), https://doi.org/10.1016/S0747-7171(88)80018-X
3. Bachmair, L., Dershowitz, N., Plaisted, D.A.: Completion without failure. In: Aït-Kaci, H., Nivat, M. (eds.) Rewriting Techniques, pp. 1–30. Academic Press (1989), https://doi.org/10.1016/B978-0-12-046371-8.50007-9
4. Claessen, K., Lillieström, A., Smallbone, N.: Sort it out with monotonicity. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction – CADE-23. Lecture Notes in Computer Science, vol. 6803, pp. 207–221. Springer (2011), https://doi.org/10.1007/978-3-642-22438-6_17
5. Claessen, K., Smallbone, N.: Efficient encodings of first-order Horn formulas in equational logic. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10900, pp. 388–404. Springer (2018), https://doi.org/10.1007/978-3-319-94205-6_26
6. Comon, H., Narendran, P., Nieuwenhuis, R., Rusinowitch, M.: Deciding the confluence of ordered term rewrite systems. ACM Transactions on Computational Logic 4(1), 33–55 (Jan 2003), https://doi.org/10.1145/601775.601777
7. Denzinger, J., Kronenburg, M., Schulz, S.: DISCOUNT - a distributed and learning equational prover. Journal of Automated Reasoning 18(2), 189–198 (Apr 1997), https://doi.org/10.1023/A:1005879229581
8. Denzinger, J., Schulz, S.: Recording and analysing knowledge-based distributed deduction processes. Journal of Symbolic Computation 21(4), 523–541 (1996), https://doi.org/10.1006/jsco.1996.0029
9. Gordon, M.J., Milner, R., Wadsworth, C.P.: Edinburgh LCF. A mechanised logic of computation. Springer, Berlin, Heidelberg (1979), https://doi.org/10.1007/3-540-09724-4
10. Korovin, K., Voronkov, A.: A decision procedure for the existential theory of term algebras with the Knuth-Bendix ordering. In: Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science. pp. 291–302. LICS '00, IEEE Computer Society, Los Alamitos, CA, USA (2000), https://doi.org/10.1109/LICS.2000.855777
11. Löchner, B.: Things to know when implementing KBO. Journal of Automated Reasoning 36(4), 289–310 (Apr 2006), https://doi.org/10.1007/s10817-006-9031-4
12. Löchner, B., Hillenbrand, T.: A phytography of WALDMEISTER. AI Communications 15(2,3), 127–133 (Aug 2002)
13. Martin, U., Nipkow, T.: Ordered rewriting and confluence. In: Stickel, M.E. (ed.) 10th International Conference on Automated Deduction. pp. 366–380. Springer Berlin Heidelberg, Berlin, Heidelberg (1990), https://doi.org/10.1007/3-540-52885-7_100
14. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) Automated Deduction – CADE 27. pp. 495–507. Springer International Publishing, Cham (2019), https://doi.org/10.1007/978-3-030-29436-6_29

15. Sekar, R., Ramakrishnan, I., Voronkov, A.: Chapter 26 - Term indexing. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1853–1964. Handbook of Automated Reasoning, North-Holland, Amsterdam (2001), https://doi.org/10.1016/B978-044450813-3/50028-X
16. Smallbone, N.: Jukebox. https://github.com/nick8325/jukebox/ (2018)
17. Smallbone, N.: Twee, an equational theorem prover. https://nick8325.github.io/twee/ (2021)
18. Socher-Ambrosius, R.: A goal oriented strategy based on completion. In: Kirchner, H., Levi, G. (eds.) Algebraic and Logic Programming. pp. 435–445. Springer Berlin Heidelberg, Berlin, Heidelberg (1992), https://doi.org/10.1007/BFb0013842
19. Winkler, S., Moser, G.: MædMax: A maximal ordered completion tool. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) Automated Reasoning. pp. 472–480. Springer International Publishing, Cham (2018), https://doi.org/10.1007/978-3-319-94205-6_31