Available online at www.sciencedirect.com
ScienceDirect
Journal of Applied Logic 7 (2009) 41–57
JOURNAL OF APPLIED LOGIC
www.elsevier.com/locate/jal

Lightweight relevance filtering for machine-generated resolution problems

Jia Meng [a], Lawrence C. Paulson [b,*]

[a] *National ICT, Australia*
[b] *Computer Laboratory, University of Cambridge, UK*

Available online 2 August 2007

Abstract

Irrelevant clauses in resolution problems increase the search space, making proofs hard to find in a reasonable amount of processor time. Simple relevance filtering methods, based on counting symbols in clauses, improve the success rate for a variety of automatic theorem provers and with various initial settings. We have designed these techniques as part of a project to link automatic theorem provers to the interactive theorem prover Isabelle. We have tested them for problems involving thousands of clauses, which yield poor results without filtering. Our methods should be applicable to other tasks where the resolution problems are produced mechanically and where completeness is less important than achieving a high success rate with limited processor time.

Keywords: Relevance filtering; Proving in large theories; Automated reasoning; Interactive theorem proving

1. Introduction

We have been working for three years on a project to link the interactive prover Isabelle with resolution-based automatic theorem provers (ATPs) [10]. Ease of invocation is our primary aim: users should be able to generate self-contained resolution problems with a single mouse click. Each problem should include a substantial library of previously-proved theorems. This spares users the tedium of listing which routine facts to include with their problem, but it can overload the ATPs with hundreds or thousands of irrelevant facts. Our initial results, for all of the leading ATPs, were poor. This paper describes how we managed to improve the success rate significantly, while increasing the "substantial library" to include *all* known theorems. We describe simple, efficient algorithms for removing irrelevant clauses from huge resolution problems. We also describe related experiments and techniques, many of them speculative. Our experiences may be valuable to other users of resolution provers.

The relevance problem dates back to the earliest days of resolution. As first defined by Robinson [19], a literal is *pure* if it is not unifiable with a complementary literal in any other clause. Deleting clauses that contain pure literals preserves the satisfiability or unsatisfiability of the remaining clauses. This process is a form of relevance test. It can be effective, but it is not a full solution. In general, demonstrating that a fact is irrelevant seems to require finding a proof

* Corresponding author.
E-mail addresses: Jia.Meng@nicta.com.au (J. Meng), LP15@cam.ac.uk (L.C. Paulson).

1570-8683/$ – see front matter
doi:10.1016/j.jal.2007.07.004

without using it. Syntactic criteria for irrelevance may be helpful, especially if we can accept a few false positives. We are happy to sacrifice solutions to some problems in return for a high overall success rate.

If a resolution theorem prover is invoked by another reasoning tool, then the problems it receives will have been produced mechanically. Machine-generated problems may contain thousands of clauses, each containing large terms. Many ATPs are not designed to cope with such problems. Traditionally, the ATP user prepares a mathematical problem with the greatest of care, and is willing to spend weeks running proof attempts, adjusting weights and refining settings until the problem is solved. Machine-generated problems are not merely huge but may be presented to the automatic prover by the dozen, with its heuristics set to their defaults and with a small time limit.

At the start of our investigations, our Isabelle-ATP linkup offered the option of generating small or large problems. All problems included a selection of previously-proved theorems, some chosen by the user, translated into clause form. A small problem, which typically comprised over 1300 axiom clauses, included the theorems that were provided by default to Isabelle's classical reasoner [11,13]. A large problem, which typically comprised 2500 clauses, also included the theorems that were provided by default to Isabelle's rewriter. Even our small problems looked rather large to a resolution prover. We ran extensive tests with a set of 285 such problems (153 small, 132 large). We were initially disappointed by the success rates, which seldom exceeded 60 percent for runtimes of 50 seconds.[1] The large numbers of clauses in our problems made it clear that we should investigate relevance filtering. Given our resource constraints, any filter would have to be fast.

The filter we eventually developed is successful enough to cope with the full set theorems known to Isabelle, reducing problems from several thousand clauses to a few hundred clauses. Users no longer have to choose which theorems are included with their problems. This push-button activation greatly improves the interface's usability.

In the course of our investigations, we found that many obvious ideas were incorrect. For example, we thought that since ATPs generate hundreds of thousands of clauses during their operation, an extra fifty clauses at the start should not do any harm; however, they do.

Paper outline. We begin with background material on Isabelle, ATPs, and our linkup between the two, mentioning related work on other such linkups (Section 2). We describe our initial attempts to improve the success rate of our linkup (Section 3). Next, we describe the evolution of our relevance filter (Section 4) and the further refinements needed to handle huge problems (Section 5). Both of these sections present empirical results as series of graphs. Finally, we present brief conclusions (Section 6).

2. Background

Resolution theorem provers work by deriving a contradiction from a supplied set of *clauses* [3]. Each clause is a disjunction of *literals* (atomic formulae and their negations) and the set of clauses is interpreted as a conjunction. Clause form can be difficult to read, and the proofs that are found tend to be unintuitive, but there is no denying that these provers are powerful. In the sequel we refer to them as automatic theorem provers or ATPs. (This term includes clausal tableau provers, but not SAT solvers, decision procedures etc.) Our experiments mainly use E [20], SPASS [25] and Vampire [17].

Interactive theorem provers allow proofs to be constructed by natural chains of reasoning, generally in a rich formalism such as higher-order logic, but their automation tends to be limited to rewriting and arithmetic. Quantifier reasoning tends to be weak: many interactive systems cannot even prove a simple theorem like $\exists x\, \forall y\, P(x, y) \rightarrow \forall y\, \exists x\, P(x, y)$ automatically. Developers of interactive tools would naturally like to give their users access to the power of ATPs without requiring them to become experts on ATPs.

We have implemented a linkup between ATPs and the interactive prover Isabelle. Of the many differences between our project and other work [5,21], a crucial one is that Isabelle already provides excellent automation. By typing *auto*, the Isabelle user causes approximately 2000 previously-proved theorems to be used as rewriting rules and for forward and backward chaining. A related tool, *blast* [13], performs deep searches in a fashion inspired by tableau provers. Even ten years ago, using early predecessors of these tools, Isabelle users could automatically prove theorems like

[1] All timings we report refer to a processor speed of approximately 2.4 GHz.

this set equality [11]:

$$\bigcup_{i\in I}(A_i \cup B_i) = \left(\bigcup_{i\in I} A_i\right) \cup \left(\bigcup_{i\in I} B_i\right).$$

Set theory problems, and the combinatory logic examples presented in that paper, remain difficult for automatic theorem provers. When we first got our linkup working, we were disappointed to find that ATPs were seldom better than *auto*—not only for hard set theory problems, but for most problems, even easy ones. We have devoted much effort to improving our success rate. Bugs in our linkup were partly to blame for the poor results. Much of our effort went to improving the problem presentation. We found a compact way of representing the type of a polymorphic constant: rather than including its full type as an additional argument, we include only the types needed to form particular instances [9, §2.4]. For example, if the constant's polymorphic type is $set(set(\alpha)) \to set(\alpha)$, then we store only the type denoted by α. We devoted some time to identifying prover settings to help ATPs cope with huge problems. Above all, we have struggled to find ways to filter out irrelevant axioms.

Sutcliffe and Dvorsky [22] propose a remarkably simple idea. Given a problem involving n axiom clauses, their RedAx ("reduce axioms") tool systematically generates its 2^n subsets. Heuristic criteria are used to remove "heavy" axioms first, but all combinations are attempted until one succeeds. Redax gives each problem variant to an ATP with a small time limit such as 60 seconds. The approach makes sense because most ATPs quickly reach a point of diminishing returns, proving few additional theorems as their time limit rises from say 60 to 6000 seconds, while the higher time limit allows proof attempts on 100 variants of the problem. A combination of RedAx and Vampire found the first-ever automatic proofs of four theorems in the TPTP library [24]. We were unaware of this work when undertaking ours, and could not have applied such an approach to problems containing thousands of clauses. However, RedAx demonstrates the power of simple means.

Of previous work, the most pertinent is the integration between the Karlsruhe Interactive Verifier (KIV) and the tableau prover 3TAP, by Ahrendt and others [1]. Reif and Schellhorn [16] present a component of that integration: an algorithm for removing irrelevant axioms. It relies on analysing the structure of the theory in which the conjecture is posed. Specifically, their method is based on four criteria for reduction, which they call the minimality, structure, specification and recursion criteria. This method is not suitable for Isabelle, where users build on a theory that already holds approximately 7000 theorems. Although in principle a user could base a project on lower levels of the theory hierarchy, to do so would require knowledge of the implementation and willingness to sacrifice functionality. Isabelle/HOL derives essential tools such as recursive function and type definitions from basic constructions such as well-founded relations and fixedpoint operators. Such tools only become available after their supporting theories have been loaded. Even the theory of lists lies near the top of this hierarchy, since it involves a recursive type definition. We decided to try other methods, which are described below.

The Isabelle-ATP linkup generates problems that contain conjecture clauses along with previously-proved theorems in clause form. At the outset of our project, giving all known theorems to the linkup was inconceivable. We expected users to select one or both of the following:

- *classical* clauses, which arise from the theorems Isabelle uses for forward and backward chaining
- *simplification* clauses, which arise from equational theorems Isabelle uses as rewrite rules, such as $0 + x = x$.

In most cases, these would have to be augmented with known theorems specific to the user's problem. In addition, each problem contained *arity* and *class inclusion* clauses, to express aspects of Isabelle's type system: the type class hierarchy.

Isabelle's *axiomatic type classes* are sets of types that meet a given specification. For instance, the type `nat` of natural numbers is a member of `order`, the class of partial orderings; we express its membership as the unit clause $order(nat)$. An arity relates the type classes of the arguments and results of type constructors. For example, an arity clause

$$\forall\tau\big[type(\tau) \to order\big(list(\tau)\big)\big]$$

says if the argument of `list` is a member of class `type`, then the resulting type of lists belong to class `order`. For more information, we refer readers to our previous papers [8,10].

Although the arity and class inclusion clauses typically number over one thousand, they pose no difficulties for modern ATPs. They are Horn clauses that contain only monadic (unary) predicates, which are not related by any equations. Most arity clauses have one literal, while class inclusion clauses consist of two literals. Pure literal elimination suffices to remove redundant arity and class inclusion clauses, so ATPs can delete most of them immediately.

Since running the experiments reported below, we have modified the linkup to ensure that these clauses are only included if they refer to types and type classes actually mentioned in the problem. While the average number of type-related clauses has dropped from 1140 to 11.7 per problem, the impact on prover performance is minimal. This confirms our view that this Horn theory is trivial.

3. Initial experiments

We have evaluated and improved the Isabelle-ATP linkup through exhaustive tests on hundreds problems in clause form. We obtained these by modifying our linkup to save the problems it was producing. Each represents a call to our system at some point in an Isabelle proof. The original Isabelle proofs for some of these problems require multiple steps, explicit quantifier instantiations, or other detailed guidance. The set used for the experiments described in this paper now numbers 285 problems; a second set of 151 problems is chiefly intended for evaluating different ways of translating higher-order logic into first-order logic [9], but we have also used this set to investigate relevance filtering.

In choosing our problems, we were mainly interested in their difficulty as indicated by the length of the existing manual proof. We aimed to have a range of difficulties, from easy to impossible. The following table breaks down the 285 problems according to their domain. The O-notation problems involve arithmetic.

domain	*number*
security protocols [12,14]	124
set theory and Zorn's lemma	48
O-notation [2]	39
combinatory logic [11]	24
Tarski's fixedpoint theorem	23
propositional logic	24
UNITY formalism	3

In this paper, the term "problem size" means "number of axiom clauses in the problem". We later refine this concept to exclude certain numerous but harmless axioms generated by the translations we use. Many of our problems have trivial proofs. For 52 of them, the refutation involves just two or three clauses and the only difficulty lies in the problem size. Other problems take minutes to prove, and a few have never been proved.

Our first task was to verify that the problems were solvable. If a problem could not be proved by any ATP, we sometimes removed irrelevant clauses manually, using our knowledge of the problem domain, in the hope of finding a proof. We could thus identify and correct problems that were missing essential axiom clauses. Bugs in our code also harmed the success rate. These ranged from the trivial (failing to notice that the original problem already contained the empty clause) to the subtle (Skolemization failing to take account of polymorphism). Over time, with the help of the techniques described below, we were able to obtain proofs for all but three of our problems.

We have partially automated the laborious process of reducing problem size. A simple idea is to note which axioms take part in any successful proofs—call them *referenced* axioms—and to remove all other axioms from the unsolved problems. We have automated this idea for the provers Vampire and E. Both clearly identify references to axiom clauses, which they designate by positive integers. Simple Perl scripts read the entire clause set into an array; referenced axioms are found by subscripting and written to a new file. We thus obtain a reduced version of the problem, containing only the clauses actually used in its proof. Repeating this process over a directory of problems yields a new directory containing reduced versions of each solved problem. If both Vampire and E prove a theorem, then the smaller file is chosen. We then concatenate the solutions, removing conjecture clauses. The result is a file containing all referenced axioms. Another Perl script intersects this file with each member of the problem set, yielding a reduced problem set where each problem contains only referenced axioms.

Auto-reduction by using only referenced axioms has an obvious drawback: some unsolved problems are likely to need axioms that have not been referenced before. Even so, this idea improved our success rate from about 60 percent

to 80 percent. It is not clear how to incorporate this idea into an interactive prover, since then its success on certain problems would depend upon the previous history of proof attempts, making the system's behaviour hard to predict. Auto-reduction's immediate benefit is that it yields evidence that the original problems have proofs: a reduced problem is a subset of the original problem, and often it is easy to prove. Fewer suspect problems require hand examination.

Using only referenced axioms does not guarantee that problems will be small. As of this writing, there are 405 referenced clauses. Approximately 150 of these correspond to theorems proved as part of the basic Isabelle/HOL development, and are common to all problems. The remainder, the majority of the referenced clauses, are specific to various problem domains. A proof about security protocols could have another 90 clauses, for a total of about 240. Still, the auto-reduced problems should be easier to solve than the original problems:

(1) The reduced problems are much smaller than the raw ones, and
(2) referenced clauses may be somehow better than other clauses for finding refutations.

This second point introduces an interesting side issue: if some clauses are better than others, then is it useful to focus on the very worst clauses? Can there exist *pathological* clauses, whose presence in a problem harms its success rate? This question is difficult to test. There is no reason to believe that the same clauses will turn out to be pathological for all ATPs. Identifying pathological clauses seems to require much guessing and manual inspection. We assumed that a pathological clause would contain highly general literals such as $X = Y$, $X < Y$, $X \in Y$, or their negations.

We investigated the question of pathological clauses by carefully examining the standard Isabelle/HOL library, eventually blacklisting around 140 theorems. A theorem could be blacklisted for various reasons, such as having too big a clause form, being logically equivalent to other theorems, or dealing with too obscure a property. This effort yielded only a small improvement to the success rate, probably because Isabelle's sets of classical and simplification rules have been hand-selected, and exclude obviously prolific facts such as transitivity. The main benefit of this exercise was our discovery that the generated problems included large numbers of functional reflexivity axioms: that is, axioms such as $X = Y \longrightarrow f(X) = f(Y)$. They are redundant in the presence of paramodulation; since we only use ATPs that use that inference rule for equality reasoning, we now omit such clauses in order to save ATPs the effort of discarding them. (This change did not improve the success rate, but at least it made the problems smaller.) This aspect of our project was in part a response to SPASS developer Thomas Hillenbrand's insistence—in an e-mail dated 23 July 2005—on "engineering your clause base once and forever".

4. Developing signature-based relevance filters

Automatic relevance filtering is clearly more attractive than any method requiring manual inspection of clauses. We decided to determine relevance with respect to the provided conjecture clauses. The simplest way of doing this is to enable the Set of Support option, if it is available. Wos's SOS heuristic [26], which dates from 1965, ensures that all inference rule applications involve at least one clause derived from the negated conjecture. It prevents inferences among the axioms and helps make the search goal-directed. It is incomplete in the presence of the ordering heuristics used by modern ATPs, but SPASS still offers SOS and it greatly improves the success rate, as the graphs presented below will demonstrate. Stephan Schulz kindly gave us two ways of simulating SOS in the E prover, but neither of them yielded improvements for our problems.

The techniques we describe below are the outcome of extensive tinkering with obvious ideas, largely of our own invention. We already knew that simple methods could be effective: in an early experiment (which we have never published), we modified the linkup to block all axiom clauses except those from a few key theories. That improved the success rate enough to yield proofs for eight hitherto unsolved problems. Clearly if such a crude filter could be beneficial, then something based on the conjecture clauses could be better still. Having automatically-reduced versions of most of our problems allows us to test the relevance filter without actually running proofs: yet more Perl scripts compare the new problems with the reduced ones, reporting any missing axioms.

The abstraction-based relevancy testing approach of Fuchs and Fuchs [7] is specifically designed for model elimination (or connection tableau) provers. It is not clear how to modify this approach for use with saturation provers, which are the type we use almost exclusively. Their approach has some curious features. Though it is based upon very general notions, the specific abstraction they implement is a *symbol abstraction*, which involves "identifying

some predicate or function symbols" and forming equivalence classes of clauses. We confess that we were not able to derive any ideas from this highly mathematical paper.

4.1. Plaisted and Yahya's strategy

Plaisted and Yahya's relevance restriction strategy [15] introduces the concept of *relevance distance* between two clauses, reflecting how closely two clauses are related. Simply put, the idea is to start with the conjecture clauses and to identify a set R_1 of clauses that contain complementary literals to at least one of the conjecture clauses. Each clause in R_1 has distance 1 to the conjecture clauses. The next round of iteration produces another set R_2 of clauses, where each of its clauses resolves with one or more clauses in R_1; thus clauses in R_2 have distance 2 to the conjecture clauses. The iteration repeats until all clauses that have distances less than or equal to some upper limit are included. This is an all-or-nothing approach: a clause is either included if it can resolve with some already-included clause, or, not included at all.

We found this method easy to implement, using Prolog, but unfortunately too many clauses are included after two or three iterations. This method does not take into account the ordering restrictions that ATPs would respect, including clauses on the basis of literals that would not be selected for resolution. Also, this strategy does not handle equality.

Plaisted and Yahya's strategy suggests a simple approach based on signatures. Starting with the conjecture clauses, repeatedly add all other clauses that share any symbols with them. We use *symbol* to mean "constant, function or predicate symbol". Symbols correspond to Isabelle constants, which can represent functions and predicates. This method handles equality, but it again includes too many clauses. Therefore, we have refined Plaisted and Yahya's strategy and designed several new algorithms that work well (Section 4.6).

4.2. A passmark-based algorithm

Our filtering strategies abandon the all-or-nothing approach. Instead, we use a measure of relevance: a clause is added to the pool of relevant clauses provided it is "sufficiently close" to an existing relevant clause. If a clause mentions n symbols, of which m are relevant, then the clause receives a score (relevance mark) of m/n. The clause is rejected unless its score exceeds a given *pass mark*, a real number between 0 and 1. If a clause is accepted, all of its symbols become relevant. This process is iterated until no new clauses are accepted. To prevent too many clauses from being accepted, somehow the test must become stricter with each iteration.

In the first filtering strategy, we attach to each clause a relevance mark that may be increased during the filtering process. The pseudo-code for our algorithm is shown in Fig. 1. The pseudo-code is largely self-explanatory. We only give a few more comments below.

- When the function `relevant_clauses` is first called, the set `W` of working relevant clauses contains the goal clauses, while `T` contains all the axiom clauses.
- In function `update_clause_mark`, `|R|` is the number of elements in the set `R`.
- The multiplication by `P_M` in function `update_clause_mark` makes the relevance test increasingly strict as the distance from the conjecture clauses increases, which keeps the process focused on the conjecture and prevents too many clauses from being taken as relevant.

Isabelle allows overloading of constants. For example, <= can denote the ordering $\leqslant$ on the integers as well as the subset relation. Therefore, the Isabelle implementation of this algorithm regards two symbols as matching only if their types match as well.

4.3. Using the set of relevant symbols

We have refined the strategy above, removing the requirement that a clause be close to one single relevant clause. It instead accumulates a pool of relevant symbols, which is used when calculating scores. This strategy is slightly simpler to implement, because scores no longer have to be stored, and it potentially handles situations where a clause is related to another via multiple equalities. To make the relevance test stricter on successive iterations, we increase the pass mark after each successive iteration by the formula $p' = p + (1 - p)/c$, where c is an arbitrary convergence

```
function relevant_clauses (W, T, P)
      # W: working relevant clauses set (each clause carries a relevance mark)
      # T: working irrelevant clauses set
      # P: pass mark
var A # accumulates relevant clauses
begin
U := {};
while W ≠ {} do {
      for each clause-mark pair (C,M) in T do
      { update_clause_mark (W, (C,M)) }
      #partition (C,M) pairs in T into two sets
      Rel := {(C,M) | P ≤ M};
      Irrel := T - Rel;
      A := W ∪ A;
      W := Rel;
      T := Irrel;
}
return A; #final set of relevant clauses
end

function update_clause_mark (W, (C,M))
      # W: relevant clauses set
      #(C,M): a clause-mark pair
effect: #updates the relevance mark of C
begin
for each clause-mark (C',M') in W do {
    CS := symbols_of C;
    R := CS ∩ symbols_of C';
    IR := CS − R;
    M := max(M, M' * |R| / (|R| + |IR|));
}
end
```

Fig. 1. A passmark-based filtering strategy,

parameter. The point of this formula is to make the pass mark converge to 1 geometrically. If $c = 2$, for example, then each iteration halves the difference $1 - p$. The algorithm appears in Fig. 2.

Since the value of c is used to modify that of p, the optimal values of these parameters need to be found simultaneously. We ran extensive empirical tests. It became clear that large values of c performed poorly, so we concentrated on 1.6, 2.4 and 3.2 with a range of pass marks. We obtained the best results with $p = 0.6$ and $c = 2.4$. These values give a strict test that rapidly gets stricter, indicating that our problems require a drastic reduction in size.

To illustrate these points, Fig. 3 presents two graphs. They plot success rates and problem sizes as the pass mark increases from 0.0 (all clauses accepted) to 0.9 (few clauses accepted).

- Success rates are for Vampire in its default mode, allowing 40 seconds per problem.
- Problem sizes refer to the average number of clauses per problem, ignoring conjecture clauses and the clauses that formalise Isabelle's type system.
- Like all the graphs in this paper, they concern the set of 285 problems introduced in Section 4 above.

Vampire's success rate peaks sharply at 0.6, by which time the average problem size has decreased from 909 to 142 clauses. If the filter has removed an essential axiom, then the problem cannot be proved, which is why the success rate drops when $p > 0.6$. We repeat that these graphs are for illustration only; our parameter settings are based on extensive testing involving several ATPs.

4.4. Taking rarity into account

Another refinement takes into account the relative frequencies of the symbols in the clause set. Some symbols are common while others are rare. An occurrence of a rare symbol would seem to be a strong indicator of relevance,

```
function relevant_clauses (RS, T, P)
        # RS: set of relevant symbols
        # T: working irrelevant clauses set
        # P: pass mark
var A # accumulates relevant clauses
begin
repeat {
     for each clause C_i in T do
        { M_i := clause_mark (RS, C_i) }
     Rel := set of all clauses C_i such that P ⩽ M_i
     T := T - Rel;
     A := A ∪ Rel;
     P := P + (1 - P) / c;
     RS := (symbols_of Rel) ∪ RS;
} until Rel = {}
return A; #final set of relevant clauses
end

function clause_mark (RS, C)
          # RS: a set of relevant symbols
          # C: a clause
begin
CS := symbols_of C;
R := CS ∩ RS;
IR := CS − R;
return |R| / (|R| + |IR|); #the relevance mark of C
end
```

Fig. 2. An improved filtering strategy using a set of relevant symbols.

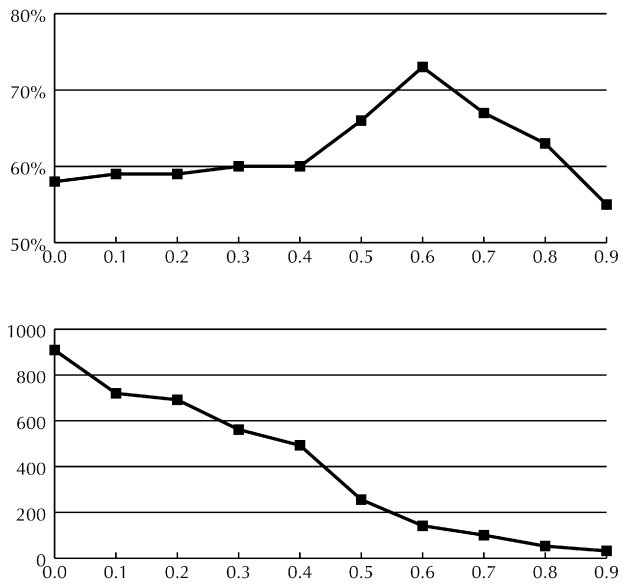

Fig. 3. Success rates and problem sizes against pass marks.

while its very rarity would ensure that not too many new clauses are included. In the relevance quotient m/n, we boost m to take rarity into account, while leaving n to denote the total number of symbols. Originally, we took rarity into account for both m and n, but consistently got poor results. In particular, clauses involving Skolem functions were being filtered out. A Skolem function is typically very rare, occurring only a few times in the set of clauses. By ignoring rarity in n, we ensure that an occurrence of a rare symbol in a clause makes it easier, and never harder, for the clause to be accepted.

```
function clause_mark (RS, C, ftab)
      # RS: a set of relevant symbols
      # C: a clause
      # ftab: a table of the number of occurrences of each symbol
      # in the clause set
begin
CS := symbols_of C;
R := CS ∩ RS;
IR := CS − R; #remaining symbols of C
M := 0;
for each symbol F in R do { M := M + func_weight(ftab, F) }
return (M / (M + |IR|)); #the relevance mark of C
end

function func_weight (ftab, F)
       # ftab: a table of the number of occurrences of each symbol
       # F: a symbol
begin
freq := number_of_occurrences (ftab, F);
return (frequency_function(freq));
end
```

Fig. 4. A filtering strategy for rarely-occurring symbols.

This strategy the same the one described in Section 4.3, except that it uses the function `clause_mark` shown in Fig. 4. The relevance mark of a clause `C` calculated by `clause_mark` is not the percentage of relevant symbols in `C` any more. Instead, we use the function `func_weight` to compute the sum of the relevant symbols' marks weighted by their frequencies.

A suitable `frequency_function` is needed to calculate a symbol's weight from its frequency. We use $f(n) = 1+2/\log(n+1)$. Thus, if a symbol occurs n times in the problem, then its contribution is $1+2/\log(n+1)$, rather than 1 as with the previous algorithm. We tested many other functions, including $1/(n+1)$, $1+1/\sqrt{n+1}$, $1+1/\log(n+1)$ and $1+1.4/\log(\log(n+2))$. Most performed worse than the constant function. Given that symbol frequencies can vary by two orders of magnitude, it was obvious at the outset that the correct formula would involve their logarithm. The log-log formula also performed well in our tests.

4.5. Other refinements

Hoping that unit clauses did not excessively increase resolution's search space, we experimented with adding all "sufficiently simple" unit clauses at the end of the procedure. A unit clause was simple unless it was a non-trivial equation, where an equation was trivial when its left- or right-hand side was variable-free. We discovered that over 100 unit clauses were often being added, and that they could indeed increase the search space. By improving the relevance filter in other respects, we found that we could do without a special treatment of unit clauses. A number of attempts to bias the relevance filter in favour of shorter clauses failed to improve the success rate.

Definition expansion is another refinement. If a symbol f is relevant, and a unit clause such as $f(X) = t$ is available, then it can be regarded as relevant. To avoid including "definitions" like $0 = N \times 0$, we check that the variables of the right-hand side are a subset of those of the left-hand side. Isabelle identifies definitions by distinctive names (ending with `_def`), but we do not use this fact. Note that Isabelle does not distinguish its internally-generated definitions, such as the fixedpoint construction of lists, from a user's definitions. Expanding the former will be harmful. Definition expansion can be beneficial provided not too many definitions are supplied. We now prefer a mode of operation in which all theorems are supplied to the filter, so we switch definition expansion off.

As of this writing, our system still contains a manually produced blacklist of 128 HOL theorems. We could probably shorten this blacklist, because a prime criterion for being blacklisted is that a theorem concerns an obscure primitive. However, some useless theorems easily survive relevance filtering. We also have a whitelist of theorems whose inclusion is forced; it contains the single theorem

$$\big[\forall x(x \in A \longrightarrow x \in B)\big] \longrightarrow A \subseteq B,$$

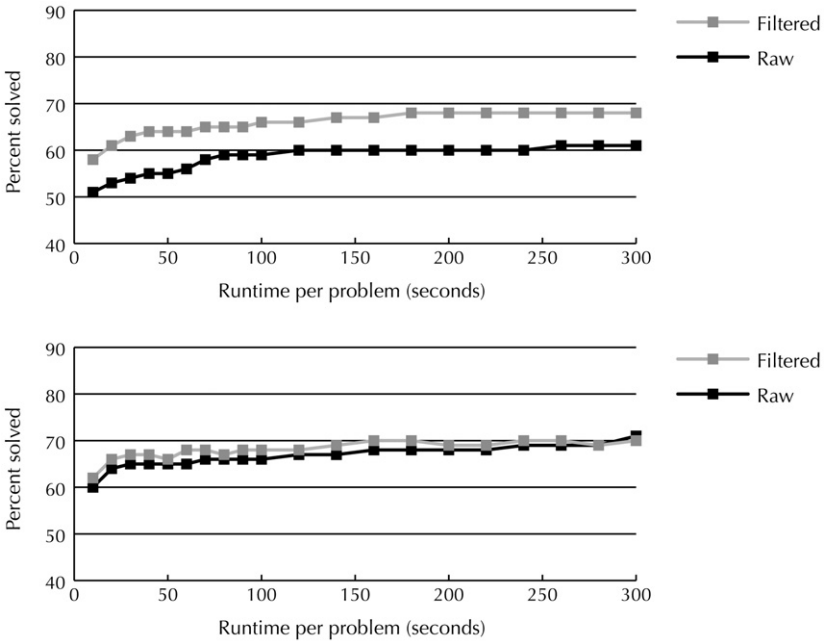

Fig. 5. E, Versions 0.9 and 0.91dev001.

which we found that the filter was frequently rejecting. We have decided to allow Isabelle users to manage these lists by occasionally attaching hints like "ATP exclude" or "ATP include" to the theorems they prove.

4.6. Empirical results

Extensive testing helped us determine which methods worked best and to find the best settings of the various parameters. We tried many ATPs during our investigations, obtaining the highest success rates with E, SPASS and Vampire. We tested these three provers in their default mode and with alternative settings. These include two different versions of E (Fig. 5): 0.9 and 0.91dev001. SPASS V2.2 performs better (Fig. 6) if SOS is enabled and splitting is disabled.[2] Vampire 8 does extremely well in its CASC mode (Fig. 7). We downloaded Vampire from the CASC-20 website [23]; it calls itself version 7.45. We ran these tests on a bank of Dual AMD Opteron processors running at 2.4 GHz. The Condor system [6] managed our batch jobs.

This section presents graphs comparing the success rates of filtered problems against raw ones. Success rates are plotted on a scale ranging from 40 to 90 percent, as the runtime per problem increases from 10 to 300 seconds. The graphs offer compelling evidence that relevance filtering is beneficial in our application. The success rate for the filtered problems exceeds that for the raw ones in virtually every case. This improvement is particularly striking given that relevance filtering can delete essential clauses. As mentioned in Section 4 above, we wrote Perl scripts to check problems for missing clauses, using our set of automatically-reduced problems as a basis. (We realise of course that a problem can have proofs using different subsets of the provided clauses.) The script indicates that eight percent of the filtered problems are missing essential clauses. Either gains are made elsewhere, or these eight percent are too difficult to prove anyway.

Our filtering gave the least benefit with the specially-modified version of the E prover (version 0.91dev001). Developer Stephan Schulz, in an e-mail dated 14 April 2006, had an explanation:

[2] The precise option string is `-Splits=0 -FullRed=0 -SOS=1`.

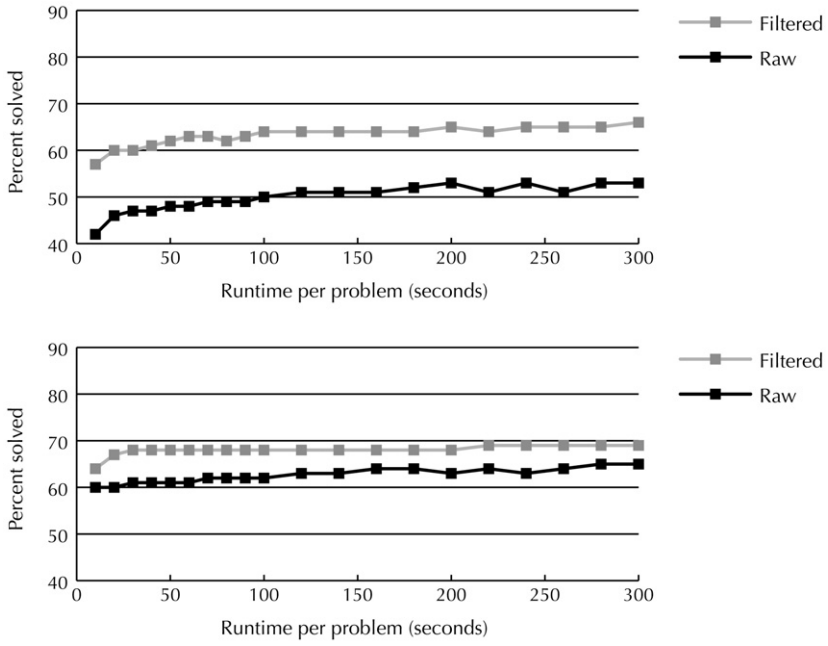

Fig. 6. SPASS 2.2, default settings and with SOS.

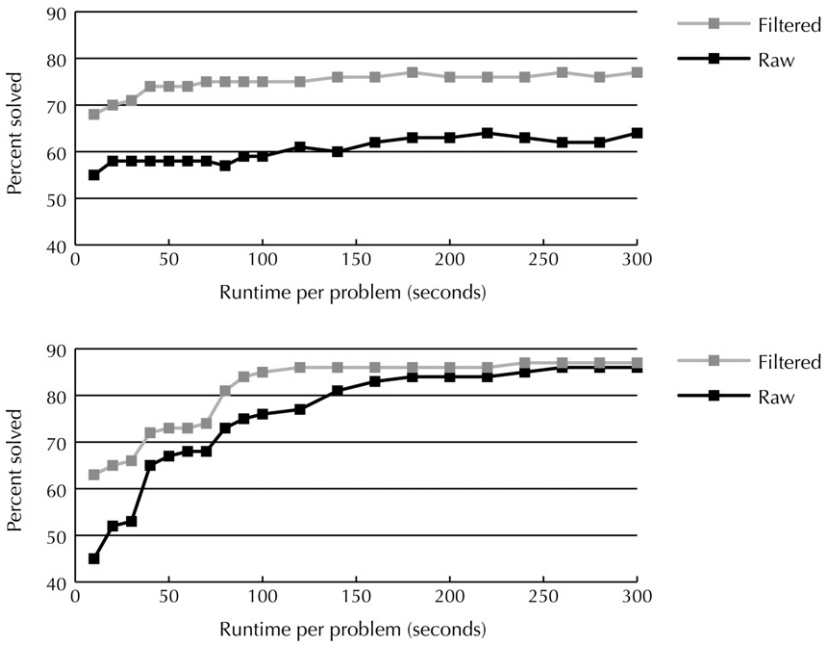

Fig. 7. Vampire 8.0, default settings and in CASC mode.

> E has a number of goal-directed search heuristics. The new version always selects a fairly extreme goal-directed one for your problems ... [which] will give a 10 times lower weight to symbols from a conjecture than to other symbols (all else being equal). I suspect that this more or less simulates your relevance filtering.

Schulz kindly produced this version after we reported that E version 0.91 did less well than 0.90 on our problem set. Such a setting could probably be added to other ATPs, and in a fashion that preserves completeness.

The success rate for raw problems should eventually exceed that of the filtered ones, since filtering sacrifices completeness. We can actually see this taking place for E version 0.91dev001 and Vampire in CASC mode. The other four graphs become nearly flat as processor time reaches 300 seconds. In some of these graphs, the benefits of filtering appear to be small. The situation changes as problems get bigger, as we shall see below.

5. A refined procedure for filtering huge problems

This success of the relevance filter has allowed us to introduce an entirely new interaction style. All theorems known to Isabelle, rather than just a selection, can be given to the linkup. For users, this is an immense improvement: no longer must they search for theorems to add to the defaults. Calling the system involves no more thought than pushing a button.

Including all theorems makes our task fundamentally more difficult. Before, though our problems were large, they consisted entirely of theorems that had been hand-designated as suitable for automation: good for forward or backward chaining or for rewriting. Now, we are including all theorems known to the system, the vast majority of which are unlikely to be useful. Many theorems in a proof development are technical lemmas intended to be used just once. In particular, the relevance filter's "expand definitions" mode (Section 4.5) performs badly if all theorems are included. This is not surprising. Before, definitions were only present if they had been specifically added. Now, all definitions in the system are included. They cause particular harm because expanding the definition of a symbol prevents the use of theorems about that symbol. Thus, we must switch this option off.

Increasing the size of the input by a factor of 20 naturally put our approach under stress. We were forced to pay more attention to efficient data structures. The relevance filter required further refinements before it could deliver acceptable results. In the end, however, the "all theorems" approach seems to deliver nearly as good a success rate as the previous approach. Given its greater convenience, this is the only approach we intend to support in the future.

The two refinements we introduced for "all theorems" mode are a relevance cap (to limit the number of clauses added per round) and dummy "theory symbols" (to allow the theory structure to influence the relevance score).

5.1. The relevance cap

Recall that the filter works iteratively. At each step, some clauses are marked as relevant provided they contain enough relevant symbols. We observed that in a few cases the resulting problems were still very large. The cause was clear: some combinations of symbols cause a large number of clauses to be marked as relevant, simply because the Isabelle library contains many theorems involving those symbols. Therefore all the symbols in those clauses are considered relevant, so in the next round even more clauses are marked as relevant, causing an avalanche. This is a defect of the "pool of symbols approach"; we might have chosen this time to revert to our previous relevance filter, where a clause would be marked as relevant only if it was sufficiently close to one other relevant clause. However, one can imagine an avalanche happening with this approach too. Our solution was simply to impose a cap—an upper bound—on the number of clauses that could be added at any iteration. Clauses in excess of the cap could still be considered in the next iteration.

As always, we ran extensive empirical tests to tune the parameters to optimal values. As another CASC had taken place, new versions of two of the provers were available. The new version of E, 0.99 "Singtom", gives almost identical results as its predecessor for our examples; we continue to use its mode specifically designed for our problems. The new version of Vampire, 8.1, offers CASC mode only. It improves slightly upon its predecessor; the improvement is marked as the runtime drops below 60 seconds, where it avoids the steep decline we see in Fig. 7.

The graphs show the effect of the upper bound on the success rate for each of E (Fig. 8), SPASS (Fig. 9) and Vampire (Fig. 10). Note that the optimum caps differ: 60 for Vampire, 100 for E and 40 for SPASS. The thick line represents the previous default, where the number of clauses added is unlimited. With E and Vampire, the improvement over the unlimited case is clear: the thick line is near the bottom. The results for SPASS are strange: most of the caps deliver worse results than the unlimited case, yet SPASS gives its best results with the smallest cap!

With the cap in place, we were able to relax the other parameters somewhat: the pass mark could decrease from 0.6 to 0.5, while the convergence could increase to 3.2. Making relevance filtering less strict reduces the danger of incompleteness, while our cap ensures that problems will be of a reasonable size. Fig. 11 shows that the average problem size increases slowly and linearly as the cap increases, while the maximum problem size rises dramatically.

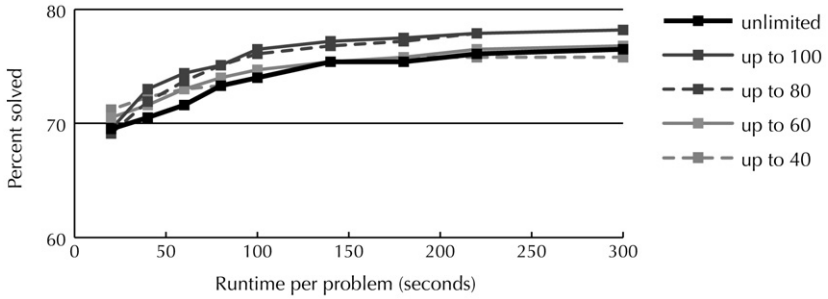

Fig. 8. E 0.99 with various caps.

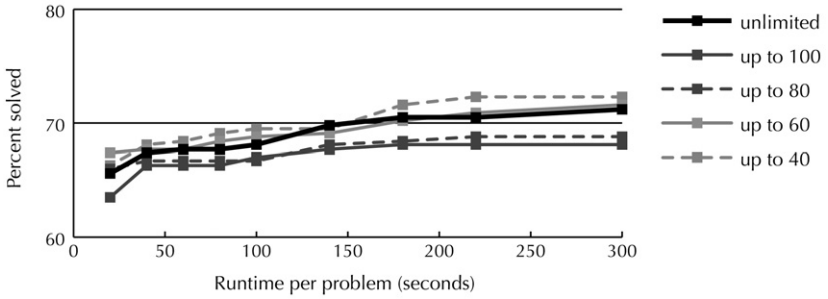

Fig. 9. SPASS 2.2 with various caps.

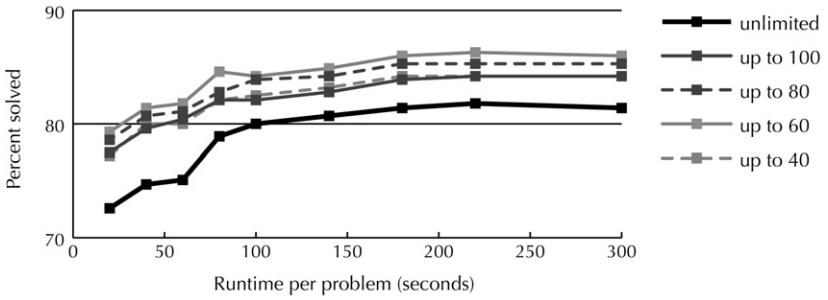

Fig. 10. Vampire 8.1 with various caps.

5.2. *Theory symbols: Recognising theory structure*

As mentioned earlier, we decided not to base our relevance filter on the structure of the theory hierarchy. Our theories are too large and their hierarchy includes constructions of basic principles such as recursion. However, it seems natural to assume that if one theorem is relevant to the problem, then other theorems proved in the same theory may also be relevant. Recall that theorems are chosen entirely on the basis of the symbols they contain. Adding a theory name as a dummy symbol gives a slight preference to other theorems that belong to the same theory as an already-relevant theorem. We call these dummy symbols *theory symbols*.

Theory symbols represent a tiny modification of the existing relevance filter. Their effect on the success rate is mixed. For SPASS, they show a clear benefit (Fig. 13). For E and Vampire, they are harmful (Figs. 12 and 14). For some other problems, not shown, they show a slight benefit for Vampire. These results are difficult to interpret. The most we can state is that the advantages of theory symbols are not proved. If two theory symbols are added to each axiom, further strengthening the role of locality, then success rates fall sharply.

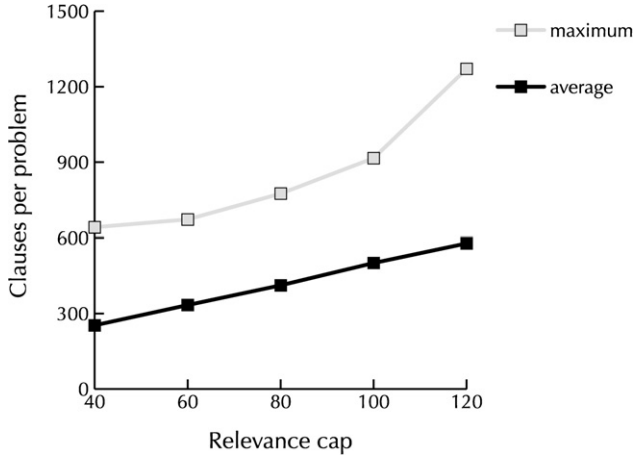

Fig. 11. Problem sizes produced by various caps.

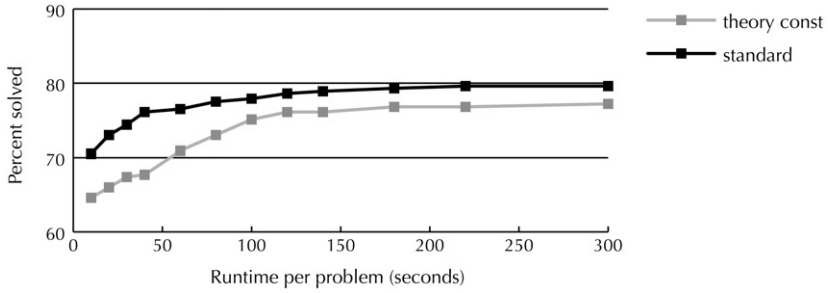

Fig. 12. E 0.99, effect of the theory symbol.

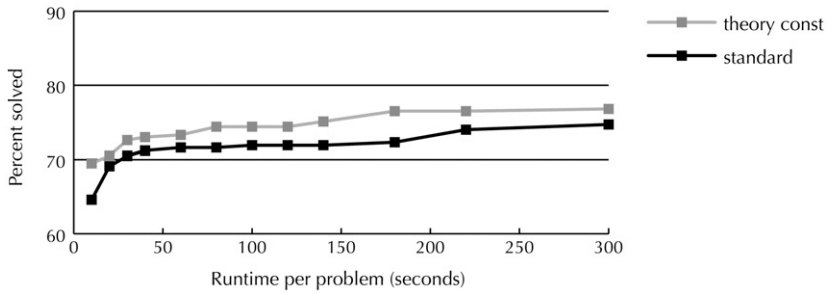

Fig. 13. SPASS 2.2, effect of the theory symbol.

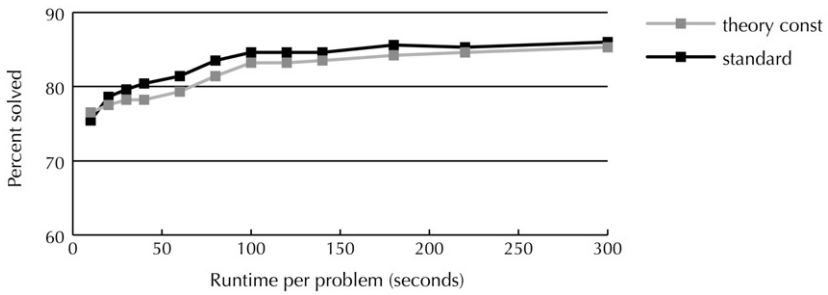

Fig. 14. Vampire 8.1, effect of the theory symbol.

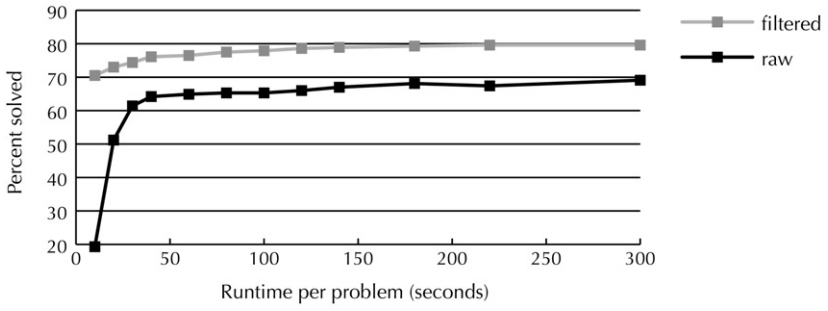

Fig. 15. E 0.99.

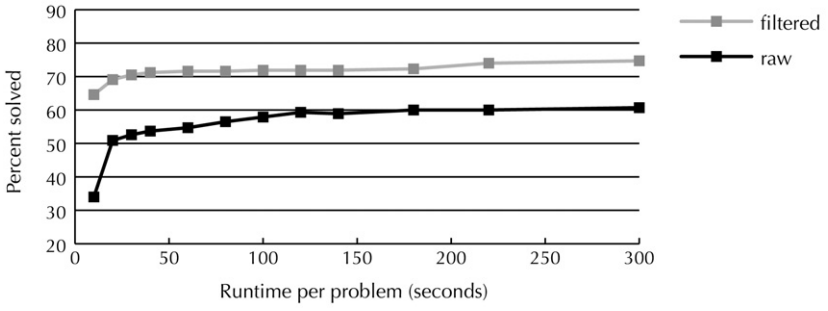

Fig. 16. SPASS 2.2, SOS enabled.

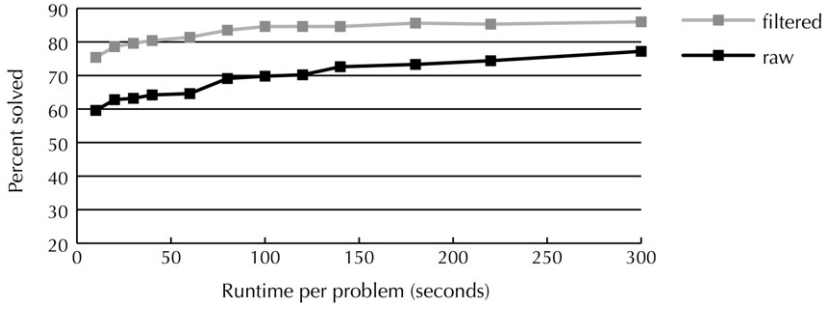

Fig. 17. Vampire 8.1, CASC mode.

5.3. *Empirical results for huge problems*

The results presented in Section 4.6 for Vampire in CASC mode and for the modified version of E may give the impression that ATPs can solve the relevance problem for themselves. That impression is false. Figs. 15 to 17 present graphs to show that none of the ATPs under consideration can cope with our huge problems. Despite its special modifications, E gets a great benefit from our relevance filter. Even Vampire's benefit at 300 seconds is a full nine percentage points. The filtered problems are based on the standard parameter settings used for testing at the time of this writing.[3] The huge problems of "all theorems" mode make relevance filtering a necessity.

[3] Theory symbols were disabled. The pass mark was 0.5, with a convergence factor of 3.2 and a relevance cap of 60. Blacklisting was enabled even for the unfiltered problems.

6. Conclusions

We wish to refute large, machine-generated sets of clauses. Experiments with the notion of "referenced axioms" demonstrate that reducing the problem size greatly improves the success rate. However, this technique introduces a dependence on past proof attempts, so we have sought methods of reducing the problem size through a simple analysis of the problem alone.

We have presented simple ideas for relevance filtering along with empirical evidence to demonstrate that they improve the success rate in a great variety of situations. The simplicity of our methods is in stark contrast to the tremendous sophistication of automated theorem provers. Although most ATPs have settings that can be adjusted to cope with large problems, their use requires expert knowledge, while our filter provides a uniform solution for all ATPs. It is surprising that such simple methods can yield benefits. We believe that the secret is our willingness to sacrifice completeness in order to improve the overall success rate.

Our method may be useful in other applications where processor time is limited and completeness is not essential. It is signature based, so it works for any problem for which the conjecture clauses have been identified. Our version of the filters operates on Isabelle theorems and assumes Isabelle's type system, but versions for standard first-order logic should be easy to devise. Users will probably have to tune our parameters—the pass mark, the convergence factor, the relevance cap—to their set of axioms.

Relevance filtering has allowed us to strengthen our approach to linking interactive and automatic theorem provers. Hand-selection of suitable lemmas is no longer necessary: we can provide a push-button interface, where all 7000 known theorems are candidates for inclusion in problems. With minor changes, our relevance filter is able to cope with this huge increase in problem sizes, delivering success rates that are only slightly worse than the previous approach. If filtering is switched off, success rates plummet (recall Figs. 15–17).

We do not feel that our methods can be significantly improved; their technological basis is too simple. More sophisticated techniques may yield more effective relevance filtering. A machine learning approach may be able to derive information from successful proofs, for example concerning which clauses work well together.

As an offshoot of the work reported above, we have submitted 565 problems to the TPTP library [24]. These are 285 raw problems plus 280 solutions: versions that have been automatically reduced as described in Section 3 above.[4]

Acknowledgements

The research was funded by the EPSRC grant GR/S57198/01 *Automation for Interactive Proof* and by the L4.verified project of National ICT Australia. Stephan Schulz has been extremely helpful throughout our work; he even provided a new version of the E prover, optimised for our problem set. Geoff Sutcliffe referred us to related work and gave advice. The referees provided detailed, valuable comments on both the workshop and journal versions of this paper.

References

[1] W. Ahrendt, B. Beckert, R. Hähnle, W. Menzel, W. Reif, G. Schellhorn, P.H. Schmitt, Integrating automated and interactive theorem proving, in: W. Bibel, P.H. Schmitt (Eds.), Automated Deduction—A Basis for Applications, vol. II. Systems and Implementation Techniques, Kluwer Academic Publishers, 1998, pp. 97–116.

[2] J. Avigad, K. Donnelly, Formalizing O notation in Isabelle/HOL, in: Basin and Rusinowitch [4], pp. 357–371.

[3] L. Bachmair, H. Ganzinger, Resolution theorem proving, in: Robinson and Voronkov [18], pp. 19–99, Chapter 2.

[4] D. Basin, M. Rusinowitch (Eds.), Automated Reasoning—Second International Joint Conference, IJCAR 2004, in: LNAI, vol. 3097, Springer, 2004.

[5] M. Bezem, D. Hendriks, H. de Nivelle, Automatic proof construction in type theory using resolution, Journal of Automated Reasoning 29 (3–4) (2002) 253–275.

[6] Condor: High throughput computing, http://www.cs.wisc.edu/condor/.

[7] M. Fuchs, D. Fuchs, Abstraction-based relevancy testing for model elimination, in: H. Ganzinger (Ed.), Automated Deduction—CADE-16 International Conference, in: LNAI, vol. 1632, Springer, 1999, pp. 344–358.

[8] J. Meng, L.C. Paulson, Experiments on supporting interactive proof using resolution, in: Basin and Rusinowitch [4], pp. 372–384.

[4] Executing the UNIX command `grep -l 'Paulson (2006)' -r Problems` in the TPTP directory generates the full list of problems.

[9] J. Meng, L.C. Paulson, Translating higher-order problems to first-order clauses, in: G. Sutcliffe, R. Schmidt, S. Schulz (Eds.), FLoC'06 Workshop on Empirically Successful Computerized Reasoning, CEUR Workshop Proceedings, vol. 192, 2006, pp. 70–80.
[10] J. Meng, C. Quigley, L.C. Paulson, Automation for interactive proof: First prototype, Information and Computation 204 (10) (2006) 1575–1596.
[11] L.C. Paulson, Generic automatic proof tools, in: R. Veroff (Ed.), Automated Reasoning and its Applications: Essays in Honor of Larry Wos, MIT Press, 1997 (Chapter 3).
[12] L.C. Paulson, The inductive approach to verifying cryptographic protocols, Journal of Computer Security 6 (1–2) (1998) 85–128.
[13] L.C. Paulson, A generic tableau prover and its integration with Isabelle, Journal of Universal Computer Science 5 (3) (1999) 73–87.
[14] L.C. Paulson, Relations between secrets: Two formal analyses of the Yahalom protocol, Journal of Computer Security 9 (3) (2001) 197–216.
[15] D.A. Plaisted, A. Yahya, A relevance restriction strategy for automated deduction, Artificial Intelligence 144 (1–2) (2003) 59–93.
[16] W. Reif, G. Schellhorn, Theorem proving in large theories, in: W. Bibel, P.H. Schmitt (Eds.), Automated Deduction—A Basis for Applications, vol. III. Applications, Kluwer Academic Publishers, 1998, pp. 225–240.
[17] A. Riazanov, A. Voronkov, The design and implementation of VAMPIRE, AI Communications 15 (2) (2002) 91–110.
[18] A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, Elsevier Science, 2001.
[19] J.A. Robinson, A machine-oriented logic based on the resolution principle, Journal of the ACM 12 (1965) 23–41.
[20] S. Schulz, System description: E 0.81, in: Basin and Rusinowitch [4], pp. 223–228.
[21] J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, I. Normann, M. Pollet, Proof development with Ωmega: The irrationality of $\sqrt{2}$, in: F. Kamareddine (Ed.), Thirty Five Years of Automating Mathematics, Kluwer Academic Publishers, 2003, pp. 271–314.
[22] G. Sutcliffe, A. Dvorsky, Proving harder theorems by axiom reduction, in: I. Russell, S. Haller (Eds.), Proceedings of the Sixteenth International Florida Artificial Intelligence Research Society Conference, AAAI Press, 2003, pp. 108–113.
[23] G. Sutcliffe, C. Suttner, CASC-20: The CADE ATP system competition, http://www.cs.miami.edu/~tptp/CASC/20/.
[24] G. Sutcliffe, C. Suttner, The TPTP problem library: CNF Release v1.2.1, Journal of Automated Reasoning 21 (2) (1998) 177–203.
[25] C. Weidenbach, Combining superposition, sorts and splitting, in: Robinson and Voronkov [18], pp. 1965–2013, Chapter 27.
[26] L. Wos, G.A. Robinson, D.F. Carson, Efficiency and completeness of the set of support strategy in theorem proving, Journal of the ACM 12 (4) (1965) 536–541.