Lightweight Relevance Filtering for Machine-Generated Resolution Problems

Jia Meng[1], Lawrence C. Paulson[2]
[1]National ICT, Australia
jiameng@nicta.com.au
[2]Computer Laboratory, University of Cambridge, U.K.
LP15@cam.ac.uk

Abstract

Irrelevant clauses in resolution problems increase the search space, making it hard to find proofs in a reasonable time. Simple relevance filtering methods, based on counting function 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. They should be applicable to other situations where the resolution problems are produced mechanically and where completeness is less important than achieving a high success rate with limited processor time.

1 Introduction

Resolution-based automatic theorem provers (ATPs) are *complete* for first-order logic: given unlimited resources, they will find a proof if one exists. Unfortunately, we seldom have unlimited resources. Removing unnecessary axioms from the problem reduces the search space and thus the effort needed to obtain proofs. Identifying unnecessary axioms while retaining completeness appears to be as difficult as proving the theorem in the first place. In general, it is hard to see how we can know that an axiom is redundant except by finding a proof without using it. Syntactic criteria may be helpful when the amount of redundancy is extreme, especially if we can relax the requirement for completeness.

The relevance problem dates back to the earliest days of resolution. As first defined by Robinson [Rob65], a literal is *pure* if it is not unifiable with a complementary literal in any other clause. Clauses containing pure literals can be removed without affecting consistency. This process is a form of relevance test, since pure literals often indicate that the axioms describe predicates not mentioned in the (negated) conjecture. It can be effective, but it is not a full solution.

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.

Our integration of Isabelle with ATPs [MQP06] generates small or large problems, depending on whether or not rewrite rules are included. A small problem occupies 200 kilobytes and comprises over 1300 clauses; a large problem occupies about 450 kilobytes and comprises 2500 clauses, approximately. Even our small problems look rather large to a resolution prover. We have run extensive tests with a set of 285 such problems (153 small, 132 large). Vampire does well on our problem set, if it is given 300 seconds per problem: it can prove 86 percent of them. Unfortunately, given 30 seconds per problem, which is more realistic for user interaction, Vampire's success rate drops to 53 percent. (See Fig. 7 in Sect. 5.) Can a cheap, simple relevance filter improve the success rate for short runtimes? Completeness does not matter to us: we are happy to forgo solutions to some problems provided we obtain a high success rate.

In the course of our investigations, we found that many obvious ideas were incorrect. For example, ATPs generate hundreds of thousands of clauses during their operation, so 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 link-up between the two, mentioning related work on other such link-ups (Sect. 2). We describe our initial attempts to improve the success rate of our link-up (Sect. 3), and then describe a family of related relevance filters (Sect. 4). We proceed to our experimental results, illustrating the benefits of filtering through a series of graphs (Sect. 5). Finally, we present brief conclusions (Sect. 6).

2 Background

Resolution theorem provers work by deriving a contradiction from a supplied set of *clauses* [BG01]. 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 versions 0.9 and 0.91dev001 [Sch04], SPASS V2.2 [Wei01] and Vampire 8 [RV01a].[1]

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 $\exists x\, \forall y\, P(x,y) \to \forall y\, \exists x\, P(x,y)$ automatically. Developers of interactive tools would naturally like to harness the power of ATPs, while preserving the intuitive reasoning style that their users expect.

We are adding automated support to the interactive prover Isabelle. There are many differences between our project and other related work [BHdN02, SBF$^+$03]. The chief difference is that Isabelle's built-in automated support gives the ATPs tough competi-

[1] We downloaded Vampire from the CASC-20 website, http://www.cs.miami.edu/ tptp/CASC/20/; it calls itself version 7.45.

tion. By typing *auto*, the Isabelle user causes approximately 2000 lemmas to be used as rewriting rules and for forward and backward chaining. A related tool, *blast*, 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 this set equality [Pau97]:

$$\bigcup_{i\in I}(A_i \cup B_i) = \Big(\bigcup_{i\in I} A_i\Big) \cup \Big(\bigcup_{i\in I} B_i\Big)$$

Set theory problems, and the combinatory logic examples presented in that paper, remain difficult for automatic theorem provers. When we first got our link-up working, we were disappointed to find that ATPs were seldom better than *auto*. We have devoted much effort to improving its success rate. We found that bugs in our link-up were partly to blame for the poor results. Much of our effort went to improving the problem presentation; for example, we found a more compact representation of types. 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.

Of previous work, the most pertinent is the integration between the Karlsruhe Interactive Verifier (KIV) and the tableau prover 3TAP, by Ahrendt and others [ABH$^+$98]. Reif and Schellhorn [RS98] present a component of that integration: an algorithm for removing redundant 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. We did not feel that this method would work with Isabelle theories, which tend to be large: a typical theory introduces many functions and definitions, and proves many theorems. For instance, the theory of the natural numbers proves nearly 200 theorems. Also, a typical Isabelle problem lies at the top of a huge theory hierarchy that includes the full Isabelle/HOL development, any included library theories, and theories for the user's problem domain. We decided to try other methods, which are described in the sequel.

The Isabelle-ATP linkup generates problems that contain conjecture clauses along with clauses from four other sources:

- *Classical* clauses arise from the theorems Isabelle uses for forward and backward chaining.

- *Simplification* clauses arise from the theorems Isabelle uses as rewrite rules. They contain equalities.

- *Arity* and *class inclusion* clauses do not correspond to Isabelle theorems. Instead, they express aspects of Isabelle's type system. 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

  $$\forall\,\tau[type(\tau) \rightarrow order(list(\tau))]$$

  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 [MP04, MQP06].

Although the arity and class inclusion clauses typically number over one thousand, they probably 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 probably suffices to remove redundant arity and class inclusion clauses, so ATPs may delete most of them immediately.

3 Initial Experiments

We have tuned the Isabelle-ATP link-up with the help of a set of problems in clause form. We obtained these by modifying our link-up to save the clause form problems it was producing. They simulate attempted calls to our system at various points in several Isabelle proofs. The original Isabelle proofs for some of these problems require multiple steps, explicit quantifier instantiations, or other detailed guidance. The set now numbers 285 problems. Many have trivial proofs and the only difficulty lies in the problem size. Other problems take a few minutes to prove, and a few remain unsolved.

Our first task was to verify that the problems were solvable. If a problem could not be proved by any ATP, we could remove irrelevant clauses manually, using our knowledge of the problem domain, in the hope of rendering it provable. This process was too laborious to carry out for every failing problem. Over time, with the help of the filtering techniques described below, we were able to obtain proofs for all but five of our problems. We also uncovered problems that were incorrectly posed, lacking crucial lemmas, and found several bugs. 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).

The laborious hand-minimization can be automated. 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 referenced. 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 some 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 easily checked to be a subset of the original problem, and with few exceptions it is easy to prove. Fewer suspect problems require hand examination.

Using only referenced axioms does not guarantee that problems will be small. At present, there are 405 referenced clauses. Most of these are specific to various problem domains; approximately 150 of these correspond to standard Isabelle/HOL theorems, and are common to all problems. A proof about protocol verification could have another 90 clauses, for a problem size of about 240. Two points are noteworthy:

1. The problems are smaller if not small, and
2. referenced clauses may be somehow more suitable for automated proof than other clauses.

This second point suggests the concept of *pathological* clauses. Is it possible that there are a few particularly bad clauses whose removal would improve the success rate?

It is difficult to test the hypothesis that the success rate is lowered by a few pathological clauses. 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. Surely a pathological clause would contain highly general literals such as $X = Y$, $X < Y$, $X \in Y$, or their negations.

We eventually blacklisted 148 theorems from the standard Isabelle/HOL library. A theorem could be blacklisted for various reasons, such as having too big a clause form, being too similar 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 already 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 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 Signature-Based Relevance Filters

Automatic relevance filtering is clearly more attractive than any method requiring manual inspection of clauses. We decided not to adopt Reif and Schellhorn's approach [RS98], which required analysis of the theory in which each problem was set, and instead to define 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 [WRC65], which dates from 1965, ensures that all inference rule applications involve at least one clause derived from a conjecture clause. It prevents inferences among the axioms and makes 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.

We tried many simple relevance filters based on the conjecture clauses. We already knew that simple methods could be effective: in an earlier experiment, we modified the link-up 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 [FF99] 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 [PY03] 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 (in 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, thereby 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 function symbols with them. 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 (Sect. 5).

4.2 A Passmark-Based Algorithm

Our filtering strategies abandon the all-or-nothing approach. Instead, we use a measure of relevance, and 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 functions, 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 functions become relevant. Iterate this process 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 a relevance mark to each clause and this mark may be increased during the filtering process. The pseudo-code for our algorithm is shown in Figure 1.

The pseudo-code should be self-explanatory. We only give a few more comments below.

- When the function `relevant_clauses` is first called, the working relevant clauses set `W` contains the goal clauses, while `T` contains all the axiom clauses and `U` is empty.
- In function `find_clause_mark`, `|R|` is the number of elements in the set `R`.
- Isabelle allows overloading of functions, so that for example `<=` can denote the usual ordering on integers as well as subset inclusion. Therefore function `find_clause_mark` regards two functions as matching only if their types match as well.
- The multiplication by `P_M` in function `find_clause_mark` makes the relevance test increasingly strict as the distance from the conjecture clauses increases, which keeps the process focussed around the conjecture clauses and prevents too many clauses from being taken as relevant.

4.3 Using the Set of Relevant Functions

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 functions, 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 parameter. If $c = 2$, for example, then each iteration halves the difference $1 - p$. The algorithm appears in Figure 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, Figure 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 formalize Isabelle's type system. Vampire's success rate peaks sharply at 0.6, by which time the average problem size has decreased from 909 to 142 clauses. Since the aim of these experiments was to determine the best

```
function relevant_clauses:
input:  W, working relevant clauses set
        T, working irrelevant clauses set
        P, pass mark
        U, used relevant clauses set
        (Each clause is attached with a relevance mark)
output: relevant clauses to be input to ATPs
begin
while W not empty do {
   for each clause-relevance-mark pair (C,M) in T do
     { update_clause_mark (W, (C,M)) }
   #partition (C,M) pairs in T into two sets
   Rel set contains (C,M) if $P \leq M$
   Irrel := T - Rel;
   U := $W \cup U$;
   W := Rel;
   T := Irrel;
}
return U;
end


function update_clause_mark:
input:  W, relevant clauses set
        (C,M), a clause-mark pair
effect: updates the relevance mark of C
begin
for each clause-mark (P,P_M) in W do {
  PC := functions_of P;
  CF := functions_of C;
  R := $CF \cap PC$;          #where names and types agree
  IR := $CF - R$;             #remaining functions of clause C
  M := max(M, P_M * |R| / (|R| + |IR|));
}
end
```

Figure 1: A Passmark-Based Filtering Strategy

```
function relevant_clauses:
input:  RF, set of relevant functions
        T, working irrelevant clauses set
        A, relevant clauses accumulator
        P, pass mark
output: relevant clauses to be input to ATPs
begin
repeat {
  for each clause $C_i$ in T do
    { $M_i$ := clause_mark (RF,$C_i$) }
  Rel := set of all clauses $C_i$ such that P $\leq$ $M_i$
  T := T - Rel;
  A := A $\cup$ Rel;
  P := P + (1 - P) / c;
  RF := (functions_of Rel) $\cup$ RF;
} until Rel is empty
return A;
end

function clause_mark:
input:  RF, a set of relevant functions
        C, a clause
output: the relevance mark of C
begin
CF := functions_of C;
R := CF $\cap$ RF;          #where names and types agree
IR := CF $-$ R;             #remaining functions of clause C
return |R| / (|R| + |IR|);
end
```

Figure 2: An Improved Filtering Strategy Using a Set of Relevant Functions

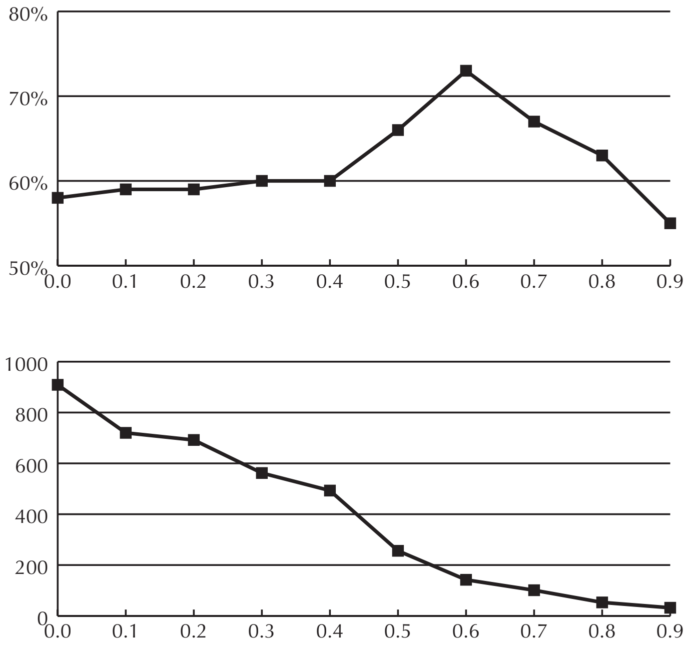

Figure 3: Success Rates and Problem Sizes Against Pass Marks

values for p and c, the choice of ATP probably makes little difference. In particular, if the filter has removed essential axioms, then no ATP will find a proof; this 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 tests that took hundreds of hours of processor time.

4.4 Taking Rarity into Account

Another refinement takes account of the relative frequencies of the functions in the clause set. Some functions are common while others are rare. An occurrence of a rare function would seem to be a strong indicator of relevance, 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 functions. Taking rarity into account in n would prevent the inclusion of clauses that involve another rare function, such as a Skolem function (each Skolem function is rare, since it only occurs in a few clauses).

This strategy requires a suitable frequency function, which calculates the weight of a symbol according to the number of its occurrences in the entire set of axiom clauses. It is similar to our strategy mentioned in Sect.4.3, except for the function `clause_mark` shown in Figure 4.

A few explanations may be helpful here.

- The relevance mark of a clause `C` calculated by `clause_mark` is not the percentage of relevant functions in `C` any more. Instead, we use `func_weight` function to compute the sum of relevant functions' marks weighted by their frequencies.

- A suitable frequency function (`frequency_function`) is needed. After much test-

```
function clause_mark:
input:  RF, a set of relevant functions
        C, a clause
        ftab, a table of the number of occurrences of each function in the clause set
output: the relevance mark of C
begin
CF := functions_of C;
R := CF ∩ RF;          #where names and types agree
IR := CF − R;          #remaining functions of C
M := 0;
for each function F in R do { M := M + func_weight (ftab, F) }
return (M / (M + |IR|));
end

function func_weight:
input:  ftab, a table of the number of occurrences of each function in the cause set
        F, a function symbol
output: the frequency weighted mark for F
begin
freq := number_of_occurrences (ftab, F);
return (frequency_function freq);
end
```

Figure 4: A Filtering Strategy for Rarely-Occurring Functions

ing we found that it should decrease gently as the frequency increases. If a function occurs n times in the problem, then we increase its contribution from 1 to $1 + 2/\log(n+1)$. An even gentler decrease, such as $1 + 1.4/\log(\log(n+2))$, can work well, but something like $1 + 1/\sqrt{n}$ penalizes frequently-occurring functions too much. Hence, our definition of `frequency_function` is

```
frequency_function n = 1 + 2/log(n+1)
```

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 provided it was not an equation; an equation was simple provided its left- or right-hand side was ground (that is, 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 favour shorter clauses failed to produce any increase in the success rate.

Definition expansion is another refinement. If a function 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. Definition expansion is beneficial, but its effect is small.

As of this writing, our system still contains a manually produced blacklist of 117 (down from 148) HOL theorems. We also have a whitelist of theorems whose inclusion is forced; it contains one single theorem (concerning the subset relation), which we found that the filter was frequently rejecting. We can probably reduce the blacklist drastically by checking which of those theorems would survive relevance filtering. However, having a high success rate is more important to us than having an elegant implementation.

5 Empirical Results

Extensive testing helped us determine which methods worked best and to find the best settings of the various parameters. The graphs compare the success rates of filtered problems against raw ones. The runtime per problem increases from 10 to 300 seconds. Success rates are plotted on a scale ranging from 40 to 90 percent. We tested the three provers we have found to be best—E, SPASS and Vampire—in their default mode and with alternative settings. E version 0.9 "Soom" is surpassed by version 0.91dev001 (Fig. 5), a development version of "Kanyam" that includes heuristics designed for our problem set. (Version 0.9 surpasses the official Kanyam release, E version 0.91, on our problems.) SPASS (Fig. 6) seems to perform better if SOS is enabled and splitting is disabled.[2] Vampire does extremely well in its CASC mode (Fig. 7).

We ran these tests on a bank of Dual AMD Opteron processors running at 2400MHz. The Condor system[3] managed our batch jobs.

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

[3] `http://www.cs.wisc.edu/condor/`

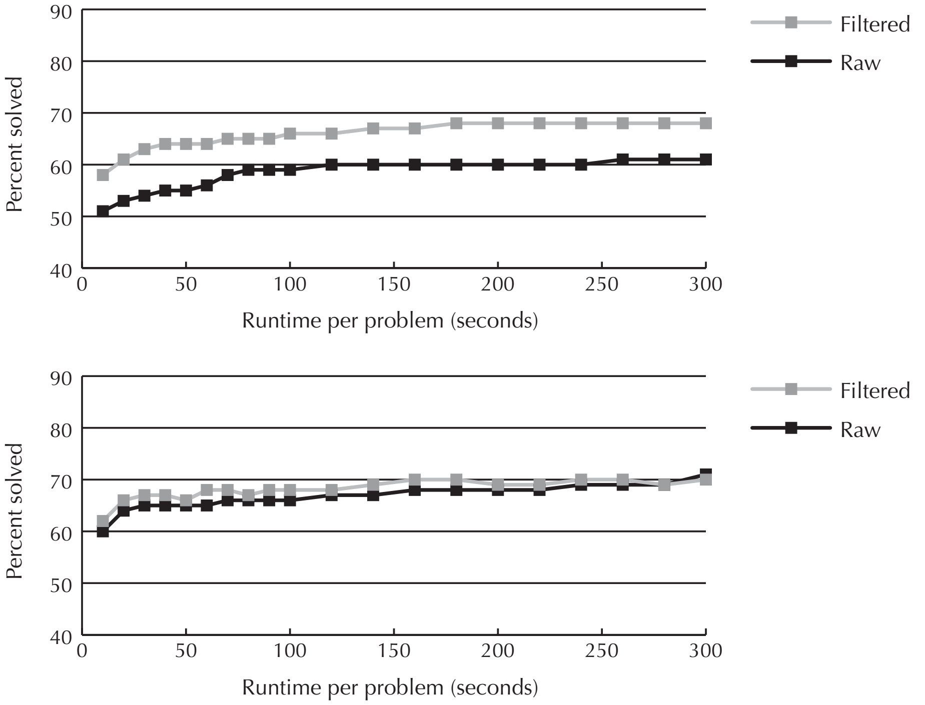

Figure 5: E, Versions 0.9 and 0.91dev001

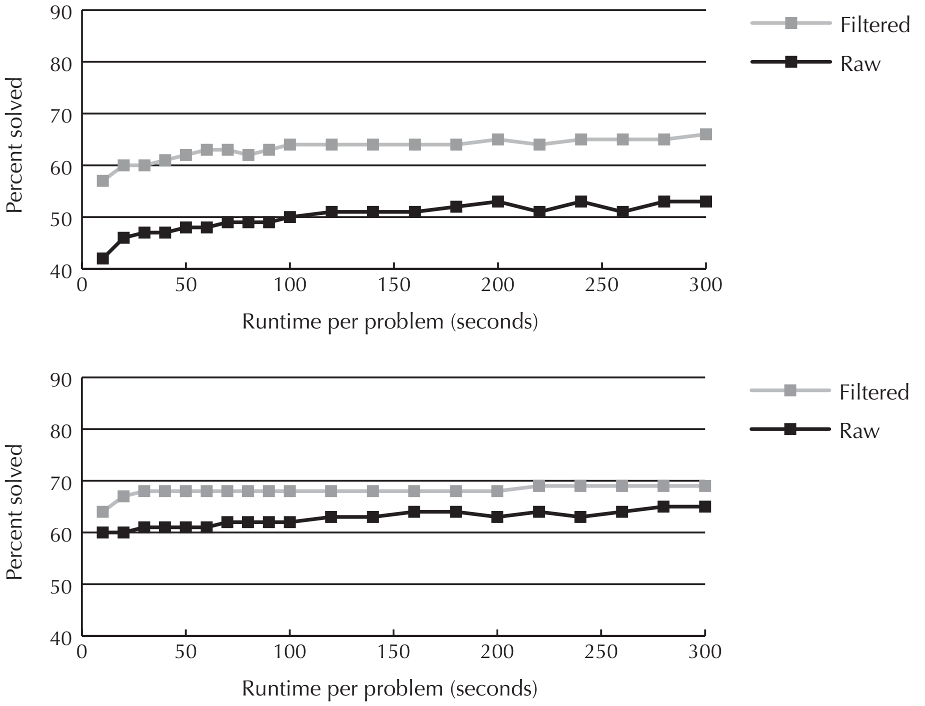

Figure 6: SPASS, Default Settings and with SOS

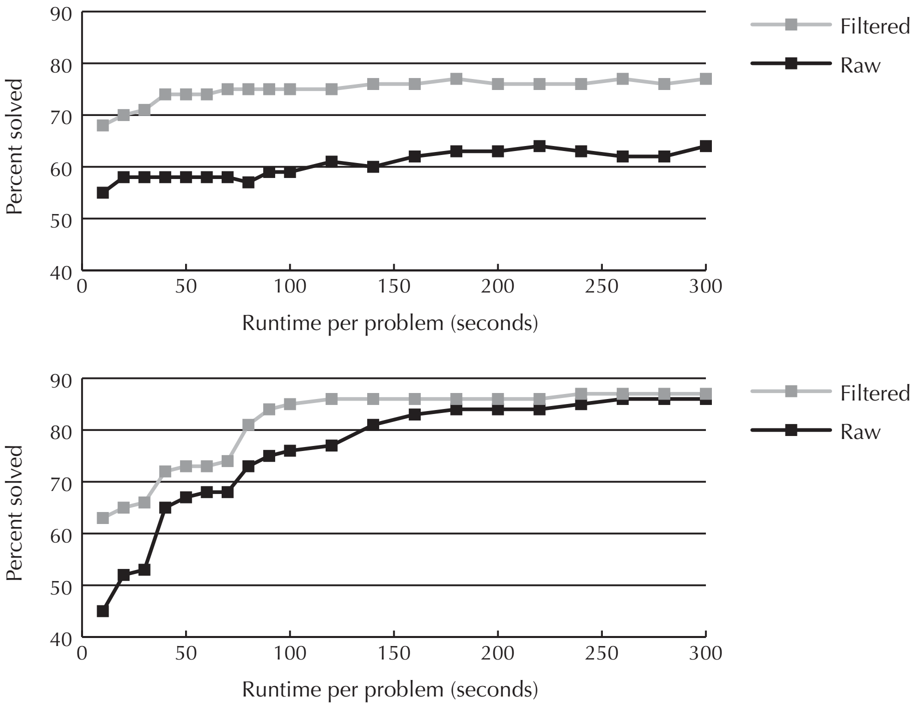

Figure 7: Vampire, Default Settings and in CASC Mode

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 eight percent of the filtered problems appear to be missing essential clauses, compared with their automatically-reduced counterparts. Perhaps gains are made elsewhere, or these eight percent are too difficult to prove anyway. A further surprise is that, with the exception of Vampire running in CASC mode, increasing processor time to 300 seconds does not give an advantage to the raw problems. The success rate for raw problems should eventually exceed that of the filtered ones, but the crossover point appears to be rather distant. Exceptions are E version 0.91dev001, where crossover appears to be taking place at 280 seconds, and Vampire in CASC mode, where by 300 seconds the two lines are close.

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. 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.

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

> 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.

Such a setting could probably be added to other ATPs, and in a fashion that preserves completeness.

The weighting mechanisms of other ATPs may be able to simulate our filter. However, finding good configurations requires expert knowledge of each ATP being used. We were never able to improve upon the default configuration of E, despite its good documentation and helpful developer. (Many ATPs do not have these benefits.) In contrast, our filter provides a uniform solution for all ATPs.

We do not feel that our methods can be significantly improved; their technological basis is too simple. More sophisticated techniques, perhaps based on machine learning, may yield more effective relevance filtering.

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

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 provided a new version of the E prover, optimized for our problem set. The referees' comments on this paper were valuable.

References

[ABH$^+$98] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter H. Schmitt. Integrating automated and interactive theorem proving. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction— A Basis for Applications*, volume II. Systems and Implementation Techniques, pages 97–116. Kluwer Academic Publishers, 1998.

[BG01] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Robinson and Voronkov [RV01b], chapter 2, pages 19–99.

[BHdN02] Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automatic proof construction in type theory using resolution. *Journal of Automated Reasoning*, 29(3-4):253–275, 2002.

[BR04] David Basin and Michaël Rusinowitch, editors. *Automated Reasoning — Second International Joint Conference, IJCAR 2004*, LNAI 3097. Springer, 2004.

[FF99] Marc Fuchs and Dirk Fuchs. Abstraction-based relevancy testing for model elimination. In Harald Ganzinger, editor, *Automated Deduction — CADE-16 International Conference*, LNAI 1632, pages 344–358. Springer, 1999.

[MP04] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In Basin and Rusinowitch [BR04], pages 372–384.

[MQP06] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 2006. in press.

[Pau97] Lawrence C. Paulson. Generic automatic proof tools. In Robert Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, chapter 3. MIT Press, 1997.

[PY03] David A. Plaisted and Adnan Yahya. A relevance restriction strategy for automated deduction. *Artificial Intelligence*, 144(1-2):59–93, March 2003.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[RS98] Wolfgang Reif and Gerhard Schellhorn. Theorem proving in large theories. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume III. Applications, pages 225–240. Kluwer Academic Publishers, 1998.

[RV01a] Alexander Riazanov and Andrei Voronkov. Vampire 1.1 (system description). In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning — First International Joint Conference, IJCAR 2001*, LNAI 2083, pages 376–380. Springer, 2001.

[RV01b] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning.* Elsevier Science, 2001.

[SBF$^+$03] Jörg Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, Immanuel Normann, and Martin Pollet. Proof development with Ωmega: The irrationality of $\sqrt{2}$. In Fairouz Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, pages 271–314. Kluwer Academic Publishers, 2003.

[Sch04] Stephan Schulz. System description: E 0.81. In Basin and Rusinowitch [BR04], pages 223–228.

[Wei01] Christoph Weidenbach. Combining superposition, sorts and splitting. In Robinson and Voronkov [RV01b], chapter 27, pages 1965–2013.

[WRC65] Lawrence Wos, George A. Robinson, and Daniel F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the ACM*, 12(4):536–541, 1965.