Simple and Efficient Clause Subsumption with Feature Vector Indexing

Stephan Schulz

Institut für Informatik, Technische Universität München,
D-80290 München, Germany
schulz@eprover.org

Abstract. This paper describes *feature vector indexing*, a new, non-perfect indexing method for clause subsumption. It is suitable for both forward (i.e., finding a subsuming clause in a set) and backward (finding all subsumed clauses in a set) subsumption. Moreover, it is easy to implement, but still yields excellent performance in practice. As an added benefit, by restricting the selection of features used in the index, our technique immediately adapts to indexing modulo arbitrary AC theories with only minor loss of efficiency. Alternatively, the feature selection can be restricted to result in *set subsumption*. Feature vector indexing has been implemented in our equational theorem prover E, and has enabled us to integrate new simplification techniques making heavy use of subsumption. We experimentally compare the performance of the prover for a number of strategies using feature vector indexing and conventional sequential subsumption.

Keywords: First-order theorem proving, indexing, subsumption.

1 Introduction

First-order theorem proving is one of the core areas of automated deduction. In this field, saturating theorem provers currently show a significant lead compared to systems based on other paradigms, such as top-down reasoning or instance-based methods. One of the reasons for this lead is the compatibility of saturating calculi with a large number of redundancy elimination techniques, as e.g. tautology deletion, rewriting, and *clause subsumption*. Subsumption allows us to discard a clause (i.e., exclude it from further proof search) if a (in a suitable sense) more general clause exists. In many cases, subsumption can eliminate between 50% and 95% of all clauses under consideration, with a corresponding decrease in the size of the search state.

Subsumption of multi-literal clauses is an NP-complete problem [6]. If some attention is paid to the implementation, the worst case is rarely (if ever) encountered in practice, and single clause-clause subsumption tests rarely form a critical bottleneck. However, the sheer number of possible subsumption relations to test for means that a prover can spend a significant amount of time in subsumption-related code. Even in the case of our prover E [17,19], which,

M.P. Bonacina and M.E. Stickel (Eds.): McCune Festschrift, LNAI 7788, pp. 45–67, 2013.

because of its DISCOUNT loop proof procedure, minimizes the use of subsumption, frequently between 10% and 20% of all time was spent on subsumption, with much higher values observed occasionally. The cost of subsumption systematically increases if other simplification techniques based on subsumption are implemented.

In a saturating prover, we are most often interested in subsumption relations between whole sets of clauses and a single clause. In *forward subsumption* (FS), we want to know if *any* clause from a set subsumes a given clause. In *backward subsumption* (BS), we want to find all clauses in a set that are subsumed by a given clause. This observation can be used to speed up subsumption, by using *indexing techniques* that return only candidates suitable for a given subsumption relation from a set of clauses, thus reducing the number of explicit subsumption tests necessary. A *perfect index* will return exactly the necessary clauses, whereas a *non-perfect* index should return a superset of candidates for which the desired relationship has to be verified.

Term indexing techniques have been used in theorem provers for some time now (see [10] for first implementations in Otter or [3,4,22] for increasingly up-to-date overviews). However, lifting term indexing to clause indexing is not trivial because the associative and commutative properties of the disjunction, and the symmetry of the equality predicate, are hard to handle. In many cases, (perfect) term indexing is used only to retrieve subsumption candidates, i.e., to implement non-perfect clause indexing (see e.g. [26]). Moreover, often two different indices are used for forward- and backward subsumption, as e.g. in the very advanced indexing schemes currently implemented in Vampire [15]. Similar problems affect term indexing modulo associativity or commutativity. McCune's EQP, for example, performs unit-subsumption, but disables all indexing as soon as some symbols are declared AC or C [9].

We suggest a new indexing technique based on *subsumption-compatible* numeric clause features. It is much easier to implement than known techniques, and the same, relatively compact data structure can be used for both forward- and backward subsumption. We have implemented the new technique for E 0.8, and, in more polished and configurable ways, for later versions, with excellent results.

In this paper, describe the new technique. We also discuss how it has been integrated into E, and how it also serves to speed up *contextual literal cutting*, a subsumption-based simplification technique that has given another boost to E. We present the results of various experiments to support our claims.

2 Preliminaries

We are primarily interested in first-order formulae in clause normal form in this paper. We assume the following notations and conventions. Let F be a finite set of function symbols. We write $f/n \in F$ to denote f as a function symbol with arity n. Functions symbols are written as lower case letters. We usually employ a, b, c for function symbols with arity 0 (constants), and f, g, h for other function

symbols. Let V be an enumerable set of variable symbols. We use upper case letters, usually X, Y, Z to denote variables. The set of all *terms* over F and V, $Term(F, V)$, is defined as the smallest set fulfilling the following conditions:

1. $X \in Term(F, V)$ for all $X \in V$
2. $f/n \in F$, $s_1, \ldots, s_n \in Term(F, V)$ implies $f(s_1, \ldots, s_n) \in Term(F, V)$

We typically omit the parentheses from constant terms, as for example in the expression $f(g(X), a) \in Term(F, V)$.

An (equational) *atom*[1] is an unordered pair of terms, written as $s \simeq t$. A *literal* is either an atom, or a negated atom, written as $s \not\simeq t$. We define a negation operator on literals as $\overline{s \simeq t} = s \not\simeq t$ and $\overline{s \not\simeq t} = s \simeq t$. If we want to write about arbitrary literals without specifying polarity, we use $s \mathrel{\dot{\simeq}} t$, or, in less precise way, $l, l_1, l_2, \ldots$. Note that $\simeq$ is commutative in this notation.

A *clause* is a multiset of literals, interpreted as an implicitly universally quantified disjunction, and usually written as $l_1 \vee l_2 \ldots \vee l_n$. Please note that in this notation, the $\vee$ operator is associative and commutative (but not idempotent). The empty clause is written as $\Box$, and the set of all clauses as $Clauses(F, V)$. A *formula* in clause normal form is a set of clauses, interpreted as a conjunction.

A *substitution* is a mapping $\sigma : V \to Term(F, V)$ with the property that $Dom(\sigma) = \{X \in V \mid \sigma(X) \neq X\}$ is finite. It is extended to a function on terms, atoms, literals and clauses in the obvious way.

A *match* from a term (atom, literal, clause) s to another term (atom, literal, clause) t is a substitution σ such that $\sigma(s) \equiv t$, where $\equiv$ on terms denotes syntactic identity and is lifted to atoms, literal, clauses in the obvious way, using the unordered pair and multiset definitions.

3 Subsumption

If we consider a (multi)set of clauses not all of the clauses necessarily contribute to the meaning of it. Often, some clauses are *redundant*. Some clauses do not add any new constraints on the possible models of a formula, because they are already implied by other clauses. Depending on the mechanism of reasoning employed, we can delete some of these clauses, thus reducing the size of the formula (and hence the difficulty of finding a proof). In the case of current saturating calculi, *subsumption* is a technique that allows us to syntactically identify certain clauses that are implied by another clause, and can usually be discarded without loss of completeness. We can specify the (multiset) subsumption rule as a deleting simplification rule (i.e., the clauses in the precondition are *replaced* by the clauses in the conclusion) as follows:

$$\text{(CS)}\ \frac{\sigma(C) \vee \sigma(R) \qquad C}{C}$$

where σ is a substitution, C and R are arbitrary (partial) clauses

[1] For our current discussion, the non-equational case is a simple special case and can be handled by encoding non-equational atoms as equalities with a reserved constant $\$true$. We will still write non-equational literals in the conventional manner, i.e., $p(a)$ instead of $p(a) \simeq \$true$.

In other words, a clause C' is subsumed by another clause C if there is an instance $\sigma(C)$ that is a sub-multiset of C'.

This version of subsumption is used by most modern saturation procedures. It is particularly useful in reducing search effort, since it allows us to discard larger clauses in favor of smaller clauses. Smaller clauses typically have fewer inference positions and generate fewer and smaller successor clauses.

Individual clause-clause subsumption relations are determined by trying to find a simultaneous match from all literals in the potentially subsuming clause to corresponding literals in the potentially subsumed clause. This is usually implemented by a backtracking search over permutations of literals in the potentially subsumed clause (and in the equational case, permutations of terms in equational literals).

Most of the techniques used to speed up subsumption try to detect failures early by testing necessary conditions. Those include compatibility of certain clause measures (discussed in more detail below) and existence of individually matched literals in the potentially subsumed clause for each literal in the potentially subsuming clause. Additionally, in many cases certain permutations of literals can be eliminated by partially ordering literals in a clause with a suitable ordering.

However, while individual subsumption attempts are reasonably cheap in practice, the number of potential subsumption relations to test for in saturation procedures is very high. Using a straightforward implementation of subsumption, we have measured up to 100,000,000 calls to the subsumption subroutine of our prover E in just 5 minutes on a 300 MHz SUN Ultra-60 for some proof tasks. Thus, the overall cost of subsumption is significant.

3.1 Subsumption Variants

In addition to standard multiset subsumption, there are a number of other subsumption variants and related techniques.

The definition of *set subsumption* is identical to that of multiset subsumption, except in that clauses are viewed as sets of literals (i.e. a single literal occurs at most once in a given clause). This allows for a slightly stronger subsumption relation: $p(X) \vee p(Y)$ can subsume $p(a)$ with set subsumption, but not with multiset subsumption. Set subsumption can e.g. be used in preprocessing. However, for most saturation-based calculi (especially those for which factorization is an explicit inference rule), the fact that a clause can subsume some of its factors causes loss of completeness.

Subsumption modulo AC is a stronger version of multiset or set subsumption, where we do not require that the instantiated subsuming clause is a subset of the subsumed clause, but only that it is equal to a subset modulo a specified theory for associative and commutative function symbols. For example, if f is commutative, then $p(f(a, X))$ subsumes $p(f(b, a)) \vee q(a)$. This variant of subsumption is useful for systems that reason modulo AC, as e.g. SNARK [25].

Equality subsumption (also known as *functional subsumption*) allows an equational unit clause to potentially subsume another clause with an equational literal

implied by the potential subsumer. It can be described by the following simplification rule:

$$\text{(ES)}\quad \frac{s \simeq t \quad u[p \leftarrow \sigma(s)] \simeq u[p \leftarrow \sigma(t)] \vee R}{s \simeq t}$$

It is typically only applied if $s \simeq t$ cannot be used for rewriting, i.e. if $\sigma(s) \simeq \sigma(t)$ cannot be oriented). This rule is implemented by E and a number of other provers, including at least the completion-based systems Waldmeister [8] and DISCOUNT [2], as well as in EQP [11].

Finally, a simplification rule that has been popularized by implementation in SPASS [28] and Vampire [14], and is sometimes called *subsumption resolution* or *clausal simplification*, combines resolution and subsumption to cut a literal out of a clause. In the context of a modern superposition calculus, we believe the rule can be better described as *contextual literal cutting*:

$$\text{(CLC)}\quad \frac{\sigma(C) \vee \sigma(R) \vee \sigma(l) \qquad C \vee \bar{l}}{\sigma(C) \vee \sigma(R) \qquad\qquad C \vee \bar{l}}$$

where $\bar{l}$ is the negation of l and σ is a substitution

It can be implemented via a normal subsumption engine (by negating each individual literal in turn, and then testing for subsumption) and is implemented thus at least in E and Vampire. Depending on how and when this rule is applied, it can increase the number of required subsumption tests by many orders of magnitude.

3.2 Saturation Procedures and Clause Set Subsumption

Most modern saturating theorem provers use a variant of the *given clause algorithm* that was popularized by Bill McCune's *Otter*. This algorithm maintains two sets of clauses, the *processed clauses* P and the *unprocessed clauses* U. At the start, all clauses are unprocessed. The algorithm repeatedly picks one clause from U, and performs all generating inferences using this *given clause* and the clauses in P as premises. It then moves the given clause to P and adds all newly generated clauses to U.

The original *Otter loop* performs simplification and subsumption between all clauses, both processed an unprocessed. A variant, originally implemented in DISCOUNT, restricts simplification to only allow processed clauses as side premises in simplification. Figure 1 shows a sketch of the main proof procedure of our prover E, an implementation of the DISCOUNT loop for full superposition.

Please observe that subsumption appears in exactly two different places and exactly two different roles in this procedure: First, we test if the *given clause* g is subsumed by *any* clause in P. In other words, we want to know if a single clause is subsumed by any clause from a set. This is usually called *forward subsumption*.

If the given clause is not redundant, we next want to find *all* clauses in P that are subsumed by g. Again, we have an operation between a single clause and a whole set, in this case called *backward subsumption*.

It is obvious that we can implement forward and backward subsumption naively by sequentially testing each clause from P against g. This implementation was

Prover state: $U \cup P$
U contains *unprocessed* clauses, P contains *processed* clauses.
Initially, all clauses are in U, P is empty.
The *given clause* is denoted by g.

```
while U ≠ {}
   g = delete_best(U)
   g = simplify(g, P)
   if g == □
      SUCCESS, Proof found
   if g is not subsumed by any clause in P (or otherwise redundant w.r.t. P)
      P = P\{c ∈ P | c subsumed by (or otherwise redundant w.r.t.) g}
      T = {c ∈ P | c can be simplified with g}
      P = (P\T) ∪ {g}
      T = T ∪ generate(g, P)
      foreach c ∈ T
         c = cheap_simplify(c, P)
         if c is not trivial
            U = U ∪ {c}
SUCCESS, original U is satisfiable
```

Remarks: delete_best(U) finds and extracts the clause with the best heuristic evaluation from U. generate(g, P) performs all generating inferences using g as one premise, and clauses from P as additional premises. simplify(c, S) applies all simplification inferences in which the main (simplified) premise is c and all the other premises are clauses from S. This typically includes full rewriting and (CLC). cheap_simplify(c, S) works similarly, but only applies inference rules with a particularly efficient implementation, often including rewriting with orientable units, but usually not (CLC).
Similarly, in this context, a clause is *trivial*, if it can be shown redundant with simple, local syntactic checks. If we test for redundancy, we also apply more complex and non-local techniques.

Fig. 1. Saturation procedure of E

used, for instance, in early versions of SPASS [28], and was used in E up to version 0.71. However, this does not make use of the fact that we are interested in subsumption relations between individual clauses and usually only slowly changing *clause sets*. The idea behind clause indexing is to *preprocess* the clause set so that subsumption queries can be answered more efficiently than by sequential search.

4 Feature Vector Indexing

Indexing for subsumption is used by a number of provers. Most existing implementations (e.g. [26,27,10]) use a variant of *discrimination tree indexing* on terms to build a index for forward subsumption, often for non-perfect indexing.

Indexing for backward subsumption is less frequent, and usually based on a variant of path indexing. We will now present a new and much simpler technique suitable for both forward and backward subsumption.

Our technique is based on the compilation of necessary conditions on numeric clause features. Essentially, a clause is represented by a vector of feature values, and subsumption candidates are identified by comparisons of feature vectors. Feature vectors for clause sets are compiled into a *trie* data structure to quickly identify candidate sets.

4.1 Subsumption-Compatible Clause Features

A *(numeric) clause feature function* (or just *feature*) is a function mapping clauses to natural numbers, $f : \mathit{Clauses}(F, V) \to \mathbf{N}$. We call f *compatible with subsumption* if $f(C) \leq f(C')$ whenever C subsumes C'. In other words, if f is a subsumption-compatible clause feature, then $f(C) \leq f(C')$ is a necessary condition for the subsumption of C' by C. Unless we specify a particular subsumption variant, we assume multiset subsumption.

We will define a number of clause features now, all of which are compatible with multiset subsumption, and many of which are compatible with other subsumption variants.

Let C be a clause. We denote the sub-multiset of positive literals in C by C^+, and similarly the sub-multiset of negative literals by C^-. Please note that both C^+ and C^- are clauses as well. $|C|$ is the number of literals in C. $|C|_f$ is the number of occurrences of the symbol f in C, e.g. $|p(a,b) \vee f(a,a) \not\simeq a|_a = 4$.

Let t be a term, and let f/n be a function symbol. We define $d_f(t)$ as follows:

$$d_f(t) = \begin{cases} 0 & \text{if } f \text{ does not occur in } t \\ \max\{1, d_f(t_1)+1, \ldots, d_f(t_n)+1\} & \text{if } t \equiv f(t_1, \ldots, t_n) \\ \max\{d_f(t_1)+1, \ldots, d_f(t_m)+1\} & \text{if } t \equiv g(t_1, \ldots, t_m), g/m \neq f, \\ & f \text{ occurs in } t \end{cases}$$

Intuitively, $d_f(t)$ is the depth of the deepest occurrence of f in t (or 0). The function is continued to atoms, literals and clauses as follows:

$$\begin{aligned} d_f(s \simeq t) &= \max\{d_f(s), d_f(t)\} \\ d_f(s \not\simeq t) &= d_f(s \simeq t) \\ d_f(l_1 \vee \ldots \vee l_k) &= \max\{d_f(l_1), \ldots, d_f(l_k)\} \end{aligned}$$

If we assume the standard representation of terms given above, the following theorems hold:

Theorem 1. *The feature functions defined by the following expressions are compatible with multiset subsumption, subsumption modulo AC, and equality subsumption:* $|C^+|$, $|C^-|$, $|C^+|_f$ *(for all* f*),* $|C^-|_f$ *(for all* f*).*

The argument is essentially always the same: instantiation can only add new symbols, and a superset (super-multiset) or superstructure always contains at least as many symbols as the subset or substructure.

Theorem 2. *The feature functions defined by the following expressions are compatible with multiset subsumption, set subsumption, and equality subsumption:* $d_f(C^+)$ *(for all* f*),* $d_f(C^-)$ *(for all* f*).*

The argument is similar: Instantiation can only introduce function symbols at new positions, never take them away at an existing depth.

If AC terms are represented in flattened form, symbol-counting features need to correct for the effects of this representation to be subsumption-compatible. On the other hand, if all terms are fully flattened, then the depth-based features also become compatible with AC-subsumption.

A final theorem allows us to combine different feature functions while maintaining compatibility with different subsumption types:

Theorem 3. *If any two feature functions* f_1*,* f_2 *are compatible with one of the listed subsumption types, then any linear combination of the two with non-negative coefficients is also compatible with that subsumption type. That is,* $f(C) = af_1(C) + bf_2(C)$ *with* $a, b \in \mathbb{R}^+$ *is also a compatible feature function.*

Many provers already use the criterion that a subsuming clause cannot have more function symbols that the subsumed one. In our notation, this can be described by the requirement that $\sum_{f \in F} |C|_f \leq \sum_{f \in F} |C'|_f$. This will, on average, already decide about half of all subsumption attempts. However, by looking at and combining more fine-grained criteria, we can do a lot better.

4.2 Clause Feature Vectors and Candidate Sets

Let π_n^i be the projection function for the ith element of a vector with n elements. A *clause feature vector function* is a function $F : \mathit{Clauses}(F, V) \to \mathbf{N}^n$. We call F subsumption-compatible (for a given subsumption type) if $\pi_n^i \circ F$ is a subsumption compatible feature for each $i \in \{1, \ldots, n\}$. In other words, a subsumption compatible feature vector function combines a number of subsumption compatible feature functions. We will now assume that F is a subsumption-compatible feature vector function. If $F(C) = v$, we call v the feature vector of C.

We define a partial ordering $\leq_s$ on vectors by $v \leq_s v'$ iff $\pi_n^i(v) \leq \pi_n^i(v')$ for all $i \in \{1, \ldots, n\}$. By definition of the feature vector, if C subsumes C', then $F(C) \leq_s F(C')$. This allows us to succinctly identify the candidate sets of clauses for forward subsumption and backward subsumption. Let C be a clause and P be a clause set. Then

$$\mathit{candFS}_F(P, C) = \{c \in P | F(c) \leq_s F(C)\}$$

is a superset of all clauses in P that subsume C and

$$\mathit{candBS}_F(P, C) = \{c \in P \mid F(C) \leq_s F(c)\}$$

is a superset of all clauses in P that are subsumed by C.

As our experiments show, if a reasonable number of clause features are used in the clause feature vector, these supersets are usually fairly small. Restricting subsumption attempts to members of these candidate sets reduces the number of attempts often by several orders of magnitude.

4.3 Index Data Structure

Whereas it is possible to store complete feature vectors with every clause in a set, this approach is rather inefficient in terms of memory consumption, and still requires the full comparison of all feature vectors. If, on the other hand, we compile feature vectors into a *trie*-like data structure, with all clauses sharing a vector stored at the corresponding leaf, large parts of the vectors are shared, and candidate sets can be computed much more efficiently.

Assume a (finite) set P of clauses with associated feature vectors $F(P)$ of length n. A *clause feature vector index* for P and F is a tree of uniform depth n (i.e., each path from the root to a leaf has length n). It can be recursively constructed as follows: If n is equal to 0, the tree consists of just a leaf node, which we associate with all clauses in P. Otherwise, let $D = \{\pi_n^1(F(C)) \mid C \in P\}$, let $P_i = \{C \mid \pi_n^1(F(C)) = i\}$ for $i \in D$ (the set of all clauses for which the first feature has a given value i), and let $F' = \langle \pi_n^2, \ldots, \pi_n^n \rangle \circ F$ (shortening the original feature vectors by the first element). Then the index consist of a root node with sub-trees T_i, such that each T_i is an index for P_i and F'. Inserting and deleting is linear in the number of features and independent of the number of elements in the index.

As an example, consider F defined by $F(C) = \langle |C^+|_a, |C^+|_f, |C^-|_b| \rangle$, the clauses $C1 = p(a) \vee p(f(a)), C2 = p(a) \vee \neg p(b), C3 = \neg p(a) \vee p(b), C4 = p(X) \vee p(f(f(b)))\}$, and the set of clauses $P = \{C1, C2, C3, C4\}$. The feature vectors are as follows: $F(C1) = \langle 2, 1, 0 \rangle$, $F(C2) = \langle 1, 0, 1 \rangle$, $F(C3) = \langle 0, 0, 0 \rangle$, $F(C4) = \langle 0, 2, 0 \rangle$. Figure 2 shows the resulting index.

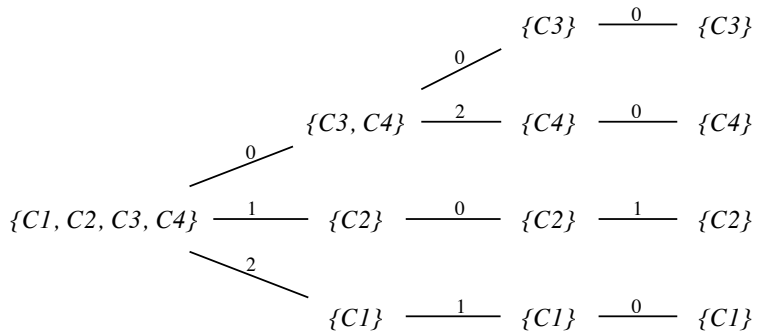

Fig. 2. Example of a Clause Feature Vector Index. For illustration purposes, each node is annotated with the clauses compatible with the features leading to that node. The actual implementation only stores clauses at the leaf nodes.

4.4 Forward Subsumption

For forward subsumption, we do not need to compute the full candidate set $candFS_F(P, C)$. Instead, we can just enumerate the elements and stop as soon as a subsuming clause is found. Assume a clause set P, a feature function F with feature vector length n, and an index I. We denote by $I[v]$ the sub-tree of I associated with value v. The clause to be subsumed is C. Figure 3a) shows the algorithm for indexed subsumption.

Note that it is trivial to return the subsuming clause (if any), instead of just a Boolean value. We traverse the sub-trees in order of increasing feature values, so that (statistically) smaller clauses with a higher chance of subsuming get tested first.

The subsumption test in the leaves of the tree is implemented by sequential search. In particular, finding the candidate sets and applying the actual subsumption test are clearly separated, i.e., it is trivially possible to use any subsumption concept as long as F is compatible with it.

4.5 Backward Subsumption

The algorithm for backward subsumption is quite similar, except that we traverse nodes with feature values greater than or equal to that of the subsuming clause, and that we cannot terminate the search early, since we have to find (and return) *all* subsumed clauses. We use the same conventions as above. Additionally, $mv(I)$ is the largest feature value associated with any sub-tree in I. Figure 3b) shows the algorithm.

4.6 Optimizing the Index Data Structure

Each leaf in the feature vector index corresponds to a given feature vector. If we ignore the internal structure of the trie, and the order of features in the vector, we can associate each leaf with an unordered set of tuples $(f, f(C))$ of individual feature functions and corresponding feature value. It is easy to see that any order of features in the feature vector will generate the same number of leaves, and that each leaf is either compatible with a given set of feature function/feature value tuples, or not. Thus, at least for a complete search as in the backward subsumption algorithm, we always have to visit the same number of leaves.

However, we can certainly minimize the internal number of nodes in the trie, and thus the total number of nodes. Consider for a simple example feature vectors with two features f_1, f_2, where f_1 yields the same value for all clauses from a set P, whereas f_2 perfectly separates the set into n individual clauses. If we test f_1 first, our tree has just one internal node (plus the root). Traversing all leaves touches $n + 2$ nodes (counting the root). If on the other hand we evaluate the more informative f_2 first, we will immediately split the tree into n internal nodes, each of which has just one leaf as the successor. Thus, to traverse all leaves we would touch $2n + 1$ nodes, or, for a reasonably sized n, nearly twice as many nodes.

(a) Forward subsumption

```
// Note that d is ranging over the length of F(C)
function search_subsuming(I, d, C)
  if I is a leaf node then
    if a clause in I subsumes C
      return true
    else
      return false
  else
    for i ∈ {0, ..., π_n^d(F(C))}
      if search_subsuming(I[i], d + 1, C)
        return true
  return false

function is_subsumed(I, C) // Return true if clause in I subsumes C
  return search_subsuming(I, 1, C)
```

(b) Backward subsumption

```
function search_subsumed(I, d, C)
  if I is a leaf node then
    return {C' ∈ I | C' subsumed by C}
  else
    res = {}
    for i ∈ {π_n^d(F(C)), ..., mv(I)}
      res = res ∪ search_subsumed(I[i], d + 1, C)
    return res

function find_subsumed(I, C) // Return clauses in I subsumed by C
  return search_subsumed(I, 1, C)
```

Fig. 3. Forward and backward subsumption with feature vector indexing

This example easily generalizes to longer vectors. In general, we want the least informative features first in a feature vector, so that as many initial paths as possible can be shared. This is somewhat surprising, since for most exclusion tests it is desirable to have the most informative features first, so that impossible candidates are excluded early. Of course, if we have totally uninformative features, we can just as well drop them completely, thus shrinking the tree depth.

Unless we want to deal with the complexity and computational cost of dynamically adapting the index, we have to determine the feature vector function before we start building the index, i.e., in practice before the proof search starts. We can estimate the informativeness of a given feature by looking at the distribution of its values in the initial clause set, and assume that this is typical for the later clauses.

For best results, we could view application of a feature function to a clause as a probability experiment and the results on the initial clause set as a sample. We

could then sort features by increasing estimated entropy[2] [23] or even conditional entropy. However, we decided to use a much simpler estimator first, namely the range of the feature value over the initial clause set. We have implemented three different mappings: *Direct mapping*, where the place of a feature in the vector is determined by the internal representation of function symbols used by the system (i.e. the first function symbol in the signature is responsible for the first 2 or 4 features, the second for the next, and so on), *permuted*, where features are sorted by feature value range, and *optimized permuted*, where additionally features with no estimated usefulness (i.e., features which evaluate to the same value for all initial clauses) are dropped.

Our experimental results show that both permuted and optimized permuted feature vectors perform much better than direct mapped ones, with optimized permuted ones being best if we allow only a few features, whereas plain permuted ones gain if we allow more features. Generally, we can decrease the number of nodes in an index by about 50% using permuted feature vectors. We explain this behaviour by noting that the degree of informativeness is generally estimated correctly, but the prediction whether a feature will be useful at all is less precise. We have especially observed the situation where only a single negative literal occurs in the initial clause set (e.g. all unit-equational proof problems with a single goal), and hence all features restricted to negative literals have an initial range of zero, although a large and varied set of negative literals is generated during the proof search.

5 Implementation Notes

We have implemented clause feature vector indexing in our prover E, using essentially simple versions of standard trie algorithms for inserting and deleting feature vectors (and hence clauses), and the algorithms described in section 4.4 and 4.5 for forward and backward subsumption. In E's implementation of the given-clause algorithm, unprocessed clauses are passive, i.e. they don't impose a computational burden once normalized. Hence we are using subsumption only between the set of processed clauses P and the given clause g and vice versa, not on the full set of unprocessed clauses. However, we have also implemented contextual literal cutting using the index. It can be optionally applied either to the newly generated clauses during simplification (using clauses from P for cutting) or between g and P, in both directions.

Feature vector indexing is used for forward and backward non-unit multiset subsumption, all versions of contextual literal cutting, (unit) equality backward subsumption, and backward simplify-reflect (equational unit cutting, see [17]) inferences. Forward equality subsumption and forward simplify-reflect have been implemented using discrimination tree indexing (on maximal terms in the unit

[2] The *entropy* of a probability experiment is the expected information gain from it, or, in other words, the expected cost of predicting the outcome. In our case, a feature with higher entropy splits the clause set into more (or more evenly distributed) parts. See e.g. [16] or, for a more comprehensive view, [5].

clause used) since early versions of E. Paramodulation and and backwards rewriting are implemented using fingerprint indexing[21].

The existing multiset subsumption code, used both for conventional subsumption and to check indexed candidates for actual subsumption, already is fairly optimized. It uses a number of simple criteria to quickly determine unsuitable candidates, including tests based on literal- and symbol count, and trying to match individual literals in the potential subsumer onto literals in the potentially subsumed clause. Only if all these tests succeed do we start the recursive permutation of terms and literals to find a common match.

The feature vector index is implemented in a fairly straightforward way, using a recursive data structure. Note that all our features in practice yield small integers. Originally, we had implemented the mapping from a feature value to the associated sub-tree via a dynamic array. However, the current implementation uses the *IntMap* data structure, a self-optimizing data structure that reorganizes itself as either a dynamic array or a *splay tree* [24], depending on the fraction of elements used.

Clauses in a leaf node are stored in a simple set data structure (which is implemented throughout E as a splay tree using pointers as keys). Empty subtrees are deleted eagerly.

It may be interesting to note that the first (and working) version of the indexing scheme took only about three (part-time) days to implement and integrate from scratch. It took approximately 7 more days to arrive at the current (production-quality) version that allows for a large number of different clause feature vector functions to be used and applies the index to many different operations, and about half a day to change the implementation to IntMaps. Compared to other indexing techniques, feature vector indexing seems to be easy to implement and easy to integrate into existing systems.

6 Experimental Results

We used all untyped first-order problems from TPTP 5.2.0 for the experimental evaluation. There are 15356 problems, about evenly split between 7712 clause normal form problems, and 7674 full first-order problems. The problems were not modified in any way. Tests were run on the University of Miami *Pegasus cluster*, under the Linux 2.6.18-164.el5 SMP Kernel in 64 bit mode. Each node of the cluster is equipped with 8 Intel Xeon cores, running at 2.6 GHz, and 16 GB of RAM.[3] Test runs were done with a CPU time limit of 300 seconds per job, a memory limit of 512 MB per job, and with 8 jobs scheduled per node. Detailed results of these and additional test runs, including an archive of the source package of the prover version, are available at `http://www.eprover.eu/E-eu/FVIndexing.html`.

[3] See [13]. Jobs were submitted on the "Small" queue, which schedules only to Intel Xeon systems.

6.1 Prover Instrumentation and Configuration

In first-order theorem proving, even small changes to the order of inferences can influence the course of the proof search significantly. Indexing affects both the order in which clauses are generated and the internal memory layout of the process. There is no guarantee that the system performs the same search with different subsumption implementations. To minimize this effect, E has been modified with an option that imposes a total ordering on newly generated clauses in each iteration of the main loop.[4] For the detailed quantitative analysis of the run times, we use only cases where the prover performs the same number of iterations of the main loop, and had the same number of processed and unprocessed clauses at termination time. These three indicators give a high likelihood that the proof searches followed very similar lines for the indexed and non-indexed case.

To gain more insights into the time spent in various parts of the prover, we have added a generic profiling mechanism to E by instrumenting the source code. The system can maintain an arbitrary number of profiling points. The system computes (at microsecond resolution) the difference between the time a profiled code segment is entered and left. The times for each profiled segment are summed over the life time of the process. Here, we use performance counters measuring the time spent for the whole proof search, non-unit clause-clause subsumption, set subsumption (i.e. forward- and backward subsumption involving a clause and a set of clauses), index maintenance, and feature vector computation.

As for all profiling solutions using portable high-level timing interfaces as defined e.g. in POSIX, the times measured are only statistically valid. Many functions have run times much shorter than the microsecond resolution of the UNIX system clock. However, over sufficiently many calls, the average values become increasingly more reliable. The same holds for the small variations invariably caused by small differences in process scheduling and even instruction scheduling in modern, pipelined multiple-issue processors. We performed tests on thousands of problems, with many thousands of calls to small profiled functions for each problem. Thus, noise effects largely average out.

[4] The exact options given to the prover were `--definitional-cnf=24 --tstp-in --split-clauses=4 --split-reuse-defs --simul-paramod --forward-context-sr-aggressive --backward-context-sr --destructive-er-aggressive --destructive-er --prefer-initial-clauses -tKBO6 -winvfreqrank -c1 -Ginvfreqconjmax -F1 -s --delete-bad-limit=512000000 -WSelectMaxLComplexAvoidPosPred -H'(4*RelevanceLevelWeight2(SimulateSOS,0,2,1,2,100,100,100,400,1.5, 1.5,1), 3*ConjectureGeneralSymbolWeight(PreferNonGoals,200,100,200,50, 50,1,100,1.5,1.5,1),1*Clauseweight(PreferProcessed,1,1,1), 1*FIFOWeight(PreferProcessed))' --detsort-new --fvindex-maxfeatures=<XXX> --fvindex-featuretypes=<YYY> --subsumption-indexing=<ZZZ>`.

Tests were done with E 1.4-011 and E 1.4-012 (for the **BI** strategy), versions of E 1.4 which has been instrumented for profiling feature vector operations.

6.2 Feature Selection

The indexed version of the prover evaluated here uses a maximum feature vector length of around 150 elements. Features used in the vectors are $|C^+|$, $|C^-|$, $|C^+|_f$, $|C^-|_f$, $d_f(C^+)$ and $d_f(C^-)$ (for some function symbols f). Some feature vector functions use a "catch-all" feature that summarizes the value of a given feature type for all function symbols not represented by an individual feature.

The vector might be slightly shorter than 150 elements if only a few symbols occur in the input formula. The lengths can vary slightly because the feature vector functions are systematically generated from a set of function symbols, with either two or four features generated per symbol.

We used 4 different feature selection schemes:

- **AC** uses the AC-compatible features, $|C^+|$, $|C^-|$, $|C^+|_f$, $|C^-|_f$, with a catch-all feature implemented individually for positive and negative literals. The catch-all sums occurrences of all otherwise uncounted symbols.
- **DF** uses the set-subsumption compatible features, $d_f(C^+)$ and $d_f(C^-)$, with a catch-all feature implemented individually for positive and negative literals. The catch-all feature represents the maximum depths of any otherwise unrepresented function symbols.
- **AL** uses all the features used by **AC** or **DF**, including the 4 catch-all features.
- **BI** was suggested by Bill McCune[5] and used by him in Prover9. McCune noted that predicate symbols always occur at depth 1, and hence $d_f(C)$-type features add no information for predicate symbols. The following features are used:
  - $|C^+|$, $|C^-|$
  - $|C^+|_f$, $|C^-|_f$ for all symbols f in the signature
  - $d_f(C^+)$ and $d_f(C^-)$ for all proper function symbols (as opposed to predicate symbols) in the signature

  BI does not use a catch-all feature.

6.3 Feature Vector Optimization

We have implemented the two optimizations described in section 4.6. So each of the 4 different feature selection schemes can be combined with 3 different orderings of features in the vectors:

- **DRT** uses direct mapping: Features are ordered by some arbitrary order naturally generated by the implementation. In practice, different feature types are grouped together, and sorted by the index of the function symbol in the symbol table.
- **PRM** uses permuted feature vectors, with features ordered by their span on the axiom set.
- **OPT** uses the same order as PRM, but completely removes features with span 0 on the axiom from the vector.

[5] Personal communication.

We know from previous experiments that **PRM** is generally the strongest of these three [18] and hence performed most tests with this schema.

6.4 Basic Performance

Table 1 shows the performance of the prover over the whole first-order part of TPTP 5.2.0 with conventional subsumption and different versions of feature-vector indexing for subsumption and contextual literal cutting. Feature vector functions are named in the obvious way. **NONE** represents the prover using conventional sequential subsumption. The table is sorted by number of solutions.

Table 1. Basic performance different subsumption methods

Index	**Number of solutions**
NONE	8,471
DRT-DF	8,721
PRM-DF	8,769
DRT-AL	8,832
PRM-BI	8,858
DRT-AC	8,881
PRM-AL	8,897
OPT-AC	8,914
PRM-AC	8,922

The weakest feature vector indexing (DRT-DF) can solve 250 problems more than the system with conventional subsumption. The best feature vector indexing (PRM-AC) can solve an extra 201 problems. We can see that in general, the use of the **AC** features is best, followed by **AL** and finally **DF**. Also, the previous result that **PRM** (marginally) outperforms **OPT** which outperforms **DRT** is confirmed.

It is somewhat surprising that **PRM-AL** outperforms **PRM-BI**. Since **BI** should capture the same information as **AL** in the regular features, the most likely reason for this is the lack of the catch-all feature in **BI**.

In all pairings of results, the larger set of solutions is very nearly a strict superset of a smaller set. As an example, there is only one problem solved by **NONE** that is not solved by **PRM-AC**, while there are 441 problems solved by **PRM-AC** but not by **NONE**.

Figure 4 is a scatter plot that illustrates the superiority of indexed (**PRM-AC**) over conventional (**NONE**) subsumption. Each cross represents the performance on a single problem. Marks below the diagonal show better performance for the indexed versions. Marks on the right border represent problems where the system with conventional subsumption timed out, marks on the upper border to problems where the indexed system timed out. It is obvious that, with very few exceptions, the indexed version is much superior in performance.

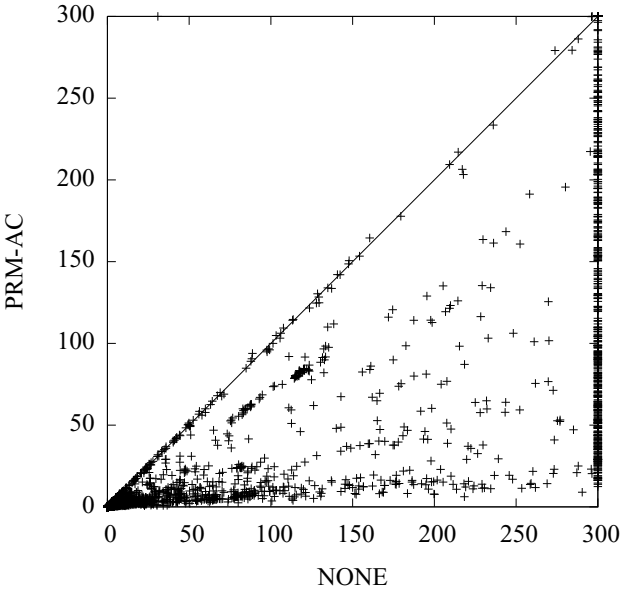

Fig. 4. Run times (in seconds) of **PRM-AC** over **NONE**

6.5 Profiling and Time Behavior

In the following we consider only problems solved by all subsumption strategies under consideration. Moreover, we consider only problems where the system has, with high likelihood, performed the same proof search. Filtering for these criteria, 8386 problems remain.

Table 2 shows where the indexed and non-indexed versions of the prover spend time on this set of problems. The order of strategies is the same as in Table 1. The columns contain the name of the index, total run-time on all 8386 problems, time spent in the computation of feature vectors, time spent on index maintenance (insert and delete), time spent directly in the clause-clause subsumption code, and time spent in forward- and backward subsumption, including the overhead of iterating through the sets or the index.

The first row shows that subsumption (including contextual literal cutting) has a drastic influence on total run time. For the non-indexed version, about 65% of total run time is spent in forward- and backwards subsumption operations, about 40% in individual non-unit clause-clause subsumption. Feature vector indexing drastically changes this, cutting the time for subsumption itself by a factor of more than 22 for **PRM-AC**. If we consider forward- and backward subsumption (including the sequential or indexed iteration over candidate clauses), we still see an improvement by a factor of more than 10.

Comparing the different indices, we can see that the run times in Table 2 are roughly, but not strictly, decreasing with the number of solutions in Table 1, indicating that the major reason for the improved performance is indeed the

Table 2. Time (in seconds) spend in various code segments

Name	Total time	FVs	Index	C/C subsumption	F&B subsumption
NONE	69,523.375	N/A	N/A	28,379.300	46,160.810
DRT-DF	34,059.599	81.250	51.460	6,848.240	17,548.180
PRM-DF	30,813.326	102.440	30.680	6,845.000	14,675.360
DRT-AL	31,301.364	75.820	86.410	2,051.890	13,322.010
PRM-BI	23,909.235	106.180	27.530	3,065.410	7,792.850
DRT-AC	25,360.085	81.490	72.590	1,217.920	9,017.900
PRM-AL	21,758.335	83.710	23.030	2,104.840	5,624.780
OPT-AC	20,963.105	100.580	35.930	1,423.630	4,550.500
PRM-AC	20,663.982	103.040	38.440	1,283.890	4,418.090

speed-up of subsumption. We can also see that index maintenance and feature vector computation are comparatively negligible. The cost for computing permuted vectors is slightly higher than for direct vectors, but this is more than compensated for by the smaller cost for index maintenance.

Table 3. Number of (non-unit) clause/clause subsumption attempts

Name	Non-unit subsumption calls	Recursive subsumption calls
NONE	120,934,644,536	14,142,932,647
DRT-DF	24,954,121,073	4,774,601,640
PRM-DF	25,897,859,434	4,851,875,239
DRT-AL	4,545,278,003	2,537,935,468
PRM-BI	8,832,113,671	4,189,693,241
DRT-AC	2,840,076,064	1,691,440,719
PRM-AL	5,795,826,572	2,936,641,523
OPT-AC	3,550,612,558	1,933,200,370
PRM-AC	3,254,649,092	1,846,439,309

The most dramatic improvement is in the time spent in actual subsumption code. The reason for this becomes obvious in Table 3. Column 2 shows the number of (non-unit) clause-clause subsumption attempts, column 3 the number of those that cannot be rejected by non-recursive tests and actually go into the exponential matching algorithm. The number of subsumption attempts drops by 97% of subsumption attempts when comparing **NONE** and **PRM-AC**. Even more dramatically, the number of recursive calls drops by nearly two full orders of magnitude. The reduction in subsumption attempts is also illustrated in the scatter plot of Figure 5.

Note the double logarithmic scale necessary to adequately display the large variation in numbers. The conventional version needs, over all problems, about 40 times more calls than the indexed version. For individual problems the improvement factor varies from 1 (for some trivial problems) to approximately 7500 e.g. for `PUZ080+2`, where the number of subsumption attempts drops from 69504083 with **NONE** to only 9284 with **PRM-AC**.

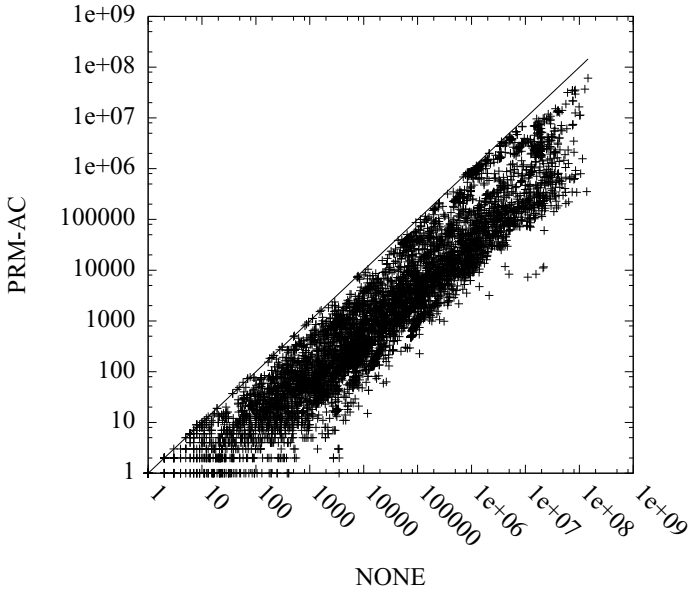

Fig. 5. Non-unit subsumption attempts of **PRM-AC** over **NONE**

6.6 Performance in Automatic Mode

The previous section listed results for a consistent search heuristic. However, much of the power of modern provers comes from an automatic selection of various search parameters in an auto-mode. We have also run E in its automatic mode with both **NONE**, **DRT-AC** and **PRM-AC**. The results are presented in Table 4.

Table 4. Performance in automatic mode

Index	**Number of solutions**
NONE	9,241
DRT-AC	9,782
PRM-AC	9,823

With E in automatic mode, the prover gains more than 600 solutions if **PRM-AC** is used.

7 Future Work

While we are very satisfied with the performance of our current implementation, there are a number of open research opportunities.

On the one side, we can still improve the technique itself. Mark Stickel has pointed out to us that additional useful features can be constructed not only from the greatest occurrence depth of a function symbol, but also from the smallest occurrence depth (if properly transformed).[6]

Recent theorem prover applications yield problems with very large signatures. In this case, the number of possible features is also very large. However, as Table 2 shows, for all the indexed strategies, time spent traversing the index is already longer than actual subsumption time. Several approaches to dealing with this are promising. If signatures are large, many feature values will be 0, since most symbols will not occur in any single clause. We can represent these default value feature implicitly in the index by annotating each node not only with the feature value, but also with the feature itself (usually represented as the numerical index describing its position in the vector). This would result in a fairly compact index even for large vectors, at the cost of slightly more complex algorithms. This is the version implemented by Korovin in iProver[7]. However, to our knowledge no systematic evaluation has taken place so far.

Secondly, we can make use of Theorem 3 to define more complex features that represent properties of the clause with respect to sets of function symbols. This requires only changes in the individual feature functions, not the other algorithms. An prototypical implementation of this in E already shows promising results, cutting total time for subsumption by another factor of two.

On a different track, while we developed feature vector indexing with the aim of finding a good solution for clause subsumption indexing, it can also be used to index terms for the retrieval relations *matches* and *is matched by*. While there are very good techniques for term indexing, most of them cannot handle AC symbols well. Feature vector indexing, on the other hand, handles associativity and commutativity easily. This might make an attractive choice for implementing forward and backward rewriting modulo AC.

8 Conclusion

Feature vector indexing has proved to be a simple, but effective answer to the subsumption problem for saturating first-order theorem provers. In our experiments, it is able to reduce the number of subsumption tests by, on average, about 97% compared to a naive sequential implementation, and thus reduces cost of subsumption in our prover to a level that makes it hard to measure using standard UNIX profiling tools. In addition to the direct benefit, this gain in efficiency has enabled us to implement otherwise relatively expensive subsumption-based simplification techniques (like contextual literal cutting), further improving overall performance of our prover.

[6] Personal communication.
[7] Personal communication.

Feature vector indexing has been successful not only in E and Prover9, but has also been implemented in the description logic reasoner KAON-2 [12], in the SMT-solver Z3 [1], and in the instance-generation based first-order prover iProver [7].

Afterword

A preliminary version of this paper was presented in 2004 at the *Workshop on Empirically Successful First-Order Reasoning (ESFOR)*, associated with IJCAR in Cork. This updated version of the paper describes some of the evolution of the implementation and adds a more detailed analysis of the costs and benefits of feature vector indexing.

ESFOR was the first of a series of workshops [20] with a renewed focus on practically useful systems and applications. I was one of the organizers, and we considered ourselves very lucky when Bill McCune joined the program committee. As it turned out, Bill was also one of the anonymous reviewers of my paper. Half a year later, he send me an email stating:

> A few weeks ago I tried your feature vector indexing in one of my provers. Powerful, easy to implement, elegant. The best new idea in indexing I've seen in many years.

It was one of the proudest moments of my life.

This informal style of cooperation was typical for Bill. He was always generous with credit, help, and advice, even when I first met him as a brand-new Ph.D. student in 1996. It is hard to accept that he is no longer around.

Acknowledgements. I thank the University of Miami's Center for Computational Science HPC team for making their cluster available for the extended experimental evaluation.

References

1. de Moura, L., Bjørner, N.S.: Engineering DPLL(T) + Saturation. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 475–490. Springer, Heidelberg (2008)
2. Denzinger, J., Kronenburg, M., Schulz, S.: DISCOUNT: A Distributed and Learning Equational Prover. Journal of Automated Reasoning 18(2), 189–198 (1997); Special Issue on the CADE 13 ATP System Competition
3. Graf, P.: Term Indexing. LNCS, vol. 1053. Springer, Heidelberg (1996)
4. Graf, P., Fehrer, D.: Term Indexing. In: Bibel, W., Schmitt, P.H. (eds.) Automated Deduction — A Basis for Applications. Applied Logic Series, vol. 9(2), ch. 5, pp. 125–147. Kluwer Academic Publishers (1998)
5. Jaynes, E.T.: Probability Theory: The Logic of Science. Cambridge University Press (2003)

6. Kapur, D., Narendran, P.: NP-Completeness of the Set Unification and Matching Problems. In: Siekmann, J.H. (ed.) CADE 1986. LNCS, vol. 230, pp. 489–495. Springer, Heidelberg (1986)
7. Korovin, K.: iProver – An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
8. Löchner, B., Hillenbrand, T.: A Phytography of Waldmeister. Journal of AI Communications 15(2/3), 127–133 (2002)
9. McCune, W.W.: EQP 0.9 Users' Guide (1999), `http://www.cs.unm.edu/~mccune/eqp/Manual.txt` (accessed December 11, 2012)
10. McCune, W.W.: Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. Journal of Automated Reasoning 9(2), 147–167 (1992)
11. McCune, W.W.: 33 Basic Test Problems: A Practical Evaluation of Some Paramodulation Strategies. In: Veroff, R. (ed.) Automated Reasoning and its Applications: Essays in Honor of Larry Wos, ch. 5, pp. 71–114. MIT Press (1997)
12. Motik, B., Sattler, U.: A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 227–241. Springer, Heidelberg (2006)
13. University of Miami Center for Computational Science. Pegasus - Introduction (2007-2011), `http://ccs.miami.edu/?page_id=3749` (accessed December 09, 2012)
14. Riazanov, A., Voronkov, A.: The Design and Implementation of VAMPIRE. Journal of AI Communications 15(2/3), 91–110 (2002)
15. Riazanov, A., Voronkov, A.: Efficient Instance Retrieval with Standard and Relational Path Indexing. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 380–396. Springer, Heidelberg (2003)
16. Schulz, S.: Information-Based Selection of Abstraction Levels. In: Russel, I., Kolen, J. (eds.) Proc. of the 14th FLAIRS, Key West, pp. 402–406. AAAI Press (2001)
17. Schulz, S.: E – A Brainiac Theorem Prover. Journal of AI Communications 15(2/3), 111–126 (2002)
18. Schulz, S.: Simple and Efficient Clause Subsumption with Feature Vector Indexing. In: Sutcliffe, G., Schulz, S., Tammet, T. (eds.) Proc. of the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving, Cork, Ireland (2004)
19. Schulz, S.: System Description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 223–228. Springer, Heidelberg (2004)
20. Schulz, S.: Empirically Sucessful Topics in Automated Deduction (2008), `http://www.eprover.org/EVENTS/es_series.html`
21. Schulz, S.: Fingerprint Indexing for Paramodulation and Rewriting. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 477–483. Springer, Heidelberg (2012)
22. Sekar, R., Ramakrishnan, I.V., Voronkov, A.: Term Indexing. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. II, ch. 26, pp. 1853–1961. Elsevier Science and MIT Press (2001)
23. Shannon, C.E., Weaver, W.: The Mathematical Theory of Communication. University of Illinois Press (1949)
24. Sleator, D.D., Tarjan, R.E.: Self-Adjusting Binary Search Trees. Journal of the ACM 32(3), 652–686 (1985)
25. Stickel, M.E.: SNARK - SRI's New Automated Reasoning Kit (2008), `http://www.ai.sri.com/~stickel/snark.html` (accessed October 04, 2009)

26. Tammet, T.: Towards Efficient Subsumption. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS (LNAI), vol. 1421, pp. 427–441. Springer, Heidelberg (1998)
27. Voronkov, A.: The Anatomy of Vampire: Implementing Bottom-Up Procedures with Code Trees. Journal of Automated Reasoning 15(2), 238–265 (1995)
28. Weidenbach, C.: SPASS: Combining Superposition, Sorts and Splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. II, ch. 27, pp. 1965–2013. Elsevier Science and MIT Press (2001)