Simple and Efficient Clause Subsumption with Feature Vector Indexing

Stephan Schulz [1]

ITC/irst, Trento, Italy
and
Technische Universität München, Germany

Abstract

We describe *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.

Key words: automated theorem proving, saturation, subsumption, indexing

1 Introduction

First-order theorem proving is one of the core areas of automated deduction. In this field, saturating theorem provers have, in the last few years, developed a significant lead compared to systems based on other paradigms, such as top-down reasoning or instance-based methods.

There are a number of reasons for this. At least one of these reasons 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

[1] Email: schulz@eprover.org

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

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 [5]. 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 [12,13], which, 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*, we want to know if *any* clause from a set subsumes a given clause. In *backward subsumption*, we want to find all clauses in a set that are subsumed by a given clause.

We can use this observation to speed up subsumption, by using *indexing techniques* that only return 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 still has to be verified.

Term indexing techniques have been used in theorem provers for some time now (see [7] for first implementations in Otter or [2,3,14] 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 only used to retrieve subsumption candidates, i.e., to implement non-perfect clause indexing (see e.g. [17]). 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 [10].

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 a more polished and configurable way, for E 0.81, with excellent results.

In this paper, we will, after some initial definitions, describe the new technique. We will also discuss how it has been integrated into E, and how it also serves to speed up *contextual literal cutting*, a subsumption-based simplifica-

tion 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 use 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, $\mathit{Term}(F, V)$, is defined as the smallest set fulfilling the following conditions:

(i) $X \in \mathit{Term}(F, V)$ for all $X \in V$

(ii) $f|_n \in F$, $s_1, \ldots, s_n \in \mathit{Term}(F, V)$ implies $f(s_1, \ldots, s_n) \in \mathit{Term}(F, V)$

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

An (equational) *atom* [2] 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 $s \dot{\simeq} t$, 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 $\square$, and the set of all clauses as $\mathit{Clauses}(F, V)$. A *formula* in clause normal form is a multiset of clauses, interpreted as a conjunction.

A *substitution* is a mapping $\sigma : V \to \mathit{Term}(F, V)$ with the property that $\mathit{Dom}(\sigma) = \{X \in V \mid \sigma(X) \neq X\}$ 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.

[2] 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 $p(a) \simeq \$true$.

3 Subsumption

If we consider a (multi-)set of clauses, that is, a formula in clause normal form, 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:

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

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

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). The first order clause subsumption problem is well-known to be NP-complete. However, there are a number of implementation techniques that can usually avoid the worst case, so that in practice individual subsumption attempts can be completed in acceptable time.

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 clauses with a suitable ordering.

However, whereas 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. We will shortly discuss some of them.

The definition of *set subsumption* is identical to that of multiset subsumption, except in that clauses are viewed as sets of literals (i.e., no multiple occurrences of the same literal are allowed). 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 be used in preprocessing or by provers not based on saturation. 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 $p(f(b, a)) \vee q(a)$. We are not aware of any system currently using subsumption modulo AC, however, it is generally believed to be useful for reasoning modulo AC.

Equality subsumption allows an equational unit clause to potentially subsume another clause with an equational literal implied by it. It can be described by the following simplification rule:

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

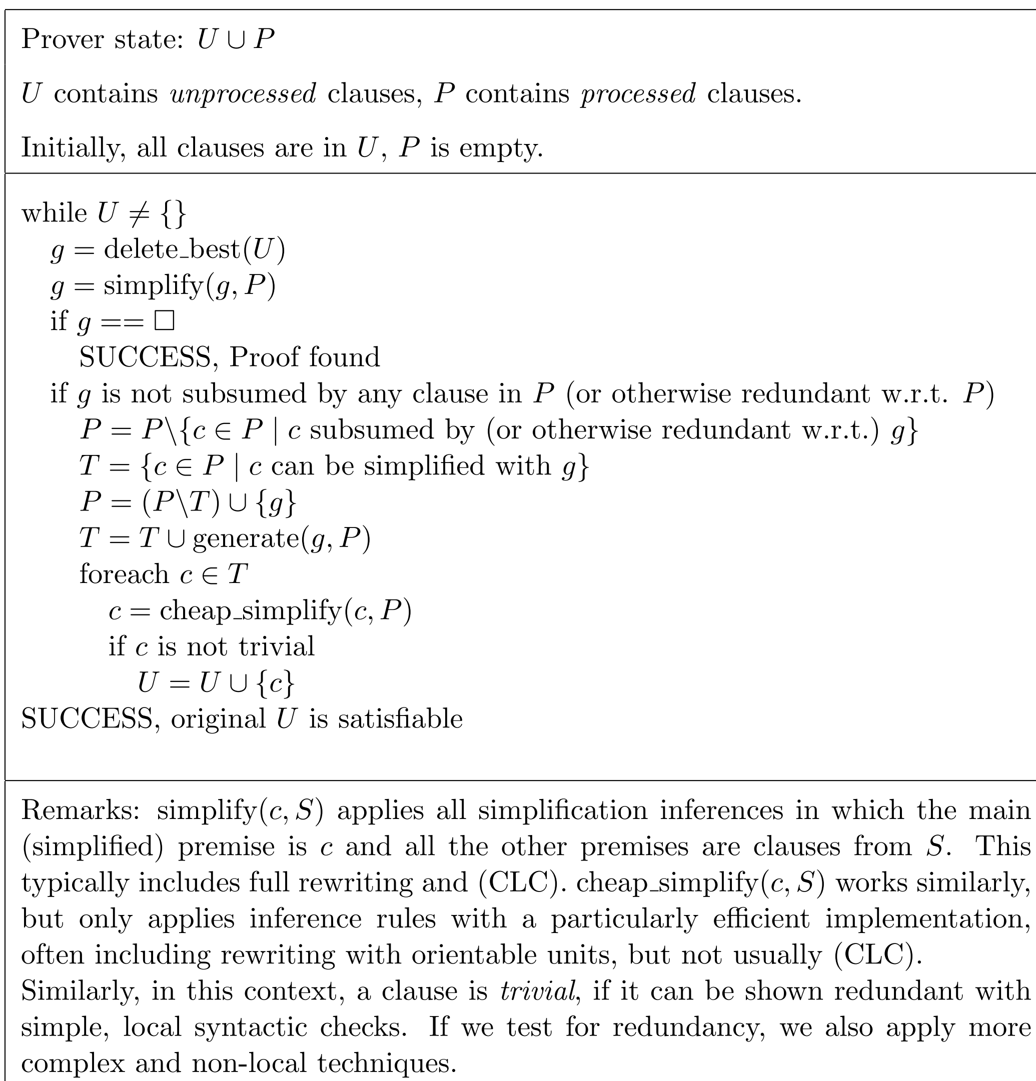
It is typically only applied if $s \simeq t$ cannot be used for rewriting. This rule is implemented by E and a number of other provers, including at least the completion-based systems Waldmeister [6] and DISCOUNT [1].

Finally, a simplification rule that has been popularized by implementation in SPASS [19] and Vampire [9], and is sometimes called *subsumption resolution*, 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*:

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

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

It can be implemented via a standard 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



Prover state: $U \cup P$

U contains *unprocessed* clauses, P contains *processed* clauses.

Initially, all clauses are in U, P is empty.

```
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: 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 not usually (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

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

Figure 1 shows a sketch of the main proof procedure of our prover E. It is representative of a modern high-performance prover using a variant of the *DISCOUNT loop* proof procedure (in which unprocessed clauses are *passive*, i.e., not even used as side premises for simplification). The alternative *Otter loop* primarily differs in that simplification and subsumption are also performed between clauses in U and using clauses from U as side premises for simplification of all clauses.

Please observe that standard 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 is e.g. used in early versions of SPASS [19], 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 [17,18,7] 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 standard 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:

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

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:

$$\begin{aligned} d_f(s \simeq t) &= \max\{d_f(s), d_f(t)\} \\ d_f(s \dot{\simeq} t) &= d_f(s \simeq t) \\ d_f(l_1 \vee \ldots \vee l_k) &= \max\{d_f(l_1), \ldots, d_f(l_k)\} \end{aligned}$$

The feature functions defined by the following expressions are compatible with standard 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.

The feature functions defined by the following expressions are compatible with standard 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 any two feature functions f_1, f_2 are compatible with a certain subsumption type, 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 $a, b \geq 0$ 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 $F : \mathit{Clauses}(F, V) \rightarrow \mathbf{N}^n$. 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 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- and backward subsumption. Let C be a clause and P be a clause set. Then

$$candFS_F(P, C) = \{c \in P | F(c) \leq_s F(C)\}$$

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

$$candBS_F(P, C) = \{c \in P \mid F(C) \leq_s F(c)\}$$

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 a suitable candidate set 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 $P_i = \{C \mid \pi_n^1(F(C)) = i \mid i \in D\}$ (the set of all clause 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 successors 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.

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

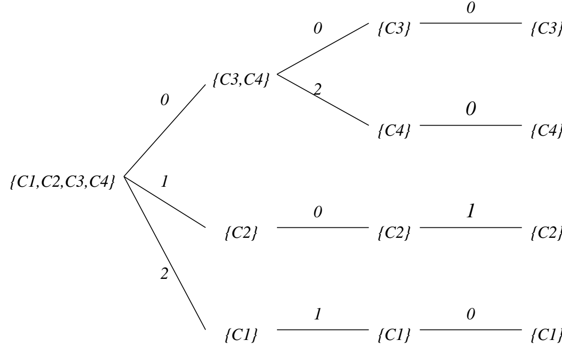


Fig. 2. Example of Clause Feature Vector Index

with feature vector length n, and an index I. We denote by $I[v]$ the subtree 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 subtrees 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 subtree 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

(a) Forward subsumption

```
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

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.

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 exclu-

sion 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.

Unfortunately, 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 only 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[3] [15] 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, *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 off.

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 that 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. We are using subsumption only between the set of processed clauses P and the given clause g and vice

[3] 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. [11] or, for a more comprehensive view, [4].

versa, but 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 [12]) inferences. Forward equality subsumption and forward simplify-reflect have been implemented using discrimination tree indexing (on maximal terms in the unit clause used) since early versions of E.

Our standard 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 individual literal matching. 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. Hence we have implemented the mapping from a feature value to the subtree as a (dynamic) array. The only special case we support is the case that a node has exactly one successor. In this case we do not use an array, but just store the feature value and a pointer to the successor, to avoid the memory overhead of the array.

Clauses in a leaf node are stored in a simple set data structure (which is implemented throughout E as a *splay tree* [16] 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. 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 5180 clause normal form problems from TPTP 2.5.1, without equality axioms, but otherwise unchanged. All test runs were performed on a cluster of 300MHz SUN Ultra-60 workstations with a time limit of 300 seconds (or equivalent configurations). The memory limit was 192 MB.

The indexed version of the prover uses a maximum feature vector length of 75. Features used in the vector are $|C^+|$, $|C^-|$, $|C^+|_f$ and $|C^-|_f$ (for some function symbols f). The vector might be shorter than 75 elements if only a

few symbols occur in the input formula. The results here were obtained with (plain) permuted feature vectors; features are ordered by increasing value range in the input clause set.

You can download detailed results of these and additional test runs at http://www.eprover.org/feature_vector_indexing.html.

6.1 Results with Aggressive Contextual Literal Cutting

Most of the strongest search strategies we found so far apply contextual unit cutting only to the given clause. To measure the effect of indexing for the worst-case scenario, we ran the system with a strong standard strategy, but with contextual literal cutting applied even to passive clauses, so that subsumption attempts are maximized. In this case, the version without feature vector indexing was able to solve 2671 problems within the time limit, whereas the prover with indexing solved 2717 problems (a strict superset). On the common subset, the indexed version needed 19857 s, whereas the plain system used 32140 s, for a speed-up of nearly 40% for the whole prover. For several harder examples, the indexed version ran as much as 5 times faster than the conventional prover.

Figure 4 shows the scatter plot of times for the conventional over the indexed version of the prover. Very few examples perform worse with indexing, and the drop is usually not very significant. A number of examples cluster around equal performance, and the majority shows moderate to large speed-ups.

We manually reran some of the few cases where the indexed version of the system was significantly slower than the non-indexed version on different hardware (Generic Intel Pentium-III/Pentium-4 PCs with GNU/Linux and a PowerPC G4 notebook with MacOS-X). In no case could we reproduce the slow-down (although it is reproducible on SUN hardware). Thus, we currently believe that it is caused by some unfortunate interplay between features of the SPARC architecture and our straight-forward recursive implementation. In particular, we suspect the large register window spills caused by deep recursions on SPARC.

Figure 5 shows how many direct clause-clause subsumption calls have been used by the conventional and the indexed version for the problems solved by both. Note the double logarithmic scale necessary to adequately display the large variation in numbers. The conventional version needs, over all problems, about 30 times more calls than the indexed version. For individual problems, the improvement factor varies from 1 (for some trivial problems) to approximately 7500 (for the problem `SYN738-1`, where the number of calls dropped from 22 552 to 3). The largest number of subsumption calls was observed for `SYN711-1` with 444 793 509. For this problem, indexing reduced the number of calls by a factor of nearly 200 to 2 229 754, and the run time from 235s to 79s.

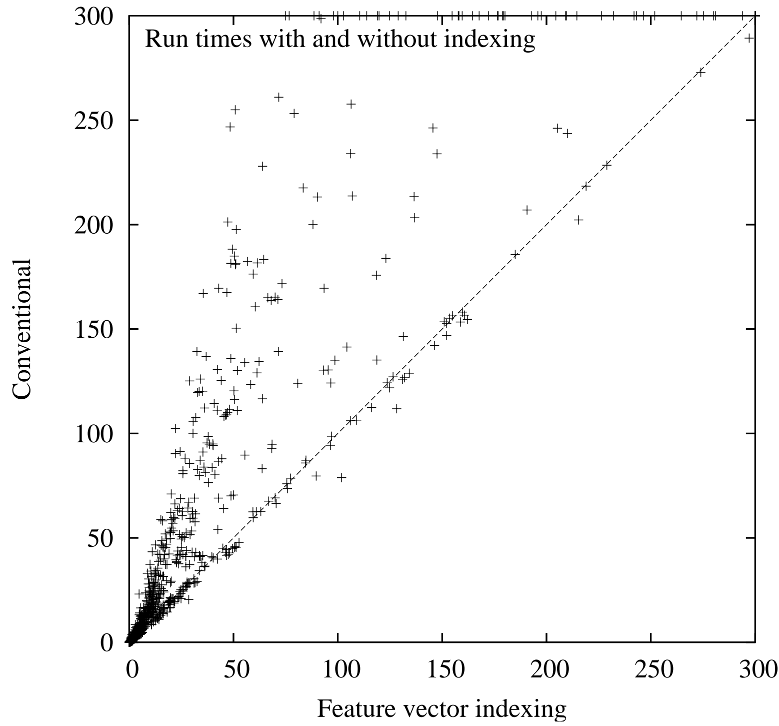


Fig. 4. Run times of indexed versus conventional implementation for aggressive (CLC)

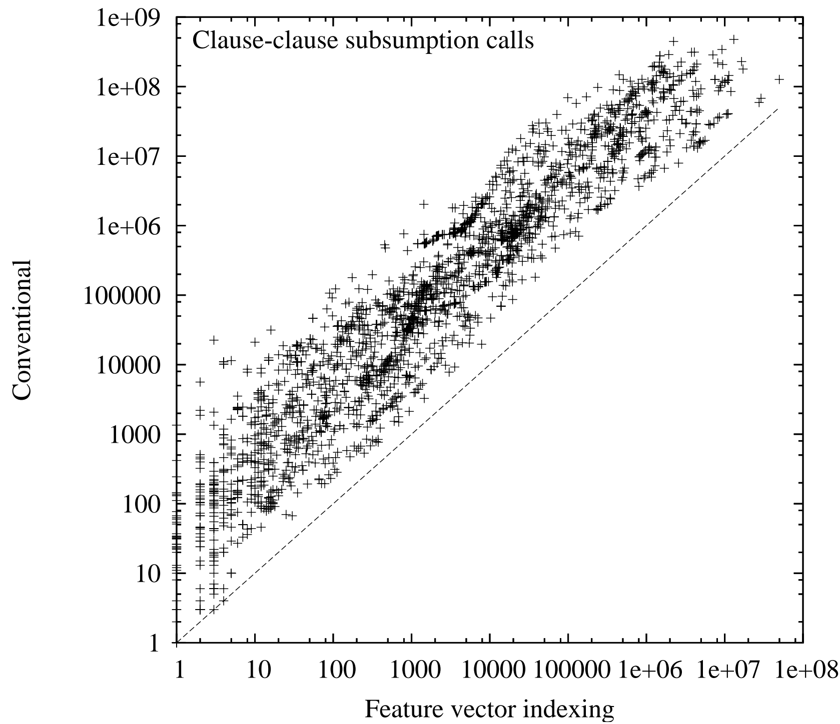


Fig. 5. Subsumption calls of indexed versus conventional implementation

6.2 Results for the Automatic Mode

Most users use E in *automatic mode*, where the prover analyses the problem, and then configures itself to use a strategy that has performed well on similar problems. In this mode, contextual literal cutting is usually only applied to the given clause, and thus the overall cost of subsumption-related techniques is lower to begin with. We have performed various experiments to measure the effect of feature vector indexing for this scenario as well. Figure 6 compares the run times of E with and without feature vector indexing using the automatic mode included with E 0.8. The conventional version solves 3405 problems, whereas the version using feature vector indexing proves a superset of 3438 problems. On the subset solved by both systems, the indexed version uses 34438 s, whereas the conventional one uses 40238 s, for a speed-up of about 15%.

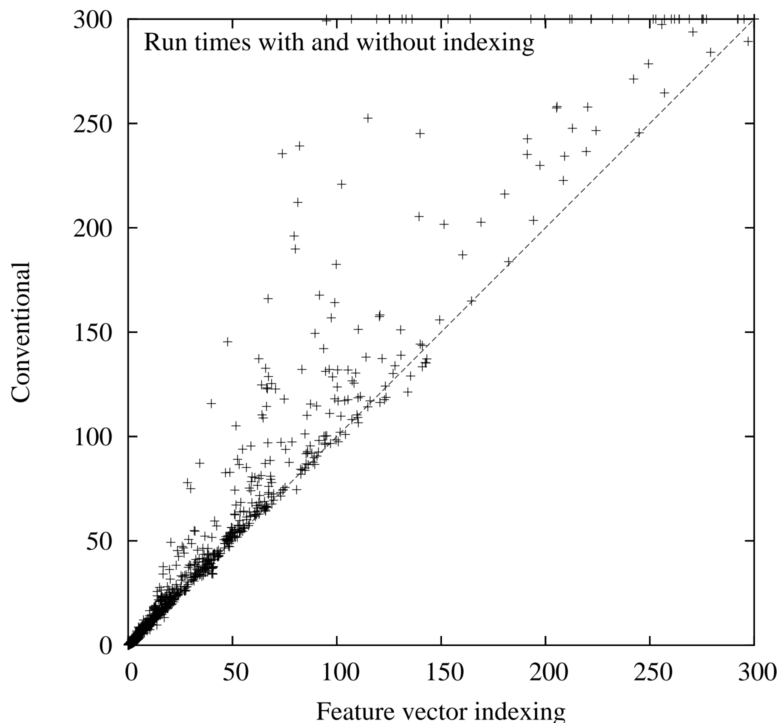


Fig. 6. Run times of indexed versus conventional implementation for E 0.8 automatic mode

The plot is similar to Figure 4, with many problems, especially for lower run times, showing similar performance for both versions, and another group showing significant improvement. Of course, since overall subsumption cost is lower for the automatic mode, the gains are not as pronounced as for the case with aggressive contextual literal cutting.

7 Future Work

While we are very satisfied with the performance of our current implementation, there are a number of research problems we are actively working on.

First, with increased speed of hardware, it has become very hard to actually quantify the time spent in different parts of a high-performance prover using standard UNIX profiling tools (which only resolve to the 1/100th second level). Thus, in this paper we only compare overall performance of the system with and without feature vector indexing. We are in discussions with some of the authors of the COMPIT framework [8] to extend it to cover (unit and non-unit) subsumption, so that more detailed measurements and a comparison of different indexing techniques become easier.

Secondly, up to now, we have only experimented with some simple and obvious features. In particular, all of the features used with our currently best parameter settings are AC compatible, and hence will not differentiate between clauses that are equal modulo AC theories. Whereas this is desirable if subsumption modulo AC is used, it is a disadvantage for our calculus, which only allows us to use some limited AC redundancy elimination. We will investigate the effect of more complex features in these cases.

Finally, we are trying to develop similar simple, but effective algorithms for other operations in the system, in particular for backward simplification (where it might be possible to use a slightly modified version of feature vector indexing) and paramodulation.

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 for 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.

References

[1] Denzinger, J., M. Kronenburg and S. Schulz, *DISCOUNT: A Distributed and Learning Equational Prover*, Journal of Automated Reasoning **18** (1997), pp. 189–198, special Issue on the CADE 13 ATP System Competition.

[2] Graf, P., “Term Indexing,” LNAI **1053**, Springer, 1995.

[3] Graf, P. and D. Fehrer, *Term Indexing*, in: W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, Applied Logic Series **9 (2)**, Kluwer Academic Publishers, 1998 pp. 125–147.

[4] Jaynes, E., “Probability Theory: The Logic of Science,” Cambridge University Press, 2003.

[5] Kapur, D. and P. Narendran, *NP-Completeness of the Set Unification and Matching Problems*, in: J. Siekmann, editor, *Proc. of the 8th CADE, Oxford*, LNCS **230** (1986), pp. 489–495.

[6] Löchner, B. and T. Hillenbrand, *A Phytography of Waldmeister*, Journal of AI Communications **15** (2002), pp. 127–133.

[7] McCune, W., *Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval*, Journal of Automated Reasoning **9** (1992), pp. 147–167.

[8] Nieuwenhuis, R., T. Hillenbrand, A. Riazanov and A. Voronkov, *On the Evaluation of Indexing Techniques for Theorem Proving*, in: R. Goré, A. Leitsch and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, LNAI **2083** (2001), pp. 257–271.

[9] Riazanov, A. and A. Voronkov, *The Design and Implementation of VAMPIRE*, Journal of AI Communications **15** (2002), pp. 91–110.

[10] Riazanov, A. and A. Voronkov, *Efficient Instance Retrieval With Standard and Relational Path Indexing*, in: F. Bader, editor, *Proc. of the 19th CADE, Miami*, LNAI **2741** (2003), pp. 380–396.

[11] Schulz, S., *Information-Based Selection of Abstraction Levels*, in: I. Russel and J. Kolen, editors, *Proc. of the 14th FLAIRS, Key West* (2001), pp. 402–406.

[12] Schulz, S., *E – A Brainiac Theorem Prover*, Journal of AI Communications **15** (2002), pp. 111–126.

[13] Schulz, S., *System Description: E 0.81*, in: *Proc. of the 2nd IJCAR, Cork, Ireland*, LNAI, 2004, (to be published).

[14] Sekar, R., I. Ramakrishnan and A. Voronkov, *Term Indexing*, in: A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, II, Elsevier Science and MIT Press, 2001 pp. 1853–1961.

[15] Shannon, C. and W. Weaver, “The Mathematical Theory of Communication,” University of Illinois Press, 1949.

[16] Sleator, D. and R. Tarjan, *Self-Adjusting Binary Search Trees*, Journal of the ACM **32** (1985), pp. 652–686.

[17] Tammet, T., *Towards Efficient Subsumption*, in: C. Kirchner and H. Kirchner, editors, *Proc. of the 15th CADE, Lindau*, LNAI **1421** (1998), pp. 427–441.

[18] Voronkov, A., *The Anatomy of Vampire: Implementing Bottom-Up Procedures with Code Trees*, Journal of Automated Reasoning **15** (1995), pp. 238–265.

[19] Weidenbach, C., *SPASS: Combining Superposition, Sorts and Splitting*, in: A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, II, Elsevier Science and MIT Press, 2001 pp. 1965–2013.