Subsumption Algorithms Based on Search Trees

Leo Bachmair
Ta Chen
C.R. Ramakrishnan
I.V Ramakrishnan

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, U.S.A.

Abstract. Clause subsumption is of fundamental importance for reducing the search space in theorem proving systems. Since the subsumption problem is NP-complete, the design of efficient heuristics is of significant interest. The core of all subsumption algorithms is the search for suitable substitutions for the variables in a given clause, which in all previously known algorithms is implicitly embedded in the control structure of the algorithm. In this paper we adopt a more abstract view of subsumption and introduce the concept of a *subsumption search tree* to separate the search control from other computational tasks, such as computing substitutions and verifying their consistency. We study key algorithmic aspects of search trees and of heuristics for constructing them. For instance, the complexity of a search-tree based algorithm depends on the height of the search tree. We show that the problem of constructing minimal-height search trees is NP-complete. We also derive improved upper bounds on the height of search trees constructed according to an analysis based on variable dependencies, as proposed by Gottlob and Leitsch; and show that the bound is essentially tight in the worst case, by establishing suitable lower bounds for arbitrary search trees. In addition to these theoretical results, we propose further algorithmic improvements based on more sophisticated data structures for computing and representing substitutions. Finally, we have implemented several variants of our proposed algorithm and report on corresponding experiments.

1 Introduction

The size of the search space (of formulas to be deduced) is a critical factor for the performance of automated theorem provers, and considerable effort has been spent on developing techniques to reduce the search space by eliminating redundant formulas. In clause-based provers, one of the most important and effective techniques is based on subsumption and allows one to delete a clause D in the presence of another clause C, if $C\sigma$ is contained in D, for some substitution σ. For instance, in [WOL91] an example is mentioned of a proof with 20,341 retained clauses, in which 4,319,586 intermediate clauses were eliminated based

on subsumption tests. Such results are not untypical, and almost all current clausal theorem provers use subsumption as key deletion strategy for eliminating redundant formulas, so that the design of efficient subsumption algorithms is a problem of undisputed importance.

It is well known that subsumption is an *NP*-complete problem [GJ79]. Subsumption may require one, for two clauses C and D with m and n literals, respectively, to explore as many as n^m combinations of matching substitutions for pairs of literals from C and D. A better bound of $O(mn^{k/2+2})$, where k is the number of variables in C, was established by Gottlob and Leitsch [GL87] for an algorithm that analyzes the dependencies among variables in C to guide the search for substitutions.

In this paper we present a more abstract approach to subsumption by introducing the concept of a *subsumption search tree*. Gottlob and Leitsch's algorithm, for instance, can be shown to be based on one such search tree. A crucial aspect of our approach is that it separates the search control from the other computational aspects of subsumption, namely computing substitutions and verifying their consistency. Some of the theoretical results in this paper require this more abstract view of subsumption, but our experience with an implementation indicates that the flexibility afforded by our formulation is also useful for practical purposes. Let us briefly summarize our main results:

- We present an abstract description of a large class of subsumption algorithms via the concept of subsumption search trees.
- The complexity of a search tree-based subsumption algorithm depends on the height of the underlying search tree. We show that the problem of constructing minimal-height search trees is NP-complete.
- We express the search control mechanisms underlying Gottlob and Leitsch's algorithm in terms of search trees and demonstrate that our approach allows us to obtain a slightly better complexity bound of $O(mn^{\lceil \frac{k}{2} \rceil + 1})$, with a simpler analysis.
- We suggest suitable trie-like data structures for computing and representing substitutions and checking their consistency, that result in a further improvement of the asymptotic complexity to $O(mn^{\lceil \frac{k}{2} \rceil})$. We show that this bound is tight for the worst case, by establishing a suitable lower bound on the height of search trees in general.

The paper is organized as follows. In Section 3 we describe the central concept of our method, subsumption search trees, and show that the complexity of a corresponding subsumption algorithm is directly related to the height of the search tree used. In section 4 we study the problem of constructing search trees. We first show that the problem of constructing minimal-height search trees is NP-complete, and then describe heuristics to construct search trees of reasonably small height. We also establish lower bounds on the height of search trees in general. A formal description of the overall algorithm is given in Section 5. In section 6 we propose data structures that allow efficient computation of substitutions and corresponding consistency checks. To validate the practicality of our

technique, we modified the theorem prover *Otter* to use several variants of our proposed subsumption method. We discuss our experience with the implementation and the experimental results in section 7.

2 Preliminaries

We consider terms built from function symbols and variables; and (atomic) formulas built from predicate symbols and terms. The letters f and g are used to denote function symbols; p and q, to denote predicate symbols; x, y and z, to denote variables; and s and t to denote terms. A *literal* is an atomic formula $P(t_1, \ldots, t_n)$ or the negation thereof, $\neg P(t_1, \ldots, t_n)$. A *clause* is a set of literals.[1] We use the letters C and D to denote clauses; and c and d to denote literals. If E is an expression (i.e., a term or formula), we denote by $var(E)$ the set of all variables occurring in E. If E contains no variables (i.e., $var(E)$ is empty), then E is said to be *ground*.

A *substitution* is a mapping from variables to terms. By $E\sigma$ we denote the result of applying a substitution σ to an expression E. By the *domain* of a substitution σ, denoted by $dom(\sigma)$, we mean the set of all variables x, for which $x \neq x\sigma$. We only need to consider substitutions with a finite domain; and write $[x_1/t_1, \ldots, x_n/t_n]$ to denote the substitution σ with finite domain $\{x_1, \ldots, x_n\}$, for which $x_i\sigma = t_i$, for all i with $1 \leq i \leq n$. By ϕ we denote the substitution with the empty set as domain.

We say that an expression E *matches* E' if there exists a substitution σ such that $E' = E\sigma$. Two substitutions σ and θ are said to be *consistent* if $x\sigma = x\theta$, for all variables x in $dom(\sigma) \cap dom(\theta)$.

The *composition* $\sigma \circ \theta$ of two substitutions σ and θ is a substitution with domain $dom(\sigma \circ \theta) = dom(\sigma) \cup dom(\theta)$, such that

$$x(\sigma \circ \theta) = \begin{cases} x\sigma & \text{if } x \in dom(\sigma) \\ x\theta & \text{otherwise} \end{cases}$$

We shall only have to consider the composition of consistent substitutions.

3 Subsumption search trees

We say that a clause $C = \{c_1, \ldots, c_m\}$ *subsumes* another clause $D = \{d_1, \ldots, d_n\}$ if there exists a substitution θ such that the *set* $C\theta$ is a subset of D. The substitution θ, if it exists, need not be unique. For example, if $C = \{p(x, y), p(y, z)\}$ and $D = \{p(a, a), p(b, b)\}$, then both $C[x/a, y/a, z/a]$ and $C[x/b, y/b, z/b]$ are subsets of D. We will assume, without loss of generality, that the clause D is ground and that $x\theta$ is a ground term, for all variables x in the domain of θ.

Evidently, a clause C subsumes another clause D if, and only if, for each literal c_i in C there exists a substitution σ_i, such that (i) $c_i\sigma_i$ is an element

[1] Sometimes clauses are defined in terms of multisets, i.e., with possible multiple occurrences of literals; which requires slightly different subsumption algorithms.

of D and (ii) the substitutions σ_i are pairwise consistent. Subsumption is thus a special case of the ACI-subterm matching problem: C subsumes D if, and only if, C matches some subclause of D modulo associativity, commutativity and idempotence.[2] In essence, subsumption is a combinatorial search problem: each literal c_i can potentially be matched with any one of the n literals d_k and one has to identify a suitable combination of consistent matching substitutions. Emphasizing this search aspect, we formulate subsumption algorithms in terms of a suitable search structure.

Definition 1. A *(subsumption) search tree* for a clause $C = \{c_1, \ldots, c_m\}$ is a (rooted) tree with set of nodes C, such that whenever a variable x occurs in two nodes c and c' on different (root-to-leaf) branches of the tree, then x also occurs in some common ancestor of c and c'.

For example, a "linear" tree with root c_1, in which each node c_{i+1} is the only child of c_i, for all i with $i < n$, is a subsumption search tree.

Given a search tree T_C for a clause C, we may test whether C subsumes a clause D by essentially traversing the tree T_C in preorder, constructing for each node c a substitution σ_c such that (i) $c\sigma_c$ matches some literal in D and (ii) σ_c is consistent with *all* the substitutions assigned to the ancestors of c. If some node c matches *none* of the literals in D, then the subsumption test fails. If suitable substitutions σ_c do exist, but none of them is consistent with any of the corresponding ancestor substitutions, then the algorithm backtracks to the predecessor node to select a different matching substitution for it, if possible. If repeated backtracking steps produce no alternative matching substitutions to be explored, the subsumption test again fails. The test succeeds only if all nodes can be traversed successfully. A more formal description of the algorithm is given in Figure 1.

Correctness of the algorithm. It can easily be seen that if $Search(T_C, \phi)$ returns $\langle \phi, FAILURE \rangle$, then C does not subsume D. On the other hand, if $Search(T_C, \phi)$ returns a pair $\langle \gamma, SUCCESS \rangle$, then to each node c_i in T_C a substitution σ_i has been assigned, such that (i) γ is the composition of all substitutions σ_i and (ii) all literals $c_i\sigma_i$ are elements of D. To show that C does subsume D, it suffices to show that any two substitutions σ_i and σ_j are consistent. This is obviously the case if one of the two nodes c_i or c_j is an ancestor of the other, as the algorithm contains an explicit consistency check for this case. Suppose c_i and c_j are on different branches of T_C. If a variable x occurs in both literals, then by the properties of a subsumption search tree, it must also occur in some common ancestor, say c_k, of c_i and c_j. Furthermore, both σ_i and σ_j must be consistent with the ancestor substitution σ_k, so that $x\sigma_i = x\sigma_k = x\sigma_j$. We may conclude that σ_i and σ_j are consistent.

The correctness proof establishes the key property of subsumption search trees, namely that the different subtrees of a node represent independent sub-

[2] If clauses are defined as multisets, subsumption corresponds to AC-subterm matching.

```
algorithm Search(T, σ)
▷
      The algorithm determines whether there exists a substitution γ
      consistent with σ, such that cγ ∈ D, for all c in T.
◁
   let c be the root of T
   let {⟨d_1, θ_1⟩, ⟨d_2, θ_2⟩, ..., ⟨d_k, θ_k⟩} be the set of all literal/substitution pairs
         such that d_i ∈ D and cθ_i = d_i
   if k = 0 then return ⟨φ, FAILURE⟩
   let {c_1, c_2, ..., c_l} be the children of c
   for i := 1 to k do
      if θ_i is consistent with σ then
         begin
            γ := θ_i
            cond := SUCCESS
            for j := 1 to l do
               begin
                  ⟨θ, cond⟩ := Search(c_j, σ ∘ θ_i)
                  if cond = FAILURE then
                     break
                  else
                     γ := γ ∘ θ
               end
            if cond = SUCCESS then
               return ⟨γ, SUCCESS⟩
         end
   return ⟨φ, FAILURE⟩
```

Fig. 1. Subsumption search algorithm

problems that can be solved separately (or in parallel, for that matter) once matching substitutions for the ancestor literals have been selected.

We also speak of a *binding occurrence* of a variable x in a node c of a subsumption search tree if x occurs in c but in no ancestor of c; and call occurrences of x in descendants of c *non-binding*. Thus, the key property of a subsumption search tree is that for each of its variables there is a unique node with binding occurrences of that variable, and that all other occurrences of the same variable are non-binding.

Complexity of the algorithm. Let T be a subsumption search tree with m nodes and D be a clause with n literals. Then $Search(T, \sigma)$ needs to compute as many as mn literal/substitution pairs $\langle d, \theta \rangle$, which requires polynomial time (in

m, n and the size of D). In addition, the algorithm has to check the consistency of certain substitutions. More specifically, at each node of T we may select from as many as n matching substitutions, so that in r recursively nested calls to *Search* as many as n^r different combinations of substitutions may have to be explored. The time needed for each consistency check is polynomial in the size of the clause D and the number of variables in literals of T. The dominant factor for the complexity of the algorithm is thus the number of consistency checks.

Lemma 2. *If T is a tree of height h with k leaves and D is a clause with n literals, then the number of consistency checks in Search(T, σ) is $O(kn^{h+1})$.*

For example, in Stillman's algorithm [Sti73] the search is organized according to a linear search tree of maximal possible depth $m-1$. The algorithm constructs consistent matching substitutions dynamically (i.e., the consistency checks are done as part of the matching attempts) and requires $O(n^m)$ matching attempts in the worst case. More specifically, the number of matching attempts in Stillman's algorithm may be $n + n^2 + \cdots + n^m = n(n^m - 1)/(n - 1)$.

All algorithms that have been proposed for clause subsumption are exponential. The subsumption problem itself is NP-complete [GJ79].

4 Construction of search trees

4.1 Minimal-depth search trees

In view of the above discussion, the problem of constructing subsumption search trees of minimal depth is of obvious interest. Unfortunately, this problem also turns out to be NP-complete. Let us sketch the basic idea of the NP-completeness proof. We consider the following decision problem:

Min-ST: Given a clause C and a positive integer K, is there a subsumption search tree T_C for C such that $height(T_C) \leq K$?

The following theorem establishes the difficulty of solving **Min-ST**.

Theorem 3. Min-ST *is NP-complete.*

Clearly, since the height of a given tree can be computed in polynomial time, **Min-ST** is in NP. The problem can be shown to be NP-hard by reduction from **Set Cover**; for details see the appendix.

4.2 Heuristics

We next present some heuristic guidelines for constructing search trees of reasonably small depth. The heuristics are based on an analysis of the variable dependencies among the literals in a given clause C and are derived from techniques proposed by Gottlob and Leitsch [GL87].

First note that if two clauses C_1 and C_2 have no variables in common, then $C_1 \cup C_2$ subsumes a clause D if, and only if, both C_1 and C_2 subsume D. In

other words, the subsumption test for a clause C can be done independently for its variable-disjoint subclauses.

For example, the clause $C = \{p(x,y), p(z,b), p(x,x)\}$ consists of two variable-disjoint subclauses, $\{p(x,y), p(x,x)\}$ and $\{p(z,b)\}$, both of which subsume $D = \{p(a,b), p(b,b)\}$. Thus, C also subsumes D.

To describe more sophisticated decomposition techniques, we need to define the notion of a "variable dependency graph."

Definition 4. If C is a clause and V is a set of variables, the corresponding *variable dependency graph* $G(C,V)$ is defined to be the labeled undirected graph (C, E, v), where E consists of all pairs (c,c') of literals in C, for which $var(c) \cap var(c') \cap V \neq \emptyset$; and, for each pair (c,c') in E, the label $v(c,c')$ is defined to be the set $var(c) \cap var(c') \cap V$. In other words, $v(c,c')$ indicates which variables from V occur in both c and c'.

Example 1. Let C be the clause $\{p_1(x,z), p_2(x,y), p_3(y), p_4(x,y), p_5(z)\}$. The variable dependency graph $G(C, \{x,y,z\})$ is shown in Figure 2.

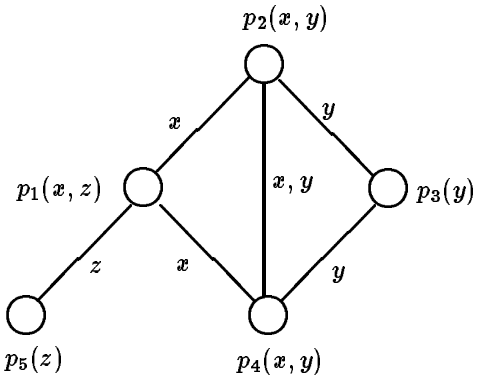

Fig. 2. A variable dependency graph

Note that if the graph $G(C, var(C))$ is not connected, then C can be decomposed into variable-disjoint subclauses. We only construct search trees for clauses with a variable dependency graph that is connected.

Definition 5. Let C be a clause and V a set of variables, such that the graph $G(C,V)$ is connected. Let c be a literal in C, for which $var(c) \cap V$ has the most number of elements[3] and let

$$G(C_1, V \setminus var(c)), \ldots, G(C_k, V \setminus var(c))$$

[3] The literal c need not be unique, and different trees may result for different selections of c.

be the connected components of $G(C \setminus \{c\}, V \setminus var(c))$. Then the search tree $T(C, V)$ is defined as the tree with root c and subtrees

$$T(C_1, V \setminus var(c)), \ldots, T(C_k, V \setminus var(c)).$$

Example 2. Figure 3 shows two possible search trees for the clause from Example 1.

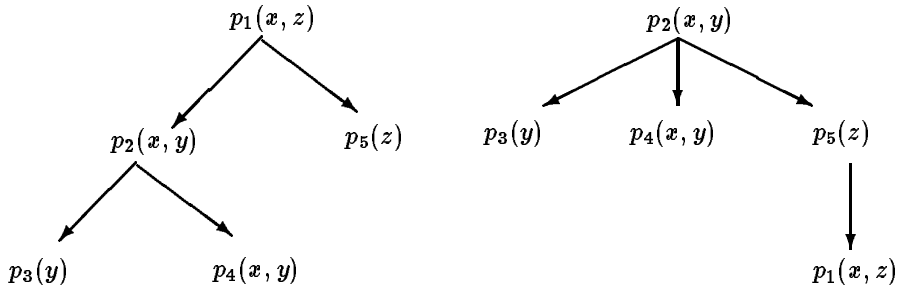

Fig. 3. Subsumption search trees

We are interested in search trees $T(C, var(C))$ for which the variable dependency graph $G(C, var(C))$ is connected. In such a search tree the subtree rooted at a node c_i is a tree $T(C_i, V_i)$, where $C_i \subseteq C$ and $V_i \subseteq var(C)$. When the algorithm *Search* is run on the tree $T(C, var(C))$, and the tree traversal reaches node c_i, all variables in $var(C) \setminus V_i$ have been instantiated by substitutions associated with ancestors of c_i. In other words, all occurrences in c_i of variables from $var(C) \setminus V_i$ are non-binding, whereas occurrences of variables from $var(c_i) \cap V_i$ are binding.

We have the following lemma:

Lemma 6. *Let c_i be a node in a search tree $T(C, var(C))$ and let $T(C_i, V_i)$ be the subtree rooted at c_i. If $var(c_i) \cap V_i = \emptyset$, then c_i is a leaf; and if $var(c_i) \cap V_i$ contains only one variable, then c_i is a leaf or the parent of a leaf.*

Proof. Let $G_i = G(C_i, V_i)$ be the variable dependency graph corresponding to node c_i. This graph is connected and each of its edges is labeled by one or more variables in $var(c_i) \cap V_i$. Thus, if $var(c_i) \cap V_i = \emptyset$, then the graph G_i consists of a single node; and hence c_i is a leaf. On the other hand, if c_i is not a leaf, then G_i contains at least one edge. If $var(c_i) \cap V_i$ is a singleton, say $\{x\}$, then all edges of G_i are labeled by x. Thus, each connected component in the variable dependency graph $G(C_i \setminus \{x\}, V_i \setminus \{x\})$ consists of a single node; and hence all children of c_i are leaves.

Theorem 7. *The height of a subsumption search tree $T(C, var(C))$ is at most $\lceil \frac{k}{2} \rceil$, where k is the total number of variables occurring in C.*

Proof. Let T_C be the search tree $T(C, var(C))$ and h be the height of T_C. Furthermore, let $c_1, \ldots, c_h, c_{h+1}$ be a longest path in T_C and $T(C_i, V_i)$ be the subtree rooted at c_i, for $1 \leq i \leq h+1$. We have $V_1 = var(C)$ and know, from Lemma 6, that all nodes c_i, with $1 \leq i < h$, have binding occurrences for at least two variables and that c_h has at least one binding occurrence. Taking k to be $|var(C)|$, we thus have $2(h-1)+1 \leq k$, which implies that $h \leq \frac{k+1}{2}$ and, hence, $h \leq \lceil \frac{k}{2} \rceil$.

The following lemma provides a lower bound on the height of subsumption search trees.

Lemma 8. *For every $k \geq 2$ there exists a clause C with k variables, such that every subsumption search tree for C has height $\frac{k}{2}$ at least.*

Proof. Suppose $k \geq 2$ and let C be the clause

$$\{p(x_1, x_2), p(x_1, x_3), \ldots, p(x_1, x_k), p(x_2, x_3), \ldots, p(x_2, x_k), \ldots, p(x_{k-1}, x_k)\}$$

containing $k(k-1)/2$ literals and k variables. Let T be a search tree for C of height h. We claim that if x_i and x_j are distinct variables with binding occurrences in c_i and c_j, respectively, then c_i and c_j must lie on the same (root-to-leaf) branch in T. For suppose, to the contrary, that c_i and c_j are on different branches. Let c_k be the literal $p(x_i, x_j)$, if $i < j$, and $p(x_j, x_i)$, if $j < i$. Since c_k shares the variable x_i with c_i and x_j with c_j, and c_i and c_j are on different branches, c_k must be a common ancestor of (and distinct from) c_i and c_j. But then c_i and c_j cannot contain binding occurrences of x_i and x_j, respectively. We conclude that c_i and c_j must be on the same branch. In short, all binding occurrences of variables in a search tree for C must lie on the same branch. Furthermore, it can easily be seen that leaves contain no binding occurrences. Since all literals in C are binary, there are at most two binding occurrences per node, so that $2h \geq k$, or $h \geq \frac{k}{2}$.

5 A subsumption algorithm

The search-based subsumption algorithm shown in Figure 4 first decomposes a clause into its variable-disjoint subclauses and then uses search trees for the resulting subclauses.

Theorem 9. *If C and D are clauses with m and n literals, respectively, and C contains k different variables, then $Subsume(C, D)$ requires atmost $O(mn^{\lceil \frac{k}{2} \rceil + 1})$ consistency checks if the construction of search trees is based on variable dependency graphs.*

Proof. By Theorem 7, the height of a search tree constructed from variable dependency graphs is at most $\lceil \frac{k}{2} \rceil$, while the number of literals in C provides a bound on the number of paths in the tree. By Lemma 2, at most $O(mn^{\lceil \frac{k}{2} \rceil + 1})$ consistency checks may be needed.

```
algorithm Subsume(C, D)
▷
        The algorithm determines whether the clause C subsumes D. It
        returns ⟨θ, SUCCESS⟩ if there exists a substitution θ, such that
        Cθ ⊆ D, and ⟨φ, FAILURE⟩ if no such substitution exists.
◁
    let C_1, ..., C_l be the variable-disjoint subclauses of C
    θ := φ
    for i := 1 to l do
        begin
            construct a search tree T for C_i
            ⟨γ, cond⟩ := Search(T, φ)
            if cond = FAILURE then
                return ⟨φ, FAILURE⟩
            else
                θ := θ ∘ γ
        end
    return ⟨θ, SUCCESS⟩
```

Fig. 4. Subsumption algorithm

The technique of decomposition based on an analysis of variable dependencies was proposed by Gottlob and Leitsch [GL87] in their subsumption algorithm. While the latter algorithm does not explicitly construct a search tree, its control structure implicitly defines a search tree that essentially corresponds to the tree we have described in Section 4.2. The only difference is that Gottlob and Leitsch do not decompose a clause if it is *simple*, that is, either is a unit clause or else contains at most one variable with a binding occurrence. The rationale for not decomposing simple clauses is that the corresponding subsumption problem can be solved in polynomial time. However, the search tree *we* construct for a simple clause also yields a polynomial search procedure, so that we achieve the same effect, but within a uniform framework. In fact, our analysis produces a slightly better bound of $O(mn^{\lceil \frac{k}{2} \rceil + 1})$, on the number of consistency checks, than the bound of $O(mn^{\frac{k}{2}+2})$, on the number of matching attempts, obtained by Gottlob and Leitsch. In Gottlob and Leitsch's algorithm consistency checking is embedded in the process of computing consistent matching substitutions, whereas we consider matching and consistency checking separately. This difference has no effect on the asymptotic complexity of the subsumption algorithm, which is dominated by the complexity of the inherent combinatorial search problem.

6 Consistency checking

We have separated consistency checking from term matching, and precompute all possible matches between literals in C and D, so as to avoid recomputing the same substitution repeatedly during the subsumption search. Let us briefly describe a possible way of computing all matching substitutions.

Given clauses C and D, first construct a discrimination net N_C for C. A discrimination net is a data structure that supports an operation $Match(d, N_C)$, which returns for a given literal d the set of all literals c in C that match d plus corresponding matching substitutions. Thus, we may find all desired matching substitutions by calling $Match(d, N_C)$ for all literals d in D.

Once the matching substitutions have been computed, they can be stored in a trie-like data structure that allows one to efficiently perform any necessary consistency checks. First let us order variables according to their binding occurrence in a given search tree T. More precisely, we define: $x <_T^b y$ if, and only if, x and y have binding occurrences at nodes c and c' in T, respectively, such that c is an ancestor of c'.

Definition 10. Let T be a subsumption search tree and D be clause. For each clause c in T let S_c be the set of all substitutions σ, such that $c\sigma \in D$, and V_c be the set of all variables with non-binding occurrences in c. The substitution trie T_c for c is a tree in which each edge (u, v) is labeled by a term $x\sigma$, for some $x \in V_c$ and substitution $\sigma \in S_c$, each node u is labeled by a subset M_u of S_c, and each interior node v is labeled by a variable $\nu(v) \in V_c$; such that

(i) if u is the root of the tree, then $M_u = S_c$;

(ii) if an edge (u, v) is labeled by a term t, then $M_v = \{\theta \in M_u : \nu(u)\theta = t\}$;

(iii) if u is an ancestor of v, then $\nu(v) \not<_T^b \nu(u)$; and

(iv) if u and v are nodes of the same depth, then $\nu(u) = \nu(v)$.

For example, suppose a literal c contains non-binding occurrences of x, y and z, with $x <_T^b y <_T^b z$, and $S_c = \{\sigma, \tau, \theta\}$, where $\sigma = [x/t_1, y/t_2, z/t_4]$, $\tau = [x/t_1, y/t_2, z/t_5]$, and $\theta = [x/t_1, y/t_3, z/t_6]$. A corresponding substitution trie is shown in Figure 5.

Given a substitution trie T_c, we may find out with which substitutions from S_c a given substitution π, with $dom(\pi) \subseteq V_c$, is consistent, by simply traversing the trie T_c, beginning at the root and choosing, for each node u with label $\nu(u) = x$, a successor node v such that the edge (u, v) labeled by $x\pi$. If no suitable edge exists, the traversal fails, indicating that π is not consistent with any substitution in S_c. If the traversal reaches a leaf v, then M_v contains all the substitutions in S_c with which π is consistent.

The use of substitution tries has the advantage that once $Search(c, \tau)$ reaches a leaf in a subsumption search tree, we may use the trie to determine whether a consistent substitution exists (and pick one such), but need not check the substitutions separately. In other words, no search among different matching substitutions is necessary at the lowest level of the search tree, which means that the bound on the number of consistency checks can be improved to $O(mn^{\lceil \frac{k}{2} \rceil})$.

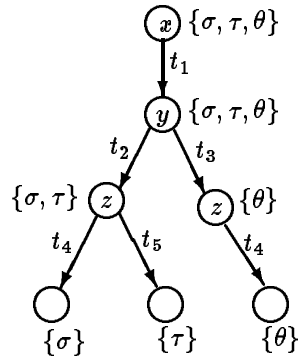

Fig. 5. A substitution trie

7 Implementation

In the previous sections we have presented a subsumption algorithm, analyzed its complexity, and suggested further improvements based on advanced data structures for representing terms and substitutions. We have implemented several variants of our algorithm and run extensive experiments. The main vehicle for our experiments was the theorem prover Otter [McC94], a well-known resolution-type theorem proving system. We selected about 70 typical problems from the TPTP problem library [SSY94] and ran them on various versions of Otter,[4] which differed only in the subsumption algorithm they used.

The native Otter system uses a straightforward subsumption algorithm, akin to Stillman's algorithm, but with some shortcuts made possible by the specific data structures used by the theorem prover. We first ran all test problems on Otter and then replaced the original subsumption algorithm by our algorithm, with all the refinements such as substitution tries. The modification did not improve the subsumption time on any example, but for a few examples actually increased it (by up to a factor of four).

We then implemented a slimmer version of our algorithm that employs search trees, but uses a straightforward method for checking consistencies (not substitution tries). For this variant we got improvements of about 5 to 10% on some examples, with no noticeable difference (changes of $\pm 1\%$) in the remaining problems.

The experiments indicate that in the context of Otter the overhead of building substitution tries outweighed any resulting improvements in the subsumption search. On the other hand, subsumption search trees require relatively little overhead, and our second algorithm is comparable in performance to the original

[4] We chose problems with different characteristics according to such parameters as number of clauses derived, etc. All selected problems could be successfully solved by Otter.

subsumption algorithm in Otter. Considering that Otter is a well-engineered system and subsumption is one of its key components [WOL91], the experiments provide evidence that our proposed search-tree based approach to subsumption is feasible in practice.

The improvements we did obtain were modest and applied only to those few test problems that involved larger clauses (or clauses with larger terms). The theorem proving search strategies used by Otter, and most other current resolution-type provers, favor the generation of small clauses. In fact, many provers may be tuned so as to actually disregard (i.e., delete) clauses that exceed a certain size limit. Consequently, subsumption tests will predominantly be applied to relatively small clauses, for which subsumption search trees may yield only minor improvements in the overall search time. More significant improvements might be expected in applications to problems requiring more complicated formulas.

In sum, our experience with the implementation shows that the search tree-based approach to subsumption is feasible and that corresponding algorithms can be easily and tightly integrated into an existing system. The modular structure of our algorithm is important as it allows one to "mix and match" the various components in a way that leads to a practical implementation.

8 Conclusion

In this paper we proposed an abstract algorithmic framework for the design of subsumption algorithms that is based on the concept of a subsumption search tree, through which we separate the search control from other computational tasks. Using this framework we have established both new theoretical results about the asymptotic complexity of subsumption algorithms and improved known results. Moreoever, we have shown that the separation of control and computation provides enough flexibility for practical implementations of different variants of the algorithm. Finally, we believe that our proposed approach and the underlying concepts can be applied in other areas, such as logic programming, where the techniques should be useful for the problem of reordering subgoals in the body of a clause, so as to improve the efficiency of backtracking.

References

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman, San Francisco, 1979.

[GL87] G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *JACM*, 32(2):280–295, April 1987.

[McC94] W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, Argonne, Ill., 1994.

[SSY94] Geoff Sutcliffe, Christian Suttner, and Theodor Yemenis. The TPTP problem library. In *Proc. 12th Int. Conf. on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 252–266. Springer-Verlag, Berlin, 1994.

[Sti73] R. B. Stillman. The concept of weak substitution in theorem-proving. *JACM*, 20(4):648–667, 1973.

[WOL91] L. Wos, R. Overbeek, and E. Lusk. Subsumption, a sometimes undervalued procedure. In J.-L. Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 3–40. MIT Press, 1991.

Appendix: Complexity of building minimum height search trees

The complexity of our algorithm depends directly on the height of the search trees used, as using a minimum height tree for a given clause guarantees the best worst case behavior for that clause. However, the problem of constructing such a tree turns out to be NP-complete. We consider the corresponding decision problem:

Min-ST: Given a clause C and a positive integer K, is there a search tree T_C for C such that $height(T_C) \leq K$?

Theorem 11. **Min-ST** *is NP-complete.*

Proof. Clearly, since height of a given tree can be computed in polynomial time, **Min-ST** is in NP. We show that the problem is NP-hard by reduction from the following problem (adapted from Garey and Johnson [GJ79]):

Set Cover: Given a collection $\mathcal{C}$ of subsets of a finite set S and a positive integer $K \leq |\mathcal{C}|$, is there a subset $\mathcal{C}' \subseteq \mathcal{C}$ with $|\mathcal{C}'| \leq K$ such that every element of S belongs to at least one member of $\mathcal{C}'$?

Let the collection $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ of subsets of $S = \{a_1, a_2, \ldots, a_m\}$ and integer $K \leq |\mathcal{C}|$ be an arbitrary instance of **Set Cover**. We construct an instance of **Min-ST** as follows.

The clause C in the instance of **Min-ST** consists of literals drawn from three sets: c-literals that correspond to gadgets representing the sets in the collection $\mathcal{C}$, a-literals that correspond to gadgets representing the elements of the set S, and b-literals used as auxiliary gadgets. Let $f_1, f_2, \ldots, f_m$ be a family of function symbols such that f_i has arity i, for all i, $1 \leq i \leq m$. For each set $c_i \in \mathcal{C}$, where $c_i = \{a_{r_1}, a_{r_2}, \ldots, a_{r_n}\}$, we construct a c-literal of the form $c_i(x^0_{\{i,1\}}, x^0_{\{i,2\}}, \ldots, x^0_{\{i,n\}}, f_k(y_{r_1}, y_{r_2}, \ldots, y_{r_k}))$. For each element $a_j \in S$, we construct a a-literal $a_j(x^j_{\{1,1\}}, x^j_{\{1,2\}}, \ldots, x^j_{\{1,n\}}, y_j)$. The last arguments of c-literals and a-literals connect the corresponding sets with their members. Apart from the n c-literals and m a-literals, the clause C contains $n(n-1)$ b-literals. Each b-literal is of the form $b_{i,j}(x^i_{\{j,1\}}, x^i_{\{j,2\}}, \ldots, x^i_{\{j,n\}})$, where $1 \leq i \leq n$ and $2 \leq j \leq n$. The clause C is such that every c-literal, say c_i, shares a unique variable with every other c-literal, and a variable with each a-literal corresponding to the members of the set c_i in $\mathcal{C}$. The b-literals are such that, a literal $b_{i,j}$ has variables in common with the a-literal a_i, and the b-literals $b_{i,k}$, for all k, $2 \leq k \leq n$.

Clearly, the clause C can be constructed in polynomial time given a collection $\mathcal{C}$ of subsets of the set S. The reduction is proved based on a property of the clause C, that minimum height search trees of C have a structure shown in Figure 6. This property is established by the following two lemmas.

Lemma 12. *In any search tree T_C built for C constructed using the process above, given any two c-literals c_i and c_j, either c_i is an ancestor of c_j, or c_i is a descendent of c_j.*

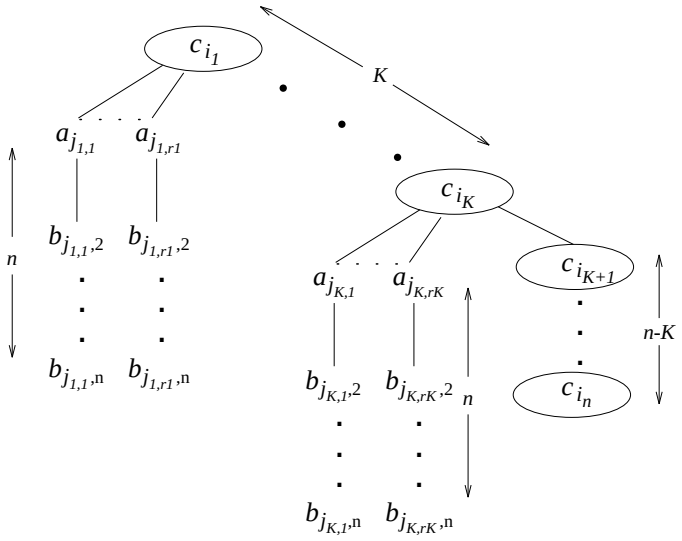

Fig. 6. Structure of minimum height search trees for C

Proof of Lemma. Note that by definition of search trees, two vertices are on distinct root-to-leaf paths iff they do not share a variable not already shared with an ancestor. From the construction of C, the literals c_i and c_j share the variable $x^0_{\{i,j\}}$ and, furthermore, $x^0_{\{i,j\}}$ does not occur in any other literal c_k, $k \notin \{i, j\}$. This leads directly to the above lemma. □

In a given search tree T_C for C, let c_l be the farthest literal from the root, among all the c-literals in C. The path from root to c_l is called as the *spine* of the search tree. We now establish that given any search tree, we can find another of smaller or equal height with a spine consisting only of c-literals.

Lemma 13. *For any search tree, say T_C, for C (as constructed above), there is a search tree T'_C, $height(T'_C) \leq height(T_C)$, such that spine of T'_C consists only of c-literals.*

Proof of Lemma. Let $b_{i,k}$ be in the spine of T_C. It is easily seen that if $b_{i,k}$ is a descendent of a_i (hence a_i is also in the spine of T_C), a search tree of smaller height can be immediately constructed by pulling $b_{i,k'}$, for all k', out of the spine since these literals share variables with no other literals in C. On the other hand, let a_i be a descendent of $b_{i,k}$. If $b_{i,k}$ has more than one child in T_C, note that all except one branch in the subtree rooted at $b_{i,k}$ will contain only literals that have no variable in common with $b_{i,k}$; hence we can attach these independent subtrees to the parent of $b_{i,k}$, without increasing the height of the search tree. Now, we can make a_i, which occurs in the chain starting at $b_{i,k}$, as the parent of $b_{i,k}$, yielding a search tree no taller than T_C. Thus, we can eliminate all b-literals

from the spine of a tree without increasing its height. Hence it is sufficient to consider trees that contain only a-literals and c-literals in their spine.

Let some a_i be in the spine of T_C. If a_i be a descendent of some c_j such that $a_i \in c_j$, then a_i can be pulled out of the spine without increasing the height of T_C. Thus we need to consider only the cases in which a_i is in some c_j, and c_j is a descendent of a_i in T_C. Let $\ldots, \alpha, a_i, \beta_1, \ldots, \beta_k, c_j, \gamma, \ldots$ be the spine of T_C. This case applies whenever a_i is not the root of the tree. The spine can be rearranged as $\ldots, c_j, \alpha, \beta_1, \ldots, \beta_k, \gamma, \ldots$, with a_i becoming a child of c_j. The descendents of c_j (other than those via γ) are closer to the root. The rest of the descendents of β_l $(1 \leq l \leq k)$ and those of γ are no farther from the root than before. The descendents of α may be one level lower than before, but the height of the non-spine subtrees at α is at most n, which is covered by the height of the subtree at a_i. Thus a_i can be removed from the spine without increasing the height of the tree.

We now consider the remaining case when the root of the tree is a a-literal. Without loss of generality we assume that there is no other a-literal in the spine, since we can eliminate such literals using the procedure in the previous paragraph. Let the spine of the tree be $a_i, \beta_1, \ldots, \beta_k, c_j, \gamma, \ldots$, were $a_i \in c_j$. We can rearrange the spine to be $c_j, \beta_1, \ldots, \beta_k, \gamma, \ldots$, making a_i a child of c_j. Note that the height of the tree is not increased as long as there is some other a-literal in the tree. If a_i is the only a-literal in the tree, then $S = c_1 = \{a_i\}$ and $\mathcal{C} = \{c_1\}$. In this degenerate case, we can clearly make c_1 as the root of the search tree (and thus its spine). □

We now show that there is a set cover of size K for $\mathcal{C}$ iff C has a search tree of height $K + n$.

if: Let C have a search tree of height $K + n$, and let T_C be a smallest such tree. Let a longest root-to-leaf path in T_C contain a_i, for some i, $1 \leq i \leq m$. Let c_j be the closest c-literal to a_i in T_C. Let the ancestors of c_j be $\{c_{k_1}, c_{k_2}, \ldots, c_{k_l}\}$. It is now easy to show that $c_j \cup \{c_{k_1}, c_{k_2}, \ldots, c_{k_l}\}$ is a cover of $\mathcal{C}$:

Assume, to the contrary, that there is some $a_{i'} \notin c_j \cup \{c_{k_1}, c_{k_2}, \ldots, c_{k_l}\}$. Then, there is some $c_{j'}$ such that $a_{i'} \in c_{j'}$ and c_j is an ancestor of $c_{j'}$. Note that the literals $b_{i',k}$ for all k, $2 \leq k \leq n$ will occur in some root-to-leaf path containing $a_{i'}$, and thus this path is longer than the path containing a_i, which is a contradiction.

only if: Let $\mathcal{C}$ have a cover of size K, say $\{c_{i_1}, c_{i_2}, \ldots, c_{i_K}\}$. Construct a search tree for C with the corresponding literals as the initial part of the spine, attaching each a-literal a_i to the earliest c_j in T_C such that $a_i \in c_j$. The chain of $b_{i,k}$ literals, $2 \leq k \leq n$ is attached to a_i. Since $\{c_{i_1}, c_{i_2}, \ldots, c_{i_K}\}$ is a cover of $\mathcal{C}$, every $a_k \in S$ also occurs in some c_l, $l \in \{i_1, \ldots i_K\}$. Hence the longest root-to-leaf path length in the tree is $K + n$.

Since **Set Cover** is NP-hard and there is a polynomial time reduction of **Set Cover** to **Min-ST** (as outlined above), **Min-ST** is NP-hard. ■