BOTTOM UP PROCEDURES TO CONSTRUCT EACH MINIMAL CLAUSE TREE ONCE

by

J.D. Horton and Bruce Spencer

TR97-115, July 1997

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
E-mail: fcs@unb.ca
www: http://www.cs.unb.ca

Bottom up Procedures to Construct each Minimal Clause Tree Once

J. D. Horton and Bruce Spencer

Faculty of Computer Science, University of New Brunswick
P.O. Box 4400, Fredericton, New Brunswick, Canada E3B 5A3
jdh@unb.ca, bspencer@unb.ca, http://www.cs.unb.ca

Abstract. Bottom up procedures are introduced for constructing only minimal clause trees. Two general methods are proposed. The first performs surgery on each non-minimal clause tree; the second ignores non-minimal clause trees and retains only minimal ones. By ranking open leaves and activating/deactivating leaves according to the ranking, the second method constructs each minimal clause tree exactly once. Methods for incorporating subsumption are described and a prototype system for propositional logic is compared against OTTER.

1 Introduction

Clause trees, introduced in [4], provide a framework for understanding the internal structure of binary resolution proofs, and a data structure that leads to implementations that exploit the new ideas. The important notion of a *minimal* clause tree extends the usual restrictions for a resolution derivation: specifically to avoid tautological clauses and to merge (factor) identical literals in a clause. In a non-minimal clause tree, tautologies or missed opportunities for merging that are hidden, can be identified easily (efficient algorithms exist). Given a non-minimal clause tree, the operation of *surgery* takes the tree apart and puts some of the pieces back together to form a new, smaller clause tree that does not contain the tautology, or does the merge that was missed. Alternately, since a minimal clause tree must exist, a clause tree procedure can avoid deriving any non-minimal trees. Many derivations correspond to one clause tree, so this allows a procedure to avoid many derivations.

Top down procedures for building (minimal) clause trees are presented in [3,4]. Bottom up procedures are described in this paper. Section 2 informally introduces clause trees, merge paths, tautology paths, derivations of clause trees, and the notion of a hidden tautology or missed merge. The formal definitions follow in Sections 3 and 4.

Section 5 describes resolution procedures in terms of the sequence of clauses they generate. One procedure is said to subsume another if each clause generated by the first subsumes some clause generated by the other. For every binary resolution procedure there exists a clause tree procedure that subsumes it – one that converts any non-minimal trees it constructs, into minimal trees. Surgery is not used in the implementations reported in this paper.

The implementation we describe retains only minimal clause trees. Minimality is combined with another strong restriction, based on *rank* and *activity* of nodes in the clause tree. This restriction is new, but bears a resemblance to a well-known ordering restriction of resolution[5]. The new restriction combined with minimality guarantees not only completeness; it also guarantees that each minimal clause tree is constructed *exactly once*.

When the minimality and the rank/activity restrictions are combined with subsumption, completeness is lost. A new variant of subsumption for clause trees, *contracting* subsumption, can be used safely with the new restrictions, but it introduces redundancy. So we provide *strict increasing* subsumption which works with rank/activity and minimality, and still avoids redundancy. Experiments with a procedure that combines minimality, rank and activity, with strict increasing subsumption and contracting subsumption, reveal that it often requires fewer inferences than OTTER 3.0[6].

2 An intuitive exposition of clause trees

When one uses binary resolution, one starts with a set of clauses, each of which is the disjunction of a set of literals, and applies resolution to them until the clause that one wants is found. This clause is the empty clause if one is looking for a contradiction. Usually a clause is represented as a set of literals, but in this paper a clause is represented by a tree in graph theory terms. An input clause is represented by a clause node connected to atom nodes each of which is labeled by an atom. A $+(-)$ sign labels the edge joining the atom node to the clause node if the atom appears positively (negatively) in the clause. Figure 1(A) shows the tree representing the clause $\{a, b, \neg c, \neg d\}$. Such a tree is called a clause tree.

Clauses can be combined using resolution. For example, the clause $\{\neg b, \neg d, e\}$ can resolve with the clause $\{a, b, \neg c, \neg d\}$ on the atom b to form the clause $\{a, \neg c, \neg d, e\}$. Clause trees are resolved by identifying the nodes that represent complementary literals from two different clauses, as shown in Figure 1(B). The leaves of the resulting tree are the literals of the resulting clause. But in this case two of the leaves are labeled by $\neg d$. The merging of the two literals $\neg d$ that occurs in resolution is not handled automatically by the clause trees. Instead the two atom nodes that correspond to the same literal can be joined with a merge path as in Figure 1(C). The literal at the tail of a merge path is no longer considered to be a literal of the corresponding clause. The operation of resolving two clause trees followed by the insertion of all leaf to leaf merge paths is analogous the usual binary resolution on clauses. A resolution between Figure 1(C) with the clause tree for the clause $\{d\}$ is done, resulting in the clause tree in Figure 1(D) whose clause is $\{a, \neg c, e\}$. Next $\{b, \neg e\}$ is resolved with Figure 1(D) to obtain the clause tree Figure 1(E), with clause $\{a, b, \neg c\}$.

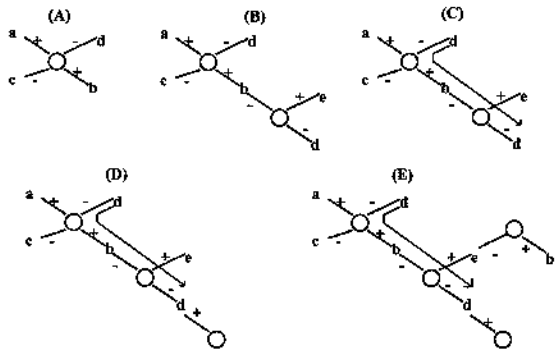

Fig. 1. Example Clause trees

Possibly many binary resolution proofs correspond to a single clause tree. The clause tree in Figure 1(E) also corresponds to the following two derivations:

1. $\{a, b, \neg c, \neg d\}$	1.$\{a, b, \neg c, \neg d\}$
2. $\{\neg b, \neg d, e\}$	2.$\{\neg b, \neg d, e\}$
3. $\{d\}$	3.$\{d\}$
4. $\{\neg e, b\}$	4.$\{\neg e, b\}$
5. $\{a, \neg c, \neg d, e\}$ 1. res 2. on b	5.$\{b, \neg b, \neg d\}$ 2. res 4. on e
6. $\{a, b, \neg c, \neg d\}$ 5. res 4. on e	6.$\{a, b, \neg c, \neg d\}$ 5. res 1. on b
7. $\{a, b, \neg c\}$ 6 res 3. on d	7.$\{a, b, \neg c\}$ 6. res 3. on d

Note that these two derivations are essentially the same, in that they compute same results and resolve the same pairs of literals in the input clauses. The only difference is the order in which the resolutions are done. The second derivation contains a tautology, clause 5. In a sense, the first derivation contains a "hidden tautology". The clause tree in Figure 1(E) represents this tautology by the existence of a legal path between the two b atom nodes, which starts and ends with edges of different signs. We call this a *tautology path*. Similarly the resolution on d might have been done before the merge on d was done. Then the resulting clause tree would have contained an unchosen legal merge path, a "missed merge." Such tautologies and missed merges can be avoided, or can be removed by surgery.

3 Definitions

We use standard definitions from first order clausal logic [1] and graph theory[2].

The definition of clause tree in this paper differs from that in [4]. There the definition is procedural, in that operations that construct clause trees are given. Here, as in [3] the definition is structural.

Definition 1 (Clause Tree). $T = \langle N, E, L, M \rangle$ is a clause tree on a set S of input clauses if

1. $\langle N, E\rangle$ is an unrooted tree.
2. L is a labeling of the nodes and edges of the tree. $L : N \cup E \rightarrow S \cup A \cup \{+, -\}$, where A is the set of instances of atoms in S. Each node is labeled either by a clause in S and called a clause node, or by an atom in A and called an atom node. Each edge is labeled $+$ or $-$.
3. No atom node is incident with two edges labeled the same.
4. Each edge $e = \{a, c\}$ joins an atom node a and a clause node c; it is associated with the literal $L(e)L(a)$.
5. For each clause node c, $\{L(a,c)L(a) | \{a, c\} \in E\}$ is an instance of $L(c)$. A path $\langle v_0, e_1, v_1, \ldots, e_n, v_n\rangle$ where $0 \leq i \leq n$, $v_i \in N$ and $e_j \in E$ where $1 < j < n$ is a *merge path* if $L(e_1)L(v_0) = L(e_n)L(v_n)$. Path $\langle v_0, \ldots, v_n\rangle$ *precedes* ($\prec$) path $\langle w_0, \ldots, w_m\rangle$ if $v_n = w_i$ for some $i = 1, \ldots, m-1$.
6. M is the set of merge paths called *chosen* merge paths such that:
   (a) the tail of each is a leaf (called a *closed* leaf),
   (b) the tails are all distinct and different from the heads, and
   (c) the relation $\prec$ on M can be extended to a partial order.

A set M of paths in a clause tree is *legal* if the $\prec$ relation on M can be extended to a partial order. A path P is *legal in* $T = \langle N, E, L, M\rangle$ if $M \cup \{P\}$ is legal. If the path joining t to h is legal in T, we say that h is *visible* from t. A path $\langle v_0, e_1, v_1, \ldots, e_n, v_n\rangle$ where $v_i \in N$ and $e_j \in E$ is a *tautology path* if $L(v_0) = L(v_n)$ and $L(e_1) \neq L(e_n)$. A path is a *unifiable tautology path* if $L(e_1) \neq L(e_n)$ and there exists a substitution θ such that $L(v_0)\theta = L(v_n)\theta$. A path is a *unifiable merge path* if there exists a substitution θ such that $L(e_1)L(v_0)\theta = L(e_n)L(v_n)\theta$.

A clause tree with a single clause node is said to be *elementary*. An *open leaf* is an atom node leaf that is not the tail of any chosen merge path. The disjunction of the literals at the open leaves of a clause tree T is called the *clause* of T, $cl(T)$.

Definition 2 (Minimal clause tree). A clause tree $\langle N, E, L, M\rangle$ is *minimal* if it contains no legal merge path not in M and no legal tautology path.

There are various operations on clause trees: creating an elementary clause tree from an input clause, resolving two clause trees, adding a merge path to the set of chosen paths and instantiation. Each of these operations results in a clause tree.

Operation 3 (Creating an elementary clause tree).
Given a clause C in S and a substitution θ for variables in C, the elementary clause tree $T = \langle N, E, L, \phi\rangle$ representing $C\theta = \{a_1, \ldots, a_n\}$ is the following:

1. N consists of a clause node and n atom nodes, where L labels the atom nodes with $a_1, \ldots, a_n$ and labels the clause node with C.
2. E consists of n undirected edges, each of which joins the clause node to one of the atom nodes and is labeled by L positively or negatively according to whether the atom is positive or negative in the clause.

Operation 4 (Resolving two mergeless clause trees).
Let $T_1 = \langle N_1, E_1, L_1, M_1 \rangle$ and $T_2 = \langle N_2, E_2, L_2, M_2 \rangle$ be two clause trees with no nodes in common such that n_1 is an atom node leaf of T_1 and n_2 is an atom node leaf of T_2. No variable may occur in a label of both an atom node in T_1 and an atom node in T_2. Let L_1 label n_1 with some atom a_1 and label the edge $\{n_1, m_1\}$ negatively, and L_2 label n_2 with the atom a_2 but label the edge $\{n_2, m_2\}$ positively. Further let a_1 and a_2 be unifiable with a substitution θ. Let $N = N_1 \cup N_2 - \{n_1\}$. Let $E = E_1 \cup E_2 - \{\{n_1, m_1\}\} \cup \{\{n_2, m_1\}\}$ where $\{n_2, m_1\}$ is a new edge. Let L be a new labeling relation that results from two modifications to $L_1 \cup L_2$: the new edge $\{n_2, m_1\}$ is labeled negatively, and θ is applied to the label of each atom node. Let M be the set of merge paths that results from replacing in $M_1 \cup M_2$ each occurrence of n_1 with n_2. Then $T = \langle N, E, L, M \rangle$ is a clause tree.

We write T_1 *res* T_2 to refer to the clause tree that results. We use a similar notation for resolving two clauses together.

Operation 5 (Adding a leaf to leaf unifiable merge path).
Let $T = \langle N, E, L, M \rangle$ and let n_1 and n_2 be two open leaves in T such that $P = path(n_1, n_2)$ is a unifiable merge path of $\langle N, E, L, \phi \rangle$, with n_2 not being the tail of any chosen merge path in M and n_1 not being the head or tail of any chosen merge path. Let θ be a substitution such that $L(n_1)\theta = L(n_2)\theta$. Let $L\theta$ be the labeling relation that results from applying θ to the label of each atom node, and otherwise leaving L the same. Then $T_1 = \langle N, E, L\theta, M \cup \{P\} \rangle$ is a clause tree.

Operation 6 (Instance of a clause tree).
A clause tree $T' = \langle N, E, L', M \rangle$ is an instance of a clause tree $T = \langle N, E, L, M \rangle$ if L' and L are identical on the clause nodes and edges, and there is a substitution θ such that for each atom node n, $L'(n) = L(n)\theta$.

Theorem 7 (Closure of Clause Tree Operations). *Each of Operations 3, 4, 5 and 6 generates a clause tree.*

4 Clause Tree Derivations

Definition 8 (Derivation of a clause tree). Given a set S of clauses, a *derivation* of T_n from S is a sequence $\langle T_1, \ldots, T_n \rangle$ of clause trees such that each T_i for $i = 1, \ldots, n$ (exactly) one of 1(a), 1(b), 1(c) or 1(d) holds and 2 holds.

1. (a) T_i is an elementary clause tree from an input clause C in S
   (b) T_i is the result of resolving T_i and T_j where $j < i$ and $k < i$. In this case T_i *depends* on T_j and T_k.
   (c) T_i is the result of choosing a leaf to leaf merge path in T_j where $j < i$. In this case T_i *depends on* T_j.
   (d) T_i is an instance of T_j for $j < i$. In this case T_i *depends on* T_j.

2. T_n transitively depends on T_i for $i = 1, \ldots, n$.

Because T_n depends on all previous T_i a derivation must not have unused clause trees. This condition does not change the essential nature of clause trees, but is added in order to make certain conditions easier to ensure.

Definition 9 (Admissible derivation). A derivation $\langle T_1, \ldots, T_n \rangle$ where $T_i = \langle N_i, E_i, L_i, M_i \rangle$ is *admissible* if for each $P \in M_n$, such that P is a path in T_i but $P \notin M_i$, then $P \in M_j$ for some $j > i$ and for all $i < k \leq j$ T_k depends on T_{k-1} and T_k is the not the result of resolving clause trees.

Thus a derivation is admissible if all leaf to leaf paths that appear in T_n are chosen as soon as possible in the derivation and before the next resolution step.

Theorem 10 (Existence of an admissible derivation). *Every clause tree has an admissible derivation.*

Proof. Let $T = \langle N, E, L, M \rangle$ be a clause tree with n_a atom nodes, n_c clause nodes, n_m merge paths and no open leaf atom nodes. For each clause node c in T, construct an elementary clause tree with atom nodes labeled the same as the atom nodes adjacent to c in T. Place these as the first n_c clause trees in the derivation. Thus all remaining clause trees in the derivation will be the result of applications of Operation 4 or 5. Extend $\prec$ to a total order on M. Process the paths $P_1, \ldots, P_{n_m}$ in this order. For each internal atom node of the path P_i not already considered, construct the clause tree corresponding to the resolution step involving this atom node on the two clause trees already in the derivation, and insert the result next into the derivation. After every step if any path P_j for $j \geq i$ is in the current clause tree but is unchosen, then insert a clause tree with that path chosen into the derivation before the next resolution step. Continue until all paths are processed and then do all remaining resolutions. The resulting derivation is admissible. □

Definition 11 (Minimal derivation). A derivation $\langle T_1, \ldots, T_n \rangle$ is *minimal* if, for any $i = 1, \ldots, n, T_i$ has two open leaves with the same label then a merge path joining these leaves is among the chosen merge paths in T_n, and for any other pair of atom nodes of T_i labeled the same, neither node is a leaf.

In particular, a derivation is minimal if it does not contain any clause tree T_i such that $cl(T_i)$ is a tautology. Also, if a leaf to leaf merge path appears, it is chosen.

Theorem 12 (Characterization of Minimal Clause Trees). *A clause tree is minimal if and only if all admissible derivations of it are minimal.*

Proof. Let clause tree T_n have a derivation $\langle T_1, \ldots, T_n \rangle$ which is admissible but not minimal, where $T_i = \langle N_i, E_i, L_i, M_i \rangle$. Then there is a clause tree T_i which has two identically labeled leaves n_1 and n_2 but the path P joining these leaves is not in M_n. Construct the derivation that chooses P after step i as follows. Let $T_i' =$

$\langle N_i, E_i, L_i, M_i \cup \{P\}\rangle$, and for $k = i, \ldots, n$, let $T'_k = \langle N_k, E_k, L_k, M_k \cup \{P\}\rangle$ if T_k depends on T_i, and $T'_k = T_k$ otherwise. Then the sequence $\langle T_1, \ldots, T_i, T'_i, \ldots, T'_n\rangle$ is a derivation. But T'_n has $M_n \cup \{P\}$ as its set of chosen paths, which implies that P is legal in T_n.

Conversely, assume that $T = \langle N, E, L, M\rangle$ is non-minimal. Then there is a legal path $P \notin M$ between atom nodes in T. Let $P_1, \ldots, P_i, P, P_{i+1}, \ldots, P_n$ be an ordering of $M \cup \{P\}$ that is an extension of the precedes relation. Then construct an admissible derivation as follows. Build a derivation of T as in the construction in Theorem 10 according to this sequence except do not insert the path P. Note that the tree in the derivation in which P could be first added as a merge path breaks the minimality condition on the admissible derivation. □

5 Comparing resolution-based procedures

Consider any binary resolution-procedure, A, working with a set I of input clauses. A produces the clauses $C = \langle C_1, C_2, \ldots, C_i, \ldots\rangle$ where

1. $C_i \in I$,
2. $C_i = C_j$ res C_k where $j < i$ and $k < i$, or
3. $C_j\theta \subseteq C_i$ where $j < i$ and θ is a substitution.

This sequence is called the *generated* clause sequence of A on input I. A refutation procedure A stops with success if $C_i = \phi$ for some i. Note that the generated sequences determine the procedure if they are known for all inputs.

Let the generated clause sequence of the procedure B on input clauses I be $\mathcal{D} = \langle D_1, D_2, \ldots, D_i, \ldots\rangle$. B is said to *subsume* A on I if for all i there is an $f(i)$, $f(i) \leq i$ and $D_{f(i)}$ subsumes C_i. Procedure B *strictly subsumes* procedure A if B subsumes A and A does not subsume B.

Lemma 13. *Let $\mathcal{C} = \langle C_1, C_2, \ldots, C_m, \ldots\rangle$ be a generated clause sequence. If $\mathcal{D} = \langle D_1, D_2, \ldots, D_n\rangle$ is a generated clause sequence on the same input clauses, of finite length, $n \leq m$ and $\mathcal{D}$ subsumes $\langle C_1, ..., C_m\rangle$ then $\mathcal{D}$ can be extended to a sequence which subsumes $\mathcal{C}$ and this extension is either an infinite sequence or contains ϕ.*

Proof. Suppose $D_{f(i)}$ subsumes C_i, $f(i) < i$. Then for each $C_k, k > m$,

1. if C_k an input clause, it can be replaced in $\mathcal{D}$ by itself.
2. if $C_k = C_j\theta$, $j < k$, C_k is subsumed by $D_{f(j)}$.
3. if $C_k = C_i$ res C_j, then C_k is subsumed by one of $D_{f(i)}, D_{f(j)}$ or $(D_{f(i)}$ res $D_{f(j)})$ when the resolution is on the instance of the literal resolved upon in C_i res C_j.

□

Thus there is no need to ever consider a generated clause sequence $\mathcal{C}$ in which step 3 (substitution) of the definition is used, since omitting from $\mathcal{C}$ the clause tree C_i, which is an instance of C_j for $j < i$, leaves a sequence $\mathcal{C}'$ and an

extension of C' subsumes C. Similarly it is pointless ever to do a resolution that does not use a most general unifier of the literals being resolved. From now on we consider only procedures that use most general unifiers and do not generate clause sequence using the substitution step.

Any resolution based procedure can be considered to be a clause tree based procedure, by substituting for each resolution of clauses, the equivalent resolution of clause trees. Thus we can define a clause tree based procedure on the set of input clauses I as the admissible derivation $\langle T_1, T_2, \ldots, T_i, \ldots \rangle$ in which 3(substitution) is not used. $\langle cl(T_1), cl(T_2), \ldots, cl(T_i), \ldots, \rangle$ is an equivalent resolution based procedure.

Clause trees also allow the operations of surgery, including inserting merge paths. Thus we can replace step 2 in the definition of the generated clause sequence with:
$2'$. $T_i = T_j$ res T_k, where $j < i$ and $k < i$. If T_i is non-minimal then generate some minimal clause tree that results from surgery step on T_i.

If the surgery step in $2'$ is ever performed and removes an open leaf, then the resulting sequence of clauses strictly subsumes the original sequence of clauses. However the sequence of clauses may no longer be a resolution based sequence, because the clause resulting from surgery may not be derivable from the preceding sequence in one step.

Theorem 14 (A subsuming minimal clause tree procedure exists). *Any resolution based procedure can be subsumed by a clause tree based procedure that performs surgery. Moreover the subsumption can be made strict unless the equivalent clause trees are all minimal.*

6 Procedures that retain only minimal clause trees

We need to produce only minimal clause trees, which can be produced using minimal derivations (Theorem 12). But a minimal clause tree is more restrictive than one which has a minimal derivation – *all* admissible derivations of it must be minimal. Although this condition seems to be computationally expensive, in fact one does not need to generate and test all derivations. When two minimal clause trees T_1 and T_2 are resolved, the resulting tree T can be non-minimal only if there is a merge or tautology path going from T_1 to T_2, or from T_2 to T_1. The only allowable merge/tautology paths are leaf to leaf merge paths. Any tautology path causes T to be non-minimal, as does any new legal merge path from an internal node to a leaf, or from an internal node to an internal node.

To implement the minimality check, one needs to construct the following for any clause tree T. Each of these can be done with an algorithm which is linear in the size of the node set of T.

1. The literals at the open leaves, $cl(T)$, and the corresponding set of atoms, $atom(T)$.
2. The atoms labeling internal nodes, $int(T)$.
3. The atoms visible from (outside) a given open leaf L, $vis(T, L)$.

For T_1 and T_2 to be resolved on open leaves L_1 of T_1 and L_2 of T_2, such that a minimal clause tree results, the following are necessary, sufficient and easily checked.

1. $(atom(T_1) \cup vis(T_1, L_1)) \cap int(T_2) = \phi$;
2. $(atom(T_2) \cup vis(T_2, L_2)) \cap int(T_1) = \phi$;
3. $cl(T_1 \text{ res } T_2)$ is not a tautology.

The procedures retain only the clause trees that pass this test.

7 Completeness and uniqueness

It is well known that resolution can be refined by imposing a preference ordering on the literals, and selecting only the most preferred literal from a clause as the literal to be resolved upon[5]. This restriction can greatly enhance the efficiency of resolution. It is natural to ask if this restriction interferes with the minimal restriction of resolution. In fact it does, as the following example shows.

Suppose the clause set $\{\{\neg p\}, \{p, \neg a\}\{p, a\}\}$ is given, and p is preferred over a. Then the only refutation will resolve the first clause against the second, resulting in $\{\neg a\}$ and the first against the third resulting in $\{a\}$, and finally resolve these two results to give $\{\}$. However the corresponding clause tree is not minimal, as there is a unchosen merge path between the p atom nodes.

We introduce the *rank/activity* restriction. For each clause tree, each leaf is given a *rank*. For an elementary clause tree with n edges, the ranks are from 1 to n. Consider two clause trees T_1 and T_2 to be resolved. The rank ordering on the leaves of T_1 res T_2 must agree with the rank orderings of T_1 and of T_2. One way to do this follows. Assume, without loss of generality, that T_1 resolves on a positive literal. For an atom node of T_1 res T_2 which is an open leaf in T_1, its rank in T_1 res T_2 is the same as its rank in T_1. For an atom node in T_2, its rank in T_1 res T_2 is its rank in T_2 increased by the number of edges in T_1. Thus the open leaves of any clause tree have ranks between 1 and the number of edges of that clause tree.

All open leaves of elementary clause trees are active. When two clause trees T_1 and T_2 are resolved, then for $i = 1, 2$ the leaves of T_i that have rank lower than the leaf resolved in T_i are deactivated or become inactive. An inactive leaf is not allowed to be resolved upon. It is reactivated only if it is the head of a chosen leaf to leaf merge path. Note that if all open leaves of a clause tree become inactive, then the clause tree can never be used in a future resolution. Hence it does not need to be retained, except perhaps to be used in subsumption. It is, however, still a minimal clause tree on the set of input clauses.

The following two theorems assume that the proof procedure has a fairness criterion: if some pair of retained clause trees can be resolved on some atom, then that resolution will be done after only a finite number of intervening resolutions – it is not delayed indefinitely. Furthermore, that resolution will be done only once by the procedure. Such a procedure is said to have a *fair selection strategy*.

A further assumption is made to accommodate first-order logic. For any clause tree that is retained, including the input clauses, for each unifiable leaf to leaf merge path whose substitution is non-empty, two clause trees must be retained. One clause tree has the substitution applied and the merge path chosen, while the other has the merge path not chosen. Such a procedure is said to *non-deterministically choose unifiable leaf to leaf merge paths.* In fact an implementation might not need this redundancy, but we defer discussing alternate strategies for implementing this non-determinism until we complete experiments involving first-order problems.

Theorem 15 (Completeness). *Any minimal clause tree procedure that has a fair selection strategy, non-determistically chooses unifiable leaf to leaf merge paths, and uses the rank/activity restriction is refutationally complete.*

Proof. Let T be any minimal clause tree on the set I of input clauses. If T is an elementary clause tree, it is produced by the procedure at initialization. Otherwise assume that T has n clause nodes. Assume as an induction hypothesis that any minimal clause tree with fewer than n nodes on any set of input clauses is constructed by the procedure given that set of input clauses.

Now T has $n-1$ internal atom nodes. At each clause node v define $p(v)$ to be the internal atom node adjacent to v that is not the head of a merge path that passes over v, and is of lowest rank in the elementary clause tree of v. The clause node v is said to point at $p(v)$. The node $p(v)$ must exist for if all of the atom nodes adjacent to v were the heads of chosen merge paths passing over v, each would be preceded by some other nodes adjacent to v which would imply a cycle in the precedes relation. Because there are n clause nodes and only $n-1$ internal atom nodes, some atom node w is pointed at by both of its neighbouring clause nodes, say v and u. Let T_v be the elementary clause tree consisting of v and its adjacent atom nodes, and similarly let T_u consist of u and its adjacent atom nodes. Let $T_1 = T_v$ res T_u on node w. T_1 is constructed by the procedure because of the fair selection strategy.

Let $I \cup \{cl(T_1)\}$ be a new set of input clauses in which the ranks of the elementary clause trees for I are the same as were derived from the procedure when T_1 is produced. There may be more than one elementary clause tree to choose for T_1 because there can be leaf to leaf merge paths produced non-deterministically when T_v and T_u are resolved, but recall that all of these are included among the initial elementary clause trees. Let T' be the clause tree T with the nodes u, v and w replaced by the appropriate elementary clause tree for $cl(T_1)$. Then T' is constructed by the procedure on the input clauses $I \cup \{cl(T_1)\}$. Now if one removes all the resolutions in which elementary clause tree from $cl(T_1)$ or its descendants are resolved, one gets exactly the clauses produced by the procedure acting on I, and T is among these. □

All the above proof shows is that at any stage in the construction of T there is always one more resolution that can be done towards the construction of T.

Theorem 16 (Uniqueness). *Any clause tree procedure that has a fair selection strategy and nondetermistically chooses unifiable leaf to leaf merge paths, and uses the rank/activity restriction produces each minimal clause tree exactly once.*

Proof. Let T be any minimal clause tree on a set I of input clauses. If T were an elementary clause tree, then it is produced only at the initial stage. Thus we can assume T is not elementary.

We show by induction that T must contain a unique interior atom node which must be resolved after all other interior atom nodes in T. This is clearly true if T has only one interior atom node. As in the proof of Theorem 15 for each clause node v in T define $p(v)$ as the atom node adjacent to v with the lowest rank in the clause of v that is not the head of a chosen merge path that passes over v.

We argue that $p(v)$ must be resolved before any other node x adjacent to v. If x is not the head of a merge path that includes v, then $rank(x) > rank(p(v))$. Then $p(v)$ must be resolved first because if x were resolved first, $p(v)$ would become inactive. Since $p(v)$ could never be reactivated it would remain inactive and could never be resolved. If x is the head of a merge path, then it is preceded by some of the other nodes adjacent to v. At least one of these nodes, y, must not be the head of a merge path, for otherwise some subset of these nodes would precede each other. Naturally y must be resolved before x because it precedes x. But either $p(v) = y$ or $p(v)$ must be resolved before y as previously argued. Hence $p(v)$ must be resolved before x. As in the proof of Theorem 15, let w be an atom node pointed at by each of its neighbours, v and u. Let $T_1 = T_u$ res T_v, and let T' be T with u, v and w replaced by a single clause node corresponding to $cl(T_1)$. Then T' is a minimal clause tree on $I \cup \{cl(T_1)\}$, with one fewer interior atom node than T. By the induction hypothesis, T' has a unique interior atom node z which must be resolved later than any other interior atom node in T'. The only node that is different in T' from T is w. But w must be resolved before any of the leaves of T_1, which themselves must be resolved before z.

Therefore T can be produced by the procedure only when the atom node z is resolved. Hence $T = T_2$ res T_3. By another induction, on the size of the clause tree for example, both T_2 and T_3 are constructed exactly once by the procedure. Hence T is also constructed exactly once by the procedure. □

8 Subsumption

Non-minimality and activity both are properties that prevent the construction of clause trees that would be removed by subsumption. Subsumption can be an expensive check because it depends on the set of retained clauses, which may be large. Wos[8] refers to this:

> If a strategy could be found whose use prevented a reasoning program from deducing redundant clauses, we would have a solution far preferable to our current one of using subsumption.

Minimality and rank/activity provide a partial solution, but do not remove every subsumed clause. For instance there may be redundancy in the input clause set, so that the same clause is derived from different input clauses. The question is, can the restrictions of minimality and rank/activity be used in conjunction with subsumption? The answer is only partially. These different techniques interfere with each other, and lose completeness. In Figure 2, the fair selection strategy is guaranteed by constructing all clause trees of one clause node, of two clause nodes, *et cetera*. Ranks are indicated by numbers in superscript, and an inactive node is denoted by $*$.

1.	$p^1 + \bullet$		
2.	$a^1 - \bullet - b^2$	input clause subsumed by 7.	
3.	$a^2 + \bullet - p^1$	input clause subsumed by 6.	
4.	$b^2 + \bullet - p^1$	input clause subsumed by 5.	
5.	$b^3 + \bullet - p + \bullet$	1. res 4.	
6.	$a^3 + \bullet - p + \bullet$	1. res 3.	
7.	$a^* - \bullet - b + \bullet - p + \bullet$	2. res 5.	inactive
8.	$b^5 - \bullet - a + \bullet - p + \bullet$	2. res 6.	

No more minimal resolutions are possible.

Fig. 2. Subsumption interacts with minimality and rank/activity

We can extend any bottom up clause tree procedure to use subsumption fully and maintain completeness, at the cost of losing some of the advantages of minimality and activity. Whenever a clause tree T subsumes a clause tree T', remove both T and T', and replace both with the elementary clause tree of a new input clause $cl(T)$. The ranks of the leaves of this clause tree are assigned as for any input clause. All of these leaves must be deemed active. We call such elementary clause trees *contracted* and we call this subsumption *contracting subsumption*. Figure 3 shows the same example as Figure 2, but using contracting subsumption.

1.	$p^1 + \bullet$	input clause		
2.	$a^1 - \bullet - b^2$	input clause	subsumed by 8.	
3.	$a^2 + \bullet - p^1$	input clause	subsumed by 6.	
4.	$b^2 + \bullet - p^1$	input clause	subsumed by 5.	
5.	$b^3 + \bullet - p + \bullet$	1. res 4.	becomes $b^1 + \bullet$, subsumes 4.	
6.	$a^3 + \bullet - p + \bullet$	1. res 3.	becomes $a^1 + \bullet$, subsumes 3.	
7.	$a^* - \bullet - b + \bullet$	2. res 5.	inactive, can be ignored	
8.	$b^3 - \bullet - a + \bullet$	2. res 6.	becomes $b^1 - \bullet$, subsumes 2.	
9.	$\bullet - b + \bullet$	5. res 8.	Done.	

Fig. 3. Contracting subsumption works with minimality and rank/activity

Contracting subsumption retains completeness, but each contracted clause tree it introduces has lost the internal structure that allows non-minimality to be detected, and all leaves in the new clauses are active. Thus the derivations are no longer unique. However, we can avoid some of this redundancy, because not all subsumptions need a contracting clause. We compute a size for each clause tree. If the subsumed clause is bigger than the subsuming clause, the subsumed clause can be safely rejected, just as the usual form of subsumption. Only if the subsumed clause is the same size or smaller, do we replace it by a contracted clause.

Definition 17 (Strict Increasing Subsumption). A clause tree T *subsumes* a clause tree $T*$ if $cl(T)$ subsumes $cl(T*)$. It is called *strict increasing subsumption* if $size(T) < size(T*)$.

There are various size functions that could be used: the number of clause nodes, the number of edges, the height of the tree, the total size where each clause node is given a weight, *et cetera*. For a given problem and selection strategy, different weight functions would give different proportions of contracting subsumptions.

Definition 18 (Properties of size functions). A size function is *consistent* if $size(T_1) < size(T_2)$ implies that $size(T_1 \text{ res } T) < size(T_2 \text{ res } T)$. We say that a size function is *stable* if, for each clause tree T, all admissible derivations of T agree on the size of T. A size function is *increasing* if $size(T_1 \text{ res } T_2) > max(size(T_1), size(T_2))$. It is *additive* if $size(T_1 \text{ res } T_2) = size(T_1) + size(T_2)$.

Theorem 19 (Completeness with all properties). *A minimal clause tree procedure is given that uses a fair selection strategy, nondeterministically chooses unifiable leaf to leaf merge paths, uses the rank/activity restriction, uses increasing subsumption and contracting subsumption otherwise, where the size function is increasing, consistent and stable. Then this procedure is refutationally complete.*

Proof. We prove that for any minimal clause tree T, there exists a minimal clause tree T' that is constructed by the procedure such that

1. $ct(T')$ subsumes $cl(T)$
2. $size(T') \leq size(T)$
3. If $size(T') = size(T)$ then the open leaves of T' correspond to open leaves of T and the rank orderings on these leaves agree. Moreover, if an open leaf of T is active and corresponds to an open leaf of T', that leaf of T' is active.

We assume here that the rank ordering is the example given in Section 7. Let T be the smallest minimal clause tree for which such a T' has not been constructed by the procedure. Assume that the input clauses do not subsume each other. Then T is not elementary. By Theorem 15, $T = A \text{ res } B$ on atom a where a is active in both A and B. As the size function is increasing, both $size(A)$ and

$size(B)$ are less then $size(T)$. Thus the procedure constructs clause trees A' and B' which satisfy the three induction conditions. If a is not in $cl(A')$ then A' will fulfill the role of T'. Similarly for B'. Thus we can assume that a is in $cl(A')$ and in $cl(B')$. Consider $T'' = A'$ res B'. If $size(A') < size(A)$ or $size(B') < size(B)$ then $size(T'') < size(T)$ by consistency of the size function. Then by induction there exists T' which satisfies the three conditions for T'' and hence will satisfy them for T. Assume $size(A') = size(A)$ and $size(B') = size(B)$. Then since a is active in A and in B, a is active in A' and B' by the third condition. Then A' res B' is either constructed because of the fair selection strategy, or not constructed either because it is nonminimal, subsumed by strict increasing subsumption, or subsumed by a contracted clause. If it is nonminimal, by surgery there exists a minimal clause tree T' which subsumes T and $size(T') < size(A'$ res $B') = size(T)$. The size is strictly less because the function is increasing. If it is subsumed by strict increasing subsumption then the subsuming clause satisfies the three conditions. If it subsumed by a contracted clause then that contracted clause must be defined to satisfy the conditions. The only remaining case is that $T' = A'$ res B' is constructed. Then T' subsumes T and $size(T') = size(T)$. To satisfy the third condition, note that any open leaf x of T must have been an open leaf x of A or B. Suppose x is a leaf of A. If x corresponds to a leaf $c(x)$ of A', $c(x)$ is a leaf of A' res B' and x as a leaf of T corresponds to $c(x)$ as a leaf of T'. This gives a natural correspondence between open leaves of T and open leaves of T' if they exist. The rank orderings of the leaves of A' and B' must agree with the rank orderings of the leaves of A and B by induction. Hence the rank orderings of the leaves of T' must agree with the rank orderings of the leaves of T . Moreover, the active leaves of A and B correspond to active leaves of A' and B' if they exist, so the active leaves of T must correspond to active leaves of T' if they exist. □

9 Experiments

Problem	OTTER inferences	Number of Clause Trees	Problem	OTTER inferences	Number of Clause Trees
GRA001-1	244	75	PUZ015-1	143	144
MSC007-1.003	91	45	PUZ030-2	2258	1669
MSC007-1.004	35820	577	SYN001-1.005	107	160
PUZ013-1	33	57	SYN010-1.005:005	216	536
PUZ014-1	79	82	SYN090-1.008	99	149
SYN094-1.005	10091	359	SYN098-1.002	127	85

We have implemented a clause tree procedure for propositional logic combining the properties of Theorem 19 with an OTTER-like algorithm based on the set of support strategy. The size function is the number of edges. This is approximately comparable to OTTER configured for binary resolution, forward and backward subsumption, and the propositional flag, although the same sequence of inferences was not attempted by both procedures. We ran each on a

number of propositional problems from TPTP[7], and the inference counts are reported above for the non-trivial problems.

10 Conclusions

Clause trees are an efficient concept to use in a resolution based automated reasoning procedure. Any resolution-based procedure is subsumed by one using surgery to produce only minimal clause trees (Theorem 14). Complete bottom up procedures exist that retain only minimal clause trees. By ranking the open leaves and activating only some of them, a procedure can construct each minimal clause tree exactly once (Theorem 16).

The concepts of minimality and rank/activity are in one sense only instances of subsumption, in that they are no more restrictive than subsumption. However, subsumption is more expensive to check than either minimality or activity. Full subsumption combined with these concepts in a theorem prover is incomplete. However a combination of increasing subsumption and contracting subsumption together with minimality and rank/activity is complete (Theorem 19) In the near future we expect to have an efficient implementation of a first-order procedure based on these concepts. A preliminary propositional implementation is encouraging in that it often uses fewer inferences that OTTER.

Acknowledgments

Joel Burrows wrote the theorem prover; NSERC provided funding.

References

1. Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York and London, 1973.
2. F. Harary. *Graph Theory.* Addison-Wesley, Reading, Mass., 1969.
3. J. D. Horton and B. Spencer. A top down algorithm to find only minimal clause trees. In L. Dreschler-Fischer and S. Pribbenow, editors, *KI-95 Activities: Workshops, Posters, Demos*, pages 79–80. Gesellschaft f r Informatik e.V. Bonn, September 1995.
4. J. D. Horton and B. Spencer. Clause trees: a tool for understanding and implementing resolution in automated reasoning. *Artificial Intelligence*, to appear summer 1997. *http://www.cs.unb.ca/profs/bspencer/htm/clause_trees/TR95 – 95.ps.*
5. R. Kowalski and P.J. Hayes. Semantic trees in automated theorem proving. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 4.* American Elsevier Publishing Company, Inc., 1969.
6. W. W. McCune. Otter 3.0 users guide. Technical Report ANL-94/6, Mathematics and Computer Science Division, Argonne National Laboratories, Argonne, IL, 1994.
7. G. Sutcliffe, C. Suttner, and T. Yemenis. The TPTP problem library. In D. Kapur, editor, *Automated Deduction CADE-12*, number 814 in Lecture Notes in Artificial Intelligence, pages 252–266. Springer-Verlag, Berlin, 1994.
8. Larry Wos. *Automated Reasoning : 33 Basic Research Problems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.