BOTTOM-UP PROCEDURES FOR MINIMAL CLAUSE TREES

by

J.D. Horton and Bruce Spencer

TR96-101, January 1996

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

Bottom-up procedures for minimal clause trees

J. D. Horton and Bruce Spencer
Faculty of Computer Science
University of New Brunswick
P.O. Box 4400
Fredericton, New Brunswick, Canada
E3B 5A3
phone 506-453-4566
jdh@unb.ca, bspencer@unb.ca, http://www.cs.unb.ca
Area: Mechanisms: resolution

Abstract

Clause trees provide a basis for reasoning procedures that use binary resolution. In this paper binary resolution trees are used as the data structure for implementing these procedures. Several properties of these procedures are explored: size, stages, minimality, disequalities, activity and rank. Procedures for uniquely building all minimal clause trees are introduced. Ordinary subsumption within these procedures does not preserve completeness, but contracting subsumption, a new restricted form of subsumption that relates to clause trees, can be integrated. The minimal restiction and unique construction address the problem of redundancy with binary resolution.

1. Introduction

A clause tree [1] is an abstract data structure which represents a proof of a resulting clause, using resolution [5] on a set of input clauses. One clause tree represents possibly many such proofs; two proofs which consist of the same resolution steps but done in a different order are represented by the same clause tree. A clause tree does not specify the order in which resolutions are done, except when necessary, to allow a merge (or factoring) of two different occurrences of a literal (or unifiable literals).

A minimal clause tree is one in which no reordering of the resolutions can produce either a merge that was not applied, or a tautology. If a merge is missed, or a tautology created in a resolution-based proof, then a shorter resolution proof exists and its result subsumes the

original result. Finding such a smaller proof corresponds to a process of removing pieces from the clause tree, called clause tree surgery [1]. Thus any procedure that resolves pairs of clause trees to construct all minimal clause trees on a set of input clauses is refutationally complete for first order logic. Three classes of such procedures have been proposed [3].

Two different clause trees can represent the same proof if the proof contains a merge; a merge path is reversed. Such clause trees are called ***reversal equivalent***. However we can restrict all clause trees so that only one of any set of reversal equivalent trees is generated. For example, Spencer's foothold restriction can be applied [6]. Such a clause tree is said to be ***representative***. The procedures mentioned above can be restricted to produce only representative minimal clause trees.

Only the most restrictive, and probably the most efficient of the three classes of procedures is considered in this paper. In this class of procedures, if a resolution is going to produce a non-minimal tree, the resolution is not allowed. In Section 3 we describe a data structure that allows one to determine quickly whether the new clause tree is minimal.

In Section 4 we describe properties for clause tree procedures. Besides minimality, we show how to restrict the set of permissible resolutions further, so that no representative minimal clause tree is produced more than once. At any point in the procedure, literals are in one of two states: active or inactive. An inactive literal is not allowed to resolve. The proofs of completeness and uniqueness are in Section 5. By greatly reducing the number of clauses produced, this technique reduces the use and hence the cost of subsumption.

Subsumption cannot be simply integrated into our procedures. However in Section 6 we define contracting subsumption, a form of subsumption that relates only to clause trees. It does not preserve minimality or uniqueness, but we beleive that a system combining all these properties will improve our ability to perform binary resolution.

2. Definitions

A ***path*** $path(v_0,v_n)$ in a graph from v_0 to v_n is an alternating sequence $\langle v_0\ e_1\ v_1\ \ldots\ e_n\ v_n\rangle$ where each v_i is a node and each e_i is an edge. The first node v_0 is the ***tail*** of the path, and the final node v_n is the ***head***.

Definition *1 $T=\langle N,E,L,M\rangle$ is a* ***clause tree*** *on a set S of input clauses if:*

(a) $\langle N,E\rangle$ is a(n unrooted) tree.

(b) L is a labeling of the nodes and edges of the tree. $L: N \cup E \rightarrow S^* \cup A \cup \{+,-\}$, where S^* is the set of instances of clauses in S and 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 −.*

(c) No atom node is incident with two edges labeled the same.

(d) Each edge $e=\{a,c\}$ joins an atom node a and a clause node c; it is ***associated*** *with the literal $L(e)L(a)$.*

(e) For each clause node c, $L(c) = \{L(\{a,c\})L(a) \mid \{a,c\} \in E\}$. A path $\langle v_0\ e_1\ v_1 \ldots e_n\ v_n\rangle$ where $0 \le i \le n$, $v_i \in N$ and $e_j \in E$ where $1 \le j \le n$ 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 $i=1,\ldots, m-1$.*

(f) M is the set of merge paths called ***chosen merge paths*** *such that:*

(i) the tail of each is a leaf (called a ***closed leaf****),*

(ii) the tails are all distinct and different from the heads, and

(iii)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)***.

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 1. Creating an elementary clause tree from an input clause

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* $C\theta = \{a_1,...,a_n\}$ *satisfies the following:*

1) *N consists of a clause node and n atom nodes, where L labels the atom nodes with* $a_1,...,a_n$ *and labels the clause node with* $C\theta$.
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 2. Resolving two 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* $M_1 \cup M_2$ *by replacing each occurrence of* n_1 *in each path of* M_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 from Operation 2. We use a similar notation for resolving two clauses together.

Operation 3. 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 4. 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, atom nodes and edges, and there is a substitution θ such that for each atom node n, $L'(n) = (L(n))\theta$.*

Theorem 1. [1] ***Closure of Clause Tree Operations***

Each of Operation 1, Operation 2, Operation 3 and Operation 4 applied to a clause tree(s) generates a clause tree.

A sequence of these operations which results in a single clause tree is called a ***derivation*** of that clause tree. See [2] for the complete definition or a derivation.

Theorem 2. [1]***Soundness and completeness of clause trees***

Let C be a clause and S be a set of clauses. Then $S \models C$ iff there is a clause tree T on S such that $cl(T) \subseteq C$.

Definition 2 *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.*

A clause tree that is not minimal can be made minimal by applying surgery on all legal tautology and legal unchosen merge paths. Surgery is an operation that involves cutting out parts of the tree, if necessary, and rearranging the remainder, possibly adding a new merge path, so that the resulting structure is a clause tree. Surgery is discussed in [1].

3. Binary resolution trees

In this section we apply the ideas developed using clause trees to a well-known data structure: a binary tree of clauses generated by binary resolution. As will be seen, derivations, visibility and minimality carry over in a natural way. This exercise illustrates that these concepts are fundamental to binary resolution. A prototype is bein impleemnted that uses a variation of binary resolution trees, called binary resolution dags (directed acyclic graphs) which allow structure sharing to enhance space efficiency.

Definition 3. *A* ***binary resolution tree*** *on an input set S of clauses is a tree with the following properties:*

- *Each directed edge points from a* ***parent*** *node to a* ***child*** *node.*
- *Each node has either zero or two parents. A node with zero parents is called a* ***leaf*** *and one with two parents is called an* ***internal node.***

- *Each node is labeled with a clause called its **clause label.** A leaf is labeled with a clause from S, whereas an internal node is labeled with the resolvent of the clause labels of its parents. Each internal node also has an **atom label**, which is the atom of the literal resolved upon. This node is the **resolution node** for this atom, and for the occurences of the literals resolved upon.*

The ***root*** of a binary resolution tree is the internal node with no descendants, which must be unique if the tree is connected. The clause label of the root is the ***result*** of the proof consisting of the resolution steps in the tree.

Every occurrence of a literal in a clause label has a history. The ***history path*** for an occurrence of a literal starts at a leaf whose clause label contains that literal, goes through its descendants and ends at either the root of the binary resolution tree or at the resolution node for this occurence. A ***merge node*** for a literal *a* in a binary resolution tree is a node whose parents both have *a* in their clause labels. A ***merge tree*** for an occurrence of a literal *a* in a binary resolution tree is the union of all initial segments of the history paths of *a* from a leaf to the last merge node for *a* on each path.

In Figure 1, a binary resolution tree is shown with its clause labels. Atom labels of internal nodes are also shown, to the left of the colon. The merge tree for *a* is circled with a dashed line, and the resolution node for *a* is underlined with a dashed line. The merge tree for *f* is circled with a dotted line and the resolution node for *f*, the root, is underlined with a dotted line.

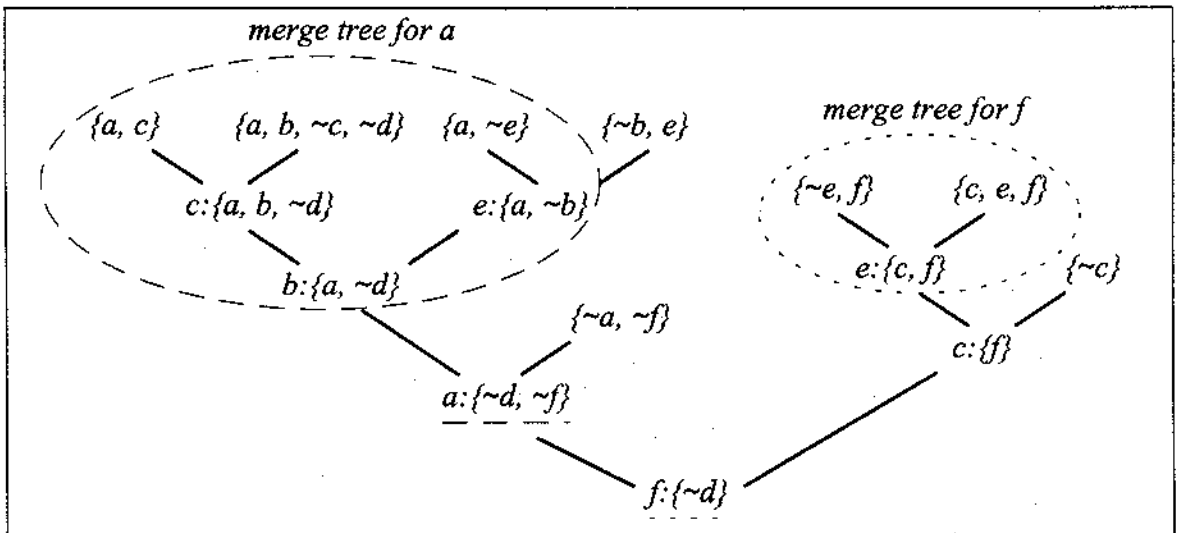

Figure 1 A non-minimal resolution tree

In a binary resolution tree, a node u can ***see*** a node v unless there exists a resolution node for an occurrence of a literal a on the path from u to v, and v is in the merge tree for that occurrence of a. We say that v is ***visible from*** u when u can see v. A binary resolution tree is ***non-minimal*** if there exist two nodes that have the same atom label and one can see the other, or if there is a literal in the clause label of the root and its atom appears as the atom label of some node in the tree. Otherwise the tree is ***minimal.*** In Figure 1, there are two nodes with the atom label e and neither one can see the other. There are also two nodes with the atom label c. Since the one on the left can see the one on the right, this tree is non-minimal.

Theorem 3. *B is a binary resolution tree and T is a corresponding clause tree. B is a minimal binary resolution tree iff T is a minimal clause tree.*

Proof Omitted. □

A non-minimal binary resolution tree T_1 is redundant in that there exists a smaller binary resolution tree T_2 on the same set of input clauses such that the clause label of the root of T_2 is a subset of an instance of the clause label of the root of T_1. Thus it is necessary only to construct minimal binary resolution trees. In Figure 2, the same set of clauses are used as in Figure 1, but the result of the tree in Figure 2 is $\{\}$, which subsumes the result $\{\sim d\}$ from Figure 1. The tree in Figure 2 is easily seen to be minimal since no two atom nodes are labeled the same and there are no literals in the clause label of the root.

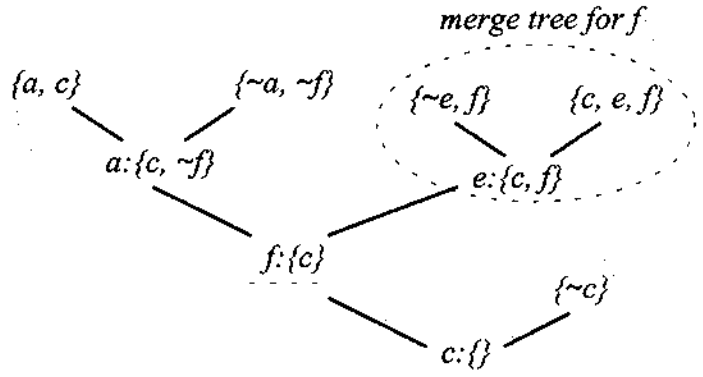

Figure 2 A minimal binary resolution tree

There is a natural construction of a binary resolution tree S from a given clause tree T. For each clause node in T construct a leaf node for S with this clause as its label. Extend the partial order on the internal atom nodes of T to a total order, $A_1,\ldots,A_n$. Consider the atom nodes in this order. For each A_i create a new internal node B_i for S and use the label

of A_i as its atom label. A_i represents the attachment of two clause trees whose atom nodes have already been considered. The clause of each of these clause trees is the clause label of a node already inserted into S. Label B_i with the clause that results from resolving these two clauses, and make it the child of these two nodes in S. When this process finishes, S is a binary resolution tree and its root is labeled with the clause $cl(T)$.

Given a binary resolution tree S, a corresponding clause tree T can be constructed. Order the nodes of the S so that for all nodes, all of its ancestors precede it. A postorder traversal suffices. Let $B_1,\ldots,B_n$ be this ordering. If B_i is a leaf, construct T_i the elementary clause tree of the input clause label of B_i. If B_i is an internal node, then the clause trees for both of its parents have already been built. Construct a new clause tree T_i by a clause tree resolution operation on these clause trees. If any literals from the parents of B_i are merged by the clause resolution operation at B_i then both of these identical literals correspond to edges of open leaves in T_i. Join these open leaves with a merge path. When this procedure finishes, $cl(T_n)$ is the clause label of B_n.

Note that the correspondence between binary resolution trees and clause trees is many to many. However the mapping from binary resolution trees to a representative clause tree is many to one. The procedures introduced in this paper construct one binary resolution tree for each clause tree. Thus for our purposes, the relation is one-to-one.

As examples, clause trees that correspond to the binary resolution trees in Figure 1 and Figure 2 are shown in Figure 3.

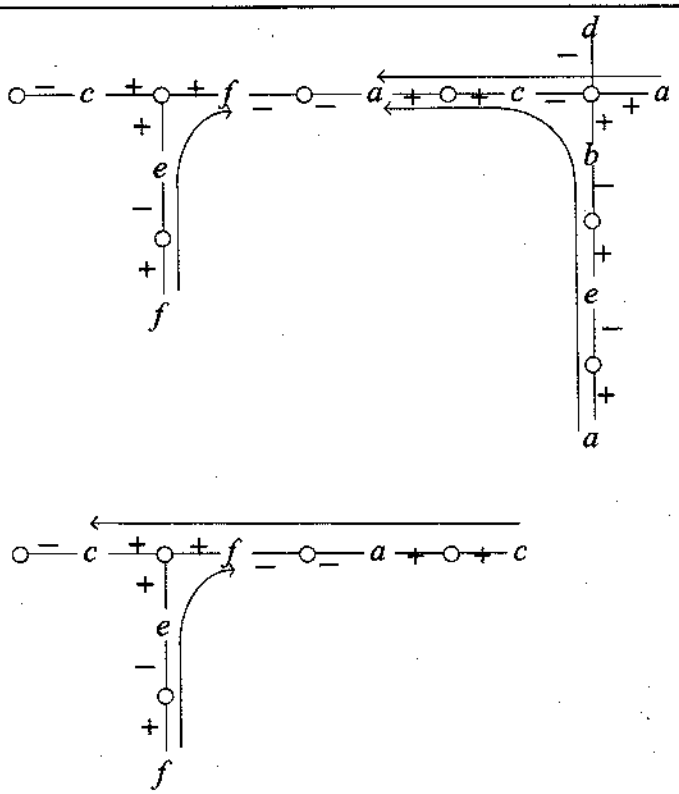

Figure 3 Clause trees corresponding to Figure 1 and Figure 2

4. Properties

The procedures we discuss in this paper have many properties in common which are described in this section.

4.1 Size and stages

In the procedures considered here, each binary resolution tree has an integer associated with it, called its ***size.*** The size function is used to control the procedure. We say that a size function is ***stable*** if, for each clause tree T, all binary resolution trees that correspond to T have the same size. In this case the size of T , written as ***size(T)*** is defined as the size of any of these binary resolution trees.

A size function is ***increasing*** if $size(T_1 \text{ res } T_2) > max(size(T_1), size(T_2))$. It is ***additive*** if $size(T_1 \text{ res } T_2) = size(T_1) + size(T_2)$. It is ***superadditive*** if $size(T_1 \text{ res } T_2) > size(T_1) + size(T_2)$.

We propose some examples of possible size functions.

a) the height of T.

$$size(T) = \begin{cases} 1 & \text{if } T \text{ is an elementary clause tree,} \\ 1 + max(size(T_1),size(T_2)) & \text{if } T = T_1 \; res \; T_2 \end{cases}$$

Example (a) gives the minimum possible size function that is increasing.

b) the number of clause nodes of T.

$$size(T) = \begin{cases} 1 & \text{if } T \text{ is an elementary clause tree,} \\ size(T_1)+size(T_2) & \text{if } T = T_1 \; res \; T_2 \end{cases}$$

c) the number of edges in T

$$size(T) = \begin{cases} size\ of\ input\ clause & \text{if } T \text{ is an elementary clause tree,} \\ size(T_1)+size(T_2) & \text{if } T = T_1 \; res \; T_2 \end{cases}$$

d) the weight of the clause tree. All input clauses are assigned a weight.

$$size(T) = \begin{cases} weight\ of\ input\ clause & \text{if } T \text{ is an elementary clause tree,} \\ size(T_1)+size(T_2) & \text{if } T = T_1 \; res \; T_2 \end{cases}$$

Example (d) generalizes both (b) and (c).

e) A subset S of the input clauses is specified and all clauses are given a weight.

$$size(T) = \begin{cases} weight\ of\ input\ clause & \text{if } T \text{ is an elementary clause tree,} \\ size(T_1)+size(T_2) & \text{if } T = T_1 \; res \; T_2, \text{ and no clause from } S \text{ occurs in } T \\ 1 + max(size(T_1),size(T_2)) & \text{otherwise} \end{cases}$$

Example (e) is similar to a set of support restriction, but small clause trees without support are allowed to be used in resolutions. A strict set of support restriction is not compatible with the minimal clause tree restriction because it is not always possible to build a given minimal clause tree from a given set of support such that all intermediate trees are minimal.

f) the number of predicate symbols, function symbols and variables in all the atom node labels of T.

Example (f) is perhaps too strict and penalizes complicated substitutions more than is desirable. It is also expensive to calculate. Thus we could weaken it to

g) $$size(T) = \begin{cases} number\ of\ symbols\ in\ the\ clause & \text{if } T \text{ is an elementary clause tree,} \\ size(T_1)+size(T_2)+increase\ in\ the & \text{if } T = T_1\ res\ T_2 \\ number\ of\ symbols\ in\ the\ open\ leaves & \end{cases}$$

h) $size(T) = size(cl(T))$ a size function that is commonly used

In the above list, only (h) is not increasing; (a), (g) and (h) are not stable. We can avoid the problem of an unstable size function in several ways. First we could define the size of a clause tree as the minimum over all the size of all of its derivations. This gives a unique size function, but still all derivations must be considered when calculating it. The second alternative, which is what we choose here, is to use procedures that allow each clause tree to be derived in only one way. Thus the size becomes dependent on the procedure used, which in turn is dependent on the size function.

Lemma 4. *For any increasing size function, there are a finite number of clause trees of size i.*

Proof Use induction on the size of the clause tree to show that any clause tree of size i has at most 2^{i-1} clause nodes. □

The procedure proceeds through a series of stages in which many resolutions are performed. Initially all distinct, most general factors of the input clauses are constructed. In stage i, all the clause trees of size i are constructed. This can be accomplished at the very least by resolving all clause trees of size less than i with all clause trees of size less than i. If the resulting clause tree is of size i, it belongs in stage i. Otherwise one can throw it away. Of course one can do better. If a produced clause tree has size $j>i$ it should be stored and considered at stage j. Secondly if all clauses of size less than $i-1$ have already been computed, one need only resolve those of size $i-1$ with all clause trees of size less than i to produce clause trees of size greater than or equal to i. This must be done if the size function is height, for example. If the size function is additive or superadditive, as (b), (c), (d), (f) and (g) are, one can resolve all clause trees of size k with clause trees of size $i-k$ for $k = 1,\ldots,\lfloor i/2 \rfloor$. This set of resolutions is used in Section **Error! Reference source not found.**.

4.2 Minimality and disequalities

Only minimal clause trees are constructed. It is not necessary to construct the clause tree to recognize that it is non-minimal. Assume that T_1 is to be resolved with T_2 at open leaves n_1 and n_2 respectively. Let $vis(n_1)$ be the internal atoms visible from beyond n_1 in T_1, and let vis(n_2) be the internal atoms visible from beyond n_2 in T_2. Let int(T) be the set of all interior atom nodes in T, and let leaves(T) be the set of all open leaf atoms nodes in T. Then in propositional logic T_1 *res* T_2 is not minimal if and only if T_1 and T_2 are not minimal and

a) $(vis(n_1) \cup leaves(T_1)) \cap int(T_2) = \varnothing$

b) $(vis(n_2) \cup leaves(T_2)) \cap int(T_1) = \varnothing$

c) $cl(T_1)$ and $cl(T_2)$ have no complementary literals other than a and $\sim a$.

We must be able to calculate $vis(n)$ and $int(n)$ quickly in order for the above to be effective. The cost of these calculations is linear in the size of the trees involved. The intersections can be checked in linear time using hash tables. An alternative way to remove non-minimal results is to apply subsumption, but the cost of subsumption depends on the number of retained results, which may be large.

The above condition does not guarantee that T_1 *res* T_2 is minimal in first order logic. A path in T_1 or T_2 that was not a legal merge or tautology, could become one when one applies the substitution arising from the resolution. The following example shows this. A clause node is shown as • and an edge is represented by its label, either – or +. The clause tree

$$f(x) - \bullet + a - \bullet + f(c),$$

is minimal. When it is resolved with the clause tree

$$\bullet + f(c)$$

the clause tree

$$\bullet + f(c) - \bullet + a - \bullet + f(c)$$

is produced. This has a tautology path between the $f(c)$ atoms, and so is non-minimal.

To prevent such unifiable paths from becoming legal merge/tautology paths, one can keep a list of pairs of atoms that are the ends of the unifiable paths. Such unifiable paths can be identified when legal merge/tautology paths are checked for. Instead of checking pairs of atoms just for identity, also check for unifiability. If such a pair of unifiable atoms is found, add the pair to the list of ***disequalities*** for the clause tree.

When the clause trees T_1 and T_2 are to be resolved at the open leaves n_1 of T_1 and n_2 of T_2, the above check for minimality is done. If it fails there is no addition to the set of retained clauses. Otherwise the clause tree T_1 *res* T_2 is contructed and the most general unifying substitution of the labels of n_1 and n_2 is applied to all of the atom labels. If there are n unifiable leaf-to-leaf merge paths in T_1 *res* T_2, then consider each subset of these paths. For each subset, the substitutions are applied that make the unifiable merge paths into merge paths and choose all leaf-to-leaf merge paths. The number of distinct clause trees that result will be bounded by 2^n. For each of these, the list of disequalities is checked; if any disequality in T_1 or in T_2 now refers to a pair of atom nodes with identical labels, the clause tree is non-minimal and so is rejected. For each tree that remains the new set of disequalities is constructed. It contains the disequalities derived from T_1, those from T_2 and the new disequalities (u,v) where u and v are atom nodes with labels that are unifible but not equal and either

- $u \in vis(n_1) \cup leaves(T_1)$ and $v \in int(T_2)$,
- $u \in vis(n_2) \cup leaves(T_2)$ and $v \in int(T_1)$, *or*
- u is an open leaf of T_1 and v is an open leaf of T_1.

As the labels of a given disequality becomes further specialized, they may become impossible to unify, so they can be removed from the disequality list.

Even an elementary clause tree may have disequalities arising from its factors. For example the input clause $\{p(X,b), p(a, Y)\}$ has the factor $\{p(a,b)\}$. The factor corresponds to a single atom elementary clause tree, while the clause tree for the original input clause would have two atom nodes and the disequality between them. If X were later bound to c, this disequality could be discarded.

4.3 Rank and activity of open leaves

For each clause tree, each leaf is given an arbitrary distinct ***rank***. For an elementary clause tree of n edges, the ranks are from 1 to n. Consider two clause trees T_1 and T_2 to be resolved and assume, without loss of generality, that T_1 is considered first. 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 will become ***reactivated*** if it is the head of a 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.

5. Completeness and uniqueness

Any procedure that is defined using the properties of Section 4 with an increasing size function is refutationally complete. Since all unsatisfiable set of clauses admit a complete minimal clause tree [1] we need to prove only that any minimal clause tree is reversal equivalent to some tree constructed by the procedure. In addition we show that any minimal clause tree is constructed only once, which shows that these procedures are efficient.

Theorem 5. Completeness
Any procedure that is defined using the properties of Section 4 with an increasing size function, 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 $T_l = T_v$ *res* T_u on node w. T_l is constructed by the procedure by stage $size(T_l)$.

Let $I \cup cl(T_l)$ be a new set of input clauses in which the size and ranks of the elementary clause trees for I are the same as were derived from the procedure when T_l is produced. We define the size of any clause tree S which contains a clause node of $cl(T_l)$ to equal the size of the clause tree with a clause node of $cl(T_l)$ replaced by the subtree T_l. There may be more than one elementary clause tree to choose for T_l because there can be merge paths produced non-deterministically when T_v and T_u are resolved, but recall that all of these 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 $cl(T_l)$. Then T' is constructed by the procedure on the input clauses $I \cup cl(T_l)$. Thus T and T' are produced in the same stage, because they are the same size. Now if one removes all the resolutions in which elementary clause tree from $cl(T_l)$ 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 6. Uniqueness

Any procedure that is defined using the properties of Section 4 with an increasing size function, 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 5 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 5, 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, both T_2 and T_3 are constructed exactly once by the procedure. Hence T is also constructed exactly once by the procedure.□

6. Subsumption

Non-minimality and activity both are properties that prevent the construction of clause trees that would be removed by subsumption. Subsumption is an expensive check because it depends on the set of retained clauses, which may be large. Non-minimality and activivty depend only on the size of the clause tree. The question is, can these procedures be used in conjunction with subsumption? The answer is, only partially. These different techniques that are trying to do the same thing can interfere with each other. An example is given in Figure 4: In this example, size is defined as the number of clause nodes, activities are indicated by numbers in superscript, and an inactive node is denoted by *. The procedure builds all clause trees of size i at stage i. Thus stage 1 shows the input set of clauses.

Stage 1

$^1p+\bullet$

$^1a-\bullet-b^2$ subsumed in stage 3

$a+\bullet-p^1$ subsumed in stage 2

$^2b+\bullet-p^1$ subsumed in stage 2

Stage 2

$^2b+\bullet-p+\bullet$

$^2a+\bullet-p+\bullet$

Stage 3

$^*a-\bullet-b+\bullet-p+\bullet$

$^2b-\bullet-a+\bullet-p+\bullet$

No more non-minimal resolutions are possible

Figure 4 Example showing incompleteness when using subsumption

Definition 4 A clause tree T ***subsumes*** a clause tree T^* if $cl(T)$ subsumes $cl(T^*)$. It is called ***strict forward subsumption*** if $size(T) < size(T^*)$ as well.

Definition 5 A size function is ***consistent*** if $size(T_1)<size(T_2)$ implies that $size(T_1 \text{ res } T) < size(T_2 \text{ res } T)$.

Theorem *7. Let A be a procedure which has the properties described in Section 4, such that A uses a size function that is stable, increasing and consistent, and A rejects clause trees that are subsumed by strict forward subsumption. Then A is refutationally complete.*

Proof. Let B be the same procedure as A, but without strict forward subsumption. B constructs all the clause trees the A constructs. If a newly constructed clause tree is subsumed by a retained clause tree, A will reject it, but B will retain it.

Let T be any minimal clause tree on the set of input clauses I. We wish to prove that some T' is constructed by A that subsumes T. T is constructed by B, by Theorem 5. If T is an elementary clause tree, then $T' = T$ is such a clause tree itself. Otherwise $T = T_1 \text{ res } T_2$ where a is the atom resolved on, and T_1 and T_2 are minimal clause trees constructed by B. We assume that a is the literal of an active node of T_1 and $\sim a$ is a literal of an active node of T_2, so that T is constructed in this way by B. We assume as an induction hypothesis that for any minimal clause tree T^* such that $size(T^*) < size(T)$, that there is a minimal clause tree T^{**} constructed by A which subsumes T^*.

The hypothesis applies to T_1 and T_2 because size is increasing, so there are clause trees T_3 and T_4 constructed by A such that T_3 subsumes T_1 and T_4 subsumes T_2, and $size(T_3) < size(T_1)$ unless $T_3=T_1$, and $size(T_4) < size(T_2)$ unless $T_4=T_2$. If both equalities are true, then either T is constructed by A because the number of clause trees of any given size is finite, or T is rejected because of strict forward subsumption. Thus we can assume that at least one of the equalities is false.

If a is not in $cl(T_3)$, then T_3 subsumes T, and $size(T_3) < size(T_1) < size(T)$. Similarly if $\sim a$ is not in $cl(T_4)$, T_4 satisfies the conditions for T'. Otherwise we can consider $T_5 = T_3$ *res* T_4, resolved on the atom a. Now T_5 is not necessarily constructed by either procedure, because a or $\sim a$ may be inactive, or because T_5 is not minimal. But T_5 must have a subtree that is a minimal clause tree, say T_6. Moreover, $size(T_6) \leq size(T_3 \text{ res } T_4) < size(T_1 \text{ res } T_2) = size(T)$. By the induction hypothesis, A must construct T_6 or a clause tree T' which subsumes T_6 and $size(T') < size(T_6)$. But then T' subsumes T and $size(T') < size(T)$. Hence A is refutationally complete.□

Any positive additive size function is increasing, stable and consistent, so that Theorem 7 applies with the size functions in Section 4.1 (b), (c) and (d). In addition one can in principle perturb the sizes of clause trees so that no two clause trees have the same size, so that strict forward subsumption corresponds to ordinary forward subsumption.

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 a new input clause $cl(T)$. Since this new clause was not used in previous stages of the algorithm, it must be processed in the way it would have been if it had been present from the beginning. Its size should be equal to the size of the smallest clause tree that it subsumes. A literal must be deemed active in the new elementary clause tree if the corresponding leaf in any clause tree subsumed by it, is active. Other literals can be deemed inactive. We call such elementary clause trees ***contracted*** and we call this subsumption ***contracting subsumption.*** Figure 5 shows the same example as Figure 4, but using contracting subsumption, and the refutation is found at stage 2.

Stage 1	
$^1p+\bullet$	
$^1a-\bullet-b^2$	subsumed in stage 3
$^2a+\bullet-p^1$	subsumed in stage 2
$^2b+\bullet-p^1$	subsumed in stage 2
Stage 2	
$^2b+\bullet-p+\bullet$	becomes $^1b+\bullet$ by contracting subsumption
$^2a+\bullet-p+\bullet$	becomes $^1a+\bullet$ by contracting subsumption
$^*a-\bullet-b+\bullet$	
$^2b-\bullet-a+\bullet$	becomes $^1b-\bullet$ by contracting subsumption
$\bullet+b-\bullet$	

Figure 5 Example from Figure 4 using contracting subsumption

If the size of the contracted clause tree were less than the size of every subsumed clause tree, then the number of clause trees of some given size may become infinite, and the completeness of the procedure may be lost. For example, consider the clauses *{~a}, {f(0)}, {~f(X),a,f(s(X))}* of weight 1, and *{b}, {~b}* of weight 2 with the size function (d) from Section 4.1. If the contracted clause trees all get weight 1, then the procedure goes into an infinite loop in stage 3, and never produces the proof of size 4. On the other hand if the size of the contracting clause tree T is larger than the size of the smallest subsumed clause tree T', then the smallest proof might use T' and the equivalent proof using T is now bigger than the smallest proof. In this case the search may be is prolonged.

Wherever a subsumed clause tree is used in a proof, a contracted clause tree can be used instead. Once a complete clause tree is found using the contracted clause tree, the original subsuming clause tree can replace it to form another complete clause tree. Thus the addition of subsumption does not affect the completeness of the procedure. The resulting complete clause tree is not necessarily minimal even if the original procedure produced only minimal clause trees, so uniqueness is lost. However the combined procedure should produce in most cases fewer resolutions than either the minimal active procedure, or the procedure without minimality or activity but with subsumption.

One can also note that one does not always need to use contracted clause trees in the case of strict forward subsumption. If the conditions of Theorem 7 are met, then strict forward

subsumption can be used without inserting a contracted clause tree, and the procedure remains complete. Only when another type subsumption is used, does one have to add a contracted clause tree to maintain completeness.

7. Conclusions

A family of bottom-up procedures are defined that produce each minimal clause tree exactly once. These procedures have been integrated with subsumption at the cost of losing uniqueness and minimality under some conditions. A prototype implementation is being developed. Preliminary experiments indicate that it often requires fewer inferences than OTTER using binary resolution and subsumption. In addition, it is quicker to check minimality and activity than to check for subsumption when the set of retained clauses is large.

The notion of a minimal clause tree has been expressed in terms of trees of clauses generated with binary resolution, a standard proof format. This addresses the problem with resolution of redundancy, Problem 6 [8].

The new technique of activity suggested here is very general and can be used with many of the standard bottom-up techniques to improve their efficiency. It can be used with clauses instead of clause trees, although the number of clauses produced will be greater than the number of minimal clause trees.

References

[1] J. D. Horton and B. Spencer, Clause trees: a tool for doing and understanding automated reasoning, TR95-095, Fac. Comp. Sc., Univ. New Brunswick, June 1995, available at http://www.cs.unb.ca/profs/bspencer/htm/clause_trees/TR95-095.ps.Z.

[2] J. D. Horton and Bruce Spencer, A top-down algorithm to find only minimal clause trees, Proceedings of CPL-95, held in conjunction with KI-95, Bielefeld, Germany, Sept.11-13, 1995, 77-78, available at http://www.cs.unb.ca/profs/bspencer/htm/clause_trees/ki95pr.ps.Z.

[3] J. D. Horton and Bruce Spencer, Reducing search with minimal clause trees,. TR95-099, Fac. Comp. Sc., Univ. New Brunswick, October 1995, available at http://www.cs.unb.ca/profs/bspencer/htm/clause_trees/TR95-099.ps.Z.

[4] W. W. McCune, OTTER 2.0 Users Guide, Technical Report ANL-90/9, Mathematics and Computer Science Division, Argonne National Laboratories, Argonne, IL (1990).

[5] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle *Journal of the ACM,* 12 (1965), 23–41.

[6] B. Spencer, Avoiding duplicate proofs with the foothold refinement, *Annals of Mathematics and Automated Reasoning,* 12 (1994) 117-140.

[7] G. Sutcliffe,C. Suttner, and T. Yemenis, The TPTP problem library, In *Proceedings of the International Conference on Automated Deduction,* 1994.

[8] Wos, L. *Automated Reasoning: 33 Basic Research Problems* (Prentice-Hall, 1988).