ELSEVIER

Artificial Intelligence 92 (1997) 25–89

Artificial Intelligence

Clause trees: a tool for understanding and implementing resolution in automated reasoning

J.D. Horton *, Bruce Spencer [1]

University of New Brunswick, Fredericton, New Brunswick, Canada E3B 5A3

Received July 1995; revised November 1996

Abstract

A new methodology/data structure, the clause tree, is developed for automated reasoning based on resolution in first order logic. A clause tree T on a set S of clauses is a 4-tuple $\langle N, E, L, M \rangle$, where N is a set of nodes, divided into clause nodes and atom nodes, E is a set of edges, each of which joins a clause node to an atom node, L is a labeling of $N \cup E$ which assigns to each clause node a clause of S, to each atom node an instance of an atom of some clause of S, and to each edge either $+$ or $-$. The edge joining a clause node to an atom node is labeled by the sign of the corresponding literal in the clause. A resolution is represented by unifying two atom nodes of different clause trees which represent complementary literals. The merge of two identical literals is represented by placing the path joining the two corresponding atom nodes into the set M of chosen merge paths. The tail of the merge path becomes a closed leaf, while the head remains an open leaf which can be resolved on. The clause $cl(T)$ that T represents is the set of literals corresponding to the labels of the open leaves modified by the signs of the incident edges. The fundamental purpose of a clause tree T is to show that $cl(T)$ can be derived from S using resolution.

Loveland's model elimination ME, the selected literal procedure SL, and Shostak's graph construction procedure GC are explained in a unified manner using clause trees. The condition required for choosing a merge path whose head is not a leaf is given. This allows a clause tree to be built in one way (the build ordering) but justified as a proof in another (the proof ordering).

The ordered clause set restriction and the foothold score restriction are explained using the operation on clause trees of merge path reversal. A new procedure called ALPOC, which combines ideas from ME, GC and Spencer's ordered clause set restriction (OC), to form a new procedure tighter than any of the top down procedures above, is developed and shown to be sound and complete.

* Corresponding author. E-mail: jdh@unb.ca.

[1] E-mail: bspencer@unb.ca.

PII S0004-3702(96)00046-X

Another operation on clause trees called surgery is defined, and used to define a minimal clause tree. Any non-minimal clause tree can be reduced to a minimal clause tree using surgery, thereby showing that non-minimal clause trees are redundant. A sound procedure MinALPOC that produces only minimal clause trees is given. Mergeless clause trees are shown to be equivalent to each of input resolution, unit resolution and relative Horn sets, thereby giving short proofs of some known results. Many other new proof procedures using clause trees are discussed briefly, leaving many open questions. © 1997 Elsevier Science B.V.

Keywords: Automated theorem proving; Redundancy; Minimality; Proof procedures

1. Introduction

This paper is concerned with automated reasoning using binary resolution. Starting with a set of clauses, each of which is the disjunction over a set of literals, one applies resolution to them until the clause that one wants is found. This clause is usually the empty clause, because the most common method starts by negating the result that one wants to prove, and then looking for a contradiction. 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. However the atom a labeling an atom node can correspond to either a positive or negative literal in the clause, a or $\sim a$, which is indicated by a + or − sign labeling the edge joining the atom node to the clause node. Fig. 1(a) shows the tree representing the clause $\{a, b, \sim c, \sim d\}$. Such a tree is called a *clause tree.*

Clauses can be combined using resolution. For example the clause $\{a, b, \sim c, \sim d\}$ can resolve with the clause $\{\sim b, \sim d, e, \sim g\}$ upon b to form the new clause $\{a, \sim c, \sim d, e, \sim g\}$. Clause trees also can be resolved, by identifying the leaf nodes that represent complementary literals from two different clauses, as shown in Fig. 1(b). The leaves of the resulting tree are the literals of the resulting clause. But two of the leaves are labeled by $\sim d$. The merging of the two literals $\sim d$ that occurs when the union of two sets occurs, is not handled automatically by clause trees. Instead the two atom nodes that correspond to the same literal can be joined with a *merge path* as in Fig. 1(c). The literal at the tail of a merge path is no longer considered to be a literal of the corresponding clause.

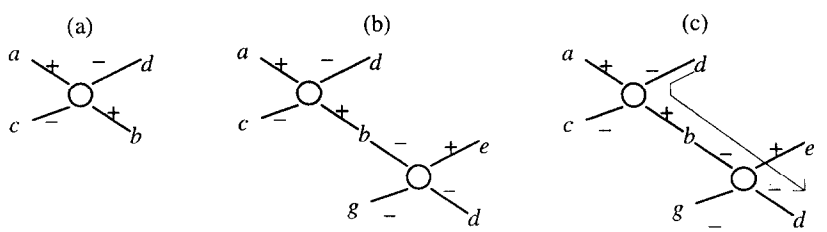

Fig. 1. Example clause trees.

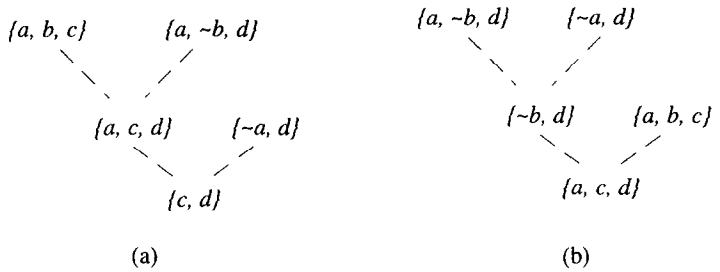

Fig. 2. Two different results from resolutions on the same clauses.

In ordinary resolution with clauses represented by sets, the order in which a sequence of resolutions is done can have a significant impact on the result. For example suppose we have the clauses $\{a, b, c\}$, $\{a, \sim b, d\}$, and $\{\sim a, d\}$. Resolving between the first two clauses on b produces $\{a, c, d\}$, and then resolving the result with the third clause on a produces $\{c, d\}$. See Fig. 2(a). However, if we begin by resolving between the second and third clause on a producing $\{\sim b, d\}$ and then resolve with the first clause on b, we obtain the inferior result $\{a, c, d\}$. See Fig. 2(b).

Fig. 3 shows the same sequences of resolutions as Fig. 2, but using clause trees instead of sets to represent the clauses. Leaves with the same label are merged as soon as they are connected by a path. However the tree on the bottom of Fig. 3(b) can be improved by choosing a merge path from the open leaf labeled a to the internal node with the same label. Thus the inferior result is improved to yield a clause tree whose clause is $\{c, d\}$. If clause trees are used it does not matter in which order the resolutions

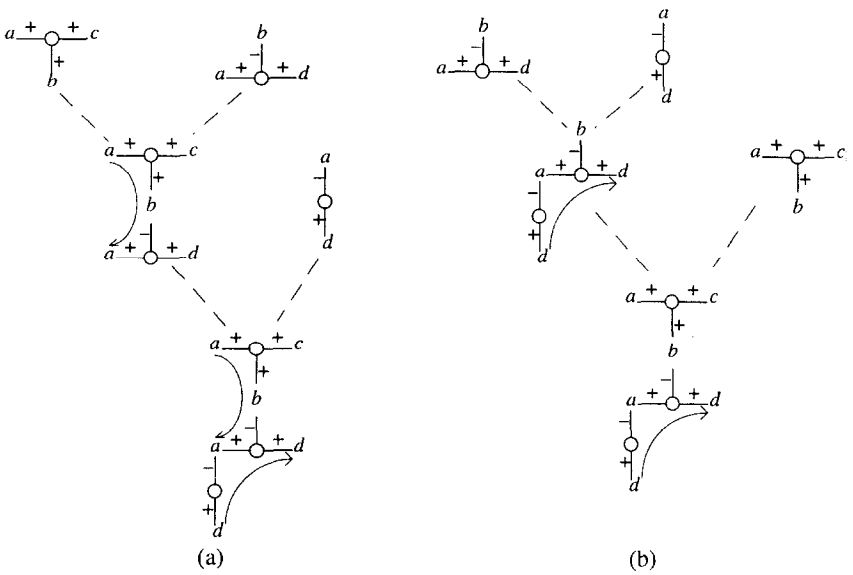

Fig. 3. Clause trees from Fig. 2.

are done, as the resulting clause tree can be made the same. Hence the clause tree corresponds to two different searches.

Fig. 3 illustrates how clause trees distinguish between the order in which the resolutions must be performed in an ordinary resolution proof (*proof ordering*), and the order in which the clause tree is constructed (*build ordering*). The final clause tree could be built using the build ordering of Fig. 3(b) but is justified by the proof ordering of Fig. 3(a). A more general example is presented in Section 3.1 where we present a way of looking at the (weak) model elimination procedure, ME, in which the proof ordering is different from the build ordering. In the proof ordering no resolutions are required with ancestor clauses, as would be required in the usual justification of the procedure. These ancestor resolutions are replaced by merges in proof ordering. In a manner similar to MESON [21], ancestor merges correspond to the insertion of merge paths in build ordering. These ideas are extended in Sections 3.3 and 3.4 to the selected literal SL procedure [16] and Shostak's graph construction GC procedure [29] respectively. Using clause trees, it is easily seen that GC is a restriction of a variant of SL, and that SL is a restriction of ME.

Section 4 develops the new concepts of visibility and support, which are relations between nodes of clause trees. The results of this paper rest on these relations. Section 5 shows that merge paths between nodes can be reversed, and that this does not essentially change the visibility relation. It also adapts the ordered clause set restriction [30,31] and the foothold score restriction [32] to clause trees.

A theorem prover can take advantage of the internal nodes of a clause tree in two basic ways. It can improve a proof, making its result more general. As shown in Fig. 3, an internal merge path can be chosen, to remove a literal. But more can be done. Most resolution-based proof procedures require that merges be done wherever possible and that tautologies be avoided. Since a clause tree represents several distinct proofs, we can detect if any one of those proofs contains a tautology or misses an opportunity to merge literals. The operation clause tree surgery, defined in Section 6, removes these internal tautologies and unused merges by cutting out some branches of the clause tree, and hence finds a smaller clause tree that subsumes the original. Clause trees to which surgery cannot be applied are said to be *minimal*.

Section 7 introduces ALP and AllPaths, top down procedures for building clause trees. Using clause trees, a theorem prover can avoid building the same clause tree that arises by reordering the resolution steps, and so can avoid redundant searching. For instance, each of these procedures in Section 7 can be further restricted by the ordered clause set restriction. Moreover, any clause tree that contains a tautology can also be avoided. As a stronger example of avoiding redundant search, the procedure Min-ALPOC produces only minimal clause trees.

Section 8 investigates clause trees without merge paths, and shows that a set of clauses admits a mergeless clause tree refutation, if and only if it admits an input refutation, if and only if it admits a unit refutation, and if and only if it contains a subset that is a relative Horn set that is unsatisfiable. Although it is well known that the latter three concepts are equivalent, the complete proofs that each is equivalent to the first concept provide examples of using clause trees to investigate and understand automated reasoning with resolution.

Our clause trees differ from the structures defined by Eisinger [8] with the same name, although the concepts are closely related.

Clause trees are closely related to connection tableaux [17]. Tableaux however are rooted trees, whereas clause trees are unrooted. The operation of tableau expansion corresponds to adding a clause tree with only one clause node to another clause tree. Clause trees can resolve with other clause trees that have more than one clause node, but this is not the case in tableaux. Choosing merge paths in a rooted clause tree corresponds to several different operations in tableaux: tableau reduction, folding up and folding down. In particular, we identify ancestor paths for reduction operations, left hooks for factoring (folding down) and left paths for c-reductions (folding up). Also all connection tableaux calculi defined in [17] are top down, whereas clause trees can be built bottom up in addition to top down [13,14].

2. Definitions

We use the standard definitions from first order clausal logic, as in [6] or [21]. An *atom* is an atomic formula. A *literal* is an atom a or the negation $\sim a$ of an atom. The *complement* $\sim b$ of a literal b is the negation of b if b is an atom, and the atom of b otherwise. A *clause* is a set of literals, possibly empty. All variables in a clause are assumed to be universally quantified. A *substitution* maps variables to terms. An *instance* of a literal or clause is the result of applying some substitution to its variables. A *ground instance* is one with no variables. A clause C_1 *subsumes* a clause C_2 if there is a substitution θ such that $C_1\theta \subseteq C_2$. A substitution θ *unifies* two literals b_1 and b_2 if $b_1\theta = b_2\theta$. A *most general unifier* θ of two literals b_1 and b_2 is one such that for each unifier σ of b_1 and b_2 there exists a substitution ρ such that $b_1\sigma = (b_2\theta)\rho$. An *interpretation* of a set of clauses consists of a non-empty domain D, an assignment to each n-place function symbol a mapping from D^n to D, and an assignment to each n-place predicate symbol a mapping from D^n to $\{true, false\}$. Given a set $S = \{C_1, \ldots, C_n\}$ of clauses and an interpretation I, I *satisfies* S if for every ground instance $S' = \{C'_1, \ldots, C'_n\}$ of S, some literal of each C'_i is mapped to *true* by I. A *model* of S is an interpretation that satisfies S. A set of clauses is *satisfiable* if there exists a model of it. S *entails* C, written $S \models C$, if every model of S is a model of C. In this paper, a *proof procedure* is a procedure that takes an input set S of clauses and an input clause C, and attempts to determine if C is *derived* from S, written $S \vdash C$. A proof procedure is *sound* if $S \vdash C$ implies $S \models C$. A proof procedure is *complete* if $S \models C$ implies there exists a clause D such that $S \vdash D$ and D subsumes C.

We also use standard terms from graph theory. A *graph* G consists of a set of nodes N, and a set E of unordered pairs of nodes called *edges*, written $G = \langle N, E \rangle$. An *edge* $e = \{v, u\}$ is *incident* with, or *joins* its endpoints v and u, and v and u are *adjacent* to each other as well as being *incident* with e. A node is of *degree* k if it is incident with exactly k edges. A *path* in a graph is an alternating sequence of nodes and edges such that adjacent elements in the sequence are incident with each other, no node appears twice, and the first and last elements are nodes. Thus $P = \langle v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k \rangle$

is a path if for all $i = 1, \ldots, k$, v_i is in N, $e_i = \{v_{i-1}, v_i\}$ is in E, and v_0 is in N. The first node v_0 is called the *tail* of the path, the last node v_k is called the *head* of the path, e_1 is called the *tail edge*, e_k is called the *head edge*, and the path is said to *join* or *connect* the tail to the head. The *length* is the number of edges. Note that a path is determined uniquely by its set of edges and its *orientation*, which is the direction that it runs. A *tree* is a graph in which there is a unique path joining any given ordered pair of nodes. We write *path*(t, h) for the path that joins node t to node h in a tree. The nodes of degree one in a tree are called the *leaves* of the tree.

A *rooted tree* is a tree with a specified node called the *root*. The nodes on the path from a given node to the root, excluding itself, are called the *ancestors* of that node. The second node on that path is called the *parent* of the first node, while the node itself is called a *child* of its parent. The children of a parent node are totally ordered; a child that comes earlier (later) in the order than a second child is said to be to the *left* (*right*) of the second child. A node n is *left* (*right*) of a node m if any ancestor of n is left (right) of any ancestor of m. The *level* of a node is the length of a path joining it to the root. As we shall see, clause trees by definition are unrooted trees with some additional structures, but while they are being constructed, the trees can be rooted.

We define clause trees in two steps. First we define mergeless clause trees and then clause trees with merge paths.

Definition 1 (*Mergeless clause tree*). Given a set S of clauses a *mergeless clause tree* T on S is a 4-tuple $\langle N, E, L, \phi \rangle$ where N is a set of atom nodes and clause nodes, E is a set of edges that each join an atom node to a clause node, and L is a labeling that maps each atom node to an atom, each clause node to a clause in S, and each edge to either $+$ or $-$. The graph $\langle N, E \rangle$ must be a tree. In addition, T conforms to either (a) or (b).

(a) (*Elementary mergeless clause tree*). Given a clause C in S and a substitution θ for variables in C, the *mergeless clause tree* $T = \langle N, E, L, \phi \rangle$ representing $C\theta = \{a_1, \ldots, a_n\}$ satisfies the following:
   (1) N consists of a clause node and n atom nodes, where L labels the atom nodes with the atoms of $a_1, \ldots, a_n$ 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.

(b) (*Resolving two mergeless clause trees*). Let $T_1 = \langle N_1, E_1, L_1, \phi \rangle$ and $T_2 = \langle N_2, E_2, L_2, \phi \rangle$ be two mergeless 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 both the label of an atom node in T_1 and the label of 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. Then $T = \langle N, E, L, \phi \rangle$ is a mergeless clause tree.

Definition 2 (*Unifiable merge and tautology paths*). Let $T = \langle N, E, L, \phi \rangle$ be a mergeless clause tree that contains two distinct atom nodes n_1 and n_2 such that the atoms that label these nodes are unifiable. A *unifiable path* P in T is the unique directed path from n_1 to n_2. If the tail and head edges of P both have the same sign, then P is a *unifiable merge path*. If the signs on the tail and head edges of P are different, then P is a *unifiable tautology path*. If the labels on n_1 and n_2 are identical and the signs on the tail and head edges are identical (respectively, opposite) then the unifiable path P is called a *merge path* (respectively, a *tautology path*).

A clause tree may have many merge paths, but not all of them need to be added to the set of chosen merge paths (the fourth parameter of the clause tree tuple). Those that do, remove a leaf from the set of open leaves. Note that a merge path is fully determined by its atom nodes and orientation, indeed simply by its head and tail.

Definition 3 (*Clause tree*). Given a set S of clauses, a *clause tree* T on S is a tuple $\langle N, E, L, M \rangle$, the set of nodes, edges, labels and chosen paths respectively, as defined by (a), (b) or (c).

(a) (*Elementary clause tree*). Given a clause C in S, and a substitution θ for variables in C, the mergeless clause tree $T = \langle N, E, L, \phi \rangle$ representing $C\theta$ is a clause tree.

(b) (*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 common nodes such that n_1 is an open leaf of T_1 and n_2 is an open leaf of T_2. Let $T' = \langle N, E, L, \phi \rangle$ be the mergeless clause tree resulting from resolving the two mergeless clause trees $\langle N_1, E_1, L_1, \phi \rangle$ and $\langle N_2, E_2, L_2, \phi \rangle$ at the nodes n_1 and n_2 as in Definition 1(b). 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.

(c) (*Choosing a 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 $\langle N, E, L, \phi \rangle$ using the substitution θ, with n_1 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 $L\theta$ 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. P is called a *chosen* merge path in T_1.

A path is a (*unifiable*) *merge or tautology path* in a clause tree $\langle N, E, L, M \rangle$ if it is a (unifiable) merge or tautology path in the mergeless clause tree $\langle N, E, L, \phi \rangle$. Note that a clause tree is based on an undirected graph (tree), but that a merge path is directed. Referring back to Fig. 1, we see that the three parts of the figure illustrate the three parts of Definition 3, respectively. We occasionally use T to refer to the tree $\langle N, E \rangle$ underlying the clause tree $T = \langle N, E, L, M \rangle$.

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

Where confusion does not arise we will use $T\theta$ for T' and $L\theta$ for L'.

Definition 5 (*Open leaf, associated literal, clause of a clause tree and subsumption of clause trees*). An *open leaf* is an atom node leaf that is not the tail of a chosen merge path. Each edge has an *associated literal* which is formed by the atom label of the incident atom node, modified by negation if the edge is labeled negatively. The *clause of a clause tree* T, written $cl(T)$, is the disjunction of literals associated with the open leaves. A clause tree T_1 *subsumes* a clause tree T_2 if $cl(T_1)$ subsumes $cl(T_2)$.

One restriction on chosen merge paths is that the head of one cannot be the tail of another. The purpose of a merge path is to indicate where a proof of the complement of the literal at the tail of a merge path can be found. If this restriction were lifted, there could be two paths based on the same edge set but oriented differently, in which the tail of one was the head of the other. Both ends would be closed, but neither would have a proof. Nor do we want two different proofs indicated for a single tail. One can imagine other definitions for chosen merge paths (such as removing the condition that n_1 in Definition 3(c) is not the tail of a path) which would sometimes allow the head of a path to be the tail of another but which would avoid this type of circular situation. However other definitions such as merge connected (see below) and path reversal (in Section 5) would become more complicated. We believe that the gain in generality is minimal, and is not needed for the development of better procedures or for the understanding of the present procedures.

Definition 6 (*Merge connected nodes*). In a clause tree T, two atom nodes are said to be *merge connected* if they are connected by a chosen merge path in either direction or if they are the tails of two chosen merge paths that have a common head. In addition every node is merge connected to itself.

Clearly atom nodes that are merge connected must be labeled by the same atom. The relation merge connected is an equivalence relation on the set of atom nodes in a clause tree. In each equivalence class there is a single node that is the head of the chosen merge paths. The remainder of the nodes in the equivalence class are closed leaves that are the tails of chosen merge paths with the same head. These tails break up naturally into two possibly empty sets: those that are incident with a negative edge and those incident with a positive edge.

Definition 7 (*Merge sets*). The *merge sets* of a clause tree are the equivalence classes defined by the merge connected relation. The *head node of a merge set* is the head of every path in the merge set, or the singleton node in the set if there are no merge paths.

See Fig. 4 for an example clause tree where the atom nodes 1, 2, 3, 4, 5 and 6 are labeled with a and are all merge connected. Node 4 is the head node of this merge set.

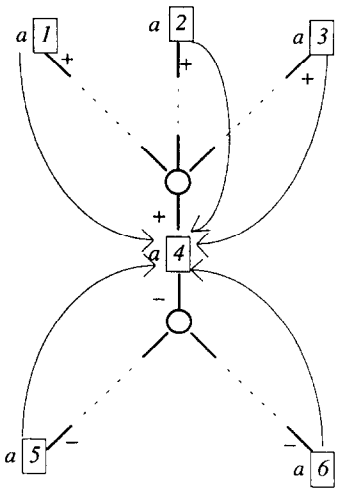

Fig. 4. {1, 2, 3, 4, 5, 6} is a merge set.

We enclose the name of an atom node in a box and the name of a clause node in a circle; a label is written either beside the node or replaces the node if the node is omitted from the figure.

Technically the clauses of the clause trees in Fig. 1(b) and (c) are the same since sets do not allow elements to occur multiple times. However the clause tree of Fig. 1(c) is more useful because there is only one open leaf for each literal in the clause. In Fig. 1(b) the literal $\sim d$ corresponds to two open leaves, whereas it corresponds to only one open leaf in Fig. 1(c). Generally one does not want a literal to correspond to multiple open leaves, so one usually chooses a merge path between open leaves as soon as it is possible to do so. However there are procedures that deliberately disobey this rule in the later sections of this paper.

Fig. 3 shows that merge paths do not have to go between two leaves, but can go from a leaf to an interior node. At first glance it is not apparent that Fig. 3(a) corresponds to a clause tree, because the heads of the merge paths are not open leaves. However it is easy to use the definition of clause tree recursively to construct it. More formally:

Definition 8 (*Derivation of a clause tree*). Given a set P of clauses, a *derivation* of T_n from P is a sequence $T_1, \ldots, T_n$ of clause trees such that each T_i for $i = 1, \ldots, n$ is the result of one of the following:

- an application of Definition 3(a) on a member of P,
- an application of Definition 3(b) on T_j and T_k where $j < i$ and $k < i$, or
- an application of Definition 3(c) on T_j where $j < i$.

Theorem 9. *Suppose that* $\langle N, E, L, \phi \rangle$ *is a mergeless clause tree. Then* $\langle N, E, L, M \rangle$ *is a clause tree if and only if it has a derivation.*

Proof. By induction. The definition of derivation mirrors the definition of clause tree. □

Theorem 10 *Let S be a set of clauses and C a clause. $S \models C$ if and only if there exists a clause tree T from S such that $cl(T)$ subsumes C.*

Proof. $S \models C$ if and only if there is a sequence of clauses $C_1, C_2, \ldots, C_n$ such that C_n subsumes C and each C_i is either in S or C_i is the result of resolving C_j and C_k, where $j < i$ and $k < i$. This is a slightly generalized restatement of Robinson's main result in [26]. Each member of the sequence C_i corresponds to either an application of Definition 3(a), or an application of Definition 3(b) followed by as many applications of Definition 3(c) as required (choosing merge paths between open leaves) so that no two open leaves are left that are labeled by the same atom. Conversely, the definition of a clause tree mirrors the definition of the derivation, so clearly $S \models cl(T)$. Further $cl(T) \models C$ since $cl(T)$ subsumes C. □

Definition 11 (*Closed clause tree*). A clause tree is said to be *closed* if it has no open leaves.

Corollary 12. *S is an unsatisfiable set of clauses if and only if there is a closed clause tree T from S.*

Corollary 13. *Any proof procedure that refutes a set of clauses by building a closed clause tree is sound.*

Definition 14 (*Isomorphic clause trees*). Two clause trees $T_1 = \langle N_1, E_1, L_1, M_1 \rangle$ and $T_2 = \langle N_2, E_2, L_2, M_2 \rangle$, defined on the same set of clauses S, are said to be *isomorphic* if there is a bijection $\Psi : N_1 \to N_2$ such that:

(a) $\{v, u\}$ is in E_1 iff $\Psi(\{v, u\}) \equiv^{\text{def}} \{\Psi(v), \Psi(u)\}$ is in E_2,

(b) $L_1(x) = L_2(\Psi(x))$ for all x in $N_1 \cup E_1$, and

(c) path $P = \langle v, e, \ldots, u \rangle$ is in M_1 iff $\Psi(P) \equiv^{\text{def}} \langle \Psi((v), \Psi(e), \ldots, \Psi(u) \rangle$ is in M_2.

3. Existing top down procedures restated in terms of clause trees

Well-known top down proof methodologies such as the weak model elimination procedure (ME), the selected literal procedure (SL) and graph construction (GC) can be translated into manipulating clause trees.

3.1. Weak model elimination

Consider the following example that proves the literal q from the clauses:

$C1 = \{q, a, b\}$,

$C2 = \{\sim a, b\}$,

$C3 = \{\sim a, \sim b\}$,

$C4 = \{\sim b, a\}$.

Table 1

Step #	Chain	ME operation	Clause tree in Fig. 5
1.	$\sim q$	query	$T1$
2.	$\{\sim q]ab$	extension with $\{q, a, b\}$	$T2$
3.	$\{\sim q]a[b]\sim a$	extension with $\{\sim a, \sim b\}$	$T3$
4.	$\{\sim q]a[b]\{\sim a]\sim b$	extension with $\{\sim b, a\}$	$T4$
5.	$\{\sim q]a[b]\{\sim a]$	reduction	$T5$
6.	$\{\sim q]a$	contraction (2×)	
7.	$\{\sim q][a]b$	extension with $\{\sim a, b\}$	$T6$
8.	$\{\sim q][a][b]\sim a$	extension with $\{\sim a, \sim b\}$	$T7$
9.	$\{\sim q][a][b]$	reduction	$T8$
10.	$\square$	contraction (3×)	

We start with the negation $\sim q$ of the theorem to be proved, and apply the weak ME procedure [21] as implemented by PTTP [35]: Table 1. The construction of the clause tree that corresponds to this proof is shown in Fig. 5. The tree is rooted at the clause node corresponding to the query (top clause). The children of a node are ordered left to right in the order that they occur in the input clause, which an algorithm could reorder. In fact the ME procedure works right to left because it works on the right hand end of the chain. Each extension step corresponds to the addition of an input clause to the clause tree to the rightmost open leaf, which is called as the current node. The trees $T1$, $T2$, $T3$, $T4$, $T6$ and $T7$ result from these additions. Each reduction step corresponds to the insertion of a merge path from the current node to an ancestor node in the tree. Trees $T5$ and $T8$ result from these steps. Not all of the 10 ME steps above have a corresponding step in the construction of the clause tree in Fig. 5. The contraction steps correspond to changing the focus of the algorithm, that is the current node, to a node at a higher level or to a node that is further to the left in the clause tree.

However the above construction is not a derivation (Definition 8) because neither the merge path introduced in $T5$ nor the one in $T8$ is between leaves. To show that clause tree $T8$ has a derivation, we show a sequence of binary resolution steps, a proof ordering, in Fig. 6. Fig. 6 is read from top to bottom. Each dashed circle surrounds one clause tree and is labeled by the clause of that tree. The dashed lines connect the clauses resolved together to their resolvents. The bottommost clause tree is $T8$.

Any ME proof is equivalent to a rooted clause tree that grows downward from the query in build ordering as the proof progresses. Any input clause that contains the appropriate literal can be resolved with the rightmost open leaf. As in ME, if a node cannot be extended, then the procedure backtracks to the previous step where another clause could be chosen to extend the clause tree at the appropriate open leaf. The equivalence of ME and this procedure is easily seen and we do not formally prove it.

The justification of the steps in a derivation or in terms of binary resolution steps can be from the bottom of the tree to the top. We do not need to use resolution steps with ancestor clauses as is usually done in order to keep the proof ordering in line with the build ordering. Ancestor resolution is replaced by insertion of a merge path to an ancestor node.

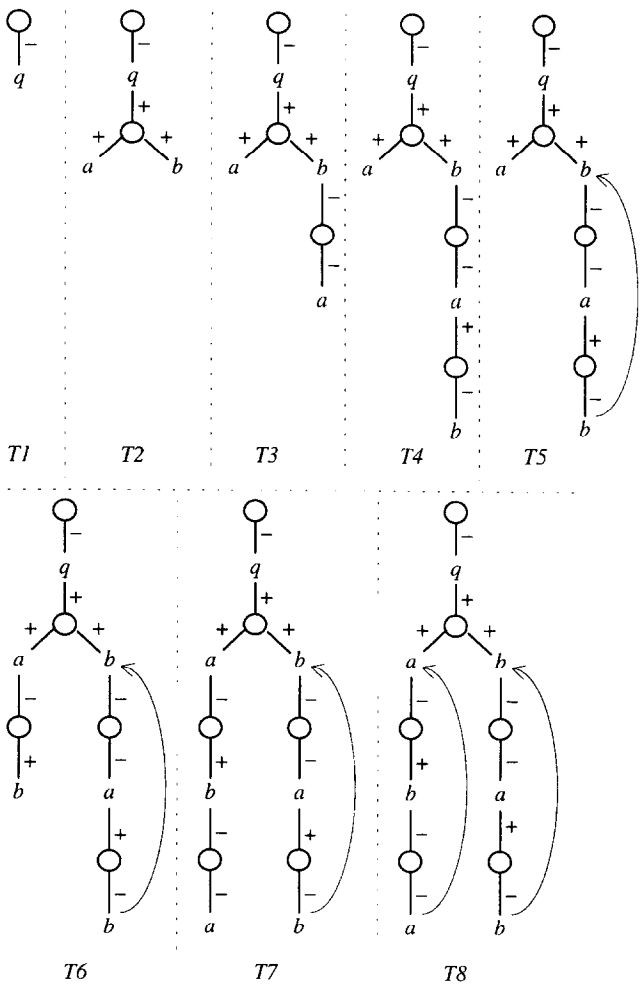

Fig. 5. The build ordering of a clause tree.

3.2. Build ordering concepts: choosing merge paths

Although the definition of a clause tree allows choosing a merge path between nodes only when both are open leaves, after it is used in a resolution step the head of the merge path is no longer a leaf. Under some conditions choosing a merge path to an internal node can be allowed. To choose such a merge path, one must show that some derivation can be constructed. The sequence of operations in the derivation does not have to be the sequence that an algorithm uses to build T. Thus two orderings of operations are considered: the *proof ordering* that satisfies Definition 3, and the *build ordering* that may include the insertion of a merge path to an internal node. An example of this is given in the previous section.

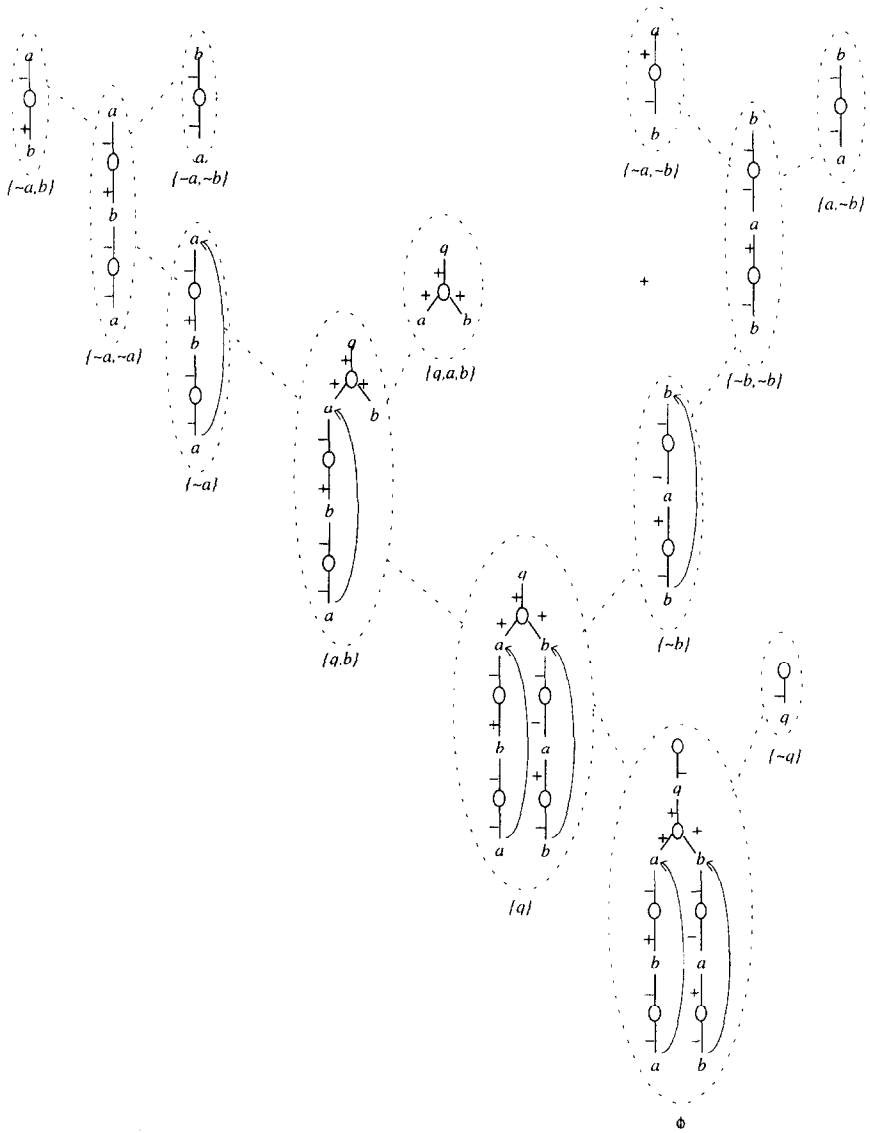

Fig. 6. The proof ordering of a clause tree.

What are the necessary conditions on these more general merge path insertions to ensure that a proof ordering (derivation) exists? First, no two merge paths A and B can end on each other, for if they do then in proof ordering the endpoints of A must both be leaves of some clause tree, with one endpoint of B not yet created. And yet at a different time the endpoints of B must both exist with one endpoint of A not yet created. Since this is clearly impossible, no such derivation can exist. An example is shown in the upper part of Fig. 7.

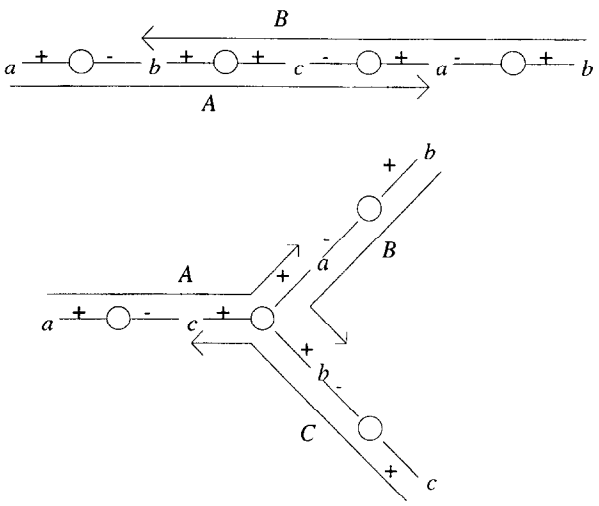

Fig. 7. Two impossible clause trees.

Moreover, if there are two merge paths A and B such that A ends on B then a third merge path C cannot be chosen if it happens that B ends on C and C ends on A. Again the reason is that no derivation exists that can produce the merge path A before B, B before C and C before A. This impossible situation is illustrated in the lower part of Fig. 7. Other conceivable situations involving more competing merge paths are also to be avoided.

The following definitions capture the notion that one merge path must be chosen before another in any derivation. The subsequent result shows what paths must be avoided and what paths can be chosen.

Definition 15 (*Precedes relation on paths*). For two paths A and B in a tree T, $A \prec B$, read *A precedes B*, if and only if the head of A is an internal node, other than the head or tail, of the path B.

Definition 16 (*Legal paths*). A set M of paths in a tree T is *legal* if the $\prec$ relation on M can be extended to a partial order $\prec^*$. A path P is *legal* in a clause tree $T = \langle N, E, L, M \rangle$ if $M \cup \{P\}$ is legal in T.

Theorem 17. *Let $T = \langle N, E, L, M_1 \rangle$ be a clause tree and let M_2 be a set of n unifiable merge paths such that the tail of each path in M_2 is a different open leaf of T, and none of the tails is the head or tail of a path in $M = M_1 \cup M_2$. Let θ be a substitution that makes all the unifiable merge paths in M_2 into merge paths. Then $T' = \langle N, E, L\theta, M \rangle$ is a clause tree if and only if M is legal. In particular, a unifiable merge path P, whose tail is an open leaf and is not the head of a chosen merge path, can be chosen if and only if P is legal in T.*

Proof. Assume M is legal, so there is a partial order $\prec^*$ on M. Extend this partial order to a total order on M. Extend this total order on M to a total order on the atom nodes in N by ordering the heads of the paths in the same order, placing the tail of each path immediately before the head of the path, and placing all the other atom nodes before these nodes. Create clause trees corresponding to each of the clause nodes by Definition 1(a). Perform a resolution by Definition 3(b) for each internal atom node and choose a merge path by Definition 3(c) for each closed leaf node, according to this total order of the atom nodes. The resolutions can always be done. The merge paths can be chosen because all the resolutions on the path have been done first, but the head is not resolved until later and is still an open leaf. Then T is a clause tree.

Conversely if T' is a clause tree, it has a derivation. Order the merge paths of M according to the order that they are inserted into M. When a path is inserted, the head of the path must be a leaf, so that it cannot be on another path that comes before it in this ordering. Hence the relation $\prec$ is a subrelation of this total order, and so $\prec^*$ is a partial order. □

The two paths in the tree $T8$ in Fig. 5 do not precede each other, so they form a trivial partial order. Thus it is easy to see that the set of merge paths is legal, and so by Theorem 17 the tree is a clause tree. More generally, ME chooses a merge path from an open leaf to an ancestor of that leaf.

Definition 18 (*Ancestor path*). An *ancestor path* in a rooted tree is a path from a leaf to one of its ancestors.

Theorem 19. *In any rooted tree, a set of ancestor paths is always legal.*

Proof. Order the paths by the depth of the head in the tree, from deepest to shallowest. Two paths with heads at equal depth are not comparable. This is clearly a partial order. Then if the head of path A falls on an internal node of path B, the head of A must be deeper than the head of B, in which case A precedes B in the order. Thus this partial order extends the precedes relation, as required. □

The result of this theorem is that the ME tree building procedure in Section 3.1 builds clause trees by Theorem 17, and so is sound by Corollary 13.

3.3. SL resolution

The selected literal (SL) procedure was independently developed by Kowalski and Kuehner [16], Loveland [20] and Reiter [25]. It is similar to the ME procedure, but in addition to the ME reduction operation that merges the current node to its ancestor, SL has an additional operation called basic factoring that merges the current node to some left sibling of some ancestor. An SL derivation of the example in Section 3.1 is Table 2. The corresponding clause tree for this proof is built as shown in Fig. 8. The clause tree is built from top to bottom, and from right to left. Since neither of the two merge paths

Table 2

Step #	Chain	SL operation	Clause tree in Fig. 8
1.	$\sim q$	query	$R1$
2.	$\{\sim q]ab$	extension with $\{q, a, b\}$	$R2$
3.	$\{\sim q]a[b]a$	extension with $\{\sim b, a\}$	$R3$
4.	$\{\sim q]a[b]$	basic factoring	$R4$
5.	$\{\sim q]a$	truncation	$R4$
6.	$\{\sim q][a]b$	extension with $\{\sim a, b\}$	$R5$
7.	$\{\sim q][a][b]\sim a$	extension with $\{\sim a, \sim b\}$	$R6$
8.	$\{\sim q][a][b]$	ancestor resolution	$R7$
9.	$\square$	truncation ($3\times$)	$R7$

precedes the other, they form a trivial partial order, so by Theorem 17, the tree $R7$ is a closed clause tree.

As in the ME procedure, the extension operation corresponds to an input clause being added to the clause tree, and the reduction operation corresponds to the insertion of a

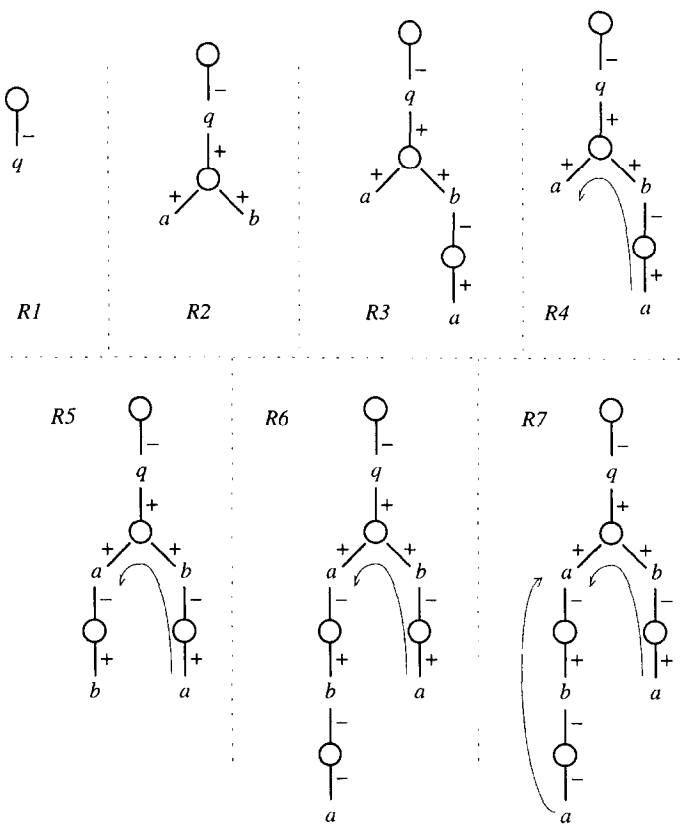

Fig. 8. An SL build ordering of clause trees.

merge path. Ancestor resolution corresponds to the insertion of a merge path from an open leaf to an ancestor, just as in ME. Basic factoring corresponds to inserting a merge path to a sibling of an ancestor, where that sibling is an open leaf. In this case, the sibling must be to the left of the ancestor. We call such a path a *left hook*.

Theorem 20. *In any rooted clause tree, any set of ancestor paths and left hooks, whose heads are atom nodes, forms a legal set of paths.*

Proof. Order the paths by the locations of their heads, first inversely by depth as done in the proof of Theorem 19, and secondarily within each level from right to left. This order is clearly a partial order. Let A be a path that ends on an interior atom node of a path B. There are two cases.

(i) B is an ancestor path. As in the proof of Theorem 19, the head of A is deeper than the head of B and so A precedes B in this partial order as needed.

(ii) B is a left hook. Either the head of A is deeper than the head of B, or they are on the same level. If they are on the same level, the head of B must be to the left of the head of A.

In either subcase A precedes B in the partial order. □

Thus the clause tree building procedure based on SL is sound by Corollary 13, Theorems 17 and 20.

Comparing ME and SL, one sees that if there were no left hooks, then the corresponding clause trees would be identical. Since SL finds all the merge paths that ME uses, but also can use more merge paths, the smallest clause tree found by SL is at least as small as the smallest clause tree found by ME. Indeed the number of resolutions required by SL cannot exceed the number required by ME in the propositional case. But one can do better than SL in this sense.

The truncation steps of the ME and SL procedures contain operations that truncate part of the data structure, so that what remains may be safely used later in reduction and basic factoring steps. Hence useful information may be lost. Procedures using clause trees do not depend on truncation; they use the partial order on nodes to define which later resolutions are legal. Some of this information can be used by the procedure in the next section.

3.4. Left paths and the GC procedure

Shostak [29] developed the graph construction (GC) procedure. It uses C-literals in addition to the A-literals (ancestors) and B-literals (open siblings of ancestors) of ME and SL. An example from his paper [29, p. 60] is used to illustrate the GC procedure. The clauses are:

$$\{\sim N,\ \sim T\},\{M, Q, N\},\{L,\ \sim M\},\{L,\ \sim Q\},\{\sim L,\ \sim P\},\{R, P, N\},$$
$$\{\sim R,\ \sim L\},\{T\}.$$

A GC derivation of the empty clause, starting with the clause $\{\sim N,\ \sim T\}$ is provided by Shostak and is reproduced in Fig. 9 along with the corresponding clause tree construc-

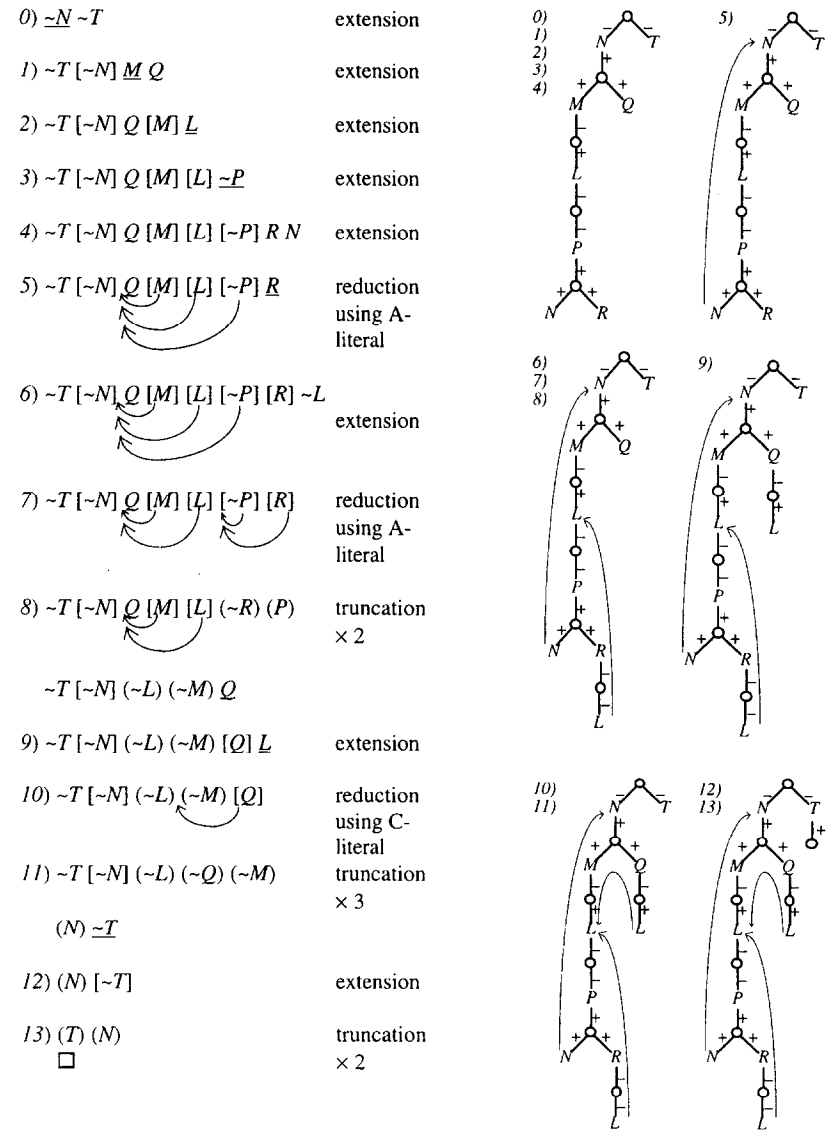

Fig. 9. The clause tree construction corresponding to the GC procedure.

tion. Note that we use ~ to denote negation, [] to denote a framed A-literal, () to denote a C-literal. Except for these changes, the example is identical to Shostak's. In the extension steps we have placed the new literals in the tree in the same order that Shostak has, and we have selected the leftmost of these new literals as the site of further extensions, which is also consistent with Shostak. Selected literals are underlined.

GC allows a merge path to go from an open leaf to an ancestor, or to a node that is to the left in the tree. Reduction with an A-literal corresponds to creating a merge path to

an ancestor node, which we have already called an ancestor path. Reduction with a C-literal corresponds to creating a merge path to some other node that is to the left of the leaf in the tree. We call such paths *left paths*.

Although Shostak says that GC has no merging operation [29, p. 60], he does allow ancestor paths and left paths. What apparently corresponds in GC to SL merges to open left siblings (basic factoring) are merges to the right siblings of ancestors, what we call *right hooks*. However the left hooks allowed by SL all correspond to left paths, and are all found by GC, albeit at a different time. A left hook is found by GC after the subtree below the head of the path has already been constructed; it is found by SL before the subtree is constructed. Thus GC's left path operation is a merging operation that finds the same merges as SL's basic factoring operation. But GC finds left paths other than left hooks, so the smallest GC proof will always be at least as small as any SL proof, just as SL was compared to ME in the previous section.

Both ME and SL prune the search when a legal tautology path is found. ME only looks at ancestor paths; SL considers both ancestor paths and left hooks. GC could also prune when a tautology path is found, either an ancestor path or a left path, but this is not mentioned by Shostak [29]. We call the clause tree version of GC with this extension the ALP procedure, and discuss it in Section 7. Again ALP is superior to SL which is superior to ME in the amount of pruning that can be done using tautologies.

These three procedures above leave open the question of whether using right hooks as well as left paths and ancestor paths improves the procedure. We call this the AllPaths procedure (Section 7), and it is the subject of ongoing investigation [27].

4. Visibility and support

Merge paths cannot be chosen from an open leaf to every internal node of a clause tree. For example if the root of the tree in Fig. 9 had had a rightmost child labeled M, the path from that node to the M in the tree could not have been added legally to the set of merge paths. The new path would be in contention with the ancestor path to N. In fact, none of the literals in the subtree under N can be reached legally by a merge path from a new rightmost child of the root. However in an altered example, other children can be reached. For example Fig. 10 shows a clause tree constructed by GC on a set of clauses that is similar to Shostak's example, but allows a child, 3, of the root to be merged to a new internal node, 9.

Thus sometimes we can insert a merge path and sometimes we cannot, which leads to the next definition.

Definition 21 (*Visibility*). A node u, which is not the tail of a merge path in M, is *visible* from a node v in a clause tree $T = \langle N, E, L, M \rangle$ if $P = path(v, u)$ is a legal path in T. If u is an atom node and the head edge in P is labeled positively (negatively) then the positive (negative) side of u is said to be *visible* from v. If t is the tail of a chosen merge path $path(t, h)$ then we say that the positive (negative) side of t is *visible* from v if the positive (negative) side of h is visible from v. We also say that v *sees* the positive (negative) side of u. Otherwise u is *invisible* from v. If a is the label of u and

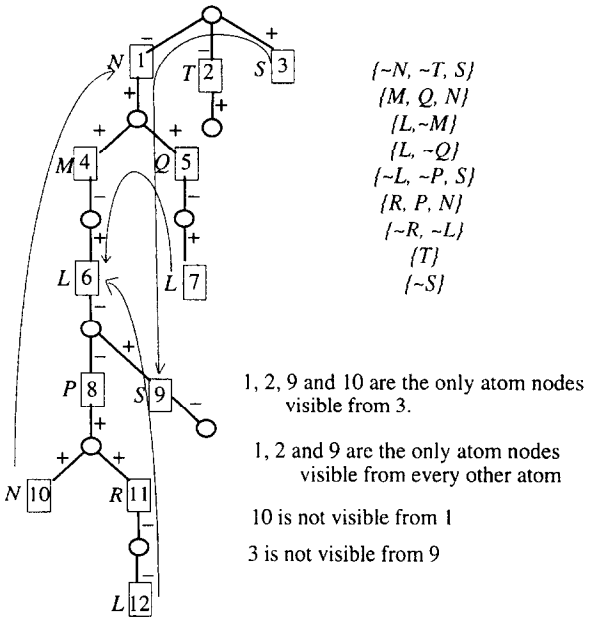

Fig. 10. Alteration of Shostak's example.

the positive (negative) side of u is visible then a ($\sim a$) is a *visible literal.* A merge set N_1 is *visible* from another merge set N_2 if the head node of N_1 is visible from the head node of N_2. By convention a node cannot see itself.

Note that although we originally intended that visibility apply to atom nodes with the same label, we also wish it to apply to any pair of nodes, including clause nodes in some cases. This definition makes merge connected nodes almost equivalent under the visibility relation. The only exception is that the head is visible from the tail, but the tail is not visible from the head. For the example in Fig. 10 node 10 is visible from all other nodes except node 1.

The following theorem shows that visibility is the property that can be used to decide which merge paths can legally be added to a clause tree. This is important for algorithms that build clause trees, as in Section 7.

Theorem 22. *A merge path $P = path(t, h)$ in a clause tree T can be chosen if and only if*

(1) *t is an open leaf,*
(2) *t is not the head of a chosen merge path,*
(3) *h is not the tail of a chosen merge path, and*
(4) *h is visible from t.*

Proof. Assume P can be chosen. By Definition 3(b), the tail t must be a leaf and it cannot be the head or tail of any chosen merge path. Also the head h cannot be the tail

of a chosen merge path. By Theorem 13, P is legal which implies h is visible from t. Conversely, since h is not the tail of a merge path and h is visible from t, then P is a legal path in T. By Theorem 17, P can be chosen. □

Lemma 23. *A path P is illegal in a clause tree $\langle N, E, L, M \rangle$ only if there is a path R in M such that the head of R is an interior point of P, and the directions of R and P disagree on their common edges, of which at least one must exist.*

Proof. Assume that $M \cup \{P\}$ is not a legal set of paths. Then there is a sequence of paths in $M \cup \{P\}$, $P_1, P_2, \ldots, P_n$ such that $P_1 \prec P_2 \prec \cdots \prec P_n \prec P_1$. We may assume that n is as small as possible, and that one of these paths must be P, since M is legal. Without loss of generality we can let $P = P_1$. Thus $P_n \prec P$ so that the head of P_n, call it v, is an interior node of P. If P_n disagrees in direction with P at v, then $R = P_n$ and we are done. Otherwise P_n agrees in direction with P. Consider the tree separated at v by splitting v into two nodes, each attached to one of its incident edges. The tree falls into two components, one of which, call it A, contains the tail of P, and the other of which, call it B, contains the head of P. P_n is entirely within A, so the head of P_{n-1} is in A. But the head of $P = P_1$ is in B. Let P_i be the smallest i such that the head of P_i is in A. Then since the head of P_{i-1} is in B, P_i must contain v. But then $P_i \prec P_{i+1} \prec \cdots \prec P_n \prec P_i$, which contradicts the legality of M. □

Lemma 23 implies that if a path does not run over the head of any other path in the opposite direction, then that path can be added to the set of paths so that the set of paths remains legal. Thus it is easy to see that any set of ancestor paths is legal as all paths point up, as was already shown by Theorem 19. Also if one has a set of ancestor paths and left hooks, then any ancestor path or left hook from an open leaf is legal. It follows easily that ME and SL always construct legal sets of merge paths. However a left path can run over the head of an ancestor path, and then it may or may not be legal.

The visibility relation is useful because it reveals what new merge paths can be added to a clause tree (Theorem 22). The related concept of support is also useful, since it reveals dependency of (the proof for) one atom node on other nodes. Consider an interior atom node n in a closed clause tree T. Suppose that no chosen merge path includes n as an interior node. Then if n is split into two nodes n_1 and n_2, each of degree one, T splits into two clause trees T_1 and T_2. This operation is the reverse of the resolution step of Definition 3(b). Without loss of generality we may assume that n_1 is in T_1 and that $cl(T_1) = \{a\}$. Then n_2 is in T_2 and $cl(T_2) = \{\sim a\}$. However there may be merge paths that include n as an interior node. Suppose there were a single chosen merge path P, shown in Fig. 11 with head and tail labeled by literal b, which has n as an interior atom node. For the sake of argument suppose that P is oriented so that the edge at n labeled "+" precedes the edge labeled "−" at n. Then if we split T at n as before, the tail of P is in T_1 and its head is in T_2. The part of P in T_2 can be discarded, and T_2 still is a proof of $\{\sim a\}$. But if the part of P in T_1 is discarded, then $cl(T_1) = \{a, b\}$, because the tail of P has become an open leaf. Thus the proof of $\{a\}$ from T_1 requires that the subtree that is at the head of P be copied at the tail of P. This subtree supports the proof of $\{a\}$, and this is the intuition behind our use of the term

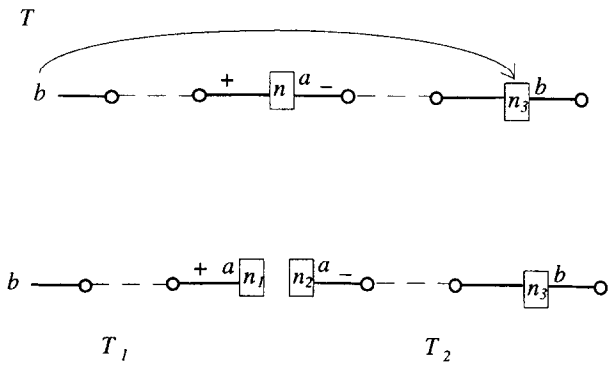

Fig. 11. n_3 is a support of n.

support. This subtree itself may require support from another chosen merge path, which in turn gives indirect support to the proof.

In the development of algorithms, we have found that it is more useful to work with nodes instead of literals. Rather than applying this terminology to the literals, we apply it to the nodes instead. Thus in some sense the node n is given support by the head of P. This discussion leads to the following definition.

Definition 24 (*Support*). In a clause tree, an atom node v *supports* an atom node u (also u *is supported by* v and v is *a support of* u) if there is a sequence of chosen merge paths $P_1, \ldots, P_k$, such that $P_1 \prec \cdots \prec P_k$, v is the head of P_k and u is an interior node of P_1. We also say that v is a *near support* of u if the path from u to v contains no other support of u. Also, v is a *far support* of u if there is no support w of u for which the path from u to w contains v.

In a rooted tree, supports can be ancestors (*ancestor supports*), to the left (*left supports*), to the right, or descendants.

The following theorem relates support and visibility, by showing that a node's near supports also serve to hide it from the view of nodes beyond the support.

Theorem 25. *In a clause tree T, an atom node h that is not the tail of any merge path is visible from an atom node t if and only if there is no near support node of h on the path $P = path(t, h)$.*

Proof. Suppose that there is a near support node of h on P. Then there is a sequence $\langle P_1, \ldots, P_k \rangle$ of chosen merge paths such that $P_1 \prec \cdots \prec P_k$, h is on P_1 and the head of P_k is on P. Then $P \prec P_1 \prec \cdots \prec P_k \prec P$. Hence P is illegal and h is not visible from t.

Conversely, assume that h is not visible from t. Then either h is the tail of a merge path or P is illegal in T. But h is not the tail of a merge path so P is illegal in T. Thus there is a sequence $\langle Q_1, \ldots, Q_j \rangle$ of chosen merge paths such that $P \prec Q_1 \prec \cdots \prec Q_j$

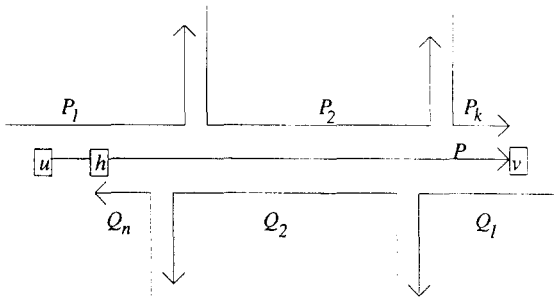

Fig. 12. Merge paths that form an illegal set.

$\prec P$. Thus h occurs on Q_1, so the head of Q_j is a support node of h that occurs on P. Hence there must be a near support node of h that occurs on P. □

Lemma 26. *If v is a support of u, then v is visible from u.*

Proof. Let $P_1 \prec \cdots \prec P_k$ be chosen merge paths such that u is an interior node of P_1 and v is the head of P_k. Let $P = path(u, v)$. Suppose P is not legal. Then there is a sequence $Q_1, \ldots, Q_n$ of chosen merge paths such that $P \prec Q_1 \prec \cdots \prec Q_n \prec P$. Thus v is an interior node of Q_1, so $P_k \prec Q_1$. (See Fig. 12.) Let h be the head of Q_n. Since $Q_n \prec P$, h is an interior node of P. Since the union of paths $P_1, \ldots, P_n$ must be connected, and includes both u and v, P is a subset of this union. Hence h is on some P_i, and so $P_i \prec \cdots \prec P_k \prec Q_1 \prec \cdots \prec Q_n \prec P_i$, a contradiction since these are all chosen merge paths. Thus P must be legal, and v is visible from u. □

Lemma 27. *If w is an ancestor support of a left support of an atom node u, then w is an ancestor of u.*

Proof. Let u have left support v and let v have ancestor support w. If w were not an ancestor of u, then the unique path P from u to v must go up to the nearest common ancestor of u and v, then down through w to v. (See Fig. 13.) Consider the sequence of chosen paths $P_1, \ldots, P_k$ such that v is on P_1, $P_1 \prec \cdots \prec P_k$ and w is the head of P_k.

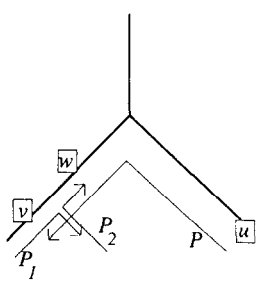

Fig. 13. An impossible ancestor support of a left support.

Then $P \prec P_1 \prec \cdots \prec P_k$, which is impossible, as v is visible from u by Lemma 26. Hence w must be an ancestor of u. □

5. Path reversal, ordered clause sets and foothold scores

When a merge path is created, it does not matter in a fundamental sense in which direction the path is oriented. We define the operation of reversing a path A in a clause tree whereby the whole subtree that is attached to the head of A is removed and reattached at the tail of A. Selected merge paths that use parts of the path A have to be redefined, but the details are not hard. This operation allows us to consider entire families of clause trees as equivalent. It also allows us to define restrictions on clause trees, including the foothold score restriction and the ordered clause set restriction.

Operation 28 (*Path reversal*). Let $T = \langle N, E, L, M \rangle$ be a clause tree, and let $A = path(t, h)$ be a chosen merge path in M. Let u be the node adjacent to h in A. The operation of reversing the path A in T, illustrated in Fig. 14, modifies T to obtain $T' = \langle N, E', L', M' \rangle$ in the following way:

(1) If there is an edge $\{v, h\}$ in E such that v does not occur on A, then the new set of edges is $E' = (E - \{v, h\}) \cup \{v, t\}$. L' is identical to L except that the new edge $\{v, t\}$ is labeled by L' with the same sign as $\{v, h\}$ is labeled by L. In case there is no such edge, $E' = E$ and $L' = L$.

(2) Define the *reversal* of A, $A^R = path(h, t)$.

(3) Let $\{B_1, \ldots, B_n\}$ be the set of chosen paths in M that contain the edge $\{u, h\}$. For each $B_i = path(t_i, h_i)$ define the path B_i' as $path(t_i, h_i)$ if $h_i \neq h$ and $path(t_i, t)$ otherwise. Note that the edge set of B_i' is the symmetric difference of the edge sets of B_i and A, with the edge $\{v, t\}$ replacing the edge $\{v, h\}$. We denote B_i' by $B_i \oplus A$, abusing notation slightly.

(4) Let $\{C_1, \ldots, C_m\}$ be the set of chosen paths in M whose head edge is $\{v, h\}$. For each $C_i = path(t_i, h)$ define $C_i' = path(t_i, t)$, with just the head edge replaced by $\{v, t\}$.

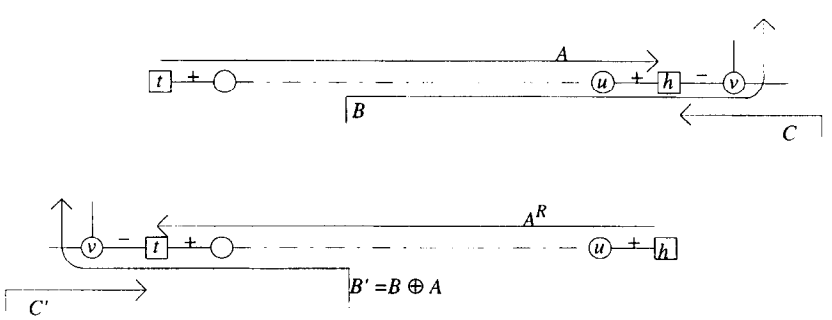

Fig. 14. Before and after path reversal.

(5) Let $M' = M - \{A\} \cup \{A^R\} - \{B_1, \ldots, B_n\} \cup \{B'_1, \ldots, B'_n\} - \{C_1, \ldots, C_m\} \cup \{C'_1, \ldots, C'_m\}$.

Lemma 29. *Let $T' = \langle N, E', L', M' \rangle$ be obtained by the path reversal of the chosen path A in $T = \langle N, E, L, M \rangle$. Then the following statements are true:*

(a) *T' is a clause tree, and $cl(T) = cl(T')$.*

(b) *T can be obtained by reversing A^R in T'.*

(c) *Two atom nodes are merge connected in T if and only if they are merge connected in T'. Thus a set of nodes is a merge set of T if and only if it is a merge set of T'.*

(d) *If a and b are atom nodes in N and $\{a, b\} \neq \{t, h\}$, a is visible from b in T if and only if a is visible from b in T'. Moreover, the same side of a is visible from b. The only difference in the visibility relations of T and T' is that h is visible from t in T and not in T', but t is visible from h in T' and not in T.*

(e) *The visibility relation between merge sets is unaffected by path reversal.*

Proof. (a) Let h, t, u, v, A, B_i and C_j be defined as in Operation 28 (path reversal). Consider any order of the internal atom nodes of T which correspond to a derivation of T, as in the proof of Theorem 17. Order the internal atom nodes of T' in the same way, except that the position of h and t in the ordering of T are interchanged in the ordering of T'. Consider any chosen path P' in M' that corresponds to a chosen path $P = path(x, y)$ in M. We must show that every internal node of P' must occur before x and y in the ordering.

If P does not have h as a node, then $P' = P$. Then all internal atom nodes of P occur before x and y in the ordering for T, so all internal atom nodes of P' occur before x and y in the ordering for T'. This same argument works if $P = C_i$. If $P = B_i$ then note that the internal atom node set of P' is the symmetric difference of the internal atom node set of P and the internal atom node set of A, with t replacing h. Since every internal node of A occurs before h in the ordering of T, every internal node of A occurs before t in the ordering of T'. As h either is y or occurs before y in the ordering of T, t is either y or occurs before y in the ordering of T'. Hence all internal atom nodes of P' occur before y. Thus the ordering of T' corresponds to a derivation of T', and T' is a clause tree.

Note that $cl(T) = cl(T')$, since the open leaves of T are the open leaves of T', except possibly for h if h is a leaf. But in this case it would be replaced by t which is associated with the same literal.

(b) Let $T'' = (N, E'', L'', M'')$ be obtained by reversing A^R in T'. It can easily be seen that $E'' = E$, $L'' = L$ and $M'' = M$.

(c) Assume that c and d are merge connected atom nodes of T.

(i) If $c = d$, then they (it) are still merge connected in T'.

(ii) Assume that $path(c, d)$ is a chosen merge path of T. There are three subcases to consider. If $path(c, d) = A = path(t, h)$, then $path(h, t) = path(d, c)$ is a chosen merge path in T'. If $d = h$ and $c \neq t$, then $path(c, t)$ and $path(d, t) = path(h, t)$ are chosen merge paths of T'. Otherwise $path(c, d)$ is a chosen merge path of T'. In any case, c and d are merge connected in T'.

(iii) Assume that $path(c, w)$ and $path(d, w)$ are chosen merge paths of T. If one of these paths is A, then without loss of generality let $c = t$ and $w = h$. Then $path(d, t) = path(d, c)$ is a chosen merge path of T'. If $w = h$ and neither c nor d is t, then $path(c, t)$ and $path(d, t)$ are both chosen merge paths of T'. Otherwise $path(c, w)$ and $path(d, w)$ are chosen merge paths of T'.

Conversely, assume that c and d are merge connected in T'. Then T is the result of reversing A^R in T' by (b). By the above argument, c and d are merge connected in T.

(d) The proof that the visibility relation on the atom nodes is essentially unchanged between T and T' is similar to the above argument in (a) for the merge paths remaining legal, and is omitted.

(e) We must show that the head node h_1 of a merge set N_1 can see the head node h_2 of a merge set N_2 in T if and only if the head node h'_1 of N_1 can see head node h'_2 of N_2 in T'. By (d) the only difference in the visibility relation is between the head and tail of a path. Since neither h_1 nor h_2 is the tail of any path, the visibility relation between head nodes is unchanged. □

The fundamental purpose of a clause tree T is to show that $cl(T)$ can be derived from the clauses corresponding to the clause nodes of the tree. Lemma 29 shows that the same derivations can be found regardless of the direction chosen to create a merge path. Thus a procedure can specify which way a merge path is oriented, without affecting completeness of the procedure.

Definition 30 (*Reversal equivalent*). Two clause trees are *reversal equivalent* if one can be obtained from the other by a series of path reversal operations.

Reversal equivalence is an equivalence relation. Fig. 15 shows four reversal equivalent clause trees from the clauses $\{\sim a\}$, $\{a, \sim b, \sim c\}$, $\{a, b\}$, and $\{\sim b, c\}$. In general, if there are n merge paths with distinct heads in a clause tree then there are 2^n different reversal equivalent clause trees.

If multiple path reversals are to be done, the order in which the path reversals are done is not important unless there are two chosen paths that have the same head.

Theorem 31. *Given a clause tree $T = \langle N, E, L, M \rangle$ and a subset S of M such that no two paths of S have the same head. Then for any ordering of the elements of S, reversing all paths of S in that order results in the same clause tree.*

Proof. Select two orderings of the elements of S, and consider the two edge sets that result from reversing the paths in these orders. Since each reversal changes exactly one edge, and since no two reversals affect the same edge, the two edge sets must be identical. Each merge path still connects the same tail to the same head. Since the underlying edge sets are trees, the path connecting a pair of nodes is unique, so the merge paths in the resulting trees must be identical. □

The situation when the set of paths being reversed includes paths with the same head is not quite as simple. The important thing is where the subtree attached to the head of

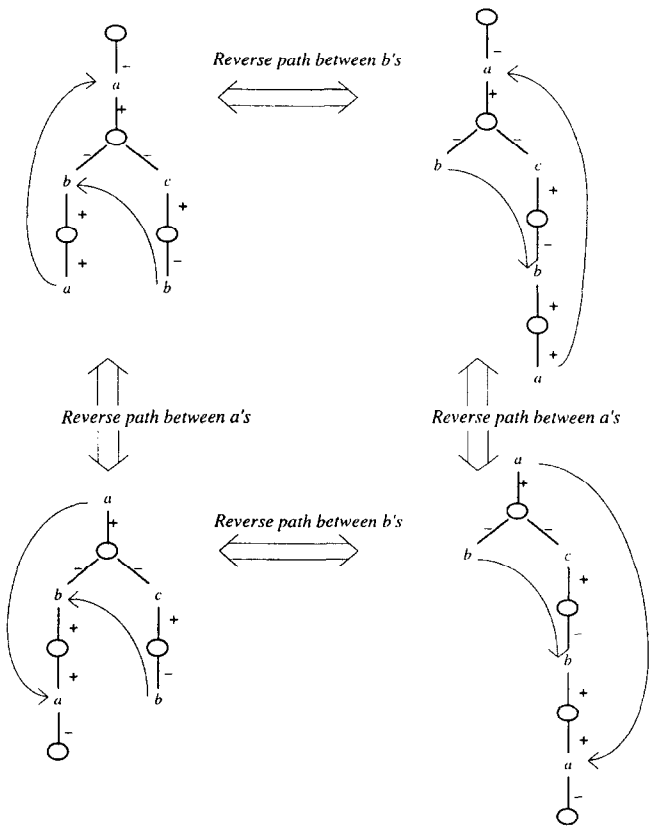

Fig. 15. Four reversal equivalent clause trees.

the paths ends up being attached at the end of the path reversals. It ends up at the tail of the last path reversed, at one of the nodes that are merge connected to the original head. But this is dependent on the order in which the paths with the same head are reversed. If two paths share a head then reversal of the first path followed by the reversal of (the image of) the second amounts to a reversal of the second. See Fig. 16. In fact, looking at this diagram one sees that under the two different operations, the roles of the two paths have been exchanged. That is the image of P_1 after reversing P_1 and then reversing P_2', is the same path as the image of P_2 after reversing P_2.

5.1. The ordered clause set restriction

The ordered clause set (OC) restriction [30,31] depends upon path reversal to decrease the search space needed for any ME-type top down procedure. These papers specifically develop OC for ME and SLI [19] proof procedures, but it can be applied to SL and GC as well. In all of these procedures some merge paths are ancestor paths. If such a path is reversed, a different proof arises. But one needs to allow only one of these

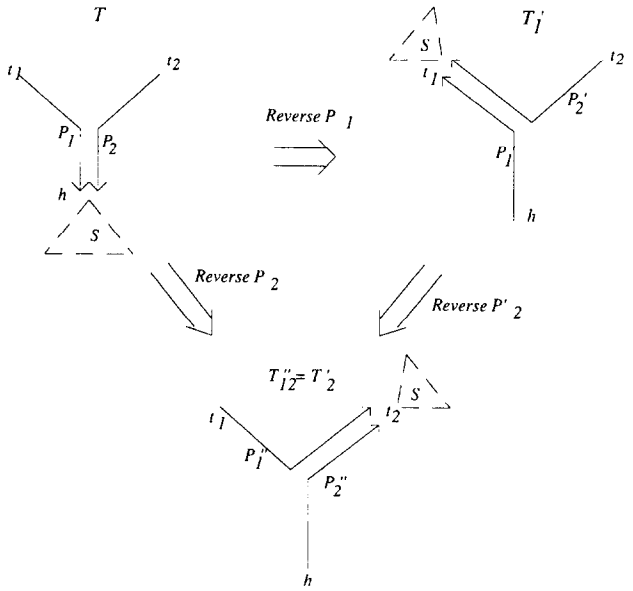

Fig. 16. Reversing two paths with the same head.

two proofs. Initially, the input clauses are assigned an arbitrary total order. The OC restriction chooses one of the two ancestor merge paths by comparing the clauses at the head and tail. The clause node adjacent to the head of any ancestor path is required to precede the clause node adjacent to the tail (or they could be the same clause in the case of first order clauses [36]).

For example in Fig. 17, an *ordered* unsatisfiable set of clauses is presented and two closed clause trees are shown. In the first, the ordered clause set restriction is obeyed since the clause by the head of the path is earlier than the clause by the tail. However the second clause tree is not accepted.

Not only does the OC condition reduce the search space, it can be detected early enough in the procedure to prevent much redundant work. It also prevents multiple redundant proofs from being found when more than one proof is required. This gives a significant improvement to the ME procedure in terms of both the number of inferences and amount of time required. Although merge paths other than ancestor paths are built by the SL and ALP procedures, the ordered clause set restriction is applied only to ancestor paths. The right hooks of SL are oriented left to right, and the left paths of ALP are oriented from right to left, so their reversal would disrupt the structure of the clause tree constructed by the procedures.

The top down procedures discussed in this paper (Section 7.2) when applied to problems in propositional logic can use the ordered clause set restriction with a strict inequality because the clauses at the head and tail of a merge path cannot be the same in these algorithms without causing tautology or unchosen merge ancestor paths (Lemma 47). But the first order logic case in general is more complicated. One wants to be able

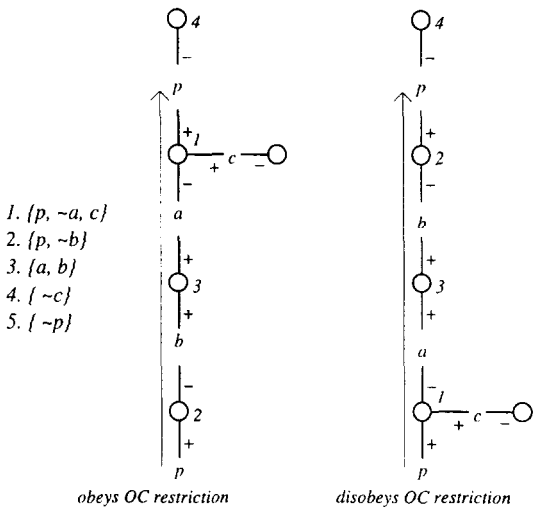

Fig. 17. An illustration of the ordered clause set restriction.

to choose one direction over the other in all ancestor merge paths, so that the direction is acceptable in all paths simultaneously. Paths may arise such that the first and last clause node are different instances of the same clause. Thus not all ancestor paths are ordered by the ordered clause set restriction. See Section 7.5 for more discussion about this. The next section offers another way to orient paths that does not suffer from this problem.

5.2. Foothold scores

Spencer [32] has a different method of orienting merge paths, the *foothold refinement*. The idea is to give each ordered pair of edges at a clause node, a *score* of $+1$, 0 or -1. The score of a path is the total of the scores of each such pair of edges along the path. If the path is reversed, the score gets multiplied by -1. If the number of clause nodes on the path with non-zero score is odd, then the total must be positive or negative. By accepting only paths that have a positive total score, the *proper* paths, one always gets only one *representative* chosen from each equivalence class of reversal equivalent clause trees (Theorem 34).

Definition 32 (*Foothold scores*). Let $T = \langle N, E, L, M \rangle$ be a clause tree and let P be a chosen merge path in T. For each clause node n in N, assume that the literals of the clause with same sign are totally ordered. Literals with different signs are not compared. Let us call such an ordering a *sign ordering* of n, and say T is a *sign ordered* clause tree. Define the *score* of n in P to be in the sign ordered clause tree T:

- 0 if the incoming and outgoing edge at n of P differ;
- $+1$ if the incoming edge at n in P precedes the outgoing edge; and
- -1 if the outgoing edge at n in P precedes the incoming edge.

P is said to be *proper* if the sum of the scores of all the clause nodes is positive; otherwise P is said to be *improper*. T is said to be a *representative* clause tree if all of its chosen merge paths are proper.

Any merge path starts and ends with edges of the same sign, and hence must have an even number of internal nodes at which the sign changes. The number of internal nodes of the path in all is odd, so there must be an odd number of internal nodes at which the sign does not change. But the sign changes at all the atom nodes, so all the nodes at which the sign does not change are clause nodes, and there must be an odd number of them. Since the score is $+1$ or -1 at these nodes and 0 elsewhere, the total score of the path must be odd, and hence either positive or negative. Note also the sign of the score changes at a node if the path runs the other way. Thus the total score of a path gets multiplied by -1 if the path is reversed. Hence either the path or its reversal is proper, but not both.

Lemma 33. *Consider three paths, path(a, b), path(b, c), and path(c, a) in a sign ordered clause tree such that a clause node n is common to the three paths. Let the score of n in each of the paths be i, j and k respectively. Then $|i+j+k| = 1$.*

Proof. If the signs on the three edges on the paths are all the same, then the literals of the three edges are all comparable in the sign ordering. Without loss of generality, we can assume that the literal on the edge towards a is the first literal in the sign ordering. Then $i = +1$ and $k = -1$. Thus $|i+j+k| = |j| = 1$. See Fig. 18. If the signs on the three edges are not all the same, one sign differs from the other two. Then two of i, j and k are 0, and the third value is $+1$ or -1. *Hence* $|i+j+k| = 1$. □

Theorem 34. *Every sign ordered clause tree is reversal equivalent to a unique representative clause tree with the same sign ordering.*

Proof. Let $T = \langle N, E, L, M \rangle$ be a sign ordered clause tree. First we show that T is reversal equivalent to a representative clause tree. Order all the chosen merge paths by some total order consistent with the $\prec$ relation. Consider each of these paths in turn according to this ordering and reverse the path if it is not proper. Since the path reversal

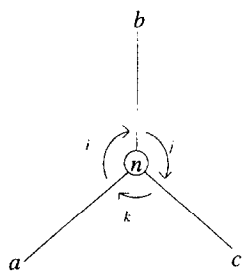

Fig. 18. Scores around a sign ordered clause node.

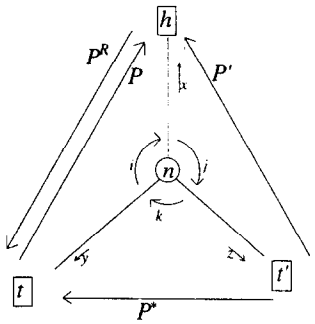

Fig. 19. If P' and P^R are proper then P^* is proper.

of a chosen merge path $P = path(t, h)$ affects only the total score of a path which includes h, any path that must precede P, which is already proper, will remain proper. Paths that include h as an internal node are processed later and it does not matter at the time P is processed whether they are proper or not. The one case that is left is if another path $P' = path(t', h)$ has the same head, from the same side, and has already been processed. We show that if P is improper and is reversed, and if P' is proper, then the path corresponding to P' in the resulting clause tree, $P^* = path(t', t)$, is proper. (See Fig. 16 for an illustration of how P^* arises from P'.) This fact is proved in [32], in the proof of Theorem 3, but we include a proof here for completeness. Let n be the node common to P, P' and P^*. Let x be the total of the scores of the nodes common to P and P', excluding n; let y be the total of the score of the nodes on P^R and P^*, excluding n. Let $-z$ be the total of the score on the nodes common to P' and P^*. Let i, j and k be the scores of n on P, on the reversal of P' and on P^* respectively. Note that x, y and z are computed by traversing away from n. See Fig. 19. Then

$$
\begin{aligned}
&-y+i+x \leqslant -1 && (P \text{ improper}),\\
&y-i-x \geqslant 1 && (P^R \text{ proper}),\\
&-z-j+x \geqslant 1 && (P' \text{ proper}),\\
&y-z-i-j \geqslant 2 && (\text{add}),\\
&y-z+k \geqslant 2+i+j+k && (\text{add } i+j+k),\\
&\qquad \geqslant 1 && (\text{by Lemma 33}).
\end{aligned}
$$

Hence P^* is proper. Thus each reversal does not change any path already processed from being proper. Hence the resulting clause tree is representative.

The remainder of the theorem requires that two reversal equivalent representative clause trees, that have the same sign ordering, be identical. Suppose T and T' are two such clause trees that are not identical but that have the minimal number of chosen merge paths. T and T' cannot be mergeless if they are reversal equivalent but not identical. There is an ordering $\langle a_1, a_2, \ldots, a_k \rangle$ of the atom nodes of T that is an extension of the $\prec$ ordering of the nodes of T, as defined in the proof of Theorem 17. There is an extension of the $\prec$ ordering of the atom nodes $\langle a'_1, a'_2, \ldots, a'_k \rangle$ of T' such

that for $i = 1, \ldots, k$, a_i and a'_i are merge connected, since the two clause trees are reversal equivalent. Now consider the first head a'_i of a chosen merge path that occurs in this ordering in T. Suppose that a_i and a'_i are different nodes. Then a'_i is the head of the corresponding path in T'. Also $path(a'_i, a_i)$ is a chosen merge path in T while $path(a_i, a'_i)$ is a chosen merge path of T'. Since the nodes on these paths must occur before a_i in the ordering, they cannot be the heads of any chosen merge paths, nor can they be the tails of chosen merge paths. Hence these two paths are the reversal of each other. But then one of these paths is not proper, a contradiction of our assumptions. Thus we can assume that $a_i = a'_i$. The set of chosen merge paths with a_i as head now must be the same in T and T'. Removing these paths from the set of merge paths leaves two clause trees that are identical, by our assumption above. Hence T and T' are identical. □

The foothold score restriction can be applied to any clause tree. In particular, one can reject any non-representative clause tree, i.e., one that contains an improper path. In a bottom up procedure, this detects redundant clauses which otherwise would be detected only by a subsumption check.

6. Tautology paths, surgery and minimal clause trees

Most proof procedures based on resolution do not allow two-literal tautologies like $\{a, \sim a\}$ to be produced as a subclause. Such tautologies cannot help in the production of a proof by resolution. This does not prevent the procedure from being complete because whenever there is a proof using a tautology, some other proof that does not have such a tautology is produced by the procedure instead. Clause trees can be used to demonstrate this.

If a clause tree has either a legal tautology path or an unchosen legal merge path, then parts of the clause tree can be removed without adding any more open leaves. In fact, open leaves may be removed. We call this operation clause tree *surgery*, and note that clause trees to which surgery cannot be applied form an important class of clause trees.

Definition 35 (*Minimal clause tree*). A clause tree $T = \langle N, E, L, M \rangle$ is *minimal* if it has no legal tautology paths and no legal merge paths not in M.

Conceptually, minimality means that one cannot reorder the resolutions in the clause tree and create a smaller clause tree that subsumes the original clause. That is, there is no smaller proof of the same result using a subset of the original resolution steps.

We start with an informal discussion of the various types of surgery on clause trees, and give a more formal unified presentation Operation 36. Let $T = \langle N, E, L, M \rangle$ be a clause tree which has a tautology or merge path $P = path(t, h)$ which is legal in T but which is not in M. We consider the case of P a unifiable path later. For the sake of the following discussion, we assume that the path goes from left to right. Thus h and t divide T up into three subtrees, the *tail subtree* A which is to the left of t, the *head*

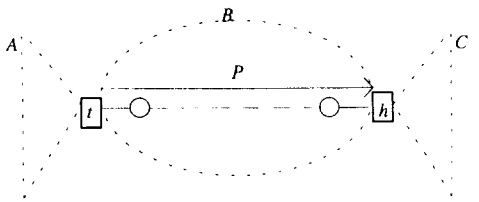

Fig. 20. Identifying the areas for tree surgery.

subtree C which is to the right of h, and the *middle subtree B* which includes all the interior nodes of P, to the left of h and to the right of t. See Fig. 20.

If P is a merge path, then we can perform the following steps, called *tail surgery at* P.

(1) Reverse any chosen merge path which has t is an interior node and whose head is in the near support of t and is in subtree A. Repeat until no chosen merge path which has t as an interior node has its head in A.
(2) Remove from M all the chosen merge paths that have their tails in A.
(3) Replace any chosen merge path Q that has t as its head by $P \oplus Q$, where $\oplus$ is defined in Operation 28.
(4) Remove the resulting subtree A from the clause tree, leaving t as an open leaf.
(5) Add P to the set of chosen paths in M, making t a closed leaf.

An example is given in Fig. 21.

If P is a tautology path, then we can do what we call *tautology surgery at* P. There are two cases to consider. First we assume that t is not visible from h. In this case there must be a support u of t on the path P. Let $P_1, \ldots, P_k$ be a minimal sequence of chosen paths such that t is on P_1, $P_1 \prec \cdots \prec P_k$, and u is the head of P_k. (By a minimal sequence, we mean that no subsequence has this property.)

(1) Reverse the merge paths $P_1, \ldots, P_k$.
(2) Perform tail surgery at P', the path joining t to h in the new clause tree.

The order in which the paths are reversed does not matter by Theorem 31, for none of these paths can have the same head by minimality. Thus we can consider the reversals being done in order $P_1, \ldots, P_k$. Each path reversal puts t onto the next path, so that immediately before the last path reversal, t is on P_k. When P_k is reversed, P is replaced by $P \oplus P_k$. Thus the sign of the edge incident with the tail of P changes, so that P becomes a merge path instead of a tautology path.

The last case to be considered is that in which T is a tautology path and the tail t is visible from the head h. We call the following operation *internal surgery at* P.

(1) For every node n in B which is a near support node of h or t and which can see both h and t, select a path $Q = path(u, n)$ such that u sees the same side of n that t sees (Lemma 26), and reverse Q.
(2) Remove all chosen merge paths that have tails in the middle subtree B.
(3) Not needed.

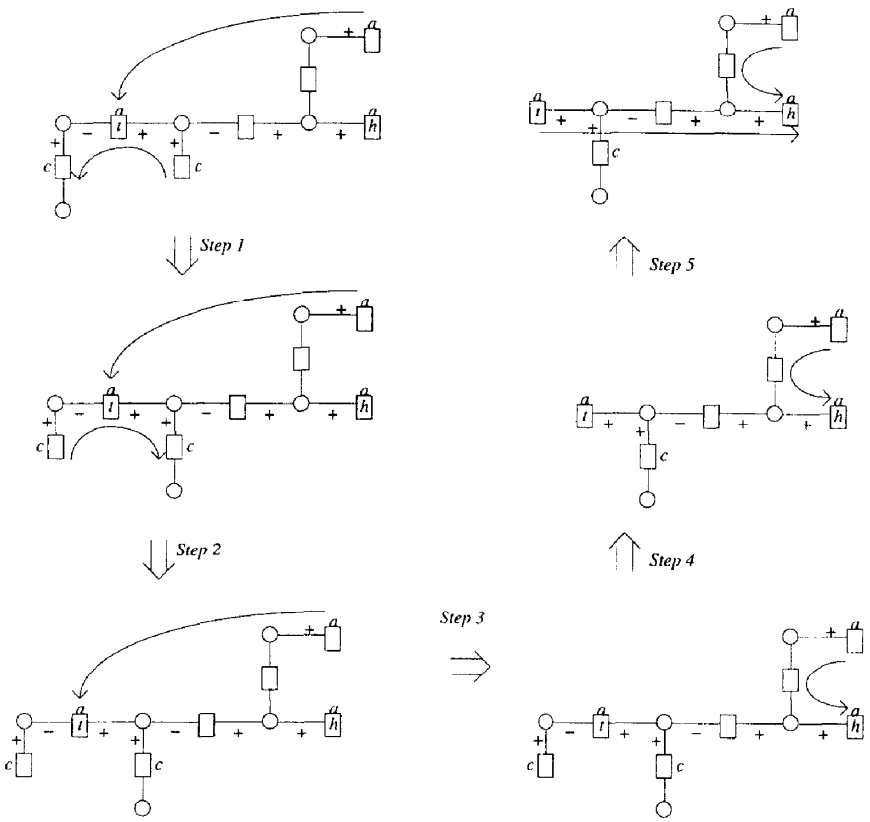

Fig. 21. Tail surgery on *path*(t, h).

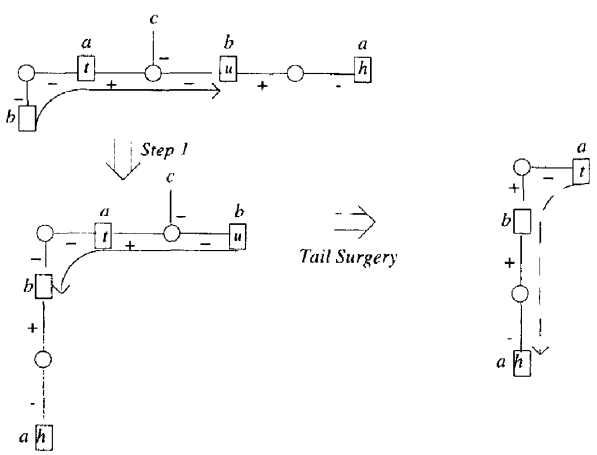

Fig. 22. Tautology surgery where t is not visible from h.

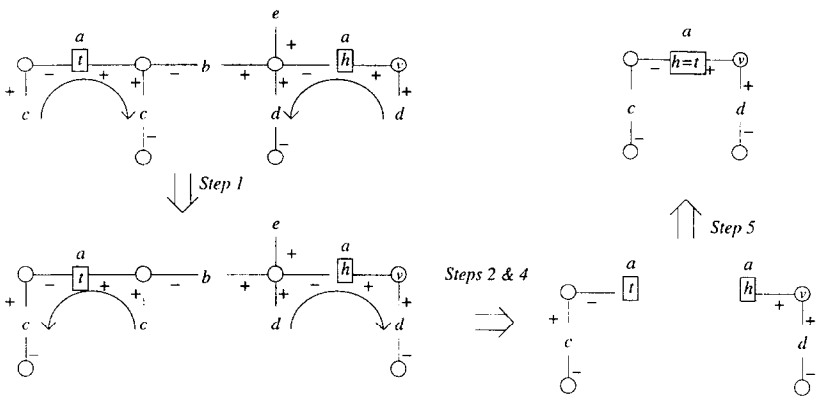

Fig. 23. Internal surgery.

(4) Remove the middle subtree B from T.
(5) Identify the two nodes t and h.

If some chosen path is affected by the removal operation in step (4), then it must have contained all of P as a subgraph. The parts of this path that were in the head subtree and the tail subtree are put back together when t and h are identified.

It is worth pointing out that surgery at a merge path P may generate a very different result than surgery at P^R, the reversal of P. For example in Fig. 24, the clause of the result of surgery at P is $\{c\}$, whereas the clause of the result of surgery at P^R is $\{a\}$.

The three different surgery operations can all be given in a single definition. The numbers on the steps in Operation 36 correspond to the numbers on the steps in Figs. 21–23.

Operation 36 (*Clause tree surgery*). Let $P = path(t, h)$ be a legal path in a clause tree $T = \langle N, E, L\ M \rangle$, unifiable by the substitution θ. Neither t nor h can be the tail of a chosen merge path. Let V be the set of nodes, excluding t, that can see the side of t complimentary to the sign of the head edge of P, but cannot see that side of h. The operation of *surgery at P on T* produces $T_P = \langle N_P, E_P, L_P, M_P \rangle$ by the following sequence of steps:

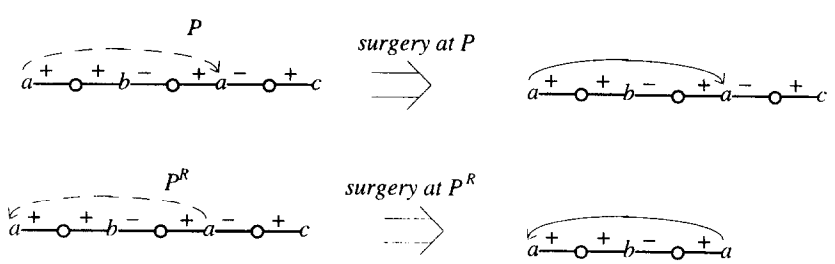

Fig. 24. Surgery at P and P^R.

(1) Let n be an atom node in V that supports t or that supports h if h is in V. Consider a chosen merge path whose tail is not in V with head n that comes from the t side of n. Reverse this path. Repeat for all such atom nodes in V. This step produces a clause tree $T' = \langle N, E', L', M' \rangle$.

(2) For each chosen merge path in M' that has its tail in V, remove it from M' to leave a set of chosen merge paths

$$M'' = M' - \{Q \mid Q \text{ in } M' \text{ and tail of } Q \text{ is in } V\}.$$

These include all the paths that were reversed in step (1).

(3) For each chosen merge path $Q = path(s, t)$ in M'', replace it with $Q' = path(s, h)$ to form

$$M''' = \{Q \mid Q \text{ is in } M'' \text{ and } t \text{ is not the head of } Q\} \cup \{path(s, h) \mid path(s, t) \text{ is in } M''\}.$$

(4) Remove from E' all edges incident to a node in $V - \{h\}$ to obtain E_P, and let $N_P = N - (V - \{h\})$.

(5) (a) *Tail surgery.* If h is not in V, then add P to the set of chosen paths $M_P = M''' \cup \{P\}$.

(b) *Internal surgery.* If h is in V, then identify the node h with t in the graph $\langle N_P, E_P \rangle$, and delete all other nodes of V from any path of M''' in which t and h occur. Let M_P be the resulting set of merge paths.

(6) Define L_P to be $L\theta$, restricted to $N_P \cup E_P$.

It is worth pointing out that while the tail of a chosen merge path cannot be seen by the head of that path, it is possible for the tail of an unchosen path to be seen from the head. Thus in Operation 36 it is possible that h is in V, since P is unchosen.

We must show that the above step (1) can be performed; that is, for each node n in V which is a support node of t (or of h if h is in V which can be handled in the same way), there is a chosen merge path $Q = path(w, n)$ with w not in V. Note that support nodes of t in V are precisely the near support nodes on the same side of t as the nodes in V. Since n is a support node of t, there are paths $P_i = path(t_i, h_i)$ for $i = 1, \ldots, k$, such that t is an interior node of P_1, $P_1 \prec \cdots \prec P_{k-1} \prec P_k$ and $n = h_k$. If $k = 1$, then P_1 is a possible choice for Q. Otherwise, since h_{k-1} is on P_k and is a support node of t, h_{k-1} must make t invisible from either t_k or h_k. But h_k is in V and so t is visible from h_k. Hence t is invisible from t_k which implies t_k is not in V. Therefore P_k is a possible choice for Q.

The above operation of surgery may produce different outcomes depending on the selection of paths in step (1), so strictly speaking it is not a well-defined operation. However if a different set of paths were selected in an alternate application of the surgery, the result is a clause tree that is reversal equivalent to the result of the original application (Theorem 38 below). In fact if one starts with reversal equivalent trees, one ends up with clause trees that are reversal equivalent up to isomorphism (Theorem 39 below). Thus the operation of surgery is well defined for the equivalence classes of the relation between clause trees T_A and T_B characterized as: T_A is isomorphic to a clause tree that is reversal equivalent to T_B.

Theorem 37. *The application of surgery on a clause tree T at a path P results in a clause tree.*

Proof. We use the symbols given in the definition of Operation 36. Assume without loss of generality that the head edge of P is positive. Then V is the set of nodes from which t is visible negatively. Since t is not the head or tail of any path reversed in step (1), we are assured by Lemma 29(d) that the set of nodes from which t is visible negatively at the end of step (1) is still V.

We show that $\langle N_P, E_P \rangle$ is a tree. This is clear if V is empty since $N = N_P$ and $E = E_P$. Assume V is not empty. Consider the nodes of T' adjacent to nodes in V, but not in V. Of course t is adjacent to a node in V. If h is in V, and h is not a leaf, then the clause node beyond h, call it v, is a second such node. See Fig. 23 for an example. Let $n \neq h$ be in V such that n is adjacent to a node $u \neq t$ and u is not in V. Then the path from u to t must be illegal while the path from n to t is legal, for otherwise u is in V. By Theorem 25, there must be a support node of t on $path(u, t)$ but not on $path(n, t)$. The only possible such node is n itself, with u further from t than n. Thus n is a support node for t, and there must be a chosen merge path with head n and tail outside V that gets reversed in step (1). See Fig. 25. But this reversal makes n a closed leaf, so u does not exist in T'. Hence only t and possibly v can be nodes of T' outside of V and adjacent to nodes in V.

Consider any node m of T' which is not deleted in step (4). Thus m is in N_P. The path joining m to t in T' cannot include any nodes of V unless the path goes through v, and includes h, and so includes all of P. In this latter case, h is in V so h and t get identified in step (5b). Thus all nodes in N_P are connected to t by a path, so $\langle N_P, E_P \rangle$ is connected. If t and h are identified, no cycle is created because the other nodes on P are deleted. Thus the underlying graph $\langle N_P, E_P \rangle$ is a tree.

It is not difficult to see that $\langle N_P, E_P, L_P, \phi \rangle$ is a mergeless clause tree. Consider the clause node c in N_P. We must show that the set of associated literals of the edges incident with c is an instance of the clause $L_P(c) = L(c)$. For every atom node in V, except possibly h, its neighbouring clause nodes are also in V and hence are deleted as well. In the exceptional case, h is in V but its neighbouring clause node v is not in V. When h is identified with t, the edge $\{t, v\}$ is created. The labels given by L_P to h and t are identical so the associated literals incident with v remain the same. All of these elementary clause tree are combined by Definition 1(b), using the substitution $L\theta$ which ensures that the atom nodes are labeled identically. The whole of $\langle N_P, E_P, L\theta, \phi \rangle$ can be built up because $\langle N_P, E_P \rangle$ is a tree.

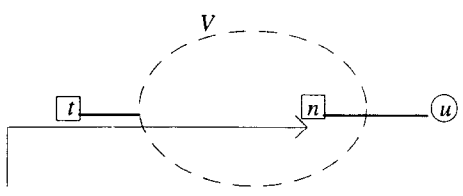

Fig. 25. In T', showing u cannot be adjacent to a node in V.

We must still show that M_P the remaining chosen merge paths, is a legal set of merge paths. Let Q be in M_P. Then the tail of Q was not in V. Nor can the head of Q have been in V, because that would imply that the head of Q was a support of t or h. But then a merge path of T with the same head as Q and the same sign would have been reversed, making this node a tail of a chosen merge path at the same time as it is the head of a merge path. This contradicts the definition of a legal set of merge paths. Q could still have had some nodes in V, by starting outside of V, entering V, and then leaving V. Because the only possible entry points to V are h and the node in V adjacent to t, Q must have included both t and h. Thus Q must have included P as a subpath. Then all the internal nodes of P are deleted in step (4), breaking Q into two pieces, but then in step (5), t and h are identified, and Q is made into a single path again. Hence each object in M_P is a path, and the head and tail of each are the head and tail of a merge path in M'. Thus each path is a merge path. That M_P is a legal set of paths follows because the precedes relation on M_P is a subrelation of the precedes relation of M', and hence extends to a partial order. □

The following theorem points out that the choice of a path on which to perform surgery determines the resulting clause tree, up to path reversal.

Theorem 38. *In any two applications of clause tree surgery at a path P in a clause tree T, the resulting clause trees are reversal equivalent.*

Proof. We use the terminology used in Operation 36. Let n be a near support node of t in V. The only non-deterministic step in clause tree surgery is the selection of paths to reverse in step (1). Suppose the two applications of surgery differ only by the selection of $Q_1 = path(t_1, n)$ and $Q_2 = path(t_2, n)$. The only difference at the end of step (1) is the path reversal of $Q_1 \oplus Q_2 = path(t_1, t_2)$. See Fig. 26. Both t_1 and t_2 are outside V,

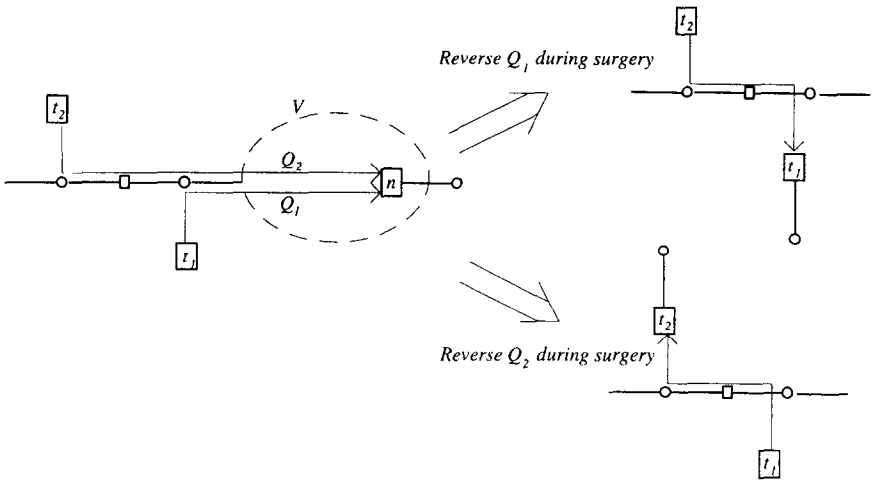

Fig. 26. Different results of different applications of surgery.

and so $path(t_1, t_2)$ is not deleted in step (2). Also, t_1 and t_2 are not deleted in step (4). Reversing $path(t_1, t_2)$ does not change the tail of any other chosen merge path and hence has no impact on whether the tails of chosen merge paths are in V. Hence the paths that are deleted in step (2) of surgery are the same regardless of whether Q_1 or Q_2 is reversed. Thus the one edge change caused by the reversal of $path(t_1, t_2)$ is the only difference between the edge sets at the end of the surgery operations. Now consider the effect of the two surgery operations on a member Q of M that is not removed in step (2). It is not possible for the head of Q to be t_1 or t_2. Thus the head and tail of Q are unchanged by the path reversals in step (1). Hence the corresponding paths resulting from Q must differ (if at all) by precisely the effect of reversing $path(t_1, t_2)$, even though it is possible that the paths Q and $path(t_1, t_2)$ may have pieces deleted out of its middle. Therefore the final clause trees differ only by the path reversal of $path(t_1, t_2)$.

If the two surgery operations differ by more than one selection in step (1), we can prove that the resulting clause trees are reversal equivalent by using induction on the number of near support nodes of t nodes at which different paths are selected. □

Theorem 39. *Let T and T^* be reversal equivalent clause trees, and let $P = path(t, h)$ be a unifiable path in T upon which surgery can be performed. Let the node t^* be the head node of the merge set of t in T^*. Similarly let the node h^* be the head node of the merge set of h in T^*. Then the result of surgery at P on T, and the result of surgery at P^* on T^* are reversal equivalent, up to isomorphism.*

Proof. First we consider applying surgery to two clause trees that differ by a single path reversal. Let $T = \langle N, E, L, M \rangle$, $P = path(t, h)$ and V be as in Operation 36. Assume without loss of generality that the head edge of P is positive. Let $Q = path(x, y)$ be a chosen merge path of M. Let T^* be the result of reversing Q in T. Consider the differences between surgery at P in T, and surgery at P^* in T^*. Without loss of generality we assume as far as possible that the two different surgeries choose the same paths to reverse in step (1). By Lemma 29(d), the set V for each surgery is the same. We have several cases to consider, depending on where x and y are relative to t, h and V. We start with $\{x, y\}$ disjoint from $\{t, h\}$, so $P^* = path(t, h)$. If x and y are both not in V, we use the same argument as used in Theorem 38 in the situation where a different path is chosen to be reversed in step (1). Then the results of the two surgeries are reversal equivalent.

If x and y are in V, then Q is removed in step (2) of surgery at P in T, and the reversal Q^R of Q is removed in step (2) of surgery at P^* in T^*. In step (4) the same nodes are removed in both surgeries, and so are the same edges, except that the edge in T beyond the head of P and the edge in T^* beyond the head of P^*, which are not in T^* and T respectively, are removed as well. Thus in this case the resulting clause trees are identical. If x is in V and y is not in V, then Q is removed in step (2) of surgery at P in T. But in T^*, Q is reversed to obtain Q^R, with head x in V and tail y outside of V. Then Q can be chosen to be reversed for x, which is now a support node for t in step (1). Thus Q^R is reversed to form Q again. Hence at the end of step (1), the same clause tree is formed from both T and T^*. A similar argument works if y is in V, x is not in V.

We are left with the case that $\{t, h\}$ is not disjoint from $\{x, y\}$. We know that neither t nor h can be x, since x is the tail of a chosen merge path. Let $t = y$. The case $h = y$ is handled similarly except working from T^* to T. There are two subcases, depending on whether x is in V or not.

Let x be not in V, so the head edge of Q is positive. Reversing Q then interchanges the role of x and y, so $path(T^*, h^*) = path(x, h)$. In both surgeries, the same paths are removed from M, since the location of the tails of other chosen merge paths are not affected by the path reversal of Q. Any chosen merge path $path(s, t)$ that remains after step (2) of surgery on T is replaced by $path(s, h)$ in step (3). In T^*, $path(s, t) = path(s, y)$ gets replaced by $path(s, x) = path(s, T^*)$, which in turn gets replaced by $path(s, h)$ in step (3) of surgery on T^*. Thus all chosen merge paths at the end of step (3) have the same heads and tails in each of the two surgery operations. The set of nodes from which x is visible negatively remains V, so the same nodes are removed in both surgeries is step (4). The edge beyond the head of Q, if it exists, joins $t = y$ to a node u in V. This u is removed in the surgery on P in T. Similarly the edge $\{x, u\}$ in T^*, which replaces the edge $\{t, u\}$ in T, is removed in the surgery on T^*. Since the node and edge sets are the same, and each corresponding merge path has the same head and tail, the merge paths are the same. Thus the resulting clause trees are identical.

Let x be in V, then Q is removed in step (2) of surgery in T. But in T^*, $T^* = x$. Now the nodes from which T^* is visible negatively is $V^* = (V \cup \{y\}) - \{x\}$. In step (4) the same nodes are removed in the two different surgeries, except that x and y interchange roles: y is kept and x discarded in surgery on T; x is kept and y discarded in surgery on T^*. The same edges get removed in step (2) as well, except that if u exists then the edge $\{y, u\}$ in T is replaced by the edge $\{x, u\}$ in T^*. Exactly the same paths are removed in step (2) in both cases. The resulting clause trees are isomorphic, with the node x replacing the node y and otherwise the mapping being the identity. This is the only case in which an isomorphism is required. In all other cases only path reversals are needed.

If two clause trees T and T^* differ by several path reversals, it follows by induction that the result of surgery on corresponding paths results in clause trees that are isomorphic to reversal equivalent trees. □

The importance of surgery, and the importance of minimal clause trees, come from the following theorem.

Theorem 40. *Let T be a clause tree on a set S of clauses. If T is not minimal, then there is a minimal clause tree T' on S such that $cl(T') \subseteq cl(T)$.*

Proof. Since T is not minimal, there is a legal tautology or unchosen merge path P in T. Then either tail surgery or internal surgery can be performed on P without a substitution on the variables. Since surgery does not affect clause nodes unless they are completely removed, the resulting clause tree is still from S. The surgery can only remove open leaves, never insert them so that the corresponding clause can only get smaller, not larger. If the resulting tree is still not minimal, then surgery can be repeated until the resulting clause tree is minimal. □

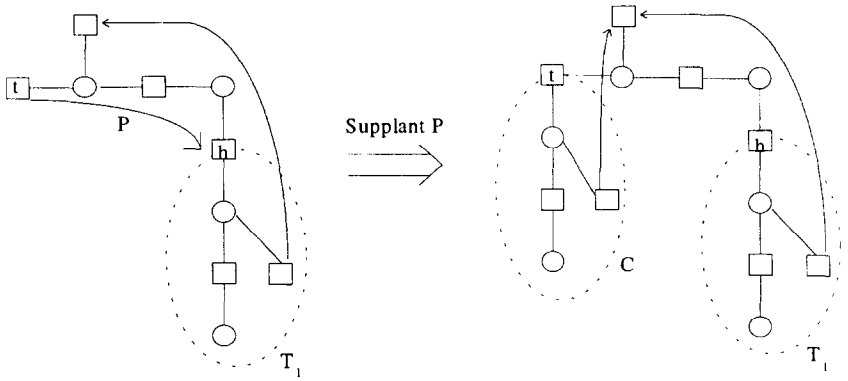

Fig. 27. Supplanting a path P by its head subtree T_1.

Thus non-minimal clause trees are redundant, if one has all the minimal clause trees. Hence it is desirable that a procedure to construct clause trees be able to prune the search whenever the clause tree being constructed becomes non-minimal.

The surgery operations are not reversible like path reversal is. One cannot know what was removed. However tail surgery does allow a one-sided inverse operation which can be reversed by surgery.

Operation 41 (*Supplanting a path*). Let P be a chosen merge path with head h and tail t in a clause tree T. The path P is *supplanted* by its head subtree when the following are done:

(1) Make a copy C of the head subtree of P.
(2) Attach C to T using t in place of h.
(3) Remove P from the set of chosen merge paths.
(4) For any chosen merge path with a tail in the subtree at h, add a chosen merge path from the copy of the tail in C to either the copy of the head if the head is in the subtree, or to the head itself if it is not in the subtree.

For example, the subtree in Fig. 27, a copy C of the subtree T_1 supplants the path P. That the result of supplanting a path is a clause tree, and that tail surgery by P inverts the operation, is omitted.

7. Top down procedures for building clause trees

In this section we develop the top down procedures ME, SL, ALP, MinALP and AllPaths for building rooted clause trees and explore some of their properties. We begin by describing the general setting, give intuitive descriptions of the procedures, followed by pseudocode, and then proofs of completeness of some of the procedures.

Each procedure starts with a satisfiable set S of input clauses (axioms), and another clause C, the negation of the theorem to be proved. C is also added to the set of input

clauses if the clauses are not propositional. Each procedure assumes that elementary clause trees from the clauses of S and from C are available. The clause tree from C is used as the top clause. One open atom node in the clause tree is deterministically selected as the *current node* by a computation rule R. Thus R determines whether the search is depth first, left to right, and so on. We usually select R as depth first, and either left to right or right to left. Some of the procedures require that R be depth first, because this guarantees that certain atom nodes are visible without having to explicitly check. All of these procedures contain one or more non-deterministic steps, such as selecting which input clause to resolve against the current atom node. This causes the search to branch. This non-determinism may be implemented by (chronological) backtracking search, or with a parallel computation. If failure results at any point then the current branch of the search is terminated. The procedure ultimately fails if all nondeterministic steps end in failure.

7.1. Visible lists in the ALP procedures

Procedures in the ALP family all use Ancestors paths and Left Paths. This family (so far) consists of ALP, MinALP for minimal ALP, ALPOC and MinALPOC. The last two use the ordered clause set restriction on ancestor paths.

All of these procedures build trees top down, left to right. At any given step of the procedure, an open leaf is being considered. It has a sequence of ancestors $a_1, a_2, \ldots, a_{n-1}$. None of the nodes to the left of a_n are open whereas all nodes to the right are open leaves and are the children of ancestors. See Fig. 28.

Some overhead work is required for the ALP procedures to guarantee that the set of chosen ancestor and left merge paths is legal. In contrast, the set of merge paths chosen by the ME and SL procedures is necessarily legal, so this overhead is not required. To determine whether a left path is legal, a list called the *visible list* is maintained,

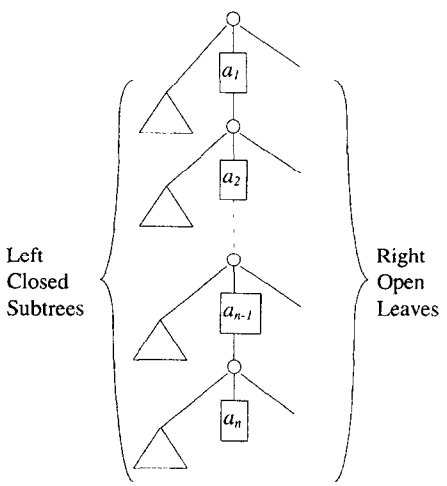

Fig. 28. Portions of the clause tree as it is built.

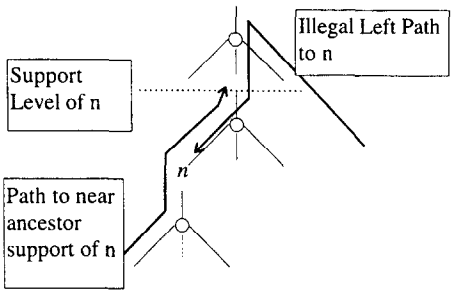

Fig. 29. Illustration of support level.

consisting of the literals that are visible from the current node. A node n in the left portion of a clause tree is visible from the current node if and only if the path from the current node does not pass over a near support node of n. The support nodes of n that could appear on this path are ancestor support nodes. Of course if any support node occurs on this path then a near support node must occur on it. Since the tree is rooted, it is sufficient to know the level of the near ancestor support node of n; this is called the *support level* of n. See Fig. 29. When the subtree beneath n is closed, the support level of n is calculated from the ancestor support lists of the atom node grandchildren of n excluding the level of n itself, by taking the maximum of the levels of the ancestor supports of these grandchildren. A subtle point: it is not sufficient to consider only the support levels of the grandchildren of n. For instance, if the support level of one grandchild is the level of n then its next nearest ancestor support could be the support level of n.

The visible list is maintained by two operations. Recall that atom nodes are selected depth first. When the subtree below an atom node n is closed, the support level of n is calculated and a pair consisting of this level and the literal corresponding to n's upper edge is inserted into the list. When the newly selected atom is at a level less than (above) the previously selected atom node, all of the entries in the visible list associated with the levels greater than the newly selected atom node are removed. This method of keeping track of visible nodes is analogous to Shostak's method of keeping a pointer into the chain for the C-literals [29]. It is also analogous to the enforced folding up procedure of [17] in connection tableaux.

To implement the ordered clause set restriction, whenever an ancestor merge path is detected, the clause numbers of the neighboring clauses must be compared, and rejected if the deeper clause has a smaller number than that of the ancestor node.

The ALPOC algorithm has been implemented [33] by adding it to Stickel's PTTP procedure [34]. These two operations for maintaining the visible list are relatively inexpensive. However, using a visible list adds some expense since, as each node is selected, the visible list must be searched for tautology paths and left paths. The overhead of ALP over PTTP slows it down by a factor of 2 to 4 per inference [33]. This agrees with the factor of 3 mentioned in [17].

The ALPOC system frequently improves upon PTTP, especially when left paths give rise to a shorter proof so that the number of levels of search is decreased. Because the

iterative deepening search strategy is not forced to look as far, the number of inferences is decreased dramatically in these examples. For instance on the TPTP [38] problem GRP008-1, PTTP required 9,061,115 inferences to find a proof within 12 levels of search (12 extension steps) [34], but our implementation of ALPOC required only 1,260,198 inferences to find a proof after 10 levels of search [33]. When we compared on the same computer the version of PTTP that compiles to Prolog [35] to our modified version of the same program to build ALPOC proofs on the TPTP problem RNG001-3, we found that PTTP required 131,224 inferences and 14.284 seconds to find a proof at level 18, but ALPOC required 6,230 inferences and 1.90 seconds to find a proof at level 14. For SYN001-1.005, the PTTP that compiles to Prolog found a proof after 26,790,196 inferences, and ALPOC found a proof after 465 inferences.

7.2. Finding minimal clause trees

ALPOC is the tightest top down procedure yet mentioned, but still it can generate proofs that are non-minimal, hence redundant. In fact the example in Fig. 30 shows a problem for which it generates a non-minimal clause tree. All ancestor paths and left paths are handled by ALP and ALPOC. But as this figure shows, not all right paths are.

All three procedures ME, SL and GC can construct the clause tree on the left of Fig. 30. However there is a legal, unchosen merge path between the nodes that are labeled with c. If the merge path between the nodes labeled a were reversed, then a legal tautology path would be created between the nodes labeled c. Thus adding the ordered clause set restriction to these algorithms does not eliminate this non-minimal proof.

We construct a procedure MinALP which avoids these non-minimal proofs, by detecting such missed paths as in Fig. 30. Let t be the tail and h be the head of an unchosen merge or tautology path P in a clause tree produced by ALP on a set of propositional clauses. If h is an ancestor of t, then when t was the current node in the ALP procedure, the path P would have been discovered. Hence this is not possible.

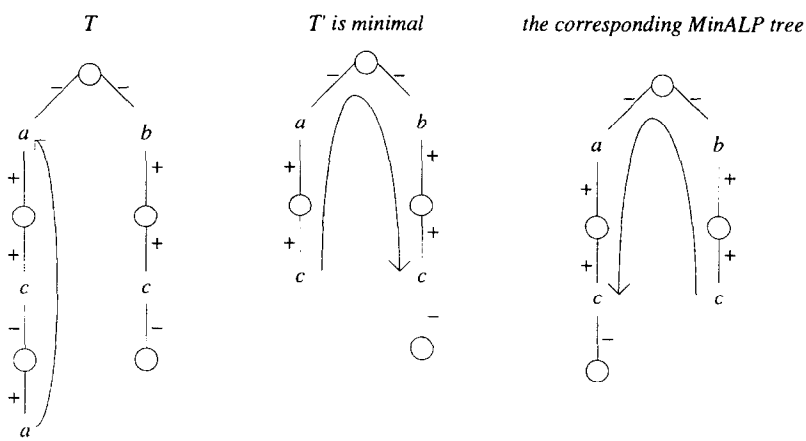

Fig. 30. A non-minimal ALP tree, the minimal tree found by surgery and the MinALP tree.

Similarly if h is a descendant of t, the ALP procedure would have discovered the reverse of P when h was the current node. Also if h is to the left of t, then when t was the current node P would have been handled. Thus we can assume that h is to the right of t. Therefore MinALP is ALP with this additional step: reject the tree if after the subtree below h is closed, h can be seen by a node t to its left that is labeled by the same atom as h.

To check this additional condition, MinALP needs to keep track of more information than ALP. Besides the three data structures required by ALP (the list of ancestors of the current node, the list of nodes visible from the current node and the list of ancestor supports of the current node) MinALP also needs to maintain two more data structures: the *invisible list* of nodes that the current node cannot see, and the near left supports of the current node. The operations on these data structures are not difficult to implement. A node is added to the invisible list when it is removed from the ALP visible list. Once on the invisible list, it stays on the invisible list. When calculating the near left supports of the current node, one must consider not only the heads of left paths on which the current node occurs, but also the heads of left paths on which those heads occur, and so on. However the ancestor supports of these left supports are not needed, since by Lemma 12, an ancestor support of a left support is an ancestor, and therefore is already maintained in the ALP ancestor support list.

The search for this path $P = path(t, h)$ can begin as soon as the subtree below h is closed because then all of the supports of h are known. Consider as candidates for t all the nodes on the invisible list with the same label as h. Eliminate from this set any t whose ancestor list does not include the near ancestor support of h, because if this support of h is on the path from t to h then it makes P illegal. Next for each left support d of h, eliminate any t that has d in its ancestor list, since these supports are on the path from t to h and so make P illegal. If any candidate remains, then the clause

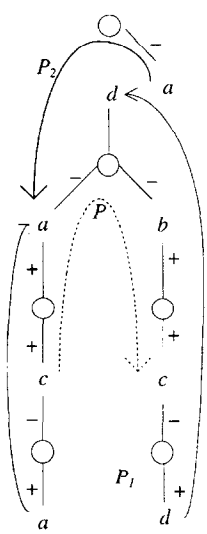

Fig. 31. Candidate bad path P made illegal by P_2.

tree so far is non-minimal. Conversely, if no candidate remains then the tree is minimal and the procedure proceeds. Thus MinALP builds only minimal clause trees.

In [12] we asked whether MinALP was complete, but we know now it is not. Even if P is legal when the subtree below h is complete, it may become illegal by a left path chosen later in the construction. In Fig. 31 the bad path P is legal before P_2 is chosen and so MinALP would have stopped. After choosing P_2, the set $\{P, P_1, P_2\}$ is illegal, and the tree in Fig. 31 is minimal, although it would not be found by MinALP. One could amend MinALP so that it rejects a tree if any remaining candidate bad path P is legal now and can never become illegal; otherwise it proceeds. It is guaranteed that P can never become illegal if no left path yet to be chosen can go over the nearest ancestor support of h and stop on P. This, in turn, is guaranteed in any of at least two ways: if the nearest ancestor support of h is h itself, or if there are no open leaves to the right of all ancestors of the nearest ancestor support of h, which is typically when the proof is nearly complete. Unfortunately, this amended MinALP does not necessarily generate only minimal clause trees, because a left path that could have made a candidate bad path illegal, may never actually be chosen.

7.3. Pseudocode for the clause tree procedures

Procedure 42 corresponds to the weak model elimination procedure [21]. This procedure is illustrated in Fig. 5 and described in Section 3.1. In that section R selects the rightmost open atom node. Such an R is always depth first. However, R does not need to be depth first for this procedure (Theorem 19).

Procedure 42 (*Weak model elimination with clause trees*).

Initially T is the clause tree from the top clause C.

1. [*Start*] Let the current atom node $g = R(T)$ be the rightmost open node.
2. [*Tautology*] If a tautology path to an ancestor of g exists, fail.
3. [*Merge*] If a merge path to an ancestor of g exists then choose this path and go to *Continue*. If a unifiable merge path to an ancestor of g exists, then non-deterministically either go to *Extension* or do the following: find the most general unifier θ of the label of g and the label of the head of the path, apply θ to the label of all atom nodes in T, choose this path and go to *Continue*.
4. [*Extension*] If any elementary clause tree from S that has no variables in common with T can be resolved with T at g, then non-deterministically select such a clause tree T_1. If none exist, fail. Let θ be the most general unifier of the label of g and the label of the selected atom in T_1. Resolve T with T_1, and then apply θ to the label of all the atom nodes. Go to *Continue*.
5. [*Continue*] Now g is a closed node. If there are no more open nodes, exit with success. Otherwise go to *Start*.

Procedure 43 corresponds to SL [16] and is described in Section 3.3. An illustration of this procedure where R selects the rightmost open atom node is given in Fig. 8. R does not need to be depth first here, but that is one way to ensure that the head of each left path and right path is a sibling of an ancestor of its tail. This condition guarantees

that the paths are legal, by Theorem 20. There are other ways to guarantee this. SLI maintains for each open literal its potential factors in the set γ, the siblings of its ancestors. Stickel [37] notes that if two goals do not have a common provable instance then factoring them is a waste of time. This inefficiency can be avoided in Procedure 43 by selecting the head of a hook as soon as the hook is chosen [28].

Changes between Procedure 43 and Procedure 42 are shown in **boldface.** This convention will be continued between successive procedures in this section to highlight the development of the procedures.

Procedure 43 (*SL with clause trees*).

Initially T is the clause tree from the top clause C.

1. [*Start*] Let the current atom node $g = R(T)$ be the rightmost open node.
2. [*Tautology*] If a tautology path to an ancestor of g **or any open leaf** exists, fail.
3. [*Merge*] If a merge path to an ancestor of g **or any open leaf** exists then choose this path and go to *Continue*. If a unifiable merge path to an ancestor of g **or any open leaf** exists, then non-deterministically either go to *Extension* or do the following: find the most general unifier θ of the label of g and the label of the head of the path, apply θ to the label of all atom nodes in T, choose this path and go to *Continue*.
4. [*Extension*] If any elementary clause tree from S that has no variables in common with T can be resolved with T at g, then non-deterministically select such a clause tree T_1. If none exist, fail. Let θ be the most general unifier of the label of g and the label of the selected atom in T_1. Resolve T with T_1, and then apply θ to the label of all the atom nodes. Go to *Continue*.
5. [*Continue*] Now g as a closed node. If there are no more open nodes, exit with success. Otherwise go to *Start*.

Procedure 44, without using tautology paths to visible internal nodes in step 2, corresponds to GC and is described in Section 3.4. An illustration where R selects the leftmost open atom node is given in Fig. 9. For Procedure 44, R should be depth first because the *Ancestor Merge* step does not check whether the ancestor path is legal. If R were not depth first, this check would be needed, but then the algorithm would not always generate ALP proofs as in Definition 48. Furthermore, Letz et al. [17, Proposition 7.1] have shown that this is not complete for arbitrary selection functions.

Procedure 44 (*ALP with clause trees*).

Initially T is the clause tree from the top clause C.

1. [*Start*] Let the current atom node $g = R(T)$ be the **left**most open node.
2. [*Tautology*] If a tautology path to an ancestor of g or any **visible internal node** exists, fail.
3. [*Merge*] If a merge path to an ancestor of g or any **visible internal node** exists then choose this path and go to *Continue*. If a unifiable merge path to an ancestor of g or any **visible internal node** exists, then non-deterministically either go to *Extension* or do the following: find the most general unifier θ of the label of g

and the label of the head of this path, apply θ to the label of all atom nodes in T, choose this path and go to *Continue*.

4. [*Extension*] If any elementary clause tree from S that has no variables in common with T can be resolved with T at g, then non-deterministically select such a clause tree T_1. If none exist, fail. Let θ be the most general unifier of the label of g and the label of the selected atom in T_1. Resolve T with T_1, and then apply θ to the label of all the atom nodes. Go to *Continue*.
5. [*Continue*] Now g as a closed node. If there are no more open nodes, exit with success. Otherwise go to *Start*.

Like Procedure 44, MinALP should use a depth first selection function R. In MinALP this is important also because the check for minimality in step 5 assumes that the subtree below h has no open nodes. We use a leftmost selection function R, although this can be relaxed to allow any of the deepest open nodes to be selected first.

Procedure 45 (*MinALP with clause trees*).

Initially T is the clause tree from the top clause C.

1. [*Start*] Let the current atom node $g = R(T)$ be the leftmost open node.
2. [*Tautology*] If a tautology path to an ancestor of g or any visible internal node exists, fail.
3. [*Merge*] If a merge path to an ancestor of g or any visible internal node exists then choose this path and go to *Continue*. If a unifiable merge path to an ancestor of g or any visible internal node exists, then non-deterministically either go to *Extension* or do the following: find the most general unifier θ of the label of g and the label of the head of this path, apply θ to the label of all atom nodes in T, choose this path and go to *Continue*.
4. [*Extension*] If any elementary clause tree from S that has no variables in common with T can be resolved with T at g, then non-deterministically select such a clause tree T_1. If none exist, fail. Let θ be the most general unifier of the label of g and the label of the selected atom in T_1. Resolve T with T_1, and then apply θ to the label of all the atom nodes. Go to *Continue*.
5. [*Continue*] Now g is a closed node.

   For each ancestor h **of** g **that has no open descendants do**
   if some internal atom node t **with the same label as** h **can see** h **then**
   fail
   endif
   end for

   If there are no more open nodes, exit with success. Otherwise go to *Start*.

The ordered clause set restriction can be implemented with any of Procedure 42, Procedure 43, Procedure 44 and Procedure 45. In Procedure 42, replace the *Merge* step with the following. Here $\leqslant$ refers to the total order on the set of clauses. In the other procedures, the ancestor merge step is the following, but other merge steps are unaffected.

[*Merge*]
If a merge path to an ancestor of g exists then
if the head clause of the merge path $\leqslant$ **the tail clause then**
choose this path and go to *Continue*.
else
fail
endif
elseif a unifiable merge path to an ancestor of g exists then
if the head clause of the merge path $\leqslant$ **the tail clause then**
non-deterministically either go to *Extension* or do the following: find the most general unifier θ of the label of g and the label of the head of the path, apply θ to the label of all atom nodes in T, choose this path and go to *Continue*.
else
go to *Extension*
endif
endif

The above procedures can be seen as special cases of a generic procedure, Procedure 46. This procedure makes use of all visible internal nodes for merges and tautologies, and it puts no restriction on its selection function.

Procedure 46 (*Generic procedure for clause trees*).
Initially T is the clause tree from the top clause C.
1. [*Start*] Let the current atom node $g = R(T)$. **Let** V **be some nodes visible from** g.
2. [*Tautology*] If a tautology path to **any member of** V exists, fail.
3. [*Merge*] If a merge path to **any member of** V exists then choose this path and go to *Continue*. If a unifiable merge path to **any member of** V exists, then non-deterministically either go to *Extension* or do the following: find the most general unifier θ of the label of g and the head of the path, apply θ to the label of all atom nodes in T, choose this path and go to *Continue*.
4. [*Extension*] If any elementary clause tree from S that has no variables in common with T can be resolved with T at g, then non-deterministically select such a clause tree T_1. If none exist, fail. Let θ be the most general unifier of the label of g and the label of the selected atom in T_1. Resolve T with T_1, and then apply θ to the label of all the atom nodes. Go to *Continue*.
5. [*Continue*] Now g is a closed node. If there are no more open nodes, exit with success. Otherwise go to *Start*.

The AllPaths procedure is Procedure 46 when R is a depth first, left to right selection function and V is all visible nodes. An AllPaths clause trees is one where all paths are either left paths, right hooks or ancestor paths. In addition it has the usual restrictions on unchosen merge and tautology paths: no ancestor paths, right hooks or legal left paths are tautology paths or unchosen merge paths. A path between a node and a sibling of one of its ancestors may be either a right hook or, reversed, a left path. This introduces

some redundancy in the definition so we add the restriction that the left paths in AllPaths trees are not reversed right hooks. It has recently been shown [11] that a closed AllPaths clause tree on an unsatisfiable set of clauses must exist. Hence the AllPaths procedure is complete. These clause trees are closely related to the folding up and folding down operations on connection tableaux [17]. We are investigating procedures for effectively building AllPaths clause trees.

All of these procedures can be improved somewhat by checking for tautology paths in the extension step from the new open leaves. This may expose tautology paths earlier, and so prevent some work. For readability, we have decided to present the searches for tautology paths from the selected literal, and show them as separate steps. In the propositional case, a tautology check in the extension step can take the place of one from the selected node. However, in the first order case, both tautology checks may be needed. The check from the selected node may find tautology paths that were only unifiable tautology paths when the check was done in the extension step. By the same reasoning, tautology paths and unchosen merge paths can arise at any point in build ordering. Detecting this and dealing with it is the subject of ongoing research [15] on the *disequality* strategy. See Section 9.1.

7.4. Completeness results

It is well known that ME and SL are complete. GC and a procedure equivalent to ALP have been shown to be complete [17, Theorem 8.2]. We provide an alternative proof which can easily be extended to the ALPOC procedure. First, a structural definition for an ALP clause tree is given which characterizes the type of clause tree that the ALP procedure constructs. Such a tree must exist if S is satisfiable and $S \cup \{C\}$ is unsatisfiable. We treat the propositional case, add the ordered clause set restriction, and then lift to first order logic.

All procedures in this section are sound because they construct closed clause trees. This is guaranteed because every step in the construction is an extension with an elementary clause tree from an input clause, or a merge path to a visible literal.

We start with the following lemma.

Lemma 47. *For propositional logic, in any closed clause tree produced by ME (Procedure 42), SL (Procedure 43), ALP (Procedure 44), or MinALP (Procedure 45), or the ordered clause set restriction of these, the top clause can only occur once, at the root.*

Proof. Suppose not, that the top clause C also occurs elsewhere in the proof at a node n. Consider the first edge on the path from the root to n. Let a be the literal associated with that edge. Then a is in C. Hence a must also be associated with an edge incident with n. If a occurs below n, then a tautology path exists from there to the atom node of a adjacent to the root. But all these algorithms reject such a path and could not construct this clause tree. Thus a must be associated with the edge above n. But then a path joining the two atom nodes labeled a, one adjacent to n and one adjacent to the root, is

a merge path. This path would have been chosen by any of these algorithms, and n could never have been extended onto the clause tree, a contradiction. □

Definition 48 (*ALP clause tree*). Given a set S of clauses, an *ALP clause tree on* S is a closed rooted clause tree on S that meets the following conditions:

(1) All chosen merge paths are either left paths or ancestor paths.
(2) No ancestor paths are tautology paths or unchosen merge paths.
(3) No legal left paths are tautology paths or unchosen merge paths.

Theorem 49. *Any ALP clause tree on* $S \cup \{C\}$ *with top clause* C *is an instance of a clause tree that can be constructed by the ALP procedure* (*Procedure* 44). *Conversely for propositional logic, the ALP procedure constructs only ALP clause trees on* $S \cup \{C\}$.

Proof. Assume we have an ALP clause tree on $S \cup \{C\}$. Consider each atom node n in the tree. Assume without loss of generality that the depth first selection function R selects nodes from left to right. As an induction hypothesis we assume that the clause tree consisting of the nodes to the left and above n are constructed by the ALP procedure. When n is selected, the tree is not rejected by step 2 of Procedure 44 since there are no left tautology or ancestor tautology paths in the clause tree. If the atom node is the tail of a merge path, the procedure would find and choose the merge path in step 3. Otherwise the procedure would non-deterministically choose a clause containing a unifiable complementary literal to extend the tree. In particular it could choose the clause labeling the clause node immediately below the current atom node.

Conversely, the procedure only looks for left paths and ancestor paths. Any left path or ancestor path is considered when the tail is the current node. If a legal tautology path is detected, the clause tree is rejected, so the resulting clause trees can have no such tautology paths. Similarly any such legal merge path is detected and chosen, so the resulting clause tree cannot have an unchosen merge left or ancestor path when the tail is the current node. Since all of the atoms are propositional, no unifiable merge (tautology) path can become a merge (tautology) path later in the construction. □

The converse of Theorem 49 does not necessarily hold in the case of first order logic. The tree generated by ALP may not satisfy condition (2) or condition (3) of Definition 48. Suppose an ancestor merge path P in the tree is not chosen in the non-deterministic part of the ancestor merge step, because when the tail t of P was the selected node, the head of P was not identical to t. Subsequent substitutions applied to the labels may make t and h identical, but the algorithm does not check for this.

Theorem 50 (Completeness of propositional ALP). *Given* S, *a satisfiable set of propositional clauses, and* C, *a propositional clause such that* $S \cup \{C\}$ *is unsatisfiable, there exists an ALP clause tree with the clause node labeled* C *as its root.*

Proof. We use induction on the number of atoms in the set of atoms in $S \cup \{C\}$. Let this set be the singleton set $\{a\}$. Then either $S = \{\{a\}\}$ and $C = \{\sim a\}$ or $S = \{\{\sim a\}\}$ and $C = \{a\}$. In either case, there is an ALP clause tree with one atom node labeled a and two clause nodes of degree one.

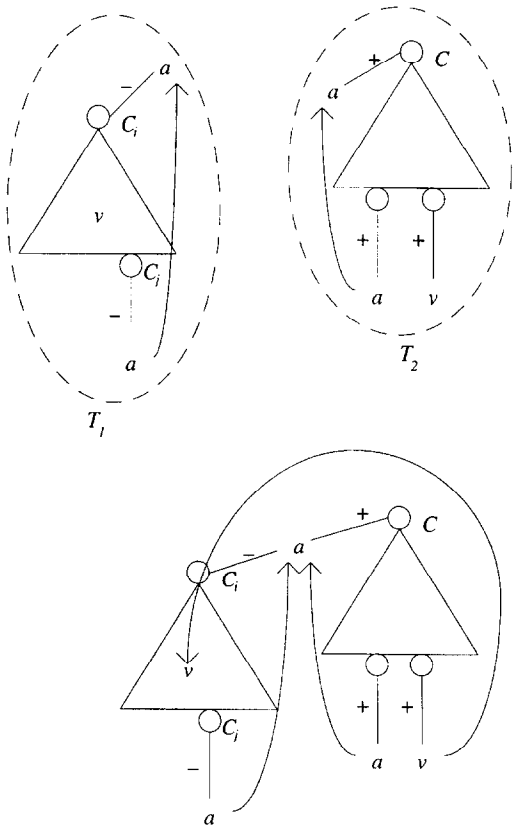

Fig. 32. The construction of an ALP tree from T_1 and T_2.

Let there be n distinct atoms in $S \cup \{C\}$, and let the leftmost literal of C be the atom a. Let $S_1 = \{X - \{\sim a\} \mid X \in S \text{ and } a \notin X\}$. Suppose there exists a model of S_1. By extending this model so that it maps a to true, we have constructed a model of $S \cup \{C\}$, a contradiction. Therefore S_1 is unsatisfiable.

Define $S_2 = S \cap S_1$. $S_2 \subseteq S$ so it is satisfiable. Let $S_1 - S_2 = \{C_1, \ldots, C_n\}$. Then find the smallest i such that $S_2 \cup \{C_1, \ldots, C_i\}$ is unsatisfiable. Let $S_3 = S_2 \cup \{C_1, \ldots, C_{i-1}\}$. By induction there exists an ALP clause tree on $S_3 \cup \{C_i\}$ with top clause C_i. For each clause node corresponding to some C_j in this tree attach a new atom node labeled a with an edge labeled $-$. Consider the new atom node added to the top clause node and make it the new root. Now all clause nodes in this tree correspond to clauses in S. Choose merge paths from these new open leaves to the new root. Name the resulting tree T_1. See Fig. 32.

Consider the clause tree that results when T_1 is resolved with an elementary clause tree for the singleton atom clause $\{a\}$. Let V be the set of literals visible from the root of this tree. Note that a is in V. Let

$$S_4 = \{X - V \mid X \in S \text{ and } X \cap \{x \mid \sim x \in V\} = \phi\}.$$

Suppose S_4 is unsatisfiable. Then there is a closed clause tree T' on S_4. For each clause node in S_4 labeled by a clause not in S, attach atom nodes labeled with atoms from V as appropriate to restore it to being a clause from S. Then the clause of this new tree is $C' \subseteq V$. Thus $S \models C'$ and so $S \models V$. Construct an elementary clause tree for V, resolve it with T_1 at a and choose merge paths as necessary from V into T_1 so that no open leaves remain. This tree ensures that $S \cup \{V\}$ is unsatisfiable, and since $S \models V$, it follows that S is unsatisfiable, a contradiction. Therefore S_4 is satisfiable.

Let $C'' = C - V$. If C'' is empty then T_1 can be resolved with C on a and all literals in C other than a can be merged with left paths to visible literals in T_1. The result is a closed clause tree. Otherwise C'' is not empty. Suppose there is a model of $S_4 \cup \{C''\}$. Extend it to a model that maps a and all literals in V to false. Thus this model satisfies any clause in $S - S_4$. It also satisfies C since $C'' \subseteq C$. Therefore $S \cup \{C\}$ is satisfied by it, a contradiction. Therefore $S_4 \cup \{C''\}$ is unsatisfiable.

By induction there exists an ALP clause tree on S_4 with top clause C''. For each clause node in the tree labeled by a clause not in S, attach new atom nodes labeled with atoms from V as appropriate to restore it to being a clause in $S \cup \{C\}$. Choose left (hook) merge paths from each atom node labeled a to the atom node labeled a in the top clause. Let T_2 be the name of this new clause tree. See Fig. 32.

Now resolve T_1 and T_2 at the atom node labeled a and choose left merge paths as necessary from T_2 to the visible nodes in T_1 to close all open leaves in T_2. The result is a closed clause tree. It is also satisfies condition (1) of Definition 48 because all paths are either provided by induction or are ancestor paths or left paths. Condition (2) is satisfied because all new ancestor merge paths merge the atom a in T_1, which only occurs as leaves of T_1. Also no tautology ancestor paths are introduced to T_1 because the literal a does not occur in T_1 (because all occurrences are negative). Condition (3) is satisfied because the construction of T_2 eliminates all literals complementary to those in V, so no tautology paths can occur. Furthermore all merge paths from T_2 to T_1 are chosen. □

Definition 51 (*ALPOC clause tree*). Consider an ordered set of clauses $S = \langle C_1, \ldots, C_n \rangle$. An *ALPOC clause tree* on S is an ALP clause tree in which each ancestor chosen merge path satisfies the ordered clause set restriction. That is, if the tail clause is C_i, and the head clause is C_j, then $j \geqslant i$.

Theorem 52. *ALPOC is complete for propositional logic.*

Proof. The proof requires just a slight addition to the proof of Theorem 50. The same induction works, with ALP clause tree replaced by ALPOC clause tree. The only extra requirement is that the ancestor chosen merge paths satisfy the ordered clause set restriction. This can be accomplished by ordering the clauses that are vying for top clause of T_1 (clauses added to S_3) in the reverse order to that required for the ordered clause set restriction. Then the top clause precedes any clause which contains the literal a in the rest of T_1, and so the chosen ancestor clauses with the top clause as head clause automatically satisfy the ordered clause set restriction. As this is the only way that

ancestor paths are chosen other than recursively, the ordered clause set restriction is satisfied. □

In the propositional case, a stronger version of the ordered clause set restriction can be used: for each ancestor path if the tail clause C_i and the head clause is C_j then $j > i$. To justify this, it suffices to apply Theorem 50 to each clause tree T_1 built by induction. The top clause cannot appear elsewhere in the clause tree, so $j \neq i$. This stronger restriction can also be applied in the first order logic case, when the head and tail clauses of a path are identical instances of the same clause. The proof of Theorem 50 suffices here.

Theorem 53. *ALP and ALPOC are complete for first order logic.*

Proof. Let S be a satisfiable set of clauses and let C be a clause such that $S \cup \{C\}$ is unsatisfiable. We shall show there is a closed ALPOC tree on $S \cup \{C\}$ with top clause C. There is a closed clause tree T on $S \cup \{C\}$. Construct the one-to-one mapping ψ from distinct instances of atoms in T to new and distinct propositional atoms. Apply ψ to the atom labels of T and the resulting clause tree T' is based on the set of propositional clauses $S' \cup C'$, where each clause of S' is derived from a clause of S and $C' = \{C_1, \ldots, C_n\}$ are the clauses derived from C in the same manner. $S' \cup C'$ is unsatisfiable, because $cl(T') = \phi$. Suppose S' is unsatisfiable. Then a closed clause tree on S' exists. By applying the inverse of ψ to it, one finds a closed clause tree on S. Thus S is unsatisfiable, which contradicts the hypothesis. Therefore S' is satisfiable. Select the minimum i such that $S'' = S' \cup \{C_1, \ldots, C_{i-1}\}$ is satisfiable, but $S'' \cup \{C_i\}$ is unsatisfiable. By completeness of the ALPOC procedure there is a tree T^* on $S'' \cup \{C_i\}$ with C_i as the top clause. By applying the inverse of ψ to T^*, a closed ALPOC clause tree on $S \cup \{C\}$ is constructed with C as its top clause.

Since any ALPOC tree is also an ALP tree, ALP is complete. □

Note that the proof of the Theorem 53 allows some unifiable merge paths to remain unchosen. There can be a legal left or ancestor unifiable merge path, which is not a merge path in T^*. However no other instance of C can exist in T^* that is identical to the top clause (Lemma 47).

7.5. Three counterexamples

For any proof format, it is important to ask whether all smallest proofs are eliminated, since many proof procedures use the size of the proof as an important criterion to guide their search. For example, PTTP [34] and SETHEO [18] use iterative deepening. The following two theorems show that ALP may increase the size of proofs and that ALPOC may increase them more.

Theorem 54. *A smallest ALP clause tree on a set of clauses may not be a smallest clause tree.*

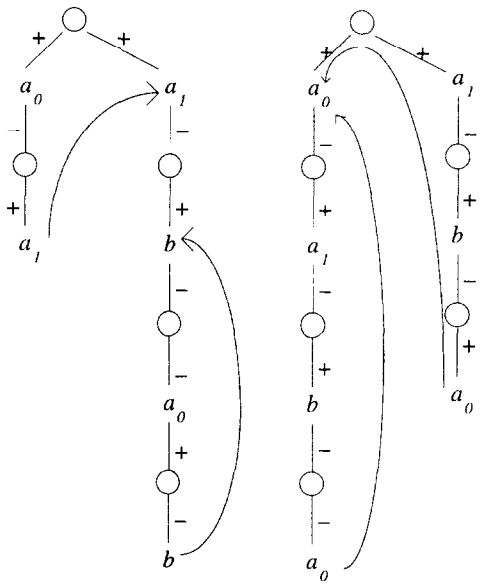

Fig. 33. A smallest proof and a smallest ALP proof.

Proof. An example is given by the following, where the first clause is the top clause and the literals within a clause are ordered from left to right:

$\{a_0, a_1\}$,

$\{\sim a_0, a_1\}$,

$\{\sim a_1, b\}$,

$\{\sim b, \sim a_0\}$,

$\{\sim b, a_0\}$.

See Fig. 33. □

It is known that both the foothold score restriction and the ordered clause set restriction can increase the size of ME proofs; that is, the smallest MEOC proof may be larger than the smallest ME proof. Because left paths can be used to shrink the size of proofs, it may seem that a smallest ALP proof is the same size as a smallest ALPOC proof. However, this is not the case.

Theorem 55. *A smallest ALPOC clause tree may be larger than a smallest ALP clause tree.*

Proof. An example is given by these clauses, where the first clause is the top clause.

1. $\{\sim q\}$,
2. $\{q, a_0, a_1\}$,
3. $\{\sim a_0, a_1\}$,
4. $\{\sim a_1, b, q\}$,
5. $\{\sim b, \sim a_0\}$,
6. $\{\sim b, a_0\}$,

See Fig. 34. □

Section 5.1 points out that for first order logic, the ordered clause set restriction fails to orient all ancestor paths. An ancestor path whose head clause is the same as its tail clause can be used in either direction. Here we discuss a generalization of the ordered clause set restriction: orient all paths lexicographically and include all the clauses, and also all the literals, on the path. Since a merge path cannot be perfectly symmetric, all merge paths are oriented by such a scheme. Unfortunately it is not possible to orient all paths at the same time by this scheme. In the example, given in Fig. 35, there are three merge paths, all oriented by this more general ordered clause set restriction. We may choose any one of the three leaves to be the head of these paths so that two of the paths

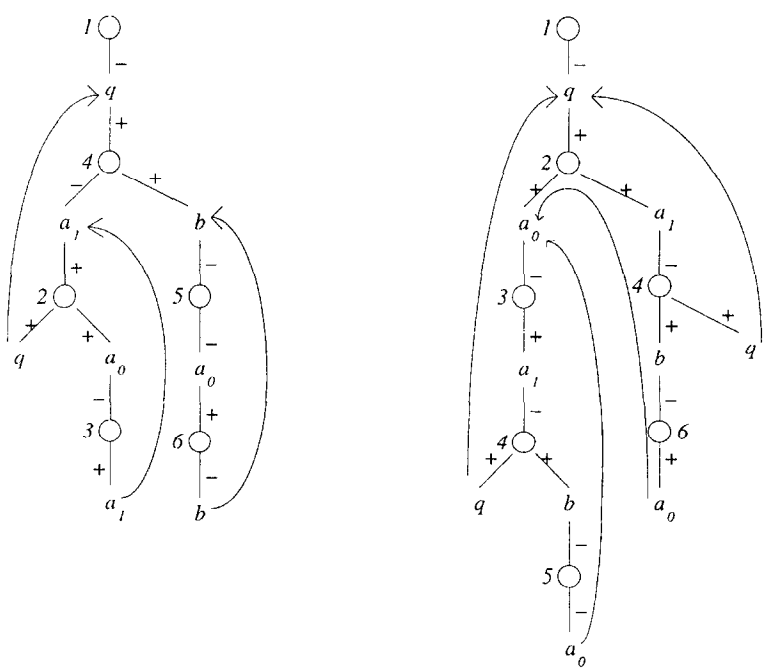

Fig. 34. A smallest ALP proof and a smallest ALPOC proof.

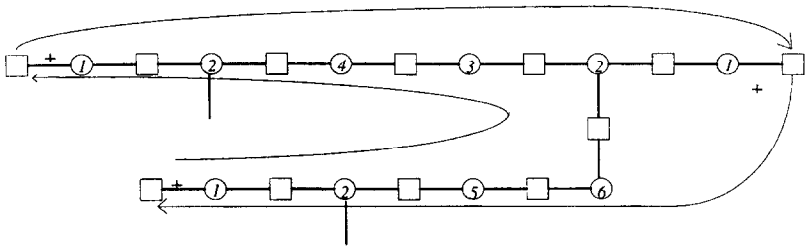

Fig. 35. This clause tree cannot obey the more general ordered clause set restriction.

become ancestor paths. But no matter which leaf we choose, one of the two chosen paths is oriented incorrectly.

8. Mergeless clause trees

The concepts of unit resolution, input resolution, and resolution on relative Horn set of clauses are known to be equivalent in a broad sense [5,21]. This section is dedicated to proving these equivalences by proving each of them is equivalent to the concept of mergeless clause trees.

First recall the following definitions (see [21]). Let S be a set of clauses. A *unit refutation* of S is a resolution proof of ϕ from S in which one parent of each resolution is a unit clause, consisting of a single literal. An *input refutation* of S is a resolution proof of the empty clause from S in which one parent of each resolution is the result of the previous resolution (linearity) and the other is a factor of an input clause. S is a *relative Horn set* of clauses if there is a set of literals (a *setting*) H, closed under substitution, in which no pair of complementary literals occur, and every clause of S contains at most one literal not from H.

Theorem 56. *Let S be a set of clauses. The following statements about S are equivalent.*

(a) *S has an input refutation.*
(b) *S has a unit refutation.*
(c) *The set of factors of S contains a relative Horn subset which is unsatisfiable.*
(d) *There exists a mergeless closed clause tree on S.*

Proof. The first three statements are well known to be equivalent. However we will prove each is equivalent to the last statement, as examples of how easy it is to work with clause trees.

(a) implies (d). Assume that S has an input refutation. Build a clause tree using the same steps as the refutation, but with the following modification. Whenever two literals are merged in the proof, instead of choosing a merge path to put in the tree, leave the two merged literals as open leaves identically labeled. When an input clause resolves on a literal, attach a clause node corresponding to that clause to all open leaves that are labeled with the literal being unified. At each step of the proof, the clause tree corresponds to the current clause in the proof, but with possibly many copies of each

literal. The final clause tree is closed as the final clause is empty, and the clause tree is mergeless since no merge paths are ever chosen.

(d) implies (a). Assume that S admits a mergeless clause tree. Order the atom nodes using any tree search sequence as long as the next node is adjacent to a node that has already been searched. Depth first and breadth first search are both acceptable. Perform the resolutions in this order. Each step resolves an input clause with the previous clause, so the proof is immediately an input refutation.

(b) implies (d). Assume that S has a unit refutation. Mirror the refutation by building clause trees using the same resolutions. Since unit clauses correspond to clause trees with a single open leaf, no clause tree generated by these resolutions can ever have two open leaves that are adjacent to different clause nodes. Thus if a merge is performed in the unit refutation, the literals merged must have come from the same input clause, in which case they could have been merged before being used in a resolution. Thus the instance of the input clause can be used in the construction of the clause tree, and no merge paths are needed. Thus the resulting closed clause tree is mergeless.

(d) implies (b). Let T be a closed mergeless clause tree based on factors of S. Number the atom nodes of T from 1 to n, the number of atom nodes, starting at any atom node and follow any tree search algorithm, as in the proof of (d) implies (a) above. Consider the resolution proof based on doing the corresponding resolutions in reverse order. When any atom node is processed, all resolutions on one side of the atom node will all have been processed already, as they must all be numbered higher in the tree search sequence. Thus that one side always corresponds to a clause tree with a single open leaf. Hence the subtree on that side corresponds to a unit clause. The proof is a unit refutation since each resolution uses a unit clause, and the last clause that corresponds to T is empty.

(c) implies (d). Assume that S is an unsatisfiable Horn set relative to a set H of literals. We may assume that for any ground literal, either it or its complement is in H. Since S is unsatisfiable, there must be a closed clause tree T based on S. Without loss of generality we can assume that T is based on ground clauses, by substituting a constant for each variable. At least one of the clause nodes of T must correspond to a clause without a literal in H, because otherwise all the nodes of T could be satisfied by setting all the literals in H to be true. Root T at this node. All the edges adjacent to the root correspond to literals that are not in H, hence the edges beyond the adjacent atom nodes correspond to literals in H. But these must be the only literals in their clauses that are in H, so the literals corresponding to edges adjacent to them are not in H. Continuing in this way, one sees that a literal corresponding to the edge towards the root at any atom node is not in H, whereas the literal corresponding to the edge away from the root at any atom node is in H.

Now consider the chosen merge paths in T. None of these merge paths can be ancestor paths since the edge on the path closest to the root must correspond to a literal in H, whereas the edge furthest from the root must not be in H. Then these two literals cannot be unified. Thus all merge paths are either left paths or right paths. Order the paths totally, in reverse order to the precedes relation, with the added constraint that if two paths are not comparable by the transitive closure of the precedes relation, then the path with its head further from the root must come earlier in the order. Thus the first

path in this order cannot have its head on any other chosen merge path, and there can be no path below the head of the first path in the tree. Supplant each of the paths in this order. When a path is supplanted, there is no part of a path in the subtree at the head of the path since such a path would have to be processed first. After all the paths are processed, the resulting clause tree is mergeless, and still closed.

(d) implies (c). Assume that T is a closed mergeless clause tree on S. Without loss of generality we can assume that T has no unifiable tautology paths, since if it did, the tautology path could be removed by internal surgery. Root T at any clause node. For each atom node n consider the path $P(n)$ from n to the root. If $P(n)$ passes over an atom node m that is unifiable with n, then perform tail surgery at $path(m, n)$. It does not matter that T is no longer mergeless. For all other atom nodes other than the tail of merge paths, place the literal corresponding to the edge above the atom node into a set H. H cannot contain two complementary unifiable literals. If the path P joining the corresponding atom nodes were an ancestor path, it would have been already turned into a merge path, whereas if it were a left or right path, it would then have been a unifiable tautology path in the original clause tree. Note that every clause node in T now contains precisely one literal not from H, except for the root which does not contain any. Thus the set of clauses corresponding to the clause nodes of T, which are a set of factors of S, form a Horn set relative to H, and are unsatisfiable. □

Corollary 57. *If T is a closed mergeless clause tree from S then there is unit refutation which, if the order of refutations is reversed, becomes an input refutation of S.*

9. Future work and conclusions

This paper contains several references to ongoing work and raises questions for future research. These are collected and expanded in this section, along with a summary of the contributions.

9.1. Open problems and future work

Open question 1 (*Completeness of ALPF* (*ALP with the foothold score restriction*)). Let S be a satisfiable set of clauses and let C be a clause such that $S \cup \{C\}$ is unsatisfiable. There is a sign ordered ALP clause tree on S with top clause C such that all ancestor paths are proper.

If Open question 1 is true then ALPF is an improvement on ALPOC because it orients all ancestor paths, including those with the same clause at the head and tail, which ALPOC does not orient.

Open question 2. Is there a complete, top down procedure that will generate only minimal clause trees, no two of which are reversal equivalent, and takes advantage of the reduced search space?

We want to exclude from consideration those procedures that use a generate and test approach to building only minimal clause trees. For instance, one could propose the ALP procedure followed by a test for minimality of the constructed tree. Such a procedure would require a large amount of additional searching of the constructed tree, without any associated benefit of reducing the search to construct it. Ideally only a small amount of checking the constructed tree would be required to ensure it is minimal.

Open question 3. Is there a polynomial p and a complete, top down procedure that will generate only (minimal) clause trees (no two of which are reversal equivalent) whose size is $p(n)$ where n is the size of a smallest clause tree on the set of input clauses?

Many implementations of theorem provers using clause tree and empirical investigations are needed. We have already developed a compiler based on PTTP to build ALPOC clause trees. We describe planned extensions to it, to use disequalities, and to build AllPaths trees. The experiments will use the TPTP problem library and a system we have written to automatically run a large number of jobs in batch mode and to collect and summarize the results [33].

It was reported in Section 7.4 that the ALP procedure on a set of first order clauses may generate non-ALP clause trees that contain unchosen merge paths or tautology paths. The check for these paths is done when the tail of the path is the current node, and at that time the path may have been a unifiable merge or tautology path. Since later substitutions can convert unifiable paths into merge or tautology paths, we are required to check again later if we want to avoid unchosen merge and tautology paths. Hynes [15] has implemented an ALP meta-interpreter in Prolog that creates a list of disequalities. A *disequality* is a pair of nodes, not labeled identically, at the head and tail of a unifiable tautology path or an unchosen unifiable merge path. The procedure maintains the list of disequalities encountered so far in the tree, and adds to this list any new disequalities from the current node. Whenever a substitution is applied, the list of disequalities is checked and if any pair now has identical labels, the tree is rejected. The current implementation applies only to ALP, and uses an iterative deepening search strategy and chronological backtracking. Preliminary experiments show that for some examples, up to 95% of the inferences are avoided, and that the number of disequalities is commensurate with the size of the tree. Work is ongoing to compare the benefits with the overhead of building and checking the disequality list. These disequalities are similar but not identical to the syntactic inequality constraints used in SETHEO [18].

The AllPaths procedure, which builds clause trees with ancestor and left paths as well as right hooks, has recently been shown to be complete. Any implementation of it must include some method preventing a left path from running over the head of an existing right hook. An existing AllPaths prototype for propositional logic [27] will be extended to a compiler to Prolog, in the PTTP style, and its effectiveness will be measured against PTTP and ALPOC.

We are using the flexible choice of selection functions offered by clause trees to improve the performance of an SL clause tree procedure. Sharpe [28] has implemented a system that replaces the usual depth first selection function with one that selects the head of a left hook as soon as the path is chosen. We call this the *head first* selection

function since it chooses the head of a right hook before other open leaves. Stickel points out that factoring used in SL often increases the search space without a compensating decrease in proof size, because the unified goals are over-instantiated and not usable in a proof [34]. By selecting the head of a left hook as the next provable goal, the system determines immediately whether or not the two factored goals have a common provable instance.

Other future work on the compiler may incorporate the use of lemmas, caches and failure caches [2,3]. Our intention is to produce one system that can generate different code depending on what options are selected, including left paths, the ordered clause set restriction, the foothold score restriction, disequalities, right hooks, AllPaths, head first selection for right hooks and lemmas.

Minimal clause trees, with all chosen merge paths restricted to being oriented in only one way, make progress towards defining a canonical form for binary resolution proofs. In the ALP family of procedures, non-ancestor chosen merge paths are forced to be left paths rather than right, and ancestor chosen merge paths can be restricted by the ordered clause set restriction or the foothold score restriction. Thus finding an effective, complete procedure for building minimal clause trees is an important question.

However in bottom up procedures, representative minimal clause trees are a canonical form as all chosen merge paths are oriented. By retaining only representative minimal clause trees, we address the problem (#6) of redundancy posed by Wos [40]. Non-minimal and non-representative clause trees are redundant, and we can detect this type of redundancy by analysing the tree. Other bottom up systems, such as OTTER use subsumption to check for the same redundancy. The cost of a subsumption check increases with the number of retained clauses. Thus the cost of producing a set of clause trees that is not redundant should grow more slowly than the cost of producing such a set of clauses with binary resolution. Any bottom up resolution based procedure can be turned into a procedure that produces only representative clause trees. Once a non-minimal clause tree is found, it can be made minimal by performing surgery, or the procedure may choose to simply reject that clause tree. The first strategy can be used with a complete bottom up procedure, and will preserve completeness. The second will preserve completeness if the procedure produces some minimal clause tree. Other procedures are being investigated [13,14].

The equality relation plays a special role in first order logic. Paramodulation is one technique for handling it. Paramodulation inference can be implemented in clause trees using paths to justify equality substitutions analogously to the way in which merge paths justify factoring.

Another open question is how to extend clause trees to handle proofs that use formulas other than just clauses, as done in [24]. Many propositional satisfiability problems can be solved only with an exponential number of resolution steps [4,6]. Hence these problems admit only exponentially large closed clause trees. However, such problems can sometimes be solved in polynomial time [4,7,39]. One method that allows the pigeonhole problem to be solved polynomially is to allow a single variable to replace a clause [9]. The negation of a clause is a conjunction, so that conjunctions can be dealt with. How such substitutions can be represented as clause trees, and when such substitutions should be performed are two more open questions.

9.2. Summary and conclusions

Clause trees offer new insights for implementing and understanding binary resolution. The distinction between binary resolution with clauses and binary resolution with clause trees is rather small, from one point of view. With clauses, two operations are always performed together: the complementary literals resolved upon are removed, and any duplicate literals are merged together. In this paper we have uncoupled these two, so that a merge may be delayed until later, or some new literal may be merged with a literal previously resolved upon.

Despite this being a subtle difference, there are several interesting advantages. Often with binary resolution on clauses, the result from one sequence of resolution steps gives a result superior to that of the same steps done in a different order, because merge opportunities are missed in the second sequence. With clause trees, both sequences can give the superior result because merges can be done as necessary to remove literals. Thus one clause tree really represents multiple different sequences of binary resolution steps and we can dynamically choose among them to find the one with the best outcome. The interaction of merge paths in the clause tree provides the information from which we can decide, quite easily, if a merge is legal.

While merge paths unify the concepts of reduction inference steps and (different forms of) factorization inference steps in other frameworks, this does not prevent different kinds of merge paths from being distinguished in implementations. In particular, ancestor merge paths are used for ancestor resolution, merge paths that are hooks correspond to SL factoring, and left paths perform Shostak's c-reductions. Thus different types of merge paths are treated differently in the procedures of Section 7. From the viewpoint of proof theory, this uniformity is an advantage because the arguments, such as those in Section 8, are simplified.

One important result here is that the precedes relation on the chosen merge paths must be extendible to a partial order. From this result it follows quickly that ME and SL are sound. It also leads to the new concepts of visibility and support. With this deeper understanding of the internal structure of binary resolution we developed the ALP procedure which we soon realized was closely related to GC.

This paper introduces the minimal property of clause trees. This is a strong property which means, intuitively, that there does not exist a tautology clause, nor a missing merge in any of the binary resolution derivations that this clause tree represents. In other words, the clause tree contains no unnecessary steps which would make the tree redundant. For every non-minimal clause tree, there is a minimal clause tree that subsumes it. The operation of surgery on clause trees removes the unnecessary steps, leaving a result that is at least as general as the original tree. Clause tree surgery can be applied on partial as well as closed trees, and it can be applied successively until a minimal tree remains. We know of no other inference technique that, without rebuilding at least part of the proof, can improve the result of a proof as much as surgery can.

The top down procedures of ME, SL, and GC are explained in a unified manner. For these procedures, it is always permissible to reverse ancestor paths, so that the head is replaced by the tail. The ordered clause set restriction and foothold score restriction take advantage of this fact. They force the path to be oriented in one of the two ways and so

cut down on the space that must be searched by these procedures. This leads to *inter alia* the ALPOC procedure.

Two new ideas in proof procedures are presented. The ability to detect non-minimal clause trees has lead to the development of a new restriction, and new top down procedures, MinALP and MinALPOC, that use it. A family of proof procedures is introduced as Procedure 46, that can use any selection function, yet are tighter in that they take full advantage of goals in the tree already proved. This generalizes both SLI, which has an unrestricted selection function but cannot use C-literals, and GC which must use a depth first selection function. A variant of Procedure 46 that uses depth first selection is called AllPaths, and is known to be complete, although the proof is not included in this paper.

Unit resolution, input resolution and resolution on relative Horn sets are each shown to be identical to mergeless clause trees. The arguments are compact, yet include a number of known results. We take this as evidence that clause trees improve our understanding of resolution.

Two equivalent theorems [29] and [1, Theorem 2] were proved in the 1970's that are related to Theorem 13. Instead of using merge paths, their graphs use simple cycles to indicate a merge. Both theorems state that a graph represents a sound proof if and only if a certain type of cycle does not exist. Such a cycle exists if and only if, in the corresponding clause tree, the set of merge paths contains a circular set under the precedes relation. That is, the precedes relation is not extendible to a partial order. The advantage of working with trees, rather than graphs is that trees are simpler structures. For example, two nodes in the tree must be connected by a unique path. Using clause graphs Shostak developed the GC procedure, a significant improvement upon SL, in the 1970's. But it was not until 1994 that the first professional implementation of it appeared [17]. Letz et al. use the connection tableaux, which is a rooted tree structure, and the folding up procedure.

Using rooted clause trees, we have extended many top down procedures and made them tighter. But they require all ancestor paths to be free of duplicate atoms, except for the head and tail of chosen merge paths, while other paths are not so restricted. Thus paths in the tree are not treated identically. This has the effect of making some clause trees larger than necessary. Unrooted clause trees may have more potential than rooted trees because all paths can be treated in the same way. We believe strongly that the study of clause trees will provide many opportunities for the development of faster automated reasoning procedures.

Acknowledgments

The authors gratefully acknowledge the efforts of the two anonymous reviewers and members of the Automated Reasoning group at UNB, including Mike Boothroyd, Kelsey Francis, Rod Hynes, Mike Lamoureux, Charlie Obimbo, David Sharpe and Qinxin Yu, as well as NSERC for funding.

References

[1] P.B. Andrews, Refutations by matings, *IEEE Trans. Comput.* **25** (1976) 801–807.

[2] O.L. Astrachan and D.W. Loveland, METEORs: high performance theorem provers using model elimination, in: R.S. Boyer, ed., *Automated Reasoning: Essays in Honor of Woody Bledsoe* (Kluwer Academic Publishers, Dordrecht, 1991).

[3] O.L. Astrachan and M. Stickel, Caching and lemmaizing in model elimination theorem provers, in: D. Kapur, ed., *Automated Deduction— CADE-11*, Saratoga Springs, Lecture Notes in Artificial Intelligence. **607** (Springer, Berlin, 1992) 224–238.

[4] W. Bibel, Short proof of the pigeonhole formulas based on the connection method, *J. Autom. Reasoning* **6** (1990) 287–297.

[5] C.-L. Chang and R.C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving* (Academic Press, New York, 1973).

[6] V. Chvátal and E. Szemerédi, Many hard examples for resolution, *J. ACM* **35** (1988) 759–768.

[7] S.A. Cook and R.A. Reckhow, The relative efficiency of propositional proof systems, *J. Symbolic Logic* **44** (1979) 36–50.

[8] N. Eisinger, *Completeness, Confluence and Related Properties of Clause Graph Resolution*, Research Notes in Artificial Intelligence (Pittman, London/Morgan Kaufmann, San Mateo, CA, 1991).

[9] A. Haken, The intractability of resolution, *Theoret. Comput. Sci.* **39** (1985) 297–308.

[10] F. Harary, *Graph Theory* (Addison-Wesley, Reading, MA, 1969).

[11] J.D. Horton and D. Sharpe, private communication.

[12] J.D. Horton and B. Spencer, A top down algorithm to find only minimal clause trees, in: L. Dreschler-Fischer and S. Pribbenow, eds., *KI-95 Activities: Workshops, Posters, Demos*, Bielefeld (Gesellschaft für Informatik, Bonn, 1995) 79–80.

[13] J.D. Horton and B. Spencer, Reducing search with minimal clause trees, Tech. Rept. TR95-099, Faculty of Computer Science, University of New Brunswick, Fredericton, NB, Canada (1995).

[14] J.D. Horton and B. Spencer, Bottom-up procedures for minimal clause trees, Tech. Rept. TR96-101, Faculty of Computer Science, University of New Brunswick, Fredericton, NB, Canada (1996).

[15] R. Hynes, The disequality strategy in theorem proving, undergraduate thesis, Faculty of Computer Science, University of New Brunswick, Fredericton, NB, Canada (1995).

[16] R. Kowalski and D. Kuehner, Linear resolution with selection function, *Artif. Intell.* **2** (1971) 227–260.

[17] R. Letz, K. Mayr and C. Goller, Controlled integration of the cut rule into connection tableau calculi, *J. Autom. Reasoning* **13** (1994) 297–337.

[18] R. Letz, J. Shumann, S. Bayerl and W. Bibel, SETHEO: a high-performance theorem prover, *J. Autom. Reasoning* **8** (1992) 183–212.

[19] J. Lobo, J. Miner and A. Rajasekar, *Fundamentals of Disjunctive Logic Programming* (MIT Press, Boston, 1992).

[20] D.W. Loveland, Mechanical theorem proving by model elimination, *J. ACM* **15** (1968) 236–251.

[21] D.W. Loveland, *Automated Theorem Proving: A Logical Basis* (North-Holland, Amsterdam, 1978).

[22] W.W. McCune, OTTER 2.0 users guide, Tech. Rept. ANL-90/9, Mathematics and Computer Science Division, Argonne National Laboratories, Argonne, IL (1990).

[23] J. Minker and G. Zanon, An extension to linear resolution with selection function, *Inform. Process. Lett.* **14** (1982) 191–194.

[24] N.V. Murray and E. Rosenthal, Dissolution: making paths vanish, *J. ACM* **40** (1993) 504–535.

[25] R. Reiter, Two results on ordering for resolution with merging and linear format, *J. ACM* **18** (1971) 630–646.

[26] J.A. Robinson, A machine-oriented logic based on the resolution principle, *J. ACM* **12** (1965) 23–41.

[27] D. Sharpe, AllPaths theorem prover, Undergraduate thesis, Faculty of Computer Science, University of New Brunswick, Fredericton, NB, Canada (1995).

[28] D. Sharpe, Backwards basic factoring: a pessimistic analog of basic factoring, in: J. Stewman, ed., *Proceedings of the Ninth Florida Artificial Intelligence Research Symposium*, Key West, FL (1996) 459–462.

[29] R.E. Shostak, Refutation graphs, *Artif. Intell.* **7** (1976) 51–64.

[30] B. Spencer, Linear resolution with ordered clauses, in: J. Lobo, D. Loveland and A. Rajasekar, eds., *Proceedings ILPS Workshop — Disjunctive Logic Programming*, San Diego, CA (1991).
[31] B. Spencer, The ordered clause restriction of model elimination and SLI resolution, in: D. Miller, ed., *Proceedings International Symposium of Logic Programming*, Vancouver, BC (1993) 678.
[32] B. Spencer, Avoiding duplicate proofs with the foothold refinement, *Ann. Math. Artif. Intell.* **12** (1994) 117–140.
[33] B. Spencer, J.D. Horton and K. Francis, Experiments with the ALPOC theorem prover, Tech. Rept. TR95-094, Faculty of Computer Science, University of New Brunswick, Fredericton, NB, Canada (1995).
[34] M. Stickel, A Prolog technology theorem prover: implementation by an extended Prolog compiler. *J. Autom. Reasoning* **1** (1988) 353–380.
[35] M. Stickel, A Prolog technology theorem prover: a new exposition and implementation in Prolog, *Theor. Comput. Sci.* **104** (1992) 109–128.
[36] M. Stickel, personal communication.
[37] M. Stickel, Upside-down meta-interpretation of the model elimination theorem-proving procedure for deduction and abduction, *J. Autom. Reasoning* **13** (1994) 189–210.
[38] G. Sutcliffe, C. Suttner and T. Yemenis, The TPTP problem library, in: D. Kapur, ed., *Automated Deduction — CADE-12*, Nancy, Lecture Notes in Artificial Intelligence **814** (Springer, Berlin, 1994) 252–266.
[39] G.S. Tseitin, On the complexity of derivations in propositional calculus, in: A.O. Stisenko, ed., *Studies in Mathematics and Mathematical Logic Part II* (Consultants Bureau, New York, 1970) 115–125.
[40] L. Wos, *Automated Reasoning: 33 Basic Research Problems* (Prentice-Hall, Englewood Cliffs, NJ, 1988).