# CLAUSE TREES and FACTOR PATHS

by

J.D. Horton and Bruce Spencer

TR94-088 October 1994

Faculty of Computer Science
University of New Brunswick
P.O. Box 4400
Fredericton, N.B.
Canada E3B 5A3

Phone: (506) 453-4566
Fax: (506) 453-3566

# Clause Trees and Factor Paths

**J. D. Horton and Bruce Spencer**
Faculty of Computer Science
University of New Brunswick
P.O. Box 4400
Fredericton, New Brunswick, Canada
E3B 5A3
email : *jdh@unb.ca* and *bspencer@unb.ca*
17 October, 1994

## Abstract

The concept of a clause is generalized to that of a clause tree which shows how the clause can be proved from a set of input clauses. Procedures are provided to find factorizations and tautologies at build-time using the internal structure of the tree. Model Elimination and SLI are specializations of this technique. Other resolution-based proof procedures, including bottom-up ones, could include these concepts to make better procedures. One example top-down procedure, ALPOC, has been implemented as part of Stickel's PTTP system.

Content areas: automated reasoning, theorem proving, disjunctive logic programming

## 1 Introduction

Unrestricted resolution[Rob65] allows many proofs. This can lead to inefficiencies in procedures that build proofs because, although many choices are possible, a great number of these choices lead to proofs that are essentially identical. This paper offers a new way to use resolution that eliminates many choices and consequently admits fewer of these identical proofs. Completeness is preserved. The new definition characterizes when factor paths and tautology paths can be used to eliminate redundant proofs. More efficient proof procedures can be developed based on it, and we provide one, ALPOC, in the Model Elimination (ME)[Lov69] family. We also expect that it can be used in bottom up proof procedures such as OTTER[McC90].

The main idea is to differentiate build-time from proof-time. Build-time determines the order that the proof is built by an procedure; proof time determines the order in which a person would write down the proof so that each step follows from previous steps. These "times" are already different in some procedures. The concepts developed in this paper explain why the MESON procedure [Lov78], the SLI procedure [LMR92], the ordered clause restriction [Spe91] and Shostak's refutation graphs work.

This paper develops the theory using propositional logic, but standard lifting techniques [C&L73] can be used to apply the results to first-order logic, as is shown by the ALPOC algorithm.

## 2 Proof-time Concepts

This paper uses the standard definitions from propositional logic [C&L73] of atoms, literals, formulas and clauses. It also uses some standard definitions from graph theory [Har69].

Clause trees, defined in this paper, generalize clauses in that any clause can be represented by a clause tree. However the clause tree in its internal structure maintains how this clause can be proved from other clauses. A clause tree consists of a graph-theoretical tree, together with a specified set of leaves that are said to be **open** (other leaves are **closed**) and a set of paths in the tree that are called factor paths. The nodes of the tree are of two types: OR nodes and atom nodes. All edges of the tree join an atom node and an OR node, and are labeled positive or negative. The atom nodes are labeled by the atoms and are either leaves or of degree two. If an atom node is of degree two, the two edges are labeled differently, one positive and one negative. An OR node represents that clause consisting of the disjunction of the labels of the neighbouring atom nodes, modified by negation if the edge joining the atom node to the clause node is labeled negative. The tree as a whole represents the disjunction of its open leaves, modified by negation if the neighbouring edge is negative. Each internal atom node represents a resolution step in a proof of the clause represented by the clause tree; each factor path represents the removal of an atom or its negation by the process of factoring in a proof of the clause represented by the clause tree. More formally, we define clause trees recursively as follows.

***Definition 1 Clause tree***
*A clause tree is a triple <N, E, F>, the set of nodes, edges and factor paths, respectively, defined by Definition 1(a)-(c).*

***Definition 1(a). Clause tree from an Input Clause***
*Given a clause $C = \{a_1, ..., a_n\}$, the **clause tree** T representing C satisfies the following.*

- *N consists of an OR node and n atom nodes, each labeled by a distinct atom from C.*
- *E consists of n undirected edges, each of which joins the OR node to one of the atom nodes and is designated as positive or negative according to whether the atom is positive or negative in the clause.*
- *F is empty.*

***Definition 1(b). Resolving two Clause trees***
*An **open leaf** of a clause tree <N, E, F> is an atom node of degree one in E that is not the tail of any factor path in F. Let $T_1$ = <$N_1$, $E_1$, $F_1$> and $T_2$ = <$N_2$, $E_2$, $F_2$> be two clause trees such that $n_1$ is an open leaf of $T_1$ and $n_2$ is an open leaf of $T_2$. Suppose that $n_1$ is labeled by an atom a and attached by a negative edge $\{n_1, m\}$, while $n_2$ is labeled by a, but attached by a positive edge. Let $E = E_1 \cup E_2 - \{\{n_1,m\}\} \cup \{\{n_2,m\}\}$ where $\{n_2,m\}$ is a negative edge. Then T = <$N_1 \cup N_2 - \{n_1\}$, E, $F_1 \cup F_2$> is a clause tree.*

***Definition 1(c). Adding a Factor Path***
*Let T = <N, E, F> and let $n_1$ and $n_2$ be two open leaves in T that are both labeled by the same node. If the edges to $n_1$ and $n_2$ have the same sign then the unique undirected path from E joining these open leaves is a **potential factor path.** Construct a path P that contains for each edge on this potential path, a directed edge so that P connects $n_1$ to $n_2$. Then $T_1$ = <N, E, $F \cup \{P\}$> is a clause tree. P is called a **factor path**. The **head** of P is $n_2$ and the **tail** of P is $n_1$. P is said to **end on** $n_2$.*

Figure 1 illustrates the parts of Definition 1. Figure 1(a) shows the clause tree for the clause {a, b, ¬c, ¬d}. Figure 1(b) shows the result of resolution with the clause tree for {e, ¬b, ¬g, ¬d}. Figure 1(c) shows the result of adding a factor path, shown by the thin line.

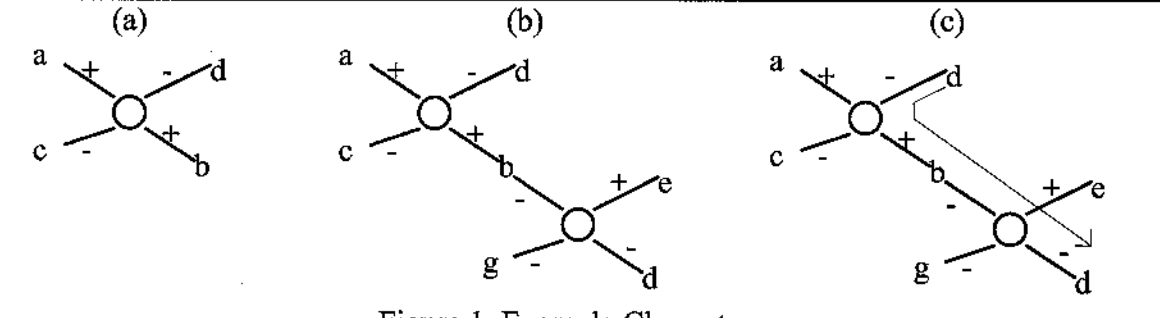

Figure 1 Example Clause trees

***Definition 2**. The **clause of a clause tree** T, written cl(T), is the set of literals formed by the atoms labeling the open leaves of T, modified by negation if the edge incident on the open leaf is designated negative.*

***Definition 3**. Given a set P of clauses, a **derivation** of $T_n$ from P is a sequence $T_1$, ..., $T_n$ of clause trees such that each $T_i$ for i = 1, ..., n is the result of one of the following:*

- *an application of Definition 1(a) on a member of P,*
- *an application of Definition 1(b) on $T_j$ and $T_k$ where j < i and k < i, or*
- *an application of Definition 1(c) on $T_j$ where j < i.*

***Theorem 1**. Let P be a set of clauses and C a clause. $P \models C$ if and only if there exists a clause tree T from P such that $cl(T) \subseteq C$.*

**Proof Sketch** Clause tree operations can perform an ordinary resolution step so they are complete. Each operation is sound so a sequence of them is also sound. □

***Corollary 2.*** *P is an unsatisfiable set of clauses if and only if there is a clause tree T from P with no open nodes.*

## 3 Build-time Concepts

Although the definition of a clause tree allows adding a factor path between nodes only when both are open leaves, after it is used in a resolution step the head of the factor path will no longer be a leaf. Under some conditions the addition of a factor path with an internal node for a head can be allowed. All that is necessary to be able to add such a factor path to a clause tree is to 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 sequences of operations are considered: the proof-time sequence that satisfies Definition 3, and a sequence of build-time operations that may include the insertion of a factor path to an internal node. Other operations will be included as well.

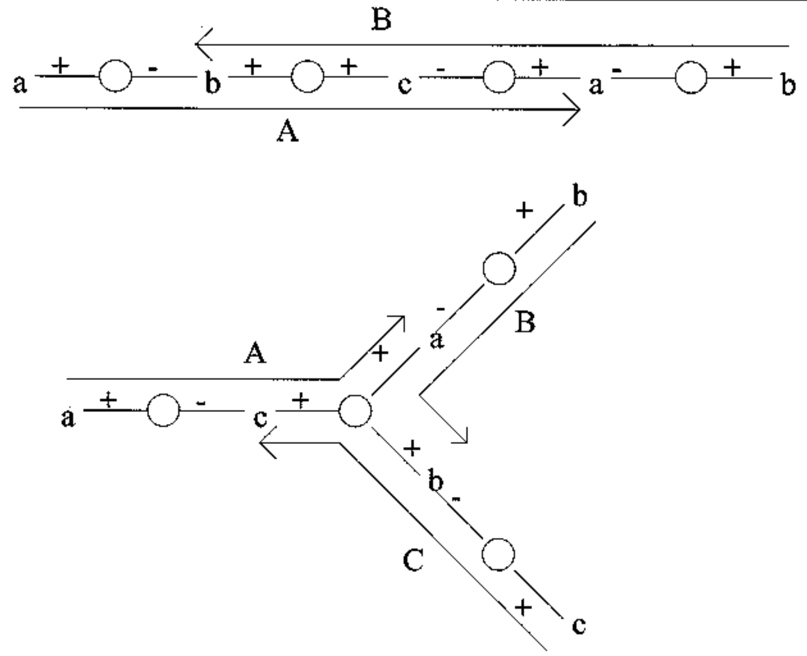

Figure 2 Two Impossible Clause trees

What are the necessary conditions on these more general factor path insertions to ensure that a proof-time sequence (derivation) exists? First, no two factor paths A and B can end on each other, for if they do then in proof-time the endpoints of A must both be leaves of some clause tree, with one endpoint of the 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 Figure 2.

Moreover, if there are two factor paths A and B such that A ends on B then a third factor path C cannot be added to the clause tree if it happens that B ends on C and C ends on A. Again the reason why is that no derivation exists that can produce the factor path A before B, B before C and C before A. This impossible situation is illustrated in the lower part of Figure 2. Other conceivable situations involving three or more competing factor paths are also to be avoided.

The following definitions and results show what situations must be avoided and what paths can be allowed.

***Definition 4.*** *For two paths A and B, $A \prec B$ ("A precedes B") if and only if an end of A occurs on B.*

***Definition 5.*** *A* ***potential path*** *P in a clause tree T is the unique directed path from a node $n_1$ to a node $n_2$ in T where the atoms that label these nodes are identical. If the first and final edges of P both have the same sign then it is a* ***potential factor path.*** *If the signs on the first and final edges of P are different, then it is a* ***potential tautology path.***

Thus factor paths are also potential factor paths.

***Definition 6.*** *A set of potential factor paths and potential tautology paths in a clause tree is* ***legal*** *if the $\prec$ relation can be extended to a partial order $\prec^*$.*

***Theorem 3.*** *Let $T = <N, E, F_1>$ be a clause tree and let $F_2$ be a set of potential factor paths in which the tail of each path is a leaf of T. Then $F = F_1 \cup F_2$ is legal if and only if $T_F = <N, E, F>$ is a clause tree.*

***Proof*** Since F is legal there is a partial order $\prec^*$ on F. Extend this partial order to a total order on F. We use this total order to build a derivation of $T_F$. First, the clause corresponding to each OR node is made into a clause tree using Definition 1(a). Next, each path in F is processed according to the total order: $P_1, P_2, ..., P_k$. For each $P_i$ all of the resolutions corresponding to the internal atom nodes of $P_i$ are done using Definition 1(b), except that any resolution already done while processing $P_1,..., P_{i-1}$ is not done again. At this time, the head of $P_i$ will be a leaf of the clause tree since only the resolutions corresponding to internal nodes of $P_1,..., P_i$ have been performed and by the ordering, the head of $P_i$ is not among them. The tail of $P_i$ is also a leaf by assumption. Thus $P_i$ can be added as a factor path of the clause tree by Definition 1(c). After all paths in F have been processed, all of the other atom nodes can be resolved to form $T_F$.

Conversely if $T_F$ is a clause tree, it has a derivation. This derivation can easily be modified to become a derivation of T by omitting the steps that add the factor paths in $F_2$ to the set of factor paths. Order the factor paths of F according to the order that they are inserted into F. 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 the ordering. Hence the relation $\prec$ is a subrelation of this total order, and so $\prec^*$ is a partial order.□

When a factor path is created, does it matter which direction the path is oriented? In a fundamental sense it does not, as shown by the next lemma.

***Lemma 4. Path Reversal***

*Let $T = (N, E, F)$ be a clause tree, and let A be a factor path in F. Let $A^r$ be the path with the same edge set as A, but oriented in the opposite direction. Then there is a clause tree $T' = (N', E', F')$ such that $N = N'$, $cl(T) = cl(T')$, and $A^r$ is in F'.*

**Proof** Let A be a factor path in F with the atom a at both ends. Let the set of factor paths that contain the head of P be $\{B_1, ..., B_n\}$. Let the set of paths that end on A be $\{C_1, ..., C_n\}$. Note that for all i and j, $C_j \prec^* B_i$, since $A \prec B_i$ and $C_j \prec A$.

Modify the clause tree in the following way. The whole subtree that is joined to the head of A is removed and reattached at the tail of A. Thus the set of nodes remains the same but one edge is changed between E and E'. Also each $B_i$ must be replaced in F' by another path B' . The edge set of $B_i'$ is the symmetric difference of the edge sets of $B_i$ and A. The head and tail of $B_i$ remain the head and tail of $B'_i$. All other paths of F remain in F'. The top of Figure 3 shows A and $B_i$; the bottom shows $A^r$ and $B'_i$.

Clearly $(N, E', \phi)$ is a clause tree, and F' consists of a set of potential factor paths. The relation $\prec$ is the same on F' as on F except between the $B_i$ and the $C_j$. However since $C_j \prec A \prec B_i$ and $C_j \prec A^r \prec B'_i$, the relation $\prec^*$ does not change between F and F'. Thus F' must be legal as F is legal, and T' is a clause tree. The proof is completed by noting that cl(T) = cl(T'), since the open leaves of T are the open leaves of T', except possibly for the head of A (if it is a leaf), in which case it would be replaced by the tail of A which has the same literal.□

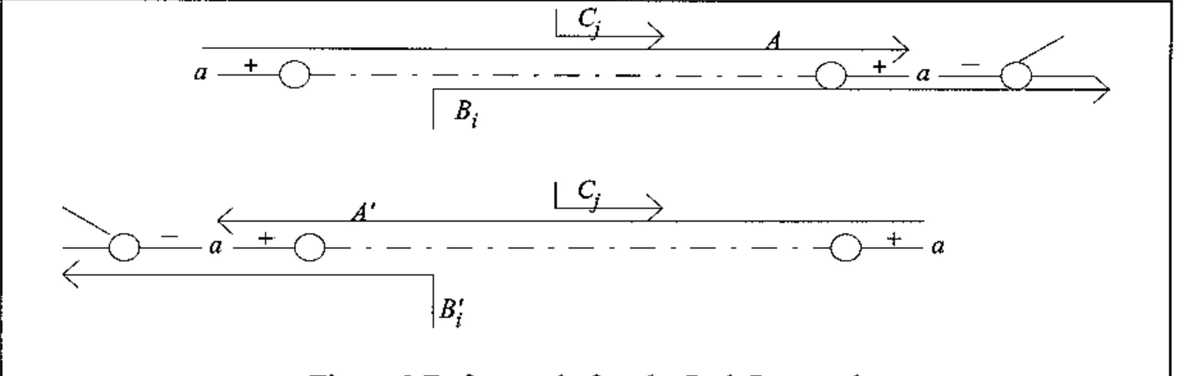

Figure 3 Before and after the Path Reversal

The fundamental purpose of a clause tree T is to show that cl(T) can be derived from the clauses corresponding to the OR-nodes of the tree. Lemma 4 shows that the same derivations can be found regardless of the direction chosen to create a factor path. Thus an algorithm can specify which way a factor path is oriented, without affecting completeness.

***Definition** 7. A clause tree $T = <N, E, F>$ is **admissible** if there is no potential tautology path $P$ such that $F \cup \{P\}$ is legal.*

***Definition 8**. A clause tree $T = <N, E, F>$ is **minimal** if there is no potential factor path $P$ such that $F \cup \{P\}$ is legal.*

***Definition 9**. Let $T_1$ and $T_2$ be clause trees. $T_1$ **subsumes** $T_2$ if $cl(T_1) \subseteq cl(T_2)$.*

***Theorem 5.** Let $T_1 = <N, E, F>$ be a clause tree. There exists a minimal admissible clause tree $T$ that subsumes $T_1$.*

**Proof.** Let $P$ be a potential factor path whose tail $n$ is not a leaf, and that would form a legal set of paths if added to $F$. Suppose that $P$ travels left to right. Reverse all of the factor paths that start to the right of $n$ and end to the left of $n$. Now the subtree to the left of $n$ can be removed, together with all factor paths whose tails have been removed.

Let $Q$ be a potential tautology path from $n_1$ to $n_2$, that would form a legal set of paths if added to $F$. Suppose $Q$ travels left to right. Reverse all paths that start to the right of $n_2$ and go through $n_2$ but not through $n_1$. Also reverse all paths that start to the left of $n_1$ and go through $n_1$ but not through $n_2$. Now remove $Q$ and all subtrees other than the two subtrees attached to $n_1$ and $n_2$, and then attach these two subtrees together. Again remove any factor path that lost a tail. The heads of the remaining paths are not affected so the partial order is still legal.

These two operations are repeated until no such potential paths remain. As these operations can remove open leaves, but can never add them, the resulting clause tree $T$ is minimal, admissible and subsumes $T_1$. □

We now have the following set of operations that can be used to construct clause trees:

- Input clause as a clause tree (Definition 1(a));
- Resolution (Definition 1(b));
- Factor path insertion from leaf to leaf (Definition 1(c));
- Factor path insertion from leaf to internal node (Theorem 3);
- Path reversal (Proof of Lemma 4);
- Factor path insertion from internal node to internal node (proof of Theorem 5);
- Tautology path removal (proof of Theorem 5);

Sequences of the above operations can only generate clause trees whose clause is entailed by the input clauses. These sequences can generate a subsuming clause for every logical consequence of the input clauses.

An algorithm to build clause trees can use its own criteria for the choice of operations that should be performed, and where to perform them, as long as the resulting set of factor paths is legal. The algorithm in the next section uses this freedom to decrease the search space needed by Model Elimination theorem provers.

## 4 A Clause Tree Algorithm

The Model Elimination [Lov78] method of theorem proving applied to propositional logic can be described as a tree-building procedure with the addition of the ancestor resolution rule. It also may be described as a procedure for building clause trees with the following specializations.

1. It builds a clause tree with one node specified as the root.
2. An open leaf node is selected as the current leaf, which means it is the site of the next resolution step.
3. One of the participants in a resolution step will be the clause tree of an input clause.
4. It searches for factor paths and tautology paths only between the current leaf node and the root.

On the basis of the fourth of these restrictions, factor paths can never form an illegal set because all factor paths point to the root. The paths can be ordered by how close they are to the root.

Another procedure in the ME family, SLI [LMR92] can be described in terms of clause trees. The procedure to build SLI clause trees adheres to the first three restrictions above. The main distinction between ME and SLI is that factor paths and tautology paths are allowed from the current leaf either to an ancestor or an ancestor's sibling that is still an open leaf. Thus SLI can detect more opportunities for factoring, and can avoid more tautologies than ME can.

To make the discussion simpler, suppose that the SLI procedure uses a left-to-right ordering of literals in the input clauses to determine which open leaf will be the current leaf. Then a proof-time ordering can be imposed on the heads of factor paths: those deeper in the tree are earlier in proof-time, and among those that end at a given level, the ones on the left are earlier. Thus it is immediate that SLI clause trees are legal.

Both ME and SLI can be restricted by the ordered clause restriction [Spe91] which imposes a total order on the set of input clauses and allows factor paths to ancestors only if the OR node adjacent to the tail is larger in the total order than the OR node adjacent to the head. Completeness is preserved since by path reversal, Lemma 4, the factor path can be oriented in either direction. The ordered clause restriction imposes *a priori* a direction based on the total order.

A new procedure has been implemented that also adheres to the first three specializations above, but generalizes ME and SLI in that it searches most of the current clause tree for factor paths and potential tautology paths. In this algorithm, the current leaf is selected in a depth-first manner and within a clause the ordering is left-to-right. Therefore the clause tree is composed of three distinct parts: the ancestors of the current open leaf, the descendants of the left siblings of these ancestors, none of which are open leaves, and the right siblings of the ancestors, all of which are open leaves. (See Figure 4.) Factor paths are allowed (and tautology paths can be avoided) from the current leaf either to an ancestor or to a descendant of a left sibling of an ancestor. Those that end at an ancestor are called ancestor paths, and the others are called left paths. Because this procedure creates **A**ncestor and **L**eft **P**aths and uses the **O**rdered **C**lause restriction, it is called ALPOC.

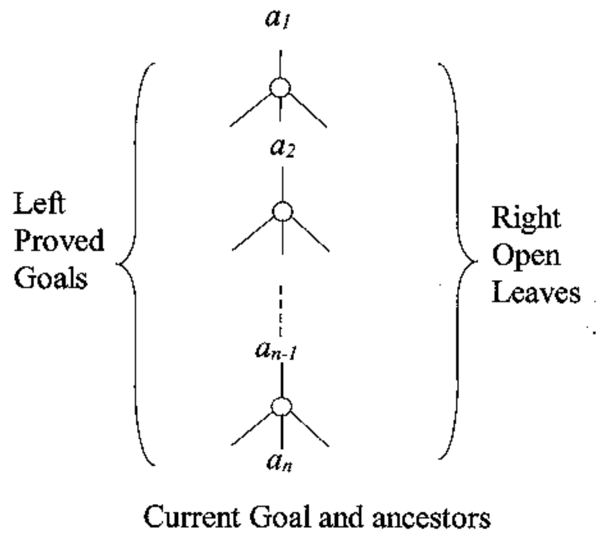

Figure 4 Left, Current and Right portions of the Goal tree

Unlike ME and SLI, the factor paths created by ALPOC will not form a legal set by virtue of the search procedure alone. Any ancestor path can be used. But a potential left path on its way down the tree may run over the head of some ancestor path running up. If so, then it cannot end anywhere on this ancestor path. Nor can it end on any other path that (transitively) precedes this ancestor path. If a path from a node u to a node v can be added to the set of factor paths and the set of paths remains legal, we say v is **visible** from u, otherwise v is **invisible** from u.

For example, consider the partial tree shown in Figure 5 as built by ALPOC. The signs on the edges are omitted. When the current goal is a right sibling of e (or any descendant thereof), left paths to any left descendants of e's parent c are possible. Later when the current goal is one level higher, at a right sibling of c, no nodes on the ancestor path *C* are visible and neither are those on the left path *D* since *D* ends on *C*. When the current leaf is at the next higher level, most of the left nodes are invisible.

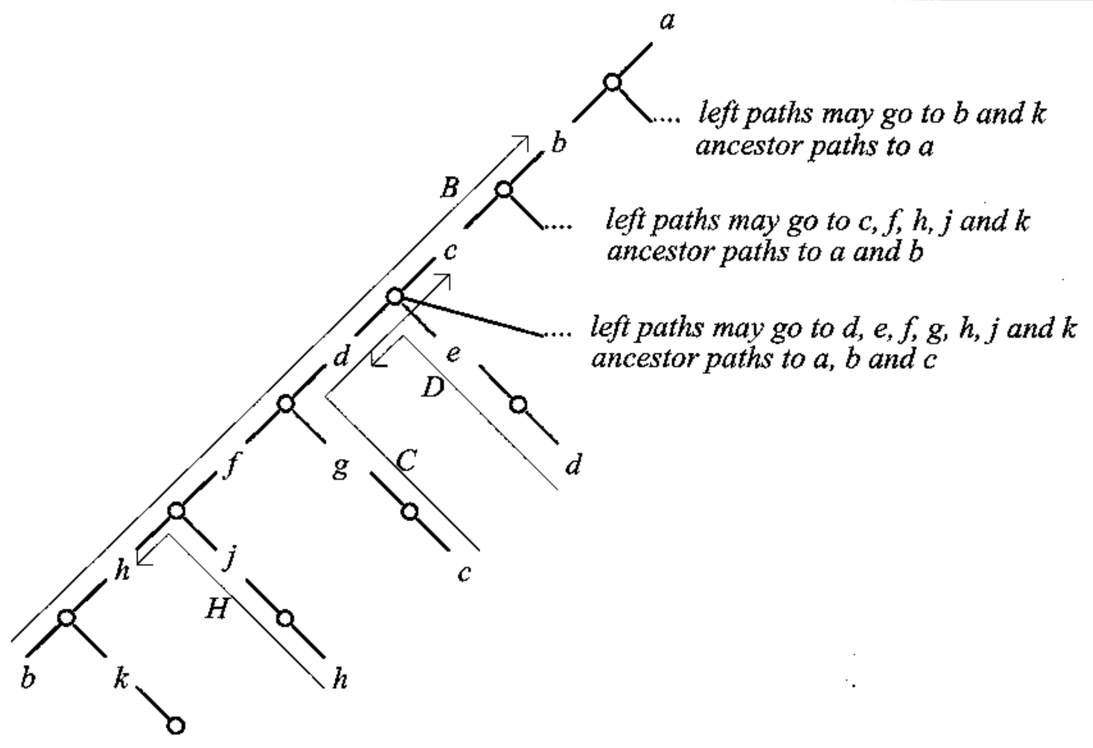
a
left paths may go to b and k ancestor paths to a
b
B
left paths may go to c, f, h, j and k ancestor paths to a and b
c
left paths may go to d, e, f, g, h, j and k ancestor paths to a, b and c
d
e
D
d
f
g
C
c
h
j
H
b
k
h

Figure 5 A partial clause tree and potential factor paths

Given that the set of paths forms a legal set, the clause tree represents a sound deduction. If the input set of clauses is satisfiable but becomes unsatisfiable with the negation of the theorem, then a clause tree exists that proves the theorem and the ALPOC procedure will find it.

***Theorem 6.*** *The ALPOC procedure is sound and refutationally complete.*
**Proof** omitted.

Although the extra paths in ALPOC add some extra work, this work can be implemented so that it increases the overall work by only a modest factor. (See Section 6.) Consider that the current goal p has just been proved. Let $D_p$ be the maximum depth of the head of a factor path preceded by some path through p. For example if there is only one ancestor path through p and it stops at level L in the tree then $D_p = L$ (See Figure 6.) In future operations, if a left path is drawn to p, it must not rise in the tree as high as $D_p$. Otherwise it will stop on a path that it precedes, which is illegal.

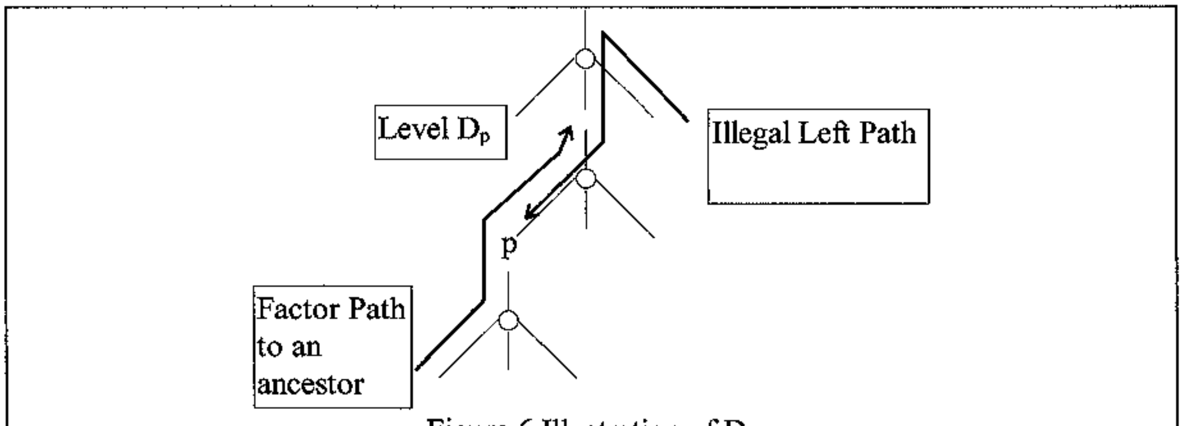

Figure 6 Illustration of $D_p$

The ALPOC procedure maintains a list of nodes that are visible from the current node, and thus are candidates for heads of left factor paths. This visible list is modified when the current goal changes so that it is always up to date. The list is organized by levels so that when the current goal p is completed, it is entered to the visible list at level $D_p$. When the level of the current goal changes from L to L-1 (i.e. a rightmost sibling was the current goal) then all the goals in the visible list at level L are removed. These two operations of inserting and removing goal from the visible list are relatively inexpensive.

The ALPOC algorithm has been implemented by adding it to Stickel's PTTP procedure [Sti88]. PTTP operates by converting each contrapositive form of each input clause to a Prolog clause and adding an argument to each goal in this clause to contain the ancestors in the goal tree. Also, for each distinct predicate and each negation, two clauses are added. One clause halts search and initiates backtracking if the current goal is identical to one of its ancestors. This corresponds to tautology ancestor paths. The other clause allows the current goal to succeed if a unifiable negated ancestor exists. This is an ancestor factor path. A special case of this occurs when that negated ancestor is identical to the current goal. In that case further searching to solve this goal is eliminated.

Three additions to PTTP are necessary to achieve ALPOC: $D_p$ is calculated for each goal that succeeds, the visible list is maintained, and left paths are drawn. Two additional arguments are added to each goal to construct the list of levels (integers) of the heads of ancestor paths that pass through the goal. Two additional arguments are added to construct the visible list. Finally one more argument contains the goal's level in the proof tree. (A pair of arguments is needed for $D_p$ and for the visible list since these are Prolog accumulators.) Left paths are drawn by adding clauses to the procedure for each goal, similar to ancestor paths. If a member of the visible list is identical to the goal, then one clause halts searching and initiates backtracking, because a legal left tautology path exists. If a negated member of the visible list is unifiable with the goal then another clause uses the legal left factor path to it. A special case occurs here also; if the negated member of the visible list is identical to the goal then further searching is disabled.

## 5 An Example

The first non-propositional example that we ran was example 8 from [C&L73], which also appeared as an example in [Sti88] and is NUM015-1 in the TPTP problem set [SSY93]. The theorem to be proved is that any natural number is divisible by a prime. The input formulae are:

1 divides(X,X).
2 not_divides(X,Y);not_divides(Y,Z);divides(X,Z).
3 prime(X);divides(divisor(X),X).
4 prime(X);less(1,divisor(X)).
5 prime(X);less(divisor(X),X).
6 not_less(1,X);not_less(X,a);prime(factor_of(X)).
7 not_less(1,X);not_less(X,a);divides(factor_of(X),X).
8 less(1,a).
9 not_prime(X);not_divides(X,a).
10 query:-prime(X),divides(X,a)

The first formula means that a number X divides itself; the second is the transitivity of divides; 3-5 define "X is prime"; 6 and 7 express an induction hypothesis, that the theorem is true for numbers less than a; 8 says that 1 is less than a; 9 is the negation of the theorem; and 10 is the query.

This problem was run using both PTTP by itself, and with ALPOC included. PTTP found a proof using 10 clauses, while ALPOC found a proof using only 9 clauses. The proofs are given below from computer output.

```
ALPOC Proof:
Goal#  Wff#  Wff Instance
-----  ----  ------------
  [0]   10   query :- [1],[12].
  [1]    3      prime(a) :- [2].
  [2]    2         not_divides(divisor(a),a) :- [3],[8].
  [3]    7            divides(factor_of(divisor(a)),divisor(a)) :- [4],[6].
  [4]    4               less(1,divisor(a)) :- [5].
  [5]   red                 not_prime(a).
  [6]    5               less(divisor(a),a) :- [7].
  [7]   red                 not_prime(a).
  [8]    9            not_divides(factor_of(divisor(a)),a) :- [9].
  [9]    6               prime(factor_of(divisor(a))) :- [10],[11].
 [10]   lfp                 less(1,divisor(a)).
 [11]   lfp                 less(divisor(a),a).
 [12]    1      divides(a,a).
```

```
PTTP Proof:
Goal#  Wff#  Wff Instance
-----  ----  ------------
  [0]   10   query :- [1],[13].
  [1]    4      prime(a) :- [2].
  [2]    6         not_less(1,divisor(a)) :- [3],[5].
  [3]    5            less(divisor(a),a) :- [4].
  [4]  red               not_prime(a).
  [5]    9            not_prime(factor_of(divisor(a))) :- [6].
  [6]    2               divides(factor_of(divisor(a)),a) :- [7],[11].
  [7]    7                  divides(factor_of(divisor(a)),divisor(a)) :-[8],[9].
  [8]  red                     less(1,divisor(a)).
  [9]    5                     less(divisor(a),a) :- [10].
 [10]  red                        not_prime(a).
 [11]    3                  divides(divisor(a),a) :- [12].
 [12]  red                     not_prime(a).
 [13]    1      divides(a,a).
```

The PTTP proof uses four ancestor factor paths (red); the ALPOC proof uses two ancestor factor paths and two left factor paths (lfp). The ALPOC proof can be extended with two more clauses to form an ME type proof with only ancestor factor paths, which is then longer than the proof found by PTTP.

PTTP uses iterative deepening in its search methodology. Thus because the ALPOC proof is smaller than the PTTP proof, it is found at an earlier level and therefore found faster than the PTTP proof. But there is another advantage for ALPOC. ALPOC does fewer inferences at most levels, as shown in the table below. The first few levels are the same, then ALPOC starts to do fewer inferences, and the number of inferences at each search level grows more slowly for ALPOC than for PTTP. The end result for this problem was that ALPOC did fewer than a fourth of the inferences (886 to 3830), and took just more than a third of the time of PTTP (0.216 to 0.617 seconds). On the other hand, ALPOC took about 50% more time per inference than PTTP.

| Search Level | Number of Inferences | | | |
|---|---|---|---|---|
| | PTTP | | ALPOC | |
| | cum | diff | cum | diff |
| 1 | 3 | 3 | 3 | 3 |
| 2 | 9 | 6 | 9 | 6 |
| 3 | 27 | 18 | 27 | 18 |
| 4 | 57 | 30 | 55 | 28 |
| 5 | 118 | 61 | 103 | 48 |
| 6 | 212 | 94 | 171 | 68 |
| 7 | 405 | 197 | 284 | 113 |
| 8 | 700 | 295 | 430 | 146 |
| 9 | 1317 | 617 | 685 | 255 |
| 10 | 2291 | 874 | 886 | 201 |
| 11 | 3830 | 1539 | - | - |

## 6 Empirical Results and Analysis

We have run many problems from the TPTP collection of problems [SSY93], using both PTTP and ALPOC, using a SUN SPARCStation 2. The theorem provers are written in Quintus PROLOG 3.1.1. All timings are reported in seconds, but we have ignored problems that ran in less than 0.2 seconds on both systems. Reading across each row, the table below reports the time taken by PTTP, by ALPOC, the factor by which PTTP is faster if it is faster, and the factor by which ALPOC is faster if it is faster. These problems were chosen from among those reported by Stickel [Sti88] and were in TPTP.

| Problem Name | PTTP Time | ALPOC Time | PTTP Factor | ALPOC Factor |
|---|---|---|---|---|
| GRP001-1 | 11.83 | 37.62 | 3.2 | - |
| GRP008-1 | 387.50 | 364.77 | - | 1.1 |
| GRP009-1 | 1.48 | 4.33 | 2.9 | - |
| GRP012-1 | 0.10 | 0.27 | 2.7 | - |
| GRP012-2 | 180.52 | 540.57 | 3.0 | - |
| GRP013-1 | 1.22 | 3.32 | 2.7 | - |
| GRP029-1 | 14.62 | 52.15 | 3.6 | - |
| GRP030-1 | 2.05 | 0.10 | - | 20.5 |
| GRP036-3 | 0.13 | 0.45 | 3.4 | - |
| GRP037-3 | 3.05 | 8.43 | 2.8 | - |
| MSC001-1 | 67.65 | 103.27 | 1.5 | - |
| MSC002-1 | 0.23 | 0.67 | 2.9 | - |
| MSC004-1 | 80.65 | 0.23 | - | 346.1 |
| NUM001-1 | 0.50 | 0.93 | 1.9 | - |
| NUM002-1 | 0.18 | 0.35 | 1.9 | - |
| NUM015-1 | 0.62 | 0.22 | - | 2.9 |
| NUM024-1 | 6.15 | 12.58 | 2.0 | - |
| NUM027-1 | 10.00 | 14.55 | 1.5 | - |
| PUZ030-0 | > 9 hours | 36.97 | - | > 876 |
| RNG001-5 | 626.02 | 2016.32 | 3.2 | - |
| RNG040-2 | 29.72 | 42.45 | 1.4 | - |
| RNG041-1 | 1.50 | 2.37 | 1.6 | - |
| SET002-1 | 0.35 | 0.35 | 1.0 | - |
| SYN034-1 | 0.72 | 0.03 | - | 21.7 |
| MSC007-1.005 | > 60 hours | 0.22 | - | > 980000 |
| MSC007-1.006 | DNR | 2.27 | - | NA |
| MSC007-1.007 | DNR | 24.38 | - | NA |

DNR means the problem was not attempted. NA means the factor is not available. A time reported as > means that the proof was not completed after that much time.

The extra maintenance and searching that ALPOC imposes do add to the runtime, by a factor of less than four. But ALPOC can also decrease the number of goals in the search tree. When this happens ALPOC runs significantly faster than PTTP. The effect is magnified by the iterative deepening search performed by PTTP, since the search at each level reproduces the search at all previous levels. By reducing number of levels for some proofs, ALPOC has improved on PTTP by large factors.

A known problem with the ordered clause restriction added to PTTP is that it may increase the number of levels that need to be searched. However ALPOC will never increase the number of levels over PTTP, thus solving the problem..

## 7 Related and Future Work

ALPOC is related to Shostak's work [Sho76]. The visible lists created by ALPOC correspond to C literals. One significant difference is that ALPOC uses the ordered clause restriction. Another is that ALPOC has been implemented using Prolog technology to achieve a high speed theorem prover.

Left paths allow a type of "lemmaizing" for non-Horn problems, similar to the proposal of Astrachan and Stickel [Ast92], except that with ALPOC the lemmas need not be single literals. As they report, lemmas can lead to a significant improvement in run-time.

The work here relates to bottom-up theorem provers, such as OTTER[McC90], as well. The clause tree that results from a resolution step may have a potential factor path from one or several of its open leaves. Thus the clause of the result may have fewer literals than the clause that would result from ordinary resolution on the clauses of its parent clause tree. Similarly there may be potential paths in a clause tree which make the clause tree either not minimal or not admissible. Therefore it can be reduced (Theorem 5). Thus the existence of a subsuming clause may be detected without performing a subsumption check as in OTTER. This seems relevant to the redundancy problem [Wos88]. One drawback to using clause trees may be that clause trees are bigger than their corresponding clauses, and so storing many of them may not be practical. Other operations are possible on clause trees that remove nodes and can be used to reduce the necessary storage.

A recent direction in theorem-proving work is to combine the high speed inference rates of top-down systems, such as METEOR[Ast91] and SETHEO[LSBB90], with the flexible control strategy, redundancy control and clause retention of bottom-up systems. For example UR-PTTP[Sti91] uses theory resolution to apply Prolog technology to the Unit Resulting rule of inference. The clause tree formalism provides a basis for a new family of proof procedures that combines ideas from both approaches.

## 8 References

[Ast91] O. L. Astrachan and D. W. Loveland, METEORs: High Performance Theorem Provers using Model Elimination, In *Automated Reasoning Essays in Honor of Woody Bledsoe,* Robert S. Boyer (ed.) 31–59, 1991.

[Ast92] O. L. Astrachan and M. Stickel, Caching and Lemmaizing in Model Elimination Theorem Provers. In D. Kapur, ed., *CADE-11*, pp 224–238, 1992.

[C&L73] C.-L. Chang and R.C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York and London, 1973.

[Har69] F. Harary, *Graph Theory.* Addison-Wesley, Reading, Mass., 1969.

[LSBB90] R. Letz, J. Schumann, S. Bayerl and W. Bibel, SETHEO- A High Performance Theorem Prover for First-Order Logic. *Journal of Automated Reasoning* 8(2), 1990.

[LMR92] J. Lobo, J. Minker and A. Rajasekar. *Foundations of Disjunctive Logic Programming.* MIT Press, 1992.

[Lov69] D. Loveland, Theorem-Provers Combining Model-Elimination and Resolution. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4,* pp 73–86, University Press, Edinburgh, 1969.

[Lov78] D. W. Loveland, Automated Theorem Proving: A Logical Basis, North-Holland, Amsterdam, the Netherlands, 1978.

[M&R90] J. Minker and A. Rajasekar. A Fixpoint Semantics for Disjunctive Logic Programs. *Journal of Logic Programming,* 9:45–74, 1990.

[McC90] W. W. McCune, OTTER 2.0 Users Guide, Tecnical Report ANL-90/9, Mathematics and Computer Science Division, Argonne National Laboratories, Argonne, IL, 1990.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle *Journal of the ACM,* 12(1):23–41, 1965.

[Sho76] R. E. Shostak, Refutation Graphs, Artificial Intelligence, 7(1), 51-64, 1976.

[Spe91] B. Spencer, Linear Resolution with Ordered Clauses. In J. Lobo, D. Loveland and A. Rajasekar, editors, *Proceedings ILPS Workshop – Disjunctive Logic Programming,* 1991.

[Sti88] M. Stickel, A PROLOG Technology Theorem Prover: Implementation by an Extended PROLOG compiler. *Journal of Automated Reasoning,* 1(4):353-380, 1988.

[Sti91] M. Stickel, PTTP and Linked Inference. In *Automated Reasoning: Essays in Honor of Woody Bledsoe,* Robert S. Boyer (ed.), Kluwer Academic Publishers, Dordrecht, 1991.

[SSY93] G. Sutcliffe,C. Sottner, and T. Yemenis, TPTP Thousands of Problems for Theorem Provers Problem Library Release v1.0.0, Internet newsgroup comp.lang.prolog, Nov 13, 1993.

[Wos88] L. Wos, *Automated Theorem Proving: 33 Basic Research Problems,* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.