Chapter 7

Resolution and Unification

1. Introduction

We have seen that the problem of deciding whether or not a formula B can be deduced from a finite set of formulae $\{A_1, A_2, \ldots, A_n\}$ can be reduced to that of deciding the consistency (satisfiability) of a set of formulae in which no quantifiers appear. We now study resolution methods. Their aim is to produce the most efficient algorithms possible for attacking this problem. There are two types of resolution method: ground resolution or resolution without variables and general resolution or resolution with variables by unification. These are of fundamental importance; they enter into almost all theorem-proving programs and especially into PROLOG as we show in Chapter 8.

2. Ground resolution or resolution without variables

Suppose we wish to establish whether or not a given set of formulae $\{A_1, \ldots, A_n\}$ of propositional calculus is satisfiable, that is, whether or not there is an interpretation i for which $i[A_1] = i[A_2] = \ldots = i[A_n] = \mathbf{T}$. The simplest and most obvious method would be to construct the truth table for $A_1 \wedge A_2 \wedge \ldots \wedge A_n$ and see if at least one T result was obtained. While this would certainly give the answer within a finite time, the time needed is of the order of 2^N where N is the total number of atoms in the set $\{A_i\}$ and the method soon becomes impractical. The method we now describe is faster and is also very simple and can be generalized to apply to formulae of predicate calculus, as we see in Section 4. It is presented as a formal system so as to highlight its simplicity with respect to the formal system of propositional calculus given in Chapter 4.

We define the formal system, GR, ground resolution, as follows:

$\Sigma_{GR} = \{p_0, p_1, \ldots, p_n, \ldots\} \cup \{\neg, v\}$

F_{GR} = the set of all clauses formed with the propositional variables p_0, $p_1, \ldots, p_n, \ldots$, i.e. of all formulae of the form:

$$p_{i1} \vee p_{i2} \vee \ldots \vee p_{in} \vee \neg p_{j1} \vee \neg p_{j2} \vee \ldots \vee \neg p_{jm}$$

The components p_{ih} and $\neg p_{jk}$ are called the *literals* of the clause; we consider only clauses having no repetition of components,

that is, in which each p_{ih} or $\neg p_{jk}$ appears only once. The clause whose literals are those of clauses f and g written without repetitions are denoted f v g. No account is taken of the order in which the literals are written in a clause. The clause having no literals (the empty clause) is denoted by $\square$ as before. This is included in F_{GR}, and cannot be satisfied by any interpretation.

$A_{GR} = \emptyset$: there are no axioms.
R_{GR} = {cut}: there is only one inferential rule:

$$f \vee p_i , g \vee \neg p_i \underset{\text{cut}}{\vdash\!\!-\!\!-} f \vee g$$

Example

(a) A deduction:

$$p_0 \vee p_1 \vee p_2 , \neg p_0 \vee p_1 \vee p_2 , \neg p_1 \vee p_2 \underset{\text{GR}}{\vdash\!\!-\!\!-\!\!-} p_2$$

is as follows:

$f_1 : p_0 \vee p_1 \vee p_2$	(given)
$f_2 : \neg p_0 \vee p_1 \vee p_2$	(given)
$f_3 : p_1 \vee p_2$	(cut with f_1 and f_2)
$f_4 : \neg p_1 \vee p_2$	(given)
$f_5 : p_2$	(cut with f_3 and f_4)

(b) The deduction:

$p_0 \vee p_1, p_0 \vee \neg p_1, \neg p_0 \vee p_1, \neg p_0 \vee \neg p_1 \underset{\text{GR}}{\vdash\!\!-\!\!-\!\!-} \square$ is as follows:

$f_1 : p_0 \vee p_1$	(given)
$f_2 : p_0 \vee \neg p_1$	(given)
$f_3 : p_0$	(cut with f_1 and f_2)
$f_4 : \neg p_0 \vee p_1$	(given)
$f_5 : \neg p_0 \vee \neg p_1$	(given)
$f_6 : \neg p_0$	(cut with f_4 and f_5)
$f_7 : \square$	(cut with f_3 and f_6)

The deduction trees (cf. Chapter 3) for these examples are given in Figs 8 and 9.

Note that this inference rule generalizes both the *modus ponens* (m.p.) and the *modus tollens* (m.t.) rules. In terms of clauses m.p. is:

$$\neg p_i \vee p_j, p_i \underset{\text{m.p.}}{\vdash\!\!-\!\!-} p_j$$

which is the 'cut' rule with f = p_i, g = $\square$; and m.t. is:

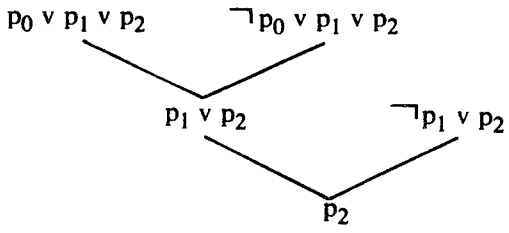


Fig. 8. Deduction tree for (a)

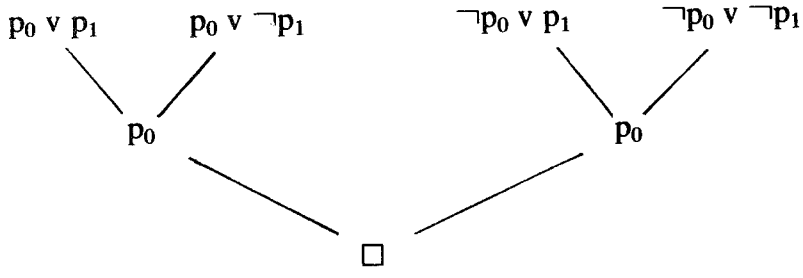


Fig. 9. Deduction tree for (b)

$$p_i \rightarrow p_j\,,\ \neg\, p_j \vdash \neg\, p_i \qquad \text{m.t.}$$

or in clause form:

$$\neg\, p_i \vee p_j,\ \neg\, p_j \vdash \neg\, p_i$$

the 'cut' rule with f = □, g = ¬ p_i.

It is easily verified that the rule of transitive implication (t.i.):

$$p_i \rightarrow p_j,\ p_j \rightarrow p_k \vdash p_i \rightarrow p_k \qquad \text{t.i.}$$

is also a particular case of this same rule. Further, if the expressions are put into clause form, repeated applications of the rule give the following:

$p_{i1}, p_{i2}, \ldots, p_{in}, (p_{i1} \wedge p_{i2} \wedge \ldots \wedge p_{in}) \rightarrow p_j \vdash p_j$
$p_j \rightarrow (p_{i1} \vee p_{i2} \vee \ldots \vee p_{in}), \neg\, p_{i1}, \neg\, p_{i2}, \ldots, \neg\, p_{in} \vdash \neg\, p_j$

The strength of the resolution resides in its great generality; with it one can perform forward or backward chaining, or both.

Proposition 1: a set $\mathscr{A}$ of propositional calculus clauses is unsatisfiable iff $\mathscr{A} \vdash_{GR} \square$

Proof

(a) We may suppose $\mathscr{A}$ to be finite, for if $\mathscr{A}$ is infinite then:

- it is unsatisfiable if and only if it contains an unsatisfiable finite subset (by the compactness theorem, Proposition 10 in Chapter 4);
- and $\mathscr{A} \vdash_{GR} \square$ if and only if it contains a finite subset $\mathscr{A}'$ such that:

$$\mathscr{A} \vdash_{GR} \square$$

as is obvious, since a deduction uses only a finite number of hypotheses.

(b) If $\mathscr{A} \vdash_{cut} \square$, then $\mathscr{A}$ is unsatisfiable

(c) If $\mathscr{A}$ is satisfiable, then:

$$\mathscr{A} \vdash_{GR} \square$$

cannot hold

Lemma 1

If $\mathscr{A}_1 \vdash_{cut} C$, then $\mathscr{A}_1 \models C$

must contain two clauses:

$f \vee p_i$, $g \vee \neg p_i$

and:

$C = f \vee g$.

Let i be any interpretation that satisfies $\mathscr{A}_1$; then:

if $i[p_i] = \mathbf{T}$, $i[\neg p_i] = \mathbf{F}$ and therefore $i[g] = \mathbf{T}$ and $i[f \vee g] = \mathbf{T}$
if $i[p_i] = \mathbf{F}$, $i[f] = \mathbf{T}$ and therefore $i[f \vee g] = \mathbf{T}$ again.

(d) If $\mathscr{A}$ is unsatisfiable, then:

$$\mathscr{A} \vdash_{GR} \square$$

This statement results from Lemma 2

The second Lemma introduces the concept of *minimal unsatisfiability*: a set is minimally unsatisfiable if every one of its proper subsets is satisfiable. It

is clear that any unsatisfiable set of clauses will have at least one, and usually several, minimally unsatisfiable subsets.

Lemma 2

If $\mathscr{S}$ is a finite, minimally unsatisfiable set of clauses then:

$$\mathscr{S} \underset{GR}{\vdash\!\!-\!\!-} \square$$

This is proved by induction on the *excess literals* of $\mathscr{S}$, el($\mathscr{S}$), defined by:

el($\mathscr{S}$) = (total number of literals in $\mathscr{S}$) − (number of clauses in $\mathscr{S}$)

where repetitions of the same literal in different clauses are counted:

$el(p_1 \vee p_2, p_2 \vee \neg p_3) = 4 - 2 = 2$

If el($\mathscr{S}$) = 0 and $\mathscr{S}$ is minimally unsatisfiable then $\mathscr{S}$ is of the form $\{p_i, \neg p_i\}$ and therefore $\mathscr{S} \underset{GR}{\vdash\!\!-\!\!-} \square$; so the result holds for el($\mathscr{S}$) = 0.

Suppose it holds for all such $\mathscr{S}$ with el($\mathscr{S}$) < n, where n > 0. There must be at least one clause $f \in \mathscr{S}$ with more than one literal; let L be a literal of f, and g the clause resulting from the deletion of L from f.

Let:

$$\mathscr{S}_1 = (F - \{f\}) \cup \{g\}$$

$\mathscr{S}_1$ is unsatisfiable, for if not any interpretation that satisfied $\mathscr{S}_1$ would satisfy $\mathscr{S}$. Let B_1 be a minimally unsatisfiable subset of $\mathscr{S}_1$. Since el($\mathscr{S}$) = n, el($\mathscr{S}_1$) < n and also el(B_1) < n and therefore there is a deduction $f_1, f_2, \ldots, f_k$ of $\square$ from B_1; suppose first that none of these f_i is g. This means that the deduction is from $\mathscr{S}$ and the result is thus proved for el($\mathscr{S}$) = n. Now suppose one of the f_i is g; if we add the literal L to each clause f_i (occurring after g in $f_1, f_2, \ldots, f_k$) we have a deduction $f_1', f_2', \ldots, f_k'$ of L from $\mathscr{S}$.

Now let $\mathscr{S}_2 = (\mathscr{S} - \{f\}) \cup \{L\}$. Then $\mathscr{S}_2$, like $\mathscr{S}_1$, is unsatisfiable; let B_2 be a minimally unsatisfiable subset of $\mathscr{S}_2$. We must have $L \in \mathscr{S}_2$, for otherwise $B_2 \subset \mathscr{S} - \{f\}$, which is satisfiable because $\mathscr{S}$ is minimally unsatisfiable.

Since el(B_2) < n there is a deduction of $\square$ from B_2; let this be $g_1, g_2, \ldots, g_h$. We have now arrived at the following:

$$\mathscr{S} \underset{GR}{\vdash\!\!-\!\!-} L \text{ by } f_1', f_2', \ldots, f_k'$$

$$\mathscr{S} \cup \{L\} \underset{GR}{\vdash\!\!-\!\!-} \square \text{ by } g_1, g_2, \ldots, g_h$$

from which it follows:

$$\mathscr{S} \underset{GR}{\vdash\!\!-\!\!-} \square \text{ by } f_1', f_2', \ldots, f_k', g_1, g_2, \ldots, g_h$$

Thus the result is proved for el($\mathscr{A}$) = n and so by induction for all $\mathscr{A}$. △

Proposition 1 gives these lemmas for the following reasons:

1. If $\mathscr{A}$ is unsatisfiable, then $\mathscr{A}$ contains at least one unsatisfiable minimally $\mathscr{A}'$, and after that Lemma 2 gives:

$$\mathscr{A}' \underset{GR}{\vdash\!\!\!-\!\!\!-} \square$$

and:

$$\mathscr{A} \underset{GR}{\vdash\!\!\!-\!\!\!-} \square$$

2. If $\mathscr{A} \underset{GR}{\vdash\!\!\!-\!\!\!-} \square$, then (according to Lemma 1):

$$\mathscr{A} \models \square$$

where $\mathscr{A}$ is unsatisfiable ($\square$ is unsatisfiable if i satisfies $\mathscr{A}$, then i($\square$) = v).

Note

$\underset{cut}{\vdash\!\!\!-\!\!\!-}$ denotes one time

$\underset{GR}{\vdash\!\!\!-\!\!\!-}$ denotes a finite number of times

Example

$\neg p_2, p_1 \vee p_2 \underset{cut}{\vdash\!\!\!-\!\!\!-} p$

$p_1 \neg p_1 \underset{cut}{\vdash\!\!\!-\!\!\!-} \square$

where:

$p_1 \vee \neg p_2, p_1 \vee p_2, \neg p_1 \underset{GR}{\vdash\!\!\!-\!\!\!-} \square$ △

Note

Although the formal system GR that we have defined enables us to decide whether or not any given set $\mathscr{A}$ is satisfiable, it is less powerful than the system P_o described in Chapter 3, for these reasons:

1. it can handle only clauses;

2. the relation that holds for P_o:

$$(h_1, h_2, \ldots, h_n) \underset{P_o}{\vdash\!\!\!-\!\!\!-} t \text{ iff } (h_1 \wedge h_2, \wedge \ldots \wedge h_n) \models t)$$

does not hold GR; for example:

$p_0 \models p_0 \vee p_1$ although it is not true that $p_0 \underset{GR}{\vdash\!\!\!-\!\!\!-} p_0 \vee p_1$.

The Davis–Putnam procedure (see Exercises in Chapter 6) generally gives a faster determination of the satisfiability or otherwise of a set of clauses than does the ground resolution method.

3. Unification

The algorithm given in Chapter 6 for deciding whether or not a formula B is a consequence of a set of formulae $\{A_1, A_2, \ldots, A_n\}$ becomes impractical as soon as the set consists of more than ten formulae. This is because the Herbrand atoms, over which the search for a contradiction is conducted, are obtained by replacing the variables by terms in a completely unsystematic manner. Thus if the problem is to find if the set:

$$\neg p(x) \vee q(y),\ p(f(z)) \vee q(u),\ \neg q(v)$$

leads to a contradiction, it is of no help to replace x, y, v by a, and z, u by f(a); whilst replacing y, z, u, v by a and x by f(a), giving:

$$\neg p(f(a)) \vee q(a),\ p(f(a)) \vee q(a),\ \neg q(a)$$

leads immediately to a contradiction and shows that there is no model.

The general idea of bringing the atoms as closely as possible into coincidence so as to reveal any contradiction quickly is what underlies the general resolution method or method of resolution with variables, which we study in Section 4 of this chapter.

The process of bringing into coincidence by an appropriate choice of replacements is called *unification*. The principle has been known in mathematical logic for a long time, having appeared in Herbrand's thesis in 1930; but its exploitation as a fundamental component of theorem-proving algorithms, and as a main tool in logical programming, is due to J A Robinson (1965).

A. Substitutions and Substitution Components

A *substitution component* is any expression of the form $(x \mid t)$ where x is a variable and t any term of predicate calculus.

If A is a formula of predicate calculus, $(x \mid t)$ A denotes the formula that results when every occurrence of the free variable x in A is replaced by t. For example:

$$(x \mid f(y,g(a))) \; (p(u) \rightarrow r(x))$$
$$= \quad (p(u) \rightarrow r(f(y,g(a))))$$

A substitution is a mapping:

σ: $F_{Pr} \rightarrow F_{Pr}$

of the form:

σ : A $c_1, c_2, \ldots, c_k$A

where the c_i are substitution components; the finite series $c_1, c_2, \ldots, c_k$ is called a decomposition of the substitution, and we write:

$\sigma = [c_1, c_2, \ldots, c_k]$

The identity substitution is written ϵ or [ ], and the set of all substitutions denoted by Sub.

We are concerned only with formulae that contain no quantifiers, so the preliminary renaming discussed in Section 3 in Chapter 5 is not necessary here.

Example

Let σ be the substitution:

$[(x \mid f(a)) \; (y \mid f(x))]$

Then:

$$\begin{aligned} \sigma \; p(x,y) &= (x \mid f(a))(y \mid f(x)) \; p(x,y) \\ &= (x \mid f(a)) \; p(x,f(x)) \\ &= p(f(a),f(f(a))) \end{aligned}$$

Two points should be noted. First, it is not in general true that $[c_1 \; c_2] = [c_2 \; c_1]$. Thus for the above example:

$$\begin{aligned} (y \mid f(x)) \; (x \mid f(a)) \; p(x,y) &= (y \mid f(x)) \; p(f(a),y) \\ &= p(f(a), f(x)) \\ &\neq \sigma \; p(x,y) \end{aligned}$$

Second, the decomposition of a substitution into components is in general not unique:

and

$$[(x \mid y) \; (z \mid y)] = [(z \mid y) \; (x \mid y)]$$
$$[(x \mid t) \; (y \mid t) \; (z \mid t)] = [(x \mid t) \; (y \mid x) \; \; (z \mid x)]$$

If $S = \{A_1, A_2, \ldots, A_n\}$ is a set of predicate calculus formulae, we say that any substitution σ such that:

$$\sigma \, A_1 = \sigma \, A_2 = \ldots = \sigma \, A_n.$$

is a *unifier* of S. Thus for the set:

$$S_1 = \{A_1, A_2, A_3\}$$

where:

$$A_1 = p(x, z) \quad A_2 = p(f(y), g(a)) \quad A_3 = p(f(u), g(a))$$

the substitution:

$$\sigma_1 = [(x \mid f(u))\ (y \mid u)\ (z \mid g(a))]$$

is a unifier, giving:

$$\sigma_1 A_1 = \sigma_1 A_2 = \sigma_1 A_3 = p(f(u), g(a))$$

The substitution:

$$\sigma_2 = [(u \mid f(a))]\ \sigma_1$$

is also a unifier of S_1, giving:

$$\sigma_2 A_1 = \sigma_2 A_2 = \sigma_2 A_3 = p(f(f(a)), g(a))$$

Clearly, if σ is a unifier of S, then so is any substitution of the form $\alpha\sigma$.
A set may have no unifiers; for example:

$$S_2 = \{p(x, f(x)), r(f(t), y)\}$$

as is obvious, because the predicates are all different; also:

$$S_3 = \{p(x, f(x)), p(f(y), y)\}$$

Let U_S denote the set of all unifiers of the set S; if σ is a unifier such that:

$$\forall\ \alpha \in U_S\ \exists\ \beta \in \mathrm{Sub} : \alpha = \beta\sigma$$

we say that σ is a *greatest* (or *most general*) *unifier* of S.

In our current example, σ_1 is a greatest unifier of S_1 but it is not the only one, for:

$$\sigma_3 = [(x \mid f(v))\ (y \mid v)\ (z \mid g(a))\ (u \mid v)]$$

is another; and:

$$\sigma_1 = (v \mid u)\sigma_3$$
$$\sigma_3 = (u \mid v)\sigma_1$$

Intuitively we may say that a greatest unifier, if one exists, of a set of formulae is a substitution that results in bringing the formulae into coincidence, and is as general as possible.

B. Unification Algorithm for a Pair of Atoms A and B

The algorithm determines the substitution Θ that forms a greatest unifier. Let the atoms be A and B:

$\Theta := \epsilon$

while $\Theta A \neq \Theta B$ **do**
find the leftmost symbol of ΘA that differs from the symbol in the same position in ΘB
find the sub-terms t_1, t_2 of ΘA, ΘB that start with this symbol
if neither of these is a variable **or** one is a variable contained in the other
then do print 'A and B are not unifiable' **stop**
else do let x be a variable which is either t_1 or t_2
let t be the one of t_1, t_2 that is not x
$\Theta := (x \mid t)\ \Theta$
end if
end while
stop

Example 1

$\Theta(A)$	$\Theta(B)$	
p(x,f(x),a)	p(u,w,w)	$\Theta = \epsilon$
↑	↑	
p(x,f(x),a)	p(u,w,w)	$\Theta = [(x \mid u)]$
↑	↑	
p(u,f(u),a)	p(u,w,w)	$\Theta = [(w \mid f(u))\ (x \mid u)]$
↑	↑	
p(u,f(u),a)	p(u,f(u),f(u))	failure: neither a nor f(u) is a variable
↑	↑	

Example 2

p(x,f(g(x)),a)	p(b,y,z)	$\Theta = \epsilon$
↑	↑	
p(x,f(g(x)),a)	p(b,y,z)	$\Theta = [(x \mid b)]$
↑	↑	
p(b,f(g(b)),a)	p(b,y,z)	$\Theta = [(y \mid f(g(b)))(x \mid b)]$
↑	↑	
p(b,f(g(b)),a)	p(b,f(g(b)),z)	$\Theta = [(z \mid a)\ (y \mid f(g(b)))(x \mid b)]$
↑	↑	

The process succeeds: A and B are unifiable and a greatest unifier is:

$[(z \mid a)\ (y \mid f(g(b)))\ (x \mid b)]$

Example 3

p(x,f(x))	p(f(y),y)	$\Theta = \epsilon$
↑	↑	
p(x,f(x))	p(f(y),y)	$\Theta = [(x \mid f(y))]$
↑	↑	
p(f(y),f(f(y)))	p(f(y),y)	failure: y is a variable of the term t = f(f(y))
↑	↑	

Loveland (1978), for example, gives a detailed proof of the validity of this algorithm, showing in particular that if there is a unifier for a given pair of atoms then there is a greatest unifier, and that all greatest unifiers differ only in the naming of the variables. This holds also for a set of more than two atoms and an algorithm for the general case can be derived from that for two atoms, as follows.

C. Unification Algorithm for Atoms $A_1, A_2, \ldots, A_n$

for i from 1 to n−1 do
  if $\sigma_{i-1}\ \sigma_{i-2} \ldots \sigma_1\ A_i$ and $\sigma_{i-1}\ \sigma_{i-2} \ldots \sigma_1\ A_{i+1}$ are unifiable
    then find a greatest unifier σ_i
    else print '$A_1\ A_2 \ldots A_n$ are not unifiable'
    stop
  end if
end for
print '$\sigma_{n-1}\ \sigma_{n-2} \ldots \sigma_1$ is a greatest unifier of $A_1\ A_2 \ldots A_n$'
stop

When i = 2 we look for a unifier of A_1 and A_2.

Example

S = {p (x, y), p(f(z), x), p(w, f(x))}
σ_1 = (y | f(z) (x | f(z)) is a greatest unifier of p(x, y) and p(f(z), x))
σ_1 p(f(z), x) = p(f(z), f(z))
σ_1 p(w, f(x)) = p(w, f(f(z)))

where the process stops because p(f(z), f(z)) and p(w, f(f(z))) are not unifiable.

In using this method care must be taken not to forget to apply $\sigma_{i-1}, \sigma_{i-2}, \ldots, \sigma_1$ to those atoms not yet unified, in advance of their turn. Thus in the above example if we had not applied σ_1 to p(w, f(x)) the second stage of the unification would have appeared possible because then p(f(z), f(z)) and p(w,f(x)) would have been unifiable to p(f(z), f(z)) and the false conclusion reached that S is unifiable.

Many variants of Robinson's unification algorithm have been proposed

and used, mostly with the aims of speeding up the process and reducing the storage space needed for the atoms. References are given in the Bibliography.

4. General resolution or resolution with variables

The treatment described in Section 2 is restricted to propositional calculus formulae; we now extend it to predicate calculus. The resolution method leads to a variety of algorithms analogous to those given in Chapter 6 for determining if a formula is a consequence of a given finite set of formulae, with the advantage that its efficiency makes it possible to use these algorithms in practice for theorem proving. Many strategies have been developed for the practical use of the method; we give some in Section 5.

The formal system of general resolution or resolution with variables, RES, is defined as follows.

$\Sigma_{RES} = \Sigma_{Pr}$, the alphabet of first-order predicate calculus (cf. Chapter 5).
F_{RES} = the set of all clauses of predicate calculus, that is, of all formulae of the form:

$$a_1 \vee a_2 \vee \ldots \vee a_n \vee \neg b_1 \vee \neg b_2 \vee \ldots \vee \neg b_m$$

where a_i and b_j are atomic formulae. The same conventions are assumed as for F_{GR}.
$A_{RES} = \emptyset$: there are no axioms.
R_{RES} = {res, dim}: there are two rules, resolution (res) and diminution (dim) defined as follows:

Resolution rule

$$f,g \vdash_{RES} h$$

iff f, g and h are of the form:

(f) $a \vee f_1$
(g) $\neg b \vee g_1$
(h) $\sigma(\Theta f_1 \vee g_1)$

where Θ is a substitution such that Θa and b have no common free variable and σ is a greatest unifier of a and b. Θ is called a *renaming substitution*.

The clause h is called the *resolvant* of f and g.

Example 1

$$p(x,c) \vee r(x), \neg p(c,c) \vdash r(c) \vee q(x)$$

where:

$$a = p(x, c), \square\ b = p(c, c)$$
$$\Theta = (x \mid y), \square\ \Theta a = p(y, c), \square\ \sigma = (y \mid c)$$

Example 2

$$p(x,f(x)),\ \neg p(a,y) \vee r(f(y)) \underset{RES}{\vdash\!\!-\!\!-} r(f(f(a)))$$

where:

$$a = p(x,f(x)),\ b = p(a,y)$$
$$\Theta = \epsilon,\quad \sigma = (y \mid f(a))\ (x \mid a)$$

Diminution rule

$$f \underset{dim}{\vdash\!\!-\!\!-} h$$

iff f, h are of the form:

(f) $a \vee b \vee f_1$
(h) $\sigma a \vee \sigma f_1$

where σ is a greatest unifier of a and b.

Example 3

$$p(x,g(y)) \vee p(i(c),z) \vee r(x,y,z) \underset{dim}{\vdash\!\!-\!\!-} r(i(c),y,g(y)) \vee p(i(c),g(y))$$

where:

$$\sigma = (x \mid i(c))(z \mid g(y))$$

To illustrate the process of deduction in the formal system RES, consider the deduction of $\square$ from the three formulae:

$\neg p(x) \vee p(f(x))$, $p(a)$, $\neg p(f(z))$

$f_1 : \neg p(x) \vee p(f(x))$	(given)
$f_2 : p(a)$	(given)
$f_3 : p(f(a))$	(RES with f_1 and f_2 and $\sigma = (x \mid a)$)
$f_4 : \neg p(f(z))$	(given)
$f_5 : \square$	(RES with f_3 and f_4 and $\sigma = (z \mid a)$)

The result shows that there is no model for this set of formulae and therefore that:

$$\exists z\ p(f(z))$$

is a consequence of the formulae:

$$\forall x\ (p(x) \rightarrow p(f(x))),\ \exists y\ p(y))$$

A study of this example shows that the deduction is in fact quite natural. Intepreting $\neg p(x) \vee p(f(x))$ as the equivalent $\forall x (p(x) \rightarrow p(f(x)))$ we have, as a particular case:

$$p(a) \rightarrow p(f(a))$$

from which p(f(a)) follows because we are given p(a). Then interpreting $\neg p(f(z))$ as $\forall z \neg p(f(z))$, we have the contradiction $\neg p(f(a))$ and hence $\square$.

Proposition 2: a set $\mathscr{A}$ of clauses of predicate calculus is unsatisfiable iff $\mathscr{A} \vdash_{RES} \square$

There is a detailed proof in Loveland (1978); the following is an outline, the steps in the argument being the same as in Proposition 1.

(a) Only the case of finite $\mathscr{A}$ need be considered.

(b) It is shown that if $\mathscr{A} \vdash_{RES} \square$ can be deduced in a single step then $\mathscr{A}$ is unsatisfiable.

(c) It is shown that if $\mathscr{A}_1 \vdash_{RES} \mathscr{A}_2$ and M is a model of $\mathscr{A}_1$ then M is a model of $\mathscr{A}_2$: this is established by an obvious induction argument. Thus $\mathscr{A} \vdash_{RES} \square$ implies that $\mathscr{A}$ is unsatisfiable because there is no model of $\square$, i.e. the condition is sufficient.

(d) To show that the condition is necessary:
   1. Herbrand's theorem is used to construct a finite unsatisfiable set B of formulae without variables by substituting terms for variables in the formulae of $\mathscr{A}$.
   2. Proposition 1 is used to deduce $\square$ from B in GR.
   3. This deduction is translated into a deduction in RES.

Δ

The following examples illustrate the deduction process.

Example 4 (from Loveland, 1978)

The formulae we consider in this example are axioms of group theory; so as to avoid the need to use the predicate '=' we introduce a predicate in three variables p(x, y, z), to be interpreted as x.y = z

A_1 $\forall x \forall y \exists z\ p(x, y, z)$
(closure)

A_2 $\forall x \forall y \forall z \forall u \forall v \forall w\ ((p(x, y, u) \wedge p(y, z, v)) \rightarrow (p(x, v, w) \leftrightarrow p(u, z, w))$
(associative law)

A_3 $\exists$ x ($\forall$ y p(x, y, y) $\wedge$ $\forall$ z $\exists$ u p(u, z, x))
(existence of a left unit and of a left inverse)

We want to establish the existence or otherwise of a right inverse, i.e. is the formulae B a consequence of the set $\{A_1, A_2, A_3\}$ where:

B = $\exists$ x ($\forall$ y p(x, y, y) $\wedge$ $\forall$ z $\exists$ u p(z, u, x))

The negation of B is:

$\neg$ B = $\forall$ x ($\exists$ y $\neg$ p(x, y, y) v $\exists$ z $\forall$ u $\neg$ p(z, u, x))

and the existence is established if it can be shown that:

$\mathscr{S}$ = $\{A_1, A_2, A_3, \neg B\}$ is unsatisfiable

Writing $\mathscr{S}$ in clausal form we have:

a_1 p(x, y, f(x, y))
a_2 $\neg$ p(x, y, u) v $\neg$ p(y, z, v) v $\neg$ p(x, v, w) v p(u, z, w)
a_3 $\neg$ p(x, y, u) v $\neg$ p(y, z, v) v $\neg$ p(u, z, w) v p(x, v, w)
a_4 p(e, y, y)
a_5 p(g(z), z, e)
a_6 $\neg$ p(x, h(x), h(x)) v $\neg$ p(k(x), u, x)

and the deduction of $\square$ is as follows:

f_1 : p(x, y, f(x, y))	(a_1)
f_2 : $\neg$ p(x, y, u) v $\neg$ p(y, z, v) v $\neg$ p(x, v, w) v p(u, z, w)	(a_2)
f_3 : $\neg$ p(x, y, u) v $\neg$ p(y, z, v) v $\neg$ p(u, z, w) v p(x, v, w)	(a_3)
f_4 : p(e, y, y)	(a_4)
f_5 : p(g(z), z, e)	(a_5)
f_6 : $\neg$ p(x, h(x), h(x)) v $\neg$ p(k(x), z, x)	(a_6)
f_7 : $\neg$ p(k(e), z, e)	(r., f_4, f_6)
f_8 : $\neg$ p(x, y, k(e)) v $\neg$ p(y, z, v) v $\neg$ p(x, v, e)	(r., f_2d, f_7)
f_9 : $\neg$p(g(v), y, k(e)) v $\neg$ p(y, z, v)	(r., f_5, f_8c)
f_{10}: $\neg$ p(g(v), e, k(e))	(r., f_4, f_9b)
f_{11}: $\neg$ p(g(v), y, u) v $\neg$ p(y, z, e) v $\neg$ p(u, z, k(e))	(r., f_3d, f_{10})
f_{12}: $\neg$ p(g(v), y, e) v $\neg$ p(y, k(e), e)	(r., f_4, f_{11}c)
f_{13}: $\neg$ p (g(v), g(k(e)), e)	(r., f_5, f_{12}b)
f_{14}: $\square$	(r., f_5, f_{13})

The graph of this deduction is shown in Fig. 10.

Example 5

Taking the formulae of Chapter 6 (Section 4):

A_1 = $\neg$ p(x) v q(x, f(x))
A_2 = $\neg$ p(x) v r(f(x))

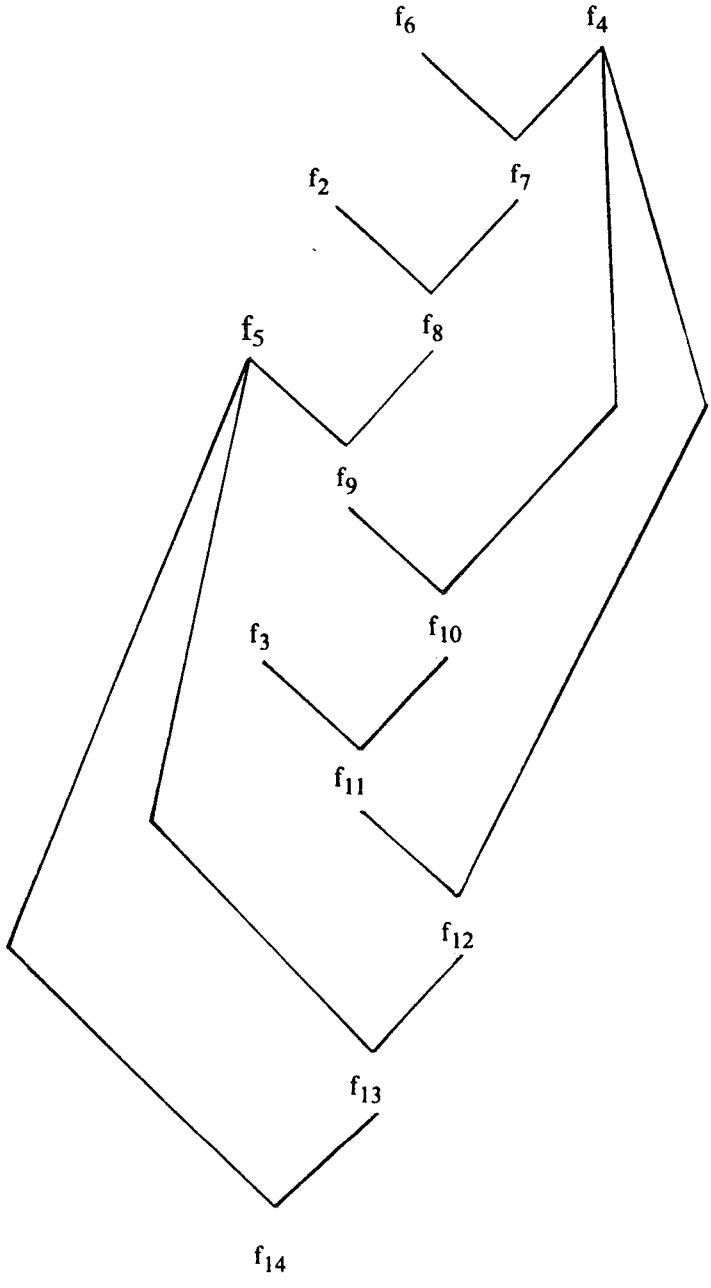


Fig. 10. Graph of deduction of □ from Example 4.

$A_3 = p(a)$
$A_4 = \neg\, q(x, y)$

the deduction of the empty clause is as follows:

f_1: $\neg\, p(x)$ v $q(x, f(x))$	(A_1)
f_2: $p(a)$	(A_3)
f_3: $q(a, f(a))$	(res with f_1, and f_2)
f_4: $\neg\, q(x, y)$	(A_4)
f_5: $\square$	(res with f_3 and f_4)

Example 6

To illustrate the advantage of resolution over using Herbrand's theorem, consider the pair of clauses:

$p(a, x_2, f(x_2), x_4, f(x_4), \ldots, x_{2n}, f(x_{2n}))$
$\neg\, p(x_1, f(x_1), x_3, f(x_3), x_5, \ldots, f(x_{2n-1}), x_{2n+1})$

where n is a given integer.

Resolution gives the empty clause in one step. With Herbrand's method the formulae have to be considered in different classes as shown in Chapter 6: first those involving a, then those with a and f(a), and so on, and the truth value of a formula $B_0 \wedge B_1 \wedge \ldots \wedge B_m$ investigated; the contradiction will not be found until $m = 2.(2n)^{2n}$ at least. For $n = 20$ this is beyond the power of any foreseeable computer.

Example 7

Given the following statements:

S1 Given a crime, there is someone who has committed it.
S2 Only criminals commit crimes.
S3 Only criminals are arrested.
S4 Criminals who are arrested do not commit crimes.
S5 Crimes are committed.

we seek to prove that:

B There are criminals who are not arrested.

We introduce the following predicates:

cr(x) : x is a crime
cm(y) : y is a criminal
co(y, x): y commits x
ar(y) : y is arrested

The statements S1 to S5 are, in terms of these,

A1 $\forall\, x\, (cr(x) \rightarrow \exists\, y\, co(y)))$
A2 $\forall\, y\, \forall\, x((cr(x) \wedge co(y,x)) \rightarrow cm(y)$

A3 $\forall$ y (ar(y) $\rightarrow$ cm(y))
A4 $\forall$ y ((cm(y) $\wedge$ ar(y)) $\rightarrow$ $\neg$ $\exists$ x (cr(x) $\wedge$ co(y,x)))
A5 $\exists$ x cr(x)
A6 $\neg$ $\exists$ y (cm(y) $\wedge$ $\neg$ ar(y))

which in clause form are as follows:

a1 : $\neg$ cr(x) v co(f(x),x)
a2 : $\neg$ cr(x) v $\neg$ co(y,x) v cm(y)
a3 : $\neg$ ar(y) v cm(y)
a4 : $\neg$ cm(y) v $\neg$ ar(y) v $\neg$ cr(x) v $\neg$ co(y,x)
a5 : cr(a)
a6 : $\neg$ cm(y) v ar(y)

To show that B follows from the set S1 to S5 we have to find a deduction in RES of the empty clause from a1 to a6. We start by setting $f_1 = a_1$, ..., $f_6 = a_6$ and continue:

f_7 : co(f(a),a)	(res f_5, f_1)	someone has committed the crime
f_8 : $\neg$ cr(a) v cm(f(a))	(res f_7, f_2)	
f_9 : cm(f(a))	(res f_8, f_5)	this person is a criminal
f_{10} : $\neg$ cm(f(a)) v $\neg$ ar(f(a)) v $\neg$ cr(a)	(res f_7, f_4)	
f_{11} : $\neg$ cm(f(a)) v $\neg$ ar(f(a))	(res f_{10}, f_5)	
f_{12} : $\neg$ ar(f(a))	(res f_{11}, f_9)	and has not been arrested
f_{13} : $\neg$ cm(f(a))	(res f_{12}, f_6)	
f_{14} : $\square$	(res f_{13}, f_9)	contradicting A6

The graph of this deduction is shown in Fig. 11.

Example 8

So far we have not used the diminution rule. It is, however, an essential rule, as the following example will show.

If S is the set of clauses:

S = {p(x) v p(y), $\neg$ p(x) v $\neg$ p(y)}

the empty clause cannot be deduced by using the resolution rule alone because the result of applying this to a pair of clauses with two literals is a clause with two literals. But with the diminution the deduction goes as follows:

f_1 : p(x) v p(y)	(given)
f_2 : p(x)	(dim f_1)

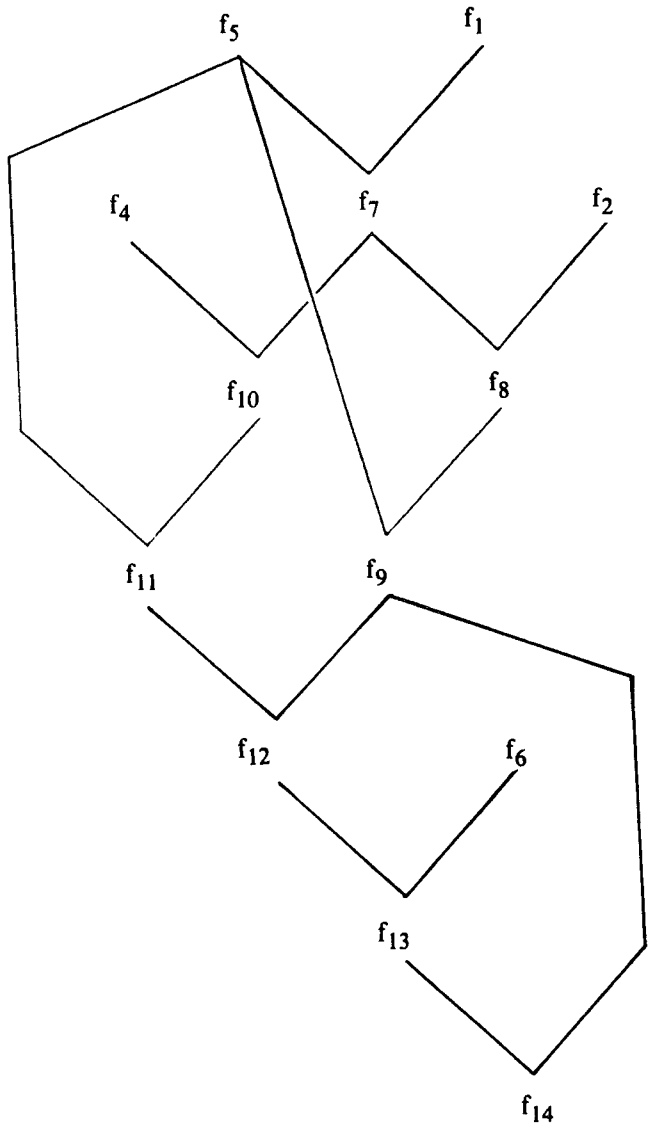


Fig. 11. Graph of deduction in Example 7.

f_3 : $\neg p(x) \vee \neg p(y)$	(given)
f_4 : $\neg p(y)$	(res f_2, f_3)
f_5 : $\square$	(res f_2, f_4)

Nevertheless, we see in Section 5 that there are circumstances in which the diminution rule is not necessary.

5. Strategies for the use of the resolution method

There are two fundamentally different ways of using the resolution method for automated theorem proving, each of which gives rise to a number of variants; we may describe these as processing sets of clauses and searching trees, respectively. For the first we start with a set S_0 of clauses and try to

establish whether or not this is satisfiable, applying the resolution and diminution rules both to extend and to simplify the original set; the operations are guided by a strategy decided at the start and are continued until either the empty clause is reached or it is seen that this will never be reached. For the second we traverse the deductions' tree, abandoning certain branches as unprofitable, until again we either reach the empty clause or can be certain that we shall never do so. We now describe the two methods in detail.

A. Processing Sets of Clauses

There are at least three variants of this strategy.

Saturation

This is certainly the simplest and the most natural. Starting with the set S_0 whose satisfiability we are investigating, we perform all the possible resolutions and diminutions, resulting in a number of new clauses which we add to S_0, giving S_1 say. We then carry out the same process with S_1, getting S_2, and so on, stopping either when $\square$ appears in some S_i or when we find $S_{i+1} = S_i$.

Example

Thus if:

$$S_0 = \{A, \neg A \vee B, \neg B \vee A, \neg B\}$$
$$S_1 = \{B, B \vee \neg B, \neg A \vee A, \neg A, A\} \cup S_0$$
$$S_2 = \{A, \square, \ldots\} \cup S_1$$

Proposition 2 guarantees that if S_0 is unsatisfiable then this will become known in a finite time because $\square$ will then appear in some set S_i. Thus the method is complete, in the sense that it will establish unsatisfiability when this is the case, and sound, in the sense that it will not show as unsatisfiable a set that is in fact satisfiable. But it is not a practical method because the sizes of the sets S_i increase exponentially. From a theoretical point of view it has no advantage over the algorithm of Chapter 6, for if S_0 is satisfiable it can happen that S_{i+1} differs from S_i for every i, so the algorithm never terminates. This is the case, for example, for:

$$S_0 = \{p(a), \neg p(x) \vee p(f(x))\}$$

Saturation with simplification

It is useful, before describing this method, to discuss the concepts of tautological clauses and subsumed clauses, respectively.

It is easy to check that all tautological clauses, that is, clauses that are true for any interpretation, are of the form:

$$A \vee \neg A \vee C$$

where A is an atom and C is a clause. For example:

$$p(f(x)) \vee q(x) \vee \neg r(x) \vee \neg p(f(x))$$

is a tautology.

A clause C is said to *subsume* a clause D if there is a substitution σ such that:

$$D = \sigma C \vee f$$

where f is a clause.

For example:

p(x) subsumes p(f(a)) v q(a)

and:

p(x) v q(a) subsumes p(a) v q(a) v r(f(x)) v r(b)

If C subsumes D, we say that D is subsumed by C, and there is the following equivalence:

> [the set S of clauses is satisfiable] iff [the set S′ is satisfiable, where S′ is derived from S by removing all its tautologies and every clause that is subsumed by some other clause]

It is obvious that any interpretation that satisfies S′ satisfies S also, and conversely; this is the very simple idea underlying the method of saturation with simplification. We start as before, but at each step we simplify S_i by deleting all tautologies and all clauses subsumed by any other clause of the set. As before, the strategy is both complete and sound.

For example:

$S_0 = \{r(a) \vee p(a), p(x) \vee \neg r(b), r(b), \neg p(b), r(a) \vee r(c) \vee \neg r(a)\}$
$S_o' = \{r(a) \vee p(a), p(x) \vee \neg r(b), r(b), \neg p(b)\}$
$S_1 = \{r(a) \vee p(a), p(x) \vee \neg r(b), r(b), \neg p(b), p(x), \neg r(b)\}$
$S_1' = \{r(b), \neg p(b), p(x), \neg r(b)\}$
$S_2 = \{\square, \ldots\}$

showing that the calculation is much simpler than straightforward saturation. Further simplifications can be made if evaluable predicates are used, cf. Nilsson (1980).

Choice of simple clauses

The basic idea here is to construct at each step, not every possible clause that can be derived from the existing clauses, as in the two previous methods, but a single 'well chosen' clause to be added to S_i to give S_{i+1}. The key problem, of course, is the criterion for the choice, and there are several possibilities.

One method would be to make a purely random choice; another to take the first in some order, an order having been defined for the possible resolutions and diminutions, depending for example on an enumeration of the clauses or of the predicates. But the most natural choice would be to take the shortest of the clauses derivable from those of S_i, meaning the clause with the smallest number of literals: the principle being that since we are seeking to arrive at the clause having no literals we shall get there most quickly if we keep all our clauses as short as possible.

For example:

$$S_0 = \{r(x) \vee \neg p(x), \neg r(b), \neg r(c) \vee q(x) \vee r(f(a)), p(b)\}$$

$$S_1 = S_0 \cup \{\neg p(b)\}$$

$$S_2 = S_1 \cup \{\square\}$$

Used crudely, this strategy is not complete: for example:

$$S_0 = \{\neg p(x) \vee p(f(x)), p(a), \neg p(a) \vee \neg p(b) \vee \neg p(c), p(b), p(c)\}$$

$$S_1 = S_0 \cup \{p(f(a))\}$$

$$S_2 = S_1 \cup \{p(f(f(a)))\}$$

etc.

This never gives the empty clause, while taking the four last clauses of S_0 and accepting a derived clause with two literals gives $\square$ in three steps.

The method can be made complete by reverting from time to time to a saturation step.

Another possibility is to give preference to all the resolutions that lead to 'unit clauses'—clauses having only a single literal. This again, used crudely, is not a complete strategy.

Several methods can be combined: for example, by introducing simplification steps as appropriate, or by adapting the strategy to the problem in hand. It is always important to be aware that one's strategy may not be guaranteed to be complete (that is, to lead to the empty clause with certainty when the starting set is not satisfiable), and that even if it is complete this may be difficult to prove (see Chang and Lee, 1973; Loveland, 1978).

B. Searching the Deduction Tree

By a deduction tree for a set of clauses we mean the tree whose root is $(\cdot)$ and each branch of which, starting from the root, passes through the nodes f_1, $f_2, \ldots, f_n \ldots$ that constitute a deduction in either of the formal systems, GR or RES, such a tree will most often be of infinite extent.

Thus for the set $\mathscr{C} = \{A, \neg A \vee B, \neg B\}$ the tree is shown in Fig. 12.

Clearly, each branch of this tree is a deduction; for example, the leftmost branch is:

f_1 : A (given)

f_2 : $\neg A \vee B$ (given)

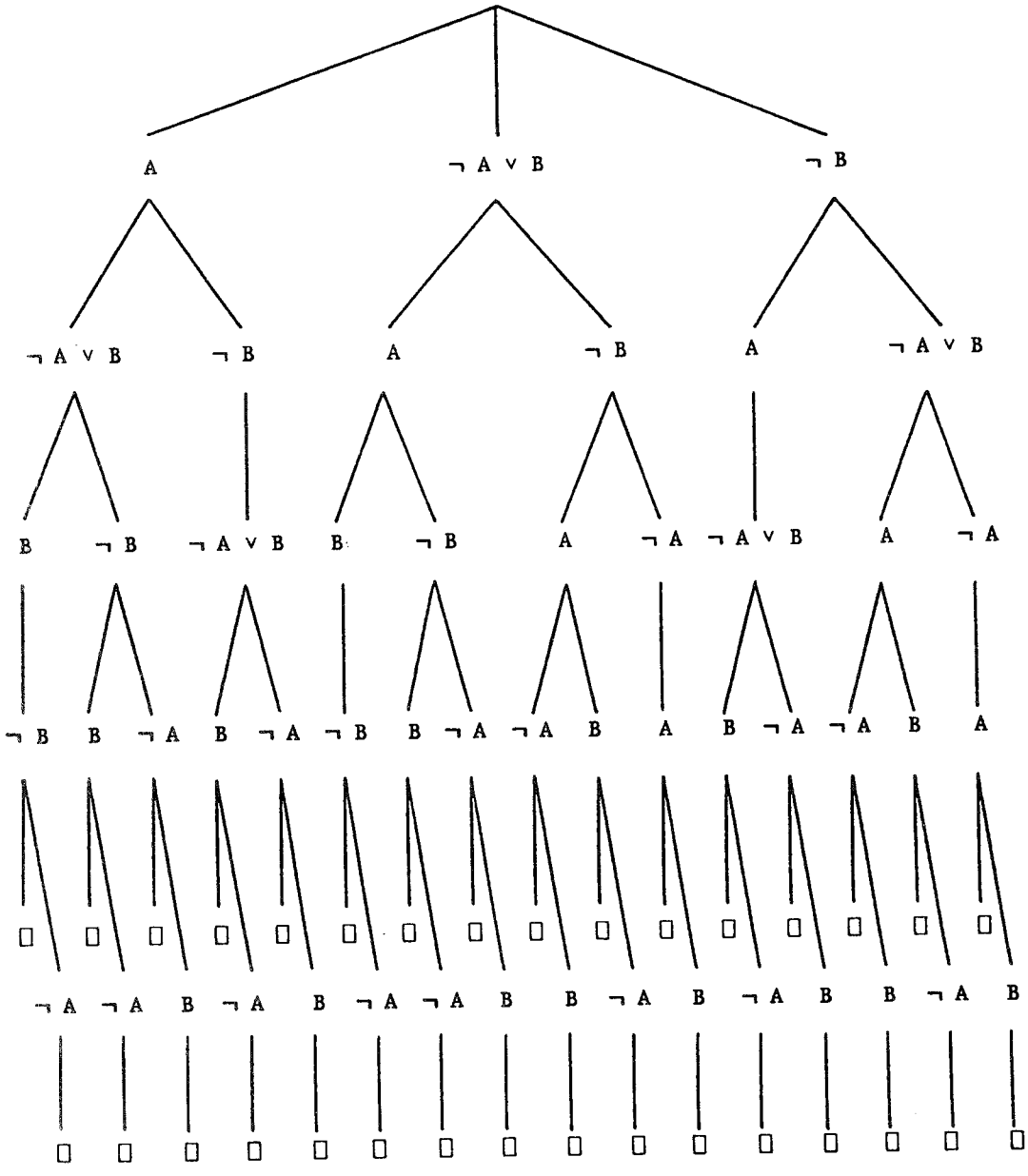


Fig. 12. Deduction tree for the set $\mathscr{C} = \{A\ \neg A \vee B,\ \neg B\}$.

f_3 :	B	(res f_1, f_2)
f_4 :	$\neg$ B	(given)
f_5 :	□	(res f_3, f_4)

In constructing the tree we restrict ourselves to deductions in which the same formula never occurs more than once: if in any deduction the same formula occurs twice, that deduction is unnecessarily long. No generality is lost as a result of this constraint. The same applies to a limitation to deductions that start by introducing the hypotheses in a fixed order; this would give the tree from Fig. 12 the form shown in Fig. 13, when only deductions which begin with the three formulae A, $\neg$ A v B, B (in this order) are considered.

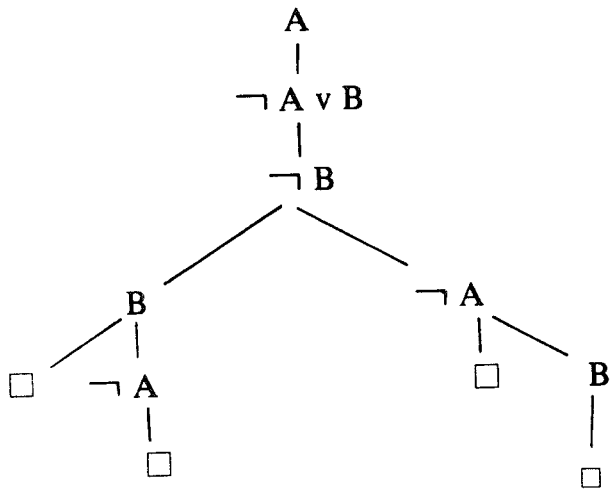


Fig. 13. Tree from Fig. 12 when the hypotheses are introduced in a fixed order.

Constraints of this type that sacrifice no generality, and which above all do not cause the empty clause to be missed when it should be found, enable us to develop strategies for searching the tree. A strategy will be based on two choices:

1. the choice of a sub-tree of the complete tree, determined by some limitation on the deductions to be taken into account;
2. the choice of a procedure for conducting the search of this sub-tree.

For the second, we consider only the two well known methods:

1. search by 'depth first with backtracking',
2. search by 'breadth first'.

These concepts are illustrated in Figs 14 and 15, respectively.

There are advantages and disadvantages in each method. The breadth first method has the advantage in the case of a tree in which each node has a finite number of descendants (as will always be true in our context) of allowing the entire tree to be searched, even, as will often be our situation, if it is infinite; and the disadvantage that if each node has a large number of descendants (the 'branching factor' is high) it takes a very long time to descend to a significant depth. The main advantage of the depth first with backtracking method is that when the search is for an object that occurs several times but rather deep in the tree (our situation again) it may find this quite quickly; another advantage is that it is easy to program and does not demand a large memory capacity. The main disadvantage of this method is that it may follow indefinitely the first infinite branch that it meets, and therefore can be relied upon only to search finite trees.

Figure 16 illustrates the conditions under which breadth first is at a relative disadvantage and Fig. 17 for depth first correspondingly.

As we have said, when we have a finite set of clauses $\mathscr{C} = \{c_0, c_1, \ldots, c_n\}$ we lose nothing if we limit ourselves to deductions starting $c_0\ c_1 \ldots c_n$ in this order; so from now on we consider only such deductions, which we write $\mathscr{C}\ f_0\ f_1 \ldots f_m$ to denote $c_0\ c_1 \ldots c_n, f_0\ f_1 \ldots f_m$. Taking the previous example $\mathscr{C} = \{A, \neg A \vee B, \neg B\}$ again, we can now show the tree as in Fig. 18.

All the search strategies that we consider from now on are sound, leading to the empty clause only when $\mathscr{C}$ is unsatisfiable; this is simply because they search a sub-tree of the full tree of all possible deductions and if $\mathscr{C}$ is satisfiable the empty clause does not occur anywhere in the full tree and therefore nowhere in any sub-tree. On the other hand, the question of completeness arises in every case studied.

We now consider some different types of strategy.

General strategies

This is analogous to the saturation strategies of the clause-processing type of method, and consist simply in searching the complete tree. Breadth

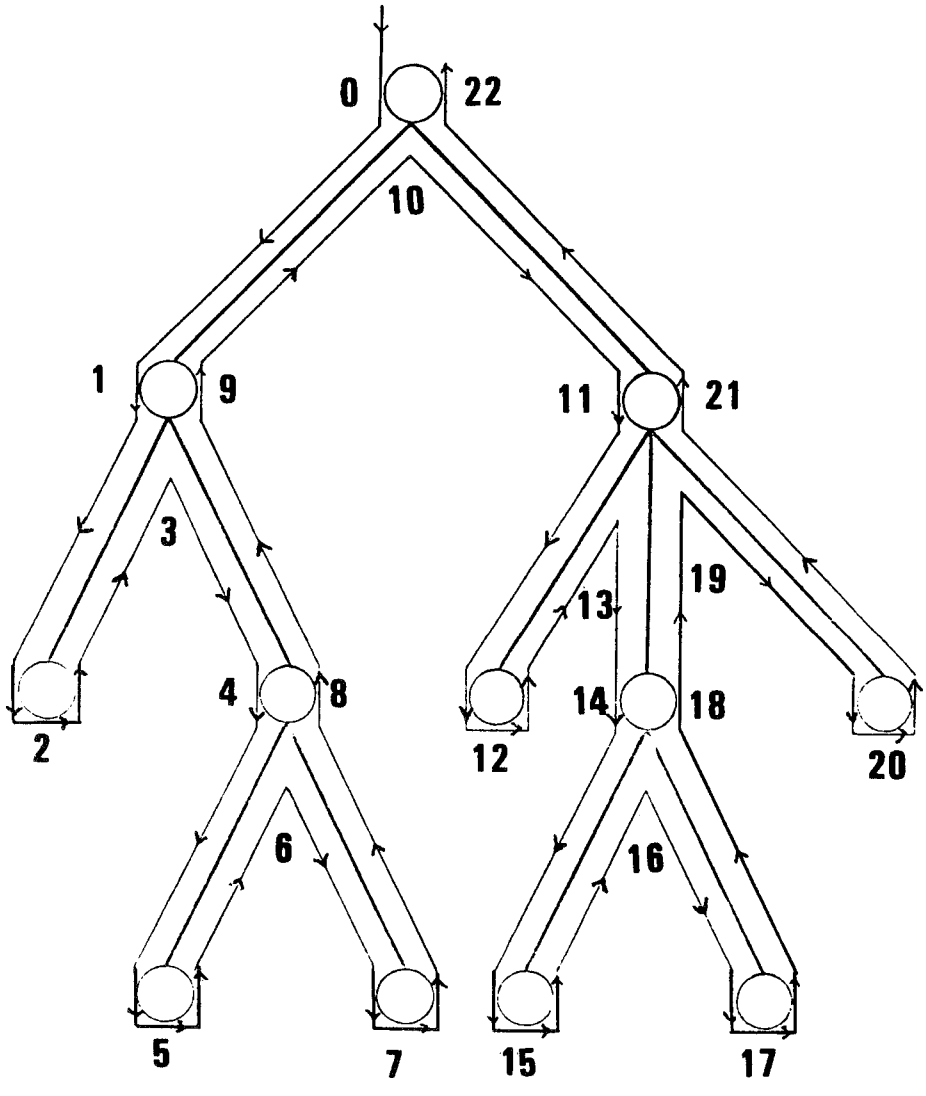


Fig. 14. Depth first search with backtracking.

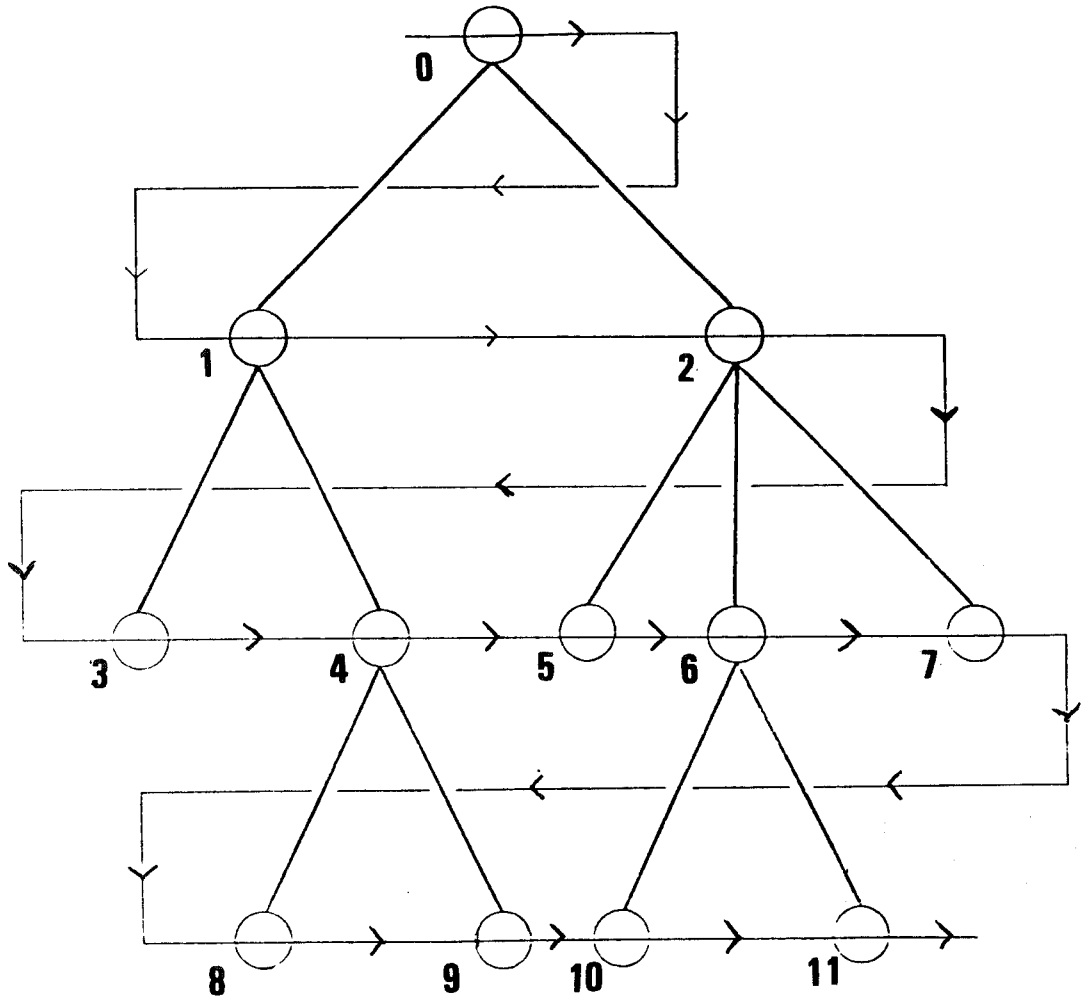


Fig. 15. Breadth first search.

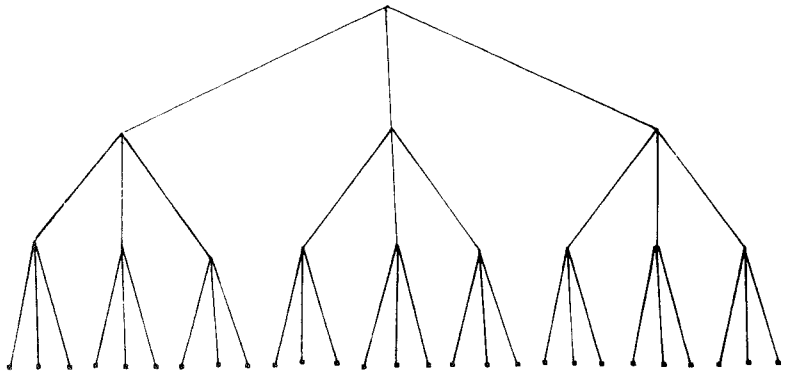


Fig. 16. Breadth first disadvantages.

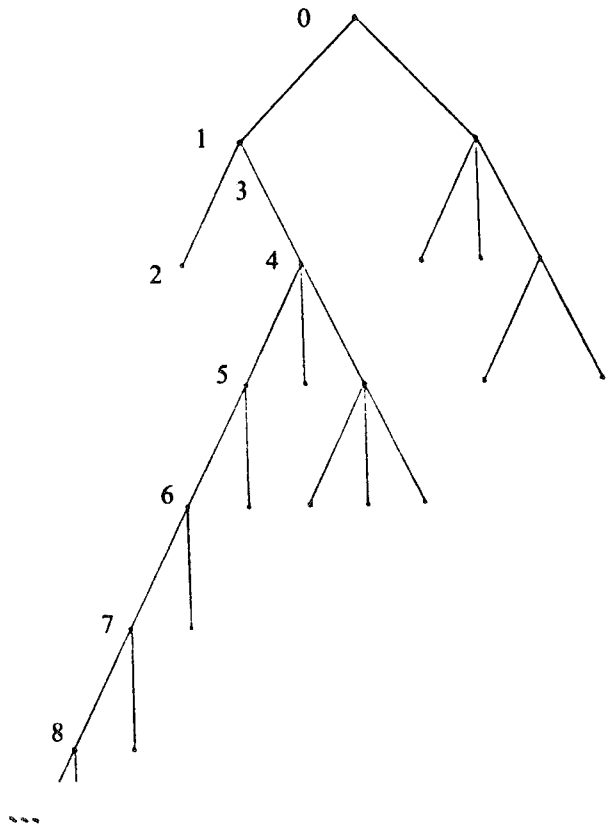


Fig. 17. Depth first disadvantages.

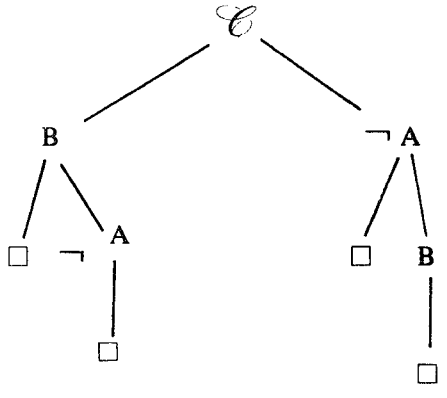


Fig. 18. Different form of tree in Fig. 12.

first is both a complete and a sound method but depth first with backtracking is a sound but not complete method.

These are very inefficient methods, more so even than saturation, because they treat as different deductions that merely change the order of the formulae.

Linear strategies

With $\mathscr{C}$ as above, we define a linear deduction with root c_0 as any deduction:

$\mathscr{C}$ f_0 f_1 ... f_m

such that:

1. f_0 is derived by resolution or diminution with clauses, one of which is c_0;
2. f_i, $i > 0$, is derived by resolution or diminution with clauses, one of which is f_{i-1}.

Example

For example, if:

$$\mathscr{C} = \{\neg A \vee \neg B, A \vee \neg C, C, B \vee \neg D, D \vee B\}$$

so:

$$c_0 = \neg A \vee \neg B$$

the deduction is:

$\mathscr{C}$

f_0 :	$\neg B \vee \neg C$	(res with c_0, $A \vee \neg C$)
f_1 :	$\neg B$	(res with f_0, C)
f_2 :	$\neg D$	(res with f_1, $B \vee \neg D$)
f_3 :	B	(res with f_2, $D \vee B$)
f_4 :	$\square$	(res with f_3, f_1)

The corresponding graph is shown in Fig. 19.
This is not the only linear deduction with root c_0; another is as follows:

f'_0 :	$\neg B \vee \neg C$	(res with c_0, $A \vee \neg C$)
f'_1 :	$\neg C \vee D$	(res with f'_0, $D \vee B$)
f'_2 :	$B \vee \neg C$	(res with f'_1, $B \vee \neg D$)
f'_3 :	B	(res with f'_2, C)
f'_4 :	$\neg C$	(res with f'_3, f'_0)
f'_5 :	$\square$	(res with f'_4, C)

Even for this simple case the complete tree of linear deduction with root c_0 is extensive; Fig. 20 shows just a part.

In this diagram the two branches marked with double lines correspond to the linear deductions $\mathscr{C}$ f_0 f_1 f_2 f_3 f_4, $\mathscr{C}$ f'_0, f'_2, f'_3, f'_4 and f'_5 respectively.

It can be shown (see Chang and Lee, 1973; Loveland, 1978) that if there is a deduction of the empty clause, starting from $\mathscr{C}$, then there is also a linear deduction; further, if $\mathscr{C} = \mathscr{C}' \cup \{c_0\}$ with $\mathscr{C}'$ satisfiable and $\mathscr{C}$ unsatisfiable, then there is a linear deduction starting from $\mathscr{C}$ and with root c_0. It follows that if the problem is to find if a theorem T can be deduced from $\mathscr{C}'$ this can

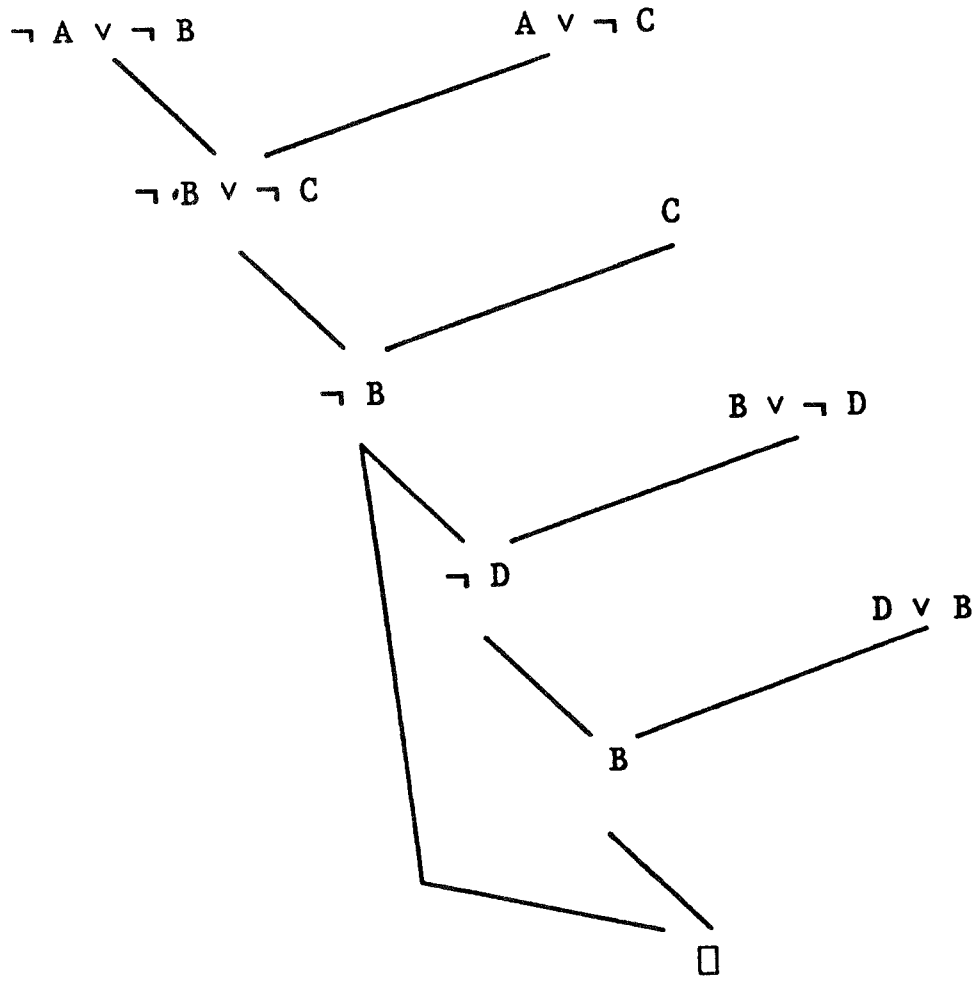


Fig. 19. Graph showing linear strategies.

be attacked by searching for the empty clause in the tree of linear deductions with root $c_0 = \neg T$.

As before, a breadth first search will always find the empty clause if it is there, while a depth first search may become trapped in an infinite branch and never succeed; this would be the case for:

$$\mathscr{C} = \{ \neg p(x) \vee p(f(x)),\ \neg p(a),\ p(x)\}$$
$$c_0 = p(x)$$

which is shown in Fig. 21, where the search of the first branch never ends although the second branch gives the empty clause immediately.

Input strategies

By this we mean any deduction $\mathscr{C}\ f_0\ f_1 \ldots f_m$ with root $c_0 \in \mathscr{C}$ such that:

1. f_0 is derived by resolution or diminution with clauses, one of which is c_0;
2. f_i, $i > 0$, is derived by resolution or diminution with clauses, one of which is contained in $\mathscr{C}$.

Clearly, if at any step f_{i-1} is not used in deriving f_i, a part of the deduction will not be used from then on; so nothing is lost if the further condition that the deduction is linear is imposed, and this we shall do.

Example

If:

$\mathscr{C} = \{\neg A, A \vee \neg B, A \vee \neg C \vee \neg D, C, D \vee \neg C\}$

$c_0 = \neg A$

the following is a linear input deduction:

$\mathscr{C}$

f_0: $\neg C \vee \neg D$ (res c_0, $A \vee \neg C \vee \neg D$)

f_1: $\neg D$ (res f_0, C)

f_2: $\neg C$ (res f_1, $D \vee \neg C$)

f_3: $\square$ (res f_2, C)

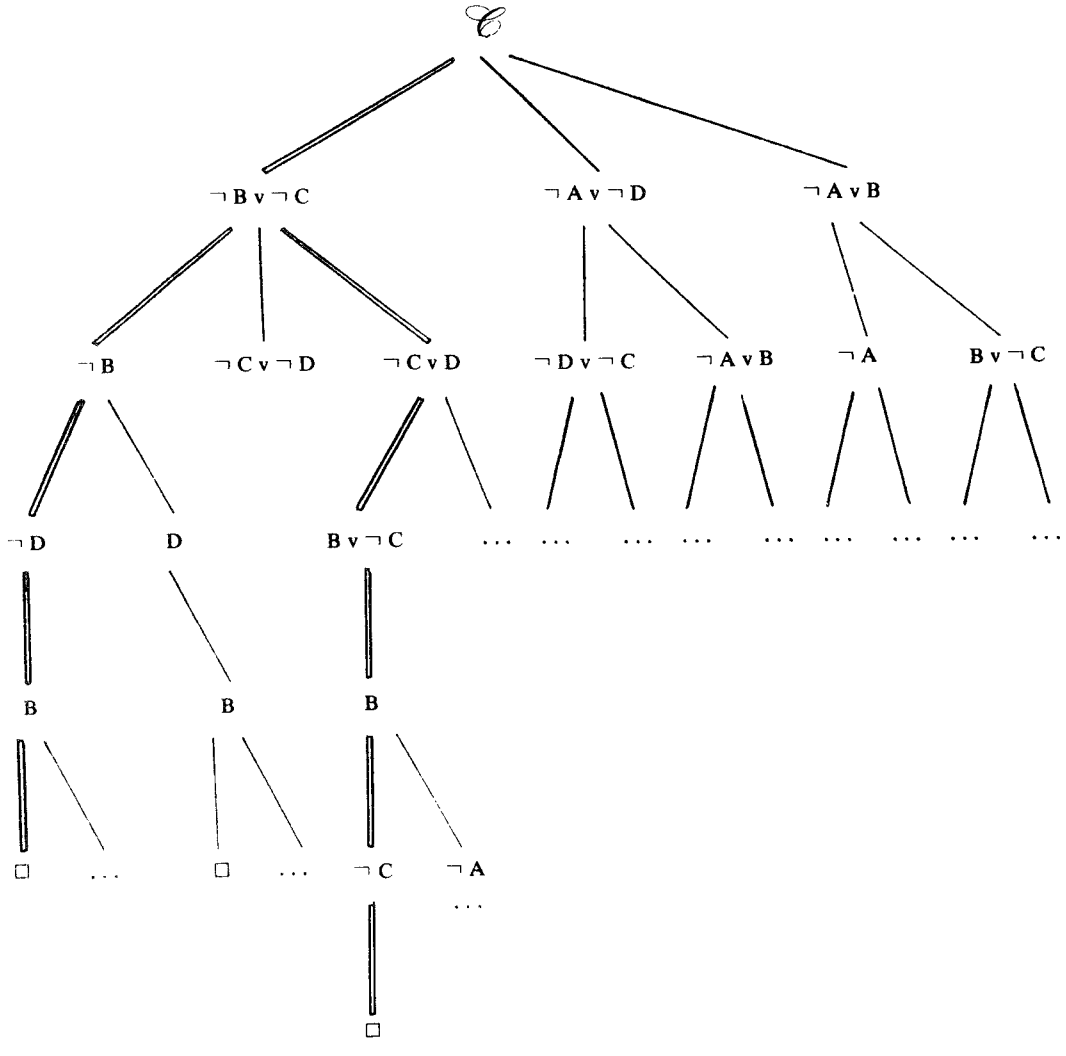


Fig. 20. Part of the extensive tree of linear deductions with root c_0.

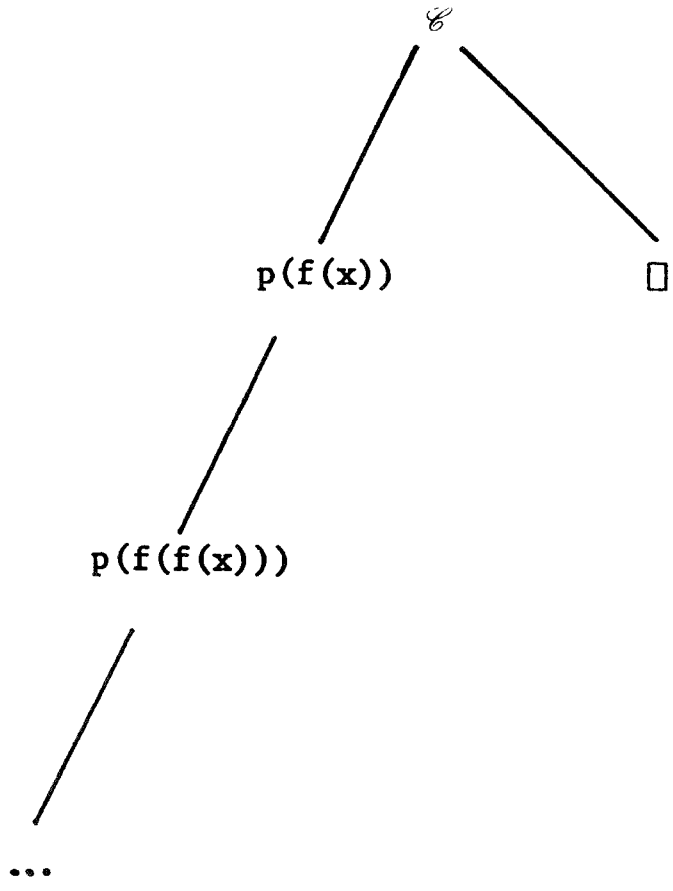


Fig. 21. The search of the first branch never ends although the second branch gives the empty clause immediately.

The complete deduction tree for this example is shown in Fig. 22. It will be seen that one of the branches leads to a dead end without reaching the empty clause; this shows that even in a simple case with a finite tree a 'depth first' search strategy must include also backtracking if it is to succeed.

In contrast to the linear strategies situation even breadth first input strategy is not complete: it can happen that the empty clause does not occur anywhere in the tree of input deductions, even though it does occur in the complete tree of unconstrained deductions. This is the case, for example, for:

$$\mathscr{C} = \{A \vee B, A \vee \neg B, \neg A \vee B, \neg A \vee \neg B\}$$

and the result can be stated: even if $\mathscr{C}$ is unsatisfiable this may not be revealed by any input strategy search.

There is however the following important result (see Henschen and Wos, 1974):

> if $\mathscr{C} = \mathscr{C}' \cup \{c_0\}$ is unsatisfiable, $\mathscr{C}'$ contains only clauses with precisely one positive literal and c_0 contains only negative literals, then there is an input deduction of the empty clause, with root c_0; and this deduction does not use the diminution rule at any step.

A clause such as c_0, with only negative literals, is called a *negative clause*; and one such as those of $\mathscr{C}$, with only a single positive literal, is called a *Horn clause*.

Consider now the example:

$$\mathscr{C}' = \{\neg\, p(x) \vee p(f(x)),\ \neg\, p(f(y)) \vee \neg\, p(y) \vee r(y),\ p(a)\}$$

$$c_0 = \neg\, r(z)$$

which corresponds to the problem:
can we deduce $\exists\, z\ r(z)$ from the three axioms:

$\forall\, x\ (p(x) \rightarrow p(f(x)))$

$\forall\, y\ ((p(f(y)) \wedge p(y)) \rightarrow r(y))$

$p(a)$

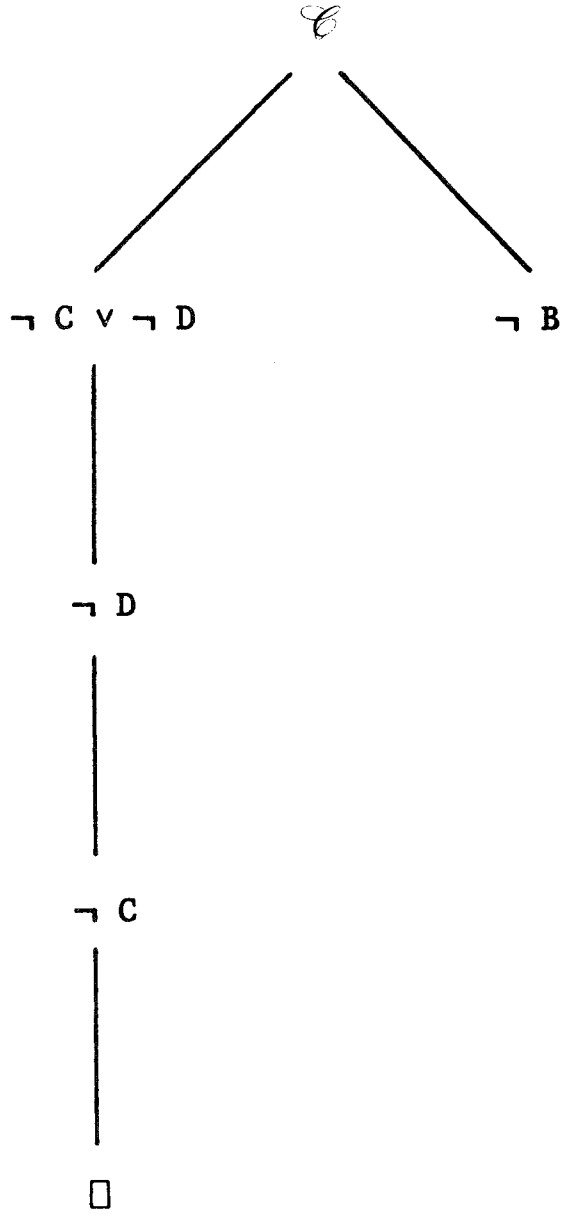


Fig. 22. Linear input deduction tree.

The input deduction with root c_0 is:

	$\mathscr{C}' \cup \{c_0\}$	
f_0 :	$\neg$ p(f(y)) v $\neg$ p(y)	(res with c_0, $\neg$ p(f(y)) v $\neg$ p(y) v r(y))
f_1 :	$\neg$ p(f(a))	(res with f_0, p(a))
f_2 :	$\neg$ p(a)	(res with f_1, $\neg$ p(x) v p(f(x)))
f_3 :	$\square$	(res with f_2, p(a))

The important consequence of the general result stated above is that a breadth first search of the tree of input deductions with root c_0 is both sound and complete in the case of problems of the type $\mathscr{C} = \mathscr{C}' \cup \{c_0,\}$ where $\mathscr{C}'$ is a set of Horn clauses and c_0 is a negative clause.

For the same class of problems and the same tree, search by depth first with backtracking is sound and complete if the tree is finite (and therefore for problems having no variables); but in general it is not complete.

It is a striking fact that the method of backward chaining described in Chapter 1 is simply a particular case of depth first with backtracking search of this tree, the root c_0 being the negation of the goal; thus the completeness of backward chaining follows from the general result given above. The situation is as follows:

1. the rules we are dealing with in backward chaining are of the form:

   $$A_1, A_2, \ldots, A_n \rightarrow B$$

   which are equivalent to:

   $$(A_1 \wedge A_2 \wedge \ldots \wedge A_n) \rightarrow B$$

   which is itself equivalent to:

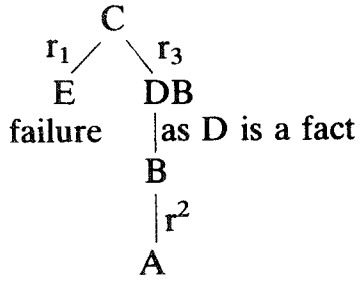
   $$\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_n \vee B$$

   which is a Horn clause.
2. the goal is C and its negation $\neg$ C is a negative clause;
3. the reasoning in backward chaining goes from C to the list of subgoals $A_1, A_2, \ldots, A_n$; this corresponds to $\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_n$ from $\neg$ C, as a step in an input strategy deduction.

Example

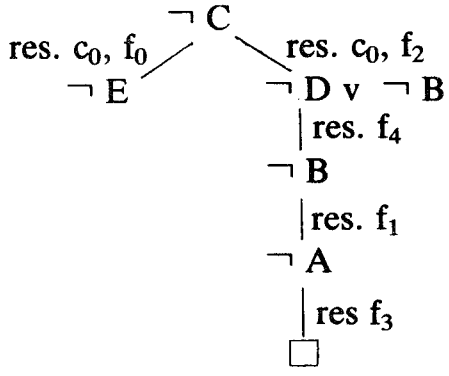
rules:	clauses:	
r1 $E \rightarrow C$	$\neg E \vee C$	f_0
r2 $A \rightarrow B$	$\neg A \vee B$	f_1
r3 $DB \rightarrow C$	$\neg D \vee \neg B \vee C$	f_2
fact: A	A	f_3
D	D	f_4
goal: C	$\neg C = c_0$	

Backward chaining



success as A is a fact

resolution tree



Ordered strategies

So far we have paid no attention to the order in which the literals appear in a clause, so that, for example, A v B and B v A have been considered identical. Here, however, we take this order into account. There are many ways in which this order can be used so as to restrict the number of deductions used when searching the tree (cf. Chang and Lee, 1973; Loveland, 1978). We describe only one, but first a definition.

If C and C′ are two ordered clauses:

$$C = L \vee L_2 \vee \ldots \vee L_n$$
$$C' = L'_1 \vee L'_2 \vee \ldots \vee L'_m$$

where L_1 is a positive literal and L'_1 is a negative literal, the *ordered resolution* of C and C′, when this is possible, gives the resolvant:

$$\Theta\,(L_2 \vee L_3 \vee \ldots \vee L_n \vee L'_2 \vee L'_3 \vee \ldots \vee L'_m$$

where Θ is the greatest unifier of L_1 and L'_1.

The constraints are imposed so that the resolution is performed on the pair of literals at the heads of C and C′ respectively and that the result is ordered by writing first the literals of the clause having a positive literal at its head (C), followed by those of the clause with the negative head (C′). For example:

if:

$$C = p(x) \vee r(x) \vee q(x,y)$$
$$C' = \neg\, p(a) \vee \neg\, r(f(a))$$

ordered resolution is possible and the resolvant is:

$$r(a) \vee q\,(a,y) \vee \neg\, r(f(a))$$

but if:

$$C = p(x) \vee r(x) \vee q(x,y)$$
$$C' = \neg\, r(f(a) \vee \neg\, p(a))$$

ordered resolution is not possible.

We call an *ordered deduction* one that uses only ordered resolutions. For example, if:

$$\mathscr{C} = \{A \vee B, \neg A \vee C, \neg B \vee D, \neg C, \neg D\}$$

an ordered deduction is:

$\mathscr{C}$

f_0: B v C	(res A v B, ¬ A v C)
f_1: C v D	(res f_0, ¬ B v D)
f_2: D	(res f_1, ¬ C)
f_3: □	(res D, ¬ D)

The most important result for us concerning ordered deduction is the following, which sharpens the result given in the section on input strategies.

> If $\mathscr{C} = \mathscr{C}' \cup \{c_0\}$ is unsatisfiable, $\mathscr{C}'$ contains only Horn clauses, each one ordered with the positive literal at the head, and c_0 is a negative clause, then there is a deduction of the empty clause having root c_0 that is ordered and of input type (cf. Apt and Emden, 1982; Lloyd, 1984).

As an example of the use of ordered deduction, Fig. 23 is the tree for:

$$\mathscr{C}' = \{p(x) \vee \neg r(x), p(x) \vee \neg q(f(x)) \vee \neg q(x), q(f(x)) \vee \neg q(x), q(a)\}$$
$$c_0 = \neg p(a)$$

The tree is finite, sparse and contains the empty clause, in agreement with the fact that:

$$\mathscr{C} = \mathscr{C}' \cup \{c_0\}$$

is unsatisfiable.

The results given in the section on input strategies concerning the properties of breadth first and depth first strategies as applied to problems of the form $\mathscr{C} = \mathscr{C}' \cup \{c_0\}$ hold equally well for ordered deductions: that is if $\mathscr{C}'$ consists only of Horn clauses and c_0 is a negative clause, breadth first is always sound and complete but depth first with backtracking is complete only for finite trees.

This underlies the language PROLOG. In this, only Horn clauses are used, and a problem is attacked by devising an algorithm to search the tree of ordered input deductions by depth first with backtracking, the root of the tree being the negation of the result sought. If the tree is finite, the above general result guarantees that a decision will be reached—either the empty clause will be found if the result sought is true or the whole tree will be searched without finding this, showing that the result is not true. This is equivalent to saying that the strategy of using PROLOG resolution is in general incomplete but is complete if the deduction tree is finite.

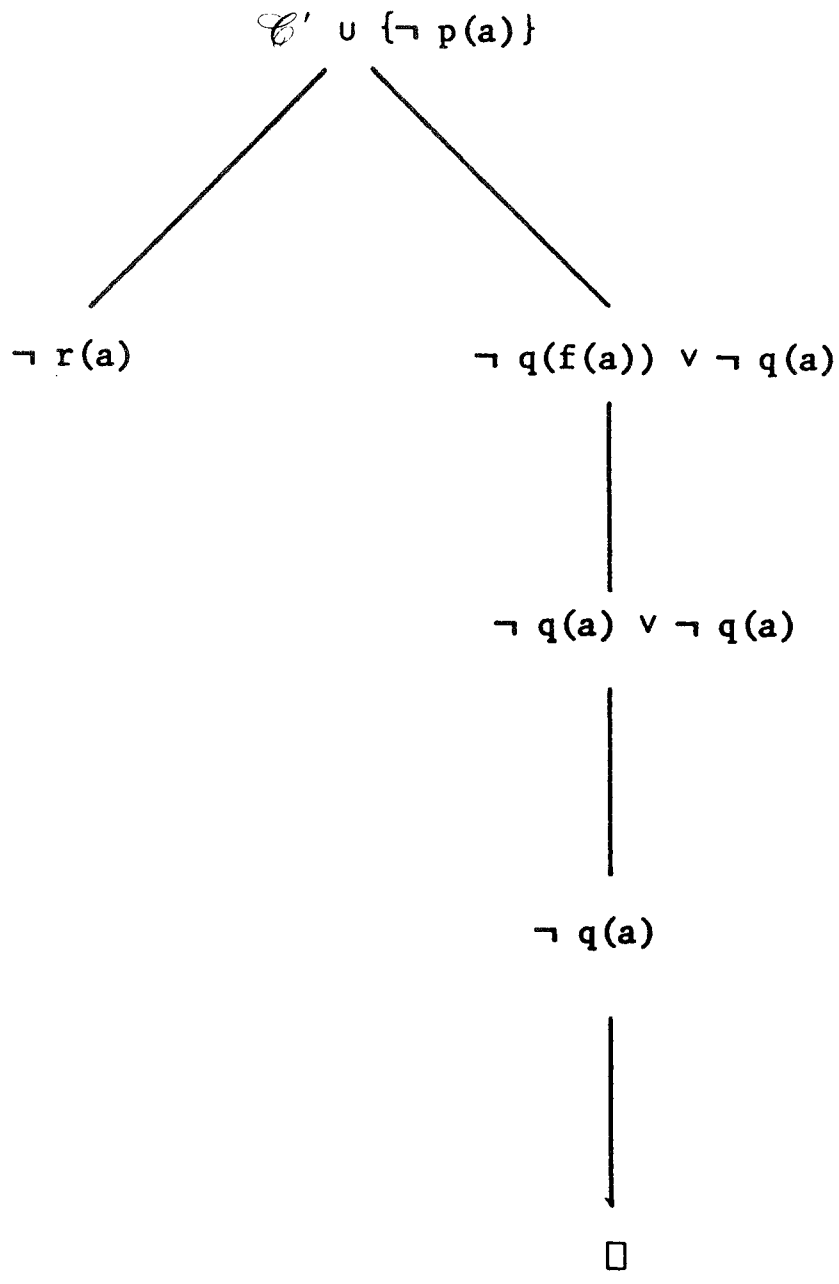


Fig. 23. Ordered deduction tree.

Ordered input strategies with extraction of responses
Consider the following:

$\mathscr{C}' = \{p(a), p(b), r(f(x)) \vee \neg p(x), q(y) \vee \neg r(y), q(c)\}$
$c_0 = \neg q(z)$

The tree of ordered input deductions is shown in Fig. 24.

The empty clause can be deduced in three different ways. The first is:

$\mathscr{C}' \cup \{\neg q(z)\}$

f_0 :	$\neg r(z)$	(res $\neg q(z)$, $q(y) \vee \neg r(y)$ together with the substitution $(y \mid z)$)
f_1 :	$\neg p(x)$	(res $r(f(x)) \vee \neg p(x)$, $\neg r(z)$ together with the substitution $(z \mid f(x))$)
f_2 :	$\square$	(res $\neg p(x)$, $p(a)$ together with the substitution $(x \mid a)$)

In the course of this deduction the variable z is replaced by f(x) and then x by a, thus replacing z by f(a).

The second is similar but using p(b) instead of p(a) in the last resolution, and the third is just the single resolution of $\neg q(z)$ with q(c). The following transformed version of the first derivation:

$\mathscr{C}' \cup \{\neg q(f(a))\}$

f_0 : $\neg r(f(a))$

f_1 : $\neg p(a)$

f_2 : $\square$

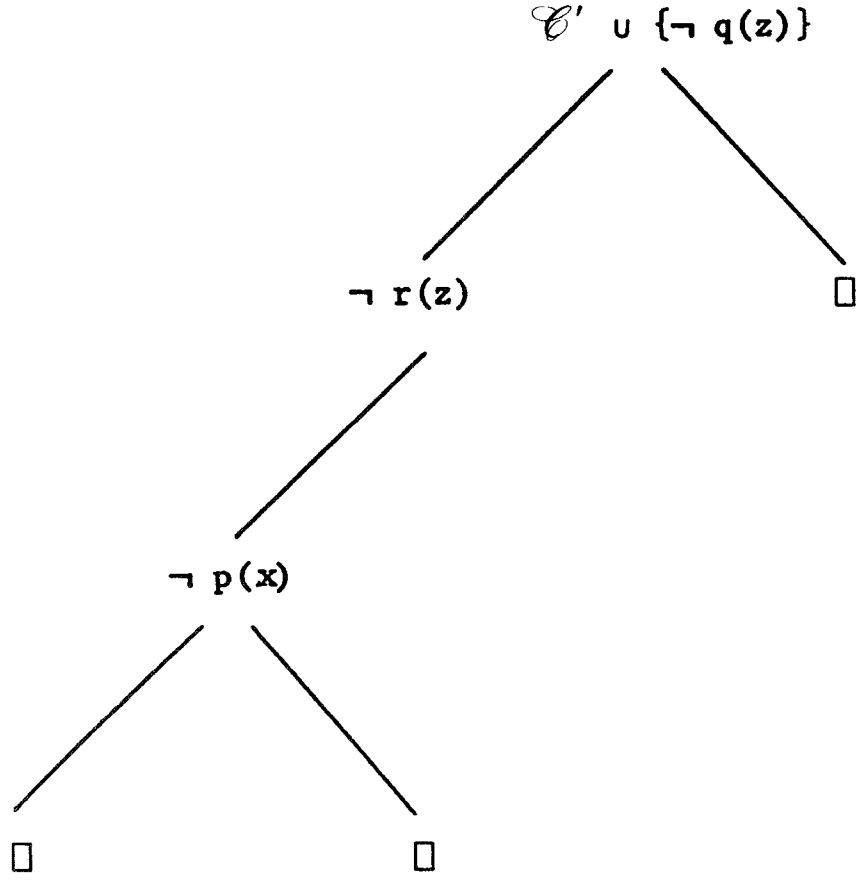


Fig. 24. Ordered input deduction tree.

shows that q(f(a)) is a consequence of $\mathscr{C}'$. Similarly it can be shown that q(f(b)) follows from $\mathscr{C}'$ by transformation of the second deduction, and q(c) by transformation of the third. Thus we can say that for clauses with variables, such as $\neg$ q(z), every deduction of the empty clause gives an object t, an element of the Herbrand universe, such that q(t) is a consequence of $\mathscr{C}'$. This is a general result, which can be stated as follows (cf. Apt and Emden, 1982; Lloyd, 1984):

> Let $\mathscr{C} = \mathscr{C}' \cup \{c_0\}$ be unsatisfiable, where $\mathscr{C}'$ contains only Horn clauses ordered with the positive literal at the head and c_0 is a negative clause:
>
> $$c_0 = \neg C(x_1, x_2, \ldots, x_n) = \neg L_1(x_1, x_2, \ldots, x_n) \vee \neg L_2(x_1 \ldots) \vee \ldots \vee \neg L_r(x_1 \ldots)$$
>
> Then every ordered input deduction that leads to the empty clause and involves the substitutions $(x_1 \mid t_1)$, $(x_2 \mid t_2)$, ..., $(x_n \mid t_n)$ defines objects of the Herbrand universe of $\mathscr{C}'$ such that:
>
> $$C(t_1, t_2, \ldots, t_n) = L_1(t_1, t_2, \ldots, t_n) \wedge L_2(t_1, t_2, \ldots, t_n) \wedge \ldots \wedge L_r(t_1, t_2, \ldots, t_n)$$
>
> is a consequence of $\mathscr{C}'$.

Conversely, if $(t_1, t_2, \ldots, t_n)$ is any n-tuplet in the Herbrand universe of $\mathscr{C}'$ such that $C(t_1, t_2, \ldots, t_n)$ is a consequence of $\mathscr{C}'$, there is an ordered input deduction of the empty clause with root $\neg C(x_1, x_2, \ldots, x_n)$ and involving substitutions $(x_i \mid t_i')$ such that $C(t_1, t_2, \ldots, t_n)$ is an *instantiation* of $(C(t_1', t_2', \ldots, t_n')$, i.e. is derived from this by replacing certain variables by terms.

It follows from this result that if we are looking for those objects t for which q(t) is a consequence of $\mathscr{C}'$— or, what is equivalent, such that q(t) is true for every Herbrand model of $\mathscr{C}'$—it is sufficient to search the ordered input deduction tree by breadth first, or depth first with backtracking if we can be certain that the tree is finite, and to note the object t associated with each empty clause as this is found.

This is the principle on which are based the methods used for extracting responses in many automated proof systems, and in some database query languages. In particular, it is used in PROLOG.

Exercises

Resolution without Variables (Ground Resolution)

1. Show, by giving a deduction of the empty clause, that the following set of formulae is unsatisfiable:

$\mathscr{C} = \{A \vee \neg B \vee C, \neg A \vee C, \neg E, A \vee B \vee E, \neg C \vee E\}$

2. Using Proposition 1, show that the following set is satisfiable:

$\mathscr{C} = \{A \vee B \vee C, A \vee \neg B \vee C \vee D, \neg A \vee B \vee D, A \vee \neg B \vee \neg C \vee D\}$

and give an interpretation that satisfies $\mathscr{C}$.

Unification

Unify, where possible, the following sets of atoms:

1. $At_1 = p(x, f(x), g(f(x), x))$
   $At_2 = p(z, f(f(a)), g(f(g(a, z)), v))$
2. $At_1 = p(x', y', z')$
   $At_2 = p(f(g(x, y), g(v, w), y))$
   $At_3 = p(f(z), x, f(x))$
3. $At_1 = p(x, f(x), f(f(x)))$
   $At_2 = p(f(f(y)), y, f(y))$
4. $At_1 = p(x, y, z, t)$
   $At_2 = p(f(y), f(z), f(t), f(x))$
   $At_3 = p(g(z), g(x), g(y), g(z))$

Let α be a mapping $\{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, n\}$. For what categories of α are $p(x_1, x_2, \ldots, x_n)$ and $p(x_{\alpha(1)}, x_{\alpha(2)}, \ldots, x_{\alpha(n)})$ unifiable?

Let $At_1 = p(b, x_1, x_1)$
$Bt_1 = p(y_1, y_1, b)$
$At_2 = p(f(b, x_1, x_1), x_2, x_2)$
$Bt_2 = p(y_2, y_2, f(y_1, y_1, b))$
$At_3 = p(f(f(b, x_1, x_1), x_2, x_2), x_3, x_3)$
$Bt_3 = p(y_3, y_3, f(y_2, y_2, f(y_1, y_1, b)))$
etc.

1. Calculate the unifier of At_1 and Bt_1, of At_2 and Bt_2, of At_3 and Bt_3 etc.
2. Defining the length of an atom as the number of symbols that it comprises, excluding brackets and commas, so that the length of At_1 above is 4, show there exist two constants $k_1 > 0$, $k_2 > 1$ such that for all $n \geqslant 0$ there are a pair At, Bt of length $> n$ such that

$$\text{length(unifier of At, Bt)} > k_1 \times k_2^{\text{length}(A_t)+\text{length}(B_t)}$$

Without using the unification algorithm, show that the following pairs are not unifiable:

1. p(x, f(x)) and p(f(y), y)
2. p(x, y, z) and p(f(z), x, y)

Resolution with Variables (General Resolution)

Having first made suitable transformations as explained in Chapter 6, use Proposition 2 to show in each of the following cases that the formula A is a consequence of the accompanying sets of formulae.

1. $A = \forall u\ q(u)$
   $A_1 = \forall x \exists y\ p(x, y)$
   $A_2 = \forall z_1 \forall z_2\ (p(z_1, z_2) \rightarrow q(z_1))$
2. $A = \forall x\ (p(x) \rightarrow p(f(f(x))))$
   $A_1 = \forall x\ (p(x) \rightarrow r(f(x)))$
   $A_2 = \forall x\ (r(x) \rightarrow p(f(x)))$
3. $A = \exists z\ q(z)$
   $A_1 = \exists y \forall x\ p(x, y)$
   $A_2 = (\forall x \exists y\ p(x, y) \rightarrow \forall z\ q(z))$
4. $A = \exists x\ s(x)$
   $A_1 = \exists x \forall y\ (p(x, y)\ v\ p(y, x))$
   $A_2 = \forall x\ p(x, x) \rightarrow (q(x)\ v\ r(x)))$
   $A_3 = \forall z\ (q(z) \rightarrow s(z))$
   $A_4 = \forall u\ (r(u) \rightarrow q(u))$
5. $A = p(s(s(s(s(a)))))$
   $A_1 = p(a)$
   $A_2 = \forall x\ (p(x) \rightarrow q(s(x)))$
   $A_3 = \forall x\ (q(x) \rightarrow p(s(x)))$

   Give an interpretation of a, p, q that will explain the meaning of this. Show that $A' = \forall x\ p(x)$ does not result from $\{A_1, A_2, A_3\}$.

6. $A = p(b, c, d, a)$
   $A_1 = \forall x \forall y \forall z \forall,\ t(p(x, y, z, t)) \rightarrow p(y, x, z, t)))$
   $A_2 = \forall x \forall y \forall z \forall,\ t(p(x, y, z, t)) \rightarrow p(z, y, x, t)))$
   $A_3 = \forall x \forall y \forall z \forall,\ t(p(x, y, z, t)) \rightarrow p((t, y, z, x))$
   $A_4 = \forall x \forall y \forall z \forall,\ t(p(x, y, z, t)) \rightarrow p((t, x, y, z))$
   $A_5 = p(a, b, c, d)$

Resolution Applied to a Specific Model

Given the following meanings for the constants and predicates—
constants:
b : a book
i : a publisher (issuer)
m: Mr Dupont
predicates:
rb(x, y) : x receives the book y
ac(x, y) : x has a bank account that is in credit
gc (x, y) : x gives a cheque to y
isb (z, y, x): publisher z sends book y to x

wb (x, y) : x would like to have book y
bai (y, z) : book y is available from publisher z

1. interpret the axioms
   a1: ac(m)
   a2: bai (b, i)
   a3: wb(m, b)
   a4: isb(z, y, x) → rb(x, y)
   a5: (gc(x, z) ∧ bai(y, z)) → isb(z, y, x)
   a6: (wb(x, y) ∧ ac(x)) → gc(x, z)
2. show, by resolution, that rb(m, b) is a consequence of the axioms; translate each step in the resolution into ordinary English, using the above meanings, so as to present the process as one of natural reasoning.

Horn clauses

Because of the form of its first member, the set:

$$S = \{p(x) \vee q(x) \vee r(x), \neg p(x) \vee \neg q(x) \vee r(x), \neg p(x) \vee q(x) \vee \neg r(x)\}$$

is not a set of Horn clauses, but if we write q', r' for $\neg q$, $\neg r$, respectively this becomes:

$$S = \{p(x) \vee \neg q'(x) \vee \neg r'(x), \neg p(x) \vee q'(x) \vee \neg r'(x), \neg p(x) \vee \neg q'(x) \vee r'(x)\}$$

which is now a set of Horn clauses.

1. Show that it is not always possible to transform an arbitrary set into a set of Horn clauses by means of such a renaming of predicates.
2. Show that if S is a minimally unsatisfiable set of variable-free clauses, it is always possible to find a renaming of the predicates that will transform S into a set consisting of Horn clauses and negative clauses (Henschen and Wos, 1974).
3. Show that every set of Horn clauses is satisfiable.
4. Let $\mathscr{C}'$ be a set of variable-free Horn clauses and c_0 a variable-free negative clause such that $\mathscr{C} = \mathscr{C}' \cup \{c_0\}$ is unsatisfiable; show that there is an input deduction of the empty clause with root c_0.
5. Let i_1 and i_2 be two models of a set $\mathscr{C}'$ of variable-free Horn clauses; then i_3 defined by:
   $i_3(A) = \mathbf{T}$ if and only if ($i_1(A) = \mathbf{T}$ and $i_2(A) = \mathbf{T}$)
   is also a model of $\mathscr{C}'$: that is, that the intersection of two models of a set of Horn clauses is also a model.
   Is this true for other than Horn clauses?

Ordered Input Resolution

Draw the ordered input resolution tree for:

$$\mathscr{C}' = \{r(x) \vee \neg r(f(x)),\ r(x) \vee \neg r(g(x)),\ r(f(g(f(a))))\}$$

with root:

$$c_0 = \neg r(a)$$

How many nodes does this have at depth n?

What is given by (a) depth first with backtracking, (b) breadth first search strategies?