Deduction revision by intelligent backtracking

M. Bruynooghe, Katholieke Universiteit Leuven, and
L. M. Pereira, Universidade Nova de Lisboa

1 INTRODUCTION

The use of traditional backtracking to explore a search space top-down starts with the initial state as the current state. Then, for each forward derivation step, one of the operators applicable to the current state is used to derive a new current state. This forward execution is repeated until either a solution state is reached and success is reported, or the set of unused operators applicable to the current state is empty. At this point, the search backtracks. The current state is dropped, its predecessor is reinstated as the current state, and forward execution recommences. If backtracking beyond the initial state is required, failure to find any more solutions is reported.

This approach does not exploit the relationships among successive states. After reaching a failed state, the system simply returns to the previous state. Sometimes, however, doing so cannot prevent the repetition of the same failure. A very inefficient thrashing behaviour can result, where the system performs an exhaustive search over a subspace which is irrelevant to the failure.

In this paper, we substantially improve the search behaviour for the case of sequential or parallel top-down executions of Horn clause logic programs. (We briefly indicate the extension to general theorem proving.)

To obtain this improvement, we observe that each derivation step extends a state into a new state. A state can be considered as a set of derivation steps. Moreover, a partial order over the derivation steps is obtained, because each new extension is only dependent on a subset of the existing ones. On failure the 'suspects' are determined: those derivation steps on which the failed extension depends which are responsible for the failure. One of them is selected as the culprit, and that derivation step as well as any derivation steps dependent on it are undone. All derivation steps not dependent on the undone ones are kept.

This paper is based on previous work of both authors (Bruynooghe, 1980; 1981; 1981a; Bruynooghe and Pereira, 1981; Pereira, 1979, Pereira and Porto, 1979; 1979a; 1980; 1980a; 1980b; 1982). Our method provides a top-down form of truth maintenance applied to resolution theorem proving, which complements bottom-up truth maintenance (Doyle, 1979; 1980). It is concerned with the backward rather than the forward component of the dynamics of logic

control, and it relies on purely syntactic information. Consequently, it is domain independent.

In the second section we briefly introduce logic programs; the third section describes a theory of intelligent backtracking; the next section shows the specialisation of the theory to the case of depth-first search and argues for the practicality of our approach; we terminate with some examples, efficiency results, and draw some conclusions.

2 LOGIC PROGRAMS

A logic program comprises a set of procedures ('Horn clauses') and a goal statement. The goal statement consists of a set of procedure calls A_i. It is written $\longleftarrow A_1, \ldots, A_n$ $(n \geqslant 1)$. The goals A_i have the form $R(t_1, \ldots, t_n)$ ('negative literals') where R stands for an n-adic relation and the t_i for terms. Terms can be distinguished into constants (first symbol an upper-case letter), variables (first symbol a lower-case letter) or compound terms of the form $f(t_1, \ldots, t_m)$ with f an m-ary function name and the t_i again terms. With $x_1, \ldots, x_m$ the variables occurring in the above goal statement, it reads 'find values for $x_1, \ldots, x_m$ which solve problems A_1 and ... and A_n'. The goal statement is the initial state of execution. It can be represented by an AND-tree where the goals A_i are the successors of the root node.

A procedure has the form $B \longleftarrow A_1, \ldots, A_n$ $(n >= 0)$ with B a positive and the A_i negative literals. B is the heading and the goals A_i form the body of the procedure. A procedure reads 'to solve problem B, solve the problems A_1 and ... and A_n'.

To perform a derivation step on a goal statement $\longleftarrow A_1, \ldots, A_n$, a goal $A_i = R(t_1, \ldots, t_p)$ is selected. A procedure for the relation R is chosen and its variables are renamed to become unique. (We consider the procedure variables as the local ones, those of the goal statement as the global ones.) The procedure ('operator') is applicable when the goal $R(t_1, \ldots, t_p)$ and the heading $R(s_1, \ldots, s_p)$ have a most general unifier θ which matches them (Robinson, 1979). Then a new goal statement $\longleftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n)\, \theta$ is derived, where the B_j form the body of the chosen procedure. This derivation step is but the result of applying the resolution principle (Robinson, 1979).

In the AND-tree 'proof tree', the terminal node containing the goal A_i is selected, the node is labelled with the substitution θ and becomes a non-terminal node with the goals B_j as successors. A simple successor labelled [ ] is generated if the body is empty. The goal statement corresponding to an AND-tree is obtained by applying all substitutions on the conjunction of all non-[ ] terminal nodes.

Whenever different procedures match the same goal, the search faces different alternatives (OR-branches). Usually, depth-first search combined with backtracking is applied to explore the search space.

3 INTELLIGENT BACKTRACKING – A THEORY

3.1 Current Approaches in Theorem Proving

Intelligent backtracking is based on analysis of the conflicts which arise during the unification process. Currently, two general approaches have appeared in the literature. An open problem is how more domain-specific information can be used within these theories to make still better backtracking choices (cf. McCarthy, 1982).

One of them is introduced by Cox (1977), in collaboration with Pietrzykowski (1981) and has been further developed by Matwin and Pietrzykowski (1982; Pietrzykowski and Matwin, 1982) while Cox (1981) also continued his investigations. The other approach started with the work of Bruynooghe (1981) and has been further developed (independently) by Bruynooghe (1980; 1981a) and by Pereira (with collaboration from Porto) (Pereira, 1979; Pereira and Porto, 1979; 1979a; 1980; 1980a; 1980b; 1982).

This paper is an attempt to define a common more sophisticated theory which encompasses the previous work of Bruynooghe and Pereira. A first version will be found in Bruynooghe and Pereira (1981).

The basic principles of the Cox–Pietrzykowski- Matwin approach and the Bruynooghe– Pereira approach are different but do not seem incompatible. They are in fact equivalent (confirmed by Cox in a personal letter, July 1982).

Cox, Pietrzykowski and Matwin base their method on 'deduction plans'. Restricting ourselves to Horn clauses, a deduction plan ('deduction tree' is a more appropriate term for the restricted case) looks like a skeletal proof tree, a proof tree with all substitutions completely ignored. (And with all necessary renaming of variables performed.) Because deduction trees are also well suited to explain our approach, we illustrate them in a simple example, see Fig. 1.

S(B) <--
R(u.u) <--

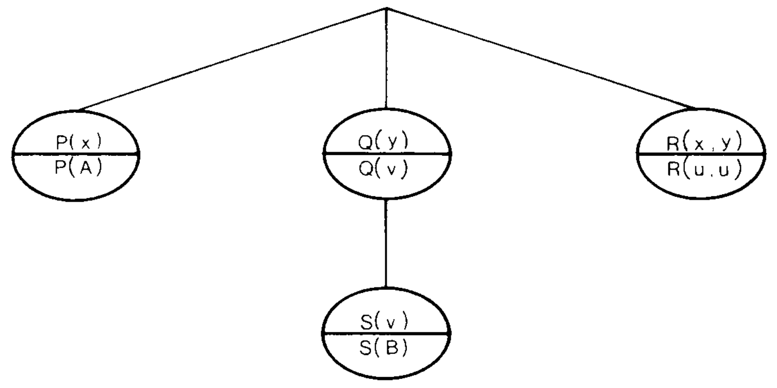


Fig. 1 – A deduction tree for Example 1.

Our representation of proof trees is inspired by the notation of Ferguson described by van Emden (1982).

Example 1

$\leftarrow$ P(x), Q(y), R(x,y)
P(A) $\leftarrow$
Q(v) $\leftarrow$ S(v)
S(B) $\leftarrow$
R(u,u) $\leftarrow$

The unifications to be performed to establish this deduction tree as a real proof tree are those between the following terms:

- P(x) and P(A)
- Q(y) and Q(v)
- S(v) and S(B)
- R(x,y) and R(u,u)

Deduction trees are so useful to explain intelligent backtracking because each deduction represents a whole set of derivations; not only the derivation in the usual (for Prolog) depth-first, left-to-right order P, Q, S, R, but also any other order allowed by the structure of the tree, e.g. Q, P, S, R. Our method is suitable for any order, which makes it appropriate for parallel processing.

As the reader can verify, the unification will fail on the above set of terms.

The Cox–Pietrzykowski–Matwin approach of analysing a failure is based on finding *maximal subtrees such that unification is possible* ('maximal consistent deduction trees'). (Each of them is obtained by removing a derivation step from a 'reduced conflict set' (Matwin and Pietrzykowski, 1982). For the above example, those trees are:

- the tree consisting of the nodes P, Q and S
- the tree consisting of the nodes Q, S and R
- the tree consisting of the nodes P, Q and R.

Each of these deduction trees serves as a starting point for a continuation of the search process. They can be handled in parallel; however, to our understanding, the search spaces have overlapping parts (confirmed by Matwin in a personal letter, November, 1982).

Our approach of analysing a failure is based on the complementary idea of finding *minimal subtrees such that unification is impossible* ('minimal inconsistent deduction trees'). Such a deduction tree is used to carefully remove a part of the deduction tree, such that a cause of the failure is removed and the serial search remains complete and nonredundant.

For the above example, the whole deduction tree is the single minimal inconsistent one.

The above description suggests that both approaches complement each other and are not fundamentally different. However, a more thorough comparison of both approaches is outside the scope of this paper.

Our method can also be interpreted as a reformulation, in a top-down fashion and applied to theorem proving, of Doyle's bottom-up truth maintenance system (Doyle, 1979; 1980).

3.2 How to Learn From Failures

We start with some terminology:

- A *closed* deduction tree: a deduction tree which represents a potential solution (e.g. Fig. 1): each call (upper half circle) is closed with the heading of a procedure (lower half circle).
- An *open* deduction tree: at least one call (upper half circle) is not closed with the heading of a procedure (lower half circle).
- If all proposed unifications are possible, then a deduction tree is *consistent*, it corresponds to a proof tree; otherwise, it is *inconsistent*.
- An open deduction tree can be *extended* by closing some of the open nodes. A deduction tree is a subtree of all its *extensions*.
- An open deduction tree is *unsolvable* if it is consistent but all its closed extensions are inconsistent.
- A deduction tree *fails* when either it is inconsistent or unsolvable.

In this section we study how to proceed after detecting an inconsistent or unsolvable deduction tree; the next section is devoted to adapting the unification algorithm so that it generates inconsistent deduction trees.

The largest possible search space consists of all possible closed deduction trees. It is infinite for recursive programs. A Prolog interpreter using naive backtracking does not consider such a large search space. It builds the deduction tree step by step; after each step, it verifies whether unification is still possible. If not, it knows that the tree fails and it never considers an extension of a failing tree.

Example 2

⟵ P(x), Q(x,y), R(y,z)		
P(A) ⟵	Q(B,D) ⟵	R(E,F) ⟵
P(B) ⟵	Q(C,D) ⟵	R(A,B) ⟵

The deduction tree shown in Fig. 2 is inconsistent and fails. The Prolog interpreter backtracks: it takes another procedure to close the call Q(x,y). It will never consider the deduction tree shown in Fig. 3.

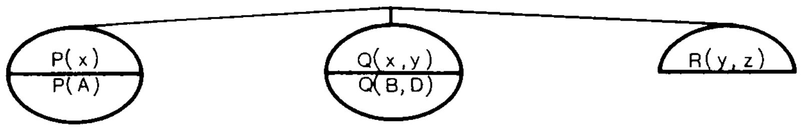


Fig. 2 – Open deduction for the problem of Example 2; it is a failing tree.

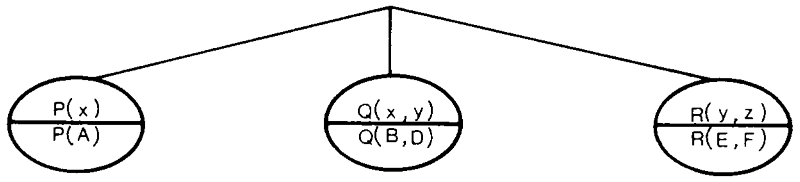


Fig. 3 – A closed deduction tree, an extension of the tree of Fig. 2.

To obtain this straightforward pruning of the search tree space, the naive interpreter enforces a total order among the nodes of the tree and keeps a list of available procedures for each node.

We can formalise this behaviour. Having a failing tree with nodes $Q_1, \ldots, Q_n$ on which the procedures $P_1, \ldots, P_n$ are applied, we can assert: 'If the procedures $P_1, \ldots, P_{n-1}$ are used to solve respectively the goals $Q_1, \ldots, Q_{n-1}$, then P_n is rejected as a solution for Q_n'.

The naive interpreter maintains this assertion without effort; it rejects P_n and tries the next available procedure to solve Q_n. Once it is forced to backtrack to Q_{n-1}, the premise becomes invalid. This is reflected by the fact that all procedures to solve Q_n become again available (if Q_n is still in the tree). Moreover, the organisation of the search is such that the premise will never become true again.

In a given failure state, the naive interpreter always considers the *whole* tree as a failing tree, the principal idea behind our intelligent backtracking is the ability to detect failing *subtrees*. This allows us to derive stronger assertions which give rise to a more substantial pruning of the search space.

Example 3

⟵ P(x), Q(y,y)	
P(A) ⟵	Q(A,B) ⟵
P(B) ⟵	Q(C,D) ⟵

Fig. 4 shows a failing deduction tree and an inconsistent subtree for the problem of Example 3. All trees which are extensions of the failing subtree are doomed to fail. A naive interpreter will consider such trees, e.g. using P(B) instead of P(A).

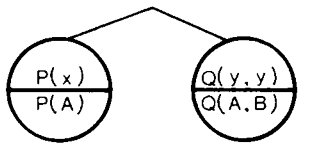


Fig. 4 (a) – A failing deduction tree.

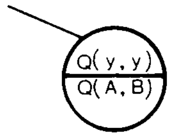


Fig. 4 (b) – A failing subtree.

This is a simple example of the famous thrashing behaviour which is at the root of the condemned inefficiency of naive backtracking. To improve the search behaviour, we must try to avoid extensions of inconsistent subtrees.

Example 4

⟵ P(x), Q(y)
a. P(A) ⟵
b. P(B) ⟵
c. Q(u) ⟵ R(u,u)
d. R(v,w) ⟵ S(v), T(w)
e. S(A) ⟵
f. S(B) ⟵
g. T(C) ⟵
h. T(D) ⟵

The deduction tree shown in Fig. 5 is inconsistent and fails. It has a failing subtree consisting of the nodes Q, R, S and T. We can make the assertion: 'If c is used to solve Q, d to solve R, e to solve S and g to solve T, then the problem has no solution' or 'Each deduction tree which is an extension of the tree using c to solve Q, d to solve R, e to solve S and g to solve T is a failing tree'.

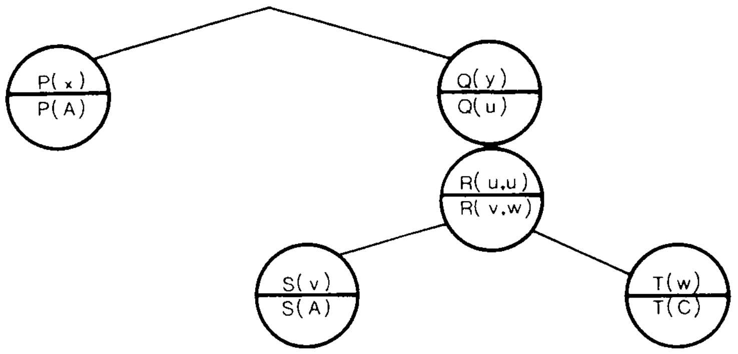


Fig. 5 – A failing deduction tree for the problem of Example 4.

Different approaches are possible in using the above knowledge to guide the search.

(1) Store the failing subtree in a database (cf. Doyle's 'no good' assertions (Doyle, 1979; 1980)).

Verifying whether a deduction tree is an extension of one of the failing trees in the database seems rather expensive. Especially for an inconsistent deduction tree, it seems that rediscovering the failure by performing the unification is more efficient than a search over the database. This is not necessarily the case for stored unsolvable deduction trees, because there we may have to consider a number of extensions (which can be arbitrarily large) before we detect the unsolvability of the tree. In the extreme case of an unsolvable problem we eventually obtain the empty tree as a failing tree. The cost of discovering this can be large, the cost of observing that a tree is an extension of the empty tree is small.

(2) We have chosen a more pragmatic solution, losing information from time to time, but resulting in less overhead. Our solution is related to the behaviour of the naive interpreter. There, on failure, we could state assertions of the form 'If we use the currently applied procedures to solve the subgoals $Q_1, \ldots, Q_{n-1}$ then the procedure p_n is rejected for solving Q_n'.

Now, for a failing subtree T_n, we state: 'If we use the currently applied procedures to solve the subgoals of T_n, then the problem is unsolvable'. To remove the cause of failure, we have to remove a node from the subtree T_n. To obtain a complete search, it has to be a leaf. As we do not require a fixed order among subgoals, it can be any leaf. The selected one is called the *culprit*. Suppose Q_n is selected as the culprit in a tree T_n and T_{n-1} is the tree obtained after removal of Q_n. Then, we can state the following *assertion:* 'If we use the currently applied procedures to solve the nodes of T_{n-1}, then p_n is rejected to solve Q_n'.

In Example 4, we can choose between S and T as culprits. Suppose we take S, then S(A) is rejected to solve the goal S(v) because of the currently applied procedures of Q, R and T. Extending again the node S(v), now using S(B), results in a new failure with the same inconsistent subtree. Again, S and T are the possible culprits. However, selecting another clause for T will invalidate the premise of the assertion obtained from the first failure and will return S(A) to the set of available procedures. To avoid this loss of information, we should again select S as the culprit.

This preferential treatment of failures extends the order among subgoals. In a failing subtree, we create order between the culprit and the other leaves. It means that in any future failure treatment, it will be selected in preference over all other nodes of the currently failing tree. With this approach, we have a loss of information each time the premise of an assertion becomes invalid, i.e. when a rejected procedure is part of some premises.

This loss of information could be prevented by turning the assertion into a 'no good' assertion *explicitly* naming the applied procedures. We have not explored this possibility. It requires consideration of (1) the cost of rediscovering it, (2) the cost of storing it and verifying whether an extension appears and (3) the probability that an extension appears.

Combining Individual Failures

Up to now, our failing subtrees were inconsistent subtrees supposedly discovered by the unification algorithm (see next section). Here we discuss the derivation of unsolvable subtrees.

It is possible that all procedures q_i available to solve a certain goal Q become rejected. Then, for each procedure q_i, we have an assertion 'If the currently applied procedures are used to solve the goals of tree T_i, then q_i is rejected to solve Q'. With T the subtree which is the union of all T_i (the set of nodes of T is the union of the sets of nodes of the T_i), we can combine these assertions into: 'If the currently applied procedures are used to solve the goals of tree T, then $q_1, \ldots, q_n$ are rejected to solve Q'; in other words, because all closed extensions of T have to, but cannot, contain a solution of Q, T is an unsolvable subtree, thus a failing subtree.

In Example 4, S(B) is also rejected because of the tree with nodes Q, R and T. This is the same tree responsible for the rejection of S(A). We conclude that this tree is unsolvable. T is its only leaf, thus it is the culprit. We obtain that T(C) is rejected because of the currently applied procedures on Q and R. We lose the assertions about S(A) and S(B); however, observe that the new assertion is stronger than the lost ones (this is not always the case).

The search takes a new start with S(A) and S(B) available to solve S(v) and T(D) to solve T(w).

When the reader continues the example, he will find an unsolvable subtree consisting of Q and R, then an unsolvable subtree with a single node Q and finally the empty subtree as unsolvable one, indicating the unsolvability of the problem. All of this without ever backtracking to the 'first' subgoal P(x).

Required Bookkeeping

Perhaps the reader wonders whether our assertions are more manageable than the optimal 'no good' assertions. Here, we briefly discuss the required structures to perform our intelligent backtracking.

With each call, we have associated

- a set of (pointers to) available procedures
- a (pointer to a) currently applied procedure
- a set of (pointers to) rejected procedures plus, for each rejected procedure, a representation of the assertion (the subtree which forms the premise of our assertion).

To represent a subtree of a deduction tree, we observe that it is uniquely determined by its leaves. We can take as representation a collection of nodes containing *at least* all leaves. This makes the operation of taking the union of two subtrees very simple.

To propagate the effects of rejecting a currently applied procedure, we need pointers to the assertions containing that procedure in their premise: another list of pointers associated with each call. A procedure returns from the set of rejected procedures to the set of available procedures when all its associated assertions are removed. (These lists of pointers also represent the partial order created by the selection of culprits.) A rejected procedure can be associated with *more than one* assertion. Indeed, an inconsistent tree can have different failing subtrees (it is also possible to extend a tree with more than one step). Each subtree represents a conflict which has to be solved by selecting a culprit. It is possible that the same call is selected as the culprit for two different conflicts.

Because a rejected procedure can have different assertions, we can derive different unsolvable subtrees for a failing tree by combining one assertion from each rejected procedure.

3.3 How To Obtain Inconsistent Deduction Trees

The naive interpreter always considers the whole deduction tree as the only inconsistent deduction tree. All our machinery is useless if we cannot obtain smaller inconsistent deduction trees. That is the goal of this section. The smaller the inconsistent deduction trees obtained, the better the behaviour.

To solve the problem of detecting *all* possible *minimal* inconsistent deduction trees, we probably have to use the latest research results of Cox (1981).

Having no idea of the computational complexity of that approach, we content ourselves with a more pragmatic solution, which is closer to conventional Prolog implementations. Each deduction tree is an extension of the previous one. Unification is incremental; it tries to establish the new part using the old part as context. We describe a unification algorithm which generates, besides the usual substitutions on successful completion, inconsistent deduction trees on failure.

To obtain this behaviour, we associate a deduction tree with each term involved in the unification. With a call, we associate the nodes necessary for the existence of that call, i.e. those on the path from the root to the call (including the call). With the heading of the applied procedure, we associate the empty deduction tree (the procedure is given). (In the case of general theorem proving, we use the minimal deduction plans resulting in the creation of both clauses involved in the resolution step.) As a result, each substitution has an associated deduction tree which is the minimal deduction tree necessary for obtaining that substitution. Detection of a unification failure results in such a tree being an inconsistent deduction tree. When the unification algorithm consults a substitution component, it requires the existence of the deduction tree associated with it. This provides a method for obtaining the deduction trees associated with terms.

The formal representation of the algorithm is as follows (where t–T denotes a term t with associated deduction tree T):

(1) matching $t–T_1$ with $t–T_2$: generate ϵ (the empty substitution) with deduction tree $T_1 \cup T_2$
(2) matching $f(t_1, \ldots, t_m)–T_1$ with $g(t_1, \ldots, t_n)–T_2$: generate 'failure' with $T_1 \cup T_2$ as an inconsistent deduction tree
(3) matching $f(t_1, \ldots, t_n)–T_1$ with $f(r_1, \ldots, r_n)–T_2$ ($n > 0$ and some $t_i =/= r_i$): match each $t_i–T_1$ with $r_i–T_2$
(4) matching $x_1–T_1$ with $t_2–T_2$ where t_2 is not a free variable and a component $x_1 \longleftarrow t$ exists with deduction tree T: match $t–T_1 \cup T$ with $t_2–T_2$
(5) matching $x_1–T_1$ with $t_2–T_2$ with x_1 a free variable: generate the component $x_1 \longleftarrow t_2$ with $T_1 \cup T_2$ as deduction tree.

It can be useful to go on with unification after generating a first failure: other failures resulting in different inconsistent deduction trees can be derived. After completing a failing unification, the minimal inconsistent deduction trees are retained while all generated substitutions are removed.

Example 5

Unifying a term f(A, A)–0 with a term f(x,y)–{Q, S} (where we represent a deduction tree by the set of its nodes and 0 denotes the empty set) with existing substitutions $x \longleftarrow B–\{Q, R\}$ and $y \longleftarrow C–\{Q\}$, we obtain:

- failure with deduction tree {Q, R, S}.
- failure with deduction tree {Q, S}.

Only the second one is minimal; indeed, the first is an extension of the second.

Example 6

$\longleftarrow$ P(x), Q(x)
P(A) $\longleftarrow$
Q(B) $\longleftarrow$

A first step unifies P(x)–{P} with P(A)–0, resulting in $x \longleftarrow A–\{P\}$. A second step unifies Q(x)–{Q} with Q(B)–0; it attempts to unify A–{Q, P} with B–0 and results in failure– {Q, P}. Notice the symmetry: both P and Q are candidate culprits; the inconsistent deduction tree is independent of the selection order between P and Q.

Keeping the deduction trees as small as possible involves some pragmatics:

- which variable to bind when unifying two free variables can only be settled later on, when one of them is bound to a term (see Pereira and Porto (1982) for a detailed discussion),
- trying to avoid dereferencing when it enlarges the deduction tree: first dereference variables whose final value is a free variable.

Undoing a procedure call S affects all executed calls having S in the deduction tree of a generated substitution; they become invalid and have to be redone. To avoid this expensive operation as much as possible, we have to consider unification as a process which also creates a partial order to be respected when selecting a culprit: P comes after S when S is part of a deduction tree of a substitution component of P.

To reduce the space requirements, Prolog interpreters try to collapse different nodes of the proof tree into one. Intelligent backtracking, to be optimal, limits this possibility to the case of 'strong determinism': whenever all but one of the available procedures is rejected due to the deduction tree consisting of the path from the root node to the parent of the strong determinisitic node. To show that such a node can be collapsed with its father, without affecting the search behaviour, suppose a node P with son Q. Suppose T is the tree consisting of the path from P to the root. Suppose $q_1, \ldots, q_{n-1}$ are rejected due to T and q_n is the only remaining procedure. When, at a later point a failing deduction tree T_1 is obtained and Q is chosen as the culprit, then q_n is rejected due to $T_2 = T_1 - Q$. Because T_1 contains Q, T_2 contains all nodes from Q to the root: it is an extension of T. Q fails and $T_2 \cup T = T_2$ is the new failing deduction tree. If Q is considered as part of node P, then we directly obtain $T_1 - Q = T_2$ as a failing deduction tree. Thus it is useless to consider strong deterministic calls as independent nodes of the deduction tree.

3.4 Examples

(1) A goal statement ⟵ P(x), Q(y), R(x,y)

Step 1. Execution of P(x) with a procedure P(A) ⟵. Unification succeeds with x ⟵ A–{P}.

Step 2. Execution of Q(y) with a procedure Q(B) ⟵. Unification succeeds with y ⟵ B–{Q}.

Step 3. Execution of R(x,y) with a procedure R(C, D). Unification fails with 'failure'–{P, R} and 'failure'–{Q, R}; selection of R as culprit results in rejection of R(C, D) due to {P} and due to {Q}.

Step 4. Execution of R(x,y) with a procedure R(A, E). Unification fails with 'failure'–{Q, R}. We have to select R as culprit; R(A, E) is rejected due to {Q}.

Step 5. Exhaustion of available procedures for R(x,y). Computation of failing subtrees:

$$\{P\} \cup \{Q\} = \{P, Q\}$$
$$\{Q\} \cup \{Q\} = \{Q\}$$

The former being an extension of the latter, the latter is the only minimal one. Q(B) is rejected, due to { }, the empty subtree, meaning Q(B) cannot contribute to a solution. R(A, E) returns to the set of available solutions, R(C, D) is still rejected because of the remaining assertion {P}.

(2) As a more elaborate example, we show a solution to the queens problem (for illustrative purposes limited to 3 queens). A configuration of queens is represented by a list of numbers, each number representing the position of a queen. The number is the row number of the queen, the column number is the position in the list, e.g. 3.1.2.Nil represents the queens on positions (3, 1), (1, 2) and (2, 3). Our solution uses the following procedures:

- Perm (x, y): the list y is a permutation of the list x.
- Del (x,y,z): the list z is obtained by removing an element x from the list y.
- Safe (s): the list represents a safe configuration of queens.
- S (s,l): the reverse of the list l represents a configuration on columns 1 to i, s represents a configuration on columns i+1 to n, where the queens on s do not attack each other and do not attack the queens on l (used to obtain an optimal ordering among the calls of Nodiag).
- Check (p,l,d) the reverse of l is a partial configuration (columns 1 to i). The distance between queen p and the last queen of that configuration is d (p on column i+d), p does not attack the queens of the configuration.
- Nodiag (p,q,d): d is the distance between p (the column i+d) and q (the column i), and p does not attack q. This relation is defined by its extension, as a database of facts.

The program:

```
   ⟵ Perm (1.2.3.Nil, s), Safe (s)
a. Perm (Nil, Nil) ⟵
b. Perm (x.y, u.v) ⟵ Del (u, x.y, w), Perm (w, v)
c. Del (x, x.y, y) ⟵
d. Del (u, x.y, x.v) ⟵ Del (u, y, v)
e. Safe (s) ⟵ S (s, Nil)
f. S (Nil, l) ⟵
g. S (p.r, l) ⟵ Check (p, l, 1), S (r, p.l)
h. Check (p, Nil, d) ⟵
i. Check (p, q.l, d) ⟵ sd = d+1, Check (p, l, sd), Nodiag (p, q, d)
```

The state of the computation at the point where the first failure occurs is given in Fig. 6. We have labelled all nodes. The root, which represents the empty subtree, has the label (0); strongly deterministic nodes have the same label as their parents; a deduction tree is represented by the labels of its leaves. The selection of calls is as in Prolog: depth-first left to right. Each node contains the list of available procedures, the applied procedure and the

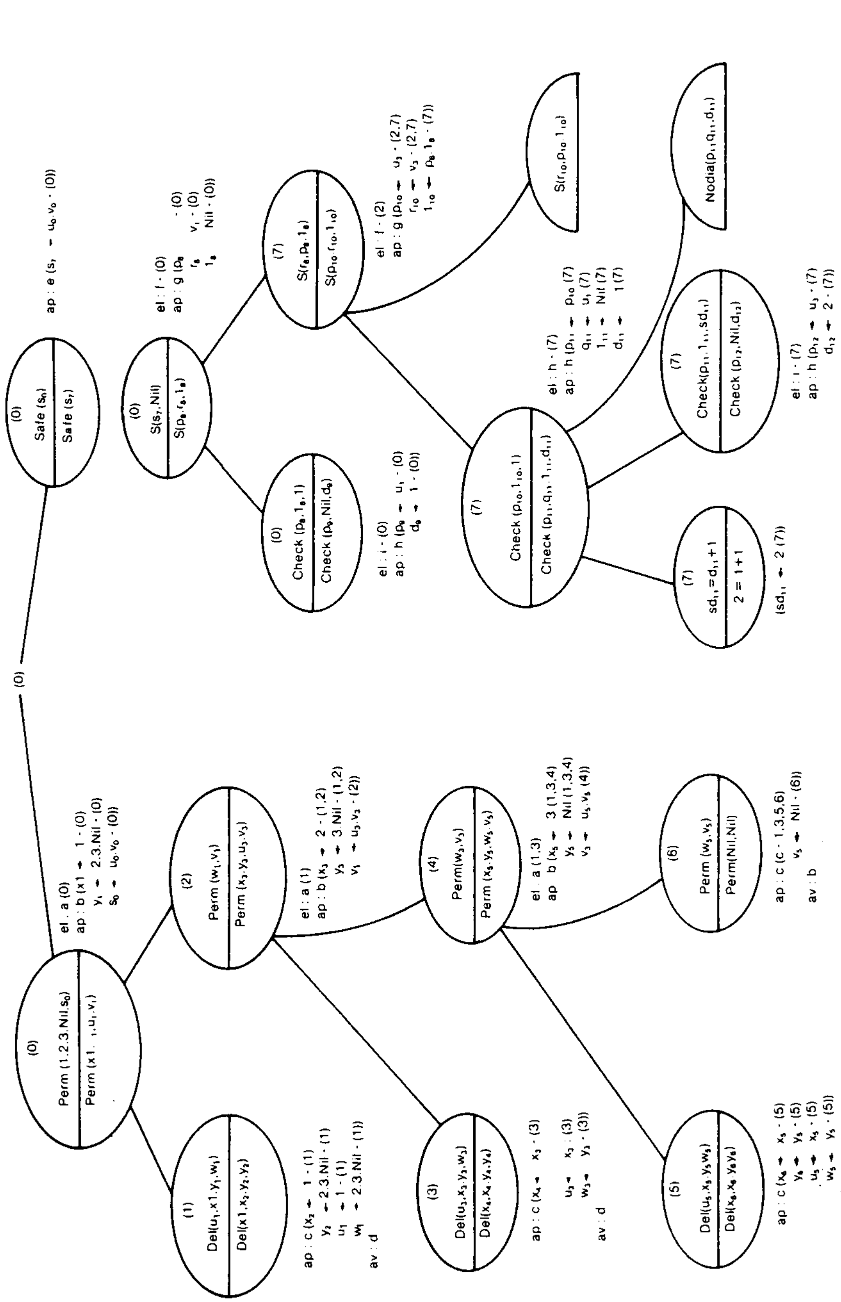


Fig. 6 – Queens computation at the first failure.

list of eliminated procedures with the deduction tree causing the elimination. The arguments of Nodiag are:

p_{11} = 2 (accessing the tree with leaves 7,1,3)
q_{11} = 1 (accessing the tree with leaves 7,1)
d_{11} = 1 (accessing the tree with leaves 7)

The call fails; the inconsistent deduction tree has as leaves 7, 1 and 3. Selecting (7) as the culprit results in the elimination of g due to {1,3}. Both f and g are eliminated, and we obtain an unsolvable deduction tree which is the union of {1,3} and {2}, that is {1,3}. Now we select (3) as the culprit; c is eliminated due to {2,1}. This means that nodes (5) and (6) completely disappear, the execution of (3) has to be undone (c is eliminated, d is available). In node (4) (a) returns to the set of available procedures and the substitutions obtained by applying procedure (b) are invalidated because they depend on (3), and they have to be redone. Normally, node (7) is undone by the first backtracking; actually, it could be saved because it does not depend on (3).

The behaviour is as desired. On detecting a conflict, the program backtracks to the point where one of the offending queens has been generated; optionally, it retains substantial parts of the Safe computation.

4 PRACTICAL EXPERIENCE

To obtain some insight into the applicability of intelligent backtracking to Prolog interpreters, we have conducted a few experiments. We expect that the overhead of our full theory is too large for the sequential execution of the majority of logic programs, and that it is only advantageous in specialised applications with a flavour of theorem proving, or for administrating backtracking in AND-parallelism. Our main interest being in normal programs, we have developed interpreters for a simplified version of the theory. One of the authors, with the help of Antonio Porto, started with a simulation: an interpreter with intelligent backtracking written in Prolog itself (Pereira, 1979, Pereira and Porto, 1979; 1979a; 1980a; 1980b; 1982). Encouraged by the results, Chris Coudron, an undergratuate student of the other author, adapted an existing Prolog interpreter (in the language C) to include a simplified form of intelligent backtracking and conducted some experiments. Given the limited amount of time, the main goal was to obtain a working system, not a very efficient one.

In this section, we discuss the simplifications, sketch the low-level implementation and show some results.

4.1 Simplifications

The selection order of calls in Prolog is controlled by the programmer. Sticking to it means that calls are executed in a strict left to right order, and that we backtrack to the most recent of candidate culprits and undo calls in a strict right to left order. The only difference with the naive interpreter is that we backtrack to the culprit instead of to the most recent call having untried alternatives.

This decision resolves the most thorny and complex issues: we have neither to bother about the partial order imposed by culprit selection nor about the partial order imposed by execution of the unification algorithm (when it consults arbitrary substitutions). Also, we do not have to worry about empty substitutions.

What remains is the following:

- a deduction tree associated with each substitution (binding of a variable);
- for each call a set of available procedures, as in a standard naive interpreter;
- for each call a set of rejected procedures, each of them associated with one or more deduction trees representing a reason for their rejection (obtained either from the unification algorithm when detecting a failure or from the intelligent backtracking mechanism when it detects an unsolvable subtree and chooses that call as the culprit).

When we undo a call, we remove all information about it: the substitutions of the applied procedures and the reasons for the rejection of the rejected procedures. The cost of rediscovering this last piece of information can be substantial, so we drop it because it allows us to retain the simple stack structure of the interpreter.

Once all procedures applicable to a call are rejected, we construct unsolvable deduction trees by taking the union of the deduction trees of the rejected procedures. Because we never use the deduction trees of the rejected procedures, but only the unions, we can equally well incrementally build those unions. Instead of having a set of deduction trees for each rejected procedure, we only have such a set for the call as a whole. Actually, we simplify further (at the cost of accuracy) and retain only one deduction tree for each call. In the simulation, unification computes different failures. Each of these failures is merged with the already obtained deduction tree and the 'best' resulting deduction tree is retained. The best one is the one giving the deepest backtracking. Formally, a tree T with nodes (in left to right order) $t_1, \ldots, t_t$ is better than a tree S with nodes $s_1, \ldots, s_s$ if there exists a $n \geqslant 0$ such that for $0 \leqslant i \leqslant n-1$, $t_{t-i} = s_{s-i}$ and t_{t-n} is less recent than s_{s-n} (the last n nodes being identical). Due to lack of development time, the low-level implementation stops unification after detecting a first failure, and this results in the undesirable effect that the behaviour depends on the ordering of arguments. Note that this simplification never guarantees optimal behaviour, e.g. in the first example of Section 3.4.

To summarise, we end up with:

- a deduction tree associated with each bound variable (produced by unification), and
- a deduction tree associated with each executed call (incrementally built each time a procedure is rejected, and used to determine the culprit once all procedures are rejected).

4.2 Low-level Implementation

As already explained, any considered deduction tree is a subtree of the current proof tree and we can represent the deduction trees by sets containing at least

all leaves. The sets themselves we have represented by binary trees with the members of the set in the leaves (a sign bit is used to distinguish between leaves and non-leaves). Taking the union of two deduction trees is very fast and simple: the creation of a new binary tree with the left branch pointing to the first tree and the right branch pointing to the second tree. However, the representation is not minimal, some nodes can have multiple occurrences and the representation can contain non-leaf nodes.

To reduce the size of the deduction trees of substitution components, we observe that the deduction tree of any substitution component generated during execution of a node Q includes node Q; as a consequence, the presence of the ancestors of Q is unnecessary because they are implicit in Q; we can initiate unification with an empty deduction tree for the call. Moreover, the bindings of the variables of a call Q are in the environment of the parent of Q; looking up such a variable means including the deduction tree associated with the variable. This deduction tree contains at least (sometimes it is the only element) the parent of Q, but that parent need not be included because Q is already present. To avoid inclusion of the parent, the deduction tree of a substitution component of a node is represented by either the node (only one leaf), or by a binary tree with the node in the left branch and the other elements in the right branch. When looking up a substitution component of node P, the father of Q, we ignore P simply by taking the right branch of the binary tree representation.

The space overhead in representing the deduction trees of substitutions consists of:

- one extra field for each variable, and
- the space of the binary tree in cases where the deduction tree has more than one element in its representation.

We started with an interpreter using different space-saving techniques; i.e. the removal of completed determinate subtrees and tail recursion optimisation. We tried to retain these techniques. Afterwards, we considered this as a serious design error. Starting again, we would remove all of them and concentrate on detecting strong determinism, and only reduce the proof tree in the case of strong determinism. The approach we were following posed several problems:

- Nodes disappear. To retain completeness, we have to add their deduction tree to the deduction tree of their parent; as a consequence, that deduction tree is larger than necessary. When the parent at some point fails, the resulting unsolvable deduction tree is larger than necessary; the backtracking is not optimal.
- Nodes disappear, but they are still present in deduction tress. Searching all deduction trees and replacing them by their parents is too expensive. We developed a labelling mechanism. Each node of the proof tree has an extra label field. A table maps labels into nodes. When a node disappears, the table is adjusted such that the label is mapped into the parent node.

The label mechanism and its space overhead would be unnecessary had we restricted ourselves to strong determinism.

Each node of the proof tree has another extra field containing (a pointer to) the associated deduction tree.

Due to the lack of time, the unification algorithm has not been adapted to discover all failures and to select the strongest one.

We take a crude approach to all impure aspects of Prolog. We keep a marker (initialised at the root of the proof tree). Up to the marker, we backtrack intelligently, beyond the marker we fall back on the naive backtracking mechanism. On meeting an impure feature (such as the 'cut') the marker is set at the current node of the proof tree (also on finding a first solution). For the 'cut', we could probably have done better, by doing complete dereferencing of all arguments of the call and taking the resulting deduction tree as the reason for the rejection of the procedures eliminated by the 'cut'. For finding subsequent solutions, we could reject a pseudo-goal with a deduction tree obtained by completely dereferencing the arguments in the top goal.

4.3 Results (as obtained by C. Coudron)

The test programs:

(1) A complex query to a small database

```
Student (Robert, Prolog) ←
Student (John, Music) ←
Student (John, Prolog) ←
Student (John, Surf) ←
Student (Mary, Science) ←
Student (Mary, Art) ←
Student (Mary, Physics) ←

Professor (Luis, Prolog) ←
Professor (Luis, Surf) ←
Professor (Maurice, Prolog) ←
Professor (Eureka, Music) ←
Professor (Eureka, Art) ←
Professor (Eureka, Science) ←
Professor (Eureka, Physics) ←

Course (Prolog, Monday, Room1) ←
Course (Prolog, Friday, Room1) ←
Course (Surf, Sunday, Beach) ←
Course (Math, Tuesday, Room1) ←
Course (Math, Friday, Room2) ←
Course (Science, Thursday, Room1) ←
Course (Science, Friday, Room2) ←
Course (Art, Tuesday, Room1) ←
Course (Physics, Thursday, Room3) ←
Course (Physics, Saturday, Room2) ←

Noteq (a, b) ← Less_than (a, b)
Noteq (a, b) ← Less_than (b, a)
```

The query : ← Student(stud, course1), Course(course1, day1, room), Professor(prof, course1), Student(stud, course2), Course(course2, day2, room), Professor(prof, course2), Noteq(course1, course2).

(2) A simple solution for the queens problem (generate and test) but taking care that the tests are in order (1, 2), (1, 3), (2, 3), (1, 4), (2, 4), (3, 4) . . . .

```
Queens(l, config) ← Perm(1, p), Pair(l, p, config), Safe(Nil, config)

Perm(Nil, Nil) ←
Perm(x.y, u.v) ← Delete(u, x.y, w), Perm(w, v)

Delete(x, x.y, y) ←
Delete(u, x.y, x.v) ← Delete(u, y, v)

Pair(Nil, Nil, Nil) ←
Pair(x.y, u.v, p(x, u).w) ← Pair(y, v, w)

Safe(left, Nil) ←
Safe(left, q.r) ← Test(left, q), Safe(q.left, r)

Test(Nil, q)
Test(r.s, q) ← Test(s, q), Notondiagonal(r, q)

Notondiagonal(p(c1, r1), p(c2, r2)) ← c=c1–c2, r=r1–r2, Noteq(c, r),
                                      nr=r2–r1, Noteq(c, nr)
Noteq(a, b) ← Less_than(a, b)
Noteq(a, b) ← Less_than(b, a)
```

A query (for 5 queens, one solution only)

```
← Safe(1.2.3.4.5.Nil, config),!
```

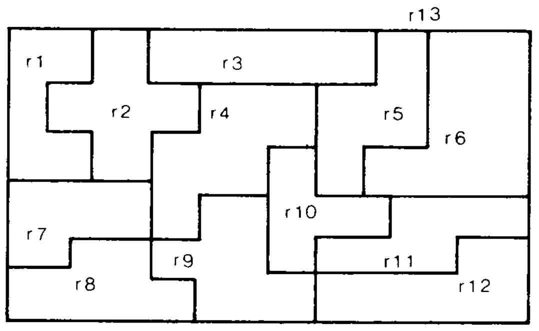
(3) A clever solution for the queens problem, merging the generate and test part, with a lot of 'cuts' (formulated for 5 queens)

```
Queens(config) ← Solution(c(0, Nil), config)

Solution(c(5, config), config) ← !
Solution(c(m, config), conf) ← Expand(c(m, config), c(m1, conf1)),
                               Solution(c(m1, conf1), config)

Expand(c(m, q), c(m1, p(m1, k).q)) ← m1=m+1, Column(k),
                                     Noattack(p(m1, k), q)

Column(1) ←
Column(2) ←
Column(3) ←
Column(4) ←
Column(5) ←

Noattack(p, Nil) ←
Noattack(p, q.1) ← Noattack(p, 1), Ok(p, q)

Ok(p(r1, c), p(r2, c)) ← !, Fail
Ok(p(r1,k1),p(r2,k2)) ←— difr=r2-r1, Abs(difr,abs), difc=k2-k1,
                                     Abs(difc,abs), !, Fail
Ok(p,q) ←—

Abs(n,n) ←— n>0, !
Abs(n,m) ←— m=0-n
```

The query : ←— Queens(config).

(4) A map colouring problem with a rather good order of colouring

```
Next(Blue, Yellow) ←—
Next(Blue, Red) ←—
Next(Blue, Green) ←—
Next(Yellow, Blue) ←—
Next(Yellow, Red) ←—
Next(Yellow, Green) ←—
Next(Red, Blue) ←—
Next(Red, Yellow) ←—
Next(Red, Green) ←—
Next(Green, Blue) ←—
Next(Green, Yellow) ←—
Next(Green, Red) ←—
```



```
Goal(r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13) ←—
    Next(r1,r13),   Next(r1,r2),    Next(r2,r13),   Next(r2,r4),
    Next(r4,r10),   Next(r6,r10),   Next(r8,r13),   Next(r6,r13),
    Next(r2,r3),    Next(r3,r4),    Next(r3,r13),   Next(r3,r5),
    Next(r5,r6),    Next(r5,r13),   Next(r4,r5),    Next(r5,r10),
    Next(r1,r7),    Next(r7,r13),   Next(r2,r7),    Next(r4,r7),
    Next(r7,r8),    Next(r4,r9),    Next(r9,r10),   Next(r8,r9),
    Next(r9,r13),   Next(r6,r11),   Next(r10,r11),  Next(r11,r13),
    Next(r9,r12),   Next(r11,r12),  Next(r12,r13).
```

(5) The same map colouring problem with a bad order of colouring

```
Goal(r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13) ←—
    Next(r1,r2),    Next(r2,r3),    Next(r3,r4),    Next(r4,r5),
    Next(r5,r6),    Next(r6,r11),   Next(r11,r12),  Next(r12,r13),
    Next(r9,r13),   Next(r9,r10),   Next(r4,r10),   Next(r4,r7),
    Next(r7,r8),    Next(r2,r7),    Next(r6,r10),   Next(r2,r13),
    Next(r6,r13),   Next(r2,r4),    Next(r2,r4),    Next(r4,r9),
    Next(r3,r5),    Next(r8,r9),    Next(r1,r13),   Next(r3,r13),
    Next(r5,r13),   Next(r7,r13),   Next(r11,r13),  Next(r9,r12),
    Next(r5,r10),   Next(r10,r11),  Next(r1,r7).
```

(6) A simple deterministic program to build an ordered binary tree

```
Tree(Nil, tree) ←—
Tree(e.1, tree) ←— Insert(e, tree), Tree(l, tree)
```

```
Insert(e,t(l,e,r)) ←— !
Insert(e,t(l,f,r)) ←— Less_than(e,f), !, Insert(e,l)
Insert(e,t(l,f,r)) ←— Insert(e,r)
```

Query:

```
←— Tree(46.11.48.46.47.6.5.9.7.5.14.17.14.22.1.32.61.14.
                                             56.11.78.Nil, tree)
```

The results, in seconds, are summarised in Table 1.

Table 1.

	Standard	Intelligent	Change
database query	0.72	0.58	−20%
6 queens, simple program	42.7	27.8	−36%
7 queens, simple program	40.8	7.05	−80%
8 queens, simple program	1015	232	−77%
6 queens, clever program	14.5	28.9	+99%
7 queens, clever program	4.12	8.4	+106%
8 queens, clever program	101.	221	+119%
map colouring, good order	0.52	0.85	+63%
map colouring, bad order	697	2.06	−99.7%
binary tree	1.28	1.84	+44%

Taking into account that it has been a first limited effort to implement intelligent backtracking, we are satisfied with the results. For the programs where the intelligent backtracking only creates overhead, the time increases by 44% to 119%, while for the other programs the results are substantially better.

5 CONCLUSIONS

We have provided a general conceptual framework to describe intelligent backtracking in resolution theorem provers.

We have explored the application of our approach in detail for top-down, depth-first execution of Horn clause logic programs.

Finally, in a pragmatic effort, we have written a Prolog interpreter which uses a simplified version. Experimental results show that implementation of intelligent backtracking at a low level is worthwhile. Such an implementation has been obtained by a not excessive modification of an existing standard backtracking implementation.

6 ACKNOWLEDGEMENT

We thank Antonio Porto for very fruitful discussions, and for his invaluable help with the implementation of the intelligent backtracking interpreter in Prolog.

In no way is the Instituto Nacional de Investigação Científica in Portugal thanked for not supporting this work.

We acknowledge the support of the Afdeling Toegepaste Wiskunde en Programmatie of the Katholieke Universiteit Leuven and of the Nationaal Fonds voor Wetenschappelijk Onderzoek.

We are indebted to Chris Coudron for the intelligent backtracking interpreter implemented in C.

7 REFERENCES

Bruynooghe, M., (1980), Analysis of dependencies to improve the behaviour of logic programs, *5th Conference on Automated Deduction*, Les Arcs, France, eds. W. Bibel and R. Kowalski, Springer, Lecture Notes in Computer Science, pp. 293–305.

Bruynooghe, M., (1981), Intelligent backtracking for an interpreter of Horn clause logic programs, Colloquium on Mathematical Logic in Programming, Salgotarjan, Hungary, 1978, in *Mathematical Logic in Computer Science*, eds. B. Dömölki and T. Gergely, North-Holland, pp. 215–258.

Bruynooghe, M., (1981a), Solving combinatorial search problems by intelligent backtracking, *Information Processing Letters*, **12**, 1, pp. 36–39.

Bruynooghe, M. and Pereira, L. M., (1981), *Revision of Top-down Logical Reasoning Through Intelligent Backtracking*, Departmento de Informática, Universidade Nova de Lisboa, Portugal, and Departement Computerwetenschappen, K. U. Leuven, Belgium.

Cox, P., (1977), *Deduction Plans: A Graphical Proof Procedure for the First Order Predicate Calculus*, Department of Computer Science, University of Waterloo, Canada.

Cox, P. and Pietrzykowski, T., (1981), Deduction plans: a basis for intelligent backtracking, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-3**, 1.

Cox, P., (1981), *On Determining the Causes of Nonunifiability*, Department of Computer Science, University of Auckland, New Zealand.

Doyle, J., (1979), A truth maintenance system, *Artificial Intelligence*, **12**, pp. 231–272.

Doyle, J., (1980), *A Model for Deliberation, Action and Introspection*, M.I.T. AI Lab, USA.

Kowalski, R. A., (1974), Predicate logic as a programming language, *IFIP*, North-Holland, pp. 569–574.

Kowalski, R. A., (1979), *Logic for Problem Solving*, North-Holland.

Matwin, S. and Pietrzykowski, T., (1982), Plan based deduction: data structures and implementation, in *Proceedings of 6th Conference on Automated Deduction*, New York, USA, Lecture Notes in Computer Science, Springer-Verlag.

McCarthy, J., (1982), *Colouring Maps and the Kowalski Doctrine*, (draft) Department of Computer Science, Stanford, USA.

Pereira, L. M., (1979), *Backtracking Intelligently in AND/OR trees*, Departamento de Informática, Universidade Nova de Lisboa, Portugal.

Pereira, L. M. and Porto, A. (1979), *Intelligent Backtracking and Sidetracking in Horn Clause Programs – The Theory*, Departamento de Informática, Universidade Nova de Lisboa, Portugal.

Pereira, L. M. and Porto, A., (1980), *An Interpreter of Logic Programs Using Selective Backtracking*, Workshop on Logic Programming, Debrecen, Hungary.

Pereira, L. M. and Porto, A., (1980a), Selective backtracking for logic programs, *5th Conference on Automated Deduction*, Les Arcs, France, eds. W. Bibel and R. Kowalski, Springer, Lecture Notes in Computer Science, pp. 306–317.

Pereira, L. M. and Porto, A., (1980b), *Selective Backtracking at Work*, Departamento de Informática, Universidade Nova de Lisboa, Portugal.

Pereira, L. M. and Porto, A., (1979a), *Intelligent Backtracking and Sidetracking in Horn Clause Programs – The Implementation*, Departamento de Informática, Universidade Nova de Lisboa, Portugal.

Pereira, L. M. and Porto, A., (1982), Selective backtracking, in *Logic Programming*, eds. K. Clark and S.-Å. Tärnlund, Academic Press, pp. 107–114.

Pietrzykowski, T. and Matwin, S., (1982), Exponential improvement of efficient backtracking, in *Proceedings of 6th Conference on Automated Deduction*, New York, USA, Lecture Notes in Computer Science, Springer-Verlag.

Robinson, J. A., (1979), *Logic: Form and Function*, Edinburgh University Press.

van Emden, M., (1982), An algorithm for interpreting Prolog programs, in *Proceedings of the First International Logic Programming Conference*, Marseille, France, pp. 56–64, and the present volume.

Warren, D. H. D., Pereira, L. M. and Pereira, F. (1977), Prolog – the language and its implementation compared with LISP, ACM Symposium on Artificial Intelligence and Programming Languages, Rochester, USA, *Sigart Newsletter*, **64**, and *Sigplan Notices*, **12**, 8.