3.38

CENTRO DE INFORMÁTICA
DA UNIVERSIDADE NOVA DE LISBOA

in Horn clause programs - the theory

Luis Moniz Pereira

Antonio Porto

Departamento de Informatica
Universidade Nova de Lisboa
1899 Lisbon, Portugal

October 1979

UNIVERSIDADE NOVA DE LISBOA
CIUNL CIUNL CIUNL CIUNL CIUNL CIUNL

Intelligent backtracking and sidetracking

in Horn clause programs - the theory

Luis Moniz Pereira

Antonio Porto

Departamento de Informatica
Universidade Nova de Lisboa
1899 Lisbon, Portugal

October 1979

report no. 2 /79 CIUNL

Abstract

This report introduces two general execution strategies for logic programs. One is an intelligent form of backtracking, the other a method of proceeding forward which postpones non-deterministic steps. Both strategies are illustrated with examples. Some optional control constructs are also presented, allowing the user to provide control advice. The implementation of these strategies, in the case of the language Prolog, is the subject of a companion report.

Keywords: logic programming, backtracking, artificial intelligence

Contents

1. Introduction

2. Prolog AND/OR trees

3. Backtracking intelligently in AND/OR trees

4. On the possible types of unification conflicts

5. On the obtention of backtrack nodes

6. Reformulation of the operational semantics of Prolog

7. Realistic applications of intelligent backtracking

    7.1 Map colouring example

    7.2 Relational database example

    7.3 Scene analysis example

    7.4 Non-attacking chessboard queens example

    7.5 Generating and parsing grammars example

8. Sidetracking in logic programs

9. Control constructs

    9.1 Intelligent backtracking control

    9.2 Sidetracking control

10. Conclusions

11. Acknowledgements

12. References

1. Introduction

> "The mental features discoursed of as the analytical, are, in themselves, but little susceptible of analysis. We only appreciate them in their effects."
>
> Edgar Allan Poe in "The murders in the Rue Morgue", p.378

The work reported herein divides naturally into two parts for exposition purposes. The first is concerned with a theory of intelligent backtracking and sidetracking, and the second with implementation.

We begin the first part, which is the subject of this report, by presenting an intelligent backtracking method for Horn clause programs, as applied to Prolog [Warren 1977] [Warren et al. 1977] [Pereira et al. 1978a] [Coelho et al. 1979], a programming language based on first order predicate calculus [Kowalski 1974] [Kowalski 1979], developed at the university of Marseille [Roussel 1975]. This method is based on the general method expounded in [Pereira 1979] for backtracking intelligently in AND/OR trees (of which we furnish a résumé).

Afterwards, we present a sidetracking method which consists on delaying the match of any goal for which there is more than one viable clause match, until all goals are matched for which there is a single clause matching them at runtime. This means that any runtime-deterministic goal is executed one step (thereby generating more goals), before any runtime-non-deterministic goal is executed one step. In particular, any pending deterministic procedure is executed before any pending non-deterministic one.

Finally, we present some control constructs allowing the user to guide intelligent backtracking and sidetracking if s/he so wishes. (The use of these techniques in conjunction with parallel execution will be elucidated in a future paper.)

Our motivation for this work was spurred by [Bruynooghe 1978]. We have however greatly improved and generalized his findings in many essential ways, due to a distinct and more thorough approach. Motivation for some kind of control over resolution theorem provers [Loveland 1978], and attendant combinatorial "diseases", has always flourished. But general search strategies have remained too general, and the "cure" too slow. One approach to the problem is to devise some kind of control language in which the user may optionally supply additional information regarding the control of search as it proceeds forward only [Clark et al. 1979]. Our main contribution to the "cure" is in the form of a general strategy for backtracking intelligently in any top-down execution of Horn clause programs. Furthermore, such backtracking is also susceptible of optional control constructs, which we illustrate. The adaptation of the techniques developed here to any resolution-logic theorem prover is, we believe, straightforward, as long as its executions can be modelled by an AND/OR tree. It should be pointed out that our theory has been tested thoroughly in its implemented form. No example has falsified it, all have corroborated it.

The rest of this paper is organized as follows. To begin with, we relate Prolog program executions to AND/OR trees in a slightly unusual way. Immediately afterwards we furnish a self-contained

summary of the theory established in [Pereira 1979] for backtracking intelligently in AND/OR trees. Next we show, by means of illustrative examples, the various types of conflicts that may arise during unification. Subsequently, we explain and exemplify how admissible backtrack nodes are obtained for each goal that fails, and we reformulate the operational semantics of Prolog accordingly. Then we describe some realistic examples of application. Afterwards, the method of sidetracking to runtime-deterministic goals is explicated and exemplified. Finally, a few control constructs are put forward which permit the user to guide, optionally, both the backtracking and the sidetracking.

The second part of our work is the subject of another report. There, a general interpreter of Prolog is presented which uses structural intelligent backtracking instead of the standard chronological blind one. (We also present a simpler specialization of this interpreter, meant only for use in relational database lookup.) The interpreter is written in Prolog itself, and is not aimed at efficiency, since we have only guided standard backtracking, provided by the compiler of Prolog, to mimick intelligent backtracking. This means that we have simulated at a high level what requires an appropriate implementation of its own to achieve competitive efficiency. Hopefully, this will be done in the near future. (The specialized interpreter, nevertheless, has proved to be competitive already, even for smallish databases.) The basic implementation design idea is to attach directly to each component of a term the nodes of the execution tree on which it depends. This information is automatically passed on by unification and forgotten on backtracking. Intelligent backtracking uses it to

know where to backtrack to.

Another interpreter is presented there, again written in Prolog, for runtime-deterministic priority execution of Prolog programs (sidetracking), and the way of combining both interpreters into one is pointed out.

2. Prolog AND/OR trees

"Yet to calculate is not in itself to analyse."

ibidem, p.379

To solve a goal a clause must be found whose head matches the goal and all the goals in its body can be solved. Execution of a goal generates an AND/OR tree, with that goal as the root, in the following way.

The clauses whose heads match a goal give rise to ORed branches at that goal. Furthermore, each such OR branch is split into ANDed branches. One such branch leads to the node where unification between the goal and the clause head must be achieved. The other ANDed branches lead to each of the goals in the body of the clause.

The unification node is itself an AND of the unifications of corresponding arguments in the goal and clause head. Each of these arguments, moreover, may be a compound term giving rise to further ANDed branching of the match of the functor names and of each of their arguments, and so on.

3. Backtracking intelligently in AND/OR trees (a résume' of [Pereira 1979])

> "It may be sufficient here to say that it forms one of an infinite series of mistakes which arise in the path of Reason through her propensity for seeking truth <u>in detail</u>."
>
> E.A.Poe in "The mystery of Marie Roget - a sequel to The murders in the Rue Morgue", p.454

A general method is presented in [Pereira 1979] for guiding standard backtracking in AND/OR trees only to those nodes where repetition of goal failures may possibly be prevented, thus avoiding much useless search. Each goal that fails originates a stack of the admissible backtrack nodes susceptible of either solving or avoiding it, and a simple rule combines individual stacks into a single new stack at AND and OR nodes. The admissible backtrack nodes of a failed goal are, besides its ancestors, where alternative branches of the tree may exist, all the nodes where modification of any conflicting objects in the attempted matches of the goal may perhaps be achieved. The former are readily obtained if each object in a goal is tagged, implicitly or explicitly, with those nodes. This way, backtracking from a failed goal is admitted only to those nodes where subsequent failure of the same goal may possibly be prevented.

Identification of the admissible backtrack nodes for a goal is done only on failure, not as the search moves forward. The goal's admissible backtrack nodes are organized in an ordered stack, with the most recent one coming first on the stack. (The nodes of the tree are numbered from the root by the search strategy.)

For a failing terminal goal node, its admissible backtrack nodes are precisely those where modifications may occur to the failure originating components of the objects it refers to, plus its parent node, where alternative branches may exist (if necessary, other ancestors will be accessed, recurrently, via the parent node).

For a failing non-terminal goal node, the stacks of admissible backtrack nodes of its failed immediate successors are combined to form its ordered backtrack stack, where its parent node is included if absent.

The rule for combining stacks is simple. The stacks coming from ORed branches are just merged to form the OR node stack, because each contains backtrack nodes susceptible of generating candidate solutions on different branches of the OR. The stack obtained at nodes with ANDed branches coincides with that of the branch stacks which is less than or equal to any other stack, according to the lexicographic order among stacks. Recall that the stacks are themselves ordered, with the most recent node (that with the highest number) coming first on the stack. Since nodes are numbered from the root, the first admissible backtrack node of the AND node stack will be the least recent of the most recent nodes of all the branch stacks, since each branch must lead to a solved node. From the point of view of that AND node, if there are no alternatives at that first backtrack node, then the next backtrack node to visit must be the next most recent node on the stack chosen as the AND node stack, for the same reason, and so on. If there are alternatives at that node, on the other hand, and should there be subsequent failing goals, perhaps at the same places, the

stacks of admissible backtrack nodes will be obtained anew.

When, finally, there are no more admissible backtrack nodes to try on backtracking, failure of the top goal is reported. This happens only when backtracking reaches the top goal itself, because every parent node is admissible on failure of its successors.

This discussion assumed that at each node information was available concerning its admissible backtrack nodes in the face of failure. Specifically, we need to know, for each object component contributing to the failure, all the nodes on which its presence in the current goal depends. Therefore, this form of backtracking is aptly called "structural".

Failure of terminal goals may occur for many reasons [Loveland 1978]. For example, the goal may be equal to, or a particular case of, an ancestor ; the goal has no potential successors; all potential successors cannot become actual successors on closer examination.

Irrespective of the reasons for failure, the parent goal is always an admissible backtrack node. In case failure does not depend on the objects involved in the goal, it will be the only such node. In case failure depends on the particular objects involved but one cannot or does not care to analyse which, then all the nodes on which the object components depend are additional admissible backtrack nodes.

In case failure is known to depend precisely on specific object components, then the only extra admissible backtrack nodes are exactly those on which such object components depend on.

Methods of object component tagging are too peculiar to problem representation and system implementation to be considered with any generality. They are treated in the implementation report in what concerns Prolog.

4. On the possible types of unification conflicts

> "When I say proficiency, I mean that perfection [...] which includes a comprehension of all the sources whence legitimate advantage may be derived. These are not only manifold but multiform, and lie frequently among the recesses of thought altogether inaccessible to the ordinary understanding. To observe attentively is to remember distinctly [...]. The necessary knowledge is that of what to observe."
>
> E.A.Poe in "The murders in the Rue Morgue", p.380

Composition of the stacks of admissible backtrack nodes generated at the branches of an AND/OR tree is governed by the rules set forth in [Pereira 1979], summarized above. In this section we concentrate on the different types of conflict that provoke failure of a goal, so that in the next section we might concentrate on how an individual stack is obtained, upon failure of a goal, in the case of Prolog.

Assume failure occurs because of a mismatch between the goal and some clause head. (Other reasons for failure, such as the absence of a clause for that goal, are dealt with as explained in the resume' of [Pereira 1979] above and illustrated in the examples). This means that at least one mismatch between two constant names has occured. If there are several mismatches, their backtrack nodes are composed according to the rule for ANDed branches, as exemplified below. Accordingly, the stack of backtrack nodes retained from the attempted match with a clause head coincides with the stack generated by just one of the conflicts (ie. the least among the stacks of individual conflicts).

We distinguish two cases.

> "For, in respect to the latter branch of the supposition, it should be considered that the most trifling variation in the facts of the two cases might give rise to the most important miscalculations."
>
> E.A. Poe in "The mystery of Marie Roget - a sequel to The murders in the Rue Morgue", p. 410

<u>First case</u>. One of the constant names textually occurs at the conflicting position in the head of the clause and so it cannot be changed.

For the match to succeed the other constant name must change. Now, if this other constant name is also textually present in the goal, no backtrack node besides the parent node is generated by this conflict, meaning that the conflict cannot be solved. This situation occurs with the match of goal 3 in Example 1 below, where a and b conflict, regardless of the value of X.

If the second constant name is not textual, then it is obtained through some variable textually present in the goal. In that case, the backtrack nodes generated are all those nodes responsible for the transmission of that constant name to the variable. This is the case of the instantiation b of variable X in goal 3 below, where goal 1 is generated as a potential backtrack node. However, the other conflict (between a and b) generates 0 as a backtrack node, and the rule for combining backtrack stacks at ANDed branches retains only 0 as a backtrack node since it is the least recent. If an extra clause was added such as

r(c,a).

backtrack node 1 would be merged into the backtrack stack for goal r(X,a), since clauses for r are ORed in the AND/OR tree.

```
        0        1         2         3
    t(X,Y):- p(X,X),q(X,a,d,Z,Z),r(X,a).

    p(b,d).
    p(W,V):- unifiable(V,b).

    q(X,X,Y,c,Y).
    q(b,a,d,c,c).

    r(c,b).

    unifiable(V,V).
```

Example 1

When, in particular, the second conflicting constant name (present in the binding of the goal variable) is textually present in the clause head itself and is transmitted to that goal variable through a multiple textual occurrence in the goal of that variable, the conflict is irrevocable. This happens when goal 1 attempts to match the first clause. The conflict between the binding of the second occurrence of X and d is inherent to the match and cannot be removed.

Second case. No constant name textually occurs at the conflicting position in the head of the clause.

There is a variable at that position, that must receive the mismatching constant name from the goal itself, at some other of its textual occurrences. In that case, the conflict is really between two constant names refered by the goal. Accordingly, the conflict may be solved if either one of them changes, and the backtrack nodes are all those on which their reference by the goal depends. The second argument position in the match of goal 2 with the first clause illustrates this situation. (a conflicts with b through X.) Again, if both conflicting constant names are textually present in either the goal or head, the conflict is also irrevocable. This is the case for the last argument position in the match of goal 2 with the first clause. (d and c conflict through Y and Z.) The conflict is inherent to the match and no amount of backtracking can possibly abolish it. Thus, this conflict overrides any potential backtrack nodes generated for the other conflicts in the same match (which are not retained).

5. On the obtention of backtrack nodes

> "I repeat that it is more than fact that the <u>larger</u> portion of all truth has sprung from the collateral; it is but in accordance with the spirit of the principle involved in this fact that I would now divert inquiry, in the present case, from [...] the event itself to the contemporary cirumstances which surround it."
>
> ibidem, p. 436

Now that we have completed the analysis of the types of conflict that may arise between constant names, we next examine, through a case analysis, how the presence of a constant name in the binding of a variable depends on the matches performed at the nodes of the AND/OR tree.

1) The constant names textually present in the goal and/or matching head depend solely on the match node. If the conflict occurs between such constant names, either another clause must be chosen to match the goal or the parent goal must resort to another clause, as we have seen in Example 1.

2) The presence of a constant name as part of the binding of a textual variable in either the goal or matching head depends solely on the matches which have transmitted that part of the binding.

2.1) If in a match a variable is bound to some non-variable term directly, ie. not through any intervening variables, then all the constant names part of that binding depend on that match. This is the case of the binding c of Z in the matching of goal 2 with the head of the second clause for q in Example 1.

2.2) The only remaining case is where a variable becomes dependent by unification on other variables, even though the latter may have not yet acquired a value. For, in case they do acquire a value, and failure occurs because of that transmitted value, the matches originating the dependencies on those variables constitute potential backtrack nodes:-

2.2.1) The simplest subcase is that of the dependency of a variable occurring in the clause head upon the actual or future binding (possibly in the same match) of a corresponding variable in the matching goal. Because every failing goal in the matching clause body (possibly failing because of the binding of one such variable) must always have its parent node as a backtrack point, the dependencies on the parent created by such variables need not be explicit. The dependency of a goal on the parent can be systematically noted by the interpreter, when a goal fails. In that case, the only dependency analysis to be performed is that of the binding of the variable the head variable has been unified with. This is the case, in Example 1, of the dependency of W in the body of the following extra third clause for q on the binding of Z in goal 2.

```
q(b,a,d,a,W):- q(b,a,d,a,f(W)).
```

In particular, the subcase where two variables in a clause head are unified to one another through one same multiple occurring variable in the goal, is already catered for, given that any goal in the clause body containing any of such head variables is systematically made to depend on the parent node whenever it fails.

2.2.2) Another case is where a goal variable becomes dependent on some future binding (possibly in the same match) of a corresponding variable in the head:-

2.2.2.1) If there is a single occurrence of the head variable in the head, then the future binding can only be made in the execution of the body of the clause. Any failure due to the transmission of that binding via the goal variable will depend on a subgoal of that goal. But this subgoal, through its parent, automatically depends on the matches of all its ancestors, and so also on the match of the goal with the clause head. Consequently, the dependency of the goal variable on the head variable established in the match need not be noted explicitly. This is the case, in Example 1, of the implicit dependency of the binding b of X on the match of goal 1 with the second clause for p.

2.2.2.2) If there is a multiple occurence of the head variable in the head, the future binding may be performed during the match itself, namely with a textual non-variable term or a (possibly bound) variable in the goal. What happens then is that the head variable is linking two or more terms in the goal. So any dependency of those terms on each other depends on this match for its transmission. This is the sole case, regarding the matching of variables, which needs explicit noting of the match node as a potential backtrack node because failure caused by the binding transmitted through the goal variable may not depend on any subgoal of the goal. The next example illustrates this well.

```
   0        1        2        3        4        5
t(X,Y):- p(X,Y),p(Y,W),p(X,Z),q(Var1),r(Var2).

p(X,X).

q(a).

r(b).
```

Example 2

At goal 5, if Var1 stands for W and Var2 for Z, then all nodes are backtrack nodes, since there could exist other clauses for p and q. If Var1 is Y and Var2 is Z, only 2 is not a backtrack node. If Var1 is Y and Var2 is X, only 2 and 3 are not backtrack nodes.

Let us sum up our case analysis. Because parent nodes are always included as backtrack nodes of failing goals by the interpreter, the node dependencies created by simple transmission of bindings up and/or down the tree (through chains of ancestors possibly linked by common variables at brother nodes) need not be noted explicitly. Any node where a textual non-variable term is bound to a variable must be retained as a potential backtrack node for any constant name now part of the binding of the variable. The only other kind of node which must also be retained as a backtrack node of bindings, is where two (or more) terms refered by the goal create a dependency between them at that node.

Finally, the next example exhibits a variety of cases in combination, and shows the dynamics of intellingent backtracking. Before we describe it some notation is required.

```
start:-example(e).
       0-0

example(V):-uncle1(X,  Y ,V),uncle2,parent(f(X), Y ).  0[0-0] 0(0)
                                                                *

 0-0    e    0-1   a g(e) e   0-2    0-3     a  g(e)
             3-1   d h(e) e   3-2    3-3     d  h(e)

parent(  W , Y ):-brother(  Y , Z ),goal(  W , Z ).     2[0-0,0-1] 2(0,1)
                                                                     *
 0-3   f(a) g(e)     0-4   g(e)   Z    0-8 f(a) g(b)
                     2-4   g(e) g(e)   1-8 f(a) h(c)      3[0-0,3-1] 3(0,1)
 3-3   f(d) h(e)     3-4   h(e)   Z    2-5 f(a) g(e)                   *
                                       4-5 f(d) h(e)      4[0-0,3-1] 4(0,1)
uncle1(a,g(V),V).   1[0-0]                                             *
 0-1        e  e             3(0)
                               *
uncle1(d,h(V),V).   3[0-0]
 3-1        e  e

uncle2.        uncle2:-uncle2.
 0-2

brother(  Y ,Z):-nephew1( Z ),nephew2( Y ).         2[0-1,0-3]
                                                               2(1,3)
  0-4   g(e) Z    0-5    g(b)  0-6    g(e)          2[0-3]       *
  3-4   h(e) Z    1-5    h(c)  1-6    g(e)
                  3-5    g(b)  3-6    h(e)          3[3-1,3-3]
                                                               4(0,1,3)
brother(  X ,  X ).                                 4[]            *
  2-4   g(e) g(e)
  4-4   h(e) h(e)

nephew1(g(b)).              1[0-1,0-3,1-4]
 0-5                                        1(1,3,4)
 3-5                                            *

nephew1(h(c)).              1[0-1,0-3,1-4]
 1-5

nephew2(e).                 3[3-1,3-4]
 0-7                                   3(0,1,4)
 1-7                                       *

nephew2(g(X)):-nephew2(X).  3[3-1,3-4]
 0-6       e      0-7     e
 1-6       e      1-7     e

goal(t(e),m).      0[0-3]      0(3,5)           1[0-3]      1(1,3,5)
                                 *                              *
goal(f(X),h(X)).   0[0-3,0-5]                   1[0-1,0-3]

                   2[0-3]          2(1,3,4)     4[3-3]           4(0,1,3,4)
                   2[2-4,0-3,0-1]      *        4[0-0,3-1,3-3,4-4]       *
```

Below each goal and successful matching clause head are displayed the bindings of each variable at ocasion i-j, where j is the AND/OR tree node number of the match, and i numbers the path started the ith time execution began moving forward (including any backtracking up to the point where execution moved forward again).

Besides each set of clauses for the same predicate, for each clause there are objects of the form k[i-j,...] for everytime a goal for that predicate finally failed. k is the path number at the time of failure, and is followed by a list of the backtrack nodes generated in the subtree of the match node for that goal and clause. As before, j is the node number and i is the path number of j. To the right of these objects is shown the complete backtrack stack at the time of the final failure of the goal. The backtrack stack, with the form

k(j1,...,jn)
*

incorporates both the backtrack nodes obtained for the goal from the composition of the backtrack nodes (according to the rule for OR nodes) of individual clauses for that predicate, and any backtrack nodes subsisting from previous paths. An * signals the next node to be backtracked to (ie. the most recent node which is on the stack). Analysis of backtrack nodes generated by conflicts is only performed after all the clauses for a goal have been tried. If at least one clause does not fail, no analysis is done at all. Note that analysis does not stop at the first conflict encountered, as unification does, but goes on to analyse all conflicts.

The first forward path is 0. Failure occurs of goal(f(a),g(b)) at 0-8. The first clause for goal provides backtrack nodes 3 and 5; f conflicts with t and was obtained at 0-3; g conflicts with m and was obtained at 0-5; dependency on the transmission at 4 is not noted because 4 is a parent of 5 and thus will be generated if 5 fails; only 3 is retained, according to the rule for AND nodes, since it is the least recent; the parent node is also 3. In the second clause, only g conflicts with h; so 5 is a backtrack node. Now the rule for OR nodes merges 3 and 5 into the single backtrack stack 0(3,5).

So backtracking returns to 5 (the most recent on the stack), and path 1 begins. Failure occurs of goal(f(a),h(c)) at 1-8. As before, the first clause provides only 3 as a backtrack node. In the second clause, the conflict is between a and c of the goal, and both give backtrack nodes which are ORed into one same stack; a was obtained at 1 through 3, and c at 5 through 4. Again 4 is not retained, but 3 is because it is the parent. The stack becomes 1(1,3,5).

Backtracking is to 5, where there are no more clauses. So 5 fails, and its clauses give only their parent 4 as a backtrack node. The stack is updated to 1(1,3,4).

At 4 there is still another clause, and path 2 begins. It fails with goal(f(a),g(e)) at 2-5. In the first clause, the conflict of f with t gives 3; of g with m gives 4 and 1. The rule for AND nodes retains only 3, which is also the parent node. The resulting stack is 2(1,3,4).

Backtracking returns to 4 where there are no more clauses. Since

there are no conflicting matches with the heads of the clauses, the backtrack nodes for each clause are just the remaining ones inherited from their bodies plus their matching goal's parent node. The stack is now 2(1,3). Backtracking then goes to 3 where, similarly, the stack becomes 2(0,1).

Intelligent backtracking now jumps to 1 over 2 (a loop for standard backtracking), and the second clause starts path 3. It fails with goal(f(d),h(e)) at 4-5. In the first clause, f conflicts with t giving 3; h conflicts with m giving 1, 4, and 0. The stack is now 4(0,1,3,4).

Back at 4 there are no more clauses, no conflicts, and the stack becomes 4(0,1,3). At 3 there are no more clauses, no conflicts, and the stack is updated to 4(0,1). At 1 there are no more clauses, no conflicts, and the stack contains only the top goal. (Uhf!)

The reader is urged to make sure s/he understands this example, to compare its dynamics with that of standard backtracking, and to become convinced that the case analysis in the preceding section covers all the situations s/he can think of. Notice that no solutions are lost compared to standard backtracking. On the contrary, some new solutions may be found due to the possibility of backtracking over infinite loops. Furthermore, the order among the solutions of standard backtracking is preserved.

6. Reformulation of the operational semantics of Prolog

> "And what means are ours of attaining the truth? We shall find these means of multiplying and gathering distinctness as we proceed."
>
> ibidem, p.451

Let us now reformulate the operational semantics of Prolog Horn clause programs, in an informal way, to take into account intelligent backtracking.

To execute a goal, the system searches for the first clause whose head matches or unifies with the goal. The unification process [Robinson 1965] [Robinson 1979] finds the most general common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then activated by executing in turn, from left to right, each of the goals (if any) in its body. If at any time the system fails to find a match for a goal, it intelligently backtracks, ie. it reconsiders in turn all the clauses for the failed goal, and for each finds the backtrack nodes of every conflict in the match of the goal with the clause ; next it composes the stacks of backtrack nodes so obtained according to the rule for AND branching of [Pereira 1979] ; then the stack contributed by each clause is merged with the ones of the other clauses into a single backtrack stack, according to the rule for OR branching of [Pereira 1979], and the parent of the failed goal and any remaining backtrack nodes are included in it; after that, the system chooses the most recent backtrack node on the stack, forgetting completely all its successor nodes, and rejects the clause activated at that node, undoing any substitutions made by the match with the head of that clause;

finally, it deletes that node from the stack, reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal.

7. Realistic applications of intelligent backtracking

"It is now rendered necessary that I give the *facts* - as far as I comprehend them myself."

E.A.Poe in "The facts in the case of M.Valdemar", p.281

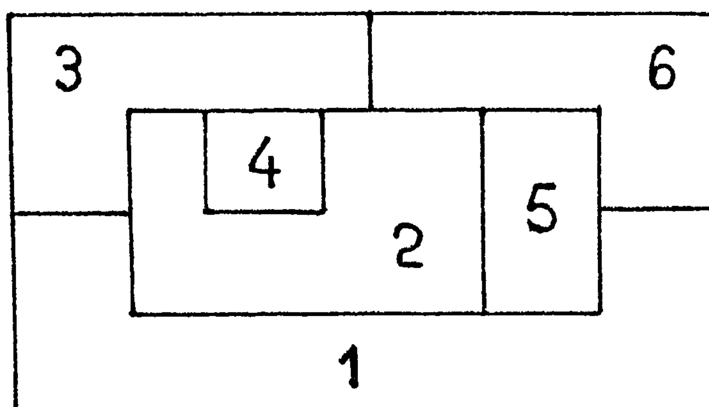
7.1 Map colouring example

Our first example is a straightforward program for colouring any planar map with at most four colours, such that no two adjacent regions have the same colour (this has been proved to be always possible).

The program consists in a complete list of pairs of different colours, taken from a collection of four. These constitute the admissible pairs of colours for regions next to each other.

```
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).
```

To obtain a colouring of a map such as the one below, we must give as a goal to the program all the pairs of regions that are next to each other. This can be done in a systematic way by first numbering every region in the map, and then, starting with the lowest number, by pairing each region with all higher numbered regions which stand next to it.



For this map we have

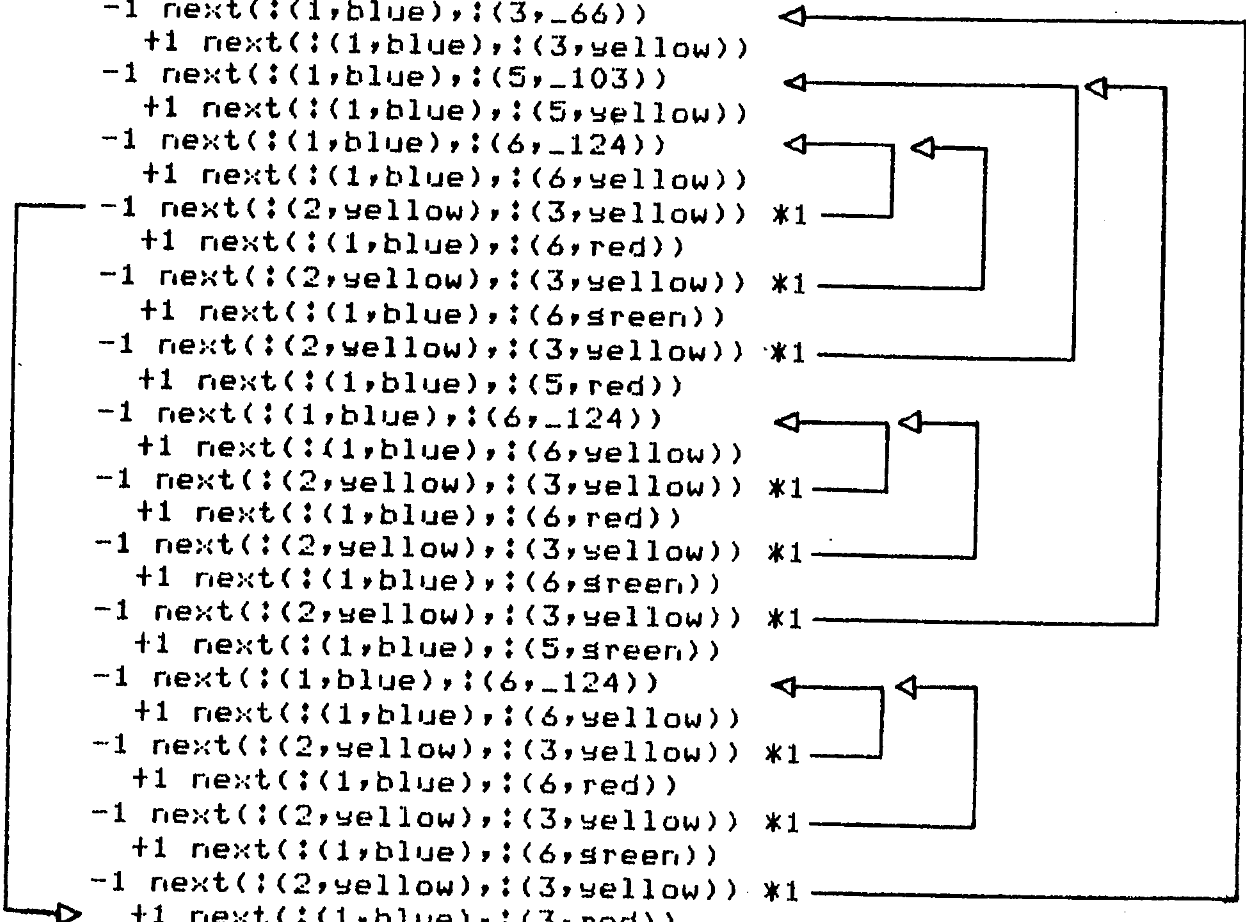
```
goal(R1,R2,R3,R4,R5,R6):-
       next(R1,R2), next(R1,R3), next(R1,R5), next(R1,R6),
       next(R2,R3), next(R2,R4), next(R2,R5), next(R2,R6),
       next(R3,R4), next(R3,R6),
       next(R5,R6).
```

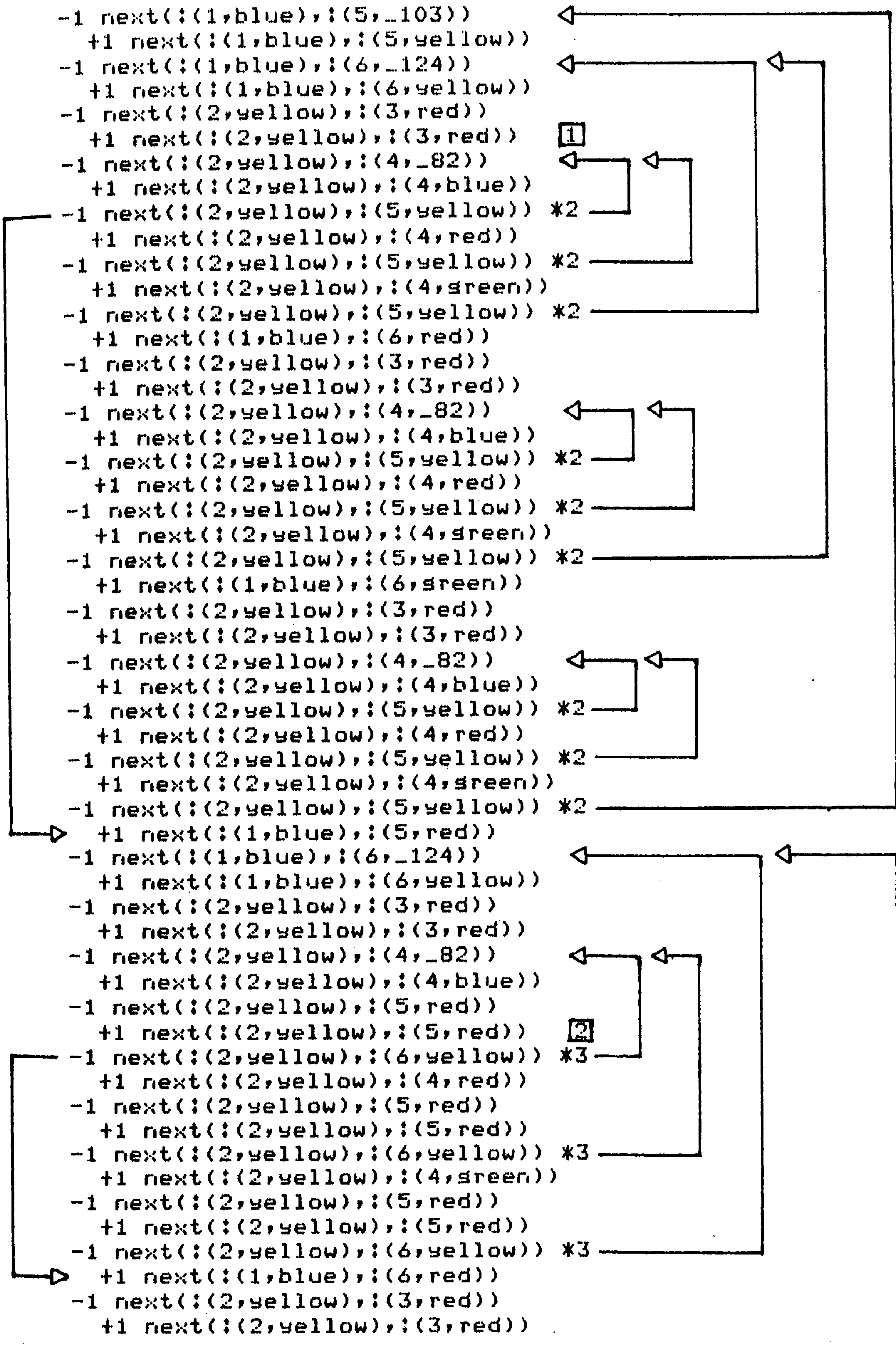
The effect of intelligent backtracking is best seen in the trace of standard backtracking below. It drastically improves the execution. To help to follow the trace we have replaced the calls in the goal, which are of the form next(Ri,Rj), by calls of the form next(i:Ri,j:Rj). Accordingly, a program clause such as next(blue,yellow) has been changed to next(Ni:blue,Nj:yellow).

In it, each procedure (or predicate) call is displayed with the current value of its arguments, and is preceded by a number with a "-" sign, which indicates the AND/OR tree level of the goal being displayed. Also, each time execution of a goal is successfully completed, the call is displayed again with the new current values of all its arguments. Again the current level of the goal is indicated, this time preceded by a "+" sign and the invocation level. In the trace, functors are in prefix notation, and numbers preceded by "_"

are arbitrary names for variables. An "*" indicates failure of a goal. Each failing goal is numbered and, on the right-hand side, a line with arrows points from the failed goal to the goal to which standard backtracking returns. Successful completion of a previously failed goal is signaled by the boxed number of that failed goal. On the left-hand side, similar lines with arrows point from a failed goal to the point further down in the trace which would correspond to an execution with intelligent backtracking. The segments of trace jumped over correspond exactly to the useless computation of standard backtracking, as compared with the intelligent version.

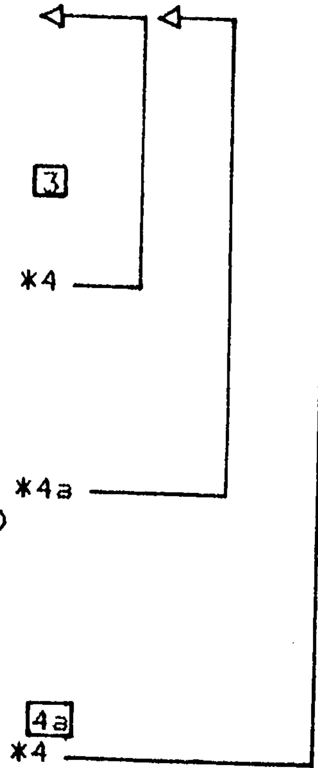
goal(R1,R2,R3,R4,R5,R6).





```
-1 next(:(1,blue),:(5,_103))
  +1 next(:(1,blue),:(5,yellow))
-1 next(:(1,blue),:(6,_124))
  +1 next(:(1,blue),:(6,yellow))
-1 next(:(2,yellow),:(3,red))
  +1 next(:(2,yellow),:(3,red))          [1]
-1 next(:(2,yellow),:(4,_82))
  +1 next(:(2,yellow),:(4,blue))
-1 next(:(2,yellow),:(5,yellow))  *2
  +1 next(:(2,yellow),:(4,red))
-1 next(:(2,yellow),:(5,yellow))  *2
  +1 next(:(2,yellow),:(4,green))
-1 next(:(2,yellow),:(5,yellow))  *2
  +1 next(:(1,blue),:(6,red))
-1 next(:(2,yellow),:(3,red))
  +1 next(:(2,yellow),:(3,red))
-1 next(:(2,yellow),:(4,_82))
  +1 next(:(2,yellow),:(4,blue))
-1 next(:(2,yellow),:(5,yellow))  *2
  +1 next(:(2,yellow),:(4,red))
-1 next(:(2,yellow),:(5,yellow))  *2
  +1 next(:(2,yellow),:(4,green))
-1 next(:(2,yellow),:(5,yellow))  *2
  +1 next(:(1,blue),:(6,green))
-1 next(:(2,yellow),:(3,red))
  +1 next(:(2,yellow),:(3,red))
-1 next(:(2,yellow),:(4,_82))
  +1 next(:(2,yellow),:(4,blue))
-1 next(:(2,yellow),:(5,yellow))  *2
  +1 next(:(2,yellow),:(4,red))
-1 next(:(2,yellow),:(5,yellow))  *2
  +1 next(:(2,yellow),:(4,green))
-1 next(:(2,yellow),:(5,yellow))  *2
  +1 next(:(1,blue),:(5,red))
-1 next(:(1,blue),:(6,_124))
  +1 next(:(1,blue),:(6,yellow))
-1 next(:(2,yellow),:(3,red))
  +1 next(:(2,yellow),:(3,red))
-1 next(:(2,yellow),:(4,_82))
  +1 next(:(2,yellow),:(4,blue))
-1 next(:(2,yellow),:(5,red))
  +1 next(:(2,yellow),:(5,red))          [2]
-1 next(:(2,yellow),:(6,yellow))  *3
  +1 next(:(2,yellow),:(4,red))
-1 next(:(2,yellow),:(5,red))
  +1 next(:(2,yellow),:(5,red))
-1 next(:(2,yellow),:(6,yellow))  *3
  +1 next(:(2,yellow),:(4,green))
-1 next(:(2,yellow),:(5,red))
  +1 next(:(2,yellow),:(5,red))
-1 next(:(2,yellow),:(6,yellow))  *3
  +1 next(:(1,blue),:(6,red))
-1 next(:(2,yellow),:(3,red))
  +1 next(:(2,yellow),:(3,red))
```

```
-1 next(:(2,yellow),:(4,_82))
  +1 next(:(2,yellow),:(4,blue))
-1 next(:(2,yellow),:(5,red))
  +1 next(:(2,yellow),:(5,red))
-1 next(:(2,yellow),:(6,red))
  +1 next(:(2,yellow),:(6,red))          [3]
-1 next(:(3,red),:(4,blue))
  +1 next(:(3,red),:(4,blue))
-1 next(:(3,red),:(6,red))               *4
  +1 next(:(2,yellow),:(4,red))
-1 next(:(2,yellow),:(5,red))
  +1 next(:(2,yellow),:(5,red))
-1 next(:(2,yellow),:(6,red))
  +1 next(:(2,yellow),:(6,red))
-1 next(:(3,red),:(4,red))               *4a
  +1 next(:(2,yellow),:(4,green))
-1 next(:(2,yellow),:(5,red))
  +1 next(:(2,yellow),:(5,red))
-1 next(:(2,yellow),:(6,red))
  +1 next(:(2,yellow),:(6,red))
-1 next(:(3,red),:(4,green))
  +1 next(:(3,red),:(4,green))           [4a]
-1 next(:(3,red),:(6,red))               *4
  +1 next(:(1,blue),:(6,green))
-1 next(:(2,yellow),:(3,red))
  +1 next(:(2,yellow),:(3,red))
-1 next(:(2,yellow),:(4,_82))
  +1 next(:(2,yellow),:(4,blue))
-1 next(:(2,yellow),:(5,red))
  +1 next(:(2,yellow),:(5,red))
-1 next(:(2,yellow),:(6,green))
  +1 next(:(2,yellow),:(6,green))
-1 next(:(3,red),:(4,blue))
  +1 next(:(3,red),:(4,blue))
-1 next(:(3,red),:(6,green))
  +1 next(:(3,red),:(6,green))           [4]
-1 next(:(5,red),:(6,green))
  +1 next(:(5,red),:(6,green))
  +0 goal(blue,yellow,red,blue,red,green)
```



```
R4 = blue,
R5 = red,
R6 = green,
R1 = blue,
R2 = yellow,
R3 = red.
```

7.2 Relational database example

This example deals with an extendable database of students who take courses, professors who teach courses, and courses held on certain weekdays and rooms. Our example query is

"Is there a student such that a professor teaches him two different courses in the same room?"

```
query(S,P):- student(S,C1),       1
             course(C1,D1,R),     2
             professor(P,C1),     3
             student(S,C2),       4
             course(C2,D2,R),     5
             professor(P,C2),     6
             C1 \== C2.           7

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(eureka,music).
professor(eureka,art).
professor(eureka,science).
professor(eureka,physics).

course(prolog,mon,room1).
course(prolog,fri,room1).

course(surf,sun,beach).

course(maths,tue,room1).
course(maths,fri,room2).

course(science,thu,room1).
course(science,fri,room2).

course(art,tue,room1).

course(physics,thu,room3).
course(physics,sat,room2).
course(music, wednesday, room3).
```

are picked up. Failure occurs again at 7 but, after backtracking is made to 4, another solution is found for C2, namely prolog. However, the professor of music is no professor of prolog. So 6 fails. Because P and C2 have originated at 3 and 4, respectively, the stack is updated to (0,3,4). And so on and so back and forth.

Could the query have been posed in a more efficiently executed way ? For example, could

```
query(S,P):- student(S,C1),      1
             student(S,C2),      2
             C1 \== C2,          3
             professor(P,C1),    4
             professor(P,C2),    5
             course(C1,D1,R),    6
             course(C2,D2,R).    7
```

dispense intelligent backtracking ? No. Failure at 7 because of R would provoke backtracking to 6, and thereafter to 2 if no compatible R was found.

What about the following query ?

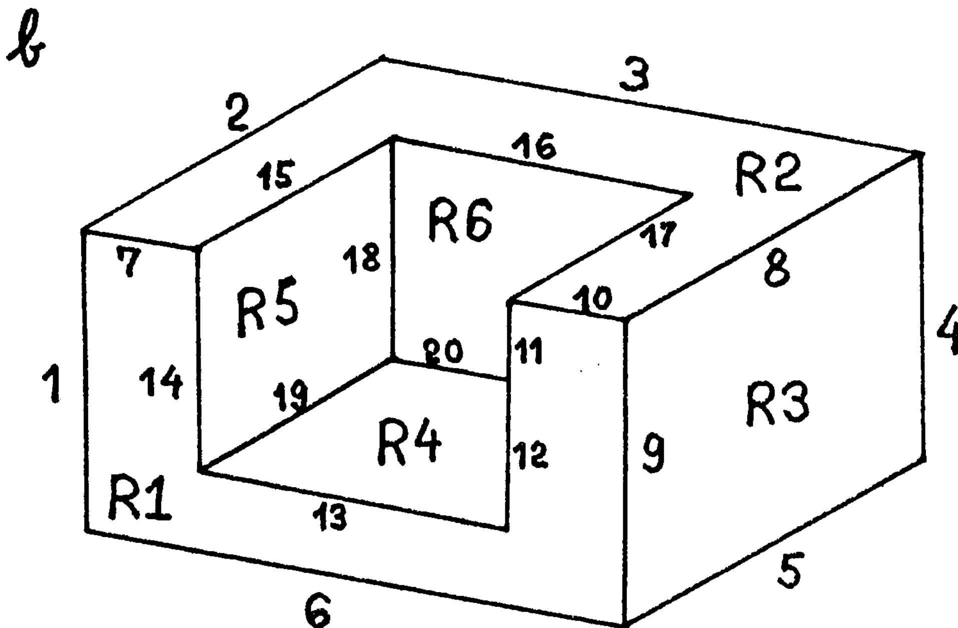
```
query(S,P):- student(S,C1),      1
             student(S,C2),      2
             C1 \== C2,          3
             course(C1,D1,R),    4
             course(C2,D2,R),    5
             professor(P,C1),    6
             professor(P,C2).    7
```

Again no. Failure at 7 because P does not teach C2 requires backtracking to 6, and subsequently to 2 if no common P is found for C1 and C2 as they stand.

One of our theses is that intelligent backtracking provides a powerful runtime form of control for efficient database lookup, which can be used in conjunction with forward forms of control. For the

moment, we postpone the discussion of control cooperation and of implementation overheads to future sections.

7.3 Scene analysis example



The admirably simple program below, due to Warren [Coelho et al. 1979], is capable of identifying each solid in a scene of trihedral solids. It does so by specifying for each plane region R the body it belongs to, and also its position relative to viewpoint : whether it is horizontal (h), leaning to the left (l), or leaning to the right (r). Viewpoints must be chosen above the scene such that all regions fall into one of these categories, ie. no vertical frontal planes are allowed. Edge segments are also identified by assigning them labels. Edge segments separating a visible region of a solid from an invisible one are labeled "f" (for "frontier"). Edge segments separating two visible regions of the same solid are labeled

'+' if they belong to a convex intersection of two visible regions, and '-' if they belong to a concave one.

The input to the program consists of a list of the edge segments present in the scene. Each edge segment E separating two regions R1 and R2, where R1 is to the right of R2 along the edge segment, produces a goal of the form

<u>type</u>(R1,R2,E)

where type is either v, p, or n, according to whether the edge is vertical, positive sloped, or negative sloped. Horizontal edges are precluded by the condition excluding vertical frontal regions. R1, R2, and E are variables to which will be assigned the corresponding identifying labels. Constraints on labels are transmitted from one goal to another through these variables.

```
:-op(500,xfy,[;]).    /* specification of infix functor */

solution(R1; R2; R3; R4; R5; R6,
         E1; E2; E3; E4; E5; E6; E7; E8; E9; E10; E11;
         E12; E13; E14; E15; E16; E17; E18; E19; E20):-

    v(b,R1,E1), p(b,R2,E2), n(R2,b,E3), v(R3,b,E4),
    p(R3,b,E5), n(b,R1,E6), n(R1,R2,E7), p(R2,R3,E8),
    v(R1,R3,E9), n(R1,R2,E10), v(R6,R1,E11), v(R4,R1,E12),
    n(R1,R4,E13), v(R1,R5,E14), p(R2,R5,E15), n(R6,R2,E16),
    p(R6,R2,E17), v(R5,R6,E18), p(R5,R4,E19), n(R4,R6,E20).
```

The program consists just of twelve unit clauses, four for each edge slope. A rule like v(l:S, r:S, +) states that a vertical edge can be a convex edge if it separates a left leaning region belonging to solid S from a right leaning region belonging to the same solid. We leave to the reader the exercise of checking out the completeness of these rules.

```
:-op(500,xfy,[:]).    /* specification of infix functor */

v(l:S, r:S, +).
v(r:S, l:S, -).
v(X, l:S, f).
v(r:S, X, f).

p(h:S, r:S, +).
p(r:S, h:S, -).
p(X, h:S, f).
p(r:S, X, f).

n(l:S, h:S, +).
n(h:S, l:S, -).
n(h:S, X, f).
n(X, l:S, f).
```

What we want to show is how intelligent backtracking improves the execution of this program; that effect is best seen in the trace of standard backtracking below.

The notation is as explained in 7.1 .

```
solution(RS,ES).

-0 solution(_24,_45)                              _106 is R1
-1 v(b,_106,_112)                                 _107 is R2
  +1 v(b,:(l,_139),f)                                 .
-1 p(b,_107,_113)                                     .
  +1 p(b,:(h,_147),f)                                 .
-1 n(:(h,_147),b,_114)
  +1 n(:(h,_147),b,f)                             _112 is E1
-1 v(_108,b,_115)                                 _113 is E2
  +1 v(:(r,_162),b,f)                                 .
-1 p(:(r,_162),b,_116)                                .
  +1 p(:(r,_162),b,f)                                 .
-1 n(b,:(l,_139),_117)
  +1 n(b,:(l,_139),f)
-1 n(:(l,_139),:(h,_147),_118)
  +1 n(:(l,_139),:(h,_139),+)
-1 p(:(h,_139),:(r,_162),_119)
  +1 p(:(h,_139),:(r,_139),+)
-1 v(:(l,_139),:(r,_139),_120)
  +1 v(:(l,_139),:(r,_139),+)
-1 n(:(l,_139),:(h,_139),_121)
  +1 n(:(l,_139),:(h,_139),+)
```

```
-1 v(_111,:(l,_139),_122)                         <----------------+
  +1 v(:(r,_139),:(l,_139),-)                                      |
-1 v(_109,:(l,_139),_123)                         <----+  <----+   |
  +1 v(:(r,_139),:(l,_139),-)                          |       |   |
-1 n(:(l,_139),:(r,_139),_124)  *1 --------------------+       |   |
  +1 v(_109,:(l,_139),f)                                       |   |
-1 n(:(l,_139),_109,_124)                                      |   |
  +1 n(:(l,_139),:(h,_139),+)   [1]               <------+     |   |
-1 v(:(l,_139),_110,_125)                         <----+ |     |   |
  +1 v(:(l,_139),:(r,_139),+)                          | |     |   |
-1 p(:(h,_139),:(r,_139),_126)                         | |     |   |
  +1 p(:(h,_139),:(r,_139),+)                          | |     |   |
-1 n(:(r,_139),:(h,_139),_127)  *2 --------------------+ |     |   |
  +1 v(:(l,_139),:(l,_237),f)                            |     |   |
-1 p(:(h,_139),:(l,_237),_126)  *2a ---------------------+     |   |
  +1 n(:(l,_139),:(l,_230),f)                                  |   |
-1 v(:(l,_139),_110,_125)                         <----+       |   |
  +1 v(:(l,_139),:(r,_139),+)                          |       |   |
-1 p(:(h,_139),:(r,_139),_126)                         |       |   |
  +1 p(:(h,_139),:(r,_139),+)                          |       |   |
-1 n(:(r,_139),:(h,_139),_127)  *2 --------------------+       |   |
  +1 v(:(l,_139),:(l,_238),f)                                  |   |
-1 p(:(h,_139),:(l,_238),_126)  *2a ---------------------------+   |
  +1 v(:(r,_221),:(l,_139),f)                                      |
-1 n(:(l,_139),:(r,_221),_124)  *1 --------------------------------+
  +1 v(_111,:(l,_139),f)
-1 v(_109,:(l,_139),_123)                         <----+
  +1 v(:(r,_139),:(l,_139),-)                          |
-1 n(:(l,_139),:(r,_139),_124)  *1 --------------------+
  +1 v(_109,:(l,_139),f)
-1 n(:(l,_139),_109,_124)
  +1 n(:(l,_139),:(h,_139),+)   [1]
-1 v(:(l,_139),_110,_125)
  +1 v(:(l,_139),:(r,_139),+)
-1 p(:(h,_139),:(r,_139),_126)
  +1 p(:(h,_139),:(r,_139),+)   [2a]
-1 n(_111,:(h,_139),_127)
  +1 n(:(l,_139),:(h,_139),+)   [2]
-1 p(:(l,_139),:(h,_139),_128)
  +1 p(:(l,_139),:(h,_139),f)
-1 v(:(r,_139),:(l,_139),_129)
  +1 v(:(r,_139),:(l,_139),-)
-1 p(:(r,_139),:(h,_139),_130)
  +1 p(:(r,_139),:(h,_139),-)
-1 n(:(h,_139),:(l,_139),_131)
  +1 n(:(h,_139),:(l,_139),-)
  +0 solution(;(:(l,_139),;(:(h,_139),;(:(r,_139),;(:(h,_139),
              ;(:(r,_139),:(l,_139)))))),
              ;(f,;(f,;(f,;(f,;(f,;(f,;(+,;(+,;(+,;(+,;(f,;(f,
              ;(+,;(+,;(+,;(+,;(f,;(-,;(-,-))))))))))))))))))))

RS = (l:_139);(h:_139);(r:_139);(h:_139);(r:_139);l:_139,
ES = f;f;f;f;f;f;+;+;+;+;f;f;+;+;+;+;f;-;-;-
```

(In the original, "-1 n(:(l,_139),:(r,_139),_124)" lines marked with a bracket arrow on the left point to the following "+1 v(_109,:(l,_139),f)"; the first "-1 n(:(r,_139),:(h,_139),_127) *2" has a left bracket arrow pointing to "+1 v(_111,:(l,_139),f)".)

7.4 Non-attacking chessboard queens example

Our next example is by now a classic of backtrack studies [Dijkstra 1972] [Gaschnig 1977] [Bruynooghe 1978]. We shall again make recourse of it when exemplifying sidetracking techniques, as [Clark et al. 1979] have done.

The problem consists in placing n chess queens on a n*n chessboard such that no two queens attack one another. For the purpose of exemplification we restrict ourselves to the case of four queens. Our program is general though. Only the input changes according to the number of queens. The input for four queens is

```
perm(4.3.2.1.nil,L), pair(4.3.2.1.nil,L,Q), safe(Q)
```

The position of a queen is specified by a pair of coordinates (row,column), and a complete board configuration is a list of such pairs. Predicate procedure perm generates a permutation of four row positions for the four queens, thereby ensuring that no two queens are on the same row. Predicate procedure pair pairs each element of the list L of row positions with a different column number, thereby ensuring that no two queens are on the same row. Predicate procedure safe inspects whether in the list of pairs Q any two queens are on the same diagonal.

The complete program is now presented.

```
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(nil).
safe(Q.R):- check(Q,R), safe(R).

check(Q,nil).
check(Q,R.S):- not_on_diagonal(Q,R), check(Q,S).

not_on_diagonal(p(R1,C1),p(R2,C2)):-
        minus(R1,R2,R),
        minus(C1,C2,C),
        C \== R,
        minus(R2,R1,NR),
        C \== NR.
```

A permutation U.V of X.Y is generated by taking some element U of X.Y as the first element, followed by a permutation V of the list W obtained from X.Y by deleting U. pair simply pairs successive elements of the list of permuted rows and the the fixed list of columns. safe checks each pair against all following pairs in the list of pairs. Thus any two pairs are checked not to be on one same diagonal. This is accomplished by subtracting corresponding coordinates (with system predicate minus), and making sure they differ in absolute value.

not_on_diagonal rejects the queenboard and causes backtracking when it finds two queens on the same diagonal. Standard backtracking would generate the next permutation, even though it may not alter the coordinates of the conflicting queens. Intelligent backtracking will return to the point in the permutation generation where one of the conflicting queens is assigned a different row if possible.

For example, the first permutation being 4.3.2.1.nil, the first two pairs, p(4,4) and p(3,3), are the first to conflict. Standard backtracking would then permute the rows of the queens on columns 2 and 1 to no avail. Intelligent backtracking, on the other hand, begins by detecting that the conflicting constants C and R at C \== R were obtained at minus(C1,C2,C) and minus(R1,R2,R), and sets them up as backtrack nodes. Since there are no other alternatives for those calls they fail. It then detects the nodes on which the presence of C1, C2, R1 and R2 in those goals depend, because C and R are made to depend on them by the system predicate minus. Therefore all such nodes are admissible backtrack nodes, since a change in any of the constants may solve the conflict arisen. The parent node, not_on_diagonal, is one such node. Other nodes are the first two calls of pair, where the constants in the pairs p(4,4) and p(3,3) were transmitted to those pairs, respectively from the first and second arguments. The presence of the constants C1 and C2 in the first argument of the two first calls to pair depends only on their ancestors up to the input goal pair(4.3.2.1.nil,L,Q). No backtrack nodes are generated by that dependency, since such dependency on ancestors is implicit in the dependency on pair already established. The presence of R1 and R2 in the second argument of the two first calls to pair depends on the two first calls to delete, where that argument was transmitted from the second to the third argument of the first clause for delete. The presence of R1 and R2 in the second argument of the first two calls to delete depends on all their ancestors up to the input goal perm(4.3.2.1.nil,L), where the two constants originated.

To sum up, the successive backtrack nodes generated upon this failure of C \== R are : first, minus(4,3,1), minus(4,3,1) and the parent not_on_diagonal(p(4,4),p(3,3)) ; next, on failure of the second minus(4,3,1), the second call to pair, pair(3.2.1.nil,3.2.1.nil,W), and the second call to delete, delete(U,3.2.1.nil,W); next, on failure of the first minus(4,3,1), the first call to pair, pair(4.3.2.1.nil,4.3.2.1.nil,Q), and the first call to delete, delete(U,4.3.2.1.nil,W).

Finally, everytime one of these backtrack goals fails because there are no more clauses for it, its parent node is included as a backtrack node, and an analysis takes place to find reasons for any failure of the goal in matching the clauses. Thus, when not_on_diagonal fails, there being no failure in entering its single clause, only its parent, check, is included. But check also fails, so an analysis takes place to examine why its first clause failed to match the goal. The reason is the conflict between the principal functor "." in the binding of R and nil. check could perhaps be solved if "." changed. A backtrack node for that change would be the point where that principal functor was obtained. It turns out to be the second call to pair, already retained as a backtrack node. The fact is that all constant components of p(4,4).W were obtained in the match of the second call of pair. Similarly, analysis of the failure of safe, ie. the conflict between the binding of Q and nil, singles out the first (input) call of pair as the culprit, which was already retained as a backtrack node.

The reader is advised to draw up the AND/OR tree generated by the execution of the top goal up to the first failure of C \== R and to follow our argument thoroughly.

7.5 Generating and parsing grammars example

Our next example concerns two grammars, S and P. The problem is to find some string common to the two grammars. We assign to S the role of generating some string, and to P the role of parsing it. The strings generated by S are either the null string or one or more a's followed by one or more b's, followed by one or more a's, etc. The strings accepted by P are composed of alternating pairs of a's and pairs of b's, starting with a pair of a's.

The two grammars may be conveniently expressed in Prolog's grammar rules notation [Colmerauer 1975] [Pereira et al. 1978a] [Pereira,FCN et al. 1978] [Pereira,FCN 1979].

Grammar S

```
s --> [].
s --> sa, sb, s.

sa --> [a].
sa --> [a], sa.

sb --> [b].
sb --> [b], sb.
```

Grammar P

```
pab --> [a,a], pb.

pb --> [b,b], pab.
```

The first of the two rules for non-terminal sa, for instance,

expresses that sa can be parsed as the terminal a, followed by a parsing of sa. In the other rules, [] expresses an empty sequence of terminals, and [b,b] a sequence of two b's.

Now, in fact, grammar rule notation is merely a convenient 'syntactic sugar' for equivalent ordinary Prolog clauses expressing the same thing. Each grammar rule takes an input string (or variable), parses (or generates) some initial portion, and produces the remaining portion as output for further analysis (or generation). The arguments required for the input and output are not written explicitly in a grammar rule, but the syntax implicitly defines them.

For the purpose of this exposition, we work directly with the equivalent Prolog clauses, where those two extra arguments are made explicit. The above clauses become

Grammar S

```
s(L,L).
s(L1,L4):- sa(L1,L2), sb(L2,L3), s(L3,L4).

sa(a.L,L).
sa(a.L1,L2):- sa(L1,L2).

sb(b.X,X).
sb(b.X1,X2):- sb(X1,X2).
```

Grammar P

```
pab(a.a.L1,L2):- pb(L1,L2).

pb(b.b.L1,L2):- pab(L1,L2).
```

while the input is   s(S,nil), pab(S,nil) .

In the sequel we show that while standard backtracking is caught in the recursive loop of s, intelligent backtracking appropriately jumps over it, and the loop of sb, to reach sa. Standard backtracking will cause the successive generation of the strings

nil a.b.nil a.b.a.b.nil ... a.b...a.b.nil

but parsing will never advance beyond the first a. On the other hand, intelligent backtracking successively generates the strings

nil a.b.nil a.b.b.nil a.a.b.nil a.a.b.a.b.nil a.a.b.a.a.b.nil a.a.b.b.nil a.a.b.b.a.b.nil...

and parsing will proceed as generation advances.

Let us see how this happens. After rejection by P of the null string (represented by nil), a simple backtracking to the previous goal takes place. The string a.b.nil is then generated. A failure arises next when pab is called. The conflict is between the b in the input string and the second a in the clause head. So the binding of S must change. Where did S acquire the constant b ? Now S was bound to L1 in the second clause for s, and L1 was subsequently bound to a.L, when the call sa(L1,L2) matched the first clause for sa. So access to b is made via this L. This L is made to depend on L2 in the same call even though L2 has not yet acquired a value. It does obtain a value in the next call, which binds it to b.X. Consequently, the backtrack nodes generated by the conflict are the two above mentioned calls (there is no parent). The next string generated is a.b.b.nil. The same conflict occurs, but now there are no more clauses for sb at the first backtrack node. So intelligent backtracking resorts to the next

backtrack node, the second clause for sa is activated, and the new string, a.a.b.nil is produced. The next conflict occurs between b and nil in the call to pb. nil was obtained at the first clause for s. After backtracking, the string a.a.b.a.b.nil is produced, and subsequently a.a.b.a.a.b.nil and a.a.b.b.nil.

We do not further here our analysis of backtracking, for these and all subsequent failures give rise to similar dependency analyses.

8. Sidetracking in logic programs

> 'In that which I now propose, we will discard the interior points [...], and concentrate our attention upon its outskirts. Not the least unusual error in investigations such as these is the limiting of enquiry to the immediate, with total disregard of the collateral or circumstancial events.'
>
> E.A.Poe in 'The mistery of Marie Roget - a sequel to The murders in the Rue Morgue', p.435

We will now be concerned with the presentation of a strategy for the execution of logic programs in clausal form which is essentially dynamic, since it relies upon runtime evaluation of the deterministic or non-deterministic character of the goals to be solved.

Any problem is viewed, at any stage in its execution, as a conjunction of single goals. A disjunction of goals appearing in the program must be viewed as a single goal for which several matching clauses exist whose bodies are precisely the goals in each disjunct.

An execution step is the replacement of one of the single goals in the conjunction that represents the problem by the conjunction of goals (possibly none) which constitute the body of the clause activated by that goal. This activation may cause instantiation of variables of the goal also present in other pending goals, thus restricting the matching possibilities of those goals.

A goal is said to be non-deterministic, at a certain stage in the computation, if at that stage it can activate more than one clause; otherwise, it is said to be deterministic.

The main idea behind the strategy which we call sidetracking is to postpone, at any stage, the replacement of non-deterministic goals until all pending deterministic ones have been replaced. This requires a runtime evaluation of determinism at each execution step.

Proceeding this way has a double advantage:

First, doing all possible replacements of deterministic goals before activating non-deterministic ones avoids reactivation of those deterministic goals if backtracking takes place among the non-deterministic goals that follow. We might say: calculate the inevitable once, then try the options.

Second, if a failing deterministic path exists which results from a replacement of a goal then we are sure to spot that failure before any (useless) choices for non-deterministic goals are ever made.

Detailed description of operation:

Starting with the goal statement, and throughout the execution, we always have a conjunction of single goals which we view as forming a circular list, one of whose goals is considered to be the current goal - at the start it will be the leftmost goal in the initial goal statement.

An execution step is performed as follows:

We find if the current goal is deterministic; if not, the next goal in the list becomes the current one and is in turn examined for determinism. Two outcomes are possible for this search:

1 - The whole list has been searched and no deterministic goal was found.

The current goal (the one where the search began) is then replaced by the (possibly empty) body of the first clause it activates, while the next goal in the list becomes the current one.

2 - A deterministic goal was found.

If there is a clause activated by the goal, this one is replaced by the clause body (which may be empty), and the next goal in the list becomes the current goal.

If no clause can be activated by the goal we are in the presence of a failure, and backtracking must ensue, undoing replacements made in the list. Backtracking only makes sense if made to a goal for which alternative replacements do exist, i.e. a goal that was non-deterministic just before it was last replaced. This fact implies that backtracking (be it standard or intelligent) can be further enhanced by automatic skipping over chains of deterministic goals. The whole problem fails, as usual, if backtracking past the top goal statement is required.

A solution to the problem is found if the list becomes empty.

When, in both outcomes, we talked about the next goal in the list becoming the current one we meant the goal next to the last replacing goal. So execution of the conjunction of goals is performed in breadth-first manner, with priority of deterministic goals over non-deterministic ones.

Ideally, a goal should only be examined for determinism if that has not already been done with its arguments in the same state as the current one. So, re-examination of a goal should only occur if that goal has variables which have been instantiated after it was last examined. This can be carried out using the tagging of variables already described in the section on intelligent backtracking, and keeping a record, for each goal, of the last time it was examined for determinism.

To illustrate the dynamics of sidetracking we will return to the non-attacking chessboard queens example, in the formulation already described in section 7.4.

We can see that the first goal perm(4.3.2.1.nil,L) is deterministic, for it can activate only one clause. It is therefore replaced by del(U1,4.3.2.1.nil,W1),perm(W1,U1). Variable L is now instantiated to U1.V1 and so the conjunction of goals to solve becomes

del(U1,4.3.2.1.nil,W1),perm(W1,V1),pair(4.3.2.1.nil,U1.V1,Q),safe(Q)

The current goal is now pair(4.3.2.1.nil,U1.V1,Q), which is again deterministic, and is thus replaced by pair(3.2.1.nil,V1,Q1), with variable Q becoming instantiated to p(4,U1).Q1 .

The next step is the replacement of deterministic goal safe(p(4,U1).Q1) by the conjunction check(p(4,U1),Q1),safe(Q1) .

Having completed a cycle on the list, the current goal is now del(U1,4.3.2.1.nil,W1). This time we are in the presence of a non-deterministic goal, so we sidetrack and proceed to analyse our

next current goal, perm(W1,V1), which is also non-deterministic, and finally arrive at deterministic goal pair(3.2.1.nil,V1,Q1), that is replaced by pair(2.1.nil,V2,Q2), with V1 now instantiated to U2.V2 and Q1 to p(3,U2).Q2 .

If the reader cares to keep on simulating sidetracking execution of this problem s/he should arrive at a point when the only goals remaining are non-deterministic, and these are:

```
del(U1,4.3.2.1.nil,W1), del(U2,W1,W2), del(U3,W2,U4.nil),
minus(U1,U2,D12), D12 \== 1, D12 \== -1,
minus(U1,U3,D13), D13 \== 2, D13 \== -2,
minus(U1,U4,D14), D14 \== 3, D14 \== -3,
minus(U2,U3,D23), D23 \== 1, D23 \== -1,
minus(U2,U4,D24), D24 \== 2, D24 \== -2,
minus(U3,U4,D34), D34 \== 1, D34 \== -1
```

We are here assuming that goals of type minus(A,B,C) are regarded as non-deterministic unless A and B are instantiated, and similarly any goal A \== k is considered non-deterministic as long as A is uninstantiated. This problem will be elucidated in a subsequent section of this paper on sidetracking control.

del(U1,4.3.2.1.nil,W1) will be the first non-deterministic goal to be replaced, by the body of the first clause it activates, our list of goals becoming

```
del(U2,3.2.1.nil,W2), del(U3,W2,U4.nil),
minus(4,U2,D12), D12 \== 1, D12 \== -1,
        .
        .
        .
```

No other goal has become deterministic yet. We next replace goal del(U2,3.2.1.nil,W2) by the (empty) body of the first clause it activates, thus getting

```
del(U3,2.1.nil,U4.nil),
minus(4,3,D12), D12 \== 1, D12 \== -1,
        .
        .
        .
```

In the next execution step minus(4,3,D12) is detected as being deterministic and so is activated, instantiating D12 to 1; The following step is activation of 1 \== 1, now deterministic, which fails. Backtracking follows, to the more recently activated non-deterministic goal, del(U2,3.2.1.nil,W2), as is indeed logical.

It is not difficult now to picture the whole path of the computation. You may notice that not even intelligent backtracking is required in this particular example, for returning to the more recently activated non-deterministic goal is sufficient to get the correct backtrack point. Note, however, that in the general case intelligent backtracking can improve sidetracking execution.

9. Control constructs

> "but it will be strange, indeed, if a comprehensive survey, such as I propose, [....] will not afford us some minute points which shall establish a <u>direction</u> for enquiry."
>
> ibidem, p.436

9.1 Intelligent backtracking control

Two (unimplemented) control constructs are presented.

1) The first allows the user to specify, in case of failure of a goal, which of its brother goals should nevertheless be executed. The aim is to obtain better backtracking information, should some of them also fail.

Consider the next query, for the database of section 7.2.

```
query(S,P):- student(S,C1),     1
             student(S,C2),     2
             C1 \== C2,         3
             professor(P,C1),   4
             course(C1,D1,R),   5
             course(C2,D2,R),   6
             professor(P,C2).   7
```

If 6 fails it may be advantageous to execute 7 anyway. The backtrack stack for 6 is (2,5), while that for 7, in case it fails, is (2,4). The rule for composing stacks at AND's retains (2,4) for backtracking rather than (2,5). Put another way, it is no use looking for another room R at 5 if P does not teach C2 anyway.

Some notation for indicating such groups of brother goals is needed. We refrain however from proposing any, until the articulation of various forms of control constructs becomes clearer.

2) The second control construct is exhibited in the following program for sorting a list

```
sort(L,S):- perm(L,S), ordered(S).

ordered(nil).
ordered(X.nil).
ordered(X.Y.nil):- X@ =< Y , ordered(Y.Z).
```

where perm generates permutations as in section 7.4. The control construct consists in the suffix @ after variable X in the last clause. It indicates that on failure of X =< Y , only the backtrack nodes for changing term X should be used. The backtrack nodes for changing Y are ignored. In fact, it is no use changing Y without changing X, since Y would still be positioned after X in the permuted list.

9.2 Sidetracking control

Because mental simulation of sidetracking execution of a program is not at all easy, the programmer loses the ability to foresee the order in which certain goals will be executed. It is nevertheless very important sometimes to guarantee that a certain goal will only be executed after some other goal or goals have been totally or partially executed. This may be for efficiency reasons or because of system restrictions upon the execution of certain goals.

This kind of problem can be solved by the introduction of a control construct that associates a goal with a condition (normally upon its arguments) such that the system will skip activation of the goal (sidetrack over it) unless the condition is satisfied.

Take as an example this definition for the predicate grand_parent :

```
grand_parent(X,Z):- parent(X,Y), parent(Y,Z).
```

If we use it for getting the value of an X whose grandparent is a given Z, when parent(Y,Z) is non-deterministic, then parent(X,Y) will be activated first. This is highly inconvenient, for any ground clause parent(a,b) will match parent(X,Y) (X and Y are both uninstantiated), and there may be an enormous amount of them. It is really stupid to find for every parent Y in the whole population if it is a child of Z.

This can be remedied as outlined above, with a condition for the activation of parent(X,Y) :

```
grand_parent(X,Z):- ( nonvar(X) or nonvar(Y) : parent(X,Y) ),
                                                 parent(Y,Z).
```

Thus parent(X,Y) will not be activated unless X or Y are instantiated.

Another example would be the use of a goal such as X<Y when we are not able to guarantee that X and Y will have integer values when activation of the goal is tried. We could write instead

```
( integer(X) and integer(Y) : X<Y )
```

Conditions may be used for other conditions, as in

```
p:- C1: C2: C3: q.
```

A second construct is next presented. In the section on sidetracking, we said that a deterministic goal is one that activates at most one clause. Activation of a clause, however, should not be construed simply as the successful matching of the goal with the clause head. Additional conditions may be imposed. The motivation for doing so is that some tests on arguments of a goal cannot be performed by unification alone, but may be carried out within the body of the clause (take for example the distinction between odd and even values). Consequently, a goal may match several clause heads and yet be deterministic in the sense that on closer inspection only one clause may execute the goal.

What we need then is a refinement of the meaning of activation of a clause by a goal to include, besides unification with the clause head, the verification of a certain condition.

Consider the two clauses for p(X)

```
p(X):- odd(X), r(X).
p(X):- even(X), s(X).
```

If activation depends solely on matching, any goal p(a) will be considered non-deterministic, while it is obvious that only one clause may possibly solve the goal.

We will write such clauses using a notation which makes explicit the conditions for their activation.

```
p(X):- odd(X):- r(X).
p(X):- even(X):- s(X).
```

Note that the declarative semantics is preserved. In fact, we are simply using the equivalence between the forms

a <- b, c and (a <- b) <- c

It is essential that the evaluation of conditions, in both constructs, be performed so that no backtracking is allowed from outside into them, although it may take place freely during their evaluation.

The next example further illustrates this second control construct. It is a Horn clause program to compute the primes, in increasing order, according to the method known as "The sieve of Eratosthenes", taken from [Bruynooghe 1979]. The method itself consists in deleting from a list of the integers all the multiples of any integer greater than 1. We use predicate procedure sift(LI,LP) below to express the result of this activity on the list LI of integers greater than 1 to obtain the list LP of primes different from 1. The result L2 of the activity of deleting from a list L1 all the multiples of a given integer P is expressed by predicate procedure filter(P,L1,L2). Predicate procedure integer_list(N,L) relates an integer N to the list of integers starting with N. divide(P,N,B) is a system predicate that merely tests of two given integers whether N is a multiple of P is B (a boolean value). plus(X,Y,Z) is a system predicate that given X and Y produces Z=X+Y.

Let us now look at the program and the optional control we may impose on sidetracking in order to achieve a more efficient computation. Note, in passing, that Prolog's standard depth-first

```
H1:- B11 :- B12.

H2:- not(B11) :- B2.
```

Finally, when more than one "!" occur in the same body as in

```
H1 :- B11, !, B12, !, B13, !, B14.
H2:- B2.
```

we have

```
H1:- B11 :- B12: B13: B14.
H2:- not(B11) :- B2.
```

This notation accurately describes two different uses of "!".

10. Conclusions

> "Conclusions such as these open a wide field for speculation and exciting conjecture."
>
> E.A.Poe in "Narrative of A.Gordon Pym", last page.

Logic deals with what follows from what. The correctness of a piece of reasoning, it is found, does not depend on what the reasoning is about, so much as on how the reasoning is done ; on the pattern of relationships between the various constituent ideas rather than on the ideas themselves. We believe this holds as much for the forward deployment of reasoning as well as for the retracing of an unsuccessful line of argument away from inevitable repeated failure.

In fact, intelligent backtracking, as we have described it, relies only on the structure of derivations, annd the way it affects the objects therein concerned, so as to direct the recover from an unsuccessful reasoning step away from alternative derivations which are bound to fail at the same step for identical reasons. It is a general strategy of retrocess, that complements extant strategies for moving forward. Furthermore, it opens the way for research into more refined forms of backtracking, where for instance failed goals would pass back information regarding the admissible patterns for their arguments, gleaned from unsuccessful match attempts.

One other reasoning strategy was presented in this paper, for guiding the forward movement of search, called sidetracking. It is but an instance of the well known principle of procrastination, which advises postponement of the problematic until the inevitable has been

acomplished, or at least attempted, in the hope that the problematic might become more restricted in the proocess, or altogether avoided as a consequence of failure of the inevitable.

Specifically, in sidetracking any runtime-deterministic pending goal is replaced by the goals in the body of the clause it activates, before any runtime-non-deterministic goal is so replaced. A runtime deterministic goal is one that activates at most one clause. If no match is found for a goal backtracking takes place, of course, be it of the standard or the intelligent variety.

A final ingredient for improving search was put forward. It incorporates advice from the programmer, in the form of optional control constructs, for guiding program execution. Ideally, such constructs should not interfere with the program's declarative semantics, but this requirement may be relaxed to allow constructs which contract the original search space, at the user's own risk, so long as they don't extend it.

Our last remark is to the effect of stating that the paradigm of tagging objects in a derivational system with object specific information will prove to be fertile ground in the near future. In particular, in what regards the interplay of parallel communicating processes with backtracking, and in what regards the administration of hypothetical reasoning.

"I know not [...] what impression I may have made, so far, upon your own understanding ; but I do not hesitate to say that legitimate deductions even from this portion of the testimony [...] are in themselves sufficient to engender a suspicion which should give direction to all further progress in the investigation of the mystery."

E.A.Poe in "The murders in the Rue Morgue", p.396

11. Acknowledgements

This work was supported by the Instituto Nacional de Investigação Científica (INIC). We are grateful to the Divisão de Informática of the Laboratório Nacional de Engenharia Civil for the computation facilities provided.

12. References

[Bruynooghe 1978] Bruynooghe,M.

Intelligent backtracking for Horn clause logic programs
Colloquium on Mathematical Logic in Programming
Salgotarjan, Hungary.
(Proceedings to be published by North-Holland.)

[Bruynooghe 1979] Bruynooghe,M.

A control regime for Horn clause logic programs (draft)

[Clark et al. 1979] Clark,K.L.; McCabe,F.G.

The control facilities of IC-Prolog
Dept. of Computing and Control, Imperial College, London.

[Coelho et al. 1979] Coelho,H.; Cotta,J.C.; Pereira,L.M.

How to solve it with Prolog
Laboratorio Nacional de Engenharia Civil, Lisbon.

[Colmerauer 1975] Colmerauer,A.

Les grammaires de metamorphose
Groupe d'Intelligence Artificielle, Univ. Marseille-Luminy
Appears as "Metamorphosis Grammars" in "Natural Language
Communication with Computers", Springer-Verlag, 1978.

[Dijkstra 1972] Dijkstra,E.W.

Notes on structured programming
in "Structured Programming" (O.-J.Dahl et al. eds),Academic Press.

[Gaschnig 1977] Gaschnig,J.

A general backtrack algorithm that eliminates most redundant tests
5th Int. Joint Conf. on Artificial Intelligence (IJCAI-77),p 457.

[Kowalski 1974] Kowalski,R.A.

Predicate logic as a programming language
IFIP 74, pp 569-574, North-Holland Publ. Co.

[Kowalski 1979] Kowalski,R.A.

"Losic for problem solving"
North-Holland Publ. Co.

[Loveland 1978] Loveland,D.

"Automated theorem provins: a losical basis"
North-Holland Publ. Co.

[Pereira,FCN et al. 1978] Pereira,F.C.N.; Warren,D.H.D.

Definite clause grammars compared with
augmented transition networks
Dept. of Artificial Intellisence, Edimbursh University.

[Pereira,FCN 1979] Pereira,F.C.N.

Extraposition grammars
Dept. of Artificial Intellisence, Edimbursh University.

[Pereira et al. 1978a] Pereira,L.M.; Pereira,F.C.N.; Warren,D.H.D.

User's guide to DECsystem-10 Prolog
Laboratorio Nacional de Engenharia Civil, Lisbon.

[Pereira et al. 1978b] Pereira,L.M.; Monteiro,L.F.

The semantics of parallelism and co-routining
in logic programs
Colloquium on mathematical Logic in Programming,
Salgotarjan, Hungary.

[Pereira 1979] Pereira,L.M.

Backtracking intelligently in AND/OR trees
Universidade Nova de Lisboa, Lisbon.

[Poe 1908] Poe,E.A.

Tales of mystery and imagination
Everyman's Library, Dent, London 1971.

[Robinson 1965] Robinson,J.A.

A machine-oriented logic based on the resolution principle
JACM vol 12, pp 23-44.