A HISTORY-BASED SCHEME FOR ACCELERATING PROLOG INTERPRETATION

Vishv Mohan Malhotra

Discipline of Computer Science
The Flinders University of South Australia
Bedford Park 5042
AUSTRALIA.

Tang Van To

Division of Computer Science
Asian Institute of Technology
G.P.O. Box 2754, Bangkok 10501
THAILAND.

ABSTRACT

An algorithm to improve the performance of a Prolog interpreter is introduced. The algorithm, unlike the intelligent backtracking schemes which improve the performance by avoiding redundant redos, avoids redundant calls. The algorithm identifies the redundant calls by maintaining a history of the program execution. The algorithm can be used in conjunction with an intelligent backtracking scheme for a further speed-up of the programs.

1. INTRODUCTION

Most intelligent backtracking (IB) schemes for *Prolog*, reported in literature [1, 4, 6, 8, 10, 12], consist of two parts: (a) an algorithm to compute the set of suspects; and (b) an algorithm to choose the backtrack-points using a data-base of these sets. Much of the work has centered on the former algorithm (see also [2, 5, 9]). The emphasis has been to design algorithms to construct smaller sets of suspects at lower costs. It is the algorithm to compute the set of suspects that distinguishes one backtracking scheme from the other.

We believe that the IB schemes for choosing the backtrack-points do not make a full use of the available information. Specifically, after a backtrack, the interpreter resumes the search without avoiding the failed calls that have not *yet been attended* to. As a consequence, the interpreter repeats several failures in the resumed search. The algorithm reported in [8] is expected to reduce the search space for 6-queens problem (clever solution) [1] by about 10%. It is shown later in this paper that the reduction can exceed 35% through a better use of the information.

In this paper, we introduce an algorithm to improve the performance of a Prolog interpreter by reducing thrashing. The algorithm achieves this improvement by avoiding calls which begin searches that are destined to eventually fail. An IB scheme, on the other hand, controls the thrashing by selecting the appropriate goals for redo. The two schemes affect the programs differently; there may be redundant calls that an IB scheme fails to eliminate. On the other hand, the scheme described in this paper may not skip some unproductive redos. Thus, the two methods can be viewed as complimenting each other. Indeed, many data-structures and procedures are common in the two schemes and can be shared.

In the *next section*, we illustrate, with an example, the calls that can be avoided to reduce the search space. Section 2 also introduces additional terminology that supplements the common Prolog terminology [3, 7]. The algorithm is introduced in Section 3. Section 4 presents the statistics obtained by running certain benchmarks using the algorithm. In the final section, the paper is concluded with some remarks about the possible extensions of the scheme.

2. PRELIMINARIES: EXAMPLE AND TERMINOLOGY

Consider the following Prolog program and its execution trace under an IB scheme:

```
    ? <-- r(A), r(B), s(A), t(B).
R1: r(1).
R2: r(2).
S1: s(C) <-- t(C).
S2: s(1).
T1: t(2).
```

Execution trace under an IB:

```
 1: goal r(A) calls clause R1. A unifies with 1.
 2: goal r(B) calls clause R1. B unifies with 1.
 3: goal s(A) calls clause S1. C unifies with 1.
 4: goal t(C) fails. Redo s(A).
 5: goal s(A) calls clause S2.
 6: goal t(B) fails. Redo r(B).
 7: goal r(B) calls clause R2. B unifies with 2.
 8: goal s(A) calls clause S1. C unifies with 1.
 9: goal t(C) fails. Redo s(A).
10: goal s(A) calls clause S2.
11: goal t(B) calls clause T1.
```

1984/89/0000/0482$01.00 © 1989 IEEE

Steps (8) and (9), in the above trace, are redundant as they repeat steps (3) and (4) under the same bindings. We call a search redundant if it is known, from the past behavior of the interpreter, that the search will end unsuccessfully. The scheme that we describe, in this paper, aims at maintaining the execution history of the interpreter in a form that can be used readily to identify and avoid repeated execution of the redundant calls.

To detect the redundant calls, the execution history of a program is maintained as an AND-OR tree. Indeed, we abstract the actions of a Prolog interpreter as a traversal (and construction) over this tree. In a history tree a goal is represented by an OR node. The children of a goal-node represent the clauses in the goal procedure. A clause is represented by an AND node with its children representing the goals (literals) in the body of the clause. A special goal-node -- we call it G0 -- constitutes the root of the tree. A clause-node, C0, is attached as the only child of G0. The user query defines the child goal-nodes of C0. The children of the goal-nodes are ordered by the sequencing of the clauses in the program. Similarly, the children of a clause-node are ordered by the literal ordering in the body of the clause.

For a goal-node, G, we use the term parent clause to refer to its parent node in the tree; the parent goal refers to the parent of the parent clause; the sibling of a goal is its right sibling -- a sibling shares the parent clause with the goal and is defined by the literal in the defining clause that immediately follows the literal defining node G. An analogous terminology is used for the clause-nodes. For a clause-node, its sibling is the node sharing the parent goal and is defined by the next clause in the goal procedure.

2.1 A PROLOG INTERPRETER

Initially, a tree consists of nodes G0, C0 and the goal-nodes representing the user query. The interpreter starts the execution by visiting the leftmost child goal-node of C0 and invoking a call for the goal.

CALL

During a call, the interpreter executes the goal that it is currently visiting by unifying the goal with the head of a clause. As a clause is selected, an AND node representing the clause is inserted as a child of the goal-node if it is not already there. (It simplifies the presentation of algorithms, in the later sections, if the interpreter also inserts a dummy node representing the sibling of the clause. No dummy sibling is needed for the last clause in the goal procedure.) The goals (literals) in the body of the clause are attached to the clause-node as its child goal-nodes. After a successful call -- a call is successful if a unifying clause is found -- the interpreter traverses to the leftmost child (goal-node) of the unifying clause. A call is then invoked for the new goal.

EXIT

The interpreter exits a goal-node if no descendent of the goal remains to invoke a call. After an exit from a goal-node the interpreter traverses to its sibling and invokes a call.

REDO

If the interpreter fails to execute a goal, it traverses back, in the reverse order of call invocations, to a previously executed goal and invokes a redo. An attempt is made to execute the goal using a different clause. After a successful redo the interpreter traverses to the leftmost goal-node in the unifying clause and invokes a call for the new goal. A goal that has unified with a head, either during a call or redo, is called an executed goal. The set of executed goal-nodes and their unifying clause-nodes is called a search.

FAIL

A goal is said to have failed if it can not unify with the head of any clause during a call or a redo step. The interpreter traverses (backtracks) to an executed goal-node, as described earlier, and performs a redo. As the interpreter backtracks it undoes the effects of the executed calls and redos.

THE MOST RECENT SUSPECT

When a goal G fails to unify with the head, H, of a clause an IB interpreter computes a set, S(G,H), of the executed goals that are suspected to be contributing bindings impeding the unification. Some IB schemes (e.g., [6]) may do so only when G fails. For a set S(G,H), let L(G,H) denote the most recently executed goal in the set. Clearly, it is unnecessary to try to execute goal G using H until the interpreter has backtracked to L(G,H). It is only after the interpreter has backtracked to goal L(G,H) that the bindings in G may change to let G unify with H.

In next section, we introduce an algorithm to control the execution of an interpreter to avoid the redundant calls.

3. AN ALGORITHM TO AVOID REDUNDANT CALLS

The algorithm for identifying the redundant calls has two main components. One component remembers for each clause-node that has been removed from a search, the most recently executed goal among the goals suspected to have caused the removal. A clause-node that was tried but failed to be a part of a successful search is not used again as a candidate clause for executing the parent goal until the interpreter has backtracked to this suspect since the clause was removed from the search. The extent of the backtracking done by the interpreter between two consecutive visits of a node is determined by the other component of the algorithm. The algorithms for these components are introduced in the following subsections.

3.1 REMEMBERING THE MOST RECENT SUSPECT

The algorithm remembers the most recently executed suspect for a clause-node by assigning tag, called L tag, to the node. An L tag of a node specifies the most recent goal to which the interpreter must backtrack before executing a call by including the node in the search. An algorithm to determine L tags for the nodes in the history tree, as the interpreter traverses over it, is described in the following paragraph. The algorithm also assigns L tags to the goal-nodes in the tree. The tags on the goal-nodes enable the interpreter to compute L tags for the clause-nodes as it traverses over the tree.

A clause-node may be removed from a search for three reasons:

(i) The clause fails to unify with the goal invoking the call or redo. In this case L(G,H), where G is the goal and H the head of the clause, is assigned as L tag with the clause-node.

(ii) The child goals of the clause-node fail. In this case L tag of the clause-node is computed from the L tags of the its child goal-nodes. For the clause-node its L tag is the earliest executed goal that appears as L tag on a child goal-node. The clause can not be in a successful search unless all its child goals can execute successfully. For a goal-node, L tag is determined by the L tags of its child clause-nodes. A goal can be executed using any one of the clauses in its procedure. Thus, L tag of a goal-node is the most recently executed goal appearing as a L tag on its child clause-nodes. Or,

(iii) A redo step is executed for the parent goal of the clause-node. The clause-node remains a potential candidate for inclusion in a search and is used as a candidate clause if the parent goal invokes a new call.

The L tags for the nodes are computed and assigned to the nodes as the interpreter backtracks from a goal to another goal to invoke a redo. The tag for a clause-node is computed when the interpreter backtracks from the leftmost goal in the clause to the parent goal.

3.2 DETERMINING THE EFFECTIVE BACKTRACKING

The other major component of the scheme is an algorithm to determine, as the interpreter visits the goal-nodes and invokes calls, the earliest goal to which the interpreter has backtracked since the previous visit of the node. This information is compared with the L tag on a clause-node to determine if the interpreter should try to execute the goal using the clause or not. The search involving a clause-node will end unsuccessfully if the interpreter has not backtracked to the goal specified by the L tag of the clause-node. The idea behind the algorithm is described below:

Imagine that each goal has a characteristic color. A goal executed earlier has a darker hue than the one executed later -- a darker hue can be painted over a lighter hue but not the visa versa. The interpreter picks the color of the goal when it performs a redo. It spreads this color by painting the sibling and the child nodes as it visits the nodes in its traversal over the tree. As the interpreter backtracks and invokes more redos, several balloons of these colors accumulate. It is essential that these colors be spread ahead of the interpreter over the history-tree. If a darker hue overtakes a lighter hue the interpreter may economize its efforts and discard the lighter hue. The color of a goal-node, when the interpreter invokes a call for the goal, is determined by the darkest hue that is painted on the node. Thus, if a goal-node is set colorless when it fails, the color of the node will determine the earliest goal to which the interpreter has backtracked since it failed. For certain other considerations, the goal-nodes are set colorless after the interpreter has invoked a call for the goal. The color is, however, remembered by assigning the same to one of its child clause-nodes.

3.3 THE ALGORITHM

Let G be a goal that has just failed. Let L be its L tag. As every goal in a search must execute successfully, no search that involves G can be successful unless the interpreter backtracks to L. The interpreter may, therefore, directly invoke a redo for L without exploring the intervening goals for a solution. There, however, may be an alternate search that does not involve goal G. The interpreter may begin this search by performing a redo for the parent goal of G. To systematically explore all searches the algorithm directs the

INITIALIZATION:

1. Initialize the history tree by creating a goal-node G0 with child clause C0. Insert goals in the user query as children of C0. Let G1 be the leftmost child. Set C0.C = G0. For each child Gi of C0 set Gi.C = no_color, and Gi.L = nil.
2. Set GOAL = G1. Set COLOR = G0.
3. Goto CALL.

CALL:

1. Set GOAL.C = no_color; set GOAL.L = nil.
2. (Paint the sibling) If GOAL has a sibling and the call for COLOR was invoked before the call for SIBLING.C then set SIBLING.C = COLOR. This assignment is also done if SIBLING.C was no_color.
3. Goto GET_UNIFYING_CLAUSE.

REDO:

(If this algorithm is used with an IB scheme, the goal chosen for redo by the IB scheme might have invoked the call before the call by GOAL. In this case set GOAL = the goal chosen by the IB scheme.)

1. Set COLOR = GOAL.
2. (Paint the sibling) If GOAL has a sibling and the call for COLOR was invoked before the call for SIBLING.C then set SIBLING.C = COLOR.
3. Goto GET_UNIFYING_CLAUSE.

GET_UNIFYING_CLAUSE:

1. Choose the next clause in the procedure for GOAL. If no clause remains then goto FAIL. Otherwise, let CLAUSE be the chosen clause. Let HEAD be its head.
2. (Insert in the tree) If CLAUSE is not a child of GOAL, insert CLAUSE as a child of GOAL. A dummy clause-node representing the sibling of CLAUSE is also inserted as a child of GOAL if CLAUSE is not the last clause in the goal procedure. Insert child goal-nodes of CLAUSE. A newly inserted node in the tree has its C tag set to no_color and L tag set to nil.
3. If the call for CLAUSE.C was invoked before the call for COLOR then set COLOR = CLAUSE.C.
4. Set CLAUSE.C = no_color. (Paint the sibling) If CLAUSE has a sibling and the call for COLOR was invoked before the call for SIBLING.C then set SIBLING.C = COLOR.
5. (Does CLAUSE begin a redundant search?) If the call for CLAUSE.L was invoked before the call for COLOR then goback to step 1.
6. If GOAL does not unify with HEAD then set CLAUSE.L = L(GOAL,HEAD) and goback to step 1.
7. (CLAUSE has unified with GOAL) Set CLAUSE.L = nil. If CLAUSE has no child goal then goto EXIT.
8. (Traverse to the the leftmost goal in the unifying clause) Select leftmost goal in CLAUSE as the next GOAL and goto CALL.

EXIT:

1. If GOAL has a sibling then set COLOR = SIBLING.C, select SIBLING as the next GOAL and goto CALL.
2. Otherwise, set parent goal of GOAL as the new GOAL and goto EXIT.

FAIL:

1. Compute and assign L tag to GOAL. This tag is chosen from the L tags of the child clause-nodes of GOAL by selecting the one that has invoked its call most recently. If only nil and GOAL appear as L tags on the child clause-nodes of GOAL then L tag assigned to GOAL is the goal that was executed just before GOAL.
2. If L, computed in step 1, is a goal that had its call invoked after the call by the parent goal of GOAL then set L as the new GOAL and goto REDO.
3. Otherwise, compute and assign L tag to the parent clause of GOAL. This tag is the goal that invoked its call earliest among those that appear as a L tag on the children of the clause-node. After assigning the L tag to the clause-node, clear the L tags (set them to nil value) of all children of the clause-node.
4. Set GOAL = the parent goal of GOAL and goto REDO.

Algorithm 1: An augmenting algorithm to control a Prolog interpreter.

interpreter, when G fails, to goal-node L, if the call for L was invoked after the call for the parent of G. Otherwise, the interpreter is directed to invoke a redo for the parent of G.

A complete description of the algorithm is given in Algorithm 1. The algorithm consists of six procedures: Initialization, Call, Redo, Get-unifying-clause, Exit and Fail. The control is passed between these procedures through goto statements. Tag C on a node specifies the color (goal) of the node. The algorithm uses a global variable, COLOR, to spread colors from the goals to their child goal-nodes. Execution trace of the algorithm when executing the example program is shown in Fig. 1 and Fig. 2.

```
0. Assume that initially tree (Fig. 2) has  nodes
   G0, C0, and G1 thru' G4.  Let
       COLOR = G0; GOAL = G1; C0.C = G0;
       Gi.C = no_color, for i := 1 thru' 4; and
       Gi.L = nil, for i := 1 thru' 4.

1. CALL G1: (1) G1.C := no_color; G1.L := nil;
       (2) G2.C := G0.
   GET_UNIFYING_CLAUSE: (1) CLAUSE := C1.
       (2) Insert C1 and (dummy) C2  as  children
           of G1.
       (4) C2.C := G0.

2. EXIT G1: (1) GOAL := G2.

3. CALL G2: (1) G2.C := no_color; G2.L := nil.
       (2) G3.C := G0.
   GET_UNIFYING_CLAUSE: (1) CLAUSE := C3.
       (2) Insert C3 and (dummy) C4  as  children
           of G2.
       (4) C4.C := G0.

4. EXIT G2: (1) GOAL := G3.

5. CALL G3: (1) G3.C := no_color; G3.L := nil.
       (2) G4.C := G0.
   GET_UNIFYING_CLAUSE: (1) CLAUSE := C5.
       (2) Insert C5 and (dummy) C6  as  children
           of G3. Insert G5 as a child of C5.
       (4) C6.C := G0.
       (8) GOAL := G5.

6. CALL G5: (1) G5.C := no_color; G5.L := nil.
   GET_UNIFYING_CLAUSE: (1) CLAUSE := C8.
       (2) Insert C8 as a child of G5.
       (6) C8.L := G1.

7. FAIL G5: (1) G5.L := G1.
       (3) C5.L := G1; G5.L := nil.
       (4) GOAL := G3.

8. REDO G3: (1) COLOR := G3.
   GET_UNIFYING_CLAUSE: (1) CLAUSE := C6.
       (3) COLOR := G0;
       (4) C6.C := no_color.

9. EXIT G3: (1) GOAL := G4.

10. CALL G4: (1) G4.C := no_color; G4.L := nil.
   GET_UNIFYING_CLAUSE: (1) CLAUSE := C7.
       (2) Insert C7 as a child of G4.
       (4) C7.C := no_color.
       (6) C7.L := G2.

11. FAIL G4: (1) G4.L := G2.
       (2) GOAL := G2.

12. REDO G2: (1) COLOR := G2.
       (2) G3.C := G2.
   GET_UNIFYING_CLAUSE: (1) CLAUSE := C4.
       (3) COLOR := G0;
       (4) C4.C := no_color.

13. EXIT G2: (1) COLOR := G2; GOAL := G3.

14. CALL G3: (1) G3.C := no_color; G3.L := nil.
             (2) G4.C := G2.
   GET_UNIFYING_CLAUSE: (1) CLAUSE := C5.
       (4) C6.C := G2.
       (5) goback to step 1.
       (1) CLAUSE := C6.
       (4) C6.C := no_color.

15. EXIT G3. (1) COLOR := G2; GOAL := G4.

16. CALL G4: (1) G4.C := no_color; G4.L := nil.
   GET_UNIFYING_CLAUSE: (1) CLAUSE :=c7.
       (4) C7.C := no_color.

17. EXIT G4: (1) GOAL := G0.

18. EXIT G0: DONE.
```

Fig. 1: Execution trace of Algorithm 1 for the example Prolog program. The numbers in the parenthesis refer to step numbers in Algorithm 1. For brevity sake, the steps that do not change values or affect the control-flow are not shown in the trace.

4. PERFORMANCE

We have implemented the scheme on a naive interpreter. An algorithm based on the scheme introduced in [8], is used to compute L(G,H). The algorithm returns a goal, contributing some binding to G, to which the interpreter must necessarily backtrack to remove a cause of the non-unifiability of goal G and head H. The algorithm (see Algorithm 2), though less selective than the one described in [8], is closer to the algorithm implicit in [6] to construct the suspect sets. Thus, it will allow us to draw meaningful conclusions about the effectiveness of the scheme by comparing the execution statistics. The algorithms used in [1] and [4] for constructing the sets of suspects are, however, more selective than the one implicit in our choice.

The statistics collected during the experiments are summarized in Table 1. The columns in the table show the data about the search space, the execution time and the size of the history tree. The search space is defined as the total number of attempted unifications, successful or otherwise, between the goals and the heads of the clauses.

Table 2 compares the speed-ups and the reductions in the search space with some IB schemes. The speed-ups mentioned under scheme KL are based on the number of Warren Abstract Machine (WAM [11]) cycles [6].

TABLE 1: A comparison between an interpreter using the proposed scheme and a naive interpreter.

PROBLEM & REFERENCE	SEARCH SPACE			EXECUTION TIME			HISTORY TREE SIZE
	Naive	Proposed	Gain %	Naive Secs.	Proposed Secs.	Speed-up X	Nodes
Naive sort [3]							
7 elements	40006	2242	94	44.5	3.6	12.4	153
8 elements	472845	9996	98	527	16.0	33	192
Circuit design [6]							
First solution	7003	2287	67	10.6	5.8	1.8	1676
Four solutions	7408	2427	67	11.1	6.0	1.9	1699
Map colour -- good order [1]							
First solution	320	260	18	0.3	0.3	~1	240
Seven solutions	1198	768	36	0.7	0.7	~1	768
Map colour -- bad order [1]							
First solution	1139248	423	99	684.5	0.4	>1500	268
Seven solutions	1377691	7337	99	826.0	8.7	95	365
6-queens -- simple solution [1]							
First solution	10450	7081	32	13.7	13.6	1.0	357
All solutions	42491	29563	37	55.9	57.1	1.0	375
6-queens -- clever solution [1]							
First solution	1868	1157	38	3.2	3.6	0.9	240
All solutions	10345	6244	40	17.7	19.9	0.9	258
6-queens -- clever solution [1] with calls in Noattack rearranged							
First solution	1084	765	29	2.2	2.7	0.8	240
All solutions	5821	3982	32	11.7	14.3	0.8	258
Database query [1]							
First solution	519	277	46	0.3	0.3	~1	55
Four solutions	759	472	37	0.3	0.3	~1	58
Quick sort							
10 elements	95	95	0	0.16	0.19	0.9	192
15 elements	151	151	0	0.30	0.32	0.9	315
Binary tree [1]	187	187	0	0.38	0.41	0.9	425

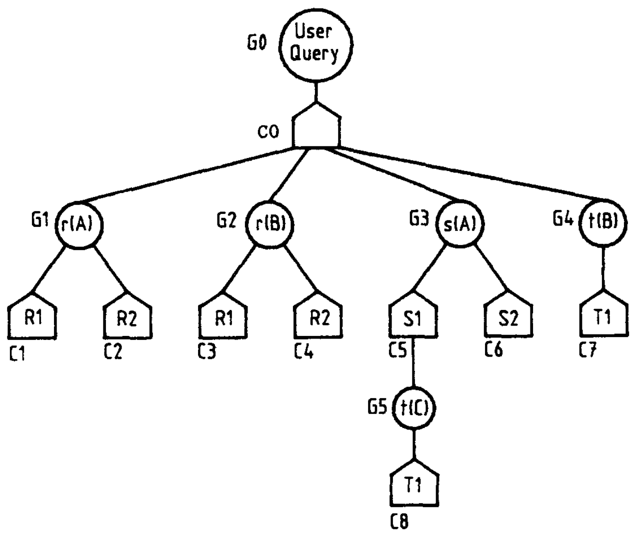

Fig. 2: The history tree for the example Prolog program.

5. CONCLUSIONS

Comparisons show that the scheme results in considerable reduction in the search space. The improvement in execution time, though moderate, is significant. Indeed, most of the overheads -- e.g., maintenance of the tags and construction of the set of suspects -- are common with the IB schemes. As a result, we expect a good reduction in the execution time if the algorithm is used along with an IB interpreter. The more selective is an IB scheme in choosing the backtrack points, the better is the expected improvement from the combined scheme. A better IB scheme is expected to create a larger number of paths over which the interpreter may make redundant searches. Indeed, the algorithm benefits the IB scheme too. Smaller number of paths over which the interpreter makes searches implies that the smaller will be the database of sets of suspects. This, in turn, will reduce the number of spurious backtrack-points -- the backtrack-points that do not result in a successful search.

The scheme can be tailored to trade search space for storage. A goal that has completed an exit step can be deleted, along with its descendents, to recover storage. The price, we pay, is the redundant search that may result when the interpreter returns to a goal that has been removed from the history tree. Perhaps, the compromise lies in the use of a heuristic to choose the goal-node for reclaiming the storage. We experimented with a heuristic that reclaims storage by removing the nodes with no descendent goal-nodes. Even this simple heuristic promises a graceful tradeoff between the search space and the storage for the history tree.

ACKNOWLEDGEMENTS

This work was done when the first author was at AIT. The work was also supported by AIT Project Initiation Grant, 1988-89. The Master's program of the second author was supported by an Australian government scholarship. Graeme Port and Kanchana Kanchanasut helped the authors in improving the clarity and focus of the paper.

REFERENCES

1. M. Bruynooghe and L.M. Pereira, Deduction Revision by Intelligent Backtracking, In: J.A. Campbell (ed.), Implementation of Prolog, Ellis Horwood, 1984, pp. 194-215.

2. T.Y. Chen, J.L. Lassez and G. Port, Maximal Unifiable Subsets and Minimal Non-unifiable Subsets, New Generation Computing, Vol. 4, 1986, pp. 133-152.

3. W.F. Clocksin and C.S. Mellish, Programming in Prolog, 2nd edition, Springer-Verlag, Berlin, 1984.

TABLE 2: A comparison with some ib schemes.

PROGRAM	SEARCH SPACE (% reduction)		EXECUTION TIME (Speed-up)			
	KL [6]	This	sDIB [4]	BP [1]	KL [6]	This
6-queens (simple)	01	32	2.6	1.5	0.4	1.0
6-queens (clever)	03	38	0.9	0.5	0.6	0.9
Circuit design	73	67	4.2		2.5	1.8
Database query	64	46	2.1	1.2	1.1	1.3
Binary tree	00	00	0.8	0.7	0.8	0.9
Map color (bad order)	99	99		340	~1250	>1500
Map color (good order)	06	18		0.6	0.9	1.1

```
function unify(g,h: term; p: goal): goal;
 begin
  if g is an unbound variable
  then begin
      bind g with h;
      remember G as the goal that binds g;
      SPECIAL CASE: if g and h are both variables
         and g is in a variable in head H
         then instead of binding g with h
              we make g an alias of h;
      return null;
  end;
  if h is an unbound variable
  then return unify(h,g,p);
  if g is a bound variable
  then begin
      let gprime be the term bound to g and
      let q be the goal that bound g to gprime;
      if q invoked its  call after p
      then return unify (gprime,h,q)
      else return unify (gprime,h,p);
  end;
  if h is a bound variable
  then return unify (h,g,p);
  if functor(g) ≠ functor(h)
     /* g and h can not unify */
  then return p;
  for each pair gi,hi of corresponding
                     arguments in g,h do
  begin
      X := unify (gi,hi,p);
      if X ≠ null
      then return X;
  end;
  return null
 end.
```

Algorithm 2: An algorithm to compute L(G,H) -- the suspect that executed the most recent call. The algorithm is a simplified version of the algorithm given in [8]. For a goal G and head H (both G and H are treated as terms), L(G,H) = unify(G,H,G0).

4. C. Codognet, P. Codognet and G. File, Yet Another Intelligent Backtracking Method, In: R.A. Kowalski and K.A. Bowen (eds.), Logic Programming: Proc. of 5th Intl. Conf. and Symp., MIT Press, Cambridge, Aug. 1988, pp. 247-265.

5. P.T. Cox, Finding Backtracking Points for Intelligent Backtracking, In: J.A. Campbell (ed.), Implementation of Prolog, Ellis Horwood, 1984, pp. 216-233.

6. V. Kumar and Y-J. Lin, A Data-Dependency-Based Intelligent Backtracking Scheme for Prolog, J. of Logic Programming, Vol. 5, Nr. 2, June 1988, pp. 165-181.

7. J.W. Lloyd, Foundations of Logic Programming, 2nd edition, Springer-Verlag, 1987.

8. V.M. Malhotra, T.V. To and K. Kanchanasut, An Improved Data-Dependency-Based Backtracking Scheme for Prolog, Information Processing Letters, Vol. 31, No. 4, May 1989, pp. 185-189.

9. G. Port, A Simple Approach to Finding the Cause of Non-Unifiability, In: R.A. Kowalski and K.A. Bowen (eds.), Logic Programming: Proc. of 5th Intl. Conf. and Symp., MIT Press, Cambridge, Aug. 1988, pp. 651-665.

10. T.V. To, A History-based Backtracking Scheme for Prolog, M. Engg. thesis, No. CS-88-6, Division of Computer Science, Asian Institute of Technology, Bangkok, December 1988.

11. D.H.D. Warren, An Abstract Prolog Instruction Set, Tech. Note 309, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025 (Oct., 1983).

12. D.A. Wolfram, Reducing Thrashing by Adaptive Backtracking, J. of Automated Reasoning, to appear, 1989.