Technical Report

UCAM-CL-TR-112
ISSN 1476-2986

Number 112

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Reducing thrashing by adaptive backtracking

D.A. Wolfram

August 1987

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

1 Form of the Problem

General methods are sought for avoiding two inefficiencies of backtracking which are shown in solving the following distinct representatives problem. Given sets $S_1 = \{2,8\}, S_2 = \{3,5\}, S_3 = \{4,5\}, S_4 = \{1,3\}$, $S_5 = \{6,7\}$, and $S_6 = \{1,3\}$, find all sets with six elements of distinct $l_i \in S_i$, where $1 \leq i \leq 6$.

This problem can be solved with backtracking, which is often pictured as a depth-first search of an associated search tree. The branches of a subtree of the search tree are listed in figure 1 below. The set from which each element was chosen has been indicated for clarity.

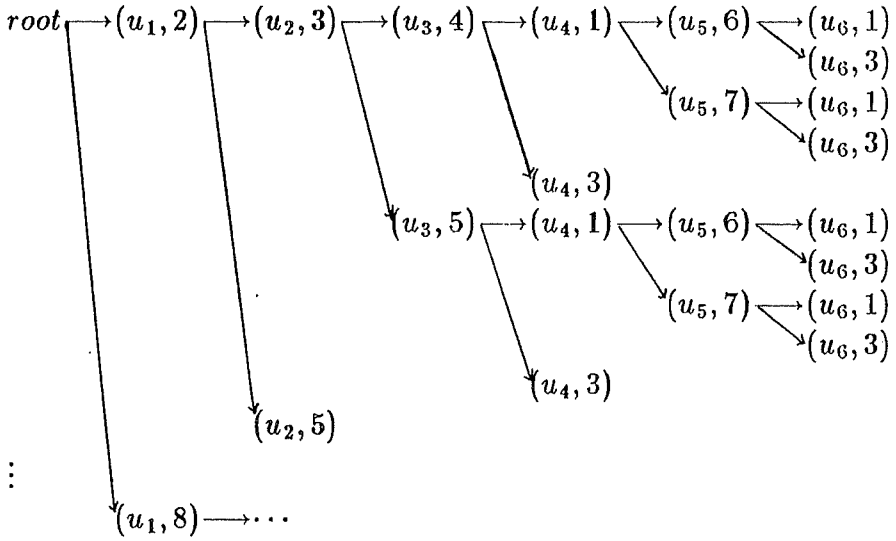

Figure 1: Subtree of the distinct representatives problem.

A *bounding function* [5] (or *bounding property*) used in solving this problem is that the elements chosen are distinct. This search problem and the others considered later have *hereditary* bounding properties: if the bounding property is true for a set, it is true for every subset of that set.

The example illustrates thrashing, which is caused by the inefficiencies of blind backtracking and recurring failures.

1. *Blind Backtracking.* The first subtree with root $(u_5,6)$ does not contain a branch leading to a solution, but searching the subtree with root $(u_5,7)$, predictably will also not contain any branches leading to a solution, because the elements 1 and 3 are still in the branch.

2. *Recurring Failures.* The subtree with root $(u_2,3)$ does not contain a branch leading to a solution, but this recurs with $(u_1,8)$.

Adaptive backtracking can speed-up finding solutions by efficiently pruning these subtrees from the search tree.

The distinct representatives problem is a form of *combinatorial search problem*, which we shall consider initially.

They consist of

1. a finite set $U = \{u_1, \ldots, u_n\}$ of *units*
2. for each unit $u \in U$, a finite set Du of *labels*, which is called the *domain* of unit u, and
3. an hereditary bounding property

$$H : (u_1 \times Du_1) \times \cdots \times (u_n \times Du_n) \rightarrow \{true, false\}.$$

We are interested in finding *all* solutions

$$\{S \mid S \in (u_1 \times Du_1) \times \cdots \times (u_n \times Du_n) \wedge H(S)\}.$$

In the distinct representatives example, each unit is a variable $u_i, 1 \leq i \leq 6$, for a representative. The sets S_1, S_2, S_3, S_4, S_5, and S_6 are the domains, and the hereditary bounding property is that the representatives are distinct. This property is hereditary because if a set of selected representatives are distinct, then so is every subset of that set.

Another example of a combinatorial search problem is the eight queens problem. Here the units correspond to the columns of the chess board, the domain of a unit is the set of rows of the chess board, and the hereditary bounding property is that queens placed on the chess board do not attack each other.

The following logic program and goal can be used to find whether the formula

$$\{x_1 \vee x_3 \vee x_8\} \wedge \{x_3 \vee x_7 \vee x_4\} \wedge \{x_8 \vee x_2 \vee x_8\}$$

is satisfiable with just one true literal per clause.

```
:- c(X1, X3, X8), c(X3, X7, X4), c(X8, X2, X8).

c(1, 0, 0).
c(0, 1, 0).
c(0, 0, 1).
```

This is also a combinatorial search problem where the units are the atoms of the goal `c(X1, X3, X8)`, `c(X3, X7, X4)`, and `c(X8, X2, X8)`; and the domain for each unit consists of the clauses `c(0, 0, 1)`, `c(0, 1, 0)`, and `c(1, 0, 0)`. Unifiability is the hereditary bounding property.

Following the distinct representatives example, each node, except for the root, of a *search tree* represents the *assignment* of a label $l \in Du$ to a unit $u \in U$. No branch in the search tree has two nodes representing assignments to the same unit. The children of a node in the search tree are the assignments $(u, e_1), \ldots, (u, e_m)$, where $u \in U$, and $Du = \{e_1, \ldots, e_m\}$.

A branch is written from the root of a search tree, as an ordered set

$$\{(u_1, l_1), \ldots, (u_n, l_n)\}.$$

The ordering is $root < (u_1, l_1) < \cdots < (u_n, l_n)$. In particular, the maximum of the empty set is *root*. A *consistent* set of assignments is one which satisfies the hereditary bounding property.

The paper is organised as follows. Modifications to ordinary backtracking are described in the next sections. Then we discuss backtracking after a solution has been found. The adaptive backtrack algorithm is then presented with a discussion of practical experience.

The method is a dynamic refinement of ordinary backtracking which finds an early point to backtrack to after a failure, which can prevent thrashing without pruning solutions. The worst case time complexity for finding this backtrack point is $O(nf(n))$ if the bounding property can be tested in time $O(f(n))$, and the worst case space complexity is $O(n^2)$.

Apart from its application to the solution of all combinatorial search problems which use hereditary bounding properties, adaptive backtracking can be generalised [11] for searches that occur in theorem proving, the execution of logic programs, database retrieval, reason maintenance systems, planning, graph theory, and combinatorics.

Dependency-directed backtracking [9,4] is a previous dynamic refinement of backtracking. At a contradiction a tracing function is used to find a backtrack point which can be chosen arbitrarily from a "nogood" set of such points, although this can prune solutions. The nogood sets are recorded permanently and used during the search to prevent regenerating subtrees of the search tree.

Intelligent backtracking was also developed specifically for linear resolution [3], and the execution of logic programs [1,8]. Versions of this approach rely on modifications to the unification algorithms used for those systems. The development of intelligent backtracking for linear resolution prunes search trees using the maximal unifiable subsets at a failure [7]. I showed [10] that in general this approach is not feasible because of the potential exponential growth in the number of these subsets. Considerable memory overheads have been observed in practice [7].

The modifications to backtracking for logic programming seem to use one minimal nonunifiable subset at a failed branch, and no information about previous failures is retained to prevent their recurrence [1].

In contrast to dependency-directed backtracking and intelligent backtracking, we prove that adaptive backtracking retains completeness. It applies uniformly to backtracking after direct and indirect failures, and backtracking after a solution has been found. We define the search problems which can be solved by this method. Sets of backtrack points are not recorded permanently because in general, their storage and use can create exponential overheads. Instead, choices which are not part of any solution are found and deleted. The polynomial space and time overheads of the algorithm are analysed.

2 The Cause Set

A direct failure is a consistent branch $\{(u_1,l_1),\ldots,(u_{n-1},l_{n-1})\}$ which cannot be extended consistently by any assignment for u_n.

Formally, the branch $\{(u_1,l_1),\ldots,(u_{n-1},l_{n-1})\}$ is a *direct failure* if

$$H(\{(u_1,l_1),\ldots,(u_{n-1},l_{n-1})\}) \wedge \\ (\forall l_n \in Du_n)\neg H(\{(u_1,l_1),\ldots,(u_n,l_n)\}) \quad (1)$$

Adaptive backtracking reduces thrashing by backtracking to an assignment, called the *backtrack point*, which could be further back than the previous assignment in the branch. The subtree beneath the backtrack point is searched less than with ordinary backtracking.

In the distinct representatives example, the branch

$$\{(u_1,2),(u_2,3),(u_3,4),(u_4,1),(u_5,6)\}$$

is a direct failure because the labels in $Du_6 = \{1,3\}$ already appear in the branch. The backtrack point is $(u_4,1)$.

A direct failure with the eight queens problem occurs when the queens in the first five columns are in rows $\{(u_1,1),(u_2,3),(u_3,5),(u_4,2),(u_5,4)\}$. There is no row in the sixth column where a non-attacking queen can be placed. No further solutions can be found by moving the queen in the fifth column. The backtrack point is $(u_4,2)$.

The definition of the backtrack point at a direct failure uses one minimal inconsistent subset of the inconsistent set $\{(u_1,l_1),\ldots,(u_n,l_n)\}$, called the *cause set*, which is denoted by $C = cause\{(u_1,l_1),\ldots,(u_n,l_n)\}$. The cause set by definition is the lexicographically least minimal inconsistent subset: it has the property that for each assignment (u,l) of C, all inconsistent subsets which contain all assignments in C greater than (u,l) contain an assignment greater or equal to (u,l).

The cause set of the inconsistent set

$$\{(u_1,2),(u_2,3),(u_3,4),(u_4,1),(u_5,6),(u_6,1)\}$$

from the distinct representatives problem, is the minimal inconsistent subset

$$C = \{(u_4,1),(u_6,1)\}$$

No solution contains $(u_6,1)$ until backtracking returns to $(u_4,1)$.

A direct failure in the Prolog execution of the following goal and logic program occurs when all but the last atom in the goal have been matched.

```
:- p(X1), t(X1, X2), t(X1, X2), t(X2, X3), t(X2, X3), r(X3).

p(0).
t(Y, Y).
r(1).
```

The direct failure is contained in the inconsistent set

```
{(p(X1), p(0)), (t(X1, X2), t(Y1, Y1)), (t(X1, X2), t(Y2, Y2)),
(t(X2, X3), t(Y3, Y3)), (t(X2, X3), t(Y4, Y4)), (r(X3), r(1))}.
```

The cause set of this set is the minimal inconsistent subset

```
{(p(X1), p(0)), (t(X1,X2), t(Y1,Y1)), (t(X2,X3), t(Y3,Y3)), (r(X3), r(1))}
```

No solution can be found with `(r(X3), r(1))` until backtracking returns at least as far as `(t(X2, X3), t(Y3,Y3))`.

The other minimal inconsistent subsets are

```
{(p(X1), p(0)), (t(X1,X2), t(Y1,Y1)), (t(X2,X3), t(Y4,Y4)), (r(X3), r(1))}
{(p(X1), p(0)), (t(X1,X2), t(Y2,Y2)), (t(X2,X3), t(Y3,Y3)), (r(X3), r(1))}
{(p(X1), p(0)), (t(X1,X2), t(Y2,Y2)), (t(X2,X3), t(Y4,Y4)), (r(X3), r(1))}
```

My earlier results [10] show that to find the backtrack point it is not feasible in general to produce all minimal inconsistent subsets, or a minimal inconsistent subset whose size exceeds a certain bound, because there are search problems for which these problems are either intractable or NP-hard. For example, the number of minimal inconsistent subsets using the previous logic program can be made to increase exponentially with the size of the goal.

However, we shall show that the algorithm 2.1 given below can efficiently compute one minimal inconsistent subset, the cause set, of a failed set

$$\{(u_1, l_1), \ldots, (u_n, l_n)\}.$$

Algorithm 2.1

precondition: $\neg H(\{(u_1,l_1),\ldots,(u_n,l_n)\}) \wedge H(\{(u_1,l_1),\ldots,(u_{n-1},l_{n-1})\})$

$M := \{(u_n, l_n)\}$;

loop invariant: $\neg H(\{(u_1,l_1),\ldots,(u_i,l_i)\} \cup M) \wedge M \subseteq \{(u_{i+1},l_{i+1}),\ldots,(u_n,l_n)\} \wedge$
$(\forall N \subset M) H(\{(u_1,l_1),\ldots,(u_i,l_i)\} \cup N)$

for $i := n-1$ **downto** 1 **do**
  if $H(\{(u_1,l_1),\ldots,(u_{i-1},l_{i-1})\} \cup M)$ **then**
    $M := M \cup \{(u_i,l_i)\}$

postcondition: $\neg H(M) \wedge (\forall N \subset M) H(N) \wedge M \subseteq \{(u_1,l_1),\ldots,(u_n,l_n)\}$

In the worst case, the running time of the algorithm occurs when the whole failed set is a minimally inconsistent set with respect to H. The number of assignments in the set tested will be $n-1$ at each test, so that if testing H has time complexity $O(f(n))$, the algorithm has time complexity $O(nf(n))$.

We now prove that the algorithm computes the cause set.

Theorem 2.1 *Let $\{(u_1,l_1),\ldots,(u_n,l_n)\}$ be an inconsistent set at a direct failure. Algorithm 2.1 computes the cause set.*

Proof Just before the assignment (u_i,l_i) is included in M in the algorithm, we have from the program assertion

$$\neg H(\{(u_1,l_1),\ldots,(u_i,l_i)\}\cup M)\wedge H(\{(u_1,l_1),\ldots,(u_{i-1},l_{i-1})\}\cup M).$$

This implies that there is no inconsistent subset N of $\{(u_1,l_1),\ldots,(u_n,l_n)\}$ with respect to H, such that $M\subseteq N$ and $\max(N-M)<(u_i,l_i)$. Otherwise we would have that $N\subseteq\{(u_1,l_1),\ldots,(u_{i-1},l_{i-1})\}\cup M$, and $\neg H(\{(u_1,l_1),\ldots,(u_{i-1},l_{i-1})\}\cup M)$, which is a contradiction. Therefore by definition, M is the cause set when algorithm 2.1 terminates. □

3 Blind Backtracking at Direct Failures

Blind backtracking means returning to an earlier node which has unsearched branches which predictably do not lead to solutions. The number of these branches can increase exponentially because each domain can contain two or more labels. Substantial reductions in search times are possible if a modified algorithm can determine accurately and efficiently the earliest node which could lead to further solutions.

The *backtrack point* for a direct failure is now defined as the maximum assignment of the set

$$L=\bigcup_{l_n\in Du_n} cause\{(u_1,l_1),\ldots,(u_n,l_n)\}-\{(u_n,l_n)\} \qquad (2)$$

As the following theorem shows, adaptive backtracking prevents blind backtracking at direct failures.

Theorem 3.1 *At a direct failure, the backtrack point is the earliest assignment to which backtracking can return without loss of solutions.*

Proof If $L=\emptyset$ then every cause set at a direct failure is a singleton, and the search problem has no solution since there is no consistent label for the most recently selected unit independently of all other assignments. In this case, the backtrack point is the root of the search tree, as required.

Otherwise, each of the branches at the direct failure whose cause sets are not singletons contain an inconsistent subset. There are no further solutions if

backtracking does not return at least as far as the backtrack point. This follows by the definition of cause set, and equation 2. □

A modified backtrack algorithm could now be written which prevents blind backtracking after direct failures. The computation of the backtrack point similarly has worst case time complexity of $O(nf(n))$ because the domains are finite.

If this were the only form of blind backtracking, the computation of the backtrack point could be optimised by modifying the earlier algorithm so that it never finds more than two elements of a cause set. However, blind backtracking can also occur after indirect failures.

4 Blind Backtracking at Indirect Failures

An indirect failure occurs when each assignment for the most recently selected unit either cannot be used to extend consistently a branch in the search tree, or it has been a backtrack point because the subtree beneath it has already been searched without finding any solutions.

The computation of the backtrack point for an indirect failure uses the sets L (equation 2), which are generated at direct failures. A set L is recorded as a *rejection set* $R(u,l) = L - \{(u,l)\}$, where $(u,l) = \max L$ so that it can be retrieved for this computation. The main property of a rejection set is that the search tree below a branch is failed and contains no solutions if the rejection set is a subset of the branch. Initially, all rejection sets are undefined ($\perp$) and are not subsets of any branch.

The branch $\{(u_1,l_1),\ldots,(u_{n-1},l_{n-1})\}$ is an *indirect failure* if there exists $D \subset Du_n$ such that

$$(\forall l_n \in D)\neg H(\{(u_1,l_1),\ldots,(u_n,l_n)\}) \wedge$$
$$(\forall l_n \in Du_n - D)R(u_n,l_n) \subseteq \{(u_1,l_1),\ldots,(u_{n-1},l_{n-1})\}.$$

The backtrack point for indirect failures is defined in a similar way to that for direct failures as the maximum assignment of the set

$$L = \bigcup_{l_n \in D} (cause\{(u_1,l_1),\ldots,(u_n,l_n)\} - \{(u_n,l_n)\}) \cup \bigcup_{l_n \in Du_n - D} R(u_n,l_n) \quad (3)$$

This set L is now recorded as a rejection set $R(u,l)$ so that it can be used in the computation of the backtrack point for other indirect failures. Cause sets and rejection sets at a failure are used to find another rejection set and backtrack point. We shall prove that this computation preserves the main properties of rejection sets and backtrack points.

In solving the earlier distinct representatives problem (Figure 1) there is an indirect failure of the branch

$$\{(u_1,2),(u_2,3),(u_3,4)\},$$

and $R(u_4, 1) = \{(u_2, 3)\}$ was produced at the first direct failure

$$\{(u_1, 2), (u_2, 3), (u_3, 4), (u_4, 1), (u_5, 6)\}.$$

From the backtrack point definition, $D = \{3\}$, $Du_4 = \{1, 3\}$, and

$$\begin{aligned} L &= \bigcup_{l_4 \in \{3\}} (cause\{(u_1, 2), (u_2, 3), (u_3, 4), (u_4, l_4)\} - \{(u_4, l_4)\}) \bigcup \{(u_2, 3)\} \\ &= (\{(u_2, 3), (u_4, 3)\} - \{(u_4, 3)\}) \cup \{(u_2, 3)\} \\ &= \{(u_2, 3)\} \end{aligned}$$

and $R(u_2, 3)$ becomes $\emptyset$, and the next partial branch is $\{(u_1, 2), (u_2, 5)\}$.

Instead of this, ordinary backtracking will fruitlessly search the subtree under $(u_3, 5)$. Adaptive backtracking skips this searching because the rejection set and computed cause set show that branches will be inconsistent for every label in Du_4 independently of every assignment to u_3.

The worst case time for retrieving and using a rejection set is $O(n)$, so that computing another backtrack point also has worst case time of $O(nf(n))$. We now prove that adaptive backtracking is accurate after all failures and that the main property of rejection sets is maintained.

Theorem 4.1 *At a failure, backtracking can return to the backtrack point without loss of solutions.*

Proof Direct failures have been considered in Theorem 3.1. This is the basis for the following inductive proof.

Let (u_m, l_m) be the backtrack point of an indirect failure,

$$\{(u_1, l_1), \ldots, (u_{n-1}, l_{n-1})\}$$

where $1 \leq m < n$. Suppose that

$$R(u_n, l_n) \subseteq \{(u_1, l_1), \ldots, (u_{n-1}, l_{n-1})\}$$

implies that there is no solution containing $\{(u_1, l_1), \ldots, (u_m, l_m)\} \cup \{(u_n, l_n)\}$. For all labels l_n in D, we have $\neg H(\{(u_1, l_1), \ldots, (u_m, l_m)\} \cup \{(u_n, l_n)\})$ since

$$cause(\{(u_1, l_1), \ldots, (u_n, l_n)\}) \subseteq \{(u_1, l_1), \ldots, (u_m, l_m)\} \cup \{(u_n, l_n)\}.$$

Therefore there is no solution containing $\{(u_1, l_1), \ldots, (u_m, l_m)\}$ since there is no solution with unit u_n, and backtracking can return to the backtrack point (u_m, l_m) without loss of solutions. Furthermore, if ever the computed rejection set $R(u_m, l_m)$ is a subset of the branch $\{(u_1, l_1), \ldots, (u_{m-1}, l_{m-1})\}$ there can be no further solutions, so that the main property of rejection sets is also preserved. In the case that L is the empty set, there cannot be any further solution since there is no previous assignment which if changed would allow further solutions to be found. □

5 Preventing Recurring Failures

Adaptive backtracking prevents recurring failures by deleting labels from domains which are not part of any further solution, so that failed subtrees which otherwise would have been generated are pruned.

The rejection sets are used to detect which assignments do not occur in any further solution. From the previous discussion, if ever a cause or rejection set is a singleton $\{(u,l)\}$, there is no earlier assignment in the branch which if changed would enable a further solution to be found with (u,l), and so l can be deleted from Du without subsequently pruning any solutions.

The algorithm can be said to learn about recurring failures, and this is a complementary aspect to the computation of the backtrack point. A singleton rejection set is not recorded in R because it would have no effect in computing backtrack points.

Label deletion is only used when a rejection set is a singleton, not when a cause set is a singleton, for this does not reduce recurring failures, and it can readily be attained in a similar way to a node consistency preprocessor [6]. The label deletion used partially produces higher levels of consistency.

It is possible to record permanently all rejection sets produced during a search and prune a branch when one of the previously generated rejection sets is a subset of it. This is a more general means of preventing recurring failures, but the storage requirements for the rejection sets can increase exponentially with the number of units, and the overhead in checking the subset relation is prohibitive.

6 Backtracking After a Solution

If at least one branch extending from the most recently selected unit u_n is a solution branch, and all assignments for u_n have been tested, then backtracking should return to (u_{n-1}, l_{n-1}) just as in ordinary backtracking, because further solutions may exist. The same method of computing the backtrack point for failures is also used when backtracking after a solution. It is easy to verify that algorithm 2.1 returns all assignments in a consistent branch. This means that the backtrack point for backtracking after a solution is always the assignment to the most recent previously selected unit, as required. It is necessary to identify backtracking after a solution during execution so that labels of u_1 are not deleted unnecessarily.

An implementation of adaptive backtracking need not compute rejection sets after a solution. One way to do this is to a associate a Boolean flag with each unit. Whenever unit u is selected, its flag is set to true. The flag is only set to false when the unit is one of the assignments in a solution. The computation of the backtrack point should only occur when the flag of the most recently selected unit is true. Otherwise, backtracking should return to the assignment to the most recent previously selected unit.

Even without this optimisation, the worst case storage requirement for rejection

sets of adaptive backtracking is $O(n^2)$ in the number of units in the search problem. This upper bound occurs if all rejection sets are of maximum size: $|R(u_n, l_n)| = n - 1$.

7 The Adaptive Backtrack Algorithm

We now present the adaptive backtrack algorithm for search problems. A partial branch is represented by the assignments in set A. It is extended by selecting an unassigned unit from a list B and finding, if possible, a consistent label from its domain. The remaining untested labels of a domain Du_i are stored in a set Eu_i. If every unit has been selected from B, the assignments in A represent a solution to the search problem.

The backtrack point is not computed using rejection sets when backtracking after a solution. A flag F_i is set to false only when (u_i, l_i) is part of a solution.

Algorithm 2.1 is used to find the cause set:

```
function findcause(n: integer): set of assignments;
begin
      M := {(u_n, l_n)};
      for i := n - 1 downto 1 do
          if H({(u_1, l_1), ..., (u_{i-1}, l_{i-1})} ∪ M) then
             M := M ∪ {(u_i, l_i)};
      findcause := M
end;
```

The following function backtrackpoint computes the backtrack point when backtracking after a failure or a solution. The function also updates rejection sets and deletes labels from domains. Function findcause is called if a cause set has to be computed when backtracking after a failure.

```
function backtrackpoint(i: integer): integer;
begin
      if F_i then begin (* Backtracking after a failure *)
        L := ∅;
        for each l_i ∈ Du_i do
                if R(u_i, l_i) = ⊥ then begin
                   L := L ∪ (findcause(i) - {(u_i, l_i)})
                end else
                   L := L ∪ R(u_i, l_i);

        m := max{k | (u_k, l_k) ∈ L}
        if |L| > 1 then
                R(u_m, l_m) := L - {(u_m, l_m)} (* Update rejection set *)
        else if |L| = 1 then  (*.Delete label *)
```

```
        Du_m := Du_m - {l_m};

    end else m := i - 1; (* Backtracking after a solution *)
    backtrackpoint := m;
end;
```

The main part of the algorithm follows. The algorithm would behave in the same way as ordinary backtracking if each call to backtrackpoint(i) always resulted in the value $i-1$.

Algorithm 7.1

```
A := ∅;
i := 1;
B := {u_2, ..., u_n};
F_1 := true;
Eu_1 := Du_1;
while i > 0 do
    if Eu_i ≠ ∅ then begin
      l_i := a label in Eu_i;
      Eu_i := Eu_i - {l_i};

      if H(A ∪ {(u_i, l_i)}) then begin
        A := A ∪ {(u_i, l_i)} (* Advance *)
        if B = ∅ then begin
          print(A); (* Solution *)
          for i := 1 to i do F_i := false;
          A := A - {(u_i, l_i)}
        end else begin (* Select next unit *)
          i := i + 1;
          B := B - {u_i};
          for all l_i ∈ Du_i do R(u_i, l_i) := ⊥;
          Eu_i := Du_i;
          F_i := true
        end
      end

    end else begin (* No more labels - Backtracking *)
      m := backtrackpoint(i); (* Resume forward searching *)
      A := A - {(u_m, l_m), ..., (u_{i-1}, l_{i-1})}; (* Prune the search tree *)
      B := B ∪ {u_k | m + 1 ≤ k ≤ i};
      i := m
    end
```

8 More General Search Problems

Until now we have been considering finding all solutions to finite search problems. However, backtracking is also used in problems where the assignment of a label to a unit can introduce new units for which consistent assignments must also be found. For example, the proof of a theorem corresponding to a unit may involve proving further theorems which have been introduced by the application of a rule of inference from a finite set of such rules. In logic programming, the introduced units correspond to the introduced atoms of a subgoal.

Adaptive backtracking also applies to these more general search problems. The ordering of units is extended so that introduced units are greater than their introduction point. The root of the search tree can be considered as the introduction point of the initial units of the problem.

The effect of this definition is that if ever a set L (equation 3) is the empty set, then backtracking should return to the introduction point of the most recently selected unit. The search ends if the introduction point is the root of the search tree, otherwise if the introduction point is (u, l) then l should be deleted from Du and the search resumed from this point.

9 An Example

The next example illustrates adaptive backtracking in the more general case. Consider the following Prolog program and the goal `:- g(X1, Y1).`

```
g(X, Z) :- f1(X, Y), huge_computation(X), f3(Y, Z).
g(X, Z) :- f1(X, Y), f3(Y, Z).
f1(a, b).
f1(b, c).
huge_computation(Y) :- ...
f3(R, L) :- h(R, T), r(T, L).
f3(c, g).
h(b, c).
h(c, d).
r(c2, e).
r(d, e).
```

At the first direct failure using the first clause for g, no consistent label can be found for `r(T, L)` and the set L computed is the singleton

$$\{(\mathtt{h(R,T)}, \mathtt{h(b,c)})\}$$

so that the label `h(b, c)` is deleted.

An indirect failure now occurs because the label `h(c, d)` gives a failed set. The L set computed for this failure is

{(f1(X, Y), f1(a, b)), (f3(Y, Z), f3(R, L)), (h(R, T), h(c, d))}.

The backtrack point is (f3(Y, Z), f3(R, L)), and the rejection set computed is R(f3(Y, Z), f3(R, L)) = {(f1(X, Y), f1(a, b))}.

A second indirect failure occurs because the label f3(c, g) gives a failed set. The L set computed is the union of the rejection set R(f3(Y, Z), f3(R, L)) and the cause set {(f1(X, Y), f1(a, b)), (f3(Y, Z), f3(c, g))}. The backtrack point is (f1(X, Y), f1(a, b)) and the computed rejection set is a singleton, so that the label f1(a, b) is deleted. Ordinary backtracking would have backtracked to an assignment in the subtree for huge_computation(X) and repeated the failure.

The other label for f1(X, Y) yields a solution. Backtracking after this solution is the same as ordinary backtracking until the subgoals for the second clause for g(X, Z) are executed. If there were only one label for f1(X, Y) initially, then adaptive backtracking would have returned to the introduction point and tried this second clause after the second indirect failure.

10 Practical Results

Adaptive backtracking and ordinary backtracking have been implemented in Pascal for the distinct representatives problem and for the n queens problem. The adaptive backtrack programs are about two hundred lines long.

For the n queens problem, $6 \leq n \leq 10$, the adaptive program was slower than ordinary backtracking and this lag increased with n to 1.3 times for the 10 queens problem. However, the number of failure nodes visited by adaptive backtracking was between 1.5 to 2.3 times fewer than those visited by ordinary backtracking.

In contrast, adaptive backtracking was between 1.4 to 41 times faster than ordinary backtracking for the four distinct representatives problems tested, and the number of failure nodes visited by adaptive backtracking was up to one thousand times fewer. The high figures occurred because adaptive backtracking can detect that a problem has no further solutions when all labels have been deleted from a domain.

The speed of adaptive backtracking compared with ordinary backtracking depends on the nature of the problem being solved; it can be much faster for problems which have early backtrack points and many recurring failures. It was slower for the n queens problem because the computed backtrack point is usually the same as that found by ordinary backtracking without the overhead of computing and using rejection sets, and there were no recurring failures. Adaptive backtracking was faster than ordinary backtracking for the distinct representatives problem because one of the computed backtrack points was eleven units earlier than the last unit in a direct failure, and there were several recurring failures. One of the distinct representatives problems used is listed below. Adaptive backtracking was 1.4 times faster than ordinary backtracking for this problem.

$$Du_1 = \{2, 8, 7\}, Du_2 = \{1, 8\}, Du_3 = \{4, 8\}, Du_4 = \{9, 8\}.$$

$$Du_5 = \{5,8\}, Du_6 = \{12,8\}, Du_7 = \{14,8\}, Du_8 = \{16,8\},$$
$$Du_9 = \{19,8\}, Du_{10} = \{18,8\}, Du_{11} = \{1\}, Du_{12} = \{2\}, Du_{13} = \{8\}.$$

11 Conclusion

Adaptive backtracking applies to all search problems which use hereditary bounding properties. This group of problems identified here includes those in theorem proving, logic programming, reason maintenance systems, planning, pattern recognition and a wide range of other problems in areas such as combinatorics, and graph theory.

Thrashing caused by blind backtracking and recurring failures in the backtrack solution of these problems is reduced by adaptive backtracking. We have proved that the algorithm finds a backtrack point without losing solutions, and that it can delete labels which do not occur in any solution.

Backtrack points after failures are computed using cause sets and rejection sets. The worst case computational overhead of using these sets is $O(nf(n))$ in time and $O(n^2)$ in space. These features prevent fruitless searches of failed subtrees, and they can result in exponential speed-ups.

Adaptive backtracking has been implemented and compared to ordinary backtracking. Four distinct representatives problems whose solution involves significant thrashing were tested, and speed-ups of up to 41 times occurred in accordance with the expected exponential speed-ups. The practical results suggest that a full-scale implementation could provide speed-ups for typical problems whose ordinary backtrack solution would involve significant thrashing.

12 Acknowledgements

This paper is based on part of my thesis which was submitted for the Degree of Master of Science to the Faculty of Science of the University of Melbourne on 13 August, 1985. I should like to thank Rodney Topor, the examiners, Larry Paulson, Martin Richards, and John Carroll for their comments. This research was supported by an Australian Government Postgraduate Research Award 1984 – 1985.

References

[1] Bruynooghe, M., and Pereira, L.M. Deduction Revision by Intelligent Backtracking, *in*: Campbell [2], pages 194–215.

[2] J. Campbell, ed., *Implementations of Prolog*, Ellis Horwood, Chichester, 1984.

[3] Cox, P.T. Finding Backtrack Points for Intelligent Backtracking, *in*: Campbell [2], pages 216–233.

[4] Doyle, J. A Truth Maintenance System, *Artificial Intelligence* **12** (1979) 231–272.

[5] Horowitz, E., and Sahni, S. *Fundamentals of Computer Algorithms*, Pitman, London, 1979.

[6] Mackworth, A.K., and Freuder, E.C. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems, *Artificial Intelligence* **25** (1985) 65–74.

[7] Matwin, S., and Pietrzykowski, T. Intelligent Backtracking in Plan-Based Deduction, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **7**, 6 (1985), 682–692.

[8] Sato, T. An Algorithm for Intelligent Backtracking, *Proceedings of the RIMS Symposia on Software Science and Engineering*, Lecture Notes in Computer Science, **147**, Springer, 1983.

[9] Stallman, R.A. and Sussman, G.J. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer Aided Circuit Analysis, *Artificial Intelligence* **9** (1977) 135–196.

[10] Wolfram, D.A. Intractable Unifiability Problems and Backtracking, *Proceedings of the Third International Conference on Logic Programming*, London, July, 1986, Lecture Notes in Computer Science, Springer, **225**, 107–121, 1986.

[11] Wolfram, D.A. *Adaptive Backtracking*, M.Sc. Thesis, Department of Computer Science, University of Melbourne, August, 1985.