Backtracking with Multi-Level Dynamic Search Rearrangement

by

Paul Walton Purdom, Jr.
Cynthia A. Brown
Edward L. Robertson

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 77
BACKTRACKING WITH MULTI-LEVEL
DYNAMIC SEARCH REARRANGEMENT
PAUL WALTON PURDOM, JR.
CYNTHIA A. BROWN
EDWARD L. ROBERTSON
REVISED FEBRUARY 1980

Research reported herein was supported in part by the National Science Foundation under grant number MCS 79 06110.

Backtracking With Multi-Level Dynamic Search Rearrangement*

Paul Walton Purdom, Jr.
Cynthia A. Brown
Edward L. Robertson

Abstract. The order in which the variables are tested in a backtrack program can have a major effect on its running time. The best search order usually varies among the branches of the backtrack tree, so the number of possible search orders can be astronomical. We present an algorithm that chooses a search order dynamically by investigating all possibilities for k levels below the current level, extending beyond k levels wherever possible by setting the variables that have unique forced values. The algorithm takes time $O(n^{k+1})$ to process a node. For k = 2 and binary variables the analysis for selecting the next variable to introduce into the backtrack tree makes complete use of the information contained in the two-level investigations. For larger k or variables of higher degree there is no polynomial-time algorithm that makes complete use of the k-level investigations to limit searching (unless P = NP). The search rearrangement algorithm is closely related to constraint propagation. Experimental studies on conjunctive normal form predicates confirm that

1-level search rearrangement saves a great deal of time compared to 0-level (ordinary backtracking), and show that 2-level saves time over 1-level on large problems. For such problems with 256 variables 2-level is better than 1-level by a factor of two.

---

*Research reported herein was supported in part by the National Science Foundation under grant number MCS 79 06110.

1. Introduction

Backtracking is a technique that is useful for solving some problems for which no polynomial time algorithm is known. (Knuth (1) gives an introduction to backtracking). Various authors have discussed the advantages of using dynamic search rearrangement to enhance the performance of backtracking. Bitner and Reingold (2) observe that dynamic search rearrangement led to a major improvement in the running time of their programs. Transforming a problem into one with binary variables gives maximum scope to search rearrangement. Knuth (3) gives an example based on the work of Tompkins and Paige (4) and of Parker (5) in which a similar recoding combined with search rearrangement resulted in an improvement by a factor of 10^{11} in the time required by a program. Constraint propagation, which has been extremely successful in scene analysis, may be viewed as a method of reducing problems to binary variables and then using dynamic search rearrangement with backtracking (avoiding actually backtracking altogether in the best case). These studies show that dynamic search rearrangement can lead to major improvements in the running time of backtrack programs.

In this paper we present a generalization of search rearrangement that investigates several levels into the backtrack tree before selecting the next variable. Empirical studies on a representative class of problems show that this method can save a significant amount of time over simple search rearrangement.

2. Notation and Basic Algorithm

Any problem that can be solved in NP-time (7) corresponds to a family of predicates that can be evaluated in polynomial time. Solving an instance of an NP-time problem may be regarded as a search for an assignment of values $X_1,\ldots,X_n$ to variables $x_1,\ldots,x_n$ such that the value of the corresponding predicate P on $X_1,\ldots,X_n$ is true. A simple exhaustive search tries every combination of the values of the variables. It can be implemented by assigning a tentative value X_1 to x_1 and attempting recursively to find values for $x_2,\ldots,x_n$, then trying successive values for x_1 and so on until the search terminates. If $\tilde{P}(x_1) = (\exists x_2) \ldots (\exists x_n)P(x_1,x_2,\ldots,x_n)$ is easy to compute, perhaps without discovering exact values for $x_2,\ldots,x_n$, then a more economical technique would be to first test whether a particular X_1 satisfies $\tilde{P}$. In this way values of x_1 that lead to "blind alleys" could be avoided. In most cases $\tilde{P}$ itself is not easy to compute, but backtracking algorithms are based on the hope of constructing predicates that warn of blind alleys a significant proportion of the time.

Let $S = \{x_1,\ldots,x_n\}$ be the ordered set of variables for predicate P. Let $T = \{x_1,\ldots,x_k\}$ be an initial segment of S, and let P_T be an arbitrary predicate over the variables in T. If P implies P_T in the sense that $P_T(X_1,\ldots,X_k)$ may be false only if

$P(X_1,\ldots,X_k,X_{k+1},\ldots,X_n)$ is false for all sets of values $X_{k+1},\ldots,X_n$ of $x_{k+1},\ldots,x_n$, then a false value of P_T would indicate a blind alley when searching for a solution to P. In this case we say P_T is <u>consistent</u> with P.

More generally, if A and B are subsets of S, $A \subset B$, and P_A and P_B are predicates over A and B, we say that P_A is consistent with P_B if P_B implies P_A. A set of predicates $\mathcal{P} = \{P_D\}_{D \in \mathcal{D}}$, where $\mathcal{D}$ is a set of subsets of S, is consistent if for any pair of predicates P_A and P_B in $\mathcal{P}$, $A \subset B$ implies P_A is consistent with P_B.

For a given $A \subset S$, $A = \{a_1,\ldots,a_i\}$, the strongest predicate consistent with P_A is $(\exists a_1) \ldots (\exists a_i)P_A$. Usually this predicate is not easy to compute. The identically true predicate is consistent with every predicate and is easy to compute, but it provides no useful information. An <u>intermediate</u> predicate for a problem is a predicate over a subset of the variables that is consistent with the predicate for the problem. A problem is a candidate for solution by backtracking if it has intermediate predicates that are strong enough to give a significant savings over exhaustive search and that are relatively easy to compute.

In some cases the structure of the intermediate predicates might suggest a good search order. For the sake of generality we will assume that the predicates are so complicated that they must be treated as "black boxes".

The only way to get information about a predicate is to evaluate it on some set of values. This precludes the use of methods that directly analyze the definitions of the intermediate predicates.

At a given point in the backtrack search, some variables will have been assigned tentative values while others are still unset. We let S' denote the (ordered) set of variables which do not yet have an assigned value; S" denotes S-S'. The value of each x in S" is denoted by X. We call the intermediate predicate that is used at this stage $P_{S"}$. One such predicate can be obtained from P by substituting the values of the variables in S" into the formula for P. The resulting formula is evaluated over the variables in S. If there is enough information to determine that the value will be false no matter what values the remaining variables get, then it returns false; otherwise the value is true. This predicate is always consistent with P. We hope that there is an efficient intermediate predicate which substantially reduces the search space, but if none is available we will use the identically true predicate. In any case, an efficient intermediate predicate does exist for every subset of S.

The general form of the backtrack algorithm for finding all solutions (not just a single solution) to a problem is given below. The code in square brackets is omitted in the more primitive versions of the algorithm.

Backtrack Algorithm

```
S" ← φ; S' ← S; BTFLAG ← true; [Initialize variable list;]
while true do
     if BTFLAG then BTFLAG ← P_S"(current node);
     if (BTFLAG and  S' = φ) then
          begin
              print current node;
              BTFLAG ← false;
          end;
     if  (not  BTFLAG)  then while  (S" ≠ φ  and the first
          variable in  S"  has no more values)  do
          begin
               put the first variable of  S"  in  S';
          end;
     [update variable list;]
     if  S" = φ  then stop
          else set the first variable of  S"  to its next
          value;
     BTFLAG ← true;
     call SELECT;
endwhile;
end backtrack algorithm.

Procedure SELECT:
     choose a variable  x  in  S' (see section 3);
     transfer  x  from  S'  to  S";
     assign  x  its first value;
```

```
        return;
    end SELECT.
```

Transfers of variables between S' and S" insert elements at the front of the ordered set. When the backtrack algorithm manipulates the first element of S", it is the element most recently transferred to S". In the algorithms below, "for" loops over ordered sets start with the first element and proceed in order.

The method of choice used in procedure SELECT is the main subject of this paper. The algorithms we describe below differ mainly in the way this procedure is implemented. For each algorithm we give the portions that are not fully specified in the basic algorithm: the code for SELECT and in some cases the code for manipulating variable lists. We begin with the simplest algorithm.

3. The 0-Level and 1-Level Search Algorithms

The simplest case of search rearrangement is to do no rearrangement. Thus the code for SELECT is quite simple. We distinguish algorithms by the depth of search during the selection and tentative setting of a variable. This simple algorithm with no search is thus called 0-level select.

0-Level Search Algorithm

```
Procedure SELECT
     x ← first element of  S';
     transfer  x  from  S'  to  S";
     set  x  to its first value  X;
     return;
end SELECT.
```

This is the traditional backtracking algorithm. Figure 1 shows the backtrack tree that is obtained when this algorithm is used to find the values of the variables that satisfy the predicate $Q = A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G \wedge H \wedge I \wedge J$, where $A = a \vee c$, $B = a \vee d$, $C = a \vee \neg e$, $D = \neg a \vee b$, $E = \neg b \vee c$, $F = c \vee d$, $G = \neg c \vee f$, $H = \neg b \vee \neg e \vee f$, $I = d \vee \neg e \vee f$, and $J = \neg b \vee \neg d \vee e \vee \neg f$. The intermediate predicate for each subset of variables is the conjunction of all the clauses that contain only variables for that subset. In Figure 1, each leaf is labelled with the name of the (first) clause that causes backtracking. The unlabelled leaves are solutions. Each interior node is labelled with the variable that is set at that node. Left branches correspond to the value false and right branches to true. Figure 1 has 39 nodes, compared to the 64 cases that would need to be examined in an exhaustive search.

The following version of SELECT gives a 1-level search rearrangement algorithm.

1-Level Search Algorithm

```
Procedure SELECT:
     SELECTFLAG ← false;
     for each  x  in  S' while (not SELECTFLAG) and
          BTFLAG  do
     begin
          count[x] ← the number of values of  x  for which
               P_{S"∪{x}} is true;
          if count[x] = 0  then  BTFLAG ← false;
          if count[x] = 1  then  SELECTFLAG ← true;
     end;
     if (not BTFLAG) then return;
     if (not SELECTFLAG) then set  x  to a variable with a
          minimum count[x];
transfer  x  from  S' to  S";
     set  x  to its first value;
     return;
end SELECT.
```

This is essentially the algorithm of Bitner and Reingold (2). It is possible to modify the algorithm so that it saves the results of testing $P_{S"\cup\{x\}}$; this will be considered further with the multilevel algorithm. (Also see (8) for such an algorithm). Figure 2 shows the result of applying the 1-level algorithm to predicate Q above. Notice that different search orders are selected for the two main branches of the tree. The 1-level algorithm reduces the number of nodes from 39 to 25. Using the 1-level algorithm

increases the time needed to process a node by about a factor of n over the 0-level algorithm. For large problems this increase is usually offset by the exponential savings in the number of nodes that can result from a good search order.

4. Multi-Level Search Rearrangement

The most obvious way of extending the 1-level algorithm is to consider all two-level extensions of the current node, and to select as the next variable the root variable of a subtree of minimum width at level two. This method requires $O(n^2)$ time to process a node. It can be improved by investigating the implications of the data contained in the 2-level search trees.

We will say problem P_1 is covered by problem P_2 if for each solution to problem P_1 there is a corresponding solution to problem P_2. A good cover is one for which problem P_2 does not have many additional solutions.

Notice that an intermediate predicate is one that is consistent with the original predicate but is weaker because it does not depend on all the variables of the original predicate. A covering predicate is consistent with the original predicate but is weaker because it does not exclude as many nodes as the original predicate. We will construct a covering problem from the conjunction of a number of intermediate predicates; this in turn leads to a good method for doing search rearrangement.

First consider the 1-level search rearrangement algorithm. Assume that the variables in set $A \subset S$ have been assigned tentative values. The problem is now to find values for the remaining variables. For each $b \notin A$, there is a predicate $P_{A \cup \{b\}}$. The predicate for the covering problem is the conjunction of all the $P_{A \cup \{b\}}$; for each subset of variables in S-A the intermediate predicate is the conjunction of all the $P_{A \cup \{b\}}$ defined on that subset. This covering problem is very simple because the permitted values of each variable are independent of the values chosen for the other variables. The smallest backtrack tree for this problem is the one with the node of lowest degree at the root. Thus considering 1-level lookahead as a covering problem leads to the correct rule for selecting the next variable.

The same considerations lead directly to the appropriate algorithm for two-level search rearrangement. For each pair of variables $b,c \notin A$, there is a predicate $P_{A \cup \{b,c\}}$. The predicate for the covering problem is the conjunction of the $P_{A \cup \{b,c\}}$ for all pairs of variables $b,c \notin A$; the intermediate predicate for a subset of variables B, $B \cap A = \emptyset$, is the conjunction of the $P_{A \cup \{b,c\}}$ for $b,c \in B$. If the variables are binary then the permitted values for each variable in the covering problem can be found in time $O(n^3)$ by the following procedure. Test each value of each variable. If some value forces other variables to have a certain value, they are temporarily set. If, during this process a contradiction

(a variable with no possible values) arises, the values being tested do not occur in any solution; otherwise they may occur in some solution. Thus, for binary variables, all the information contained in 2-level lookahead trees can be obtained in a time $O(n^3)$.

If the variables can take on more than two values, the problem of finding all the implications of the information in 2-level trees is NP-complete. To see this, we map the NP-complete problem of formulas in conjunctive normal form having three literals per term (7) into a conjunction of binary relations on three-valued variables. In the conjunctive normal form formulas each literal is a Boolean variable x_m, $1 \le m \le n$, or its negation. Let the i^{th} term be $a_i \vee b_i \vee c_i$, where $a_i \in \{x_j, \neg x_j\}$, $b_i \in \{x_k, \neg x_k\}$, $C \in \{x_1, \neg x_\ell\}$. Define the predicate $Q_r(a,b)$, for $a \in \{0,1\}$, $b,r \in \{0,1,2\}$ as $Q_r(a,b) =$ *if* $r = b$ *then* a *else* 1. A new conjunctive normal form formula can be constructed using the Q_r's, the original variables x_m and their negations, and new variables y_i, $1 \le i \le n$, taking values from $\{0,1,2\}$. The i^{th} term of the original formula is replaced by $Q_0(a_i,y_i) \wedge Q_1(b_i,y_i) \wedge Q_2(c_i,y_i)$. In any assignment satisfying the conjunction of the Q's, $y_i = 0$ (or 1 or 2) implies that a_i (or b_i or c_i, respectively) is true. Thus the original formula is satisfiable if and only if the conjunction of the Q's is. The problem is thus NP-hard. Since a solution satisfying the conjunction of any

collection of predicates can be guessed (if it exists) and verified in polynomial time, the problem is NP-complete.

It is also well known that the problem of using all the information in three level binary trees is NP-complete. (7) Thus, while in the case of 2-level search rearrangement on binary variables it is reasonable to expect the search rearrangement algorithm to process each node in polynomial time and obtain all the information from the lookahead trees, it is unreasonable to expect this in the more difficult cases.

The covering problem for the two-level searches on binary variables can also be solved in time $O(n^2)$ by using the Putnam-Davis (9) procedure. This method does not permit us to easily recover the information contained in the poles built by our algorithm. (See below). Moreover, it is not clear how to generalize the Putnam-Davis procedure to the case of nonbinary variables or searches of more than two levels. For two-level searches and binary variables an investigation of the result of using the Putnam-Davis procedure might be worthwhile.

The question of how much effort to expend processing each node is an interesting one. Since the backtrack predicates can be viewed as conjunctions of relations on the variables, the work of Schaefer (10) on relations is relevant. Schaefer establishes six classes of relations that can be solved in polynomial time: 0-valid, 1-valid, weakly positive, weakly

negative, bijunctive, and affine. Regarding the k-level search (omitting the implication poles -- see below) as defining a set of relations on k variables (one relation for each set of variables) we would like to establish that the set of relations does not have a solution, so that we can back up. If the set of relations falls under one of Schaefer's classes and doesn't have a solution, then our method will discover in polynomial time that no solution exists, thereby using the information in the k-level trees to its fullest extent. Our method takes n times longer than Schaefer's, since it cannot examine the definitions of the relations, but by following implications as described below it may discover extra information that allows us to backtrack even when the set of relations has a solution. When it is not the case that all the relations being considered fall into one of the above classes, the problems can become NP-complete. In this case our algorithm still runs in polynomial time, but it does not always notice when a set of relations has no solution. This suggests that the algorithm does about the right amount of work in analyzing the implications of the information in the k-level search trees it builds.

The algorithm includes as an option the opportunity to use the results of the analysis of the relations that are affine (equivalent to linear equations). This option is indicated by a call to procedures LINEAREQUATIONS. The option is included to ensure that all of Schaefer's classes

can be provided for. Linear equations appear in many practical problems. On the other hand, we give no details for this case, since linear equation analysis may be too specialized for a general purpose backtracking method. A relation that corresponds to a modulo two linear equation must give false for half the values of its variables (ignoring the useless equation 0=0). Therefore a random relation is unlikely to be a linear equation or even to have a covering problem that is a linear equation. This is in contrast to the bijunctive, weakly positive, and weakly negative cases, which appear to be very useful in covering a general relation.

The multi-level search rearrangement algorithm is based on two ideas. The first is to build all possible (k-1)-level search trees and then to extend the bottom level as far as possible with nodes of degree one, which correspond to variables with forced values. Ordinarily this results in a "pole" of length zero or more extending from level k-1 to the next to bottom level, which will have a node of degree two or greater (or a solution). Occasionally the process leads instead to a contradiction, and that branch is pruned from the tree. The considerations mentioned earlier suggest that this k-level analysis will be effective in reducing the size of many backtrack trees.

The second idea is to maintain, for each variable, a list of its permitted values. For each value that is not

permitted, the level in the backtrack tree where it became illegal is noted, and the value becomes legal again on backtracking above that level. This idea is a feature of constraint propagation methods and for non-binary variables it helps reduce the time spent doing search rearrangement. Binary variables are always introduced into the backtrack tree as soon as one of their values is eliminated, so there is no reason to keep a list of permitted values in that case.

It takes time $O(n^{k+1})$ to analyze a node using k-level search, but the upper limit occurs only when a substantial number of long poles are encountered during the analysis. It takes time $O(n^{k-1})$ to build the (k-1)-level search tree, and time $O(n^2)$ to follow a long pole. Only small values of k should be considered for practical algorithms. The algorithm can lead to an exponential decrease in running time, and will not cause it to increase by more than $O(n^{k+1})$.

The following code segments are added to the backtrack algorithm in the indicated places to maintain the lists of permitted values.

Initialize Variable List

```
for each x in S' and for each value X of x do
    permitted[x,X] ← yes;
mainlevel ← 0;
```

Update Variable List

for each x in S' and for each value X of x do

if permitted[x,X] ≥ mainlevel then permitted[x,X] ← yes;

mainlevel ← mainlevel -1;

The value "yes" is not greater than or equal to any positive number. If permitted[x,X] is a non-negative number, the value X is not permitted. The double for loops in update variable list can be implemented by maintaining linked lists of the values that need updating. We omit the details.

We are now ready to present the multi-level SELECT algorithm. It involves a series of nested loops, each of which is presented as a separate function. SELECT chooses the next variable by determining a cost and a degree for each unset variable. The degree is the number of values that remain after the k-level search has pruned as many as it can. If a variable of degree zero is found, backtracking is done immediately. If a variable of degree one is found, it is selected immediately. Otherwise, after the analysis is completed for all variables, the variable with the lowest cost is chosen. The function we use for cost is

$$\underset{\substack{\text{k-plus level trees}}}{\text{minimum}} \left(\sum_{\substack{\text{nodes on} \\ \text{level k-1}}} d(\text{node}) \times \beta^{-\text{level(node)}} \right)$$

where d(node) is the degree of the node at the bottom of the pole of degree one nodes extending from the original node, level(node) is the length of the pole, the sum is

over all poles that survive pruning, the k-plus level trees are the k-level trees with their associated poles, and β is an arbitrary parameter. Our method of assigning cost is similar to a method developed by Johnson for satisfying most clauses of a formula in conjunctive normal form (11). A good value for β is the average branching factor of the backtrack tree.

Multi-level SELECT Algorithm

```
Procedure SELECT:
     (at this point BTFLAG is true)
mainlevel ← mainlevel + 1;
SELECTFLAG ← false;
for each  x  in  S' while (not SELECTFLAG) and BTFLAG do
     begin
          cost [x] ← 0; degree ← 0; transfer  x  from  S' to S";
          for each value  X  of  x  for which  P_S"  is true do
               call TESTVALUE; (TESTVALUE sets degree,
               cost[x], and permitted[x,X])
          if degree = 0 then  BTFLAG ← false;
          if degree = 1 then  SELECTFLAG ← true;
          reinsert  x  from  S" to S'; (reinsert maintains
          the original order of the list  S')
     end;
if (not BTFLAG) then return;
if (not SELECTFLAG) then
```

```
     begin
          sort the variables in  S'  in order of increasing
          cost;
          set  x  to the first element in  S';
     end;
transfer  x  from  S'  to  S"  and set it to its first
     permitted value;
return;
end SELECT.
```

The set S' is sorted in the hope that nodes of degree zero or one will be found sooner on the next transfer to SELECT.

TESTVALUE counts the number of values X of x that survive the k-level search. It also totals the costs for the various X, using the value intermediatecost calculated by SEARCH.

```
Procedure TESTVALUE:
     call SEARCH; [call LINEAREQUATIONS;]
     if SEARCH returns true [and LINEAREQUATIONS returns true]
          then
          begin
               degree ← degree + 1;
               cost[x] ← cost[x] + intermediate cost;
          end;
     else permitted[x,X] ← mainlevel;
end TESTVALUE.
```

SEARCH selects each possible set of variables for building levels 2 through k-1 of the search tree. It returns false if it finds some set of variables for which all value assignments lead to a contradiction. Otherwise it computes the cost of the least expensive variable set. The variable treecost is set by TRYVALUES.

```
Procedure  SEARCH:
     intermediate cost ← ∞;
     for each  (k-2)  element subset  A of S' (note that
          this is empty if  k=2  and by convention has
          |S'|  elements if  |S'| < k-2)  do
          begin
               call TRYVALUES;
               if  TRYVALUES  returns false then return false;
               intermediatecost ← min (intermediatecost,
               treecost);
          end;
     return true;
end  SEARCH.
```

TRYVALUES sums the cost of all nodes at level k-1 in the search tree. If all nodes at level k-1 lead to contradictions, it returns false.

```
Procedure  TRYVALUES:
    branches ← false; treecost ← 0;
    for each assignment of values  p  to the variables in
        A  for which  P_{S"∪A}  is true do
        begin
            transfer the variables in  A  from  S'  to  S";
            set the variables in  A  to the values in  p;
            call IMPLICATIONS; (IMPLICATIONS assigns a
            value to bottomcost in addition to returning
            true or false)
            if IMPLICATIONS returns true then do
                begin
                    treecost ← treecost + bottomcost;
                    branches ← true
                end;
        end;
    return branches;
end TRYVALUES.
```

Given a node on level k-1 of the search tree, IMPLICATIONS considers all unset variables. If some variable has no viable values, IMPLICATIONS returns false. If some variable has only one viable value, IMPLICATIONS sets it to that value and then reconsiders all the unset variables. When either all remaining unset variables have more than one viable value or there are no more unset variables, a cost is assigned to the node on level k-1. The cost is the degree of the remaining variable of lowest degree (or zero if there is no remaining

variable) times β^{-level}, where level is the number of variables whose values were forced.

```
Procedure IMPLICATIONS:
    bottomdegree ← ∞; bottomcost ← ∞; level ← -1; list ← Ø;
    descendingloop:  while  S'  is not empty do
        begin
            level ← level +1;
            costloop:  for each  x  in  S'  do
                begin
                    bottomdegree ← number of values  X
                     of  x  for which  P_{S"∪{x}}  is true;
                    if bottomdegree = 0 then exit
                     descendingloop;
                    if bottomdegree = 1 then
                        begin
                            transfer  x  from  S'  to  S"
                            and set the value of  x  to
                            the one value  X  that makes
                            P_{S"}  true;
                            put  x  on list;
                            exit costloop;
                        end;
                    bottomcost ← min (bottomcost, bottom-
                    degree x β^{-level});
                end;
```

```
        end costloop;
    if bottomdegree > 1 then exit descendingloop;
      end;
end descendingloop;
if  S'  is empty then bottomcost ← 0;
        for each  x  on list do reinsert  x  from  S"  to  S";
        if bottomdegree ≠ 0 then return true else return false;
end IMPLICATIONS.
```

This completes the algorithm. If it is used to find only one solution rather than all solutions, it should be modified by replacing the array "permitted" by sorted lists of values. TESTVALUE should be modified to keep these lists sorted according to the cost of investigating each value of a variable. A program for this algorithm with $k > 3$ should combine the code for SEARCH and TRYVALUES so that the predicate is tested level by level, using ordinary backtracking. In some cases it might also be advantageous to combine IMPLICATIONS and SEARCH. We do not do that here because it is not clear how to do it best, it would make the explanation of the algorithm more complex, and we believe $k = 2$ to be the most interesting case for the algorithm.

The algorithm as written may evaluate P a number of times on the same set of variables and values. A hash table for storing the results of evaluations (with provisions for forgetting them when they become old), such as suggested by Zobrist (12), would help reduce repeated function evaluations.

When k is small (in practice, when k=2), an alternative is to store the results in an indexed table. It is often helpful to precede the k-level search by a k-1 level search (and so on, down to a one level search), and allow the preliminary searches to continue to the next level only if they do not find a node of degree zero or one.

Figure 3 shows the result of using two-level search rearrangement on the problem of Figure 2. The two-level analysis selects variables c and a. The L2 on the figure indicates a node that failed as a result of the two-level analysis, rather than as a result of not satisfying some one clause. For all nodes other than the ones where c and a were chosen, a one-level analysis found a variable with a forced value to introduce. Although the tree in Figure 3 is only slightly smaller than the one in Figure 2 (21 vs. 25 nodes), it is an optimum backtrack tree for this problem. As with the previous algorithm, the increased time per node means that the two level algorithm improves the overall performance only on large trees.

5. Experimental Results

Previous investigators [2] have shown that 1-level search rearrangement leads to substantial savings over ordinary backtracking. For $k \geq 2$ the amount of extra work done per node is large enough that it is not clear *a priori* when multi-level search should be used. We

compared 1-level and 2-level search rearrangement experimentally on random conjunctive normal form predicates. Problem sizes ranged from two to sixteen: for problem size p the number of variables was p^2 and the number of terms p^3; there were three literals per term. Problems of this type lead to backtrack trees that have many properties similar to those of the backtrack trees the authors have encountered in realistic problems. The performance of 0-level backtracking on these problems was analyzed in [13]. In the 2-level tests we first tried 1-level search on each node; the 2-level search was done only if the 1-level search did not find any variables of degree zero or one. As a measure of work we used the number of calls to the function evaluator. The results of the tests on problem size two, twelve, and sixteen are shown in Table 1. Already on problem size two both 1 and 2-level search rearrangement outperformed 0-level backtracking. The 1-level algorithm did better than the 2-level up to problem size 12, where their performance was about equal; by problem size 16 the 2-level algorithm was better than the 1-level by about a factor of two. We expect that the 2-level algorithm would have an increasingly greater relative advantage on harder problems.

Many optimizations can be found when implementing the algorithms presented in this paper. For example, when $k=2$ function calls can be reduced by about a half by remembering

the result when one variable is being tested and a second is being set at level k-1, and not doing the function calls in the converse case. Similar savings can be realized for larger k at the cost of increasingly complex bookkeeping. When k=2 or the variables are binary many parts of the algorithms can be omitted or simplified.

Our preliminary experimental results are encouraging; they indicate that 2-level search rearrangement can be a valuable tool for solving large problems. In future work we plan a more detailed empirical study and comparison of 0, 1, and 2-level search rearrangement.

5. Conclusion

We have presented a multi-level search rearrangement algorithm which appears to be nearly optimal for two-level searches with binary variables. Using time $O(n^3)$, it makes use of all the information available from two-level search trees (plus some additional information) to select the most promising variable to introduce next. Any algorithm that uses all the information from two-level trees to decide which variables correspond to nodes of degree zero and one must use at least time $O(n^2)$ per node if it processes each node independently. Therefore there are probably no significantly better two-level rearrangement algorithms. For k=2 and binary variables our experimental studies show that 2-level search rearrangement is a valuable practical

tool for solving some large problems.

For $k \geq 2$ and variables of degree greater than two the algorithm is a natural extension of the basic idea. Since these cases correspond to NP-complete problems, it is harder to tell how close we have come to a good algorithm.

16 variables, 64 terms, 3 literals per term

Algorithm	Nodes	Function calls
0 level	$(1.955 \pm 2.103) \times 10^3$	$(1.955 \pm 2.103) \times 10^3$
1 level	$(1.793 \pm 1.097) \times 10^2$	$(7.799 \pm 3.953) \times 10^2$
2 level	$(6.748 \pm 0.724) \times 10^2$	$(3.435 \pm 3.435) \times 10^3$

144 variables, 1728 terms, 3 literals per term

Algorithm	Nodes	Function calls
0 level	6.665×10^{12}	6.665×10^{12}
1 level	$(9.025 \pm 6.833) \times 10^3$	$(1.771 \pm 0.864) \times 10^4$
2 level	$(1.824 \pm 0.706) \times 10^2$	$(1.661 \pm 0.842) \times 10^4$

256 variables, 4096 terms, 3 literals per term

Algorithm	Nodes	Function calls
0 level	1.092×10^{19}	1.092×10^{19}
1 level	$(4.453 \pm 3.236) \times 10^4$	$(1.232 \pm 0.821) \times 10^6$
2 level	$(3.432 \pm 1.110) \times 10^2$	$(7.725 \pm 2.660) \times 10^5$

table 1

The average number of nodes and function calls for problem with 16,144, and 256 variables. The expected standard deviation for a single run is given after the plus-minus sign. For zero level the average is calculated using [13] and the deviation is obtained from 10 runs. For one level and two level the average and the deviation are obtained from 100 runs. Thus the average for zero level is exact, while the error on the averages for the other two cases is about 10% of the listed standard deviation.

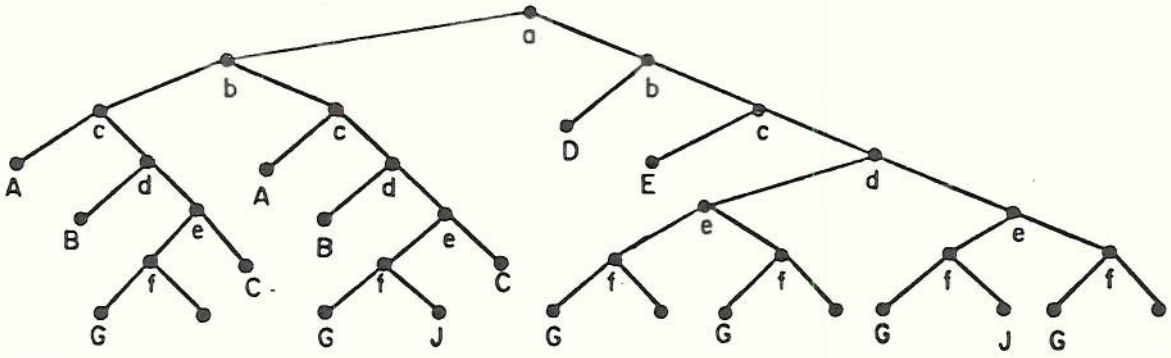

Figure 1. A fixed order backtrack tree with 39 nodes and 6 variables.

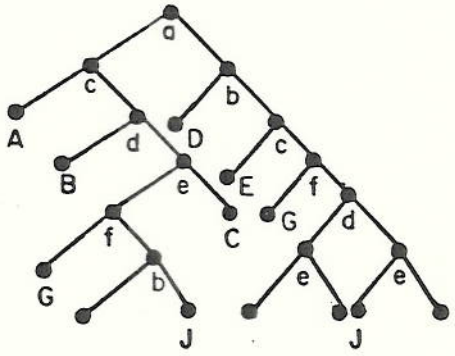

Figure 2. A 1-level backtrack tree with 25 nodes and 6 variables. The tree is for the same problem as the tree in figure 1.

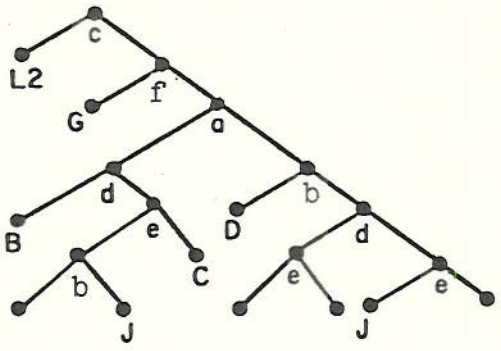

Figure 3. A 2-level backtrack tree with 21 nodes and and 6 variables. The tree is for the same problem as the tree in figure 2.

References

1. Donald E. Knuth, "Estimating the Efficiency of Backtrack Programs," Math. Comp., v. 29 (1975) pp. 121-136.

2. James R. Bitner and Edward M. Reingold, "Backtrack Programming Techniques," CACM, v. 18 (1975) pp. 651-655.

3. Donald E. Knuth, "Mathematics and Computer Science: Coping with Finiteness," Science, v. 194 (1976) pp. 1235-1242.

4. C. Tompkins, "Machine Attacks on Problems Whose Variables are Permutations," Proc. Symp. Appl. Math., v. 6, Amer. Math. Monthly, (1956), p. 195; L. J. Paige and C. Tompkins, "The Size of the 10x10 Latin Square Problem," ibid., v. 10 (1960) p. 71.

5. E. T. Parker, "Computer Investigations of Orthogonal Latin Squares of Order Ten," Proc. Symp. Appl. Math., v. 15, Amer. Math. Soc., Providence, R.I. (1963), p. 73.

6. David Waltz, "Understanding Line Drawings of Scenes with Shadows," in The Psychology of Computer Vision, edited by Patrick Henry Winston, McGraw-Hill, New York (1975).

7. Stephen A. Cook, "The Complexity of Theorem-Proving Procedures," Third Annual ACM Symposium on Theory of Computing, (1971), pp. 151-158.

8. Paul Purdom, "Tree Size by Partial Backtracking," SIAM J. Comp., v. 7 (1978).

9. Martin Davis and Hilary Putnam, "A Computational Procedure for Quantification Theory," JACM, v. 7 (1960), pp. 201-215.

10. Thomas J. Schaefer, "The Complexity of the Satisfiability Problem," Tenth Annual ACM Symposium on Theory of Computing, (1978), pp. 216-226.

11. David S. Johnson, "Approximation Algorithms for Combinatorial Problems," Journal of Computer and Systems Sciences, v. 9 (1974) pp. 256-278.

12. Albert L. Zobrist, "A Hashing Method with Application to Game Playing," Tech. Report 88, Computer Sci. Dept., U. of Wisconsin, Madison, Wis. (1970).

13. Cynthia A. Brown and Paul W. Purdom, Jr., "An Average Time Analysis of Backtracking," Indiana University Computer Science Department Tech. Report No. 86, November (1979).