Increasing Tree Search Efficiency for Constraint Satisfaction Problems

Robert M. Haralick and Gordon L. Elliott
Department of Electrical Engineering and Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VI 24061, U.S.A.

Recommended by Nils Nilsson

ABSTRACT

In this paper we explore the number of tree search operations required to solve binary constraint satisfaction problems. We show analytically and experimentally that the two principles of first trying the places most likely to fail and remembering what has been done to avoid repeating the same mistake twice improve the standard backtracking search. We experimentally show that a lookahead procedure called forward checking (to anticipate the future) which employs the most likely to fail principle performs better than standard backtracking, Ullman's, Waltz's, Mackworth's, and Haralick's discrete relaxation in all cases tested, and better than Gaschnig's backmarking in the larger problems.

1. Introduction

Associated with search procedures are heuristics. In this paper we provide a theory which explains why two heuristics used in constraint satisfaction searches work. The heuristics we discuss can be given a variety of one line descriptions such as:

—Lookahead and anticipate the future in order to succeed in the present.
—To succeed, try first where you are most likely to fail.
—Remember what you have done to avoid repeating the same mistake.
—Lookahead to the future in order not to worry about the past.

We will attempt to show that for a suitably defined random constraint satisfaction problem, the average number of tree search operations which employs these principles will be smaller than that required by the standard backtracking tree search.

To begin our discussion, we need a precise description of the constraint satisfaction problem we are attempting to solve by a search procedure. We

Artificial Intelligence 14 (1980) 263–313

assume that there are N units (some authors call these variables instead of units). Each unit has a set of M possible values or labels. The constraint satisfaction problem we consider is to determine all possible assignments f of labels to units such that for every pair of units, the corresponding label assignments satisfy the constraints. More formally, if $\boldsymbol{U}$ is the set of units and $\boldsymbol{L}$ is the set of labels, then the binary constraint R can be represented as a binary relation on $\boldsymbol{U} \times \boldsymbol{L}$: $R \subseteq (\boldsymbol{U} \times \boldsymbol{L}) \times (\boldsymbol{U} \times \boldsymbol{L})$. If a pair of unit-labels $(\boldsymbol{u}_1, l_1, \boldsymbol{u}_2, l_2) \in R$, then labels l_1 and l_2 are said to be consistent or compatible for units $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$. A labeling f of all the units satisfies the constraints if for every pair $\boldsymbol{u}_1$, $\boldsymbol{u}_2$ of units $(\boldsymbol{u}_1, f(\boldsymbol{u}_1), \boldsymbol{u}_2, f(\boldsymbol{u}_2)) \in R$. Haralick et al. [8] call such a labeling a consistent labeling.

The problem of determining consistent labelings is a general form of many problems related to artificial intelligence. For example, scene labeling and matching [1, 14], line interpretation [16], edge labeling [5], graph homomorphisms and isomorphisms [15], graph coloring [9], boolean satisfiability [8], and proposition theorem proving [10] are all special cases of the general consistent labeling problem. Ullman [15], Waltz [16], Rosenfeld et al. [14], Gaschnig [2, 3, 4], and McGregor [12] attempt to find efficient methods to solve the consistent labeling problem. Knuth [17] also analyzes the backtracking tree search, which is the basis of most methods used to solve the consistent labeling problem.

For the purpose of illustrating the search required to solve this problem, we choose the N-queens problem, how to place N-queens on an $N \times N$ checkerboard so that no queen can take another. Here, the unit set corresponds to the row coordinates on a checkerboard and we denote them by positive integers. The label set corresponds to the column coordinates on a checkerboard and we denote them by alphabetic characters. Hence, the unit-label pair (1, A, 2, D) satisfies the constraint R, [(1, A, 2, D) $\in R$], since a queen on row 1 column A cannot take a queen on row 2 column D. But, the unit-label pair (1, A, 3, C) does not satisfy the constraint R because queens can take each other diagonally (see Fig. 1).

Using the number letter convention for unit-label pairs, Fig. 2 illustrates a portion of a backtracking tree trace for the 6-queens problem. Notice how the unit 5 labels A, C, E, and F occur twice in the trace, each time being tested and failing for the same reason: incompatibility with units 1 or 2. These redundant tests can be eliminated if the fact they failed can be remembered or if units 1 or 2 could lookahead and prevent 5 from taking the labels A, C, E, or F. The remembering done by Gaschnig's backmarking [2] and the forward checking approach described in this paper help eliminate these problems. Notice that once unit 3 takes label E (Fig. 1(a)) the only labels left for units 4 and 6 are incompatible. The forward checking algorithm will not discover this future incompatibility. However, the first time label B is associated with unit 4, there

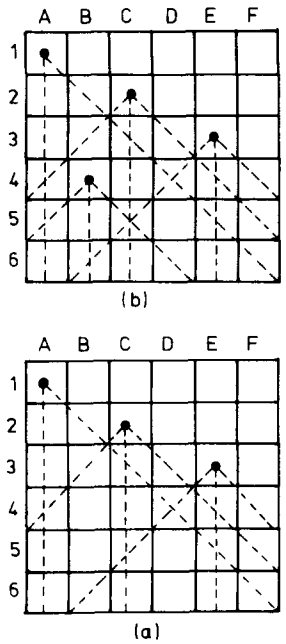

FIG. 1. (a) illustrates how the labeling A, C, E for units 1, 2, 3 implies that the only labels for units 4 and 6 are incompatible in the 6 queens problem. (b) illustrates how the labeling A, C, E, B for units 1, 2, 3, 4 implies that there is no label for unit 6 in the 6 queens problem.

```
1 A
  2 A,B
  2 C
    3 A,B,C,D
    3 E
      4 A
      4 B
        5 A,B,C
        5 D
          6 A,B,C,D,E,F
        5 E,F
      4 C,D,E,F
    3 F
      4 A
      4 B
        5 A,B,C,D,E,F
      4 C,D,E,F
```

FIG. 2. A segment of a tree trace that the standard backtracking algorithm produces for a 6 queens problem. No solutions are found in this segment. The entry 2 A,B, for example, indicates that labels A and B were unsuccessful at level 2, but 2 C succeeds when checked with past units, and the tree search continues with the next level.

is absolutely no label possible for unit 6. Hence, the search through the labels for 5 and 6 are entirely superfluous and forward checking will discover this (Fig. 1(b)). The lookahead procedures (discrete relaxation) of [11, 13, 14, 15, 16] help alleviate the problem illustrated in Fig. 1(a) as well as in Fig. 1(b).

Section 2 gives a description of the full and partial looking ahead, forward checking, backchecking, and backmarking procedures. In Section 3 we compare the complexity of these algorithms as they solve the N-queens problem and problems generated randomly. We measure complexity in terms of number of table lookups and number of consistency checks. These results show that standard backtracking is least efficient in most cases and bit parallel forward checking is most efficient for the cases tried.

In Section 4, we give a statistical analysis of constraint satisfaction searches and demonstrate the statistical reason why forward checking requires fewer expected consistency checks than standard backtracking. In Section 5 we explore other applications of the fail first or prune early tree search strategies and show that such particular strategies as choosing the next unit to be that unit having fewest labels left and testing first against units whose labels are least likely to succeed reduce the expected number of consistency tests required to do the tree search. Finally, by changing the unit search order dynamically in every tree branch so that the next unit is always the one with fewest labels left, we show experimentally that performance improves for each procedure and that forward checking even increases its computational advantage over the other algorithms.

2. Some Procedures for Tree Search Reducing

In this section we give brief descriptions of five procedures, and a variation of data structure in one, which can be used within the standard backtracking framework to reduce tree search operations. They are called full and partial looking ahead, forward checking, backchecking, and backmarking. Each of these procedures invests resources in additional consistency tests or data structures at each point in the tree search in order to save (hopefully) more consistency tests at some point later in the tree search.

For ease in explaining these procedures, we call those units already having labels assigned to them the past units. We call the unit currently being assigned a label the current unit and we call units not yet assigned labels the future units. We assume the existence of a unit-label table which at each level in the tree search indicates which labels are still possible for which units. Past units will of course have only one label associated with each of them. Future units will have more than one. The tree search reducing procedures invest early to gain later. Hence, the result of applying any of them in the tree search will be to decrease the number of possible labels for any future unit or reduce the number of tests against past units.

2.1. Looking ahead

Waltz filtering [16], a procedure by Ullman [15], discrete relaxation [14], and the Ψ operator of Haralick et al. [8] are all examples of algorithms that look ahead to make sure that (1) each future unit has at least one label which is compatible with the labels currently held by the past and present units and (2) each future unit has at least one label which is compatible with one of the possible labels for each other future unit. Looking ahead prevents the tree search from repeatedly going forward and then backtracking between units u and v, $v < u$, only to ultimately discover that the labels held by units 1 through v cause incompatibility of all labels between some unit w, $w > u$, and some past, current, or future unit.

Because looking ahead in this manner cannot remember and save most of the results of tests performed in the lookahead of future units with future units for use in future lookaheads, the full savings of looking ahead are not realized for many problems. A partial look ahead that does not do all the checks of full look ahead will perform better, and one that checks only future with present units (neglects future with futures) will do much better because all tests it performs can be usefully remembered.

The procedure L_A_TREE SEARCH and its associated subroutines CHECK_FORWARD and LOOK_FUTURE (Fig. 3(a), (b) and (c)) is a formal description of the full looking ahead algorithm, which can easily be translated into any structured recursive language. U is an integer representing the unit, and will increment at each level of the tree search. It takes on the value 1 at the initial call. F is a one dimension array indexed by unit, where entry $F(u)$ for unit u is the label assigned to u. T and NEW_T are tables, which can be thought of as an array of lists, $T(u)$ is a list of labels which have not yet been determined to be not possible for unit u. (We implemented T as a 2 dimension array, with the number of entries in each list (or row) stored in the first position of the row. This implementation uses approximately (NUMBER_OF_UNITS)^2 × (NUMBER_OF_LABELS) words of memory for table storage since there can be NUMBER_OF_UNITS levels of recursion.) The tree search is initially called with T containing all labels for each unit. All other variables can be integers. EMPTY_TABLE and NUMBER_OF_UNITS and NUMBER_OF_LABELS have obvious meanings.

The function RELATION(u_1, l_1, u_2, l_2) returns *TRUE* if $(u_1, l_1, u_2, l_2) \in R$, otherwise it returns *FALSE*. CHECK_FORWARD checks that each future unit-label pair is consistent with the present label $F(u)$ for unit u as it copies the table T into the next level table NEW_T, LOOK_FUTURE then checks that each future unit-label pair in NEW_T is consistent with at least one label for every other unit, and deletes those that are not.

In this implementation CHECK_FORWARD and LOOK_FUTURE return a flag, EMPTY_ROW_FLAG, if a unit is found with no possible consistent

```
1. RECURSIVE PROCEDURE L_A_TREE_SEARCH(U, F, T);
2. FOR F(U) = each element of T(U) BEGIN
3.   IF U < NUMBER_OF_UNITS THEN BEGIN
4.     NEW_T = CHECK_FORWARD(U, F(U), T);
5.     CALL LOOK_FUTURE(U, NEW_T);
6.     IF NEW_T is not EMPTY_ROW_FLAG THEN
7.       CALL L_A_TREE_SEARCH(U + 1, F, NEW_T);
8.     END;
9.   ELSE
10.    Output the labeling F;
11.  END;
12. END L_A_TREE_SEARCH;
```

(a)

```
1. PROCEDURE CHECK_FORWARD(U, L, T);
2. NEW_T = empty table;
3. FOR U2 = U + 1 TO NUMBER_OF_UNITS BEGIN
4.   FOR L2 = each element of T(U2)
5.     IF RELATION(U, L, U2, L2) THEN
6.       Enter L2 into the list NEW_T(U2);
7.   IF NEW_T(U2) is empty THEN
8.     RETURN (EMPTY_ROW_FLAG); /* No consistent labels */
9.   END;
10. RETURN (NEW_T);
11. END CHECK_FORWARD;
```

(b)

```
1. PROCEDURE LOOK_FUTURE(U, NEW_T);
2. IF U + 1 ≧ NUMBER_OF_UNITS THEN RETURN;
3. FOR U1 = U + 1 TO NUMBER_OF_UNITS BEGIN
4.   FOR L1 = each element of NEW_T(U1)
5.     FOR U2 = U + 1 TO NUMBER_OF_UNITS except skipping U1 BEGIN
6.       FOR L2 = each element of NEW_T(U2)
7.         IF RELATION(U1, L1, U2, L2) THEN
8.           BREAK for L2 loop; /* consistent label found */
9.       IF no consistent label was found for U2 THEN BEGIN
10.        Delete L1 from list NEW_T(U1);
11.        BREAK for U2 loop; /* unit U2 has no label consistent with U1, L1 */
12.        END;
13.      END for U2 loop;
14.    END for L1 loop;
15.  IF NEW_T(U1) is empty THEN BEGIN
16.    NEW_T = EMPTY_ROW_FLAG;
17.    RETURN;
18.    END;
19.  END for U1 loop;
20. RETURN;
21. END LOOK_FUTURE;
```

(c)

FIG. 3 (continued).

```
1. PROCEDURE PARTIAL_LOOK_FUTURE(U, NEW_T);
2. IF U+1 ≧ NUMBER_OF_UNITS THEN RETURN;
3. FOR U1 = U+1 TO NUMBER_OF_UNITS - 1 BEGIN
4.   FOR L1 = each element of NEW_T(U1)
5.     FOR U2 = U1 + 1 TO NUMBER_OF_UNITS BEGIN
6.       FOR L2 = each element of NEW_T(U2)
7.         IF RELATION(U1, L1, U2, U2) THEN
8.           BREAK for L2 loop; /* consistent label found */
9.       IF no consistent label was found for U2 THEN BEGIN
10.        Delete L1 from list NEW_T(U1);
11.        BREAK for U2 loop; /* unit U2 has no label consistent with U1, L1 */
12.        END;
13.      END for U2 loop;
14.    END for L1 loop;
15.  IF NEW_T(U1) is empty THEN BEGIN
16.    NEW_T = EMPTY_ROW_FLAG;
17.    RETURN;
18.    END;
19.  END for U1 loop;
20. RETURN;
21. END PARTIAL_LOOK_FUTURE;
```

(d)

FIG. 3. (a), (b), and (c) express the full looking ahead algorithm. Replace the call to LOOK_FUTURE at line 5 in (a) with an identical call to PARTIAL_LOOK_FUTURE (d) and the partial looking ahead algorithm is obtained. Forward checking consists of (a) and (b) with line 5 of (a), the call to LOOK_FUTURE, deleted so that only CHECK_FORWARD is called. Forward checking does no checks of future units with future units. (c) is the LOOK_FUTURE procedure, which deletes future labels which are not consistent with at least one label for every unit other than the label's own unit. (d) is the PARTIAL_LOOK_FUTURE procedure. It differs from LOOK_FUTURE (c) only at lines 1, 3, and 5. Each future unit-label pair is checked only with units in its own future.

labels. Thus the next level of the tree search will not be called, otherwise each entry in NEW_T is consistent with $\boldsymbol{u}$, $F(\boldsymbol{u})$, and therefore, all the past unit-label pairs.

2.2. Partial looking ahead

Partial looking ahead is a variation of looking ahead which does approximately half of the consistency checks that full looking ahead does while checking future with future units. Each future unit-label pair is checked only with units in its own future, rather than all other future units. Thus partial looking ahead is less powerful than full looking ahead in the sense that it will not delete as many unit-label pairs from the lists of potential future labels. We will, however, see that partial looking ahead does fewer total consistency checks than full looking ahead in all cases tested.

The checks of future with future units do not discover inconsistencies often enough to justify the large number of tests required, and these results cannot be usefully remembered. Since partial looking ahead does fewer of these less useful tests, it is more efficient. A look ahead that checks only future with current or past units can have better performance since these more powerful tests can also be usefully remembered.

The formal algorithm for partial looking ahead is L_A_TREE_SEARCH (Fig. 3(a)), with the call to LOOK_FUTURE on line 5 replaced with an identical call to PARTIAL_LOOK_FUTURE (Fig. 3(d)).

2.3. Forward checking

Forward checking is a partial lookahead of future units with past and present units, in which all consistency checks can be remembered for a while. This method is similar to looking ahead, except that future units are not checked with future units, and the checks of future units with past units are remembered from checks done at past levels in the tree search. Forward checking begins with a state of affairs in which there is no future unit having any of its labels inconsistent with any past unit-label pairs. This is certainly true at the root of the tree search, since there are no past units with which to be inconsistent. Because of this state of affairs, to get the next label for the current unit, forward checking just selects the next label from the unit table for the current unit. That label is guaranteed to be consistent with all past unit-label pairs. Forward checking tries to make a failure occur as soon as possible in the tree search by determining if there is any future unit having no label which is consistent with the current unit-label pair. If each future unit has consistent labels, it remembers by copying all consistent future unit-label pairs to the next level's unit-label table. If every future unit has some label in the unit label table which is consistent with the current unit-label pair, then the tree search can move forward to the next unit with a state of affairs similar to how it started. If there is some future unit having no label in the unit label table which is consistent with the current unit-label pair, then the tree search remains at the current level with the current unit and continues by selecting the next label from the table. If there is no label, then it backtracks to the previous unit and the previous label table.

The formal algorithm for forward checking is the Procedure L_A_TREE_SEARCH (Fig. 3(a)) with line 5, the call to LOOK_FUTURE, removed. Forward checking is just looking ahead, omitting the future with future checks.

An improvement in efficiency can be gained in the lookahead type of algorithms by using a data structure for the unit-label tables that is suggested by McGregor [12]. McGregor simultaneously developed a weaker form of the forward checking algorithm and compared them on subgraph isomorphism

problems. In L_A_TREESEARCH and CHECK_FORWARD the lists $T(u)$ or NEW_$T(u)$ can be stored as bit vectors, with one bit in the machine word for each possible label (this is essentially a set representation). Lines 4, 5 and 6 in CHECK_FORWARD (Fig. 3(b)) can then be replaced with the following statement:

$$\text{NEW\_}T(U2) = AND(T(U2), \text{RELATION\_BIT\_VECTOR}(U, L, U2)).$$

This single statement replaces a loop, taking advantage of the parallel bit handling capabilities of most computers. RELATION_BIT_VECTOR(u, l, u_2) returns a bit vector with a bit on in each position corresponding to a label $l2$ for which RELATION($u, l, u2, l2$) would have been true. This is essentially the set $\{l_2 \mid (u, l, u_2, l_2) \in R\}$. Thus NEW_$T(u_2)$ becomes the set $\{l_2 \in T(u_2) \mid (u, l, u_2, l_2) \in R\}$, precisely what lines 4, 5, and 6 do. If the number of labels is less than the word length of the computer used, then the relation can be directly stored in an array of size (NUMBER_OF_UNITS)$^2 \times$ (NUMBER_OF_LABELS), and the tables T and NEW_T will take approximately (NUMBER_OF_UNITS)^2 words of storage. Reduced forms of the relation exist for some problems, such as the N-queens problem or the subgraph isomorphism problem, in which only two dimensional tables need be stored and a quick calculation will generate the needed list. The same technique can be applied to the full and partial looking ahead algorithms, but they will not be compared here since the three algorithms will have approximately the same relationships of efficiency, to each other, with or without the improved data structure.

2.4. Backchecking

Backchecking is similar to forward checking in the way it remembers unit-label pairs which are known to be inconsistent with the current or any previous unit label. However, it keeps track of them by testing the current unit label only with past unit-label pairs and not future ones. So if, for instance, labels A, B, and C for unit 5 were tested and found incompatible with label B for unit 2, then the next time unit 5 must choose a label, it should never have A, B, or C as label possibilities as long as unit 2 still has the label B.

Each test that backchecking performs while looking back from the current unit u to some past unit v, forward checking will have performed at the time unit v was the current unit. Of course, at that time, forward checking will also have checked all future units beyond unit u. Hence, backchecking performs fewer consistency tests, an advantage. But backchecking pays the price of having more backtracking and at least as large a tree as forward checking. Backchecking itself is not as good as forward checking.

2.5. Backmarking

Backmarking (defined in Gaschnig [2] and also discussed in Gaschnig [3]) is backchecking with an added feature. Backmarking eliminates performing some consistency checks that were previously done, had not succeeded, and if done again would again not succeed. Backmarking also eliminates performing some consistency checks that were previously done, had succeeded, and if done again would again succeed. To understand how backmarking works, recall that the tree search by its very nature goes forward, then backtracks, and goes forward again. We focus our attention on the current unit u. We let v be the lowest ordered unit to which we have backtracked (has changed its label) since the last visit to the current unit u. Backmarking remembers v. If $v = u$, then backmarking proceeds as backchecking. If $v < u$, then since all the labels for unit u had been tested in the last visit to unit u, any label now needing testing, needs only to be tested against the labels for units v to $u - 1$, which are the ones whose labels have changed since the last visit to unit u. That is, the tests done previously against the labels for units 1 through $v - 1$ were successful and if done again would again be successful because labels for units 1 through $v - 1$ have not changed and the only labels permitted for the current unit u are those which have passed the earlier tests (see Fig. 4).

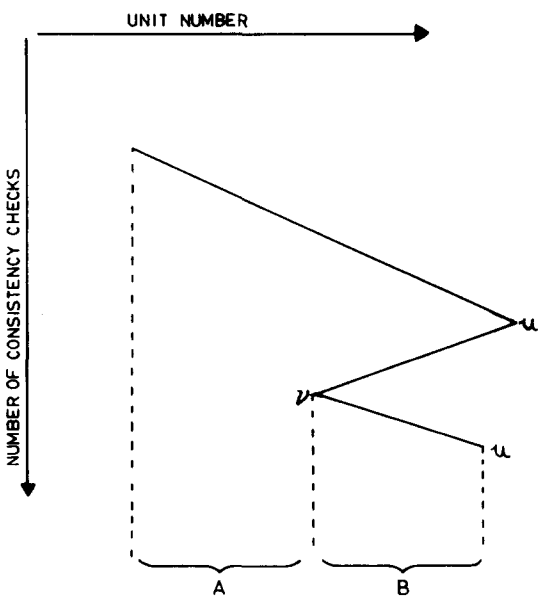

FIG. 4. A segment of a tree trace. Because backmarking remembers from the first visit to unit u which labels for u were compatible with the labels for units 1 through $v - 1$ (segment A) do not have to be performed. Only those for units v through $u - 1$ (segment B) have to be performed.

The formal algorithm for Backmarking appears in Fig. 5. It is essentially Gaschnig's algorithm [2], but modified to find all solutions. The variable U and array F are the same as in looking ahead. LOWUNIT is a one dimensional array of NUMBER_OF_UNITS entries, and LOWUNIT(i) will indicate the lowest level at which a change of label has occurred since the last time the MARK array is set. MARK is dimensioned NUMBER_OF_UNITS by NUMBER_OF_LABELS, and MARK(u, l) will indicate the lowest level at which a consistency test failed when the unit-label pair (u, l) at the current level was last tested against the previous unit-label pairs on previous levels. At any point if MARK(u, l) is less than LOWUNIT(u), then the algorithm knows that (u, l) has already been tested against the unit-label pairs at levels below the value in LOWUNIT(u) and will fail at level MARK(u, l), so there is no need to repeat the tests. If MARK(u, l) is greater or equal to LOWUNIT(u), then all tests will succeed below and level LOWUNIT(u) and only tests against units at LOWUNIT(u) to the current unit need be tested.

Before the initial call to BACKMARK, all entries in LOWUNIT and MARK are initialized to 1, and BACKMARK is called with the initial $u = 1$. Since the same MARK and LOWUNIT arrays are used at all levels of recursion of the tree search, approximately (NUMBER_OF_UNITS)× (NUMBER_OF_LABELS) words of table storage are needed.

```
 1. RECURSIVE PROCEDURE BACKMARK(U, F, MARK, LOWUNIT);
 2. FOR F(U) = 1 TO NUMBER_OF_LABELS BEGIN
 3.   IF MARK(U, F(U)) ≧ LOWUNIT(U) THEN BEGIN
 4.     TESTFLAG = TRUE;
 5.     I = LOWUNIT(U);
 6.     WHILE (I < U) BEGIN /* Find lowest failure */
 7.       TESTFLAG = RELATION(I, F(I), U, F(U));
 8.       IF NOT TESTFLAG THEN BREAK while loop;
 9.       I = I + 1;
10.       END while loop;
11.     MARK(U, F(U)) = I; /* Mark label with lowest failure */
12.     IF TESTFLAG THEN
13.       IF U < NUMBER_OF_UNITS THEN
14.         CALL BACKMARK(U + 1, F, MARK, LOWUNIT);
15.       ELSE
16.         Output the labeling F;
17.     END;
18.   END for F loop;
19. LOWUNIT(U) = U − 1; /* Previous level will now change */
20. FOR I = U + 1 TO NUMBER_OF_UNITS;
21.   LOWUNIT(I) = MIN(LOWUNIT(I), U − 1);
22. RETURN;
23. END BACKMARK;
```

FIG. 5. Gaschnig's backmarking procedure as it was modified to find all solutions to constraint satisfaction problems (see [2]).

3. Experimental Results

In this section we compare the six procedures, partial and full looking ahead, backtracking, backchecking, forward checking, and backmarking, on the N-queens problem for $4 \leq N \leq 10$, and on a random constraint problem. We assume that the unit order is fixed in its natural order from 1 to N and that all consistency tests of the current unit with past units or future units begin with the lowest ordered unit. The label sets will consist of all N columns; no consideration is given to the various symmetries peculiar to the N-queens problem.

The random constraint problems are generated using a pseudorandom number generator. A random number is generated for each possible consistency check (u_1, l_1, u_2, l_2) for the relation R, so that each entry in the relation will be made with probability p. A probability of $p = 0.65$ is chosen so that problems will be generated that are somewhat similar to the N-queens problem.

The comparison among the tree search reducing procedures indicates that backtracking is least efficient in most cases, and that backmarking and forward checking are more efficient for the cases tested. Bit parallel forward checking, which takes advantage of machine parallelism, is the most efficient for all cases tried.

Our comparison of algorithm complexity will be in terms of nine criteria involving number of consistency tests, number of table lookups, and number of nodes in the tree search. There are a variety of ways of presenting these results including

(1) Number of consistency tests performed to obtain all solutions (Figs. 6 and 8).

(2) Number of table lookups used in finding all solutions (Figs. 10 and 11).

(3) Number of nodes in the tree search to obtain all solutions (Fig. 12).

(4) Number of nodes visited at each level in the tree search (Fig. 13).

(5) Number of nodes found to be consistent at each level in the tree search, or consistent labelings to depth (Figs. 17 and 18).

(6) Number of consistency checks at each level in the tree search (Fig. 19).

(7) Number of table lookups at each level in the tree search (Fig. 20).

(8) Percentage of nodes at each depth that fail because an inconsistency was found at that depth (Figs. 21 and 22).

(9) Average number of table lookups per consistency check (Figs. 23 and 24).

Fig. 6 indicates that the number of consistency tests performed to obtain all solutions seems to increase exponentially with N for the N-queens problem. The number of solutions to the N-queens problem also appears to increase exponentially (see Fig. 7). The number of bit vector operations is also shown,

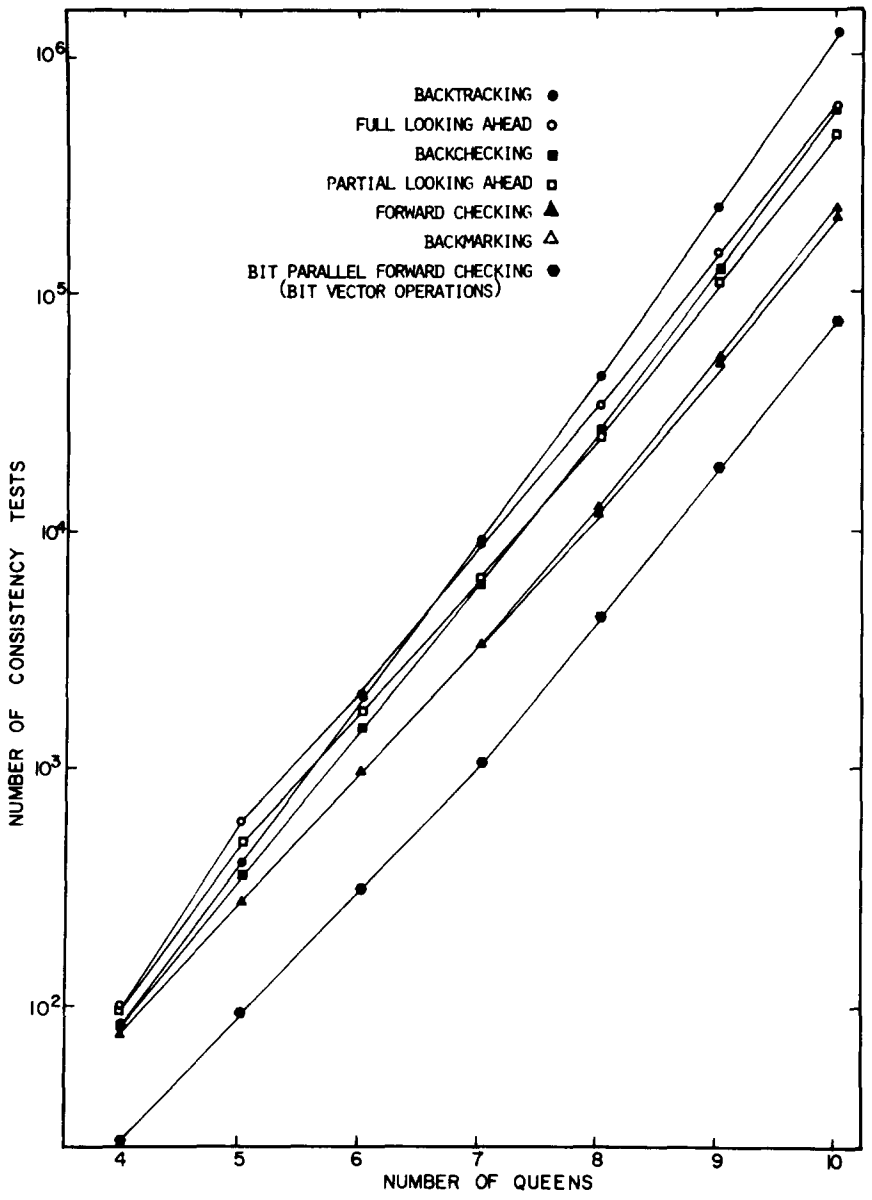

FIG. 6. This figure compares the efficiency of the standard backtracking procedure with backchecking, looking ahead, forward checking, and backmarking, for the N-queens problem in the natural unit order. The number of bit vector operations for bit parallel forward checking is also shown.

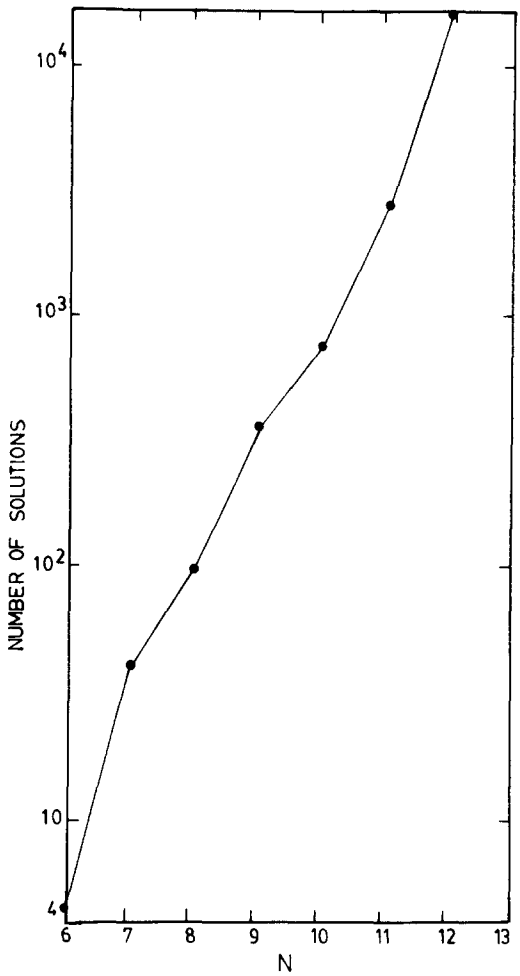

FIG. 7. The number of solutions for the N-queens problem.

for forward checking done in the bit vector data structure. Though backmarking appears to do slightly fewer consistency checks than forward checking on the N-queens problem, the use of machine parallelism gives bit parallel forward checking a clear advantage over all the other algorithms.

Random constraint problems with fixed probability 0.65 of consistency check success probability of 0.65 are tested in Figs. 8 and 9. The number of consistency tests appears to grow exponentially in Fig. 8, until a sufficiently large problem size is reached. At this point the number of solutions drops, as is indicated in Fig. 9, and the number of consistency tests appears to grow more slowly. Fig. 9 explains the uneveness of the curves in Fig. 8. Too few random

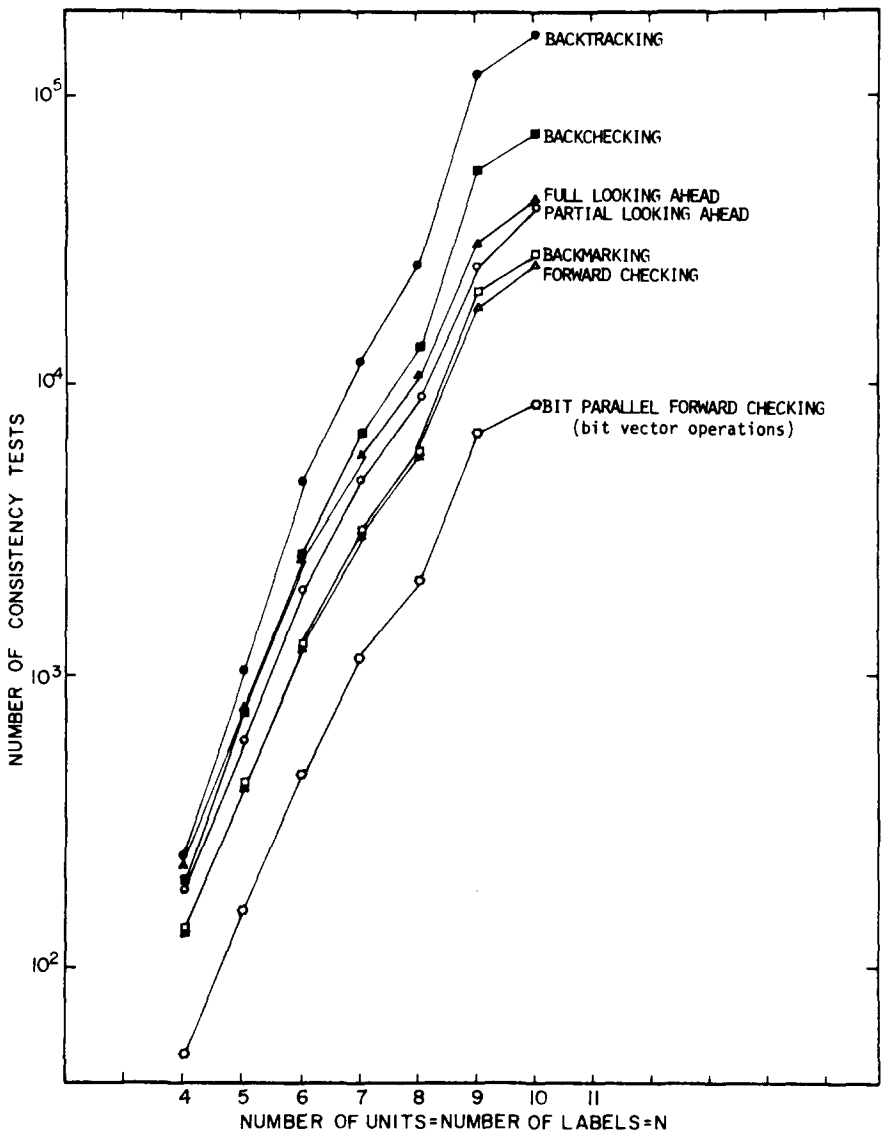

FIG. 8. The number of consistency tests in the average of 5 runs of the indicated programs. Relations are random with consistency check probability $p = 0.65$ and number of units = number of labels = N. Each random relation is tested on all 6 methods, using the same 5 different relations generated for each N. The number of bit-vector operations in bit parallel forward checking is also shown for the same relations.

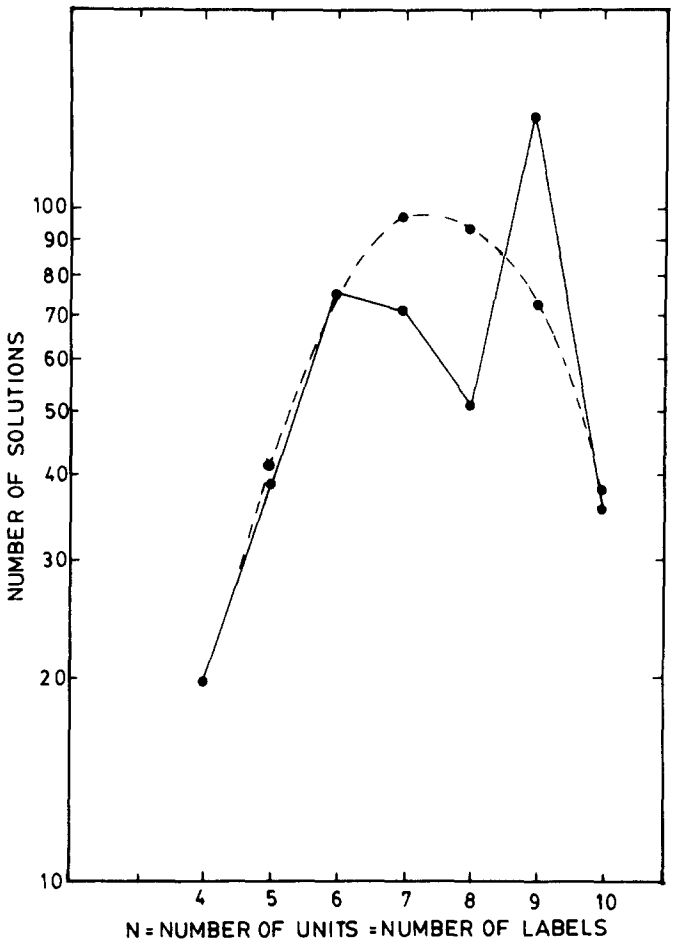

FIG. 9. The average number of solutions for the probabilistic relations in Fig. 8. This is the average of 5 experiments for each problem size, and relations are random with consistency check success probability $p = 0.65$. The dotted curve is the expected number of solutions.

relations were tested for the means to settle closely to the expected values for this type of problem, and the average number of solutions varies eratically high and low.

In the random problems, forward checking does slightly fewer consistency checks than backmarking in the larger problem sizes, and once again machine parallelism gives bit parallel forward checking a clear advantage.

The number of table lookups for the N-queens and random relation problems are compared in Figs. 10 and 11. Only the lookups in the MARK array in backmarking and backchecking, and the T or NEW_T tables in the lookahead

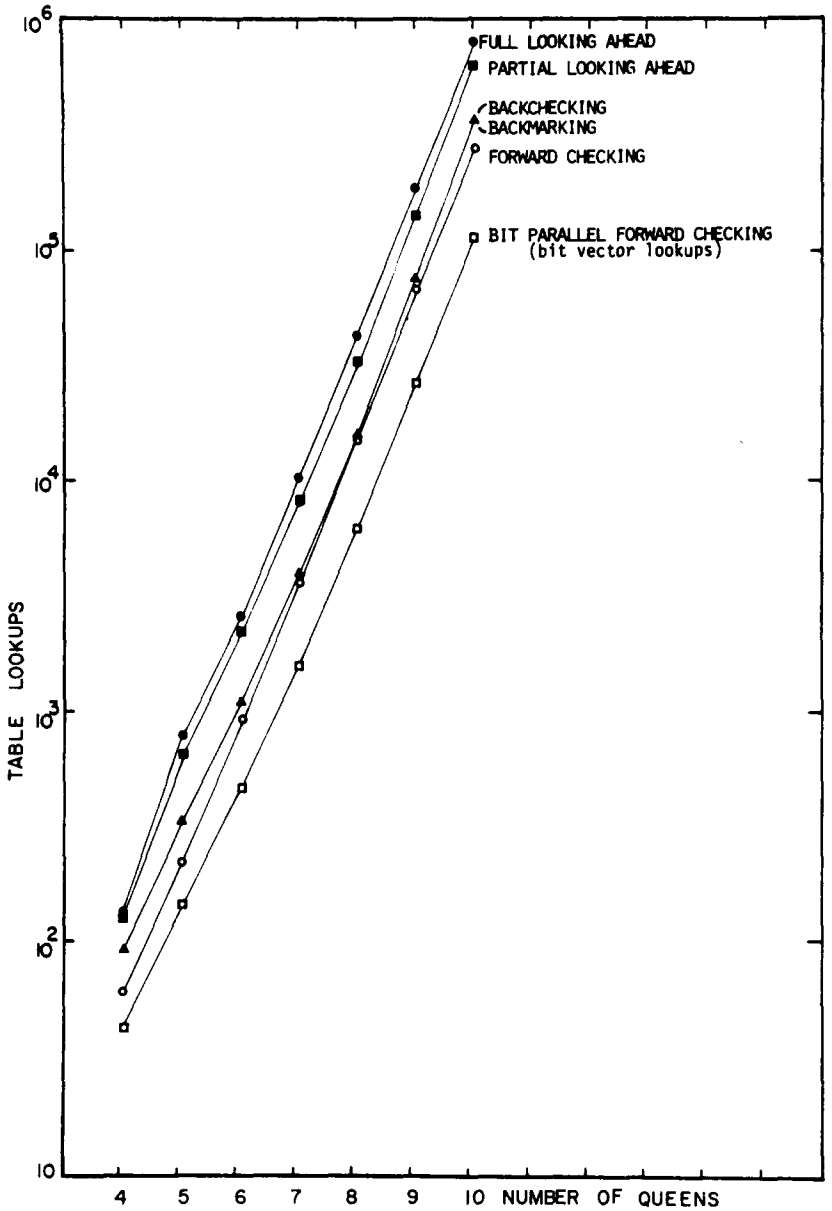

FIG. 10. The number of table lookups used to find all solutions to the N-queens problem for varying N, with the natural unit order.

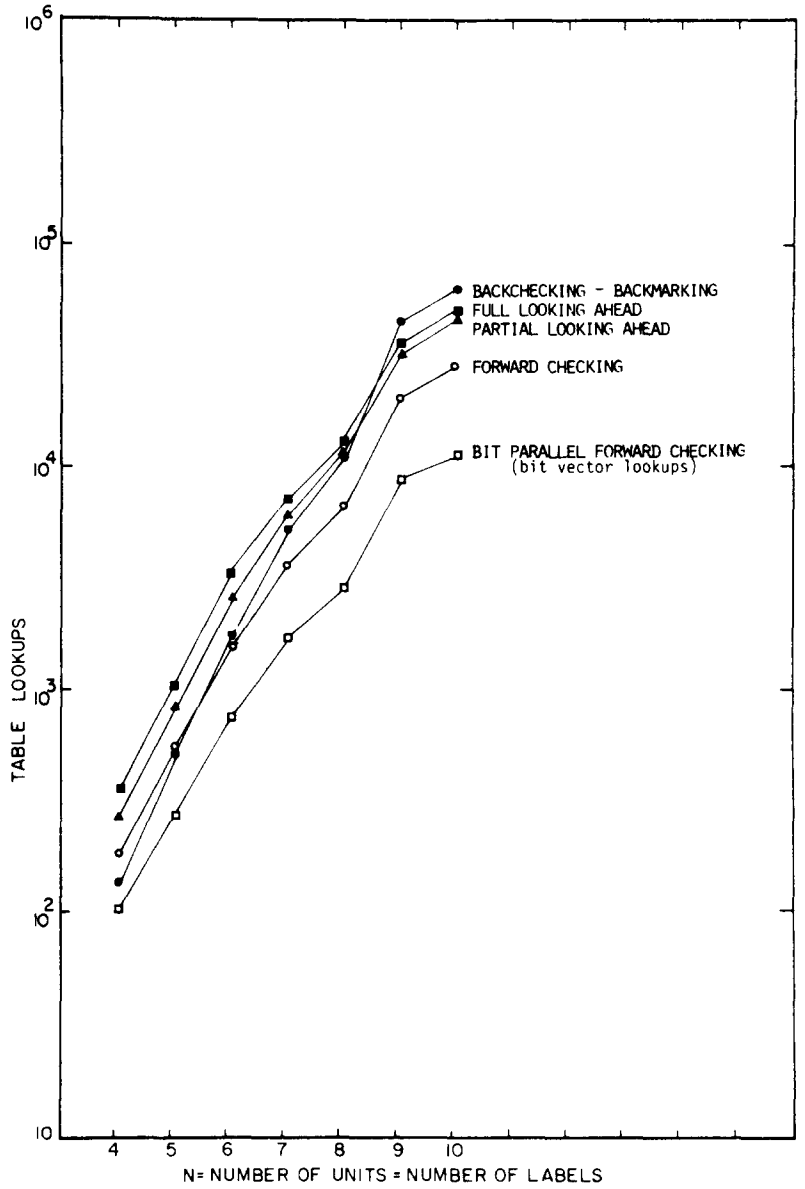

FIG. 11. The number of table lookups for the average of 5 random relations at each N, with number of units = number of labels = N, and probability of consistency check success $p = 0.65$. These random problems are the same as shown in Figs. 8 and 9.

type algorithms are considered. These table lookups occur at line 3 in Fig. 5, backmarking, line 2 in Fig. 3(a), line 4 in Fig. 3(b), and lines 4 and 6 in each of Figs. 3(c) and 3(d), the lookahead type algorithms. The entering of values into the tables are not considered, since they always follow at least one consistency check, and never happen more often than consistency checks.

Backtracking is not shown, since it does no table lookups of the type considered. Partial and full looking ahead always do more table lookups than forward checking in these cases, and forward checking does better than backmarking in the larger problem sizes. Even full looking ahead does fewer table lookups than backmarking in the larger random problems. The number of table lookups into bit vectors is smaller than the number of table lookups in other algorithms, when the bit parallel data structure is used in forward checking.

Fig. 12 demonstrates that full looking ahead visits the fewest nodes in the tree search, since it eliminates the most potential nodes during its examination of future unit-label pairs. Fig. 13 indicates that the number of nodes visited in the tree search is largest for the middle levels in the tree search, with the full looking ahead procedure having the fewest nodes at each level.

Figs. 14, 15, and 16 show segments of the trace of nodes visited by the full and partial looking ahead, and forward checking algorithms for the 6-queens problem. Backmarking and backchecking will have the same tree trace as backtracking (see Fig. 2). More detailed trace of the action of backmarking can be found in [3, 4].

Fig. 17 shows the number of consistent labelings at each depth of the tree search for the 8-queens problem, and Fig. 18 shows the average number of consistent labelings for random problems. This is the number of nodes at each level which have not yet been found to be inconsistent. Backmarking and backchecking will have the same search tree as backtracking, and consequently has the same number of nodes and consistent labelings at each depth (see Figs. 13, 17, and 18). Their efficiencies are gained by reducing the amount of work spent at each node, checking against past units. However, the lookahead algorithms perform extra work at each node to reduce the number of nodes, and as Figs. 19 and 20 show, the relation checks and table lookups for the lookahead type algorithms are concentrated more at the shallow depths of the treesearch. As Figs. 6, 8, 10, and 11 show, full and partial looking ahead do too much work at each node for the problems shown, and forward checking and backmarking do better.

The percentage of nodes at each depth in the tree search that fail because some inconsistency is discovered are shown in Figs. 21 and 22. In the cases shown, in the backtracking, backchecking, and backmarking algorithms, over 95% of the nodes (instantiated labels) fail at the deepest level of the treesearch, because they are inconsistent with some past unit-label pair.

The lookahead type algorithms reduce the number of nodes in the treesearch

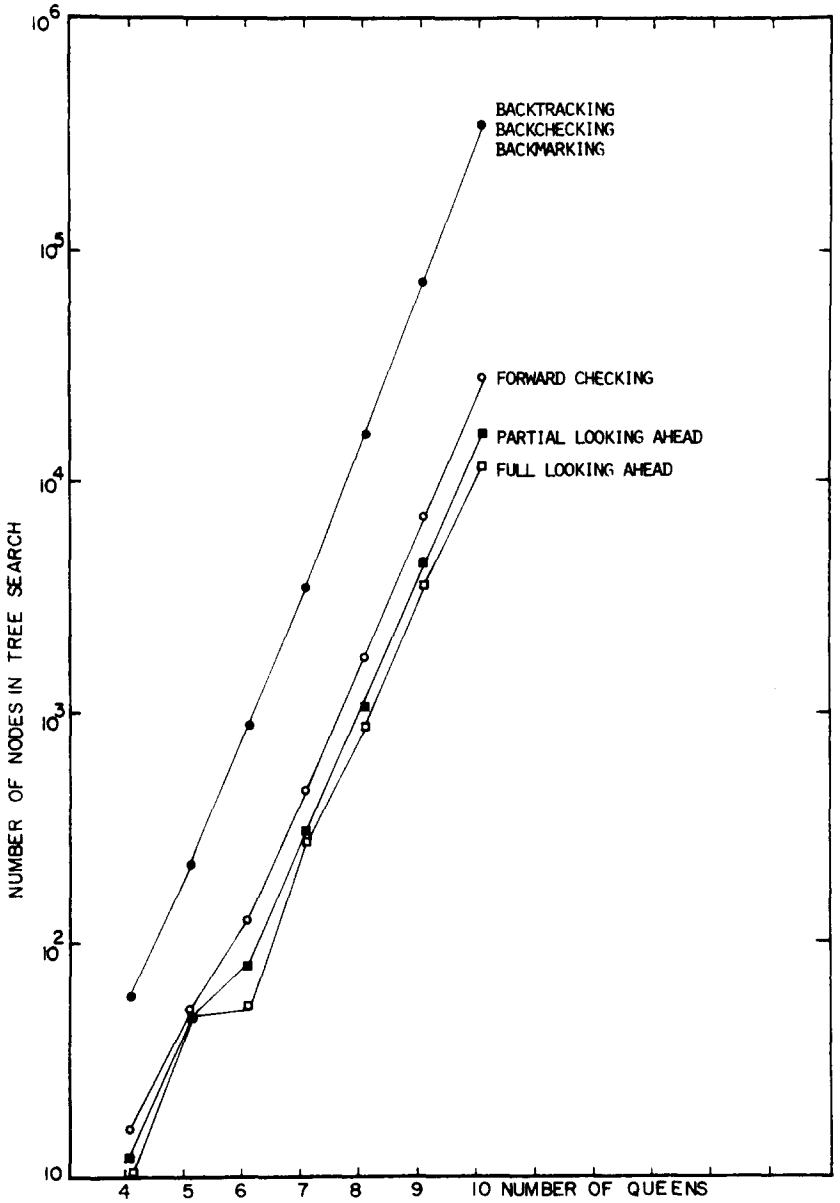

FIG. 12. The number of nodes in the treesearch to find all solutions to the N-queens problem for varying N.

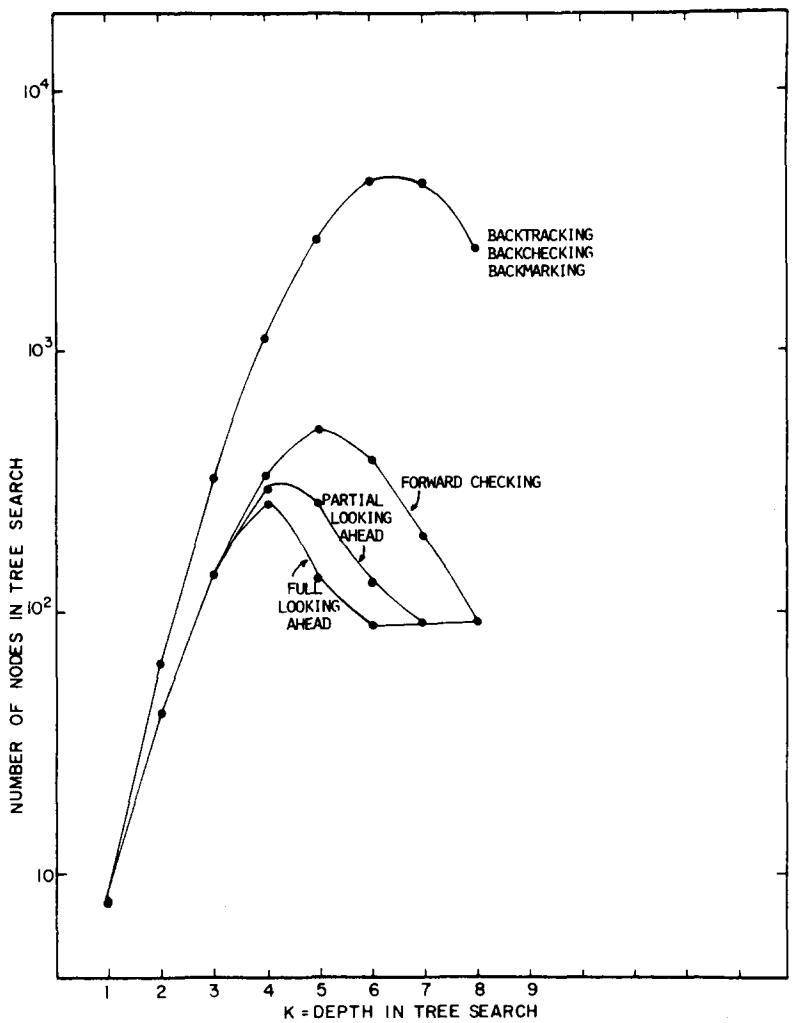

FIG. 13. The number of nodes visited at each depth of the treesearch to find all solutions of the 8-queens problem, for the various algorithms.

in two ways. First by removing entries from the tables of potential unit-label pairs, and second by noticing that some future units may have no possible labels associated with them. In Figs. 21 and 22 all nodes that fail in the lookahead type algorithms do so for this second reason, since they would not have occurred as nodes if their labels were deleted from the future unit-label tables. If lines 7 and 8 are removed from the CHECK_FORWARD procedure in Fig. 3(b), then no nodes would fail in forward checking (this is McGregor's restricted arc consistency algorithm in [12]). This weaker form of forward

```
1 A
  2 C
  2 D
    3 B,F
  2 E,F
1 B
  2 D
    3 A
    3 F
      4 A
        5 C
          6 E
  2 E
    3 A,C
  2 F
```

FIG. 14. A segment of a tree trace made by the full looking ahead procedure in a 6-queens problem. One consistent labeling, 1B 2D 3F 4A 5C 6E, appears in this portion of the trace. 1A 2C fails to spawn any further nodes because the LOOK_FUTURE algorithm will, after deleting several potential labels, discover that one future unit has no possible labels.

```
1 A
  2 C
    3 E,F
  2 D
    3 B,F
  2 E
    3 B
  2 F
1 B
  2 D
    3 A
      4 C
    3 F
      4 A
        5 C
          6 E
  2 E
    3 A
      4 F
    3 C
  2 F
```

FIG. 15. A segment of a tree trace showing the nodes of the tree search in the partial looking ahead procedure in a 6-queens problem. One consistent labeling, 1B 2D 3F 4A 5C 6E, appears in this portion of the trace. 1A 2C 3A,B,C, and D do not appear because CHECK_FORWARD removes them from the table at nodes 1A and 2C. However 1A 2C 3E fails to have successors because the only labels left for future units 4 and 6 are incompatible and are removed by LOOK_FUTURE (see Fig. 1(a)).

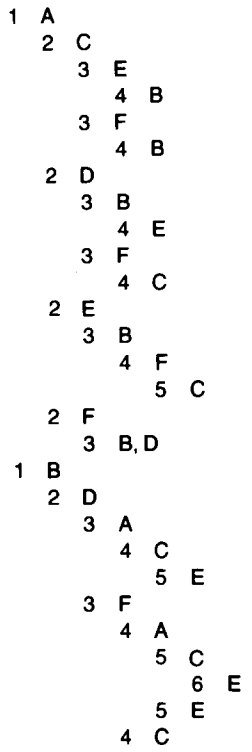

FIG. 16. A segment of a tree trace made by the forward checking procedure in a 6-queens problem. One consistent labeling, 1B 2D 3F 4A 5C 6E, is found. Notice that 1A 2C 3E 4B fails because the CHECK_FORWARD procedure discovers that there are no labels remaining for unit 6 at the 4B node (see Fig. 1(b)).

checking algorithm will find all the consistent labelings at each level of the tree search that backtracking does, but at a higher cost than the original forward checking algorithm. The replacement of these lines in forward checking will realize a 15% saving of consistency checks and table lookups in the 10-queens problem, and over 40% savings in the 10 units by 10 labels random problem.

Figs. 23 and 24 address the question of what measure best determines the algorithmic time complexity. A careful check of all the algorithms will show that no step is executed more often than the maximum of the number of consistency checks or the number of table lookups. As the problem size increases in the lookahead type of algorithms, the ratio of table lookups to consistency checks seems to decrease from a maximum of about 2 to no more than 1.5 table lookups per consistency check in both the N-queens and random problems. This ratio is guaranteed to be greater than or equal to one, by the

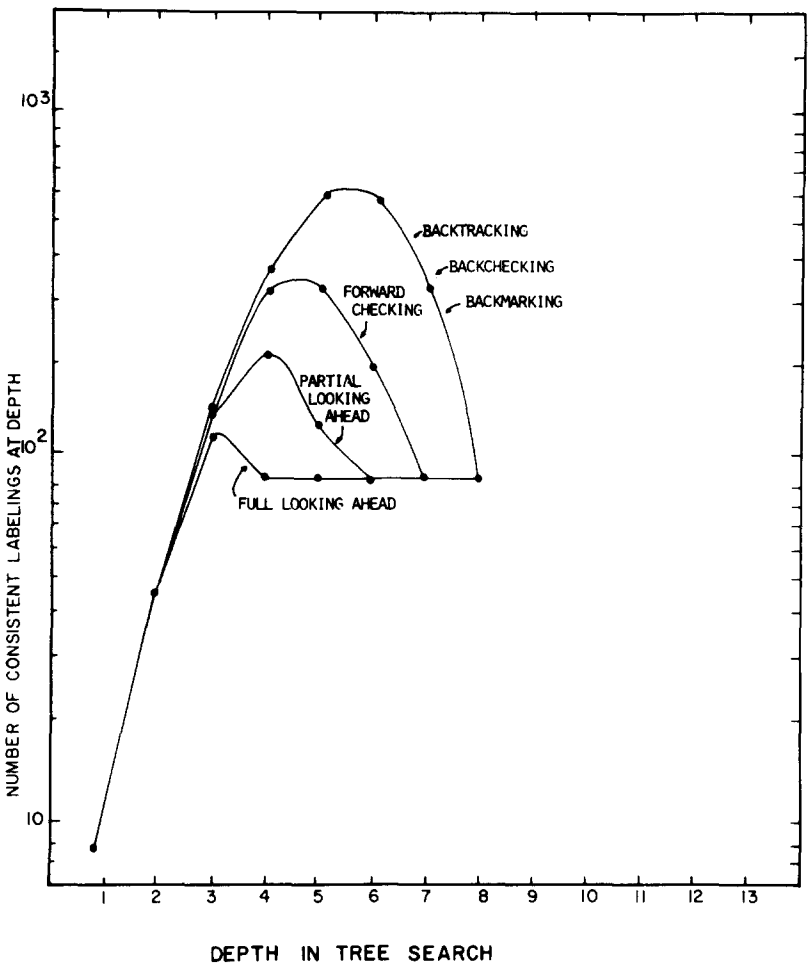

FIG. 17. The number of consistent labelings as a function of tree depth, for the 8-queens problem with the natural unit order.

algorithms structure, thus either may be used as a measure of algorithmic time complexity for the lookahead type of algorithms.

Because in both the N-queens and random problems backmarking seems to have a steadily increasing ratio of table lookups to consistency checks as problem size grows, only table lookups (which equals the number of nodes in the treesearch in this case) can be used as a true measure of algorithmic time complexity for backmarking. Only in the case that a computation of a relation

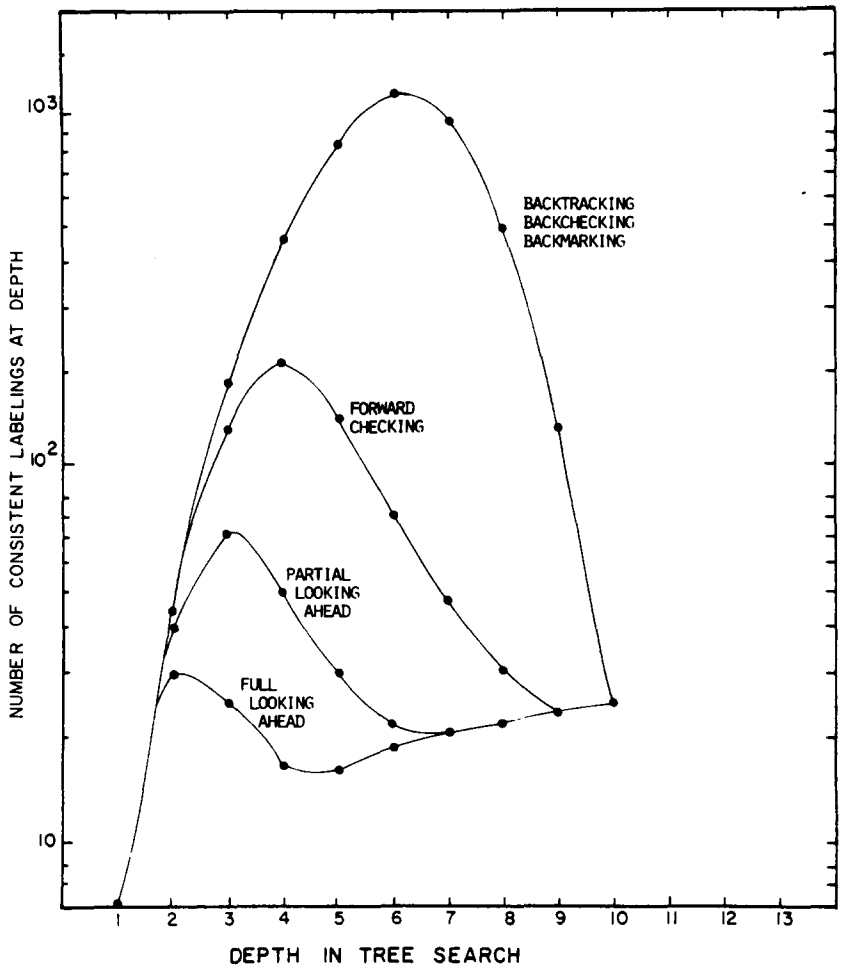

FIG. 18. The average number of nodes visited at each depth of the treesearch for solutions to 5 random relations with number of units = number of labels = 10, and probability of consistency check success of $p = 0.65$. These are the 5 relations shown at the $N = 10$ case in Figs. 8, 9, and 11.

check is significantly more expensive than the cost of a node's loop control and a table lookup will relation checks be a useful practical measure for the time complexity of backmarking. The reason that it is a practical measure in this case is that the node and table lookups cost will dominate the cost of execution only in very large problem sizes, so large that the problems can not be solved in a reasonable time, and relation tests will dominate the cost in the smaller problems which can be solved in a practical amount of time.

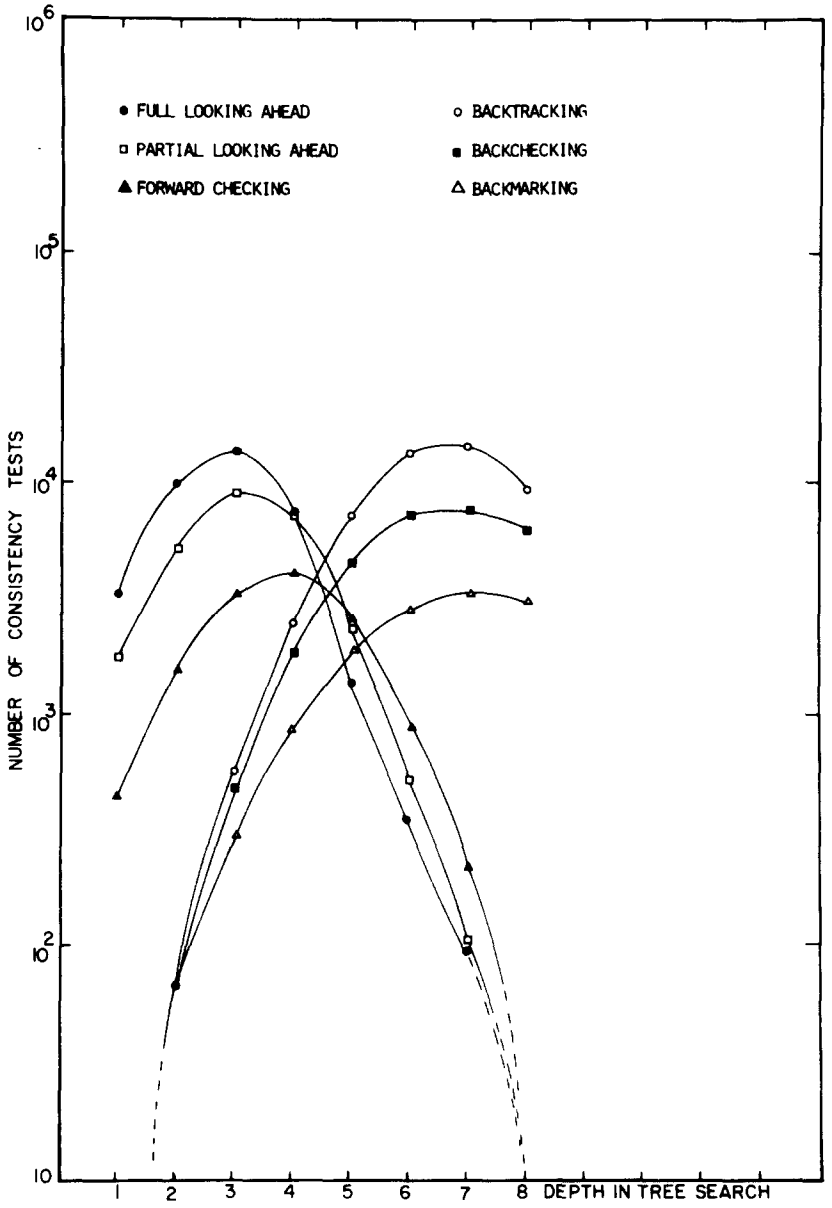

FIG. 19. This figure compares the number of consistency tests made at each level in the tree search for six different procedures, for the 8-queens problem, in the natural unit order.

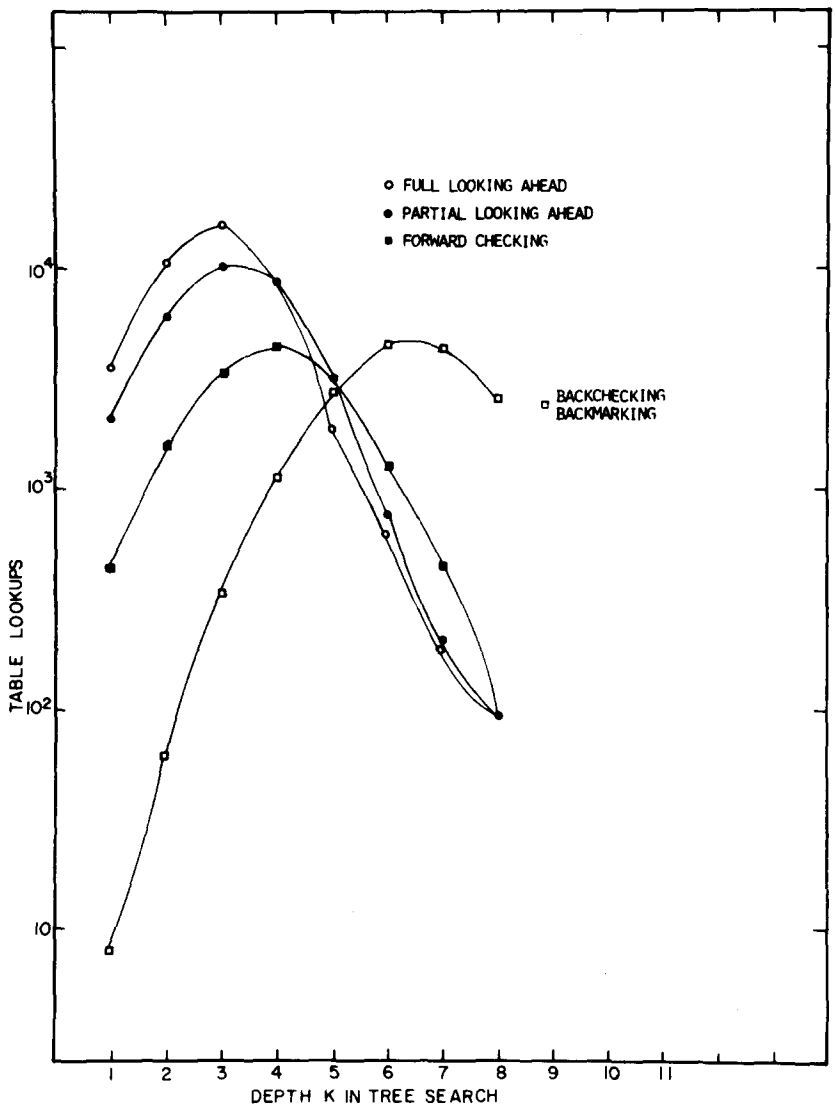

FIG. 20. The number of table lookups at each possible depth k in the treesearch to find all solutions to the 8-queens problem, for the various algorithms.

4. Statistical Model for Constraint Satisfaction Searches

Our statistical model for random constraint satisfaction is simple. The probability that a given consistency check succeeds is independent of the pair of units or labels involved and is independent of whatever labels may already

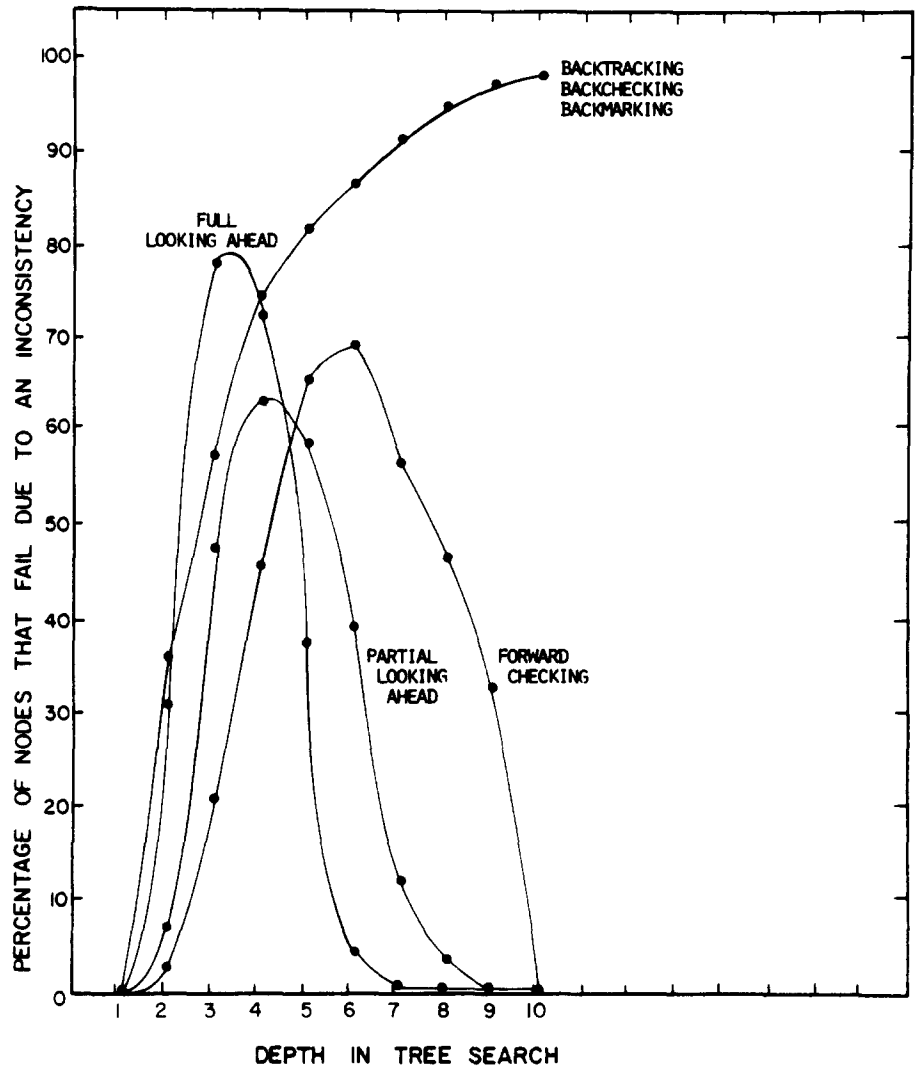

FIG. 21. This figure shows the percentage of nodes at a given depth in the tree search which fail because some inconsistency is detected at that node. Results are shown for the various algorithms in the 8-queens problem. Backchecking and backmarking often discover that there is an inconsistency by using table lookups, rather than performing all the checks. Nodes in the lookahead type algorithms fail because a future unit fails to have any remaining labels after inconsistent future labels are removed.

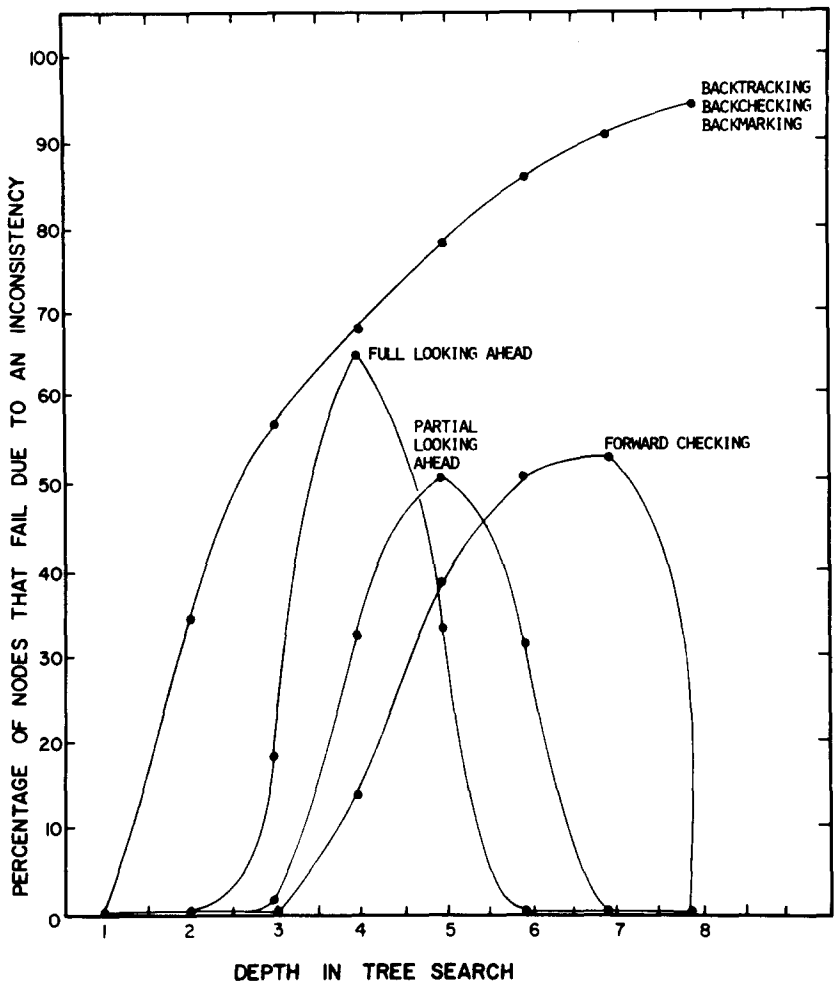

FIG. 22. This figure shows the percentage of nodes at a given depth in the tree search which fail because some inconsistency is detected at that node. Results are shown for the average over 5 random relations, with consistency check probability $p = 0.65$ and number of units = number of labels = 10.

have been assigned to past units. Hence,

$$P((u_{k+1}, l_{k+1}, u, l) \in R \mid l_1, \ldots, l_k \text{ are consistent labels of } u_1, \ldots, u_k) = P((u_{k+1}, l_{k+1}, u, l) \in R) \text{ for every } u, l.$$

The N-queens problem is a more difficult problem, with fewer solutions but requiring more consistency tests than the corresponding random constraint problem with the same probability of consistency check success. A comparison

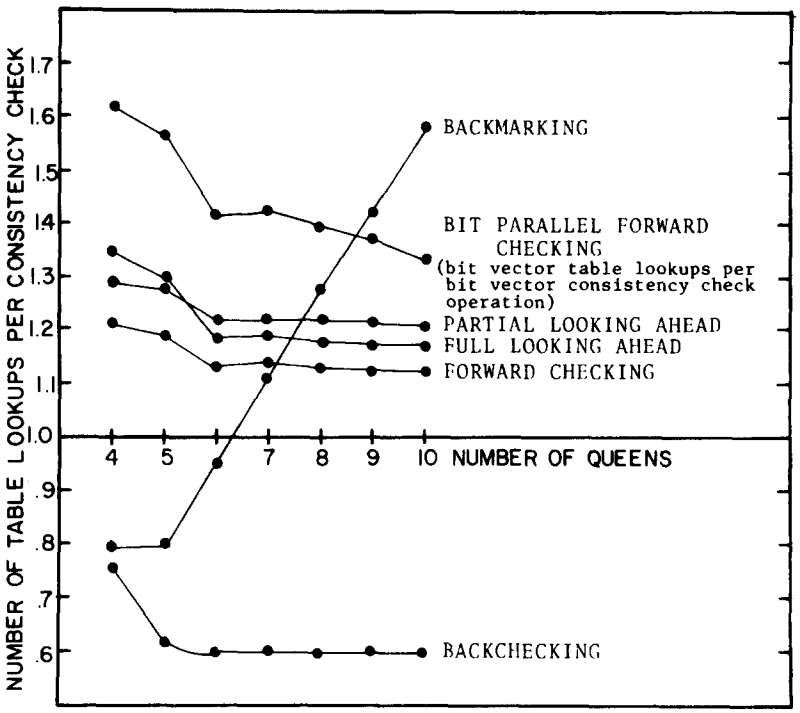

FIG. 23. This figure shows that in the N-queens problem, all the algorithms that keep tables, except backmarking, reference those tables no more than 35% more often than they reference the relation. However, backmarking appears to have a continually growing ratio of table lookups to relation tests, as problem size grows.

of the graphs for the two problems in Section 3 will show that while the numerical values of the quantities vary considerably, the basic character of the algorithms operation is similar for both problems.

In our analysis, we will assume that a given pair of units with a given pair of labels is consistent with probability p, p being independent of which units, which labels, or any past processing. If each unit has the same number, M, of possible labels, then any k-tuple of labels for any k units has probability $p^{k(k-1)/2}$ of being consistent since each labeling must satisfy $\frac{1}{2}k(k-1)$ consistency checks. Since there are M^k possible labelings of k units, the expected number of consistent labelings is

$$M^k p^{k(k-1)/2}.$$

The expected number of nodes processed at level k in a standard backtracking search will be M, the number of possible labels, times the number of

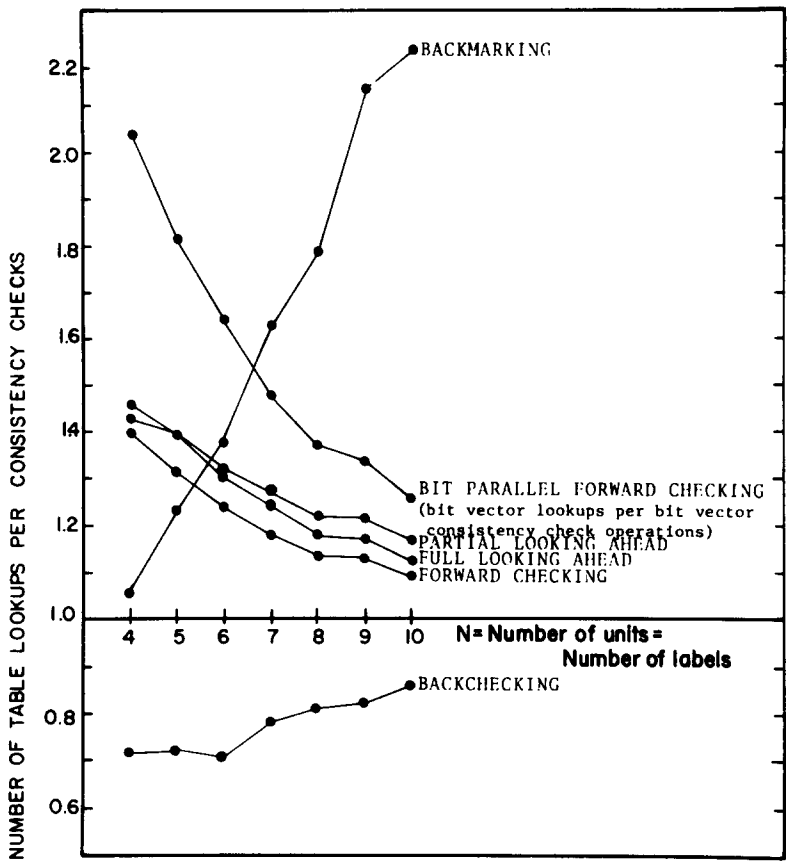

FIG. 24. This figure demonstrates that in the average over 5 random relations for each problem size that was tested, that the ratio of table lookups to consistency checks seems to approach one, except in the backmarking algorithm, in which the ratio seems to steadily increase with problem size. The random relations are the same as those shown in Figs. 8, 9, and 11.

consistent labelings at the previous level, $M^{k-1}p^{(k-1)(k-2)/2}$. Thus there are

$$M^k p^{(k-1)(k-2)/2}$$

tree search nodes at level k.

We can also count the expected number of consistency checks performed by backtracking. We expect $M^k p^{(k-1)(k-2)/2}$ level k nodes and at each node a label must be tested for consistency with the labels given the previous $k-1$ units. The first consistency check fails with probability $1-p$. If it fails, we have spent 1 test. If it succeeds we have spent 1 test and are committed to make another one which might also succeed with probability p. All $(k-1)$ tests will succeed

with probability $p^{(k-1)}$. Hence the expected number of consistency checks performed at each node is

$$\sum_{i=1}^{k-1} ip^{i-1}(1-p)+(k-1)p^{k-1}.$$

This may be simplified by recognizing the telescopic nature of the sum which is equal to

$$\sum_{i=0}^{k-2} p^i.$$

But this is a geometric sum and is equal to

$$\frac{1-p^{k-1}}{1-p}.$$

Therefore the expected number of consistency checks at level k will be $M^k p^{(k-1)(k-2)/2}$, the number of nodes at level k times $(1-p^{k-1})/(1-p)$, the expected number of consistency checks at a node, making

$$M^k p^{(k-1)(k-2)/2} \frac{1-p^{k-1}}{1-p}$$

consistency checks at level k. Of course the expected total number of consistency checks will be the summation of the expected number of consistency checks for each level k for k ranging from 1 to N, the number of units.

The computation of the number of labelings for the forward checking algorithm is somewhat more complicated because the algorithm stops checking when a future unit has no labels that are consistent with the past and present unit-label pairs. A consistent labeling to depth k occurs when the tree search successfully reaches a given label for unit k and forward checking of that unit-label pair produces no future unit that has no remaining labels. Thus the consistent labelings to depth k for forward checking meet exactly the following conditions:

(1) $\boldsymbol{u}_1 l_1, \boldsymbol{u}_2 l_2, \ldots, \boldsymbol{u}_k l_k$ are consistent unit-label pairs.

(2) There is no future unit $\boldsymbol{u}$ in levels $k+1, \ldots, N$ for which there is no label l so that $\boldsymbol{u}, l$ is consistent with $\boldsymbol{u}_1 l_1, \ldots, \boldsymbol{u}_k l_k$.

The k unit-label pairs are consistent with probability $p^{k(k-1)/2}$, and there are M^k possible labelings to depth k (condition (1)). A future unit-label pair is consistent with the k past and present unit-label pairs with probability p^k and there are M possible labels for a future unit, so the probability that a future unit has no label that is consistent with the k past and present units is $(1-p^k)^M$. Since there are N units, there are $N-k$ future units, and the probability that all of these has at least one label that is consistent is $[1-(1-p^k)^M]^{N-k}$ (condition (2)). Thus the expected number of consistent labelings to depth k for forward

checking is

$$M^k p^{k(k-1)/2}[1-(1-p^k)^M]^{N-k}.$$

The expression for the expected number of nodes in the forward checking tree search at level k is very similar to that for the number of consistent labelings to depth k, since each node will perform forward checks to determine if its label will become a consistent labeling. The labels for a node must meet condition (1) above, but the future units are required to have succeeded with at least one label only for checks with the unit-label pairs $u_1l_1, \ldots, u_{k-1}l_{k-1}$, since each node was in the table for a consistent labeling to depth $k-1$, and if any future unit as seen from level $k-1$ failed to have a label, then it would not have spawned nodes at the next level. Thus in each node the future units will have at least one label and the second condition occurs with probability $[1-(1-p^{k-1})^M]^{N-k}$. Thus the expected number of nodes at depth k in forward checking is

$$M^k p^{k(k-1)/2}[1-(1-p^{k-1})^M]^{N-k}.$$

A slight overapproximation for the expected number of consistency checks at depth k in the tree search can be found by multiplying the expected number of nodes at the depth times the expected number of labels remaining for each future unit times the number of future units, $N-k$. Since each future unit will have at least one label, this expected number of labels will be

$$\frac{Mp^{k-1}}{1-(1-p^{k-1})^M}.$$

Thus the expected number of consistency checks in forward checking will be

$$M^{k+1} p^{(k+2)(k-1)/2}[1-(1-p^{k-1})^M]^{N-k-1}(N-k).$$

The exact expected value can be obtained by replacing the number of future units term, $(N-k)$, with the expected number of future units tested, since forward checking will stop testing as soon as a future unit is discovered to have no possible labels. Each of these tests of a future unit will succeed with probability $1-(1-p^k)^M$, and reasoning similar to that for the number of consistency checks at each node in backtracking will give

$$\frac{1-[1-(1-p^k)^M]^{N-k}}{(1-p^k)^M}$$

for the expected number of future units tested. Thus the expected number of consistency checks at level k in forward checking will be

$$M^{k+1} p^{(k+2)(k-1)/2}[1-(1-p^{k-1})^M]^{N-k-1} \cdot \frac{1-[1-(1-p^k)^M]^{N-k}}{(1-p^k)^M}.$$

The number of table lookups in forward checking is the sum of the number

of consistency checks and the number of nodes. Thus the expected number of table lookups at depth k in the tree search will be

$$M^k p^{k(k-1)/2}[1-(1-p^{k-1})^M]^{N-k}\left[1+\frac{Mp^{k-1}}{1-(1-p^{k-1})^M}\cdot\frac{1-[1-(1-p^k)^M]^{N-k}}{(1-p^k)^M}\right].$$

The expected number of bit vector operations with the bit parallel data structure in forward checking can easily be found, by removing the term for the number of labels remaining for each future unit from the expression for the number of consistency tests, since only one operation will be performed for each unit, giving

$$M^k p^{k(k-1)/2}[1-(1-p^{k-1})^M]^{N-k}\frac{1-[1-(1-p^k)^M]^{N-k}}{(1-p^k)^M}$$

for the expected number of bit vector operations at level k in bit parallel forward checking.

The number of table lookups in bit vectors is still the sum of the number of bit vector operations and the number of nodes, for

$$M^k p^{k(k-1)/2}[1-(1-p^{k-1})^M]^{N-k}\left[1+\frac{1-[1-(1-p^k)^M]^{N-k}}{(1-p^k)^M}\right]$$

table lookups into bit vectors in bit parallel forward checking.

To illustrate the general form of the expressions we computed for the expected number of consistency checks and expected number of solutions, we present a few graphs. Fig. 25 illustrates the graph of the expected number of consistency checks as a function of tree depth for a random constraint satisfaction problem having $N = 17$ units and labels and a probability $p = 0.70$ of a constraint being satisfied. Notice that the greater number of tests forward checking does early in the tree search pays off later in the tree search both in terms of number of consistency tests and in number of successful instantiations at each tree depth (Fig. 26).

Fig. 27 illustrates the expected number of solutions as a function of N and p parameters of a random constraint satisfaction problem. Increasing N for a fixed p eventually causes fewer solutions to exist because the number of constraints is increasing quadratically.

We, of course, expect the number of consistency tests to increase as N increases and p remains fixed since the search space is becoming large very rapidly. This is shown for the forward checking procedure in Fig. 28. Also expected is for the average number of consistency checks per labeling to increase as N increases and p remains fixed (Fig. 29). As N increases, the problem of finding the first solution as well as all solutions is becoming more and more difficult. Therefore, it is not expected for the number of consistency

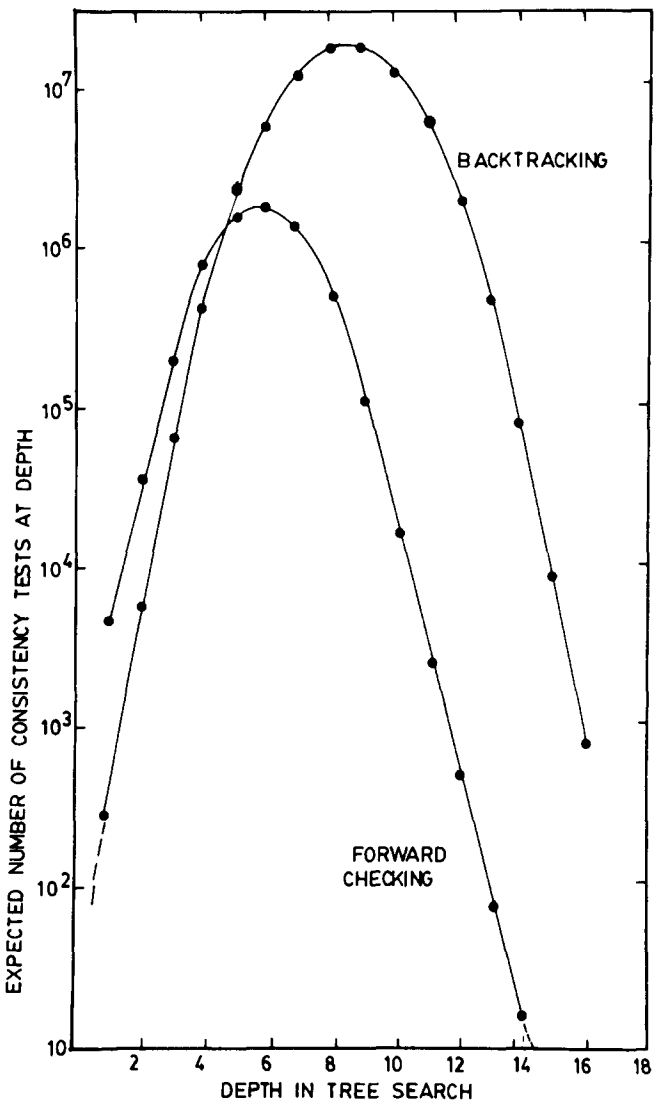

FIG. 25. The number of consistency tests as a function of tree depth for an $N = 17$, $p = 0.70$ random constraint satisfaction problem.

tests per solution to decrease as the number of solutions increases. The reason for this is that as the number of solutions increases more of the tests required to verify a solution become shared because the solutions have common segments. This is illustrated in Fig. 30.

An experimental check of the theoretical equations for the number of

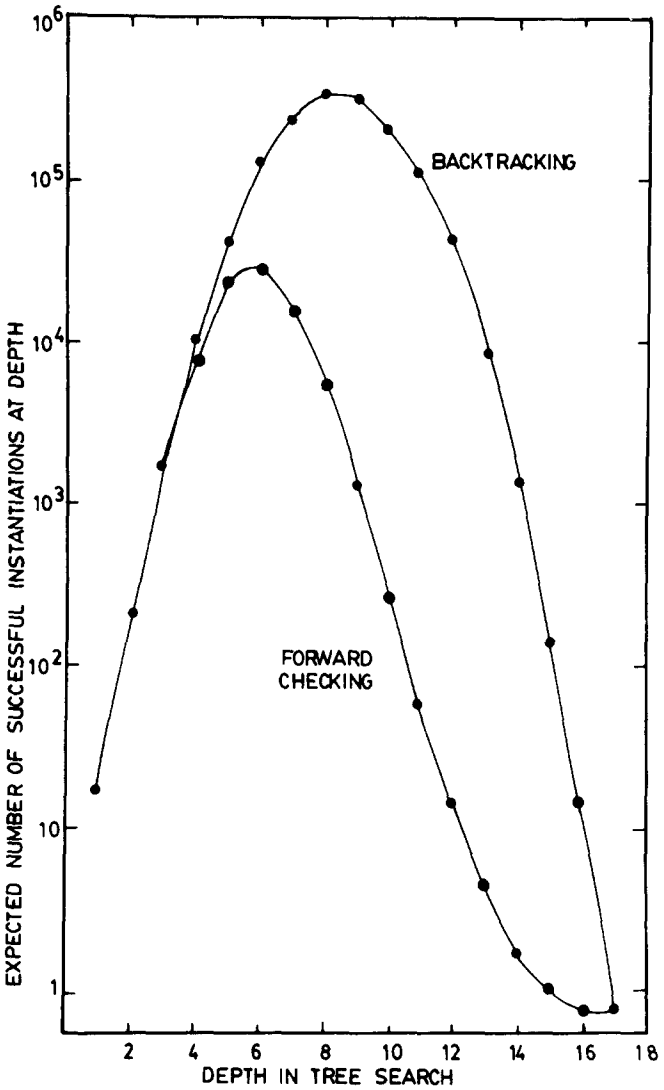

FIG. 26. The number of consistent labelings as a function of tree depth for an $N = 17$, $p = 0.70$ random constraint satisfaction problem.

solutions and number of solutions at a given depth in the tree search with random relations is given in Figs. 31 and 32. Although the average number of solutions is close to the theoretical result, the individual relations vary widely. The I bars mark a distance of one standard deviation of the mean above and below the average of the trials.

Fig. 33 demonstrates the accuracy of the theoretical expression for the

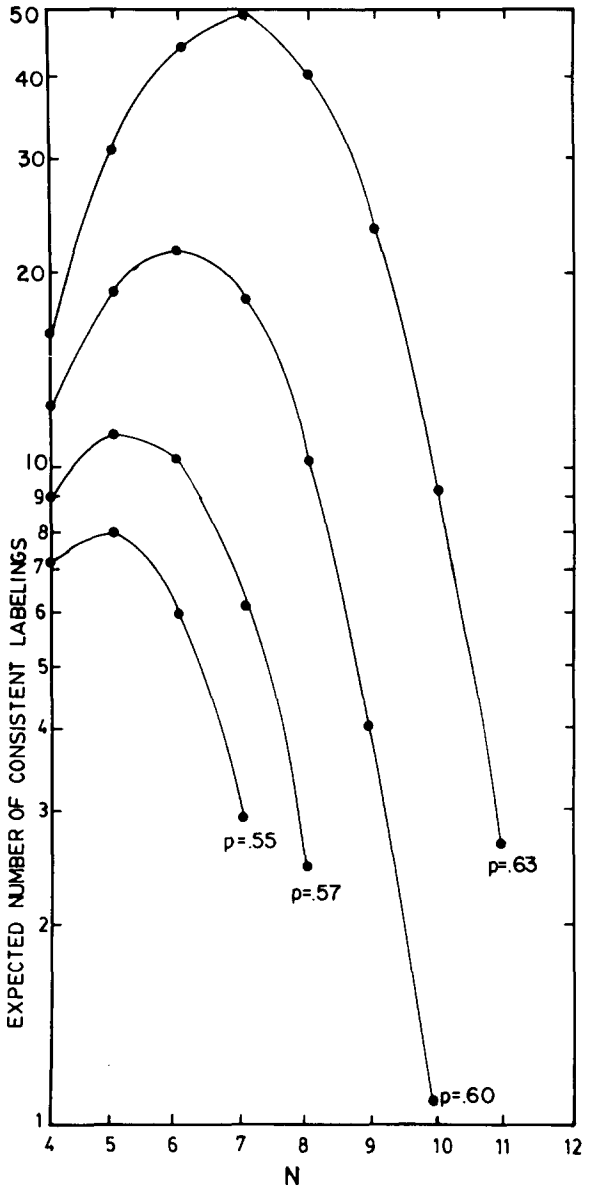

FIG. 27. The expected number of solutions as the N and p parameters change for a random constraint satisfaction problem.

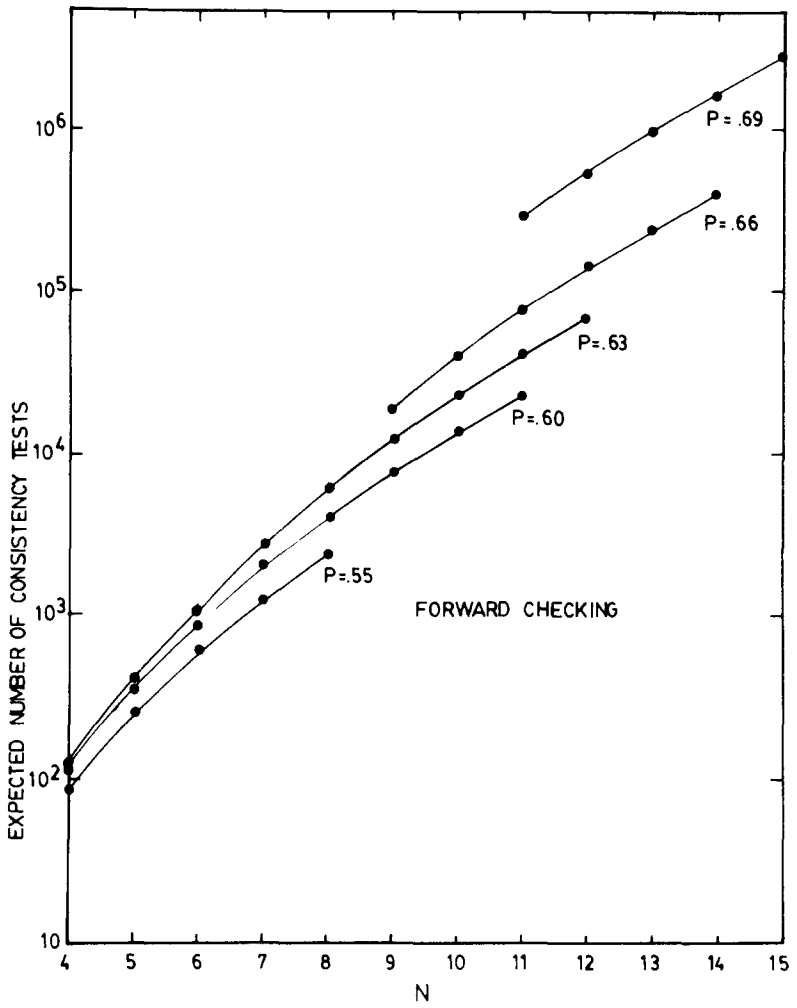

FIG. 28. This figure shows how the number of consistency tests increases as N increases, and p is held constant.

expected number of consistency tests with random relations in the forward checking algorithm. The total expected number of consistency tests shown is calculated from the sum of the expected number of tests for each level in the various problem sizes. For this expression to be correct, the expression for the expected number of nodes at each level in the forward checking algorithm must also be correct.

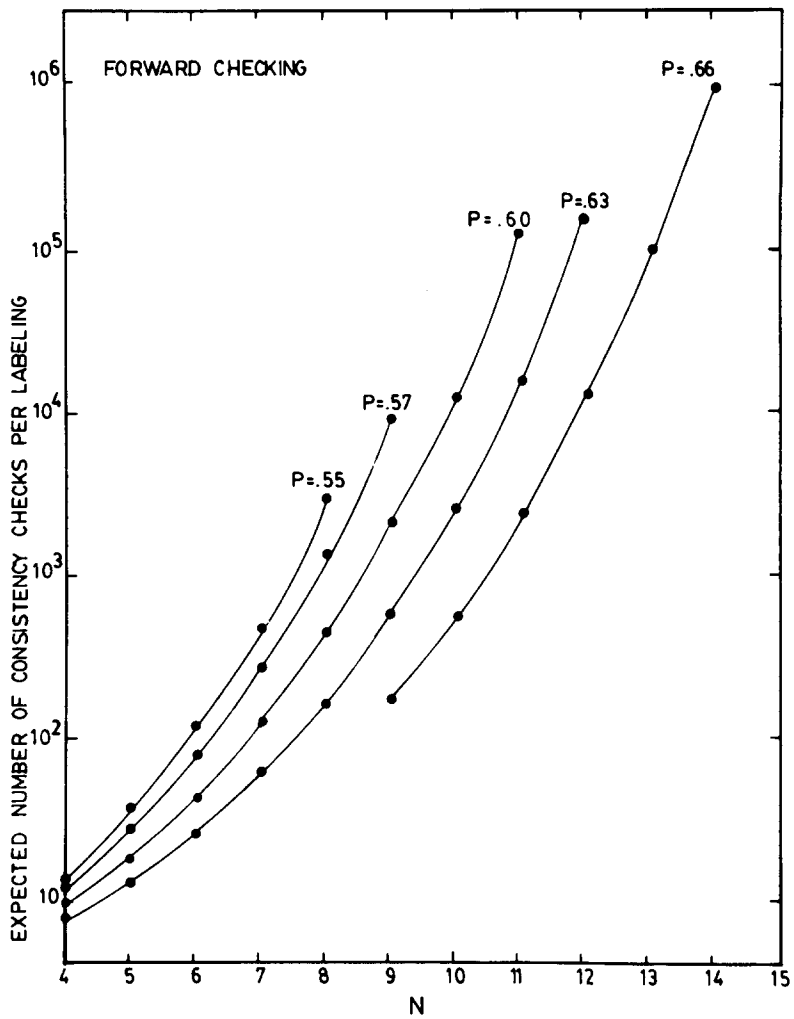

FIG. 29. This figure shows how the number of consistency tests per solution increases as N increases and p is held constant, in forward checking.

5. The Fail First Principle

One of the strategies which helps tree searching for constraint satisfaction problems is the fail first or prune early strategy of the looking ahead and forward checking procedures. There are other ways that we can apply the general principle of trying to fail first (and of course remember that fact so that there are no unnecessarily repeated mistakes). In this section we discuss two

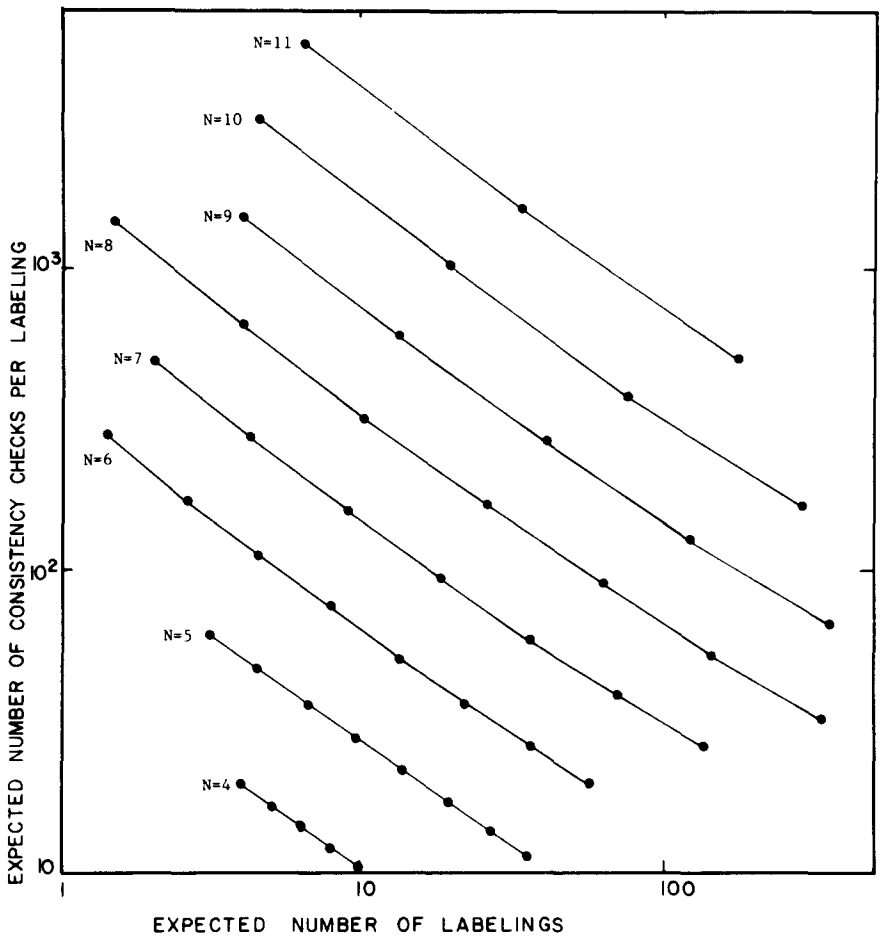

FIG. 30. This figure illustrates that as the expected number of solutions increases, the expected number of consistency checks per solution decreases, for varying probability p and fixed problem size N = number of units = number of labels, in the forward checking algorithm.

other applications of this strategy. The first is by optimizing the order in which we do consistency tests. The fail first principle states that we should first try those tests in the given set of tests that are most likely to fail since if they do fail we do not have to do the remainder of the tests in the set.

The second application is in dynamically choosing the optimal order in which to process units in each branch of the tree search. Optimal unit order choosing, even on a local basis, will not only lower the number of expected consistency tests per problem as compared with a random ordering, but it also lowers the variance of this average. For the unit order choice, the fail first principle states

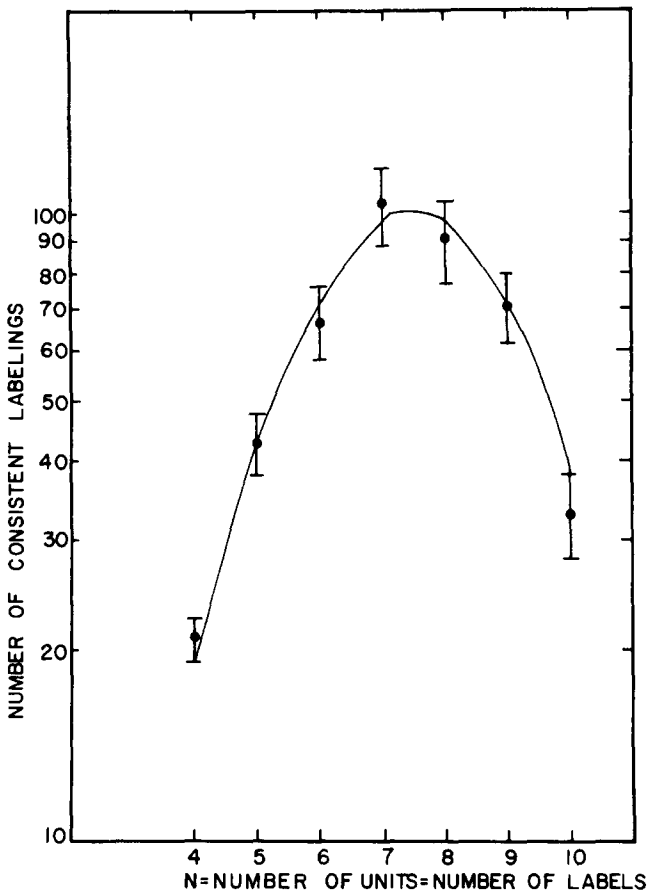

FIG. 31. The expected number of consistent labelings for random relations with number of units = number of labels = N, and consistency check success probability $p = 0.65$. Dots represent the average number of consistent labelings for 25 random relations tested at each problem size, and the I bars indicate one standard deviation of the mean above and below the experimental mean.

that the next unit to choose should be that one with the fewest possible labels left.

5.1. Optimizing the consistency check order in tree searching

Suppose we are solving a constraint satisfaction problem and suppose units $1, \ldots, K$ have already been assigned labels $l_1, \ldots, l_K$ and we are trying to find a label l_{K+1} for unit $K+1$. The label l_{K+1} must come from some set S_{K+1} of labels and it must be consistent with each of the previous labels $l_1, \ldots, l_K$, that is, we must have $(k, l_k, K+1, l_{K+1}) \in R$ for $k = 1, \ldots, K$. To determine the label l_{K+1},

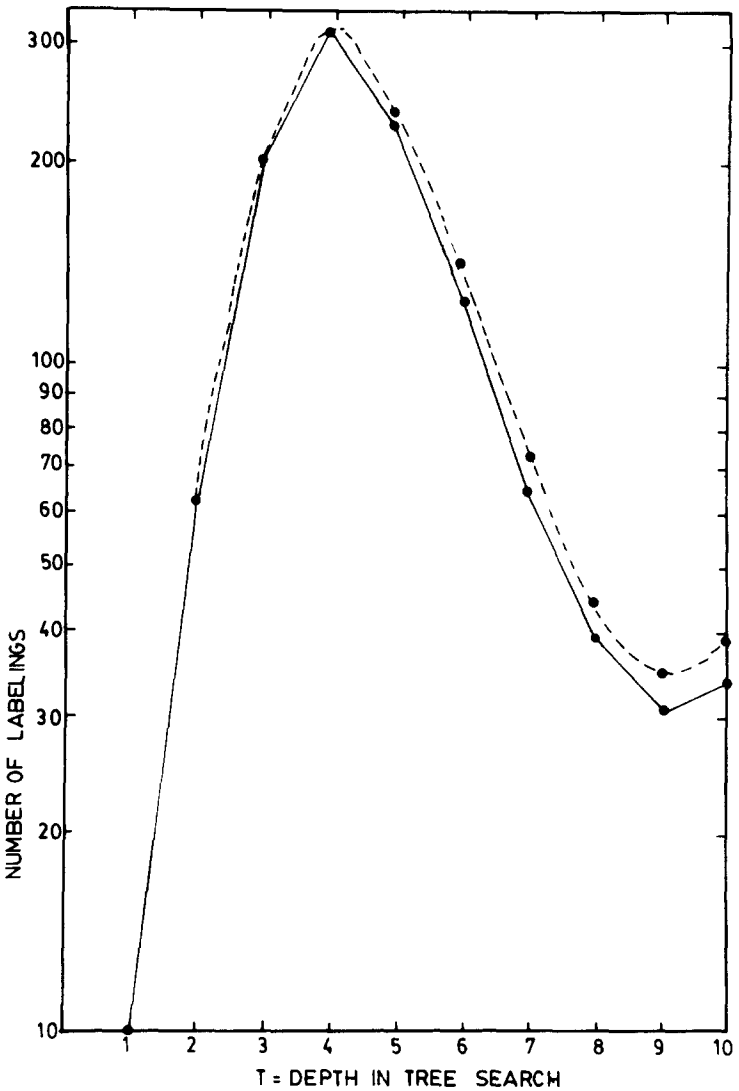

FIG. 32. The number of consistent labelings to depth k in the tree search for the average of 25 random constraint satisfaction problems with probability of consistency check success of 0.65 and number of units = number of labels = 10. The dotted curve is the theoretical expected number of labelings for such a problem.

we sequentially go through all the labels in S_{K+1} and perform the K consistency checks: $(k, l_k, K+1, l_{K+1}) \in R$. If one check fails, then we try the next label in S_{K+1}. If all checks succeed, then we can continue the depth first search with the next unit.

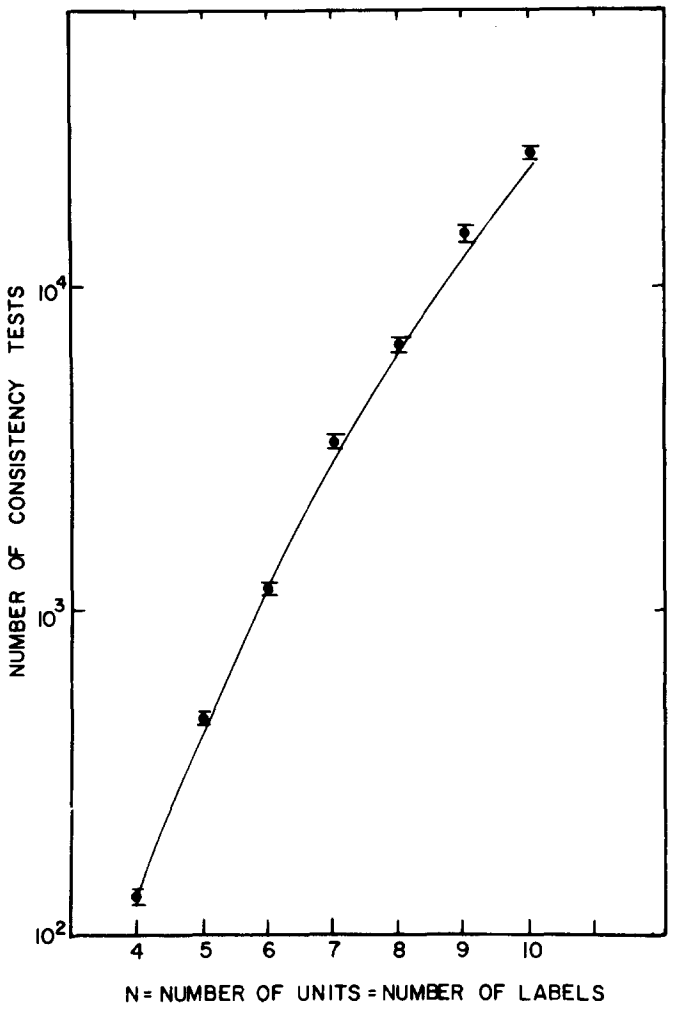

FIG. 33. The expected number of consistency checks for random relations with number of units = number of labels = N, and consistency check success probability $p = 0.65$. Dots represent the average number of consistency checks for 25 random relations tested at each problem size, and the I bars indicate one standard deviation of the mean above and below the experimental mean.

The optimizing problem for consistency checking is to determine an order in which to perform the tests which minimizes the expected number of tests performed. To set up the optimizing problem, we must have some knowledge about the degree to which a previous unit's label constrains unit $(K+1)$'s label. For this purpose we let $P(k)$ be the probability that the label l_k for unit k is consistent with some label for unit $K+1$. We assume that the consistency

checks are independent events so that the probability of the tests succeeding on units 1 through K is $\prod_{k=1}^{K} P(k)$.

For each order of testing, these probabilities determine the expected number of tests in the following way. Let $k_1, \ldots, k_K$ be a permutation of $1, \ldots, K$ designating the order in which the consistency checks will be performed. The test $(k_1, l_{k_1}, K+1, l_{K+1}) \in R$ will succeed with probability $P(k_1)$ and fail with probability $1 - P(k_1)$. If it fails, we incur a cost of one consistency check and we try the next label. If it succeeds, we will have incurred a cost of one consistency check and we are committed to try the next test $(k_2, l_{k_2}, K+1, l_{K+1}) \in R$. This test succeeds with probability $P(k_2)$ and fails with probability $1 - P(k_2)$. At this point, we have incurred a cost of two tests and may be committed to make more tests if this one succeeded.

Fig. 34 shows the tree of $K+1$ possible outcomes. Since the tests are assumed independent, the probability for each outcome can be computed by multiplying probabilities. For example, the probability of failing on consistency check with unit k_3 is $P(k_1)P(k_2)(1 - P(k_3))$. Also associated with each outcome is the number of tests performed to get there. For example, failure on the test with unit k_3 incurs a cost of 3 tests.

The expected number of tests C performed is computed by

$$C = \sum_{i=1}^{K} i[1 - P(k_i)] \prod_{j=1}^{i-1} P(k_j) + K \prod_{i=1}^{K} P(k_i).$$

Upon rearranging and simplifying this expression we obtain

$$C = 1 + \sum_{i=1}^{K-1} \prod_{j=1}^{i} P(K_j).$$

Now by the proposition at the end of Section 5, this is minimized by having $k_1, \ldots, k_K$ be any permutation of $1, \ldots, K$ satisfying $P(k_1) \leqslant P(k_2) \leqslant \cdots \leqslant P(k_K)$. Hence, to minimize expected numbers of tests, we must choose the order so that the tests with units most likely to fail are done first.

To illustrate the advantage of using optimum consistency test order, we consider the 10-queens problem when the units are naturally ordered from 1 to N and the current unit is K, then the fail first principle states that tests with past units must be done in the order of decreasing constraints. Since the row previous to row k has the strongest constraint on row k, the test order should be first unit $K-1$, then $K-2$, up to unit 1, in the N-queens problem. Backtracking requires 1,297,488 tests when done in the wrong order (unit $1, 2, \ldots, K-1$) and 1,091,856 tests when done in the right order. It is interesting to note that Gaschnig's backjumping procedure [3] when done with the consistency tests in the wrong order (1,131,942 tests) performs worse than standard backtracking with consistency tests in the right order. Furthermore, for the N-queens problem, backjumping with consistency tests in the right order for the N-queens problem is equivalent to standard backtracking with

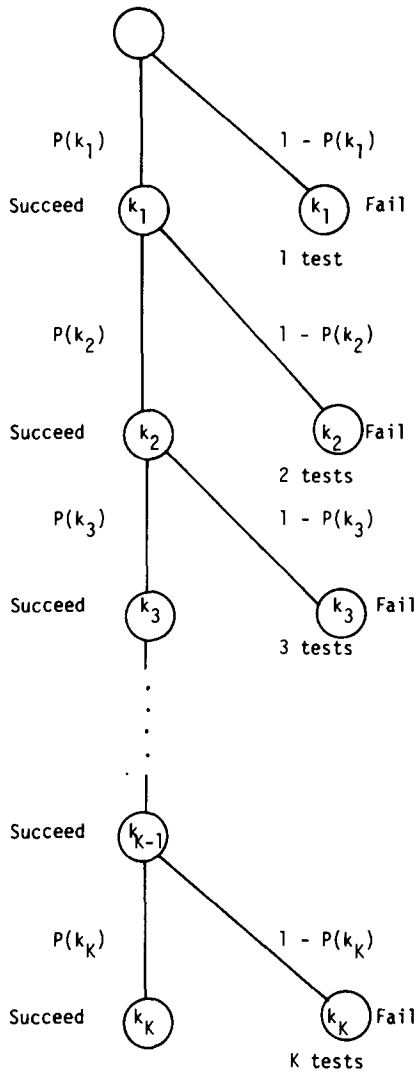

FIG. 34. The $K + 1$ outcomes of K tests.

consistency tests in the right order because backjumping backtracks to the highest level at which a failure is detected, and there is always at least one label at a given level which fails when checked with the immediately preceding level.

5.2. Optimizing tree search order

Every tree search must assume some order for the units to be searched in. The order may be uniform throughout the tree or may vary from branch to branch.

It is clear from experimental results that changing the search order can influence the average efficiency of the search. In this section we adopt the efficiency criterion of branch depth and we show how by always choosing the next unit having smallest number of label choices we can minimize the expected branch depth.

Suppose units $1, \ldots, N$ are units which are yet to be assigned labels. Let $n(m)$ be the number of possible or available labels for unit m. We assume that each of the $n(m)$ labels possible for unit m has the same probability q of succeeding and that success or failure of one of the labels is an independent event from success or failure for any of the other labels. Thus, the probability that a unit m will not have any label that succeeds is $(1-q)^{n(m)}$. The probability that some label for unit m succeeds is, therefore, $P(m) = 1-(1-q)^{n(m)}$. Unfortunately, this analysis holds only for the first level of the tree.

Let $k_1, \ldots, k_N$ be the order in which the units are searched on the tree. Let $P_n(k_n \mid k_1, \ldots, k_{n-1})$ be the conditional probability that some label for unit k_n will succeed when unit k_n is the nth one in the tree search order given that units $k_1, \ldots, k_{n-1}$ are the first $n-1$ units searched in the branch. We assume that the probability of a label for unit k_n succeeding depends only on the number of units preceding it in the tree search and not upon which particular units they are. That is,

$$P_n(k_n \mid k_1, \ldots, k_{m-1}) = P_n(k_n \mid l_1, \ldots, l_{n-1})$$

for all labels $l_1, \ldots, l_{n-1}$. This conditional independence assumption justifies the use of the notation $P_n(k_n)$ to designate the probability that some label succeeds for unit k_n when it is the nth unit in the tree search, and we will call the probability that an arbitrary label for unit u will succeed when checked against another arbitrary unit-label pair the success probability for unit u.

Units which are searched later in the tree typically have lower probability for a label succeeding since the label must be consistent with the labels given all the earlier units. We want some way to compare the probability of success for the same unit in different tree searches. Since the success probability depends only on the unit and its level in the tree and since units later in the tree have lower success probabilities, we assume that the success probability for a unit u when it is at level i in one tree search is related to the success probability of unit u when it is at the first level of another tree search by a constant factor α^{i-1} where $0 < \alpha \leq 1$:

$$P(u)\alpha^{i-1} = P_i(u).$$

The best search order is the one which minimizes the expected length or depth of any branch. When the units are searched in the order $k_1, \ldots, k_m$, the expected branch depth is given by

$$1 + \sum_{n=1}^{N-1} \prod_{j=1}^{n} P_j(k_j).$$

By the proposition at the end of Section 5, this is minimized when the unit chosen at each level is that unit whose success probability is smallest. Thus at level j we choose unit k_j, where

$$P_j(k_j) \leq P_j(u) \quad \text{for } u \neq k_1, \ldots, k_{j-1}.$$

Now, $P_j(k_j) = \alpha^{j-1}[1-(1-q^{n(k_j)})]$. Since $0 \leq q \leq 1$, this expression is minimized by choosing k_j to be that unit having the smallest number of possible labels.

To illustrate the advantage of using a locally optimal unit order for each branch in the tree search, we consider the improvement achieved on the N-queens problem and random relation problems. The number of consistency tests required is given in Tables 1 and 2. Some improvement is shown in the larger N-queens problems, and considerable improvement appears in the larger random relation problems. Fig. 35 demonstrates that the improvement increases with problem size in the random relation problems with $p = 0.65$ and number of units = number of labels = N.

The reason why optimal unit order usually improves forward checking more than backmarking is that forward checking has more information about future units than backmarking. Therefore, forward checking's choice of the next unit most likely to fail is more likely to produce a unit which fails than backmarking's choice.

PROPOSITION. *Let $0 < \alpha \leq 1$ be given. For each unit u let $P(u)$ be its initial success probability. Let $k_1, \ldots, k_N$ be any permutation of $1, \ldots, N$ satisfying $P(k_1) < P(k_2) < \cdots < P(k_N)$. Define $P_n(u) = \alpha^{n-1}P(u)$. Then,*

$$\sum_{n=1}^{N-1} \prod_{j=1}^{n} P_j(k_j) \leq \sum_{n=1}^{N-1} \prod_{j=1}^{n} P_j(u_j) \qquad \text{for any permutation } u_1, \ldots, u_N \text{ of } 1, \ldots, N.$$

Proof. Let $u_1, \ldots, u_N$ be any permutation of $1, \ldots, N$ minimizing

$$\sum_{n=1}^{N-1} \prod_{j=1}^{n} P_j(u_j).$$

If $u_1, \ldots, u_N$ equals $k_1, \ldots, k_N$ we are done. If $u_1, \ldots, u_N$ does not equal $k_1, \ldots, k_N$, let m be the smallest index such that $u_m \neq k_m$. Also let m' be the index such that $u_m \neq k_m$. Also let m' be the index such that $u_{m'} = k_m$. Define the permutation $i_1, \ldots, i_N$ by

$$i_n = u_n, \qquad n \neq m \text{ or } m',$$

$$i_m = u_{m'},$$

$$i_{m'} = u_m.$$

We will prove a contradiction by showing that

$$\sum_{n=1}^{N-1} \prod_{j=1}^{n} P_j(i_j) < \sum_{n=1}^{N-1} \prod_{j=1}^{n} P_j(u_j)$$

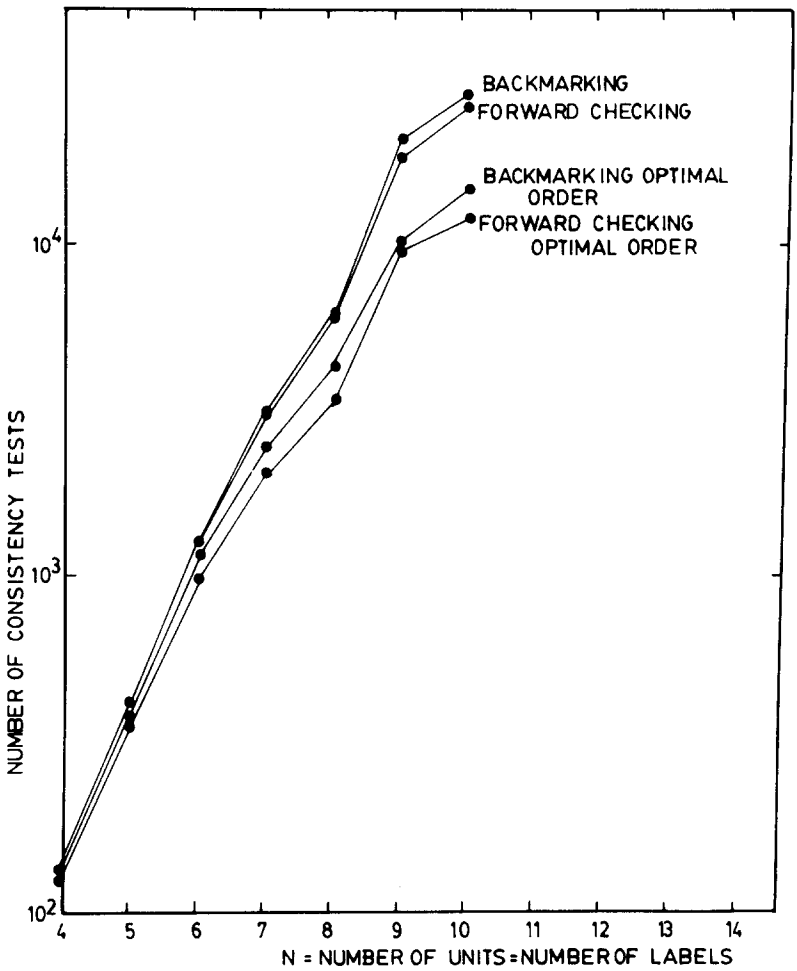

FIG. 35. The improvement in efficiency for the average of 5 random relation problems with probability $p = 0.65$ and number of units = number of labels = N, when locally optimized unit order is used. The relations are the same as those shown in Figs. 9 and 10.

by looking at the products. There are three cases: $n < m$, $m \leq n \leq m'$, and $m' \leq n$.

Case 1: $n < m$. Here since $i_j = u_j$, $j = 1, \ldots, n$, we obtain

$$\prod_{j=1}^{n} P_j(i_j) = \prod_{j=1}^{n} P_j(u_j).$$

Case 2: $m \leq n < m'$.

$$\prod_{j=1}^{n} P_j(i_j) = \frac{P_m(u_m)}{P_m(u_m)} \prod_{j=1}^{n} P_j(i_j) = \frac{P_m(u_{m'})}{P_m(u_m)} \prod_{j=1}^{n} P_j(u_j).$$

TABLE 1. Number of consistency tests in N-queens, problem for normal and optimal unit order

	Backtracking	Backchecking	Full Looking Ahead		Part Looking Ahead		Backmarking		Forward Checking	
N	Normal	Normal	Normal	Optimal	Normal	Optimal	Normal	Optimal	Normal	Optimal
4	84	80	99	99	97	97	76	76	76	76
5	405	356	598	578	485	431	276	276	282	282
6	2,016	1,496	2,095	2,082	1,703	1,708	944	921	964	946
7	9,297	6,042	8,942	8,941	6,511	6,318	3,236	3,168	3,338	3,229
8	46,752	27,450	35,323	35,211	25,882	25,062	12,308	12,095	13,024	12,108
9	243,009	131,538	153,455	151,275	112,327	106,247	50,866	50,027	55,326	49,856
10	1,297,558	643,658	661,017	636,377	496,455	449,666	220,052	211,635	242,174	205,970

TABLE 2. Number of consistency tests in average of 5 random constraint satisfaction problems with consistency check success probability 0.65, for normal and optimal unit order. N = number of units = number of labels

	Backtracking	Backchecking	Full Looking Ahead		Part Looking Ahead		Backmarking		Forward Checking	
N	Normal	Normal	Normal	Optimal	Normal	Optimal	Normal	Optimal	Normal	Optimal
4	243	195	230	223	184	174	132	133	133	121
5	1,043	731	760	712	599	554	425	382	414	355
6	4,637	2,514	2,543	2,288	1,966	1,700	1,288	1,159	1,273	979
7	12,040	6,722	5,722	5,156	4,697	3,875	3,175	2,486	3,057	2,069
8	25,893	13,490	10,779	9,306	9,111	7,233	6,089	4,300	5,978	3,425
9	118,086	55,318	30,799	25,904	26,788	19,174	21,170	10,246	18,616	8,673
10	163,983	73,260	44,655	36,675	41,232	28,872	28,314	14,892	26,288	12,022

Now, $P_m(u_{m'}) = P_m(k_m) < P_m(u)$ for any $u \neq k_1, \ldots, k_{m-1}$. Since $u_m \neq k_1, \ldots, k_{m-1}$, $P_m(u_{m'}) < P_m(u_m)$ and

$$\prod_{j=1}^{n} P_j(i_j) = \frac{P_m(u_{m'})}{P_m(u_m)} \prod_{j=1}^{n} P_j(u_j) < \prod_{j=1}^{n} P_j(u_j).$$

Case 3: $m' \leq n$.

$$\begin{aligned}
\prod_{j=1}^{n} P_j(i_j) &= \frac{P_m(u_m)P_{m'}(u_{m'})}{P_m(u_m)P_{m'}(u_{m'})} \prod_{j=1}^{n} P_j(i_j) \\
&= \frac{P_m(i_m)P_{m'}(i_{m'})}{P_m(u_m)P_{m'}(u_{m'})} \prod_{j=1}^{n} P_j(u_j) \\
&= \frac{P_m(u_{m'})P_{m'}(u_m)}{P_m(u_m)P_{m'}(u_{m'})} \prod_{j=1}^{n} P_j(u_j) \\
&= \frac{\alpha^{m-1}P(u_{m'})\alpha^{m'-1}P(u_m)}{\alpha^{m-1}P(u_m)\alpha^{m'-1}P(u_m)} \prod_{j=1}^{n} P_j(u_j) \\
&= \prod_{j=1}^{n} P_j(u_j).
\end{aligned}$$

Hence,

$$\sum_{n=1}^{N-1} \prod_{j=1}^{n} P_j(i_j) < \sum_{n=1}^{N-1} \prod_{j=1}^{n} P_j(u_j),$$

contradicting the minimality of $u_1, \ldots, u_N$. Therefore, $u_1, \ldots, u_N = k_1, \ldots, k_N$.

6. Conclusion

Using complexity criteria of number of consistency checks and number of table lookups we have shown analytically and experimentally the efficacy of the remembering and fail first principles in constraint satisfaction tree search problems. A new search procedure called forward checking has been described and it combined with optimal unit order choice leads to a more efficient tree search than looking ahead or backmarking. A data structure that takes advantage of a computer's natural ability to process bit vectors in parallel can make forward checking even more efficient. This suggests that the entire set of lookahead operators described by Haralick et al. [3], Haralick and Shapiro [6, 7], the discrete relaxation described by Waltz [16] and Rosenfeld et al. [14] would be more efficiently implemented by omitting the consistency tests required by future units against future units. Further analytic and experimental work needs to be done to determine if this in fact is generally true, whether it is only true when every unit constrains every unit or in the problem discussed in this paper. Applicability of the forward checking idea to inference and theorem proving algorithms needs to be tested and this will be the topic of a future paper.

REFERENCES

1. Barrow, H.G. and Tenenbaum, J.M., MYSIS: A system for reasoning about scenes, Stanford Research Institute, Menlo Park, CA, SRI AI Techn. Rep. 121 (1976).
2. Gaschnig, J., A general backtrack algorithm that eliminates most redundant tests, *Proceedings of the International Conference on Artificial Intelligence*, Cambridge, MA (1977), 457.
3. Gaschnig, J., Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing-assignment problems, *Proceedings of the 2nd National Conference of the Canadian Society for Computational Studies of Intelligence*, Toronto, Ont., July 19–21 (1978).
4. Gaschnig, J., Performance measurement and analysis of certain search algorithms, Thesis, Department of Computer Science, Carnegie-Mellon University (May 1979).
5. Haralick, R.M., Scene analysis: Homomorphisms, and arrangements, in: Hanson and Reisman (Eds.), *Machine Vision* (Academic Press, New York, 1978).
6. Haralick, R. and Shapiro, L., The consistent labeling problem: Part I, *IEEE Trans. Pattern Analysis and Machine Intelligence* **1** (2) (April 1979).
7. Haralick, R. and Shapiro, L., The consistent labeling problem: Part II, *IEEE Trans. Pattern Analysis and Machine Intelligence* **2** (3) (1980).
8. Haralick, R.M., Davis, L.S., Rosenfeld, A. and Milgram, D.L., Reduction operations for constraint satisfaction, *Information Sci.* **14** (1978) 199–219.
9. Harary, F., *Graph Theory* (Addison-Wesley, Reading, MA, 1969).
10. Kowalski, R., A proof procedure using connection graphs, *J. Agg. Comput. Mach.* **22** (1975) 572–595.
11. Mackworth, A.K., Consistency in networks of relations, *Artificial Intelligence* **8** (1) (1977) 99–118.
12. McGregor, J.J., Relational Consistency algorithms and their applications in finding subgraph and graph isomorphisms, *Information Sci.* **18** (1979).
13. Montanari, V., Networks of constraints: Fundamental properties and applications to picture processing, *Information Sci.* **7** (1976) 95–132.
14. Rosenfeld, A., Hummel, R. and Zucker, S., Scene labeling by relaxation operations, *IEEE Trans. Systems, Man and Cybernetics* **SMC-6** (1976) 420–433.
15. Ullman, J.R., Associating parts of patterns, *Information and Control* **9** (6) (1966) 583–601.
16. Waltz. D.L., Generating semantic descriptions from drawings of scenes with shadows, Rep. MAC AI-TR-271 MIT, Cambridge, MA (1972).
17. Knuth, D., Estimating the efficiency of backtrack programs, *Mathematics of Computation* **29** (129) (1975), 121–136.

Received 21 December 1979