

INCREASING TREE SEARCH EFFICIENCY
FOR
CONSTRAINT SATISFACTION PROBLEMS

Robert M. Haralick
Electrical Engineering Dept.
Computer Science Dept.
Virginia Polytechnic Institute
and State University
Blacksburg, Virginia 24061

Gordon L. Elliott
Electrical Engineering Dept.
Virginia Polytechnic Institute
and State University
Blacksburg, Virginia 24061

In this paper we explore the number of consistency checks made by a tree search in order 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 remember 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 remember your 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 consistency checks performed by a search procedure 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 assume that there are M units (some authors call these variables instead of units). Each unit has a set of L possible values or labels for each pair of units. 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 U is the set of units and L is the set of labels, then the binary constraint R can be represented as a binary relation on $U \times L$: $R \subseteq (U \times L) \times (U \times L)$. If a pair of unit labels $(u_1, \ell_1, u_2, \ell_2) \in R$, then labels ℓ_1 and ℓ_2 are said to be consistent or compatible for units u_1 and u_2 . A labeling f of all the units satisfies the constraints if for every pair u_1, u_2 of units $(u_1 f(u_1), u_2, f(u_2)) \in R$. Haralick et. al. (1978) 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 (Barrow and Tenenbaum, 1976, and Rosenfeld et. al., 1976), line interpretation (Waltz, 1972), edge labeling (Haralick, 1978), graph homomorphisms and isomorphisms (Ullman, 1969), graph coloring (Harary, 1969) boolean satisfiability (Haralick et. al., 1978), and proposition theorem proving (Kowalski, 1975) are all special cases of the general consistent labeling problem. Ullman (1966), Waltz (1972), Rosenfeld et. al. (1978 and 1979), and Gaschnig (1977 and 1978) attempt

to find efficient methods to solve the consistent labeling problem. Knuth (1975) 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. 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.

Using the number letter convention for unit-label pairs, Figure 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 (1977) and the forward checking approach described in this paper help eliminate these problems. Notice that once unit 3 takes label E (Figure 1a) 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 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 (Figure 1b). The lookahead procedures (discrete relaxation) of Ullman (1966), Waltz (1972), Rosenfeld (1976), Mackworth (1977), and Montanari (1974) help alleviate the problem illustrated in Figure 1a as well as in Figure 1b.

Section 2 gives a brief description of the full and partial looking ahead, forward checking, backchecking, and backmarking procedures and concludes with a comparison of these problems created randomly. These results show that standard backtracking is least efficient in most cases, and backmarking and forward checking are the most efficient for the cases tried.

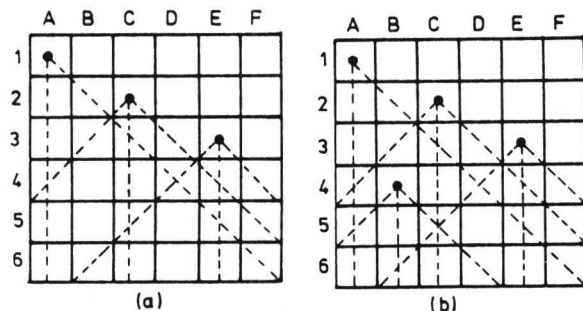


Figure 1a illustrates how the labeling A,C,E for units 1,2,3 implies that the only labels for units 4 and 6 are incompatible in the 6 Queens problem.

Figure 1b 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

```

Figure 2 illustrates 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.

In section 3, 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 4 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 not only does performance improve for each procedure, but that forward checking now does better than backmarking in the larger problems tested.

2. SOME PROCEDURES FOR TREE SEARCH REDUCING

In this section we give brief descriptions of five procedures which can be used within the standard backtracking framework to reduce tree search operations. They are called full and partial looking ahead, forward checking, back-checking, and backmarking. Each of these procedures invests resources in additional consistency tests 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.

The section concludes with a comparison among the tree search reducing procedures which indicates that backtracking is least efficient in most cases, and that backmarking and forward checking are the most efficient for the cases tested. The lookahead procedures can be ordered in increasing order of efficiency as full looking ahead, partial looking ahead, and forward checking.

2.1 Looking Ahead

Waltz filtering (Waltz, 1972), a procedure by Ullman (1966), discrete relaxation (Rosenfeld, Hummel, Zuckor, 1976), and the Ψ operator of Haralick et. al. (1978) 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 than backtracking and one that checks only future with past and 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 (Figure 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 $(NUMBER_OF_UNITS)^2 \times (NUMBER_OF_LABELS)$ words of memory for table storage). 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 have obvious meanings.

The function RELATION($U1, L1, U2, L2$) 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 labels. Thus the next level of the tree search will not be called, otherwise each entry in NEW_T is consistent with $U, F(u)$, and therefore, all the past unit-label pairs.

```

RECURSIVE PROCEDURE L_A_TREE_SEARCH(U,F,T);
FOR F(U) = each element of T(U) BEGIN
  IF U < NUMBER_OF_UNITS THEN BEGIN
    NEW_T = CHECK_FORWARD(U,F(U),T);
    CALL LOOK_FUTURE(U,NEW_T);
    IF NEW_T is not EMPTY_ROW_FLAG
      CALL L_A_TREE_SEARCH(U+1,F,NEW_T)
    END;
  ELSE
    Output the labeling F;
  END;
END L_A_TREE_SEARCH;

```

Figure (3a)

```

PROCEDURE CHECK_FORWARD(U,L,T);
NEW_T = empty table;
FOR U2 = U+1 TO NUMBER_OF_UNITS BEGIN
  FOR L2 = each element of T(U2)
    IF RELATION(U,L,U2,L2) THEN
      ENTER L2 in the list NEW_T(U2);
  IF NEW_T(U2) is empty THEN
    RETURN (EMPTY_ROW_FLAG); /* NO consistent
    Labels */
  END;
RETURN (NEW_T);
END CHECK_FORWARD;

```

Figure (3b)

```

PROCEDURE LOOK_FUTURE(U,NEW_T);
IF U+1 ≥ NUMBER_OF_UNITS THEN RETURN;
FOR U1 = U+1 TO NUMBER_OF_UNITS BEGIN
  L1 CONSISTENT = FALSE;
  FOR L1 = each element of NEW_T(U1)
    FOR U2 = U+1 TO NUMBER_OF_UNITS except
      skipping U1 BEGIN
      L2 CONSISTENT = FALSE;
      FOR L2 = each element of NEW_T(U2);
        IF RELATION(U1,L1,U2,L2) THEN BEGIN
          L2 CONSISTENT = TRUE;
          BREAK for L2 loop;
        END;
      IF NOT L2 CONSISTENT THEN
        BREAK for U2 loop;
      END for U2 loop;
    IF L2 CONSISTENT THEN
      L1 CONSISTENT = TRUE;
    ELSE
      Delete L1 from list NEW_T(U1);
    END for L1 loop;
  IF NOT L1 CONSISTENT THEN BEGIN
    NEW_T = EMPTY_ROW_FLAG;
    RETURN;
  END;
END for U1 loop;
RETURN;
END LOOK_FUTURE;

```

Figure (3c)

2.2 Partial Looking Ahead

Partial looking ahead is a variation of looking ahead which does approximately half of the consistency 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 and standard backtracking 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 requires only a minor modification of the LOOK_FUTURE procedure (Figure 3b). To change LOOK_FUTURE into PARTIAL_LOOK_FUTURE, the second and fifth lines must be changed. The "FOR U1..." loop becomes "FOR U1 = U+1 TO NUMBER_OF_UNITS -1 BEGIN" and "FOR U2..." becomes "FOR U2 = U1+1 TO NUMBER_OF_UNITS".

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 base 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 label 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 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 label table.

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

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 by itself is not as good as forward checking.

2.5 Backmarking

Backmarking (defined in Gaschnig, 1977, and also discussed in Gaschnig, 1978) is backchecking with an added feature. Backchecking 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.

2.6 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, finding all solutions. 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.

Our comparison will be in terms of three kinds of graphs and tables:

1. The number of nodes visited is largest for the middle levels of the tree search, with the looking ahead procedure having fewest nodes at each level. (Figures 4 and 6)
2. The bulk of the consistency tests are done at shallow depths of the tree-search for the look ahead types of procedures (Figure 5)
3. Forward checking and backmarking do the fewest consistency tests among the algorithms using the normal unit orders in most cases tested. (Tables 1 and 2)

3. 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 have been assigned to past units. Hence, $P((u_{K+1}, \ell_{K+1}, u, \ell) \in R | \ell_1, \dots, \ell_K \text{ are consistent labels of } u_1, \dots, u_K) = P((u_{K+1}, \ell_{K+1}, u, \ell) \in R \text{ for every } u, \ell)$.

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; and there will be M units total. If each unit has the same number L 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 $K(K-1)/2$ consistency checks. Since there are L^K possible labelings of K units, the expected number of consistent labelings is

$$L^K p^{K(K-1)/2}$$

Using our statistical model, the expected number of consistency checks at level K in the tree search can be derived:

$$L^K p^{K(K-1)/2} \frac{1-p^{K-1}}{1-p}$$

We might note that the N-Queens problem differs some from the assumption of our statistical model. In terms of number of solutions, N-Queens has fewer solutions than the expected number of solutions of its associated random constraint satisfaction problem. In terms of consistency checks, N-Queens requires more checks than the expected number of checks of its associated random constraint satisfaction problem.

The computation of the number of labelings at any depth K for the forward checking algorithm is considerably more complicated, but the result is nearly as simple as for backtracking. The expected number of consistent labelings at level K out of M units will be

$$L^K p^{K(K-1)/2} [1 - (1-p)^L]^{M-K}$$

Notice that for $K = M$ this is the expected number of solutions to the problem, and the computation agrees with the one for the standard backtracking tree search. The expected number of consistency checks at level K will be

$$L^{K+1} p^{(K-1)(K+2)/2} [1 - (1-p)^L]^{M-K-1} (M-K)$$

Figure 6 shows that the expected number of consistent labelings for random constraint satisfaction problems agrees closely with the actual number in the average of several experiments.

4.1 Optimizing the Consistency Check Order in Tree Searching

Suppose we are solving a constraint satisfaction problem and suppose units $1, \dots, K$ have already been assigned labels $\ell_1, \dots, \ell_K$ are 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 $\ell_1, \dots, \ell_K$, that is, we must have $(k, \ell_k, K+1, \ell_{K+1}) \in R$ for $k=1, \dots, K$. To determine the label ℓ_{K+1} , we sequentially go through all the labels in S_{K+1} and perform the K consistency checks: $(k, \ell_k, K+1, \ell_{K+1}) \in R$. 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.

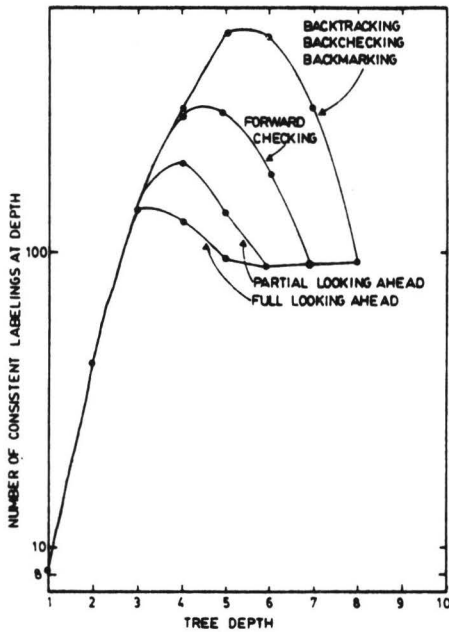


Figure 4 illustrates number of consistent labelings as a function of tree depth, for the 8 queens in the natural unit order.

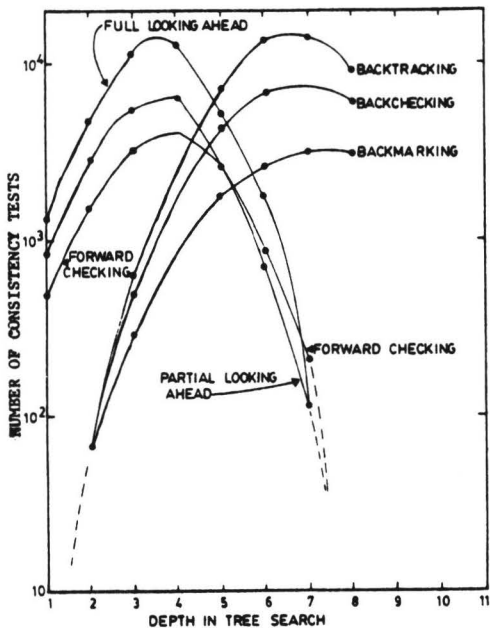


Figure 5 compares the number of consistency tests made at each level in the tree search for six different procedures, for the 8 queens problem in the natural unit order.

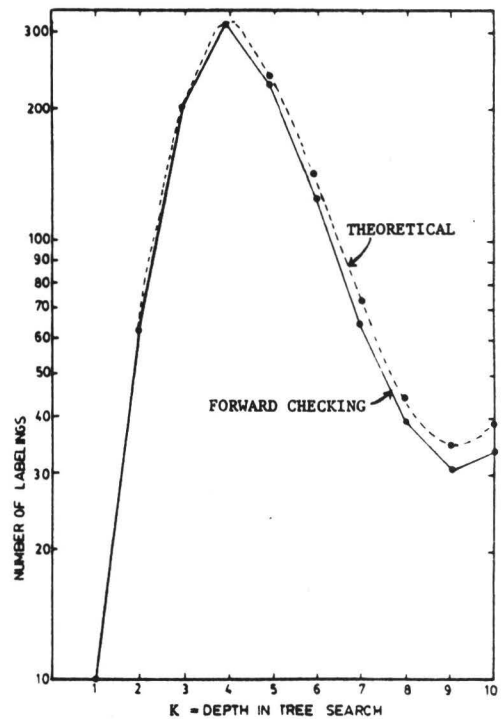


Figure 6 indicates the number of consistent labelings to depth K in the tree search for the average of 25 random constraint satisfaction problems with probability of consistency check success of .65 and number of units = number of labels = 10. The dotted curve is the theoretical expected number of labelings for such a problem.

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 $k_1, \dots, k_K$ be a permutation of $1, \dots, K$ designating the order in which the consistency checks will be performed. The expected number of tests performed can be shown

to be

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

We can minimize the expected number of tests by having $k_1, \dots, k_K$ be any permutation of $1, \dots, K$ satisfying $P(k_1) \leq P(k_2) \leq \dots \leq P(k_K)$. Hence, we must choose to test against those past units whose label is most likely to fail against the current unit label pair.

To illustrate the advantage of using an optimum consistency test order, we consider the 10 Queens problem for the standard back tracking procedure which checks the current units' label against the past units' labels. In the N-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. Hence, the test order should be first unit K-1, then K-2, up to unit 1. Back-tracking requires 1,297,448 tests when done in the wrong order (unit 1, 2, ..., K-1) and 1,091,856 tests when done in the right order.

4.2 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.

Let $k_1, \dots, k_M$ be the order in which the M units are searched on the tree. Let $P_n(k_n | k_1, \dots, k_{n-1})$ be the conditional probability that some label for unit k_n will succeed when unit k_n is the n^{th} one in the tree search order given that units $k_1, \dots, k_{n-1}$ 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,

$$P_n(k_n | k_1, \dots, k_{n-1}) = P_n(k_n | v_1, \dots, v_{n-1}) \text{ for all units combinations } v_1, \dots, v_{n-1}$$

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 n^{th} 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's label 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 $0 < \alpha \leq 1$:

$$P(u) \alpha^{i-1} = P_1(u)$$

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 $k_1, \dots, k_m$, the expected branch depth is given by

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

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

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

Letting the unconditioned success probability of one unit with another be q we have, $P_j(k_j) = \alpha^{j-1} [1 - (1 - q^{n(k)})]$. 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 the order N random relation problem for $4 \leq N \leq 10$. The number of consistency tests required is given in Tables 1 and 2.

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.

Some improvement is shown in the larger N-Queens problems, and considerable improvement appears in the larger order N random constraint problems. This improvement increases with problem size in the random constraint problem with $p = .65$.

V. CONCLUSION

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 in the large problems tested. This suggests that the entire set of look ahead operators described by Haralick et. al. (1978), Haralick and Shapiro (1979a, 1979b), the discrete relaxation described by Waltz (1972) and Rosenfeld et. al. (1976) would be more efficiently implemented by omitting the consistency tests required by future units against future units. Future analytic and experimental work needs to be done to determine if this in fact is generally true. Further work also needs to be done to explore the tradeoff between data structure overhead and number of consistency tests so that the lowest CPU time algorithms can be determined.

Table 1 Number of Operations in N-Queens Problem for Normal and Optimal Unit Order

N	Backtracking	Backchecking	Full Looking Ahead		Part Looking Ahead		Backmarking		Forward Checking	
	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	2016	1496	2095	2082	1703	1708	944	921	964	946
7	9297	6042	8942	8941	6511	6318	3236	3168	3338	3229
8	46752	27450	35323	35211	25882	25062	12308	12095	13024	12108
9	243009	131538	153455	151275	112327	106247	50866	50027	55326	49856
10	1297558	643658	661017	636377	496455	449666	220052	211635	242174	205970

Table 2 Number of Operations in Average of 5 Random Constraint Satisfaction Problems with Consistency Check Success Probability .65, for Normal and Optimal Unit Order

N = number of units = number of labels

N	Backtracking	Backchecking	Full Looking Ahead		Part Looking Ahead		Backmarking		Forward Checking	
	Normal	Normal	Normal	Optimal	Normal	Optimal	Normal	Optimal	Normal	Optimal
4	243	195	230	223	184	174	132	133	133	121
5	1043	731	760	712	599	554	425	382	414	355
6	4637	2514	2543	2288	1966	1700	1288	1159	1273	979
7	12040	6722	5722	5156	4697	3875	3175	2486	3057	2069
8	25893	13490	10779	9306	9111	7233	6089	4300	5978	3425
9	118086	55318	30799	25904	26788	19174	21170	10246	18616	8673
10	163983	73260	44655	36675	41232	28872	28314	14892	26288	12022

References

(sorry, no space)