December 1981

IMPROVED CONSTRAINT SATISFACTION ALGORITHMS USING INTER-VARIABLE COMPATIBILITIES

by

Bernard Nudel

DCS-TR-109

#214

Department of Computer Science
Rutgers University
New Brunswick, New Jersey

This work was started as part of course CS602 with Saul Amarel and is continuing in collaboration with Bill Steiger and Saul Amarel

TABLE OF CONTENTS

List of Figures

List of Tables

ABSTRACT

This report addresses the problem of improving algorithms for solving **consistent-labeling** (also called **constraint-satisfaction**) problems. The concept of **compatibility** between variables in such problems is introduced. How to obtain compatibilities analytically and empirically is discussed, and various compatibility-based heuristics (as well as some useful but less effective non compatibility-based heuristics) are developed to improve a version of the Waltz algorithm which was found best of a set of consistent-labeling problem algorithms tested by Haralick [5]. Empirical results with these heuristics are very encouraging, with over an order of magnitude improvement in performance with respect to the basic algorithm on a set of randomly generated consistent-labeling problems.

1. INTRODUCTION

The **consistent-labeling (C-L) problem**, or **constraint-satisfaction problem**, is a generalization which subsumes many specific problems of interest in Artificial Intelligence and Operations Research. Some of these are: packing, scene labeling and matching, graph homomorphism and isomporphism, boolean satisfiability and proposition theorem proving.

A consistent-labeling problem is characterized by:

1. A finite set of variables, each one having an associated finite domain in which it can take values.

2. A set of constraints between variables. Each such constraint can in general be of any "arity" between 1 and the number of variables. If some constraints are between two variables, but no constraints are between more than two, then we have a **binary consistent-labeling problem**. Note that each domain specification in point 1 above is also a type of (unary) constraint.

3. A goal, which is to find one or more sets of assignments of variables to values in their corresponding domains, such that for each assignment set all constraints are simultaneously satisfied.

Quite a range of algorithms exist to solve consistent-labeling problems. Contributors to this range include Golomb [3], Waltz [9], Mackworth [8], Gaschnig [2], and Haralick [4, 5]. Haralick, in his 1980 paper [5], concentrates on binary consistent-labeling problems and studies the complexity of various algorithms in finding *all* consistent-labeling solutions. He presents a very nice empirical and analytical comparison between several algorithms. Like Haralick [5] we study the complexity of obtaining all consistent-labelings for binary C-L problems, but we are not interested in comparison of algorithms whose differences are non problem-specific; rather we take the **Forward Checking** algorithm, which Haralick finds to have the best general (non problem-specific) strategy, and study various ways of introducing problem-specific improvements to it (other algorithms could be improved in a similar way).

More specifically, our approach differs from most others (for exceptions see [1] and [7]) in emphasizing the role of the constraint-dependent information for a given problem in improving the control strategy of C-L algorithms. We ask "How much more can an algorithm be improved by incorporation of problem-specific information which has previously not been taken advantage of?" To be worthwhile this information must be:

- Efficiently extractable from a problem statement: Its derivation should not offset the advantage of knowing it.

- Generally extractable from the problem statement: It should be of a type that can be derived for all, or a large subclass, of C-L problems otherwise any improved algorithm based on this information will not be relevant.

- Effectively employable: it must result in worthwhile gains in algorithm efficiency.

In the following we introduce the notion of pair-wise inter-variable **compatibilities** which satisfy all the above requirements. They are:

- often extractable analytically from a problem statement, at no (machine) computing cost. Examples of this are given in section 3.1. Any difficulty in terms of human effort in doing this can often be justified in that compatibilities for an infinite class of constraints can be obtained once and for all, and the (human) cost amortized over a great number of solutions.

  On the other hand, computer extraction of exact compatibilities can be carried out at a worst case complexity of O(n↑4) where n is the number of variables, each having domain size n.

  A spectrum of methods exists (chapter 3) that can provide exact or approximate compatibilities at costs intermediate between the above zero and O(n↑4) costs, dependent on the particular C-L problem constraints being considered.

- extractable analytically for many C-L problems and computer-extractable for all C-L problems.

- effectively usable on the average C-L problem, judging from the empirical study of chapter 6. There more than an order of magnitude improvement in performance is obtained over the basic Forward Checking algorithm, and nearly that much compared to Haralick's [5] improved algorithm which does incorporate a degree of problem dependent guidance.

Haralick's mathematical analysis of some of his algorithms is in fact in terms of what we are calling compatibilities, but he models C-L problems as having the same compatibility between all pairs of problem variables; this will not usually be the case in practice. It may be an adequate model for Haralick's purpose, which was to get a rough analytic comparison between several non problem-specific strategies for solving C-L problems[1] . However, it is the very non-equality of the intervariable compatibilities in the usual C-L problem that allows us to obtain our compatibility-based heuristics for increasing C-L algorithm efficiency.

Ten new heuristics are presented and tested, and all are found to result in improved average performance, even two of the ten which make no use of compatibilities. However, we find that the more an algorithm makes use of compatibilities the better is its performance.

---

[1] In work carried out after that of this report, we have obtained the analogues of Haralick's analytic results for the more general situation of non-equal intervariable compatibilities. These expressions show analytically, and experiments confirm empirically, that Haralick's complexity analysis in terms of one fixed average compatibility value for a problem can be highly inaccurate when the actual compatibilities show a significant spread about their mean value. That is, the predicted complexity, for a given problem and a given algorithm, given by his expressions can be significantly in error. This suggests then that even Haralick's intended goal of understanding how various algorithms compare for a given problem, may not have been achieved for problems with non-equal compatibilities. His empirical results don't help here because they are also all for problems having essentially zero spread amongst the compatibilities.

2. COMPATIBILITIES

The **pair-wise compatibility**, p_{ij} , between variables v_i and v_j is simply the proportion of the possible pairs (unordered) of a value for v_i and a value for v_j , that are consistent according to the constraints of the given problem. Values are chosen from the domains, D_i and D_j , of sizes M_i and M_j, for the respective variables. Thus p_{ij} is the proportion of the cartesian product D_i X D_j that is consistent. If the two variables don't mutually constrain each other then p_{ij} is 1. We then call v_i and v_j **totally consistent**. If there are no mutually compatible instantiations of the two variables, from their respective domains, then p_{ij} is 0. We then call the two variables **totally inconsistent**. The full compatibility information for a problem of n variables is given by the p_{ij} for the $\binom{n}{2} = n \times (n-1)/2$ ways of choosing two variables, v_i and v_j, from the n variables. We define the **compatibility matrix** to be formed from these p_{ij} in the obvious way, noting that $p_{ij} = p_{ji}$ and that p_{ii} is undefined.

Triple-wise compatibilities p_{ijk} can be defined analogously, and similarly for k-wise compatibilities. As k increases, the extraction of $p_{i1i2..iK}$ for each combination of subscripts becomes a more and more difficult sub-C-L problem of the original, and we need to invest more effort to extract the compatibilities than they are worth. However, it may still be worthwhile to extract triple-wise compatibilities for use in refinements of the pair-wise compatibility-based heuristics presented here. This has not been looked into in the present report.

In chapter 3 we discuss the extraction of the compatibilities, given a problem. Chapter 4 discusses the particular C-L problem algorithm, Forward Checking, we will be successively refining, and gives two non-compatibility based improvements, one from Haralick [5] and one new one. In chapter 5 we discuss how compatibilities may be used to further improve the complexity of the Forward Checking algorithm. Chapter 6 presents results of an empirical study of the various compatibility-based (and non compatibility-based) heuristics developed. We find that the more that compatibility is used in guiding the Forward Checking algorithm, the greater is the improvement in complexity.

3. EXTRACTING COMPATIBILITIES

It seems that in many cases, inter-variable compatibilities can be found analytically, hence incuring zero (machine) computation cost while affording often significant reductions in search complexity. Below, we outline several cases where compatibilities are analytically available. These are not being cited because they are *the* important cases for Computer Science, but rather as examples that compatiblities are derivable analytically. The second example shows that large classes of common constraints can be handled this way.

Where an analytic solution is not forthcoming, there are several alternatives:

1. **Analytic approximation** may be possible and adequately accurate.
2. **Stochastic probing** is always possible and may be adequately accurate.
3. **Full enumeration** by computer is always possible and is fully accurate. Its cost is $O(M_i x M_j)$ at worst, for the pair of variables v_i and v_j having domain sizes M_i and M_j; the cost is much less in many cases when combined with certain partial analytic analyses beforehand; we call this **analytically-aided enumeration**. Even in the worst case, the overall cost of this preprocessing is no more than $O(n\uparrow 4)$ when each domain size equals the number of variables, n, since there are $\binom{n}{2}$ variable pairs to process. A preprocessing cost of $O(n\uparrow 4)$ is often more than recovered in search reduction. It is a central concern for the future, to characterize when this trade off is justified.

The following presents a spectrum of approaches to obtaining compatibilities, presented roughly in order of increasing reliance on the computer. The first two approaches are purely analytic making no use of the computer (or at least no use whose complexity is a non-constant function of the problem size). The rest are computer-based to an increasing extent and therefore correspond to **preprocessing** algorithms to be employed prior to the search itself.

3.1 Exact Compatibilities Analytically

3.1.1 The Constraints of the n-Queens Problem

For arbitrary n, the compatibility between v_i and v_j, the variables respectively for the i-th and j-th row queen position, can be readily found analytically to be:

$$p_{ij} = (n\uparrow 2 - 3n + 2|i-j|)/n\uparrow 2$$

3.1.2 The Class of Linear Equality Constraints: ax + by = c

The compatibility between two consecutive-integer valued variables, x and y, satisfying a mutual linear constraint ax+by=c, with a, b and c integers, can be found analytically using some basic results of Diophantine Equation theory. We give below the results for the number of compatibilities, I_{xy}, in the Cartesian product of the domains of variables x and y; the actual compatibility, p_{xy}, between the variables is this number divided by the product of the domain sizes.

Call the lower and upper bounds of the domain of x: x_L and x_U, and the lower and upper bounds for y: y_L and y_U. Make the following definitions:

$$\text{Alpha} = \text{Min}[(c-by_L)/a,\ (c-by_U)/a]$$
$$\text{Beta} = \text{Max}[(c-by_L)/a,\ (c-by_U)/a]$$

If Alpha and Beta are less than x_L or if Alpha and Beta are greater than x_U then $I_{xy}=0$, because ax+by=c does not pass through the Cartesian product of the domains of x and y.

If neither of the above hold, then define:

g = the greatest common divisor of a and b.

If g does not divide c then $I_{xy}=0$, because ax+by=c has no integer solutions for x and y. If g divides c then there exists x_0 and y_0 satisfying $ax_0 + by_0 = g$. Finding such an (x_0, y_0) pair and finding g is possible using Euclid's algorithm, whose complexity is independent of the problem size as characterized by the domain sizes M_x and M_y of x and y.

If we find that g does divide c then define:

$$x^L = \text{Max}[\text{Alpha},\ x_L]$$
$$x^U = \text{Min}[\text{Beta},\ x_U]$$

$$x_1 = (c/g)x_0$$
$$y_1 = (c/g)y_0$$

We then have that the number of compatibilities between x and y is:

$$I_{xy} = \text{Floor}[(g/b)(x^U - x_1)] - \text{Ceiling}[(g/b)(x^L - x_1)] + 1$$

3.1.3 The Class of Linear Non-Equality Constraints: ax + by ≠ c

For the same situation as in section 3.1.2 but where the constraints between variables x and y are now ax + by ≠ c, it is a simple extension of the result of the previous section which we call I_{xy}^{equality}, that the number of compatibilities is:

$$I_{xy}^{\text{non-equality}} = M_x * M_y - I_{xy}^{\text{equality}}$$

3.1.4 Other Classes of Diophantine Equality and Non-Equality Constraints

The problem of section 3.1.2 was solved by recognizing that we have a Diophantine equation with a twist: we are concerned with solutions in a restricted region of the x–y plane, and we actually require the number of solutions rather than the solutions themselves. Diophantine theory supplied a parameterized expression (not given above) for the possibly infinite number of solutions in the x–y plane. This was manipulated to give the desired results. A similar strategy should be possible for any constraint that is a Diophantine equation, where the parameterized family of solutions are available from Diophantine Equation theory. Also, where the constraint is a non–equality (i.e. LHS $\neq$ RHS) that would otherwise be a Diophantine equality constraint, then the method of taking the compliment, as in section 3.1.3, can be used.

3.2 Approximate Compatibilities Analytically

Sometimes the exact compatibility for a given constraint may be hard to obtain analytically, but a close approximation can be found by analytic reasoning. The latter may also be quite useful in reducing search complexity.

3.2.1 The Class of Inequality Constraints: g(x,y) < c

When successive–integer valued variables x and y are constrained to satisfy $g(x,y) < c$, it may not be easy to analytically derive the number of compatibilities, I_{xy} (even the linear inequality constraint, $ax + by < c$ for a, b and c integer–valued appears to be difficult; See [6] for a partial solution to a more general version of this). However, using integral calculus, usually one can readily find the area on the appropriate side of the curve $g(x,y) = c$ and within the rectangle delimited by x_L, x_U, y_L and y_U. This can lead to a quite accurate analytic approximation to the exact I_{xy}. A common case where this can be done is when $rx + sy < t$ for real valued r, s and t. This can often be done even when x and y are not functionally related. For example, with constraint $(x-r)^2+(y-s)^2 < t^2$ one can approximate I_{xy} as the area of the the circle of radius t centered on (r,s) that falls in the x–y plane rectangle delimited by x_L, x_U, y_L and y_U.

3.3 Approximate Compatibilities by Stochastic Probing

Where not even approximations can be derived analytically, one may want to approximate the compatibilities by stochastic probing (N.B. it may be preferable to get exact compatibilities by full–enumeration or preferably by analytically–aided enumeration as in

sections 3.5.1 and 3.4). Stochastic probing is a random sampling form of full enumeration. For each pair of variables, value-pairs in the Cartesian product are generated at random and the proportion of compatible value-pairs generated to the total generated is taken as the approximation to the true compatibility for that pair of variables. This probing could be carried out for a pair of variables until a predetermined desired degree of accuracy is obtained.

3.4 Exact (or Approximate) Compatibilities by Analytically-Aided Enumeration

It may not be possible to get an exact (or good approximate) expression for a compatibility, but it may still be possible to get an exact (or good approximate) analytic expression for a subexpression from which the exact (or approximate) compatibility can be built up by computer. The building up process is the residual non-analytic computational effort required because of the inability to carry the analysis beyond the individual subproblems involved. This non-analytic part we call the aided enumeration because it can be seen as a full enumeration of section 3.5.1, which has by the aid of a partial analysis, been reduced to one over analytically solved subproblems rather than over all the points of the Cartesian product.

3.4.1 The Class of Inequality Constraints (again): g(x,y) < c

In section 3.2.1 we discussed approximation of compatibilities for inequality constraints g(x,y) < c. However partial analysis of the problem may allow the exact results to be obtained by an aided enumeration.

For example, whenever the constraint can be written as y < f(x) where f is a function, i.e mapping x into only one value we may proceed as follows. The essential part of finding the number of (x, y) pairs that satisfy y < f(x) is to find the number of such pairs under the curve y = f(x) for the set, X, of x values between where the curve enters and leaves the rectangle bounded by x_L, x_U, y_L and y_U. This can be expressed as the sum over all x in X of the number, s(x), of such pairs for that x value. Now, finding s(x) analytically is simple, as it is just Floor[f(x)], but analytically suming these Floors over x in X is not simple. Simplification formulae for sums of Floors are few and far between (the sum of Floors is certainly not the Floor of the sum). Analysis has taken the place of enumeration over the y-value rows of the Cartesian product, but a residual enumeration (i.e the explicit suming steps) must be carried out by machine over the x-value columns for x in X. The complexity for a pair of variables x and y will then be $O(|X|)$ which can be no more than $O(M_x)$. If the inverse of f, f^{-1}, is also a function then we can choose which variable, x or y, should correspond to the columns and which to the rows, and the worst case complexity is $O(\text{Min}[M_x, M_y])$.

Even when f above is not a function, a similar approach may be possible. For example, if the constraint is $(x-r)^2 + (y-s)^2$ i.e t^2 so that

y i.e. sqrt$[t^2 - (x-r)^2]$ + s = f(x)

then we need just sum

s(x) = Floor[the larger f(x) value] - Ceiling[the smaller f(x) value] + 1

over integer x values in the interval [r-t,r+t]. Since we could have done the same with respect to the y variable instead, then the complexity is O(Min[M_x, M_y, 2t+1]).

3.4.2 The Class of Equality Constraints: g(x,y) = c

Whenever the constraint between variables x and y can be expressed so that y is a function of x, y=f(x), then the enumeration over the y-rows can be avoided simply because there is only one candidate y-value to consider for a given x, viz y=f(x). For at most each value in the domain of x we need just generate this y=f(x) and test if this y value is in the domain of y. The number of such y values actually in the domain of y is the desired I_{xy}. The worst case complexity is O(M_x), and if the x values were also functionally related to y, then by choosing over which variable to enumerate, the worst case complexity is O(Min[M_x, M_y]).

If writing g(x,y) < c in the form y < f(x) gives an f that is not a function, but which returns at most w > 1 values of y for any given x, then we can generalize the previous method and each of the several y values returned can be tested for membership in the domain of variable y. The complexity is at worst O(wM_x), or O(Min[wM_x, vM_y]) if we can solve x=h(y) as for y=f(x), where v is the maximum multiplicity analogous to w in this case. In practice though, one would need to consider the time costs for evaluating f(x) versus h(y), and when comparing to full-enumeration, the relative costs of these versus one consistency-check.

3.5 Exact Compatibilities by Full Enumeration

3.5.1 Full-Enumeration

As a last resort, exact compatibilities can be obtained for any C-L problem by simply exhaustively testing each possible value pair for the domains of each different combination of two variables. The complexity per pair of variables x_i and x_j is M_i*M_j. Since there are $\binom{n}{2}$ ways to choose a pair of variables from the problem's n variables this process requires O(n↑4) consistency checks for C-L problems (such as the n-queens type) where each of the n variables has a domain of size n. Even this last-resort approach to obtaining the compatibilities results in no more than a fourth-order polynomial complexity cost during

preprocessing. Seeing that the complexity of solving consistent labeling problems tends to grow exponentially with n, we can expect that an O(n↑4) cost for obtaining the compatibilities may very well prove worthwhile for larger problems.

Of course, one of the former approaches may be useable for the problem at hand to derive the compatibilities with a lesser, possibly zero, preprocessing cost. And to the extent that the preprocessing cost can be reduced, the algorithms presented below that use compatibilities to reduce search complexity, will be more useful.

3.5.2 Fringe-Benefits of Full Enumeration

Doing full enumeration preprocessing may be the least desirable approach if all that is wanted are the compatibilities but it can, at zero or slight extra cost, provide some significant additional benefits for the main algorithm.

- Firstly, having done once every possible consistency check in the problem during this form of preprocessing, the results are available for storing in an array for doing fast lookup when it is necessary to redo the same check successively during the main search phase. (Significant amounts of many-fold consistency-check duplication is a feature of all C-L algorithms to date; see point 6 in section 7). If consistency checks by the normal method ("real" checks) are expensive then array lookup, after each check has been done once normally, can save significant time.

  An approximation to this strategy could be used also when not using full enumeration preprocessing, by filling the array on the fly as the main search algorithm proceeds. This approach would however require an extra test whenever a consistency check is to be done to see if the the corresponding array cell has had the result ("consistent" or "inconsistent") stored there yet. If not, the check would be done in the regular way and then stored there. If it had been already done in the regular way once it would have been stored there already and table look-up would replace doing it again. This would still be much preferable to just doing every repetition of all checks by the regular method, as is done in all algorithms I have seen. But doing a full-enumeration preprocessing gives the advantage that after preprocessing it is known that all possible $\binom{n}{2}*n^2$ checks have been done once and it is not necessary to test for each subsequent consistency check during the main search algorithm, whether it has yet been carried out before it can be simulated by a table look-up. But on the other hand, if we filled the array on the fly and didn't do the $\binom{n}{2}*n^2$ checks during full-preprocessing, the search algorithm in general does less than the $\binom{n}{2}*n^2$ possible *non-identical* checks, although each check it does do is repeated many times in general. So we would do less of the "real checks" which are the expensive ones, but do a little extra work for "has-it-been-checked-yet" testing for each of the much larger total number of checks during the main search.

  Which approach has the advantage, array-filling based on full-enumeration preprocessing or array-filling on the fly during search, is not obvious and depends on the relative times for a regular consistency check versus a table look-up versus a "has-it-been-checked-yet" test as well as on how many of the full $\binom{n}{2}*n^2$ non-identical "real" checks can be avoided by forgoing preprocessing. But whatever the case, it is clear that array-filling by either

of the above two approaches, to allow duplicated checks to be done by table-look-up is of great value for all constraints except those corresponding to the simplest (fastest) types of "real" consistency check. This, it seems, has not been employed yet in algorithms to date.

- More importantly, at slight extra cost, arc-inconsistent [8] values of variables can be identified during a full enumeration preprocessing and eliminated at that stage. This will reduce the complexity of the subsequent preprocessing as well as the main search itself. At worst, no arc inconsistencies will be detected and you get the compatibilities at O(n↑4) and still have the original problem to solve. At best you can solve the whole problem during preprocessing at no more than O(n↑4). Development of this form of preprocessing algorithm is proceeding.

4. THE FORWARD CHECKING ALGORITHM AND SOME NON COMPATIBILITY-BASED IMPROVEMENTS

Haralick in his 1980 paper [5] compares empirically a range of algorithms for C-L problems and finds that the one he calls Forward Checking has the best average performance. We use this algorithm as our starting point[2] from which to improve via a range of heuristic refinements, both compatibility-based and non compatibility-based. The basic algorithm is shown in fig 4-1, copied directly from [5]. Line 5 of fig 4-1a needs to be deleted (it is there only for the other algorithms that Haralick was testing). For the sake of brevity we assume that the reader will refer to [5] for any clarification needed regarding the algorithm.

Forward Checking differs from standard backtracking in an essential way: at each node after a new variable has been instantiated, a "forward search" is done over the still uninstantiated variables to eliminate any values from their domains that are inconsistent with the last instantiation. At each node a table is carried along giving the still viable values for the uninstantiated variables. Thus instantiating at a node in this algorithm, the instantiation values that will be consistent with past variables' instantiations, are known *in advance* for each uninstantiated variable by looking in the table of value sets at that node. There is thus never any need to check the consistency of the last instantiation against earlier ones along the path back to the root, as in standard backtracking. Thus "back-checking" against individual past instantiations after each new instantiation is avoided by "forward checking" against whole sets of still viable values for the uninstantiated variables in order to thin these sets. It might then seem counter-intuitive that Forward Checking is better than Backtracking, except that what appears to be a harder job of forward checking for consistency gets rid of the futile tries once and for all, that Backtracking makes again and again (thrashing) in generating descendants.

The algorithm as it stands carries out instantiation of variables (line 7, fig 4-1a) in the order $v_1, v_2, \ldots, v_n$, where the indexing is done in some standard way by the programmer. This instantiation order stays fixed throughout the algorithm execution for all branches of the search. We call this STD-IO, for Standard Instantiation Order. Similarly, the basic algorithm checks the sets of still viable values of the remaining uninstantiated variables for consistency with the most recent instantiation (line 5, fig 4-1b) - in increasing numerical order of subscript assigned in the chosen standard order. We call this STD-CO, for Standard Consistency-Checking Order. Figure 4-2 shows the search tree for the basic

---

[2] Haralick does find that this algorithm can be made particularly efficient by use of a certain data structure, resulting in an algorithm he calls Bit Parallel Forward Checking. We do not use this version as our starting point because the improved versions we later obtain should have essentially the same relationship to the original whether or not the data structure is used

```
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)

Figure 4-1: The Forward Checking algorithm (line 5 of (a) needs to be deleted).

Forward Checking algorithm applied to the 6-queens problem[3] . By definition, STD-IO and STD-CO are used and the variables have been numbered in the usual way: v_i for the column value of the queen in the i-th row.

4.1 Improving the Instantiation Order

The STD-IO used in the basic algorithm is an obvious place for improvement. Haralick discusses this and incorporates Least-Alternative-First variable selection as an improved instantiation order - i.e. choose the uninstantiated variable with the least number of currently viable values still left in its domain. This will result in the least number of surviving children at a node; the number of surviving children being equal to the domain size of the variable when it is chosen, since each value in a domain is by the nature of the Forward Checking algorithm, consistent with prior instantiations made (see page 12). We will call this the LAF-IO heuristic, for Least-Alternative-First Instantiation Order. Ties are broken by defaulting to the arbitrary order of least-index variable first as in STD-IO.

[3] We choose the 6-queens problem, which might seem large for an example, because it is the smallest n-queens problem for which any performance difference occurs between the basic algorithm and the modifications to be presented later in this chapter

Half of the symmetrical search tree for the 6-queens problem, using both the <STD-IO STD-CO> and <STD-IO LAF-CO> versions of the basic Forward Checking algorithm. Arcs are labeled with the number of consistency checks to get child node. The value for the <STD-IO LAF-CO> version is given in parentheses when this measure differs for the two algorithms.

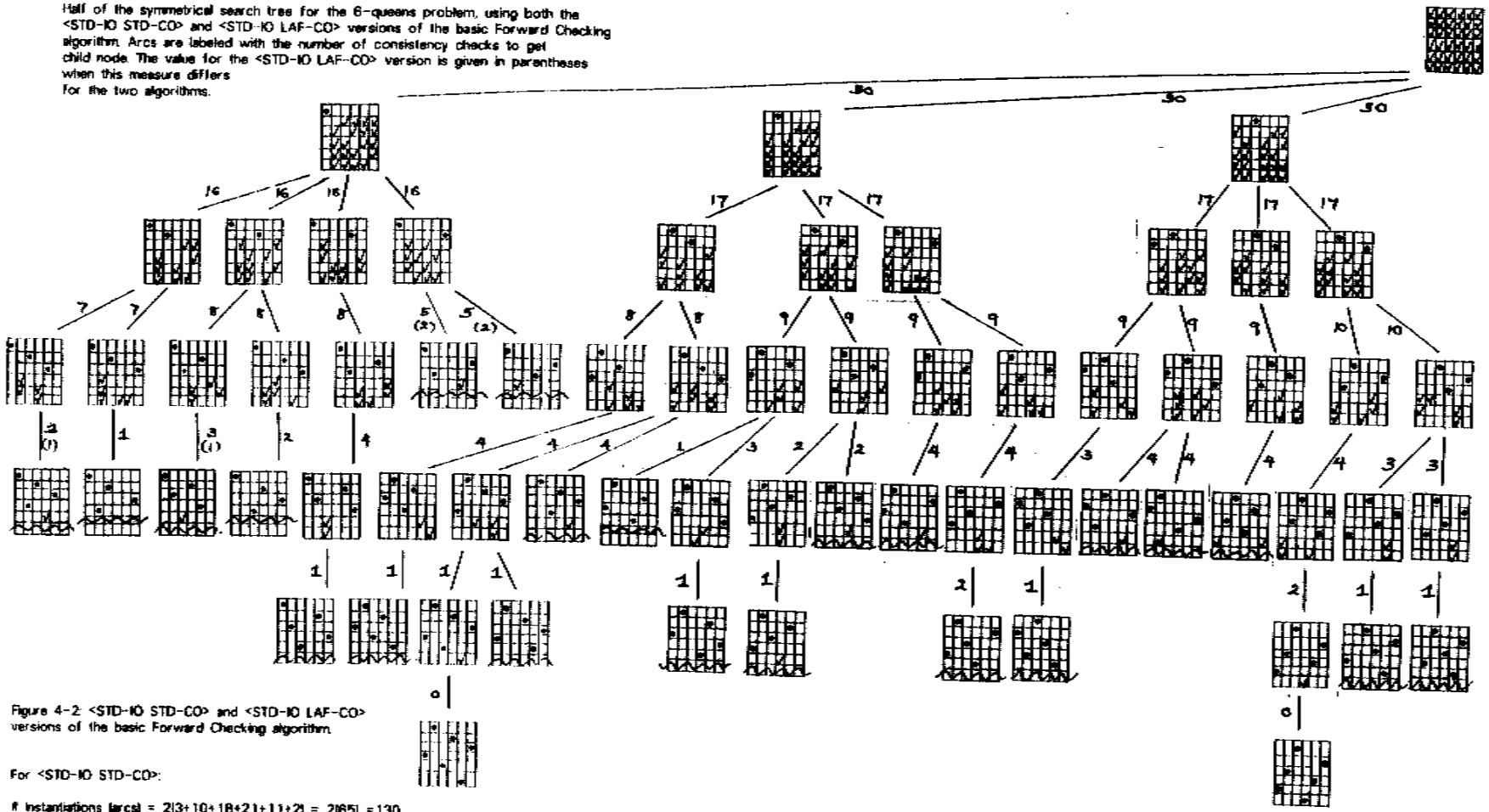

Figure 4-2: <STD-IO STD-CO> and <STD-IO LAF-CO> versions of the basic Forward Checking algorithm.

Dots indicate instantiations for the respective variables (rows); ticks indicate the still viable (consistent) values for the respective as yet uninstantiated variables.

For <STD-IO STD-CO>:

instantiations (arcs) = 2(3+10+18+21+11+2) = 2(65) = 130

Consistency Checks = 2(30+30+30) + (16+16+16+16+17+17+17+17+17+17)
+(7+7+8+8+8+5+5+8+8+9+9+9+9+9+9+10+10)
+(3+1+3+2+4+4+4+4+1+3+2+2+4+4+3+4+4+4+4+3+3)
+(1+1+1+1+1+1+2+1+2+1+1)

= 2(90+166+147+66+13)

= 964

Wavy lines indicate a row found, by <STD-IO STD-CO>, to have no still viable (consistent) values.

For <STD-IO LAF-CO>:

instantiations (arcs) = same as shown at left for <STD-IO STD-CO>.
Changing the CO-heuristic never changes the search tree and thus neither the number of instantiations.

Consistency Checks = 964 - 2(5-2+5-2+3-1+3-1)

= 964 - 2(10)

= 944

Figure 4-3 shows the search tree when basic Forward Checking is augmented with the LAF-IO heuristic on the same 6-queens problem as in fig 4-2. Note that the total number of consistency checks required (line 5, fig 4-1b), decreases from 964 with STD-IO to 956 with LAF-IO, and the total number of arcs or instantiations (line 2, fig 4-1a), decreases from 130 to 118.

4.2 Improving the Consistency-Checking Order

A less obvious but often also significant consideration is: 'given that one uninstantiated variable has been chosen and instantiated, in what order should subroutine CHECK_FORWARD choose from the other uninstantiated variables when checking the consistency of the latest instantiation with their still viable value sets (line 5, fig 4-1b)?

The consistency-checking order will not effect the search tree generated and hence neither the number of arcs, which is a function only of the instantiation order chosen. But it will effect the total number of consistency checks required for a problem, since the sooner we select for consistency checking an uninstantiated variable that retains no viable values given the last (and prior) instantiations, when such a variable exists, the less irrelevant consistency checking need be done. (Where no such uninstantiated variable results from the last instantiation, the consistency-checking order for that node is irrelevant.)

But will decreasing the total number of consistency checks really improve the algorithm's complexity if it doesn't effect the total number of arcs (instantiations) in the search tree? Haralick [5] shows that it *is* the total number of consistency checks that is an appropriate measure of the basic Forward Checking algorithm's complexity, and not the number of arcs in the generated search tree[4]. Of course, decreasing the number of arcs in a search will in general itself decrease the total number of consistency checks, so that appropriate heuristics for *both* the instantiation order (IO-type heuristics) and the consistency-checking order (CO-type heuristics) will contribute to improved complexity as measured by total number of consistency checks.

Haralick does discuss improving the consistency-checking order as a way of improving C-L algorithm efficiency, but his analysis is not applicable to the Forward Checking algorithm (or any of the other Forward Checking-like algorithms he deals with - Full Looking Ahead and Partial Looking Ahead). Rather it is applicable to standard Backtracking and its relatives dealt with by him - Backchecking and Backmarking.

We suggest, as a first way to improve the Forward Checking algorithm and its relatives, the Least-Alternatives-First heuristic below, LAF-CO, for ordering variables in consistency

---

[4] See section 6.3 for a discussion of related issues

Half of the symmetrical search tree for the 6-queens problem, using both the <LAF-IO STD-CO> and <LAF-IO LAF-CO> versions of the basic Forward Checking algorithm. Arcs are labeled with the number of consistency checks to get child node. The value for the <LAF-IO LAF-CO> version is given in parentheses when this measure differs for the two algorithms.

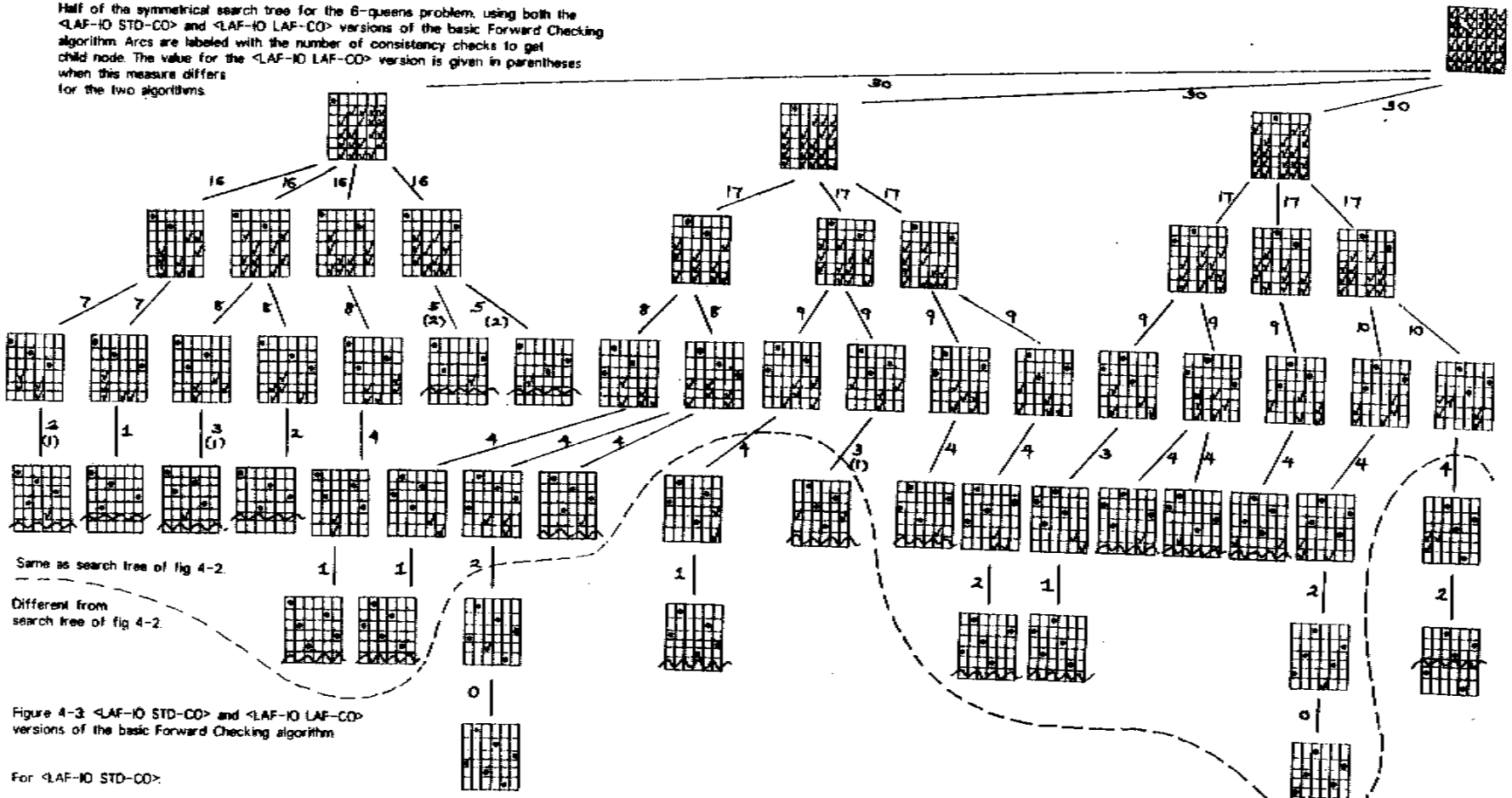

Figure 4-3: <LAF-IO STD-CO> and <LAF-IO LAF-CO> versions of the basic Forward Checking algorithm.

For <LAF-IO STD-CO>:

instantiations (arcs) = 2(3+10+18+18+8+2) = 2(59) = 118

Consistency Checks = 2(30+30+30) + (16+16+16+16+17+17+17+17+17+17)
+(7+7+8+8+8+5+5+8+8+9+9+9+9+9+9+10+10)
+(3+1+3+2+4+4+4+4+4+3+4+4+3+4+4+4+4+4)
+(1+1+2+1+2+1+2+2)

= 2(90+166+147+63+12)

= 956

Dots indicate instantiations for the respective variables (rows); ticks indicate the still viable (consistent) values for the respective as yet uninstantiated variables.

For <LAF-IO LAF-CO>:

instantiations (arcs) = same as shown at left for <LAF-IO STD-CO>. Changing the CO-heuristic never changes the search tree and thus neither the number of instantiations.

Consistency Checks = 956 - 2(5-2)+(5-2)+(3-1)+(3-1)+(3-1)

= 956 - 2(12)

= 932

Wavy lines indicate a row found, by <LAF-IO STD-CO>, to have no still viable (consistent) values.

checking. This is analogous to the LAF-IO heuristic above for instantiation ordering and neither heuristic makes use of compatibility information. Compatibility-based extensions of both LAF-IO and LAF-CO will be made in chapter 5.

Since in consistency checking we want to quickly find the variable, when one exists, amongst the uninstantiated variables whose still viable set of values is reduced to be empty by the last instantiation made - it would be reasonable to choose variables for checking in order of increasing size of their still viable domains. This is because one would expect the variables with the smaller currently viable domains to be more likely to have all their domain values clash with the last instantiation. This is what we call the LAF-CO heuristic.

When the basic Forward Checking algorithm is augmented by the LAF-CO heuristic for consistency checking but still using the basic STD-IO heuristic for instantiation we refer to it as the Forward Checking <STD-IO LAF-CO> algorithm. Since as mentioned, changing only the consistency checking order does not effect the search tree itself, figure 4-2 is also used to show the performance of <STD-IO LAF-CO> as well as that of <STD-IO STD-CO> introduced above. Similarly, figure 4-3 is used to show the performance of <LAF-IO LAF-CO> as well as that of <LAF-IO STD-CO> introduced above. In both figures, where the LAF-CO version results in a different number of consistency checks after an instantiation, the corresponding arc is labeled with that number in parentheses. Non-parenthesised arc labels indicate the common number of consistency checks for both algorithms of a figure.

4.3 Overview and Prospectus

We have presented explicit search trees for the 6-queens problem solved with all four of the possible combinations of the above two IO-type heuristics and two CO-type heuristics incorporated into the basic Forward Checking algorithm. Table 4-1 gives the total number of consistency checks and arcs for all four combinations for the 6-queens problem.

The LAF-IO heuristic was used by Haralick in [5] while the LAF-CO heuristic is new. Together they are seen to be more effective than individually in the case of table 4-1. In chapter 6 a fuller comparison of the above four combinations will be made - viz. for the range of n-queens problems with n = 4 to 11 and some combinations will be compared over sets of random C-L problems for n = 4, 6, 8, 10 and 12 variables. This will also be done for the compatibility-based extensions of both LAF-IO and LAF-CO that will be developed in chapter 5.

Instantiation Order \ Consistency-Checking Order	STD-CO	LAF-CO
STD-IO	964, 130	944, 130
LAF-IO	956, 118	932, 118

Table 4-1: Total number of consistency checks and arcs for four possible combinations of heuristics with the Forward Checking algorithm applied to the 6-queens problem

5. USING COMPATIBILITIES

Chapter 4 discussed the basic Forward Checking algorithm for C-L problems, and presented two independent heuristics for refining the algorithm. One heuristic, LAF-IO, was from [5] and improved the variable-instantiation order. The other heuristic, LAF-CO, was new and improved the consistency-checking order for variables. Including the basic algorithm's two "heuristics" for these two issues, STD-IO and STD-CO, four different algorithms were possible by choosing independently one of the two IO heuristics and one of the two CO heuristics. None of these however yet make use of the inter-variable compatibilities introduced in chapter 2. In this chapter we present two more IO-class heuristics COMP1-IO and COMP2-IO, and one more CO-class heuristic COMP-CO, each making use of COMPatibilities. By selecting independently from the now enlarged set of four IO heuristics and three CO heuristics, we will be able to obtain twelve variations of the basic Forward Checking algorithm. The results of chapter 6 show that the more an algorithm makes use of compatibilities, the better it performs; significant improvement over the basic <STD-IO STD-CO> algorithm is obtained. Even better compatibility-based IO-class and CO-class heuristics are no doubt possible.

5.1 Instantiation Order Heuristic COMP1-IO

This heuristic instantiates variables as in the non compatibility-based LAF-IO heuristic of section 4.1, except for a deceptively minor change: ties are broken based on the use of compatibilities. A tie occurs whenever one or more yet uninstantiated variables has the same size domain of viable values left so that LAF-IO cannot choose between them. (In LAF-IO itself ties are broken arbitrarily by defaulting to the STD-IO method of least-index variable first.)

It seems that choosing for instantiation the variable with the least number of values, as does the LAF-IO heuristic, has a first order effect on the complexity of the Forward Checking algorithm by minimizing the number of surviving first generation descendants of a node. The rationale for breaking ties in COMP1-IO is to aim at getting the most from the second order effect of decreasing the number of second generation descendants. Minimizing the number of *surviving* first generation descendants of a node can be done without error in the Forward Checking algorithm since at each node, variables have associated with them a table of all, and only, their values that are still consistent with prior instantiations along the path to the root from this node[5] (see page 13). However this table

[5] In standard backtracking the number of first generation descendants of a node that will survive the consistency check is not known, and at best could be known only approximately if compatibilities were used. In Forward Checking this kind of uncertainty is pushed off one level further, to the second generation below a node.

allows only a weak indication of how many second generation descendants will survive; taking the product of the domain sizes for the variables proposed to be instantiated at the two next levels will give an upper bound on number of second generation survivors. A better estimate uses compatibilities. Take the product $p_{ij} \ |v_i| \ |v_j|$ for the variables v_i and v_j proposed to be instantiated at the next two levels, where $|v_i|$ is the domain size of currently still viable values for variable v_i. This is a good estimate of the number of survivors amongst the second generation descendants of a node, being the *exact* number of compatible value pairs formable for these two variables *when their whole original value sets are used*.

Explicitly then, at each node: use the LAF-IO heuristic if no ties occur. However if a set T, of variables tie from amongst the set U of as yet uninstantiated variables, choose to instantiate the variable that satisfies

$$\min_{v_i \text{ in } T} \ \min_{v_j \text{ in } U \setminus v_i} p_{ij} \ |v_i| \ |v_j|$$

If there is more than one such variable we choose arbitrarily according to the STD-IO method of least-index variable first. This could be improved by breaking these last ties based on an estimate of the number of third generation survivors, using the estimate $p_{ij} \ p_{ik} \ p_{jk} \ |v_i| \ |v_j| \ |v_k|$, but we didn't investigate this and the obvious sequence of such improvements possible.

5.2 Instantiation Order Heuristic COMP2-IO

This heuristic uses a simple and drastic way to effect the instantiation order: the algorithm should simply give up immediately realizing that there is *no* instantiation order that can solve the problem whenever any compatibility p_{ij} value is zero! If no pair of values from the domains of two specific variables is consistent then no n-tuple of values from the domains of the problem's n variables can be consistent. When no $p_{ij} = 0$ this heuristic proceeds as in COMP1-IO.

This use of compatibilities is possible of course only if the compatibilities extracted, analytically or empirically, for the problem are exact so that $p_{ij} = 0$ really means that no value-pair is consistent in $D_i \times D_j$. If the compatibilities are extracted empirically but by random sampling rather than exhaustive sampling, then we have only approximations q_{ij}, to the true compatibilities p_{ij}. A q_{ij} of zero only means that for the *tested* v_i and v_j value pairs, none were found to be consistent. However there may yet be an untested consistent pair which allows a consistent value n-tuple extension to be formed. Even approximate compatibilities however, may perhaps be used in this way if the goal is only a probabilistically good algorithm.

It seems that the average random C-L problem tends towards having no solutions as the number of variables increases[6]. If this manifests itself at the pairwise level (i.e. there is at least one $p_{ij} = 0$), as opposed to being manifested only at the (k > 2)-wise level[7], then using this COMP1-IO heuristic could be sufficient to cause average performance on random problems for large n to become O(constant), apart from the cost of extracting the compatibilities, since the algorithm can immediately give up based on seeing the zero compatibilities that will be present for the average problem. Of course, the problem-space corresponding to C-L problems *required solved in real life* may not have the property that there are zero solutions to the average problem as n goes to infinity. (Haralick [5] shows that the n-queens problem has an exponentially increasing number of solutions as n increases.) Nevertheless, whenever a problem does have no solutions because of a pair-wise total incompatibility of two variables, then this heuristic can be of great use. If k-wise compatibilities were used then problems that have no solutions, because of a k-wise total incompatibility of some k variables (even though no (k-1)-wise or lower order subsets of variables were totally inconsistent so that pair-wise compatibilities say, wouldn't help), would also benefit from this type of heuristic.

5.3 Consistency-Checking Order Heuristic COMP-CO

As explained in section 4.2, the goal of optimizing the order of consistency checking (of the last instantiation with the still viable value sets of the yet uninstantiated variables), is to pick as soon as possible that uninstantiated variable having no values left consistent with the last instantiation, when such a variable exists. An inconsistent path can thus be abandoned sooner. LAF-CO reasons that the smaller the value set the more likely it is to have no consistent member. But the size of a variable's value set is only an upper bound on the number of values consistent. Compatibilities can be employed to give a better estimate of the number of values consistent. Given that variable v_i has just been instantiated according to whatever IO heuristic is used, and that the set of variables U, remains uninstantiated, select variable v_j from U for consistency checking with the value just set for v_i according to increasing value of:

$$p_{ij} \; |v_j|$$

---

[6] This needs to be checked and the various (if any) sample spaces that this holds for be explicitly described. It did seem to hold empirically for the sample space of section 6.2, where we get 27, 2, 0, 0, and 0 total number of solutions respectively for each of the sets of ten random problems with n = 4, 6, 8, 10 and 12 variables. See Appendix A

[7] It is not clear empirically in section 6.2 if this is the case. There, instead of a sequence of increasing values, we get 3, 3, 3, 5, and 3 problems respectively whose lack of solutions manifests itself at the pairwise level, from the sets of ten random problem with n = 4, 6, 8, 10 and 12 variables. See the zeros introduced when using the COMP2-IO heuristic in the tables of Appendix A

The reason for this is the same as given in section 5.1 to justify the COMP1-IO heuristic. There we talk about $p_{ij}\ |v_i|\ |v_j|$, but since the i-th variable remains fixed here we can drop the factor $|v_i|$ in forming the estimates.

5.4 Better Heuristics

The above heuristics are based on the attitude that optimization of instantiation order and of consistency checking order can be usefully considered independently or decoupled. The empirical results of chapter 6 confirm this view in that an IO-class heuristic could be added to a CO-class heuristic, or vice-versa, with always an overall gain in performance. However, by taking explicitly into account the way in which the instantiation order interferes with the effect of the consistency-checking order, even better heuristics could be expected. That is, if we knew when, it might be worthwhile making a less than optimal choice (in terms of the COMP2-IO heuristic say) for the instantiation order so that when combined with a non-decoupled CO heuristic, the overall effect is superior. This kind of approach requires a mathematical analysis, not undertaken here, which allows an analytic treatment of the complexities of the coupling effect.

6. RESULTS

The heuristics introduced in chapters 4 and 5 are here tested empirically, in various combinations, on a sequence of n-queens problems and on a sequence of sets of random C-L problems. We study the complexity of the resulting algorithms as a function of problem size measured by the number of problem variables. The complexity is measured in terms of the number of consistency checks required by the algorithms, a measure determined by Haralick [5] to be appropriate for the basic Forward Checking algorithm, as opposed to the number of instantiations (arcs in the search tree) required. Results for the latter measure are included for interest's sake. The applicability of the number of consistency checks as a complexity measure for the *modified* Forward Checking algorithms presented here, is discussed in section 6.3

6.1 Experiments on the n-Queens Problem Sequence

Table 6-1 presents results for algorithms applied to n-queens problems for n = 4 to 11. Twelve variations of the basic Forward Checking algorithm are formable by independently selecting from the four IO-class heuristics and the three CO-class heuristics presented in chapter 5. Results for nine of these are presented. The ones using the COMP2-IO heuristic are not given separately because for n-queens problems, all of which have solutions when n > 3, the COMP2-IO heuristic gives the same results as the COMP1-IO heuristic. The COMP2-IO heuristic has an advantage only for instituting "early give-up" for certain problems having no solutions. Its usefulness will be seen in the following section where random problems are dealt with.

6.2 Experiments on a Sequence of Random C-L Problems

For each value of n, a set of ten random C-L problems was generated, each problem having n variables and each such variable having a domain of n values. The n-queens problems used in section 6.1 are examples of C-L problems where each variable has the same size domain as the number of variables in the problem. However, in general a C-L problem of n variables needn't have all its variables of domain size n. Thus the sample space for our random sampling is a subspace of the full C-L problem space. Haralick in [5] also restricts himself to this subspace. If the full space had been used it would have taken considerably more problem samples to get an average which is representative. However the relative performance of the various algorithms tested could be expected to reflect that for the full C-L problem space. This is because once the first level or two of the search tree has been passed using a Forward Checking type algorithm, the still viable

IO \ CO		STD-CO			LAF-CO	COMP-CO
	n	#Checks	#Arcs	#Solns	#Checks	#Checks
STD-IO	4	76	16	2	76	76
	5	282	53	10	282	282
	6	964	130	4	944	944
	7	3338	463	40	3248	3248
	8	13024	1724	92	12732	12732
	9	55326	7031	352	54030	54030
	10	242174		724		
	11					
LAF-IO	4	76	16		76	76
	5	282	53		282	282
	6	956	118		932	932
	7	3244	393		3164	3164
	8	12066	1360		11876	11856
	9	49814	5398		49216	49168
	10	204954	19744			
	11	835124	85939			
COMP1-IO (also COMP2-IO)	4	76	16		76	76
	5	282	53		282	282
	6	956	118		932	932
	7	3252	397		3180	3178
	8	12018	1340		11822	11798
	9	49378	5337		48650	48588
	10	203802	19820			199394
	11		85527			904622

Table 6-1: Nine variations on basic Forward Checking applied to a sequence of n-queens problems. The rectangle at the top left indicates the two versions tested by Haralick [5]. Those to the right and/or under the line of X's are new versions making use of compatibilities. The two other versions are new but do not make use of compatibilities. number of arcs is same for algorithms with same IO heuristic, and so is not repeated in each column.

value sets for the various still uninstantiated variables have widely different sizes in general, approximating the size differences we would get by sampling from the full space of problems.

Random relations were generated as follows[8] : For each pair of variables v_i and v_j[9], a "target probability" P_{Tij} is randomly generated between 0 and 1 inclusive. Then for each pair in the cartesian product D_i X D_j the pair is added with probability P_{Tij} to the list of consistent value pairs for the random relation. (Note that since they are added with probability P_{Tij} then P_{Tij} will be close to, but will not be the exact compatibility between variables v_i and v_j. The exact compatibility, to be used by the compatibility-based heuristics in the various algorithms being tested, is computed on the fly as the pairs are actually added.) Such a value pair for two given variables is represented in our list as the four-tuple (v_i v_i-value v_j v_j-value).

Table 6-2 presents the results. For each n the values presented are the average over 10 random problems. Appendix A gives the detailed results on a problem-by-problem basis. Figure 6-1 presents these average results graphically.

6.3 Complexity Measures and True Complexity

As mentioned earlier, Haralick [5] concludes that the number of consistency checks is an appropriate measure of complexity for the basic Forward Checking algorithm. We have presented all our empirical results in terms of this measure (as well as the number of instantiations or arcs in the search tree; included for interest). But is it still an appropriate measure for the modified versions of the basic algorithm? The modified algorithms use heuristics that incur a cost of their own in computing the information needed at each node to base the heuristic decision upon. Does some measure other than number of consistency checks, then become asymptotically limiting?

One way to decide is to analyze the algorithm theoretically. However, average case complexity will be a complex function of the problem compatibilities. Another way, at least as a first step, is an empirical comparison of actual *execution times* to see if they are consistent with the variation in the *number of consistency checks* measure.

But there is a difficulty in doing this for the random problems we are working with. Each

---

[8] This method is different than that in [5].It is not clear however whether it corresponds to the same implicit assumption about frequency distribution of problems in the sample space. Nevertheless it seems to be an equally reasonable way to select random C-L problems.

[9] only those for i > j were treated explicitly, since if values a and b are consistent (inconsistent) for v_i and v_j respectively, then values b and a will be consistent (inconsistent) for v_j and v_i respectively. Also, the consistency of v_i with itself is undefined.

IO \ CO	n	STD-CO Av. #Checks	STD-CO Av. #Arcs	STD-CO Av. #Solns	LAF-CO Av. #Checks	COMP-CO Av. #Checks
STD-IO	4	68	16	3	68	60
	6	263	33	0	237	207
	8	597	28	0	466	336
	10	1597	61	0		
	12					
LAF-IO	4	67	14		67	59
	6	164	11		157	126
	8	469	17		455	323
	10	953	18			
	12					
COMP1-IO	4	56	12		55	44
	6	123	8		118	70
	8	312	10		306	125
	10		11			152
	12		13			248
COMP2-IO	4	46	11		46	39
	6	89	6		84	59
	8	234	7		228	106
	10		6			102
	12		9			206

Table 6-2: Table of averaged results for the random C-L problems. (See caption of table 6-1 for meaning of upper left rectangle and line of X's)

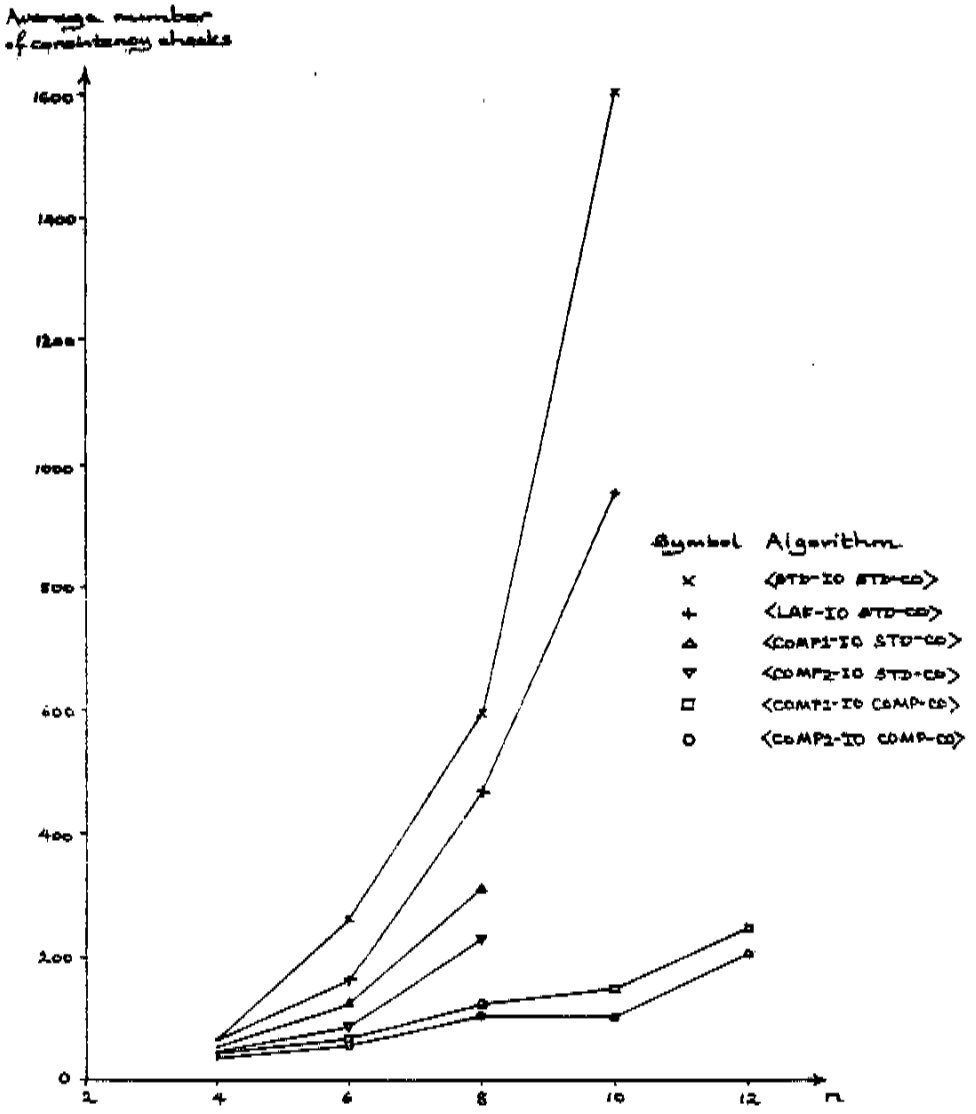

Figure 6-1: Plot of averaged results for some of the algorithms applied to the random C-L problems

such problem is defined explicitly in terms of a long list of 4-tuples, (v_i v_i-value v_j v_j-value), that give compatible instantiations for pairs of variables. As the number of variables, n, in the problem increases these lists become longer, on average for a random problem. Since a consistency check involves scanning the list of four-tuples for the current four-tuple being checked, such a check becomes more time consuming as n increases; a time cost that is irrelevant to the algorithm's time complexity. Thus for different n, the execution times of an algorithm are not directly comparable. Nevertheless, for a fixed n the different algorithms' times can be compared, but this only gives their relative performance for the specific average unit consistency check time that results for that value of n.

This leads us to ask: Even if for the experiments, the average consistency check could be made in time constant with n, what time would we choose? If the time is high then this biases in favor of algorithms that reduce the number of consistency checks. If the time is low, then this biases against such algorithms since the reduction in consistency checking that they achieve will be relatively less significant, and their extra time for evaluation of their more sophisticated heuristics will be incurred with less benefit.

We can consider two extremes:

1. If the time for a consistency check is infinite for all n, then the relative time complexity of the various algorithms will be the same as the ratios of the number of consistency checks for the algorithms.

2. If the time for a consistency check is zero for all n, then the (relative) time complexity of the various algorithms will be the same as (the ratios of) their non consistency-checking execution times.

Figure 6-2 shows these two *envelope* or *limiting-case* ratios of time for algorithm <STD-IO STD-CO> to time for algorithm <COMP2-IO COMP-CO>. Also plotted is the *actual* ratio for the execution times of these two algorithms. The latter, as mentioned, is *not* a meaningful time complexity ratio as n varies, because of the increasing unit consistency check time incurred as n grows. But it is interesting to see how it fits within the envelope. For larger n we expect it to assymptote towards the top (the infinite unit check time assumption) of the envelope as n, and hence average unit check time, grows.

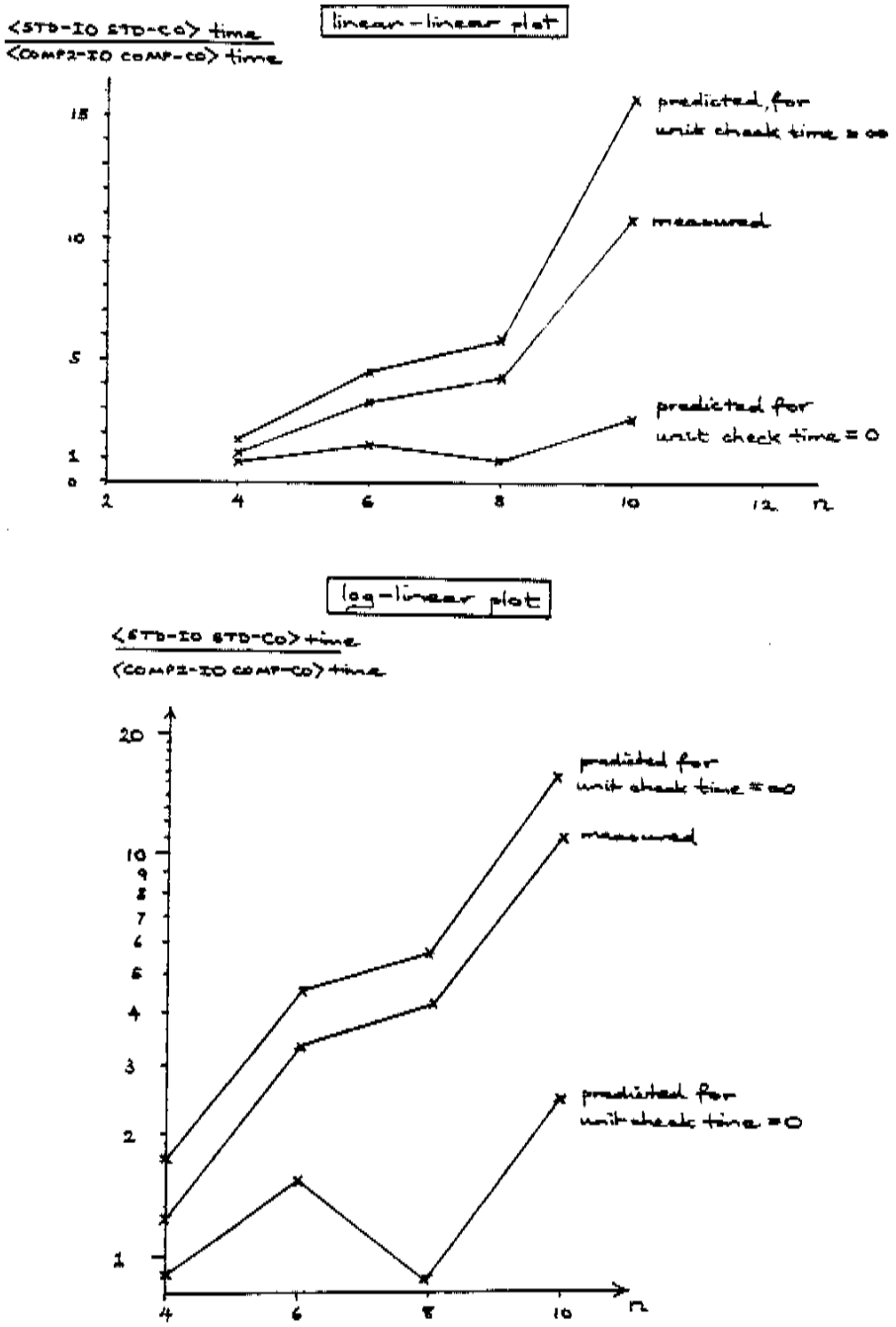

Figure 6-2: Ratio of averaged execution times for <STD-IO STD-CO> to <COMP2-IO COMP-CO> for the sets of random C-L problems.

7. DISCUSSION AND FUTURE WORK

Between completing the above experiments and writing this section, much further work was carried out on compatibility-based C-L algorithms. The following discussion will incorporate *some* of the insights from the latter where appropriate.

The following points deserve note:

1. The above experiments all show that use of compatibilities to guide search leads to a reduction in search cost, and algorithms that used compatibilities more extensively give more extensive savings. This is what matters when we can get the compatibilities at zero computational cost by analytic methods, which is often the case.

2. When obtaining compatibilities requires one or another of the preprocessing methods of chapter 3, hence incuring some computational expense, the preprocessing expense must be weighed against the savings resulting during search. One of the central goals for future work is to characterize empirically and/or analytically the savings resulting during seach when the compatibilities are used. This characterization would need to be as a function of the compatibilities of the problem and would indicate when the preprocessing cost to find compatibilities is justified by the later savings. Unfortunately, to use this characterization in a practical situation the compatibilities will need to be known before the savings due to having them can be predicted, in order to decide if they are worth computing (shades of Catch-22). However, if the prospective algorithm user can consider his problem as a random sample from a given distribution of C-L problems then he can get probabilistic advice from such a characterization, as to whether preprocessing is justified. Similarly, if he has a rough idea of the compatibilities, this may be sufficient for a meaningful determination of whether preprocessing is justified to get the (more) exact compatibilities.

3. Why was the improvement due to use of compatibilities in the n-queens problem experiments so minimal? Is it simply that we have here a class of problems for which compatibility-based algorithms don't gain you much for some reason? The answer is: Yes and No. The "Yes" is explained in the point below. The "No" arises because we, and Haralick [5], carried out our experiments for STD-IO and STD-CO order being (1,2,3...,n) where i is the variable giving the column position for the queen in row i and rows are successively numbered from the top of the board. By good (or bad) luck, this happens to be one of the better fixed orders for the n-queens problem. Therefore, the compatibility based algorithms had less room left for improvement over the <STD-IO STD-CO> algorithm results, than would otherwise have been available. If we are to model the algorithms' relative worth in an everyday context then we need to recognize that the user will not know which instantiation order or consistency check order to use. Our experiments should be averaged over a random selection of orders whenever an algorithm uses a STD ordering for instantiating and/or consistency checking (this is also true for the other algorithms actually, since they used the STD order (1,2,3,...,n) in the capacity of a default order when heuristics gave tied results, and the optimum order (1,2,...,n) favorably biased these algorithms too. But the bias in these cases was much less than when using STD-IO and/or STD-CO ordering, since the dynamic heuristic ordering worked to reduce it). Preliminary results for doing the n-queens experiments in this more

meaningful way, do show a greater relative improvement in search efficiency with increasing use of compatibilities.

Even the random C-L problem experiments should have been carried out with a randomized instantiation and consistency-checking order to ensure a more random sampling. However, in this case we expect no systematic biases to have resulted when this is not done, since even though the same fixed order was always used over all problems, each problem was randomly generated and hence the fixed order gave no systematic bias to an algorithm's performance.

4. The above reason for the small relative improvement on n-queens problems when using dynamic ordering heuristics was that there was simply an oversight in the experimental design. There is however one reason that n-queens problems as a class of problems are intrinsically less susceptible to improvement by compatibility-based methods than are the random problems investigated. This is because the range of compatibility values for an n-queens problem is smaller than that for the average random C-L problem generated. The closer the compatibility values are in a given problem, the less advantage compatibility based algorithms will have, since there is less information that they can base their differential judgements on. In the limiting case, when all compatibilities are the same, there can be no improvement at all from using any compatibility-based heuristic. (If however, compatibilities can be updated at each node to reflect the current accurate value which will vary due to reductions in the domains of the variables, then even if all compatibilities were initially equal it would not make the heuristics unuseable, since the compatibilities would in general diverge from equality at a node as the node's level increased; see point 7 below.)

   This leads to the aim, for future work to characterize the improvements when using compatibilities, as a function of the spread of compatibilities in a problem, and preferably as a function of the exact set of compatibilities themselves, and not just as a function of the mean compatibility value for a problem as Haralick has done [5]. He did not have as strong a reason for doing so as us, since he was not dealing with compatibility based algorithms where the effect is significant. We have recently developed a random C-L problem generator that takes a set of problem compatibilities as its input and gives a random problem having these exact compatibilities. This allows a much more incisive study to be carried out of the compatibility spread effect, than if only the mean is an input parameter and the compatibilities are random samples from a given distribution P(p) having that mean. Haralick uses such a P(p) with spread essentially zero; in section 6.2 we use a uniform P(p) from 0 to 1, so that the mean was fixed at 1/2. In future work, using the new random problem generator, we will be able to study the algorithms' performance not only as a function of the mean and spread of the compatibilities of a random problem, but also as a function of the now individually controllable compatibility values themselves.

   Preliminary results show us even for the non compatibility-based algorithms dealt with by Haralick, that the performance is not just a function of the mean compatibility (although Haralick's analysis only models it this way) but significantly varies with the spread of compatibilities in a problem. Therefore Haralick's analytic complexity results even for these algorithms can be quite misleading when any significant spread exists about the mean value. Of course, when there is negligeable such spread about the mean compatibility, as in Haralick's experiments, his analysis of the <STD-IO STD-CO> algorithm applies and we have found his analytic results to be very accurate.

5. Haralick's Full and Partial Lookahead extensions of his Forward Checking algorithm (used here) are versions of the oft discussed approach of interleaving what is sometimes called "reasoning by constraint analysis", "relaxation" or "deduction" (which correspond to the lookahead) with instantiation, "reasoning by case analysis" or "hypothesizing". Haralick [5] found that his Full or Partial Lookahead algorithms were not as effective as plain Forward Checking even though, as expected, they did reduce the size of the search tree measured in nodes. This is because algorithm complexity needs to measured in consistency checks, not in nodes generated, and the consistency checking at each node during Lookahead was just too expensive to pay for the domain values, and hence nodes, that it was able to eliminate. Haralick mentions the possibility that there may be problems for which the Lookahead methods have an advantage. We propose an even more interesting possibility: that there are nodes in a *given* problem's search tree where Lookahead is justified even if a blanket Lookahead approach is not justified for the problem's whole search tree. Furthermore, now that we have seen the value of using compatibilities to optimize search we can expect that there are also compatibility-based heuristics that will allow a dynamic decision to be made as to which nodes are appropriate for Lookahead, and how much Lookahead (full or partial) is justified there. A refinement of the compatibility notion may be useful here and is being considered – the values p_{ij}^{k} for the compatibility of value k of variable i over all values of variable j.

   The resulting algorithm would dynamically and intelligently fluctuate between an approach that interleaves Lookahead with instantiation plus Forward Checking, and an approach that forgoes the Lookahead option when it is predicted not to be worthwhile. Also, *within* a given Lookahead phase Haralick employs no inteligent strategy to provide a dynamic ordering capability. Again, compatibilities seem potentially useful in providing optimized ordering within a Lookahead phase when the algorithm does choose to do Lookahead at a node.

6. As mentioned in section 3.5.2, there is a duplication of consistency checking in the Forward Checking algorithm and in fact in all current C-L problem algorithms. There are no more than $\binom{n}{2}*n^2$, O(n↑4), *non-identical* consistency checks for a C-L problem of n variables where each has domain size n, yet the number of checks done (since duplication occurs) is an exponential function of n in general. This seems like a furtile area for algorithm improvement. If some kind of more global scheme could be used for keeping track of consistency checks, rather than having each node blind to the duplication that occurs in non-ancestor nodes, then tremendous savings could result. This seems particularly promising for using Haralick's lookahead extensions, because lookahead consistency-check duplication occurs even in ancestor nodes, and such duplication should be relatively easy to avoid.

7. We had said in an earlier point above that " In the limiting case, when all compatibilities are the same, there can be no improvement at all from using any compatibility-based heuristic. If however, compatibilities can be updated at each node to reflect the current accurate value which will vary due to reductions in the domains of the variables, then even if all compatibilities were initially equal it would not make the heuristics unuseable, since the compatibilities would in general diverge from equality at a node as the node's level increased." But dynamic updating of compatibilities would not only make compatibility-based heuristics more generally useable but also more accurate, since the information they use would be kept current at each node. Thus dynamic updating of compatibilities seems a useful developement to consider in future work to see if it can be done with cost small enough to pay for itself.

APPENDIX A. DETAILED RESULTS FOR THE EXPERIMENTS ON RANDOM C-L PROBLEMS

The following tables give the detailed results, on a problem-by-problem basis, from which the averages of table 6-2 are obtained. For each value n = 4, 6, 8, 10, 12 a set of random C-L problems was generated as described in section 6.2. The set corresponding to each n is used to exercise the up to 12 different versions of the basic Forward Checking algorithm formable by independently selecting one of the four IO-class heuristics and one of the three CO-class heuristics introduced in chapters 4 and 5.

Algorithms having the same IO-class heuristic result in the same search tree and hence the same number of arcs (#Arcs below) in the tree. The #Arcs column is thus given only for the STD-CO representative of a set of algorithms with common IO heuristic. The number of solutions is a property of the problem, not of an algorithm, and is given only once per table. Rows labeled "Av" give average performance for the corresponding algorithms on the set of ten random problems.

n = 4

	STD-CO			LAF-CO	COMP-CO
	Prob #	#Checks	#Arcs	#Solns	#Checks	#Checks
STD-ID	1	48	4	0	48	16
	2	28	4	0	28	16
	3	85	33	9	81	73
	4	58	12	0	58	58
	5	16	4	0	16	16
	6	96	18	0	100	100
	7	148	42	17	148	148
	8	69	15	4	66	50
	9	43	9	2	43	39
	10	88	21	4	88	88
	Av	68	16	3	68	60
LAF-ID	1	48	4		48	16
	2	28	4		28	16
	3	77	21		77	69
	4	58	9		58	58
	5	16	4		16	16
	6	97	15		101	102
	7	148	42		148	148
	8	66	13		66	50
	9	43	9		43	39
	10	84	19		88	84
	Av	67	14		67	59
COMP1-ID	1	48	4		48	16
	2	28	4		28	16
	3	77	21		77	69
	4	28	4		28	20
	5	16	4		16	16
	6	56	6		48	28
	7	119	37		119	111
	8	66	13		66	50
	9	43	9		43	39
	10	74	16		74	70
	Av	56	12		55	44
COMP2-ID	1	0	0		0	0
	2	0	0		0	0
	3	77	21		77	69
	4	28	4		28	20
	5	0	0		0	0
	6	56	6		48	28
	7	119	37		119	111
	8	66	13		66	50
	9	43	9		43	39
	10	74	16		74	70
	Av	46	11		46	39

n = 6

	STD-CO				LAF-CO	COMP-CO
	Prob #	#Checks	#Arcs	#Solns	#Checks	#Checks
STD-IO	1	101	9	0	101	95
	2	146	12	0	124	136
	3	120	6	0	120	48
	4	72	6	0	72	36
	5	848	130	0	748	710
	6	48	6	0	48	42
	7	143	9	0	123	99
	8	734	117	2	621	569
	9	96	6	0	96	36
	10	322	30	0	317	302
	Av	263	33	0	237	207
LAF-IO	1	101	8		100	99
	2	137	10		139	151
	3	120	6		120	48
	4	72	6		72	36
	5	346	22		314	280
	6	48	6		48	42
	7	133	8		123	99
	8	322	23		289	238
	9	96	6		96	36
	10	268	16		264	224
	Av	164	11		157	125
COMP1-IO	1	84	6		84	42
	2	157	9		156	96
	3	78	6		78	42
	4	72	6		72	36
	5	174	6		174	36
	6	48	6		48	42
	7	114	8		100	70
	8	322	23		289	238
	9	96	6		96	36
	10	84	6		84	60
	Av	123	8		118	70
COMP2-IO	1	84	6		84	42
	2	157	9		156	96
	3	78	6		78	42
	4	0	0		0	0
	5	0	0		0	0
	6	48	6		48	42
	7	114	8		100	70
	8	322	23		289	238
	9	0	0		0	0
	10	84	6		84	60
	Av	89	6		84	59

n = 8

		STD-CO			LAF-CO	COMP-CO
	Prob #	#Checks	#Arcs	#Solns	#Checks	#Checks
STD-ID	1	772	24	0	471	224
	2	786	45	0	661	691
	3	839	43	0	668	583
	4	479	16	0	460	583
	5	609	26	0	459	295
	6	475	12	0	459	178
	7	1579	87	0	1053	852
	8	192	8	0	192	64
	9	72	8	0	72	72
	10	165	9	0	149	114
	Av	597	28	0	466	336
LAF-ID	1	467	11		468	220
	2	763	30		780	669
	3	506	16		464	376
	4	416	11		384	222
	5	438	12		408	237
	6	474	11		474	188
	7	1184	52		1156	1067
	8	192	8		192	64
	9	72	8		72	72
	10	165	9		149	114
	Av	469	17		455	323
COMP1-ID	1	424	8		424	64
	2	282	9		273	115
	3	578	17		582	115
	4	160	8		160	64
	5	438	10		402	162
	6	474	11		474	188
	7	343	9		329	121
	8	192	8		192	64
	9	72	8		72	72
	10	155	9		149	115
	Av	312	10		306	125
COMP2-ID	1	0	0		0	0
	2	282	9		273	115
	3	578	17		582	281
	4	0	0		0	0
	5	438	10		402	162
	6	474	11		474	188
	7	343	9		329	121
	8	0	0		0	0
	9	72	8		72	72
	10	165	9		149	115
	Av	234	7		228	105

n = 10

	STD-CO			LAF-CO	COMP-CO
	Prob #	#Checks	#Arcs	#Solns	#Checks	#Checks
STD-IO	1	889	50	0		
	2	868	20	0		
	3	1257	66	0		
	4	841	32	0		
	5	2259	99	0		
	6	1344	58	0		
	7	2830	120	0		
	8	3089	114	0		
	9	800	10	0		
	10	1794	37	0		
	Av	1597	61	0		
LAF-IO	1	847	21			
	2	743	15			
	3	708	15			
	4	837	16			
	5	1274	22			
	6	1088	22			
	7	1282	24			
	8	1104	22			
	9	800	10			
	10	849	14			
	Av	953	18			
COMP1-IO	1		10			100
	2		10			100
	3		11			197
	4		11			185
	5		10			100
	6		11			190
	7		12			268
	8		11			181
	9		10			100
	10		10			100
	Av		11			152
COMP2-IO	1		0			0
	2		0			0
	3		11			197
	4		11			185
	5		0			0
	6		11			190
	7		12			268
	8		11			181
	9		0			0
	10		0			0
	Av		6			102

n = 12

	STD-CO			LAF-CO	COMP-CO
	Prob #	#Checks	#Arcs #Solns	#Checks	#Checks
STD-ID	1		0		
	2		0		
	3		0		
	4		0		
	5		0		
	6		0		
	7		0		
	8		0		
	9		0		
	10		0		
	Av		0		
LAF-ID					
COMP1-ID	1	12			144
	2	12			144
	3	13			313
	4	12			144
	5	13			283
	6	12			168
	7	13			274
	8	15			543
	9	12			156
	10	14			311
	Av	13			248
COMP2-ID	1	0			0
	2	0			0
	3	13			313
	4	0			0
	5	13			283
	6	12			168
	7	13			274
	8	15			543
	9	12			156
	10	14			311
	Av	9			205

References

[1] Fikes, R. E.
REF-ARF: A system for solving problems stated as procedures.
Artificial Intelligence 1:27-120, 1970.

[2] Gaschnig, J.
Performance measurement and analysis of certain search algorithms.
PhD thesis, Dept. Computer Science, Carnegie-Mellon U., 1979.

[3] Golomb, S. W. and Baumert, L. D.
Backtrack programming.
J. Assoc. Computing Machinery 12:516-524, 1965.

[4] Haralick, R. M. and Shapiro, L. G.
The consistent labeling problem: Part I.
IEEE Trans. Pattern Analysis and Machine Intelligence PAMI-1(2):173-184, 1979.

[5] Haralick, R. M. and Elliot, G.L.
Increasing tree search efficiency for constraint satisfaction problems.
Artificial Intelligence 14:263-313, 1980.

[6] Hardy, G. H. and Littlewood, J. E.
Some problems of Diophantine approximation: the lattice-points of a right-angled triangle.
Proc. London Mathematical Society 20:15-36, 1920.

[7] Lauriere, J. L.
A language and a program for stating and solving combinatorial problems.
Artificial Intelligence 10:29-127, 1978.

[8] Mackworth, A. K.
Consistency in Networks of Relations.
Artificial Intelligence 8:99-118, 1977.

[9] Waltz, D. L.
Understanding line drawings of scenes with shadows.
In Winston, P. H. (editor), *The Psychology of Computer Vision*, pages 19-91. McGraw-Hill, New York, 1975.