Constraint satisfaction algorithms[1]

BERNARD A. NADEL[2]

Computer Science Department, Wayne State University, Detroit, MI 48202, U.S.A.

Received January 6, 1988

Revision accepted September 25, 1989

Constraint satisfaction problems are ubiquitous in artificial intelligence and many algorithms have been developed for their solution. This paper provides a unified survey of some of these, in terms of three classes: (*i*) tree search, (*ii*) arc consistency (AC), and (*iii*) hybrid tree search/arc consistency algorithms. It is shown that several important algorithms, when slightly rearranged, are of the latter hybrid form, but with arc consistency components that do not necessarily achieve full arc consistency at the tree nodes. Accordingly, we define several new *partial* AC procedures, AC1/5, AC1/4, AC1/3, and AC1/2, analogous to the well-known *full* AC algorithms which Mackworth has called AC1, AC2, and AC3. The fractional suffixes on our AC algorithms are roughly proportional to the degree of partial arc consistency they achieve. Unlike traditional versions, our AC algorithms (full and partial) are presented in a parameterized form to allow them to be embedded efficiently at the nodes of a tree search process. Algorithm complexities are compared empirically, using the *n*-queens problem and a new version called confused *n*-queens. Gaschnig's Backmarking (a tree search algorithm) and Haralick's Forward Checking (a hybrid algorithm) are found to be the most efficient. For the hybrid algorithms, we find that it pays to do little arc consistency processing at the nodes, incurring more nodes, but sufficiently reducing the work per node so as to obtain less work over the whole tree. The unified view taken here suggests several new algorithms. Preliminary results show one of these to be the best algorithm so far.

Key words: constraint satisfaction problem, network consistency algorithms, arc consistency algorithms, tree search algorithms, Backtracking, Backjumping, Backmarking, Forward Checking.

Les problèmes de satisfaction des contraintes sont omniprésents dans le domaine de l'intelligence artificielle et bon nombre d'algorithmes ont été élaborés afin de les résoudre. Cet article fait état d'une étude de ces algorithmes selon trois classes : les algorithmes de (*i*) recherche arborescente, de (*ii*) consistance d'arc (AC) et de (*iii*) recherche hybride arborescente - consistance d'arc. Il est démontré que plusieurs algorithmes importants, lorsque légèrement modifiés, font partie de la dernière classe dite hybride, avec cependant des composantes de consistance d'arc qui ne permettent pas nécessairement d'obtenir une consistance d'arc complète aux nœuds d'arbre. Par conséquent, nous avons élaboré plusieurs nouvelles procédures AC *partielles*, AC1/5, AC1/4, AC1/3 et AC 1/2, qui sont analogues aux algorithmes bien connus *complets* AC que Mackworth a appelé AC1, AC2 et AC3. Les suffixes fractionnaux de nos algorithmes AC sont plus ou moins proportionnels au degré de consistance d'arc partielle qu'ils obtiennent. Contrairement aux versions traditionnelles, nos algorithmes AC (complets et partiels) sont présentés dans une forme paramétrée afin de leur permettre d'être emboîtés efficacement aux nœuds dans un processus de recherche arborescente. La complexité des algorithmes est comparée de façon empirique, à l'aide du problème *n*-reines et d'une nouvelle version, dite *n*-reines confuses. Le marquage arrière de Gaschnig (un algorithme de recherche arborescente) et la vérification avant de Haralick (un algorithme hybride) se sont révélés les plus efficaces. Dans le cas des algorithmes hybrides, nous avons constaté qu'il était profitable de peu traiter la consistance d'arc aux nœuds, car, bien que cela entraîne davantage de nœuds, le travail par nœud s'en trouve réduit, au point de donner moins de travail pour l'arbre entier. Le point de vue unifié adopté ici suggère plusieurs nouveaux algorithmes. Les résultats préliminaires permettent de classer l'un d'entre eux comme le meilleur algorithme jusqu'à présent.

Mots clés : problème de satisfaction des contraintes, algorithmes de consistance de réseau, algorithmes de consistance d'arc, algorithmes de recherche arborescente, retour-arrière, saut arrière, marquage arrière, vérification avant.

[Traduit par la revue]

Comput. Intell. 5, 188-224 (1989)

1. Introduction

The Constraint Satisfaction Problem (CSP) is ubiquitous in artificial intelligence.[3] It has received intense study from many researchers, including Fikes (1970), Waltz (1975), Gaschnig (1974, 1977, 1978, 1979), Rosenfeld *et al.* (1976), Montanari (1974), Mackworth (1977*a*), McGregor (1979), Haralick and Elliot (1980), Haralick and Shapiro (1979, 1980), Haralick *et al.* (1978), Purdom (1982, 1983), Freuder (1978, 1982), Nadel (1986, 1988*a*, *b*), Nudel (1982, 1983*a*, *b*), Mohr and Henderson (1986), Dechter and Pearl (1988), and Dechter and Dechter (1987). Section 2 introduces the CSP problem class. The *n*-queens problem and a new variant, confused *n*-queens, are presented there and provide a convenient test-bed for the example traces and empirical comparisons that follow.

The importance of CSP is due to the wide range of practical problems it can be used to model. Applications of the standard form of the problem or a close relative have included such diverse areas as theorem proving (Purdom and Brown 1981; Van Hentenryck and Dincbas 1986), belief maintenance (Dechter 1987*a*; DeKleer 1986; Doyle 1979), graph problems (Fowler *et al.* 1983; McGregor 1979; Ullman

[1]A preliminary version appeared as "Tree search and arc consistency in constraint satisfaction algorithms" in *Search in Artificial Intelligence*, edited by L. Kanal and V. Kumar, Springer-Verlag, New York, 1988, pp. 287-342. Portions of this article that are the same or similar are reprinted by permission of Springer-Verlag, New York.

[2]Previously Nudel.

[3]We will use CSP to refer to the problem class, and csp to refer to an individual problem instance. The word *problem* will be used both for the class and an instance. The meaning should be clear from context.

Printed in Canada / Imprimé au Canada

1976), machine vision (Davis and Rosenfeld 1981; Mackworth 1977*b*; Waltz 1975), event scheduling (Rit 1986; Tsang 1987), floor-plan design (Eastman 1972), and planning genetic experiments (Stefik 1981).

As might be expected, many algorithms have been developed for solving constraint satisfaction problems. This paper provides a unified comparison for some of these, in terms of three classes: (*i*) *tree search*, (*ii*) *arc consistency*, and (*iii*) *hybrid tree search/arc consistency* algorithms. Section 3 presents three tree search algorithms: the classic Backtracking algorithm and two algorithms by Gaschnig, Backjumping and Backmarking. Backjumping is closely related to dependency-directed backtracking in truth-maintenance systems (Doyle 1979) and intelligent backtracking in Prolog (Bruynooghe and Pereira 1984; Kumar and Lin 1988). Backmarking is important in being one of the most efficient algorithms considered here.

Section 4 treats the class of arc consistency (AC) algorithms (Mackworth 1977*a*; Mohr and Henderson 1986; Waltz 1975). These are simplification algorithms which convert the initial problem into a simpler version with the same solutions. Path consistency algorithms (Mackworth 1977*a*; Mohr and Henderson 1986; Montanari 1974) are analogous in that they are also a form of simplification algorithm. Freuder (1978) has generalized these simplification approaches with the concept of *j*-consistency. In his terms, arc consistency corresponds to 2-consistency and path consistency to 3-consistency. We do not consider here *j*-consistency algorithms for $j \geq 3$ because past experiments (McGregor 1979) show them to be not cost-effective in general. In fact we go the other way. Section 4.1 introduces a set of *partial* arc consistency algorithms, which could be said to achieve *j*-consistency for $1 < j < 2$. These we call AC1/5, AC1/4, AC1/3, and AC1/2, where the fractional suffixes are more or less proportional to the degree of arc consistency they attain. They are reduced analogs of the classic *full* ($j = 2$) arc consistency algorithms which Mackworth (1977*a*) has called AC1, AC2 and AC3. The latter are treated in Sect. 4.2.

Section 5 considers 13 hybrid algorithms that embed arc consistency processing at each node of a tree search. This section builds a bridge between the "network consistency school of thought" and the "tree search school." The tree search/arc consistency hybrid algorithms are considered along a spectrum according to the degree of arc consistency they achieve at their search tree nodes. Several new hybrid algorithms are presented, which arise naturally to fill gaps in this spectrum. Also, several important known algorithms — Haralick's Full and Partial Lookahead and even the classic Backtracking algorithm — are seen as having this hybrid form when certain rearrangements are made in the nesting of their loops. However, the embedded arc consistency components are not necessarily full arc consistency algorithms. Rather, they may be the above-mentioned partial AC algorithms, AC1/5, AC1/4, AC1/3, and AC1/2, of Sect. 4.1. The hybrid algorithms of Sect. 5 are treated in two groups: nine that guarantee full arc consistency[4] at each search tree node (Sect. 5.1) and four that guarantee successively lesser degrees of partial arc consistency at the nodes (Sect. 5.2).

[4]This does not necessarily mean that they use only full arc consistency procedures at the nodes, as will be seen in Sect. 5.

In anticipation of their embedding into a tree search shell, each of our AC algorithms is parameterized to allow arc consistency processing to be carried out on appropriate, adjustable subgraphs of the constraint network local to the corresponding tree node. Figure 8 gives a unified single algorithm that subsumes all our 13 hybrids. Table 1 gives a summary of the schematic structures of these hybrids. As seen there, the structures are all of the form TS + AC_1 or TS + AC_1 + AC_2, where AC_1 and AC_2 are one of the full or partial AC algorithms of Sect. 4, and TS stands for the tree search shell in which they are embedded. Rationalized new names are suggested for these algorithms which reflect their structural relationship.

Section 6 presents empirical complexity results for the tree search algorithms of Sect. 3 and the hybrid algorithms of Sect. 5, together with general conclusions. We find that the Backmarking tree search algorithm and the hybrid algorithm Forward Checking are the (essentially equal) best two of the algorithms studied. We also find that of the hybrid algorithms, partial AC hybrids are more efficient than full AC hybrids. In fact, of the partial AC hybrids, the best one, Forward Checking, does the second least amount of arc consistency processing at the tree nodes. The exact results are given in Tables 2 and 3 and are summarized in the schematic plot of Fig. 13. We see from these results that even though less arc consistency at the tree nodes results in more nodes per tree, reduction of the work at each node can more than make up for the extra nodes incurred and can lead to less work over the whole tree.

There is a break-even point, however, along this spectrum of degree of arc consistency attained at the nodes, where the optimum efficiency is achieved. For the hybrid algorithms studied here, this is at Forward Checking. However, the spectrum view suggests considering algorithms on either side of Forward Checking on the spectrum for a possibly better hybrid (see Fig. 13). This has led us to a new algorithm which preliminary experiments do in fact show to be the better than all other algorithms considered here. It is discussed briefly in Sect. 6 and will be reported more fully in a later paper.

Appendix I further clarifies the algorithms by presenting more-detailed explanations and traces for them. Appendix II discusses aspects of the Pascal-like programming language used to present the algorithms. This paper is an extension of an earlier version (Nadel 1988*a*). Several algorithms have been added and the presentation has been significantly restructured to make the main concepts more transparent. The earlier version, however, may be useful in providing a different slant on this material and for some additional detailed examples. Another useful review article in the same spirit is Mackworth (1987).

2. Constraint satisfaction problems

Constraint satisfaction problems have three components: *variables*, *values*, and *constraints*. The goal is to find assignments of the variables to their candidate values such that all the constraints are satisfied. We will here be concerned with the version of the problem where *all* such assignments are sought. The well-known *n*-queens puzzle is often used to exemplify constraint satisfaction problems. This is the problem of finding all ways to place *n* queens on an $n \times n$ chess board so that no two queens attack each other. To avoid notational conflict, we will henceforth refer to it as

q-queens rather than n-queens, reserving n for the number of variables in a problem.

It should be kept in mind that q-queens itself is not a constraint satisfaction problem, but rather can be naturally formulated as one. There are in fact quite a few natural alternative formulations of q-queens as a constraint satisfaction problem, as discussed in Nadel (1988*b*). In general, these have different numbers of variables, and different values and constraints.

The standard constraint satisfaction formulation of q-queens is to associate a variable z_i with each of the $1 \leq i \leq q$ rows of the board. Since there must be exactly one queen per row if q queens are to be placed on the board with no two attacking, then we need only know what column the queen is in for each row. Thus each variable z_i can have a domain of candidate values, $d_{z_i} = \{1\ 2\ \ldots\ q\}$, whose members denote the corresponding board column, with assignment $z_i = j$ meaning that the queen in row i is in column j. The condition that no two queens attack each other can be considered as $c = \binom{q}{2}$ binary constraints, expressible analytically as $(z_i \neq z_j) \wedge (|i - j| \neq |z_i - z_j|)$, for $1 \leq i < j \leq q$, since this ensures that no two queens are in the same column or diagonal of the board. (That no two queens are in the same row is already taken care of by allowing only one value per variable.) In terms of the Pascal-like programming language we will be using throughout (see Appendix II), these constraints can be expressed by the following Boolean function for testing whether value zi for z_i and value zj for z_j are compatible.

```
FUNCTION check (i, zi, j, zj): Boolean;
check ← (zi ≠ zj) and (abs(i − j) ≠ abs(zi − zj));
END;
```

We will make use of 4-queens (and in Appendix I, also of 5-queens) under the above formulation to exemplify the working of the hybrid algorithms of Sect. 5. For the tree search algorithms of Sect. 3, however, a more convenient running example will be *confused* 4-queens. Confused (or inverse) q-queens is a new variant of q-queens for which one seeks all ways to place q queens on a $q \times q$ chess board, one queen per row, so that each pair of queens *does* attack each other. A constraint satisfaction formulation is obtained as for regular q-queens, but with the constraints now being $(z_i = z_j) \vee (|i - j| = |z_i - z_j|)$, for $1 \leq i < j \leq q$. These are programmable as for *check* above with the obvious changes.

Confused q-queens is not really much of a puzzle, since it is easy to discover the pattern in generating the set of solutions for any q. There are $q + 2$ solutions, one for each board column and one for each of the two principal diagonals. Each arrangement of the q queens along one of these $q + 2$ straight lines generates one of the solutions. The only exception is for $q = 3$, in which case there are nine solutions (four of which do not correspond to the above-mentioned pattern):

(1 1 1) (1 2 1) (1 2 3) (2 1 2) (2 2 2) (2 3 2) (3 2 1) (3 2 3) (3 3 3)

Although not particularly challenging for people, confused q-queen nevertheless provides a convenient, nontrivial test-bed for the constraint satisfaction algorithms below, since these cannot take advantage of the pattern in the solutions as can humans. In fact, confused q-queens perhaps provides a more appropriate test-bed than does the oft-used q-queens problem, in the sense that the number of solutions for confused q-queens and for q-queens are respectively linear and (apparently) exponential in q (see Tables 2 and 3 of Sect. 6). The former is more representative of realistic problems, since it is unlikely that real problems would swamp one with solutions as does q-queens.

Related to this, we find that the constraints of q-queens grow increasingly loose with q, whereas those of confused q-queens grow tighter. Looser constraints mean more solutions, and also larger search trees and more complex searches. The usefulness of tightness parameters in obtaining precise complexity expressions, and in designing effective problem-solving heuristics based on those expressions, has been seen in Nadel (1986, 1988*b*) and Nudel (1983*a*). Specifically, the *constraint looseness* or the *constraint satisfiability ratio*, $R(z_i\ z_j)$, for a constraint $C(z_i\ z_j)$ was defined there as the fraction of the tuples that actually satisfy the constraint out of all the candidate tuples in the corresponding cartesian product $d_{z_i} \times d_{z_j}$. For q-queens and confused q-queens it can be shown that the satisfiability ratios are given respectively by

$$R(z_i\ z_j) = (q^2 - 3q + 2|i - j|)/q^2$$

and its complement

$$R'(z_i\ z_j) = 1 - R(z_i\ z_j) = (3q - 2|i - j|)/q^2.$$

It should be kept in mind that actually neither of the above two families of queens problems is particularly representative of constraint satisfaction problems in general — which is not surprising, since each family is generated by only a single parameter q. First, both kinds of queens problems (under the above formulations) have variables all with the *same* domain. Second, the instances are what might be called *complete binary* constraint satisfaction problems in that (*i*) each of their constraints involves only two variables (hence *binary*) and (*ii*) *all* pairs of problem variables have such a binary constraint on them (hence *complete*).

The algorithms below assume complete binary instances because this considerably simplifies the presentation. Also, for the same reason, the algorithms all instantiate variables and (or) check constraints in the natural order. However, in most cases these order restrictions can be readily overcome by a slight extension of the algorithms or simply by a reindexing of the variables and constraints to correspond to the new orders desired. On the other hand, it is not clear how one could, or whether one should, generalize algorithms Backjump and Backmark to allow constraints to be checked in any but the natural order. The instantiation order though could quite conceivably be changed for any of the algorithms.

A more general treatment of constraint satisfaction problems and some of their algorithms can be found in Nadel (1986). The algorithms there are written so as to allow arbitrary instantiation order and constraint-check order. The problems are allowed to have variables z_i each with an arbitrary domain, of an arbitrary number, m_{z_i}, of values of arbitrary type. And instances are allowed to have an arbitrary number of constraints C_j, each over an arbitrary subset Z_j of the n problem variables z_i, with possibly zero, one, or even more than one constraint on a given subset of

variables. All our algorithms below assume complete binary instances on n variables, each with integer domain {1 2 ... m_{z_i}}, although the domains are not necessarily of the same size. Our running examples (formulating q-queens and confused q-queens) are all of this form, but with all domains of the same size, equal to the number of variables, so that $n = q = m_{z_i}$ for each z_i.

3. Tree search algorithms

This section treats three tree search algorithms for solving constraint satisfaction problems: the traditional Backtracking algorithm and two algorithms due to Gaschnig, Backjumping and Backmarking. The action of these algorithms is shown graphically in Figs. 1 and 2 (and in more detail in Appendix I), using confused 4-queens as an example.[5] We will see that there are certain inefficiencies in the standard Backtracking approach. These have been noted by previous researchers (Gaschnig 1974; Haralick and Elliot 1980; Mackworth 1977*a*) and are often grouped under the heading of *thrashing behavior*. We will see that Backjumping and Backmarking are able to avoid some of these inefficiencies. Backjumping can be seen as achieving what we call "horizontal" savings compared to Backtracking, while Backmarking's savings are "vertical." Note that although Backtracking is the prototypical tree search algorithm, we will see later (Sect. 5.2.4) that a modified version involving loop renesting can be viewed as belonging to the class of hybrid tree search/arc consistency algorithms dealt with in Sect. 5.

3.1. Backtracking

Apart from generating all $\Pi_{i=1}^{n} m_{z_i}$ possible n-tuples and checking each against the constraints, the most straightforward approach to solving constraint satisfaction problems is via the traditional Backtracking algorithm (Bitner and Reingold 1975; Golomb and Baumert 1965; Walker 1960), a version of which is given below. This generates a tree of all instantiations (assignments) of values to variables, checking each instantiation against all earlier ones along the corresponding branch of the tree. Only if no incompatibility occurs between the current instantiation and a past one does the branch get extended by instantiating the next variable to each of the values in its candidate domain. This of course has a potentially great advantage over the brute-force generation and tesing of all possible n-tuples in that large subsets of inconsistent n-tuples may be avoided each time a branch of the search tree is pruned. The algorithm may be implemented as follows.

```
PROCEDURE BT(k, VAR z);
FOR z[k] ← 1 TO m[k] DO
  BEGIN
  Consistent ← True;
  FOR p ← 1 TO k−1 WHILE consistent DO
    consistent ← check(p, z[p], k, z[k]);
  IF consistent THEN
    IF k = n THEN output(z) ELSE BT(k+1, z)
  END
END;
```

The initial call to BT(k, z) has k = 1, with the value of z being arbitrary. Parameter z is an array of integer components z[1] to z[n] for storing instantiations respectively for z_1 to z_n. Note that z is a *reference* parameter of BT, since it is preceded by VAR. This is only in order to save space and is not a logical necessity. The domains d_{z_k} = {1 2 ... m_{z_k}} for variables z_k are represented by array m whose components m[k] = m_{z_k} store the upper-bounds for the corresponding domains, $1 \le k \le n$.[6] This array is available to BT as a global variable.

A trace of the $z_1 = 2$ subtree for BT solving the confused 4-queens problem is shown in Fig. 1. (This figure also applies to the Backjumping algorithm discussed below. The greyed-out parts and the "Backjump!" arrows are for that algorithm and should be ignored for the time being.) The rectangles into which the tree is partitioned in the figure denote nodes. These are labeled A to I in the order of their generation by BT. (There are of course nodes generated between A and B, and after I, that are not shown.) Note that, as for any recursive process, there is more than one way to partition a search tree, such as that in Fig. 1, into nodes. We define a node to be the processing done within the corresponding call of BT, exclusive of any recursive subcalls. The number of nodes is the number of calls of BT. (Similar definitions apply for the other algorithms below.)

Given the above form of BT, at each node we first instantiate the corresponding variable, then check for consistency. We might have rewritten BT, and correspondingly repartitioned the tree, so that a node involved first checking consistency (of the instantiation made at the parent node) and then instantiating the next variable.[7] This is in fact how the version of Backtracking in Nadel (1986) works. Section 5.2.4 below introduces yet a third variation. It retains the node structure used here, but changes the order of processing at a node. (Analogous differences exist between the alternative versions of the Forward Checking, Partial Lookahead, and Full Lookahead algorithms below and in Haralick and Elliot (1980) and Nadel (1986).

For discussion purposes, the following terminology will be useful. The node corresponding to a call BT(k, z) we call a level-k node; the root node being at level $k = 1$. At such a node, variable z_k is called the *current* variable. It is the variable that is instantiated at the node. Variables z_1 to z_{k-1} have already been instantiated, and these we call the *past* variables, while variables z_{k+1} to z_n are yet to be instantiated and are called the *future* variables. To include the current and future variables together, we sometimes use the term *nonpast* variables. These naming conventions will apply also to all subsequent algorithms below.

At the left of Fig. 1 (and Fig. 2) are shown the search tree level numbers $1 \le k \le n$ and the current variables z_k that are being instantiated at the corresponding level's nodes. (The numbers at the ends of the arcs within the nodes are

[5] We use the confused version of the problem because the difference between Backtracking and Backjumping already shows up at confused 4-queens, but does not show up using regular q-queens till $q = 6$. See Table 3.

[6] Note that we use k to denote the algorithm variable and k to denote its generic value. Extending this, we use m or m[k] to denote the algorithm array variable, m[k] to denote the variable that is the kth component of m[k], and $m[k]$ to denote the generic value of the component variable m[k]. Similar notation will be used throughout to denote simple and compound algorithm variables and their values.

[7] In other words, using I for instantiation and C for constraint checking, the sequence $ICICIC$ ... along a branch of a search tree could be partitioned either as $|IC|IC|IC|$ ... or as $I|CI|CI|C$ ...

of course the values to which the corresponding variable is instantiated). Also shown on successive rows for a given level are the constraints $C(z_p, z_k)$ for $1 \leq p \leq k-1$, in the order in which they are checked at that level's nodes. Within nodes of the figure, we use crosses and check marks respectively to denote whether the corresponding constraint (the one on that row of the figure) was violated or satisfied when checked.

Checking of a given instantiation against past instantiations of course stops when the first violation is found, and the next instantiation is then tried. Thus, for example, the constraint checks at node E in Fig. 1 are performed in the order given by their labels *a* to *f* which we have added. Recursion takes place at each point that all applicable checks have succeeded for a given instantiation. Thus, for example, node F is generated after check *c* at node E. Node F is then completed before returning to instantiation $z_3 = 3$ and check *d* at node E. As we will see (in connection with Fig. 12), our modified BT of Sect. 5.2.4 orders checks and interleaves recursion quite differently at a node, although the same checks and nodes occur, and nodes are generated in the same order.

3.2 Backjumping

There are certain inefficiencies in straight Backtracking. For example, look at node H of the BT trace in Fig. 1. Each instantiation of z_4 performed there fails against a past instantiation no deeper than for z_2. But BT returned to level $k = 3$ from node H and tried different values for z_3 in node G. There is no point in doing this since each possible z_4 value has just been found incompatible with instantiations at even shallower levels. Changing the z_3 values, and not those for z_1 or z_2, will only allow the same incompatibilities to reoccur when the z_4 instantiations are performed again, as is in fact the case at node I.

To avoid this inefficiency, Backjumping backs up possibly more than one level. When backing up from a node where all values of a variable were found to be incompatible with some past instantiation along that branch, Backjumping backs up all the way to the level of the deepest such past incompatible instantiation. Of course, the bigger the number of levels backjumped over, the greater the savings. For node H, the deepest past variable whose instantiation is incompatible with a z_4 value is z_2, not z_3. We therefore may avoid irrelevant reinstantiation of z_3 and jump back from node H (fleetingly through node G) to node B to try the next instantiation there, $z_2 = 4$. This is shown by the rightmost arrow labeled "Backjump!" in the figure. A similar backjump is shown by the leftmost "Backjump!" arrow.

As shown by the greyed-out regions of the BT search tree, the rightmost backjump avoids two instantiations and three constraint checks in node G and avoids generating node I completely, saving four instantiations and five constraint checks there. The leftmost backjump does not avoid any node generation (since both remaining z_3 instantiations at node C fail anyway, preventing any more subnodes of C), but avoids two instantiations and three constraint checks at node C. These savings are reflected in the counts for BJ in the table at the right of Fig. 1. For the section of the tree shown in the trace, BJ beats BT with 32 versus 43 checks and 8 versus 9 nodes. For the whole problem, due to several more backjumps, BJ beats BT with a total of 139 versus 160 checks, and 27 versus 29 nodes. Further comparisons occur in Tables 2 and 3 of Sect. 6.

This *multi-level* or *nonchronological* backtracking is implemented by the BJ algorithm below, which is Gaschnig's Backjump (Gaschnig 1978, 1979) modified to find all solutions. (Another variant appears at the end of this section.) Backjump is closely related to dependency-directed Backtracking in truth maintenance systems (Doyle 1979) and intelligent backtracking in Prolog (Bruynooghe and Pereira 1984; Kumar and Lin 1988).

```
1  FUNCTION BJ(k, VAR z) : integer;
2  returndepth ← 0;
3  FOR z[k] ← 1 TO m[k] DO
4    BEGIN
5    Consistent ← True;
6    For p ← 1 TO k−1 WHILE consistent DO
7      consistent ← check(p, z[p], k, z[k]);
8    IF not consistent THEN faildepth ← p−1;
9          {gives p value of last completed FOR cycle.}
10   IF consistent THEN
11     IF k = n THEN BEGIN
12                      output(z);
13                      faildepth ← n−1
14                      END
15               ELSE BEGIN
16                      faildepth ← BJ(k+1, z);
17                      IF faildepth < k THEN
18                                    Return(faildepth)
19                      END;
20   returndepth ← max(returndepth, faildepth)
21   END;
22 Return(returndepth)
23 END;
```

Parameters k and z of BJ are as for BT, with the same initial values. Again z is a reference parameter in order to save space, but not because of logical necessity. Figure A1 of appendix I gives a more detailed view of the processing in Fig. 1 in terms of the returndepth and faildepth variables.

BJ can be rewritten as BJ2 below to make clearer its correspondence with algorithm BM of the next section. Note that BJ above sets faildepth to $p-1$ at line 8 only when an inconsistency has been found.[8] But BJ2 makes this assignment, to MaxCheckLevel, whether an inconsistency is found or not. Thus in the case that an inconsistency is not found, the counterpart variables faildepth and MaxCheckLevel apparently get different values in the two versions of BJ. However, a little thought shows that this is not the case[9] and the two algorithms are functionally equivalent.

[8] See Appendix II regarding why $p - 1$ and not p is used here and also in algorithm BM below.

[9] In BJ when an inconsistency is not found, there are two possibilities: (*i*) If $k = n$ then faildepth gets the value $n - 1$. But in this case, this is the same value as already given to MaxCheckLevel at line 8 in BJ2 (and thus the explicit assignment to $n - 1$ at line 13 of BJ is avoided in BJ2). (*ii*) If $k \neq n$ then faildepth gets the value returned by the recursion at line 16 of BJ. This will generally not be the value of MaxCheckLevel from line 8 of BJ2, but on return from recursion by BJ2 at line 16, MaxCheckLevel will in any case be overwritten and set correctly to the same value as faildepth gets.

The faildepth variable is renamed MaxCheckLevel in the new version of BJ because for a given instantiation it is set to the index p (or level) of the deepest past variable z_p whose value is tested against the instantiation. It thus necessarily corresponds to an actual "fail depth" only when its value is less than $k - 1$. If it is equal to $k - 1$, then a fail (inconsistency) may or may not have occurred. Note that BJ2 has been written as a procedure, not a function, in order to make closer the connection with BM of Sect. 3.3. Further discussion of BJ/BJ2 in relation to BM appears in that section.

```
1  PROCEDURE BJ2(k, VAR z, VAR MaxCheckLevel);
2  returndepth ← 0;
3  FOR z[k] ← 1 TO m[k] DO
4    BEGIN
5    Consistent ← True;
6    FOR p ← 1 TO k-1 WHILE consistent DO
7      consistent ← check(p, z[p], k, z[k]);
8    MaxCheckLevel ← p-1;
9          {gives p value of last completed FOR cycle.}
10   IF consistent THEN
11     IF k = n THEN output(z)
12               ELSE BEGIN
13                    BJ2(k+1, z, MaxCheckLevel);
14                    IF MaxCheckLevel < k
                                        THEN Return
15                    END;
16   returndepth ← max(returndepth, MaxCheckLevel)
17   END;
18 MaxCheckLevel ← returndepth
19 END;
```

3.3. Backmarking

There would still seem to be room for improvement in the Backjumping approach, since many constraint checks are still repeated. Any such repetitions are of course wasteful, but the trick is to avoid them without incurring an inordinate penalty in terms of space required. Note also that the naive approach of simply keeping a large table storing the results of past checks requires too much memory in general. Moreover, it only replaces function calls by faster table look-ups. Wasteful repetitions of the same table look-ups still occur, thus reducing the average check time but not really avoiding check repetitions. Gaschnig's Backmark algorithm (Gaschnig 1977) manages to avoid a large number of repetitive checks in a way that avoids both these pitfalls.

Algorithm BM below is Gaschnig's Backmark modified to find all solutions. Another version of the algorithm and a useful discussion are given by Haralick and Elliot (1980). Figure 2 shows a trace of BM solving the same $z_1 = 2$ subproblem as used in Fig. 1 for BT and BJ. The BM algorithm avoids some of BT's constraint checks (but not its nodes or instantiations) in the following two ways:

(a) If, at the most recent node where a given instantiation was checked, the instantiation failed against some past instantiation that has not yet changed, then it will fail against it again. Therefore all constraint checks involving it may as well be avoided and the next instantiation tried.

(b) If, at the most recent node where a given instantiation was checked, the instantiation succeeded against all past instantiations that have not yet changed, then it will succeed against them again. Therefore we may as well check the instantiation only against the more recent past instantiations which have changed.

As before, the processing avoided compared to BT is indicated in Fig. 2 by greyed-out regions. Grey circles denote constraint check savings of type (a) above and grey squares denote savings of type (b). Note that type (a) savings always correspond to a column of zero or more circled check marks under a given instantiation, ending in a circled cross. The only example of this with more than zero circled check marks occurs under instantiation $z_4 = 2$ in node I of Fig. 2. Type (b) savings always correspond to a column of one or more squared check marks, followed by uncircled check marks or crosses. No examples of this with more than one squared check mark occur in the figure.

The savings by BM compared to BT are reflected in the counts in the table at the right of Fig. 2. In terms of checks, BM is significantly better than both BT and BJ for the segment of the trace shown and for the whole problem. More data appears in Tables 2 and 3, where BM will be seen to be one of the most efficient of all the algorithms studied, agreeing with the findings of Haralick and Elliot (1980).

```
PROCEDURE BM(k, VAR z, VAR MaxCheckLevel,
                    VAR MinBackupLevel);
FOR z[k] ← TO m[k] DO
  IF MaxCheckLevel[k, z[k]] ≥ MinBackupLevel[k]
  THEN
  BEGIN{Type (a) savings when this block is avoided.}
  Consistent ← True;
  FOR p ← MinBackupLevel[k] TO k-1
        {Type (b) savings if MinBackupLevel[k] > 1.}
    WHILE consistent DO
      consistent ← check(p, z[p], k, z[k]);
  MaxCheckLevel[k, z[k]] ← p-1;
      {gives p value of the last completed FOR cycle.}
  IF consistent THEN
    IF k = n THEN output(z)
             ELSE BM(k+1, z, MaxCheckLevel,
               MinBackupLevel)
  END;
MinBackupLevel[k] ← k-1;
FOR i ← k+1 TO n DO
  MinBackupLevel[i] ← min(MinBackupLevel[i], k-1)
END;
```

Parameters k and z of BM are as for BT and BJ, with array z again being a reference parameter only to save space but not because of logical necessity. However, arrays MaxCheckLevel and MinBackupLevel must be reference parameters. The former is an $n \times m$ array, where $m = \max_{i=1}^{n}\{m_z\}$, and the latter is a $1 \times n$ array like z. As for BT and BJ, the initial call to BM inputs k = 1, with the initial value of array z being irrelevant. Arrays MaxCheckLevel and MinBackupLevel start with all elements initialized to 1. Note that since all array formal parameters are reference parameters, the memory requirements are quite manageable in general.

MinBackupLevel[k] in BM stores the minimum level to which backup has occurred since the last level-k node was completed. MaxCheckLevel in BM is a generalization of that variable in algorithm BJ2 above. It is used to remember the

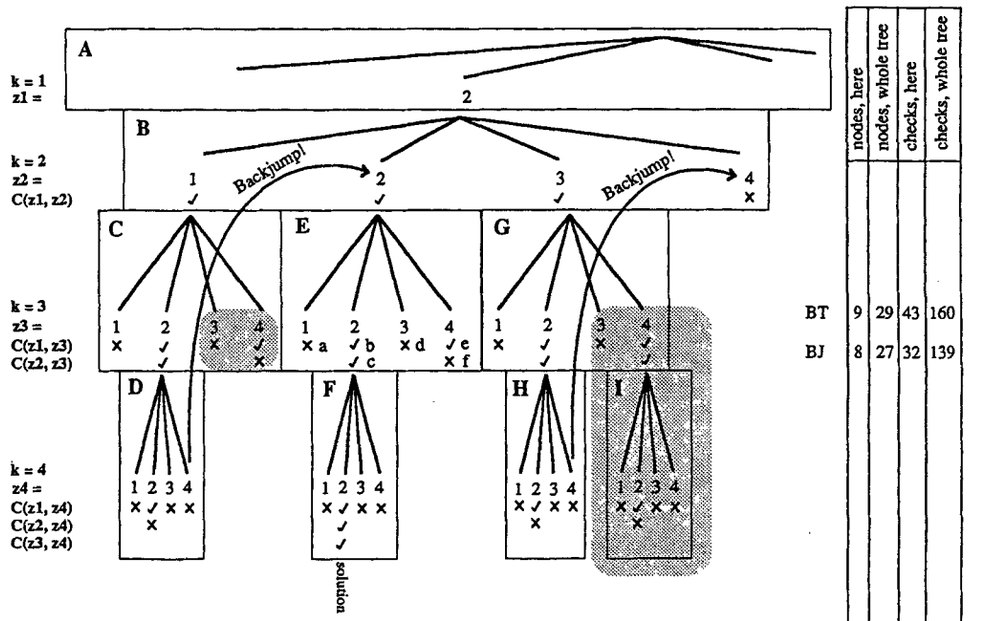


FIG. 1. The $z_1 = 2$ subtree when solving confused 4-queens by BT and BJ. (Greyed-out instantiations, constraint checks, and nodes are avoided by BJ.)

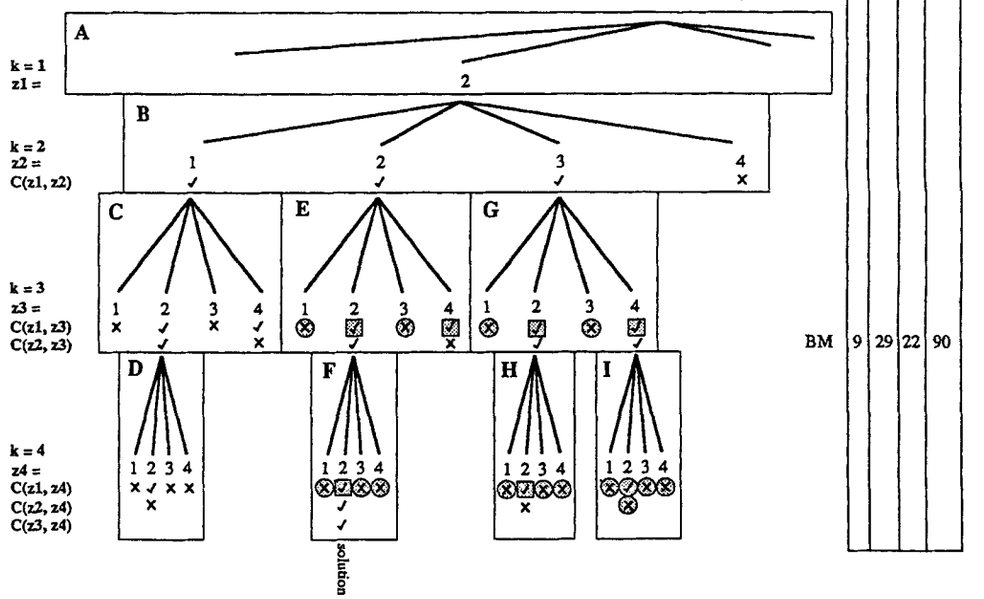


FIG. 2. The $z_1 = 2$ subtree when solving confused 4-queens by BT and BM. (Greyed-out constraint checks are avoided by BM.)

individual MaxCheckLevel values of BJ2, whereas these are forgotten in the latter algorithm. This remembering is achieved by (*i*) having MaxCheckLevel of BM be an array, as opposed to an integer variable in BJ2, and storing individual MaxCheckLevel values for (*k*, *val*) pairs in MaxCheckLevel[*k*, *val*], and by (*ii*) having MaxCheckLevel of BM be a reference parameter so that its data from a given node is available to chronologically later nodes in the search. Figure A1 of Appendix I gives a more detailed view of the processing in Fig. 2, showing values of the MaxCheckLevel[*k*, *val*] and MinBackupLevel[*k*] variables.

We can think of the BM's savings compared to BT as being "vertical" savings, while BJ's savings are "horizontal." BM saves checks by possibly doing less checks for a given instantiation. We call this a vertical savings since the checks for a given instantiation appear vertically in our traces. BJ, on the other hand, once it makes a given instantiation, has no way of avoiding any of the checks made by BT. Rather, by backjumping over multiple levels, BJ may avoid some instantiations at ancestor nodes, and hence avoid the checks and possible descendant nodes corresponding to these avoided instantiations. Avoiding instantiations at a node is a horizontal savings because successive instantiations occur horizontally in our traces. Whereas, compared to BT, BJ may avoid instantiations (and corresponding checks and nodes), but may not avoid checks for a given instantiation

once made; BM may avoid checks for a given instantiation, but may not avoid any instantiations (or nodes). These effects are perhaps clearer in the more detailed traces of Fig. A1 in Appendix I.

Something to think about would be a synthesis of BM and BJ into an algorithm called, say, BMJ (BackMarkJump). We see in Figs. 1 and 2 that each algorithm avoids some checks that the other doesn't. Is it possible to combine both approaches while retaining all, or most, of the power of each? Our preliminary attempt at such an algorithm suggests that the answer may be no. This is perhaps why Gaschnig did not suggest such a synthesized algorithm. However, more thought on this is warranted.

4. Arc consistency algorithms

An important class of algorithms for (partially) solving constraint satisfaction problems is what Mackworth (1977*a*) has called *arc consistency* (AC) algorithms. Their development can be traced back to the apparently independent work of Waltz (1975), Ullman (1973, 1976), and Fikes (1970). Gaschnig (1974, 1978, 1979) was also one of the first in this area. Others that have been active in developing (full or partial) arc consistency algorithms are Rosenfeld (1975), Rosenfeld *et al.* (1976), McGregor (1979), Mackworth (1977*a*), Freuder (1978), Haralick *et al.* (1978), Haralick and Shapiro (1979, 1980), Haralick and Elliot (1980), and Mohr and Henderson (1986).

An arc consistency algorithm can be thought of as a simplification algorithm which transforms the original problem into a simpler version that has the same solutions. In some cases the resulting problem is so simple that the solutions (or lack thereof) become manifest and the original problem is solved. Often, however, the simplification process "dries up" before the solutions are exposed and a nontrivial problem remains to be solved. More extensive simplifications are possible. The path consistency algorithms PC1 and PC2 of Montanari (1974) and Mackworth (1977*a*) and PC3 of Mohr and Henderson (1986) represent another level in this simplification approach. Arc and path consistency are further generalized by Freuder's concept of j-consistency (Freuder 1978), in terms of which the former become 2-consistency and 3-consistency respectively. A problem on n variables is always solved when n-consistency is attained, but this approach is usually grossly inefficient. We therefore concentrate here on the relatively low-level simplification achieved by arc consistency algorithms (full and partial) in anticipation of later using these in hybrid tree search/arc consistency algorithms which (*i*) are relatively efficient and (*ii*) are guaranteed to find all solutions.

Arc consistency algorithms are best discussed in terms of the *constraint network* representation of a consistent labeling problem. A constraint network is a labeled graph with a node for each problem variable and an arc between each pair of nodes for which there is a constraint between the corresponding two variables. The nodes and (optionally) the arcs are labeled respectively with the names of the corresponding variables and constraints. Examples for 3-queens and 4-queens problems (under the standard CSP formulation) are seen in Fig. 3. Of course, for q-queens problems the constraint networks involve *complete* graphs (on q nodes) because there is a constraint for each pair of variables. In the above form, the constraint network representation is applicable only to problems whose constraints each involve no more than two variables. If constraints having three or more argument variables are involved, a graphical depiction will require hypergraphs (Montanari and Rossi 1988) or some other generalization (Freuder 1978). Since for simplicity we are assuming only binary constraints, contraint network representations can be used here.

Mackworth (1977*a*) used G to denote the constraint network on nodes 1 to n. We will instead use $G_{1:n}$ for this, since we will find it useful below to generalize to arbitrary subnetwork on variables i to j, which we denote by $G_{i:j}$. By definition, the latter subnetwork includes all the arcs between the variables i to j that were in the full constraint network.

Not all the information about a problem instance is captured in its constraint network. In particular, the latter representation does not show the domain values for each variable, nor the specific nature of the constraints involved. This can be overcome by use of an *expanded constraint network* representation, an example of which is seen in Fig. 4 for 3-queens. Such a representation has also been used by Gaschnig (1974, 1978), Haralick *et al.* (1978), and Haralick and Shapiro (1980). Each network node is expanded to include one subnode for each domain value of the corresponding variable, and each pair of values that is consistent for two constrained variables is joined by its own (sub)arc. Two variables that have no problem constraint between them can be thought of as being constrained by an implicit universal (i.e., all-permitting) constraint. If the consistent value pairs for such universally constrained pairs of variables are also linked by an arc in the expanded constraint network, then a solution to an n-variable problem corresponds to an n-clique, and the problem of finding all solutions becomes the problem of finding all n-cliques.

A different graphical representation that will be convenient here involves what we call the *domain array*. Figure 5 shows an example for 4-queens. A domain array has a row for each problem variable. The rows therefore correspond to nodes in the constraint network representation. Each row is labeled by the corresponding variable name (unless some standard ordering is implicit, such as the natural order z_1 to z_n running from top to bottom). The ith cell of a row corresponds to the ith domain value (under some assumed ordering) for the corresponding variable; array rows will therefore be of different lengths when variables have different domain sizes. For q-queens problems (regular or confused) this representation is particularly natural, as the domain array, using the natural order for rows and columns, is isomorphic to the underlying chess board. (We assume such a correspondence in the examples below). But, of course, the domain array representation applies to any csp, not just those based on chess boards.

In a domain array, a value that has been eliminated from the domain of a variable (by arc revision, as discussed below) is denoted by the corresponding domain array cell being white. A still viable domain value is denoted by a corresponding grey cell in the array. Later in the context of tree search, we will also use black cells for values that have been instantiated to the corresponding variable.

■ = assigned value
□ = eliminated value
▫ = still possible value

A domain array per se does not show which variables are mutually constraining. This information can, however, be

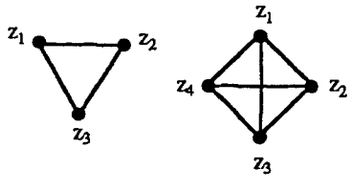


FIG. 3. Constraint networks for 3-queens and 4-queens.

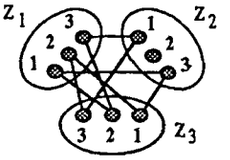


FIG. 4. Expanded constraint network for 3-queens.

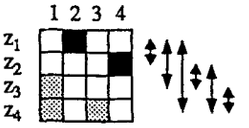


FIG. 5. Domain array for 4-queens.

easily added in the form of lines or bidirectional arrows at the side of the array, between the corresponding array rows, as shown in Fig. 5 for the 4-queens case. Later when discussing arc revision, these lines or bidirectional arrows become unidirectional arrows to correspond to *directed* arcs in the constraint network. The directed arc *from* node z_i *to* node z_j will be written as $(i\ j)$. Node z_i will be referred to as the *source* of the directed arc and node z_j as the *target*. We will sometimes loosely refer to node (variable) z_i as simply node (variable) i.

The detailed constraint information (legal value-pairs) denoted by the arcs of an expanded constraint network is not intended to be incorporated into a domain array representation. The strength of the representation is in tracking the process of *arc revision* by showing which values of which domains have been eliminated from consideration, which are still valid candidates and, in the context of tree search, which have already been instantiated to the corresponding variable.

Arc revision is the basic simplification process in the arc consistency algorithms that follow. A domain value for variable z_i for which there is no corresponding domain value for z_j compatible with respect to the constraint $C(z_i, z_j)$ between the variables may be removed from consideration. Such a value for z_i can never appear in a solution, since it has no compatible partner value for z_j. Removing *all* domain values of z_i that do not have at least one compatible z_j value is known as *revising* the directed arc $(i\ j)$. The directed arc $(i\ j)$ is said to then be *consistent*. Our version of Mackworth's revise(i, j) function for doing this is given below. An interesting variation is Gaschnig's revise-both(i, j) procedure (Gaschnig 1978, 1979), which revises both arcs $(i\ j)$ and $(j\ i)$ in one call, at no more cost than the corresponding two calls to revise separately.

```
PROCEDURE revise(i, j, VAR d, VAR empty_domain,
                 VAR change);
change ← False;
di ← d[i];
FOR vali ← each element of di DO
  BEGIN
  support_found_for_vali ← False;
  FOR valj ← each element of d[j]
    WHILE not support_found_for_vali DO
    IF check(i, vali, j, valj) THEN
      support_found_for_vali ← True;
  IF not support_found_for_vali THEN
    BEGIN d[i] ← d[i] - (vali); change ← True END
  END;
IF d[i] = empty THEN empty_domain ← True
END;
```

The reference array parameter d contains in d[i] the current (possibly filtered) version of the domain d_{z_i} for variable z_i, $1 \le i \le n$. Variables empty_domain and change are included as reference parameters to inform the calling routine respectively whether the domain of z_i was totally depleted and whether any deletion occurred at all. The latter parameter will be useful later only occasionally.[10] In a call where it is not needed we will use an actual parameter called dummy as a place-holder.

The above version of revise works by deleting values from the domain in d[i]. However, an "additive" version of revise is also possible (as implicit in Fig. 3 of McGregor (1979) and in Check_Forward of Haralick and Elliot (1980), but not in their Look_Future or Partial_Look_Future which both use the deleting approach above). In the additive version, the filtered domain in d[i] is obtained by successively adding on to an initially empty list, each value that is found to have support from a value in d[j], rather than by successively deleting from the original list d[i] the values found not to have any support. The additive approach is a little less concise in our pseudo-language. But it may be easier to implement efficiently for languages without good list-processing facilities, because it allows an array to be used to represent a list d[i] without the need to move up nondeleted elements to fill the holes of deleted ones.

Revising several arcs of a problem's constraint network may be sufficient to solve the problem. An example is shown in Fig. 6 where we see that for 3-queens, revising only three arcs is sufficient to eliminate all values of a domain and hence to show that there are no solutions. For the purposes of comparison, the figure shows the domain array, the constraint network, and the expanded constraint network representations of the arc revisions taking place. Filtering a domain down to zero remaining values we call a *domain wipe-out*. In domain arrays, its occurrence is indicated by a row all of whose cells are white (remember, a white cell denotes an eliminated domain value), and for emphasis, we also draw a wavy line through the row as in Fig. 6.

To clarify the details of arc revision processing, we include the following listing of the specific constraint checks performed in the three arc revisions of Fig. 6.

(1 2) 1121F 1122F 1123T 1221F 1222F 1223F 1321T
(3 2) 3121F 3122F 3123T 3221F 3222F 3223F 3321T
(3 1) 3111F 3113F 3311F 3313F

[10]Specifically, it will be needed only in defining the full arc consistency algorithms AC1, AC2, AC3 and in defining AC½ and revise in that they are needed for AC1.

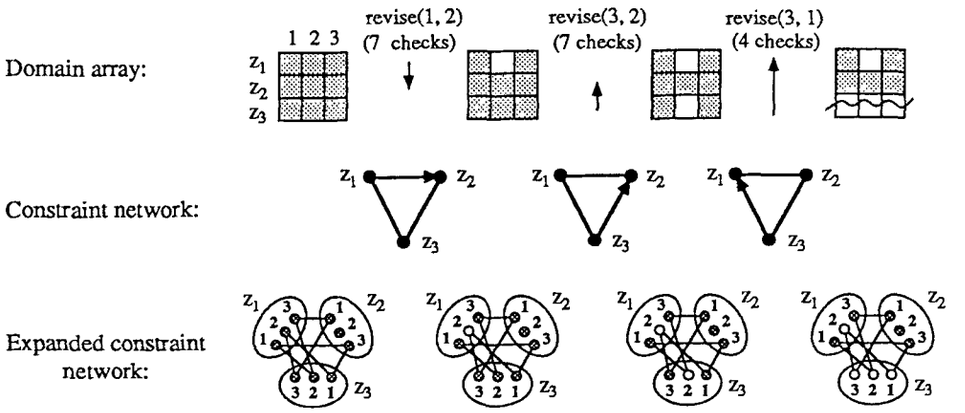


FIG. 6. Revising three directed arcs for 3-queens is sufficient to show there are no solutions.

Following Gaschnig (1979), we use 5-tuples here to denote constraint checks. The meaning of the 5-tuple ABCDE is that instantiation z_A = B was checked against z_C = D, and that the result was E, where E can be either T for true or F for false, indicating respectively that the corresponding binary constraint was found to be satisfied or violated by the pair of instantiations. For example, tuple 3123T means that z_3 = 1 was checked against z_2 = 3, and these were found to be consistent. Successive checks for a given arc revision appear left to right on a given line above. Successive arc revisions appear on successive lines, preceded by the corresponding arc (A C). Similar conventions are used in Appendix I where detailed constraint-check-level traces are given. Note that in revising arc (3 1) above, checks involving z_1 = 2 and z_3 = 2 are not tried, since these values have been removed for the corresponding variables by the first two arc revisions.

Unlike for the above 3-queens example, in general, revising arcs is not sufficient to solve a constraint satisfaction problem. (See the related discussion near the start of Sect. 4.2.) It may not even suffice to achieve any significant simplification of the domains. Extreme examples are the q-queens problems for $q \geq 4$. In each case, one can revise (at significant total cost) each of the $2\binom{q}{2}$ directed arcs of the constraint network and not find a single domain value that is eliminated. We are thus led to several possibilities:

1. One may pursue further degrees of simplification, such as path consistency, along the j-consistency spectrum. But short of attaining n-consistency, this approach on its own is still not guaranteed to solve the problem, and if it does, it is usually not cost-effective.

2. One may apply a simplification process as a preliminary to a tree search algorithm such as those of Sect. 3. Again, high-order simplification is usually not cost-effective in preprocessing, but low-order may be.

3. One may apply a simplification process (again, low-order appears best) at each node of the tree search algorithm. We will see that many important algorithms can in fact be seen to be of this form. Approach 2 above is a special case of this where simplification is applied only at the root node.

In anticipation of approach 3, the next subsection defines certain *partial* arc consistency algorithms for revising various subsets of the arcs in a constraint network. The subsection after that defines some important *full* arc consistency algorithms. The sense in which these are full and partial AC algorithms will be discussed later. Section 5 shows how both full and partial AC procedures may be incorporated into a tree search process. To allow this, each AC procedure below (unlike the original AC1, AC2, and AC3 of Mackworth (1977*a*) has a parameter k, to later correspond to the depth in the search tree, and, like revise itself, has a parameter d to later store the state of each domain local to a given search tree node. Also, like revise (but unlike the original AC1, AC2, and AC3), each of our AC procedures has a parameter empty_domain for letting its calling routine know if a domain was filtered to empty. In each of our AC algorithms, arc revision ends as soon as a domain is made empty.

Unlike for the other two classes of algorithms (tree search and hybrid), the AC algorithms of this section are given without any accompanying example traces. This is because the traces of the hybrid algorithms of Sect. 5 can double as traces for the AC algorithms, since the latter algorithms are used as components at the nodes of the former. Thus the reader should refer to the node processing of the hybrid algorithm traces in Figs. 10–12 of Sect. 5 and also in Figs. A2–A.7 of Appendix I, for examples of the working of this section's algorithms.

4.1. *Partial arc consistency algorithms*

This section introduces several partial arc consistency procedures for revising various combinations of arcs in the constraint network $G_{1:n}$. The particular combinations revised are motivated by the hybrid algorithms of Sect. 5, in which the present AC procedures are used as components. We name these procedures ACi, for various fractional i. This is by analogy with the classic full arc consistency algorithms (discussed in Sect. 4.2) which Mackworth (1977*a*) has called AC1, AC2, and AC3. Our fractional suffixes are intended to denote partial arc consistency, with the fraction being more or less proportional to the degree of arc consistency attained.

```
PROCEDURE AC⅓ {or Check_Backward}
  (k, VAR d, VAR empty_domain);
empty_domain ← False;
FOR p ← 1 TO k - 1 WHILE not empty_domain DO
  revise(k, p, d, empty_domain, dummy)
END;
```

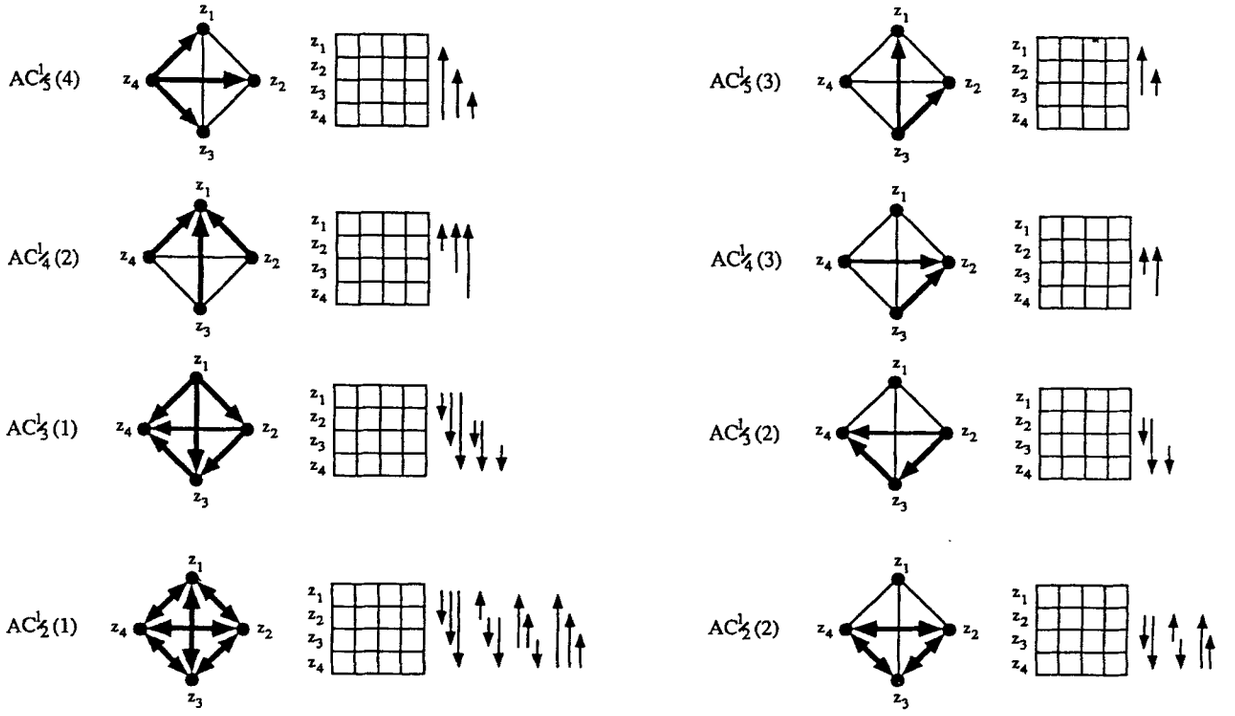


FIG. 7. Examples of arcs revised by various calls to our parametric partial arc consistency algorithms AC$i(k)$, i = 1/5, 1/4, 1/3, 1/2.

The above procedure revises the arcs (k, p), $1 \leq p < k$, once each, in lexographical order. It is essentially the unnamed procedure of McGregor (1979, p. 241). This was found after it was independently arrived at here for the purpose of reformulating (in Sect. 5.2.4) the Backtracking algorithm of Sect. 3.1 as a tree search/arc consistency hybrid algorithm. This reformulation corresponds to a loop-nesting interchange of the original BT. If in AC1/5 one expands out the call to revise, we see that the nesting of loops is essentially nesting (a) below. BT of Sect. 3.1, on the other hand, corresponds to the loop nesting shown in (b). We will see that when AC1/5 is used in the context of the reformulated, revise-based BT, there is in fact only one value in each d[p], so the last loop of nesting (a) may be ignored, leaving nesting (b), but with the order reversed. This will be discussed further in Sect. 5.2.4.

```
(a) FOR p ← 1 TO k−1 DO
      FOR each valk in d[k] DO
        FOR each valp in d[p] DO ...
(b) FOR each valk in d[k] DO
      FOR p ← 1 TO k−1 DO ...
```

The following procedure revises the arcs $(f, k-1)$, $k \leq f \leq n$, once each, in lexographical order. It is essentially the procedure that Haralick and Elliot (1980) called Check_Forward. However, it also appears earlier, unnamed, in Fig. 5 of McGregor (1979). It will be useful later in defining several hybrid algorithms of Sect. 5.

```
PROCEDURE AC1/4 {or Check_Forward}
  (k, VAR d, VAR empty_domain);
empty_domain ← False;
FOR f ← k TO n WHILE not empty_domain DO
  revise (f, k−1, d, empty_domain, dummy)
END;
```

In Sect. 5 when it is used as a component in hybrid algorithms, AC1/4 will also use revise in the specialized manner implied above for AC1/5. Due to the instantiations that will have occurred at ancestor tree nodes, both these procedures will be used to only revise arcs $(i\ j)$ whose target nodes j have exactly one domain value. The other partial and full arc consistency procedures below, when used in the hybrid algorithms of Sect. 5, will, however, use revise in its full generality with both nodes i and j usually containing more than one value. It is convenient to introduce the following subroutine rrevise (note the double r) for making multiple calls to revise. In particular, rrevise revises the arcs $(f1, f2)$, f2_min $\leq f2 \leq$ f2_max, $f2 \neq f1$, once each, in lexographical order.

```
PROCEDURE rrevise (f1, f2_min, f2_max, VAR d,
                   VAR empty_domain,
                   VAR deletion_occurred);
deletion_occurred ← False;
empty_domain ← False;
FOR f2 ← f2_min TO f2_max WHILE not
  empty_domain DO
  IF f2 ≠ f1 THEN
    BEGIN
    revise(f1, f2, d, empty_domain, change);
    deletion_occurred ← deletion_occurred or change
    END
END;
```

We can use rrevise to write more concisely several useful arc consistency algorithms. For example, the above AC1/5 can now be rewritten simply as

```
PROCEDURE AC1/5 {or Check_Backward}
  (k, VAR d, VAR empty_domain);
rrevise(k, 1, k−1, d, empty_domain, dummy)
END;
```

Note that the deletion__occurred parameter of rrevise serves the same purpose as the change parameter of revise: to flag whether any domain value at all was deleted. As with the change parameter itself, the value of deletion__occurred is not usually used by the calling routine, in which case (as in AC⅕ above) a corresponding actual parameter dummy will occur in the call to rrevise.

Unlike AC⅕, which varies the target node (the second parameter of revise) of the arcs being revised, AC¼ varies the source node (the first parameter of revise) and thus cannot be rewritten in terms of rrevise. However, rrevise is helpful in defining the two partial arc consistency procedures AC⅓ and AC½ below.

```
PROCEDURE AC⅓ {or re-nested Partial__Look__Future}
                    (k, VAR d, VAR empty__domain);
empty__domain ← False;
FOR f ← k TO n − 1 WHILE not empty__domain DO
  rrevise(f, f + 1, n, d, empty__domain, dummy)
END;
```

This procedure revises the arcs $(f1, f2)$, $k \leq f1 < f2 \leq n$, once each, in lexographical order. It is a revise-based version of the procedure which Haralick has called Partial__Look__Future, used in his Partial Lookahead algorithm (Sect. 5.2.2 below). However, to allow it to be based on revise we have had to introduce a loop renesting. If in AC⅓ one expands out the call to rrevise and its calls to revise, we see that the nesting of loops is essentially nesting (a) below, while Haralick's version uses nesting (b).[11]

```
(a) FOR f1 ← k TO n − 1 DO
      FOR f2 ← f1 + 1 TO n DO
        FOR each val1 in d[f1] DO
          FOR each val2 in d[f2] DO ...

(b) FOR f1 ← k TO n − 1 DO
      FOR each val1 in d[f1] DO
        FOR f2 ← f1 + 1 TO n DO
          FOR each val2 in d[f2] DO ...
```

Another partial arc consistency algorithm that will be useful is AC½ below. It is a more extensive version of AC⅓. It revises the arcs $(f1, f2)$, $k \leq f1 \neq f2 \leq n$, once each, in lexographical order, rather than just arcs $(f1, f2)$, $k \leq f1 < f2 \leq n$. Thus AC½(k) revises once each directed arc in the subnetwork $G_{k:n}$.

```
PROCEDURE AC½ {or re-nested Look__Future}
  (k, VAR d, VAR empty__domain,
  VAR deletion__occurred);
empty__domain ← False;
deletion__occurred ← False;
FOR f ← k TO n WHILE not empty__domain DO
  BEGIN
  rrevise(f, k, n, d, empty__domain, deletion);
  deletion__occurred ← deletion__occurred or deletion
  END
END;
```

We will see that AC½ provides a convenient way to define the full arc consistency algorithm which Mackworth has called AC1 (Sect. 4.2.1). It is for this purpose only that AC½ (unlike AC⅓, AC¼, and AC⅕) needs the extra parameter deletion__occurred, to keep track of whether any call to rrevise (via any of its calls to revise) caused any domain to undergo a deletion.

Also, AC½ is a revise-based version of the procedure which Haralick has called Look__Future, used in his Full Lookahead algorithm (Sect. 5.2.1 below). Again however, this revise-based version introduces a re-nesting of loops. AC½ uses the loop nesting (a) below, while Haralick's Look__Future uses nesting (b).[12]

```
(a) FOR f1 ← k TO n DO
      FOR f2 ← k TO n, skipping f1, DO
        FOR each val1 in d[f1] DO
          FOR each val2 in d[f2] DO ...

(b) FOR f1 ← k TO n DO
      FOR each val1 in d[f1] DO
        FOR f2 ← k TO n, skipping f1, DO
          FOR each val2 in d[f2] DO ...
```

The three loop-nesting rearrangements above are conceptually important. Through them we convert (Sect. 5.2) the standard Backtracking algorithm, and Haralick's Partial Lookahead and Full Lookahead algorithms, into revise-based versions. This simplifies their structure and unifies them with a whole spectrum (Sect. 5) of tree search/arc consistency hybrid algorithms, which use the partial arc consistency procedures above and the full arc consistency procedures below, as components.

Figure 7 shows the arcs revised for various example calls to the above partial arc consistency algorithms, for a problem with a complete constraint graph on 4 nodes. Both the constraint graph representation and the domain array representation are shown. As will be our convention from now on, the arcs beside a domain array representation are drawn left to right in the order that they are revised by the corresponding algorithm. To emphasize and clarify the k-parameterization of the algorithms, two different k values are used for each algorithm so as to vary the arcs that are revised. For simplicity, only the value of argument k is shown in the calls.

Note that, as exemplified in Fig. 7, AC⅕(k) applies to nodes 1 to k, but revises only a subset of the directed arcs in the corresponding subnetwork $G_{1:k}$. AC¼(k) applies to nodes $k - 1$ to n, but revises only a subset of the directed arcs in the corresponding subnetwork $G_{k-1:n}$. AC⅓(k) applies to nodes k to n, but revises only a subset of the directed arcs in the corresponding subnetwork $G_{k:n}$. AC½(k) applies also to nodes k to n and revises *all* the directed arcs in the corresponding subnetwork $G_{k:n}$. Thus all these procedures, except AC½(k), revise only a subset of the directed arcs between the nodes to which they apply. Moreover, none of these procedures, including AC½(k), can guarantee consistency on termination of even those arcs

[11] Actually, Haralick's Partial__Look__Future has an outer loop of FOR f1 ← k+1 TO n−1 DO, instead of FOR f1 ← k TO n − 1 DO. However, this is just an artifact of the difference, mentioned in connection with footnote 7, in what constitutes a node in our respective approaches.

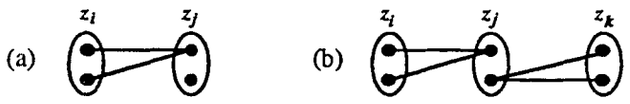
[12] Actually, Haralick's Look__Future has f1 and f2 loops from $k + 1$ to n, rather than from k to n. However, again this is just an artifact of the difference in what constitutes a node in our respective approaches, as discussed in connection with footnote 7.

which they do revise. This is discussed, and remedied, in the next section. However, we will see later that these apparent drawbacks are in fact usually advantages in the context of our hybrid algorithms of Sect. 5.

4.2. Full arc consistency algorithms

The arc consistency algorithms of the previous section revise various subsets of the arcs in the constraint network $G_{1:n}$ on nodes 1 to n. Each time an arc $(i\ j)$ is revised, values in node i that have no consistent supporting value in node j are deleted. Arc $(i\ j)$ is said to then be *consistent*. A whole constraint (sub)network is said to be *arc consistent* when all its arcs are consistent. An algorithm that guarantees arc consistency of the (sub)network on the nodes to which it applies is said to be a *full* arc consistency algorithm for that (sub)network. None of the algorithms of the previous section were full AC algorithms. The algorithms of this section are all full AC algorithms for the particular subnetwork $G_{k:n}$.

A full AC algorithm for a given subnetwork must revise all *directed* arcs in that subnetwork, since making directed arc $(i\ j)$ consistent does not necessarily ensure that directed arc $(j\ i)$ is consistent. This can be seen in example (a) below. (We are using here the expanded constraint network conventions. See Fig. 4). The need to revise each arc in *both* directions was the motivation behind Gaschnig's reviseboth procedure (Gaschnig 1978, 1979) mentioned earlier. Even revising each directed arc once, however, still does not necessarily achieve arc consistency of a subnetwork. This is because a consistent arc $(i\ j)$ may be made inconsistent again by a subsequent revision of some arc $(j\ k)$, as in example (b) below.



The algorithms of the previous section were *partial* AC algorithms in that they either did not revise all directed arcs in the corresponding subnetwork or they did not guarantee the eventual consistency of these arcs. AC1/5, AC1/4, and AC1/3 were partial in both these senses. AC1/2 was partial in only the latter sense. This section's algorithms revise all directed arcs of the subnetwork $G_{k:n}$, and ensure their consistency on termination, later arc revisions involving an arc's target node notwithstanding.

Note that there are three cases in which a problem is provably solved by achieving full arc consistency for its constraint network: (*i*) if a domain wipe-out is found to occur then it is known that no solutions exist, (*ii*) if all n domains end up with a single value, then the instantiation of the variables to their unique corresponding value is clearly a solution (and the only solution), and (*iii*) if $n-1$ domains end up with a single value, and the other domain, for variable z_i say, has $m > 1$ values, then there are m solutions. These correspond to the different instantiations of z_i to one of its values, each combined with the unique instantiations of the other $n-1$ variables. These three cases are implicit in Mackworth's NC algorithm (Mackworth 1977*b*). However, often none of the above apply, and we end up with two or more domains having multiple values. In such cases, full arc consistency may have achieved a substantial simplification, but the problem is still not actually solved.

4.2.1. AC1

AC1/2(k) revised once each directed arc of $G_{k:n}$, but did not guarantee their eventual arc consistency, because a latter revision may have undone the consistency of a previously revised arc. The most straightforward way to ensure arc consistency of subnetwork $G_{k:n}$ is then to simply repeat AC1/2(k) till no change occurs. It is for this reason that we included the extra deletion_occurred parameter in AC1/2 (but did not in AC1/3, AC1/4, and AC1/5). Repeating AC1/2 until no change occurs is essentially the arc consistency algorithm that Mackworth (1977*a*) has called AC1. Our version of the algorithm is as follows.

```
PROCEDURE AC1 (k, VAR d, VAR empty_domain);
{AC1/2 repeated till arc consistency of subnetwork G_k:n.}
REPEAT
  AC1/2(k, d, empty_domain, deletion_occcurred)
UNTIL (not deletion_occurred) or empty_domain
END;
```

Mackworth's AC1 (1977*a*) is given essentially by AC1′ below, and the version of Rosenfeld (1975) and Rosenfeld *et al.* (1976) (originally called Δ) is given by AC1″.

```
PROCEDURE AC1′;
Q ← { (i j) | (i j) is an arc in G_1:n };
REPEAT
  deletion_occurred ← False;
  FOR each (i j) in Q DO
    BEGIN
    revise(i, j, d, dummy, change);
    deletion_occurred ←
      deletion_occurred or change;
    END;
UNTIL not deletion_occurred;
END;

PROCEDURE AC1″;
Q ← { (i j) | (i j) is an arc in G_1:n };
REPEAT
  old_d ← d;
  deletion_occurred ← False;
  FOR each (i j) in Q DO
    BEGIN
    d[i] ← revise(i, j, d[i], old_d[j], change);
    deletion_occurred ←
      deletion_occurred or change;
    END;
UNTIL not deletion_occurred;
END;
```

AC1″ is less efficient than AC1′ on a regular machine, but offers the opportunity for a more efficient implementation on a parallel processor. AC1″ does not take advantage of deletions that occurred during a given pass over all arcs to improve the extent of filtering of other domains during the same pass. Instead, all deletions become effective for filtering other domains only at the next pass. It is because of this that each domain could, if processors were available, be filtered in parallel during a given pass. In AC1″ the call to procedure revise(i, j, d, empty_domain, change) of AC1′ has been replaced by a slightly different call to a function revise(i, j, di, dj, change) which filters domain di of z_i with respect to domain dj of z_j and returns the resulting filtered di as the value of the function; change is a reference para-

meter with the same meaning as before. Note that parallel approaches to solving constraint satisfaction problems are currently receiving considerable attention — see, for example, Freuder and Quinn (1985), McCall *et al.* (1985), and Kasif (1986). The latter paper sounds a note of restraint by showing that Constraint Satisfaction is in a sense an inherently nonparallizeable problem.

To make explicit the correspondence between Mackworth's AC1′ and our AC1, we expand out the call to AC½ in AC1. Employing Mackworth's list-of-arcs notation we obtain the following equivalent version of our AC½-based AC1.

```
PROCEDURE AC1(k, VAR d, VAR empty_domain);
REPEAT
  empty_domain ← False;
  deletion_occurred ← False;
  Q ← {(i j) | k ≤ i ≠ j ≤ n, in lexographical order};
                                        {arcs in G_k:n}
  WHILE (Q ≠ empty) and (not empty_domain) DO
    BEGIN
    (r s) ← pop(Q);
    revise(r, s, d, empty_domain, change);
    deletion_occurred ← deletion_occurred or change;
    END;
UNTIL (not deletion_occurred) or empty_domain;
END;
```

The phrase "in lexographical order" means that the pairs $(i\ j)$ appear left to right in the list Q with j ranging over all its values (except $j = i$) in order before the next i value is used. This is required so as to reflect the specific ordering induced by the $f1$-loop of AC½ and the $f2$-loop in its subroutine rrevise. (See the corresponding loop expansion at the end of Sect. 4.1.) Comparing this version of our AC½-based AC1 with Mackworth's version AC1′, we see that they are in fact equivalent except that

1. AC1 is more general than AC1′ in that it (like all our earlier partial arc consistency algorithms) has formal parameters k and d so that it can be incorporated at each node of a search tree (as in Sect. 5 below), rather than just at what in our context is the root, or the $k = 1$, node of the search tree. AC1 achieves arc consistency for any subnetwork $G_{k:n}$, $1 \leq k \leq n$, whereas AC1′ achieves it only for $G_{1:n}$.
2. AC1 is more specialized than AC1′ in being for *complete* binary constraint satisfaction problems (those that are binary and, moreover, have a binary constraint on each pair of variables). AC1′ does assume binary problems, but not necessarily complete binary problems.
3. AC1 is more specialized than AC1′ in using a specific order of processing: each cycle processes arcs $(i\ j)$ in lexographic order. AC1′ leaves the processing order arbitrary.
4. AC1 also differs from AC1′ by terminating as soon as a domain wipe-out occurs. In such a case AC1′ comes to a halt the hard way, by filtering every domain till it is empty.
5. Mackworth's AC1 makes an initial call to a *node consistency* algorithm NC for each variable. All algorithms here leave out such a call, on the assumption that no unary predicates are given for the problem or that they have already been incorporated in establishing the domains d_{z_i} used by the algorithms. For details see Mackworth (1977*a*).

The same differences will also apply between Mackworth's and our versions of AC2 and AC3 below. Note, that the restrictions mentioned in points 2 and 3 above may be removed from our AC1, while retaining the added generality of difference 1, by simply replacing the line that initializes Q with the new line[13] Q ← { (i j) | (i j) is an arc in $G_{k:n}$}, the form used by Mackworth but with $G_{1:n}$ in place of our $G_{k:n}$.

4.2.2. AC3

AC1 worked by making successive calls to AC½ till a call occurred in which no domain deletion took place. However, as pointed out by Mackworth (1977*a*), this is an unnecessarily wasteful way to achieve arc consistency. The obvious inefficiency is that any update of an arc $(r\ s)$ of $G_{k:n}$ on a given pass causes all arcs of $G_{k:n}$ to be revised on the next cycle, when in fact only the arcs $(j\ r)$ could possibly be affected.

This insight is embodied in the Mackworth's AC3 algorithm, a version of which follows.[14] Note that not only are arcs $(j\ i)$, $i \neq r$, not added to Q when revision of $(r\ s)$ causes a deletion in the domain of z_r, but neither is arc $(s\ r)$. This is because if arc $(s\ r)$ wasn't on Q already then it was consistent — and a consistent arc $(s\ r)$ cannot become inconsistent directly because of the revision of $(r\ s)$, because any z_r value removed was deleted precisely because it has no support in the domain of z_s and hence no z_s value was supported by it. On the other hand, if $(s\ r)$ *was* already on Q then it also needn't be added (again).

```
PROCEDURE AC3(k, VAR d, VAR empty_domain);
empty_domain ← False;
Q ← {(i j) | k ≤ i ≠ j ≤ n, in lexographic order};
                                        {arcs in G_k:n}
WHILE (Q ≠ empty) and (not empty_domain) DO
  BEGIN
  (r s) ← pop(Q);
  revise(r, s, d, empty_domain, deletion_occurred);
  IF ((deletion_occurred) and (not empty_domain))
    THEN
    BEGIN
    Q_extra ← {(j r) | k ≤ j ≤ n, j ≠ r, j ≠ s,
                  in lexographic order};
    Q ← post-union(Q_extra, Q);
    END
  END
END;
```

By post-unioning of Q_extra onto Q, achieved by the call post-union (Q_extra, Q), we mean that the arcs in list Q_extra that are not already in Q are unioned onto Q with new arcs being appended to the rear of Q, in the order of their occurrence in Q_extra. Since arcs are added to the rear and popped from the front, the list Q is therefore maintained as a *queue* — hence the name Q. This, together with the fact that Q is initialized to the same value in AC3 as in AC1, means that the first part of the AC3 processing is

[13]Note that at this level of generality, AC1 would then differ from AC1′ in the additional respect that it allows the order of arc processing to vary at each cycle because Q of AC1, but not Q of AC1′, is initialized anew on each cycle.

[14]Our AC1, AC2, and AC3 all differ from Mackworth's versions, in the ways discussed for AC1 in the previous section. In particular, there is no indeterminacy in the arc revision orders of our versions.

always the same as for the first pass of AC½ in AC1. This can be seen, for example, in Fig. 10.

A *stack-based* (rather than *queue-based*) version of AC3 might just as well have been written. This would use a procedure pre-union (instead of post-union) to add to the *front* of Q the arcs not already there from Q_extra. Or, any more-intelligent way of ordering the new arcs in Q could be used if appropriate heuristics are known. More research on this ordering issue would be appropriate. It is the analog for AC3 of the constraint-check order issue that arises for most, if not all, constraint satisfaction algorithms, and which was studied for Backtracking and Forward Checking in Haralick and Elliot (1980), Nadel (1986), and Nudel (1983*a*). An analogous list-ordering issue arises in the state space search algorithm A* where heuristic $\hat{h}$ evaluations are used to rank entries. A similar approach should be applicable here using mathematically derived heuristics analogous to those obtained in Haralick and Elliot (1980), Nadel (1986), and Nudel (1983*a*). Whichever version of adding Q_extra to Q is used, AC3 (and also AC2 below) usually attains arc consistency considerably more efficiently than its precursor AC1. See, for example, the corresponding AC1 and AC3 (and AC2) counts in Fig. 10, and in Tables 2 and 3.

4.2.3. AC2

Another full arc consistency algorithm, of historic as well as practical interest, is that which Mackworth (1977*a*) called AC2. It is essentially the version used by Waltz (1975) in his seminal work on machine vision, in which the blocks world line-labeling problem is formulated and solved as a constraint satisfaction problem. The following is a version of AC2 analogous to our AC1 and AC3 above.

```
PROCEDURE AC2(k, VAR d, VAR empty_domain);
empty_domain ← False;
FOR i ← k+1 TO n WHILE (not empty_domain) DO
  BEGIN
  Q1 ← {(i j) | k ≤ j < i, in lexographic order};
                              {arcs from node i in G_k:i}
  Q2 ← {(j i) | k ≤ j < i, in lexographic order};
                              {arcs to node i in G_k:i}
  WHILE (Q1 ≠ empty) and (not empty_domain) DO
    BEGIN
    WHILE (Q1 ≠ empty) and (not empty_domain) DO
      BEGIN
      (r s) ← pop(Q1);
      revise(r, s, d, empty_domain,
             deletion_occurred);
      IF ((deletion_occurred) and (not empty_domain))
        THEN
        BEGIN
        Q2_extra ← {(j r) | k ≤ j ≤ i, j ≠ r, j ≠ s,
                    in lexographic order};
        Q2 ← post-union(Q2_extra, Q2);
        END
      END
    Q1 ← Q2; Q2 ← empty
    END
  END
END;
```

In spite of its more complex structure, AC2 above is essentially just the earlier AC3 with another arc-revision ordering, so as to achieve arc consistency of $G_{k:n}$ in one pass through the nodes from z_k to z_n. For example, when $k = 1$ and $n = 4$ and, for simplicity, when no empty domains occur and $G_{1:n}$ is already arc consistent so that no deletions occur, AC2(k) revises arcs in the order

$$\overbrace{(2,1)(1,2)}^{i=2}\ \overbrace{(3,1)(3,2)(1,3)(2,3)}^{i=3}\ \overbrace{(4,1)(4,2)(4,3)(1,4)(2,4)(3,4)}^{i=4}$$

as in the AC2(1) processing in Fig. 10 (even though that example does not conform to the present assumption of initial arc consistency). AC3(1), however, would use the order (1,2)(1,3)(1,4)(2,1)(2,3)(2,4)(3,1)(3,2)(3,4)(4,1)(4,2)(4,3), the same as a single cycle of AC½(1) shown in Fig. 10. In the same case but with $k = 2$, AC2(k) revises arcs in the order

$$\overbrace{(3,2)(2,3)}^{i=3}\ \overbrace{(4,2)(4,3)(2,4)(3,4)}^{i=4}$$

as in the AC2(2) processing in Fig. 10 (even though again that example does not conform to the present assumption of initial arc consistency). AC3(2) revises in the order (2,3)(2,4)(3,2)(3,4)(4,2)(4,3), the same as a single cycle of AC½(2) shown in Fig. 10.

Note that in Nadel (1988*a*), the outer loop of AC2 starts with $i \leftarrow k$, generalizing Mackworth's nonparameterized AC2 (Mackworth 1977*a*) which starts with $i \leftarrow 1$. However, such lower bounds for i correspond to Q1 and Q2 lists that are empty, and hence to a first cycle which does nothing. One may as well, therefore, start with $i \leftarrow k + 1$ as above (or with $i \leftarrow 2$ in Mackworth's version). Again, note also that the arcs of Q2_extra might just as well have been unioned on to the front of Q1, or inserted in any other order, with the same outcome but with usually different efficiency. Our examples and experiments below all use the above post-unioning versions of AC2 and AC3, with initial lexographic ordering of arc lists as given.

Actually pre-unioning is more efficient than post-unioning when considered in isolation, because lists are usually referenced by pointers to their front, not to their ends, so that pre-unioning's updating at the front of a list is quicker. However (apart from possibly imposing an arc revision order that is better for the overall process of attaining arc consistency), post-unioning gives an order that is better for pedagogical purposes because its first-come-first-served order for arcs is easier to follow in traces and, as mentioned, allows the initial AC3 processing to be always the same as the first cycle of AC½ processing in AC1.

In general, we find AC2 to be better than AC3 (our specific version), as in the examples of Fig. 10 and in most cases where AC2 and AC3 are used in the hybrid algorithms compared in Tables 2 and 3 of Sect. 6. Also, of course, both AC2 and AC3 are generally better than AC1. There are examples, however, where AC3 is better than AC2 (Table 2, RFL2 versus RFL3) and, surprisingly, where even AC1 is better than AC2 (Table 2, RFL1 versus RFL2). This latter possibility does not seem to have been noted before. It is further discussed in Sect. 6 and Appendix I. Supposedly there are also examples where AC1 is better than AC3. Note that other approaches to efficient full arc consistency processing also exist. Gaschnig's DEE (1978, 1979) and Mohr and Henderson's AC4 (1986) are of particular interest. All these full arc consistency algorithms of course achieve the

same final state for a problem's constraint graph, except possibly that a domain wipe-out, if one occurs, may be discovered by different algorithms to occur at different net nodes.

5. Hybrid tree search/arc consistency algorithms

In Sects. 3 and 4 we considered respectively tree search algorithms and arc consistency algorithms. The prototypical tree search algorithm, Backtracking, was seen to have certain *thrashing* inefficiencies, which could be ameliorated by the more refined tree search algorithms Backjumping and Backmarking. No doubt certain types of inefficiencies still remain in these algorithms too. We saw that arc consistency algorithms could be used to simplify a constraint satisfaction problem, but that often such simplification was insufficient to actually solve the problem.

It is conceivable that a marriage between tree search and arc consistency algorithms would be beneficial, each type overcoming the weaknesses of the other, and building on the other's strengths. It is certainly easy to overcome the above-mentioned "incompleteness" problem of arc consistency algorithms by extending the AC processing into a tree-structured form, where a simplification phase is followed by a decomposition into subproblems by instantiating an as-yet uninstantiated variable in all ways, with the process being recurvisely repeated on each of the subproblems. Such an extension of arc consistency processing can readily be designed to ensure the finding of all solutions.

Conversely — although it really amounts to the same thing — one could embed arc consistency processing at the nodes of a search tree as a way of reducing thrashing. For example, in Fig. 1 we saw that Backtracking was wasteful in trying more instantiations of z_3 at node G after failing at node H, because the reason for failure at H would only repeat itself (as it did at node I). The problem was that no value of z_4 was compatible with both $z_1 = 2$ and $z_2 = 3$, and since these instantiations don't change in trying alternative children of G, then other such children will fail as did child H. Backjumping was one way around the problem. However, an alternative would be to revise arcs (4 1) and (4 2) of the constraint graph after the $z_2 = 3$ instantiation at node B. This would result in all domain values for z_4 being eliminated. Nodes G and H would not even be generated, let alone node I.

Thus we see that the hybridization of tree search and arc consistency may very well be a useful approach. It turns out that quite a few important CSP algorithms (or slight rearrangements of them) are in fact of this type. These will be studied in this section. Hybrids that achieve full arc consistency of the whole constraint network at each search tree node are treated in Sect. 5.1. Hybrids that achieve partial arc consistency of the network at the tree nodes are treated in Sect. 5.2. Note that achieving full (partial) arc consistency at a tree node is not the same thing as using a full (partial) arc consistency algorithm at the node. For instance, algorithms RFL*i* below use as one component the partial AC algorithm AC¼, even though they achieve full arc consistency at each node. Similarly, it is possible to achieve only partial arc consistency at a node when using a full AC algorithm if an inadequate subgraph (of the constraint graph) is processed at the node.

We can expect that the lower the degree of simplification attained at the tree nodes, the larger will be the search tree. However, less simplification per node means less effort per node and possibly less effort for the whole tree, even though the tree is larger. The big question is: where is the break even point? In other words, what is the optimal amount of simplification to apply at the search tree nodes? It is conceivable that the optimal amount of simplification to achieve is path consistency, or even some higher level of *j*-consistency Freuder (1978). However, this has not been found to be the case for the problem classes studied in previous experiments McGregor (1979). The results here and in Gaschnig (1978, 1979), Haralick and Elliot (1980), and McGregor (1979) support this observation by showing that even full arc consistency per node is excessive. Though the break-even point no doubt depends on the type of problems involved, we will see that for the problems studied here, the optimum approach is to use only a very restricted form of arc consistency per node. For pedagogical reasons, it will be convenient to present the algorithms in decreasing order of degree of arc consistency attained per node. (This order, however, is the opposite of that used in Nadel (1988*a*).)

Now that we are discussing arc consistency in the context of tree search, it is important to keep clear the distinction between nodes of the search tree and nodes of the underlying constraint network. We will therefore distinguish them as *tree nodes* and *net(work) nodes* respectively. When unqualified, the term *node* will refer to a tree node. Remember that network nodes correspond to problem variables z_i and hence to the rows of a domain array diagram, such as that in Fig. 5. Thus when we speak of *variable*, *variable domain*, or *domain array row* below, these are essentially just synonyms for *net node*. In our graphical traces of the algorithms of this section, domain arrays will be drawn at the nodes of the search trees, giving a clear indication of the state of the network nodes in the context of each of the search tree nodes.

Note that in this section we begin to realize the advantage of parameterizing our arc consistency algorithms with *k*, intended to correspond to the search tree level, with d, intended to store the list of value domains local to a given tree node, and with empty_domain, to tell us whether a domain wipe-out occurred during consistency processing. Such parameters do not appear in the traditional formulations of consistency algorithms (Freuder 1978; Mackworth 1977*a*; Mohr and Henderson 1986; Montanari 1974). The purpose of our parameters can be seen from the following procedure TS, which is the common tree search shell from which all our hybrid algorithms below are built.

```
PROCEDURE TS (k, d);
{To expand, add your favorite parametric arc consistency
  procedure(s) here}
dk ← d[k];
IF not empty_domain THEN
  FOR zk ← each element of dk DO
    BEGIN d[k] ← (zk);
      IF k = n THEN output(d) ELSE TS(k+1, d) END
END;
```

Parameter d of TS is as described for revise of Sect. 4.1. At the initial call, we require that k = 1 and that each component d[i] of d contains the original domain d_{z_i} of variable z_i, $1 \leq i \leq n$. Note that d may not be a reference parameter of TS, unlike the corresponding array z in BT,

```
PROCEDURE TSVAC(k, d, variant);
IF k > 1 THEN BEGIN
              call1(variant, k, d, empty_domain);
              IF not empty_domain THEN call2(variant, k, d, empty_domain);
              END;
dk ← d[k];
IF not empty_domain THEN
  FOR zk ← each element of dk DO
    BEGIN d[k] ← (zk); IF k = n THEN output(d) ELSE TSVAC(k+1, d, variant) END
END;


PROCEDURE call1(variant, k, VAR d, VAR empty_domain);
CASE variant OF
  'BT'      : AC1/5(k,   d,  empty_domain);
  'FC', 'PL', 'FL', 'RFL1', 'RFL2', 'RFL3' : AC1/4(k,  d,  empty_domain);
  'TSAC1'   : AC1( k-1, d,  empty_domain);
  'TSAC2'   : AC2( k-1, d,  empty_domain);
  'TSAC3'   : AC3( k-1, d,  empty_domain);
  'TSRAC1'  : AC1( 1,   d,  empty_domain);
  'TSRAC2'  : AC2( 1,   d,  empty_domain);
  'TSRAC3'  : AC3( 1,   d,  empty_domain);
END; END;


PROCEDURE call2(variant, k, VAR d, VAR empty_domain);
empty_domain ← False;
CASE variant OF
  'PL'      : AC1/3(k,   d,  empty_domain);
  'FL'      : AC1/2(k,   d,  empty_domain,  dummy);
  'RFL1'    : AC1( k,   d,  empty_domain);
  'RFL2'    : AC2( k,   d,  empty_domain);
  'RFL3'    : AC3( k,   d,  empty_domain);
  'BT', 'FC', 'TSAC1', 'TSAC2', 'TSAC3', 'TSRAC1', 'TSRAC2', 'TSRAC3' :  {do nothing} ;
END; END;
```

FIG. 8. A combined algorithm TSVAC (Tree Search + Variable Arc Consistency) for BT, FC, PL, FL, and RFLi, TSACi, TSRACi for $1 \le i \le 3$.

TABLE 1. Structures of our 13 hybrid tree search/arc consistency algorithms (subsumed by TSVAC of Fig. 8)

Algorithms that achieve full arc consistency of constraint network at each tree node											
TSRAC$i(k)$	=	TS(k)			+	AC$i(1)$					i = 1, 2, 3
TSAC$i(k)$	=	TS(k)			+	AC$i(k-1)$					i = 1, 2, 3
RFL$i(k)$	=	TS(k)	+	AC$^1/_4(k)$	+	AC$i(k)$	=	FC(k)	+	AC$i(k)$	i = 1, 2, 3
Algorithms that achieve partial arc consistency of constraint network at each tree node											
FL(k)	=	TS(k)	+	AC$^1/_4(k)$	+	AC$^1/_2(k)$	=	FC(k)	+	AC$^1/_2(k)$	
PL(k)	=	TS(k)	+	AC$^1/_4(k)$	+	AC$^1/_3(k)$	=	FC(k)	+	AC$^1/_3(k)$	
FC(k)	=	TS(k)	+	AC$^1/_4(k)$							
BT(k)	=	TS(k)	+	AC$^1/_5(k)$							

BJ, and BM of Sect. 3, and unlike d in all the arc consistency algorithms of Sect. 4.

As indicated by the comment in TS, our hybrid algorithms are built from TS by inserting various arc consistency procedures immediately after the header line. At each level-k call TS (k, d), the domains of some of the nonpast variables z_k to z_n will be filtered; which specific domains are filtered, and using which arc revisions, depends on the choice of arc consistency procedure(s) incorporated into TS. The resulting filtered domains are stored in d[i], $k \le i \le n$. If no domain wipe-out occurs, the current variable z_k is then instantiated in turn to each of the values still viable in its current domain d[k], and for each instantiation the process is repeated recursively at the next level $k + 1$ using the updated array d of domains. The instantiation for z_k is stored in d[k] itself as a single-member list, replacing the earlier domain list stored there. (The term (zk) in the statement d[k] ← (zk) denotes this singleton list). Therefore on entry to a level-k node, the instantiations of the past variables z_1 to z_{k-1} are all available as the singleton-list components d[1] to d[$k - 1$] of d, and d[k] to d[n] contain the lists of still viable values for the nonpast variables.

Since the various hybrid algorithms below vary only as to which arc consistency procedure(s) is inserted after the header in the above tree search shell TS, they are so similar in structure that it is convenient to consider them here as special cases of a single combined algorithm, as shown in Fig. 8. This figure expresses in a unified way the 13 hybrid

CSP algorithms we describe below. Schematically, this unified structure may be expressed as TS + AC_i or TS + AC_1 + AC_2, where TS denotes the basic tree search skeleton above, and AC_1 and AC_2 denote one of our earlier full or partial arc consistency algorithms applied at the search tree nodes. The combined algorithm is thus called TSVAC, for Tree Search with Variable (i.e., changeable) Arc Consistency. Selection of the required component algorithm is done by inputting the corresponding value of the string parameter *variant*; for example 'BT' to obtain algorithm BT. Table 1 summarizes the structure of the hybrid algorithms we study in this section.

Note that our combined algorithm in Fig. 8 has a k > 1 test which excludes any arc consistency processing at the root node ($k = 1$) in each of the subsumed algorithms. Apart from the fact that some of the arc consistency calls do not make sense when $k = 1$ (AC1/5(k), AC1/4(k), and ACi($k-1$)), this is because arc consistency processing at the root node is usually not cost-effective. (Remember the extreme example of q-queens for $q \geq 4$, discussed in Sect. 4, for which revising every arc filters not a single domain value). In general, when an arc (i j) is revised at the root, too many supporting domain values exist in the target net node j to allow effective filtering of the values from the source node i. Moreover, compounding the problem, this filtering is not only ineffective but also particularly costly at the root because the relatively large domains there mean that many values for i must be attempted to be filtered, each time possibly checking against many values for j. At levels $k > 1$, however, instantiation of variables at ancestor nodes has reduced some domain sizes and removed potential supporting values, which allows the filtering process to be both more effective and more efficient. Extensions of this selective consistency-processing approach are discussed in Sect. 6.

Traces of our hybrid algorithms solving 4-queens and confused 4-queens appear below in Figs. 10–12. They show individual nodes, or node subtrees, generated by the algorithms and the arc revisions, and corresponding numbers of constraint checks, performed at each node. A node in a trace is shown in terms of the contents of array d at that node, using the domain array representation and shading conventions described near the start of Sect. 4. In particular, black cells denote an instantiated value, grey cells denote still viable values, and white cells denote values eliminated from consideration. Unless otherwise stated, the domain array for a node will show the contents of d just *after* all arc consistency processing (call1 and call2 in Fig. 8) has taken place, but before any instantiation (d[k] ← (zk) in Fig. 8) of the current variable. In some cases, the domain array for d prior to any AC processing at the node, and (in Appendix I) at various intermediate stages, may also be shown.

For simplicity, the traces of this section do *not* indicate the individual constraint checks performed for the arc revisions. The reader should be able to hand simulate the processing at this level of detail him/herself, coming up with the same constraint check numbers and final domain arrays as given at each node in the diagrams. Some traces at this detailed level of individual constraint checks appear in Appendix I below and in Nadel (1988*a*). As pointed out in connection with the AC algorithms of Sect. 4, the hybrid algorithm traces in the present section and in Appendix I also double as traces of the AC algorithms, since the latter are components of the former so that the AC algorithm processing shows up at the nodes of the hybrid algorithm traces.

5.1. Nine full arc consistency hybrids

This section presents nine hybrid tree search/arc consistency algorithms in three families of three algorithms each: TSRACi, TSACi, and RFLi for $1 \leq i \leq 3$. Each of the algorithms ensures that at each search tree node, full arc consistency is attained for the whole constraint graph $G_{1:n}$ corresponding to the set of all n problem variables. Within a given family of three, the algorithms differ in whether they use the full arc consistency procedure AC1, AC2, or AC3 (possibly in conjunction with the partial AC procedure AC1/4). The families themselves differ as to what subgraph of the constraint graph they apply their AC procedures to. The TSRACi, TSACi, and RFLi families correspond to successively smaller subgraphs and hence are successively more efficient.

Since each of the nine algorithms achieves the same state of full arc consistency at corresponding nodes, they all generate search trees with the same node structure and hence with the same number of nodes. This will be seen in the traces below and in the experiments of Sect. 6. The work done at a given node, and hence the overall work for the search tree, will of course differ though.

Many algorithms in the literature are of the type discussed in this section, for example, Gaschnig's CS2 (1974) and DEEB (Domain Element Elimination with Backtracking) (1978, 1979), and Mackworth's NC (1977*b*). Similarly, McGregor (1979) cites Ullman (1976) as having used such an approach, although application-specific details in Ullman's implementation obscure this. Since most of the latter algorithms, however, have not been given in sufficient detail, it is often not clear whether they correspond to the TSRACi, TSACi, or FRLi families. As we will see, the efficiency difference can be considerable between and within these families.

5.1.1. Tree search and redundant arc consistency: TSRACi(k) = TS(k) + ACi(1)

The calls AC1(k), AC2(k), and AC3(k) each achieve full arc consistency of the constraint network $G_{k:n}$. Thus the most straightforward way at each node of a search tree to achieve full arc consistency of the constraint network $G_{1:n}$ corresponding to all n variables is to call AC1(k), AC2(k), or AC3(k) with $k = 1$. Doing this gives us the variants TSRAC1, TSRAC2, and TSRAC3 of our combined algorithm in Fig. 8. The R in the names stands for *redundant*; we will see why shortly. These names reflect the algorithm structures, which can be written schematically as

$$\text{TSRAC}i(k) = \text{TS}(k) + \text{AC}i(1), \qquad i = 1, 2, \text{ or } 3$$

Column A of Fig. 9 shows the domain array for d at nodes along a given search tree branch when solving an arbitrary problem with 4 variables. (The contents of the domains of nonpast variables z_k to z_4 at these nodes are irrelevant for the purposes of Fig. 9.) Column B of the figure shows, for each node, the set of arcs that are ultimately made consistent by the corresponding ACi(1) call of TSRACi. The braces around the arcs in column B emphasize that we are displaying the *set* of arcs made consistent at a node, rather than the actual sequence of arc revisions. AC1(1), AC2(1),

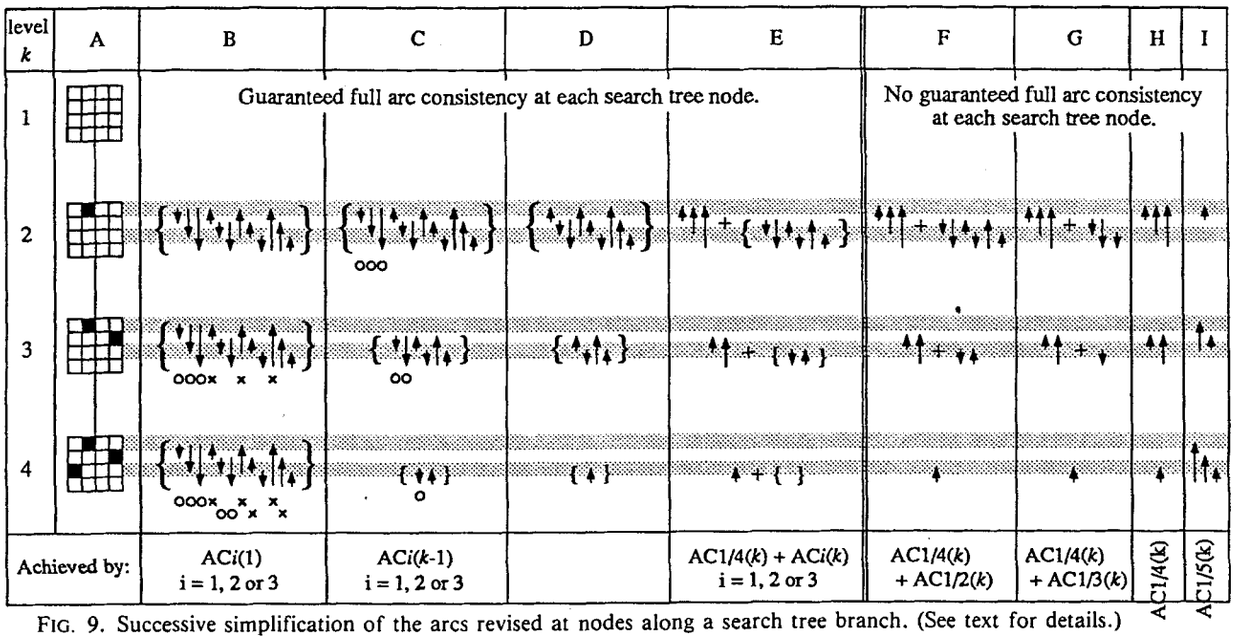


FIG. 9. Successive simplification of the arcs revised at nodes along a search tree branch. (See text for details.)

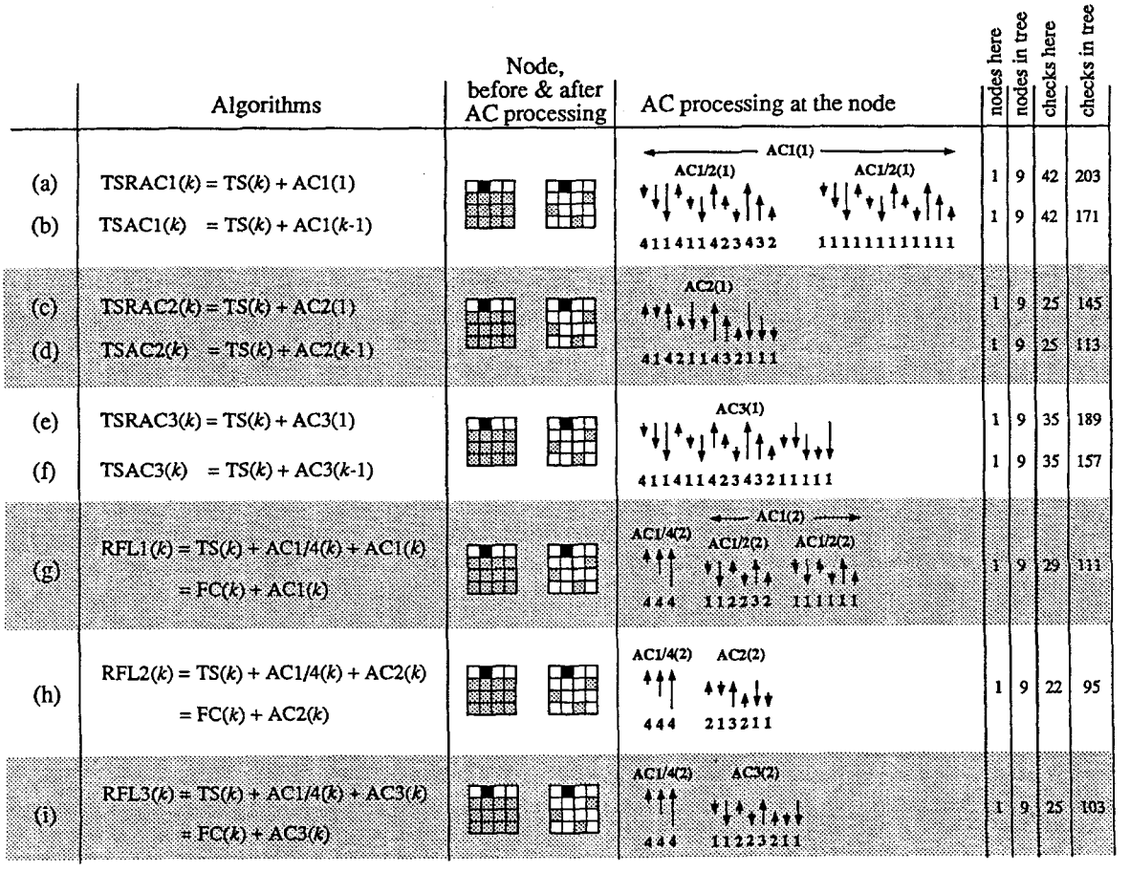


FIG. 10. Processing at the level $k = 2$ node $z_1 = 2$, when solving 4-queens by the nine full arc consistency hybrid algorithms corresponding to columns B, C, and E of Fig. 9.

and AC3(1) all make the same set of arcs consistent, but they do so in generally different order and with varying amounts of multiple revisions of some arcs.

This is seen in Fig. 10, which shows the specific arc consistency processing at the single $z_1 = 2$ node when solving the 4-queens problem by each of the nine hybrid algorithms

of Sect. 5.1. The present three TSRAC*i* algorithms correspond to rows (a), (c), and (e) of the figure. We see there how the AC processing at the node shown, and the associated cost in constraint checks, differs with the AC*i* component used, even though the same set of arcs (as opposed to sequence or multiset of arcs) is revised, and the same final state is achieved at the node.

In Fig. 11, the single node of Fig. 10 is placed, as node C, in the context of the whole left subtree for the 4-queens problem. The conventions used in Figs. 10–12 are essentially the same, as follows. To the right of each node is a sequence of arrows showing the arcs revised and their order of revision at the node. These are grouped, and labeled, according to the AC procedure call to which they correspond. The number of constraint checks performed for each arc revision is shown below the corresponding arrow. The tables at the right in the figures show respectively the number of nodes in the corresponding example and in the whole tree for the problem, and the number of constraint checks performed in the example and in the whole tree. The number of constraint checks in the example is, of course, the sum of the numbers under the corresponding individual arc revision arrows shown.

Figure 10 shows the domain array for d both before and just after the AC processing at the node. To save space, in the search tree traces of Figs. 11 and 12 we give only the domain array for d *after* the AC processing at a node. The domain array *before* the AC processing (which equals that on entry to the node) can, however, be easily inferred in such tree traces. Given the workings of our algorithm in Fig. 8, the domain array for d on entry at a level-k node must be simply the domain array inherited from the node's parent in the tree diagram, since the latter corresponds to d just after AC processing at the parent node. However, the corresponding instantiation for variable z_{k-1} must be added, since this instantiation occurred just before exit from the parent node and just after the state represented by the domain array shown for the parent. The *before* node in Fig. 10 is an example of the above. Figure 3 of Nudel (1983*a*) may also be helpful.

Figure 11*a* shows the left half of the TSRAC1 search tree in solving the 4-queens problem, together with the corresponding statistics, and those for the TSRAC2 and TSRAC3 versions (whose trees are not shown). In the TSRAC1 tree that is shown, only one cycle of the AC1(1) subroutine AC½(1) is carried out at node B. This is because a domain wipe-out is found to occur during that cycle. At node C, on the other hand, two cycles of AC½(1) are executed. No domain wipe-out occurs, but the second cycle is the last since it causes no change. At nodes D and E only single cycles of AC½(1) occur, since even these first cycles cause no change.

From the statistics of Fig. 11*a* we see that, as required, each TSRAC*i* version generates the same number of nodes due to their all achieving the same state of full arc consistency at corresponding nodes. But in terms of constraint checks, there is a difference due to the different AC*i* components used. As expected, the $i = 2$ and $i = 3$ forms of TSRAC*i* are more efficient than the $i = 1$ form, due to the general superiority of the components AC2 and AC3 over AC1, which is made explicit in the case of node C, at lines (a), (c), and (e) of Fig. 10.

5.1.2. Tree search and arc consistency:
TSACi(k) = TS(k) + ACi(k − 1)

As implied by the R in the name TSRAC*i*, those algorithms contain some redundant processing. The reason can be seen by reference to Fig. 9. Under some arcs in column B are crosses and circles. These are respectively the arcs into and out of net nodes 1 to $k-2$. Revision of such arcs at a level-k tree node is redundant because

(*i*) the in-arcs have been made consistent at some earlier level in the tree. Since at that time the target net nodes 1 to $k-2$ had only one domain value, due to instantiation, the arcs into them cannot become inconsistent by subsequent instantiation of z_{k-1} at level $k-1$ and the other arc revisions at level k.

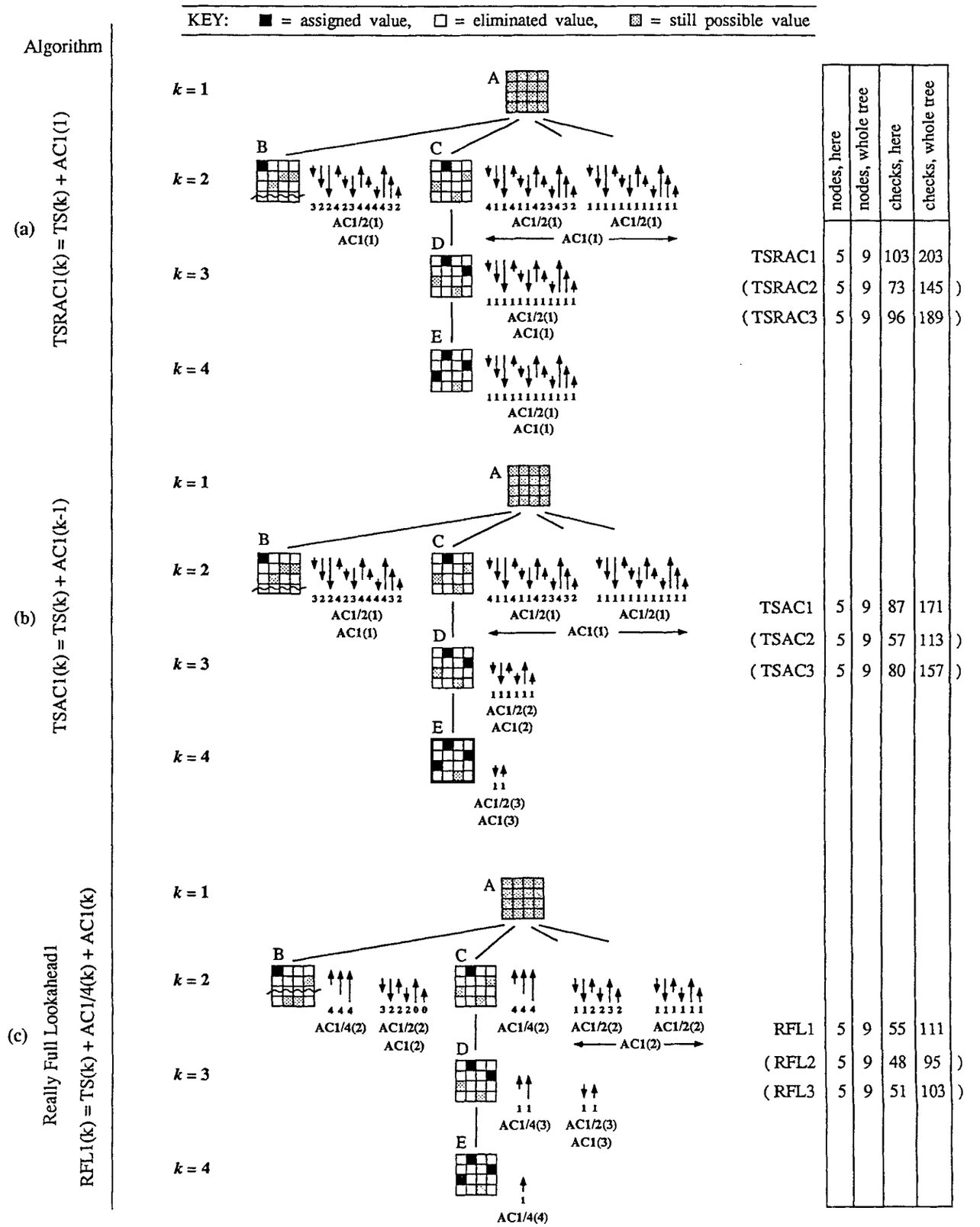
(*ii*) since, by (*i*), the in-arcs are ensured consistent, then so are the out-arcs. This is again because the network nodes 1 to $k-2$ all have only one value, and the out-arc from any net node with one value is consistent if the reverse in-arc is consistent.

Removing the redundant crossed and circled arcs in column B, we get the reduced sets of arcs in column C. Since we have removed all arcs into and out of net nodes 1 to $k-2$ from the network $G_{1:n}$, we are left with the arcs in the subnetwork on nodes $k-1$ to n, which is the subnetwork $G_{k-1:n}$. This can be made arc consistent by the call AC*i*$(k-1)$. Thus we can achieve arc consistency of the whole network $G_{1:n}$ at each level-k tree node by calls AC*i*$(k-1)$, rather than the more expensive calls AC*i*(1) in TSRAC*i*(k) above. Doing this gives us the variants TSAC1, TSAC2, and TSAC3 of our combined algorithm in Fig. 8. These names reflect the corresponding algorithm structure

$$\text{TSAC}i(k) = \text{TS}(k) + \text{AC}i(k-1), \qquad i = 1, 2, \text{ or } 3$$

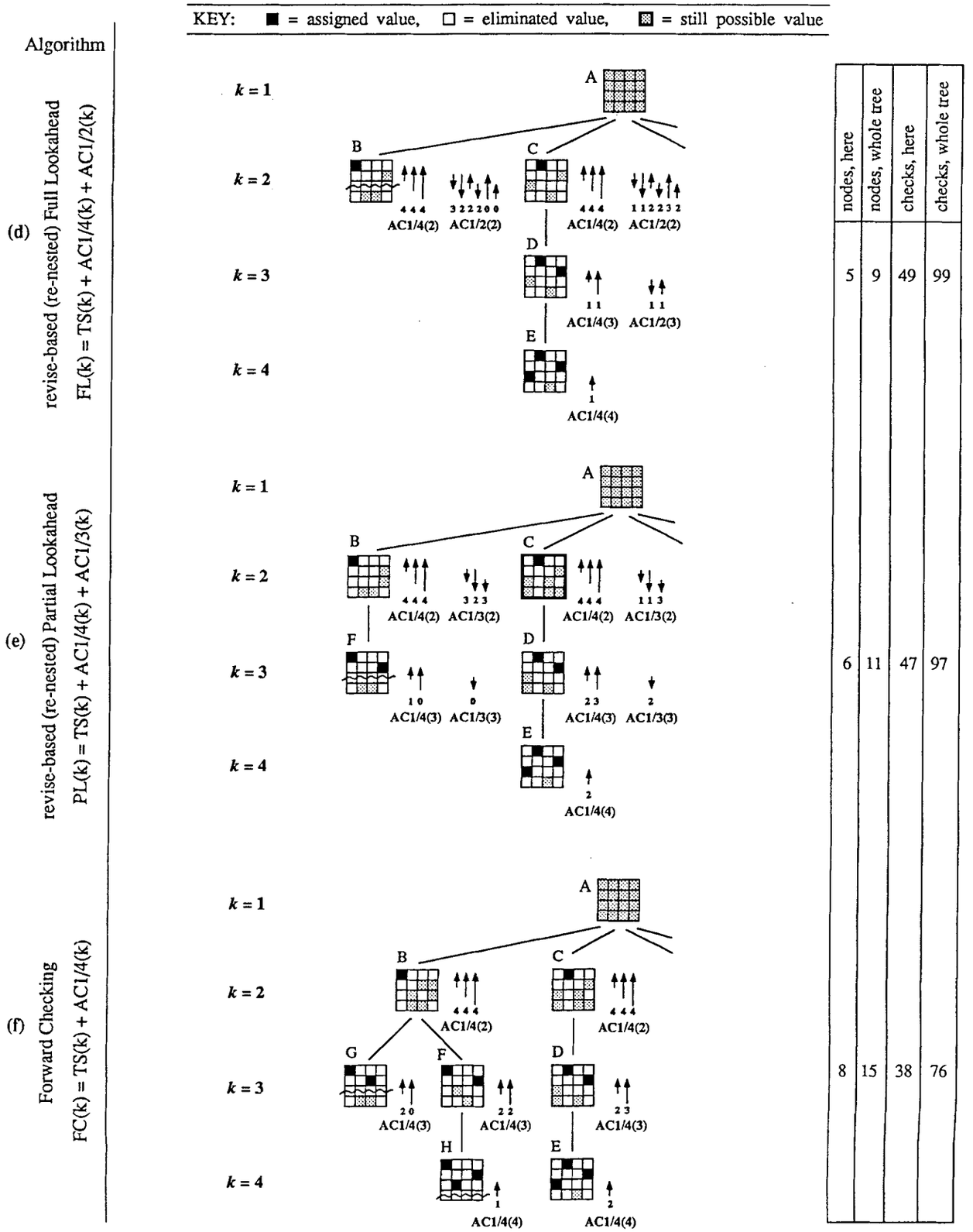
Figure 11*b* shows the left half of the TSAC1 search tree in solving the 4-queens problem, together with the corresponding statistics and those for the TSAC2 and TSAC3 versions (whose trees are not shown). Again, as required, each TSAC*i* version generates the same number of nodes, and the same number as for the three TSRAC*i* above, due to their all achieving the same state of full arc consistency at corresponding nodes. But as seen in the statistics of Figs. 11*a* and 11*b*, the number of checks performed by TSAC*i* is less than TSRAC*i* for a given *i*, due to the simpler arc consistency processing at nodes of the TSAC*i* version. Actually, at corresponding $k = 2$ nodes the processing is the same for TSAC*i* and TSRAC*i* for a given *i*. This is simply because their respective calls AC*i*$(k-1)$ and AC*i*(1) are the same when $k = 2$. It is only at levels $k > 2$ that the TSAC*i* processing is actually less at a node than that of the TSRAC*i* version. These effects can of course be seen in comparing corresponding $k = 2$ nodes and corresponding $k > 2$ nodes in Figs. 11*a* and 11*b*.

As for TSRAC*i* above, the statistics at the right of Fig. 11*b* show the $i = 2$ and $i = 3$ forms of TSAC*i* to be better than the $i = 1$ form because of the general superiority of the components AC2 and AC3 over AC1. An explicit comparison of the difference between the action of these TSAC*i* algorithms at node C can be seen in lines (b), (d), and (f) of Fig. 10. Actually, since node C is a $k = 2$ node, these lines in the figure have the same processing respectively as for lines (a), (c), and (e).



	nodes, here	nodes, whole tree	checks, here	checks, whole tree
TSRAC1	5	9	103	203
(TSRAC2	5	9	73	145)
(TSRAC3	5	9	96	189)
TSAC1	5	9	87	171
(TSAC2	5	9	57	113)
(TSAC3	5	9	80	157)
RFL1	5	9	55	111
(RFL2	5	9	48	95)
(RFL3	5	9	51	103)

FIG. 11. Solving 4-queens by the hybrid algorithms.



	nodes, here	nodes, whole tree	checks, here	checks, whole tree
(d)	5	9	49	99
(e)	6	11	47	97
(f)	8	15	38	76

FIG. 11. (*Continued*)

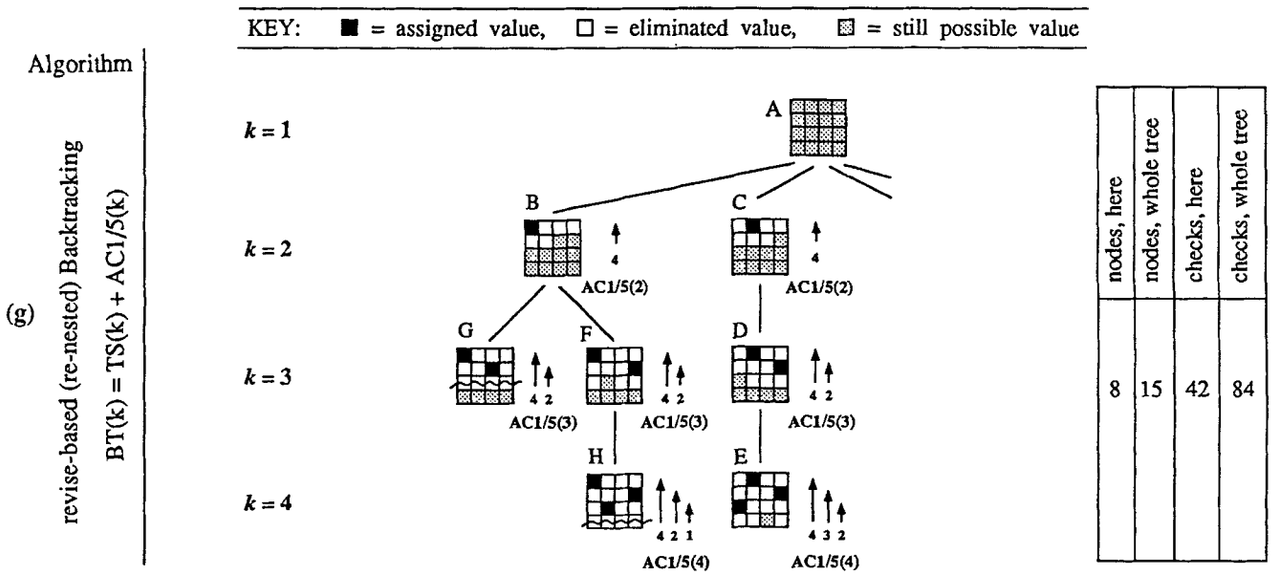
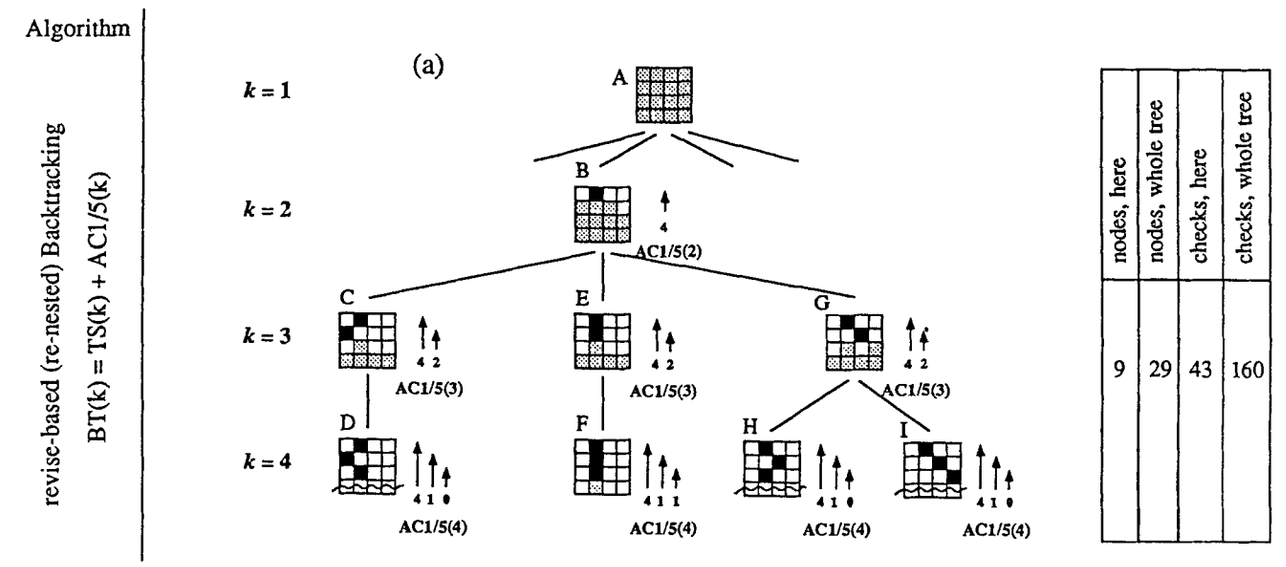


FIG. 11. (*Concluded*)



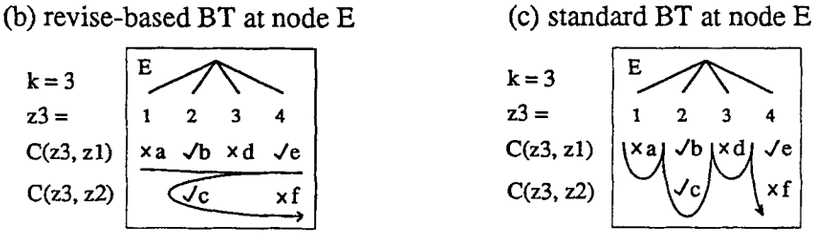


FIG. 12. Solving confused 4-queens by revise-based BT. Compare with trace for standard BT in Fig. 1 (and Fig. 2).

5.1.3. Really Full Lookahead: RFLi(k) = TS(k) + $AC^{1}/_{4}(k)$ + ACi(k) = FC(k) + ACi(k)

The above simplification of the processing at nodes may be taken even further without sacrificing guaranteed full arc consistency. As in column B of Fig. 9, some arcs in column C also have circles under them, denoting still remaining redundant arc revisions. The circled arcs of column C are the out-arcs from net node $k-1$ to net nodes k to n. The reason their revision is redundant is that for each such out-arc the corresponding in-arc is also in the set (of column C) to be made arc consistent. If the in-arc to net node $k-1$ is made consistent, then the out-arc will automatically be consistent since net node $k-1$ has only one value. (This is the same reason as used in point (*ii*) of the previous sec-

tion's argument). Removing the redundant arcs of column C, we obtain the further-reduced arc sets of column D.

The arcs in $G_{k-1:n}$ of column C have thus been reduced in column D to the set of in-arcs to net code $k-1$ coming from net nodes k to n, and the set of arcs on net nodes k to n, i.e., the arcs in the subnetwork $G_{k:n}$. The partition into these two groups (type A and type B arcs respectively) is shown explicitly in column E. Note further that each arc type A need to be revised only once to guarantee its eventual consistency. Subsequent revision of another arc of type A or B cannot undo the arc consistency of a type A arc, because the latter is an in-arc to a single-value net node (the argument used above in point (*i*) of the section on TSAC*i*). Hence we have removed the set braces around the type A arcs in column E, denoting that they are each to be revised only once.

The job of revising each type A arc just once at a level-k tree node is precisely what is achieved by the call AC¼(k) to our earlier partial arc consistency algorithm AC¼. Unlike the type A arcs, the type B arcs may possibly need multiple revisions to achieve the goal of arc consistency for the corresponding subnetwork $G_{k:n}$. This can be achieved by a call to AC*i*(k), $i = 1$, 2, or 3. In conclusion, we can replace the call AC*i*($k-1$) of TSAC*i*(k) above by the more efficient pair of calls AC¼(k) and AC*i*(k), and still achieve arc consistency of the whole network $G_{1:n}$ at all tree nodes at each level k. Doing this gives us the variants RFL1, RFL2, and RFL3 of our combined algorithm in Fig. 8. The corresponding algorithm structures are

$$\begin{aligned} \text{RFL}i(k) &= \text{TS}(k) + \text{AC¼}(k) + \text{AC}i(k) \\ &= \text{FC}(k) + \text{AC}i(k), \qquad i = 1, 2, \text{ or } 3 \end{aligned}$$

The second equality here is because, as we will see later, TS(k) + AC¼(k) is Haralick's Forward Checking algorithm, FC(k). The names RFL*i* stand for Really Full Lookahead*i*, since these algorithms are essentially extensions of Haralick's Full Lookahead, discussed next. The algorithms' structures in terms of FC suggests FCAC1, FCAC2. and FCAC3 as alternative systematic names.

Note that the AC¼(k) call is made before the AC*i*(k) call. This is because revisions of type A arcs made by the former call are more likely to lead to domain wipe-out (and hence early termination of the corresponding path through the tree) than are revisions of type B arcs made by the latter call. This is because the type A arcs are the only ones going to a guaranteed single-value network node[15], viz, network node $k-1$; a single-value target net node offers relatively little chance of support for values in the source node of their connecting arc, and hence increases the chance of a revision that actually removes a domain value in the source node.

Figure 11*c* shows the left half of the RFL1 search tree in solving the 4-queens problem, together with the corresponding statistics and those for the RFL2 and RFL3 versions (whose trees are not shown). Note that for completeness, otherwise scheduled arcs that were not revised due to a preceding domain wipe-out are included, but with a 0 (denoting no checks performed) below them. Again, as required, each RFL*i* version generates the same number of nodes, and the same number as for the three TSRAC*i* and the three TSAC*i* above, due to their all achieving the same state of full arc consistency at corresponding nodes. But as seen in the statistics of Figs. 11*a*–11*c*, there is a successive improvement in the number of checks in going from TSRAC*i* to TSAC*i* to RFL*i* for a given *i*, due to the successive simplification in arc consistency processing at corresponding nodes for these algorithms.

As for the TSRAC*i* and TSAC*i* above, the statistics at the right of Fig. 11*c* show the $i = 2$ and $i = 3$ forms of RFL*i* to be better than the $i = 1$ form because of the general superiority of the corresponding components AC2 and AC3 over AC1. (However, see Sect. 6 and Appendix I regarding the surprising possibility of AC1 being better than AC2 and AC3). An explicit comparison of the difference between the action of these RFL*i* algorithms at node C can be seen in lines (g), (h), and (i) of Fig. 10. Another, more detailed, comparison of the processing by RFL1 and RFL3 at a node appears in Figs. A5 and A6 in Appendix I.

5.2. *Four partial arc consistency hybrids*

The nine hybrid algorithms above were all designed to ensure full arc consistency of the whole constraint network $G_{1:n}$ at each search tree node. We saw that this can be achieved with successively less arc revision as we went from the TSRAC*i* set of algorithms to the TSAC*i* set, then to the RFL*i* set. In this section we maintain the above hybrid algorithm structure and continue the process of reducing the amount of arc consistency attained at the nodes. Four new algorithms are obtained in this way.

Unlike for the reductions of the previous section, however, the reductions here are sufficient to lose us guaranteed full arc consistency of the whole network $G_{1:n}$ at tree nodes. (Full arc consistency *may* nevertheless still be achieved forfuitously at some, or even all, tree nodes, depending on the problem.) Each algorithm does, however, still preserve guaranteed full arc consistency of the subnetwork $G_{1:k}$ at each level-k node. (For brevity, we forgo the proofs in each case. They are similar to those showing that $G_{1:n}$ remains fully arc consistent in the algorithms above). Thus at least at the last level, $k = n$, these algorithms all ensure full arc consistency of the whole network $G_{1:n}$. Since at $k = n$ nodes, all variables but z_n have already been instantiated to only a single value, the full arc consistency of $G_{1:n}$ at such nodes is sufficient to ensure that the solutions output by our tree search shell are indeed all valid solutions. (This corresponds to case (*iii*) of when full arc consistency is sufficient to solve a problem, discussed near the start of Sect. 4.2.)

The nine hybrid algorithms above all generated search trees with the same node structure, since each achieved the same state of full arc consistency at each node. The trees of the following algorithms, however, may have a different node structure due to their achieving only partial arc consistency, of various degrees, at the nodes. In spite of the differences in node structure of the search trees, we will still be talking below of *corresponding* nodes in two trees. By this we mean simply two nodes which correspond to the same set of instantiations of their past variables. Corresponding nodes have the same letter labeling them in the traces of Fig. 11.

The lower the degree of arc consistency achieved at nodes by an algorithm, the more nodes we can expect in the algorithm's tree since less arc consistency processing leaves more

[15] Hence the comment in Sect. 4.1 that the arc revisions performed by AC¼(k) in these hybrid algorithms are always of the specialized type where the target net node has only a single domain value.

values still viable in net nodes (variable domains), and this results in extra descendant tree nodes. This need not necessarily be less efficient, however. Even though there are more nodes in the tree, the extra nodes are a result of less arc consistency processing at each node. More nodes, but with less constraint checks per node, may result in less total checks for the overall tree. The experiments of Sect. 6, and the traces of the present section, will show that further reduction of arc consistency processing is indeed cost-effective, but only up to a point.

5.2.1. Full Lookahead: FL(k) = TS(k) + AC1/4(k) + AC1/2(k) = FC(k) + AC1/2(k)

Our successive simplifications of the previous section, culminated in three algorithms of structure RFL$i(k)$ = TS(k) + AC1/4(k) + AC$i(k)$, i = 1, 2, or 3. Let us concentrate on the i = 1 version, which uses AC1, the most straightforward of the three full arc consistency algorithms AC1, AC2, and AC3 treated in Sect. 4.2. Remember that AC1(k) works by simply making multiple calls to the partial arc consistency algorithm AC1/2(k), until no change occurs for such a call or until a domain wipe-out occurs. Thus we might express AC1(k) schematically as having the structure AC1(k) = AC1/2(k) + AC1/2(k) + ..., and hence express RFL1(k) as having the structure

$$\text{RFL1}(k) = \text{TS}(k) + \text{AC}^1\!/_4(k) + \text{AC}^1\!/_2(k) + \text{AC}^1\!/_2(k) + \ldots$$

This suggests a simplified algorithm, where only a single call is made to AC1/2(k), corresponding to column F of Fig. 9. Accordingly, as already occurred for the AC1/4(k) arcs in column E, the other arcs in column F have now also lost the set braces around them. This denotes that we are simply revising each arc once, rather than necessarily guaranteeing its eventual consistency (which another arc's subsequent revision may now undo). This reduction gives us the variant FL of our combined algorithm in Fig. 8. The corresponding algorithm structure is

$$\begin{aligned}\text{FL}(k) &= \text{TS}(k) + \text{AC}^1\!/_4(k) + \text{AC}^1\!/_2(k)\\ &= \text{FC}(k) + \text{AC}^1\!/_2(k)\end{aligned}$$

We call this FL because it is essentially Haralick and Elliot's (1980) Full Lookahead algorithm. Its above structure in terms of FC suggests the alternative systematic name FCAC1/2. The algorithm first revises the arcs $(f, k-1)$, $k \le f \le n$ (from the nonpast variables to the most recent past variable) and then the arcs $(f1, f2)$, $k \le f1 \ne f2 \le n$ (between nonequal, nonpast variables).

Note that our FL above and the Full Lookahead algorithm of Haralick and Elliot do differ in two respects. The first is the trivial difference of how the algorithms implicitly partition the search tree into nodes (see footnote 7). The second is the more substantial difference that the Look_Future subroutine of Haralick and Elliot's Full Lookahead and the AC1/2 subroutine of our FL correspond to different loop nestings, as explained in Sect. 4.1. As a result, the constraint-check order (but not the final state) is different at corresponding tree nodes.

It is this loop-nesting rearrangement that makes it possible to write FL compactly in terms of revise-based subroutines, and to thus consider it as one of our family of hybrid tree search/arc consistency algorithms. The resulting economy and unity of code makes the present version FL preferable, at least for pedagogical purposes. Which version is best in terms of efficiency is a different question. There is no obvious efficiency reason to prefer one nesting scheme over the other, although it would appear that a difference in the number of constraint checks can occur. Surprisingly, however, all experiments so far have shown no such difference for the two versions, both overall and at each corresponding pair of nodes. (The reader may like to try and show that this must necessarily be the case or to discover counter examples.)

Fig. 11*d* shows the left half of the FL search tree in solving the 4-queens problem, together with the corresponding statistics. The node structure of the FL tree is the same as that of the common tree for the full arc consistency hybrids in Figs. 11*a*–11*c*. However, this is just because for 4-queens, the reduced arc consistency processing done by FL is apparently still sufficient to ensure full arc consistency at each node. In general, since the subroutine AC1/2(k) of FL is not guaranteed to achieve full arc consistency of the subgraph it applies to, there will be more values left in the domains of variables, and hence more descendants per node and thus bigger trees for FL. For 5-queens and higher, we see in Table 3 that the FL tree indeed has more nodes than those of the full arc consistency hybrids, as expected.

Of course, in our 4-queens example, since the FL tree has no more nodes than the RFL1 tree, but has less or equal number of constraint checks per node, the total number of checks is less. This reduction in constraint checks compared to RFL1 is found for other problems, as in Tables 2 and 3, even when the number of nodes does increase over that for RFL1 and the other full arc consistency hybrids. The general decrease[16] in work per node is sufficient to reduce the work summed over all nodes, in spite of the increase in the number of nodes. At least for the problems studied here, the extra work per node of the full arc consistency hybrids is not cost-effective.

5.2.2. Partial Lookahead: PL(k) = TS(k) + AC1/4(k) + AC1/3(k) = FC(k) + AC1/3(k)

Remember from Sect. 4.1 that procedure AC1/3(k) revises a subset of the arcs revised by AC1/2(k). We can thus further reduce the amount of arc consistency achieved at search tree nodes if we replace the call to AC1/2(k) in FL(k) by a call to AC1/3(k). This corresponds to column G of Fig. 9 and gives us the variant PL of our combined algorithm in Fig. 8. The corresponding algorithm structure is

$$\begin{aligned}\text{PL}(k) &= \text{TS}(k) + \text{AC}^1\!/_4(k) + \text{AC}^1\!/_3(k)\\ &= \text{FC}(k) + \text{AC}^1\!/_3(k)\end{aligned}$$

[16]The number of constraint checks at corresponding nodes is not always less for FL than for RFL1, even though the former revises only a subset of the arcs revised by the latter at a node. This is because less arc revisions at an FL node may result in larger filtered domains being inherited at a descendant node, and hence possibly more constraint checks being necessary at the lower node. This is not seen in comparing nodes of our example traces for FL and RFL1. The same phenomenon is seen, however, in comparing nodes D of PL and FL, and may occur at corresponding nodes of any two hybrid algorithms where one algorithm revises a subset of the arcs revised by the other at the nodes. The algorithm revising the arc subsets will generally have less checks per node, but not necessarily at all nodes.

We call this PL because it is essentially Haralick and Elliot's (1980) Partial Lookahead algorithm. Its structure in terms of FC suggests the alternative systematic name FCAC1/3. The algorithm first revises the arcs $(f, k-1)$, $k \leq f \leq n$ (from the nonpast variables to the most recent past variable) and then the arcs $(f1, f2)$ $k \leq f1 < f2 \leq n$ (from the nonpast variables to more-future nonpast variables, rather than, as in FL, from the nonpast variables to any nonequal, nonpast variable).

Note that our PL differs from Haralick and Elliot's Partial Lookahead algorithm in the same two ways that our FL differed from their Full Lookahead. In particular, the loop-nesting is different as discussed in connection with AC1/3 in Sect. 4.1. Again, however, no complexity difference has been detected, although it seems that one should sometimes exist between the two versions of Partial Lookahead.

Figure 11*e* shows the left half of the PL search tree in solving our running 4-queens example, together with the corresponding statistics. Note that corresponding nodes in the PL and FL trees (such as node C) do show different states for the domains after arc consistency processing. As expected, due to the lesser degree of arc consistency attained at nodes of the PL tree, the domains there generally have more still viable (grey) values than corresponding domains in the FL tree, and hence there are extra children nodes and more overall nodes in the PL tree. In spite of the extra nodes for PL, we see that the lower amount of arc consistency, and hence the generally lower amount of constraint checks, per node is sufficient to cause a reduction in the overall number of constraint checks compared to FL. Note the interesting phenomenon, discussed in footnote 16, of there being more checks at node D of the PL tree than of the FL tree, in spite of less arcs being revised.

5.2.3. Forward Checking: FC(k) = TS(k) + AC1/4(k)

Continuing the trend above, we can still further reduce the amount of arc consistency achieved at search tree nodes by retaining only the AC1/4(k) component of PL. This corresponds to column H of Fig. 9 and gives us the variant FC of our combined algorithm in Fig. 8. The corresponding algorithm structure is

$$\text{FC}(k) = \text{TS}(k) + \text{AC}^{1}\!/_{4}(k)$$

We call this FC because it is essentially Haralick and Elliot's Forward Checking algorithm, also studied by McGregor (1979). Its structure suggests the alternative systematic name TSAC1/4. Since it retains only AC1/4(k), FC revises only the arcs $(f, k-1)$, $k \leq f \leq n$, from nonpast variables to the most-recent past variable $k-1$, avoiding the revision of arcs between pairs of nonpast variables done by FL and PL.

As with FL and PL above, our FC differs from Haralick and Elliot's version in how the tree is partitioned into nodes. But unlike with FL and PL, there is no difference in loop nesting between our FC and Haralick and Elliot's version. Figure 11*f* shows the left half of the FC search tree in solving our running 4-queens example, together with the corresponding statistics. We see that the same trend as before is still continuing: the total number of checks is still dropping, due to the reduced number of checks per node, even though the number of nodes increases. This trend of improved efficiency as a result of less AC processing at the nodes is finally reversed with our next algorithm. It takes us full circle, being a version of Backtracking, the first algorithm treated above.

5.2.4. Revise-based Backtraking: BT(k) = TS(k) + AC1/5(k)

Somewhat surprisingly, even the standard Backtracking algorithm, when slightly rearranged, can be formulated as a tree search/arc consistency hybrid. The new form, though arrived at independently here, was subsequently found to have been developed by McGregor (1979, p. 241). McGregor did not, however, identify his hybrid algorithm as being a form of plain backtracking. We first note that our original version of BT from Sect. 3.1 may be rewritten as follows, by interchanging the nesting of its two FOR-loops.

```
PROCEDURE BT(k, VAR z);
dk ← (1 2 ... m[k]); {initialize domain for variable zk.}
FOR p ← 1 TO k-1 WHILE dk ≠ empty DO
  BEGIN
  dk__copy ← dk
  FOR z[k] ← each element of dk__copy DO
    IF not check(k, z[k], p, z[p]) THEN dk ← dk - z[k];
  END
IF dk ≠ empty THEN
  FOR z[k] ← each element of dk DO
    IF k = n THEN output(z) ELSE BT(k + 1, z)
END;
```

The first statement here denotes an assignment to variable dk of the list of integers 1 to m[k], being the domain of problem variable z_k. (m[k] = m_{z_k}, the domain size of variable z_k, as for the algorithms of Sect. 3.) At a level-k node, the original BT checked an instantiation for the current variable z_k against the instantiations for each past variable z_1 to z_{k-1}, and then (in general, after returning from a recursive child generation) repeated such checks for a different instantiation of z_k. The new version of BT, on the other hand, checks *all* domain values of z_k against the single instantiation for z_1, then all *surviving* domain values for z_k against the instantiation for z_2, repeating till it has checked all surviving values for z_k against the instantiation for z_{k-1}.

This change of ordering means that no instantiations are made nor children nodes generated until all surviving z_k values have been determined. Thus it is not a truly *depth-first* formulation as was that of Sect. 3.1, but is rather what Horowitz and Sahni (1978, chap. 7) have called *D-first* (*D-search* actually). Nodes are still generated in the same order but the order of processing at a given node is changed, and if only one solution is sought then the new formulation may waste some effort in unnecessarily finding more than one viable value for z_k. However, this will only be the case for the n nodes actually on the search-tree branch leading to the first solution found and will thus usually not be significant. In any case, when all solutions are of interest, as we are assuming throughout, then the two formulations are equivalent in all respects except the order in which checks are done, and in which recursion is interleaved, at a node.

The above loop-interchange has allowed us to factor the processing at a node into a constraint-checking part followed by an instantiation and child-generation part (rather than interleaving the two). The constraint-checking part can be seen as a succession of calls to the revise procedure of Sect. 4.1, embedded in our earlier tree search shell TS(k). In particular, arcs (k, p) for $1 \leq p < k$ are successively revised. That is, arcs from the current variable to all past variables are revised, corresponding to column I of Fig. 9. This is precisely what is achieved by a call AC1/5(k) to

TABLE 2. Number of constraint checks (and nodes, in parentheses) for solving confused q-queens

Algorithm	q / No. of solutions	3 / 9	4 / 6	5 / 7	6 / 8	7 / 9	8 / 10	9 / 11	10 / 12
BT (TSAC1/5)		41(11)	160(29)	332(47)	590(69)	949(95)	1 428(125)	2 042(159)	2 810(197)
BJ		41(11)	139(27)	288(44)	509(65)	816(90)	1 225(119)	1 747(152)	2 399(189)
BM		29(11)	90(29)	192(47)	346(69)	563(95)	856(125)	1 234(159)	1 710(197)
FC (TSAC1/4)		29(11)	90(23)	188(35)	334(49)	537(65)	808 (83)	1 154(103)	1 586(125)
PL (FCAC1/3)		37(11)	117(17)	270(27)	525(39)	915(53)	1 482 (69)	2 266 (87)	3 316(107)
FL (FCAC1/2)		43(11)	146(17)	345(27)	688(39)	1 222(53)	2 014 (69)	3 125 (87)	4,638(107)
RFL1 (FCAC1)		43(11)	162(17)	393(27)	792(39)	1 412(53)	2 326 (69)	3 601 (87)	5 326(107)
RFL2 (FCAC2)		43(11)	158(17)	392(27)	806(39)	1 439(53)	2 422 (69)	3,746 (87)	5 622(107)
RFL3 (FCAC3)		43(11)	146(17)	347(27)	696(39)	1 241(53)	2 052 (69)	3 190 (87)	4 742(107)
TSAC1		96(11)	367(17)	853(27)	1 681(39)	2 954(53)	4 825 (69)	7 427 (87)	10 950(107)
TSCA2		56(11)	194(11)	466(27)	938(39)	1 645(53)	2 732 (69)	4 180 (87)	6 218(107)
TSAC3		68(11)	260(17)	654(27)	1 358(39)	2 468(53)	4 145 (69)	6 514 (87)	9 774(107)
TSRAC1		136(11)	509(17)	1 195(27)	2 399(39)	4 308(53)	7 175 (69)	11 249 (87)	16 852(107)
TSRAC2		88(11)	300(17)	760(27)	1 596(39)	2 927(53)	4 998 (69)	7 906 (87)	12 012(107)
TSRAC3		102(11)	372(17)	958(27)	2 030(39)	3 768(53)	6 433 (69)	10 266 (87)	15 598(107)

our partial arc consistency procedure AC1/5 (or Check_Backward) of Sect. 4.1. Thus our rearranged BT can be expressed as a hybrid tree search/arc consistency algorithm with structure

$$BT(k) = TS(k) + AC1/5(k)$$

corresponding to variant BT of the combined algorithm in Fig. 8. The structure suggests the alternative systematic name of TSAC1/5 for this revise-based version of BT. As with AC1/4(k) in all earlier hybrid algorithms, note the specialized use of revise by AC1/5(k) here. Due to the instantiation of variables at ancestor tree nodes, all the arcs revised by AC1/5(k) or AC1/4(k) have target net nodes with only one domain value. This was also noted in Sect. 4.1 where the AC procedures were first introduced.

A trace of our revise-based BT solving our running 4-queens example appears in Fig. 11g. We see that though the above trend of successively more nodes (or no less nodes) is continuing, the trend of successively less constraint checks has been reversed. Thus with the previous algorithm FC we had reached the limit of the usefulness of reducing the degree of arc consistency attained at tree nodes.

Actually, it is not really obvious that BT achieves a lower degree of arc consistency at a node than the earlier algorithms, since unlike for our previous succession of algorithms, the arcs revised by BT at a given node are not simply a subset of those revised by the earlier algorithms. (This is seen clearest in Fig. 9.) However, as seen in Tables 2 and 3, we find empirically that BT does generate more (or the same number of) nodes than even FC, and we take this as indicating that BT in effect is attaining the lowest degree of arc consistency of all our hybrid algorithms.

Besides comparing traces of revise-based BT with those of the other hybrid algorithms as in Fig. 11, it is instructive to similarly compare the two forms of BT. Since regular BT was traced on *confused* 4-queens in Fig. 1 (for the reason given in footnote 4), we include in Fig. 12 a trace of revise-based BT on the confused version of the problem. As required, the trees in Figs. 1 and 12 are essentially the same, with the same number of nodes and constraint checks. However, the constraint check order at a node differs in general. For example, the two arrows beside node E in Fig. 12a indicate that arc (3 1) is first revised then arc (3 2). This corresponds to constraint checks in the order a, b, d, e (the revision of arc (3 1)) then c, f (the revision of arc (3 2)), as shown in Fig. 12b. Regular BT, however, checks in the order a, b, c, d, e, f, as shown in Fig. 12c.

Schematically, we can say that regular BT does checking at a node in column-wise order or "vertically" and revise-based BT does it in row-wise order or "horizontally." (The same kind of difference exists between our revise-based versions of FL and PL above and those of Haralick and Elliot (1980), due to the analogous loop-nesting interchanges involved.) In terms of interleaving recursion, we have in our Fig. 12 example that regular BT generates node F after check c and before check d, whereas (as with all hybrid algorithms above) revise-based BT completes all checks at the node before recursing to generate subnodes. (Unlike the check-order difference above, this interleaving difference does not also exist between Haralick and Elliot's and our version of FL or PL.)

6. Empirical comparison and discussion

We have now completed our presentation of 15 algorithms for solving constraint satisfaction problems. Some of these (Sect. 3) were what we called tree search algorithms and some (Sect. 5) were hybrids of a tree search shell with various parameterized arc consistency procedures (Sect. 4) applied at the search tree nodes. The parameterization of our arc consistency procedures is in contrast to the usual practice in presenting consistency algorithms (Mackworth 1977a; Montanari 1974; Freuder 1978; Mohr and Henderson 1986). Without parameterization the implication, whether intended or not, is either that consistency algorithms are adequate in themselves to solve a csp or, if not, that they be used only for preprocessing before applying some tree search algo-

TABLE 3. Number of constraint checks (and nodes, in parentheses) for solving regular q-queens

Algorithm	q / No. of solutions	3 / 0	4 / 2	5 / 10	6 / 4	7 / 40	8 / 92	9 / 352	10 / 724
BT (TSAC1/5)		17(6)	84(15)	405(44)	2 016(149)	9 297(512)	46 752(1965)	243 009(8042)	1 297 558(34815)
BJ		17(6)	84(15)	405(44)	1 864(147)	8 309(489)	41 862(1869)	219 997(7742)	1 131 942(33000)
BM		17(6)	76(15)	276(44)	944(149)	3 236(512)	12 308(1965)	50 866(8042)	220 052(34815)
FC (TSAC1/4)		17(6)	76(15)	282(44)	964(127)	3 338(424)	13 024(1633)	55 326(6680)	242 174(27109)
PL (FCAC1/3)		17(4)	97(11)	485(40)	1 703 (79)	6 511(284)	25 882 (977)	112 327(4014)	496 455(15005)
FL (FCAC1/2)		17(4)	99 (9)	598(40)	2 095 (51)	8 942(248)	35 323 (777)	153 455(3144)	661 017(10737)
RFL1 (FCAC1)		17(4)	111 (9)	915(38)	2 744 (41)	12 009(232)	42 923 (677)	185 030(2786)	815 599 (9085)
RFL2 (FCAC2)		17(4)	95 (9)	595(38)	1 957 (41)	8 781(232)	33 765 (677)	148 893(2786)	637 448 (9085)
RFL3 (FCAC3)		17(4)	103 (9)	636(38)	2 101 (41)	9 320(232)	35 999 (677)	157 222(2786)	677 213 (9085)
TSAC1		29(4)	171 (9)	1 359(38)	3 622 (41)	18 405(232)	69 179 (677)	309 346(2786)	1 321 662 (9085)
TSAC2		19(4)	113 (9)	677(38)	2 093 (41)	9 521(232)	35 967 (677)	157 801(2786)	668 108 (9085)
TSAC3		29(4)	157 (9)	901(38)	2 850 (41)	13 285(232)	51 188 (677)	224 812(2786)	960 552 (9085)
TSRAC1		29(4)	203 (9)	1 913(38)	4 624 (41)	29 829(232)	121 881 (677)	613 796(2786)	2 692 076 (9085)
TSRAC2		19(4)	145 (9)	1 131(38)	2 883 (41)	17 799(232)	72 171 (677)	362 421(2786)	1 558 494 (9085)
TSRAC3		29(4)	189 (9)	1 387(38)	3 704 (41)	22 143(232)	90 924 (677)	449 484(2786)	1 949 272 (9085)

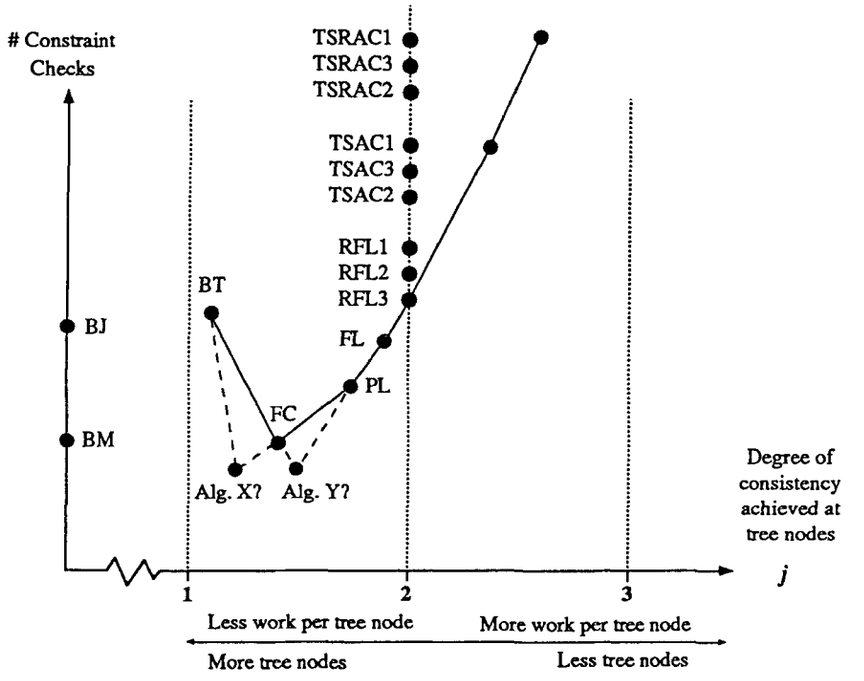


FIG. 13. Schematic plot of the complexities of our 15 CSP algorithms.

rithm. With parameterization, consistency algorithms may be hybridized with a tree search shell to allow arc consistency processing on the subproblem corresponding to each individual search tree node.

A combined algorithm for our hybrid algorithms appeared in Fig. 8. The structure of each hybrid was given schematically in Table 1 in the form TS + ACi_1 or TS + ACi_1 + ACi_2. The ACi are either one of the full arc consistency algorithms that Mackworth (1977*a*) has called AC1, AC2, and AC3, or one of the partial arc consistency algorithms AC1/5, AC1/4, AC1/3, and AC1/2 introduced above, being essentially subroutines due to McGregor (1979) and Haralick and Elliot (1980). (Note that there are other full arc consistency algorithms of interest, such as AC4 (Mohr and Henderson 1986) and DEE (Gaschnig 1978, 1979), but we have not used these in making hybrids.)

We saw that the tree search/arc consistency hybrid structure applied even to the prototypical tree search algorithm, Backtracking, when the order of its two nested loops is switched. Similarly, by rearranging the loop nesting in two of Haralick and Elliot's algorithms, Partial Lookahead and Full Lookahead, they also were able to be expressed in this

common hybrid form. In this section we compare the above algorithms (tree search and hybrid) empirically and discuss the implications for future work.

We use for our experiments the q-queens problems and the confused q-queens problems (Sect. 2) for $3 \leq q \leq 10$. Tables 2 and 3 show the results for these two problem types. (Table 3 overlaps somewhat with Table 1 in Haralick and Elliot (1980).) Note that in all hybrid algorithms that use AC2 and AC3 as a component (algorithms RFLi, TSACi, and TSRACi, $i = 2, 3$), we are using the queue-based version of AC2 and AC3 whose code appears above, rather than the stack-based, or any other, version. The general relationship between the algorithms' efficiencies for a given problem instance (that is, for a given column of Table 2 or Table 3) is summarized schematically in Fig. 13. It will be useful to keep this figure in mind in reading the following, which is a discussion of our experimental results and of possible directions for future research.

1. Our results show that the nine full arc consistency hybrids (RFLi, TSACi, and TSRACi, $i = 1, 2, 3$) always generate the same number of nodes for a given csp. This is as required, because all these algorithms achieve the same state of full arc consistency at each tree node and hence generate search trees with the same node structure.

2. The other algorithms all generate more nodes than the full arc consistency hybrids. In the case of the other hybrid algorithms, this is because, due to the incomplete degree of arc consistency achieved, more domain values are left surviving for the variables (net nodes) at a tree node and this gives rise to more children nodes. As the degree of arc consistency at the tree nodes decreases, the total number of nodes in the tree increases. Actually, even a partial arc consistency algorithm may fortuitously achieve full arc consistency for some, or even all, nodes of a tree. This is why, in Table 2, FL and even PL achieve the same minimal number of nodes as do the full arc consistency hybrids. This, however, is of course not guaranteed to occur, as seen in Table 3. As required, BM always has the same number of nodes as BT, while BJ has less (or at least no more).

3. Amongst the nine full arc consistency hybrids, for a given i, the RFLi form is better than the TSACi form, which is better than the TSRACi form. This is expected from the successive removal of redundant arc revisions, without sacrifice of guaranteed full arc consistency at the nodes, in going from TSRACi to TSACi to RFLi (as discussed in connection with Fig. 9).

4. Within a given triple (the $i = 1$, 2 or 3 forms) of these nine algorithms, the $i = 2$ and $i = 3$ forms are generally better than the $i = 1$ form, as expected from the way AC2 and AC3 refine the brute-force approach to arc consistency taken by AC1. Surprisingly however, this is not always the case. This is seen in the RFLi rows of Table 2 for $q \geq 6$. There we see that RFL1 is better than RFL2, and hence that AC1 is better than AC2 at some tree nodes. Presumably in other cases, AC1 may be better than AC3 also. These possibilities have apparently not previously been noted in the literature. Appendix I discusses this in more detail.

5. Comparing corresponding $i = 2$ and $i = 3$ amongst the nine full arc consistency algorithms, we see that generally the $i = 2$ form is better, indicating that AC2 is better than AC3. Sometimes, however, AC3 may be better, as seen in comparing the RFL2 and RFL3 data of Table 2. Of course, our data is for the specific arc revision orderings assumed by our versions of AC2 and AC3 above (initial lexographic order of arcs, with lists maintained as queues during processing). Other orderings may change the relative ranks of AC2 and AC3, and hence of corresponding $i = 2$ and $i = 3$ form algorithms. Arc-revision ordering is a potential source of significant efficiency improvement and deserves more study.

6. In our hybrid algorithms, less arc consistency per tree node means more tree nodes, but may also mean less constraint checks over the whole tree. This is because even though there are more nodes, there are less checks per node and hence possibly less checks per tree. (Number of checks, not nodes, is the more meaningful measure of complexity, since for our problems each node besides the root corresponds to at least one check.) This overall reduction of checks per tree does indeed occur as we decrease the degree of arc consistency. Both Tables 2 and 3, and the schematic plot of Fig. 13, show that the break-even point amongst our hybrid algorithms occurs for algorithm FC, after which the number of checks rises again for BT. FC is in fact the best algorithm in Table 2, and very nearly the best in Table 3, being beaten slightly by BM.

Thus in hybrid algorithms it does not necessarily pay to pursue much arc consistency at the tree nodes. And this is a strengthening of the observation by other researchers that it does not in general pay to pursue degrees of j-consistency (Freuder 1978) higher than $j = 2$ at the tree nodes (remember that 2-consistency is arc consistency, 3-consistency is path consistency). Figure 13 also indicates this deterioration of efficiency in pursuing higher degrees of j-consistency for $j > 2$.

7. There are of course algorithms conceivable with degrees of arc consistency between BT and FC or between FC and PL. The true minimum complexity (maximum efficiency) may very well occur for one of these. This has in fact led us to the discovery of a new hybrid algorithm better than FC, corresponding to Algorithm Y in Fig. 13. It revises at a tree node only those arcs $(i\,j)$ whose target net node j has only a single domain value left. Note that most of our hybrid algorithms above call AC¼(k) at a level-k node, and all arcs revised by this call are of this singleton-target-node type.

Revision of arcs $(i\,j)$ which have target nodes j with only one value has a triple advantage: (*i*) filtering of source node i is more likely to occur, (*ii*) arc $(i\,j)$ once revised need not be revised again, and (*iii*) arc $(j\,i)$ is implicitly revised when $(i\,j)$ is revised. The arc processing at a tree node in our new algorithm is orchestrated as for AC3, but modified to only consider singleton-target-node arcs. The modified AC3 also differs in that it remembers which arcs have been revised along a branch of the tree, so as not to revise again an arc already revised, explicitly or implicitly, at an ancestor tree node. Each branch through the search tree thus corresponds to one directed-arc revision for each undirected arc in the overall constraint graph, $\binom{n}{2}$ arc revisions in the case of complete graphs on n nodes. We tentatively call this algorithm TSSTAC3 for *Tree Search + Singleton Target Node AC3*. More details will be given in a future paper.

8. It is interesting to note that in Table 3 the trends seem to indicate that every algorithm in the table (even the grossly inefficient TSAC1 and TSRAC1) is better than traditional BT for large enough q. In the examples of Table 2, however, BT seems to be holding its own except against BJ, BM, and FC. This variation with q, and the different effects in the

two tables, is no doubt due at least in part to the change in constraint looseness that occurs with q, and the different way it changes for the two problem classes. We saw in Sect. 2 that looseness increases with q for regular q-queens problems, but decreases with q for confused q-queens problems. How exactly this affects the relative ranking of our algorithms as a function of q is far from obvious, however. Only mathematical analyses can really clarify such issues (see below).

9. The above point generalizes to the observation that the ranking of CSP algorithms depends on the problem instance being solved. And even more generally, any empirical study such as that here, or those in Gaschnig (1978, 1979), Haralick and Elliot (1980), McGregor (1979), and Nadel (1986), must always be taken with a grain of salt because of the inevitably limited nature of the problem set used. In our case, remember that both the q-queens and the confused q-queens problems used here are *complete*, *binary* CSP instances, for which, moreover, each domain size m_i and the number of variables n are all equal to the same value q (see Sect. 2). Moreover, specific instantiation orders, constraint-check orders, and arc-revision orders were used by the algorithms. Only mathematically derived complexity expressions can serve as a truly general basis for comparing these algorithms for arbitrary problem instances and processing orders. Mathematical results of the form in Haralick and Elliot (1980), Mackworth and Freuder (1985), Nadel (1986), and Nudel (1983*a*, *b*) would thus be desirable for the wider range of algorithms considered here. The notion of *instance-specificity* or *precision* of such analytic results, so that they capture the variation of complexity as a function of individual problem instance, is discussed in Nadel (1986, 1988*b*, *c*) and Nudel (1983*a*).

10. The hybrid algorithms above were all subsumed by the meta-algorithm of Fig. 8. Actually, this meta-algorithm could be further generalized to subsume the whole spectrum of full and partial arc consistency hybrid algorithms, which achieve j-consistency for $1 \leq j \leq 2$ at the tree nodes. An analysis of the resulting algorithm would serve as a simultaneous analysis of all its subsumed algorithms, including those here. The same holds for the next level of generalization of the meta-algorithm to allow j-consistency for $1 \leq j \leq n$ at the tree nodes. Whatever the range of j allowed, an analysis of such a "continuous-j" meta-algorithm provides a convenient theoretical basis for choosing the best j-consistency/tree search hybrid from a continuum of possible j values. This would be done by finding the j value that minimized the complexity of the meta-algorithm as a function of j, for the problem of interest.

Such a theory-based and instance-specific approach to decision-making has already been used in deciding on the best search order (Nadel 1986; Nudel 1983*a*) and on the best representation (Nadel 1988*b*) in solving constraint satisfaction problems. It has also been used (Nudel 1983*a*, *b*) in deciding on the best from a small number of algorithms. It has not yet been used, as proposed here, in deciding on the best from a parameterized continuum of algorithms.

11. The hybrid algorithms we have studied, or proposed above, apply the same j-consistency procedure(s) at all nodes of the search tree. If efficiency can benefit from a good choice for a common degree of j-consistency at all the nodes of the tree, then all the more so if this optimization is allowed separately for each level or even for each individual node (assuming that the extra decision-making cost does not outweigh the extra cost saved).

Actually, we already have a special case of this in our above algorithms' avoidance of any arc consistency processing at the root node. Such processing is not generally cost-effective, as discussed near the start of Sect. 5. Gaschnig's DEELEV(i) algorithm (1979) extends this notion, avoiding arc consistency processing till level i. However, the switch-on level, i, must still be decided by the user. Algorithms that dynamically decide (level-wise or node-wise) what degree of j-consistency processing to do where deserve more consideration. A mathematical analysis should be useful in guiding this kind of decision-making. Basically, the decision should depend on the chances of attaining useful filtering. This in turn depends on what set of constraints are involved, their tightness, and how large are the current domains of the variables to be filtered (source net nodes) and of the variables they are to be filtered against (target net nodes).

12. The hybrid algorithms studied above all combined j-consistency processing (for $j = 2$) with the simple search tree mechanism of Backtracking. Combining j-consistency with Backjump or Backmark should also be possible, as suggested by Gaschnig (1979, p. 172). And Backmark and Backjump may themselves perhaps be combined, as suggested in Sect. 3.3 above. Such algorithms deserve attention.

The above CSP algorithms, and suggested improvements, far from exhaust all possibilities. Seidel (1981) has developed an algorithm that is apparently of a totally new type. Other important new directions have also been taken by Dechter and Pearl (1987, 1988), Dechter and Dechter (1987), and Dechter (1986, 1987*a*, *b*). And parallel approaches to solving constraint satisfaction problems are opening up new possibilities (Freuder and Quinn 1985; McCall *et al.* 1985; Kasif 1986). Besides improving algorithms, considerable attention has also been given to formulating and solving alternative versions of the problem. Fuzzy, probabilistic, inexact, and weighted versions of CSP have been studied (Shapiro and Haralick 1981; Faugeras and Berthod 1981; Rosenfeld *et al.* 1976). The latter work is particularly relevant for machine vision because the image being analyzed usually is noisy to some extent. Thus with (*i*) finding new applications, (*ii*) understanding existing algorithms better, both empirically and theoretically, (*iii*) developing new algorithms, and (*iv*) generalizing the problem and its algorithms, research on the Constraint Satisfaction Problem will no doubt remain a central endeavor in artificial intelligence for considerable time to come.

Acknowledgements

Many thanks to Alan Mackworth and Robert Haralick for helpful discussions concerning some of the above algorithms, and also to the reviewers for suggesting useful improvements.

BITNER, J.R., and REINGOLD, E. 1985. Backtrack programming techniques. Communications of the ACM, **18**: 651-656.

BRUYNOOGHE, M., and PEREIRA, L.M. 1984. Deduction revision by intelligent backtracking. *In* Implementations of Prolog. *Edited by* J.A. Campbell. Ellis Horwood, Chichester, England. pp. 194-215.

DAVIS L.S., and ROSENFELD, A. 1981. Cooperating processes for low-level vision: a survey. Artificial Intelligence (Special Issue on Computer Vision), **17**: 245-263.

DECHTER, A., and DECHTER, R. 1987. Removing redundancies in constraint graphs. Proceedings of the 6th National Conference on Artificial Intelligence, Seattle, WA, pp. 105–109.

DECHTER, R. 1986. Learning while searching in constraint satisfaction problems. Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, PA, pp. 178–183.

———1987*a*. A constraint-network approach to truth-maintenance. Technical Report R-870009, Cognitive Systems Laboratory, Computer Science Department, University of California at Los Angeles, Los Angeles, CA.

———1987*b*. An integrated strategy for improved backtrack. Technical Report R-77, Computer Science Department, University of California at Los Angeles, Los Angeles, CA.

DECHTER, R., and PEARL, J. 1987. The cycle-cutset method for improving search performance in AI applications. Proceedings of the Third IEEE Conference on AI Applications, Orlando, FL.

———1988. Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence, **34**(1): 1–38. Also *in* Search in artificial intelligence. *Edited by* L. Kanal and V. Kumar. Springer-Verlag, New York, NY. 1988.

DEKLEER, J. 1986. An assumption-based TMS. Artificial Intelligence, **28**: 127–162.

DOYLE, J. 1979. A truth maintenance system. Artificial Intelligence, **12**: 231–272.

EASTMAN, C. 1972. Preliminary report on a system for general space planning. Communications of the ACM, **15**: 76–87.

FAUGERAS, O.D., and BERTHOD, M. 1981. Improving consistency and reducing ambiguity in stochastic labeling: an optimization approach. IEEE Transactions on Pattern Analysis and Machine Intelligence, **PAMI-3**(4): 412–424.

FIKES, R.E. 1970. REF-ARF: a system for solving problems stated as procedures. Artificial Intelligence, **1**: 27–120.

FOWLER, G., HARALICK, R., GRAY, F.G., FEUSTEL, C., and GRINSTEAD, C. 1983. Efficient graph automorphism by vertex partitioning. Artificial Intelligence (Special Issue on Search and Heuristics), **21**(1 and 2): 245–269. Also *in* Search and heuristics, North-Holland, Amsterdam, The Netherlands.

FREUDER, E.C. 1978. Synthesizing constraint expressions. Communications of the ACM, **21**: 958–966.

———1982. A sufficient condition for backtrack-free search. Journal of the ACM, **29**(1): 24–32.

FREUDER, E.C., and QUINN, M.J. 1985. Parallelism in an algorithm that takes advantage of stable sets of variables to solve constraint satisfaction problems. Technical Report 85-21, Department of Computer Science, University of New Hampshire. Durham, NH.

GASCHNIG, J. 1974. A constraint satisfaction method for inference making. Proceedings of the 12th Annual Allerton Conference on Circuit System Theory, University of Illinois, Chicago, IL, pp. 866–874.

———1977. A general Backtracking algorithm that eliminates most redundant tests. Proceedings of the 5th International Joint Conference on Artificial Intelligence, Massachusetts Institute of Technology, Cambridge, MA, p. 457.

———1978. Experimental case studies of Backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. Proceedings of the 2nd Biennial Conference of the Canadian Society for Computational Study of Intelligence, Toronto, Ont., pp. 268–277.

———1979. Performance measurement and analysis of certain search algorithms. Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

GOLOMB, S.W., and BAUMERT, L.D. 1965. Backtrack programming. Journal of the ACM, **12**: 516–524.

HARALICK, R.M., and ELLIOT, G.L. 1980. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, **14**: 263–313.

HARALICK, R.M., and SHAPIRO, L.G. 1979. The consistent labeling problem: part I. IEEE Transactions on Pattern Analysis and Machine Intelligence, **PAMI-1**(2): 173–184.

———1980. The consistent labeling problem: part II. IEEE Transactions on Pattern Analysis and Machine Intelligence, **PAMI-2**(3): 193–203.

HARALICK, R.M., DAVIS, L.S., and ROSENFELD, A. 1978. Reduction operations for constraint satisfaction. Information Sciences, **14**: 199–219.

HOROWITZ, E., and SAHNI, S. 1978. Fundamentals of computer algorithms, Computer Science Press, Inc., Rockville, MD.

KASIF, S. 1986. On the parallel complexity of some constraint satisfaction problems. Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, PA, pp. 349–353.

KUMAR, V., and LIN, Y. 1988. A data-dependency-based intelligent backtracking for Prolog. Journal of Logic Programming, pp. 165–181.

MACKWORTH, A.K. 1977*a*. Consistency in networks of relations. Artificial Intelligence, **8**: 99–118.

———1977*b*. On reading sketch maps. Proceedings of the 5th International Joint Conference on Artificial Intelligence, Massachusetts Institute of Technology, Cambridge, MA, pp. 598–606.

———1987. Constraint satisfaction. *In* Encyclopedia of artificial intelligence. *Edited by* S.C. Shapiro. Wiley, New York, NY.

MACKWORTH, A.K. and FREUDER, E.C. 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. Artificial Intelligence, **25**: 65–74.

MCCALL, J.T., TRONT, J.G., GRAY, F.G., HARALICK, R.M., and MCCORMICK, W.M. 1985. Parallel computer architectures and problem solving strategies for the consistent labeling problem. IEEE Transactions on Computers, **C-34**(11): 973–980.

MCGREGOR, J. 1979. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. Information Sciences, **19**: 229–250.

MOHR, R., and HENDERSON, T.C. 1986. Arc and path consistency revisited. Artificial Intelligence, **28**: 225–233.

MONTANARI, U. 1974. Networks of constraints: fundamental properties and applications to picture processing. Information Sciences, **7**: 95–132.

MONTANARI, U., and ROSSI, F. 1988. Fundamental properties of networks of constraints: a new formulation. *In* Search in artificial intelligence. *Edited by* L. Kanal and V. Kumar. Springer-Verlag, New York, NY. pp. 426–449.

NADEL, B.A 1986. The consistent labeling problem and its algorithms: towards exact-case complexities and theory-based heurestics. Ph.D. dissertation. Computer Science Department, Rutgers University, New Brunswick, N.J.

———1988*a*. Tree search and arc consistency in constraint satisfaction algorithms. *In* Search in artificial intelligence. *Edited by* L. Kanal and V. Kumar. Springer-Verlag, New York, NY. pp. 287–342.

———1988*b*. Representation selection for constraint satisfaction problems: a case study using *n*-queens. Technical Report CSC-88-006, Department of Computer Science, Wayne State University, Detroit, MI; IEEE Expert, **5**(3). June 1990. To appear.

———1988*c*. Precision complexity analysis: a case study using insertion sort. Technical Report CSC-88-008, Department of Computer Science, Wayne State University, Detroit, MI.

NUDEL, B.A. 1982. Consistent-labeling problems and their algorithms. Proceedings of the National Conference on Artificial Intelligence, Pittsburgh, PA, pp. 128–132.

———1983*a*. Consistent-labeling problems and their algorithms: expected-complexities and theory-based heuristics. Artificial Intelligence (Special Issue on Search and Heuristics), **21**(1 and 2): 135–178. Also *in* Search and heuristics. North-Holland, Amsterdam, The Netherlands.

———1983*b*. Solving the general consistent labeling (or constraint satisfaction) problem: two algorithms and their expected complexities. Proceedings of the National Conference on Artificial

Intelligence, Washington, DC, pp. 292-296.
PURDOM, P.W., Jr. 1982. Evaluating search methods analytically. Proceedings of the National Conference on Artificial Intelligence, Pittsburgh, PA, pp. 124-127.
———1983. Search rearrangement backtracking and polynomial average time. Artificial Intelligence (Special Issue on Search and Heuristics), **21**(1 and 2): 117-133.
PURDOM, P.W., Jr., and BROWN, C. 1981. An average time analysis of backtracking. SIAM Journal on Computing, **10**(3): 583-593.
RIT, J. 1986. Propagating temporal constraints for scheduling. Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, PA, pp. 383-388.
ROSENFELD, A. 1975. Networks of automata: some applications. IEEE Transactions on Systems, Man and Cybernetics, **SMC-5**(5): 380-383.
ROSENFELD, A., HUMMEL, R.A., and ZUCKER, S.W. 1976. Scene labeling by relaxation operations. IEEE Transactions on Systems, Man and Cybernetics, **SMC-6**(6): 420-433.
SEIDEL, R. 1981. A new method for solving constraint satisfaction problems. Proceedings of the 7th International Joint Conference on Artificial Intelligence, Vancouver, B.C., pp. 338-342.
SHAPIRO, L.G., and HARALICK, R.M. 1981. Structural descriptions and inexact matching. IEEE Transactions on Pattern Analysis and Machine Intelligence, **PAMI-3**(5): 504-519.
STEFIK, M. 1981. Planning with constraints (MOLGEN: Part I). Artificial Intelligence, **16**: 111-140.
TSANG, P. 1987. The consistent labeling problem in temporal reasoning. Proceedings of the Sixth National Conference on Artificial Intelligence, Seattle, WA, pp. 251-255.
ULLMAN, J.R. 1973. Pattern recognition techniques. Crane Russak, New York, NY. p. 198.
———1976. An algorithm for subgraph isomorphism. Journal of the ACM, **23**: 31-42.
VAN HENTENRYCK, P., and DINCBAS, M. 1986. Domains in logic programming. Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, PA, pp. 759-765.
WALKER R.L. 1960. An enumerative technique for a class of combinatorial problems. Combinatorial Analysis (Proceedings of the Symposium on Applied Mathematics, Vol. X), American Mathematical Society, Providence, RI, pp. 91-94.
WALTZ, D. 1975. Understanding line drawings of scenes with shadows. *In* The Psychology of computer vision. *Edited by* P.H. Winston. McGraw-Hill, New York, NY. Originally in Technical Report AI-TR-271, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA. 1972.

Appendix I. More-detailed examples

To avoid losing sight of the forest for the trees, our hybrid algorithm traces above (Figs. 10-12) did not show the actual constraint checks that were performed. The tree search algorithm traces (Figs. 1 and 2) did show the constraint checks, but not their order nor the values of important variables which controlled the search. (The only exceptions were in connection with the E nodes in Figs. 1 and 12, where the constraint check sequencing was explicitly given, as it was also in connection with Fig. 6.)

This avoidance of detail allowed the main features of our algorithms to be more clearly exposed, but at the cost of conveying a possibly incomplete understanding of what was really going on. In this appendix we fill in these details. Earlier traces, and some new ones, are given, showing the sequencing of individual constraint checks. For the tree search algorithms BJ and BM, the effect of their control variables (faildepth, returndepth, MaxCheckLevel, MinBackupLevel) is also discussed in more detail, and their values are shown in the corresponding traces.

In all figures below, individual constraint checks are denoted by 5-tuples as described in Sect. 4 in connection with Fig. 6. Specifically, the 5-tuple ABCDE denotes that instantiation z_A = B was checked against z_C = D, and that the result was E, where E can be either T for true or F for false, indicating respectively that the corresponding binary constraint was found to be satisfied or violated by the pair of instantiations. This convention, and most others for the BT, BJ, and BM traces below, are from Gaschnig (1979).

Tree search algorithms

Figure A1 gives traces for BT, BJ, and BM solving confused 4-queens. These are more-detailed versions of the graphical traces which appeared above in Figs. 1 and 2. For ease of comparison, the block of processing at a node is labeled before with "L start" and after with "L end," where L is the same letter as used for that node in Figs. 1 and 2. Corresponding node starts and ends are aligned in the three traces here. The resulting gaps left in the BJ trace emphasize the instantiation and node savings of this algorithm compared to BT. The lack of such gaps in the BM trace emphasize that BM carries out all instantiations, and generates all nodes, that BT does. Remember, BJ avoids checks by avoiding instantiations, but cannot save checks once a given instantiation is made. BM cannot avoid instantiations, but may save checks by avoiding them at the instantiations it does make. In Sect. 3.3 these were distinguished as "horizontal" versus "vertical" savings respectively.

In Fig. A1, the constraint checks ABCDE corresponding to a given instantiation z_A = B are placed on a single line in their order of execution, preceded by the pair AB. Thus the line 34 3412T 3421F occurring for algorithm BT means that after setting z_3 = 4, this instantiation was checked against past instantiation z_1 = 2 and found to satisfy the corresponding constraint, then it was checked against z_2 = 1 and found to violate the constraint. Note that all the check 5-tuples in such a line start with the same AB as precedes the line, and that the third component increases from 1 by 1 in successive 5-tuples, since the three algorithms check against past variables in the order z_1, z_2, z_3, ...

Lines of our traces here are indented right in proportion to the level of their node in the tree. The traces thus correspond to trees "lying on their side" with the root to the left. Top-to-bottom traversal in Figs. 1 and 2 corresponds to left-to-right traversal in Fig. A1. And left-to-right traversal in the former traces corresponds to top-to-bottom traversal in the latter.

For the purpose of comparison, at the beginning and end of the traces in Fig. A1, two complexity measures are given in the form (x:y). x is the cumulative number of constraint checks performed to the end of the corresponding line's checks. y is the number of nodes that have been generated (started, though not necessarily completed) by the algorithm to that stage.

Backtracking

The BT trace of Fig. A1 makes explicit the depth-first order in which BT traverses the search tree shown graphically in Fig. 1. In particular, note that the order of constraint checking at node E, and the interleaving of subnode generation, is as discussed in Sect. 3.1. In the trace of Fig. A7 below, we contrast this order with that of revise-based BT.

BT	BJ	BM
12 (37:7) B start 21 2112T C start 31 3112F 32 3212T 3221T D start 41 4112F 42 4212T 4221F 43 4312F 44 4412F D end 33 3312F 34 3412T 3421F C end 22 2212T E start 31 3112F 32 3212T 3222T F start 41 4112F 42 4212T 4222T 4232T Solution = 2222 43 4312F 44 4412F F end 33 3312F 34 3412T 3422F E end 23 2312T G start 31 3112F 32 3212T 3223T H start 41 4112F 42 4212T 4223F 43 4312F 44 4412F H end 33 3312F 34 3412T 3423T I start 41 4112F 42 4212T 4223F 43 4312F 44 4412F I end G end 24 2412F (80:15) B end	12 [3] (37:7) B start 21 [0] 2112T C start 31 [0] 3112F [1] 32 [1] 3212T 3221T D start 41 [0] 4112F [1] 42 [1] 4212T 4221F [2] 43 [2] 4312F [1] 44 [2] 4412F [1] Backjump! D end C end 22 [2] 2212T E start 31 [0] 3112F [1] 32 [1] 3212T 3222T F start 41 [0] 4112F [1] 42 [1] 4212T 4222T 4232T [3] Solution = 2222 43 [3] 4312F [1] 44 [3] 4412F [1] F end 33 [3] 3312F [1] 34 [3] 3412T 3422F [2] E end 23 [3] 2312T G start 31 [0] 3112F [1] 32 [1] 3212T 3223T H start 41 [0] 4112F [1] 42 [1] 4212T 4223F [2] 43 [2] 4312F [1] 44 [2] 4412F [1] Backjump! H end G end 24 [3] 2412F [1] (69:14) B end	12 [1:1]Y [0] (23:7) B start 21 [1:1]Y 2112T [1] C start 31 [2:1]Y 3112F [1] 32 [1:1]Y 3212T 3221T [2] D start 41 [2:1]Y 4112F [1] 42 [1:1]Y 4212T 4221F [2] 43 [1:1]Y 4312F [1] 44 [3:1]Y 4412F [1] 1111 -> 1113 D end 33 [2:1]Y 3312F [1] 34 [1:1]Y 3412T 3421F [2] 1113 -> 1122 C end 22 [1:1]Y 2212T [1] E start 31 [1:2]N aF 32 [2:2]Y bT 3222T [2] F start 41 [1:2]N aF 42 [2:2]Y bT 4222T 4232T [3] Solution = 2222 43 [1:2]N aF 44 [1:2]N aF 1122 -> 1123 F end 33 [1:2]N aF 34 [2:2]Y bT 3422F [2] 1123 -> 1122 E end 23 [1:1]Y 2312T [1] G start 31 [1:2]N aF 32 [2:2]Y bT 3223T [2] H start 41 [1:2]N aF 42 [3:2]Y bT 4223F [2] 43 [1:2]N aF 44 [1:2]N aF 1122 -> 1123 H end 33 [1:2]N aF 34 [2:2]Y bT 3423T [2] I start 41 [1:3]N aF 42 [2:3]N aT aF 43 [1:3]N aF 44 [1:3]N aF 1123 -> 1123 I end 1123 -> 1122 G end 24 [1:1]Y 2412F [1] (45:15) 1122 -> 1111 B end
Nodes, here 9 Nodes, whole tree 29 Checks, here 43 Checks, whole tree 160	8 27 32 139	9 29 22 90

FIG. A1. Detailed traces of BT, BJ, and BM solving confused 4-queens, for the same subtrees as shown graphically in Figs. 1 and 2.

Backjumping

The BJ trace in Fig. A1 is similar to that for BT. However, lines of the BJ trace include an additional one or two numbers in square brackets. The first is the value of the local returndepth variable before, and just after, the corresponding instantiation took place. The second number in square brackets occurs only on lines where the constraint checking turned up an inconsistency, or on a line just before a solution is found. It is the value assigned to faildepth after that line's constraint checking. This value equals the number of checks appearing on the corresponding line of the trace. As required by BJ, the first of these two values (returndepth) on a line is the running maximum over the second of these values (faildepth) on previous lines — previous instantiations — for the same node, and over values returned by earlier recursive calls to BJ from that node, if there were any.

Consider node D of the BJ trace, for example, where z_4 is being instantiated. The local value of returndepth at that

node starts with value 0, and then successively takes on values 1, 2, 2, 2 after the four assignments of faildepth to 1, 2, 1, and 1, corresponding to failures of the z_4 values against the instantiations for past variables z_1, z_2, z_1, and z_1 respectively. (The assignment of returndepth to its final value at a node is not recorded in the trace. Its value is easily determined though, as the maximum of the returndepth and faildepth values appearing on the line for the last instantiation at that node.)

On exiting node D, returndepth has value 2. On return to the parent $k = 3$ node C, faildepth is set to this value of 2, and the test faildepth < k succeeds. Node C is therefore immediately exited, avoiding the constraint checks corresponding to instantiations $z_3 = 3$ and $z_3 = 4$. Node C returns to set faildepth to 2 at its parent $k = 2$ node B. At this latter node, the test faildepth < k fails; the backjumping thus stops and the next value, $z_2 = 2$, is tried at node B. A similar process occurs on return from node H, but note the avoidance then not only of instantiations and their checks in the parent node G but also the avoidance of the whole node I (as seen graphically in Fig. 1.)

Backmarking

As discussed in Sect. 3.3, constraint checks done by BT are avoided by BM in two ways, which we called type (a) and type (b) savings. Both types of savings are achieved by use of BM's two array variables MinBackupLevel and MaxCheckLevel. In terms of these variables, the reasoning behind these savings is as follows:

(a) MaxCheckLevel[k, v] < MinBackupLevel[k] means that since the last node at which BM checked $z_k = v$ against instantiations of the past variables z_1, z_2, ..., BM has not yet backed up to the level p = MaxCheckLevel[k, v] of the deepest past variable z_p reached during that checking. It also means that the check of $z_k = v$ failed then against the value of z_p. This is because it is always true that MinBackupLevel[k] < k, and this together with MaxCheckLevel[k, v] < MinBackupLevel[k] implies that MaxCheckLevel[k, v] < $k - 1$. This means that $z_k = v$ failed when it was last tested, as discussed for the analogous integer variable MaxCheckLevel of algorithm BJ2 in Sect. 3.2. Since the last time it was tested, $z_k = v$ failed against the value of z_p and we have not yet backed up to level p to change the value of z_p, then $z_k = v$ will again fail against the unchanged value of z_p if tested. Thus MaxCheckLevel[k, v] < MinBackupLevel[k] means that we can avoid all checks of $z_k = v$ against past variables and go on to the next instantiation of z_k. This saves the p checks that BT would otherwise have performed.

(b) MaxCheckLevel[k, v] ≥ MinBackupLevel[k] means that since the last node at which BM checked $z_k = v$ against instantiations of the past variables z_1, z_2, ..., BM *has* backed up to a level q = MinBackupLevel[k] which is equal to, or shallower than, the level p = MaxCheckLevel[k, v] of the deepest past variable z_p reached during that checking. The check of $z_k = v$ against the value of z_p may or may not have succeeded, but the checks against the instantiations of z_1 to z_{p-1} must have succeeded, else the checking of $z_k = v$ could not have reached z_p. Thus checks now against the past instantiations of z_1 to z_{q-1} will again succeed because we have since backed up only to level $q \leq p$ so that these past instantiations are still unchanged. The check of $z_k = v$ against z_q and deeper variables may fail, however, because they have since changed their values during backup. Thus MaxCheckLevel[k, v] ≥ MinBackupLevel[k] means that we may avoid the $q - 1$ checks of $z_k = v$ against z_1, z_2, ..., z_{q-1}, which are guaranteed to succeed, and need only check $z_k = v$ against the instantiations of z_q to z_{k-1} (stopping of course at the first failure, if one occurs).

In the BM trace of Fig. A1, constraint-check savings of type (a) are indicated by aT and aF symbols, the T and F indicating respectively that the corresponding check was one that would have succeeded or failed. (The aT and aF symbols correspond respectively to the grey circled check marks and the grey circled crosses in Fig. 2.) Type (b) check savings are indicated by bT symbols in the trace, the T denoting that these checks were each destined to succeed. (The bT symbols correspond to the grey squared check marks in Fig. 2.)

The BM trace also includes MaxCheckLevel and MinBackupLevel values to clarify the use of these arrays. A line of the trace corresponding to an instantiation z_A = B, after giving the initial juxtaposed pair AB as in the other traces, also includes a pair [X:Y] where X and Y are respectively the values MaxCheckLevel[A, B] and MinBackupLevel[A] *before* the constraint checking for that instantiation. For example, in node F of the BM trace, the [1:2] in the line 43 [1:2]N aF indicates that initially MaxCheckLevel[4, 3] = 1 and that MinBackupLevel[4] = 2 (both of which can be seen to be correct from a look at the preceding part of the trace). The [X:Y] pair is followed by a (redundant) Y or N, indicating respectively whether the BEGIN-END block of BM was actually entered or not. An N corresponds to type (a) savings. A Y may correspond to type (b) savings, or to no savings.

For lines with a Y, a final number in square brackets appears giving the value assigned to MaxCheckLevel[A, B] *after* the constraint checking for that instantiation. These values are the same as the corresponding faildepth value in the BJ trace, where one is shown. For both algorithms, this value is the number of checks appearing in the corresponding line of output of the trace (including possibly type (b) avoided checks in the case of BM).

Just before completing any node, BM updates the MinBackupLevel array to reflect the backup that is about to occur. The trace shows this update in the form ABCD -> EFGH, where ABCD and EFGH are the array values before and after update respectively. The updates are shown on a separate line following the last instantiation at each node.

Hybrid tree search/arc consistency algorithms

This section shows the detailed traces for most of our hybrid algorithms of Sect. 5. For all but the revise-based BT algorithm, the traces show the processing at a single node of the search tree for 5-queens: the level $k = 2$ node having past instantiation $z_1 = 2$. This is the 5-queens analog of the 4-queens node which was treated above in Fig. 10, and which was extended to the whole half-tree in Fig. 11. Revise-based BT is also traced, but using confused 4-queens as the example. In each case, the trace is at the detailed level of individual constraint checks performed.

Algorithms FC, PL, FL, RFL1, RFL2, and RFL3

Figures A2–A6 give the traces respectively for algorithms FC, PL, FL, RFL1, and RFL3 at the $z_1 = 2$ node for solving 5-queens. Each trace shows the domain array (as contained in array parameter d of the corresponding algorithms)

FIG. A2. The $z_1 = 2$ node when solving 5-queens by $FC(k) = TS(k) + AC^1/_4(k)$.

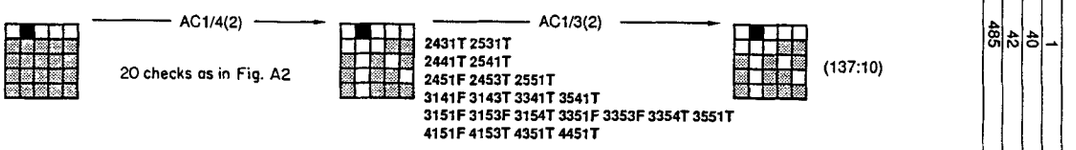


FIG. A3. The $z_1 = 2$ node when solving 5-queens by $PL(k) = TS(k) + AC^1/_4(k) + AC^1/_3(k)$.

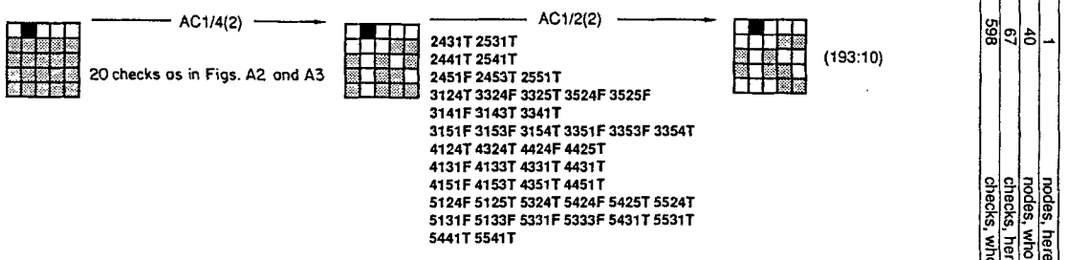


FIG. A4. The $z_1 = 2$ node when solving 5-queens by $FL(k) = TS(k) + AC^1/_4(k) + AC^1/_2(k)$.

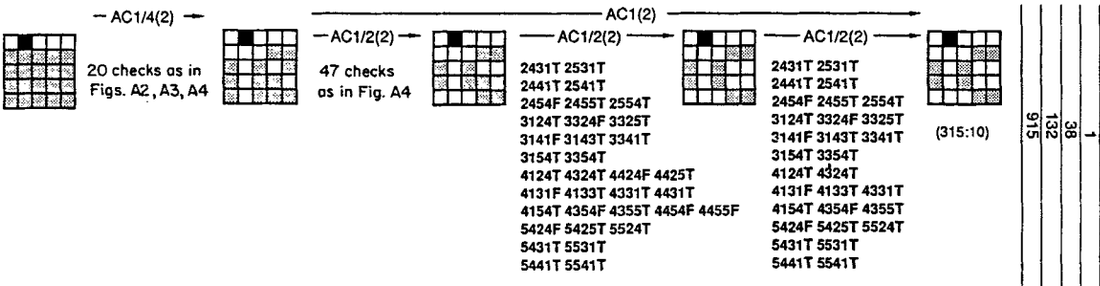


FIG. A5. The $z_1 = 2$ node when solving 5-queens by $RFL1(k) = TS(k) + AC^1/_4(k) + AC1(k)$.

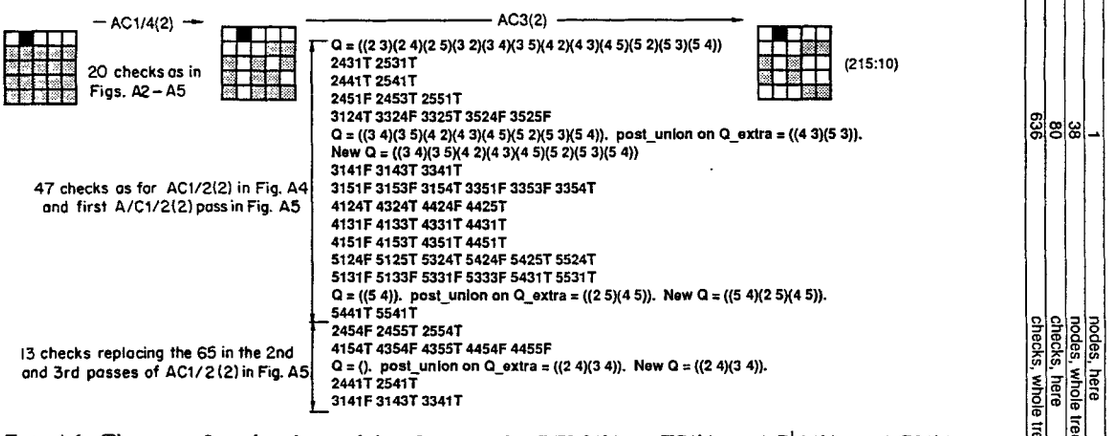


FIG. A6. The $z_1 = 2$ node when solving 5-queens by $RFL3(k) = TS(k) + AC^1/_4(k) + AC3(k)$.

before and after each arc consistency procedure call at the node, these calls being listed left to right in their order of occurrence. For each AC procedure call the figure shows the corresponding block of constraint checks performed, each check being given in the same 5-tuple format as used above. Checks of a block are grouped by arc revision, with successive arc revisions corresponding to successive lines in a block. For a given arc revision (line of a block), checks

TABLE A1. Statistics for solving 5-queens by RFLi, i = 1, 2, and 3

Algorithm RFLi = TS + AC¼ + ACi	Checks for z_1 = 2 node: AC¼	Checks for z_1 = 2 node: ACi	Checks for problem	Nodes for problem
RFL1 = TS + AC¼ + AC1	20	112	915	38
RFL2 = TS + AC¼ + AC2	20	49	595	38
RFL3 = TS + AC¼ + AC3	20	60	636	38

TABLE A2. Statistics for solving confused 6-queens by RFLi, i = 1, 2, and 3

Algorithm RFLi = TS + AC¼ + ACi	Checks for z_1 = 2 node: AC¼	Checks for z_1 = 2 node: ACi	Checks for problem	Nodes for problem
RFL1 = TS + AC¼ + AC1	30	52	792	39
RFL2 = TS + AC¼ + AC2	30	67	806	39
RFL3 = TS + AC¼ + AC3	30	35	696	39

are listed left to right in the order of their performance. The (x:y) pair shown at the end of each node's processing gives the number of checks performed, and the number of nodes generated, to that stage.

Note that extra information is added to the RFL3 trace of Fig. A6. Following each arc revision by AC3 in which a domain-value deletion actually occurred, the figure gives the current value of the arc-list Q, the value of the incremental arc-list Q__extra to be post-unioned onto Q due to the deletion, and the resulting new value of the list Q.

From the traces and associated statistics shown, we see again the ineffectiveness of increasing the amount of arc consistency processing at a node. (Or in terms of the order of presentation used in the body of the paper, we see the effectiveness of reducing the amount of processing at a node.) PL removes no more domain values at our node than does FC. FL removes three more than PL. RFL1 and RFL3 remove one more value than FL. But large amounts of extra constraint checks are expended at the node by the successive algorithms to achieve these small gains. Over the whole tree, the extra filtering at nodes does result in some small decreases in the total number of nodes in the tree. The total number of checks, however, certainly does not drop, but increases significantly. Despite having the most nodes, we see again that FC, with the least amount of processing per node of the algorithms in Figs. A2–A6, has the least checks for the whole tree.

Owing to space restrictions we do not present a trace of RFL2 analogous to those for RFL1 and RFL3. However, the corresponding statistics for the z_1 = 2 node and for the whole tree are given in Table A1, where they are compared with the statistics seen above for RFL1 and RFL3.

We see that for our 5-queens node, AC2 is more efficient than AC3, which is more efficient than AC1, with corresponding order for RFL2, RFL3, and RFL1 over the whole tree. Of course, as will always be the case for any problem, the three algorithms generate the same number of nodes because, at each corresponding node, they all achieve the same state of full arc consistency. (Note the same final domain array in Figs. A5 and A6).

Though AC2 is preferable to AC3 for the above example and for q-queens more generally, as seen in Table 3, this is certainly not always the case, as seen in Table 2 for confused q-queens. Table 2 in fact shows that not only AC3 but even AC1 can be preferable to AC2 — a possibility apparently not previously noted in the literature. A specific example where we see this is the z_1 = 2 node for confused 6-queens. For that node and problem, the analog of Table A1 is given in Table A2.

Revise-based BT

The action of revise-based BT (Sect. 5.2.4) on the above 5-queens node is straightforward and is therefore not given here. Instead, we give in Fig. A7 the trace of revise-based BT solving confused 4-queens. In particular, the figure is a more-detailed view of the processing given graphically in Fig. 12. The detailed trace here also provides the counterpart to the trace given in Fig. A1 for regular BT. As in Fig. A1 for ease of comparison, the block of processing at a node is labeled before with "L start" and after with "L end," where L is the same letter as used for that node in Fig. 12 (and Figs. 1 and 2).

In Fig. A7, constraint checks for a given node all appear on the same line in their order of execution. Vertical lines are used to separate the checks corresponding to successive arc revisions at a node. The sequence of arc revisions, and associated number of checks, is of course that shown in simplified form using arrows, and associated numbers, at the corresponding nodes of Fig. 12.

Note that as required, the trace here has exactly the same 43 checks as shown for regular BT in Fig. A1, but that the order is in general different at corresponding nodes. In particular, the order of checks at a node in Fig. A7 is that obtained from a column-wise ordering of the checks at the corresponding node in Fig. A1. This is the same "horizontal" versus "vertical" difference in ordering of constraint checks as noted earlier for the individual node E in comparing regular and revise-based BT in parts (b) and (c) of Fig. 12.

Note also that as required, the same 8 nodes are generated, in the same order, by the two versions of BT. As seen in the figure, revised-based BT proceeds in D-first manner (Horowitz and Sahni 1978) and does not interleave subnode generation till all constraint checking is completed at a parent node. On the other hand, in Fig. A1 we saw that regular BT proceeds in true depth-first manner, interleaving subnode generation between constraint checking at a parent node. This difference in interleaving is why we could place

```
...                                                              (37:7)
B start 2112T 2212T 2312T 2412F B end                            (41:8)
    C start 3112F 3212T 3312F 3412T | 3221T 3421F C end
        D start 4112F 4212T 4312F 4412F | 4221F D end
    E start 3112F 3212T 3312F 3412T | 3222T 3422F E end
        F start 4112F 4212T 4312F 4412F | 4222T | 4232T Solution = 2222 F end
    G start 3112F 3212T 3312F 3412T | 3223T 3423T G end
        H start 4112F 4212T 4312F 4412F | 4223F H end
        I start 4112F 4212T 4312F 4412F | 4223F I end       (80:15)
```

FIG. A7. Trace of subtree for solving confused 4-queens by revise-based $BT(k) = TS(k) + AC^{1}/_{5}(k)$.

all checks for a given node on a single line in the trace of Fig. A7, whereas in Fig. A1 only checks for a given instantiation at a node could appear on the same line.

Appendix II. Programming conventions

Our algorithms above are written in a pseudocode modeled essentially on Pascal. Using pseudocode has the advantage of brevity but the disadvantage of potentially introducing ambiguities. The following points are intended to clarify the programming conventions used.

- Due to their expressive power, we have made extensive use of FOR-WHILE loops of the form

FOR v := lower TO upper WHILE condition DO body

This kind of loop does not exist in Pascal, but is modeled on those in Algol and Sail (Stanford Artificial Intelligence Language). We are assuming that such a loop works by first initializing the loop variable v to the integer value "lower." If (*i*) this value does not exceed "upper" and (*ii*) the Boolean "condition" expression evaluates to true, then the body of the loop is executed. On each subsequent cycle the loop variable is incremented by one, after which the same two tests must be passed before performing the loop body again. The loop variable is assumed to retain its most recent value after termination of the loop. Thus it terminates with a value (possibly even upper + 1) one greater than during the last *completed* loop cycle. This is the reason for having to subtract 1 from p in algorithms BJ and BM to obtain the value at the last completed FOR cycle.

- As in Pascal, reference parameters are preceded in a formal parameter list by the word VAR. For clarity, each formal parameter that needs it gets its own preceding VAR qualifier, rather than covering several reference parameters with one VAR as allowed in Pascal. Care has been taken to include a formal reference parameter for each variable that is modified and passed back by a subroutine, rather than allowing updates as side effects. This makes for clearer code and also allows our subroutines to be used in a lexically scoped language such as Pascal or Common Lisp, without the need to physically include their definition in every (sub)program that uses them.

- For brevity, a Return(x) or Return statement is sometimes used respectively in a function or procedure (see the two versions of BJ in Sect. 3.2), although these are not available in standard Pascal. If such explicit returns are not used in a function, the normal Pascal mode of return applies where the returned value of the function is that which is last assigned to the function name before exit (as in function *check* of Sect. 2).

- As in Pascal, semicolons are used to divide between successive statements, but need not appear just to terminate a statement when there is not an immediate successor statement.

- Comments are delimited by brace symbols {, }, and comments are allowed anywhere, including in the header line of a subroutine declaration.

- For brevity of code, type declarations for variables (but not functions) are left implicit or, if necessary, are described in the text. Also for brevity, the BEGIN that precedes the body of a subroutine in Pascal is left out.

- We have assumed the language does its own garbage collection. (Pascals usually don't. Lisps usually do.) All arc consistency procedures of Sect. 4 and all the hybrid algorithms of Sect. 5 which use them are based ultimately on the revise procedure which stores the domain lists in an array d such that d[i] contains the current version of domain d_{z_i} of variable z_i. However, as mentioned in Sect. 4, since only deletions are performed on the domain lists, an additive version of revise is possible for which it suffices to use an array to implement the domain list d[i]. Using this approach, garbage collection becomes unnecessary for domain filtering. However, besides simple deletion of values from domain lists, arc consistency procedures AC2 and AC3 also require more complicated manipulations on their arc lists Q, Q_extra, Q1, Q2, and Q2_extra. Thus for AC2 and AC3 and the algorithms which use them, true dynamically allocated lists will be convenient for the arc lists. In that case in a language such as Pascal, algorithms AC2 and AC3 and their utilities (e.g., pop) will need to be augmented to include explicit disposal of unneeded nodes. Don't forget on exit of AC2 and AC3 to dispose of lists Q, Q1, and Q2, since they can be non-nil at that point if early termination has occurred due to an empty_domain = True condition.