Algorithms for Constraint-Satisfaction Problems: A Survey

Vipin Kumar

A large number of problems in AI and other areas of computer science can be viewed as special cases of the constraint-satisfaction problem. Some examples are machine vision, belief maintenance, scheduling, temporal reasoning, graph problems, floor plan design, the planning of genetic experiments, and the satisfiability problem. A number of different approaches have been developed for solving these problems. Some of them use constraint propagation to simplify the original problem. Others use backtracking to directly search for possible solutions. Some are a combination of these two techniques. This article overviews many of these approaches in a tutorial fashion.

A large number of problems in AI and other areas of computer science can be viewed as special cases of the constraint-satisfaction problem (CSP) (Nadel 1990). Some examples are machine vision (Chakravarty 1979; Davis and Rosenfeld 1981; Mackworth 1977b; Montanari 1974; Hummel 1976), belief maintenance (Dechter 1987; Dechter and Pearl 1988b; Dhar and Croker 1990), scheduling (Dhar and Ranganathan 1990; Fox 1987; Fox, Sadeh, and Baykan 1989; Petrie et al. 1989; Prosser 1989; Rit 1986), temporal reasoning (Allen 1983, 1984; Dechter, Meiri, and Pearl 1991; Vilain and Kautz 1986; Tsang 1987), graph problems (McGregor 1979; Bruynooghe 1985), floor plan design (Eastman 1972), the planning of genetic experiments (Stefik 1981), the satisfiability problem (Zabih and McAllester 1988), circuit design (de Kleer and Sussman 1980), machine design and manufacturing (Frayman and Mittal 1987; Nadel and Lin 1991; Navinchandra 1990), and diagnostic reasoning (Geffner and Pearl 1987).

The scope of this article is restricted to those constraint-satisfaction problems that can be stated as follows: We are given a set of variables, a finite and discrete domain for each variable, and a set of constraints. Each constraint is defined over some subset of the original set of variables and limits the combinations of values that the variables in this subset can take. The goal is to find one assignment to the variables such that the assignment satisfies all the constraints. In some problems, the goal is to find all such assignments. More general formulations of CSP can be found in Freuder (1989); Gu (1889); Mackworth, Mulder, and Havens (1985); Mittal and Frayman (1987); Mittal and Falkenhainer (1990); Freeman-Benson, Maloney, and Borning (1990); Navinchandra and Marks (1987); Shapiro and Haralick (1981); and Ricci (1990).

I further restrict the discussion to CSPs in which each constraint is either unary or binary. I refer to such CSP as binary CSP. It is possible to convert CSP with *n*-ary constraints to another equivalent binary CSP (Rossi, Petrie, and Dhar 1989). Binary CSP can be depicted by a constraint graph in which each node represents a variable, and each arc represents a constraint between variables represented by
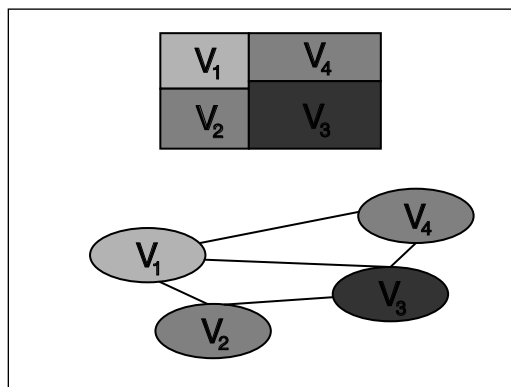

Figure 1. An Example Map-Coloring Problem and Its Equivalent Constraint-Satisfaction Problem.

the end points of the arc. A unary constraint is represented by an arc originating and terminating at the same node. I often use the term CSP to refer to the equivalent constraint graph and vice versa.

For example, the map-coloring problem can be cast as CSP. In this problem, we need to color (from a set of colors) each region of the map such that no two adjacent regions have the same color. Figure 1 shows an example map-coloring problem and its equivalent CSP. The map has four regions that are to be colored red, blue, or green. The equivalent CSP has a variable for each of the four regions of the map. The domain of each variable is the given set of colors. For each pair of regions that are adjacent on the map, there is a binary constraint between the corresponding variables that disallows identical assignments to these two variables.

Backtracking: A Method for Solving the Constraint-Satisfaction Problem

CSP can be solved using the generate-and-test paradigm. In this paradigm, each possible combination of the variables is systematically generated and then tested to see if it satisfies all the constraints. The first combination that satisfies all the constraints is the solution. The number of combinations considered by this

method is the size of the Cartesian product of all the variable domains.

A more efficient method uses the backtracking paradigm. In this method, variables are instantiated sequentially. As soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial instantiation violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. Clearly, whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domains. The backtracking method essentially performs a depth-first search (Kumar 1987) of the space of potential CSP solutions.

Although backtracking is strictly better than the generate-and-test method, its run-time complexity for most nontrivial problems is still exponential. One of the reasons for this poor performance is that the backtracking paradigm suffers from *thrashing* (Gaschnig 1979); that is, search in different parts of the space keeps failing for the same reasons. The simplest cause of thrashing concerns the unary predicates and is referred to as *node inconsistency* (Mackworth 1977a). If the domain D_i of a variable V_i contains a value a that does not satisfy the unary constraint on V_i, then the instantiation of V_i to a always results in an immediate failure. Another possible source of thrashing is illustrated by the following example: Suppose the variables are instantiated in the order V_1, V_2, ...,V_i, ..., V_j, ..., V_N. Suppose also that the binary constraint between V_i and V_j is such that for $V_i = a$, it disallows any value of V_j. In the backtrack search tree, whenever V_i is instantiated to a, the search will fail while instantiation is tried with V_j (because no value for V_j would be found acceptable). This failure will be repeated for each possible combination that the variables V_k ($i < k < j$) can take. The cause of this kind of thrashing is referred to as a lack of *arc consistency* (Mackworth 1977a). Other drawbacks of simple backtracking are discussed in Intelligent Backtracking and Truth Maintenance.

Thrashing because of node inconsistency can be eliminated by simply removing those values from the domains D_i of each variable V_i that do not satisfy unary predicate P_i. Thrashing because of arc inconsistency can be avoided if each arc (V_i, V_j) of the constraint graph is made consistent before the search starts. In the next section, I formally define the notion of arc consistency and consider algorithms for achieving it.
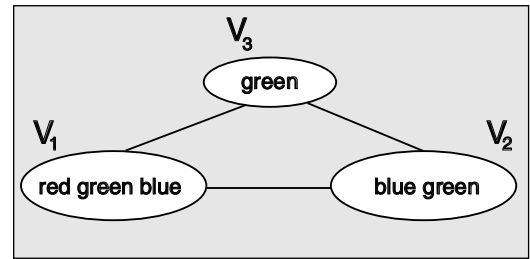

Figure 2. A Constraint Graph for Some Map-Coloring Problems.

Propagating Constraints

Arc (V_i, V_j) is arc consistent if for every value x in the current domain of V_i, there is some value y in the domain of V_j such that $V_1 = x$ and $V_j = y$ are permitted by the binary constraint between V_i and V_j. The concept of arc consistency is *directional*; that is, if an arc (V_i, V_j) is consistent, then it does not automatically mean that (V_j, V_i) is also consistent. Consider the constraint graph of another map-coloring problem given in figure 2. In this constraint graph, arc (V_3,V_2) is consistent because green is the only value in the domain of V_3, and for V_3 = green, at least one assignment for V_2 exists that satisfies the constraint between V_2 and V_3. However, arc (V_2, V_3) is not consistent because for V_2 = green, there is no value in the domain of V_3 that is permitted by the constraint between V_2 and V_3.

Clearly, an arc (V_i, V_j) can be made consistent by simply deleting those values from the domain of V_i for which the previous condition is not true. (Deletions of such values do not eliminate any solution of original CSP.) The following algorithm, taken from Mackworth (1977a), does precisely that.

```
procedure REVISE(V_i, V_j);
DELETE ← false;
for each x ε D_i do
        if there is no such v_j ∈ D_j
        such that (x, v_j ) is consistent,
        then
                delete x from D_i ;
                DELETE ← true;
        endif;
endfor;
return DELETE;
end_REVISE
```

To make every arc of the constraint graph consistent, it is not sufficient to execute REVISE for each arc just once. Once REVISE reduces the domain of some variable V_i, then each previously revised arc (V_j, V_i) has to be revised again because some of the members
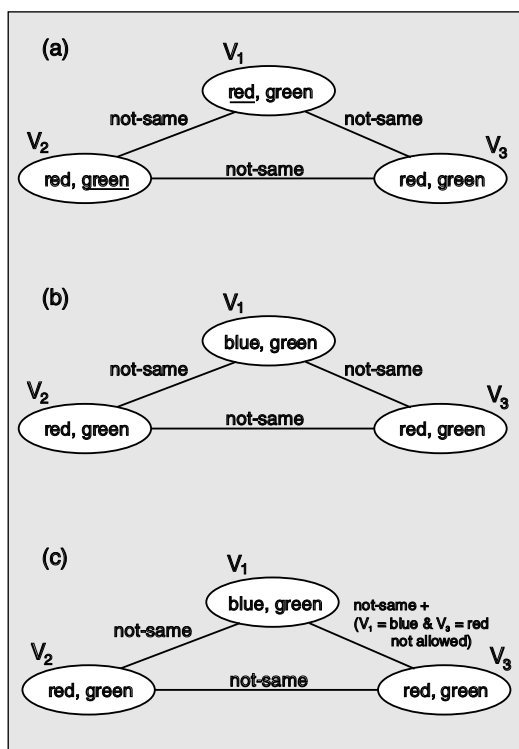
Figure 3. Examples of Arc-Consistent Constraint Graphs.

(a) The graph has no solutions. (b) The graph has two solutions ((blue,red,green), (blue,green,red)). (c) The graph has exactly one solution (blue,red,green).

of the domain of V_j might no longer be compatible with any remaining members of the revised domain of V_i. For example, in CSP in figure 2, arc (V_1, V_2) is initially consistent. After arc (V_2, V_3) is made consistent by deleting green from the domain of V_2, arc (V_1, V_2) no longer remains consistent. The following algorithm, taken from Mackworth (1977a), obtains arc consistency for the whole constraint graph G:

    procedure AC-1
    $Q \leftarrow \{(V_i, V_j) \in \text{arcs}(G), i \neq j\}$;
    repeat
            CHANGE $\leftarrow$ false;
            for each $(Vi, V_j) \in Q$ do
                    CHANGE $\leftarrow$ (REVISE(V_i, V_j)
    or CHANGE);
            endfor;
    until not(CHANGE);
    end_AC

The major problem with the previous algorithm is that successful revision of even one arc in some iteration forces all the arcs to be revised in the next iteration, even though only a small number of them are affected by this revision. Mackworth (1977a) presents a

variation (called AC-3) of this algorithm that eliminates this drawback. This algorithm (given here) performs re-revision only for those arcs that are possibly affected by a previous revision. The reader can verify that in AC-3, after applying REVISE(V_k, V_m), it is not necessary to add arc (V_m, V_k) to Q. The reason is that none of the elements deleted from the domain of V_k during the revision of arc (V_k, V_m) provided support for any value in the current domain of V_m.

    procedure AC-3
    $Q \leftarrow \{(V_i, V_j) \, \varepsilon \, \text{arcs}(G), i \neq j\}$;
    while Q not empty
            select and delete any arc (V_k, V_m)
    from Q;
            if (REVISE(Vk, Vm) then
                    $Q \leftarrow \text{»} \{(V_i, V_k)$ such that $(V_i,$
    $V_k) \, \varepsilon \, \text{arcs}(G), i \neq k, i \neq m\}$
            endif;
    endwhile;
    end_AC

The well-known algorithm of Waltz (1975) is a special case of this algorithm and is equivalent to another algorithm, AC-2, discussed in Mackworth (1977a). Assume that the domain size for each variable is d, and the total number of binary constraints (that is, the arcs in the constraint graph) is e. The complexity of an arc-consistency algorithm given in Mackworth (1977a) is $O(ed^3)$ (Mackworth and Freuder 1985). Mohr and Henderson (1986) present another arc-consistency algorithm that has a complexity of $O(ed^2)$. To verify the arc consistency, each arc must be inspected at least once, which takes $O(d^2)$ steps. Hence, the lower bound on the worst-case time complexity of achieving arc consistency is $O(ed^2)$. Thus, Mohr and Henderson's algorithm is optimal in terms of worst-case complexity. Variations and improvements of this algorithm were developed in Han and Lee (1988) and Chen (1991).

Given an arc-consistent constraint graph, is any (complete) instantiation of the variables from current domains a solution to CSP? In other words, can achieving arc consistency completely eliminate the need for backtracking? If the domain size of each variable becomes one after obtaining arc consistency, then the network has exactly one solution, which is obtained by assigning the only possible value in its domain to each variable. Otherwise, the answer is no in general. The constraint graph in figure 3a is arc consistent, but none of the possible instantiations of the variables are solutions to CSP. In general, even after achieving arc consistency, a network can have (1) no solutions (figure 3a), (2) more than one solution (figure 3b), or (3)
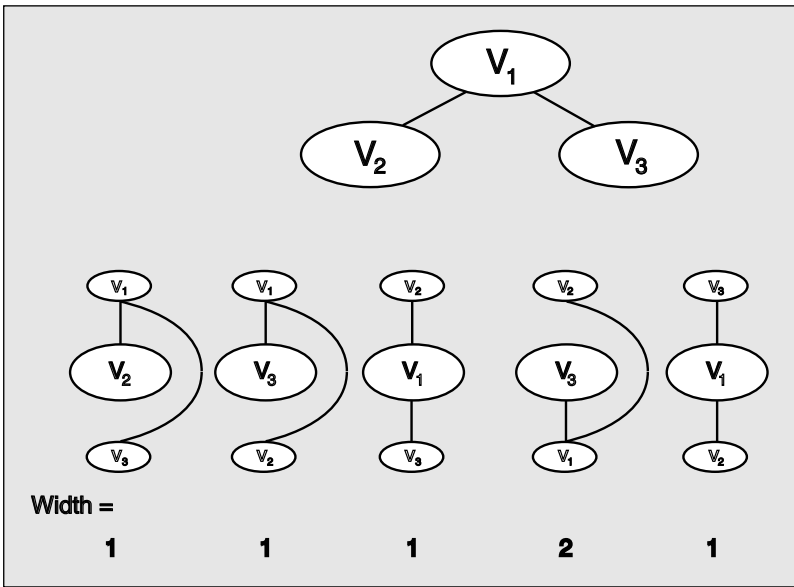
*Figure 4. A Constraint Graph and Its Six Different
Ordered Constraint Graphs.*

exactly one solution (figure 3c). In each case, search might be needed to find the solution(s) or discover that there is no solution. Nevertheless, by making the constraint graph arc consistent, it is often possible to reduce the search done by the backtracking procedure. Waltz (1975) shows that for the problem of labeling polyhedral scenes, arc consistency substantially reduces the search space. In some instances of this problem, the solution was found after no further search.

Given that arc consistency is not enough to eliminate the need for backtracking, can another stronger degree of consistency eliminate the need for search? The notion of *K* consistency captures different degrees of consistency for different values of *K*.

A constraint graph is *K* consistent if the following is true: Choose values of any *K* - 1 variables that satisfy all the constraints among these variables, then choose any K'th variable. A value for this variable exists that satisfies all the constraints among these *K* variables.

A constraint graph is strongly *K* consistent if it is *J* consistent for all *J* ≤ *K*.

Node consistency, discussed earlier, is equivalent to strong 1 consistency. Arc consistency is equivalent to strong 2 consistency. Algorithms exist to make a constraint graph strongly *K* consistent for *K* > 2 (Cooper 1989; Freuder 1988). Clearly, if a constraint graph containing *n* nodes is strongly *n* consistent, then a solution to CSP can be found without any search. However, the worst-case complexity of the algorithm for obtaining *n* consis-

tency in an *n*-node constraint graph is also exponential. If the graph is *K* consistent for *K* < *n*, then in general, backtracking cannot be avoided. Now the question is, Are there certain kinds of CSPs for which *K* consistency for *K* < *n* can eliminate the need for backtracking? Before answering this question, I define some terms.

An *ordered constraint graph* is a constraint graph whose vertices have been ordered linearly. Figure 4 shows six different ordered constraint graphs corresponding to the given constraint graph. Note that in the backtracking paradigm, the CSP variables can be instantiated in many different orders. Each ordered constraint graph of CSP provides one such order of variable instantiations. (The variables that appear earlier in the ordering are instantiated first. For example, in figure 4, for each ordered graph, the variable corresponding to the top node is instantiated first.) It turns out that for some CSPs, some orderings of the constraint graph are better than other orderings in the following sense: If the backtracking paradigm uses these orderings to instantiate the variables, then it can find a solution to CSP without search (that is, the first path searched by backtracking leads to a solution). Next, I define the width of a constraint graph that is used to identify such CSPs.

The width at a vertex in an ordered constraint graph is the number of constraint arcs that lead from the vertex to the previous vertices (in the linear order). For example, in the leftmost ordered constraint graph, the width of the vertex V_2 is 1, and the width of V_1 is 0. The width of an ordered constraint graph is the maximum of the width of any of its vertices. The width of a constraint graph is the minimum width of all the orderings of the graph. Hence, the width of the constraint graph given in figure 4 is 1. The width of a constraint graph depends on its structure.

Theorem 1: If a constraint graph is strongly *K* consistent, and *K* > *w*, where *w* is the width of the constraint graph, then a search order exists that is backtrack free (Freuder 1988, 1982).

The proof of the theorem is straightforward. If *w* is the width of the graph, then an ordering of the graph exists such that the number of constraint arcs that lead from any vertex of the graph to the previous vertices (in the linear order) is, at most, *w*. Now, if the variables are instantiated using this ordering in the backtracking paradigm, then whenever a new variable is instantiated, a value for this variable that is consistent with all the previous assignments can be found. Such a value can be found because (1) this value has to be

If the domain size of each variable becomes one after obtaining arc consistency, then the network has exactly one solution…

consistent with the assignments of, at most, w other variables (that are connected to the current variable) and (2) the graph is strongly $(w + 1)$ consistent.

It is relatively easy to determine the ordering of a given constraint graph that has a minimum width w (Freuder 1988, 1982). It might appear that all we need to do is to make this constraint graph strongly $(w + 1)$ consistent using the algorithms in Freuder (1978). Unfortunately, for $K > 2$, the algorithm for obtaining K consistency adds extra arcs in the constraint graph, which can increase the width of the graph. Hence, a higher degree of consistency has to be achieved before a solution can be found without any backtracking. In most cases, even the algorithm for obtaining strong 3 consistency adds so many arcs in the original n-variable constraint graph that the width of the resulting graph becomes n. However, we can use node-consistency or arc-consistency algorithms to make the graph strongly 2 consistent. None of these algorithms adds any new nodes or arcs to the constraint graph. Hence, if a constraint graph has width 1 (after making it arc and node consistent), we can obtain a solution to the corresponding CSP without any search.

Interestingly, all tree-structured constrained graphs have width 1 (that is, at least one of the orderings of a tree-structured constraint graph has width equal to 1) (Freuder 1988, 1982). As an example, figure 5 shows a tree-structured constraint graph and one of its ordered versions whose width is 1. Hence, if given CSP has a tree-structured graph, then it can be solved without any backtracking once it has been made node and arc consistent.

In Dechter and Pearl (1988a, 1988b), the notion of adaptive consistency is presented along with an algorithm for achieving it. Adaptive consistency is functionally similar to K consistency because it also renders CSP backtrack free. Its main advantage is that the time and space complexity of applying it can be determined in advance. For other techniques that take advantage of the structure of the constraint graphs to reduce search, see

Dechter and Pearl (1988a, 1988c, 1986); Freuder and Quinn (1985); Freuder (1990); Dechter, Meir, and Pearl (1990); Zabih (1990); Perlin (1991); and Montanari and Rossi (1991).

How Much Constraint Propagation Is Useful?

To this point, I have considered two rather different schemes for solving CSP: backtracking and constraint propagation. In the first scheme, different possible combinations of variable assignments are tested until a complete solution is found. This approach suffers from thrashing. In the second scheme, constraints between different variables are propagated to derive a simpler problem. In some cases (depending on the problem and the degree of constraint propagation applied), resulting CSP is so simple that its solution can be found without search. Although, any n-variable CSP can always be solved by achieving n consistency, this approach is usually even more expensive than simple backtracking. A lower-degree consistency (that is, K consistency for $K < n$) does not eliminate the need for search except for certain kinds of problems. A third possible scheme is to embed a constraint-propagation algorithm inside a backtracking algorithm, as follows:

A root node is created to solve original CSP. Whenever a node is visited, a constraint-propagation algorithm is first used to attain a desired level of consistency. If at a node, the cardinality of the domain of each variable becomes 1, and corresponding CSP is arc consistent, then the node represents a solution. If in the process of performing constraint propagation at the node, the domain of any variable becomes null, then the node is pruned. Otherwise one of the variables (whose current domain size is >1) is selected, and new CSP is created for each possible assignment of this variable. Each such new CSP is depicted as a successor node of the node representing parent CSP. (Note that each new CSP is smaller than parent CSP because we need to choose assignments for one less variable.) A back-
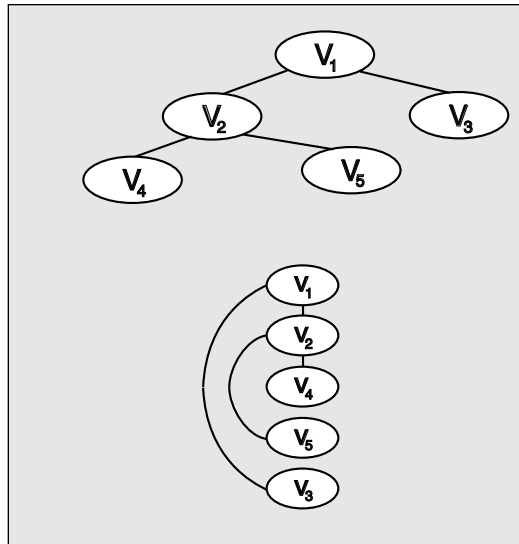
Figure 5. A Tree-Structured Constraint Graph and One of Its Width-1 Ordering.

tracking algorithm visits these nodes in the standard depth-first fashion until a solution is found. The question now is how much constraint propagation to do at each node. If no constraint propagation is done, then the paradigm reverts to simple backtracking (actually, as we see shortly, even simple backtracking performs some kind of constraint propagation). More constraint propagation at each node will result in the search tree containing fewer nodes, but the overall cost can be higher because the processing at each node will be more expensive. At one extreme, obtaining *n* consistency for the original problem would completely eliminate the need for search, but as mentioned before, this method is usually more expensive than simple backtracking.

A number of algorithms for solving CSPs that essentially fit the previous format have been investigated by various researchers (Haralick and Elliot 1980; Nadel 1988; Fikes 1970; Gaschnig 1974; Ullman 1976; Haralick, Davis, and Rosenfeld 1978; McGregor 1979; Dechter and Meiri 1989). In particular, Nadel (1988) empirically compares the performance of the following algorithms: generate and test (GT), simple backtracking (BT), forward checking (FC), partial lookahead (PL), full lookahead (FL), and really full lookahead (RFL). These algorithms primarily differ in the degrees of arc consistency performed at the nodes of the search tree.

Figure 6 (adapted from Nadel [1988])

depicts each of these algorithms as a combination of pure tree search and some fraction of arc consistency. On one extreme, GT is simple generate and test, and at the other extreme, RFL is a complete 2-consistency algorithm embedded in backtracking. The other algorithms—BT, FC, PL, and FL—are increasingly complete implementations of partial arc consistency. Note that even backtracking incorporates a limited degree of arc consistency because whenever a new variable is considered for instantiation, any of its values that are inconsistent with any previous instantiations cause immediate failure. Hence, the domain of this variable is effectively filtered to contain only those values that are consistent with the instantiations of variables higher in the tree. FC incorporates a greater degree of arc consistency than BT, as follows: Whenever a new variable instantiation is made, the domains of all as-yet-uninstantiated variables are filtered to contain only those values that are consistent with this instantiation. If the domains of any of these uninstantiated variables becomes null, then failure is recognized, and backtracking occurs. PL, FL, and RFL are essentially augmented versions of FC that perform arc consistency even between uninstantiated variables. Nadel (1988) presents a comprehensive evaluation of these algorithms in the context of *n*-queens and confused-*n*-queens problems. The *n-queens problem* is to place *n* queens on an *n* x *n* chess board in such a way that no pair of queens attacks each other. The *confused-n-queens problem* is a variation of the *n*-queens problem. Here, the *n* queens are to be placed on the *n* x *n* chess board, one queen to a row, such that each pair of queens attacks each other (that is, each pair of queens is either on the same column or the same diagonal). For these problems, FC (which implements only a fraction of the consistency achieved by the arc-consistency algorithm) turns out to be the best algorithm. Experiments by other researchers with a variety of problems also indicate that it is better to apply constraint propagation only in a limited form (Haralick and Elliot 1980; McGregor 1979; Gaschnig 1978, 1979; Dechter and Meiri 1989a; Prosser 1991).

Intelligent Backtracking and Truth Maintenance

There are two major drawbacks to the standard backtracking scheme. One is thrashing. Thrashing can be avoided by *intelligent backtracking*; that is, by a scheme in which backtracking is done directly to the variable that

caused the failure. The other drawback is having to perform redundant work. To see how backtracking can perform redundant work, consider the following example:

Assume that variables V_1, V_2, and V_3 have been assigned values a_1, b_3, and c_1, respectively. Assume that none of the values of V_3 were found compatible with the values b_1 and b_2 of V_2. Now assume that all possible values of V_4 conflict with the choice of $V_1 = a_1$. Because the conflict is caused by the inappropriate choice of V_1, clearly an intelligent backtracking scheme will perform backtracking to V_1 and, thus, assign a different value to V_1. However, even this scheme will undo the assignments of V_2 and V_3 and will rediscover, from scratch, the fact that none of the values of V_3 are compatible with the values b_1 and b_2 of V_2.

There is a backtracking-based method that eliminates both of these drawbacks to backtracking. This method is traditionally called *dependency-directed backtracking* (Stallman and Sussman 1977) and is used in truth maintenance systems (Doyle 1979; McDermott 1991). CSP can be solved by Doyle's RMS (Doyle 1979; Stallman and Sussman 1977), as follows: A variable is assigned some value, and a justification for this value is noted (the justification is simply that no justification exists for assigning other possible values). Then, similarly, a default value is assigned to some other variable and is justified. At this time, the system checks whether the current assignments violate any constraint. If they do, then a new node is created that essentially denotes that the pair of values for the two variables in question is not allowed. This node is also used to justify another value assignment to one of the variables. This process continues until a consistent assignment is found for all the variables. Such a system, if implemented in full generality, never performs redundant backtracking and never repeats any computation.

Although the amount of search performed by such a system is minimal, the procedures for determining the culprit of constraint violation and choosing a new value of the variables are complex (Petrie 1987). Hence, overall the scheme can take more time than even simple backtracking for a variety of problems. Hence, a number of simplifications to this scheme were developed by various researchers (Bruynooghe 1981; Rosiers and Bruynooghe 1986; Dechter 1986, 1990; Haralick and Elliot 1980; Gaschnig 1977). For example, a scheme developed by Dechter and Pearl is much simpler and less precise than the original dependency-directed backtracking scheme of Stallman and Sussman.
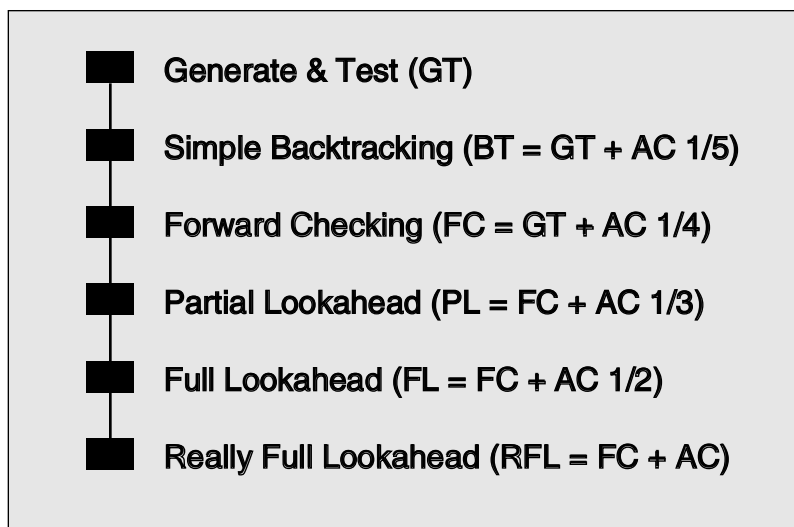
Even the schemes that perform only intelli-

Figure 6. Different Constraint-Satisfaction Algorithms Depicted as a Combination of Tree Searching and Different Degrees of Constraint Propagation.

gent backtracking can be complex depending on the analysis done to find the reason for failure. Recall that these schemes make no effort to avoid redundant work. A simple intelligent backtracking scheme can turn out to have less overall complexity than a more complicated intelligent backtracking method. The scheme presented in Freuder and Quinn (1985) can be viewed as a simple intelligent backtracking scheme that takes advantage of the structure of the constraint graph to determine possible reasons for failure.

Intelligent backtracking schemes developed for Prolog (Kumar and Lin 1988; Bruynooghe and Pereira 1984; Pereira and Porto 1982) are also applicable to CSPs. De Kleer developed an assumption-based truth maintenance system (ATMS) (de Kleer 1986a, 1986b) that also tries to remedy the drawbacks to simple backtracking. As discussed in de Kleer (1989), there are strong similarities between the constraint-propagation methods discussed in Propagating Constraints and ATMS for solving CSPs.

Variable Ordering and Value Instantiation

If backtracking is used to solve CSP, then another issue is the order in which variables are considered for instantiations. Experiments and analysis by several researchers show that the ordering in which variables are chosen for instantiation can have substantial impact on

A set of variables with no direct constraints between any pair of them is called a stable set *of variables.*

the complexity of backtrack search (Bitner and Reingold 1975; Purdom 1983; Stone and Stone 1986; Haralick and Elliot 1980; Zabih and McAllester 1988). Several heuristics have been developed and analyzed for selecting variable ordering. One powerful heuristic, developed by Bitner and Reingold (1975), is often used with the FC algorithm. In this method, the variable with the fewest possible remaining alternatives is selected for instantiation. Thus, the order of variable instantiation is, in general, different in different branches of the tree and is determined dynamically. This heuristic is called the *search-rearrangement method*. Purdom and Brown extensively studied this heuristic, as well as its variants, both experimentally and analytically (Purdom 1983; Purdom and Brown 1981, 1982; Purdom, Brown, and Robertson 1981). Their results show that for significant classes of problems, search-rearrangement backtracking is a substantial improvement over the standard backtracking method. For the *n*-queens problem, Stone and Stone (1986) experimentally show that the search-rearrangement heuristic led to an improvement of dozens of orders of magnitude for large values of *n*. With this heuristic, they were able to solve the problem for $n \leq 96$ using only a personal computer. The standard backtracking method could not solve the problem in a reasonable amount of time even for $n = 30$. Nadel (1983) presents an analytic framework that can be used to analyze the expected complexities of various search orders and select the best one. Feldman and Golumbic (1989) apply these ideas to the student scheduling problem and suggest some further extensions.

Another possible heuristic is to instantiate those variables first that participate in the highest number of constraints. This approach tries to ensure that the unsuccessful branches of the tree are pruned early. Freuder and Quinn (1985) discuss an ordering that is somewhat related to this heuristic. A set of variables with no direct constraints between any pair of them is called a *stable set* of variables. In this heuristic, the backtracking algorithm instantiates all the members of a stable set at the end. The instantiation of these variables

contributes to the search space only additively (that is, not multiplicatively, which is usually the case). If the constraint graph has *n* vertices, and a stable set of *m* vertices exists, then the overall complexity of the backtrack is bounded by dn-$m \star md$ as opposed to d^n. Hence, it makes sense to find the maximal stable set of the constraint graph before deciding on the order of instantiation. Unfortunately, the problem of finding a maximal stable set is NP-hard. Thus, one has to settle for a heuristic algorithm that finds a suboptimal stable set. Fox, Sadeh, and Baykan (1989) use many different structural characteristics of CSP to select variable order and show its utility in the domains of spatial planning and factory scheduling.

Recall that the tree-structured constraint graphs can be solved without backtracking simply at the cost of achieving arc consistency. Any given constraint graph can be made a tree after deleting certain vertices such that all the cycles from the graph are removed. This set of vertices is called the *cycle cutset*. If a small cycle cutset can be found, then a good heuristic is to first instantiate all the variables in the cycle cutset and then solve the resulting tree-structured CSPs without backtracking (Dechter 1986). If the size of a cycle cutset of an *n*-variable CSP is *m*, then the original CSP can be solved in $d^m + (n - m) \star d^2$ steps.

Once the decision is made to instantiate a variable, it can have several values available. The order in which these values are considered can have substantial impact on the time to find the first solution. For example, if CSP has a solution, and a correct value is chosen for each variable, then a solution can be found without any backtracking. One possible heuristic is to prefer those values that maximize the number of options available for future assignments (Haralick and Elliot 1980). By incorporating such a value-ordering heuristic in Stone and Stone's algorithm for solving the *n*-queens problem, Kale (1990) developed a backtracking-based algorithm that can be used to solve the problem with little backtracking even for large values of *n* (= 1000). Without incorporating Kale's heuristic, Stone and Stone's algorithm is unable to solve the *n*-queens problem for *n* much larger than 100.

Minton, Johnston, Philips, and Laird (1990) use similar value- and variable-ordering heuristics in a somewhat different framework to obtain solutions to the *n*-queens problem for $n = 1,000,000$. In their scheme, backtracking starts after a good initial assignment of values to the variables in CSP (that is, an assignment to the variables that violates only

a few of the constraints) has been obtained through some greedy algorithm. Now the values of the variables that conflict with other variables are systematically changed in the backtracking paradigm. Minton and his colleagues present empirical results using this scheme for many problems as well as an analytic model that explains the performance of this scheme with different kinds of problems. This method was originally developed as a *hill-climbing method* (that is, nonbacktracking method). It starts with a reasonable assignment of values to variables and then continues to repair the values of variables until a correct solution is obtained. Sosic and Gu also developed a similar algorithm (Gu 1989).

Another heuristic is to prefer the value (from those available) that leads to CSP that is easiest to solve. Dechter and Pearl (1988a) discuss one way of estimating the difficulty of solving CSP. In this method, CSP is converted into a tree-structured CSP by deleting a minimum number of arcs; resulting CSP is solved for all solutions. The number of solutions found in corresponding tree-structured CSP is taken as the measure of difficulty of solving CSP (higher the solution count, easier CSP). They also present an experimental evaluation of this heuristic, as well as its variations, on randomly generated CSPs and show that a variation of this heuristic helps in reducing the overall search effort. Good value-ordering heuristics are expected to be highly problem specific. For example, Sadeh (1991) shows that Dechter and Pearl's value-ordering heuristic performs poorly for the job shop scheduling problem. Sadeh presents other variable- and value-ordering heuristics that work well for this problem.

Concluding Remarks

CSP can always be solved by the standard backtracking algorithm, although at substantial cost. The reason for the poor performance of backtracking is that it does not learn from the failure of different nodes. The performance of a backtracking algorithm can be improved in a number of ways: (1) performing constraint propagation at the search nodes of the tree, (2) performing reason maintenance or just intelligent backtracking, and (3) choosing a good variable ordering or a good order for the instantiation of different values of a given variable. By performing constraint propagation, given CSP is essentially transformed into different CSP whose search space is smaller. In the process of constraint propagation, certain failures are identified, and the search space is effectively reduced so that these failures are

not encountered at all in the search space of transformed CSP. In the second technique, CSP is not transformed into a different problem, but the search space is searched carefully. Information about a failure is kept and used during the search of the remaining space. Based on previous failure information, whenever it is determined that search in some new subspace will also fail, then this subspace is also pruned. Good variable ordering reduces the bushiness of the tree by moving the failures to upper levels of the search tree. Good value ordering moves a solution of CSP to the left of the tree so that it can be found quickly by the backtracking algorithm. If applied to an extreme, any of these techniques can eliminate the thrashing behavior of backtracking. However, the cost of applying these techniques in this manner is often more than that incurred by simple backtracking. It turns out that simplified versions of these techniques can be used together to reduce the overall search space. The optimal combination of these techniques is different for different problems and is a topic of current investigation (Dechter and Meiri 1989).

References

Allen, J. 1984. Toward a General Theory of Action and Time. *Artificial Intelligence* 23(2): 123–154.

Allen, J. 1983. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM* 26:832–843.

Bitner, J., and Reingold, E. M. 1975. Backtrack Programming Techniques. 18:651–655.

Bruynooghe, M. 1985. Graph Coloring and Constraint Satisfaction, Technical Report, CW 44, Department Computerwetenschappen, Katholieke Universiteit Leuven.

Bruynooghe, M. 1981. Solving Combinatorial Search Problems by Intelligent Backtracking. *Information Processing Letters* 12(1): 36–39.

Bruynooghe, M., and Pereira, L. M. 1984. Deduction Revision by Intelligent Backtracking. In *Implementations of Prolog*, ed. J. A. Campbell, 194–215. Chichester, England: Ellis Horwood.

Chakravarty, I. 1979. A Generalized Line and Junction Labelling Scheme with Applications to Scene Analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1(2): 202–205.

Chen, Y. 1991. Improving Han and Lee's Path-Consistency Algorithm. In Proceedings of the Third

International Conference on Tools for AI, 346–350. Washington, D.C.: IEEE Computer Society.

Cooper, M. C. 1989. An Optimal k-Consistency Algorithm. Artificial Intelligence 41:89–95.

Davis, A. L., and Rosenfeld, A. 1981. Cooperating Processes for Low-Level Vision: A Survey. *Artificial Intelligence* 17:245–263.

Dechter, R. 1990. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence* 41(3): 273–312.

Dechter, R. 1987. A Constraint-Network Approach to Truth Maintenance, Technical Report, R-870009, Cognitive Systems Laboratory, Computer Science Dept., Univ. of California at Los Angeles.

Dechter, R. 1986. Learning While Searching in Constraint-Satisfaction Problems. In Proceedings of the Fifth National Conference on Artificial Intelligence, 178–183. Menlo Park, Calif.: American Association for Artificial Intelligence.

Dechter, R., and Dechter, A. 1988. Belief Maintenance in Dynamic Constraint Networks. In Proceedings of the Seventh National Conference on Artificial Intelligence, 37–42. Menlo Park, Calif.: American Association for Artificial Intelligence.

Dechter, R., and Meiri, I. 1989. Experimental Evaluation of Preprocessing Techniques in Constraint-Satisfaction Problems. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 290–296. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Dechter, R., and Pearl, J. 1988a. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence* 34:1–38.

Dechter, R., and Pearl, J. 1988b. Network-Based Heuristics for Constraint-Satisfaction Problems. In *Search in Artificial Intelligence*, eds. L. Kanal and V. Kumar, 370–425. New York: Springer-Verlag.

Dechter, R., and Pearl, J. 1988c. Tree-Clustering Schemes for Constraint Processing. In Proceedings of the Seventh National Conference on Artificial Intelligence, 150–154. Menlo Park, Calif.: American Association for Artificial Intelligence.

Dechter, R., and Pearl, J. 1986. The Cycle-Cutset Method for Improving Search Performance in AI Applications, Technical Report, R-16, Cognitive Systems Laboratory, Computer Science Dept., Univ. of California at Los Angeles.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal Constraint Networks. *Artificial Intelligence* 49:61–95.

Dechter, R.; Meiri, I.; and Pearl, J. 1990. Tree Decomposition with Applications to Constraint Processing. In Proceedings of the Eighth National Conference on Artificial Intelligence, 10–16. Menlo Park, Calif.: American Association for Artificial Intelligence.

de Kleer, J. 1989. A Comparison of ATMS and CSP Techniques. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 290-296. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

de Kleer, J. 1986a. An Assumption-Based TMS. *Artificial Intelligence* 28:127–162.

de Kleer, J. 1986b. Problems with ATMS. *Artificial Intelligence* 28:197–224.

de Kleer, J., and Sussman, G. J. 1980. Propagation of Constraints Applied to Circuit Synthesis. *Circuit Theory and Applications* 8:127–144.

Dhar, V., and Croker, A. 1990. A Knowledge Representation for Constraint-Satisfaction Problems, Technical Report, 90-9, Dept. of Information Systems, New York Univ.

Dhar, V., and Ranganathan, N. 1990. Integer Programming versus Expert Systems: An Experimental Comparison. *Communications of the ACM* 33:323–336.

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.

Feldman, R., and Golumbic, M. C. 1989. Constraint Satisfiability Algorithms for Interactive Student Scheduling. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 1010–1016. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Fikes, R. E. 1970. REF-ARF: A System for Solving Problems Stated as Procedures. *Artificial Intelligence* 1(1): 27–120.

Fox, M. S. 1987. *Constraint-Directed Search: A Case Study of Job Shop Scheduling.* San Mateo, Calif.: Morgan Kaufmann.

Fox, M. S.; Sadeh, N.; and Baykan, C. 1989. Constrained Heuristic Search. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 309–315. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Frayman, F., and Mittal, S. 1987. COSSACK: A Constraint-Based Expert System for Configuration Tasks. In *Knowledge-Based Expert Systems in Engineering: Planning and Design*, eds. D. Sriram and R. A. Adey, 143–166. Billerica, Mass.: Computational Mechanics Publications.

Freeman-Benson, B. N.; Maloney, J.; and Borning, A. 1990. An Incremental Constraint Solver. *Communications of the ACM* 33:54–63.

Freuder, E. 1990. Complexity of *K*-Tree–Structured Constraint-Satisfaction Problems. In Proceedings of the Eighth National Conference on Artificial Intelligence, 4–9. Menlo Park, Calif.: American Association for Artificial Intelligence.

Freuder, E. 1989. Partial Constraint Satisfaction. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 278–283. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Freuder, E. 1988. Backtrack-Free and Backtrack-Bounded Search. In *Search in Artificial Intelligence*, eds. L. Kanal and V. Kumar, 343–369. New York: Springer-Verlag.

Freuder, E. 1982. A Sufficient Condition for Back-

track-Free Search. *Journal of the ACM* 29(1): 24–32.

Freuder, E. 1978. Synthesizing Constraint Expression. *Communications of the ACM* 21(11): 958–966.

Freuder, E., and Quinn, M. 1985. Taking Advantage of Stable Sets of Variables in Constraint-Satisfaction Problems. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence, 1076–1078. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Gaschnig, J. 1979. Performance Measurement and Analysis of Certain Search Algorithms. Ph.D. diss., Dept. of Computer Science, Carnegie Mellon Univ.

Gaschnig, J. 1978. Experimental Case Studies of Backtrack versus Waltz-Type versus New Algorithms for Satisfying Assignment Problems. In Proceedings of the Second Biennial Conference of the Canadian Society for Computational Studies of Intelligence. Toronto: Canadian Information Processing Society.

Gaschnig, J. 1977. A General Backtrack Algorithm That Eliminates Most Redundant Tests. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 457–457. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Gaschnig, J. 1974. A Constraint-Satisfaction Method for Inference Making. In Proceedings of the Twelfth Annual Allerton Conference on Circuit Systems Theory, 866–874. Urbana, Ill.: Univ. of Illinois at Urbana-Champaign.

Geffner, H., and Pearl, J. 1987. An Improved Constraint-Propagation Algorithm for Diagnosis. In Proceedings of the Tenth International Joint Conference on Artificial Intelligence, 1105–1111. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Gu, J. 1989. Parallel Algorithms and Architectures for Very Fast AI Search. Ph.D. diss., Computer Science Dept., Univ. of Utah.

Han, C. C., and Lee, C. H. 1988. Comments on Mohr and Henderson's Path-Consistency Algorithm. *Artificial Intelligence* 36:125–130.

Haralick, R., and Elliot, G. 1980. Increasing Tree Search Efficiency for Constraint-Satisfaction Problems. *Artificial Intelligence* 14(3): 263–313.

Haralick, R.; Davis, L. S.; and Rosenfeld, A. 1978. Reduction Operations for Constraint Satisfaction. *Information Science* 14:199–219.

Hummel, R. A.; Rosenfeld, A.; and Zucker, S. W. 1976. Scene Labelling by Relaxation Operations. *IEEE Transactions on Systems, Man, and Cybernetics* 6(6): 420–433.

Kale, L. V. 1990. A Perfect Heuristic for the *N* Non-Attacking Queens Problem. *Information Processing Letters* 34(4): 173–178.

Kumar, V. 1987. Depth-First Search. In *Encyclopaedia of Artificial Intelligence*, volume 2, ed. S. C. Shapiro, 1004–1005. New York: Wiley.

Kumar, V., and Lin, Y. 1988. A Data-Dependency–Based Intelligent Backtracking Scheme for Prolog. *The Journal of Logic Programming* 5(2): 165–181.

McDermott, D. 1991. A General Framework for Reason Maintenance. *Artificial Intelligence* 50:289–329.

McGregor, J. 1979. Relational Consistency Algorithms and Their Applications in Finding Subgraph and Graph Isomorphism. *Information Science* 19:229–250.

Mackworth, A. K. 1977a. Consistency in Networks of Relations. *Artificial Intelligence* 8(1): 99–118.

Mackworth, A. K. 1977b. On Reading Sketch Maps. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 598–606. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Mackworth, A. K., and Freuder, E. 1985. The Complexity of Some Polynomial Network-Consistency Algorithms for Constraint-Satisfaction Problems. *Artificial Intelligence* 25:65–74.

Mackworth, A. K.; Mulder, J. A.; and Havens, W. S. 1985. Hierarchical Arc Consistency: Exploiting Structured Domains in Constraint Satisfaction. *Computational Intelligence* 1(3): 118–126.

Minton, S.; Johnston, M.; Phillips, A.; and Laird, P. 1990. Solving Large-Scale Constraint-Satisfaction and Scheduling Problems Using a Heuristic Repair Method. In Proceedings of the Eighth National Conference on Artificial Intelligence, 17–24. Menlo Park, Calif.: American Association for Artificial Intelligence.

Mittal, S., and Falkenhainer, B. 1990. Dynamic Constraint-Satisfaction Problems. In Proceedings of the Eighth National Conference on Artificial Intelligence, 25–32. Menlo Park, Calif.: American Association for Artificial Intelligence.

Mittal, S., and Frayman, F. 1987. Making Partial Choices in Constraint-Reasoning Problems. In Proceedings of the Sixth National Conference on Artificial Intelligence, 631–636. Menlo Park, Calif.: American Association for Artificial Intelligence.

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 Science* 7:95–132.

Montanari, U., and Rossi, F. 1991. Constraint Relaxation May Be Perfect. *The Journal of Artificial Intelligence* 48:143–170.

Nadel, B. 1990. Some Applications of the Constraint-Satisfaction Problem, Technical Report, CSC-90-008, Computer Science Dept., Wayne State Univ.

Nadel, B. 1988. Tree Search and Arc Consistency in Constraint-Satisfaction Algorithms. In *Search in Artificial Intelligence*, eds. L. Kanal and V. Kumar, 287–342. New York: Springer-Verlag.

Nadel, B. 1983. Consistent-Labeling Problems and Their Algorithms: Expected Complexities and Theory-Based Heuristics. *Artificial Intelligence* 21:135–178.

Nadel, B., and Lin, J. 1991. Automobile Transmission Design as a Constraint-Satisfaction Problem: Modeling the Kinematic Level. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing* 5(2). Forthcoming.

Navinchandra, D. 1990. *Exploration and Innovation in Design*. New York: Springer-Verlag.

Navinchandra, D., and Marks, D. H. 1987. Layout Planning as a Consistent Labelling Optimization Problem. Presented at the Fourth International Symposium on Robotics and AI in Construction, Haifa, Israel.

Pereira, L. M., and Porto, A. 1982. Selective Backtracking. In *Logic Programming*, eds. K. Clark and Sten-Ake Tarnlund, 107–114. Boston: Academic.

Perlin, M. 1991. Arc Consistency for Factorable Relations. In Proceedings of the Third International Conference on Tools for AI, 340–345. Washington, D.C.: IEEE Computer Society.

Petrie, C. 1987. Revised Dependency-Directed Backtracking for Default Reasoning. In Proceedings of the Sixth National Conference on Artificial Intelligence, 167–172. Menlo Park, Calif.: American Association for Artificial Intelligence.

Petrie, C.; Causey, R.; Steiner, D.; and Dhar, V. 1989. A Planning Problem: Revisable Academic Course Scheduling, Technical Report AI-020-89, MCC Corp., Austin, Texas.

Prosser, P. 1991. Hybrid Algorithms for the Constraint-Satisfaction Problem, Research Report, AISL-46-91, Computer Science Dept., Univ. of Strathclyde.

Prosser, P. 1989. A Reactive Scheduling Agent. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 1004–1009. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Purdom, P. 1983. Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence* 21:117–133.

Purdom, P., and Brown, C. 1982. An Empirical Comparison of Backtracking Algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence on Pattern Analysis and Machine Intelligence* PAMI-4: 309–316.

Purdom, P., and Brown, C. 1981. An Average Time Analysis of Backtracking. *SIAM Journal of Computing* 10(3): 583–593.

Purdom, P.; Brown, C.; and Robertson, E. L. 1981. Backtracking with Multi-Level Dynamic Search Rearrangement. *Acta Informatica* 15:99–113.

Ricci, F. 1990. Equilibrium Theory and Constraint Networks, Technical Report 9007-08, Istituto per la Ricerca Scientificae Tecnologica, Povo, Italy.

Rit, J. F. 1986. Propagating Temporal Constraints for Scheduling. In Proceedings of the Fifth National Conference on Artificial Intelligence, 383–386. Menlo Park, Calif.: American Association for Artificial Intelligence.

Rosiers, W., and Bruynooghe, M. 1986. Empirical Study of Some Constraint-Satisfaction Problems. In *Proceedings of AIMSA 86*. New York: Elsevier.

Rossi, F.; Petrie, C.; and Dhar, V. 1989. On the Equivalence of Constraint-Satisfaction Problems, Technical Report ACT-AI-222-89, MCC Corp., Austin, Texas.

Sadeh, N. 1991. Look-Ahead Techniques for Micro-Opportunistic Job Shop Scheduling. Ph.D. diss., CMU-CS-91-102, Computer Science Dept., Carnegie Mellon Univ.

Shapiro, L., and Haralick, R. 1981. Structural Descriptions and Inexact Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 3(5): 504–518.

Stallman, R., and Sussman, G. J. 1977. Forward Reasoning and Dependency-Directed Backtracking. *Artificial Intelligence* 9(2): 135–196.

Stefik, M. 1981. Planning with Constraints (MOLGEN: Part I). *Artificial Intelligence* 16:111–140.

Stone, H. S., and Stone, J. 1986. Efficient Search Techniques: An Empirical Study of the *N*-Queens Problem, Technical Report RC 12057, IBM T. J. Watson Research Center, Yorktown Heights, New York.

Tsang, E. P. K. 1987. The Consistent Labeling Problem in Temporal Reasoning. In Proceedings of the Sixth National Conference on Artificial Intelligence, 251–255. Menlo Park, Calif.: American Association for Artificial Intelligence.

Ullman, J. R. 1976. An Algorithm for Subgraph Isomorphism. *Journal of the ACM* 23:31–42.

Vilain, M., and Kautz, H. 1986. Constraint-Propagation Algorithms for Temporal Reasoning. In Proceedings of the Fifth National Conference on Artificial Intelligence, 377–382. Menlo Park, Calif.: American Association for Artificial Intelligence.

Waltz, D. 1975. Understanding Line Drawings of Scenes with Shadows. In *The Psychology of Computer Vision*, ed. P. H. Winston, 19–91. Cambridge, Mass.: McGraw Hill.

Zabih, R. 1990. Some Applications of Graph Bandwidth to Constraint-Satisfaction Problems. In Proceedings of the Seventh National Conference on Artificial Intelligence, 46–51. Menlo Park, Calif.: American Association for Artificial Intelligence.

Zabih, R., and McAllester, D. 1988. A Rearrangement Search Strategy for Determining Propositional Satisfiability. In Proceedings of the Seventh National Conference on Artificial Intelligence, 155–160. Menlo Park, Calif.: American Association for Artificial Intelligence.

Vipin Kumar is an associate professor in the department of computer sciences at the University of Minnesota. His current research interests include parallel processing and AI. He is author of over 50 journal and conference articles on these topics and is coeditor of *Search in Artificial Intelligence* (Springer-Verlag, 1988) and *Parallel Algorithms for Machine Intelligence and Vision* (Springer-Verlag, 1990).