CONSTRAINT NETWORKS[1]

Rina Dechter
Information and Computer Science
University of California
Irvine, CA 92717-3425

1 Introduction

Constraint-based reasoning is a paradigm for formulating knowledge as a set of constraints without specifying the method by which these constraints are to be satisfied. A variety of techniques have been developed for finding partial or complete solutions for different kinds of constraint expressions. These have been successfully applied to diverse tasks such as design, diagnosis, truth maintenance, scheduling, spatiotemporal reasoning, logic programming and user interface. *Constraint networks* are graphical representations used to guide strategies for solving *constraint satisfaction problems (CSPs).*

1.1 Basic definitions

A *constraint network (CN)* consists of a finite set of *variables* $X = \{X_1, \ldots, X_n\}$, each associated with a *domain* of discrete values, $D_1, \ldots, D_n$ and a set of *constraints*, $\{C_1, \ldots, C_t\}$. Each of the constraints is expressed as a relation, defined on some subset of variables, whose tuples are all the simultaneous value assignments to the members of this variable subset that, as far as this constraint alone is concerned, are legal[2] Formally, a constraint C_i has two parts: (1) the subset of variables $S_i = \{X_{i_1}, \ldots, X_{i_{j(i)}}\}$, on which it is defined, called a *constraint-subset*, and (2) a *relation*, rel_i defined over $S_i : rel_i \subseteq D_{i_1} \times \cdots \times D_{i_{j(i)}}$. Because many properties of a CN depend on the structure of the constraint subsets, the *scheme* of a CN is defined as the set of subsets on which constraints are defined, namely, $scheme(CN) = \{S_1, S_2, \ldots, S_t\}, S_i \subseteq X$. The projection of a relation ρ on a subset of variables $U = U_1, \ldots, U_l$ is given by $\Pi_U(\rho) = \{x_u = (x_{u1}, \ldots, x_{ul}) \mid \exists\, \overline{x} \in \rho, \overline{x}$ is an extension of $x_u\}$.

[1]Published in the Encyclopedia of Artificial Intelligence, second edition, Wiley and Sons, pp 276-285, 1992

[2]This does not mean that the actual representation of any constraint is necessarily in the form of its defining relation, but that the relation can, in principle, be generated using the constraint's specification without the need to consult other constraints in the network.

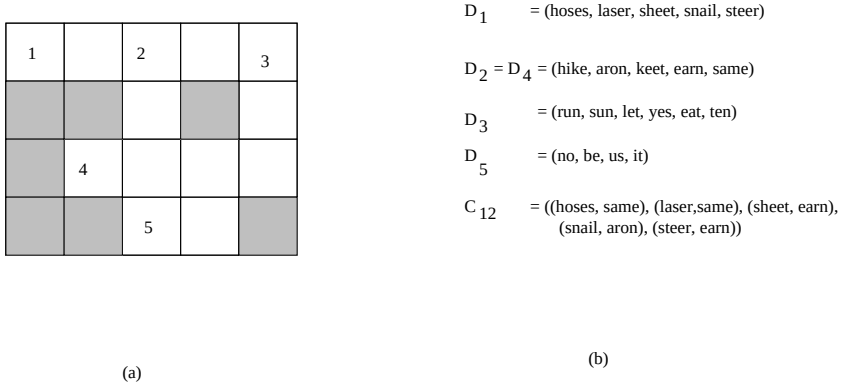

Figure 1: A crossword puzzle and its CN representation.

An assignment of a unique domain value to each member of some subset of variables is called an *instantiation*. An instantiation is said to satisfy a given constraint C_i if the partial assignment specified by the instantiation does not violate C_i (i.e., it belongs to the projection of rel_i on the common variables). An instantiation is said to be *legal* or *locally consistent* if it satisfies *all* the (relevant) constraints of the network.

A legal instantiation of *all* the variables of a constraint network is called a *solution* of the network, and the set of all solutions is a relation, ρ, defined on the set of all variables. This relation is said to be *represented* by the constraint network. Formally,

$$\rho = \{(X_1 = x_1, \ldots, X_n = x_n) \mid \forall\, S_i \in scheme,\ \Pi_{S_i}\rho \subseteq rel_i\}$$

Example 1: Figure 1a presents a simplified version of a crossword puzzle (see *constraint satisfaction*). The variables are X_1 (1, horizontal), X_2 (2, vertical), X_3 (3, vertical), X_4 (4, horizontal), and X_5 (5, horizontal). The scheme of this problem is $\{X_1X_2, X_1X_3, X_4X_2, X_4X_3, X_5X_2\}$. The domains and some constraints are specified in Figure 1b. A tuple in the relation associated with this puzzle is the solution: $(X_1 = sheet, X_2 = earn, X_3 = ten, X_4 = aron, X_5 = no)$.

Typical tasks defined in connection with constraint networks are to determine whether a solution exists, to find one or all of the solutions, to determine whether an instantiation of some subset of the variables is a partial solution (i.e., is part of a global solution), etc. These tasks are collectively called *constraint satisfaction problems*.

Techniques used in processing constraint networks can be classified into three categories. The first category consists of search techniques for systematic exploration of the space of all solutions. The most common algorithm in this class is *backtracking* which traverses the

search space in a depth-first fashion. The second category is *consistency algorithms* for transforming a CN into more explicit representation. These are used primarily in a preprocessing phase, to improve the performance of the subsequent backtracking search, but can be incorporated into the search procedure itself. Third are the *structure-driven algorithms*, which exploit the topological features of the network to guide the search. Structure-driven algorithms can support both the consistency algorithms as well as the backtracking search.

This survey concentrates on techniques of the third kind, namely, structure-based algorithms. These together with *backtracking* and *consistency algorithms* (see *constraint satisfaction*) give a complete picture of the available techniques. A brief summary of *backtracking* and *consistency enforcing procedures* is presented next.

2 Backtracking and Consistency-Enforcing Strategies

The standard solution procedure for solving constraint satisfaction problems is backtracking search. The algorithm typically considers the variables in some order and, starting with the first, assigns a provisional value to each successive variable in turn as long as the assigned values are consistent with those assigned in the past. When, in the process, a variable is encountered such that none of its domain values are consistent with previous assignments (a situation referred to as a *dead-end*), backtracking takes place. That is, the value assigned to the immediately preceding variable is replaced, and the search continues in a systematic way until either a solution is found or until it may be concluded that no such solution exists.

Improving backtracking efficiency amounts to reducing the size of its expanded search space. This depends on the way the constraints are represented, (i.e., on the extent of their explicitness), the order of variables instantiation, and, when one solution suffices, on the order in which values are assigned to each variable.

Using these factors to improve the performance of backtracking algorithms, researchers have developed procedures of two types: those that are employed *in advance of* performing the search, and those that are used *dynamically* during search. The former include a variety of *consistency-enforcing* algorithms. [Montanari 74, Mackworth & Freuder 84a, Freuder 85] These transform a given constraint network into an equivalent, yet more explicit, network by deducing new constraints to be added on to the network.

Intuitively, a *consistency-enforcing* algorithm will make any partial solution of a small subnetwork extensible to some surrounding network. For example, the most basic consistency algorithm, called *arc-consistency* or *two-consistency* (also known as *constraint propagation*

and *constraint relaxation*), ensures that any legal value in the domain of a single variable has a legal match in any other selected variable. *Path-consistency* (or three-consistency) algorithms ensure that any consistent solution to a *two-variable* subnetwork is extensible to any third variable, and, in general, *i-consistency* algorithms guarantee that any *locally consistent* instantiation of $i-1$ variables is extensible to any i^{th} variable.

Deciding the level of consistency that should be enforced on the network is not a clear-cut choice. Generally speaking, backtracking will benefit from representations that are as explicit as possible, having higher consistency level. However, the complexity of enforcing i-consistency is exponential in i. As a result, there is a trade-off between the effort spent on preprocessing and that spent on search (backtracking.) Experimental analyses of this trade-off have been published [Dechter & Meiri 89, Dechter 90, Haralick & Elliott 80].

Variable orderings' decisions have also received much consideration, and several heuristics have been proposed [Freuder 82, Dechter & Pearl 89], all following the intuition that tightly *constrained* variables should come first. Strategies for *dynamically* improving the pruning power of backtracking can be conveniently classified as *look-ahead schemes* and *look-back schemes*. Look-ahead schemes are invoked whenever the algorithm is preparing to assign a value to the next variable. Some of the functions that such schemes perform are:

1. Calculate and record the way in which the current instantiations restrict future variables. This process has been referred to as constraint propagation. Examples include Waltz's algorithm [Waltz 75] and forward checking [Haralick & Elliott 80].

2. Decide which variable to instantiate next (when the order is not predetermined). Generally, it is advantageous to first instantiate variables that maximally constrain the rest of the search space. Therefore, the variable participating in the highest number of constraints is usually selected. [Freuder 82, Purdom 83, Stone & Stone 86]

3. Decide which value to assign to the next variable (when there is more than one candidate). Generally, for finding one solution, an attempt is made to assign a value that maximizes the number of options available for future assignments [Haralick & Elliott 80, Dechter & Pearl 87].

Look-back schemes are invoked when the algorithm encounters a dead-end and prepares for the backtracking step. These schemes perform two functions:

1. Decide how far to backtrack. By analyzing the reasons for the dead-end, it is often possible to go back directly to the source of failure instead of to the immediate predecessor in the ordering. This idea is often referred to as *backjumping* [Gaschnig 79].

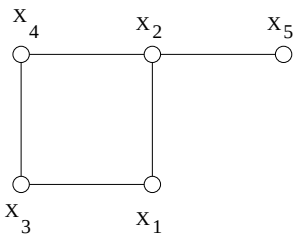

Figure 2: A constraint graph of the crossword puzzle.

2. Record the reasons for the dead-end in the form of new constraints so that the same conflicts will not arise again in a later search. Terms used to describe this idea are *constraint recording* and *no-good* constraints. *Dependency-directed backtracking* incorporates both backjumping and no-goods recording [Stallman & Sussman 77]. Constraint recording can also be viewed as a form of explanation-based learning (EBL).

3 Graph-Based Algorithms

3.1 Graphical representations

Graphical properties of CN were initially investigated through the class of *binary constraint networks*. [Freuder 82] A *binary constraint network* is one in which every *constraint subset* involves at most two variables. In this case the network can be associated with a constraint graph, where each node represents a variable, and the arcs connect nodes whose variables are explicitly constrained; namely, they are members of the network's *scheme*. Figure 2 shows the constraint graph associated with the crossword puzzle in Figure 1.

A graphical representation of higher order networks can be provided by *hypergraphs*, where again, nodes represent the variables, and *hyperarcs* (drawn as regions) group those variables that belong to the same constraint. Two variations of this representation that can be used to facilitate structure-driven algorithms are *primal-constraint graph* and *dual-constraint graph*. A *Primal-constraint graph* (a generalization of the binary constraint graph) represents variables by nodes and associates an arc with any two nodes residing in the same constraint. A *dual-constraint-graph* represents each *constraint subset* by a node (also called a *c-variable*) and associates a labeled arc with any two nodes whose *constraint subsets* share variables. The arcs are labeled by the shared variables.

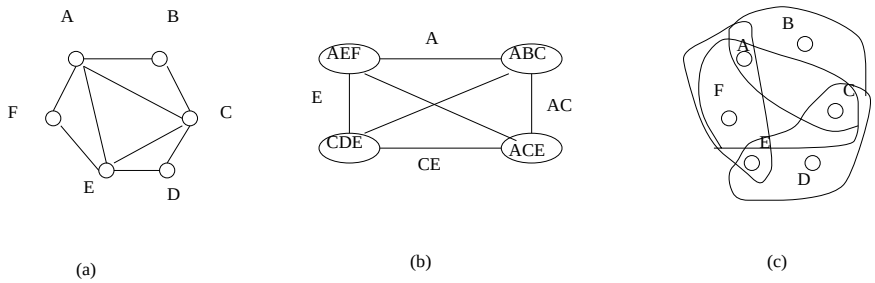

Figure 3: Primal and dual constraint graphs of a CSP.

For example, Figure 3 depicts the *primal*, the *dual*, and the *hypergraph* representations of a CN with variables A, B, C, D, E, F and constraints on the subsets (ABC),(AEF), (CDE) and (ACE). The constraints themselves are symbolically given by the inequalities: $A + B \leq C$, $A + E \leq F$, $C + D \leq E$, $A + C \leq E$, where the domains of each variable are the integers $[2, 3, 4, 5, 6]$.

The *dual* constraint graph can be viewed as a transformation of a nonbinary network into a special type of *binary* network: the domain of the c-variables ranges over all possible value combinations permitted by the corresponding constraints, and any two adjacent c-variables must obey the restriction that their shared variables should have the same values (i.e., the c-variables are bounded by equality constraints). For instance, the domain of the the c-variable ABC is {224, 225, 226, 235, 236, 325, 326, 246, 426, 336} and the binary constraint between ABC and CDE is given by the relation: $rel_{ABC,CDE} = \{(224, 415), (224, 426)\}$. Viewed in this way, any network can be solved by binary networks' techniques.

3.2 Solving Tree-Networks

Almost all the known structure-based techniques rely on the observation that *binary* constraint networks whose constraint graph is a *tree* can be solved in linear time [Freuder 82, Mackworth & Freuder 84b, Dechter & Pearl 87]. The solution of tree-structured networks are discussed, and later it is shown how they can be used to facilitate the solution of general CN.

Given a tree-network over n variables (Fig. 4a), the first step of the *tree-algorithm* is to generate a *rooted-directed* tree. Each node in this tree (excluding the root) has one *parent node* directed toward it and may have several *child* nodes, directed away from it. Nodes with no *children* are called *leaves*. An ordering, $d = X_1, X_2, \ldots, X_n$, is then enforced such that a parent always precedes its children. In the second step, the algorithm processes

each arc (and its associated constraint) from leaves to root, in an orderly layered fashion. For each directed arc from X_i to X_j it removes a value from the domain of X_i if it has *no consistent match* in the domain of X_j. Finally, after the root is processed, a backtracking algorithm is used to find a solution along the ordering d.

It can be shown that the algorithm is linear in the number of variables. In particular, backtracking, which in general is an exponential procedure, is guaranteed to find a solution without facing any dead-ends. The tree algorithm is sketched by the following procedures:

Tree-Algorithm (T)
1. begin
2. generate a *rooted tree* ordering, $d = X_1, \ldots, X_n$.
3. for i=n to 1 by -1 do
4. *revise* $(X_{p(i)}, X_i); X_{p(i)})$ denotes the parent of X_i.
5. if the domain of $X_{p(i)}$ is empty, stop. (no solution exists).
6. end
7. use backtracking to instantiate variables along d.
8. end.

The *revise* procedure [Mackworth & Freuder 84a] is defined by:

Revise(X_j, X_i)

1. begin
2. for each $v \in D_j$ do
3. if there is no $u \in D_i s.t.(X_j = v, X_i = u)$ is consistent,
4. delete v from D_j.
5. end.
6. end.

The complexity of the *tree-consistency* algorithm is bounded by $O(nk^2)$ steps where k bounds the domain size, because an ordering (step 2) can be produced in linear time, whereas the *revise* procedure, which is bounded by k^2 steps, is executed at most n times (loop 3-6).

The tree-algorithm is an instance of a general classes of *ordered* algorithms, to be discussed next.

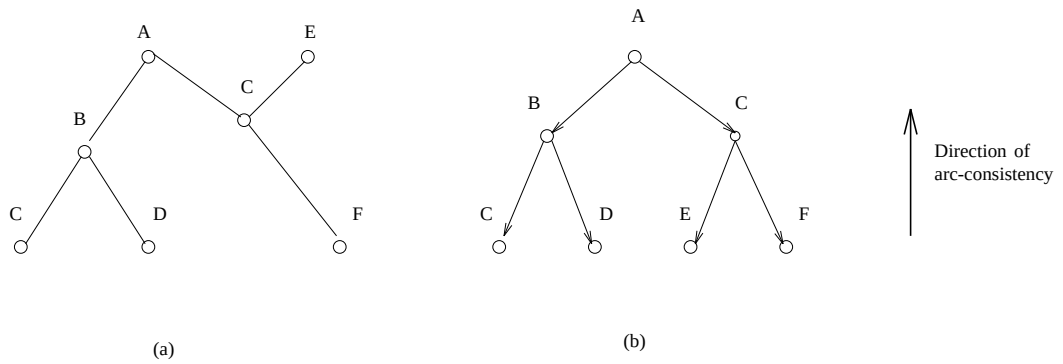

Figure 4: A tree network

4 Directional and Adaptive-Consistency

In general, a problem is considered *easy* when it admits a solution in polynomial time. In the context of *constraint networks*, a problem is easy if an algorithm like backtracking can solve it in a *backtrack-free* manner, i.e., without dead-ends, thus producing a solution in time linear in the number of variables and constraints. This concept has prompted a theoretical investigation (see Freuder [Freuder 82, Freuder2 85] and Dechter and Pearl) [Dechter & Pearl 87, Dechter & Pearl 89], into the level of *local consistency* that suffices for ensuring a *backtrack-free search*. The theory had identified topological features that determine this level of consistency, and has yielded tractable algorithms for transforming some networks into backtrack-free representations. The following paragraphs present a summary of this theory.

The theory is centered on a graphical parameter called *width*, and the definitions are relative to the *primal* constraint graph. An *ordered (primal) constraint graph* is defined as one in which the nodes are linearly ordered to reflect the sequence of variable assignments executed by *backtracking* algorithm. The *width of a node* is the number of arcs that connect that node to previous ones, the *width of an ordering* is the maximum width of all nodes, and the *width of a graph* is the minimum width of all orderings of that graph.

Figure 5 presents three possible orderings of the constraint graph of Figure 2. The width of node X_2 in the first ordering (from the left) is three, whereas in the second ordering it is two. It can be shown that no ordering can achieve width lower than two, hence the width of this constraint graph is two. (The graph has cycle, and it is known that *only trees are width-one graphs* [Freuder 82].)

The width of a graph can be determined by a greedy algorithm. The algorithm selects a

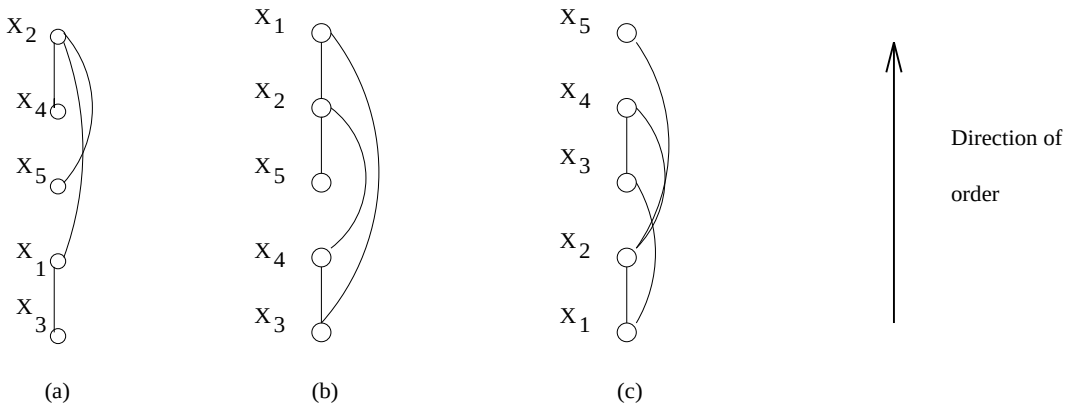

Figure 5: Three orderings of a constraint graph, representing widths of 3, 2, and 2, respectively.

node having the least number of neighbors and puts it last in the ordering. This node is then removed (together with its adjacent edges), and the algorithm proceeds recursively on the remaining graph. The ordering of Figure 5c, for instance, could have been generated by this procedure.

The connection between width and local consistency requires further elaboration. A constraint network is said to be *i-consistent* if for any set of $i-1$ variables along with values for each that satisfy all the constraints among them, there exists a value for any i^{th} variable, such that the i values together satisfy all the constraints among the i variables. *Strong i-consistency* holds when the problem is *j-consistent* for every $j \leq i$. Given an ordering d, *directional i-consistency* along d (or $d-i-$consistency) requires that any consistent instantiation of $i-1$ variables can be consistently extended only by variables that *succeed all of them* in the ordering d. *Strong-$d-i-$consistency* is defined accordingly. The general relationship between the *width* of a network and the amount of *local consistency* required for tractability is summarized in the following theorem:

> **Theorem**: An ordered constraint graph is *backtrack-free* if the *level* of directional strong consistency along this order is greater then the *width* of the ordered graph.

In particular, if the graph has *width-one* (i.e., it is a tree) a directional two-consistency is sufficient. If it is *width-two*, strong directional three-consistency would suffice. The intuition behind this theorem rests on the fact that when backtrack works along a given

ordering, it tests for consistency only among past and current variables, considering the relevant local constraints. If these constraints already ensure that a *locally consistent* partial solution will remain consistent relative to future variables, dead-end will not occur. This required level of *local-consistency* is related to the number of constraints future variables have with current variables. That is, when a future variable is constrained with many past variables (i.e. when it has a high *width*) the required level of local consistency among past variables is higher.

Because most problem instances will not satisfy the desired relationship between the *width* and the *consistency level*, it is possible to try to push one of these two factors until the relationship holds. One possibility is to increase the level of directional consistency until it matches the width of the problem. Specifically, if a width-$i-1$ problem is not i-consistent, algorithms enforcing directional i-consistency can be applied to it. [Dechter & Pearl 87]

Consider, first, the case of $width - 1$. According to the theorem, if a tree is ordered along a width-one ordering and then enforced with directional two-consistency (i.e., arc consistency), the result is a backtrack-free problem. Indeed, the *tree-algorithm* presented earlier does exactly that: the rooted-tree ordering is a width-one ordering (each node has only one adjacent predecessor) and its internal loop (steps 3-6) enforces directional arc consistency along this ordering.

This seems to lead to a general scheme: given a constraint network, find its width w and enforce directional (strong) $(w+1)$ consistency along the appropriate ordering, followed by a backtrack-free instantiation of the variables. Unfortunately, enforcing directional i-consistency ($i > 2$) often requires the addition of new constraints, and these constraints are reflected by additional arcs in the constraint-graph, which may cause the width to increase. The resulting problem will be directional consistent, but its width may now be greater than w, thus backtrack-free search is no longer guaranteed. The next algorithm [Dechter & Pearl 87, Siedel 81] overcomes this difficulty.

Given an ordering d, algorithm *adaptive consistency* establishes *directional i-consistency* recursively, when i changes from node to node to match its *width* at the time of processing. This is accomplished by processing nodes in decreasing order, so that by the time a node is processed its final *width* is determined and the required level of consistency can be achieved. Let $parents(X)$ denote the set of predecessors connected to X, when it is called for processing.

Adaptive-Consistency ($X_1, \ldots, X_n$)

begin
1. for i=n to 1 by -1 do
2. Compute parents(X_i)
3. connect all elements in parents(X_i) (if they are not yet connected)
4. perform consistency(X_i, parents(X_i))
5. endfor
End

The procedure *consistency(V, set)* generates and records tuples of those variables in the *set* that are *consistent* both internally and with at least one value of V. The procedure may impose new constraints over clusters of variables as well as tighten existing constraints. When adaptive consistency terminates, backtracking can solve the problem in the order prescribed without any dead-ends. It is important to realize that the topology of the resulting graph, called an *induced graph*, can be found prior to executing the procedure by recursively (in a decreasing order) connecting any two parents sharing a common successor.

Consider the ordering X_1, X_2, X_3, X_4, X_5 shown in Figure 5c. *Adaptive-consistency* proceeds from X_5 to X_1 and imposes constraints on the parents of each processed variable. X_5 is chosen first and because it has only one parent, X_2, the algorithm merely tightens the domain of X_2, if necessary (which amounts to enforcing arc consistency on (X_2, X_5).) X_4 is selected next and, having width two, the algorithm enforces a three-consistency on its parents $\{X_3, X_2\}$. This operation may require that a constraint between X_2 and X_3 be added, and in that case an arc (X_2, X_3) is added to the constraint graph. When the algorithm reaches node X_3, its width is two and, therefore a three-consistency is enforced on X_3's parents $\{X_2, X_1\}$. The arc (X_1, X_2) already exists so this operation may merely tighten the corresponding constraint. The resulting graph is given in Figure 6b.

Let $w(d)$ be the width of the ordering d and let $w^*(d)$ be the width of the induced graph. The complexity of solving a problem using the *adaptive consistency* preprocessing phase and then backtracking (freely) along the order d is dominated by the former. The worst-case complexity of the *consistency(V, parents(V))* step, is exponential in the cardinality of V and its *parent set*, because it actually solves a network of constraints having that many variables. Because the maximal size of the parent set is equal to the width of the induced graph, solving the constraint network along the ordering d is bounded by $O(n \cdot exp(w^*(d) + 1))$. Notice that had *adaptive-consistency* been applied on the ordering in Figure 6b, the resulting *induced width* would have been three.

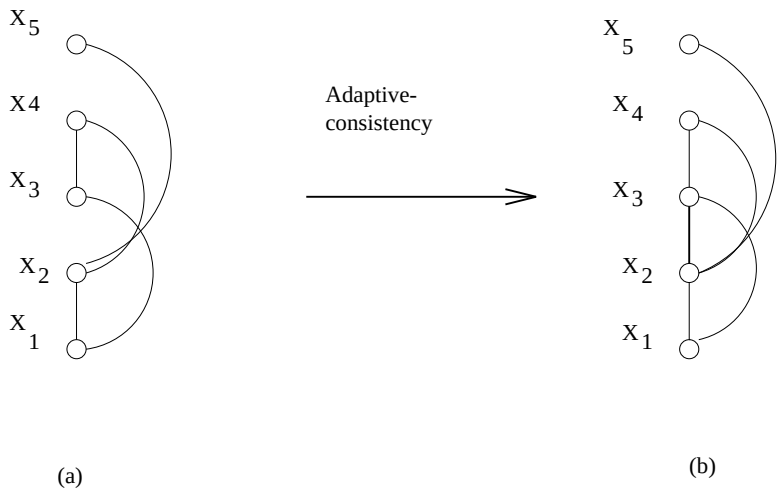

Figure 6: A constraint graph before (a) and after (b) *adaptive consistency*.

4.1 w*-Tractability

It seems that w^*, the minimum induced width, can be used to identify classes of easy problems. Namely, if the *primal graph* of a constraint network has $w^* \leq r$ then the problem can be solved in $O(exp(r))$ steps. However, finding the smallest *induced width* of a graph and its corresponding ordering is an NP-complete problem [Arnborg 85]. Nevertheless, deciding whether the w^* of a problem is less or equal to r is polynomial in r. In particular, deciding if a problem instance has small *induced width*, say $w^* = 1$, $w^* = 2$, or $w^* = 3$, can be efficiently determined. In trees, the *width* is equal to the *induced width* $(= 1)$; hence any *minimal width ordering*, is also an optimal *induced-width* ordering, and it can be found in linear time. A linear time algorithm recognizing problems having $w^* > 2$ is also available [Arnborg 85, Bertele & Brioschi 72]. The algorithm selects as last a node having a smallest degree, eliminates it, *connects its neighbors* in the residual graph (if they were not previously connected), and continues recursively. If the result is an ordering having $w^* > 2$ it can be concluded that the graph, too, has $w^* > 2$. Otherwise, the network has induced width equals two (also called *regular width-two* network).

In spite of the nice structure and complexity guarantees that are provided by *adaptive consistency*, experimental results have shown that unless w^* is very low (namely, one or two) the algorithm is too expensive on the average. Its cost stems from the determination to ensure an absolutely backtrack-free search, often investing a disproportional amount of computation trying to eliminate just a few remaining dead-ends. Simple backtracking, which can potentially encounter all such dead-ends, would often be more efficient. This

suggests that a less vigorous *consistency enforcing* algorithm can be appropriate, striking a compromise between preprocessing and search. Indeed, bounded *directional i-consistency algorithms* [Dechter & Pearl 87] fulfill such a compromise by enforcing a limited directional consistency and eliminating as many dead-ends as possible within some predetermined computational bounds. Instead of recording one constraint on all the parents of a node, these procedures record a set of smaller constraints on size-i subsets of the parents. It was shown that on classes of artificially generated CN, directional two-consistency eliminates a large subset of the dead-ends whereas directional three-consistency eliminates almost all [Dechter & Meiri 89].

4.2 Acyclic Networks and Tree-Clustering

Although w^* provides a measure of tractability, some problems admit easy solution, independently of their *width*. This happens when the *induced width* of an ordering is identical to its *width*, (namely no arcs are added by *adaptive consistency*), and when constraint recording consumes only a linear amount of computation (in the problem input). *Acyclic constraint networks (ACNs)* or *acyclic CSPs* have these two properties, and were first characterized and evaluated in the relational database literature [Beeri, et al 83]. These can be viewed as trees in the *dual-graph* representation. Clearly, if the *dual-graph* of a nonbinary CN is a tree, the *tree-algorithm* would apply. But even when the *dual graph* is not a tree, some of its arcs may be *redundant*, and their removal might result in a tree structure. An arc in the dual graph can be deleted if its variables are shared by every arc along an alternative path between the two end points. The subgraph resulting from removal of redundant arcs is called a *join graph*.

For instance, the arc between (AEF) and (ABC) in Figure 7a can be eliminated because the variable A is common along the cycle $(AEF)-A-(ABC)-AC-(ACE)-AE-(AEF)$, and so a consistent assignment to A is ensured by the remaining arcs. By a similar argument it is possible to remove the arcs labeled C and E, thus turning the join graph into a tree, called a *join tree*. (see Figure 7b). In general, finding whether such a transformation exists is a tractable problem [Maier 83].

Constraint networks that can be represented by a *join tree*, are called *acyclic networks* and can be solved efficiently as follows. If there are p constraints in the *join tree* (i.e., p c-variables), each allowing at most l tuples, then a straightforward application of the algorithm developed for a tree of singletons (using $O(nk^2)$ steps) would yield a solution in $O(pl^2)$ steps. A further refinement based on indexing can reduce the complexity to $O(p \cdot l \cdot logl)$ steps [Dechter & Pearl 89].

A generalization of acyclic networks called webs [Dalkey 91] permits backtrack-free solutions for a larger class of network topologies. This requires, however, that the constraints

Figure 7: A dual-constraint graph and its join tree.

possess special properties, typical of causal mechanisms [Dechter & Pearl 91]. Web structures are conveniently represented by a form of directed constraint networks (or causal networks) which indicate the ordering along which solutions can be obtained backtrack-free.

4.2.1 Recognizing Acyclic Networks

Several efficient procedures for identifying an ACN and finding a representative *join tree* have been described [Maier 83]. One scheme that proved particularly useful is based on the observation that a CN is acyclic if, and only if, its primal graph is both *chordal* and *conformal* [Beeri, et al 83]. A graph is *chordal* if every cycle of a length of at least four has a chord, i.e., an edge joining two nonconsecutive vertices along the cycle. A graph is *conformal* if each of its maximal *cliques* (i.e. subsets of nodes that are completely connected) corresponds to a constraint in the original CN. The *cordality* of a graph can be identified via an ordering called the *maximal cardinality ordering*, (*m-ordering*); it always assigns the next number to the node having the largest set of already numbered neighbors (breaking ties arbitrarily). For instance, the ordering in 5c is an m-ordering, whereas in Figures 5a and 5b it is not.

It can be shown [Tarjan & Yannakakis 84] that in an m-ordered chordal graph, the parents of each node must be completely connected. If, in addition, the maximal cliques coincide with the *constraint-subsets* of the original CN, both conditions for acyclicity would be satisfied. Because for chordal graphs each node and its parent set constitutes a clique, the maximal cliques can be identified in linear time, and then a *join tree* can be constructed by connecting each maximal clique to an ancestor clique with which it shares the largest set of variables.

As noted, *acyclic networks* have a chordal primal graph, thus their *width* and *induced*

width are identical along an m-ordering. Hence, if applied to such ordered CNs, *adaptive-consistency* will add no arcs to the graph. Also, because all tuples on each parent set are already *locally consistent*, the amount of constraint recording is bounded by $O(l \cdot log\ l)$, resulting in an overall complexity bound of $O(n \cdot l \cdot log\ l)$ steps.

4.2.2 Tree Clustering

The above recognition process suggests a scheme for combining subsets of constraints into higher level constraints until a join tree emerges (when the network is not acyclic to begin with). Such a *tree-clustering* scheme is based on a triangulation algorithm [Tarjan & Yannakakis 84] that transforms any graph into a chordal graph by *filling in* edges (recursively) in a reverse order of the m-ordering, connecting any two nonadjacent nodes that are connected via nodes higher up in the ordering. The maximal cliques of the resulting chordal graph are the clusters necessary for forming an ACN. These clusters represent subproblems that must be independently solved, an operation that is exponential in the clique's size.

It can be shown that the maximal clique size, generated that way, equals $w^* + 1$; thus the whole transformation (into a join tree) is, once again, exponential in w^*. Although *tree clustering* differs conceptually from *adaptive consistency*, it effectively results in the same behavior and same performance. When applied on the same ordered constraint graph, both algorithms produce the same *induced graph*. In other words, *adaptive consistency* can be viewed as an effective scheme for assembling $ACNs$. It seems desirable to use adaptive consistency when one-time solutions are required, and to use tree-clustering when the network is used as a knowledge base subjected to repeated queries. Note that although *tree clustering* can be applied in any ordering, the m-ordering produces close to optimal induced width (for chordal graphs it is indeed optimal.)

A *subclass* of $ACNs$ for which all maximal cliques have the same size is often characterized by a special class of chordal graphs called *k-trees*. A *k-tree* is a chordal graph whose maximal cliques are of size $k+1$, and it can be defined recursively as follows: (1) A complete graph with k vertices is a *k-tree*. (2) A *k-tree* with r vertices can be extended to $r+1$ vertices by connecting the new vertex to the vertices in any clique of size k. In particular, one-trees are ordinary trees.

The addition of each vertex (step 2) generates a new clique of size $k+1$, and by associating each new clique with one parent clique that shares k vertices with it, a join tree is obtained. The example of an acyclic CN given in Figure 7 is indeed a two-tree because its primal graph could be constructed in the order A, B, C, E, D, F. *k-trees* were investigated extensively in the graph theoretical literature. In particular, it was shown that a graph can be *embedded* in a k-tree if and only if it has an induced width $w^* = k$. Detailed discussions

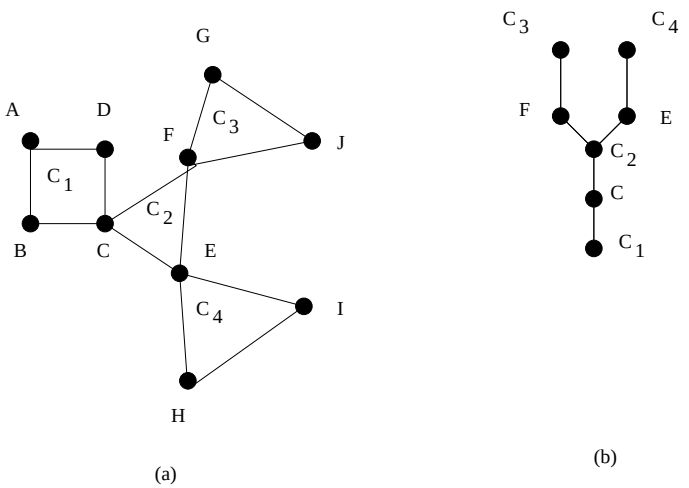

Figure 8: A Graph and its decomposition into nonseparable components.

of the properties are available [Arnborg 85, Freuder 90, Rossi & Montanari 89].

4.3 Decomposition into Nonseparable Components

Another approach that exploits the structure of the constraint graph involves the notion of *nonseparable components* [Freuder2 85, Dechter & Pearl 87]. Similar to *tree-clustering*, the idea is to identify subsets of variables that, when grouped together, transform the problem into a tree; the *nonseparable components* of a graph have this property [Even 79].

A connected graph, $G = (V, E)$ (V, a set of nodes, E, a set of edges), is said to have a *separation node* v if there exists nodes a and b such that all paths connecting a and b pass through v. A graph that has a *separation node* is called *separable*, and one that has none is called *nonseparable*. A subgraph with no *separation nodes* is called a *nonseparable component*. An $O(\mid E \mid)$ algorithm exists for finding all the *nonseparable components* and the *separation nodes*; it is based on a *depth-first search* traversal of the graph, called a *DFS ordering* [Even 79].

Let G be a graph and $super{-}G$ the *tree* whose nodes represents the components $C_1, C_2, \ldots, C_r$ and the separating nodes $V_1, V_2, \ldots, V_t$ (Figure 8b). Figure 8 shows a graph G, its components, and its separating vertices. Once the *components* are recognized, each represents a subproblem that, when solved, defines the domains of a new compound variable. The *tree-algorithm* can then be applied to the resulting problem, treating each component as a compound variable.

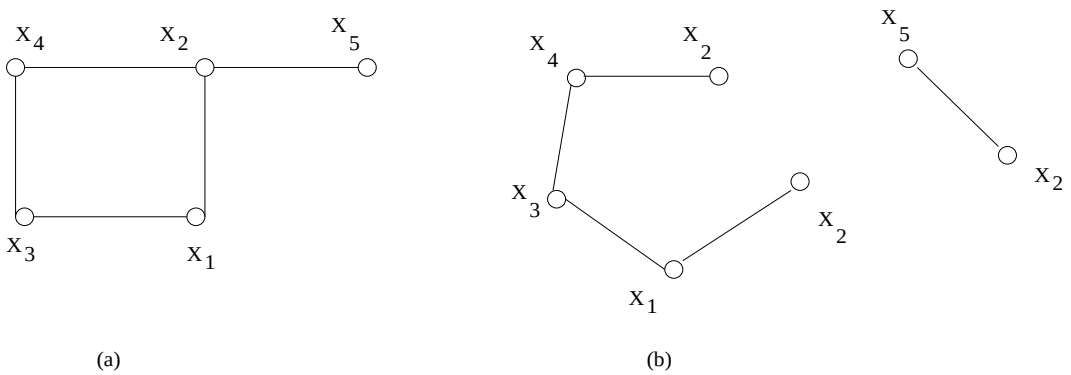

Figure 9: An instantiated variable cuts its own cycles.

The complexity of this approach is $O(nk^r)$ where r is the size of the largest component. Therefore, in cases where the constraint network has a decomposition into small clusters of *nonseparable* components, the resulting performance is improved. In comparing the *nonseparable component* method with either *tree clustering* or *adaptive consistency* it is immediately realized that it does not improve the worst-case complexity, namely, $w^* \leq r$ and, frequently $w^* < r$. Nevertheless, this scheme is the most natural extension of trees and can also be extended to the dual-graph representation.

4.4 The Cycle Cutset Scheme

The decomposition method presented in this section is based on identifying a *cycle cutset*, that is, a set of nodes that, once removed, would render the constraint graph cycle-free. The method uses trees in a different way than previous schemes, exploiting the fact that variable instantiation changes the effective connectivity of the constraint graph. In Figure 9, for example, instantiating X_2 to some value, say *hike*, renders the choices of X_1 and X_5 independent as if the pathway $X_1 - X_2 - X_5$ were blocked at X_2. Similarly, this instantiation blocks the pathway $X_1 - X_2 - X_4$, leaving only one path between any two variables. The effective constraint graph for the rest of the variables is shown in Figure 9b, where the instantiated variable X_2 is duplicated for each of its neighbors.

When the group of instantiated variables constitutes a cycle cutset, the remaining network is cycle free, and can be solved by the *tree algorithm*. In the example above, X_2 cuts the single cycle $X_1 - X_2 - X_3 - X_4$ and renders the graph in Figure 9b cycle free. In most practical cases it would take more than a single variable to cut all the cycles in the graph. Thus a general way of solving a problem of which the constraint graph contains

cycles is to find a consistent instantiation of the variables in a cycle cutset and solve the remaining problem by the *tree algorithm*. If a solution to the restricted problem is found, then a solution to the entire problem is at hand. If not, another instantiation of the cycle cutset variables should be considered until a solution is found. Thus if the task is to solve our crossword puzzle (Figure 1), first $X_2 = hike$ must be assumed, and the remaining tree problem is solved. If no solution is found, it is assumed that $X_2 = keel$ and another attempt is made until a solution is found.

The complexity of the *cycle cutset* scheme is bounded by $O(exp(c))$ steps, where c is the size of the *cycle cutset* because the utmost number of times the tree algorithm is invoked equals the number of partial solutions to the cutset variables. Because finding a minimal-size cycle cutset is NP hard, it will be more practical to incorporate this scheme within a general problem solver such as backtracking. Because *backtracking* works by progressively instantiating sets of variables, all that is necessary is to keep track of the connectivity status of the constraint graph. As soon as the set of instantiated variables constitutes a cycle cutset, the search algorithm is switched to the *tree algorithm* on the remaining problem, i.e., either finding a consistent extension for the remaining variables (thus finding a solution to the entire problem) or concluding that no such extension exists (in which case backtracking takes place and another instantiation tried) [Dechter 90].

4.5 Graph-Based Schemes for Improving Backtracking

Two ideas for improving the *look-back* phases of backtracking have received wide attention [Gaschnig 79, Stallman & Sussman 77, Doyle 79, Dechter 90]. These have often been referred to as *backjumping* and *constraint recording* in the constraint literature, but are more commonly recognized under the umbrella name *dependency-directed backtracking*, in the truth-maintenance literature. *Backjumping* suggests *jumping back* several levels in the search tree to a variable that may have relevance to the current dead-end, whereas *constraint recording* suggests storing the reasons for the dead-end in the form of new constraints, so that the same conflict will not arise again later in the search (i.e., recording *no-goods*).

In this section, graph-based variants of both *backjumping* and *constraint recording* are presented. Exploiting the structure of the problem often simplifies the implementation of these schemes and enables an assessment of the complexity, using network parameters.

4.6 Backjumping

The idea of going back several levels (in a dead-end situation) rather than retreating to the chronologically most recent decision was exploited independently in [Gaschnig 79] where the term "backjumping" was introduced, and in [Stallman & Sussman 77]. The idea has since been used in truth-maintenance systems, [Doyle 79] and in *intelligent backtracking* in PROLOG [Bruynooghe & Pereira 84]. Gaschnig's algorithm uses a marking technique where each variable maintains a pointer to the highest ancestor found incompatible with any of its values. In case of a dead-end, the algorithm can safely jump directly to the ancestor pointed to by the dead-end variable. Although this scheme retains only one bit of information with each variable, it requires an additional computation with each consistency check.

Graph-based backjumping [Dechter 90] extracts knowledge about dependencies from the constraint graph alone. Whenever a dead-end occurs at a particular variable X, the algorithm backs up to the most recent variable connected to X in the graph. Consider, for instance, the ordered constraint graph in Figure 5a. If the search is performed in the order X_1, X_2, X_3, X_4, X_5 and a dead-end occurs at X_5, the algorithm will jump back to variable X_2 because X_5 is not connected to either X_3 or X_4. If the variable to which the algorithm retreats has no more values, it should back up to the most recent parent of both the original variable and the new dead-end variable, and so on.

Whereas the implementation of this backjumping scheme would, in general, require a careful maintenance of each variable's parents set [Dechter 90], some orderings facilitate an especially simple implementation. If a depth-first search is used on the constraint graph (to generate a DFS tree) and then backjumping is conducted in an in-order traversal of the DFS tree [Even 79], finding the jump-back destination amounts to following a very simple rule: if a dead-end occurred at variable X, go back to the parent of X in the DFS tree. Consider, once again, the example in Figure 2. A DFS tree of this graph is given in figure 10b, and an in-order traversal of this tree is $(X_1, X_2, X_5, X_4, X_3)$. If a dead-end occurs at node X_4, the algorithm retreats to it parent X_2. When *backjumping* is performed on a DFS ordering of the variables, its complexity can be bounded by $O(exp(m))$ steps, m being the depth of the DFS tree. However, like many other parameters encountered, finding a minimal-depth DFS tree is NP-hard.

4.6.1 Constraint-Recording or Dependency-Directed Backtracking

An opportunity to *learn* or *deduce* a new constraint is presented whenever backtracking encounters a dead-end, i.e., when the current instantiation $s = (X_1 = x_1, \ldots, X_{i-1} = x_{i-1})$ cannot be extended by *any* value of the next variable X_i. In such a case s is *in conflict* with

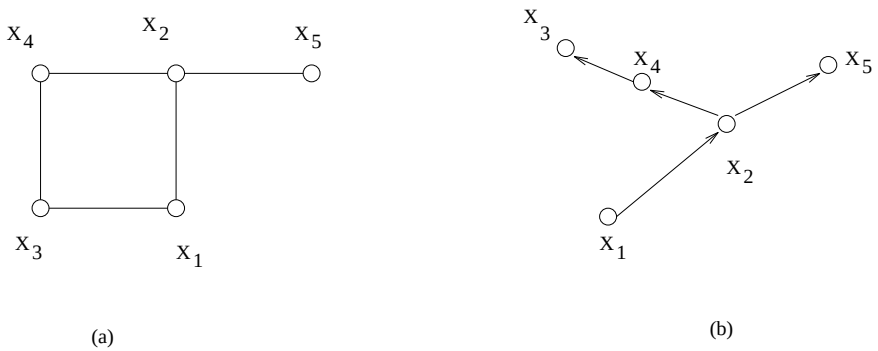

Figure 10: A DFS tree and its ordering.

X_i or that s is a *conflict set*. Had the problem included an explicit constraint prohibiting the instantiation s, the current dead-end would have been avoided. However, there is no point recording such a constraint at this stage, because under the backtracking control strategy it will not recur. If, on the other hand, the set s contains one or more subsets that are also in conflict with X_i, then recording this information in the form of new explicit constraints might prove useful in the future because future states may contain these subsets. The constraint graph provides an easy way for identifying subsets of s that are in conflict; by removing from s all assignments of variables that are *not connected* to X_i, a subset is obtained that is still in conflict with X_i, because all the removed assignments are irrelevant to this dead-end.

The procedure of *graph-based dependency-directed backtracking* (sometimes called *graph-based constraint recording* in [Dechter 90]), implements this idea by recording these conflict sets as a new constraint on each dead-end. Specifically, if the subsets $s' = (X_{i1} = x_{i1}, \ldots, X_{it} = x_{it})$ is the assignments in s connected to X_i, the procedure records a constraint on variables $X_{i1}, \ldots, X_{it}$ which disallows the tuple s'. For instance, suppose that backtracking solves the crossword puzzle using the ordering $(X_1, X_2, X_5, X_4, X_3)$ and is currently at state $(X_1 = snail, X_2 = aron, X_5 = no, X_4 = dock)$. This state cannot be extended by any value of X_4. Obviously, the tuple $(X_1 = snail, X_2 = aron, X_5 = no, X_4 = dock)$ is a conflict set; however, both the instantiations $X_2 = aron$ and $X_5 = no$ are irrelevant to this conflict because there is no explicit constraint between X_3 and X_2 or between X_3 and X_5. Therefore, the tuple $(X_4 = down,\ X_1 = snail)$ will be disallowed by recording a new constraint on X_1 and X_4.

Dependency-directed backtracking can be performed on any variable ordering. Its worse-case complexity is $O(exp(w^*))$ steps, thus providing yet another scheme whose performance is governed by the induced width.

5 Conclusion

Throughout this survey several techniques were presented that exploit the structure of the given network. Four graph parameters stood out in the analysis: the *induced-width* w^*, (appearing in *adaptive-consistency, tree-clustering* and *constraint recording* in dependency-directed backtracking), the *cycle-cutset size* c (appearing in the cycle-cutset method), the *depth of a DFS-tree* m (in backjumping), and the *size of largest non-separable component* r (appearing in the tree-component scheme). It is clear that for any problem structure, the relationships $m \geq w^*$, $r \leq w^*$ holds, and it can also be shown that $w^* \leq c + 1$ [Bertele & Brioschi 72]. m and r are not comparable, sometimes $m < r$ (e.g., trees) and sometimes $r < m$ (e.g., mesh). Another parameter mentioned in the literature, bandwidth [Zabih 90] is also dominated by w^*. It can be concluded, therefore, that w^* provides the most informative graph parameter, and it can be regarded as an intrinsic measure of the worse-case complexity of any constraint network.

References

[Arnborg 85] Arnborg, S., "Efficient algorithms for combinatorial problems on graphs with bounded decomposability – a survey," *BIT*, Vol. 25, pp. 2–23, 1985.

[Beeri, et al 83] Beeri, C., Fagin, R., Maier, D., and Yannakakis, M., "On the desirability of acyclic database schemes," *Journal of the ACM,* Vol. 30, No. 3, pp. 479–513, July, 1983.

[Bertele & Brioschi 72] Bertele, U., and Brioschi, F., *Nonserial dynamic programming,* Academic Press, New York, 1972.

[Bruynooghe & Pereira 84] Bruynooghe, M., and Pereira, L. M., "Deduction revision by intelligent backtracking," in *Implementation of PROLOG*, J. A. Campbell, Ed. Ellis Harwood, pp. 194–215, 1984.

[Dalkey 91] Dalkey, N. C., *Modeling Probability Distributions with WEB Structures*, Technical Report R-164, University of California, Los Angeles, 1991.

[Dechter & Pearl 87] Dechter, R., and Pearl, J., "Network-based heuristics for constraint-satisfaction problems," *Artificial Intelligence,*" Vol. 34, No. 1, pp. 1–38, 1987.

[Dechter & Meiri 89] Dechter, R., and Meiri, I., "Experimental evaluation of preprocessing techniques in constraint satisfaction," in *Proceedings of the 11th International Conference on AI (IJCAI-89)*, Detroit, MI, August, 1989.

[Dechter & Pearl 89] Dechter, R., and Pearl, J., "Tree clustering for constraint networks," in *Artificial Intelligence,* pp. 353–366, 1989.

[Dechter 90] Dechter, R., "Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition," *Artificial Intelligence,* Vol. 41, No. 3, pp. 273–312, January, 1990.

[Dechter & Pearl 91] Dechter, R., and Pearl, J., "Directed Constraint Networks: A Relational Framework for Causal Modeling," *Proceedings of the Twelfth IJCAI,* Sydney, Australia, Morgan-Kaufmann, San Mateo, CA, 1991.

[Doyle 79] Doyle, J., "A truth maintenance system," *Artificial Intelligence,"* Vol. 12, pp. 231–272, 1979.

[Even 79] Even, S., *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.

[Freuder 85] Freuder, E. C., "Synthesizing constraint expression," *Communications of the ACM,* Vol. 29, No. 1, pp., 24–32, 1985.

[Freuder 82] Freuder, E. C., "A sufficient condition for backtrack-free search," *Journal of the ACM*, Vol. 29, No. 1, pp. 24–32, 1982.

[Freuder2 85] Freuder, E. C., "A sufficient condition for backtrack-bounded search," *Journal of the ACM*, Vol. 32, No. 4, pp. 755–761, October, 1985.

[Freuder 90] Freuder, E. C., "Complexity of k-structured constraint satisfaction problems," in *Proceedings of AAAI-90*, pp. 4–9, Boston, MA, July, 1990.

[Gaschnig 79] Gaschnig, J., "Performance measurement and analysis of certain search algorithms," Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, Tech. Report CMU-CS-79-124, 1979.

[Haralick & Elliott 80] Haralick, R. M., and Elliott, G. L., "Increasing tree-search efficiency for constraint satisfaction problems," *Artificial Intelligence,* Vol. 14, No. 3, pp. 263–313, 1980.

[Mackworth & Freuder 84a] Mackworth, A. K., "Consistency in networks of relations," *Artificial Intelligence*, Vol. 8, No. 1, pp. 99–118, 1977.

[Mackworth & Freuder 84b] Mackworth, A. K., and Freuder, E. C., "The complexity of some polynomial network consistency algorithms for constraint satisfaction problems," *Artificial Intelligence,* Vol. 25, No. 1, 1984.

[Maier 83] Maier, D., *The theory of relational databases,* Computer Science Press, Rockville, MD, 1983.

[Montanari 74] Montanari, Ugo., "Networks of constraints: Fundamental properties and applications to picture processing," *Information Sciences,* Vol. 7, No. 2, pp. 95–132, 1974.

[Purdom 83] Purdom, P., "Search rearrangement backtracking and polynomial average time," *Artificial Intelligence*, Vol. 21, No. 1–2, pp. 117–133, 1983.

[Rossi & Montanari 89] Rossi, F. and Montanari, U., "Exact solution in linear time of networks of constraints using perfect relaxation," in *Proceedings First International Conference on Principles of Knowledge Representation and Reasoning,* pp. 394–399, Toronto, Ontario, Canada, May, 1989.

[Siedel 81] Seidel, R., "A new method for solving constraint-satisfaction problems," in *Proceedings IJCAI,* pp. 338–342, 1981.

[Stallman & Sussman 77] Stallman, R. M. and Sussman, G. J., "Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis," *Artificial Intelligence*, Vol 9, No. 2, pp. 135–196, 1977.

[Stone & Stone 86] Stone, H. S. and Stone, J. M., "Efficient search techniques – An empirical study of the N-Queens problem," IBM T. J. Watson Research Center, Tech. Report RC 12057 (#54343), Yorktown Heights, NY, 1986.

[Tarjan & Yannakakis 84] Tarjan, R. E. and Yannakakis, M., "Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectivity reduce acyclic hypergraphs," *SIAM Journal of Computing,* Vol. 13, No. 3, pp. 566–579, 1984.

[Waltz 75] Waltz, D., "Understanding line drawings of scenes with shadows," in *The Psychology of Computer Vision,* P. H. Winston, Ed., McGraw-Hill, New York, 1975.

[Zabih 90] Zabih, R., "Some applications of graph bandwidth to constraint satisfaction problems," in *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90),* pp. 46–50, Boston, MA, July, 1990.