NETWORK-BASED HEURISTICS FOR CONSTRAINT SATISFACTION PROBLEMS

Rina Dechter
Judea Pearl

July 1987
CSD-870037

To Appear in "Artificial Intelligence," (Journal) 1988.

TECHNICAL REPORT

R-90

May 1987

NETWORK-BASED HEURISTICS FOR CONSTRAINT-SATISFACTION PROBLEMS *

Rina Dechter & Judea Pearl

Cognitive Systems Laboratory

Computer Science Department

University of California, Los-Angeles, CA. 90024

ABSTRACT

Many AI tasks can be formulated as Constraint-Satisfaction problems (CSP), i.e., the assignment of values to variables subject to a set of constraints. While some CSPs are hard, those that are easy can often be mapped into sparse networks of constraints which, in the extreme case, are trees. This paper identifies classes of problems that lend themselves to easy solutions, and develops algorithms that solve these problems optimally. The paper then presents a method of generating heuristic advice to guide the order of value assignments based on both the sparseness found in the constraint network and the simplicity of tree-structured CSPs. The advice is generated by simplifying the pending subproblems into trees, counting the number of consistent solutions in each simplified subproblem, and comparing these counts to decide among the choices pending in the original problem.

* This work was supported in part by the National Science Foundation Grant, DCR 85-01234.

1. BACKGROUND AND MOTIVATION

1.1 Introduction

An important component of human problem-solving expertise is the ability to use knowledge about solving easy problems to guide the solution of difficult ones [16]. Only few works in AI have attempted to equip machines with similar capabilities [22, 2]. Gaschnig [10], Guida et al. [11], and Pearl [19] suggested that knowledge about easy problems could serve as a heuristic in the solution of difficult problems, i.e., that it should be possible to manipulate the representation of a difficult problem until it is approximated by an easy one, solve the easy problem, and then use the solution to guide the search process in the original problem.

The implementation of this scheme requires three major steps: a) simplification, b) solution, and c) advice generation. Additionally, to perform the simplification step, we must have a simple, **a-priori** criterion for deciding when a problem lends itself to easy solution.

This paper uses the domain of constraint-satisfaction tasks to examine the feasibility of these three steps. It establishes criteria for recognizing classes of easy problems, develops optimal procedures for solving them, demonstrates a scheme for generating approximate easy models, and introduces an efficient method for extracting advice thereof. Finally, the utility of using the advice is evaluated in a synthetic domain of randomly generated problem instances.

Constraint-satisfaction problems (CSPs) involve the assignment of values to variables subject to a set of constraints. Constraint specification represents a convenient form of expressing declarative knowledge, allowing the system designer to focus on local relationships among entities in the domain. Solving a CSP amounts to generating explicit interpretations to knowledge given in implicit declarative form. Examples of CSPs are map coloring, understanding line drawings, electronic circuit analysis, and truth maintenance systems. These are normally solved by some version of backtrack search which, in the worse case, may require exponential search time (for example, the map coloring problem is a CSP known to be NP-complete.)

The following paragraphs summarize the basic terminology of the theory of CSP as presented in [17].

1.2 Definitions and Nomenclature

A CSP involves a set of n variables $X_1, \dots, X_n$ having domains $D_1, \dots, D_n$, where each D_i defines the set of values that variable X_i may assume. An **n-ary relation** on these variables is a subset of the Cartesian product:

$$\rho \subseteq D_1 \times D_2 \times, \dots, \times D_n \ . \tag{1}$$

A **binary constraint** R_{ij} between two variables is a subset of the Cartesian product of their domains, i.e.,

$$R_{ij} \subseteq D_i \times D_j \ . \tag{2}$$

When $i=j$, R_{ii} stands for a **unary** constraint on X_i.

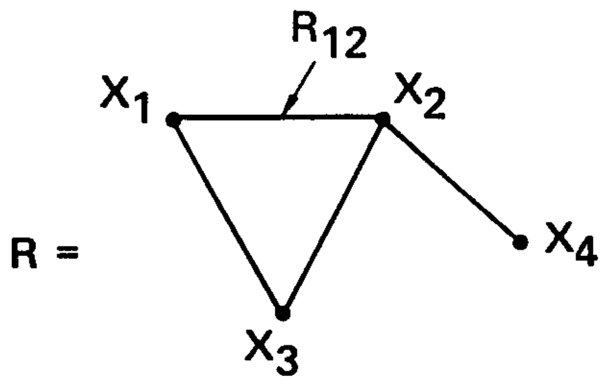
A **network of binary constraints** is a set of variables $X_1, \dots, X_n$ plus a set of binary and unary constraints imposed on the variables. It represents an n-ary relation defined by the set of all n-tuples satisfying all the constraints. The relation ρ represented by a given network of constraints is:

$$\rho = \{(x_1, x_2, \dots, x_n) \mid x_i \in D_i, \text{ and } (x_i, x_j) \in R_{ij} \text{ for all } ij\} \ . \tag{3}$$

A constraint is **symmetric** if for all $x_i \in D_i$ and $x_j \in D_j$, if $(x_i, x_j) \in R_{ij}$ then also $(x_j, x_i) \in R_{ji}$.

Not every n-ary relation can be represented by a network of binary constraints with n variables, and the issue of finding a network that best approximates a given relation is addressed in [17].

Each network of binary constraints can be represented by a **constraint graph** where the variables are represented by nodes and the constraints by arcs, and each constraint specifies the set of permitted pairs of values. Figure 1 displays a typical network of constraints (a), where constraints are given using matrix notation (b), where the entries "0" and "1" indicate forbidden and permitted pairs of values, respectively.



(a)

$$R_{12} = \begin{array}{c} \\ a \\ b \\ c \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

(b)

Figure 1 - A constraint Network (a) and matrix representation (b)

Although this paper deals mainly with binary constraints, the results are easily generalized to non-binary cases. CSP would still be characterized by a graph in which variables, participating in the same constraints, will be mutually adjacent.

Several operations on constraints can be defined. The useful ones are: union, intersection, and composition. The **union** of two constraints between the same two variables is a new constraint that allows all pairs allowed by either one of them. The **intersection** of two constraints allows only pairs that are allowed by both. The **composition** of two constraints, R_{12}, R_{23} "induces" a constraint R_{13} defined as follows: A pair

(x_1,x_3) is allowed by R_{13} if there is at least one value $x_2 \in D_2$ such that $(x_1,x_2) \in R_{12}$ and $(x_2,x_3) \in R_{23}$. In matrix notation the induced constraint R_{13} can be obtained by matrix multiplication:

$$R_{13} = R_{12} \cdot R_{23} \tag{4}$$

A partial order among the constraints can be defined as follows: we say that R_{ij} is **tighter** than R'_{ij}, denoted $R_{ij} \subseteq R'_{ij}$, iff every pair allowed by R_{ij} is also allowed by R'_{ij}. We can also say that R'_{ij} is a **relaxation** of R_{ij}. The tightest constraint between variables X_i and X_j is the **empty constraint**, denoted Φ_{ij}, which does not allow any pair of values, while the most relaxed is the **universal constraint**, denoted U_{ij}, which permits all possible pairs. Note that universal constraints are not associated with arcs in the constraint graph. A corresponding partial order can be defined among networks of constraints having the same set of variables. We say that $R \subseteq R'$ if the partial order is satisfied for all the corresponding constraints in the networks. Two networks of constraints with the same set of variables are **equivalent** if they represent the same n-ary relation.

Consider, for example, the network of figure 2, representing a problem of four bi-valued variables. The constraints, (representing equalities and inequalities), are attached to the arcs and are shown explicitly by the sets of allowed pairs. The direction of the arcs only indicates the way by which constraints are specified. In this example, the constraint between X_1 and X_4, displayed in part (b), can be induced by R_{12} and R_{24}. Therefore, adding this constraint to the network will result in an equivalent network. Similarly, since the constraint R_{21} can be induced from R_{23} and R_{31} it can be deleted without changing the relation represented by the network.

The process of inducing new constraints in a given network makes the constraints tighter and tighter, while leaving the networks equivalent to each other. Montanari called the tightest network of binary constraints which is equivalent to a given network R, The

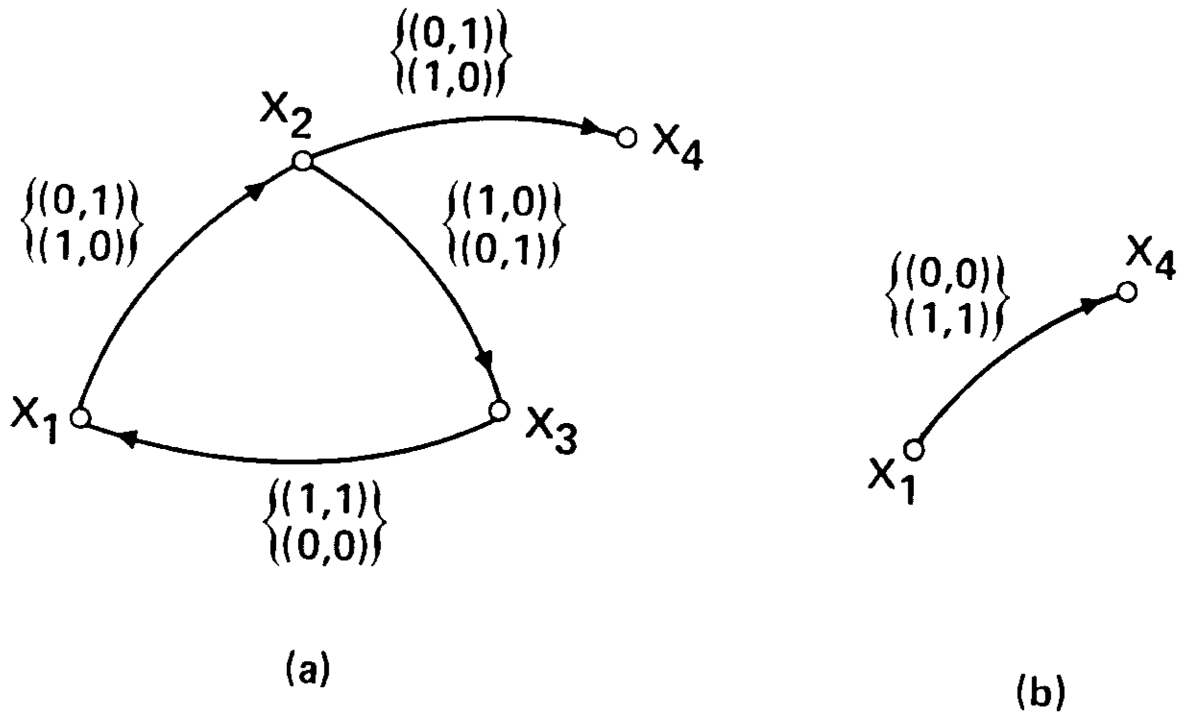


Figure 2 - A constraint network, constraints represented by set of pairs.

Minimal Network. The minimal network of constraints makes the "global" constraints on the network as "local" as possible. In other words, it is maximally explicit.

Every binary CSP can be represented by a network of constraints. A tuple in the relation represented by the network is called a solution. The problem is either to find all solutions, one solution, or to verify that a certain tuple is a solution. The last problem is fairly easy while the first two problems can be difficult and have attracted a substantial amount of research.

1.3 Backtrack for CSP

The common algorithm for solving CSPs is the **Backtrack** search algorithm. In its primitive version, backtrack traverses the variables in a predetermined order, provisionally assigning consistent values to a subsequence $(X_1, \ldots, X_i)$ of variables and attempting to append to it a new instantiation of X_{i+1} such that the whole set is consistent. (An assignment of values to a subset of variables is **consistent** if it satisfies all the constraints applicable to this subset.) If no consistent assignment can be found for the next variable X_{i+1}, a dead end situation occurs, the algorithm "backtracks" to the most recent variable, changes its assignment and continues from there. Algorithm Backtrack for finding one solution is given below. It is defined by two recursive procedures, **Forward** and **go-back.** The first extends a current partial assignment if possible, and the second handles dead end situations. The procedures maintain lists of candidate values (C_i) for each variable X_i.

```
Forward ($x_1, . . . , x_i$)
Begin
  1. if i = n exit with the current assignment.
  2. $C_{i+1}$ <-- Compute-candidates($x_1, . . . , x_i, X_{i+1}$)
  3. if $C_{i+1}$ is not empty then
  4.   $x_{i+1}$ <-- first element in $C_{i+1}$, and
  5.   remove $x_{i+1}$ from $C_{i+1}$, and
  6.   Forward($x_1, . . . , x_i, x_{i+1}$)
  7. Else
  8.   Go-back($x_1, . . . , x_i$)
End.

go-back($x_1, . . . , x_i$)
Begin
  1. if $i$=0, exit. No solution exists.
  2. if $C_i$ is not empty then
  3.   $x_i$ <-- first in $C_i$, and
  4.   remove $x_i$ from $C_i$, and
  5.   Forward($x_1, . . . , x_i$)
  6. Else
  7. Go-back($x_1, . . . , x_{i-1}$)
End
```

Backtrack is initiated by calling "forward" with i=0, namely, the instantiated list is empty. The procedure "compute-candidates($x_1, . . . , x_i, X_{i+1}$)" selects all values in the domain of X_{i+1} which are consistent with the previous assignments w.r.t. all applicable constraints $R_{j(i+1)}$, $j \leq i$.

The primitive version of backtrack contains many maladies and several cures have been offered and analyzed in the AI literature. The recent increase of interest in this topic can be attributed to the use of backtrack in PROLOG [15,3,1] and in truth-maintenance systems [23,7,14]. The terms "intelligent backtracking", "selective backtracking", and "dependency-directed backtracking" describe various efforts of improving backtrack's performance. The spectrum of research on CSPs can be classified along the following dimensions:

1. **Improving representation prior to search:** Montanari [17], considering the task of finding all solutions, introduced methods of achieving local consistency by propagating the constraints and deleting pairs of incompatible values. Mackworth [13] extended Montanari's work by introducing higher levels of consistency for curing Backtrack's maladies. Freuder [8] considered the problem of finding one solution and provided a preprocessing procedure for selecting a good ordering of variables prior to running the search. Dechter [4] introduced algorithms for removing redundancies from the input constraints and exploiting the sparseness of the resulting network.

2. **Improving backtrack during search:**. The various strategies here can be classified as follows:

    1. **Look-ahead schemes:** guiding the decision of what variable to instantiate or what value to assign next among all the consistent choices available.

        a. **Variable Ordering:** An attempt is made to instantiate that variable which maximally constrains the rest of the search space. Usually the variable participating in the highest number of constraints is selected [8,20,24].

b. **Value Ordering:** An attempt is made to assign a value that maximizes the number of options available for future assignments [12].

2. **Look-back schemes:** Guiding the decision of where and how to backtrack in case of a dead end situation. Look-back schemes are centered around two fundamental ideas:

   a. **Go-back to source of failure:** An attempt is made to detect and change decisions that caused the dead end while skipping irrelevant decisions [9]. Most work on intelligent backtracking in PROLOG is focused on such facilities [15, 3, 1].

   b. **Constraint-recording (learning):** The reasons for the dead-end are recorded so that the same conflicts will not arise again in the continuation of the search. The notion of "Dependency-directed-backtracking" developed for Truth-maintenance systems (TMSs), incorporates both types of look-back schemes [23, 7, 14]. Dechter [6] studied the trade-offs involved, by controlling the amount of information recorded.

Analysis of the average performance of Backtrack were reported by [18, 21] and [12], all estimating the size of the tree exposed by Backtrack while searching for all solutions. Stone [25] analyzed the average size of the search tree generated when backtrack searches for the first solution.

It seems that the parameter which received the least attention is the order by which values are assigned to variables. Part of the reason can be that, under a fixed order of variables, the search tree exposed by backtrack aiming for all solutions, is invariant to the order of value selection [5]. Moreover, backtrack's performance is uneffected by the order of value selection (assuming no pruning), in problems having no solutions, irrespective of the objective of the search.

In this paper we address the task of finding a **single solution** to CSPs. Although this problem is easier then finding all solutions, it can still be very difficult (e.g. 3-colorability) and it occurs frequently. Theorem proving, planning and even vision problems are examples of domains where finding one solution will normally suffice, and the order by which values are selected may have a profound effect on the algorithm's performance. In the following subsection we outline a general approach to devising value selection strategies for finding one solution to a CSP.

1.4 A General Approach to Automatic Advice Generation

Backtrack builds partial solutions and extends them as long as they show promise to be part of a whole solution. When a dead-end is recognized it backtracks to a previous variable. Assuming that the order of variables is fixed, the selection of the next node amounts to choosing a promising assignment of a value from a set of pending options. Clearly, if the next value can be guessed correctly, and if a solution exists, the problem will be solved in linear time with no backtracking. The advice we wish to generate should order the candidates according to the confidence we have that they can be extended further to a full solution.

Such confidence can be obtained by making simplifying assumptions about the continuing portion of the search graph and estimating the likelihood that it will contain a solution even when the simplifying assumptions are removed. It is reasonable to assume that, if the simplifying assumptions are not too severe, the number of solutions found in the simplified version of the problem would correlate positively with the number of solutions present in the original version, and, hence, is indicative of the chance to retain at least one surviving solution. We, therefore, propose to count the number of solutions in the simplified model and use it as a measure of confidence that the options considered will lead to an overall solution.

To incorporate this advice into the backtrack algorithm, an advice procedure should estimate the number of possible solutions stemming from each candidate value in C_i and order their instantiations accordingly.

Section 2 provides theoretical grounds for the advice-giving scheme, establishes criteria for recognizing classes of easy CSPs and develops algorithms for their solutions. Section 3 describes the algorithm, introduces an efficient method of counting the number of solutions, and report experimental evaluation of its performance. Conclusions are given in section 4.

2. THE ANATOMY OF SIMPLE PROBLEMS

2.1 Introduction and background

In general, a problem is considered easy when its representation permits a solution in polynomial time. In our context, since we are dealing mainly with backtrack algorithms, we will consider a CSP **easy** if it can be solved by a **backtrack-free** procedure. A backtrack-free search is one in which Backtrack terminates without backtracking, thus producing a solution in time linear in the number of variables.

The feasibility of achieving backtrack-free search relies heavily on the topology of the constraint graph. Freuder [15] has identified sufficient conditions for a CSP to yield a backtrack-free solution and has shown, for example, that tree-like constraint graphs can be made to satisfy these conditions with a small amount of preprocessing. Our main purpose here is to further study classes of constraint graphs lending themselves to backtrack-free solutions and to devise efficient algorithms for solving them. Once these classes are identified and their solution complexities assessed, they can be used as targets for a problem simplification scheme: constraints can be selectively deleted from the original specification so as to transform the original problem into a backtrack-free one. Furthermore, the number of consistent solutions in the simplified problem will be used as a figure-of-merit to establish priority of value assignments in the backtracking search of the original problem. We show that this figure of merit can be computed in time comparable to that of finding a single solution for a class of easy problems.

Definition: ([15]) **An ordered constraint graph** is a constraint graph in which the nodes are linearly ordered to reflect the sequence of variable assignments executed by the Backtrack search algorithm. The **width of a node** is the number of arcs that lead from that node to previous nodes, 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.

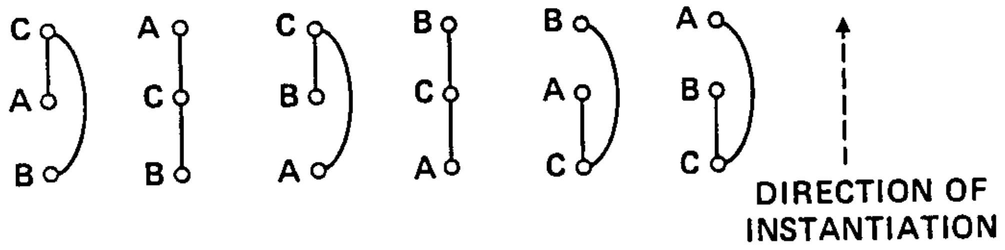


Figure 3 - Ordering of a constraint graph

Figure 3 presents six possible orderings of a constraint graph. The width of node C in the first ordering (from the left) is 2, while in the second ordering it is 1. The width of the first ordering is 2, while that of the second is 1. The width of the constraint graph is, therefore, 1. Freuder provided an efficient algorithm for finding both the width of a graph and the ordering corresponding to this width. He further showed that a constraint graph is a tree iff it is of width 1.

Montanari [28] and Mackworth [22] have introduced two kinds of local consistencies among constraints named **arc-consistency** and **path-consistency**. Their definitions assume that the graph is directed, i.e., each symmetric constraint is represented by two directed arcs.

Let $R_{ij}(x,y)$ stand for the assertion that (x,y) is permitted by the explicit constraint R_{ij}.

Definition: ([22]) Directed arc (X_i,X_j) is **arc consistent** iff for any value $x \in D_i$ there is a value $y \in D_j$ such that $R_{ij}(x,y)$.

Definition: ([28]) A path of length m through nodes $(i_0, i_1, \ldots, i_m)$ is **path consistent** if for any value $x \in D_{i_0}$ and $y \in D_{i_m}$ such that $R_{i_0 i_m}(x,y)$, there is a sequence of values $z_1 \in D_{i_1}, \ldots, z_{m-1} \in D_{i_{m-1}}$ such that

$$R_{i_0 i_1}(x,z_1) \text{ and } R_{i_1 i_2}(z_1,z_2) \text{ and} \cdots R_{i_{m-1} i_m}(z_{m-1},y). \tag{5}$$

$R_{i_0 i_m}$ may also be the universal relation, e.g., permitting all possible pairs.

A constraint graph is arc consistent if each of its directed arcs is arc consistent. Similarly, a constraint graph is path consistent if each of its paths is path consistent. "Achieving arc consistency" means deleting certain values from the domains of certain variables such that the resultant graph is arc-consistent while still representing the same overall set of solutions. To achieve path-consistency, certain **pairs of values** that were initially allowed by the input constraints should be disallowed. Montanari and Mackworth have proposed polynomial-time algorithms for achieving arc-consistency and path-consistency. In [23] it is shown that arc-consistency can be achieved in $O(ek^3)$ while path consistency can be achieved in $O(n^3k^5)$, where n is the number of variables, k is the number of possible values, and e is the number of edges.

Theorem 1([15]):

a. If the constraint graph has a width 1 (i.e. it is a tree) and if it is arc-consistent then it admits backtrack-free solutions.

b. If the width of the constraint graph is 2 and it is also path-consistent then it admits backtrack-free solutions.

□

The above theorem suggests that tree-like CSPs (CSPs whose constraint graph are trees) can be solved by first achieving arc consistency and then instantiating the variables in any width-1 order. Since this backtrack-free instantiation takes $O(ek)$ steps, and on trees $e=n-1$, the entire problem can be solved in $O(nk^3)$ [23]. The test for simplicity is also easily verified: it amounts to testing whether a given graph is a tree, and can be accomplished by an $O(n^2)$ spanning tree algorithm. Thus, tree-like CSPs are easy since they can be made backtrack-free after a preprocessing phase of low complexity.

The second part of the theorem tempts us to conclude that a width-2 constraint graph should admit a backtrack-free solution after passing through a path-consistency algorithm. In this case, however, the path-consistency algorithm may add arcs to the graph and increase its width beyond 2. This happens when the algorithm disallows value-pairs from a non adjacent variables (i.e. variables that were initially related by the universal constraint) and it is often the case that passage through a path-consistency algorithm renders the constraint graph complete. It may happen, therefore, that no advantage could be taken of the fact that a CSP possesses a width-2 constraint graph if it is not already path consistent. We are not even sure whether width-2 suffices to preclude exponential complexity.

In the following section we give weaker definitions of arc and path consistency which are also sufficient for guaranteeing backtrack-free solutions and have two advantages over those defined by [28] and [22] :

1. They can be achieved more efficiently, and

2. They add fewer arcs to the constraint-graph, thus preserving the graph width in a larger class of problems.

Note that adding an arc means that when the consistency algorithm rules out a pair of values, the implicit universal constraint must be replaced by an explicit constraint between the variables considered.

2.2 Algorithms for achieving directional consistency

a. The case of Width-1 (Trees)

Securing full arc-consistency is more than is actually required for achieving backtrack-free solutions. For example, if the constraint graph in figure 4 is ordered by (X_1,X_2,X_3,X_4), nothing is gained by making the directed arc (X_3,X_1) consistent.

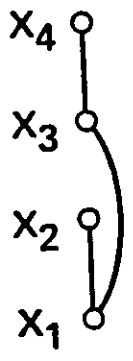


Figure 4 - Ordered constraint graph demonstrating the sufficiency of making arc (X_1,X_3) directional consistent.

To ensure backtrack-free assignment, we need only make sure that any value assigned to variable X_1 will have at least one consistent value in D_3. This can be achieved by making only the directed arc (X_1,X_3) consistent, regardless of whether (X_3,X_1) is consistent or not. We, therefore, see that arc-consistency is required only w.r.t. a single direction, the one in which Backtrack selects variables for instantiations. This motivates the following definitions.

Definition: Given an order d on the constraint graph R, we say that R is **d-arc-consistent** if all edges directed along d are arc-consistent.

Theorem 2:

Let d be a width-1 order of a constraint tree T. If T is d-arc-consistent then the backtrack search along d is backtrack-free.

proof:

Suppose that $X_1, X_2, \ldots, X_k$ were already instantiated. The variable X_{k+1} is connected to at most one previous variable (from the width-1 property), say X_i, which was assigned the value x_i. Since the directed arc (X_i, X_{k+1}) is along the order d, its arc-consistency implies the existence of a value x_{k+1} such that the pair (x_i, x_{k+1}) is permitted by the constraint $R_{i(k+1)}$. Thus, the assignment of x_{k+1} is consistent with all previous assignments. Reasoning by induction over k proves the theorem.

□

An algorithm for achieving directional arc-consistency for any ordered constraint graph is given next (The order $d = (X_1, X_2, \ldots, X_n)$ is assumed)

DAC- *d*-arc-consistency

```
1. begin
2.   For $i=n$ to 1 by -1 do
3.     For each arc $(X_j,X_i)$; $j<i$ do
4.       REVISE$(X_j,X_i)$
5.     end
6.   end
7. end
```

The algorithm REVISE(X_j,X_i), given in [22] , deletes values from the domain D_j until the directed arc (X_j,X_i) is arc-consistent.

REVISE(X_j,X_i)

```
1. begin
2.   For each $x \in D_j$ do
3.     if there is no value $y \in D_i$ such that $R_{ji}(x,y)$ then
4.       delete $x$ from $D_j$
5.   end
6. end
```

To prove that the algorithm achieves *d*-arc-consistency we have to show that upon termination, any arc (X_j,X_i) along d $(j<i)$, is arc-consistent. The algorithm revises each *d*-directed arc once. It remains to be shown that the consistency of an already processed arc is not violated by the processing of coming arcs. Let arc (X_j,X_i) $(j<i)$ be an arc just processed by REVISE(X_j,X_i). to destroy the consistency of (X_j,X_i) some values should be deleted from the domain of X_i during the continuation of the algorithm. However, according to the order by which REVISE is performed from this point on, only lower indexed variables may have their set of values updated. Therefore, once a directed arc is made arc-consistent its consistency will not be violated.

For comparison we give the algorithm AC-3 [22], that achieves full arc-consistency:

AC-3

```
1. begin
2.  $Q \leftarrow \{ (X_i,X_j) \mid (X_i,X_j) \in$ arcs, $i \neq j \}$
3.   while $Q$ is not empty do
         select and delete arc $(X_k,X_m)$ from $Q$
5.        REVISE($X_k,X_m$)
6.        if REVISE($X_k,X_m$) caused any change then
7.          $Q \leftarrow Q \cup \{(X_i,X_k) \mid (X_i,X_k) \in$ arcs, $i \neq k,m\}$
8.   end
9. end
```

The complexity of AC-3, is $O(ek^3)$. The advantage of directional arc-consistency algorithm is that it eliminates the need for looping when REVISE makes a change and hence each arc is processed exactly once. It is also optimal, because even to verify directional arc-consistency each arc should be inspected once, and that takes k^2 tests. Note that when the constraint graph is a tree, the complexity of the directional arc-consistency algorithm is $O(nk^2)$.

Theorem 3:

A tree-like CSP can be solved in $O(nk^2)$ steps and this is optimal.

proof:

Once we know that the constraint graph is a tree, finding an order that will render it of width-1 takes $O(n)$ steps. A width-1 constraint tree can be made *d*-arc-consistent in $O(nk^2)$ steps, using the DAC algorithm. Finally, the backtrack-free solution on the resultant tree is found in $O(nk)$ steps. Summing up, finding a solution to tree-like CSP's takes, $O(nk) + O(nk^2) + O(n) = O(nk^2)$. This complexity is also optimal since any algorithm for solving a tree-like problem must examine each constraint at least once, and each such examination may take, in the worst case, k^2 steps, especially when no solution exists and the constraints permit very few pairs of values (see Appendix for a formal proof of optimality).

□

Interestingly, if we apply DAC w.r.t. order *d* and then DAC w.r.t. the reverse order we get a full arc-consistency for trees. We can, therefore, achieve full arc-consistency on trees in $O(nk^2)$. Algorithm AC-3, on the other hand, has a worst case performance on trees of $O(nk^3)$ as is shown next. Figure 5 illustrates a CSP problem that has a chain-like constraint graph.

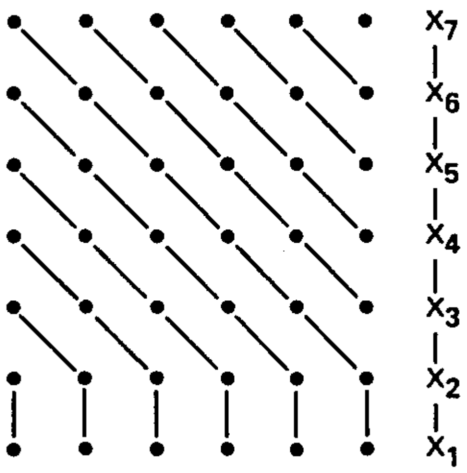


Figure 5 - Tree-constraint network showing worst-case of AC-3

There are 7 variables, each with k values, constrained as shown in the figure. Each row represent a variable, the points represent values and the connecting lines describe the allowed pairs of values. Since AC-3 does not determine the order in which the arcs enter REVISE, we will impose an ordering which will be particularly bad for it: in increasing node index, that is, first (X_1,X_2) then (X_2,X_3) etc. However, because AC-3 employs a last-in-first-out policy, when a new arc is inserted to the queue it is given the highest priority. Therefore, Arc (X_1,X_2) will be processed first, then arc (X_2,X_3) and, since a value was deleted from X_2, arc (X_1,X_2) will be inserted back to the queue and processed. No change occurred and, therefore, the next arc to be processed is (X_3,X_4), this will cause processing of (X_2,X_3) again which, in turn, causes the processing of (X_1,X_2) and so on. We see that each arc will be processed $k-1$ times resulting in a total complexity of $O(nk^3)$. It has recently been found [27], that using special data structures, the general arc-consistency on graphs can be improved to achieve performance of O(ek^2). This is done at the expense of additional book-keeping, maintaining, for each value, the number of values that match it in each neighboring variable. An arc may be processed several times but the amount of processing is carefully controlled so that the total complexity is reduced. The advantage of the directional arc-consistency algorithm is that it is much simpler to implement and is more suitable for parallel implementation.

b. The case of Width-2

Order information can also facilitate backtrack-free search on width-2 problems by making path-consistency algorithms directional.

Montanari had shown that if a network of constraints is consistent w.r.t. all paths of length 2 (in the complete network) then it is path-consistent. We will show that directional path-consistency w.r.t. length-2 paths is sufficient for ensuring backtrack-free search on a width-2 problems.

Definition: A constraint graph, R, is **d-path-consistent** w.r.t. ordering $d = (X_1, X_2, \ldots, X_n)$, if for every pair of values (x,y), $x \in X_i$ and $y \in X_j$ such that $R_{ij}(x,y)$ and for every $k > i,j$. there exists a value $z \in X_k$, $k > j$, such that $R_{ik}(x,z)$ and $R_{kj}(z,y)$.

Theorem 4:

Let d be a width-2 ordering of a constraint graph, R. If R is directional arc and path-consistent w.r.t. d then it is backtrack-free.

Proof:

To ensure that a width-2 ordered constraint graph is backtrack-free it is required that each variable selected for instantiation will have some values consistent with all previously chosen values. Suppose that $X_1, X_2, \ldots, X_k$ were already instantiated. The width-2 property implies that variable X_{k+1} is connected to at most two previous variables. If it is connected to X_i and X_j, $i,j \leq k$ then directional path consistency ensures that for any assignment of values to X_i, X_j there exists a consistent assignment for X_{k+1}. If X_{k+1} is connected to one previous variable, then directional arc-consistency ensures the existence of a consistent assignment.

□

An algorithm for achieving directional path-consistency on ordered graphs must manage not only the changes made to the constraints but also the changes made to the graph, i.e., the addition of new arcs. To describe the algorithm we use the matrix representation of constraints, with a diagonal matrix R_{ii} representing the set of values permitted for variable X_i. The algorithm is described using the operations of intersection and composition, writing R'_{ij} & R''_{ij} for the intersection of R'_{ij} and R''_{ij}.

Given a network of constraints $R = (V,E)$ and an ordering $d = (X_1, X_2, \ldots, X_n)$, the next algorithm achieves path-consistency and arc-consistency relative to d.

DPC-d-path-consistency

```
 begin
1. $Y = R$
2. for $k=n$ to 1 by -1 do
     (a) $\forall$ $i \le k$ connected to $k$ do
           $Y_{ii} \longleftarrow Y_{ii}$ & $Y_{ik} \cdot Y_{kk} \cdot Y_{ki}$ /* this is REVISE($i,k$)
     (b) $\forall$ $i,j \le k$ s.t. $(X_i,X_k),(X_j,X_k) \in E$ do
           $Y_{ij} \longleftarrow Y_{ij}$ & $Y_{ik} \cdot Y_{kk} \cdot Y_{kj}$
           $E \longleftarrow E \cup (X_i,X_j)$
3. end
 end
```

Step 2a is equivalent to REVISE(i,k), and performs directional arc-consistency. Step 2b starts after completing 2a for $i < k$. Step 2b updates the constraints between pairs of variables transmitted by a third variable which ranks higher in d. If X_i, X_j, $i,j<k$ are not connected to X_k then the constraint between the first two variables is not affected by X_k. If only one variable, X_i, is connected to X_k, the effect of X_k on the constraint (X_i,X_j) will be computed by step 2a of the algorithm. The only time a variable X_k affects the constraint between a pair of earlier variables is when it is connected to both, and it is in this case that a new arc may be added to the graph. The DPC algorithm will connect any parent-nodes having a common successor. Note that the convenience of writing the algo-

rithm using matrix notation does not imply that it should be implemented that way, actually, representing constraints as sets of compatible pairs of values is easier and often requires less space.

The complexity of the directional-path-consistency algorithm is $O(n^3k^3)$. The number of times the inner loop, 2b, is executed for variable X_i is at most $O(deg^2(i))$ (the number of different pairs of parents of i), and each step is of order k^3. The computation of loop 2a is completely dominated by the computation of 2b, and can be ignored. Therefore, the overall complexity is

$$\sum_{i=2}^{n} deg^2(i)k^3 = O(n^3k^3) \tag{6}$$

It is also bounded by $O(DEG \cdot ek^3)$ when DEG is the maximum degree in the graph, thus, for sparse networks, the complexity is linear in the number of arcs.

Applying directional-path-consistency to a width-2 graph may increase its width and therefore, does not guarantee backtrack-free solutions. Consequently it is useful to define the following subclass of width-2 problems.

Definition: A constraint graph is **regular width-2** if there exists a width-2 ordering of the graph which remains width-2 after applying d-path-consistency, DPC.

A ring constitutes an example of a regular-width-2 graph. Figure 6 shows an ordering of a ring's nodes and the graph resulting from applying the DPC algorithm to the ring. Both graphs are of width-2.

Theorem 5:

A regular width-2 CSP can be solved in $O(n^3k^3)$

Proof:

Regular width-2 problem can be solved by first applying the DPC algorithm and then performing a backtrack-free search on the resulting graph. The first takes $O(n^3k^3)$ steps

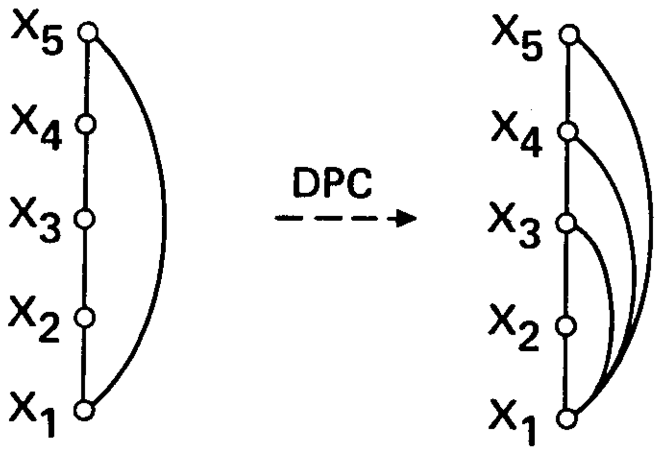


Figure 6 - A regular width-2 graph

and the second $O(ek)$ steps.

□

The nice feature of regular width-2 CSPs is that they can be easily recognized and therefore can also be used as targets for simplification.

A graph may have many width-2 orderings, only some of which remain width-2 after being processed by the DPC algorithm. Arnborg [1] describes a linear time algorithm for recognizing regular width-2 graphs, and generating the corresponding ordering (see also [3]). The algorithm eliminates nodes from the graph in the following recursive manner: If there is a node of degree smaller or equal to 2, eliminate any node having the smallest degree, connect its neighbors in the residual graph (if they were not previously connected), and continue. The procedure terminates when the graph is empty or when no more nodes can be eliminated. In the former case, the problem is regular width-2 and the elimination order provides the required ordering (in reverse). Otherwise, the graph is not regular-width-2.

The relationship between the width of the graph and the tractability of the problem can be further generalized to higher widths and higher levels of consistencies, see

[10]. One such extension is given in the following subsection.

2.3 Adaptive-consistency: a general strategy for achieving a backtrack-free solution.

The analysis of easy problems has two objectives. First, easy problems may serve as sources of advice, and hence, they must be recognized and solved efficiently. Second, the original problem itself may be categorized as "easy" or may undergo structural transformations to admit an easy, backtrack-free solution. In such cases it is advisable to conduct or complete the search using specialized strategies, tailored to exploit the unique features that render the problem easy. The procedure described in this subsection, adaptive-consistency, was devised to meet this second objective.

Both arc-consistency and path-consistency can be thought of as preprocessing procedures that install uniform levels of local consistency throughout the constraint graph. A natural generalization of these methods is **K-consistency** defined by Freuder [14]. K-consistency implies the following condition: Choose any set of $K-1$ variables along with values for each that satisfy all the constraints among them. Now choose any K^{th} variable. There exists a value for the K^{th} variable such that the K values taken together satisfy all constraints among the K variables. Freuder has shown that a K-consistent CSP having a width-(K-1) ordering admits a backtrack-free solution in that ordering. This suggests that one should be able to preprocess a CSP for a backtrack-free solution by passing it through an algorithm establishing K-consistency. However, since algorithms that achieve K-consistency change the width of the graph, it is difficult to determine in advance the level of K desired, namely, the amount of pre-processing required to facilitate backtrack-free solution.

An alternative generalization, specifically suited for directional consistency, would be to adjust the level of consistency on a node by node basis. We call this method **adaptive-consistency**, as it lets the evolving structure of the (directional) constraint-graph dictate the amount of pre-processing required for each node.

Adaptive-consistency process nodes in decreasing order. Given an ordering *d*, adaptive-consistency lets each variable impose consistency among *i* variables which precede it and are connected to it at the time of processing. The size of this set represents the current width of the node. We will say that variable X_i **imposes i-consistency** on a set of $i-1$ variables if any $i-1$-tuple of the later is consistent with at least one value of X_i. The procedure is illustrated in figure 7. Consider the constraint-graph of figure 7a, shown in one of its minimal-width orderings. Adaptive-consistency in the order (*E*,*D*,*C*,*A*,*B*) starts at node *B* and, having width-1, *B* imposes 2-consistency on *D* (i.e., establishing arc-consistency on (D,B)). The domain of *D* is thus tightened. When *A* is processed next, it imposes 3-consistency on (C,D) (A's width is 2) which may amount to adding a constraint and an arc between *C* and *D* (as in figure 7b). When the algorithm reaches node *C*, *C*'s width is 2 and, therefore, a 3-consistency needs to be imposed on (E,D) but, since the arc between (*E*,*D*) already exists, the effect would be to merely tighten this constraint. The resulting graph is shown in Figure 7b.

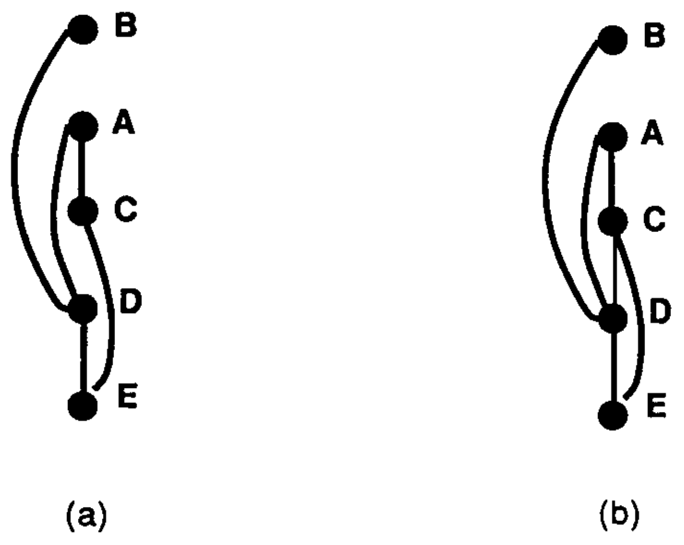


Figure 7: An ordered graph before and after being pre-processed

Formally, adaptive-consistency can be described by the following procedure. For each variable, X, PARENTS(X) is the set of all predecessors of X currently connected to it which precede it. The parent-set of each variable is computed only when it needs to be processed.

```
adaptive-consistency( $X_1, \ldots, X_n$)

1. Begin
2.   for r=n to 1 by -1 do
3.   compute PARENTS($X_r$)
4.   perform consistency($X_r$, PARENTS($X_r$))
5.   connect by arcs all elements in PARENTS($X_r$) (if they are not yet connected)
6. End
```

The procedure **consistency(V,SET)** records a constraint among the variables in SET, induced by V. In other words, those instantiations of variables in SET consistent with at least one (consistent) value of V are retained, while the rest are ruled out. To determine consistency, the procedure considers **all** constraints, old and new, which were generated up to this point by the algorithm, and which are applicable to the variables in V union SET. A constraint is considered applicable if it involves variable V and a subset (possibly empty) of variables from SET. In figure 7, for instance, when consistency(C,$\{D,E\}$) is executed the applicable constraints are the input constraint (C,E) and the recorded constraint between C and D. These two constraints will be considered to determine the set of value-pairs of (D,E) consistent with at least one value of C.

The induced graph, generated by adaptive-consistency, is identical to the one generated by executing directional path-consistency on the ordered graph. It can be found prior to applying the procedure by connecting, recursively, any two parents sharing a common successor, processing nodes in descending order. When adaptive-consistency terminates, backtrack can solve the problem without any dead-end, simply consulting the input constraints and those recorded by the procedure.

In general, an ordered constraint graph will be backtrack-free if for every subset of consistently instantiated variables $(X_1, \ldots, X_r)$ there exists a value for X_{r+1} which is consistent with the current instantiation of $(X_1, \ldots, X_r)$.

Theorem 6:

An ordered constraint-graph processed by adaptive-consistency is backtrack-free.

Proof: Suppose that the first r variables were already instantiated, and assume X_{r+1} has a parent-set of size t, $X_1, \ldots, X_t$. This parent set was identified and processed by adaptive-consistency, namely, the procedure consistency(X_{r+1}, $\{X_1, \ldots, X_t\}$) recorded a constraint on the t variables ensuring that any t-tuple which is not consistent with at least one consistent value of X_r is ruled out. Therefore, any consistent t-tuple, which passed this restriction, is extendible to X_{r+1}.

□

An important feature of the adaptive-consistency scheme is that, unlike the undirectional K-consistency method, it permits the calculation of a bound on its complexity prior to conducting the search. Let $W(d)$ be the width associated with ordering d and $W^*(d)$ the width of the graph induced by adaptive-consistency in that ordering. The worst-case complexity of adaptive-consistency along d is exp($W^*(d)$+1) since the consistency procedure records a constraint on $W^*(d)$ variables induced by their common successor, a process comparable to solving a CSP with $W^*(d)$+1 variables. $W^*(d)$ can be found in O($n+e$) time [35] prior to the actual processing. Let $W^* = \min_d \{W^*(d)\}$. If we had a way of finding the ordering that minimizes $W^*(d)$, a tighter bound, $exp(W^*+1)$, could be given for the overall time complexity of CSPs. However, since W* is hard to compute (an NP-complete task [2]), W*(d*) can be used as a good upper bound, where d* realizes the width of the constraint-graph. The space complexity is exp(W*(d)) since at most n constraints are added by adaptive-consistency, each involves W*(d) variables or less, and the specification of each such constraint requires O($k^{W^*(d)}$) value combinations in the worst case. For more details see [10].

2.4 Other use of network-based features

Another approach which also exploits the structure of the constraint graph and offers a general method of improving Backtrack's performance, involves the notion of **nonseparable components** [13].

definition: A connected graph $G(V,E)$ is said to have a **separation vertex** v if there exist vertices a and b, such that all paths connecting a and b pass through v. A graph which has a separation vertex is called **separable**, and one which has none is called **nonseparable**. A subgraph with no separation vertices is called a **nonseparable component**.

An $O(|E|)$ algorithm for finding all the nonseparable components and the separation vertices is given in [13]. It is also known that the nonseparable components are interconnected in a tree-structured manner.

Let R be a graph and SR be the tree in which the nonseparable components $C_1,C_2,\ldots,C_r$ and the separating vertices $V_1,V_2,\ldots,V_t$ are represented by nodes. A width-1 ordering of SR dictates a partial order on R, d^s, in which each separating vertex precedes all the vertices in its children components of SR. The DAC algorithm can be applied to the resulting tree, treating each component as a compound variable. Figure 8 shows a graph with 10 nodes identified by its components and its separating vertices together with an ordering on R which is induced by a width-1 ordering on SR..

Let SR be the directed tree associated with a given graph R. Backtrack, given both R and SR will work on each component starting from leaf components towards the

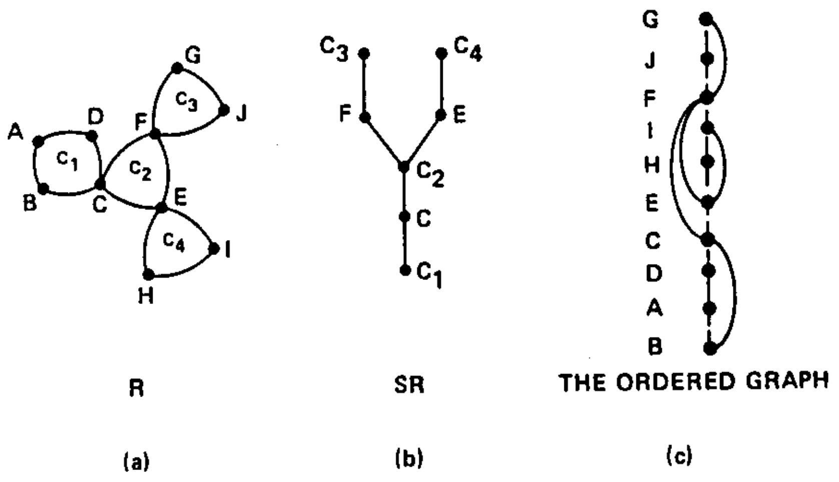


Figure 8- A Graph and its decomposition into nonseparable components

root component. The algorithm is described next:

```
Backtrack($R$,$SR$)
begin
 0. assign $TSR \Leftarrow SR$. /*$TSR$ stands for temporary $SR$.*/
 1. If $TSR = \Phi$ then generate-solutions($SR$).
 2. let $C_1, \ldots, C_t$ be the leaf components in $TSR$.
 3.  for each $C_i$ a leaf, do
 4.    perform backtrack on this subgraph $C_i$
 5.    associate with the component $C_i$ all solutions found, denoted by $\rho_{C_i}$
 6.    delete from the domain of $P_{C_i}$ values which do not appear in $\rho_{C_i}$
       ($P_{C_i}$ is the parent separating vertex of $C_i$)
 7.    delete $C_i$ from $TSR$.
 8. end.
 9. goto 1.
end.
```

The procedure "generate-solution(*SR*)" will generate all the solutions by traversing the *SR* tree from the root to leaves and concatenating the partial solutions found for each component that can be "joined" w.r.t. the separating vertices. i.e. for two connected components that share the same separating vertex all partial solutions having the same value for that vertex can be joined. The complexity of this algorithm is $O(nk^r)$ when r is the size of the largest component. We therefore see that if R has a decomposition into small clusters of non-separable components then backtrack can utilize this structure and improve its performance. The use of this decomposition is also discussed in [15] although the solution procedure is more involved (there, the terms "separation vertices" and "non-separable components" are named "articulation nodes" and "biconnected components" respectively).

The properties of tree-structured constraint-graphs can also be used to guide the ordering of variables. A scheme which promises several possibilities is based on the following observation: If, in the course of a backtrack search, we remove from the constraint-graph the nodes corresponding to instantiated variables and find that the remaining subgraph is a tree, then the rest of the search can be completed in linear time (e.g. by the DAC algorithm presented before). Consequently, the aim of ordering the variables should be to instantiate as quickly as possible, a set of variables that cut all cycles in the graph. Indeed, if we identify m variables which form such a cycle-cutset, The entire CSP can be solved in at most $O(k^m nk^2)$ steps; we simply solve the trees resulting form each of the k^m possible instantiations of the variables in the cutset. Thus, in graphs where m is small, enormous savings can be realized using simple heuristics for selecting small cycle-cutsets. For a detailed evaluation of this approach see [11].

The two approaches, non-separable components and cycle-cutset, can be combined as follows: Separate the graph into nonseparable components, and solve each component using the cycle-cutset scheme. If the size of the largest component is r and the largest cycle-cutset in any component is m, then the combined algorithm has the complexity $O(n \cdot r \cdot k^{m+2})$. It should be interesting to experimentally compare this approach with adaptive-consistency, which also requires exponential space.

2.5 Summary

1. This section analyses several topological features of constraint networks that render them amenable to backtrack-free solution. These features can be used either as sources of advice to universal problem-solving schemes such as backtrack or as pointers to invite specialized problem solving strategies when certain conditions are matched.

2. The introduction of directionality into the notions of arc and path consistency enabled us to extend the class of recognizable easy problems beyond trees, to include regular width-2 problems.

3. Using directionality we were able to devise improved algorithms for solving simplified problems and to demonstrate their optimality. In particular, it is shown that tree-structured problems can be solved in $O(nk^2)$ steps, and regular width-2 problems in $O(n^3k^3)$ steps.

4. We have introduced a new scheme called "adaptive-consistency" which renders any CSP backtrack-free and requires time and space complexities of $O(exp(W^*(d^*)))$ which can be determined in advance.

5. We have shown how the tree processing algorithm can be applied to general CSPs by separating the graph into its connected components and using the cycle-cutset method within each component. CSPs can be then solved in $O(nrk^{m+2})$ when r is the size of the largest component and m, $(m<r)$, is the size of largest cycle-cutset.

3. ADVICE-GENERATION

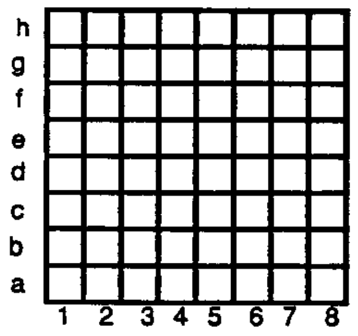
3.1 Description of scheme

Returning to our primary aim of studying easy problems, we now show how advice can be generated using a tree approximation. Suppose that we want to solve an n variables CSP using Backtrack with $X_1, X_2, \ldots, X_n$ as the order of instantiation. Let X_i be the variable to instantiate next, with $x_{i1}, x_{i2}, \ldots, x_{ik}$ the possible candidate values. Ideally, to minimize backtracking we should first try the values which are more likely to lead to a consistent solution but, since this likelihood is not known in advance, we may estimate it by counting the number of consistent solutions that each candidate admits in some approximate, easily solved problem. We generate a tree-like relaxation of the remainder of the problem (i.e. the problem induced by the previous instantiations) by deleting some of the explicit constraints given, then count the number of solutions consistent with each of the *k* possible assignments, and finally use these counts as a figure of merit for scheduling the candidate values for assignment.

In [8] we discuss ways of generating minimal spanning tree (MST) approximations, and justify two alternative heuristic measures for the arcs' weights. The weights used in the experiments described here is the **number of compatible value pairs in the constraint** represented by the arc. The MST could be generated during search, in which case for each value assignment a new tree, with arc-weight's based on the restrictions imposed by past instantiations, would be determined. This would require $O(n^2)$ operations (the cost of generating an MST), to each node in the search space. Instead we simplify the implementation by generating a fixed MST for each level in the search **prior to search.** After ordering the variables in decreasing order of their degrees, the MST for each level is computed, based on the structure of the residual problem starting from that level. The complexity of generating the various trees in this way is $O(n^3)$ and is independent of the size of the search space.

The **Advised Backtrack algorithm** (abbreviated ABT) uses an **advising procedure** to select a value for the next variable. The selection of value for the i^{th} variable is done as follows: For each of the remaining variables all the values that are not consistent with previous instantiations are eliminated (temporarily, for the current advice generation). Using the tree precomputed for level i, the number of solutions consistent with each candidate value is computed, considering only constraints present in the tree. The advising procedure returns to backtrack the set of candidates with their associated counts and backtrack then chooses the one with the highest count.

Consider, for example, the 8-queen problem where the task is to assign 8 queens to the board such that queens will not attack one another (figure 9a). A trace of running a regular-backtrack's on this problem is given in figure 9b, where rows, (representing variables), are instantiated in the order (d,e,c,f,b,g,h,a), and queens are assigned from left to right. (This order was chosen to accentuate backtrack's maladies). The number of backtracking points in this trace is 11.



(a)

d	e	c	f	b	g	h	a
2	5	4	3	6			
				1			
			8	1			
				6			
				7	1		
					6		
			7	1			
				6			
			8	3	6		
					1		
			1	6			
				3	6	4	7

(b)

Figure 9: A trace of 8-queen problem by regular backtrack

When ABT solves this problem it assigns the first three rows, d,e,c queens on cells 8,1,5, respectively. In order to choose a value for the next row, f, the remaining subproblem is consulted, involving the variables, f,g,h,a,b. Figure 10a represents this subproblem showing the cells compatible with past assignments. The first task is to simplify the problem by generating a spanning tree. The weights associated with each constraint are computed based on the number of compatible value-pairs between any two rows (figure 10b) and the minimum spanning tree is accordingly given in figure 10c. (This computation is performed prior to the search). On this simplified problem, (considering only the constraints among the pairs (f,g) (g,h),(f,b),(b,a)), the number of solutions associated with each candidate value of f, are computed, yielding 3 solutions compatible with f =3, 10 solutions compatible with f =7 and 12 solutions are compatible with f =4. These counts are returned to backtrack which chooses the assignment f =4 as the most promising guess. Using ABT on this problem instance, no backtracking was required.

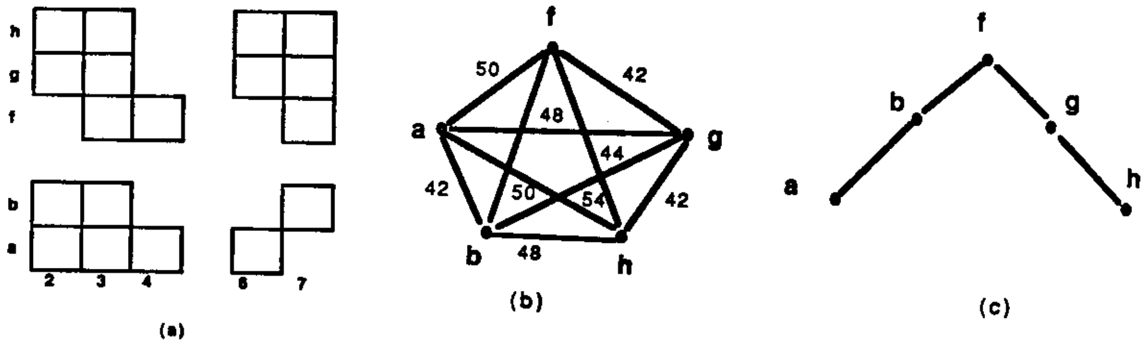


Figure 10: Advice-generation for the 8-queen

Note that the advising procedure and backtrack are completely separated, namely, there is no flow of information from advice to backtrack beside the priority of values. For instance, if in the course of counting the number of solutions, certain values could be pruned by counting algorithm, these values should not and **were not pruned**, by backtrack. The reason is that conclusions of the advising procedure reflect the restrictions imposed by instantiating the first $i-1$ variables, once those are changed, the restricted subproblem would change as well. Nevertheless, more complex interaction with backtrack could be used to advantage, for example, considering future instantiations suggested by the tree and checking whether they satisfy the entire set of constraints. Another possibility would be to maintain a table of currently consistent values for each variable and, rather than recomputing the consistent values of future variables with each invocation of advice, to update the table when a new assignment occurs. In the experiments reported, however, such interactions were not considered.

We shall next show how counting the number of consistent solutions can be embedded within the d-arc-consistency algorithm, DAC, on trees.

Any width-1 order, d, on a constraint tree determines a directed tree in which a parent always precedes its children in d (arcs are directed from the parent to its children). Let $N(x_{jt})$ stand for the number of solutions in the subtree rooted at X_j consistent with the assignment of x_{jt} to X_j. Consider a node X_j with all its successor nodes as in figure 11. Looking first on the relation between X_j and a specific child node X_c, it is clear that the value x_{jt} can participate in a solution with each compatible value of X_c, no matter what values are assigned to variables in the subtree rooted at X_c. Therefore the number of partial solutions consistent with x_{jt}, in the subtree rooted at X_c, is a sum of those contributed by each compatible value of X_c. Namely:

$$N(x_{jt} \mid \text{in the tree rooted at } X_c) = \sum_{\{x_{cl} \in D_c \mid R_{jc}(x_{jt}, x_{cl})\}} N(x_{cl}) \tag{7}$$

Since the partial solutions coming from different successor nodes can be combined in all possible ways, the number of solutions in the subtree rooted at X_j will be their product. Therefore, $N(\cdot)$ satisfies the following recurrence:

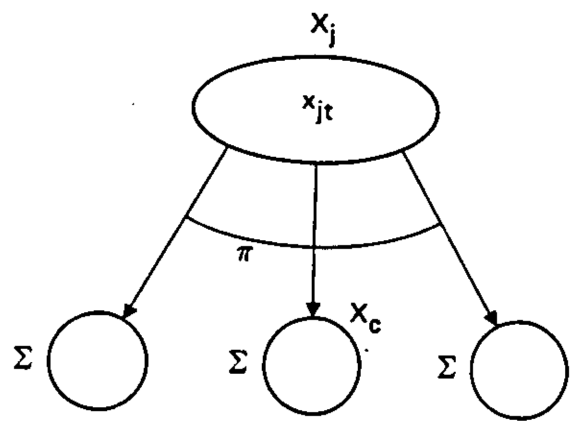


Figure 11 - Schematic computation of number of solutions in trees.

$$N(x_{jt}) = \prod_{\{c \mid X_c \text{ is a child of } X_j\}} \sum_{\{x_{cl} \in D_c \mid R_{jc}(x_{jt}, x_{cl})\}} N(x_{cl}) \tag{8}$$

From this recurrence it is clear that the computation of $N(x_{jt})$ may follow the exact same steps as in DAC: simultaneously with testing that a given value x_{jt} is consistent with each of its children nodes, we simply transfer from each child of X_j to x_{jt} the sum total of the counts computed for the child's values that are consistent with x_{jt}. The overall value of $N(x_{jt})$ will be computed later on by multiplying together the summations obtained from each of the children. The computation starts at the leaves, initialized to $N = 1$, and progresses towards the root. Each variable performs the counting only after all its child's nodes computed their counts as in the counting procedure COUNT that follows.

The COUNT procedure contains the arc-consistency algorithm using REVISE described earlier and performs the calculation according to the recurrence (8) above. The algorithm terminates when the root is assigned counts for all its values. The COUNT procedure is defined for a parent node V_p and all its children., $V_1,...,V_r$. Notice that COUNT could be implemented without explicitly performing arc-consistency (line 3 can be deleted). A value that gets the count of "0" can be eliminated.

```
COUNT($V_p, V_1, V_2, \ldots, V_r$)

1. begin
2.   For each $(V_p, V_l)$ do
3.     REVISE$(V_p, V_l)$
4.     For each $v_{ps} \in D_p$ (for each value of $V_p$) do
5.       $n_l(v_{ps}) = \sum_{v_{lt}; R_{pl}(v_{ps}, v_{lt})} N(v_{lt})$
6.   end
7.   For each $v_{ps} \in D_p$ do
8.     $N(v_{ps}) = \prod_{l; V_l \text{ a child}} n_l(v_{ps})$
9.   end.
10. end
```

Like REVISE, line 4 takes k^2 steps, and therefore, for each parent node, V_p, processing takes $k^2 \cdot deg(V_p)$ steps. Thus, the counting for all the n variables in the subtree, sums up to $O(nk^2)$, not increasing the complexity of the directional arc-consistency by more then a constant.

In the remainder of this section we compare the performance of Advised Backtrack (ABT) with that of Regular Backtrack (RBT) analytically, via worst case analysis, and experimentally, on randomly generated problems.

3.2 Worst case analysis

An upper bound is derived for the number of consistency checks performed by the two algorithms as a function of the problem's parameters and the number of backtracks performed. A consistency check occurs each time the algorithm checks to verify whether a pair of values is consistent w.r.t. the corresponding constraint.

Let B_A and B_R be the number of backtracks, and C_A and C_R the number of consistency checks performed by ABT and RBT, respectively. The problem's parameters are n, the number of variables, k, the number of values for each variable, $|E|$, the number of arcs, and deg, the maximum degree over the variables in the graph.

The number of backtracks performed by an algorithm is equal to the number of explicated leaves in the search tree. We assume that

$$\text{Number of nodes expanded} = \alpha \cdot B \tag{9}$$

approximately holds for some constant α. (This truly holds only for uniform trees where α is the branching factor.) Therefore we use the number of backtracks as a surrogate for the number of nodes expanded. Let $\hat{c}_A$ and $\hat{c}_R$ be the maximum number of consistency checks performed at each node by ABT and RBT, respectively. We have:

$$C \leq B \cdot \hat{c} \tag{10}$$

First consider RBT. The number of consistency checks performed at the i^{th} node in the order of instantiation is less then $k \cdot deg(i)$. That is, each of this variable's values should be checked against the previous assigned values for variables which are connected to it. This yields:

$$C_R \leq k \cdot deg \cdot B_R \tag{11}$$

The ABT algorithm performs all its consistency checks within the advice generation phase. For the i^{th} variable , a tree of size $n-i$ is generated. The consistency checks performed on this tree occur in two phases. In the first phase, for each variable in the tree, the values consistent with the previous assignments are determined. The number of consistency checks for a variable v in the tree equals $k \cdot w(v)$, where $w(v)$ is the number of variables connected to v which were already instantiated. Therefore, for all variables in the tree, we have

$$k \cdot \sum_{v \in tree} w(v) \leq k \cdot |E| \, . \tag{12}$$

(Were we to use the tables of current values suggested earlier, this number would be reduced to O(nk)). The second phase counts the number of solutions. We have showed that counting takes no more then $(n-i) \cdot k^2$ steps which is bounded by nk^2. Hence,

$$C_A \leq (k \cdot |E| + n \cdot k^2) \cdot B_A \tag{13}$$

We now want to determine the ratio $\frac{B_A}{B_R}$ for which it will be worthwhile to use Advised Backtrack instead of Regular Backtrack and, as a first approximation, will treat the upper-bounds (11,13) as tight estimates. In other words, even though

$$C_A \leq C_R \tag{14}$$

is not implied by

$$(k \cdot |E| + n \cdot k^2) \cdot B_A \leq k \cdot deg \cdot B_R\ , \tag{15}$$

we take (15) as an indicator for the utility of ABT. From (15) we get

$$\frac{B_R}{B_A} \geq \frac{|E|}{deg} + \frac{nk}{deg}\ , \tag{16}$$

and, using

$$\frac{|E|}{deg} \leq n\ , \tag{17}$$

(16) holds when

$$\frac{B_R}{B_A} \geq n + \frac{nk}{deg}\ . \tag{18}$$

Thus, ABT is expected to result in a reduction of the number of consistency checks only if it reduces the number of backtracks by a factor greater than $(n+\frac{nk}{deg})$. Therefore, the potential of the proposed method is greater in problems where the number of backtrackings is exponential in the problem size.

3.3 The utility of advice-generation

Test cases were generated at random using four parameters: the number of variables n, the number of values for each variable k, the probability p_1 of having a constraint (an arc) between any pair of variables, and the probability p_2 that a constraint allows a given pair of values. Two performance measures were recorded: the number of backtrackings (B) and the number of consistency checks performed, C. The latter being an indicator of the overall running time of the algorithm. What we expect to see is that the more difficult the problem, the larger the benefits resulting from using advised Backtrack.

Two classes of problems were tested. The first, containing 10 variables and 5 values, were generated using $p_1 = p_2$=0.5, and the second with 15 variables and 5 values, generated using p_1=0.5 and p_2=0.6. 10 problems from each class were generated and solved by both ABT and RBT. For both algorithms, variables were instantiated in decreasing order of their degrees. The order of value selection was determined by the advice mechanism in ABT and at random in RBT. Therefore, while ABT solved each problem instance just once, RBT was used to solve each problem several (five) times to account for variations in value-selection order.

Figures 12 and 13 display performance comparisons for both classes of problems. In figure 12, the horizontal axis gives the number of backtrackings performed by RBT, the vertical axis gives the number of backtrackings performed by ABT, and each point represent one problem instance. The darkened circles correspond to instances from the first (easier) class while empty circles correspond to instances of the second (harder) class. We observe an impressive saving in B when advice is used, especially for the second (harder) class. Figure 13 compares the number of consistency checks. Here, we observe that in many instances the number of consistency checks in ABT is larger than in RBT, indicating that, in these cases, the extra effort spent in "advising", was not worthwhile.

These results are consistent with the theoretical prediction of the preceding subsection. If we substitute the parameters of the first class of problems in (18) we get that B_A should be smaller than B_R by at least a factor of 20 (25 for the second class of problems) to yield an improvement in overall performance. Many of the problems, however, were not hard enough (in terms of the number of backtrackings) to achieve these levels. In appendix II we give an average analysis of RBT performance for this class of problems and, indeed, we observe that, on the average, these problems are linear in *n*, i.e. easy to solve.

Figure 14 compares the two algorithms in only those problems that turned out to be difficult; it displays the number of consistency checks associated with instances requiring at least 70 backtrackings in RBT. We see that the majority of these instances were solved more efficiently by ABT than RBT.

As a result of these findings we concluded that the "advising" scheme was too sophisticated for the tested problems, i.e. it produced very good advise at too high a cost. In order to reduce this costs, the following scheme was devised: instead of generating advice based on a full spanning tree of the remaining problem we trimmed the tree to contain only a fixed number , *l*, of variables. The number of consistent solutions was counted in the same way, based only on this partial tree. This approach enabled us to test ABT using a wide range of advice **strength**, starting from a strong advice by considering all nodes in the tree, through the weakest possible advice, considering just one node (providing no useful information).

Figures 15 and 16 summarize the performance of this **variable advice** scheme. The x-axis displays the strength of the advice, i.e. the parameter *l*. The left vertical axis gives the number of consistency checks and the right vertical axis gives the number of backtracking. Figure 15 displays average performances w.r.t. problems in class 1 while figure 16 gives the results on problems from class 2. Empty dots indicate the average number of backtrackings, full dots indicate the average number of consistency checks. We see that the amount of backtrackings reduces exponentially with *l*. This suggest that even slight improvement in the implementation of this scheme (e.g. making the advice scheme dependent on different levels) would be worthwhile. The amount of consistency checks has a minimum at a relatively weak level of advice. In the first class of problem the advice based on just two nodes gave the best performance (on the average), namely a simple lookahead of just one level was the best choice. In the second class of problems with n=15 an advice based on 3 nodes was the best. In both cases the best performance of ABT was indeed better then RBT's performance. We conjecture therefore, that problems in this class will need only weak advice to enhance their performance considerably.

In conclusion, advice should be invoked with an appropriate strength. One needs therefore, a way of recognizing the difficulty of a problem instance prior to solving it. Knuth [21] has suggested a simple sampling technique to estimate the size of the search tree. These estimates can be used to tailor the advice strength to the expected size of the tree. Smaller problems should be guided by cheap and weak advice (e.g. based on partial trees) and harder problems by a stronger advice (based on full spanning trees).

Experiments related to the ones reported here were also performed by Haralick et. al [20]. Their three lookahead schemes: "full-lookahead", "partial-lookahead" and "forward-checking" can also be viewed as heuristics which are generated from simplified models. Since the task considered was finding all solutions, the heuristic information was used to prune, rather than order values. (The order of values is irrelevant in this case). The full-lookahead scheme considers the whole remaining problem (no elimination of constraints) and uses a relaxation algorithm, arc-consistency, to prune values. Partial-lookahead is limiting the relaxation procedure even further by doing only directional arc-consistency. The procedure forward-checking is the closest to our scheme. It can be viewed as a simplification of the remaining problem into a star-shaped tree, where the current variable is the center of the star. In other words, only constraints between the next variable and its successors were considered, and a directional arc-consistency was performed on this relaxed problem. The experimental results of [20] (evaluated on queen problems and random problems having complete graphs) reinforce our conclusions, namely, the weakest heuristic procedure, forward-checking, yields the best overall performance.

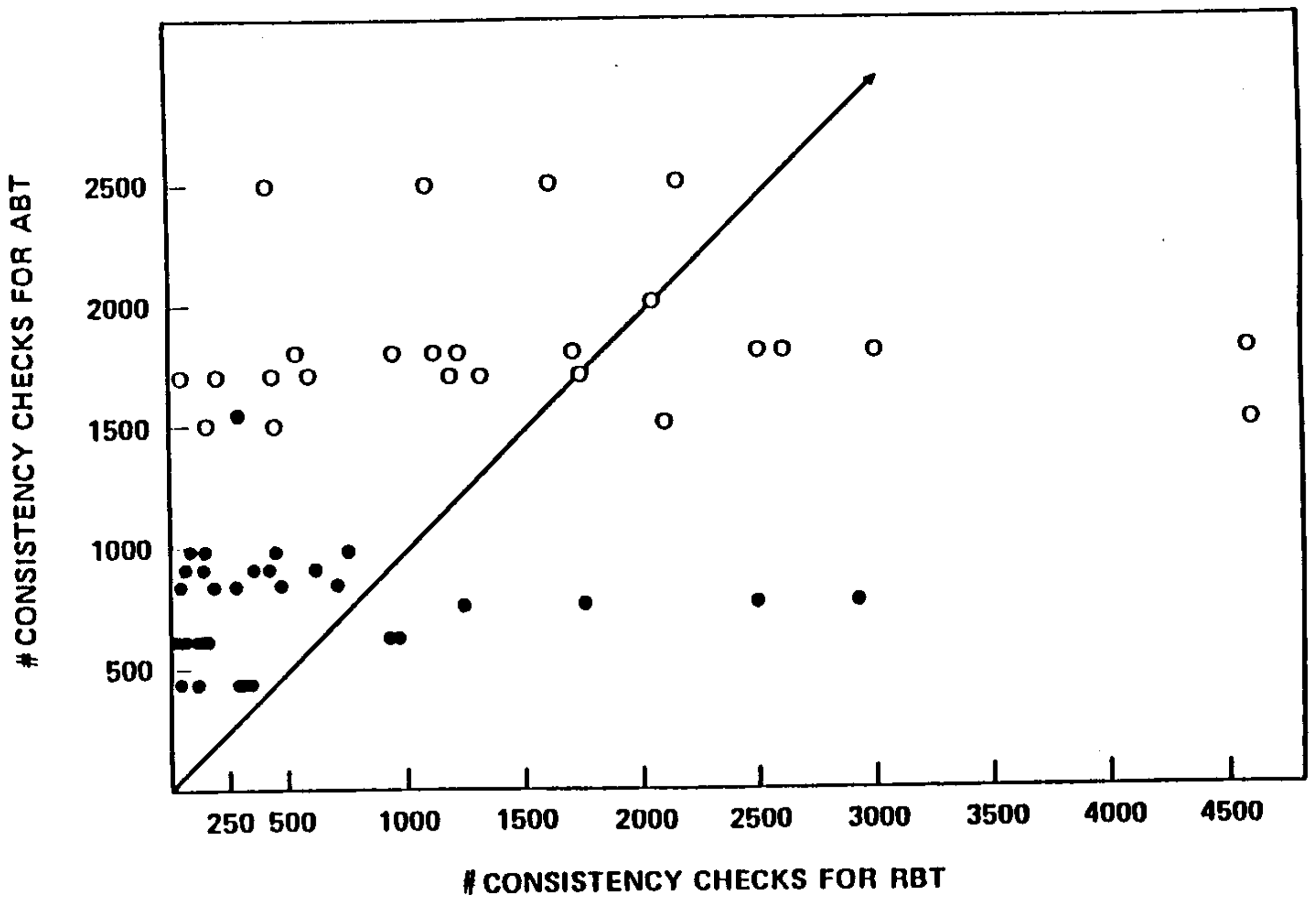


Figure 12 - Comparison of #consistency checks in ABT and RBT

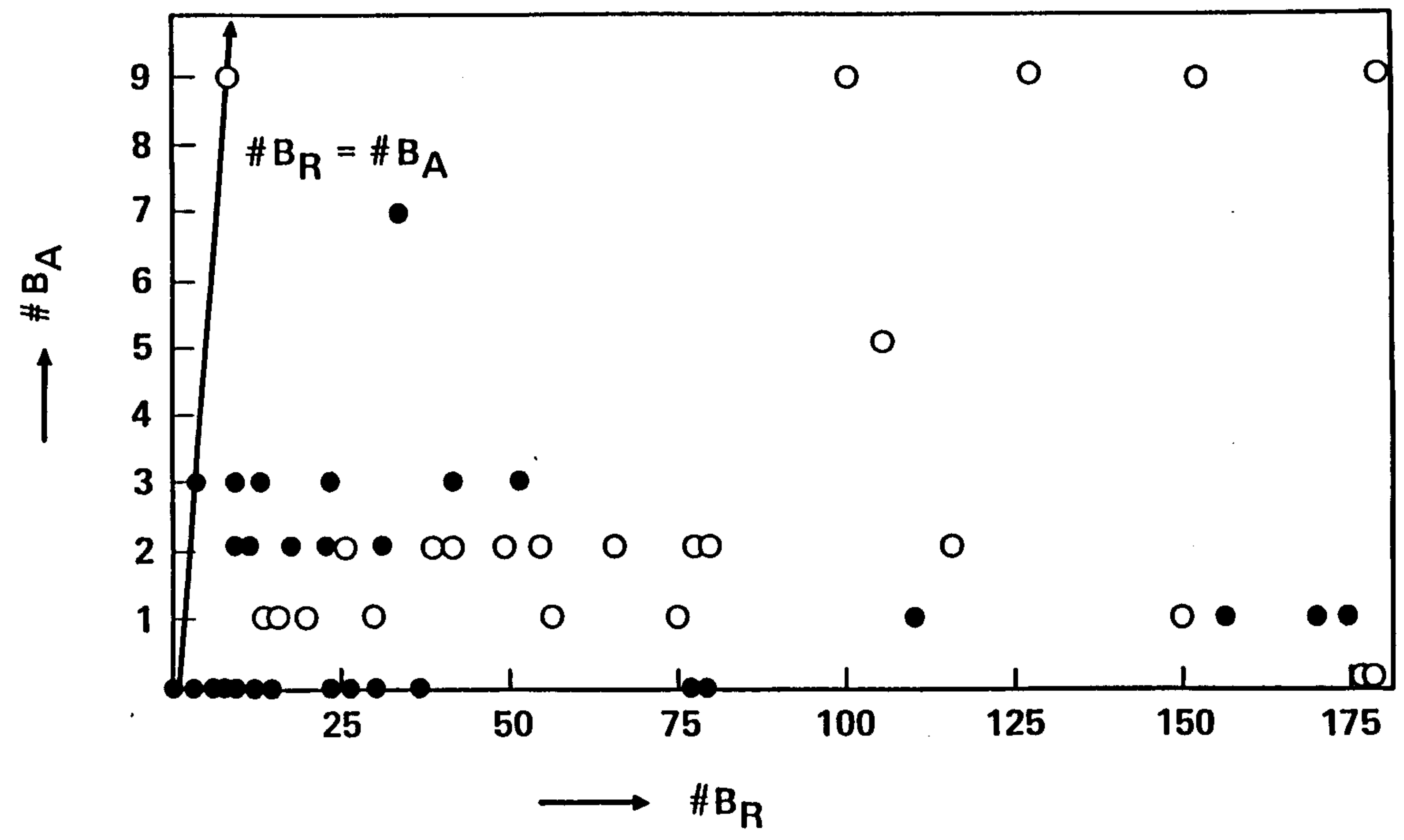


Figure 13 - Comparison of #backtrackings in ABT and RBT

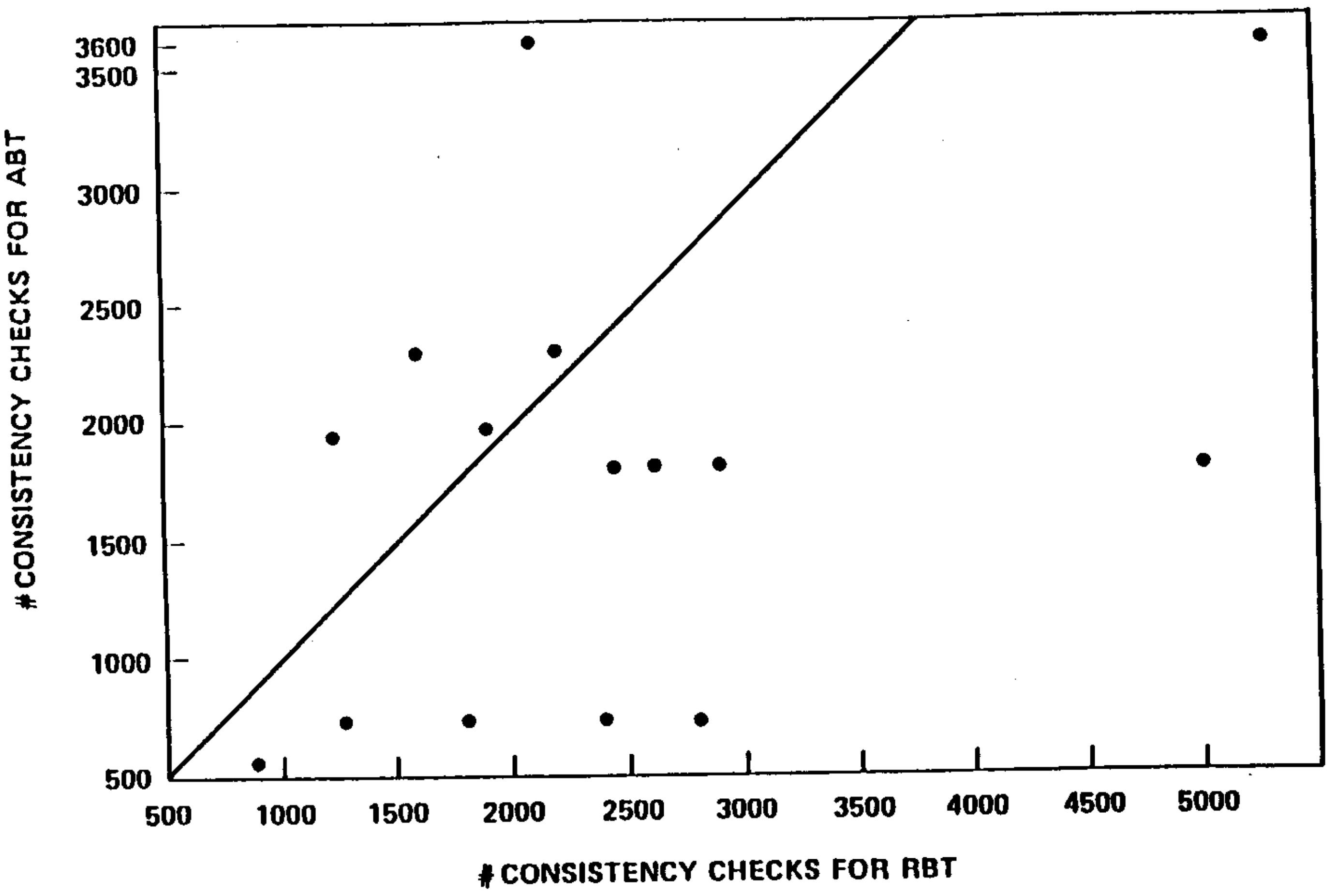


Figure 14 - Comparing #consistency checks on difficult problems

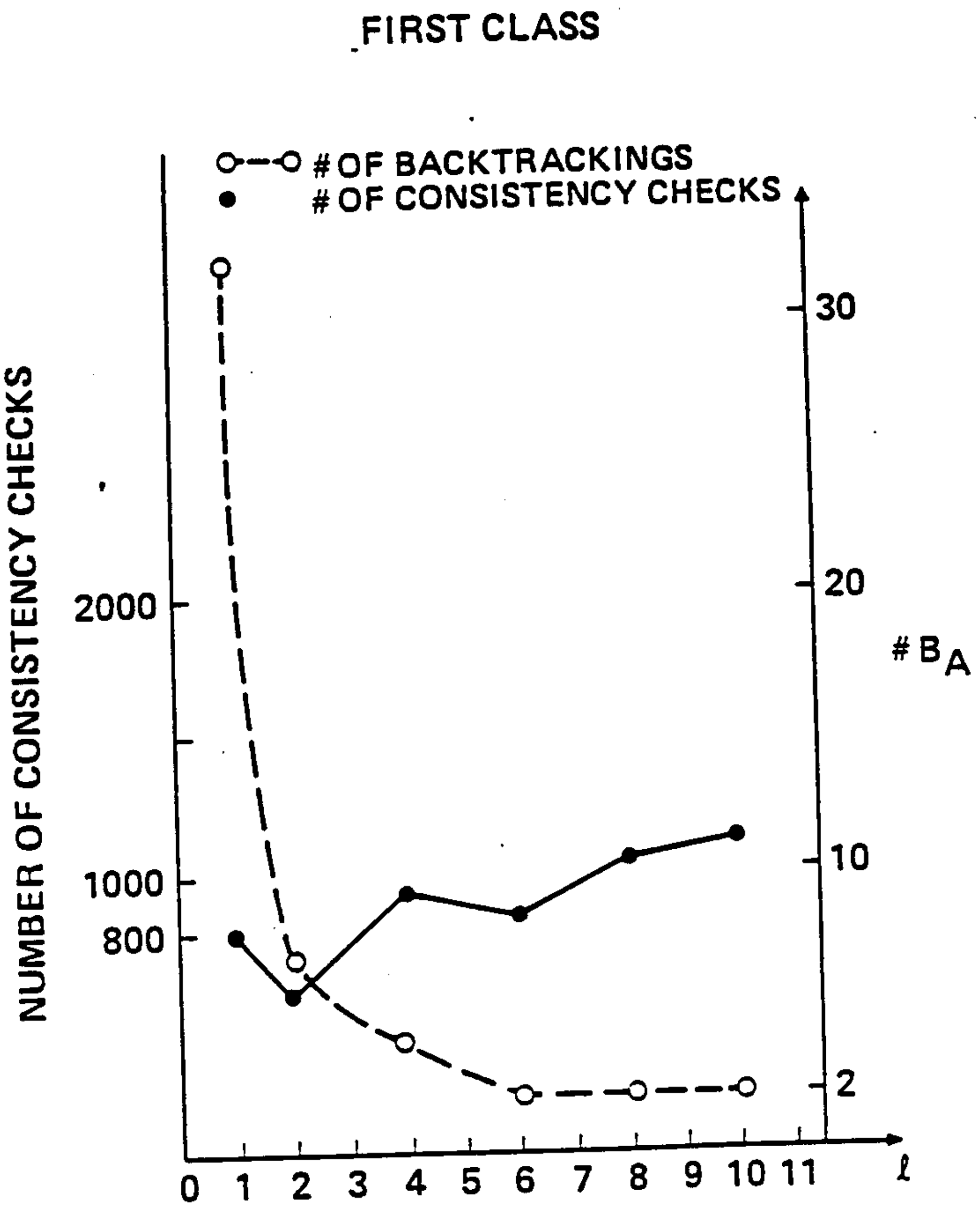


Figure 15 - #consistency checks and #backtrackings with parametrized advice (first class of problems)

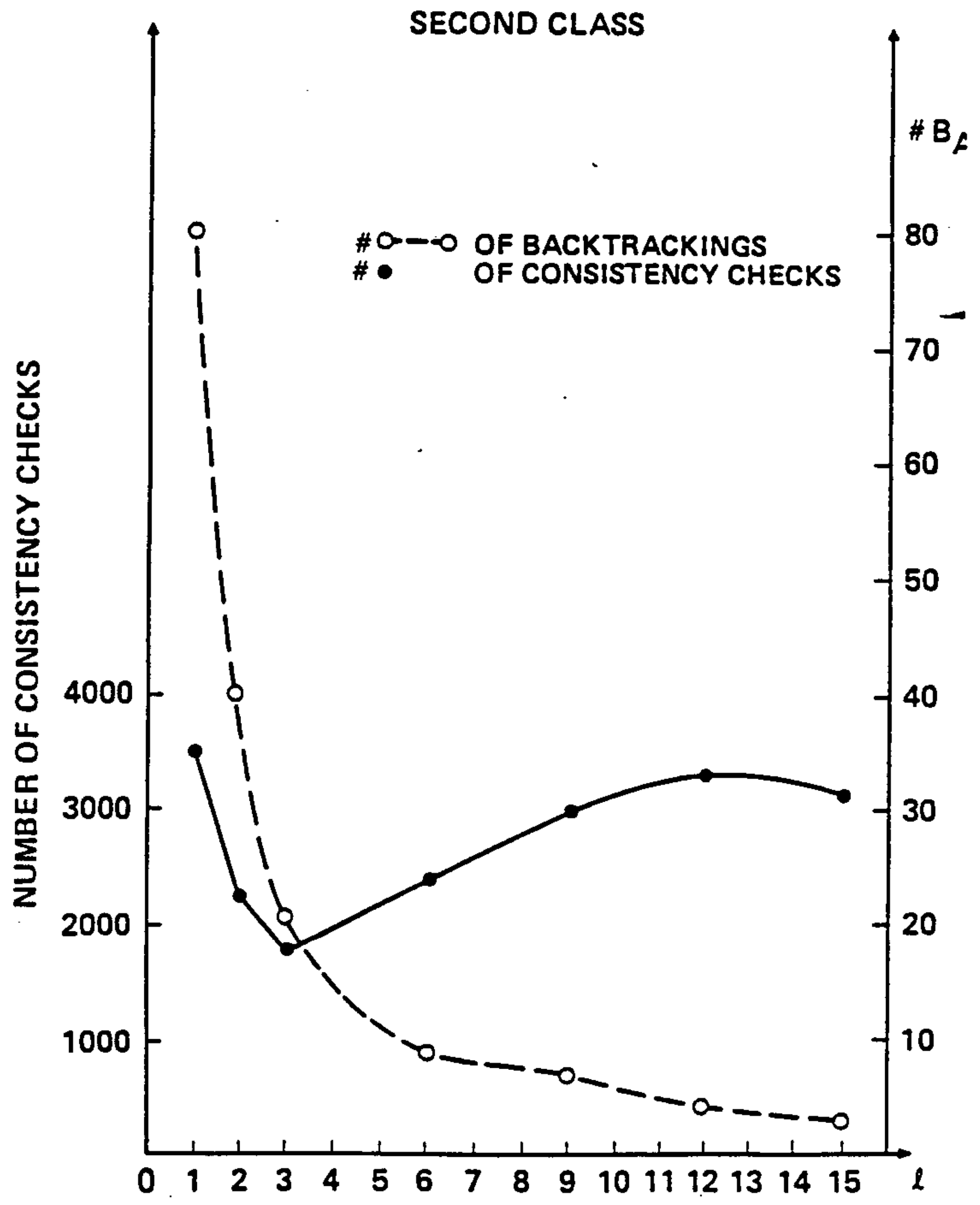


Figure 16 · #consistency checks and #backtrackings with parametrized advice (second class of problems)

4. CONCLUSIONS

The central theme of this paper is that easy problems often advertise their simplicity via salient features of their constraint networks. These features can be useful either as sources of heuristics advice for guiding universal weak methods such as backtrack, or as triggering mechanisms for invoking specialized methods, tailored to the identified structure of the problem.

The easiest class of constraint-satisfaction problems are characterized by tree-structured constraints, and these can be solved in $O(nk^{2)}$ steps using the directional-arc-consistency algorithm introduced in Section 2.2. The next recognizable class comprises regular width-2 problems, solvable in $O(n^3 k^3)$ steps using directional path consistency. The hierarchy can be generalized to regular width-K problems, which can be recognized in $O(n^{K+2})$ and solved in O(k^K) steps. An adaptive scheme is introduced in Section 2.3 where the level of K is adjusted on a node by node basis, according to conditions prevailing during the search. Other features advertising problem simplicity are nonseparable-component and cycle-cutset decompositions. A method is introduced for solving such problems in $O(nrk^m)$ steps where r is the size of the largest component and m is the size of the largest cycle-cutset (over all components).

Section 3 demonstrates a scheme of approximating general CSPs by easy, tree-structured models and introduces an efficient, $O(nk^2)$ method of extracting advice thereoff. Experiments testing the utility of the scheme on random problems revealed that advice generated by examing full spanning tree approximations is too precise, i.e., it helped backtrack avoid all but few deadends but at a rather high cost. Coarsening the advice by examining only partial trees led to improved overall performance. For the class of problems tested, the optimal advice was generated by consulting only 2-3 nodes ahead but, in harder problems, it appears feasible to adopt the level of advice to the complexity of the problem at hand, thus reaching an optimum balance between the cost of extracting advice and the benefits it provides.

APPENDIX I: Lower bound to the complexity of tree-CSP

We will show that any algorithm that solves tree-CSP with n variables and k values may require in the worst case $(n-1)k^2$ steps. We assume that the basic step of the search algorithm consists of testing the consistency of a pair of values. The algorithm terminates with the first solution found or by concluding that no solution exist. The proof uses an oracle which, for any given algorithm, creates a problem instance on which that algorithm must spend $(n-1)k^2$ steps.

Consider tree problems that have a star structure as depicted in the figure 17.

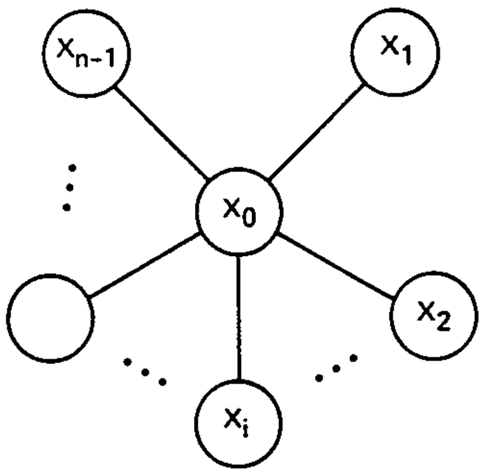


Figure 17 - A star constraint-tree

X_0 is the center variable and $X_1, \dots, X_{n-1}$ are all connected to it. Let v represent an arbitrary value of the center variable. The idea is to have the oracle answer the tests so that, for each v, it will not be possible to conclude the existence of a consistent solution without checking v against all possible values of all the peripheral variables. Since there are $(n-1)k$ such values for each value of X_0, this will prove the claim.

Oracle specification:

At any point of time, the information available to the algorithm (and the oracle) consists of three sets of pairs: positively determined, negatively determined and undetermined. In response to the query $R(X_i,x,X_0,v)$? (testing if the pair (x,v) is consistent) the oracle responds as follows:

a. If for X_i and v there are still undetermined pairs (X_i,y,X_0,v) , $y \neq x$, the answer is "NO", else

b. the response is "YES" unless

c. v in X_0 has a positively determined matching value in each of the other $n-2$ variables (excluding X_i). In which case the answer is again "NO".

This oracle creates a problem instance with no solution so that any algorithm will have to check all pairs of values. Notice that case (b) is executed w.r.t. v and a specific variable X_j only when all the pairs in X_j and v were determined. After this there is one positive match to v in X_j and all the other pairs are negatively determined. Therefore, when case (c) is executed with regard to value v of X_0 all the values which are connected to v are determined and there are exactly one matching value to v in each of the other variables. Part (c) ensures that there will not be a full solution consistent with v.

Suppose that the claim is not true, i.e., that the algorithm halts while there are still undetermined pairs. Let v be a value of X_0 which participates in one or more of these undetermined pairs. Any variable X_i whose values were all determined w.r.t v must have a positive match with v (due to case (b)). For all other variables we make the undetermined pairs positive, thus creating a problem instance that has a solution which the algorithm fails to detect. This yields a contradiction.

□

APPENDIX II: Average case analysis

Given a random CSP generated with the parameters p_1 and p_2, the probability, Q, that a pair of values is consistent is:

$$Q = 1 - p_1 + p_1 \cdot p_2 \tag{19}$$

The average number of nodes expanded when backtrack looks for one solution, denoted by $E(B_{one})$ is smaller then the average number of nodes expanded if backtrack looks for all solutions, denoted by $E(B_{all})$. i.e.

$$E(B_{one}) \leq E(B_{all}) \tag{20}$$

We will therefore, find a bound to $E(B_{one})$ by bounding $E(B_{all})$.

Given a fixed order of variables' instantiation, the probability that the j^{th} value in the sequence is consistent with all the previous $j-1$ values is Q^{j-1}. Therefore, the probability that a node at depth l in the search tree will be expanded is: (see also [20])

$$p(X \geq l) = Q^{\frac{(l-1)l}{2}} \tag{21}$$

where X is a random variable indicating the length of a consistent set of values. Since the number of possible nodes at level l is k^l, we get:

$$E(B_{all}) = \sum_{l=1}^{n} k^l Q^{\frac{(l-1)l}{2}} = \sum_{l=1}^{n} (kQ^{\frac{(l-1)}{2}})^l \tag{22}$$

Let $l_{\max}$ be the level in the search tree that has the maximum average number of nodes. Replacing each term in (22) by the highest, we get:

$$E(B_{all}) \leq n \cdot (k \cdot Q^{\frac{(l_{\max}-1)}{2}})^{l_{\max}} \tag{23}$$

$l_{\max}$ can be found by differentiating the function:

$$f(l) = k^l Q^{\frac{(l-1)l}{2}} \tag{24}$$

yielding

$$l_{\max} = -\frac{\ln k}{\ln Q} + \frac{1}{2} \tag{25}$$

If $l_{\max} \leq n$ we have:

$$E(B_{all}) = O(n \cdot (k^{c(k,Q)} \cdot Q^{(-\frac{1}{2} + c(k,Q))c(k,Q)})) \tag{26}$$

where

$$c(k,Q) = \frac{\ln k}{\ln \frac{1}{Q}} \tag{27}$$

For fixed k and Q we see that, asymptotically, the average complexity is linear in n. For a fixed Q and varying k we have:

$$E(B_{all}) = O(n \cdot k^{c(k,Q)}) = O(n \cdot k^{\frac{1}{\ln \frac{1}{Q}}} \cdot k^{\ln k}) = O(n \cdot k^{\ln k}) \tag{28}$$

No wonder, therefore, that most instances generated by our model were not too difficult. On the average these problems are linear in n and O($k^{\ln k}$) in k.

Acknowledgement

A preliminary version of this paper appears in [8] also large parts of section 2 were reported in [9]

References

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

[2] Arnborg, S., D. G. Corneil, and A. Proskurowski, ''Complexity of finding embeddings in a k-tree,'' *Siam Journal of algorithm and Discrete Math.,* Vol. 8, No. 2, 1987., pp. 277-184.

[3] Bertele, U. and F. Brioschi, *Nonserial Dynamic Programming,* New York: Academic press, 1972.

[4] Bruynooghe, Maurice and Luis M. Pereira, ''Deduction Revision by Intelligent backtracking,'' in *Implementation of Prolog,* J.A. Campbell, Ed. Ellis Harwood, 1984, pp. 194-215.

[5] Carbonell, J.G., ''Learning by analogy: Formulation and generating plan from past experience,'' in *Machine Learning,* Michalski, Carbonell and Mitchell, Ed. Palo Alto, California: Tioga Press, 1983.

[6] Cox, P.T., ''Finding backtrack points for intelligent backtracking,'' in *Implementation of Prolog,* J.A. Campbell, Ed. Ellis Harwood, 1984, pp. 216-233.

[7] Dechter, A. and R. Dechter, ''Removing redundencies in constraint networks,'' in *Proceedings AAAI-87,* Seattle, Washington: 1987.

[8] Dechter, R. and J.Pearl, ''A problem simplification approach that generates heuristics for constraint satisfaction problems.,'' UCLA-Eng-rep.8497. To appear in Machine Intelligence 11., 1985.

[9] Dechter, R. and J. Pearl, ''The anatomy of easy problems: a constraint-satisfaction formulation,'' in *Proceedings Ninth International Conference on Artificial Intelligence,* Los Angeles, Cal: 1985, pp. 1066-1072.

[10] Dechter, R., ''Learning while searching in constraint-satisfaction-problems,'' in *Proceedings AAAI-86,* Philadelphia, Pensilvenia: 1986.

[11] Dechter, R. and J. Pearl, ''A tree-clustering scheme for CSPs,'' UCLA, Cognitive-Systems Lab., Los Angeles, Cal., Tech. Rep. R-86, 1987.

[12] Dechter, R. and J. Pearl, ''The cycle-cutset method for improving search performance in AI applications,'' in *Proceeding of the 3rd IEEE on AI Applications*, Orlando, Florida: 1987.

[13] Doyle, John, ''A truth maintenance system,'' *Artificial Intelligence*, Vol. 12, 1979, pp. 231-272.

[14] Even, S., *Graph Algorithms*, Maryland, USA: Computer Science Press, 1979.

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

[16] Freuder, E.C., ''A sufficient condition for backtrack-bounded search,'' *Journal of the Association of Computing Machinery*, Vol. 32, No. 4, 1985, pp. 755-761.

[17] Gaschnig, J., ''Performance measurement and analysis of certain search algorithms.,'' Carnegie-Mellon University, Pitsburg, Pensilvenia, Tech. Rep. CMU-CS-79-124, 1979.

[18] Gaschnig, J., ''A problem similarity approach to devising heuristics: first results,'' in *Proceedings 6th international joint conf. on Artificial Intelligence.*, Tokyo, Jappan: 1979, pp. 301-307.

[19] Guida, G. and M. Somalvico, ''A method for computing heuristics in problem solving,'' *Information Sciences*, Vol. 19, 1979, pp. 251-259.

[20] Haralick, R. M. and G.L. Elliot, ''Increasing tree search efficiency for constraint satisfaction problems,'' *AI Journal*, Vol. 14, 1980, pp. 263-313.

[21] Knuth, D. E., ''Esimating the efficiency of backtrack programs,'' *Mathematics of computation*, Vol. 29, No. 129, 1975, pp. 121-136.

[22] Mackworth, A.K., ''Consistency in networks of relations,'' *Artifficial intelligence*, Vol. 8, No. 1, 1977, pp. 99-118.

[23] Mackworth, A.K. and E.C. Freuder, ''The complexity of some polynomial network consistancy algorithms for constraint satisfaction problems,'' *Artificial Intelligence* , Vol. 25, No. 1, 1984.

[24] Martins, Joao P. and Stuart C. Shapiro, ''Theoretical Foundations for belief revision,'' in *Proceedings Theoretical aspects of Reasoning about knowledge*, 1986.

[25] Matwin, Stanislaw and Tomasz Pietrzykowski, ''Intelligent backtracking in plan-based deduction,'' *IEEE Transaction on Pattern Analysis and Machine Intelligence*, Vol. PAMI-7, No. 6, 1985, pp. 682-692.

[26] Minsky, M., ''Steps towards Artificial Intelligence,'' in *Computer and Thought,* Feigenbaoum Feldman, Ed. McGraw-Hill, 1963, p. 442.

[27] Mohr, R. and T.C. Henderson, ''Arc and Path consistency revisited,'' *Artificial Intelligence,* Vol. 28, No. 2, 1986, pp. 225-233.

[28] Montanari, U., ''Networks of constraints :fundamental properties and applications to picture processing,'' *Information Science,* Vol. 7, 1974, pp. 95-132.

[29] Nudel, B., ''Consistent-Labeling problems and their algorithms: Expected complexities and theory based heuristics.,'' *Artificial Intelligence,* Vol. 21, 1983, pp. 135-178.

[30] Pearl, J., ''On the discovery and generation of certain heuristics,'' *AI Magazine,* No. 22-23, 1983.

[31] Purdom, P., ''Search rearrangement backtracking and polynomial average time,'' *AI Journal,* Vol. 21, 1983, pp. 117-133.

[32] Purdom, P.W. and C.A. Brown, *The Analysis of Algorithms*: CBS College Publishing, Holt, Rinehart and Winston, 1985.

[33] Sacerdoti, E. D., ''Planning in a hierarchy of abstraction spaces,'' *Artificial Intelligence,* Vol. 5, No. 2, 1974, pp. 115-135.

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

[35] Stone, H. S. and J. M. Stone, ''Efficient search techniques- An empirical study of the N-queens problem.,'' IBM T.J. Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC 12057 (#54343), 1986.

[36] Stone, H.S. and P. Sipala, ''The average complexity of depth-first search with backtracking and cut-off,'' *IBM JR&D,* Vol. 30, No. 3, 1986, pp. 242-258.

[37] Tarjan, Robert E. and Mihalis Yannakakis, ''Simple Linear-Time Alsorithms to test Chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs,'' *SIAM Journal of Computing,* Vol. 13, No. 3, 1984, pp. 566-579.