Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition*

Rina Dechter**

Cognitive System Laboratory, Computer Science Department, University of California, Los Angeles, CA 90024, USA

ABSTRACT

Researchers in the areas of constraint satisfaction problems, logic programming, and truth maintenance systems have suggested various schemes for enhancing the performance of the backtracking algorithm This paper defines and compares the performance of three such schemes "backjumping," "learning," and "cycle-cutset" The backjumping and the cycle-cutset methods work best when the constraint graph is sparse, while the learning scheme mostly benefits problem instances with dense constraint graphs An integrated strategy is described which utilizes the distinct advantages of each scheme Experiments show that, in hard problems, the average improvement realized by the integrated scheme is 20–25% higher than any of the individual schemes

1. Introduction

Backtracking search is a prominent processing technique in Artificial Intelligence, in particular due to its use in PROLOG [2, 4, 20, 35], truth maintenance systems (TMSs) [9, 21, 26, 29, 32], and constraint satisfaction problems (CSPs) [5, 12–14, 17, 24, 25, 28]. In recent years there has been great emphasis in all of these areas on developing methods for improving backtracking efficiency. The terms "intelligent backtracking," "selective backtracking," and "dependency-directed backtracking" denote such improvements. The main trust of these efforts is to enhance the chronological nature of the textbook backtracking algorithm with mechanisms that facilitate more informed decisions at various stages of the search.

The standard backtracking search algorithm attempts to assign values to a set

*This work is supported in part by the National Science Foundation, Grant #IRI-8501234 A substantial part of this work was performed while the author was at the Artificial-Intelligence Center at Hughes, Calabassass

** Current address Computer Science Department, Technion—Israel Institute of Technology, Technion City, Haifa 32000, Israel

0004-3702/90/$3 50 © 1990, Elsevier Science Publishers B V (North-Holland)

of variables so that all the given constraints among the variables are satisfied. The algorithm typically considers the variables in some order and, starting with the first, assigns a value, taken from a set of possible values, to each successive variable in turn as long as the assigned value is consistent with the values assigned to the preceding variables. When, in the process, a variable is encountered such that none of its possible values is 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 (i.e , assignment of values to all variables) is found or until it may be concluded that no such solution exists

Improving backtracking efficiency amounts to reducing the size of the search space expanded by the algorithm. The size of the search space is greatly dependent on the way the constraints are represented. Generally speaking, the algorithm will benefit from representations which are more *explicit*, that is, when constraints are spelled out explicitly, even if they are implied by other constraints. The size of the search space is also heavily influenced by user-specific choices such as the variable ordering and, when one solution suffices, the order in which values are assigned to each variable.

In trying to use 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 Techniques designed to be applied in advance include a variety of consistency enforcing algorithms [18, 24]. These algorithms transform a given constraint network into an equivalent, but more explicit network by using the given constraints to deduce new constraints which are then added to the network There are also several heuristics that have been proposed for the purpose of deciding the ordering of the variables prior to performing backtracking search [7, 12].

Strategies for *dynamically* improving the pruning power of backtracking can be conveniently classified as *lookahead schemes* and *lookback schemes*.

Lookahead 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 [36] and forward checking [14].

(2) Decide which variable to instantiate next (when the order is not predetermined). Generally, it is advantageous to first instantiate variables which maximally constrain the rest of the search space. Therefore, the variable participating in the highest number of constraints is usually selected [12, 28, 33, 37].

(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 [5, 14].

Lookback 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 previous variable in the ordering. This idea is often referred to as *dependency-directed backtracking* [32] or *backjumping* [13].

(2) Recording the reasons for the dead-end in the form of new constraints so that the same conflicts will not arise again later in the search (terms used to describe this idea are *constraint recording* and *nogood* constraints).

This paper reports the results of theoretical and empirical investigations of three enhancement schemes for the backtracking algorithm using the model of constraint satisfaction problems. The advantage of using the CSP model is that, unlike models based on PROLOG and TMSs, it permits an implementation-free definition of the techniques involved and simple criteria for their evaluation At the same time, the CSP model is rich enough to exhibit the basic problem areas, so that the conclusions of this study are likely to apply to other domains where backtracking is used. The choice of enhancement schemes for this study was largely motivated by insight gained through the graphical representation of CSPs, called the *constraint graphs*. The three schemes are:

(1) *Graph-based backjumping*. An implementation of the idea of going back to the source of the failure by using only knowledge extracted from the constraint graph.

(2) *Learning while searching*. An adaptation of the constraint recording idea to CSPs We view this process as that of learning, since it results in modified data structure and improved performance. We consider the effects of learning on the structure of the problem and identify several levels of learning, characterized by the degree to which the structural properties of the problems are affected, and the amount of computational resources required.

(3) *Cycle-cutset decomposition*. A new method [6] which exploits algorithms for solving tree-structured CSPs in linear time. Variable instantiation by the backtracking algorithm reduces the effective connectivity of the constraint graph, possibly until such point that the remaining constraint graph is a tree. At this point, rather than continue the search blindly, a special-purpose algorithm is invoked that completes the rest of the search in linear time.

We have simplified the presentation of the methods discussed in this paper by considering only *binary* constraint satisfaction problems, namely problems whose constraints involve no more than two variables. However, these methods are easily extendible to the general CSPs using a hypergraph generalization of the constraint graph [7].

The paper is organized as follows. Section 2 introduces the CSP model and its search space, the backtracking algorithm, and the way it is modified by the graph-based backjumping Section 3 discusses the method of learning, discusses the computational aspects of several dialects of this method, and reports the results of a computational evaluation of these dialects. The cycle-cutset decomposition is discussed in Section 4. Bounds on its performance are calculated, and the results of experimental evaluation are provided. Section 5 reports on the results of experiments where all three methods are integrated in one algorithm. A summary and conclusions are given in Section 6.

2. Constraint Satisfaction, Backtracking, and Backjumping

2.1. The CSP and its search space

A *constraint satisfaction problem* (CSP) is represented by a *constraint network*, consisting of a set of n variables, $X_1, \ldots, X_n$; their respective value domains, $R_1, \ldots, R_n$; and a set of constraints. A *constraint* $C_i(X_{i_1}, \ldots, X_{i_j})$ is a subset of the Cartesian product $R_{i_1} \times \cdots \times R_{i_j}$, consisting of all tuples of values for a subset $(X_{i_1}, \ldots, X_{i_j})$ of the variables which are compatible with each other. A *solution* of the network is an assignment of values to all the variables such that all the constraints are satisfied. Solving the CSP requires finding either one solution or all the solutions of the network. A *binary CSP* is one in which each of the constraints involves at most two variables. A binary CSP is associated with a *constraint graph* consisting of a node for each variable and an arc connecting each pair of variables which has a constraint in the network.

Consider, for example, a constraint satisfaction problem involving three variables, X, Y, and Z, whose domains are $X = \{5, 2\}$ $Y = \{2, 4\}$ and $Z = \{5, 2\}$. The problem is to assign a value for each variable, taken out of the domain of that variable, so that the value of Z divides the values of both X and Y. This restriction can be expressed in terms of two binary constraints, one between X and Z represented by the set of tuples $(XZ) = ((55)(22))$, and the other between Y and Z consisting of the set of tuples $(YZ) = ((22)(42))$. The constraint graph of this problem is shown in Fig. 1(a).

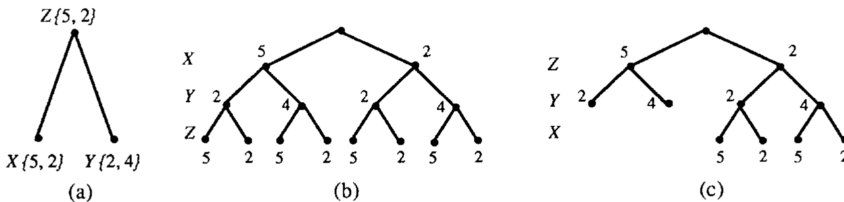


Fig 1 A CSP and its state space representation

The state space explored by the backtracking algorithm depends on the ordering of the variables. For instance, Figs. 1(b) and 1(c) display two state spaces associated with two different orderings for the CSP of Fig. 1(a). The states of this space are consistent assignments of values to a prefix of the variables in the ordering, and the operators are all consistent value assignments to the next variable in the ordering For example, given an ordering $d = (X_1, \ldots, X_n)$, a state at depth i would consist of the assignment $(X_1 = x_1, \ldots, X_{i-1} = x_{i-1})$ and the operators are the consistent extensions $X_i = x_i$ of such a state. An assignment of values to a subset of the variables is *consistent* if it satisfies all the constraints applicable to this subset. A constraint is *applicable* to a set S of variables if it is defined over any subset of S. The consistent states in depth n represent all the solutions to the problem.

2.2. Backtracking

In its standard version, the backtracking algorithm 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. 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. A backtracking algorithm 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$(x_1, \ldots, x_i)$
begin
1. if $i = n$ exit with the current assignment.
2. $C_{i+1} \leftarrow$ **Compute-candidates**$(x_1, \ldots, x_i, X_{i+1})$
3. if C_{i+1} is not empty then
4. $\quad x_{i+1} \leftarrow$ first element in C_{i+1}, and
5. $\quad$ remove x_{i+1} from C_{i+1}, and
6. $\quad$ **Forward**$(x_1, \ldots, x_i, x_{i+1})$
7. else
8. **Go-back**$(x_1, \ldots, x_i)$

end.

Go-back($x_1, \ldots, x_i$)
begin

1 if $i = 0$, exit. No soluton exists.
2. if C_i is not empty then
3. $x_i \leftarrow$ first in C_i, and
4. remove x_i from C_i, and
5 **Forward**($x_1, \ldots, x_i$)
6. else
7. **Go-back**($x_1, \ldots, x_{i-1}$)

end.

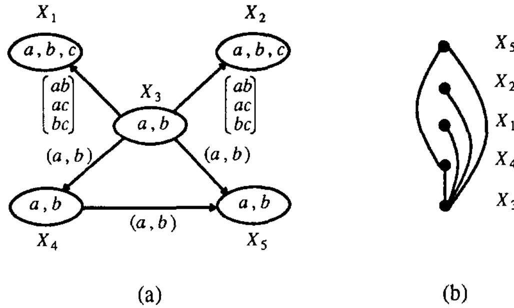
Backtracking search is initiated by calling **Forward** with $i = 0$, namely, the instantiated list is empty. The procedure **Compute-candidates**($x_1, \quad, x_i, X_{i+1}$) selects all values in the domain of X_{i+1} which are consistent with the previous assignments.

2.3. Graph-based backjumping

The idea of going back several levels in a dead-end situation, rather than going back to the chronologically most recent decision was exploited independently in [13], where the term "backjumping" is introduced, and in [32] as a part of the well known *dependency-directed backtracking*. The idea has been used since in truth maintenance systems [9], and in *intelligent backtracking* in PROLOG [2]. Gaschnig's [13] algorithm uses a marking technique that maintains, for each variable, a pointer to the highest level variable with which any value of this variable was found to be incompatible. In case of a dead-end the algorithm can safely jump directly to the variable pointed to by the dead-end variable. Although this scheme retains only one bit of information with each variable, it requires an additional (constant) computation with each consistency check.

Graph-based backjumping 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 The additional computation at each consistency check is eliminated, at the cost of less refined information about the potential cause of the dead-end.

Consider, for instance, a CSP represented by the graph in Fig. 2(a). Each node represents a variable that can take on any of the values indicated, and the constraint between connected variables is a strict lexicographic order along the arrows (the allowed pairs are specified along the arcs). If the search on this problem is performed in the order X_3, X_4, X_1, X_2, X_5 (see Fig. 2(b)), then when a dead-end occurs at X_5 the algorithm will jump back to variable X_4 since



(a) (b)

Fig 2 An example CSP

X_5 is not connected to either X_2 or X_1. If the variable to which the algorithm retreated has no more values it should back up more to the most recent variable among those connected both to the original variable and to the new dead-end variable, and so on. In short, the backjumping algorithm backs up to the most recent variable among those that are both connected to it by a path of preceding variables, and from which it can continue forward.

The implementation of graph-based backjumping requires that both the **Forward** and **Go-back** procedures must be modified. They should carry a global variable P, indicating the parent set of variables that need to be consulted upon the next dead-end. The parent set is updated only when going back. Procedure **Forward** has P as an additional parameter. Its lines 6 and 8 are changed as follows:

6. **Forward**$(x_1, \ldots, x_i, x_{i+1}, P)$
8. **Jump-back**$(x_1, \ldots, x_i, X_{i+1}, P)$

Go-back is replaced by **Jump-back** which has the partial instantiation, the dead-end variable, and the set P as parameters. Procedure **Jump-back** is given below. It uses a procedure **Parents**(X) that computes the variables connected to X that precede it in the ordering (e.g., in the ordering of Fig. 2(b), **Parents**$(X_5) = \{X_3, X_4\}$).

Jump-back$(x_1, \ldots, x_i, X_{i+1}, P)$
begin
1. if $i = 0$, exit. No solution exists.
2. PARENTS $\leftarrow$ **Parents**(X_{i+1})
3. $P \leftarrow P \cup$ PARENTS
4. Let j be the largest indexed variable in P,
5. $P \leftarrow P - X_j$
6. if $C_j \neq \emptyset$ then

7 x_j = first in C_j, and
8 remove x_j from C_j, and
9 **Forward**($x_1, \ldots, x_j, P$)
10. else
11. **Jump-back**($x_1, \ldots, x_{j-1}, X_j, P$)
end.

In the next section we describe the learning scheme which, although it is not based on structural properties of the CSP in the way graph-based backjumping does, its implementation can exploit graph-based information.

3. Learning While Searching

3.1. Introduction

By the term learning we mean the task of usefully recording information which becomes known to the problem solver while solving the problem. This information can be used later either in the same problem instance or in solving other instances of the same problem. One of the first applications of this idea involved the creation of *macro-operators*. These are sequences and subsequences of atomic operators that have proven useful as solutions to earlier problem instances from the domain. This idea was exploited in STRIPS with MACROPS (Fikes and Nilsson [11], and by Korf [15]). Other examples of learning in problem solving are the work on analogical problem solving (Carbonell [3]), learning heuristic problem solving strategies through experience as described in the program LEX (Mitchell et al. [23]), and developing a general problem solver (SOAR) that learns about aspects of its behavior using chunking (Laird et al [16]). Recently, a set of methods, commonly referred to as *explanation-based learning* (EBL), has captured the attention of the machine learning community. These methods identify, during search or theorem proving process, the most general description of a set of conditions (a concept) that can be learned from one example [22].

In the context of CSPs, the process of making *implicit* constraints into *explicit* constraints can be viewed as a form of explanation-based learning. This process can be performed independently of backtracking search (e.g., consistency algorithms), but its potential as a means of improving search efficiency may be even greater when it is incorporated into the backtracking algorithm itself

3.2. Controlled learning

An opportunity to learn new constraints is presented whenever the algorithm encounters a *dead-end*, i.e., when the current state $S = (X_1 = x_1, \ldots, X_{i-1} =$

x_{i-1}) cannot be extended by *any* value of the next variable X_i. In such a case we say that S is *in conflict* with X_i or, in short, that S is a *conflict set*. Had the problem included an explicit constraint prohibiting the set 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 this state will not reoccur. (Recording this constraint may still be useful for answering future queries on the same initial set of constraints.) If, on the other hand, the set S contains one or more subsets which are also in conflict with X_i, then recording this information in the form of new explicit constraints might prove useful in the continued exploration of the search space because future states may contain these subsets.[1]

When a conflict set has at least one subset which is also a conflict set, then recording the smaller conflict set as a constraint is sufficient for preventing the larger one from occurring. The target of this learning scheme is to identify subsets of the original conflict set S which are as small as possible because small conflict sets will occur earlier in the search than larger sets. A *minimal conflict set* [1] is one which does not have any subset which is a conflict set. Minimal conflict sets can be thought of as the sets of instantiations that "caused" the conflict.

A first step in discovering a subset of S which is in conflict with X_i consists of removing from S variable-value pairs which are *irrelevant* to X_i. A pair consisting of a variable and its value, (X, x), is said to be irrelevant to X_i if it is consistent with *all* values of X_i. Irrelevant pairs can be safely removed from S because they cannot belong to any minimal conflict set.

The process of recording constraints generated by the removal of irrelevant pairs from S will be called *shallow learning*. We say that *full shallow learning* is performed when *all* irrelevant pairs are removed, and denote by conf(S, X_i), or, in short, *conf-set*, the resulting conflict set. The constraint graph provides an easy way for identifying irrelevant pairs, namely those whose variables are not connected to the dead-end variable. Using the constraint graph as the sole source for detecting irrelevant pairs is called *graph-based shallow learning*. Graph-based shallow learning may fail to detect all irrelevant variable pairs, because a variable connected with the dead-end variable may still be irrelevant. Consequently, the resulting conflict set, called *graph-based conf-set*, contains the conf-set.

Consider the CSP presented in Fig 2(a). Suppose that a backtracking algorithm, using the ordering (X_1, X_2, X_3, X_4, X_5) is currently at state ($X_1 = b$, $X_2 = b$, $X_3 = a$, $X_4 = b$). This state cannot be extended by any value of X_5

[1] The type of learning discussed here can be viewed as *explanation-based learning*, in which learning can be done by recording an explanation or a proof to some concept of interest. Here the target concept is a dead-end situation, its proof is the conflict set and we record a summary of this proof which could be useful later [31]

since none of its values is consistent with all the previous instantiations Obviously, the tuple $(X_1 = b, X_2 = b, X_3 = a, X_4 = b)$ should not have been allowed in this problem, but as we pointed out above, there is no point in recording this fact as a constraint among the four variables involved. A closer look reveals that the instantiations $X_1 = b$ and $X_2 = b$ are both irrelevant to this conflict because there is no explicit constraint between X_1 and X_5 or between X_2 and X_5. Neither $X_3 = a$ nor $X_4 = b$ can be shown to be irrelevant and, therefore, the conf-set is $(X_3 = a, X_4 = b)$, which could be recorded by eliminating the pair (a, b) from the set of pairs permitted by constraint $C(X_3, X_4)$. The conf-set is not minimal, however, since the instantiation $X_4 = b$ is, by itself, in conflict with X_5 Therefore, it would be sufficient to record only this information, by eliminating the value b from the domain of X_4.

Identifying and recording only conflict sets which are known to be minimal constitutes *deep learning* Discovering *all* minimal conflict sets amounts to acquiring all the possible information out of a dead-end and is called *full deep learning*. However, instead of improving the overall work in searching, full deep learning may result in a substantial addition of time and space overhead which may outweight its benefits

If r is the cardinality of the conf-set, we can envision a worst case where all subsets of conf(S, X_i) having $\frac{1}{2}r$ elements are in conflict with X_i The number of minimal conflict sets should then satisfy

$$\#\text{min-conflict-sets} = \binom{r}{\frac{1}{2}r} \simeq 2^r ,$$

which amounts to exponential time and space complexity at each dead-end Even if performed efficiently, learning from *each* dead-end is too space-expensive, since it is equivalent to recording almost the entire search space explored

In addition, it is not clear that adding constraints necessarily reduces the work of the backtracking algorithm. The presence of more constraints, while potentially pruning the search space, increases the cost of generating states in the search space because more constraints need to be tested with each new instantiation. Thus backtracking may be slowed down considerably.

For these reasons, it is imperative to focus the learning process on those constraints having good pruning capability. This can be accomplished by recording only constraints with a small number of variables, thus limiting both the *number* of constraints recorded, their *size*, and the time needed for their detection. In addition, the pruning power of lower-arity constraints is higher than the pruning power of constraints involving more variables, as they stand a higher chance of reoccurring.[2] We may decide, for example, to record only

[2] In light of this discussion, it is somewhat surprising that most researchers in the area of truth maintenance systems have adopted the approach to *learn from each dead-end* (recording nogood sets, e g Doyle [9], de Kleer [8], Martins and Shapiro [19]) Indeed some of these systems suffer from space problems

conflict sets consisting of a single variable. This can be implemented by eliminating the value from the domain of its variable, and is referred to as *first-order learning*.[3]

Second-order learning is performed by recording only conflict sets involving either one or two variables.[4] In a similar vein we can define and execute an ith-order learning algorithm, recording every constraint involving i or less variables. Obviously, as i increases time and space complexities increase. Moreover, since not all subsets of variables are directly constrained in the initial representation, learning of second order and higher may also change the topology of the constraint graph. This can be avoided, however, by further restricting the algorithm to *modify* only existing constraints without creating new ones. This restriction leaves the structure of the constraint graph unchanged; a desirable property, particularly in the presence of graph-based techniques which benefit from the graphs' sparseness (see Section 4).

Controlling the size of the constraints recorded is applicable to both shallow and deep learning. For example, shallow second-order learning means recording the conf-set as a constraint only if it has no more than two variables. Deep second-order learning means identifying and recording as constraints all minimal conflict sets of either one or two variables.

3.3. Computational aspects of controlled learning

The complexity of graph-based shallow learning is just $O(l)$, where l is the number of variables in the initial conflict set, S, that are connected to the dead-end variable. Full shallow learning (i.e., identifying all irrelevant variables) requires the examination of each of the l connected variables and testing whether each of its values is consistent with each of the values of the dead-end variable. Therefore, the complexity is $O(kl)$, where k is the number of values for X_i. The computation of the conf-set can be enhanced by remembering some information already explicated during the **Forward** phase of the backtrack procedure. For instance, a marking procedure similar to the one suggested by Gaschnig [13] can be worked out with only a small amount of additional cost.

A detailed trace of the performance of shallow second-order learning on the example of Fig. 2 is given in Appendix C. Figure 3 gives the search space explicated by naive backtracking vis à vis that generated by incorporating

[3] First-order learning amounts to making a subset of the arcs *arc-consistent*. An arc (X, Y) is arc-consistent if for any value x in the domain of X, there is a value y in the domain of Y s.t. $(x, y) \in C(X, Y)$ [17]. In first-order learning only those arcs that are encountered during the search are made arc-consistent.

[4] Second-order learning performs partial *path consistency* [24] since it only adds and modifies constraints from paths discovered during the search. A pair (X, Y) is path-consistent w.r.t. Z if for any pair of values $(x, y) \in C(X, Y)$ there is a value z of Z s.t. $(x, z) \in C(X, Z)$ and $(y, z) \in C(Y, Z)$.

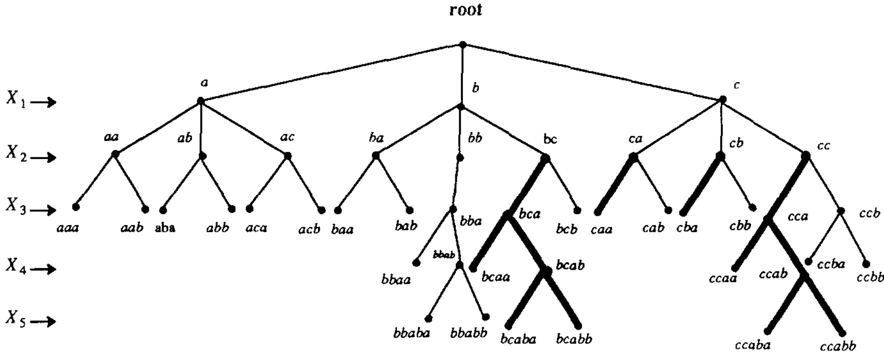


Fig 3 Explicated search space with naive backtracking with and without shallow learning

shallow second-order learning; all marked branches in Fig. 3 are not generated by the latter.

Restricting the level of shallow learning does not affect its time complexity but does affect the storage required. For example, first-order learning does not require any additional space beyond that of naive backtracking. Second-order learning may increase the size of the problem and changes its constraint graph. Since there are at most $\frac{1}{2}n(n-1)$ binary constraints, each having at most k^2 pairs of values, the increase in storage is still reasonably bounded and may be compensated by savings gained in search. Of course, when the algorithm is restricted to modifying existing constraints, the constraint graph does not change.

Full deep learning, i.e , discovering all minimal conflict subsets of the conf-set, can be implemented enumeratively: it first recognizes all conflict sets of one element, then all those with two elements, and in general, given that all minimal conflict sets of size 1, . . , $i-1$ are recognized, it finds all the size-i conflict sets which do not contain any smaller conflict sets. The time and space complexity of this mode of learning is exponential in the size of the conf-set.

Restricting the level of deep learning affects both space and time The effect on space is similar to the case of shallow learning, as discussed above The time complexity of *deep first-order learning* is $O(kr)$, where r is the size of the conf-set, since each of the r value-assignments in the conf-set is tested against all values of X_i. For *deep second-order learning* the complexity can rise to $O(\frac{1}{2}r(r-1)k)$ since each pair of instantiations should be checked against each value of X_i. We will replace r with an upper bound w, defined as follows. Given an ordered constraint graph, the *width of a node* is the number of its adjacent predecessors (parents) in the ordering. The *width of the ordering*, w, is the maximum width of all nodes in that ordering. Since the size of the conf-set cannot exceed w (i.e., any dead-end variable is connected to at most w variables), w can be used to bound r (i.e., $r \leq w$). The width, w, also bounds

Table 1

	Shallow learning			Deep learning		
Complexities	1st-order	2nd-order	all	1st-order	2nd-order	all
time/dead-end	$O(w)$	$O(w)$	$O(w)$	$O(wk)$	$O(kw^2)$	$O(kw2^w)$
space/dead-end	$O(1)$	$O(1)$	$O(w)$	$O(1)$	$O(w^2)$	$O(w2^w)$
#added-constraints	$O(1)$	$O(n^2)$	$O(2^n)$	$O(1)$	$O(n^2)$	$O(2^n)$
Pruning factor	$\left(1-\frac{1}{k}\right)$	$\left(1-\frac{1}{k^2}\right)$	$\left(1-\frac{1}{k^w}\right)$	$\left(1-\frac{1}{k}\right)$	$\left(1-\frac{1}{k^2}\right)$	$\left(1-\frac{1}{k^w}\right)$

the number of constraints needed to be checked when determining a smaller conflict subset of the conf-set (i.e., $l \leq w$). (For an additional analysis of the effects of recording nogoods, see [27].)

Table 1 summarizes the time and space bounds for controlled learning in each dead-end situation (lines 1 and 2). The expressions for shallow learning are based on computing the graph-based conf-set. Line 3 of the table gives bounds on the total number of contraints that may be recorded during the whole search, and line 4 gives a rough indication on the pruning power of different sizes of conflict sets. The expressions in line 4 describe the utmost pruning potential, assuming that the overall search space is $O(k^n)$ and that the conflict set recorded is the only constraint used for pruning the search space. For instance, eliminating a tuple of size i (i.e., recording an i-ary constraint) results in a worse-case number of solutions of $k^{n-i}(k^i - 1) = k^n - k^{n-i}$. Thus, the search space is reduced by a factor of $(1 - 1/k^i)$

3.4. Experimental evaluation

3.4.1. *Classes of problems*

The learning scheme, implemented with a graph-based backjumping algorithm, was tested on several classes of problems with varying degrees of difficulty. The first is the *class-assignment problem*, a database type problem adapted[5] from [2]. The problem statement and the associated constraint graph are given in Appendix A. The second is a more difficult problem known as the *zebra problem*. The problem can be represented as a binary CSP by defining 25 variables each with 5 values (a detailed statement of the problem and its formulation as a binary CSP is given in Appendix B). Several instances of each problem have been generated by randomly varying the order of variables. As explained in Section 2, each ordering results in a different search space for the problem and, therefore, is considered as a different instance.

[5] Our problem is an approximation of the original problem where only binary constraints are used

Two other classes of problems, random CSPs and random planar CSPs were tested. The random CSPs were generated using two parameters p_1 and p_2. The first is the probability that any pair of variables is directly constrained, and the second is the probability that pairs of values belonging to constrained pairs of variables are compatible. Planar problems are CSPs whose constraint graph is planar. Problems that arise in vision have this property. These problems were generated from an initial maximally connected planar constraint graph with 16 variables, as shown in Fig. 4. The parameter p_1 in this case determines the probability that an arc will be deleted from this graph, while p_2 controls the generation of each constraint as in the case of random CSPs. We tested the algorithms on random CSPs with 10 and 15 variables, having 5 or 9 values. The planar problems were tested with 16 variables and 9 values. The planar and the random problems were solved in a fixed order, nonincreasing with the variable's degree This is a reasonable heuristic since it estimates the notion of *width* of the graph as described by Freuder [12]

3 4 2. *The algorithms*

The modes of learning used in the experiments were controlled by three parameters: the *depth* of learning (i.e., shallow or deep), the *level* of learning (i.e , first-order or second-order), and either *adding* constraints or only *modifying* existing ones. This results in six types of learning: shallow first-order (SF), shallow second-order modify (SSM), shallow second-order add (SSA), deep first-order (DF), deep second-order modify (DSM), and deep second-order add (DSA). For shallow learning we computed the graph-based conf-set All learning modes were implemented on top of a graph-based backjumping algorithm Thus, each problem instance was solved by eight search strategies: naive backtracking, graph-based backjumping (no learning), and graph-based backjumping enhanced with each of the six possible modes of learning.

3.4 3. *Performance measures*

Two measures were recorded· the *number of backtrackings* during search and the *number of consistency checks* performed A consistency check is performed

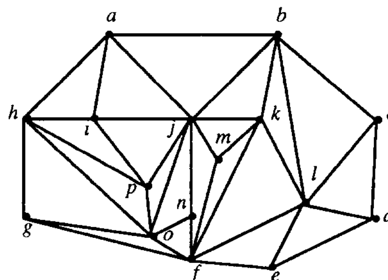


Fig 4 A 16-node, fully triangular planar graph

each time the algorithm checks if two values of a variable are consistent w.r.t. the direct constraint between them. (Notice that had we implemented our algorithms on constraints with more than two variables, the time for testing the consistency of a tuple could not have been regarded as constant.) The number of backtrackings measures the size of the search space which was explicated (every dead-end is a leaf in this space), while the amount of consistency checks measures the overall work invested in generating this search space.

3.4.4. *Results*

Tables 2 and 3 present the results for six problem instances of the class-assignment problem, and for six problem instances of the zebra problem, respectively. For these problems we omitted the distinction between adding or modifying constraints since there was only negligible difference between these

Table 2
The class-assignment problem

#	NB	BJ	SF	SS	DF	DS
1	(25, 219)	(25, 219)	(25, 218)	(25, 221)	(25, 218)	(22, 194)
			(0, 44)	(0, 44)	(0, 44)	(0, 44)
2	(12, 123)	(12, 123)	(12, 123)	(12, 133)	(12, 137)	(12, 137)
			(0, 43)	(0, 43)	(0, 42)	(0, 42)
3	(24, 266)	(24, 266)	(24, 266)	(24, 267)	(20, 260)	(7, 125)
			(10, 140)	(10, 140)	(10, 140)	(0, 51)
4	(42, 407)	(42, 407)	(42, 406)	(42, 409)	(40, 423)	(39, 509)
			(8, 108)	(8, 108)	(4, 91)	(0, 50)
5	(42, 433)	(42, 433)	(42, 433)	(42, 435)	(40, 445)	(39, 527)
			(8, 116)	(8, 116)	(4, 91)	(0, 50)
6	(85, 559)	(85, 559)	(85, 559)	(53, 391)	(85, 692)	(57,619)
			(68, 441)	(2, 55)	(54, 461)	(0, 49)

Table 3
The zebra problem

#	NB	BJ	SF	SS	DF	DS
1	(180, 2066)	(57, 1241)	(56, 1234)	(56, 1234)	(56, 1272)	(37, 884)
			(53, 1214)	(53, 1214)	(53, 1252)	(11, 322)
2	(5378, 37541)	(1315, 20542)	(1279, 19498)	(12799, 19498)	(1279, 21371)	(302, 4523)
			(1276, 19416)	(1276, 19416)	(1276, 19945)	(97, 1584)
3	(31097, 241204[a])	(2848, 46771)	(2723, 44719)	(2721,44716)	(2512, 41911)	(530, 10670)
			(2719, 44693)	(2719, 44693)	(2300, 38421)	(113, 1861)
4	(23778, 237152)	(738, 21946)	(738, 21946)	(655, 21205)	(738, 22117)	(190, 5521)
			(738, 21946)	(301, 10591)	(738, 22117)	(16, 572)
5	(5378, 37541)	(4159, 30100)	(4158, 30091)	(1621, 11777)	(4158, 30091)	(414, 4604)
			(4155, 30009)	(1566, 11180)	(4155, 30009)	(117, 1579)
6	(105215, 11533666[a])	(6362, 91026)	(6362, 91026)	(6362, 93610)	(6362, 93610)	(961, 17367)
			(1975, 33071)	(1975, 33126)	(1975, 33126)	(626, 9535)

[a] Count recorded at the time the run was stopped

two types of learnmg. Each entry records the number of backtrackmgs and the number of consistency checks respectively.

Each of the problem instances was solved twice by the same strategy; the second run using a new representation that included all the constraints recorded in the first run. This was done in order to check the effectiveness of these strategies in finding a better problem representation. The results of these runs are shown in a second pair of numbers in the corresponding entries in Tables 2 and 3 Figures 5 and 6 provide a graphic display of these results depicting only the number of consistency checks performed for the two problems.

The third and fourth classes of problems are similar in their behavior for the various learning algorithms. Figures 7 and 8 present these results for the random and planar problems respectively, after grouping similar instances into clusters and averaging on each cluster.

Our experiments (implemented in LISP on a Symbolics LISP Machine) show that the behavior of the algorithms is different for different problems. From these results we see that the class-assignment problem turned out to be very easy, and is solved efficiently even by naive backtracking (see Fig. 5 and Table 2). The effects of backjumping and learning are, therefore, minimal, except for

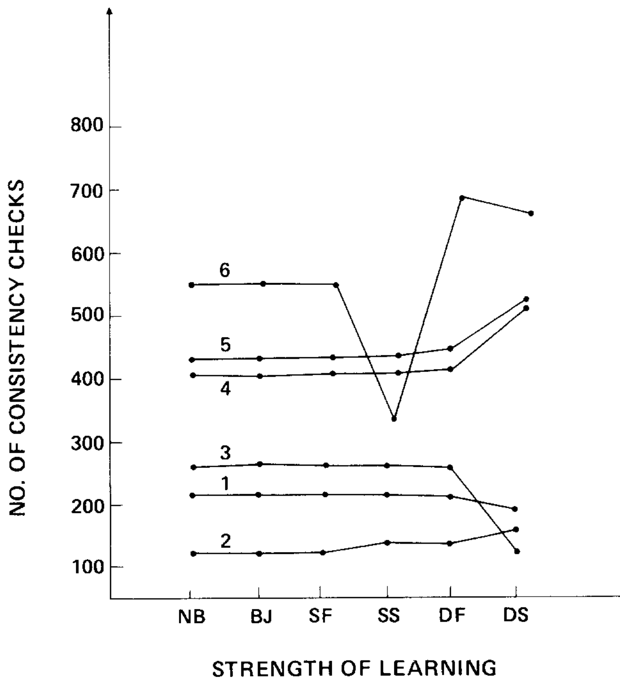


Fig 5 Performance of learning schemes on the class-assignment problem

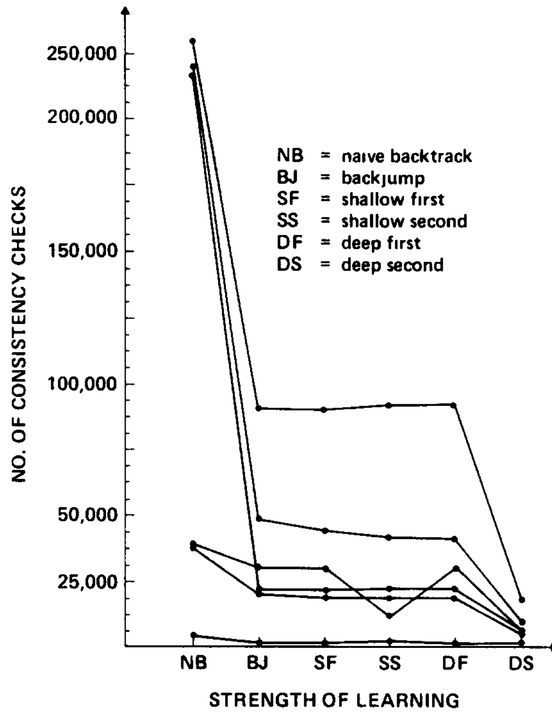


Fig 6 Performance of learning schemes on the zebra problem

deep second-order learning where some gains are evident. In all these cases, the second run gave a backtrack-free performance.

The zebra problem, on the other hand, is apparently much more difficult, and in some cases could not be solved by naive backtracking in a reasonable amount of time (in these cases the numbers reported in Table 2 are the counts recorded at time the run was stopped). The enhanced backtrack schemes show dramatic improvements in two stages (see Fig. 6). First, the introduction of backjumping by itself improved the performance substantially, with only moderate additional improvements due to the introduction of first-order or shallow second-order learning. Deep second-order learning caused a second leap in performance, with gains over no learning backjump by a factor of 5 to 10.

For the planar and random problems which appear to be of moderate difficulty the behavior pattern is different (see Figs. 7 and 8). In almost all cases we see a big improvement in the performance going from naive backtracking to backjumping. Additional, more moderate improvement continues for shallow first- and shallow second-order learning when constraints are modified only, then the performance deteriorates when deeper forms of

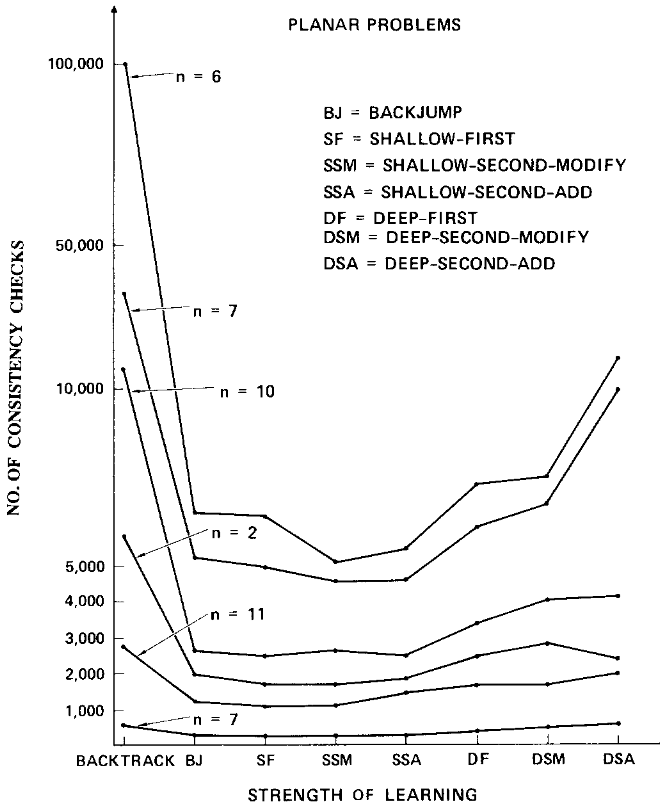


Fig 7 Performance of learning schemes on the planar problems

learning are used. The amount of work invested in these deeper learning schemes outweighs the savings in the search.

The experiments demonstrate that learning might be beneficial in solving CSPs. For difficult and moderately difficult problems (e.g. the zebra problem, planar and random problems) backjumping, coupled with shallow (first- or second-order) learning, largely outperformed naive backtracking. In contrast, for easy problems (e.g. the class-assignment problem) their performance is roughly similar. These results strongly support the work on intelligent backtracking in PROLOG which is centered on various backjumping schemes. In recent work Rosiers and Bruynooghe [30] also demonstrated that these schemes compare favorably with some known lookahead schemes.

Deeper forms of learning perform well on difficult problems, like the zebra problem, on which the greatest improvement was achieved by the strongest form of learning tested: deep second-order learning. It still remains to be seen

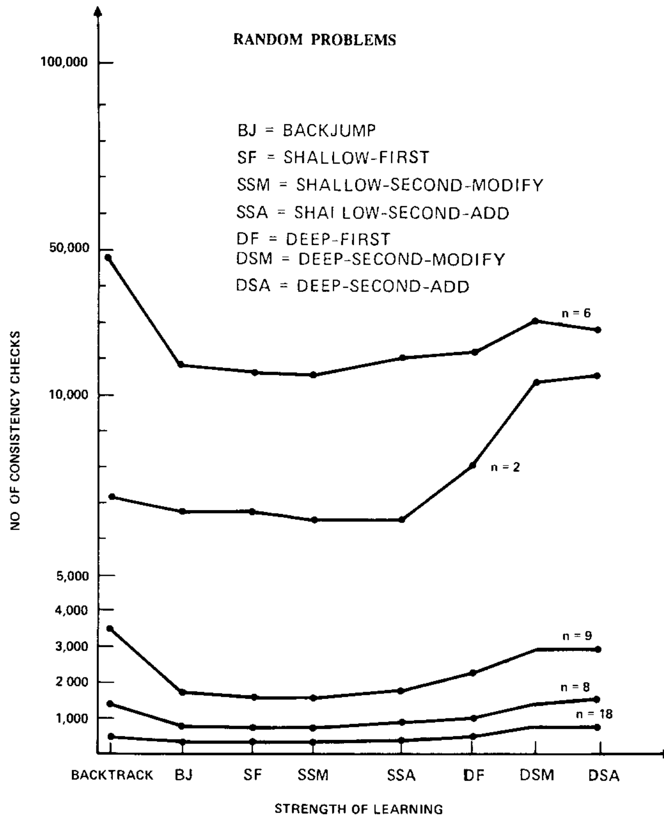


Fig 8 Performance of learning schemes on the random problems

whether higher degrees of learning are worth pursuing in view of the additional bookkeeping storage space they require. For easier problems, however, (e.g. planar problems, random problems and the class-assignment problem), their performance was generally worse in comparison to backjumping.

We showed that with the use of learning the "knowledgeable" representation achieved upon termination is significantly better than the original one, when answering exactly the same query. This could be used when a CSP is representing a knowledge base on which many different queries can be posed. Each query assumes a world that satisfies all the domain constraints plus some additional *query constraints*. In such an environment it may be worthwhile to keep the knowledge in the form of a set of constraints enriched by those learned during past searches. It is in such environments that the benefits of learning across instances could be best demonstrated. However, such a study is beyond the scope of this paper.

Additional experimental results are required for establishing how first- and second-order learning compare with the pre-processing approaches of arc and path consistency. However, theoretical considerations by themselves reveal that pre-processing may be too costly and may perform unnecessary work. For instance, the path consistency algorithm is known to have a lower-bound worse-case complexity of $O(n^3k^3)$, and the best performance is $O(n^3k^2)$ [18][6] For the zebra problem these consistency checks range between 388625 and 1953125, which is far greater than those encountered in deep second-order learning on all problem instances presented. Finally, we conjecture that the effect of learning would be higher if all solutions were generated.

In the next section we present another graph-based technique, called the *cycle-cutset scheme*, and subsequently demonstrate its effect on backjumping and learning.

4. The Cycle-Cutset Decomposition

4.1. The cycle-cutset method

Whereas learning methods attempt to prune the search space by explicating deducible constraints, decomposition methods keep the problem representation intact and attempt, instead, to exploit special features of the constraint graph which admit simple solutions. The decomposition method investigated in this section utilizes the simplicity of tree-structured problems We call it the *cycle-cutset method* since it is based on identifying a set of nodes that, once removed, would render the constraint graph cycle-free.

The cycle-cutset method is based on two facts: one is that tree-structured CSPs can be solved very efficiently [5, 12, 18], and the other is that variable instantiation changes the effective connectivity of the constraint graph In Fig. 2(a), for example, instantiating X_3 to some value, say a, renders the choices of X_1 and X_2 independent of each other as if the pathway X_1—X_3—X_2 was "blocked" at X_3 Similarly, this instantiation "blocks" the pathways X_1—X_3—

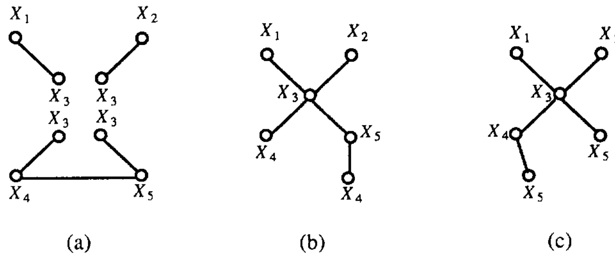


Fig 9 An instantiated variable cuts its own cycles

[6] Recent experimental results are also available in R Dechter and I Meiri, Experimental evaluation of preprocessing techniques in constraint satisfaction problems, in *Proceedings IJCAI-89*, Detroit, MI (1989)

X_5, X_2—X_3—X_4, X_4—X_3—X_5 and others, leaving only one path between any two variables. The constraint graph for the rest of the variables is shown in Fig. 9(a), where the instantiated variable, X_3, is duplicated for each of its neighbors. The method of *cutting loops* in the "CONSTRAINTS" language [34] is a variant of the same idea.

When the group of instantiated variables constitute a cycle-cutset, the remaining network is cycle-free, and the efficient algorithm for solving tree-constrained problems is applicable. In the example above, X_3, cuts the single cycle X_3—X_4—X_5 in the graph, and the graph in Fig. 9(a) is cycle-free. Of course, the same effect would be achieved by instantiating either X_4 or X_5, resulting in the constraint trees shown in Figs. 9(b) and 9(c). In most practical cases it would take more than a single variable to cut all the cycles in the graph (see Fig. 10).

Therefore, a general way of solving a problem whose constraint graph contains cycles is to instantiate the variables in a cycle-cutset in a consistent way and solve the remaining tree-structured problem by a 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 we wish to solve the problem in Fig. 2, we first assume $X_3 = a$ and solve the remaining problem. If no solution is found, then assume $X_3 = b$ and try again.

Since the complexity of this scheme is dominated by the exponential complexity of instantiating the cutset variables, and since finding a minimal size cycle-cutset is an NP-complete problem, it will be more practical to incorporate it within a general problem solver like backtracking. This could be done by keeping the ordering in which backtracking instantiates variables unchanged and enhancing its performance once a tree-structured problem is encountered.

Since all backtracking algorithms work by progressively instantiating sets of variables, all we need to do is 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 a specialized tree-solving algorithm on the remaining problem, i.e., either finding a consistent instantiation for the remaining variables (thus, finding a solution to the entire problem)

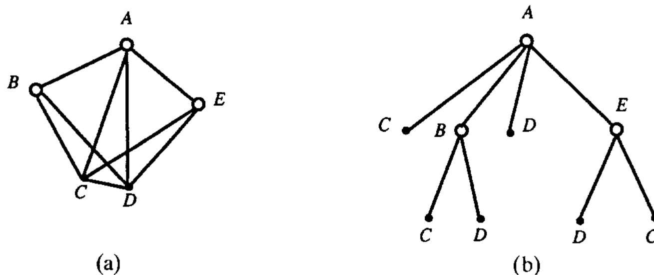


Fig 10 A constraint graph and a constraint tree generated by the cutset $\{C, D\}$

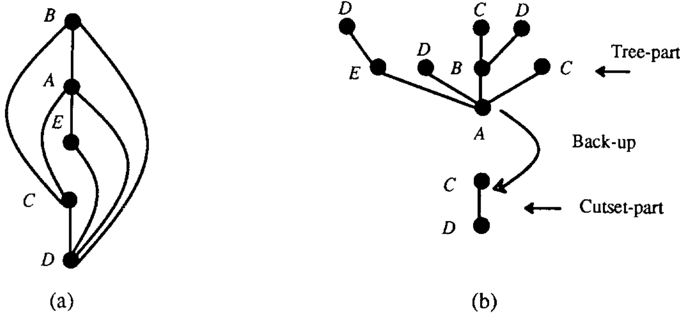


Fig 11 The constraint graph of backtracking with cutset

or concluding that no consistent instantiation for the remaining variables exists (in which case backtracking must take place) For instance, assume that backtracking instantiates the variables of the CSP represented in Fig. 10(a) in the order D, C, E, A, B (Fig. 11(a)). Backtracking with the cutset provision will instantiate variables C and D, and then, realizing that these variables cut all cycles, will invoke a tree-solving routine on the rest of the problem (Fig. 11(b)). If no solution is found, control returns to backtracking which will go back to variable C

Observe that the applicability of this idea is independent of the particular type of backtracking algorithm used (e.g., naive backtracking, backjumping, backtracking with learning, etc., see Fig. 12). When the original problem has a tree constraint graph, the enhanced backtracking scheme coincides with a tree algorithm, and when the constraint graph is complete, the algorithm reverts to naive backtracking.

4.2. Bounds on the performance of the cutset method

In [5] it is shown that tree CSPs can be solved in $O(nk^2)$ and that no algorithm can do better in the worst case. It seems reasonable, therefore, that any backtracking algorithm should improve its worst case bound if it cooperates with a tree algorithm via the cycle-cutset method. This, however, may only be

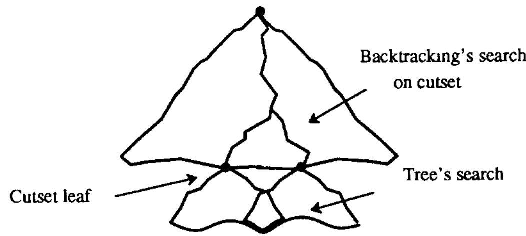


Fig 12 The search space of algorithm backtracking with cutset

true for naive backtracking. Let $M(A)$ denote the worst-case complexity of an algorithm A, where complexity is measured by the number of consistency checks performed.

Theorem 4.1. *Let B be the naive backtracking for solving a CSP, and let B_c be the algorithm resulting from incorporating a tree algorithm in B via the cycle-cutset approach Then*

$$M(B_c) \leq M(B) .$$

Proof. Let S_{cutset} be the search space explored by B, truncated at the depth corresponding to the cutset states, and let $M(S_{cutset})$ be the number of consistency checks used by B to explore this search space. Each leaf node in S_{cutset} corresponds either to a leaf state in the full search space, one which cannot be extended by any consistent assignment, or to an instantiated cycle-cutset. Denote the latter type leaves by CUTSET-LEAVES, and let $M_i(B)$ stand for the effort spent by B in exploring the subtree rooted at the ith state of the CUTSET-LEAVES. The overall complexity of B is given by

$$M(B) = M(S_{cutset}) + \sum_{i \in \text{CUTSET-LEAVES}} M_i(B) .$$

Algorithm B, being naive backtracking, does not acquire any information from searching the subtree rooted at i. Namely, if an oracle were to inform backtracking that a certain state in the CUTSET-LEAVES leads to a dead-end, the rest of the search would be the same, had B discovered this information on its own. Therefore, the truncated search space and the set of CUTSET-LEAVES are the same for both B and B_c. Let TREE be the tree-solving algorithm. The complexity of B_c is given, therefore, by:

$$M(B_c) = M(S_{cutset}) + \sum_{i \in \text{CUTSET-LEAVES}} M_i(\text{TREE}) .$$

Each state in the CUTSET-LEAVES induces a new CSP problem, which, as explained, has a tree-structured graph and therefore can be solved efficiently by a tree algorithm. Moreover, since for naive backtracking $M_i(B)$ is dependent only on state i, we get

$$M_i(\text{TREE}) \leq M_i(B)$$

yielding

$$M(B) \geq M(B_c) . \qquad \square$$

The performance of *any* backtracking algorithm, B_c, that incorporates the cutset approach can be bounded as follows. Let $d = X_1, \ldots, X_n$, be an ordering of the variables, and let $C = X_1, \ldots, X_c$ be a cycle-cutset in the graph.[7] For a state in the CUTSET-LEAVES, only $n - c$ variables remain to be instantiated. Therefore, all tree-structured CSPs induced by these states have $n - c$ variables and they can be solved by a tree algorithm in $O((n - c)k^2)$ (k is the number of values) Switching to a representation required by the tree algorithm may take $O((n - c)ck))$ consistency checks, since each cutset variable must propagate its value to all its neighbors which are not in the cutset. The complexity (i.e., the number of consistency checks) of the TREE algorithm at state i including the transition work is therefore $O((n - c)k^2 + (n - c)ck)$. The number of consistency checks required for generating all CUTSET-LEAVES and the cardinality of this set are bounded by k^c since the CUTSET-LEAVES are the solutions of a CSP restricted to the cutset variables whose cardinality is c. We get:

$$M(B_c) = O(k^c) + O(k^c\{(n - c)k^2 + (n - c)ck\})$$

and, therefore,

$$M(B_c) = O(k^c\{(nk^2) + (n^2k)\}) .$$

Obviously, as the size of the cycle-cutset diminishes the exponential term in the above expression is reduced and we get better upper bounds for the performance of the algorithm. We see that using the cutset method, the exponential term in the upper bound on the performance of any algorithm can be reduced from $O(k^n)$ to $O(k^c)$.

Practically, however, the algorithms rarely exhibit their worst-case performance, and their average-case performance is of greater interest. We do not expect to see the superiority of the cycle-cutset method in every problem instance. This is so because there is no tree algorithm which is superior to all other algorithms for all trees; so, the tree algorithm used in the cycle-cutset method may occasionally perform worse than the original backtracking algorithm

4.3. Experimental evaluation

We compared the performance of backtracking enhanced by the cycle-cutset approach to that of naive backtracking, on the random and planar problems and on one instance of the zebra problem. As before, variables were instantiated in a decreasing order of their degrees.

[7] Observe that, when the ordering of variables is not fixed, each state should be tested for the cycle-cutset property which would render the cycle-cutset method computationally unattractive

The tree algorithm which was used in the cutset method is the one presented in [5], which is optimal for tree CSPs. The algorithm performs directional arc consistency (DAC) from leaves to root, i.e., a child always precedes its parent. If, in the course of the DAC algorithm a variable becomes empty of values, the algorithm concludes immediately that no solution exists. When a solution exists, the tree algorithm assigns values to the variables in a backtrack-free manner, going from the root to the leaves. Many total ordering will satisfy the partial order dictated by the DAC algorithm (e.g., child precedes its parent) and the choice may have a substantial effect on its average performance. The ordering we implemented is the reverse of "in-order" traversal of trees [10]. This ordering compared favorably with other orderings tried. It realizes empty-valued variables early in the DAC algorithm, thus concludes that no solution exists as soon as possible. For completeness we present the tree algorithm:

Tree-backtrack($d = X_1, \ldots, X_n$)
1. begin
2.   call **Dac**(d)
3.   if completed then **Find-solution**(d)
4.   else (return, no solution exists)
5. end

Dac-d-arc-consistency
(the order d is assumed)
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.   if X_j is empty, return (no solution exists)
5.   end
6.   end
7. end

The procedure **Find-solution** is a simple backtracking algorithm on the order d which, in this case, is expected to find a solution with no backtrackings and therefore its complexity is $O(nk)$. The algorithm **Revise**(X_j, X_i) [17] deletes values from the domain of X_j until the directed arc (X_j, X_i) is arc-consistent. The complexity of **Revise** is $O(k^2)$.

In Fig. 13 and Fig. 15 we compare the two algorithms graphically on random CSPs, and in Fig. 14 and Fig. 16 the comparison is on planar CSPs. In Figs. 13 and 14 the x-axis displays the number of consistency checks (on a log-log

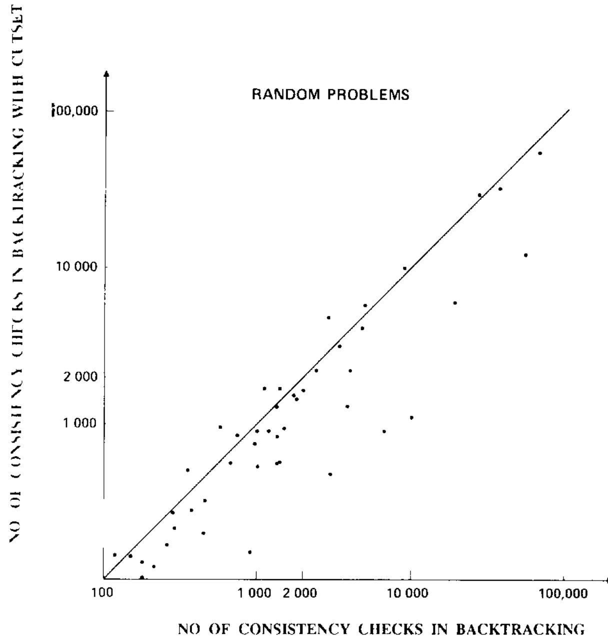


Fig 13 Performance of the cutset method on random problems

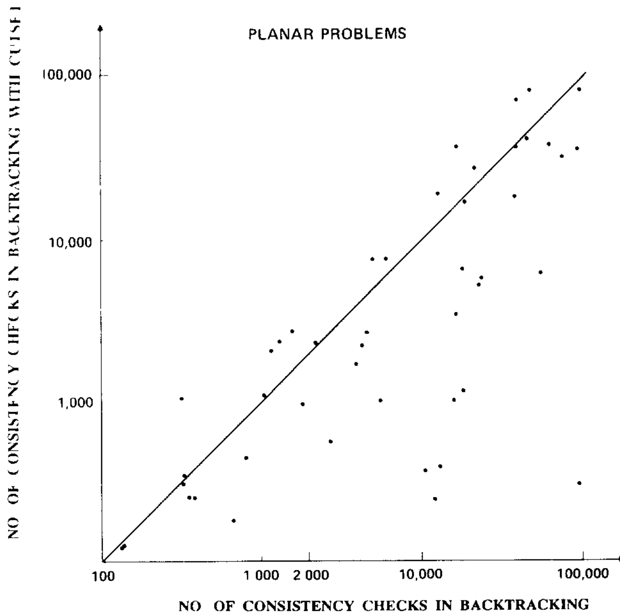


Fig 14 Performance of the cutset method on planar problems

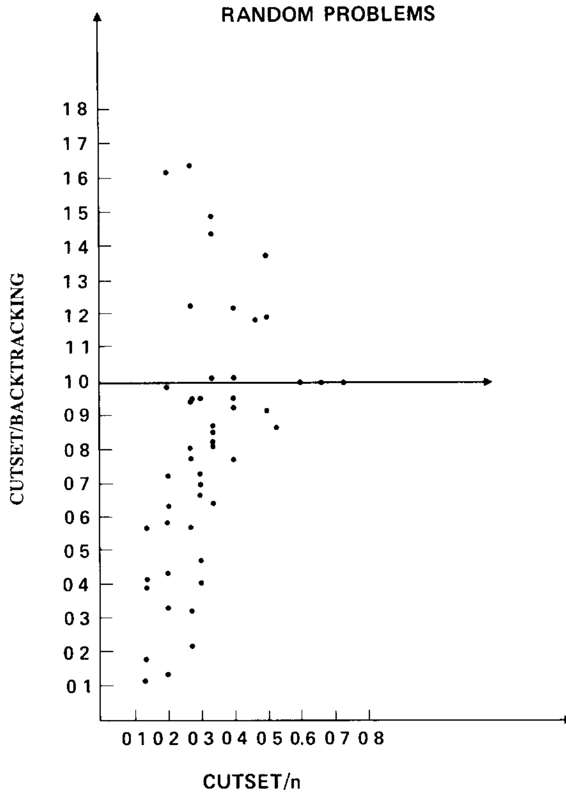


Fig 15 Evaluating the cutset method relative to cutset size on random problems

paper) performed by backtracking (denoted by B) and the y-axis displays the same information for backtracking with cutset (denoted by B_c). Each point in the graph corresponds to one problem instance. The 45-degree line represents equal performance for both algorithms; above this line backtracking did better; and below this line backtracking with cutset did better. We see that most problem instances lie underneath this line.

In Figs. 15 and 16 the graph displays the relationship between the performance of B_c and the size of the cycle-cutset. The x-axis gives the ratio of the cutset size to the number of variables, and the y-axis gives the ratio between the performances of B_c and B. When the ratio of the cutset size to n is less than 0.3, almost all problems lie underneath the line $y = 1$, for which B_c outperformed B. Unlike backjumping, the cycle-outset method does not always improve naive backtracking's peformance. This indicates that for some problem instances the tree algorithm was less efficient than naive backtracking on the tree part of the search space (although its worst-case performance is better). Indeed, while no algorithm for trees can do better than $O(nk^2)$ in the

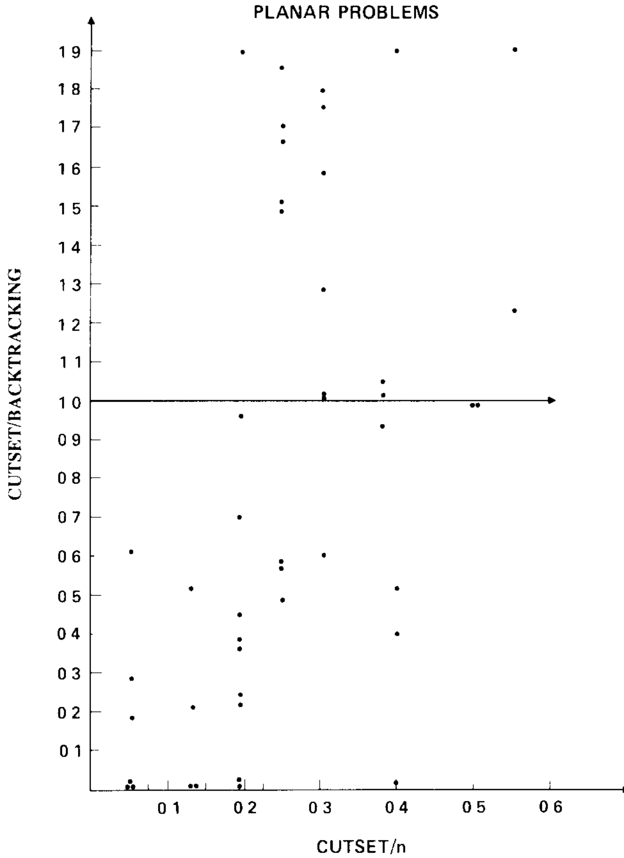


Fig 16 Evaluating the cutset method relative to cutset size on planar problems

worst case, the performance of such algorithms ranges between $O(nk)$ and $O(nk^2)$ when there is a solution, and it can be as good as $O(k^2)$ when no solution exists. It depends mainly on the order of visiting variables, either for establishing arc consistency or for instantiation. Backtracking may unintentionally step in the right order and, since it avoids the preparation work required for switching to a tree representation (which may cost as much as $O(n^2k)$), it may outperform backtracking with cutset. On the average, however, the cutset method improved backtracking by 25% on the planar problems, and by 20% on the random CSPs. Observe that when the size of the cutset is small B_c outperformed B more often.

On the zebra problem the performance of backtracking with and without the cutset method was almost the same (we tested only one instance of the zebra problem). This can be explained by the fact that the constraint graph of this

problem is very dense and 20 out of the 25 variables were required to cut all cycles. Since most of the search is performed by naive backtracking anyway, the impact of the tree algorithm was quite negligible.

As we see, the cycle-cutset method provides a promising approach for improving backtracking algorithms. The experiments demonstrate that the effectiveness of this method depends on the size of the cutset, which provides an a priori criterion for deciding whether or not the method should be utilized in any specific instance.

The effectiveness of this method also depends on the efficiency of the tree algorithm employed and on the amount of adjustment required while switching to a tree representation. The development of an algorithm that exploits the topology of tree-structured problems without intentional pre-processing would be very beneficial.

Since the applicability of the cycle-cutset method seems independent of the particular version of backtracking used, it is interesting to see whether it has a similar effect when it is incorporated with nonnaive versions of backtracking. In the next section we answer this question by combining the enhancement schemes presented in this paper: backjumping and learning, with the cutset method.

5. Integrating the Three Schemes

5.1. Integration principles

In principle the cycle-cutset method can be used with any backtracking scheme, not necessarily naive backtracking. The version of backtracking used will instantiate variables in a fixed order, until a cutset is realized and then it will switch to a tree-solving algorithm. This suggests that the cutset method may improve any backtracking scheme and thus provide a universal improvement. This conclusion, however, is only valid when there is no flow of information which is used by the specific backtracking scheme when it is between the first part of the search (called the *cutset part*) and that corresponding to the tree search (the *tree part*) (see Fig. 12). This assumption is true for naive backtracking but not for all its enhancements. For instance, when backjumping alone searches the tree part of the search space, it gathers some valuable information that helps it prune the search in the cutset part by jumping back efficiently. If the integrated scheme backs up naively from the tree part to the cutset part, no such information will be available.

Consider again the constraint graph of Fig. 10(a) and suppose that backjumping works on this problem in the order $(D \to C \to E \to A \to B)$ (Fig. 17(a)). If, for instance, there is a dead-end at E, backjumping will back up to node D. If the cutset method is integrated "naively" into backjumping, it will instantiate D and C (the cutset variables) and give control to the tree algorithm

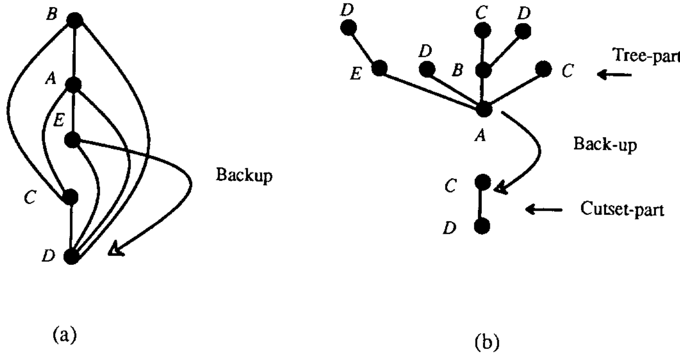


Fig 17 Incorporating backjumping with cutset

(see Fig. 17(b)). When the dead-end at E is encountered the tree algorithm will indifferently switch back to backjumping, providing it no information for skipping C. This difficulty may be corrected if we equip the tree-solving algorithm with the ability to gather the same information needed by backjumping, namely identifying the subset of variables which may be responsible for a failure.

Therefore, the tree algorithm which we integrated with backjumping will return a subset of responsible variables, given a "no solution" situation. If the domain values of variable X_i become empty during DAC (as a result of **Revise**) it implies that only the variables which are located within the processed part of the subtree rooted at X_i may be relevant to this situation The cutset variables of this subtree can be regarded, therefore, as the parent set (also as the conf-set) of this dead-end. These variables will be returned to backjumping which will back up to the most recent among them. If, for instance, the tree part of the problem in the example of Fig. 17 is solved in left to right order and if the algorithm finds that the domain values of E are empty after performing **Revise** on (E, D) it will return D as its only parent (D is the only cutset leaf in the subtree rooted at E) and backjumping will back up to it and not to C as in the naive integration The difference between naive integration and the one suggested here were profound in our experiments and only by this kind of integration would the combined scheme improve w.r.t its individual components.

As to the integration with learning, the same kind of information gathering process, as with backjumpng alone, was used. Namely, upon a "no solution" situation identified at node X_i of the tree, the conf-set is identified (same as the parent sets for backjumping) and returned back for analysis. Shallow learning can be performed on this set. For deep learning an additional analysis of the conf-set should be performed when X_i is considered the dead-end variable

5.2. Experimental evaluation

Figures 18 and 19 compare the performance of backjumping against the performance of backjumping with cutset on random and planar problems. For most *hard* instances (i.e. those requiring more than 1000 consistency checks for backjumping) the integration improved the performance and in some cases quite significantly (the comparison is displayed on a log-log paper as in Fig. 13). On the average backjumping was improved by 25% on these two classes. For the easiest problems, when backjumping performed 1000 consistency checks or less, the integration didn't pay off. The deterioration, however, is not severe: 50% for planar problems and 10% for random problems.

Figure 20 compares the integrated learning and backjumping schemes with their unintegrated counterparts on planar problems. On the right-hand side of the y-axis we repeat the results appearing in Fig. 7 while on the left-hand side we added the corresponding results of the integrated strategy. The actual numbers (without the deep learning results) are given in Table 4. The name of each integrated learning scheme is preceded by "C" to indicate the cutset method which is embedded into it (e.g. CSF stands for cycle-cutset method integrated with shallow first-order learning), "ratio" gives the ratio between

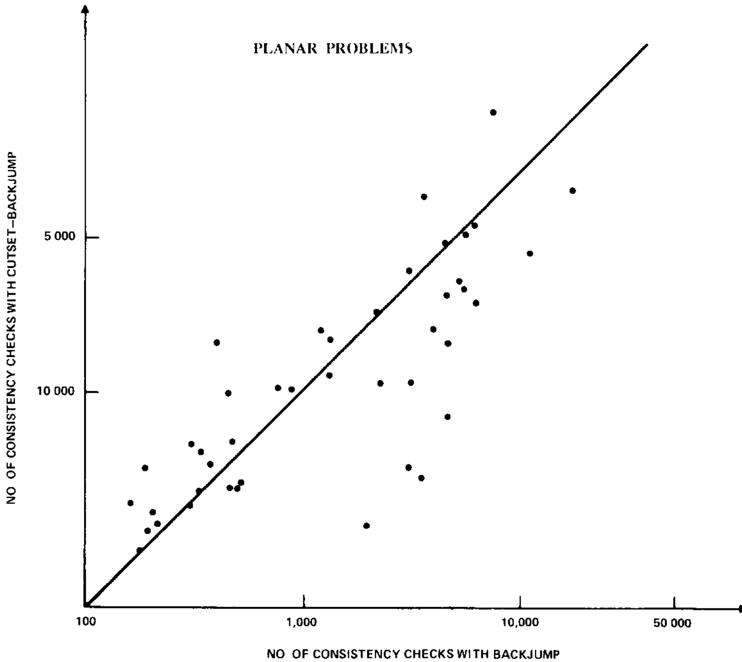


Fig. 18 Comparing backjumping to cutset backjumping on planar problems

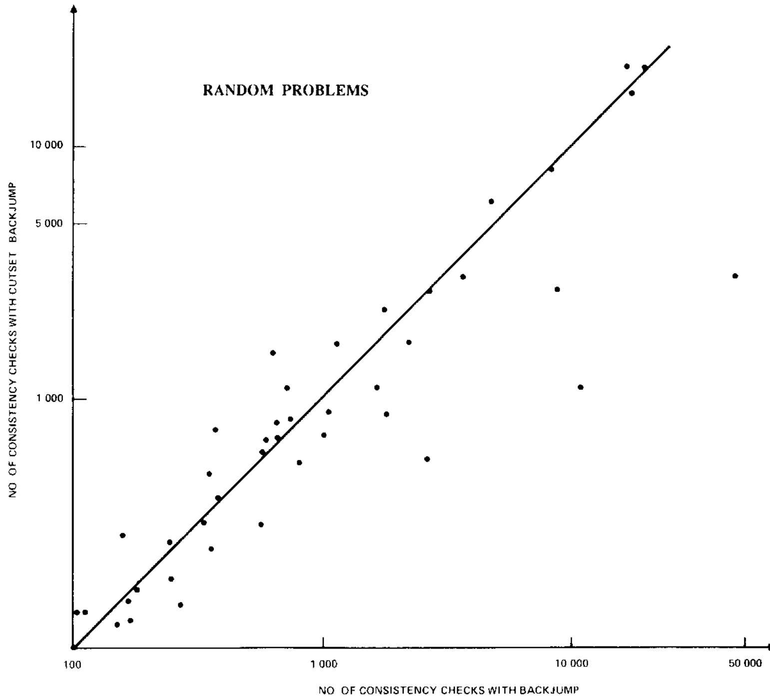


Fig 19 Comparing backjumping to cutset backjumping on random problems

the average cutset size and the number of variables. Thus, we compared the performances of naive backtracking, the cutset method, backjumping, shallow first-order, shallow second-order modify, shallow second-order add, deep first-order, deep second-order modify and deep second-order add, averaged over clusters of instances, with and without the cutset method. We see that the curves to the left of the y-axis are generally below those to the right, indicating an improvement in performance. In two clusters one corresponding to easy problems and one corresponding to 5000–10000 consistency checks (comprised of only two instances) a small deterioration is detected.

We tested the integrated scheme on one instance of the zebra problem, the one on which naive backtracking showed the best performance, and the results are tabulated in Table 5. Backjumping alone improves performance by 50% and combining it with the cutset method produced an additional improvement of 40% despite the large size of the cutset (20 out of the 25 variables are in the cutset). The class-assignment problem was tested on few instances as well, however, due to the inherent easiness of this problem the cutset approach caused a deterioration in performance.

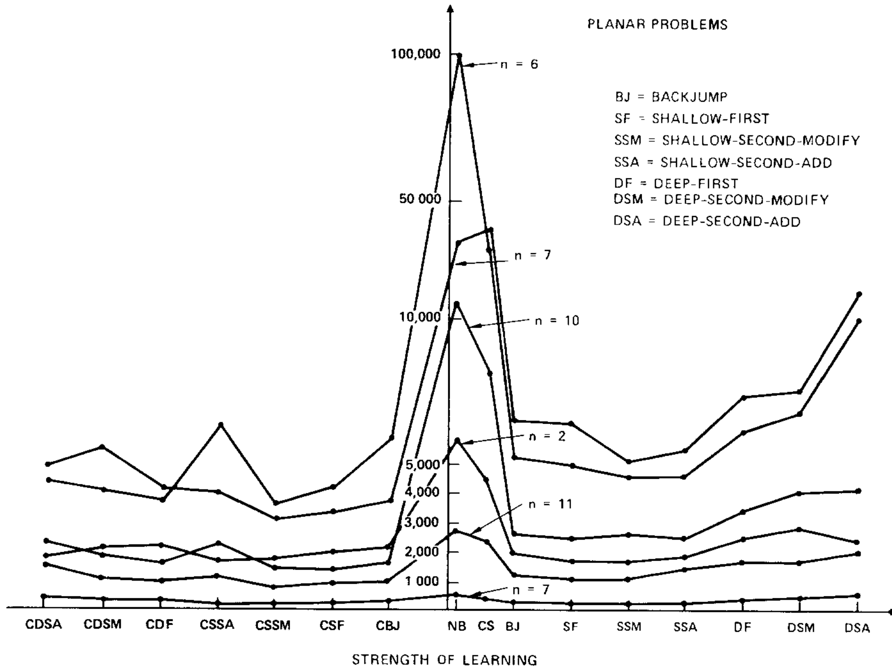


Fig 20 Performance of learning with and without cutset

Table 4
Average number of consistency checks for different backtracks

Range	Number of instances	Ratio	NB	Cutset	BJ	SF	SSM	SSA	CBJ	CSF	CSSM	CSSA
0–1000	7	0 19	454	404	261	248	252	263	285	270	267	267
1000–5000	11	0 29	2702	2360	1229	1152	1125	1423	1023	996	985	1201
5000–10000	2	0 23	5782	4335	1966	1769	1778	1872	2256	1988	1737	1601
10000–20000	10	0 17	15423	8683	2740	2508	2668	2545	1730	1437	1399	2235
20000–50000	7	0 39	35157	39188	5225	5050	4683	4672	3857	3611	3437	3974
50000–100000	7	0 28	104730	33150	6655	6500	5260	5574	5809	4229	3525	6065

Table 5
The zebra problem

NB	BJ	SF	SS	DF	DS	CBJ	CSF	CSS	CDF	CDS
2066	1241	1234	1234	1272	884	782	759	759	813	1189

We see that the integrated strategy provides an improvement on each of its individual constituents; backjumping, learning, and the cycle-cutset method. Each of the individual schemes shows its strength in different classes of problem instances and the integrated scheme takes advantage of each scheme's power when appropriate. For instance, when the constraint graph is sparse, backjumping and the cutset method are most effective When it is highly densed, backjumping and the cutset method lose their effectiveness and the learning schemes take over. For intermediate cases both the cutset method and backjumping cooperate and, on the average, they do better than each one alone.

6. Conclusions

This paper presents and evaluates three schemes for improving the performance of backtracking: backjumping, learning, and cycle-cutset decomposition. Variants of these schemes can be found in various disguises in the literature and this paper presents a unifying formal framework within which such schemes can be defined, evaluated and integrated.

Our experiments and analyses conclude that backjumping defeats naive backtracking on an instance-by-instance basis and, when the constraint graph is sparse, the improvement is significant Since this scheme doesn't trade off anything for its virtues we recommend it should always replace naive backtracking

Learning does involve trade-off consideration. We showed that the amount of learning effects the overall performance. Difficult problems (e.g , the zebra problem) benefited from deeper forms of learning even though it required more time and space, while other problems improved their performance with shallow learning but the additional work required by deep learning was not justified. We conclude, therefore, that learning should be used selectively—recording *all* constraints can cause serious deterioration of performance. Therefore, when no prior knowledge is available regarding the nature of the problem, only shallow forms of learning should be attempted. When the problem is expected to be difficult, deeper, though level-restricted learning should be performed. This advice is particularly relevant for the current versions of TMSs which indiscriminantly record all constraints.

Recording all constraints may be justified in frequently queried knowledge bases, since the cost of learning is amortized over many queries. Our experiment provide a limited evidence to such behavior, showing that the representation resulting from any amount of learning improved the efficiency of answering the same queries again.

The cycle-cutset too improves the worse-case performance of naive backtracking, and integrating it with learning and backjumping revealed its potential for improving any backtracking algorithm. Although this improvement is

not on a case-by-case basis, the average improvements justify incorporating this idea in any algorithm especially if a way can be found to minimize the overhead of integrating the tree algorithm with the hosting algorithm.

Appendix A. The Class-Assignment Problem

A constraint between a student and a class

1. Student(Robert, Prolog)
2. Student(John, Music)
3. Student(John, Prolog)
4. Student(John, Surf)
5. Student(Mary, Science)
6. Student(Mary, Art)
7. Student(Mary, Physics)

A constraint between a Professor and a class

1. Professor(Luis, Prolog)
2. Professor(Luis, Surf)
3. Professor(Maurice, Prolog)
4. Professor(Eureka, Music)
5. Professor(Eureka, Art)
6. Professor(Eureka, Science)
7. Professor(Eureka, Physics)

A trinary constraint among a class, a day, and a place

1. Course(Prolog, Monday, Room1)
2. Course(Prolog, Friday, Room1)
3. Course(Surf, Sunday, Beach)
4. Course(Math, Tuesday, Room1)
5. Course(Math, Friday, Room2)
6. Course(Science, Thursday, Room1)
7. Course(Science, Friday, Room2)
8. Course(Art, Tuesday, Room1)
9. Course(Physics, Thursday, Room3)
10. Course(Physics, Saturday, Room2)

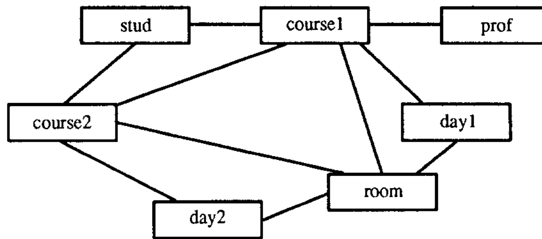


Fig 21 The constraint graph of the class-assignment problem

The *query* is: find Student(stud, course1) and Course(course1, day1, room) and Professor(prof, course1) and Student(stud, course2) and Course(course2, day2, room) and noteq(course1, course2)

The constraint graph corresponding to the problem's representation as a binary CSP is shown in Fig. 21.

Appendix B. The Zebra Problem

1. There are five houses, each of a different color and inhabited by men of different nationalities, with different pets, drinks, and cigarettes.
2. The Englishman lives in the red house.
3. The Spaniard owns a dog
4. Coffee is drunk in the green house.
5. The Ukranian drinks tea.
6. The green house is immediately to the right of the ivory house
7. The Old-Gold smoker owns snails.
8. Kools are being smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house on the left.
11. The Chesterfield smoker lives next to the fox owner.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky-Strike smoker drinks orange juice.
14. The Japanese smokes Parliament.
15. The Norwegian lives next to the blue house.

The *query* is: Who drinks water and who owns the zebra?

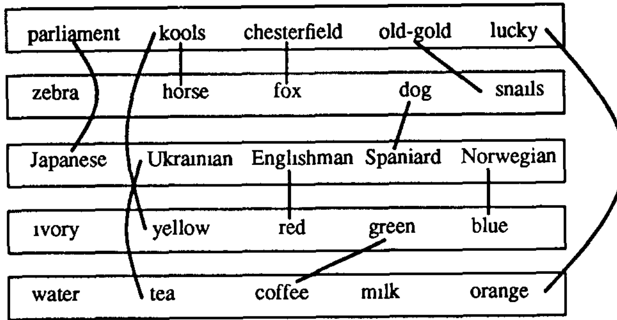


Fig. 22 The constraint graph of the zebra problem

The problem can be represented as a binary CSP using 25 variables divided into 5 clusters as follows:

1. red; blue; yellow; green; ivory.
2. norwegian; ukrainian; englishman; spaniard; japanese.
3. coffee; tea; water; milk; orange.
4. zebra; dog; horse; fox; snails.
5. old-gold; parliament; kools; lucky; chesterfield.

Each of the variables has the domain values $\{1, 2, 3, 4, 5\}$ associating a house number with the characteristic represented by the variable (e.g., assigning the value 2 to the variable *red* mean that the second house is red, etc.)

The constraints of the puzzle are translated into binary constraints among the variables. For instance, the sentence "The Spaniard owns a dog" describes a constraint between the variable *spaniard* and the variable *dog* that allows only the pairs: $\{(1, 1)(2, 2)(3, 3)(4, 4)(5, 5)\}$. In addition, there is a constraint between any pair of variables of the same "cluster" ensuring that they are not assigned the same value. The constraint graph for this problem is given in Fig. 22 (the constraints among the variables of each cluster are omitted for clarity).

Appendix C. A Detailed Example

This appendix illustrates the operation of backtracking with learning on the CSP in Fig. 2, assuming that the search is conducted in the order $(X_1, X_2, X_3, X_4, X_5)$. In Fig. 23 we follow step by step the performance of backtracking with shallow second-order learning. The steps in Fig. 23 correspond to consistent states generated by backtracking and each string of values corresponds to an instantiation according to the above order. $C_{i,j}$ denotes the constraint between variables X_i and X_j and the arrows represent backtrack

1	*a*	
2	*aa*	Conf = $\{X_1 = a, X_2 = a\}$ → **add** $C_{1,2}$, delete (a, a) from $C_{1,2}$
3	*ab*	Conf = $\{X_1 = a, X_2 = b\}$ → **modify** $C_{1,2}$, delete (a, b) from $C_{1,2}$
4	*ac*	Conf = $\{X_1 = a, X_2 = c\}$ → **modify** $C_{1,2}$, delete (a, c) from $C_{1,2}$
5		Conf = $\{X_1 = a\}$ → **modify** C_1, delete a, from C_1
6	*b*	
7	*ba*	Conf = $\{X_1 = b, X_2 = a\}$ → **modify** $C_{1,2}$, delete (b, a) from $C_{1,2}$
8	*bb*	
9	*bba*	
10	*bbab*	Conf = $\{X_3 = a, X_4 = b\}$ → **modify** $C_{3,4}$, delete (a, b) from $C_{3,4}$
11		Conf = $\{X_3 = a\}$ → **modify** C_3, delete a from C_3
12		Conf = $\{X_1 = b, X_2 = b\}$ → **modify** $C_{1,2}$, delete (b, b) from $C_{1,2}$
13	*bc*	Conf = $\{X_1 = b\}$ → **modify** C_1, delete b from C_1
14	*c*	
15	*ca*	Conf = $\{X_2 = a\}$ → **modify** C_2, delete a from C_2
16	*cb*	Conf = $\{X_2 = b\}$ → **modify** C_2, delete b from C_2
17	*cc*	
18	*ccb*	
19	*ccba*	Conf = $\{X_3 = b, X_4 = a\}$ → **modify** $C_{3,4}$, delete (b, a) from $C_{3,4}$
20	*ccbb*	Conf = $\{X_3 = b, X_4 = b\}$ → **modify** $C_{3,4}$, delete (b, b) from $C_{3,4}$
21		Conf = $\{X_3 = b\}$ → **modify** C_3 delete b from C_3

Fig 23 Simulating shallow second-order learning

points, that is, states which cannot be extended. With each such point the learning scheme identifies the conf-set and records the constraint discovered. Consecutive back-arrows correspond to several consecutive backtracks needed to reach a state that can be extended. With each such backtrack the learning procedure either adds or modifies a constraint.

When the search starts, X_1 is assigned the value a and then X_2 is assigned the value a (this is indicated by the first two steps in Fig. 23). This state cannot be extended by any value of X_3 and therefore there is a backtrack point. The conf-set includes the whole dead-end state and therefore a binary constraint between X_1 and X_2 is *added* excluding the pair (a, a) from the constraint $C_{1,2}$ Later on, the constraint $C_{1,2}$ is modified and the pairs (a, b) and (a, c) are deleted. When the search reaches state (b, b, a, b) (step 10), which cannot be extended by X_5, three consecutive backtracks are performed. The first results in modifying the constraint $C_{3,4}$ and the second in deleting the value a from C_3. This last modification helps prune the search. For instance, in step 13, the state (b, c) is not extended to (b, c, a) since a is no longer in the domain of X_3

ACKNOWLEDGEMENT

I thank the AI Lab at Hughes Aircraft for providing me with a very supportive environment for this work In particular Dave Payton, Dave Keirsey, Ken Rosenblatt, and Charles Dolan offered valuable help in getting these experiments going. I also thank Itay Meiri, Judea Pearl and Greg Provan for valuable comments on previous versions of this manuscript I am especially grateful to my husband, Avi Dechter, who helped me shape these ideas and bring the manuscript to readable form

REFERENCES

1 M Bruynooghe, Solving combinatorial search problems by intelligent backtracking, *Inf. Process Lett* **12** (1981) 36–39

2 M Bruynooghe and L M Pereira, Deduction revision by intelligent backtracking, in J A. Campbell, ed , *Implementation of Prolog* (Ellis Horwood, Chichester, 1984) 194–215

3 J G Carbonell, Learning by analogy Formulation and generating plan from past experience, in· R S Michalski, J G Carbonell and T M Mitchell, eds , *Machine Learning: An Artificial Intelligence Approach* (Tioga, Palo Alto, CA, 1983)

4 P T Cox, Finding backtrack points for intelligent backtracking, in: J A Campbell, ed., *Implementation of Prolog* (Ellis Horwood, Chichester, 1984) 216–233

5 R Dechter and J Pearl, Network-based heuristics for constraint-satisfaction problems, *Artificial Intelligence* **34** (1987) 1–38

6. R Dechter and J Pearl, The cycle-cutset method for improving search performance in AI applications, in *Proceedings 3rd IEEE on AI Applications*, Orlando, FL (1987)

7 R Dechter and J. Pearl, Tree clustering for constraint networks, *Artificial Intelligence* **38** (1989) 353–366

8. J de Kleer, An assumption-based TMS, *Artificial Intelligence* **28** (1986) 127–162

9 J Doyle, A truth maintenance system, *Artificial Intelligence* **12** (1979) 231–272

10. S Even, *Graph Algorithms* (Computer Science Press, Rockville, MD, 1979)

11 R E. Fikes and N.J Nilsson, STRIPS A new approach to the application of theorem proving to problem solving, *Artificial Intelligence* **2** (1971) 189–208

12 E C Freuder, A sufficient condition of backtrack-free search, *J ACM* **29** (1) (1982) 24–32.

13 J Gaschnig, Performance measurement and analysis of certain search algorithms, Tech Rept CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh, PA (1979)

14 R.M Haralick and G L Elliott, Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence* **14** (1980) 263–313

15 R E Korf, A program that learns how to solve Rubik's cube, in: *Proceedings AAAI-82*, Pittsburgh, PA (1982) 164–167

16 J E Laird, P S Rosenbloom, and A. Newell, Towards chunking as a general learning mechanism, in *Proceedings AAAI-84*, Austin, TX (1984)

17 A K Mackworth, Consistency in networks of relations, *Artificial Intelligence* **8** (1977) 99–118

18 A K Mackworth and E C Freuder, The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, *Artificial Intelligence* **25** (1985) 65–74

19 J P Martins and S C Shapiro, Theoretical foundations for belief revision, in *Proceedings Theoretical Aspects of Reasoning about Knowledge* (1986)

20 S Matwin and T Pietrzykowski, Intelligent backtracking in plan-based deduction, *IEEE Trans Pattern Anal Mach Intell* **7** (1985) 682–692

21 D A McAllester, An outlook on truth-maintenance, Tech Rept , AI Memo No 551, MIT, Boston, MA (1980)

22 T M Mitchell, R.M Keller, and S T. Kedar-Cabelli, Explanation-based generalization· A unifying view, *Machine Learning* **1** (1986) 47–80

23 T Mitchell, P E Utgoff and R Banerji, Learning by experimentation, acquiring and refining problem solving heuristics, in R S Michalski, J G Carbonell and T M Mitchell, eds , *Machine Learning An Artificial Intelligence Approach* (Tioga, Palo Alto, CA, 1983) 163–190
24 U Montanari, Networks of constraints Fundamental properties and applications to picture processing, *Inf Sci* **7** (1974) 95–132
25 B Nudel, Consistent-labeling problems and their algorithms Expected-complexities and theory-based heuristics, *Artificial Intelligence* **21** (1983) 135–178
26 C J Petrie, Revised dependency-directed backtracking for default reasoning, in *Proceedings AAAI-87*, Seattle, WA (1987) 167–172
27 G Provan, Complexity analysis of multiple-context TMSs in scene representation, in *Proceedings AAAI-87*, Seattle, WA (1987) 173–177
28 PW Purdom, Search rearrangement backtracking and polynomial average time, *Artificial Intelligence* **21** (1983) 117–133
29 R Reiter and J de Kleer, Foundations of assumption-based truth maintenance systems Preliminary report, in *Proceedings AAAI-87*, Seattle WA (1987)
30 W Rosiers and M Bruynooghe, Empirical study of some constraint satisfaction algorithms, Tech Rept CW 50, Katholieke Universiteit Leuven, Leuven, Belgium (1986)
31 M Sims, An AI approach to analytic discovery in mathematics, Departments of Computer Science and Mathematics, Rutgers University, New Brunswick, NJ (1988)
32 R M Stallman and G J Sussman, Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence* **9** (1977) 135–196
33 H S Stone and J M Stone, Efficient search techniques An empirical study of the *N*-queens problem, Tech Rept RC 12057 (#54343), IBM T J Watson Research Center, Yorktown Heights, NY (1986)
34 G J Sussman and G L Steele Jr, CONSTRAINTS A language for expressing almost-hierarchical descriptions, *Artificial Intelligence* **14** (1980) 1–39
35 P van Hentenryck and M Dincbas, Forward checking in logic programming, in J -L Lassez, ed , *Proceedings 4th International Conference on Logic Programming* (MIT Press, Cambridge, MA, 1987) 229–255
36 D Waltz, Understanding line drawings of scenes with shadows, in P H Winston, ed , *The Psychology of Computer Vision* (McGraw-Hill, New York, 1975)
37 R Zabih and D McAllester, A rearrangement search strategy for determining propositional satisfiability, in *Proceedings AAAI-88*, St Paul, MN (1988)

Received December 1987; revised version received August 1988