TECHNICAL REPORT
R-121-AI

ELSEVIER

Artificial Intelligence 68 (1994) 211–241

Artificial Intelligence

Experimental evaluation of preprocessing algorithms for constraint satisfaction problems

Rina Dechter[a,*], Itay Meiri[b]

[a]Information and Computer Science Department, University of California, Irvine, CA 29717–3425, USA

[b]Cognitive Systems Laboratory, Computer Science Department, University of California, Los Angeles, CA 90024, USA

Received February 1992; revised July 1993

Abstract

This paper presents an experimental evaluation of two orthogonal schemes for preprocessing constraint satisfaction problems (CSPs). The first of these schemes involves a class of local consistency techniques that includes *directional arc consistency*, *directional path consistency*, and *adaptive consistency*. The other scheme concerns the prearrangement of variables in a linear order to facilitate an efficient search. In the first series of experiments, we evaluated the effect of each of the local consistency techniques on *backtracking* and *backjumping*. Surprisingly, although *adaptive consistency* has the best worst-case complexity bounds, we have found that it exhibits the worst performance, unless the constraint graph was very sparse. *Directional arc consistency* (followed by either *backjumping* or *backtracking*) and *backjumping* (without any preprocessing) outperformed all other techniques: moreover, the former dominated the latter in computationally intensive situations. The second series of experiments suggests that *maximum cardinality* and *minimum width* are the best preordering (i.e., static ordering) strategies, while *dynamic search rearrangement* is superior to all the preorderings studied.

1. Introduction

Constraint satisfaction tasks belong to the class of NP-complete problems and, as such, normally lack realistic measures of performance. Worst-case analysis, because it depends on extreme cases, may yield an erroneous view of typical performance of algorithms used in practice. Average-case analysis, on the other

* Corresponding author. E-mail: dechter@ics.uci.edu.

0004-3702/94/$07.00 © 1994 Elsevier Science B.V. All rights reserved
SSDI 0004-3702(93)E0057-S

hand, is extremely difficult and is highly sensitive to simplifying theoretical assumptions. Thus, theoretical analysis must be supplemented by experimental studies.

The most thorough experimental studies reported so far include Gaschnig's comparisons of *backjumping*, *backmarking* and constraint propagation [12], Haralick and Elliot's study of look-ahead strategies [14], Brown and Purdom's experiments with dynamic variable orderings [21, 22], and, more recently, Dechter's experiments with structure-based techniques [3], and Prosser's hybrid tests with *backjumping* and *forward-checking* strategies [20]. Additional studies were reported in [6, 13, 23, 26, 27].

Experimental studies are most informative when conducted on a "representative" set of problems from one's own domain of application. However, this is very difficult to effect. Real-life problems are often too large or too ill-defined to suit a laboratory manipulation. A common compromise is to use either randomly generated problems or canonical examples (e.g., n-queens, crossword puzzles, and graph-coloring problems). Clearly, conclusions drawn from such experiments reflect only on problem domains that resemble the experimental conditions and caution must be exercised when generalizing to real-life problems. Such experiments do reveal the crucial parameters of a problem domain, and so help establish the relative usefulness of various algorithms.

Our focus in this paper is on algorithms whose performance, as revealed by worst-case analysis, is dependent on the topological structure of the problem. Our aim is to uncover whether the same dependency is observed empirically and to investigate the extent to which worst-case bounds predict actual performance. Our primary concern is with preprocessing algorithms and their effect on *backtracking*'s performance. Since our preprocessing algorithms are dependent on a static ordering of the variables they invite different heuristics for variable ordering. We tested the effect of such orderings on the preprocessing algorithms as well as on regular *backtracking* and *backjumping*.

We organized our experimental results into two classes. The first class concerns consistency enforcing algorithms, which transform a given constraint network into a more explicit representation. On this more explicit representation, any *backtracking* algorithm is guaranteed to encounter fewer deadends [16]. Since these algorithms are polynomial while *backtracking* is exponential, and since they always improve search, one may hastily conclude that they should always be exercised. Our aim was to test this hypothesis. The three consistency enforcing algorithms tested are *directional arc consistency* (*DAC*), *directional path consistency* (*DPC*), and *adaptive consistency* (*ADAPT*) [6]. These algorithms represent increasing levels of preprocessing effort as well as an increasing improvement in subsequent search. Although *DAC* and *DPC*, whose complexities are quadratic and cubic, respectively, can still be followed by exponential search (in the worst case), *ADAPT* is guaranteed to yield a solution in time bounded by $O(\exp(W^*))$, where W^* is a parameter reflecting the sparseness of the network.

Our results show, contrary to predictions based on worst-case analysis, that the average complexity of *backtracking* on our randomly generated problems is far

from exponential. Indeed the preprocessing performed by the most aggressive scheme, *ADAPT*, did not pay off unless the graph was very sparse, in spite of its theoretical superiority to *backtracking*. On the other hand, the least aggressive scheme, *DAC*, came out as a winner in computationally intensive cases. Apparently, *DAC* performs just the desired amount of preprocessing. Additionally, while *ADAPT* showed that its average complexity is exponentially dependent on W^*, the dependence of all other schemes on W^* seems to be quite weak or even non-existent.

In the second class we report the effect of various static ordering strategies on *backtracking* and *backjumping* without preprocessing. Static orderings, in contrast to dynamic orderings, are appealing in that they do not require any overhead during search. We tested four static heuristic orderings, *minimum width* (*MIN*), *maximum degree* (*DEG*), *maximum cardinality* (*CARD*), and *depth-first search* (*DFS*). Those orderings are advised when analyzing their effect on the preprocessing algorithms *ADAPT* and even *DPC* as they yield a low W^*. Although no worst-case complexity ties *backtracking* or *backjumping* to W^*, we nevertheless wanted to discover whether a correlation exists, and which of these static orderings yields a better average search. Lastly, in order to relate our experiments with other experiments reported in the literature, we compared our static ordering with one dynamic ordering, *dynamic search rearrangement* (*DSR*) [21]. We tested two implementation styles of *DSR*, presenting a tradeoff between space and time overhead.

Our results show that *minimum width* and *maximum cardinality* clearly dominated the *maximum degree* and *depth-first search* orderings. However, the exact relationship between the first two is still unclear. While *dynamic ordering* was only second or third best when implemented in a brute-force way it outperformed all static orderings when a more careful implementation that restricted its time overhead was introduced.

The remainder of the paper is organized as follows: we review the constraint network model and general background in Section 2, present the tested algorithms in Section 3, describe the experimental design in Section 4, discuss the results in Section 5, and provide a summary and concluding remarks in Section 6.

2. Constraint processing techniques

A **constraint network** (CN) consists of a set of **variables** $X = \{X_1, \ldots, X_n\}$, each associated with a **domain** of discrete values $D_1, \ldots, D_n$, and a set of **constraints** $\{C_1, \ldots, C_t\}$. Each constraint is a relation defined on a subset of variables. The tuples of this relation are all the simultaneous value assignments to this variable subset which, as far as this constraint alone is concerned, are legal.[1]

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

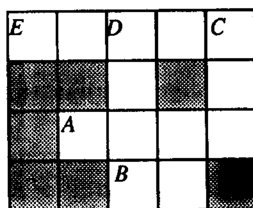
Formally, constraint C_i has two parts: a subset of variables $S_i = \{X_{i_1}, \ldots, X_{i_{j(i)}}\}$ on which it is defined, called a **constraint-subset**, and a **relation** rel_i defined over S_i: $rel_i \subseteq D_{i_1} \times \cdots \times D_{i_{j(i)}}$. The **scheme** of a CN is the set of its constraint subsets, namely, $scheme(CN) = \{S_1, S_2, \ldots, S_t\}$, $S_i \subseteq X$. An assignment of a unique domain value to each member of some subset of variables is an **instantiation**. An instantiation is a **solution** only if it satisfies *all* the constraints. The **set of all solutions** is a relation ρ defined on the set of all variables. Formally,

$$\rho = \{(X_1 = x_1, \ldots, X_n = x_n) \mid \forall S_i \in scheme,\ \Pi_{S_i}\rho \subseteq rel_i\}\ , \tag{1}$$

where $\Pi_X\rho$ is the projection of relation ρ over a subset of its variables X, namely it is the set of all subtuples over X that can be extended to a full tuple in ρ.

A CN may be associated with a constraint graph in which nodes represent variables and arcs connect variables that appear in the same constraint. For example, the CN depicted in Fig. 1(a) represents a crossword puzzle. The variables are E, D, C, A, and B. The scheme is $\{ED, EC, CA, AD, DB\}$. For instance, the pair DE is in the scheme since the word associated with D and the word associated with E share a letter. The constraint graph is given in Fig. 1(b).

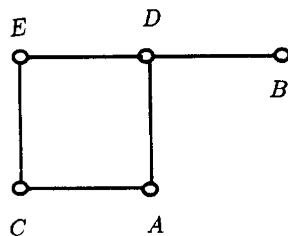
Typical tasks defined on a CN are determining whether a solution exists, finding one solution or the set of all solutions, and establishing whether an instantiation of a subset of variables is part of a global solution. Collectively, these tasks are known as **constraint satisfaction problems** (CSPs).



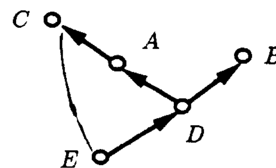
D_E = {hoses,laser,sheet,snail,steer}
D_A=D_D = {hike,aron,keet,earn,same}
D_C = {run,sun,let,yes,eat,ten}
D_B = {no,be,us,it}

C_{AB} = {(hoses,same),(laser,same),(sheet,earn),(snail,aron),(steer,earn)}

(a)



(b)



(c)

Fig. 1. (a) A crossword puzzle (D denotes domains of variables, C_{AB} is the constraint between variables A and B), (b) its CN representation, and (c) a depth-first search preordering.

Techniques used in processing constraint networks can be classified into three categories: (1) **search algorithms**, for systematic exploration of the space of all solutions, which all have backtracking as their basis; (2) **consistency enforcing algorithms**, that enforce consistency on small parts of the network, and (3) **structure-driven algorithms**, which exploit the topological features of the network to guide the search. Hybrids of these techniques are also available. For a detailed survey of constraint processing techniques, see [4, 15].

Backtracking traverses the search space in a depth-first fashion. The algorithm typically considers the variables in some order. It systematically assigns values to variables until either a solution is found or the algorithm reaches a **deadend**, where a variable has no value consistent with previous assignments. In this case the algorithm backtracks to the most recent instantiation, changes the assigned value, and continues. It is well known that the worst-case running time of *backtracking* is exponential.

Improving the efficiency of *backtracking* amounts to reducing the size of the search space it expands. Two types of procedures were developed: preprocessing algorithms that are employed prior to performing the search, and dynamic algorithms that are used during the search.

The preprocessing algorithms include a variety of **consistency enforcing algorithms** [9, 16, 18]. These algorithms transform a given CN into an equivalent, yet more explicit form, by deducing new constraints to be added to the network. Essentially, a consistency enforcing algorithm makes a small subnetwork consistent relative to its surrounding constraints. For example, the most basic consistency algorithm, called arc consistency or 2-consistency (also known as constraint propagation or constraint relaxation), ensures that any legal value in the domain of a single variable has a legal match in the domain of any other variable. Path consistency (or 3-consistency) ensures that any consistent solution to a two-variable subnetwork is extensible to any third variable, and, in general, i-consistency algorithms guarantee that any locally consistent instantiation of $i-1$ variables is extensible to any ith variable. The algorithms, *DAC*, *DPC*, and *ADAPT* are all restricted (because they take into account the direction in which *backtracking* instantiates the variables) versions of these consistency enforcing algorithms.

The preprocessing algorithms also include algorithms for ordering the variables prior to search. Several heuristics for **static orderings** have been proposed [7, 10]. The heuristics used in this paper—*minimum width*, *maximum cardinality*, *maximum degree*, and *depth-first search*—follow the intuition that tightly constrained variables should be instantiated first.

Strategies that dynamically improve the pruning power of backtracking can be classified as either **look-ahead schemes** or **look-back schemes**. Look-ahead schemes are invoked whenever the algorithm is about to assign a value to the next variable. Some schemes, such as *forward-checking*, use constraint propagation [14, 28] to predict the way in which the current instantiation restricts future assignments of values to variables. An example of a look-ahead scheme is *dynamic search rearrangement*, which decides what variable to instantiate next

[10, 21, 26]. Look-back schemes are invoked when the algorithm encounters a deadend and prepares to backtrack. An example of such a scheme is **backjumping** [12]. By analyzing the reasons for the deadend it is often possible to go back directly to the source of failure instead of to the immediate predecessor in the ordering. The algorithm may also record the reasons for the deadend so that the same conflicts will not arise again later in the search (terms used to describe this idea are constraint recording and no-good constraints). *Dependency-directed backtracking* incorporates both *backjumping* and no-good recording [3, 25].

Structure-based techniques, such as *graph-based back-jumping directional i-consistency*, *adaptive consistency*, and *cycle-cutset scheme* can be viewed as structure-based improvements of some of the above techniques [4].

3. The tested algorithms

We first present our two search algorithms, *backtracking* and *backjumping*, and then describe the consistency enforcing algorithms and the ordering heuristics we used.

3.1. Backtracking and backjumping

A backtracking algorithm for finding one solution is given in Fig. 2. It is defined by two recursive procedures, *forward* and *go-back*. The first extends a current partial assignment if possible, and the second handles deadend situations. The procedures maintain lists of candidate values, C_i, for each variable, X_i. Their initial values are computed by *compute-candidates*$(x_1, \ldots, x_i, X_{i+1})$, which selects all values in the domain of variable X_{i+1} that are consistent with previous assignments. *Backtracking* starts by calling *forward* with $i = 0$, namely, the instantiated list is empty.

Backjumping improves the *go-back* phase of *backtracking*. Whenever a deadend occurs at variable X, it backs up to the most recent variable Y connected to X in the constraint graph. If variable Y has no more values, then it should back up more, to the most recent variable Z connected to both X and Y, and so on. This algorithm is a graph-based variant of Gaschnig's *backjumping* [12] which extracts knowledge about dependencies from the constraint graph alone. *Graph-based backjumping* has been shown to outperform *backtracking* on an instance-by-instance basis [3]. For simplicity, *backjumping* refers to *graph-based backjumping* throughout the remainder of this paper.

In our implementation of *backjumping*, both *forward* and *jump-back* (the *go-back* variant of *backjumping*) carry a parameter P, that stores the set of variables that need to be consulted upon the next deadend (see Fig. 3). Accordingly, lines 6 and 8 of *forward* are changed to *forward*$(x_1, \ldots, x_i, x_{i+1}, P)$ and *jump-back*$(x_1, \ldots, x_i, X_{i+1}, P)$. Procedure *jump-back* is shown in Fig. 3. Its parameters are the partial instantiation $x_1, \ldots, x_i$, the deadend variable X_{i+1}, and P.

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. $x_{i+1} \leftarrow$ first element in C_{i+1}, and
5. remove x_{i+1} from C_{i+1}, and
6. *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 solution 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

Fig. 2. Algorithm *backtracking*.

jump-back$(x_1, \ldots, x_i, X_{i+1}, P)$
Begin
1. if $i = 0$, exit (no solution exists)
2. $PARENTS \leftarrow Parents^2(X_{i+1})$
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

Fig. 3. Procedure *jump-back*.

[2] $Parents(X_i)$ are those variables connected to X_i that precede X_i in the ordering.

Consider, for instance, the ordered constraint graph in Fig. 1(a). If the search is performed in the order E, D, A, C, B, and a deadend occurs at B, the algorithm will jump back to variable D since B is not connected to either C or A.

In general, the implementation of *backjumping* requires careful maintenance of each variable's parent set. Some orderings, however, facilitate a simple implementation. If we perform a *depth-first search* (*DFS*) on the constraint graph (to generate a *DFS* tree) and apply *backjumping* along the resulting *DFS* numbering [8], finding the *jump-back* destination amounts to going back to the parent of X in the *DFS* tree. A *DFS* tree of the graph in Fig. 1(b) is given in Fig. 1(c). The directed arcs are part of the *DFS* tree. The rest of the arcs are undirected [8]. The *DFS* ordering (which amounts to an in-order traversal of this tree) is (E, D, B, A, C). Again, if a deadend occurs at node A, the algorithm retreats to its parent in the *DFS* tree, D. When *backjumping* is performed along a *DFS* ordering of the variables, its complexity can be bounded by $O(\exp(m))$ steps, where m is the depth of the *DFS* tree [2].

The *backjumping* procedure we use here is relatively conservative in that variables are eliminated from the parent set only when they are jumped back to (see step 5 of *jump-back*). This may result in deterioration of performance however. As the size of the parent set gets larger, *backjumping* will have less opportunities to execute big jumps back, and its performance will converge into that of naive *backtracking*. This can be corrected by a more sophisticated "parent releasing" method. One approach is to index each variable with the trigger variable that introduced it to the parent set, and releasing it upon processing that trigger variable by *forward*. For a discussion of several such improvements of *backjumping* see [11, 20].

3.2. Local consistency algorithms

Deciding the consistency level that should be enforced on the network is not a clear-cut choice. Generally, *backtracking* will benefit from representations that are as explicit (therefore of a higher consistency level) as possible. However, the complexity of enforcing i-consistency is exponential in i (and may also require exponential memory). Thus, there is a tradeoff between the effort spent on preprocessing and that spent on search. The primary goal of our paper is to uncover this tradeoff.

Algorithms *DAC*, *DPC*, and *ADAPT*, being the directional versions of *arc consistency*, *path consistency and n-consistency*, respectively, have the advantage that they take into account the direction in which *backtracking* will eventually search the problem. Thus, they avoid processing many constraints that are not encountered during search.

We start with *ADAPT*, then generalize its underlying principle to describe a class of preprocessing algorithms that contains both *DAC* and *DPC*. Given an ordering of the variables, the **parent set** of a variable X is the set of all variables connected to X (in the constraint graph) that precede X in the ordering. The **width of a node** is the size of its parent set. The **width of an ordering** is the maximum width of nodes in that ordering, and the **width of a graph** is the minimal

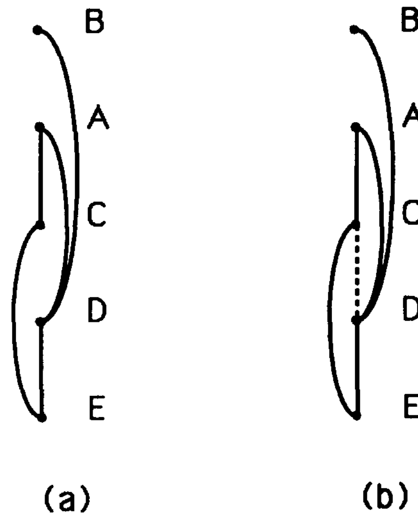


Fig. 4. Ordered constraint graphs.

width of all its orderings. For instance, given the ordering (E, D, C, A, B) in Fig. 4(a), the width of node B is 1, while the width of this ordering is 2 (since the width of A is 2 and there is no node having a larger width). Since there is no width-1 ordering, the width of this graph is 2. Algorithm *ADAPT*, shown in Fig. 5, processes the nodes in a reverse order, that is, each node is processed before any of its parents.

The procedure *record-constraint*(X, SET) generates and records those tuples of variables in SET that are consistent with at least one value of X. For instance, in our example, if A has only one feasible word, *aron*, in its domain and C and D each have their initial domains, then the call for *record-constraint*(A, $\{C, D\}$) will result in recording a constraint on the variables $\{C, D\}$, allowing only the pairs {(*earn*, *run*), (*earn*, *sun*), (*earn*, *ten*)}. *ADAPT* may tighten existing constraints as well as impose constraints over new clusters of variables. It has been shown [6] that, when the procedure terminates, *backtracking* can solve the problem, in the

---

ADAPT($X_1, \ldots, X_N$)
Begin
1. for $i = n$ to 1 by -1 do
2. compute *parents*(X_i)
3. perform *record-constraint*(X_i, *parents*(X_i))
4. in the constraint graph, connect all unconnected elements in *parents*(X_i)

End

Fig. 5. Algorithm *ADAPT*.

order prescribed, without encountering any deadend. The topology of the new induced graph can be found prior to executing the procedure by recursively connecting in the graph any two parents sharing a common successor .

Consider Fig. 4(a). Variable B is chosen first, and since it has only one parent, D, the algorithm records a unary constraint on D's domain. Variable A is processed next, and a binary constraint is enforced on its two parents, D and C, eliminating all pairs that have no common consistent match in A. This operation may require that an arc be added between C and D. The resulting *induced graph* contains the dashed arc in Fig. 4(b).

Let $W(d)$ be the width of the ordering d, and $W^*(d)$ be the width of the induced graph along this ordering. It can be shown that solving the problem along the ordering d, using *ADAPT* is $O(n \cdot \exp(W^*(d)+1))$ [6].

The directional algorithms *DAC*, *DPC*, and *directional i-consistency* differ from *ADAPT* only in the amount and size of constraint recording performed in Step 3. Namely, instead of recording one constraint among all parents, they record a few smaller constraints on subsets of the parents. Let *level* be a parameter indicating the utmost cardinality of the recorded constraints. The class of algorithms *adaptive(level)* is described in Fig. 6. It uses a procedure, *new-record(level, var, set)*, that records only constraints of size *level* from subsets of *set*.

Adpative(level = 1) reduces to *DAC*, while for *level* = 2 it becomes *DPC*. The graph induced by all these algorithms (excluding the case of *level* = 1, where the graph does not change) has the same structure as the one generated by *ADAPT*. Since *adaptive(level* = $W^*(d)$) is the same as *ADAPT*, it is guaranteed to generate a backtrack-free solution.

```
adaptive(level, X_1, ..., X_n)
Begin
  1.  for i = n to 1 by -1 do
  2.     compute parents(X_i)
  3.     perform new-record(level, X_i, parents(X_i))
  4.     for level >= 2, connect in the graph all elements in parents(X_i)
End

new-record(level, var, set)
Begin
  1. if level <= |set| then
  2.    for every subset S in set, subject to |S| = level do
  3.    record-constraint(var, S)
  4.    end
  5. else do record-constraint(var, set)
End
```

Fig. 6. Procedures *adaptive* and *new-record*.

The complexity of *adaptive(level)* is both time- and space-dominated by the procedure *new-record(level)* which is

$$O\left(\binom{W^*(d)}{level}\cdot(k^{\min\{level,W^*(d)\}})\right),$$

k being the maximal domain size. Clearly, the bound can be tightened if the ordering d results in a smaller $W^*(d)$. However, finding the ordering that has the minimum induced width is an NP-complete problem [1].

3.3. Preordering of variables

It is well known that the ordering of variables, whether static throughout search or dynamic, may have a tremendous impact on the size of the search space explored by backtracking algorithms. Finding an ordering that minimizes the search space is difficult and consequently researchers have concentrated on devising heuristics for variable ordering.

We consider four static orderings. The **minimum width** (*MIN*) heuristic [10] orders the variables from last to first by selecting, at each stage, a node having a minimal degree in the subgraph restricted to all nodes not yet selected. (The degree of a node in a graph is the number of its adjacent nodes.) As its name indicates, the heuristic results in a minimum width ordering. The **maximum degree** (*DEG*) heuristic orders the variables in a decreasing order of their degree in the constraint graph. This heuristic also aims at (but does not guarantee) finding a minimum width ordering. The **maximum cardinality** (*CARD*) ordering selects the first variable arbitrarily, then, at each stage, selects the variable that is connected to the largest set of already selected variables. A **depth-first search** ordering (*DFS*) is generated by a *DFS* traversal of the constraint graph. It can be combined with any of the previous orderings to create a tie-breaking rule. In our experiments, the tie-breaking rule was random.

The best known dynamic ordering is the **dynamic search rearrangement** (*DSR*), which was investigated analytically via average-case analysis in [14, 19, 21], and experimentally in [23, 24, 26]. This heuristic selects as the next variable to be instantiated a variable that has a minimal number of values that are consistent with the current partial solution. Heuristically, the choice of such a variable minimizes the remaining search. Other, more elaborate estimates of the remaining search space were also considered [1, 29].

4. Experimental design

Our experiments were performed at two different locations, **site-1** and **site-2**. They were conducted completely independently using different implementations of algorithms, different test instances, and different algorithm combinations. Overall, 42 algorithm combinations were tested. At site-1, we executed *backtracking* (*BTK*) and *backjumping* (*BJ*) on each problem instance twice, once directly

without any preprocessing and once after preprocessing the network by either *DAC*, *DPC*, or *ADAPT* (8 combinations). We ran each algorithm combination along with each one of the static orderings *DEG*, *CARD*, and *MIN* (24 combinations). Two more runs, of *backtracking* and *backjumping* using *DSR* ordering, were also performed. At site-2 we studied only the effect of ordering strategies, with and without preprocessing by *arc consistency*. In this case we executed *backtracking* and *backjumping* on each problem instance twice, once without any preprocessing and once after preprocessing by *DAC* (4 combinations). Each algorithm combination was tested with the static orderings *CARD*, *MIN*, *DFS*, and with the *DSR* (16 combinations).

Table 1 summarizes the algorithm combinations tested and their corresponding sites. For instance, we see that *DPC-BJ* (i.e., *DPC* followed by *backjumping*) was tested only at site-1, while *BJ* was tested at both site-1 and site-2. Note that an instance-by-instance comparison is feasible only within sites.

The test problems at each site were generated using the same random model, but with different parameters. The generator used four parameters: n, the number of variables; k, the number of values; and p_1 and p_2. The parameter p_1 denotes the probability that a given pair of variables will not be constrained (and thus will not have an arc in the constraint graph), while p_2 is the probability that a given pair of values won't be allowed by a given constraint. In other words, the generator created a random graph uniformly using probability p_1 and then created a relation for each arc in the graph using probability p_2 (p_2 is the probability of a no-good). This random model has been used by others [3, 12, 14].[3] The problem instances on which we reported were not selected uniformly from a given distribution. Our aim was to select a subset of problems that appear to be more difficult. We first determined values for p_1 and p_2 that gave rise to relatively difficult instances, while producing sparse graphs. Then, from those distributions we hand-picked only the harder instances. Difficulty was determined by the number of deadends encountered when running *backtracking* on a *DEG* ordering of the instance; if the number of deadends acceded a certain

Table 1
Algorithms and test combinations (1—site-1, 2—site-2).

	BTK	*BJ*	*DAC BJ*	*DPC BJ*	*ADAPT BJ*	*DAC BTK*	*DPC BTK*	*ADAPT BTK*
DEG	1	1	1	1	1	1	1	1
CARD	1 2	1 2	1 2	1	1	1 2	1	1
MIN	1 2	1 2	1 2	1	1	1 2	1	1
DSR	1 2	1 2	2			2		
DFS	2	2	2			2		

[3] Additional, more recently used random models are discussed in the conclusion.

Table 2
Parameters of problem instances at site-1

	W^*	#	Parameters
10,	1	5	{(80,70) (80,73) (80,73) (80,75) (80,77)}
consistent	2	17	{(65,65) (68,55) (68,58) (68,65) (70,63) (70,65) (70,67) (75,60) (75,63) (75,63) (75,65) (80,60) (80,60) (80,70) (85,70) (85,70) (85,70)}
	3	15	{(63,55) (63,55) (65,55) (65,58) (68,55) (68,58) (68,58) (68,65) (70,60) (70,60) (70,60) (70,63) (70,63) (73,58) (73,58)}
	4	5	{(60,50) (63,50) (63,50) (65,58) (68,50)}
10,	2	5	{(75,65) (80,70) (80,75) (80,75) (80,77)}
inconsistent	3	14	{(65,65) (68,60) (68,65) (70,65) (70,65) (70,67) (70,67) (73,58) (75,60) (75,60) (75,63) (75,65) (80,73) (80,77)}
	4	6	{(65,55) (65,58) (65,55) (65,68) (68,55) (68,60)}
15, consistent	2	11	(79,61) (81,62) (84,62) (84,61) (85,63) (85,66) (85,70) (87,64) (88,72) (88,75) (88,75)
	3	9	(77,56) (78,59) (79,56) (80,59) (80,59) (81,62) (83,61) (85,68) (86,63)
	4	9	(77,56) (78,55) (79,54) (79,58) (80,60) (80,60) (82,60) (82,60) (82,62)
	5	6	(71,48) (73,50) (74,52) (75,50) (77,50) (78,56)
15,	2	3	{(85,63) (85,70) (89,75)}
inconsistent	3	17	{(79,58) (80,59) (82,61) (82,61) (82,61) (83,62) (85,61) (85,63) (85,66) (85,66) (85,66) (85,66) (85,70) (80,63) (87,64) (88,70) (88,75)
	4	16	{(78,57) (78,59) (78,59) (79,55) (79,56) (79,61) (79,61) (82,60) (82,62) (82,62) (82,65) (82,65) (82,65) (83,61) (83,62) (88,72)
	5	3	{(76,56) (77,54) (78,55)}

threshold we retained the problem. Table 2 lists the parameters of all the consistent and inconsistent problem instances we reported at site-1, clustered in groups of common W^* along a *DEG* ordering. The induced widths along a *MIN* or *CARD* orderings were highly correlated. Note that most problem instances come from a narrow range of parameters. Overall, we experimented with two sets of random problems: one containing 66 instances (24 instances were inconsistent), each having 10 variables and 5 values, and the other containing 71 instances (39 instances were inconsistent), each with 15 variables and 5 values. As mentioned, these instances represent the more difficult problems among a much larger set of problems, from which all the easy problems were omitted. Larger problems that have more variables required too much time and space for our machine to handle, because of the *ADAPT* overhead.

We restricted the set of test problems to binary CSPs primarily because problems with constraints of higher order tend to have denser constraint graphs, for which consistency enforcing algorithms have a higher overhead. We should point out, however, that *ADAPT* adds non-binary constraints to the network,

and thus the implementation of *backtracking* and *backjumping* had to accommodate general, non-binary cases.

At site-2 we experimented with two sets of random problems: one consisting of 104 instances (40 instances were consistent), each having 10 variables and 5 values, and the other consisting of 107 instances (23 instances had solutions), each with 15 variables and 7 values. The first group was generated with $p_1 = p_2 = 0.5$, while the second was generated with $p_1 = 0.64$ and $p_2 = 0.55$. This combination of parameters generated relatively difficult instances.

We recorded the number of consistency checks and the number of deadends (or back-trackings) in each run. The number of consistency checks is considered a realistic measure of the overall performance, while the number of deadends is indicative of the size of the search space explored. The implementation of *DSR* at site-2 used an additional data structure in the form of a set of tables. In this case, we added the number of table look-ups to the number of consistency checks.

Each algorithm was run twice on each problem instance; once to find one solution and once to find all solutions. The results were clustered into six groups by problem size (10 or 15 variables), and the following three cases: for finding one solution (called *first*), for finding all solutions (called *all*), and for cases where no solution exists (called *failure*).

5. Experimental results

5.1. Evaluation of consistency preprocessing algorithms

We first report our results at site-1. Our initial goal was to compare the effects of the three preprocessing algorithms, *DAC*, *DPC*, and *ADAPT*, on *backtracking* and *backjumping*. Figs. 7 and 8 present bar-charts displaying the average number of consistency checks, and are classified according to the width of the induced graph W^* when using the *CARD* ordering. The first displays results for 10-variable instances, while the second focuses on 15-variable instances. The results for *DEG* and *MIN* were similar, and the ones for *MIN* are presented in Figs. 9 and 10.[4] Each horizontal pair of graphs presents the average results from a group of instances having the same W^*. The left column contrasts the results of all algorithms. However, because *ADAPT's* performance for large W^* is so out of scale when compared to most other algorithms, we used a different scale for the *backjumping*, *DAC*, and *backtracking* algorithms in the right column. The results reported for *DAC*, *DPC*, and *ADAPT* are for cases where they were complemented by *backjumping*. When using *backtracking*, the same behavior was observed, because following consistency enforcing, most deadends were eliminated.

Since the *backtracking*, *backjumping* and *DAC* algorithms do not show a clear

[4] Full detailed results can be requested from the authors.

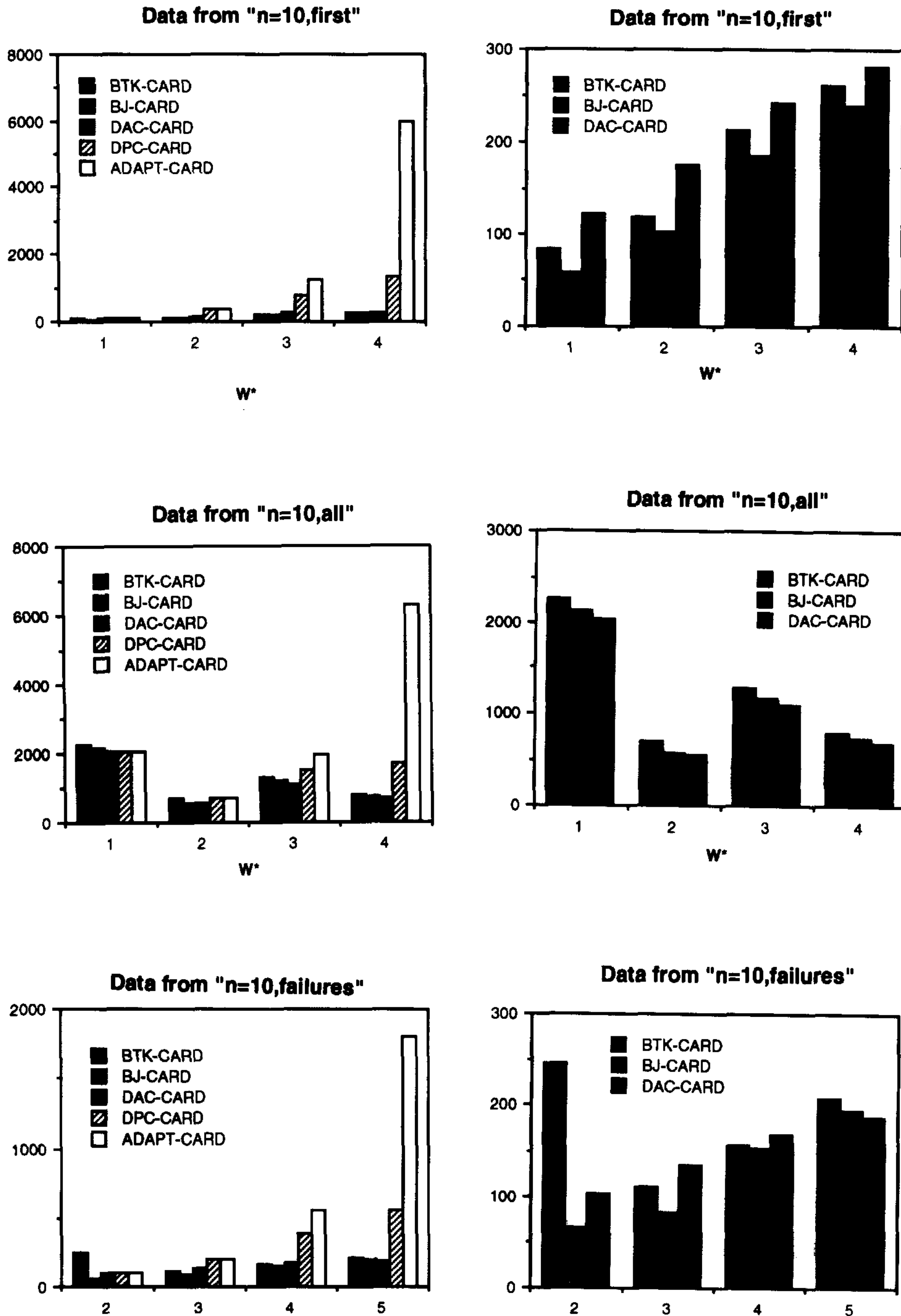


Fig. 7. Number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with the *CARD* ordering on 10-variable random problems.

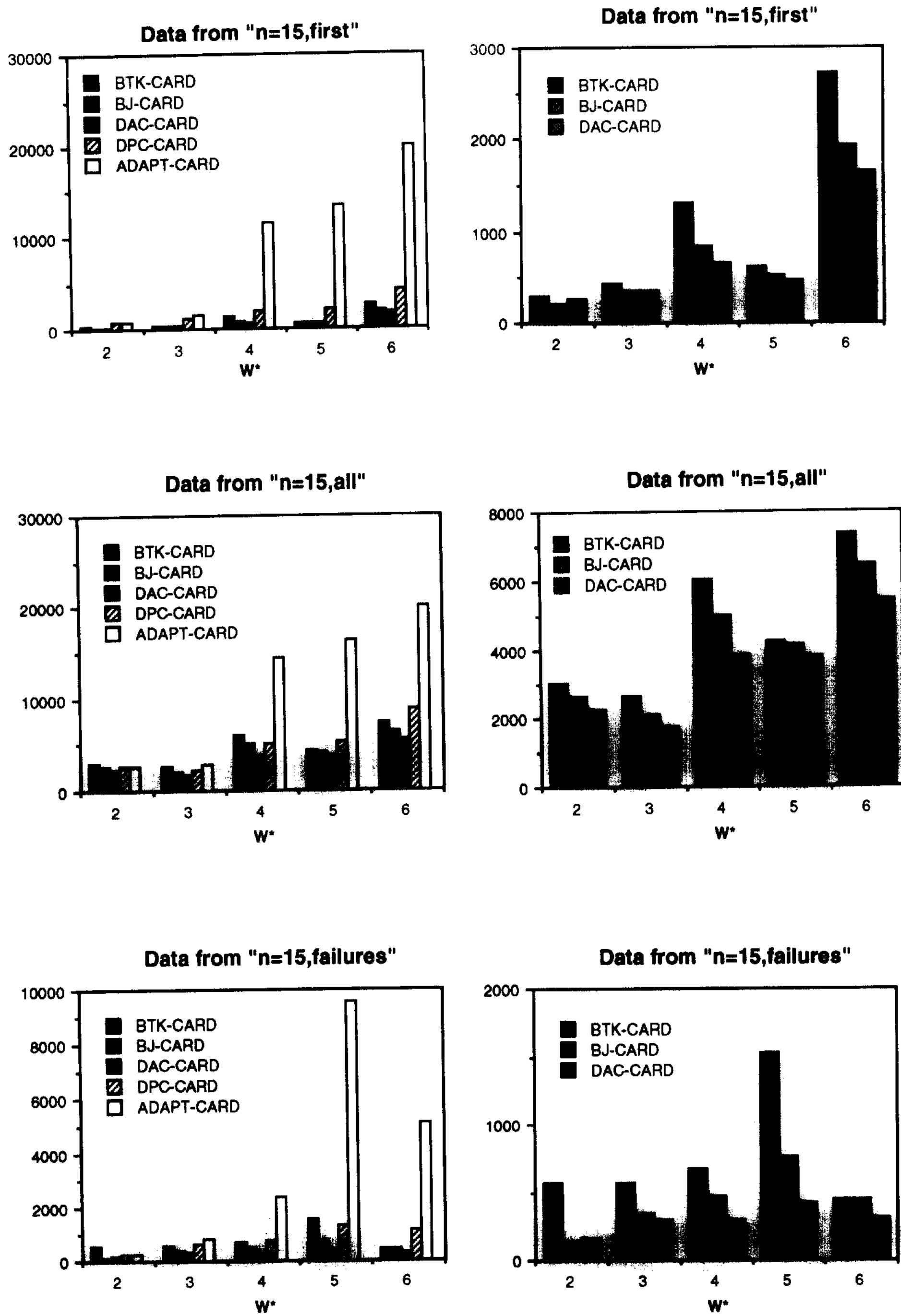


Fig. 8. Number of consistency checks for preprocessing algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with the *CARD* ordering on 15-variable random problems.

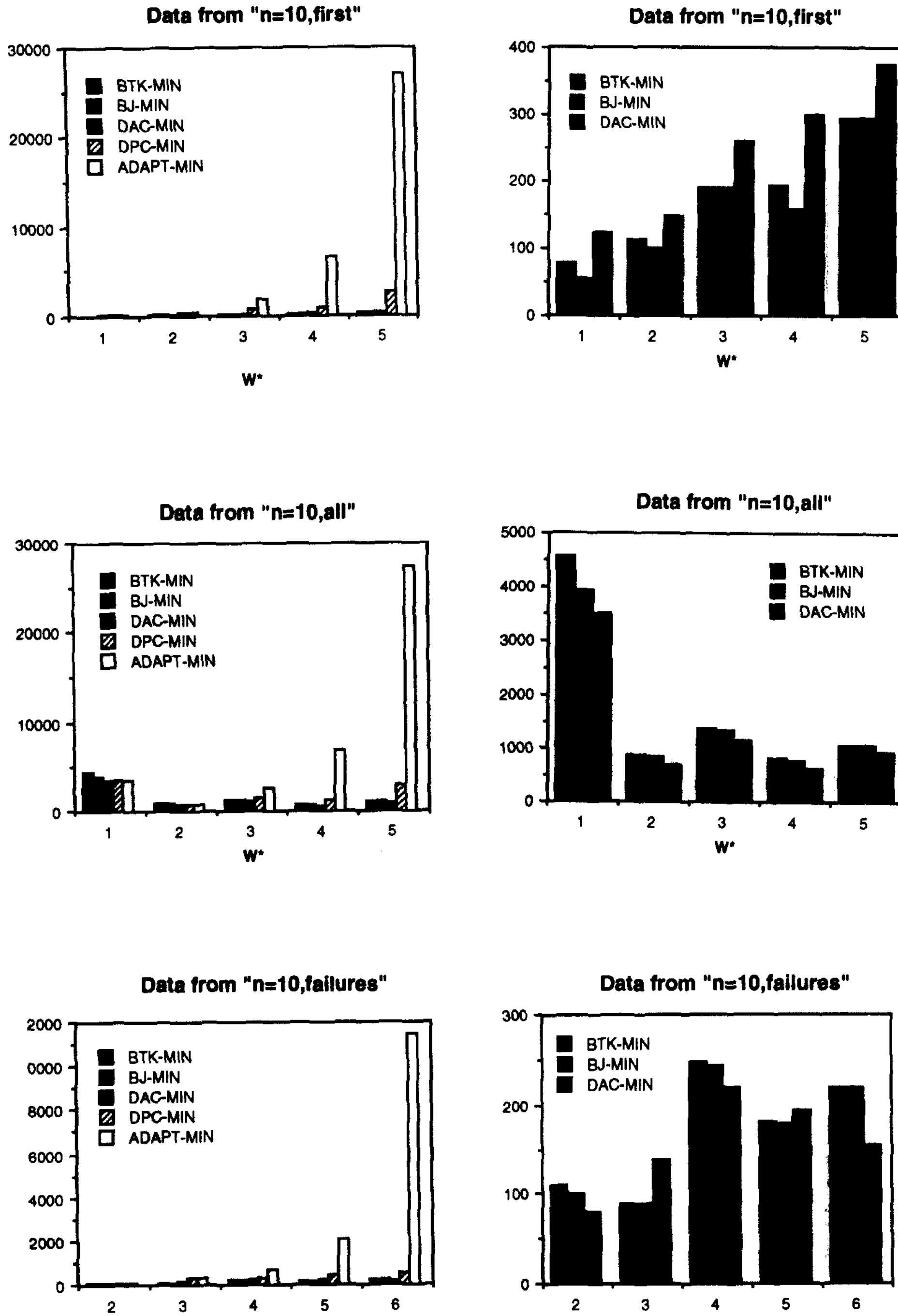


Fig. 9. Number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with the *MIN* ordering on 10-variable, 5-value random problems

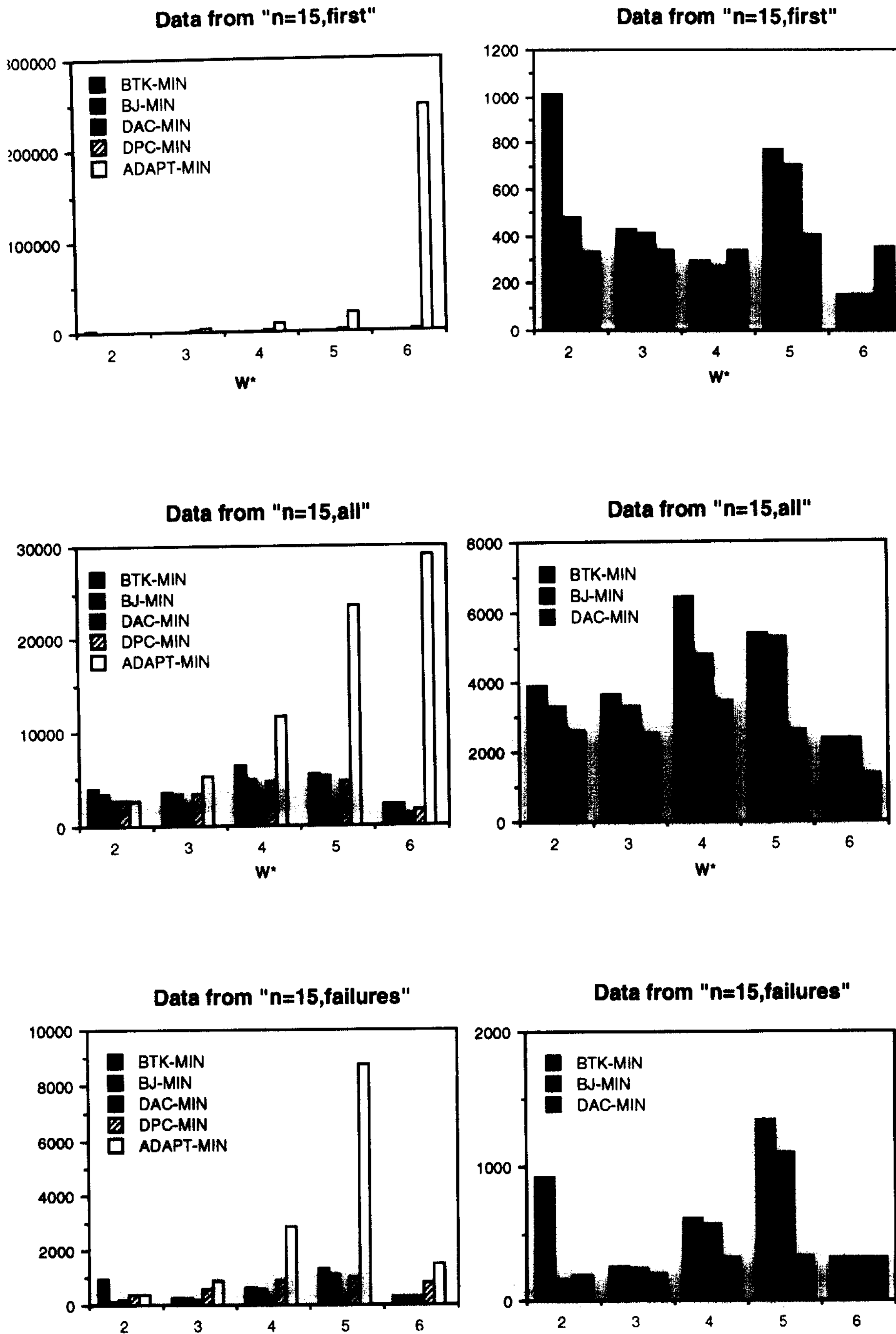


Fig. 10. Number of consistency checks for preprocessing algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with the *MIN* ordering on 15-variable, 5-values random problems

monotonic dependence as a function of W^* (right columns in figures), we presented in Fig. 11 a bar-chart comparing their overall average performances, in the case of *CARD* ordering. The corresponding results for *MIN* and *DEG* were similar and are given in Figs. 12 and 13. In Fig. 14, we separately plotted the performances of all algorithms for small W^* to emphasize the dependence of *ADAPT* and *DPC* on this parameter. (Note that, when $W^* = 2$, *ADAPT* and *DPC* coincide.)

Now we will consider the main relationship observed between the different algorithms as demonstrated by their average performance. In most cases these observations can be backed up by statistical guarantees of a 90% confidence level. In other cases we relied more on our general impression by looking at the detailed data, although we cannot claim that with a high level of confidence.

By looking at the left-hand columns of the graphs of Figs. 7, 8 and 11, we see that simple *backtracking* generally outperforms *ADAPT*. We can also see that even on the average, *ADAPT* has an exponential behavior as a function of W^*. *Backtracking* on the other hand exhibits a mauch more moderate, almost linear behavior. Figs. 7, 8, 14 and 15 suggest that the average performance of *ADAPT* tends to be better than *backtracking* for small values of W^* (for $W^* = 1$ on

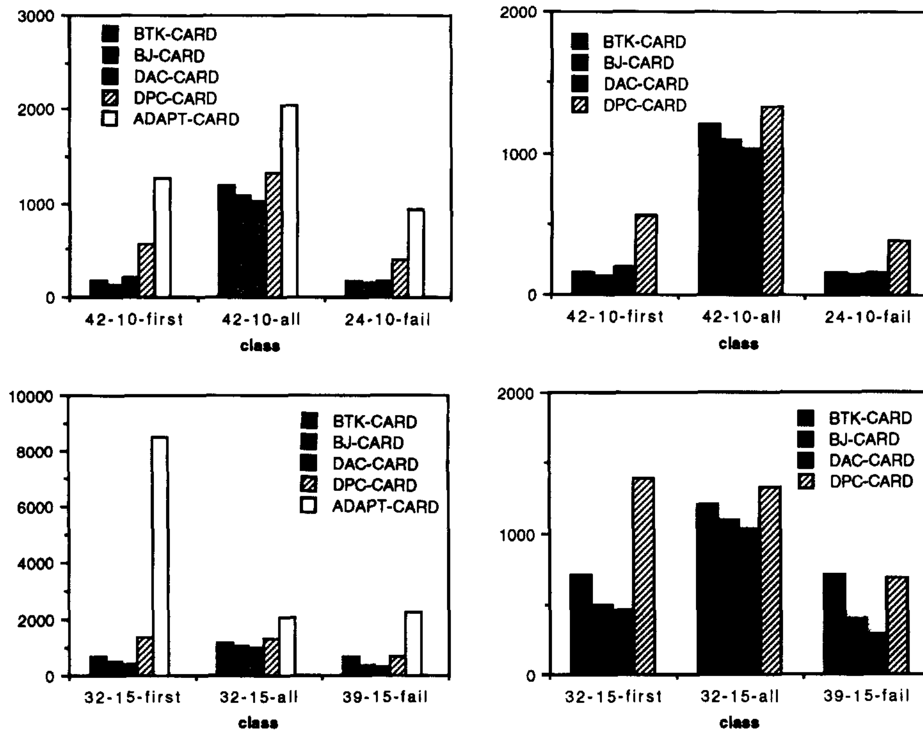


Fig. 11. Number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with the *CARD* ordering on 15-variable random problems, disregarding W^*.

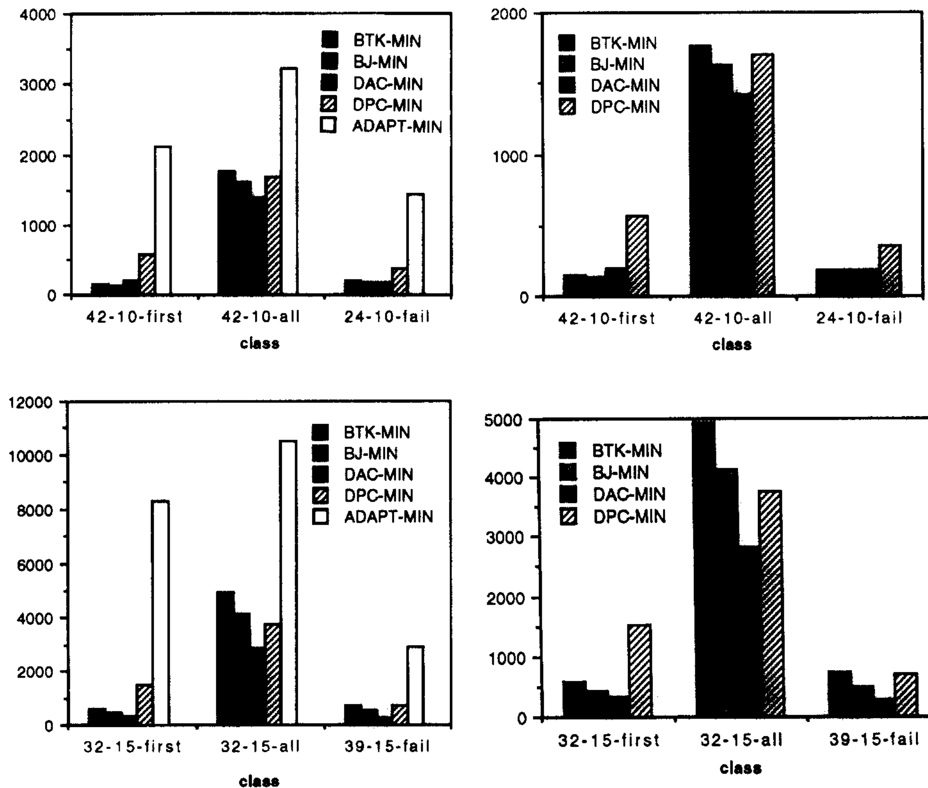


Fig. 12. Number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with the *MIN* ordering, disregarding W^*.

10-variable instances and $W^*=2$ for 15-variable instances). In those cases *ADAPT* becomes *DAC* ($W^*=1$) and *DPC* ($W^*=2$), respectively. W^* values become even more significant for the task of finding all solutions. Evidently, the amount of preprocessing performed by *ADAPT* is generally too heavy to be justified when generating one solution only, but becomes worthwhile when shared by several solutions. When comparing *backtracking* with *DPC* we see that *DPC*'s overhead is in many cases also too high, but we observe some dominance of *DPC* for the task of finding all solutions when W^* is small (Figs. 14 and 15).

When compared to *backjumping*, *ADAPT* seems even worse. It rarely outperformed *backjumping*, and never with more than just a small margin. This data suggests that *backjumping* exploits the structure of the problem more efficiently than *ADAPT* and should be preferred, especially considering that it does not need the additional memory that *ADAPT* consumes. What remains to be tested is how *ADAPT* compares with *backjumping* for larger problems having sparse graphs, when all solutions are required. Algorithm *DPC*, although generally inferior to *backjumping*, sometimes outperformed *backjumping* when

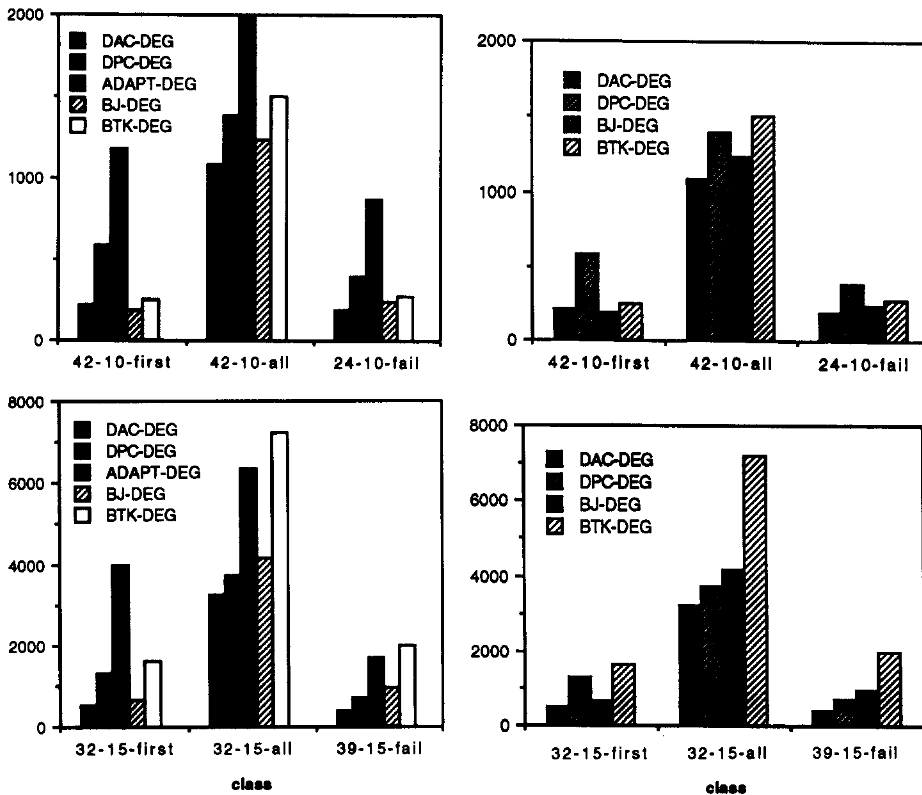


Fig. 13. Number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with the *DEG* ordering, disregarding W^*.

W^* was small (see Fig. 15). *Backjumping* is always at least as good as backtracking as can be guaranteed theoretically.

The disappointing results of *ADAPT* are explained when comparing it with the other two, less ambitious, preprocessing algorithms *DAC* and *DPC*. Almost always, *DAC* is better than *DPC* which is better than *ADAPT*. Indeed, when we counted the number of deadends left after preprocessing (Fig. 16), we found that in many problem instances even *DPC* eliminated all future deadends. It becomes clear, therefore, that for this class of problem instances *ADAPT* is doing unnecessary preprocessing. Moreover, the number of deadends left by *DAC* shows that a substantial portion of the work is accomplished even by this algorithm, which performs the smallest amount of constraint recording.

Although *ADAPT* generally does not seem to be a sensible choice for finding a one time solution, it is still useful for finding a better representation of a network of constraints, such as when the network represents some knowledge base on which many queries are to be answered over time. In such cases, the work for generating the new representation can be ignored [7]. We also observed, as

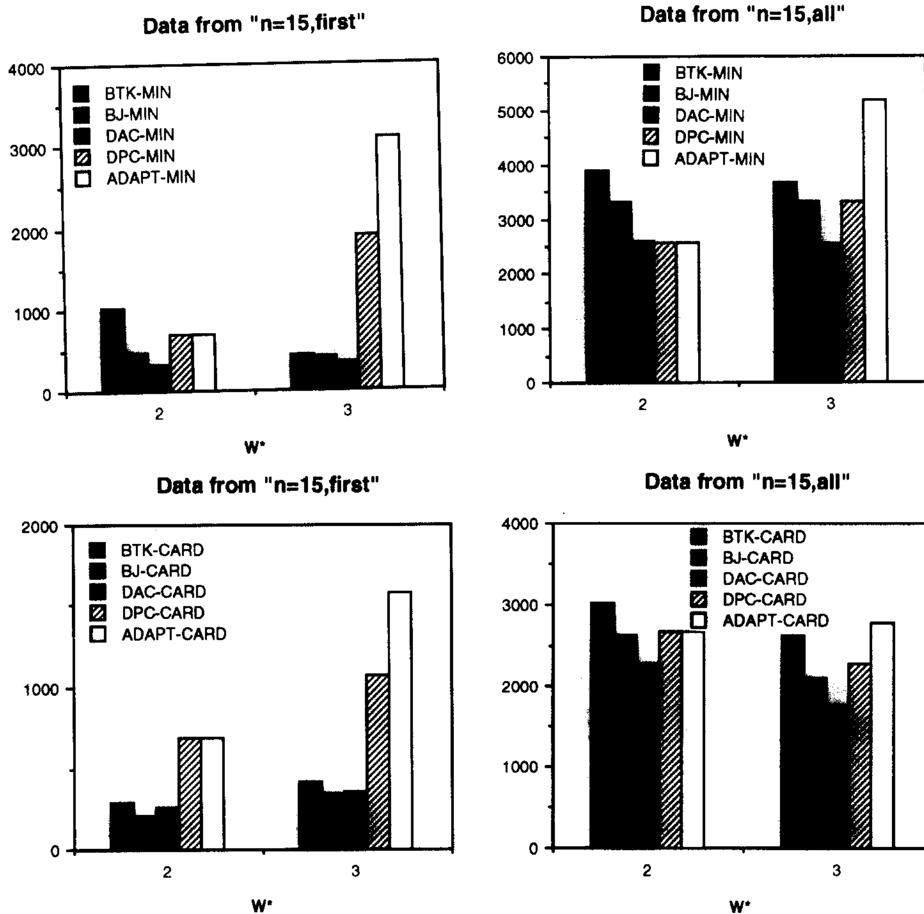


Fig. 14. Number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with *CARD* and *MIN* orderings for small W^*.

dictated by theory, that the amount of processing required to generate one or all solutions is considerably less after using *ADAPT*. (The results are not displayed.)

We turn now to the right-hand columns of Figs. 7–11, which compare *backjumping*, *DAC*, and *backtracking*. Clearly, the two algorithms that stand out in these experiments are *backjumping* and *DAC*. Both outperformed *backtracking* (and *DPC* and *ADAPT*) in almost all cases, but neither dominated the other. When carefully observing their relative performance according to the problem's size, the ordering used, and the task at hand, we see that the *backjumping* algorithm was somewhat better than *DAC* only when finding the first solution and for problems of small size (10 variables), irrespective of the ordering. In the more demanding cases, when the problems were large (15 variables), and for the task of finding all solutions, *DAC* was better. Thus, the charts suggest that for heavy

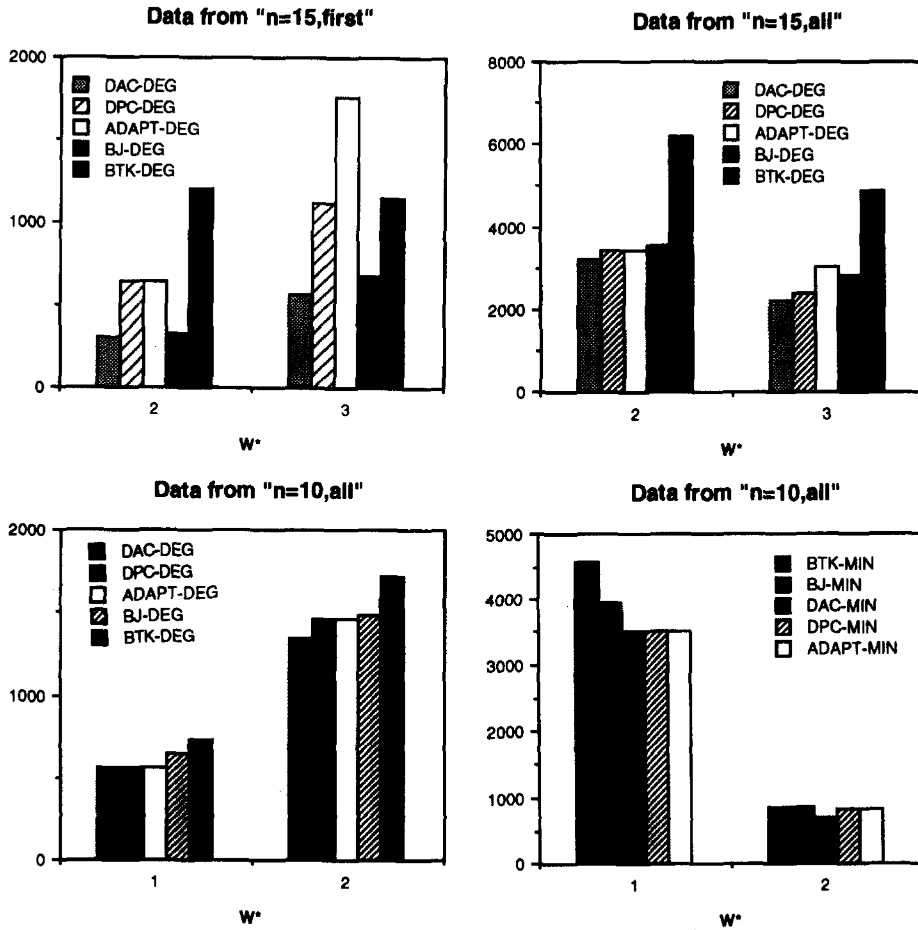


Fig. 15. Number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with the *DEG* and *MIN* orderings for small W^*.

tasks, the overhead presented by *DAC* is outweighed by its gain. This hypothesis requires further testing on larger problem instances.

5.2. Effects of variable ordering

We now focus on characterizing the effect of variable ordering, be it static or dynamic, on the various algorithm combinations, particularly on *backtracking* and *backjumping*. Here we presented results from both sites. Fig. 17 presents results from site-1. It shows the results of running *backtracking* and *backjumping* using the four orderings: *DEG*, *CARD*, *MIN*, and *DSR*. Each graph presents the average number of consistency checks over all instances, disregarding W^*. As

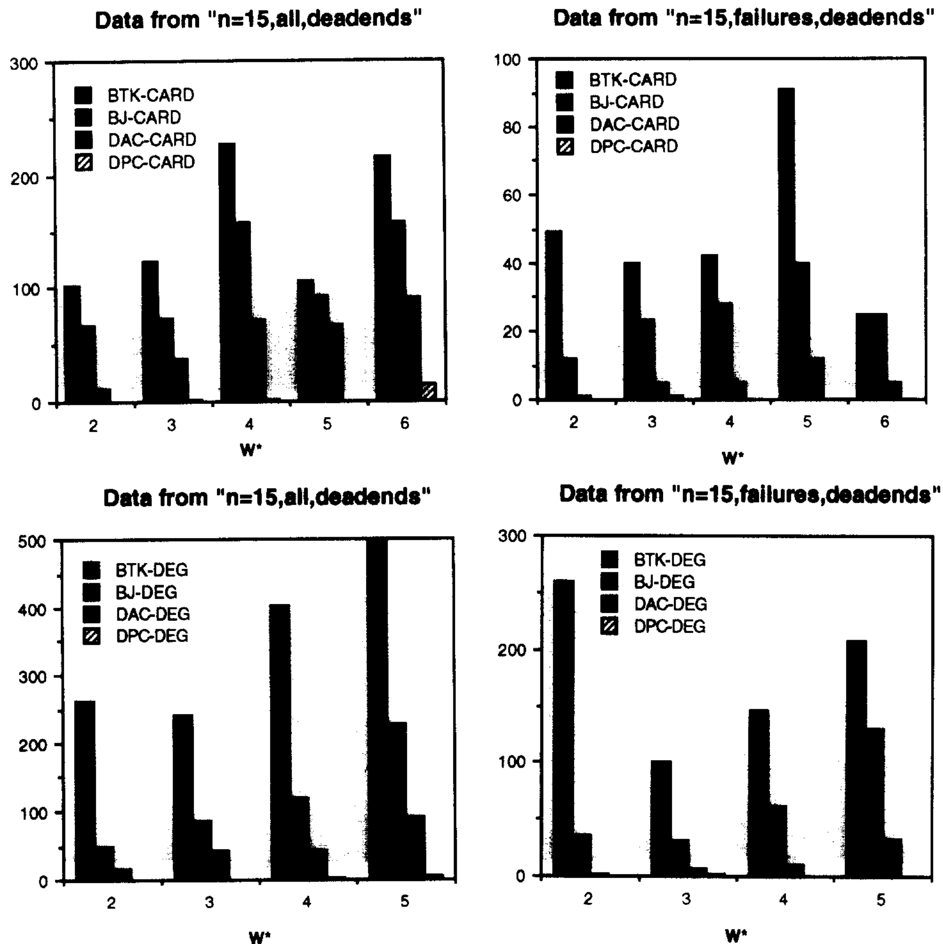


Fig. 16. Number of deadends for preprocessing algorithms (*DPC*, *DAC*, *backjumping* and *backtracking*) with the *CARD* and *DEG* orderings on 15-variable random problems.

before, we grouped the results according to the problem size (10 or 15 variables) and the three cases (*first*, *all*, *failure*).

Fig. 17 shows that the *DEG* ordering almost always comes out a loser (accept for 10-variable instances when looking for all solutions), yet there is no clear winner. Again we observed some patterns in the role of ordering, relative to the task and to the problem size. Specifically, *MIN* was the best ordering for the task of finding the first solution (except for *backtracking* in large problems), *CARD* was the best ordering for finding all solutions, and *DSR* was generally best for *failure* instances (with the exception of *backjumping* in small problems). When we compared the number of deadends associated with each ordering it became clear that *DSR* expands the smallest search space (i.e., it has the least number of deadends on an instance-by-instance basis). Therefore, if we had better im-

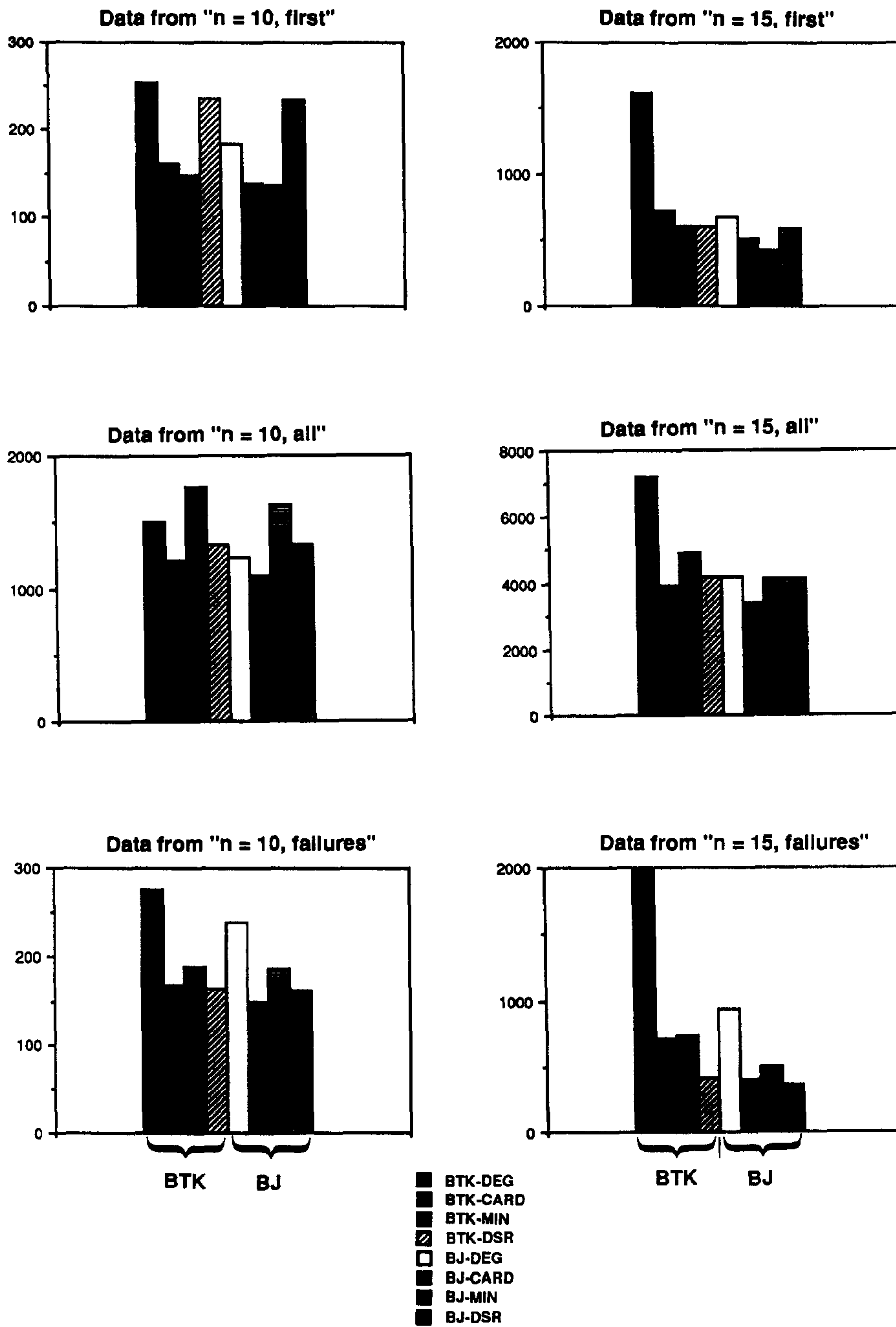


Fig. 17. Number of consistency checks for *backjumping* and *backtracking* in site-1 for 10- and 15-variable problems. *DEG*, *CARD*, *MIN*, and *DSR*.

plemented this technique, we may have had a better overall performance. Indeed, at site-1's implementation of *DSR*, no data structure was used to reduce redundant consistency checks, as is done in other look-ahead schemes such as *forward-checking* [14].

This problem was corrected in our experiments at site-2. Here we compared three static orderings and one dynamic ordering. We used *MIN* and *CARD*, as at site-1, but instead of *DEG* (it had been relatively bad and almost never first at site-1) we used a *DFS* ordering with a random tie-breaking rule. At this site, *DSR* was implemented more efficiently, using data structures similar to those used by Haralick and Elliot [14] which take at most quadratic space. Therefore, when collecting the data, we added the number of table look-ups to the number of consistency checks.

As can be seen from Figs. 18 and 19, in this implementation, *DSR* dominated all other orderings. Contrary to our observations at site-1, we saw some superiority of *MIN* over *CARD* in these instances.

6. Summary and conclusions

We have evaluated the computational benefits of several techniques for preprocessing constraint satisfaction problems. First, we tested the effect of various consistency preprocessings on *backtracking* and *backjumping*, and then we tested the effect of five variable ordering schemes. The conclusions are schematized in Figs. 20 and 21. These graphical representations summarize the relative merits of the algorithms as reflected by the average number of consistency checks. An arrow from A to B indicates that algorithm A is generally superior to algorithm B. Superiority in these graph means that with a confidence level of 0.95 the average performance of the algorithm in the tail of the arrow is less than that of the algorithm in its head, with exceptions annotated on the arrow. The exceptions mean either reverse superiority or an inconclusive relationship. For example, Fig. 20 indicates that *backtracking* outperforms *ADAPT* except when W^* is very small and all solutions are computed. Similarly, *DAC* outperforms *backjumping* except when finding the first solution. This arrow is wigly to indicate that the dominance relationship displayed is relatively weak or inconclusive. Likewise, Fig. 21 presents the relative strength of different orderings with respect to *backtracking* (Fig. 21(a)) and *backjumping* (Fig. 21(b)). The solid arrows show results taken at site-1 while the dotted arrows show results taken at site-2. An inconclusive relationship is denoted by an undirected arc, labeled "?".

Our experiments indicate that in most cases *DAC* followed by *backjumping* outperforms all other schemes, while *DSR* remains the most promising ordering scheme. When static ordering is imposed, the experiments suggest that combining *DAC* with *MIN* or *CARD* yields the best results, on average.

In previous empirical results conducted with the same random model, we observed that weak enhancement schemes were the most effective (due to their low overhead). Stronger schemes did not pay off. For instance, in testing different

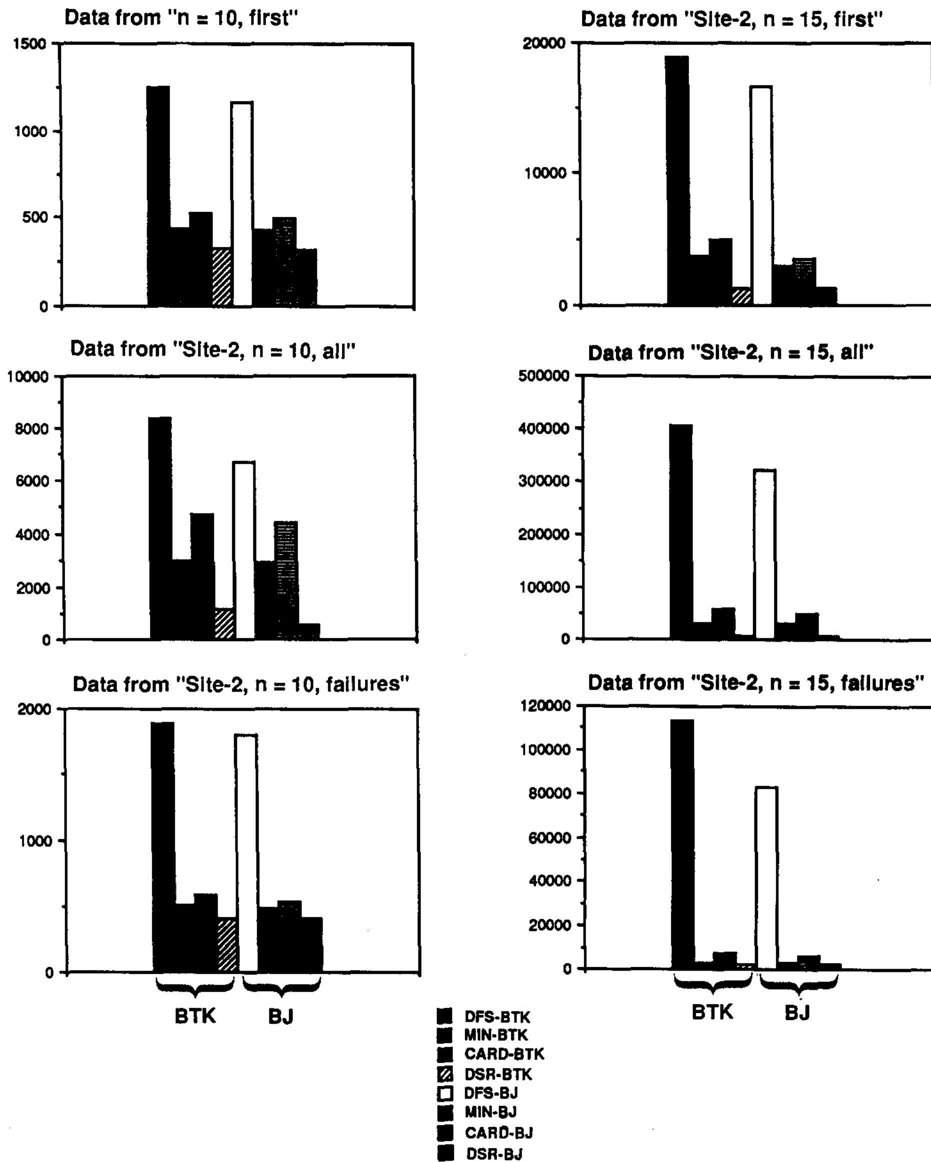


Fig. 18. Number of consistency checks for *backjumping* and *backtracking* on orderings *DEG*, *CARD*, *MIN*, and *DSR* at site-2 for 10- and 15-variable problems.

levels of look-ahead schemes, Haralick and Elliot concluded that *forward-checking*, the least intensive look-ahead mechanism, performed much better than the more intensive partial-look-ahead and full-look-ahead (see [14, Figs. 6, 9, 10, and 11]). The same conclusions are evident in Gaschnig's experiments with *DEEB*, which was his way of incorporating full arc consistency into search (see [12, Figs.

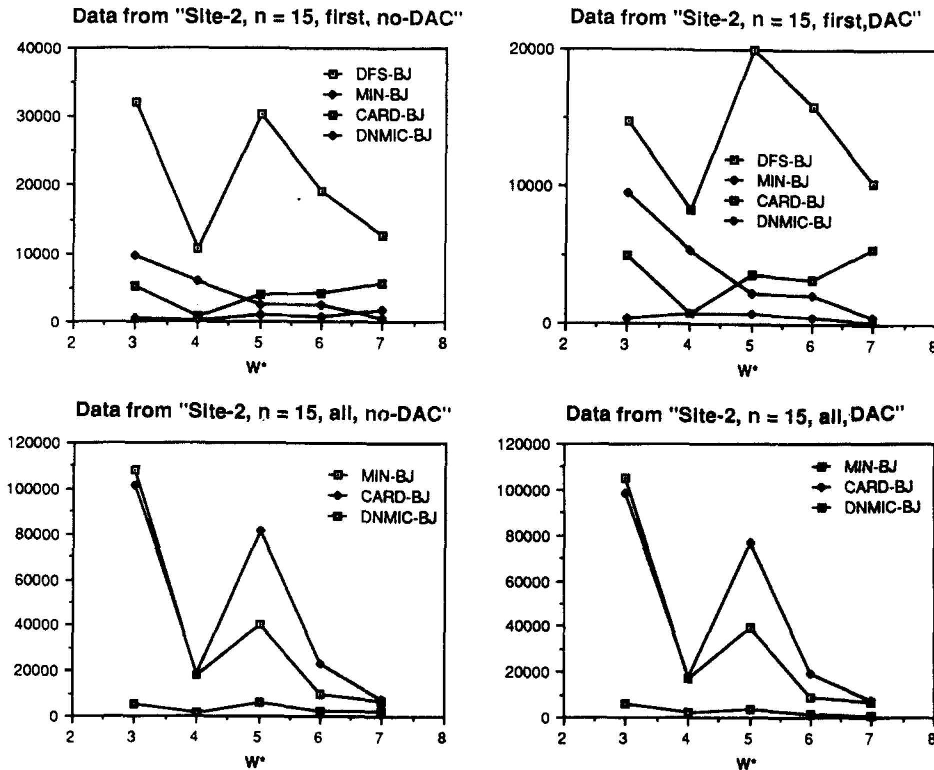


Fig. 19. The effect of variable ordering on the number of consistency checks parameterized by W^* (with and without preprocessing by *DAC*) at site-2.

4.3-1 and 4.4.2-2]). Similarly, when generating heuristics for value selection preferences, only very shallow advice improves the search (see [6, Figs. 15 and 16]). We witnessed the same phenomenon while assessing various look-back schemes. When augmenting *backjumping* with various levels of constraint recording (i.e., learning no-goods parameterized by the size of the constraints and the depth of reasoning), it becomes evident that only very shallow learning of only small constraints is worth undertaking (see [3, Figs. 7 and 8]).

A disclaimer is in order here. Since these experimental comparisons were conducted on small-size problem instances, limited by *ADAPT* time and space complexity, we cannot confidently extrapolate our findings for larger problem instances. Another point is that the random generator we used tends to generate relatively easy instances. It has been shown that the average complexity for solving such problems by *backtracking* is polynomial [14], and it has also been observed recently [11] that a random generator that takes the number of constraints as a parameter (rather than the probability of a constraint) tends to generate much harder instances. Therefore, the comparisons should be continued on additional random models, and larger instances, before a definite conclusion can be reached.

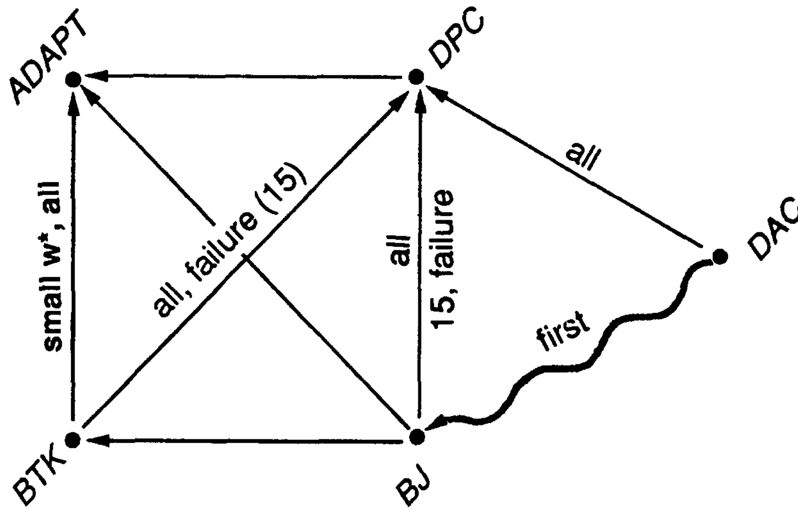


Fig. 20. Relative performance of consistency enforcing algorithms. An arrow from A to B means that algorithm A is generally superior to B with *exceptions* annotated on the arrows.

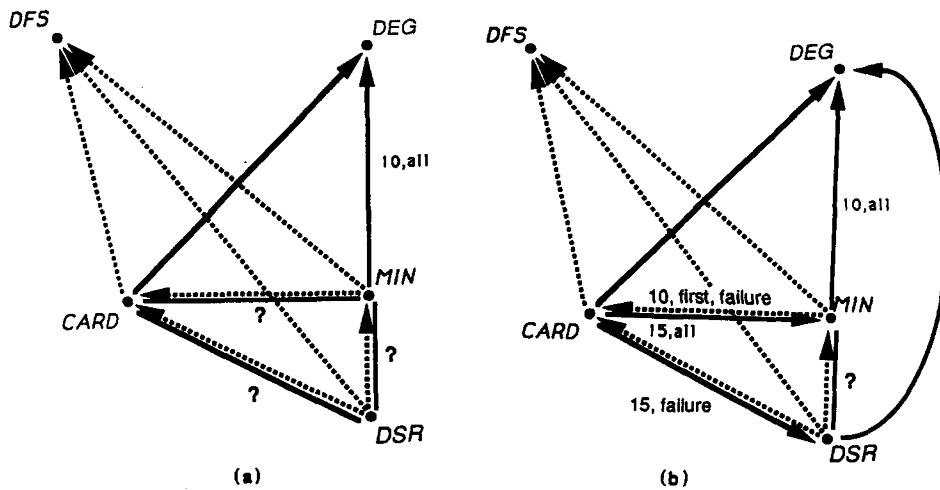


Fig. 21. Relative merits of ordering schemes for (a) *backtracking* and (b) *backjumping.*

It is conceivable that on larger, more difficult instances, intensive preprocessing algorithms may actually pay off. Indeed, we recently observed that for hard 20-variable, 5-value instances, full *path consistency* followed by *backjumping* considerably outperformed *backjumping* [5]. We also observed from our as well as from others' data that the performance variance on problems generated with the same parameter is very large. Thus, performance averages may not be the adequate measure for comparisons. Relative measures such as the *average performance ratio* might be more informative.

Acknowledgments

We would like to thank Tal Sela and Nadav Eran for implementing the techniques at site-2. Thanks also go to Judea Pearl for commenting on the last version of this manuscript. This work was supported in part by the National Science Foundation, Grants #IRI-881552 and IRI-9157636, by GE Corporate of research, by Toshiba of America, by a Xerox grant and by the Air Force Office of Scientific Research, Grant #AFOSR-88-0177.

References

[1] S. Arnborg, D.G. Corneil and A. Proskurowski, Complexity of finding embeddings in a k-tree, *SIAM J. Algebraic Discrete Methods* **8** (2) (1987) 177–184.
[2] Z. Collin, R. Decher and S. Katz, On the feasibility of distributed constraint satisfaction, in: *Proceedings IJCAI-91*, Sidney, Australia (1991) 318–324.
[3] R. Dechter, Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition, *Artif. Intell.* **41** (3) (1990) 273–312.
[4] R. Dechter, Constraint networks, in: S. Shapiro, ed., *Encyclopedia of Artificial Intelligence* (Wiley, New York, 2nd ed., 1992) 276–285.
[5] R. Dechter and K. Kask, Combining GSAT and path consistency for solving constraint satisfaction problems, 94-17, Information and Computer Science Department, University of California, Irvine, CA (1994).
[6] R. Dechter and J. Pearl, Network-based heuristics for constraint-satisfaction problems, *Artif. Intell.* **34** (1) (1987) 1–38.
[7] R. Dechter and J. Pearl, Tree clustering for constraint networks, *Artif. Intell.* **38** (1989) 353–366.
[8] S. Even, *Graph Algorithms* (Computer Science Press, Potomac, MD, 1979).
[9] E.C. Freuder, Synthesizing constraint expression, *Commun. ACM* **21** (11) (1978) 958–965.
[10] E.C. Freuder, A sufficient condition for backtrack-free search, *J. ACM* **29** (1) (1982) 24–32.
[11] D. Frost and R. Dechter, Dead-end driven learning, in: *Proceedings AAAI-94*, Seattle, WA (1994).
[12] J. Gaschnig, Performance measurement and analysis of certain search algorithms, Tech. Report CMU-CS-124, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA (1979).
[13] M. Ginsberg, Search lesson learned from crossword puzzles, in: *Proceedings AAAI-90*, Boston, MA (1990) 210–215.
[14] R.M. Haralick and G.L. Elliott, Increasing tree-search efficiency for constraint satisfaction problems, *Artif. Intell.* **14** (3) (1980) 263–313.
[15] A.K. Mackworth, Constraint satisfaction, in: S. Shapiro, ed., *Encyclopedia of Artificial Intelligence* (Wiley, New York, 2nd ed., 1992) 285–293.
[16] A.K. Mackworth, Consistency in networks of relations, *Artif. Intell.* **8** (1) (1977) 99–118.
[17] D. Mitchell, B. Selman and H.J. Levesque, Hard and easy distributions of SAT problems, in: *Proceedings AAAI-92*, San Jose, CA (1992) 459–465.
[18] U. Montanari, Networks of constraints: fundamental properties and applications to picture processing, *Inf. Sci.* **7** (2) (1974) 95–132.
[19] B. Nudel, Consistent-labeling problems and their algorithms: expected-complexities and theory-based heuristics, *Artif. Intell.* **21** (1–2) (1983) 135–178.
[20] P. Prosser, Hybrid algorithms for the constraint satisfaction problem, *Comput. Intell.* **9** (3) (1993) 268–299.
[21] P. Purdom, Search rearrangement backtracking and polynomial average time, *Artif. Intell.* **21** (1–2) (1983) 117–133.

[22] P.W. Purdom, E.L. Robertson and C.A. Brown, Backtracking with multi-level dynamic search rearrangement, *Acta. Inf.* **15** (2) (1981) 99–114.
[23] W. Rosiers and M. Bruynooghe, Empirical study of some constraint satisfaction algorithms, Tech. Report CW 50, Katolieke Universiteit Leuven, Leuven, Belgium (1986).
[24] N. Sadeh and M. Fox, Variable and value ordering heuristics for hard constraint satisfaction problems: an application to job shop scheduling, Tech. Report. CMU-RI-TR-91-23, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1992).
[25] R.M. Stallman and G.J. Sussman, Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artif. Intell.* **9** (2) (1977) 135–196.
[26] H.S. Stone and J.M. Stone, Efficient search techniques—an empirical study of the *N*-Queens problem, Tech. Report RC 12057 (#54343), IBM T.J. Watson Research Center, Yorktown Heights, NY (1986).
[27] P. Van Hentenryck and M. Dincbas, Forward checking in logic programming, in: J.-L. Lassez, ed., *Proceedings Fourth International Conference on Logic Programming* (MIT Press, Cambridge, MA, 1987) 229–255.
[28] D. Waltz, Understanding line drawings of scenes with shadows, in: P.H. Winston, ed., *The Psychology of Computer Vision* (McGraw-Hill, New York, 1975).
[29] R. Zabih and D. McAllester, A rearrangement search strategy for determining propositional satisfiability, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 155–160.