Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

EXPERIMENTAL EVALUATION OF CONSTRAINT PROCESSING

R. Dechter
I. Meiri

January 1992
CSD-920015

Technical Report
R-174
January 1992

EXPERIMENTAL EVALUATION OF CONSTRAINT PROCESSING*

Rina Dechter

Information and Computer Science Department
University of California, Irvine,
Irvine, CA, 29717.

Itay Meiri

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

Abstract

This paper presents an experimental evaluation of two orthogonal schemes for solving constraint satisfaction problems (CSPs). The first scheme involves a class of pre-processing techniques including *directional arc consistency*, *directional path consistency* and *adaptive consistency*. The second is concerned with the order in which variables are chosen for evaluation during the search. In the first part of the experiments we measured the performance of *backtracking* and its common enhancement -- *backjumping* with and without each of the pre-processing techniques above. The results show that *directional arc-consistency*, and *backjumping* (without any preprocessing) outperform all other preprocessing techniques; the first dominated the second in computational intensive situations. The second part of the experiments suggest that *dynamic search rearrangement*, is the best ordering while among the static orderings both *max-cardinality* and *min-width* yield the the best pre-ordering.

*This work was supported in part by the National Science Foundation, Grants #IRI-8815522 and IRI-9157636 and by the Air Force Office of Scientific Research, Grant #AFOSR-88-0177.

1. Introduction

Constraint Satisfaction Problems (CSPs), belong to the class of NP-complete problems. Such problems normally suffer from lack of realistic measure of performance; worse-case analysis fails to reveal average-case behavior. It is either not refined enough to distinguish between different algorithms or, since based on extreme cases, it may give an erroneous view of typical performance. Hence, it fails to identify those algorithms that are good in practice. Consequently, theoretical analysis must be supplemented by experimental studies.

The most thorough experimental studies so far include Gaschnig comparisons of *backjumping*, *backmarking* and *constraint propagation*, [Gaschnig 1979] Haralick and Elliot's study of look-ahead strategies [Haralick 1980] and, more recently, Dechter's experiments with structure-based techniques, [Dechter 1990] and Prosser's hybrid tests with *backjumping* and *forward checking* strategies [Prosser 1991]. Additional studies were reported in [Dechter 1987, Stone 1986, Rosiers 1986, Ginsberg 1990].

Idealy, experimental studies are most informative when conducted on a "representative" set of problems from one's domain of application. This, however, is a very difficult task. Real life problems are often too large or ill defined to suit a laboratory manipulation. Thus, a common compromise is to use either randomly-generated problems or canonical examples that received attention (e.g., *n*-queens, cross-word 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. They do reveal, however, the problem's parameters that effect the relative usefulness of various algorithms.

The experiments reported in this paper were performed in two different sites and can be categorized into two classes. The first concerns **pre-processing** algorithms, which transform a given constraint network into a more explicit representation. The three pre-processing algorithms tested are *directional arc consistency (DAC)*, *directional path consistency (DPC)*, and *adaptive consistency (ADAPT)* [Dechter 1987]. These algorithms respectively represent increasing levels of pre-processing effort. The second category tests the effect of various ordering strategies on *backtracking* and *backjumping*. We tested four **static** heuristic orderings: *min-width*, *max-degree*, *max-cardinality*, *depth first search (DFS)*, and one **dynamic** ordering, *dynamic search rearrangement* [Purdom 1983].

The results show upfront, that, contrary to predictions based on worse-case analysis, the average complexity of backtracking on randomly-generated problems is far from exponential; it is rather almost linear. This has a prominent effect on the benefit generated by pre-processing. Indeed, the pre-processing performed by the most aggressive scheme, *ADAPT*, did not pay off, except in very sparse graphs, in spite of its theoretical superiority over *backtracking*. On the other hand, the least aggressive scheme, *DAC*, came out as a winner in computationally intensive cases. Apparently, *DAC* performs just the right amount of pre-processing. As to ordering strategies, *dynamic* ordering outperformed all static orderings. It apparently yields the smallest search space with a reasonable amount of overhead. Among the static orderings, *min-width* and *max-card* clearly dominated the *max-degree* and *DFS* orderings, however, the exact relationship between the first two is still inconclusive.

Viewing our results in the context of earlier experimental evaluation, we see a common pattern. In practice *Backtracking*, is not as bad as its worse-case bound indicates, and significant improvement can be realized with only minor help. More intense help would be ineffective yielding an overall worse performance.

The paper is organized as follows. Section 2 presents the constraint network model and general background. Section 3 presents the tested algorithms. Sections 4 describes the experimental design, while Section 5 presents the results. Section 6 provides a summary and concluding remarks.

2. Constraint Processing Techniques -- Background

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]. Formally, a constraint, C_i, has two parts: 1). a subset of variables, $S_i = \{X_{i_1}, \ldots, X_{i_{j(i)}}\}$, on which it is defined, called a **constraint-subset**, and 2). a **relation**, rel_i, defined over S_i: $rel_i \subseteq D_{i_1} \times \cdots \times D_{i_{j(i)}}$. The **scheme** of a *CN* is the set of its constraint subsets, namely, $scheme(CN) = \{S_1, S_2, \ldots, S_t\}$, $S_i \subseteq X$. The projection of a relation, ρ, on a subset of variables, $U = U_1, \ldots, U_l$, is given by:

$$\Pi_U(\rho) = \{x_u = (x_{u_1}, \ldots, x_{u_l}) \mid \exists \; \bar{x} \in \rho, \bar{x} \text{ is an extension of } x_u\}.$$

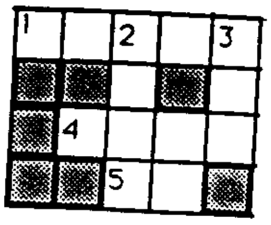
An assignment of a unique domain value, to each member of some subset of variables is called an **instantiation**. An instantiation is called a **solution** if it satisfies *all* the constraints. The set of all solutions is a relation, ρ, defined on the set of all variables. Formally,

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

A *CN* may be associated with a constraint graph, where nodes represent variables and arcs connect variables that appear in the same constraint. For example, the *CN* depicted in figure 1a presents a cross-words puzzle. The variables are *E* (1, horizontal), *D* (2, vertical), *C* (3, vertical), *A* (4, horizontal), and *B* (5, horizontal). The scheme is {*ED*, *EC*, *CA*, *AD*, *DB* } and the constraint graph is given in figure 1b.

Typical tasks defined on a *CN* are to determine whether a solution exists, to find one solution or the set of all solutions, and to determine whether an instantiation of a subset of variables is part of a global solution. These tasks are collectively called **Constraint Satisfaction**

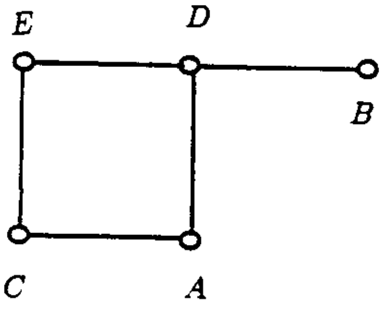
[1] This does not mean that the actual representation of any constraint is necessarily in the form of its defining relation, rather, the relation can in principle be generated using the constraint's specification without the need to consult other constraints in the network.



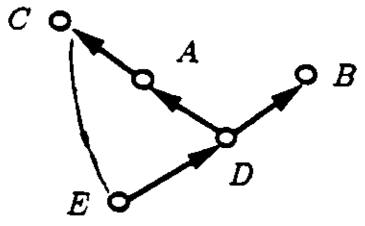
$D_A = \{$`hoses,laser,sheet,snail,steer`$\}$
$D_B = \{$`hike,aron,keet,earn,same`$\}$
$D_C = \{$`run,sun,let,yes,eat,ten`$\}$
$D_D = \{$`no,be,us,it`$\}$

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

(a)



(b)



(c)

Figure 1: A cross word puzzle (a), its *CN* representation (b), and a *DFS* pre-ordering (c).

Problems (CSPs).

Techniques used in processing constraint networks can be classified into three categories. 1. Search techniques for systematic exploration of the space of all solutions. The basis for all search algorithms is **backtracking**. 2. **Consistency algorithms**, for transforming a *CN* into a more explicit representation. These algorithms are used either in a preprocessing phase to improve the performance of the subsequent backtracking, or they can be incorporated into the search procedure itself. 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 [Mackworth 1991, Dechter 1991].

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 it reaches a **dead-end**, 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 worse-case running time of *backtracking* is exponential.

Improving the efficiency of *backtracking* amounts to reducing the size of the search space it expands. Researchers have developed two types of procedures: pre-processing algorithms that are employed **prior to** performing the search, and algorithms that are used dynamically **during** search.

The pre-processing algorithms include a variety of **consistency enforcing** algorithms [Montanari 1974, Mackworth 1977, Freuder 1978]. These algorithms transform a given constraint network into an equivalent, yet more explicit form, by deducing new constraints to be added to the network. In a nutshell, 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 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 i-**th** variable. Algorithms, *directional arc consistency (DAC)*, *directional path consistency (DPC)*, and *adaptive consistency (ADAPT)* are all restricted versions of these consistency-enforcing algorithms that take into account the direction in which backtracking instantiates the variables.

Ordering the variables prior to search is another class of pre-processing algorithms that has attracted much attention. Several heuristics for **static orderings** have been proposed [Freuder 1982, Dechter 1989]. Heuristics, such as *min-width*, *max-cardinality*, *max-degree* and *DFS* ordering, most follow the intuition that tightly-constrained variables should be instantiated first.

Strategies that **dynamically** improve the pruning power of backtracking can be classified as **look-ahead schemes** and **look-back schemes.**

Look-ahead schemes are invoked whenever the algorithm is about to assign a value to the next variable. Some schemes, like *forward-checking*, use constraint propagation [Waltz 1975, Haralick 1980] to predict the way in which the current instantiation restricts future assignments of values to variables. Other schemes, like *dynamic search rearrangement*, decide which variable to instantiate next [Freuder 1982, Purdom 1983, Stone 1986].

Look-back schemes are invoked when the algorithm encounters a dead-end and prepares to backtrack. There are two main issues here. 1. Decide how far to backtrack. By analyzing the reasons for the dead-end it is often possible to go back directly to the source of failure instead of to the immediate predecessor in the ordering. This idea is often referred to as **backjumping** [Gaschnig 1979]. 2. Record 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 **no-good** constraints). **Dependency-directed backtracking** incorporates both backjumping and no-goods recording [Stallman 1977].

Structure-based techniques, such as *backjumping*, *directional i consistency* and *adaptive consistency* can be viewed as structure-based improvements of some of the above techniques [Dechter 1991].

3. The tested algorithms

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

3.1 Backtracking and Backjumping

Backtracking systematically assigns values to variables until either a solution is found or there is a dead-end. It then *goes back* to the most recent instantiation, changes it and continues. A backtracking algorithm for finding one solution is given in figure 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 dead-end situations. The procedures maintain lists of candidate values, C_i, for each variable, X_i. Their initial values is computed by "*compute-candidates*$(x_1, \ldots, x_i, X_{i+1})$" that selects all values in the domain of variable X_{i+1} which are consistent with previous assignments. Backtracking starts by calling *forward* with $i=0$, namely, the instantiated list is empty.

```
forward ( $x_1, \ldots, x_i$ )
Begin
  1. if $i = n$ exit with the current assignment.
  2. $C_{i+1}$ <-- compute-candidates($x_1, \ldots, x_i, X_{i+1}$)
  3. if $C_{i+1}$ is not empty then
  4.   $x_{i+1}$ <-- first element in $C_{i+1}$, and
  5.   remove $x_{i+1}$ from $C_{i+1}$, and
  6.   forward($x_1, \ldots, x_i, x_{i+1}$)
  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.   $x_i$ <-- 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
```

Figure 2: Algorithm *backtracking*

Backjumping improves the ''go-back'' phase of backtracking and whenever a dead-end 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*, [Gaschnig 1979] which extracts knowledge about dependencies from the constraint graph alone. It was shown that graph-based *backjumping* outperforms *backtracking* on an instance by instance basis. [Dechter 1990] For a comparison between *backjumping* and graph-based *backjumping* see [Prosser 1991].

In our implementation of graph-based backjumping, both *forward* and *jump-back* (the *go-back* variant of backjumping) carry a parameter P, storing the set of variables that need to be consulted upon the next dead-end. Accordingly, lines 6 and 8 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 Figure 3. Its parameters are the partial instantiation, $x_1, \ldots, x_i$, the dead-end variable, X_{i+1} and P.

```
jump-back($x_1,...,x_i,X_{i+1},P$)
Begin
 1. if $i$=0, exit. no solution exists.
 2. PARENTS <- 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 $C_j \neq \Phi$ 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.
```

Figure 3: Procedure *jump−back*

Consider, for instance, the ordered constraint graph in Figure 1a. If the search is performed in the order E,D,A,C,B and a dead-end 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 a careful maintenance of each variable's parents 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 [Even 1979], finding the jump-back destination amounts to following a very simple rule: if a dead-end occurred at variable X, go back to the parent of X in the *DFS* tree. A *DFS* tree of the problem in figure 1b is given in figure 1c. The DFS numbering (which amounts to an inorder traversal of this tree) is (E,D,B,A,C). Again, if a dead-end 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, m, being the depth of the *DFS*-tree.

3.2 Pre-processing algorithms

Deciding the consistency level that should be enforced on the network is not a clear cut choice. Generally speaking, *backtracking* will benefit from representations which are as explicit as possible, having higher consistency level. However, the complexity of enforcing i-consistency is exponential in i. Thus, there is a trade-off between the effort spent on pre-processing and that spent on search. Indeed, the primary goal of our paper is to explicate this trade-off.

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

We start with *adaptive-consistency*, then generalize its underlying principle to describe a class of pre-processing algorithms which contain 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* which precede it 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 width of all its orderings. For instance, given the ordering (*E,D,C,A,B*) in Figure 4a, the width of node *B* is 1, while the width of this ordering is 2, and so is the width of this graph. Algorithm *adaptive consistency (ADAPT)*, shown in Figure 5, processes the nodes in a reverse order, i.e., each node is processed before any of its parents.

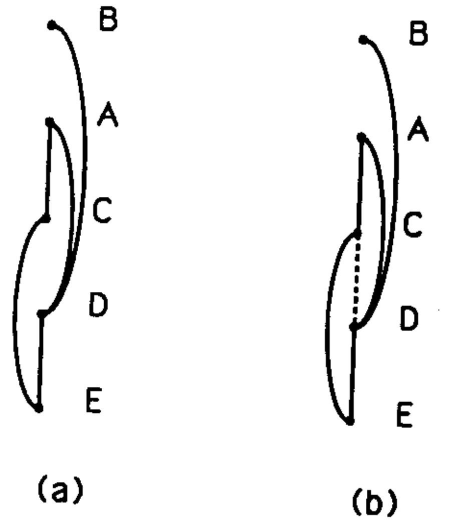


Figure 4: Ordered constraint graphs

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

Begin
1. for i=n to 1 by -1 do
2. Compute $parents(X_i)$
3. perform $record\text{-}constraint(X_i, parents(X_i))$
4. connect all elements in $parents(X_i)$ (if they are not yet connected)
End
```

Figure 5: Algorithm *adaptive consistency*

The procedure *record-constraint(V,SET)* generates and records those tuples of variables in *SET* that are consistent with at least one value of V. For instance if, in our example, A has only one feasible word, *aron*, in its domain, and if 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 clusters of variables. It was shown [Dechter 1987], that when the procedure terminates *backtracking* can solve the problem, in the order prescribed, without encountering any dead-end. The topology of the new **induced graph** can be found prior to executing the procedure, by recursively connecting any two parents sharing a common successor (see Figure 4b).

Consider our example of Figure 4a. 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 which have no common consistent match in A. This operation may require that an arc be added between C and D, and so on. The resulting *induced graph* contains the dashed arc in Figure 4b.

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 is $O(n \cdot \exp(W^*(d)+1))$ [Dechter 1987].

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 recoreded constraints. The class of algorithms *adaptive* (*level*) is described in Figure 6. It uses a procedure, *new−record(level, var, set)*, that records only constraints of size *level* from subsets of *set*.

```
adaptive(level, $X_1, \ldots, X_n$)
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 $level \geq 2$, connect all elements in $parents(X_i)$ (if they are not yet connected)
End

new-record(level, var, set)
Begin
1. if $level \leq |set|$ then
2.   for every subset $S$ in set, s.t $|S| = level$ do
3.      record−constraint(var,S)
4.   end
5.  else do record−constraint (var,set)
end
```

Figure 6: Procedures *adaptive* and *new−record*

Adaptive(*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 doesn't change), has the same structure as the one generated by *adaptive consistency*. Since *adaptive*(*level* = $W^*(d)$) is the same as *adaptive consistency*, it is guaranteed to generate a backtrack-free solution.

The complexity of *adaptive* (*level*) is both time and space dominated by the procedure *new−record*(*level*) which is $O(\binom{W^*(d)}{level} \cdot (k^{\min(level, W^*(d))}))$. This bound can be tightened if the ordering d results in a smaller $W^*(d)$. However, finding the ordering which has the minimum induced width is an NP-complete problem [Arnborg 1987].

3.3 Variable Ordering

It is well known that the ordering of variables, be it **static** throughout search, or **dynamic**, may have a tremendous influence on the size of the search space explored by backtracking algorithms. Finding an ordering which would minimize the search space is a difficult problem and, consequently, researchers have concentrated on devising heuristics for variable ordering.

The best known dynamic ordering is the *dynamic search rearrangement*, which was investigated analytically via average-case analysis in [Purdom 1983, Haralick 1980, Nudel 1983], and experimentally in [Stone 1986, Rosiers 1986]. This heuristic selects as the next variable to be instantiated a variable that has a minimal number of values which are consistent with the current partial solution. Heuristically, the choice of such variable minimizes the remaining search. Other, more elaborate estimates of the remaining search space were also considered [Purdom 1981, Zabih 1988].

We consider four static orderings. The **minimum width** (*min−width*) heuristic [Freuder 1982], orders the variables from last to first by selecting, at each stage, a node having a minimal degree in the subgraph induced by all nodes which have not been selected yet. As its name indicates, the heuristic results in a minimum width ordering.

The **maximum degree** (*max−degree*) 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** (*max−card*) ordering selects the first variable arbitrarily, then, at each stage, it selects a variable which is connected to the largest set of already selected variables. This heuristic can be thought of as the fixed version of dynamic search rearrangement: the next variable to be selected is the most constrained one, being connected to the largest set of already instantiated variables.

A **DFS ordering** is generated by a depth-first-search traversal of the constraint graph. It can be combined with any of the previous ordering for tie-breaking rule. In our experiments the tie breaking rule was random.

4. Experimental Design

Our experiments were performed in two different locations to be named **site-1** and **site-2**. In each location different problem instances were generated and different algorithm combinations were tested. Overall, 35 algorithm combinations were tested. In site-1, *backtracking (BTK)* and *backjumping (BJ)* were executed on each problem instance twice, once directly, without any pre-processing, and once after pre-processing the network by either *directional arc consistency (DAC)*, *directional path consistency (DPC)* or *adaptive consistency (ADAPT)* (8 combinations). Each algorithm combination was run along each one of the static ordering, *max-degree, max-cardinality*, and *min-width,* (yielding 24 combinations). Two more runs of *backtracking* and *backjumping* (without pre-processing) were performed in conjunction with dynamic ordering. In site-2, *backtracking* and *backjumping* were executed twice on each problem instance: once without any pre-processing, and once after pre-processing it by full *arc−consistency* (*AC*) (4 combinations). Each algorithm combination was tested with each one of the static orderings, *max−cardinality, min−width* and *DFS* ordering, and with the *dynamic ordering* (yielding 16 combinations).

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

	BTK	BJ	DAC BJ	DPC BJ	ADAPT BJ	DAC BTK	DPC BTK	ADAPT BTK
max-degree	1	1	1	1	1	1	1	1
max-card	1 2	1 2	1 2	1	1	1	1	1
min-width	1 2	1 2	1 2	1	1	1	1	1
dynamic	1 2	1 2						
dfs	2	2	2					

1 — site-1
2 — site-2

Table 1: Algorithms and test combinations

The test problems in each site were selected from a randomly-generated CSPs. The random problems were created by generating random graphs and associating each arc with a randomly-generated binary constraint. We purposely concentrated on parameters (e.g., probability of an arc) which result in *more difficult problems* for backtracking. We chose to restrict the set of test problems to binary CSPs primarily because problems with constraints of higher order tend to have denser constraint graphs for which the pre-processing algorithms have higher overhead. It should be pointed out, however, that *adaptive consistency* adds to the network non-binary constraints, hence the implementation of *backtracking* and *backjumping* had to accommodate general, non-binary, CSPs.

The problem instances experimented with in both sites had similar characteristics. In site-1 we experimented with two sets of random problems: one containing 42 instances, each having 10 variables and 5 values and the other, containing 35 instances, each with 15 variables and 5 values. These instances represent the more difficult problems among a much larger set of problems, from which all the easy problems were omitted. Larger problems took too much time and space for our machine to handle, especially for *adaptive consistency*. Similarly, in site-2 we experimented with two sets of random problems. One consisting of 104 instances, each having 10 variables and 5 values and the other with 107 instances having 15 variables and 7 values each.

We recorded the number of consistency-checks and the number of dead-ends (number of backtrackings) in each run. The number of consistency-checks is considered a realistic measure of the overall performance, while the number of dead-ends is indicative of the size of the search space explored. The implementation of *dynamic ordering* in site-2 used an additional data structure in the form of a set of tables. In this case we counted the number of table-lookups and added it to the number of consistency checks.

Each algorithm was run twice on each problem instance: once for finding one solution and once for finding all solutions. The results were clustered into 6 groups according to the 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 Pre-processing Algorithms

We first report our results in site-1. Our first goal is to compare the effects of the three pre-processing algorithms *DAC*, *DPC* and *ADAPT* followed by *backtracking* and *backjumping*. Figures 7-12 present graphs displaying the average number of consistency checks, classified according to the width of the induced graph, W^*. Figures 7 and 8 present data for the *max-degree* ordering. The first displays results for 10-variable instances, while the second focuses on 15-variable instances. Similarly, Figures 9 and 10 display *min-width* results, and Figures 11 and 12 summarize *max-cardinality* results. Each horizontal pair of graphs presents the results for a group of instances, where the left graph contrasts the results for algorithms *ADAPT*, *BTK* and *BJ*, while the right one shows (using a different scale) the results for algorithms *BJ*, *DAC* and *DPC*. The results reported for *DAC*, *DPC*, and *ADAPT* are for the cases where they were complemented by *backjumping*. When we used *backtracking*, we observed a similar behavior, since after pre-processing most of the dead-ends were eliminated. Comparing *ADAPT* to *BTK* and *BJ* (left columns in all figures), we see that even on the average, adaptive-consistency has an exponential behavior as a function of W*. *BTK* and *BJ*, on the other hand, exhibit a much more moderate, maybe even linear, behavior.

The average performance of *ADAPT* is better than BTK only for small values of W^*, mostly for finding all solutions. Evidently, the amount of pre-processing performed by *ADAPT* is too heavy to be justified by just one solution (see left, upmost graphs), but when it is shared by several solutions, it becomes worth-while (see left, middle figures). For the case of n=10, when looking for one solution, BTK outperformed *ADAPT* even for $W^* = 1,2$.

When compared to *BJ*, however, *ADAPT* appears as a complete looser. BJ outperformed *ADAPT* in almost all instances. Clearly, BJ exploits the structure of the problem in a more efficient way than *ADAPT* and should be preferred, especially considering the fact that it doesn't need the additional space which is consumed by *ADAPT*. Although *ADAPT* does not seem to be a sensible choice for a one time solution, it still can be used for finding a better representation of a network of constraints, for example, 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 [Dechter 1989].

The disappointing results of *ADAPT* can be explained by comparing it with the two other, less ambitious, pre-processing algorithms, *DAC* and *DPC*. When we counted the number of dead-ends left after pre-processing (Figure 13), we found out that in almost all problem

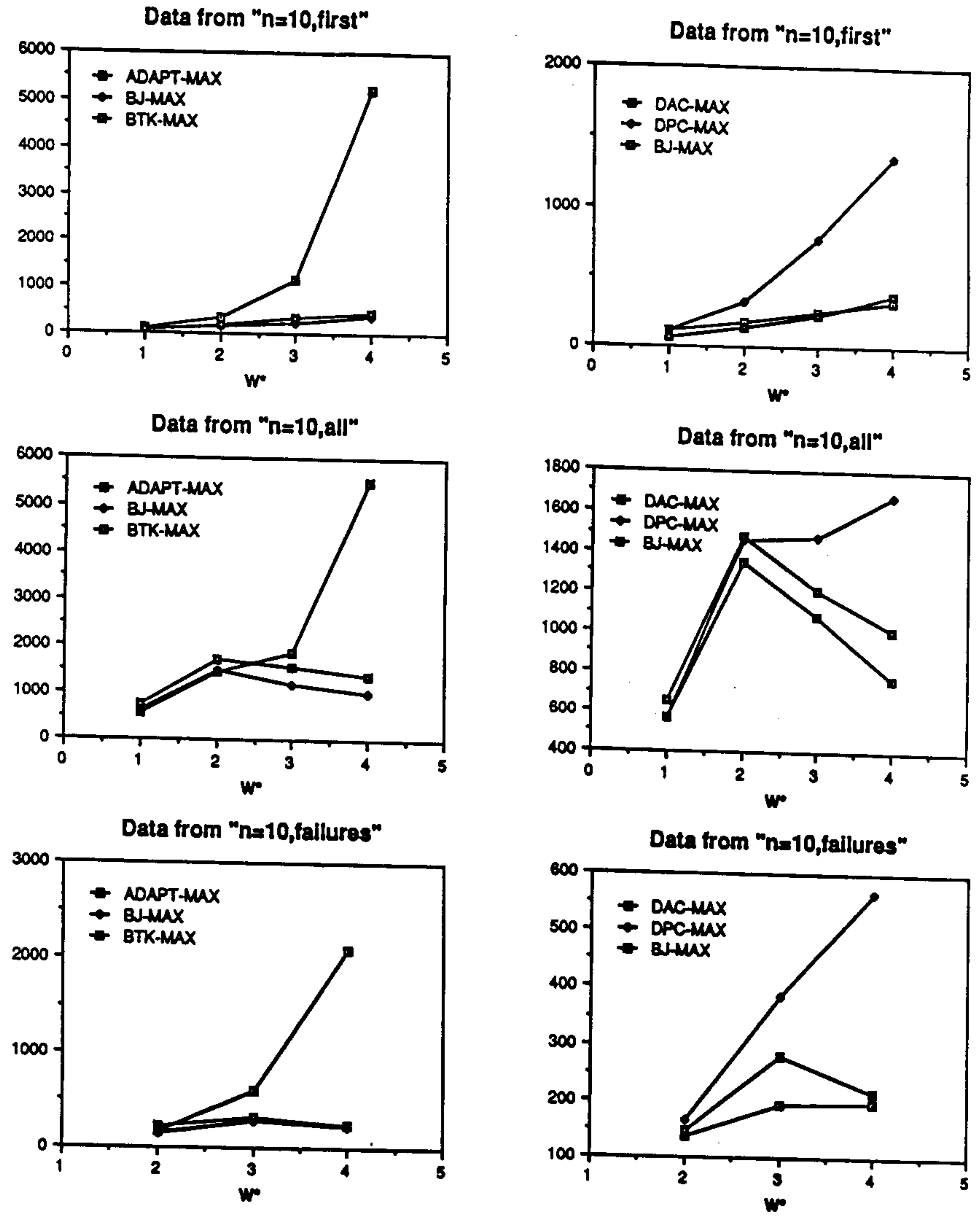


Figure 7 : number of consistency-checks for algorithms *DAC*, *DPC*, *ADAPT*, *BJ* and *BTK* with *max−degree* ordering on 10-variables random problems.

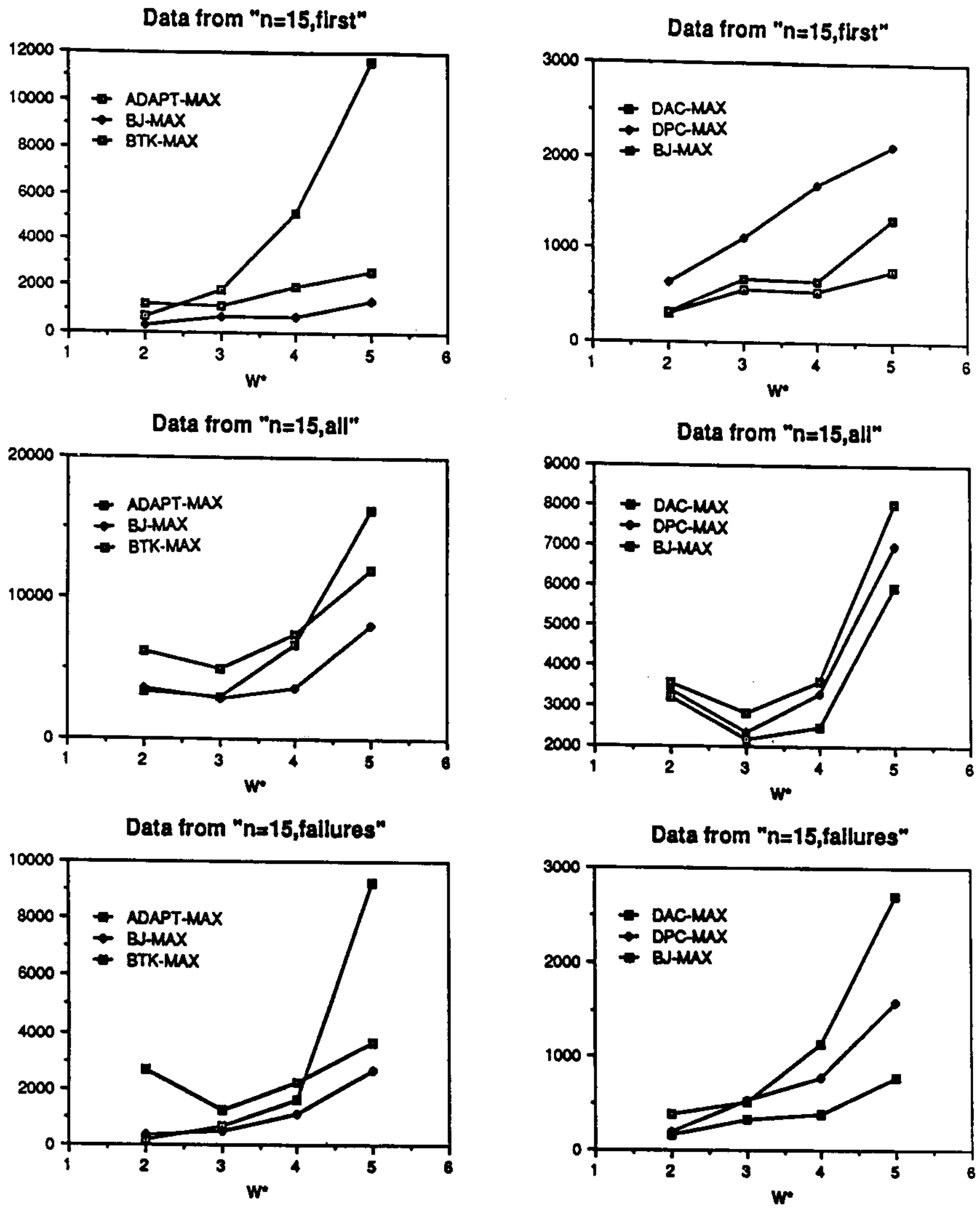


Figure 8 : number of consistency-checks for pre-processing algorithms *DAC*, *DPC*, *ADAPT*, *BJ* and *BTK* with *max−degree* ordering on 15-variables random problems.

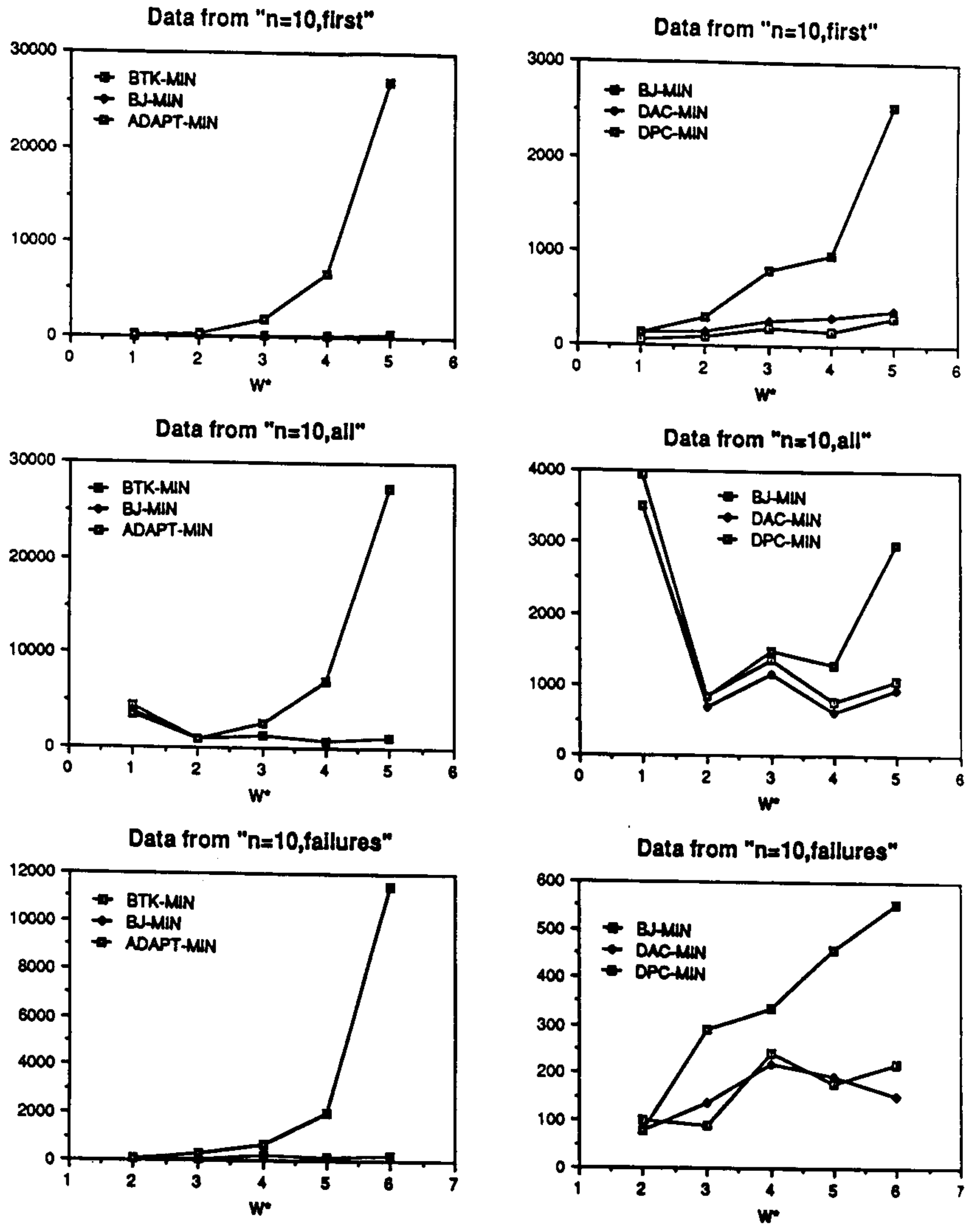


Figure 9 : number of consistency-checks for pre-processing algorithms *DAC*, *DPC*, *ADAPT*, *BJ* and *BTK* with *min−width* ordering on 10-variables random problems.

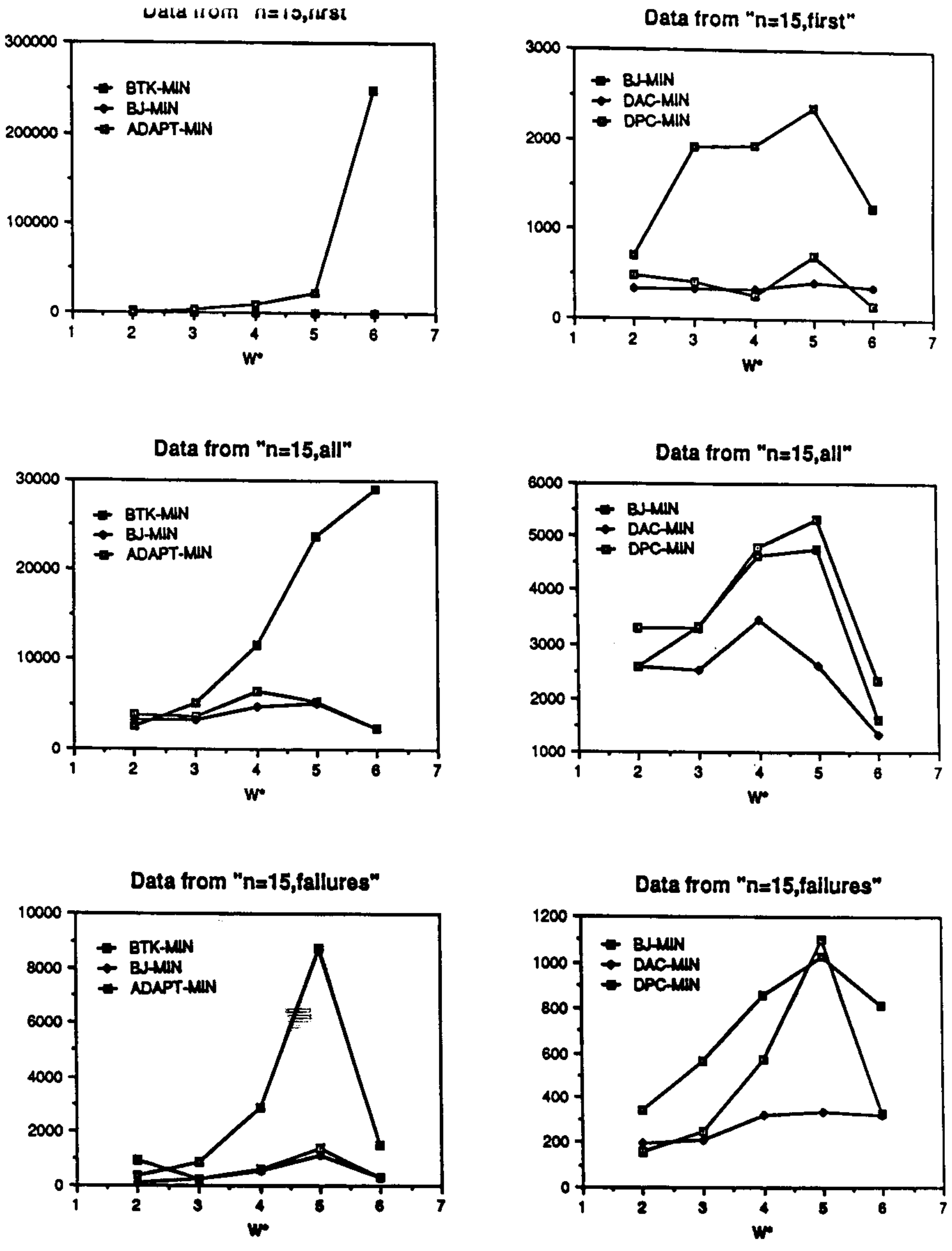


Figure 10 : number of consistency-checks for pre-processing algorithms *DAC*, *DPC*, *ADAPT*, *BJ* and *BTK* with *min–width* ordering on 15-variables random problems.

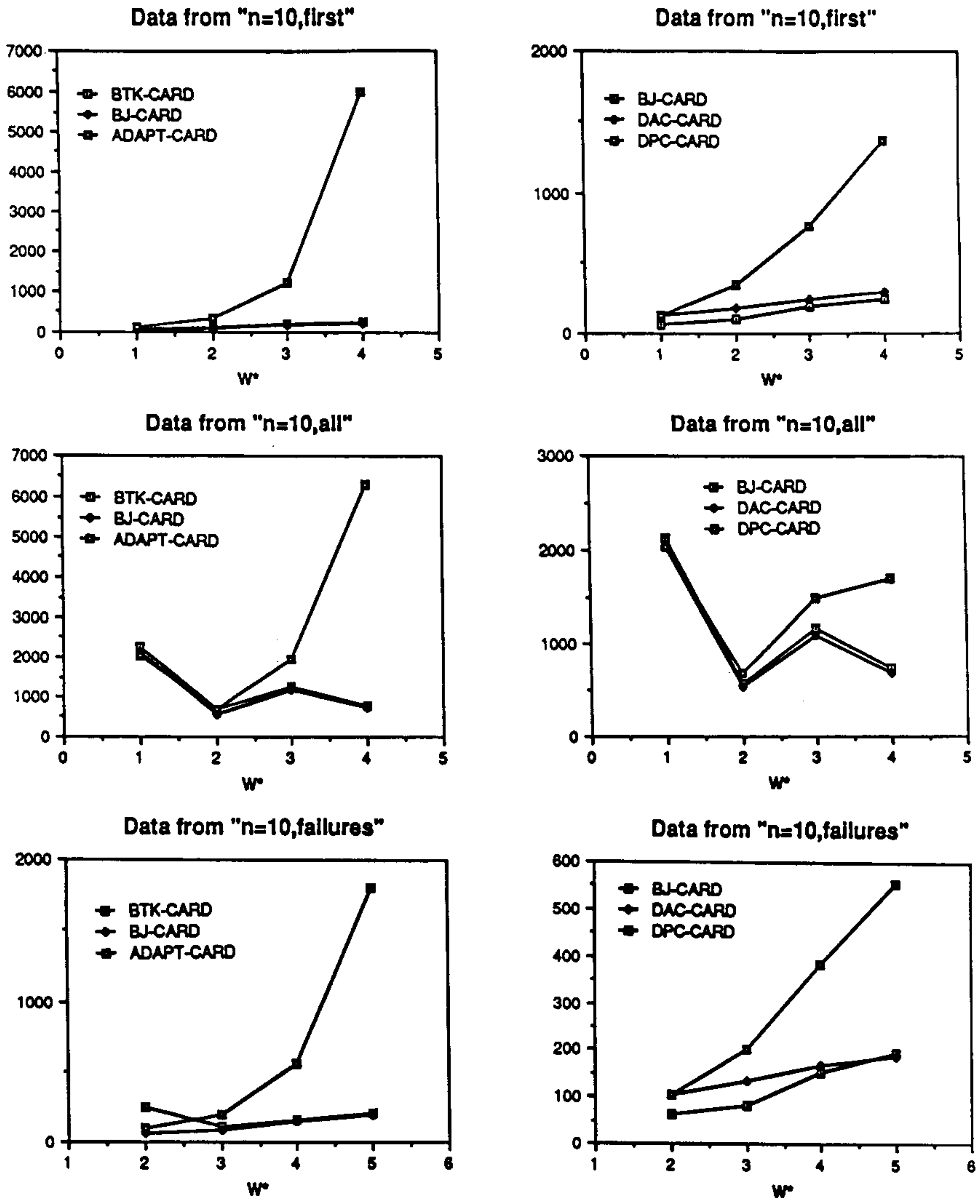


Figure 11 : number of consistency-checks for pre-processing algorithms *DAC*, *DPC*, *ADAPT*, *BJ* and *BTK* with *max−cardinality* ordering on 10-variables random problems.

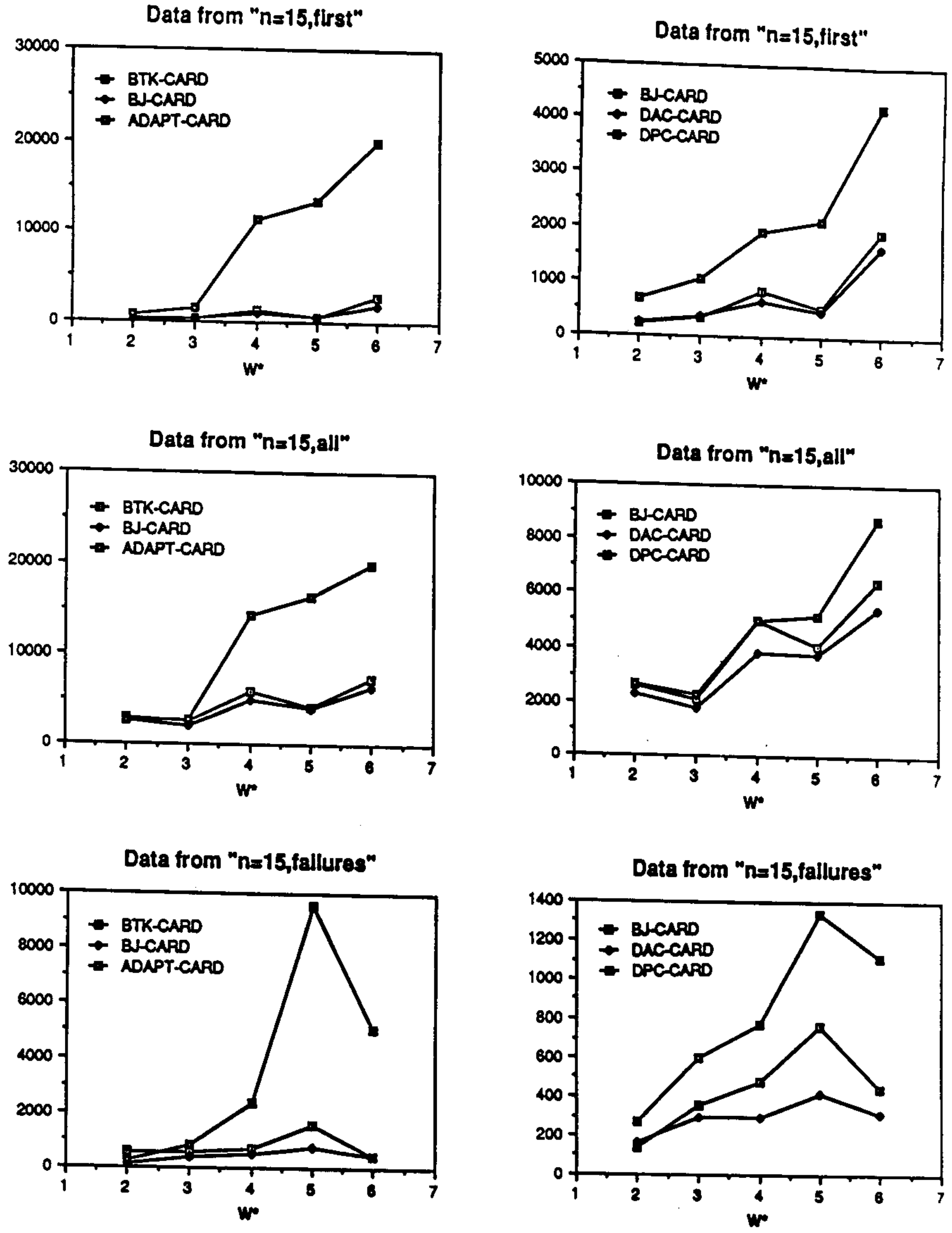


Figure 12 : number of consistency-checks for pre-processing algorithms *DAC*, *DPC*, *ADAPT*, *BJ* and *BTK* with *max−cardinality* ordering on 15-variables random problems.

instances even algorithm *DPC* eliminated all future dead-ends. It is clear, therefore, that for problem instances of this type, *ADAPT* is doing unnecessary pre-processing. Moreover, the number of dead-ends left by algorithm *DAC* (see figure) shows that a substantial amount of the work is accomplished even by this algorithm which performs the smallest amount of constraint recording. Observing the graphs of Figures 7-12, we see from their left columns that *BJ* outperformed both *BTK* and *ADAPT*.

Let us focus now on the right-hand columns which compare *BJ*, *DAC* and *DPC*. Clearly, the two algorithms that stand out in these experiments were *BJ* and *DAC* (followed by *BJ*). Both outperformed *DPC* in almost all cases, however none of them dominated the other. When looking carefully at their relative performance according to the problem's size, the ordering used and the task at hand, we see the following pattern. Algorithm *BJ* was better than *DAC* only for the task of 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) or for the task of finding all solutions, *DAC* was better. This result is easy to explain: for heavy tasks, the overhead presented by *DAC* will be outweighed by its gain.

5.2 The Effects of Variable Ordering

We now focus on characterizing the effect of variable ordering, be it static or dynamic, on the various algorithm combinations, in particular on *backtracking* and *backjumping*. We present results from both sites. Figure 14 presents results from site-1. It shows the results of running *backtracking* and *backjumping* using four orderings: 1. *max-degree (max)*, 2. *max-cardinality (card)*, 3. *min-width (min)*, and 4. *dynamic ordering (dnmic)*. Each graph presents the average number of consistency checks over all instances, disregarding W^*. Again, we group the results according to the problem size (10 or 15 variables), and the three cases, *first*, *all*, and *failures*.

To see the effect of the induced width, a selected set of graphs is presented in Figure 15. To maintain continuity we averaged the same set of instances, and therefore W* indicates the

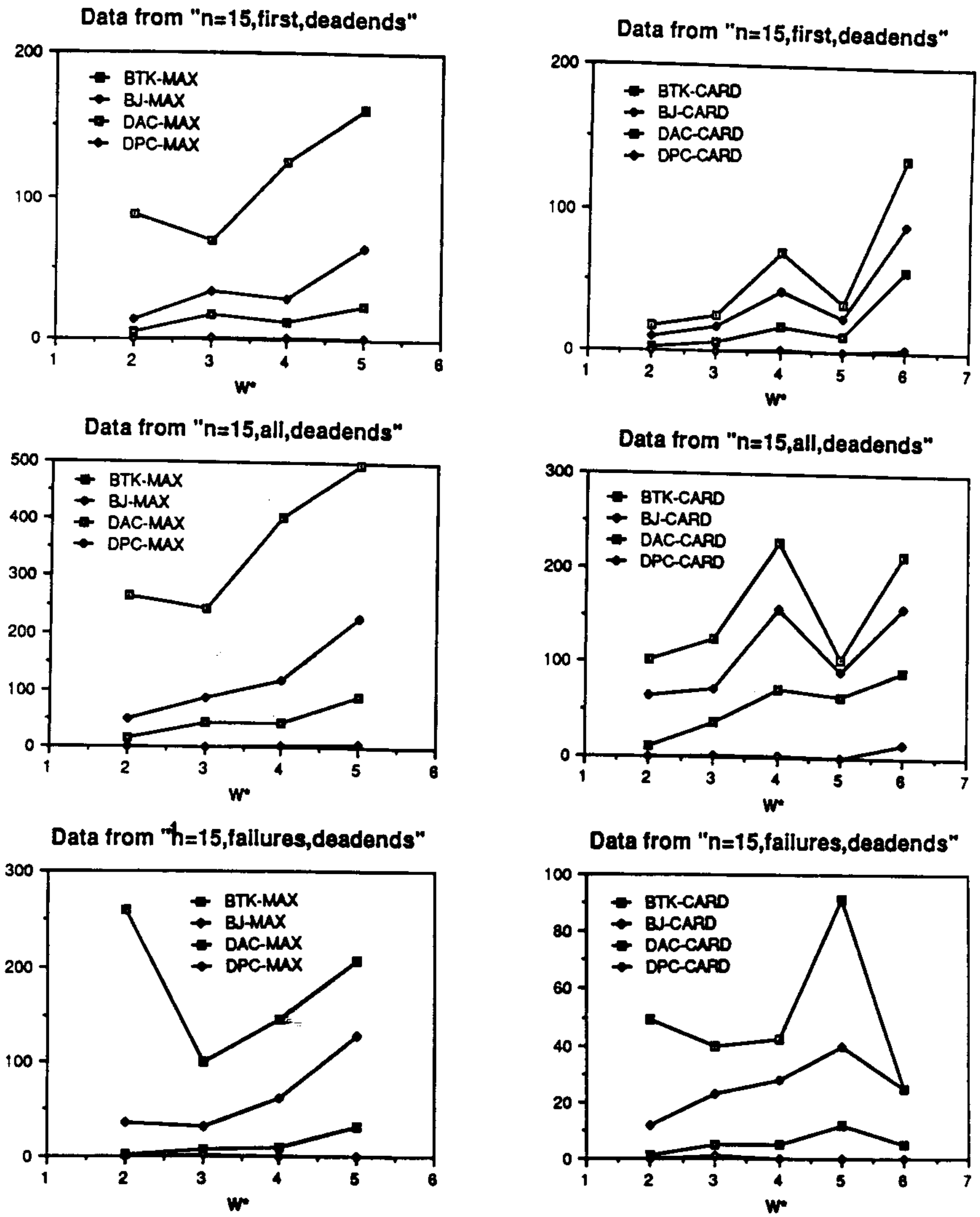


Figure 13 : number of deadends for processing algorithms (*DPC*, *DAC*, *BJ* and *BTK*) with *max−cardinality* and *max−degree* ordering on 15-variables random problems.

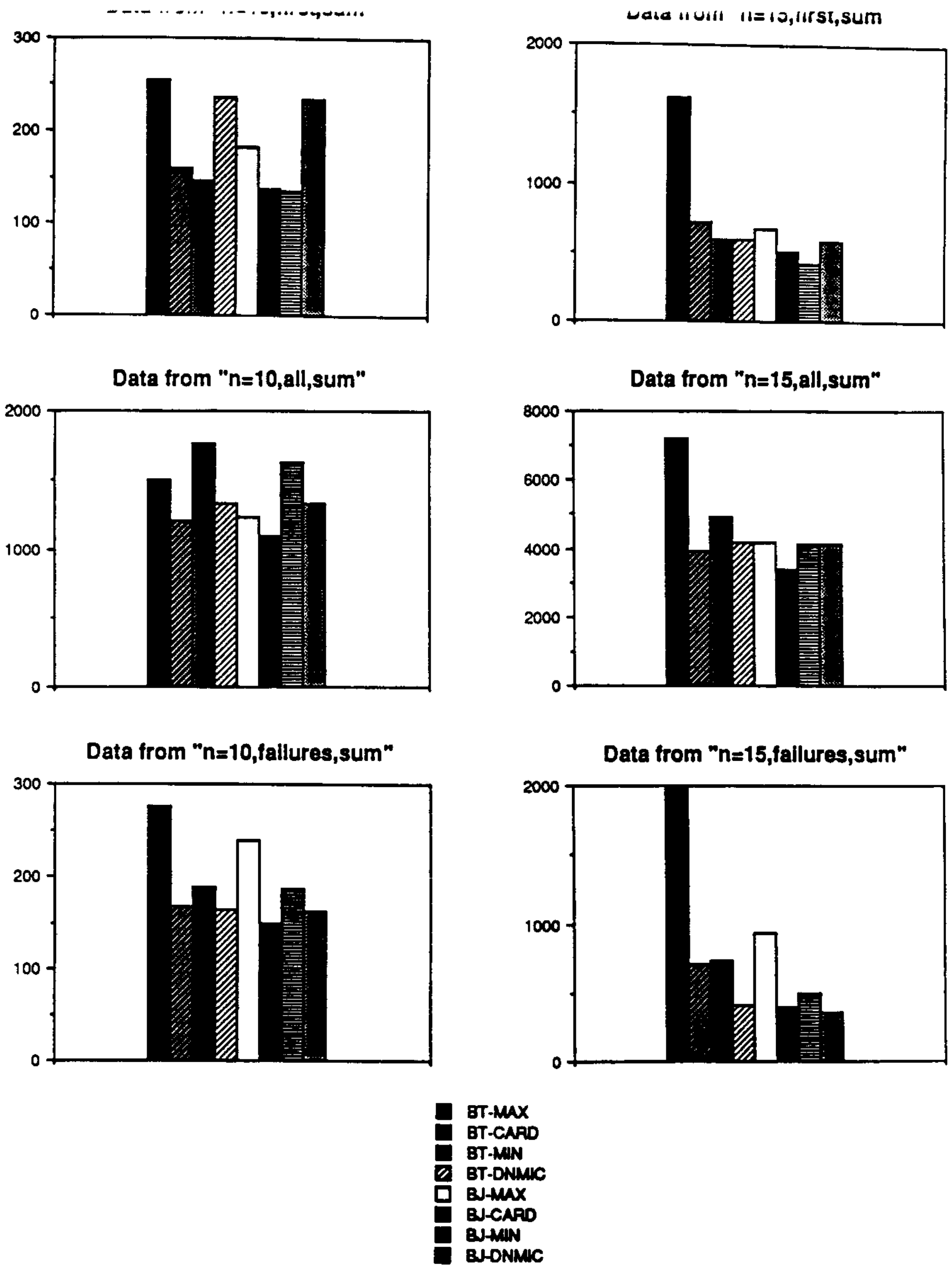


Figure 14 : number of consistency-checks for *BJ* and *BTK* on orderings: *max−degree*, *max−cardinality*, *min−width* and *dynamic* in site-1 for 10 and 15 variable problems.

induced width of *max-degree* ordering.

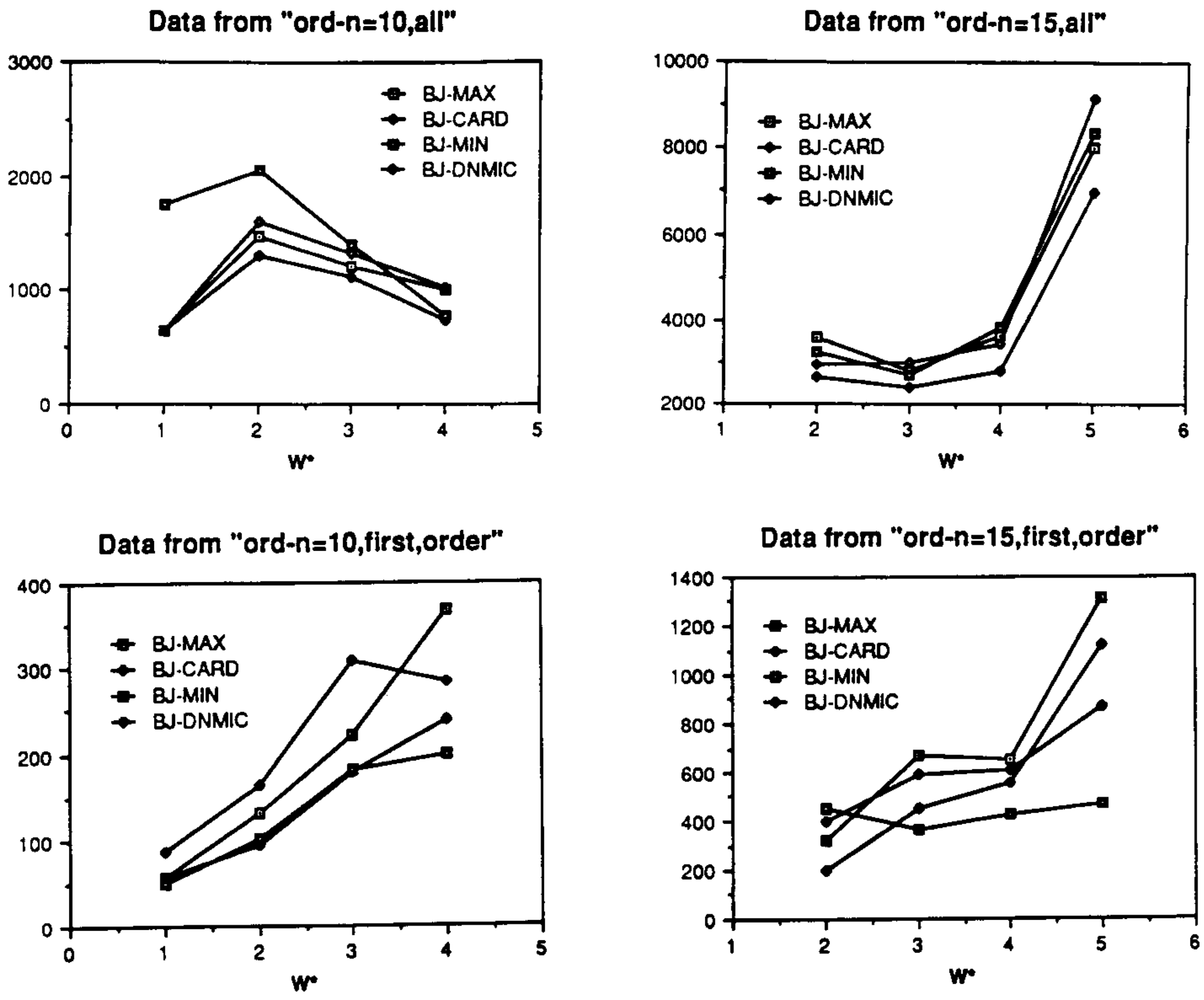


Figure 15: The effect of variable ordering on consistency checks parametrized by $w*$.

Figure 14 shows that the *max−degree* ordering comes out as a complete looser, while there is no clear ''winner''. Again we observe some patterns governing the role of ordering relative to the task and the problem size. Specifically, *min−width* was the best ordering for the task of finding the first solution (except for *BTK* in large problems), *max-card* was the best ordering for finding all solutions, and *dynamic* ordering was generally best for *failure* instances (with the exception of *BJ* in small problems). When we compared the number of dead-ends associated with each ordering (see Figure 16), it becomes clear that *dynamic* ordering expands the smallest search space (i.e., it has the least number of dead-ends on an instance by instance basis, almost).

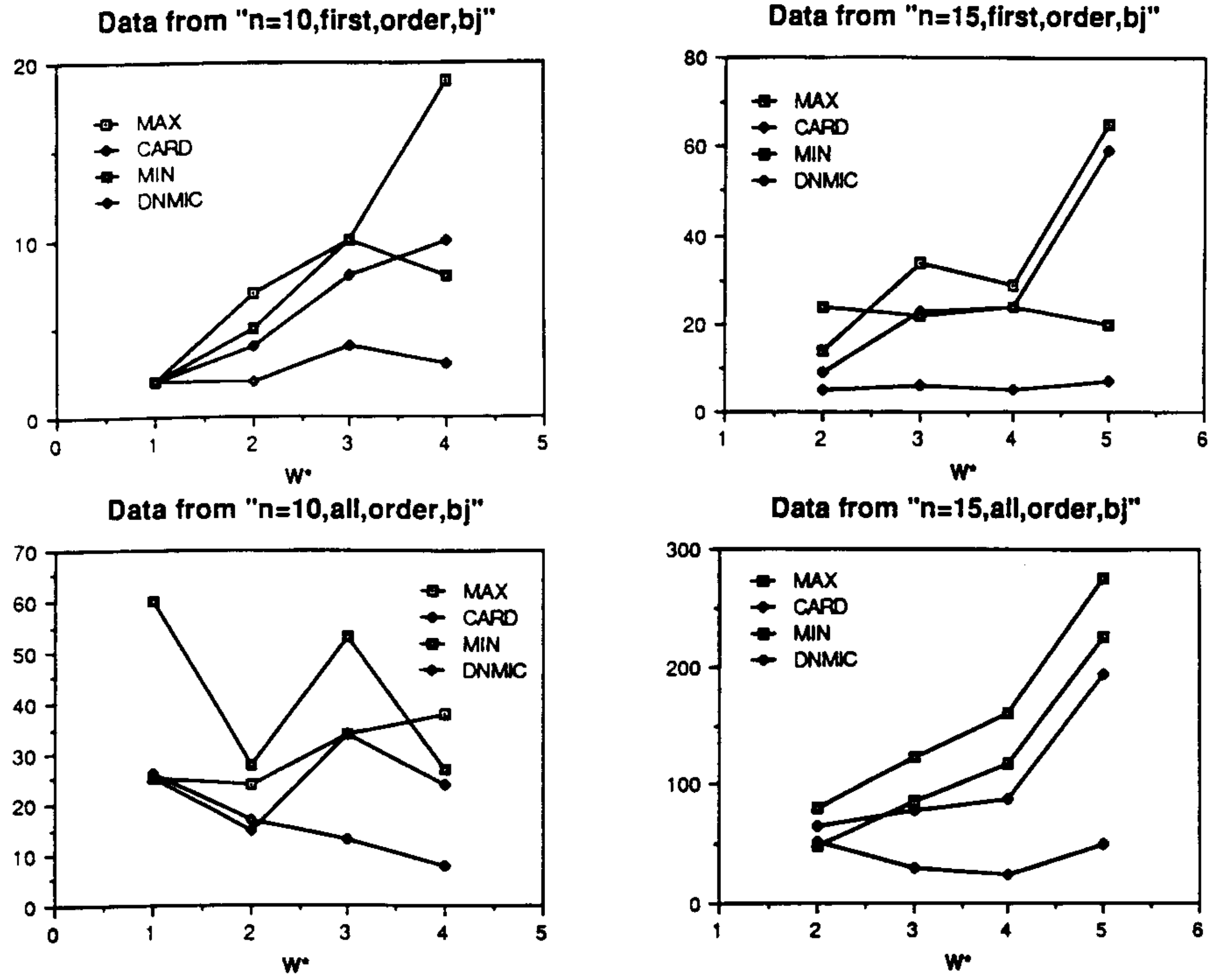


Figure 16: The effect of variable ordering on search space (number of deadends) parametrized by w^*.

Therefore, had we better implemented this technique we may have had a better overall performance. Indeed in site-1's implementation of *dynamic* ordering no data structure was used to alleviate redundant consistency checks as was done in other look-ahead schemes such as *forward checking* [Haralick 1980].

This problem was corrected in our experiments in site-2. Here we compared three static orderings and one *dynamic* ordering. We used *min−width* and *max−card* as in site-1, but instead of *max−degree* (as it had been so bad in site-1) we used a *DFS* ordering with a random tie-breaking rule. In this site, *dynamic* ordering was implemented more efficiently, using data structures similar to those used in [Haralick 1980] that take at most quadratic space. Consequently, when collecting the data, we added the number of table look-ups to the number of consistency checks.

As can be seen from Figures 17 and 18, with this implementation *dynamic* ordering dominated all other orderings. Contrary to our observations in site-1, we see a clear superiority of *min-width* over *max-card* in these instances.

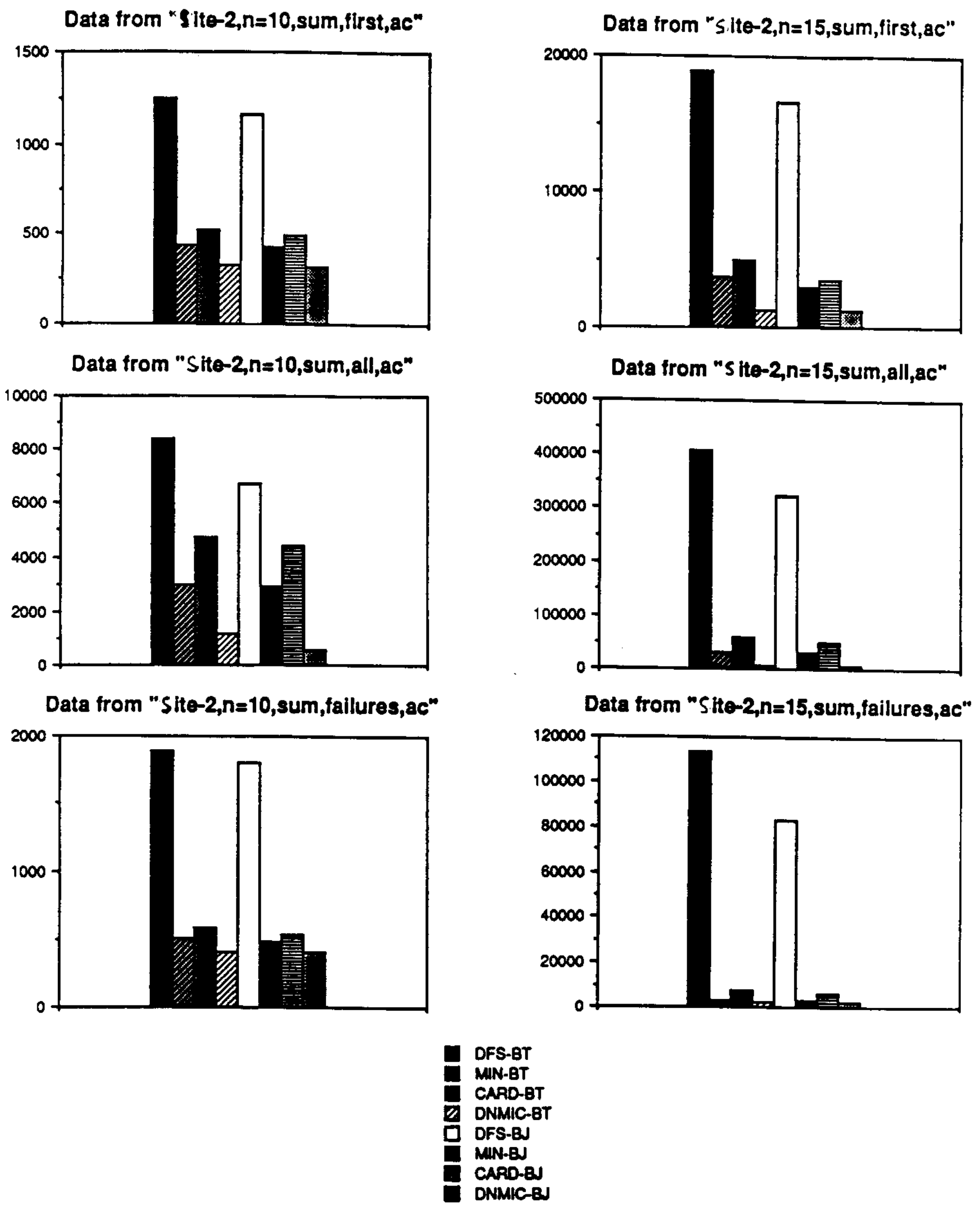


Figure 17 : number of consistency-checks for *BJ* and *BTK* on orderings: *max−degree*, *max−cardinality*, *min−width* and *dynamic* in site-2 for 10 and 15 variable problems.

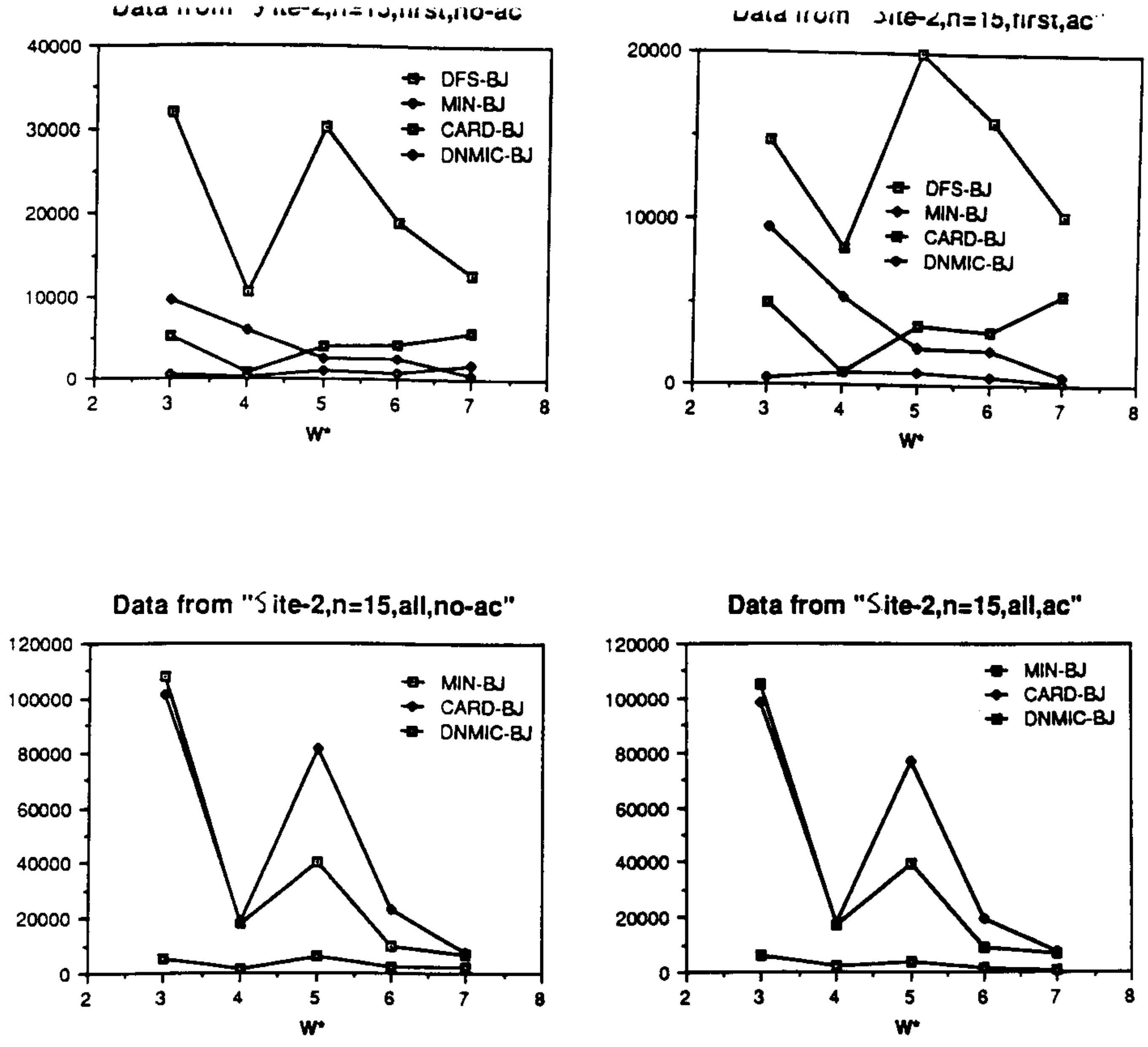


Figure 18: The effect of variable ordering on number of consistency checks parametrized by w^* (with and without pre-processing by *arc−consistency*) in site-2.

6. Summary and Conclusions

We evaluated the performance of several backtracking techniques for solving CSPs. First, we tested the effect of various pre-processing algorithms on *backtracking* and *backjumping*, using static orderings, and second, we tested the effect of five variable ordering schemes. The conclusions are summarized in Figures 19 and 20. We used a graphical representation to summarize the relative merits of the algorithm as reflected by the number of consistency checks. An arrow from *A* to *B* indicates that algorithm *A* is superior to algorithm *B*, with perhaps some exceptions as annotated on the arrows. For instance, figure 19 indicates that *DAC* outperforms *BJ* except on finding first solution on size-10 problems. Similarly, Figure 20 presents the relative strength of different orderings w.r.t. *BTK* (Figure 20a) and *BJ* (Figure 20b). The solid lines show results taken at site-1 while the dotted lines show results taken at site-2. An inconclusive relationship is denoted by an undirected arc labeled by "≠".

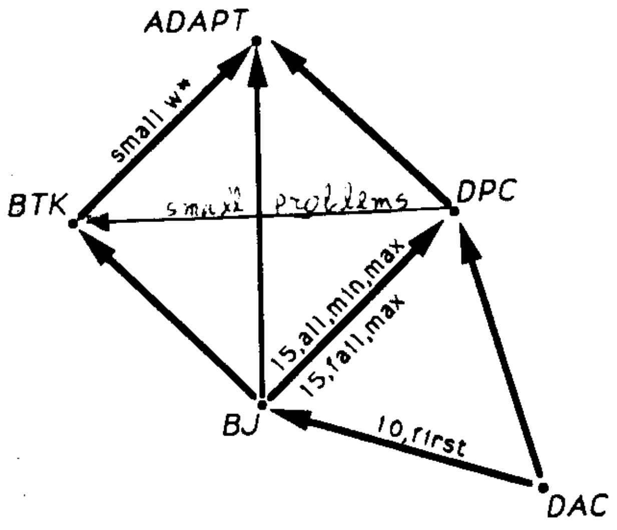


Figure 19: Relative performance of pre-processing algorithms

In summary, our experiments indicate that in most cases *directional arc consistency* followed by *backjumping* outperforms all other schemes, while *dynamic search rearrangement* is the most promising ordering scheme. When static ordering is used, the experiments suggest that combining *DAC* with *min-width* or *max-cardinality* orderings will yield best results on the average.

Viewed from the perspective of prior experiments our results fit into a general pattern. In all techniques tested, be it pre-processing or in-processing, weak enhancement schemes were the most effective (due to their low overhead). Stronger schemes did not pay off. For example, in testing different levels of look-ahead schemes, Haralick and Elliot concluded that, *forward-checking*, the least intensive look-ahead mechanism performed much better then the more

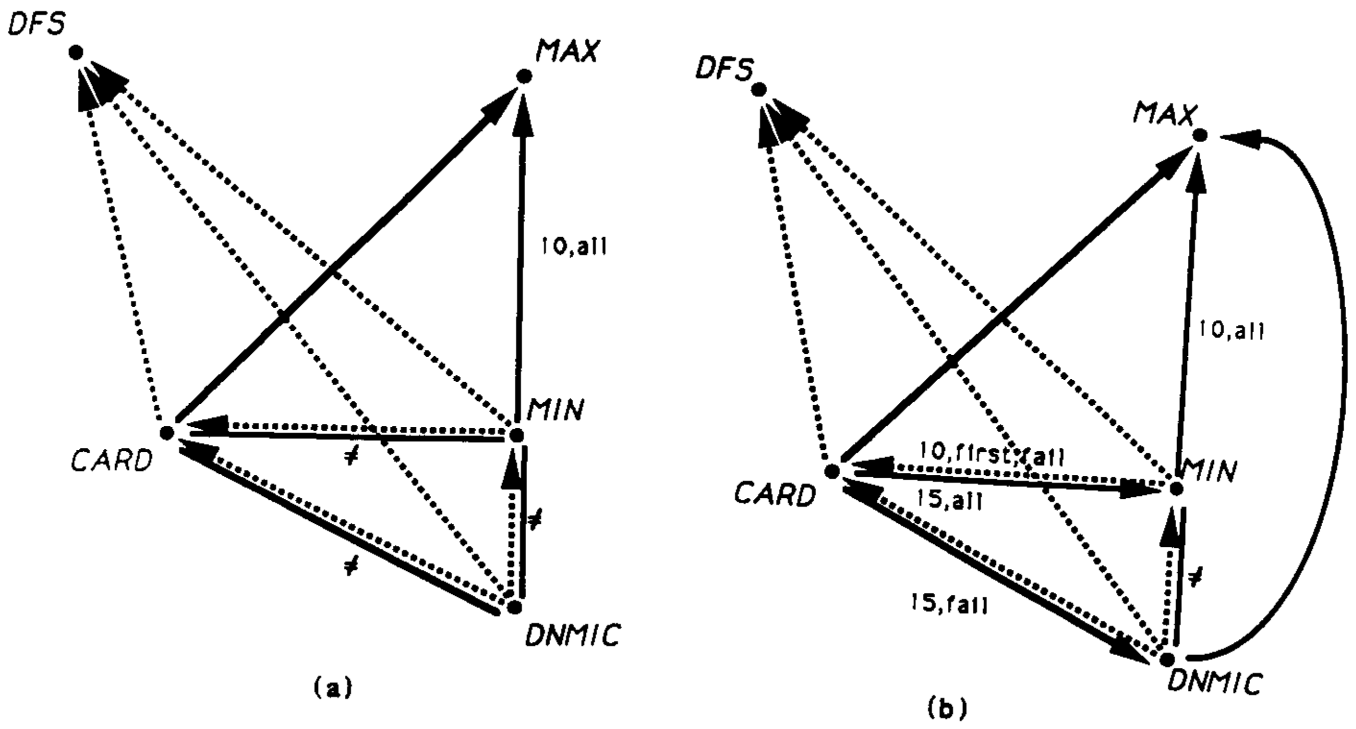


Figure 20: Relative merits of ordering schemes for *BTK* (a) and *BJ* (b).

intensive *partial-look-ahead* and *full-look-ahead* (see figures 6,8,10,11 in [Haralick 1980]). The same is evident from Gaschnig's experiments with *DEEB*, which was his way of incorporating full arc-consistency into the search (see figures 4.3-1, 4.4.2-2 in [Gaschnig 1979]). Similarly, in generating heuristics for value selection preferences it was shown that only very shallow advise improved the search (see figures 15 and 16 in [Dechter 1987]). In assessing various look-back schemes we found the same phenomenon. When augmenting *backjumping* with various levels of constraint recordings, (i.e., learning no-goods parametrized by the size of the constraints and the depth of reasoning) it became evident that only very shallow learning of only small constraints was worth undertaking (Figure 7,8 in [Dechter 1990]).

These results, current and previous ones alike should be qualified before further extrapolated. The conclusions are valid only relative to problem domains with statistics similar to those used in generating the test samples. Consequently, the next phase of experimentation should focus on testing whether this pattern of behavior scales up to larger random problems and to real-life applications.

Acknowledgement

We would like to thank Tal Sela and Nadav Eran for implementing the techniques in site 2. We would also like to thank Judea Pearl for commenting on the last version of this manuscript.

References

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

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

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

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

[Dechter 1991] Dechter, R., "Constraint Networks," in *Encyclopedia of Artificial Intelligence*, S. Shapiro, Ed. Wiley and Sons, December, 1991.

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

[Freuder 1978] Freuder, E.C., "Synthesizing constraint expression," *Communication of the ACM*, Vol. 21, No. 11, 1978, pp. 958-965.

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

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

[Ginsberg 1990] Ginsberg, M., "Search lesson learned from crossword puzzles," in *Proceedings Eighth National Conference on Artificial Intelligence*, Boston, MA: July 29 - August 3, 1990, pp. 210-215.

[Haralick 1980] Haralick, R. M. and G. L. Elliott, "Increasing tree-search efficiency for constraint satisfaction problems," *Artificial Intelligence*, Vol. 14, 1980, pp. 263-313.

[Mackworth 1991] Mackworth, A., "Constraint Satisfaction," in *Encyclopedia of Artificial Intelligence*, S. Shapiro, Ed. Wiley & Sons inc., December 1991.

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

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

[Nudel 1983] Nudel, B., ''Consistent-Labeling Problems and their Algorithms: Expected Complexities and Theory-based Heuristics,'' *Artificial Intelligence,* Vol. 21, 1983, pp. 135-178.

[Prosser 1991] Prosser, P., ''Hybrid Algorithms for the Constraint Satisfaction Problem,'' University of Strathclyde, Computer Science Dept., Glasgow, Scotland, Tech. Rep. AISL-46-91, September 1991.

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

[Purdom 1981] Purdom, P. W., E. L. Robertson, and C. A. Brown, ''Backtracking with Multi-level Dynamic Search Rearangement,'' *Acta Informatica,* Vol. 15, No. 2, 1981, pp. 99-114.

[Rosiers 1986] Rosiers, W. and M. Bruynooghe, ''Empirical Study of Some Constraint Satisfaction Algorithms,'' Katholieke Universiteit Leuven, Leuven , Belguim, Tech. Rep. CW 50, July, 1986.

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

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

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

[Zabih 1988] Zabih, R. and D. McAllester, ''A Rearrangement Search Strategy for Determining Propositional Satisfiability,'' in *Proceedings AAAI-88,* St. Paul, Minnesota: August, 1988.