UC Irvine
ICS Technical Reports

**Title**
Experimental evaluation of preprocessing algorithms for constraint satisfaction problems

**Permalink**
https://escholarship.org/uc/item/66x1p6q3

**Authors**
Dechter, Rina
Meiri, Itay

**Publication Date**
1992

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Z
699
C3
no. 92-63
Rev.

# Experimental Evaluation of Preprocessing Algorithms for Constraint Satisfaction Problems

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

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

Technical Report 92–63

January, 1992
revised August, 1992

Notice: This Material may be protected by Copyright Law (Title 17 U.S.C.)

This work was supported in part by the National Science Foundation, Grants #IRI-8815522 and IRI-9157636, by the Air Force Office of Scientific Research, Grant #AFOSR-88-0177, and by GE Corporate R&D and by Toshiba of America.

# EXPERIMENTAL EVALUATION OF PREPROCESSING ALGORITHMS FOR CONSTRAINT SATISFACTION PROBLEMS*

**Rina Dechter**

Information and Computer Science Department
University of California, 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 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 its common enhancement, *backjumping*. Surprizingly, 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 pre-processing) 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 pre-ordering (i.e., static ordering) strategies, while *dynamic search rearrangement* is superior to all the preorderings studied.

**August 28, 1992**

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

## 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 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 ( [Gaschnig 1979] ), Haralick and Elliot's study of look-ahead strategies ( [Haralick 1980] ), Brown and Purdom's experiments with dynamic variable orderings [Purdom 1985], 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, Hentenryck 1987].

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 pre-processing algorithms and their effect on backtracking's performance. Since our pre-processing 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 pre-processing 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 [Mackworth 1977]. 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)* [Dechter 1987]. These algorithms represent increasing levels of pre-processing 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 pre-processing performed by the most aggressive scheme, *ADAPT*, did not paid 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 pre-processing. Additionaly, while *ADAPT* showed that its average complexity is exponentialy dependent on $W^*$, the dependence of all other schemes on $W^*$ seems to be quite weak or even non existing.

In the second class we report the effect of various static ordering strategies on *backtracking* and *backjumping* without pre-processing. 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 pre-processing 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)* [Purdom 1983]. We tested two implementation styles of *DSR*, presenting a trade-off 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.

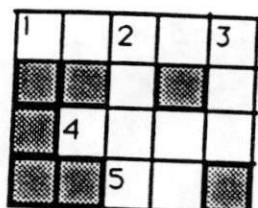
## 2. Constraint Processing Techniques

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] 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 $\rho$ defined on the set of all variables. Formally,

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

[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.

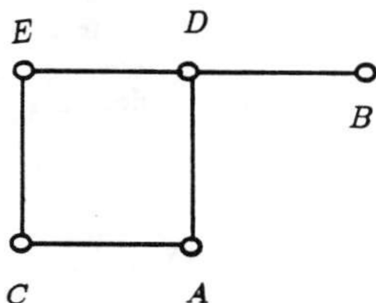
A CN may be associated with a constraint graph in which nodes represent variables and arcs connect variables that appear in the same constraint. For example, the CN depicted in Figure 1a represents a crossword puzzle. The variables are $E$ (1, horizontal), $D$ (2, vertical), $C$ (3, vertical), $A$ (4,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



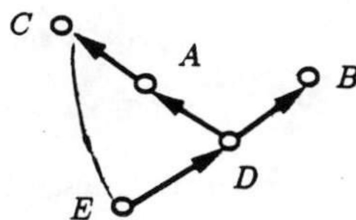

$$D_E = \{\texttt{hoses,laser,sheet,snail,steer}\}$$
$$D_A = D_D = \{\texttt{hike,aron,keet,earn,same}\}$$
$$D_C = \{\texttt{run,sun,let,yes,eat,ten}\}$$
$$D_B = \{\texttt{no,be,us,it}\}$$

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

(a)




(b)




(c)

Figure 1: (a) A crossword puzzle, (b) its CN representation, and (c) a depth-first search pre-ordering

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).

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 [Mackworth 1992, Dechter 1992].

*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 **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 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: pre-processing algorithms that are employed prior to performing the search, and dynamic algorithms that are used during the search.

The pre-processing algorithms include a variety of **consistency-enforcing algorithms** ( [Montanari 1974, Mackworth 1977, Freuder 1978] ). 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 $i$-th 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 pre-processing algorithms also include algorithms for ordering the variables prior to search. Several heuristics for **static orderings** have been proposed [Freuder 1982, Dechter 1989]. The heuristics used in this paper *minimum width*, *maximum cardinality*, *maximum degree*, and *depth-first search* 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 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, like such as *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. An example of a look-ahead scheme is *dynamic search rearrangement*, which decides what 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. An example of such a scheme is **backjumping** [Gaschnig 1979]. 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. The algorithm may also record the reasons for the dead-end 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 [Stallman 1977, Dechter 1990].

**Structure-based techniques**, such as graph-based *backjumping*, *directional i-consistency*, *adaptive consistency*, and *cycle-cutset scheme* can be viewed as structure-based improvements of some of the above techniques [Dechter 1992].

## 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 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 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.

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

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

Figure 2: Algorithm *backtracking*

*Backjumping* improves the ''*go-back* phase of *backtracking*. 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. *Graph-based backjumping* has been shown to outperform *backtracking* on an instance-by-instance basis [Dechter 1990]. 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 dead-end (see Figure 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 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 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 [Even 1979], 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 Figure 1b

is given in Figure 1c. The *DFS* ordering (which amounts to an in-order 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, where *m*, is the depth of the *DFS*-tree [Collin 1991].

### 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*. Thus, there is a trade-off between the effort spent on pre-processing and that spent on search. The primary goal of our paper is to uncover this trade-off.

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 processieg many constraints that are not encountered during search.

We start with *ADAPT*, then generalize its underlying principle to describe a class of pre-processing 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 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 *ADAPT*, shown in Figure 5, processes the nodes in a reverse order, that is, each node is processed before any of its parents.

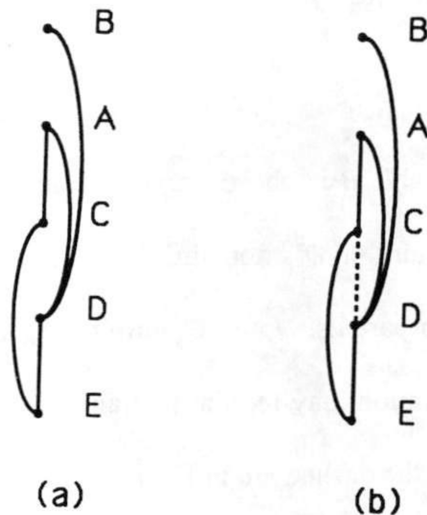



Figure 4: Ordered constraint graphs

*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. connect all elements in *parents*($X_i$) (if they are not yet connected)
End

Figure 5: Algorithm *ADAPT*

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, *earn*, 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 clusters of variables. It has been 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 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 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 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 recorded 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, . . . ,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 all elements in parents(X_i) (if they are not yet connected)
End

new-record(level, var, set)
Begin
1. if level ≤ |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 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.

The complexity of *adaptive* (*level*) is both time and space dominated by the procedure *new−record*(*level*) which is $O(\binom{W^*(d)}{level} \cdot (k^{\min\{level, W^*(d)\}}))$, $k$ 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 [Arnborg 1987].

### 3.3 Pre-ordering 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 [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 not yet selected. 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 [Purdom 1983, Haralick 1980, Nudel 1983], and experimentally in [Stone 1986, Rosiers 1986, Sadeh 1991.]. 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 [Purdom 1981, Zabih 1988].

## 4. Experimental Design

Our experiments were performed in two different locations, **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 *DAC, DPC,* or *ADAPT* (8 combinations). Each algorithm combination was run along with one of the static orderings *max-degree, max-card*, and *min-width* (24 combinations). Two more runs, of *backtracking* and *backjumping* both without pre-processing and in conjunction with *DSR* were also performed. In site-2, *backtracking* and *backjumping* were executed twice on each problem instance, once without any pre-processing and once after pre-processing *DAC* (4 combinations). Each algorithm combination was tested with the static orderings *max−card, min−width*, and *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 in site-1, while *BJ* was tested in both site-1 and site-2. Note that an instance-by-instance comparison is feasible only within sites.

The test problems in each site were selected from 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) that 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

| | 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<br>2 | 1<br>2 | 1<br>2 | 1 | 1 | 1 | 1 | 1 |
| *min-width* | 1<br>2 | 1<br>2 | 1<br>2 | 1 | 1 | 1 | 1 | 1 |
| *dynamic* | 1<br>2 | 1<br>2 | | | | | | |
| *dfs* | 2 | 2 | 2 | | | | | |

1 — site-1
2 — site-2

Table 1: Algorithms and test combinations

have denser constraint graphs, for which consistency enforcing algorithms have higher overhead. It should be pointed out, however, that *ADAPT* adds non-binary constraints to the network, hence the implementation of *backtracking* and *backjumping* had to accommodate general, non-binary CSPs.

The problem instances experimented with in each site had similar characteristics. In site-1, 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. These instances represent the more difficult problems among a much larger set of problems, from which all the easy problems were omitted. Larger problems required too much time and space for our machine to handle, especially for *ADAPT* and *DPC*. 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 of 107 instances, each with 15 variables and 7 values.

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 overall performance, while the number of dead-ends is indicative of the size of the search space explored. The implementation of *DSR* 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 six 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

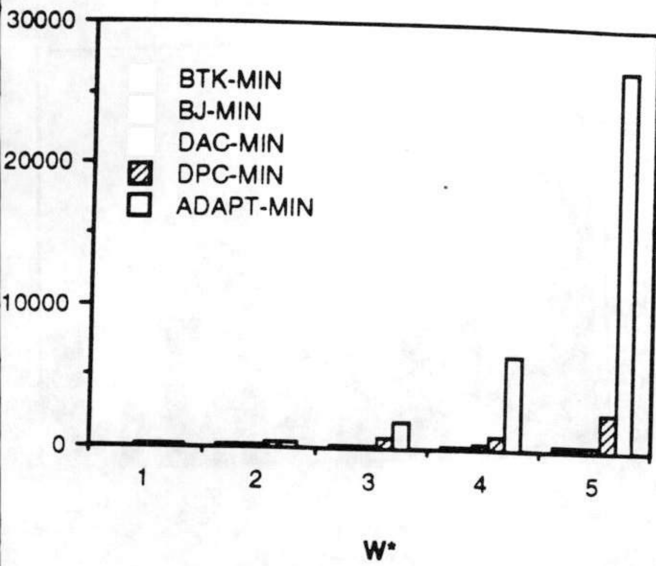
### 5.1 Evaluation of consistency pre-processing algorithms

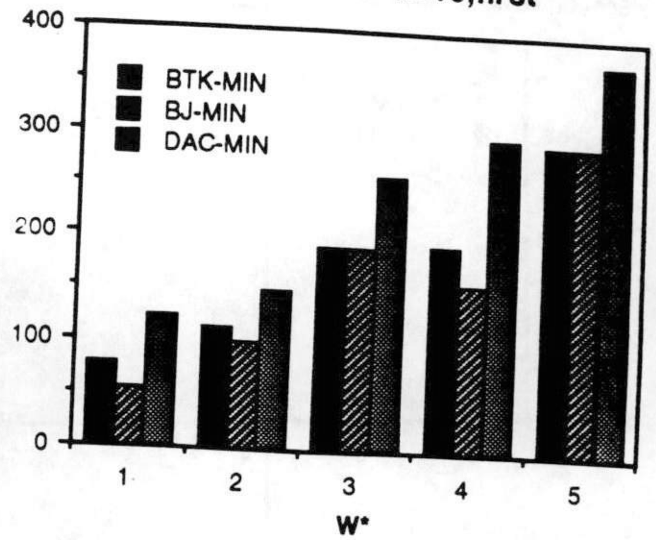
We first report our results in site-1. Our initial goal was to compare the effects of the three pre-processing algorithms *DAC*, *DPC*, and *ADAPT* on *backtracking* and *backjumping*. Figures 7 and 8 present bar-charts displaying the average number of consistency checks, classified according to the width of the induced graph $W^*$ when using the min-width ordering. The first displays results for 10-variable instances, while the second focuses on 15-variable instances. The results for *DEG* and *CARD* were similar, and the ones. for *CARD* are presented in the Appendix (Figures 17,18). Each horizontal pair of graphs presents the average results over a group of instances

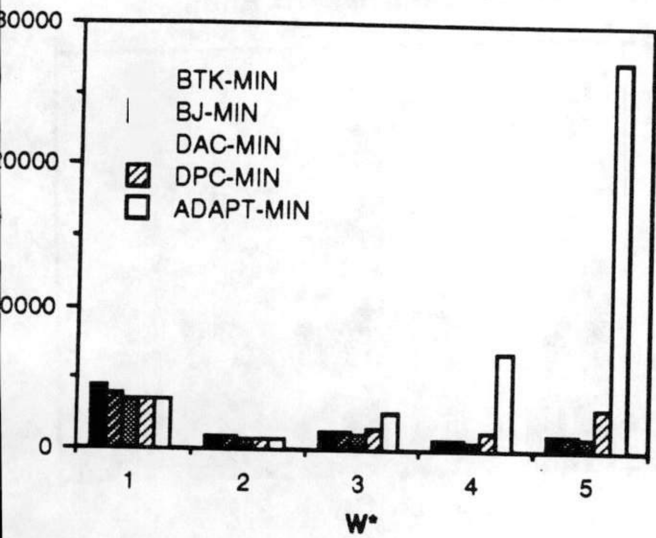
having the same induced width. The left column contrasts the results of all algorithms, however since ADAPT's performance for large $W*$ is so out of scale relative to most other algorithms, we used a different scale for algorithms *backjumping*, *DAC*, and *backtracking* in the right column. The results reported for *DAC*, *DPC*, and *ADAPT* are for the cases where they were complemented by *backjumping*. When using *backtracking*, same behavior was observed, since following consistency enforcing most dead-ends were eliminated.

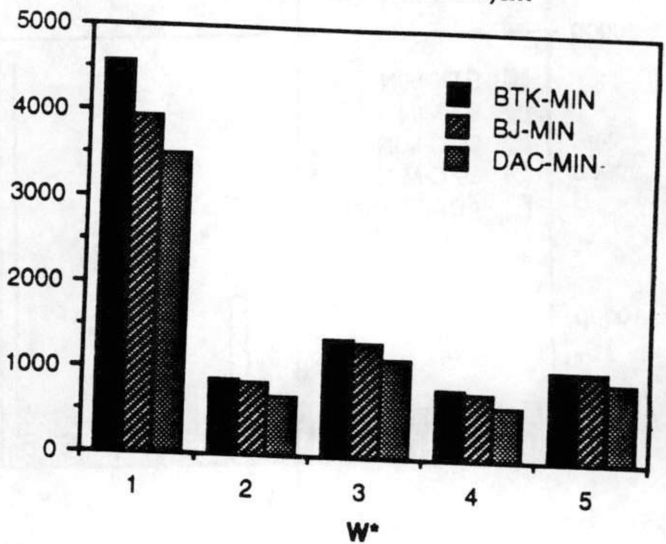
Since algorithms *backtracking*, *backjumping* and *DAC* do not show a monotonic dependence as a function of $W*$ (right columns in figures) we present in Figure 9 a bar-chart comparing their overall average performances for the case of *CARD* ordering. The corresponding results for *MIN* and *DEG* were similar and are given in the Appendix (Figures 19, 20). In Figure 10 we plot separately the performances of all algorithms for small $W*$ to emphasize the dependence of *ADAPT* and *DPC* on this parameter.

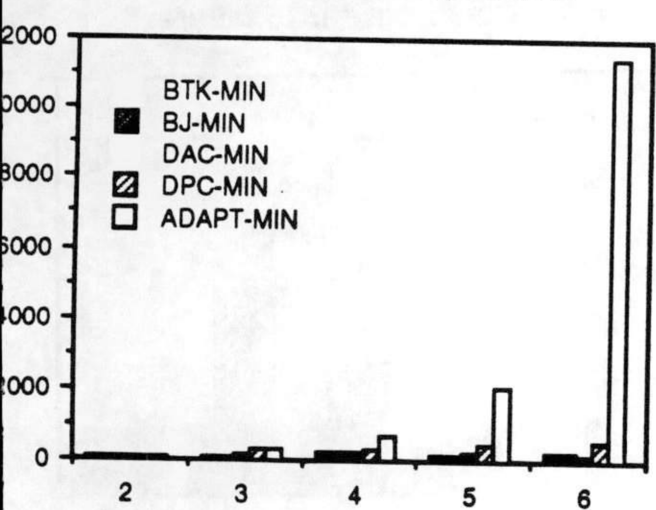
By looking at the left-hand columns of graphs of Figures 7-9, we see that simple *backtracking* generally outperform both *ADAPT* and *DPC*. We see that even on the average *ADAPT* has an exponential behavior as a function of W* and so does *DPC* albeit a weaker one. *Backtracking* and *backjumping*, on the other hand, exhibit a much more moderate, perhaps even linear behavior. However, as can be seen from Figures 7,8 and 10, the average performance of *ADAPT* and *DPC* is better than *BTK* for small values of $W^*$ (for $W*=1$ on 10-variable instances and $W*=2$ for 15 variable instances), mostly for the task of finding all solutions. Evidently, the amount of pre-processing performed by *ADAPT* is generally too heavy to be justified when generating one solution only (see left, upmost graphs), but when it is shared by several solutions, it becomes worthwhile. When comparing *backtracking* with *DPC* we see a clear dominance of *DPC* for the



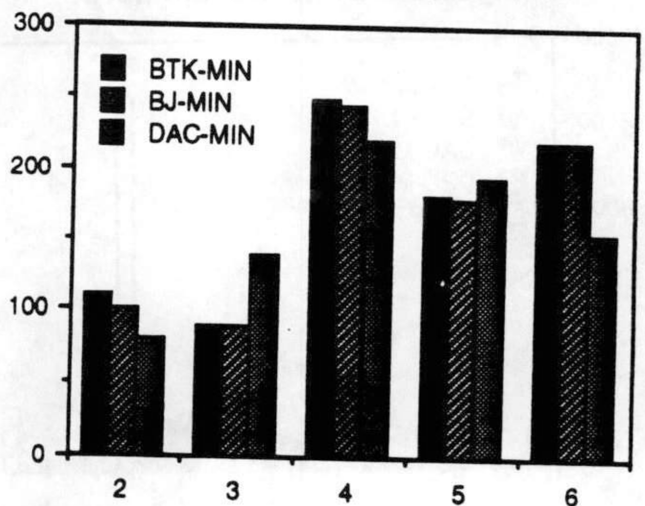
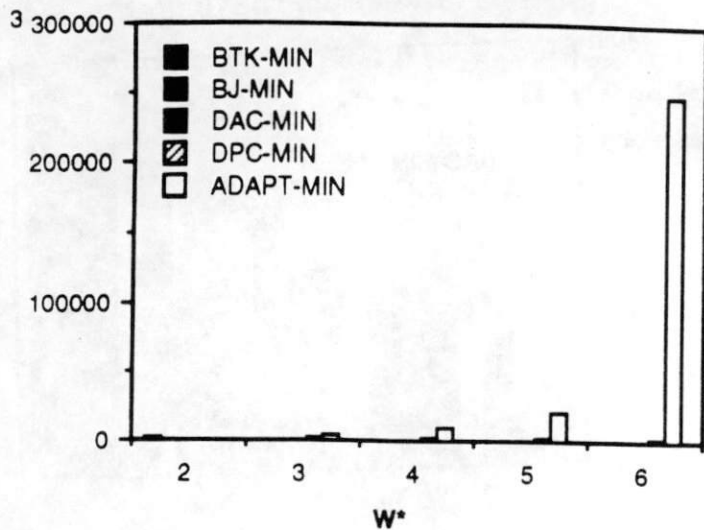
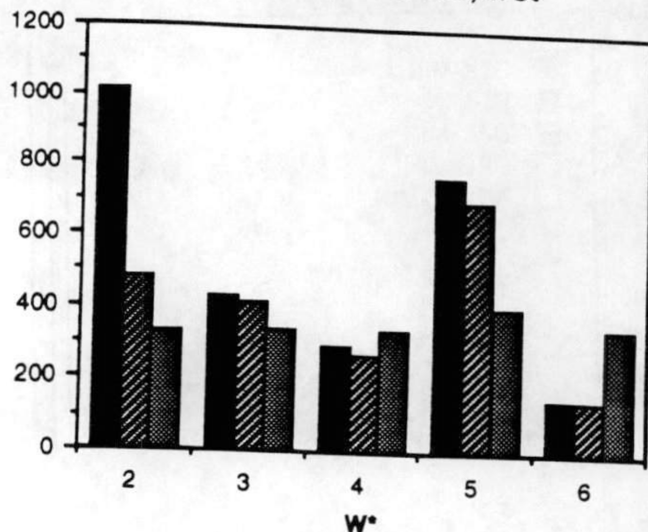
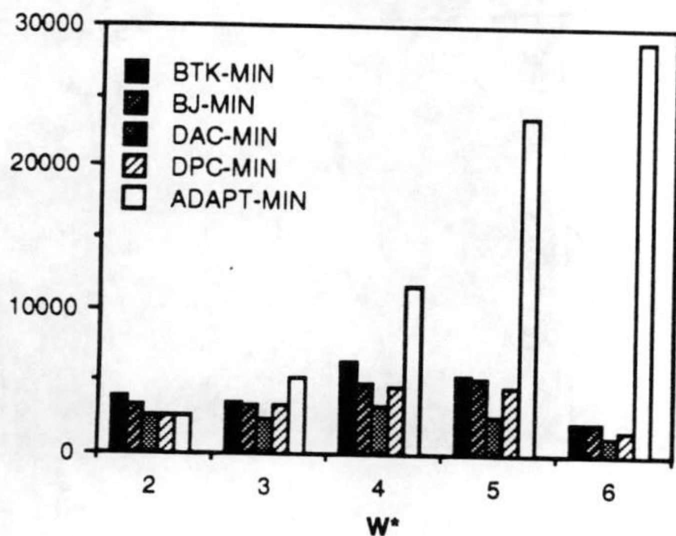
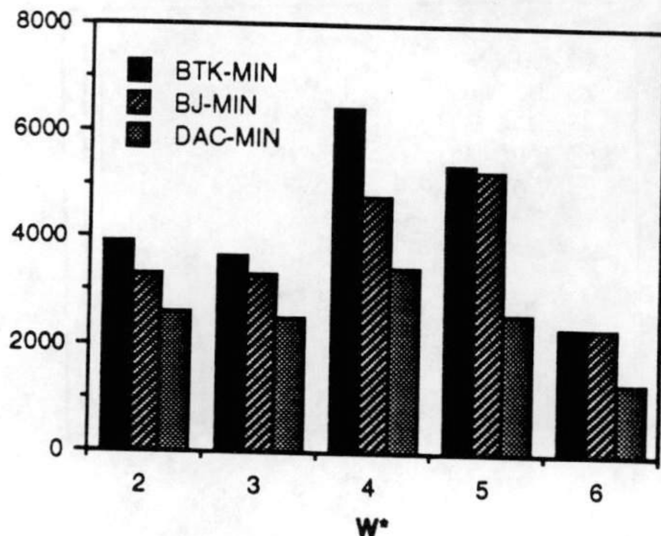



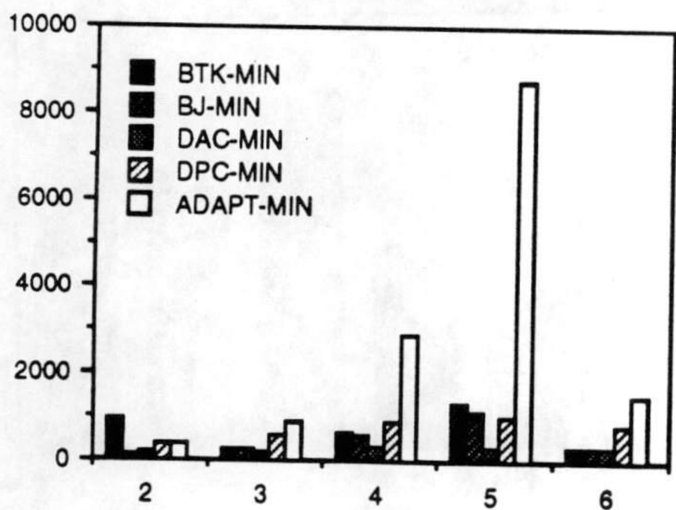

Figure 7: number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with *min–width* ordering on 10-variable, 5-value random problems



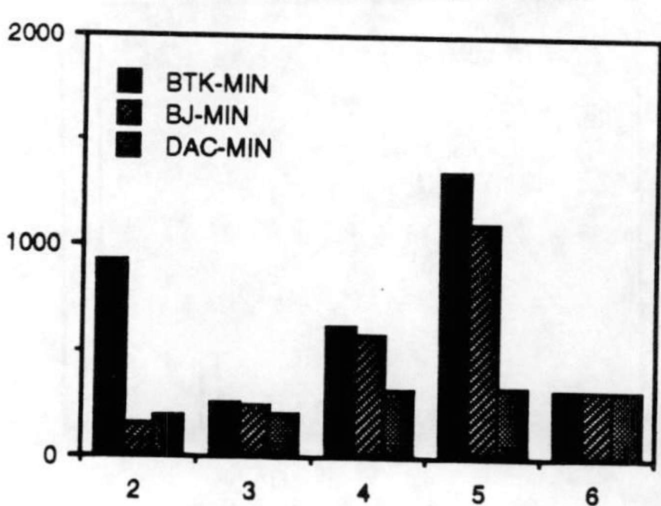
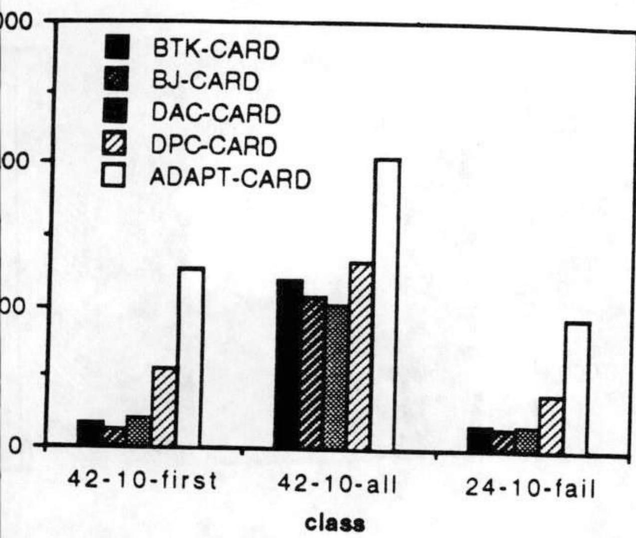
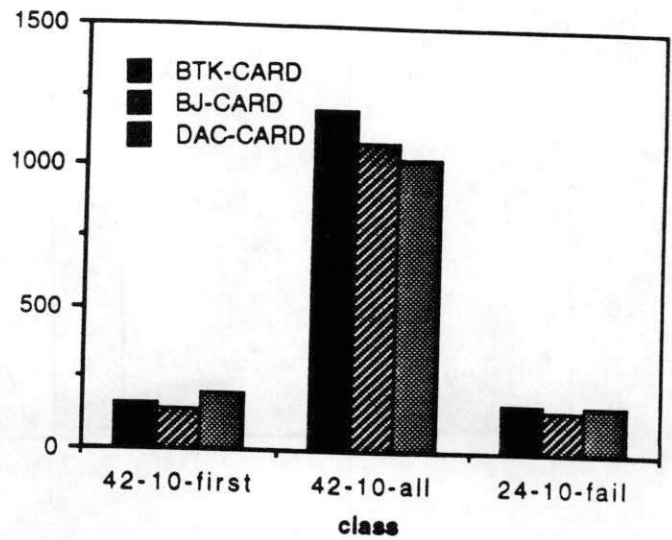
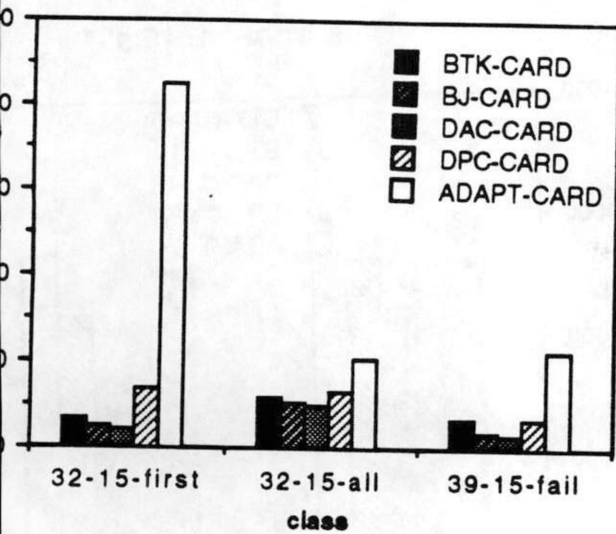




Figure 8: number of consistency checks for pre-processing algorithms *DAC*, *DPC*, *ADAPT*, *backjumpin* and *backtracking* with *min–width* ordering on 15-variable, 5-values random problems.



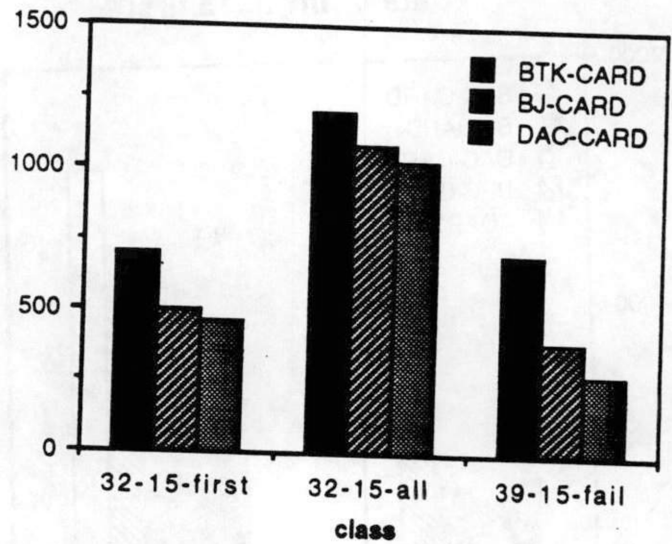
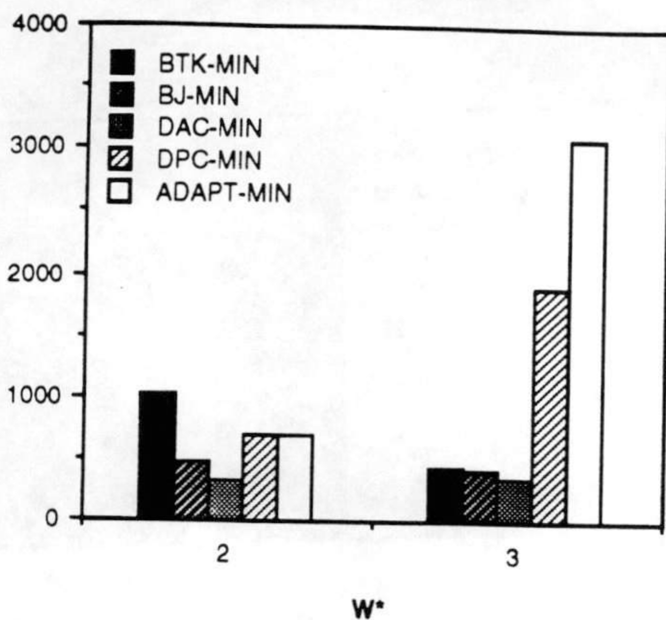
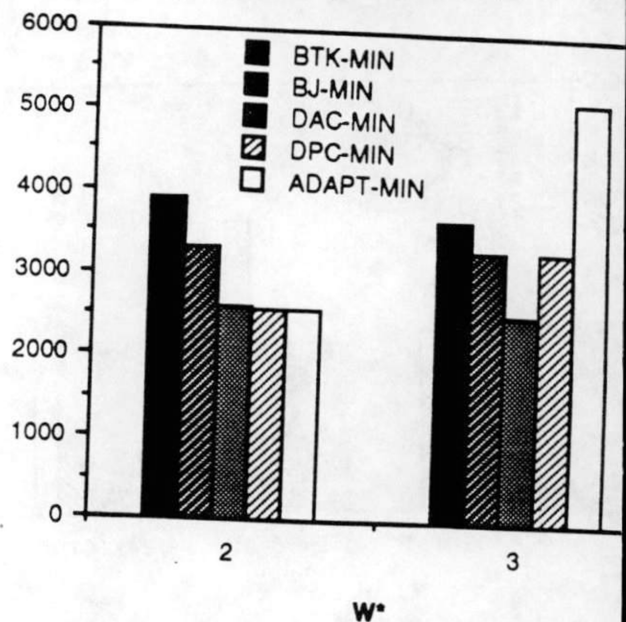
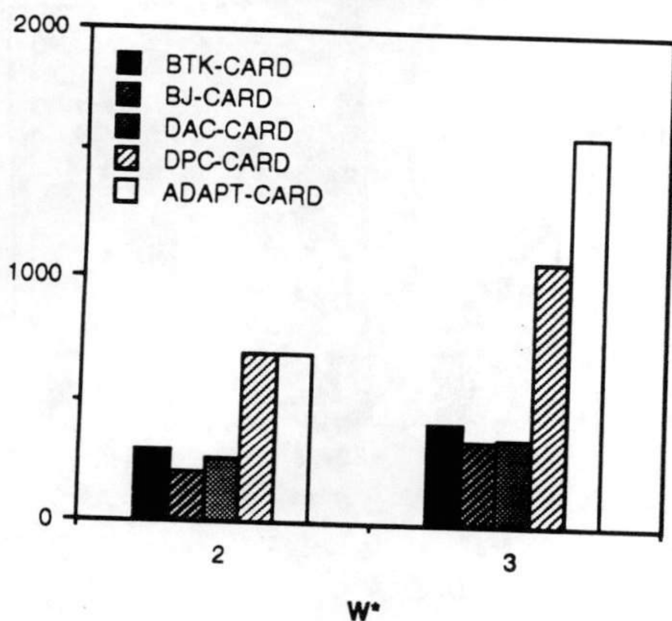




Figure 9: number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with *max−cardinality* ordering on 15-variable random problems, disregarding $W^*$



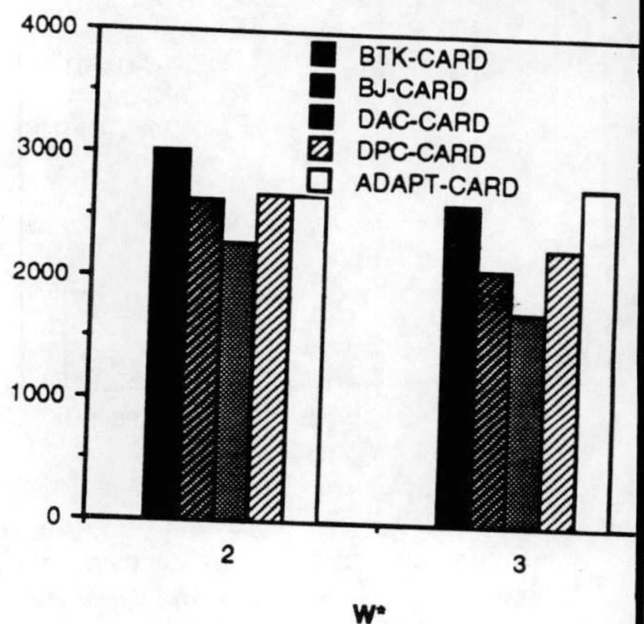



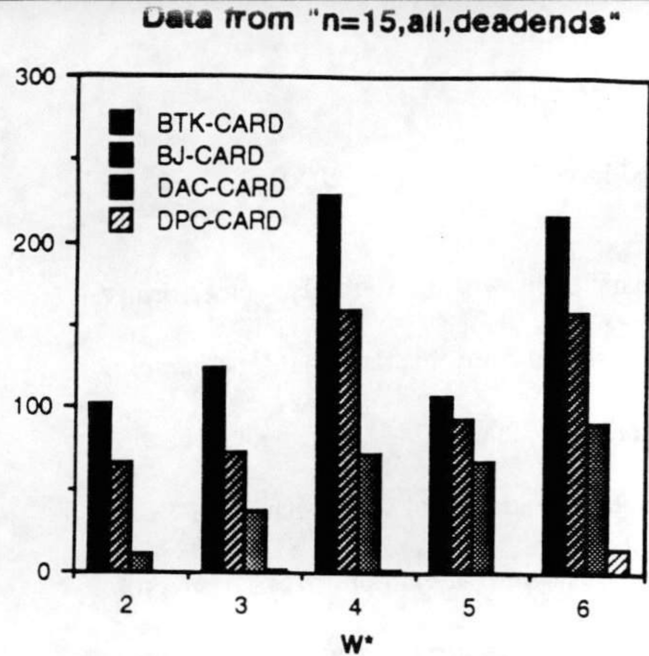

Figure 10: number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with *CARD* and *MIN* orderings for small $W^*$.

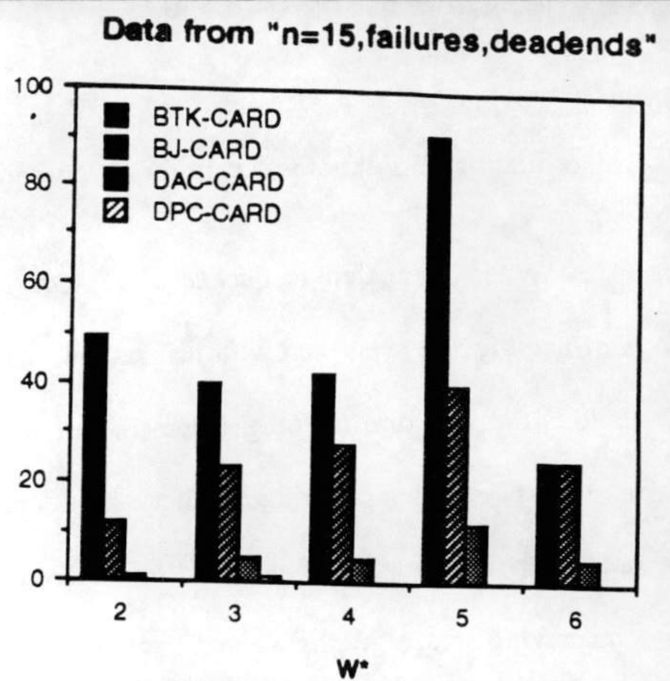
task of finding all solutions, accept for *MIN* ordering and large $W^*$ (Figures 9, 20, 21).

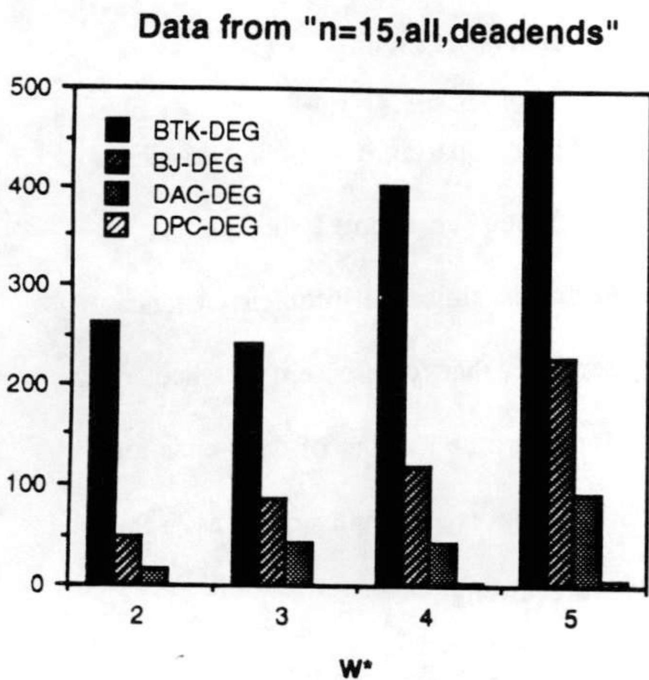
When compared to *backjumping*, *ADAPT* appears even worse. It rarely outperformed *backjumping* and even then with just a small margin. It seems from this data that, *backjumping* exploits the structure of the problem in a more efficient way than *ADAPT* and should be preferred, especially considering the fact that it does not need the additional space that *ADAPT* consumes. It still remains to be tested how ADAPT compares with backjumping for larger problems having sparse graphs when all solutions are required. Algorithm *DPC*, although generally inferior to *backjumping*, it mostly outperformed *backjumping* when $W^*$ is small.

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 11), we found that in almost all problem instances even *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 *DAC* (see Figure 11) shows that a substantial portion of the work is accomplished by even 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 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].



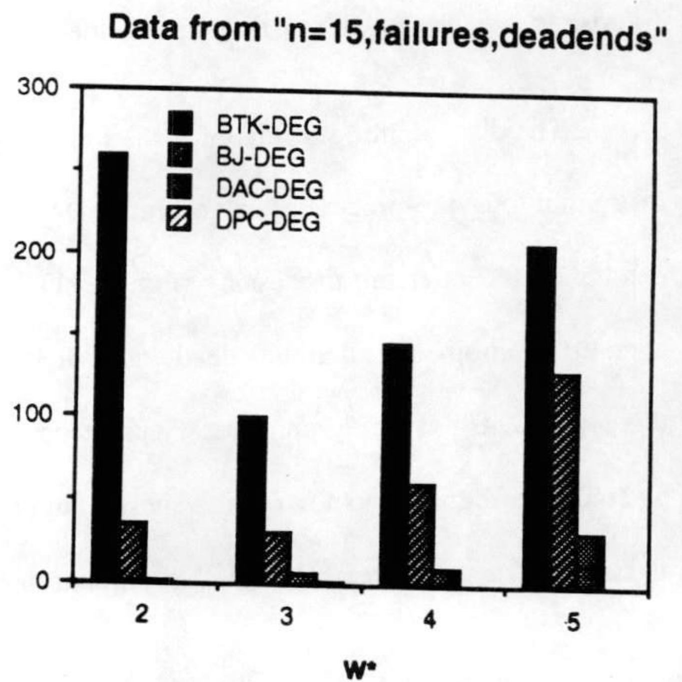




Figure 11: Number of dead-ends for processing algorithms (*DPC*, *DAC*, *BJ* and *BTK*) with *max−card* and *max−degree* orderings on 15-variable random problems

Let us focus now on the right-hand columns of Figure 7-9, which compare *backjumping*, *DAC*, and *backtracking*. Clearly, the two algorithms that stand out in these experiments were *backjumping* and *DAC*. Both outperformed *backtracking* (and *DPC* and *ADAPT*) in almost all cases, but neither 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. Thus, for heavy tasks, the overhead presented by *DAC* will be outweighed by its gain.

### 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, in particular on *backtracking* and *backjumping*. We present results from both sites. Figure 12 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 $W^*$. Again, we group the results according to the problem size (10 or 15 variables) and the three cases (*first*, *all*, *failure*).

Figure 12 shows that the *max–degree* ordering comes out as a complete looser, yet there is no clear winner. Again we observe some patterns in 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 *backtracking* in large problems), *max-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 compare the number of dead-ends associated with each ordering it becomes clear that *DSR* expands the smallest search space (i.e., it has the least number of dead-ends on an instance by instance basis, almost). Therefore, had we better implemented this technique we may have had a better overall performance. Indeed in site-1's implementation of *DSR*, no data structure was used to reduce redundant consistency checks, as is done

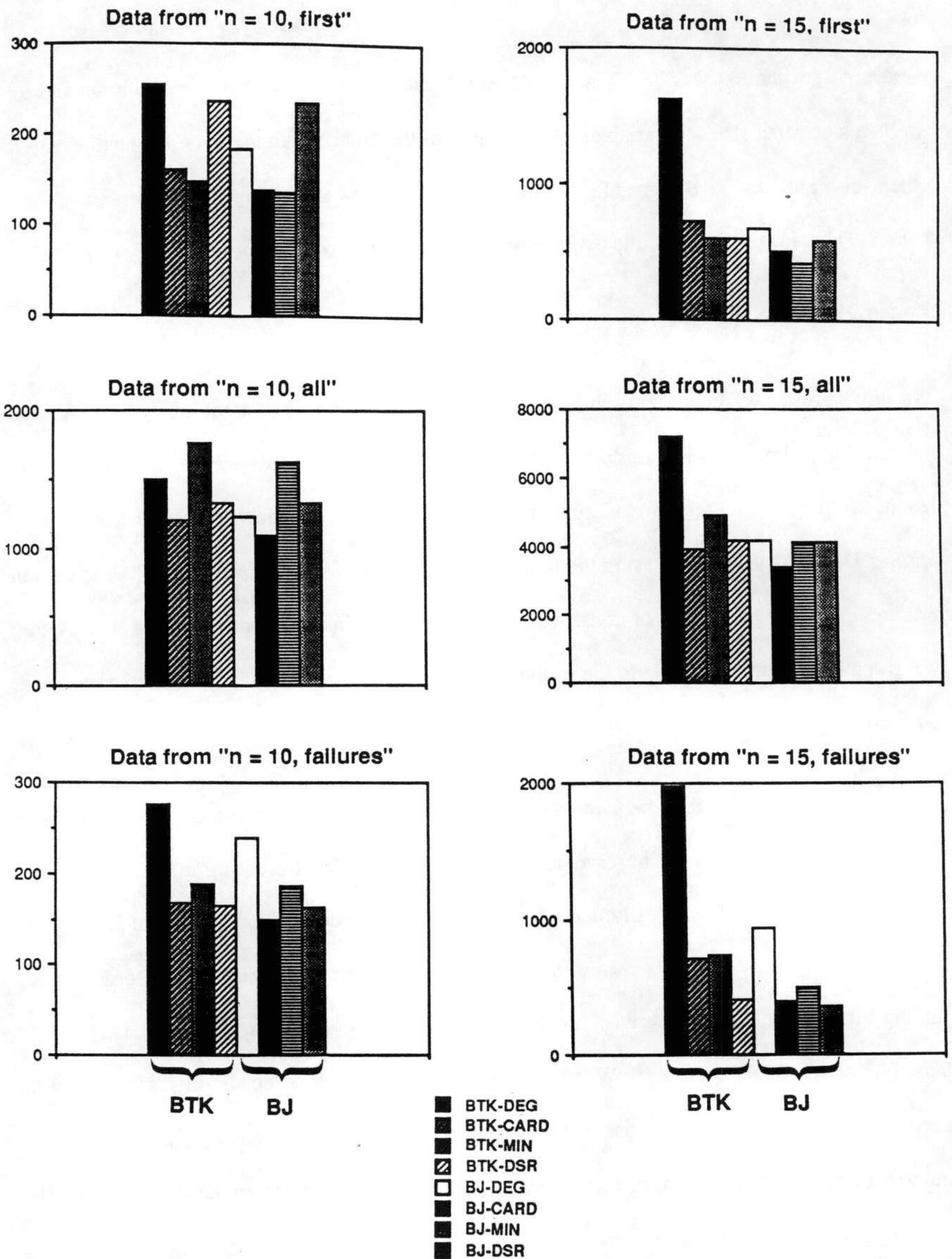



Figure 12 : number of consistency-checks for *BJ* and *BTK* on orderings: *max−degree*, *max−cardinality*, *min−width* and *dynamic* in site-1 for 10 and 15 variable problems.

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, *DSR* was implemented more efficiently, using data structures similar to those used in [Haralick 1980] which 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 13 and 14, in this implementation *DSR* 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.

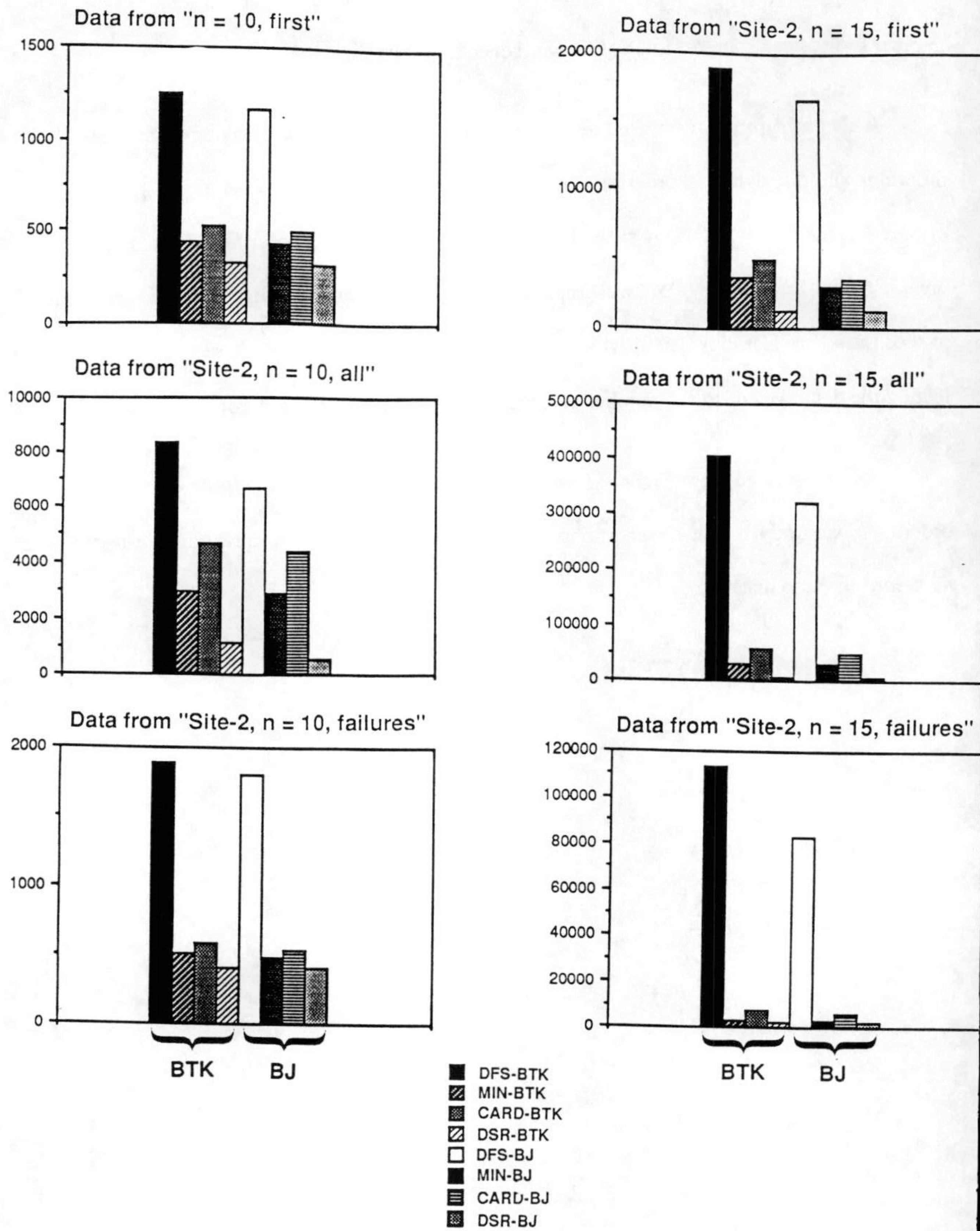



Figure 13 : number of consistency-checks for *BJ* and *BTK* on orderings: *max−degree*, *max−cardina* *min−width* and *dynamic* in site-2 for 10 and 15 variable problems.

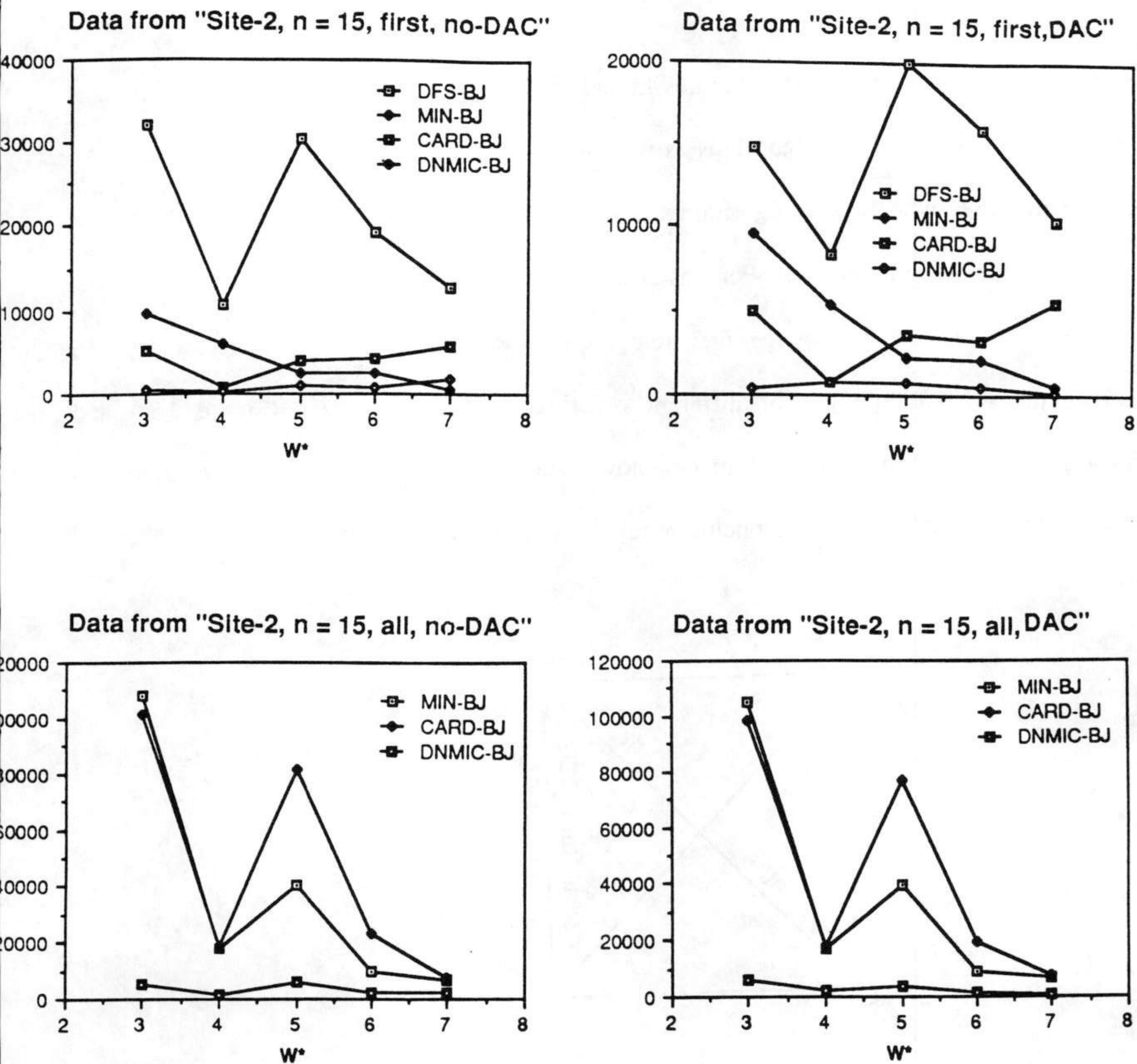



Figure 14: The effect of variable ordering on number of consistency checks parametrized by $w^*$ (with and without pre-processing by DAC) in site-2

## 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 pre-processings on *backtracking* and *backjumping*, and, second, we tested the effect of five variable ordering schemes.

The conclusions are schematized in Figures 15 and 16. These graphical representations summarize the relative merits of the algorithms as reflected by the number of consistency checks. An arrow from *A* to *B* indicates that algorithm *A* is generally superior to algorithm *B*, with the exceptions annotated on the arrows. For instance, Figure 15 indicates that *DAC* outperforms *backjumping* except on finding the first solution in size-10 problems. Similarly, Figure 16 presents the relative strength of different orderings w.r.t. *backtracking* (Figure 16a) and *backjumping* (Figure 16b). 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 "?".

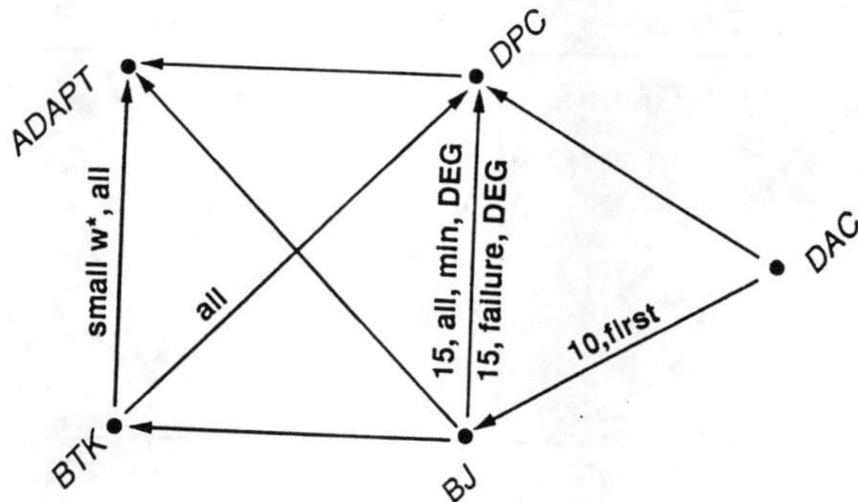



Figure 15: Relative performance of consistency enforcing algorithms

In summary, our experiments indicate that in most cases *DAC* followed by *backjumping* outperforms all other schemes, while *DSR* is still the most promising ordering scheme. When static ordering is imposed, the experiments suggest that combining *DAC* with *min-width* or *max-card* will yield the best results, on the average.

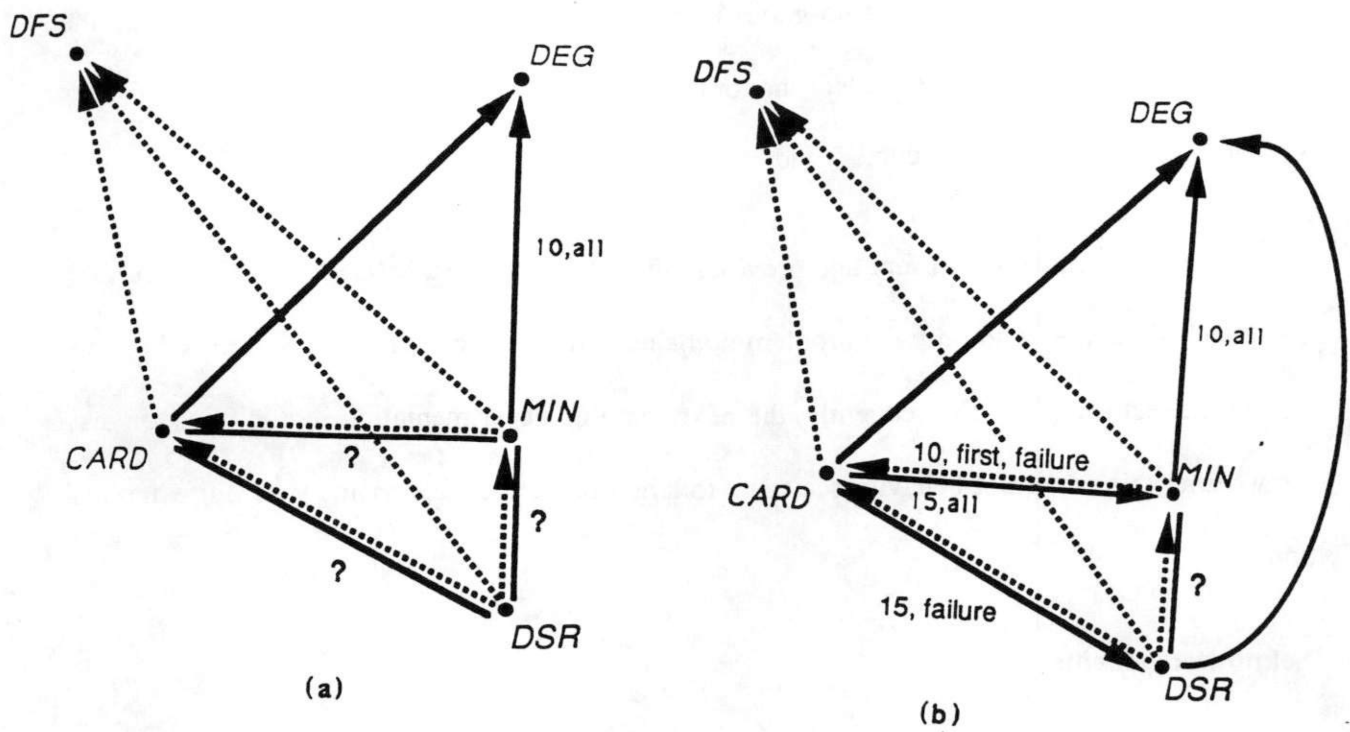



Figure 16: Relative merits of ordering schemes for (a) *backtracking* and (b) *backjumping*

Viewed together with results from prior experiments, our results fit a general pattern. In all techniques tested, be it pre-processing or in-processing, weak enhancement schemes are the most effective (due to their low overhead). Stronger schemes do 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 than the more intensive partial-look-ahead and full-look-ahead (see Figures 6,8,10, and 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 and 4.4.2-2 in [Gaschnig 1979] ). Similarly, in generating heuristics for value selection preferences, only very shallow advice improves the search (see Figures 15 and 16 in [Dechter 1987] ). We found the same phenomenon in assessing various look-back schemes. When augmenting *backjumping* with various levels of

constraint recording (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 (see Figures 7 and 8 in [Dechter 1990] ).

These results, both current and previous, should be qualified before extrapolated further. The conclusions are valid only for 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.

**Acknowledgements**

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 Algebraic and Discrete Methods,,* Vol. 8, No. 2, 1987, pp. 277-184.

[Collin 1991] Collin, Z., R. Dechter, and S. Katz, ''Self stabilizing distributed constraint satisfaction,'' in *Proceedings Ijcai-91,* Sidney, Australia: August, 1991.

[Dechter 1987] Dechter, R. and J. Pearl, ''Network-based heuristics for constraint-satisfaction problems,'' *Artificial Intelligence,* Vol. 34, No. 1, 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 1992] Dechter, R., ''Constraint networks,'' in *Encyclopedia of Artificial Intelligence (2nd ed.),* S. Shapiro, Ed. New York: Wiley, 1992, pp. 276-285.

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

[Freuder 1978] Freuder, E.C., ''Synthesizing constraint expression,'' *Communications 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, 1982, pp. 24-32.

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

[Ginsberg 1990] Ginsberg, M., ''Search lesson learned from crossword puzzles,'' in *Proceedings of the Eighth National Conference on Artificial Intelligence,* Boston, MA: 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, No. 3, 1980, pp. 263-313.

[Hentenryck 1987] Hentenryck, P. Van and M. Dincbas, ''Forward checking in logic programming,'' in *Proc. 4th Int. Conf. on Logic Programming,* J-L Lassez, Ed. MIT Press, 1987, pp. 229-255.

[Mackworth 1992] Mackworth, A., ''Constraint satisfaction,'' in *Encyclopedia of Artificial Intelligence (2nd ed.),* S. Shapiro, Ed. New York: Wiley, 1992.

[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 Sciences,* Vol. 7, No. 2, 1974, pp. 95-132.

[Nudel 1983] Nudel, B., ''Consistent-labeling problems and their algorithms: Expected-complexities and theory-based heuristics,'' *Artificial Intelligence,* Vol. 21, No. 1-2, 1983, pp. 135-178.

[Prosser 1991] Prosser, P., ''Hybrid algorithms for the constraint satisfaction problem,'' Computer Science Dept, I University of Strathclyde, Glasgow, Scotland, Tech. Rep. AISL-46-91, 1991.

[Purdom 1983] Purdom, P., ''Search rearrangement backtracking and polynomial average time,'' *Artificial Intelligence,* Vol. 21, No. 1-2, 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.

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

[Rosiers 1986] Rosiers, W. and M. Bruynooghe, ''Empirical study of some constraint satisfaction algorithms,'' Katholieke Universiteit Leuven, Leuven, Belguim, Tech. Rep. CW 50, 1986.

[Sadeh 1991.] , Sadeh, N. , ''Lookahead Techniques for Activity-based Job-shop Scheduling.,'' in *School of Computer Science, Technical Report No. CMU-CS-91-102,* , Carnegie Mellon University,: 1991..

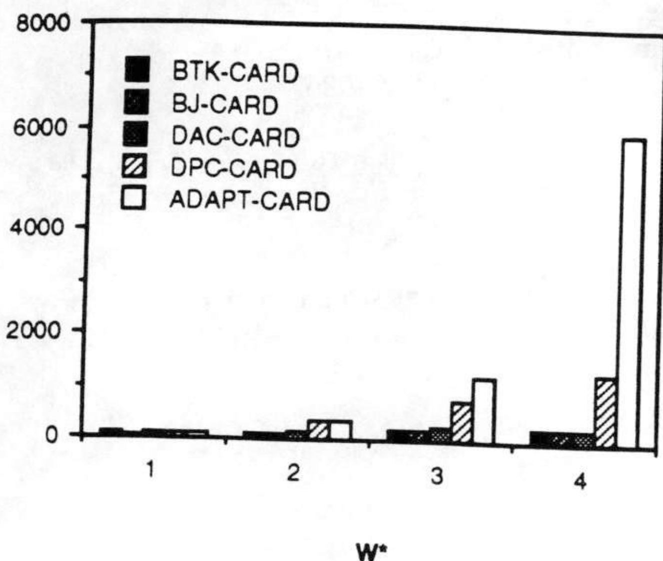
[Stallman 1977] Stallman, R.M. and G. J. Sussman, ''Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis,'' *Artificial Intelligence,* Vol. 9, No. 2, 1977, pp. 135-196.

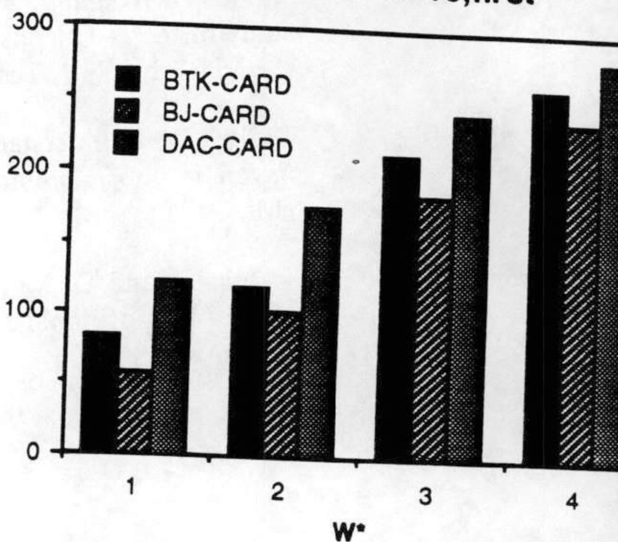
[Stone 1986] Stone, H. S. and J. M. Stone, ''Efficient search techniques- An empirical study of the N-Queens problem.,'' IBM T.J. Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC 12057 (#54343), 1986.

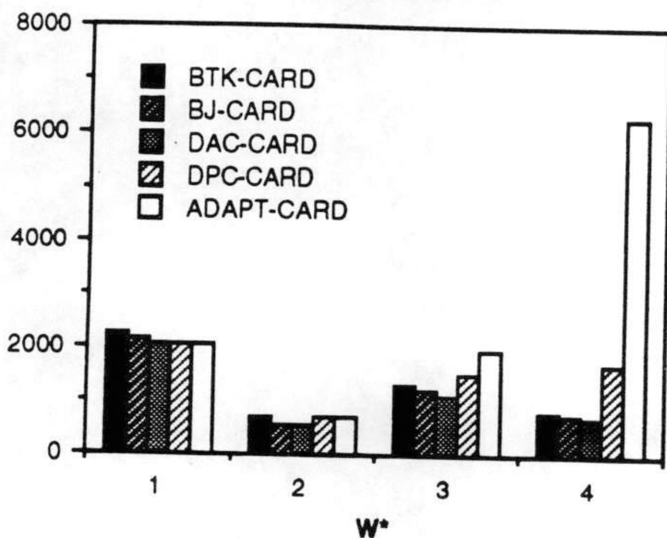
[Waltz 1975] Waltz, D., ''Understanding line drawings of scenes with shadows,'' in *The Psychology of Computer Vision,* P. H. Winston, Ed. New York: McGraw-Hill, 1975.

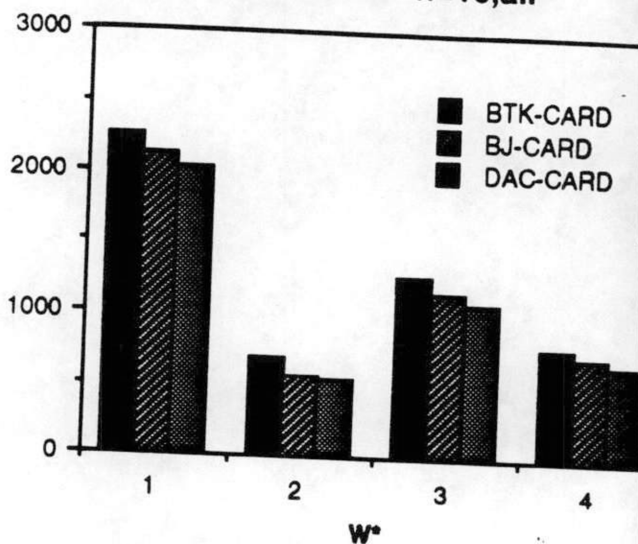
[Zabih 1988] Zabih, R. and D. McAllester, ''A rearrangement search strategy for determining propositional satisfiability,'' in *Proceedings AAAI-88,* St. Paul, Minnesota: 1988.

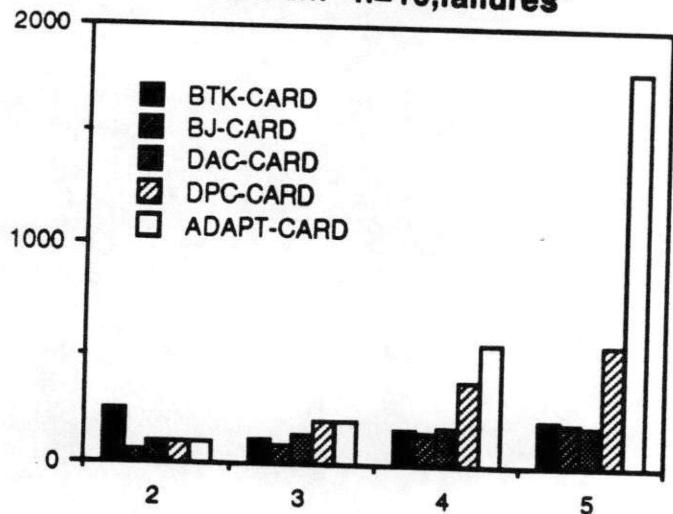
## Appendix: Additional graphs



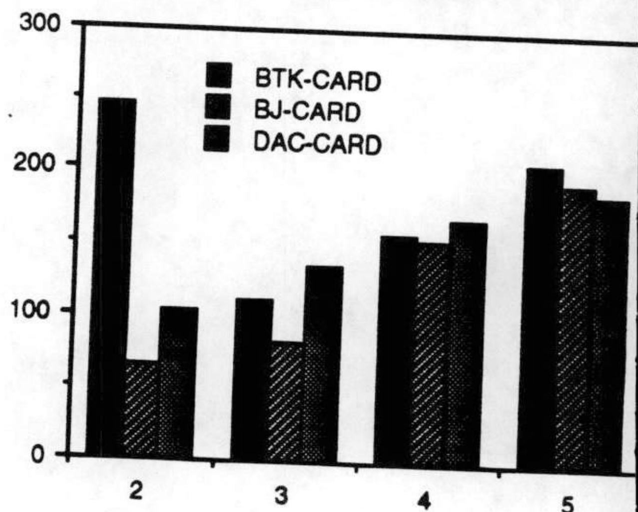
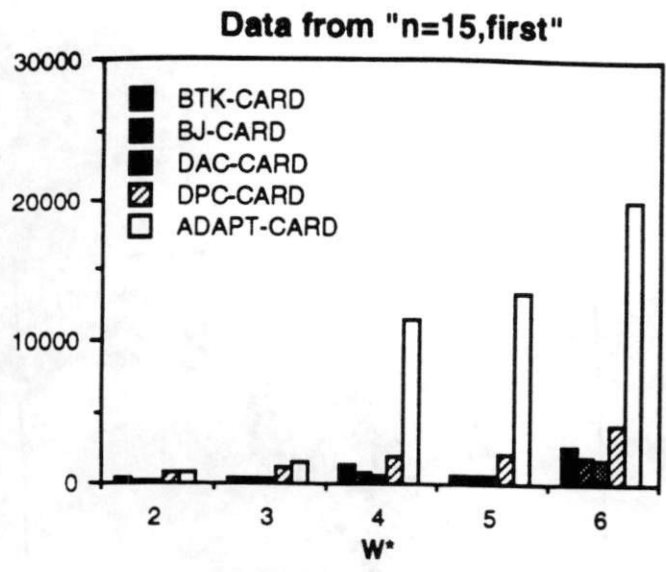
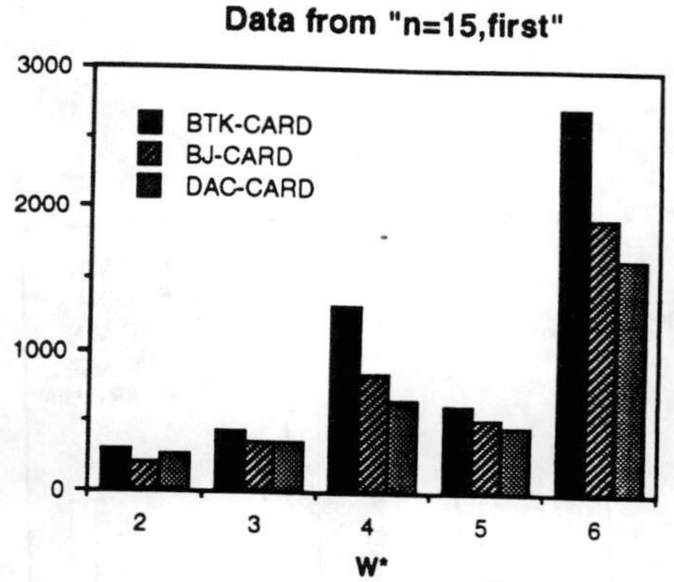
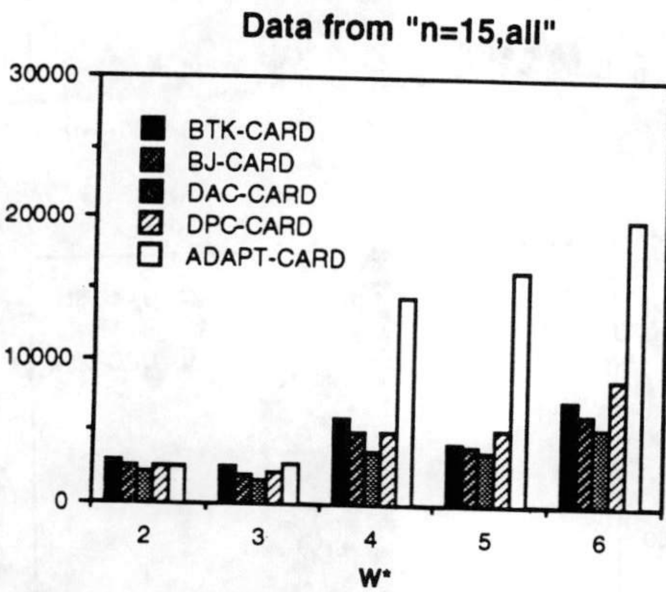
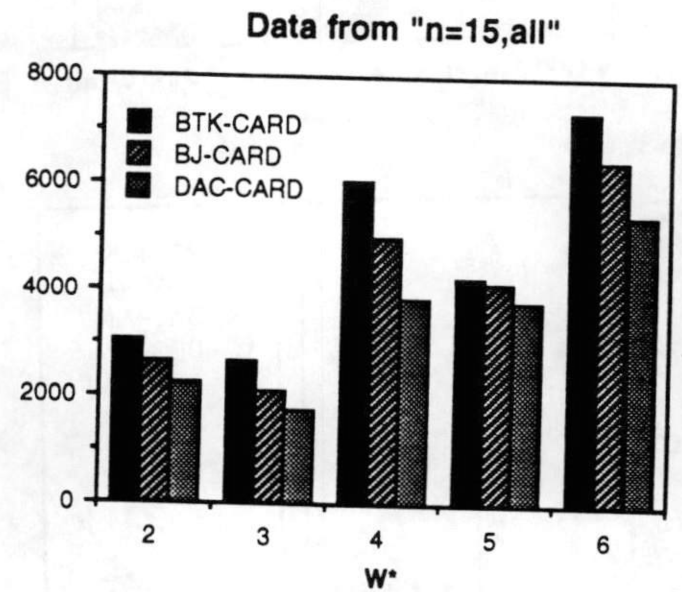



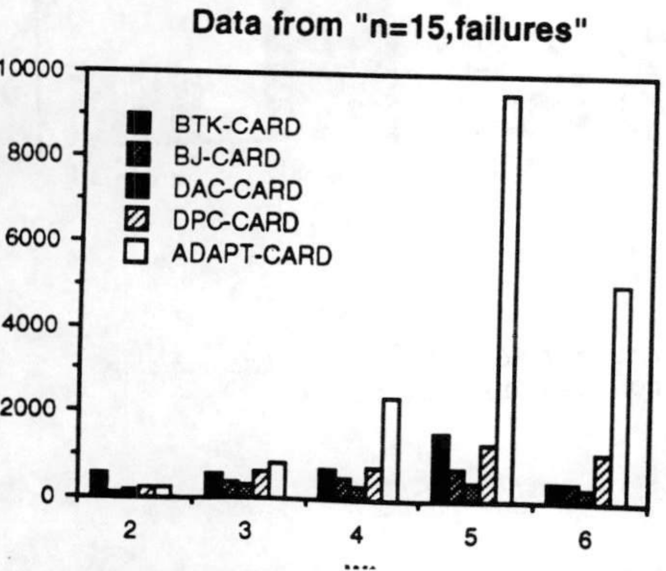

Figure 17: number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with *max−card* ordering on 10-variable random problems



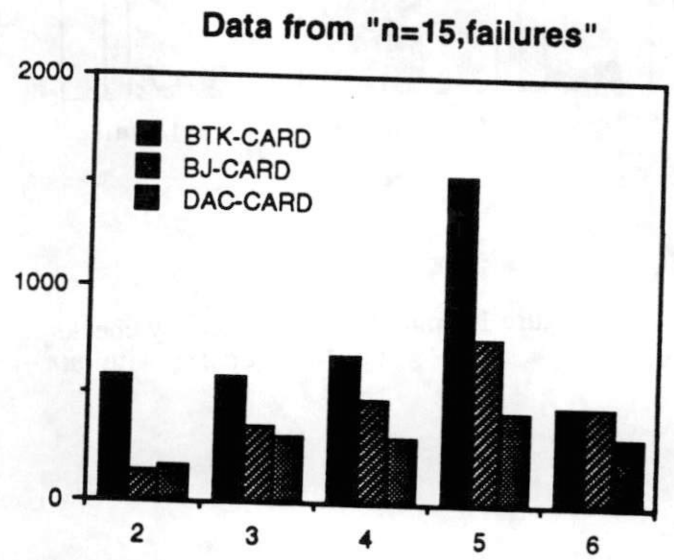




Figure 18: number of consistency checks for pre-processing algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with *max−card* ordering on 15-variable random problems.

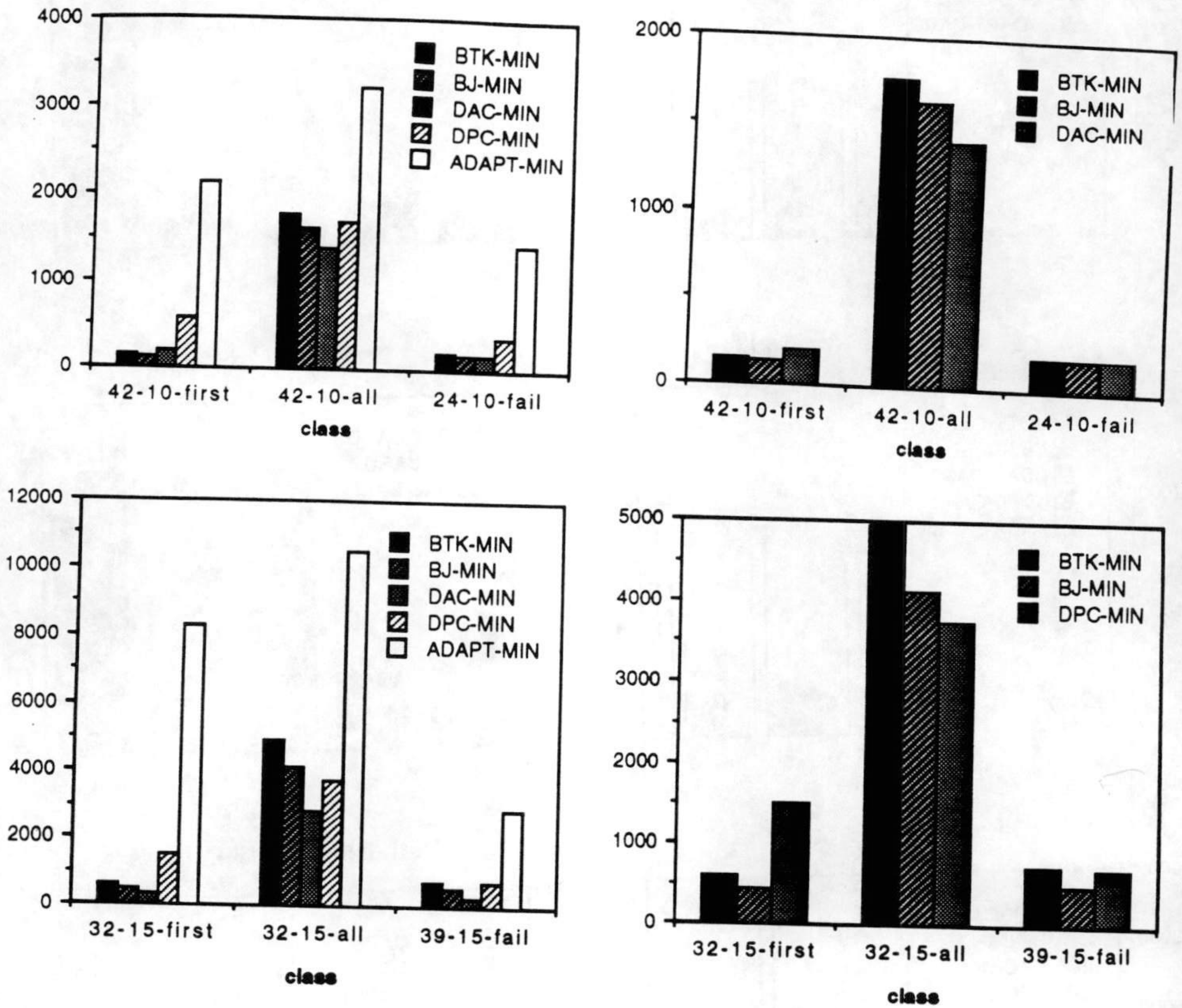
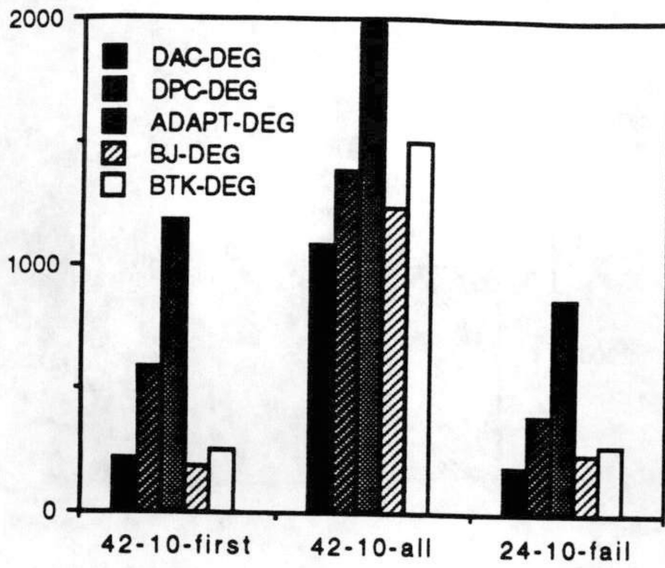
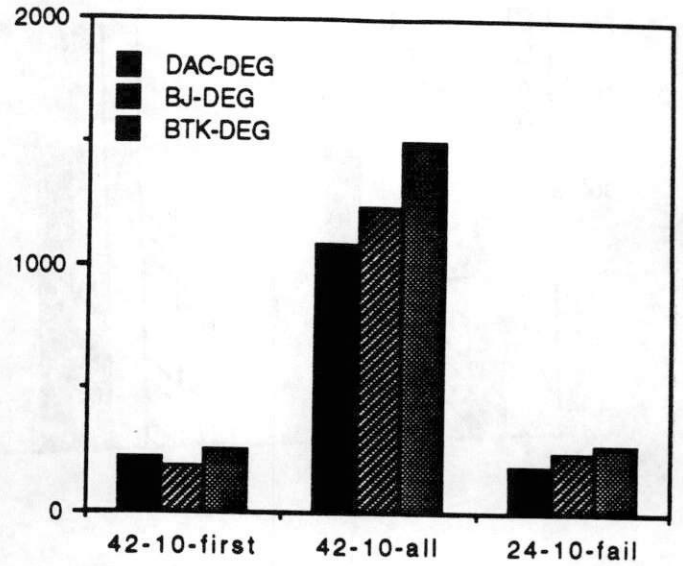
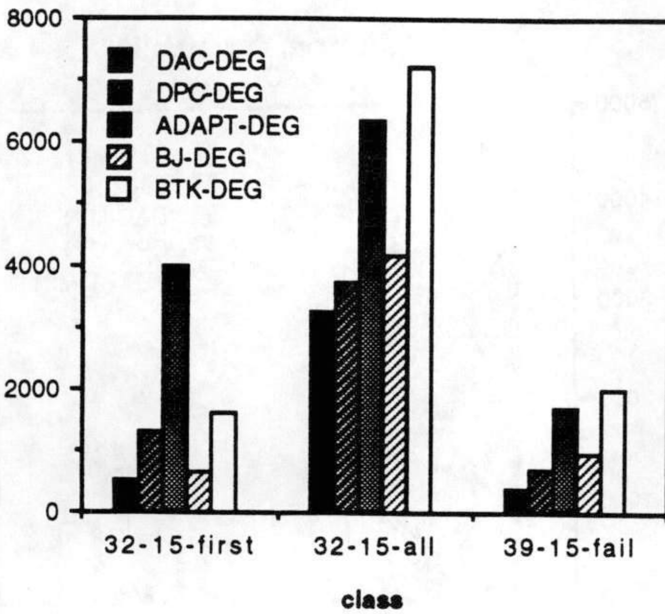



Figure 19: number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with *min−width* ordering, disregarding $W^*$.



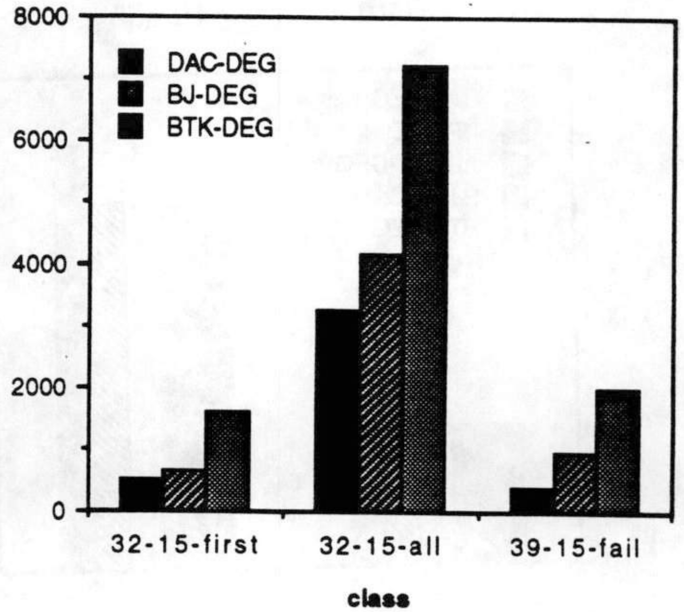
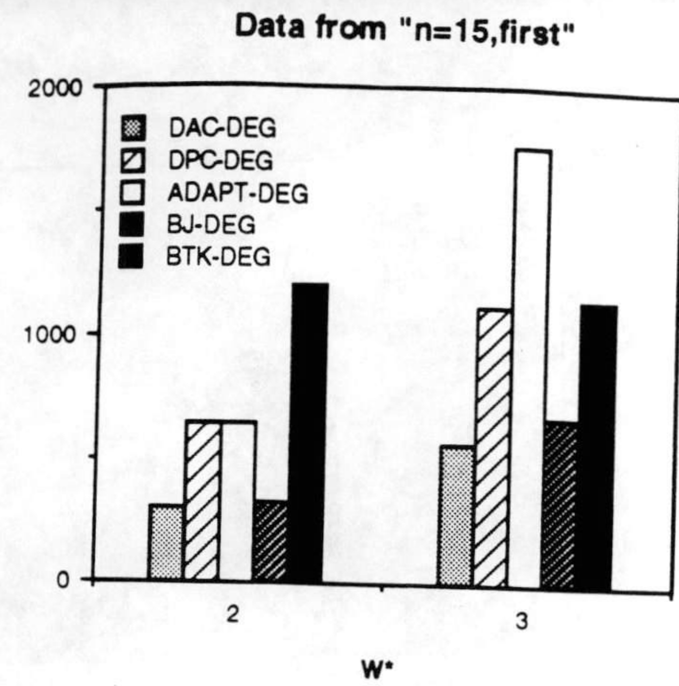
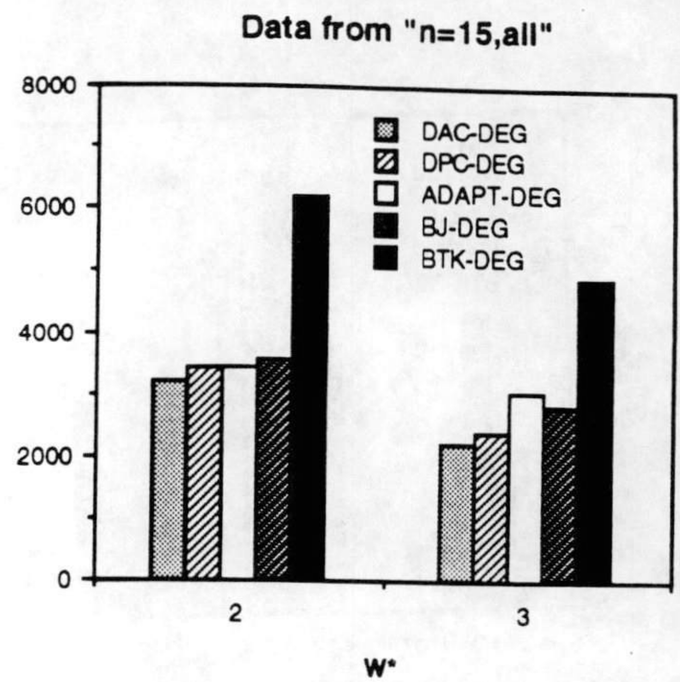
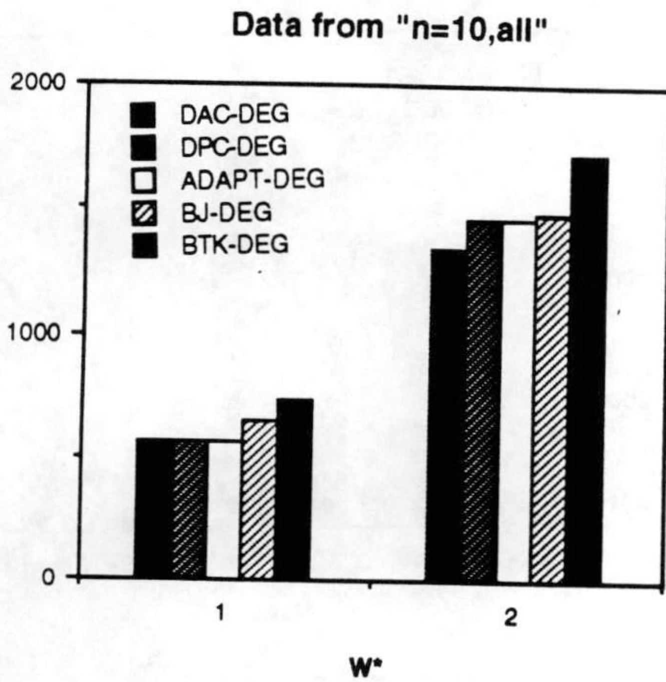




Figure 20: number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with *max−degree* ordering, disregarding *W**



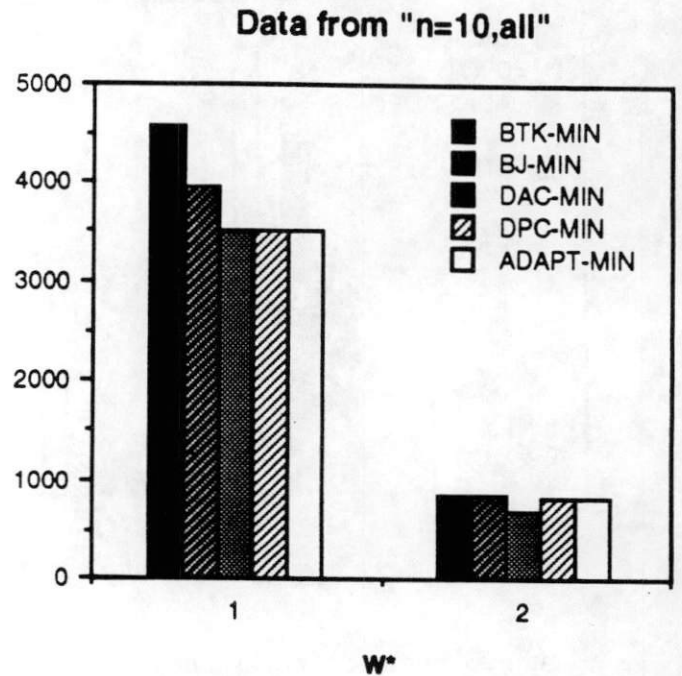




Figure 21: number of consistency checks for algorithms *DAC*, *DPC*, *ADAPT*, *backjumping* and *backtracking* with *max−degree* and *min−width* ordering, for