AN INTEGRATED STRATEGY FOR IMPROVED BACKTRACK

Rina Dechter

February 1987
CSD-870010

TECHNICAL REPORT
R-77
CSD-8700##
January 1987

AN INTEGRATED STRATEGY FOR IMPROVED BACKTRACK* †

Rina Dechter
Cognitive Systems Laboratory
UCLA Computer Science Department
Los Angeles, California 90024

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

AN INTEGRATED STRATEGY FOR IMPROVED BACKTRACK

Rina Dechter

Artificial Intelligence Center
Hughes Aircraft Company
and
Computer Science Department
University of California, Los Angeles
Net address: dechter@locus.ucla.edu
Tel: (213) 825-3243

Paper type: full paper
Track: Science track
Topic area: reasoning.
Keywords: problem solving, learning, constraint-satisfaction, backtracking, cycle-cutset.

ABSTRACT

In previous work researchers in the areas of Constraint-Satisfaction Problems (CSPs) and Prolog had suggested various enhancements to the Naive Backtrack algorithm. Each scheme was presented and tested individually, and the average performances were compared.

The contribution of this paper is in devising a backtrack strategy that integrates three improvement schemes: "**Backjump**", "**Learning while searching**" and the"**cycle-cutset method**"; Backjump and the cycle-cutset method work best when the constraint-graph is sparse, while the learning scheme mostly benefits problem instances with dense constraint graphs. The **Integrated-strategy** proposed here lets each scheme dominates when instances favorable to its performance are presented and makes them cooperate on intermediate instances. The experiments show that, in hard problems, the average improvement realized by the integrated scheme is by 20-25 % higher then any of the individual schemes.

1. Introduction

The extensive use of backtrack as the main control mechanism in many AI programs has prompted researchers to suggest various schemes for enhancing its performance. Each scheme was presented and tested individually, and comparisons among the different schemes were based on average performance [Haralick 1980, Gaschnig 1979a, Bruynooghe 1984]. However, no attempt has yet been made to integrate several kinds of improvements into one algorithm or, discuss the trade-offs involved in such effort, or even characterize the domain of instances on which each scheme will work best.

The contribution of this paper lies in integrating three improvement schemes named "**Backjump**", "**Learning while searching**" and the **cycle-cutset method**"; the first being a graph-based simplification of a method proposed by Gaschnig [Gaschnig 1979b] while the later two were proposed by the author [Dechter 1986, Dechter 1987.] and evaluated individually. Graph-based backjump and the cycle-cutset method work best when the constraint-graph is sparse, while the learning scheme mostly benefits problem instances with dense constraint graphs. In order to exploit the distinct merits of each scheme it is necessary to insure the continued influx of information that each component would receive were it to operate alone. The **Integrated-Backtrack** proposed here lets each scheme dominates when instances favorable to its performance are presented and makes them cooperate on intermediate instances.

The paper is organized as follows: Section 2 presents definitions and preliminaries. Sections 3,4, 5 and 6 summarize the three schemes and their individual performances, section 7 presents the **integrated-Backtrack** scheme and describes a set of experiments for evaluating it, and section 8 contains concluding remarks.

2. Definitions and Preliminaries

A constraint satisfaction problem involves a set of n variables $X_1,...,X_n$, each represented by its domain values, $R_1,\ldots,R_n$, and a set of constraints. A constraint $C_i(X_{i_1},\cdots,X_{i_j})$ is a subset of the Cartesian product $R_{i_1}\times\cdots\times R_{i_j}$ that specifies which values of the variables are compatible with each other. A solution is an assignment of values to all the variables which satisfy all the constraints, and the task is to find one or all solutions. A constraint is usually represented by the set of all tuples permitted by it. A **Binary CSP** is one in which all the constraints are binary, i.e., they involve only pairs of variables. A binary CSP can be associated with a **constraint-graph** in which nodes represent variables and arcs connects pairs of variables which are constrained explicitly. Consider, for instance, the CSP presented in figure 1 (from [Mackworth 1977]). Each node represents a variable whose values are explicitly indicated, and the constraint between connected variables is a strict lexicographic order along the arrows.

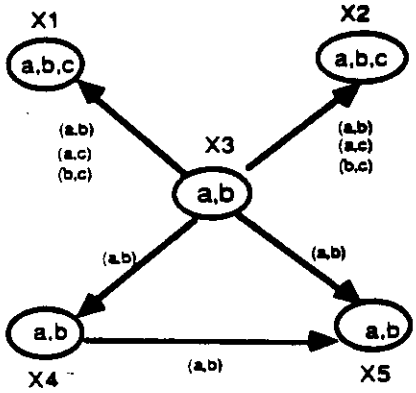

Figure 1: An example CSP

The search space associated with a CSP has states being consistent assignments of values to subsets of variables. A state $(X_1=x_1,\ldots,X_i=x_i)$ can be extended by any consistent assignment to any of the remaining variables. The states in depth n which are consistent represent solutions to the problem, namely n-tuples satisfying all the constraints. If the order by which variables are instantiated is fixed, then the search space is limited to contain only states in that specific order. The efficiency of various search algorithms is determined by the size of the search space they visit and the amount of computation invested in the generation of each state. It is common to evaluate the performance of such algorithms by the number of consistency checks they make rather than the size of the search space they explicate, where a consistency check occurs each time the algorithm query about the consistency of any two values.

The enhancements to backtrack described in this paper are given in terms of binary CSPs using the concept of constraint-graph but they are not limited to this class. By associating hyper-graph with non-binary CSPs all these schemes can be generalized.

3. Backjump

The idea of going back several levels up in a dead-end situation, rather then going back to the chronologically most recent decision made, is first mentioned by Gaschnig [Gaschnig 1979b] who also gave the name for this method. Most recent schemes for improving backtrack's performance like **dependency-directed-backtrack** [Doyle 1979] in Truth-maintenance systems, and **intelligent backtrack** in Prolog [Bruynooghe 1984] are variations of that idea. Gaschnig's algorithm uses a marking technique that summarizes previous consistency checks and utilizes this information in dead-ends. Specifically, for each value that failed instantiation the algorithm records the furthest level with which that value was incompatible, so in case of a dead-end variable, it jumps back to the most recent among the levels recoreded. Although this scheme retains only one bit of infor-

mation with each variable, it requires an additional computation with each consistency check.

Graph-based-Backjump, extract knowledge about dependencies from the constraint-graph. Whenever a dead-end occur at a particular variable, the algorithm backs up to the most recent variable connected to it in the graph. In that way, the additional computation at each consistency check is saved at the expense of a less refined information about the potential cause of the dead-end. For example, if the search on the problem in figure 1 is performed in the order X_3, X_4, X_1, X_2, X_5, then when a dead-end occurs at X_5 the algorithm will jump back to variable X_4 since X_5 is not connected to either X_2 nor X_1.

4. Learning while searching

Each time the algorithm encounters a dead-end situation it has an opportunity to learn, or **explicate** a constraint. Whenever the current state $S = (X_1 = x_1, \ldots, X_{i-1} = x_{i-1})$ cannot be extended by **any** value of X_i, we say that S **is in conflict** with X_i or, in short, that S is a **conflict-set**. An obvious constraint that can be learned at that point is one that prohibits the set S. Recording this constraint, however, is of no help since under the backtrack control strategy this state will never re-occur. If, on the other hand, the set S contains one or more subsets which are also in conflict with X_i, then recording this information in the form of new explicit constraint might prove useful in future search.

In the process of identifying smaller conflict-subsets we first remove from S all the instantiations which are irrelevant to X_i, i.e. those that do not constraint any value of X_i. Recording this set, named **Conf-set**, as a new constraint is called **shallow learning** since, on one hand its discovery doesn't require much effort and on the other hand many more potentially explicable constraints may be overlooked. The explication of these

constraints is referred to as **deep-learning**, and is performed by identifying subsets of the Conf-set which are still in conflict with X_i, in particular those which are **minimal**, namely which do not contain any conflict-set.

Consider again the problem in figure 1. Suppose that the backtrack algorithm is currently at state $(X_1 = b, X_2 = b, X_3 = a, X_4 = b)$. This state cannot be extended by any value of X_5 since none of its values is consistent with all the previous instantiations. As pointed out above, there is no point recording this tupple as a constraint among the four variables involved and smaller constraints should be looked for. A closer look reveals that the instantiation $X_1 = b$ and $X_2 = b$ are both irrelevant in this conflict simply because there is no explicit constraint between X_1 and X_5 or between X_2 and X_5. Neither $X_3 = a$ nor $X_4 = b$ can be shown to be irrelevant and, therefore, the Conf-set is $(X_3 = a, X_4 = b)$. This could be recorded by eliminating the pair (a,b) from the set of pairs permitted by $C(X_3,X_4)$. This Conf-set is not minimal, however, since the instantiation $X_4 = b$ is, by itself, in conflict with X_5. Therefore, it would be sufficient to record this information only, by eliminating the value b from the domain of X_4.

Discovering all minimal conflict-sets amounts to acquiring all the possible information out of a dead-end. Yet, such deep learning requires an exponential time and storage space and as an alternative we proposed several schemes of **controlled learning**.

Independently of the depth of learning chosen, one may restrict the **dimensionality** of the constraints actually recorded, i.e, the number of variables recorded. Constraints involving only a small number of variables require less storage and have a better chance of prunning future search than constraints with many variables. We can record only conflict-sets consisting of a single instantiation, by simply eliminating a value from the domain of the variable, which is referred to as **first-order-learning**. First-order learning does not increase the storage of the problem beyond the size of the input and it

prunes the search each time the deleted value is a candidate for assignment.

Second-order learning is performed by recording conflict-sets involving only one or two variables. Since not every pair of variables appear as a constraint in the initial representation (e.g. when all pair of values are permitted nothing is written), second-order learning may increase the size of the problem, but still in a manageable amount. In general, an **i^{th}-order learning** algorithm will record every constraint involving i or less variables. Obviously, as i increases, the amount of analysis required increases and also storage increases.

The additional storage required for higher order learning can be avoided, however, by further restricting the algorithm to only **modify** existing constraint without creating new ones. This approach does not change the structure of the constraint-graph associated with the problem, a property which is sometimes desirable [Dechter 1985]. We therefore include this option into the various learning scheme and we can either **add** constraint or only **modify** existing ones. For a detailed description of the learning schemes see [Dechter 1986].

When deep learning is used in conjunction with restricting the order of learning we get **deep first-order learning** (identifying minimal conflict sets of size 1), **deep second-order learning** (i.e. identifying minimal conflict-sets of sizes 1 and 2), and in general **deep i^{th}- order -learning**. The combination of the three learning parameters: depth (deep vs shallow), order (first-order vs second-order), and the possibility of "add" or "modify" only yield 8 learning schemes.

5. The cycle-cutset method

The cycle-cutset method is based on two facts; one is that tree-structured CSPs can be solved very efficiently [Dechter 1985], and the other is that variable instantiation

changes the effective connectivity of the constraint graph. In Figure 1, for example, instantiating X_3 to some value, say a, renders the choices of X_1 and X_2 independent of each other as if the pathway $X_1 - X_3 - X_2$ were "blocked" at X_3. Similarly, this instantiation "blocks" the pathways $X_1 - X_3 - X_5$, $X_2 - X_3 - X_4$, $X_4 - X_3 - X_5$ and others, leaving only one path between any two variables. The constraint graph shown in Figure 2a reflects this situation, where the instantiated variable, X_3, is duplicated for each of its neighbors.

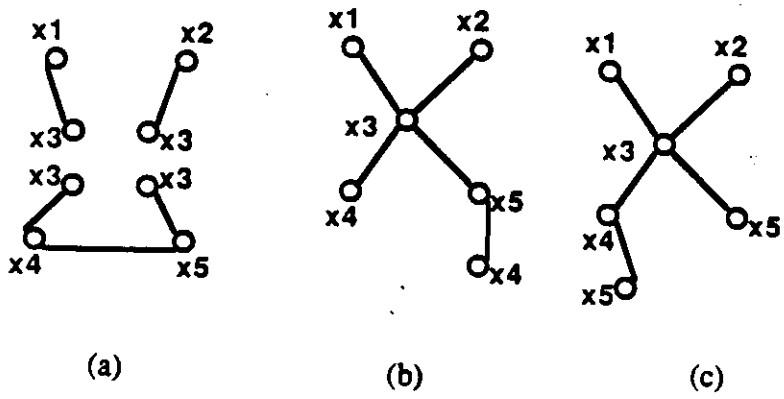

Figure 2: An instantiated variable cuts its own cycles.

When the group of instantiated variables constitute a cycle-cutset, the remaining network is cycle-free, and the efficient algorithm for solving tree-constraint problems is applicable. In the example above, X_3 cuts the single cycle $X_3 - X_4 - X_5$ in the graph, and the graph in Figure 2a is cycle-free. Of course, the same effect would be achieved by instantiating either X_4 or X_5, resulting in the constraint-trees shown in Figure 2b and 2c. In most practical cases it would take more than a single variable to cut all the cycles in the graph (see Figure 3).

One way of exploiting the simplicity inherent in tree-structured problems works as follows: To solve a problem whose constraint graph contains cycles, instantiate the

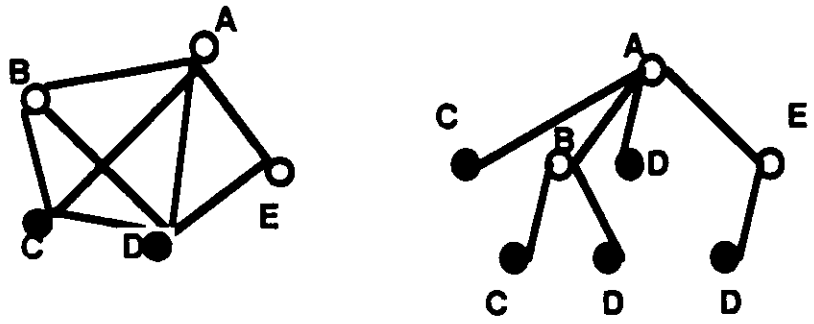

Figure 3: A constraint graph and a constraint-tree generated by the cutset {C,D}

variables in a cycle-cutset in a consistent way and solve the remaining tree-structured problem. If a solution to the restricted problem is found, then a solution to the entire problem is at hand. If not, consider another instantiation of the cycle-cutset variables and continue. Thus, if we wish to solve the problem in Figure 1, we first assume $X_3 = a$ and solve the remaining problem. If no solution is found, then assume $X_3 = b$ and try again. Since the tree-CSP can be solved in $O(nk^2)$, the cycle cutset method can bound the worst-case solution of CSPs to $O(c^k)$, when c is the size of the cutset.

This version of the cutset method is practical only when the cycle-cutset is very small because, in the worst case, we may examine all consistent instantiations of the cycle-cutset variables, the number of which grows exponentially with the size of the cutset.

A more general version of the cycle-cutset method would be to incorporate it within a backtrack algorithm, i.e., to keep the ordering of variables, used by backtrack, unchanged, and to enhance performance once a tree-structured problem is encountered. Since all backtracking algorithms work by progressively instantiating sets of variables,

all one needs to do is to keep track of the connectivity status of the constraint graph. Whenever the set of instantiated variables constitutes a cycle-cutset, the search algorithm is switched to a specialized tree-solving algorithm on the remaining problem, i.e., either finding a consistent instantiation for the remaining variables (thus, finding a solution to the entire problem) or concluding that no consistent instantiation for the remaining variables exists (in which case backtracking must take place).

Observe that the applicability of this idea is entirely independent on the particular type of backtracking algorithm used (e.g., naive backtracking, backjumping, backtracking with learning, etc.). Let B be any algorithm for solving CSPs and let B_c be its enhanced cycle-cutset version. Both algorithms will explore the cutset-part of the search space in the same manner (dictated by the specifics of algorithm B), with algorithm B_c, using a tree-algorithm for exploring the remainder of the search space (see Figure 4). In cases where the problem has a tree-constraint graph, B_c coincides with a tree-algorithm, and when the constraint graph is complete, the algorithm becomes regular B again. In general the cycle-cutset method can be shown to potentially improve any Backtrack scheme. For details see [Dechter 1987.]

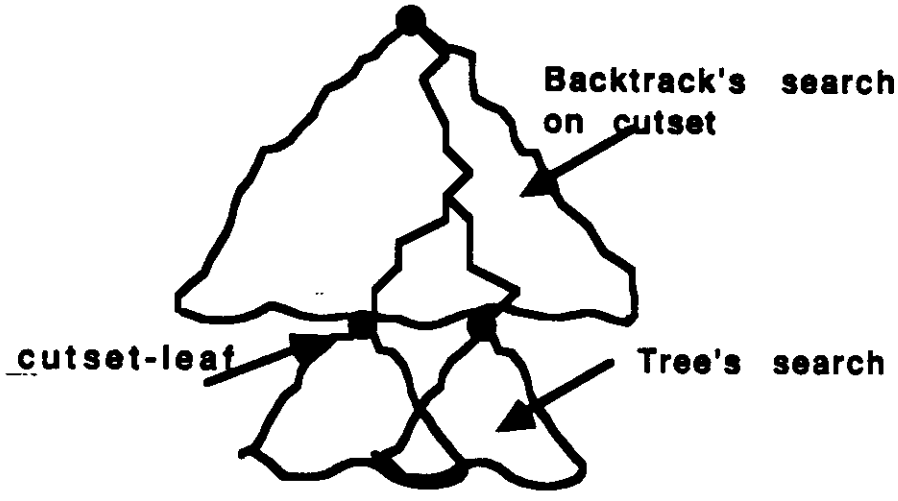

Figure 4: The search space of algorithm B_c.

The tree algorithm used is the one presented in [Dechter 1985], which is optimal for tree-CSPs. The algorithm performs directional arc-consistency (DAC) from leaves to root, i.e., a child always precedes its parent. If, in the course of the DAC algorithm, a variable becomes empty of values, the algorithm concludes immediately that no solution exists. Many orderings will satisfy the partial order above (e.g. child precede its parent) and the choice may have a substantial effect on the average performance. The ordering we implemented is the reverse of "in-order" traversal of trees [Even 1979]. This orderings had the potential of realizing empty-valued variables early in the DAC algorithm and thus concluding that no solution exist as soon as possible. When a solution exists, the tree-algorithm assigns values to the variables in a backtrack-free manner, going from the root to the leaves. The tree-algorithm is presented next.

Tree-backtrack ($d = X_1, \ldots, X_n$)

```
1. begin
2.   call DAC(d)
3.   If completed then find-solution(d)
4.   else ( return,  no solution exist)
5. end
```

DAC- d-arc-consistency
(the order d is assumed)

```
1. begin
2.   For i=n to 1 by -1  do
3.     For each arc (X_j,X_i); j < i do
4.       REVISE(X_j,X_i)
5.       If X_j is empty, return (no solution exit)
5.     end
6.   end
7. end
```

The procedure **find-solution** is a simple backtrack-algorithm on the order d which, in this case, is expected to find a solution with no backtrackings. The algorithm REVISE(X_j,X_i) [Mackworth 1977] deletes values from the domain of X_j until the

directed edge (X_i,X_j) is arc-consistent i.e., each value of X_j is consistent with at least one value of X_i.

6. Summary of previous results

We experimented with Backjump and learning schemes on various classes of problems among which we will focus on the two that were most characteristics. One is the **Zebra problem** which represents a class of difficult problems and can be modeled as a binary CSP by defining 25 variables each with 5 values (the problem statement is given in the appendix). Several instances of this problem were generated by randomly varying the order of variables' instantiation. The other class is a set of randomly generated **Planar CSPs** which yield easy and moderately difficult problems. Planar-problems are CSPs whose constraint-graph is planar and therefore can be regarded as representative for problems in vision. These problems were generated from an initial maximally connected planar constraint-graph with 16 variables (see figure 5).

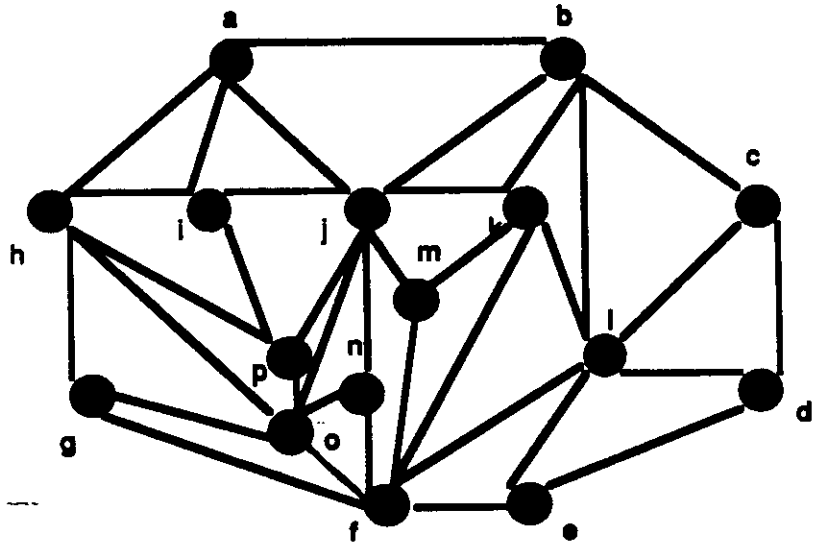

Figure 5: A 16-node, fully triangular planar graph

Two parameters p_1 and p_2, were used in the generation. p_1 determines the probability that an arc will be deleted from this graph, while p_2 determines the probability that pairs

of values belonging to constrained variables are compatible.

Backjump and the learning schemes were immediately integrated. We compared backjump to naive backtrack without any learning and then added to backjump each one of the learning scheme to see their additional impact. Each problem instance was solved by eight search strategies: naive backtrack, backtrack with backjump (no learning), and backjump coupled with each of the six possible modes of learning. The results (i.e. the number of consistency checks performed) for six problem instances of the zebra problem are given in figure 6 (the distinction between adding or modifying constraints are omitted since the difference in impact between these two types of learning was negligible for this problem). Figure 7 depicts the results for the planar-problems after grouping them into clusters (of roughly equal instances) and averaging over each cluster.

Our experiments (implemented in LISP on a Symbolics LISP Machine) show that the behavior of the algorithms is different for different problems. In all problem instances we see an impressive improvement in performance due to backjump alone, and an additional more moderate improvement for the shallow learning schemes. For the zebra problem, the **second-order-deep** learning caused a second leap in performance, with gains over no-learning-backjump by a factor of 5 to 10. For the planar-problems the behavior pattern is different. The improvement gained by Backjump and shallow-learning deteriorates by deeper forms of learnings. For this class, the amount of work invested in these deeper learning schemes outweighs the saving in the search. Note also, that the **add** learning mode is compared unfavorably with the **modify** mode, which can be explained by the fact that adding constraints make the graph denser and cause backjump to be less effective.

Two other classes of problems we experimented with are **class-scheduling problems** and **random CSPs**. The first class represent very easy problems (instances of this

problem were also generated by changing the order of variables) and their solution by naive-backtrack was very efficient. We couldn't therefore see an improvement by the learning schemes but no significant deterioration was observed either. The random CSPs were created by generating random constraint-graphs. For this class the results were of the same nature as for the planar-problems and therefore are omitted here.

The performance of the **cycle-cutset method** was compared to that of the naive backtrack on all the instances of the planar problems as well as on one instance of the zebra problem. In figure 8 the performance of naive backtrack and backtrack with the cutset method (denoted by $Backtrack_c$) are compared. The X-axis is the number of consistency checks (on a log-log paper) performed by *Backtrack* and the Y-axis displays the same information for $Backtrack_c$. Unlike Backjump, the cycle-cutset method do not always improve naive-backtracks performance. This indicates that, for some problem instances the tree algorithm was less efficient then naive backtrack on the tree-part of the search space (although its worst case performance is better). On the average, however, the cutset method improved backtrack by 25% (by 20% on the random CSPs). We also observed that when the size of the cutset is small $Backtrack_c$ outperformed *Backtrack* more often [Dechter 1987.]. On the Zebra-problem the performance of backtrack with and without the cutset method was almost the same (we tested only one instance of the Zebra problem). This can be explained by the fact that the constraint-graph of this problem is very dense and 20 out of the 25 variables were required to cut all cycles. Since most of the search is performed by naive-backtrack anyway, the impact of the tree-algorithm is quite negligible.

Both the cutset method and graph-based backjump exploits the structure of the constraint-graph, however it seems that backjump itself does it more successfully then the cycle-cutset method alone. This is seen in figure 7, depicting the cycle-cutset performance, averaged over the different clusters, alongside the other strategies. The reason

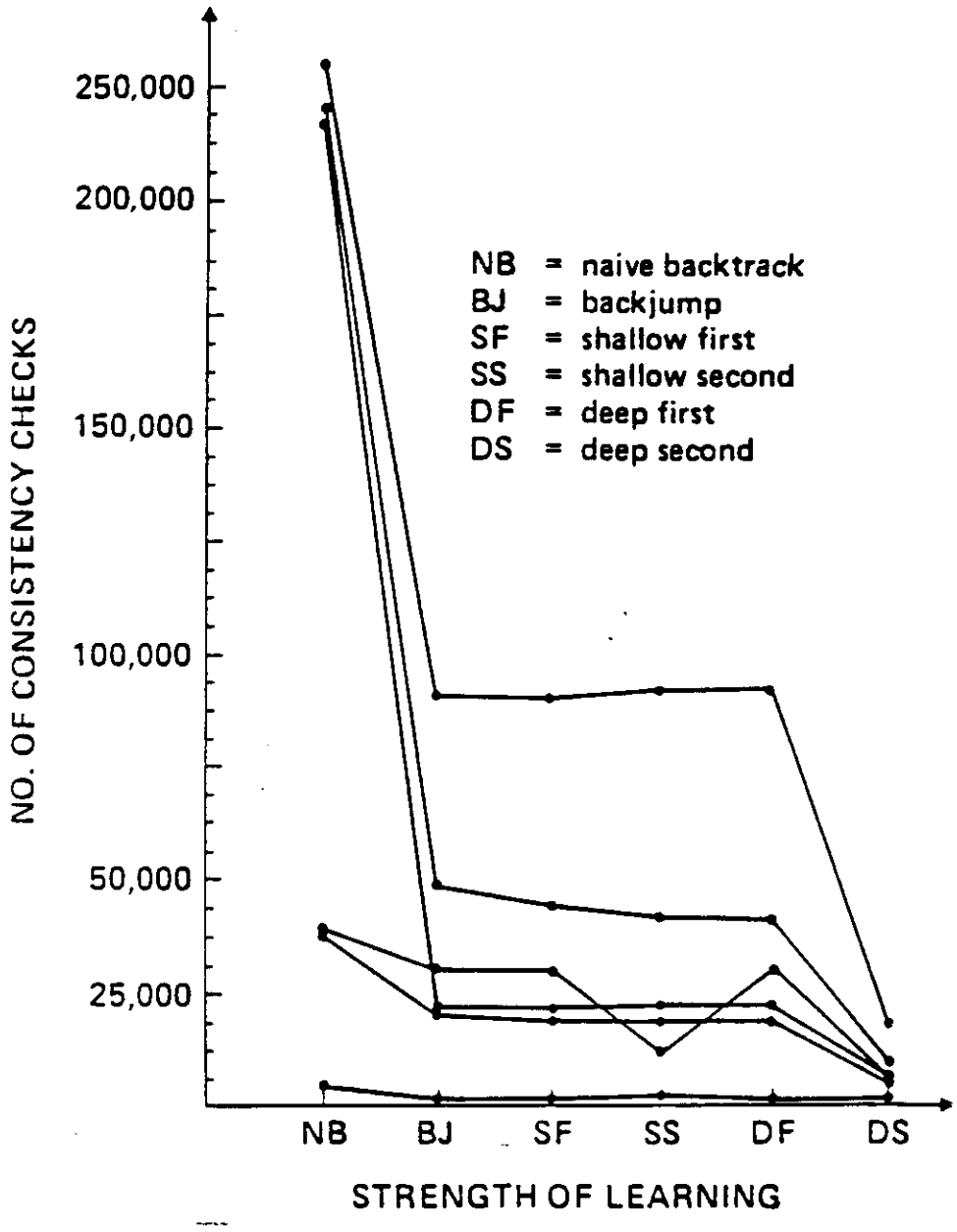
THE ZEBRA PROBLEM
250,000
200,000
150,000
100,000
50,000
25,000
NO. OF CONSISTENCY CHECKS
NB = naive backtrack
BJ = backjump
SF = shallow first
SS = shallow second
DF = deep first
DS = deep second
NB
BJ
SF
SS
DF
DS
STRENGTH OF LEARNING

Figure 6

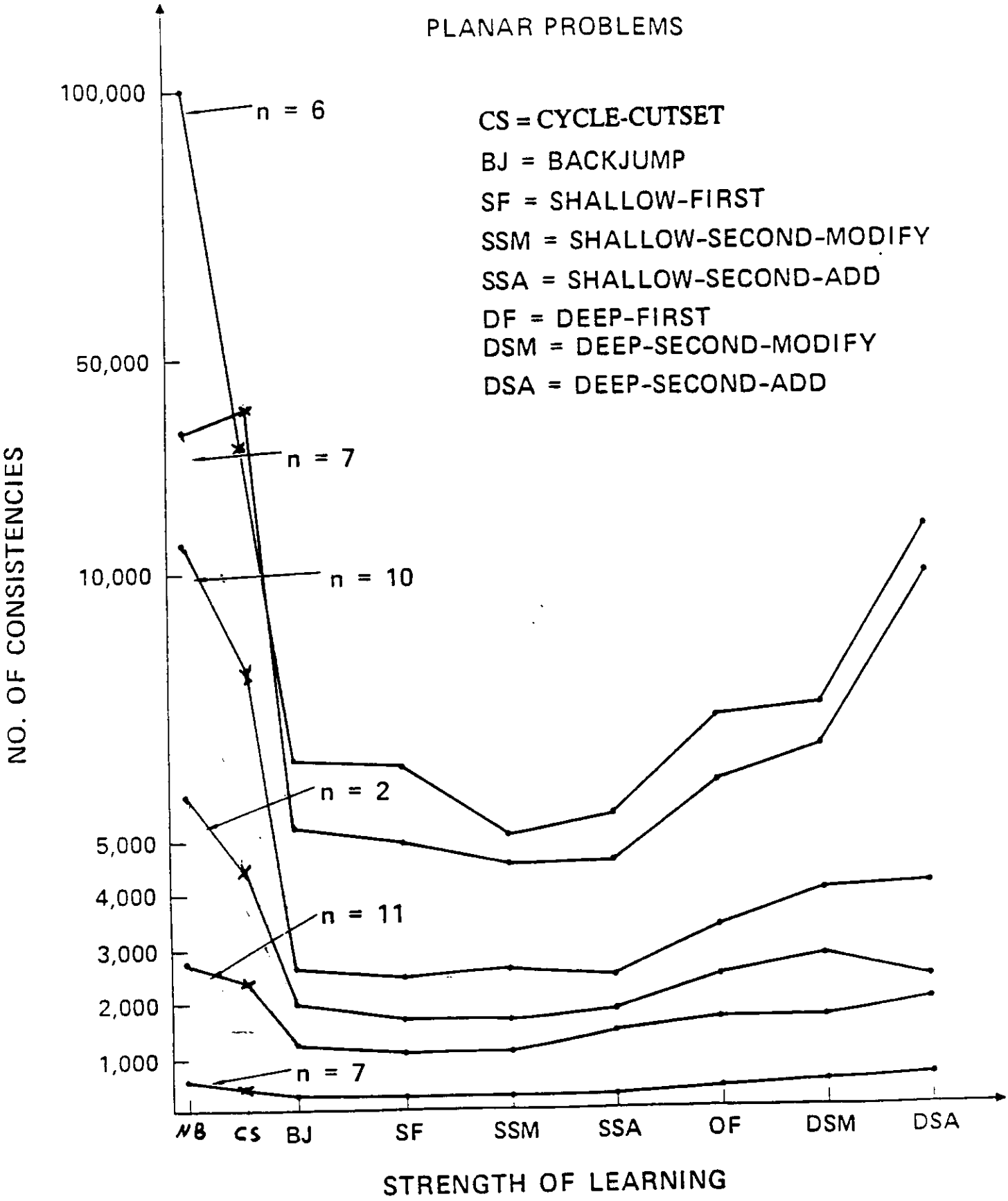
PLANAR PROBLEMS
CS = CYCLE-CUTSET
BJ = BACKJUMP
SF = SHALLOW-FIRST
SSM = SHALLOW-SECOND-MODIFY
SSA = SHALLOW-SECOND-ADD
DF = DEEP-FIRST
DSM = DEEP-SECOND-MODIFY
DSA = DEEP-SECOND-ADD
NO. OF CONSISTENCIES
100,000
50,000
10,000
5,000
4,000
3,000
2,000
1,000
n = 6
n = 7
n = 10
n = 2
n = 11
n = 7
NB
CS
BJ
SF
SSM
SSA
OF
DSM
DSA
STRENGTH OF LEARNING

Figure 7

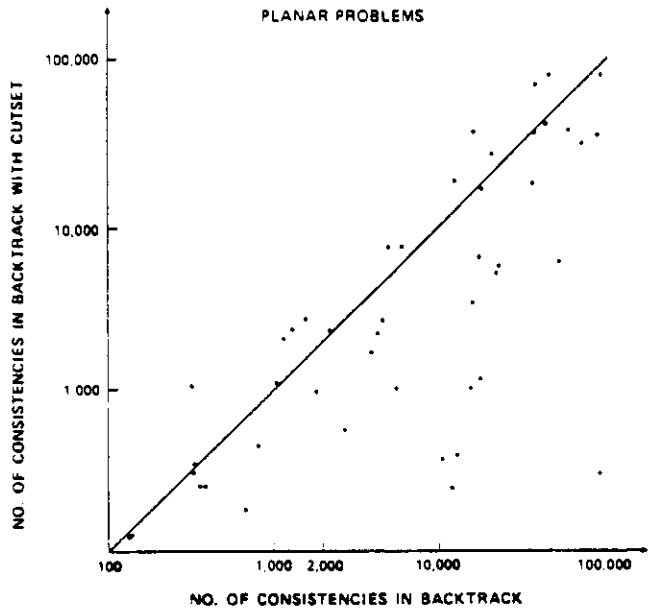

Figure 8

may be that the initial phase of the cutset scheme is performed by naive backtrack and its inefficiency is not compensated enough by the second phase. This motivates the possibility that integrating the cycle-cutset method with a more advance backtrack scheme like Backjump and learning, might improve each individual scheme.

7. The integrated backtrack scheme

In principle the cycle-cutset method can be used with any backtrack scheme not necessarily naive-backtrack. The backtrack algorithm will instantiate variables in a fixed order, until a cutset is realized and then it will switch to a tree-solving algorithm. This suggests that the cutset method may improve any backtrack scheme and thus provide a universal improvement. This conclusion, however, is only valid when there is no flow of information which is used by backtrack when it is between the first part of the search, (denoted the **cutset part**), and that corresponding to the tree-search (denoted the **tree part**) (see figure 4). This assumption is true for naive backtrack but not for all its

enhancements. For instance, when Backjump alone searches the tree-part of the search space, it gathers some valuable information that helps it prune the search in the cutset part by jumping back efficiently. If the integrated scheme backs up naively from the tree part to the cutset part, no such information will be available.

Consider for example the constraint-graph of figure 3 and suppose that backjump works on this problem in the order ($D \rightarrow C \rightarrow E \rightarrow A \rightarrow B$). The ordered graph is given in figure 9 (a). If for instance, there is a dead-end at E, backjump will back-up to node D. If the cutset method is integrated "naively" into backjump, it will instantiate D and C (the cutset variables) and give control to the tree algorithm (see figure 9 (b)).

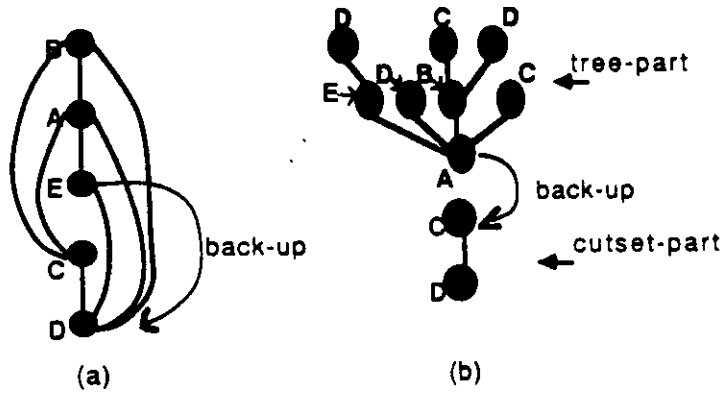

Figure 9

When the dead-end at E is encountered the tree algorithm will indifferently switch back to backjump, providing it no information for skipping C.

This difficulty may be corrected if we equip the tree-solving algorithm with ability to gather the same information needed by backjump, namely identifying the subset of variables which may be responsible for a failure.

The tree algorithm which is integrated with Backjump will return, in a "no-solution" situation, a subset of responsible variables. As soon as the tree-algorithm finds that the domain-values of variable X_i becomes empty (as a result of *REVISE*) it can conclude that only the variables which are located within the processed part of the subtree rooted at X_i may be relevant to the situation. The cutset-leaves of this subtree can be regarded as the Conf-set of this dead-end, they will be returned to backjump which will back-up to the most recent variable among them. If, for instance, the tree-part of the problem in the example of figure 9, is solved in left to right order and if the algorithm finds that the domain values of E are empty after performing *REVISE* on (E,D) it will return D as the Conf-set since D is the only cutset-leaf in the subtree rooted at E, and backjump will back up to it and not to C as in the naive integration. The difference between naive integration and the one suggested here were profound in our experiments and only by this kind of integration the combined scheme was improved.

Learning schemes introduce interaction not only between the tree-part and the cutset-part of the search but also between successive solutions of the tree-part, i.e., successive executions of the tree-part improve due to the learning process. We have not attempted to achieve this capability in the integrated scheme, since the tree algorithm was already fairly efficient and the improvement due to learning was estimated to be meager. Regarding the interaction between the tree part and the cutset part in "no-solution" situations, the same kind of information gathering process, as with backjump alone, can be used. Namely, upon a "no-solution" situation identified at node X_i of the tree, the Conf-set is identified (like for backjump) and returned back for analysis. Shallow learning can be performed on this set. For deep learning an additional analysis of the Conf-set should be performed when X_i is considered the dead-end variable.

The integrated scheme was tested on random problems, random planar problems and the Zebra problem. Figures 9 and 10 compare the performance of Backjump against

the performance of Backjump-with-cutset on the first two classes. For most hard instances (i.e. those requiring more than 1000 consistency checks for backjump) the integrated scheme improved the performance and in some cases quite significantly (the comparison is displayed on a log-log paper as in figure 8). On the average backjump was improved by 25% on both planar and random-problems. For the easiest problems, requiring less then 1000 consistency-checks for Backjump the integration didn't pay off. The deterioration, however, is not severe; 50% for planar-problems and by 10% for random problems.

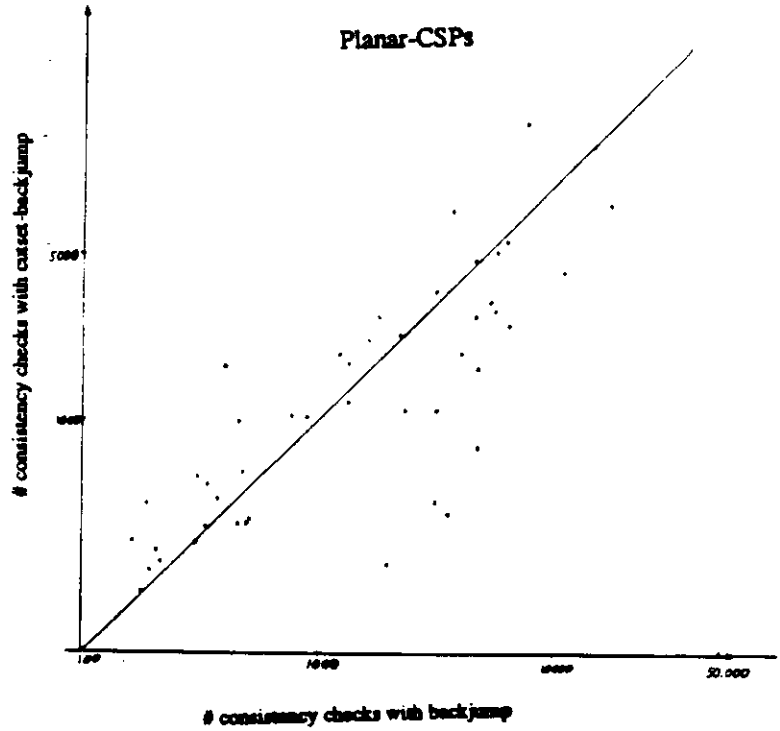

Figure 9: comparing Backjump to cutset-backjump on Planar-problems

Figure 11 compares the integrated learning and backjump schemes with their unintegrated counter parts on planar-problems. On the right hand-side of the Y-axis we repeat the results appearing in figure 7 while on the left hand-side we added the corresponding results of the integrated strategy. The actual numbers (without the deep learning results) are given in table 1. The name of each integrated learning scheme is

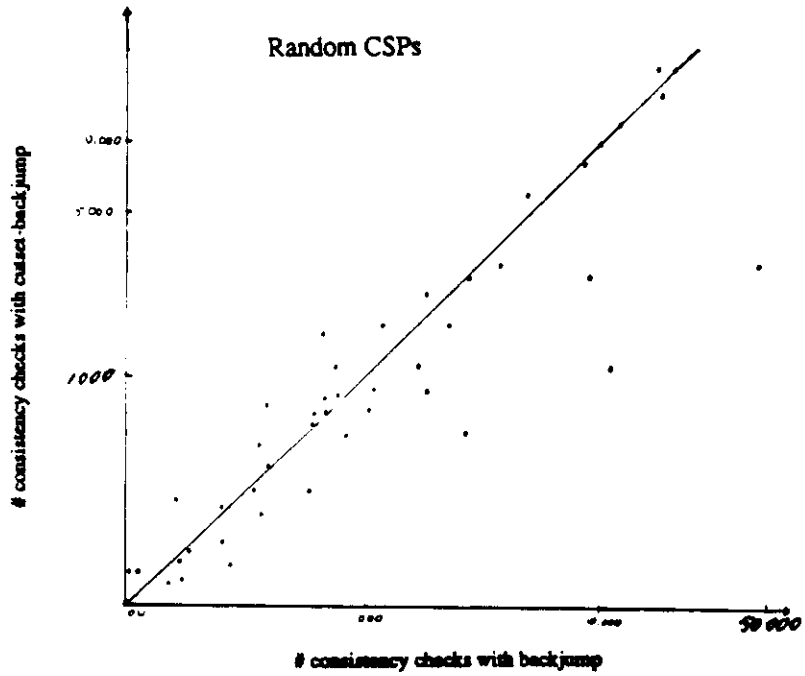

Figure 10: Comparing Backjump to cutset-backjump on random-problems

preceded by "C" to indicate the cutset method which is embedded into it (e.g. CSF stands for cycle-cutset method integrated with shallow-first-order-learning), "ratio" gives the average cutset size to the number of variables. Thus we compare the performances of naive-backtrack, the cutset method, Backjump, Shallow-first-order, shallow-second-order-modify, shallow-second-order-add, deep-first-order, deep-second-order-modify and deep-second-order-add averaged over clusters of instances with and without the cutset method. We see that the curves on the left handside part of the graph are generally below the right hand-side, indicating an improvement in performance. In two clusters, one corresponding to easy problems and one corresponding to 5000-10,000 consistency checks (that contains only two instances) a small deterioration is detected.

We tested the integrated scheme on one instance of the Zebra problem, the one on which Naive backtrack showed the best performance. The results of running all the algorithms on this problem instance are tabulated in table 2. Backjump alone improves per-

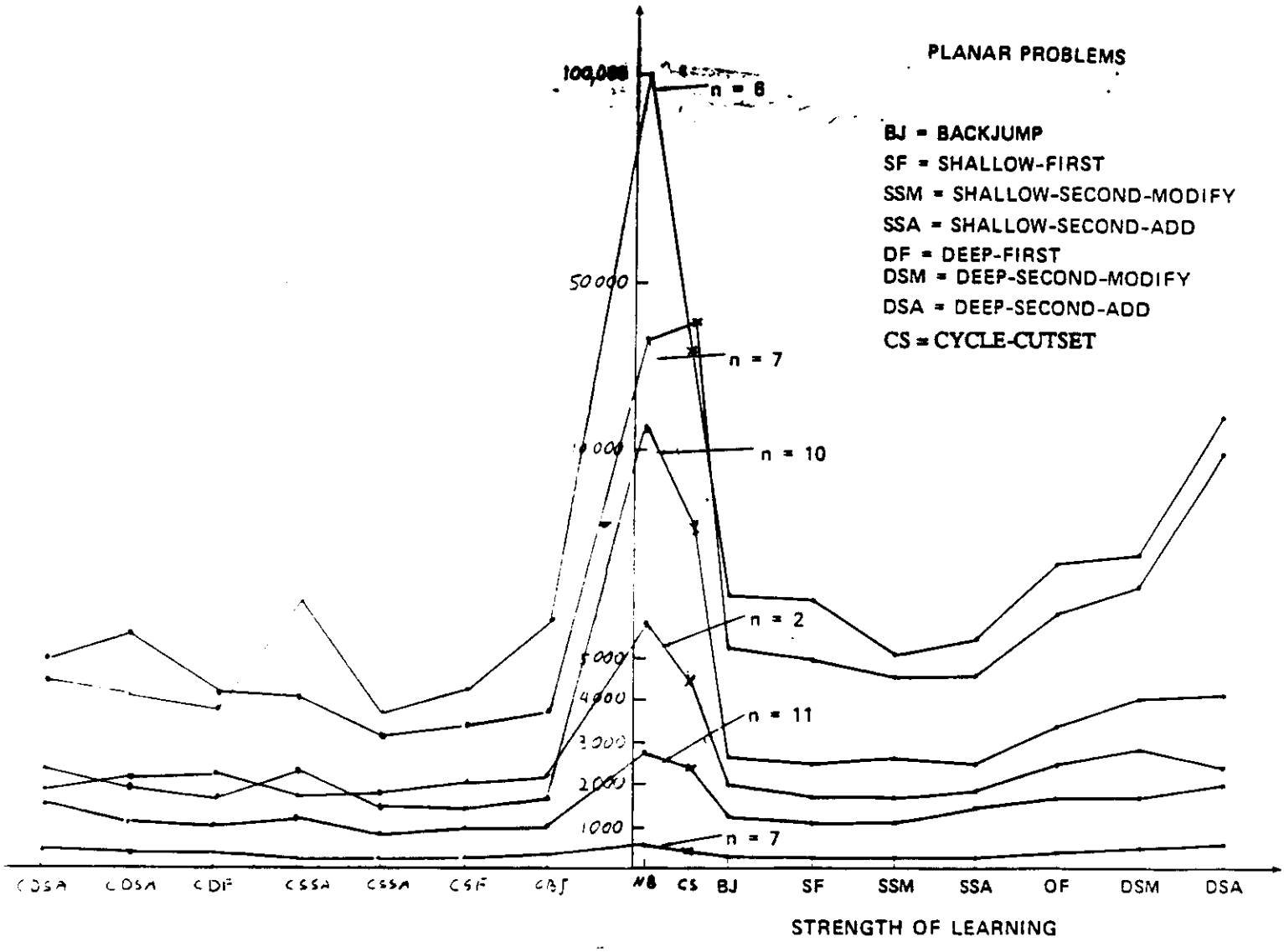
PLANAR PROBLEMS
BJ = BACKJUMP
SF = SHALLOW-FIRST
SSM = SHALLOW-SECOND-MODIFY
SSA = SHALLOW-SECOND-ADD
DF = DEEP-FIRST
DSM = DEEP-SECOND-MODIFY
DSA = DEEP-SECOND-ADD
CS = CYCLE-CUTSET
n = 6
n = 7
n = 10
n = 2
n = 11
n = 7
50000
1000
2000
BJ
SF
SSM
SSA
OF
DSM
DSA
STRENGTH OF LEARNING

Figure 11

range	#-of-instances	ratio	NB	cutset	BJ	SF	SSM		CBJ	CSF	CSSM	CSSA
0-1000	7	0.19	454	404	261	248	252	263	285	270	267	267
1000-5000	11	0.29	2702	2360	1229	1152	1125	1423	1023	996	985	1201
5000-10000	2	0.23	5782	4335	1966	1769	1778	1872	2256	1988	1737	1601
10000-20000	10	0.17	15423	8683	2740	2508	2668	2545	1730	1437	1399	2235
20000-50000	7	0.39	35157	39188	5225	5050	4683	4672	3857	3611	3437	3974
50000-100000	7	0.28	104730	33150	6655	6500	5260	5574	5809	4229	3525	6065

Table 1: average number of consistency checks for different backtracks.

formance by 50% and incorporating it with the cutset method produced an additional improvement of 40% and this is inspite the fact the size of the cutset (20 out of the 25 variables are in the cutset). The shallow-learning scheme did not have a substantial effect on the Zebra problem, only deep learning provided an additional improvement.

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

Table 2: The zebra problem

8. Conclusions

The experiments presented in the paper show that the integrated strategy provides an improvement on each of its individual constituents; Backjump, learning, and the cycle-cutset method. Each of the individual schemes shows its strength in a different

classes of problem instances and the integrated scheme takes advantage of each scheme's power when appropriate. For instance, when the constraint-graph is sparse, backjump and the cutset method are most effective. When it is a highly densed, backjump and the cutset-method lose their effectiveness and learning schemes take over. For intermediate cases both the cutset method backjump cooperate and, together, they do better then each one alone.

For easy problems the integrated scheme showed some deterioration. However, at this range of performances, the variations are small (in absolute terms) and cancel against the improvements in difficult problems.

APPENDIX: The Zebra problem

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

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

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

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

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

The constraints of the puzzle are translated into binary constraints among the variables. For instance, the sentence "The spaniard owns a dog" describes a constraint between the variable **spaniard** and the variable **dog** that allows only the pairs : {(1,1) (2,2) (3,3) (4,4) (5,5)}. In addition, there is a constraint between any pair of variable of the same "cluster" ensuring that they are not assigned the same value. The constraint graph for this problem is given in figure 12 (the constraints among the variables of each cluster are omitted for clarity).

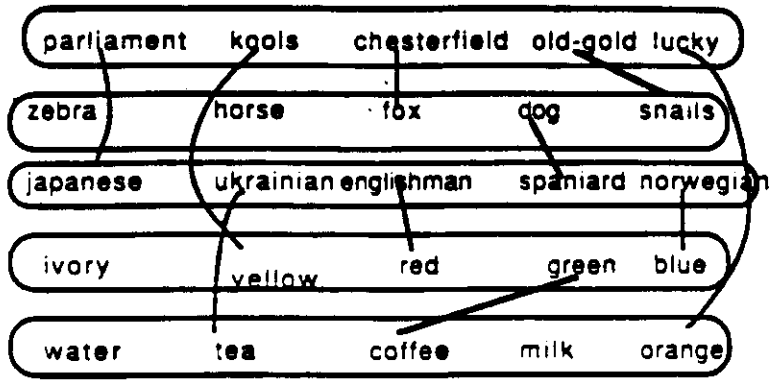

Figure 12: The constraint graph of the zebra problem

References

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

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

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

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

[Doyle 1979] Doyle, Jon, "A truth maintenance system," *Artificial Intelligence*, Vol. 12, 1979, pp. 231-272.

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

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

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

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

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