THE CYCLE-CUTSET METHOD FOR IMPROVING SEARCH PERFORMANCE IN AI APPLICATIONS*

Rina Dechter

Artificial Intelligence Center
Hughes Aircraft Company, Calabasas, CA 91302
and
Cognitive Systems Laboratory, Computer Science Department
University of California, Los Angeles, CA 90024

Judea Pearl

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

ABSTRACT

This paper introduces a new way of improving search performance by exploiting the availability of efficient methods for solving tree-structured problems. The scheme is based on the following observation: If, in the course of a backtrack search, we remove from the constraint-graph the nodes corresponding to instantiated variables and find that the remaining subgraph is a tree, then the rest of the search can be completed in linear time. Thus, rather than continue the search blindly, we invoke a tree-searching algorithm tailored to the topology of the remaining subproblem. The paper presents this method in detail and evaluates its merit both theoretically and experimentally.

1. Introduction

The subject of improving search efficiency has been on the agenda of researchers in the area of Constraint-Satisfaction-Problems (CSPs) for quite some time [16, 13, 10, 11, 12, 5]. A recent increase of interest in this subject, concentrating on the backtrack search, can be attributed to its use as the control strategy in PROLOG [15, 2, 3] and in Truth Maintenance Systems [8, 7, 14].

The various enhancements to Backtrack suggested for both the CSP model and its extensions can usually be classified as being either, **Look-ahead schemes**, affecting the choice of the next variable binding [12, 19, 17, 5], or **Look-back schemes**, affecting the decision of where and how to go in case of a dead-end situation [1, 6].

Another approach for improving the solution of CSPs is to characterize classes of easy CSPs that can be identified directly from their representation and to provide efficient algorithms for their solutions. Recent effort in this direction is focused on the use of graph representations of CSPs and identifying easy problems based on special properties of their graphs [10, 5]. The best known and most useful result in this area is that binary-CSPs whose constraint graph is a tree can be **optimally solved** in $O(nk^2)$ time where n is the number of variables and k is the number of values for each variable.

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

Solving easy problems can be instrumental in the solution of general problems since it provides valuable guidance for efficiently searching for the general solution [18]. This approach was investigated in a look-ahead scheme [4] where heuristic advice is generated to guide the order by which backtrack assigns values to new variables.

In this paper we investigate a new method of utilizing the simplicity of tree-structured problems. The approach is presented for binary CSPs, but it can be generalized to any non-binary CSP using Hyper-graphs instead of graphs. We call it the **cycle-cutset method** since it is based on identifying a set of nodes that, once removed, would render the constraint-graph cycle-free. The method is based on the following observation: If, in the course of a backtrack search, we remove from the constraint-graph the nodes corresponding to instantiated variables and find that the remaining subgraph is a tree, then the rest of the search can be completed in linear time. Thus rather then continue the search blindly, we invoke a tree searching algorithm tailored to the topology of the remaining subproblem. Section 2 presents definitions and preliminaries, section 3 describes the basic ideas involved, and section 4 provides theoretical bounds on the complexity of the method. Section 5 describes a set of experiments for evaluating this method, and section 6 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 [13]). Each node represents a variable whose values are explicitly indicated, and the constraint between connected variables is a strict lexicographic order along the arrows.

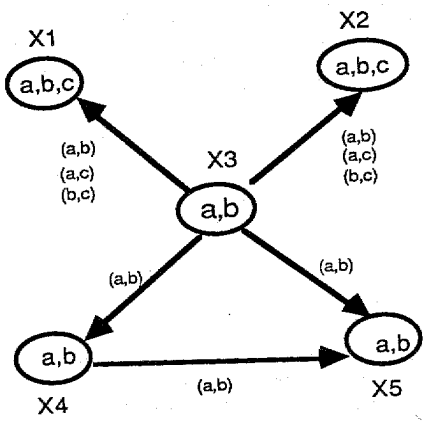

Figure 1: An example CSP

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.

3. The cycle-cutset method

The cycle-cutset method is based on the fact 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 for the rest of the variables is shown in Figure 2a, where the instantiated variable, X_3 is duplicated for each of its neighbors.

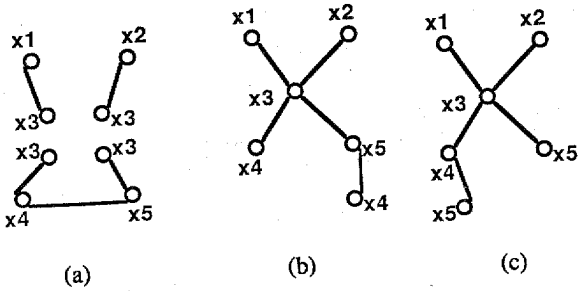

Figure 2: An instantiated variable cuts its own cycles.

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).

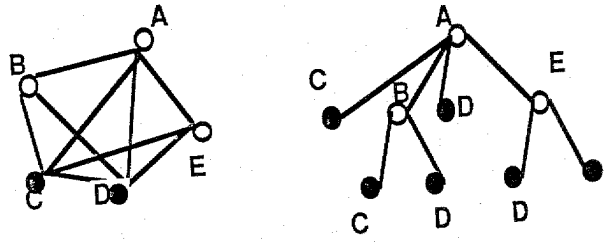

Figure 3: A constraint graph and a constraint-tree generated by the cutset {C,D}

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

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 keep the ordering of variables unchanged, but 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 version. Suppose the variables are instantiated in a fixed order $(d = X_1, \ldots, X_n)$ and that $c = \{X_1, \ldots, X_j\}$ is the first cutset reached. Both algorithms will explore the search space up to depth j in precisely 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).

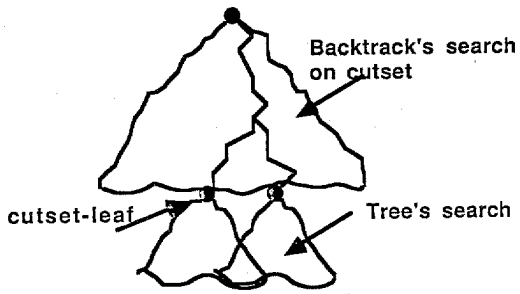

Figure 4: The search space of algorithm B_c.

In the case when the problem has a tree-constraint graph, $Backtrack_c$ coincides with a tree-algorithm, and when the constraint graph is complete, the algorithm becomes regular backtrack again.

4. Bounds on the performance of the cutset method

In [5] it is shown that tree-CSPs can be solved in $O(nk^2)$ and that no algorithm can do better in the worst case. This leads to the following important observation. Backtracking algorithm will improve its worst case bound if it cooperates with a tree algorithm via the cycle-cutset method. Denote the worst case complexity of algorithm A by $M(A)$, when complexity is measured by the number of consistency checks performed.

Theorem 1:

Let B be the naive backtrack algorithm for solving a CSP, and let B_c be the algorithm resulting from incorporating a tree-algorithm in B via the cycle-cutsets approach. Then

$$M(B_c) \leq M(B) \tag{1}$$

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

$$M(B) = M(S_{cutset}) + \sum_{i \in CUTSET-LEAVES} M_i(B) \tag{2}$$

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

$$M(B_c) = M(S_{cutset}) + \sum_{i \in CUTSET-LEAVES} M_i(Tree) \tag{3}$$

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

$$M_i(Tree) \leq M_i(B) \tag{4}$$

yielding

$$M(B) \geq M(B_c)$$

□

Other types of backtrack algorithms may not satisfy the conditions of the above theorem since their behavior at each state of the search may depend not only on that state alone but also on the entire history of the search. We may no longer assume either that the truncated search spaces are the same for an algorithm ,A, and its cycle-cutset version, nor that the tree algorithm solves a tree-structured problem better than A. A may have acquired useful information not available to the tree-algorithm. Equipping the tree-algorithm with equivalent information gathering features, would enable us to extend the theorem over some other versions of backtrack. However this option is beyond the scope of the current paper.

Observe that, when ordering of variables is not fixed, each state should be tested for the cycle-cutset property which would render the cycle-cutset method computationally unattractive. In the analysis that follows we assume, therefore, that the algorithm instantiates variables in a fixed order. Let $d = X_1, \ldots, X_n$, be an ordering of the variables, and let $C = X_1, \ldots X_c$ be a cycle-cutset in the graph. The performance of B_c can be bounded as follows. For a state in the *CUTSET–LEAVES*, only $n-c$ variables remain to be instantiated. Therefore, all tree-structured CSPs induced by these states have $n-c$ variables and can be solved by a tree-algorithm in $O((n-c)k^2)$ (k is the number of values). To switch from the original representation of the problem to a representation required by the tree algorithm may take $O((n-c)ck))$ consistency checks, since each cutset-variable must propagate its value to all its neighbors which are not in the cutset. The complexity of *Tree* algorithm at state i is given therefore by:

$$M_i(Tree) \leq O((n-c)k^2 + (n-c)ck) \tag{5}$$

The number of consistency checks required for generating all CUTSET-LEAVES and the cardinality of this set are bounded by c^k since the CUTSET-LEAVES are the solutions of a CSP restricted to the cutset variables whose cardinality is c. We get:

$$M(B_c) \leq O(k^c) + O(k^c\{(n-c)k^2 + (n-c)ck\}) \tag{6}$$

and therefore,

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

Obviously, as the size of the cycle-cutset diminishes the exponential term in the above expression is reduced and we get better upper-bounds for the performance of the algorithm. In general, given a cycle cutset of size c, the upper bound on the performance of any algorithm is reduced from $O(k^n)$ to $O(k^c)$.

Practically, however, the algorithms rarely exhibit their worst-case performance, and their average case performance is of greater interest. We do not expect to see the superiority of the cycle-cutset method on an instance-by-instance basis. This is so because there is no tree-algorithm

which is superior to all other algorithms for all trees; so, the tree-algorithm used in the cycle-cutset method may occasionally perform worse then the original backtrack algorithm.

5. Experimental evaluation

We compared the performance of the cycle-cutset approach to that of naive backtrack on several CSPs. Backtrack works by provisionally assigning consistent values to a subset of variables and attempting to append to it a new instantiation such that the whole set is consistent. An assignment of values to a subset of the variables is **consistent** if it satisfies all the constraints applicable to this subset.

Variables were instantiated in a fixed order, non-increasing with the variables' degrees. This is a reasonable heuristic since it estimates the notion of **width** of the graph as described by [10]. Whenever $Backtrack_c$ reaches the first cutset in this ordering it switches to a tree-algorithm. If a solution is found, the algorithm stops and returns the solution, otherwise, $backtrack_c$ finds a new consistent cutset-state and proceeds with the tree algorithm until either a solution is found or there is no solution.

The tree algorithm was the one presented in [5], 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 [9]. 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. This ordering compared favourably with other orderings tried. When a solution exists, the tree-algorithm assigns values to the variables in a backtrack-free manner, going from the root to the leaves. For completeness we present the tree-algorithm 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 exist)
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 and therefore its complexity is $O(nk)$. The algorithm REVISE(X_j, X_i) [13] deletes values from the domain of X_j until the directed arc (X_j, X_i) is arc-consistent, i.e., each value of X_j is consistent with at least one value of X_i. The complexity of REVISE is $O(k^2)$.

We compared *Backtrack* to $Backtrack_c$ on two classes of problems, randomly generated CSPs, and Planar problems. Two probabilistic parameters were used in the generation of each class; For the random CSPs, p_1 determines the probability that any two variables are directly connected and p_2, the probability that any two values in an existing constraint are permitted. Two other parameters are n, the number of variables, and k, the number of values for each variable. The Planar problems are CSPs whose constraint-graph is planar. These problems were generated from an initial maximally connected planar constraint-graph with 16 variables. In this case, the parameter p_1 determines the probability that an arc will be deleted from the graph, while p_2 controls the generation of each constraint as in the case of random CSPs.

We tested the algorithms on random-CSPs with 10 and 15 variables, having 5 or 9 values. Tables 1,2, and 3 present the results. Each row in a table describes the performance on one problem instance, i.e., it gives the size of the cutset, the ratio between the cutset size and the number of variables, the number of consistency checks performed by each algorithm, and the ratio between the performance of the two algorithms. We see that in most cases $Backtrack_c$ outperformed *Backtrack*, but not in all cases. This indicates that, for some CSPs, the tree-algorithm was less efficient than regular backtrack. Indeed, while no algorithm for trees can do better then $O(nk^2)$ in the worst case, the performance of such algorithms ranges between $O(nk)$ to $O(nk^2)$ when there is a solution, and it can be as good as $O(k^2)$ when no solution exists. It depends mainly on the order of visiting variables, either for establishing arc-consistency or for instantiation. Regular backtrack may unintentionally step in the right order and, since it avoids the preparation work required for switching to a tree representation (which may cost as much as $O(n^2k)$), it may outperform $Backtrack_c$.

On the average, the cutset method improved backtrack by 20%, for this class of problems. When the size of the cutset is relatively small, $Backtrack_c$ outperforms *Backtrack* more often. Also, the superiority of $Backtrack_c$ is more pronounced when the number of values is smaller (see the comparisons between tables 2 and 3). We conjecture that, since the worst-case performance increases quadratically with the number of values, the tree-algorithm exhibit its worst performance more often, while the performance of regular backtrack remains closer to the average. Notice that, in some instances, the performance of the two algorithms is exactly the same. This happens when the search goes no deeper then cycle-cutset states; so the tree-algorithm is not invoked.

The planar problems were tested with 16 variables and 9 values. The results on this class differ only slightly from the results on random CSPs. An average improvement of 25% is observed for this class of CSPs.

In Figure 5 and Figure 6 we compare the two algorithms graphically on the random CSPs, and in Figure 7 and Figure 8 we do the same for the planar CSPS (due to space considerations the tables for this class are omitted). In Figures

5 and 7 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 $Backtrack_c$. Each point in the graph corresponds to one problem instance. The 45-degree line represents instances for which both algorithms performed equally well; above this line *Backtrack* did better; and below this line $Backtrack_c$ did better. We see that most problem instances lie underneath this line.

In figures 6, and 8, the graph displays the relationship between the performance of $Backtrack_c$ and the size of the cycle-cutset. The X-axis gives the ratio of cutset size to the number of variables, and the Y-axis gives the ratio between the performances of $Backtrack_c$ and *backtrack*. When the ratio of the cutset size to n is less than 0.3, almost all problems lie underneath the line Y=1, for which $Backtrack_c$ outperformed *Backtrack*.

Table 1: CSPs with 10 variables, and 5 values

cutset	ratio	*Backtrack*	$Backtrack_c$	$\frac{Backtrack_c}{Backtrack}$
4	0.4	1701	1572	0.9
5	0.5	1100	997	0.91
3	0.3	705	842	1.19
4	0.4	1392	458	0.33
3	0.3	150	142	0.95
4	0.4	283	268	0.95
3	0.3	247	185	0.75
5	0.5	367	504	1.37
3	0.3	184	87	0.47
4	0.4	384	284	0.75
4	0.4	1188	903	0.76
3	0.3	483	317	0.66
3	0.3	177	130	0.73
mean		676	550	0.84

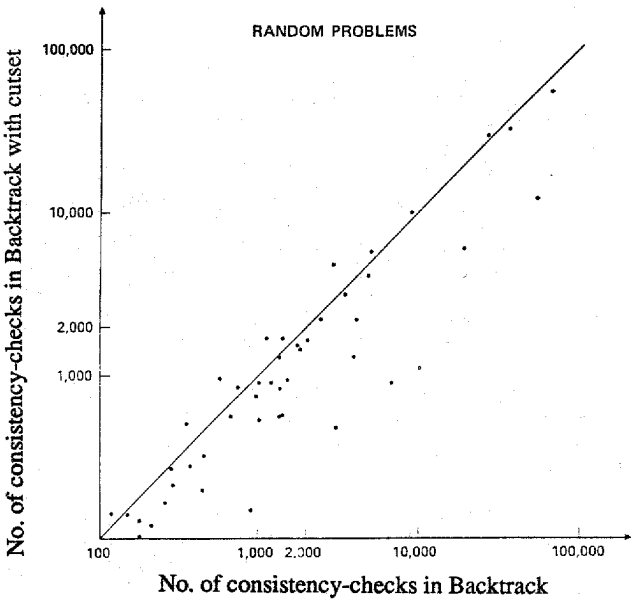

Figure 5

Table 2: CSPs with 15 variables and 5 values

cutset	ratio	*Backtrack*	$Backtrack_c$	$\frac{Backtrack_c}{Backtrack}$
9	0.6	623	623	1
10	0.66	840	840	1
3	0.2	6779	914	0.13
3	0.2	214	125	0.58
2	0.13	117	143	1.22
3	0.2	589	950	1.61
3	0.2	3849	1288	0.33
4	0.27	971	750	0.77
2	0.13	903	153	0.17
3	0.2	1322	1301	0.98
3	0.2	450	199	0.44
3	0.2	264	165	0.63
4	0.27	3443	3190	0.93
3	0.2	292	210	0.72
2	0.13	1368	533	0.39
4	0.27	2042	1631	0.8
4	0.27	1531	870	0.57
5	0.33	1363	839	0.64
5	0.33	1137	1699	1.49
5	0.33	669	567	0.85
mean		1438	849	0.59

Table 3: CSPs with 15 variables and 9 values

cutset	ratio	*Backtrack*	$Backtrack_c$	$\frac{Backtrack_c}{Backtrack}$
11	0.7	2753	2753	1
5	0.33	4832	4167	0.86
4	0.27	56774	12094	0.21
5	0.33	67686	55350	0.82
4	0.27	2414	2265	0.94
5	0.33	71063	102627	1.44
7	0.47	5069	5975	1.18
7	0.47	38316	32792	0.86
5	0.33	1805	1460	0.81
5	0.33	27934	28996	1.04
5	0.33	9280	10045	1.08
2	0.13	4014	2263	0.56
4	0.27	19534	6316	0.32
5	0.33	2989	4902	1.64
6	0.4	1397	1688	1.21
3	0.2	10505	1140	0.11
mean		17177	20397	0.81

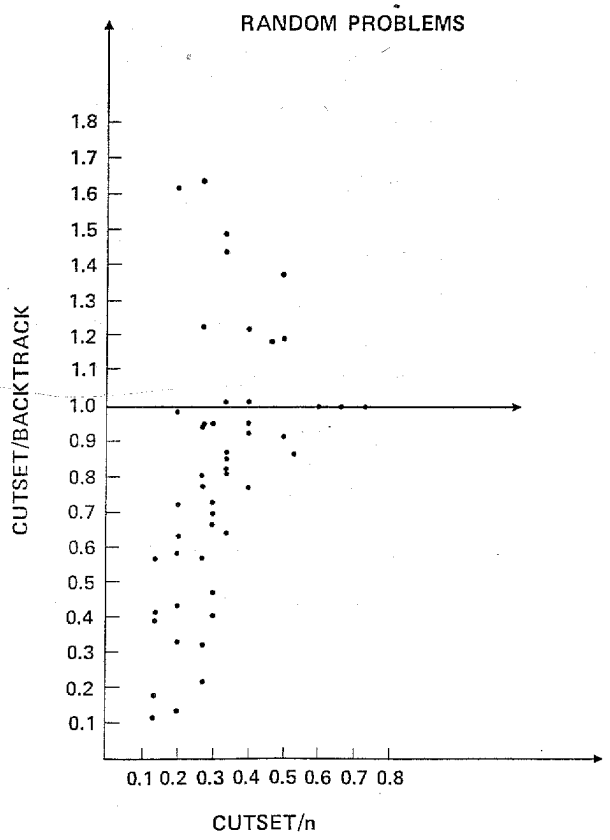

Figure 6

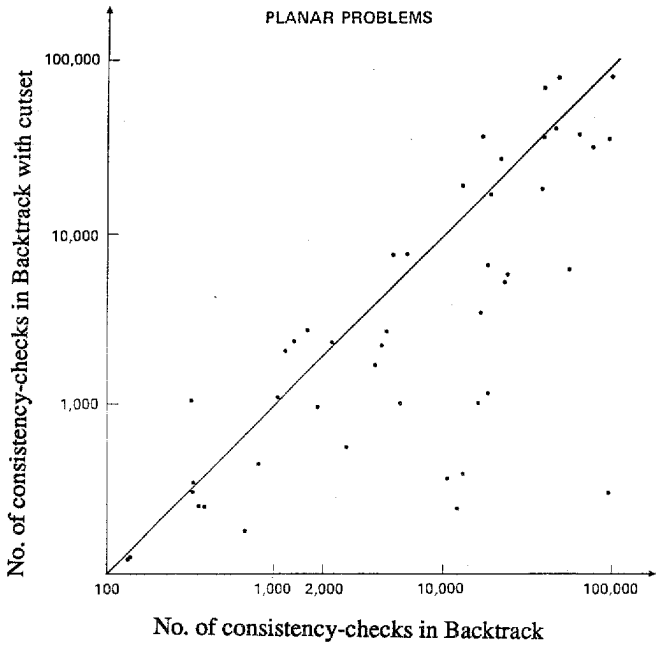

Figure 7

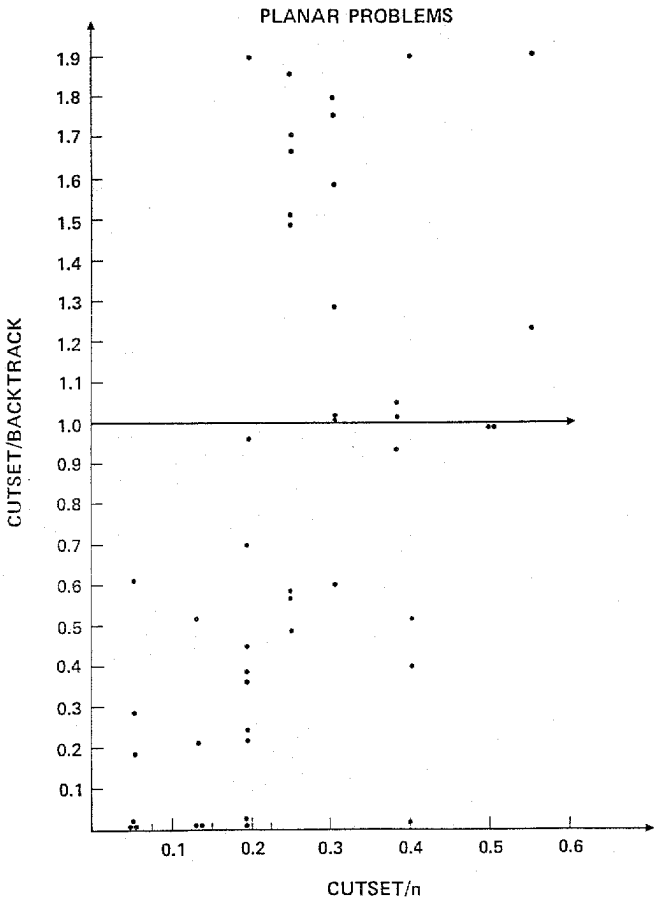

Figure 8

6. Conclusions

The cycle-cutset method provides a promising approach for improving a wide range of search algorithms. The experiments presented demonstrate that the effectiveness of this method depends on the size of the cutset. This provides an a-priori criterion for deciding whether or not the method should be utilized in any specific instance.

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

References

[1] Bruynooghe, Maurice, "Solving combinatorial search problems by intelligent backtracking," *Information Processing Letters*, Vol. 12, No. 1, 1981.

[2] 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.

[3] Cox, P.T., "Finding backtrack points for intelligent backtracking," in *Implementation of Prolog,* J.A. Campbell, Ed. Ellis Harwood, 1984, pp. 216-233.

[4] Dechter, R. and J.Pearl, "A problem simplification approach that generates heuristics for constraint satisfaction problems.," UCLA-Eng-rep.8497. To appear in Machine Intelligence 11., 1985.

[5] 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.

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

[7] De-Kleer, Johan, "Choices without backtracking," in *Proceedings AAAI,* Washington D.C.: 1983, pp. 79-85.

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

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

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

[11] 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.

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

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

[14] Martins, Joao P. and Stuart C. Shapiro, "Theoretical Foundations for belief revision," in *Proceedings Theoretical aspects of Reasoning about knowledge,* 1986.

[15] Matwin, Stanislaw and Tomasz Pietrzykowski, "Intelligent backtracking in plan-based deduction," *IEEE Transaction on Pattern Analysis and Machine Intelligence,* Vol. PAMI-7, No. 6, 1985, pp. 682-692.

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

[17] Nudel, B., "Consistent-Labeling problems and their algorithms: Expected complexities and theory based heuristics.," *Artificial Intelligence,* Vol. 21, 1983, pp. 135-178.

[18] Pearl, J., "On the discovery and generation of certain heuristics," *AI Magazine,* No. 22-23, 1983.

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