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

Rina Dechter
Judea Pearl

October 1986
CSD-860022

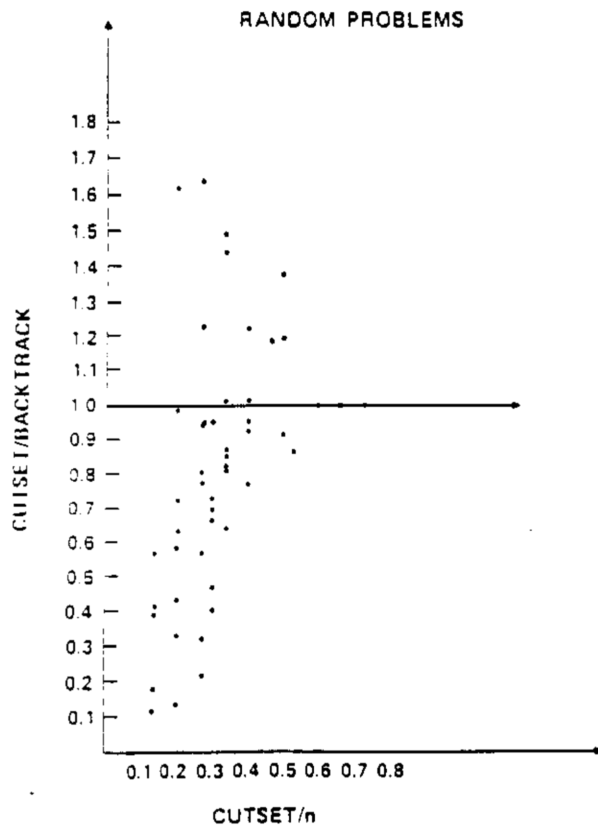

Figure 6

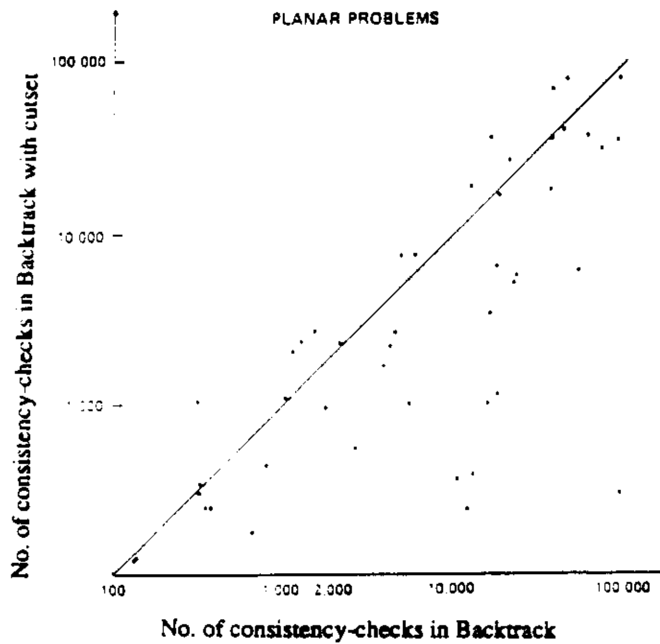

Figure 7

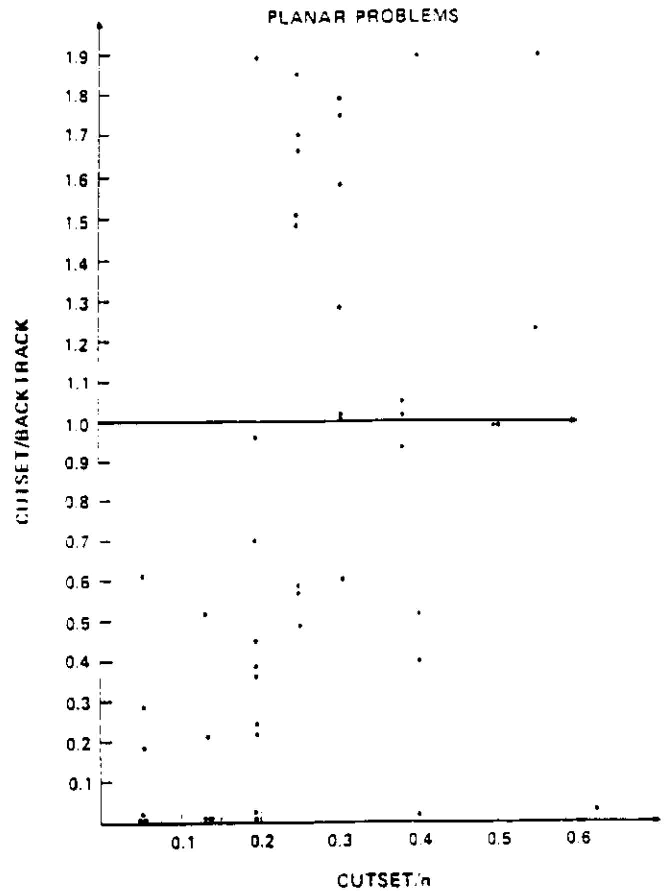

Figure 8

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.

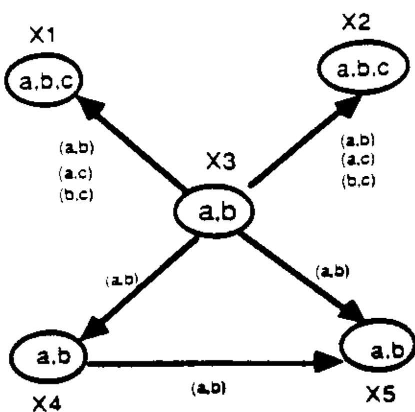

Figure 1: An example CSP

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.

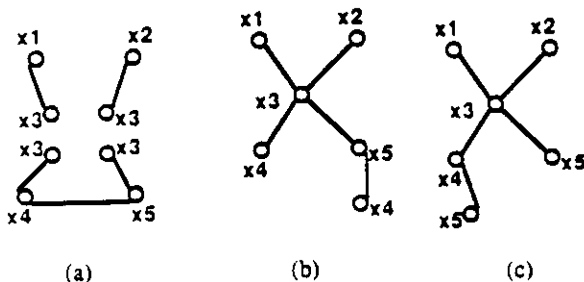

Figure 2: An instantiated variable cuts its own cycles.

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).

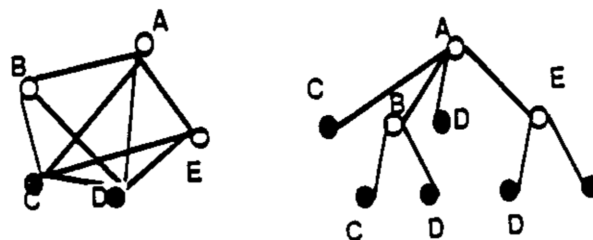

Figure 3: A constraint graph and a constraint-tree generated by the cutset {C,D}

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

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 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 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