An Efficient Cross Product Representation of the Constraint Satisfaction Problem Search Space[1]

Paul D. Hubbe and Eugene C. Freuder

Computer Science Department
University of New Hampshire
Durham, NH 03824 USA
pdh@cs.unh.edu; ecf@cs.unh.edu

Abstract

Constraint satisfaction problems involve finding values for variables subject to constraints on which combinations of values are permitted. They arise in a wide variety of domains, ranging from scene analysis to temporal reasoning. We present a new representation for partial solutions as cross products of sets of values. This representation can be used to improve the performance of standard algorithms, especially when seeking all solutions or discovering that none exist.

Introduction

Constraint-based reasoning has long been recognized as a primary component of AI problem solving and has seen increasing interest and application in recent years. Constraint-based reasoning has been used in many areas of artificial intelligence: vision, language, planning, diagnosis, scheduling, configuration, design, temporal reasoning, defeasible reasoning, truth maintenance, qualitative physics, logic programming, expert systems.

This research focuses on the constraint satisfaction problem (CSP) paradigm, which underlies many of these applications [Mackworth 87]. Constraint satisfaction problems involve finding values for a set of problem variables that simultaneously satisfy a set of constraints or restrictions on which combinations of values are permissible. A variety of approaches have been developed for solving these problems; the basic algorithmic tools are backtracking and constraint propagation [Meseguer 89].

A basic problem in constraint satisfaction problem search is a phenomenon that we might call "the battle of the bulge". Search commonly proceeds by developing partial solutions, discarding them once it is clear that they cannot be extended to complete solutions. Initially we start with a small number of partial solutions. Ultimately, for some problems, there are relatively few complete solutions. However, in between, there may be a huge number of partial solutions, candidates for completion to full solutions.

Of course, this phenomena is common to other AI search domains. For example, beam search is an effort to address the problem in the classic state space domain. In fact the technique proposed here for addressing the problem was inspired by a very similar technique demonstrated by Wu [Wu 90] in what is, at least superficially, a very different search domain: diagnosis in the presence of multiple diseases.

The general principle involved might be stated as follows:

Find a method of representing search subspaces that permits operating on the subspaces more efficiently than operating on their individual elements.

In the CSP context:

Find a method of representing sets of partial solutions such that further values can be tested against a set more efficiently than against each member individually.

We implement this principle by using a cross product representation of sets of partial solutions. Suppose, for example, that a, b, c, k are the potential values for variable X and c, d, e, h are the potential values for variable Y. If the pairs of values that satisfy the constraint between X and Y are:

{(a d) (a e) (a f) (b d) (b e) (b f)
(c d) (c e) (c f) (k e) (k h)}

this set can be represented by two cross products:

{a b c} X {d e f} and {k} X {e h}

A value for another variable can be tested against one of these cross product sets by testing against each value for each variable in the cross product. If the value g, for variable Z, is consistent with every value but b and h, we arrive at the extended partial solution sets:

{a c} X {d e f} X {g} and {k} X {e} X {g}.

[1]This material is based upon work supported by the National Science Foundation under Grant No. IRI-8913040. The Government has certain rights in this material.

If the value m for Z is also consistent with every value but b and h we have the new partial solution sets:

{a c} X {d e f} X {g m} and {k} X {e} X {g m}.

The cross product representation permits additive as opposed to multiplicative complexity. For example, value g for variable Z can be tested against the cross product representation {a b c} X {d e f} with six (3 + 3) consistency tests : 3 to see if g is consistent with a, b and c and 3 more to see if g is consistent with d, e and f. With the standard backtrack search tree representation we need to see if g is consistent with each of the 3 * 3 = 9 pairs, and for each pair 1 or 2 tests must be made, depending on whether the first test succeeds or not, resulting in a total of between 9 and 18 tests.

A related technique was utilized in [Freuder and Quinn 85] to take advantage of variables that are not directly constrained. The lack of a direct constraint between variables implies that all combinations (the cross product of the variable domains) are consistent. Here all combinations in a cross product of subsets of the domains are consistent.

This representation supports search algorithms that can be time as well as (relatively) space efficient, especially in searching for all solutions. We call this representation the *cross product representation* or CPR.

It might seem that there would be a problem with a potential combinatorial space explosion in utilizing this representation. However, we will see that the cross product representation can be profitably utilized within an essentially depth-first search context where space is not a serious concern.

CPR can be applied to both classical backtracking and *forward checking,* one of the most highly regarded variants of backtrack search in the literature [Golumb and Baumert 65; Haralick and Elliott 80; Nadel 89]. In both cases, when finding all solutions, adding CPR can *not increase* the number of constraint checks, a standard measure of CSP algorithm performance. This is true, in particular, in the special case where there are no solutions, and finding all solutions means discovering that the problem is insoluble. (A *constraint check* is counted every time we ask the basic question "is value a for variable X consistent with value b for variable Y", i.e. is the pair (a b) allowed by the constraint between X and Y.)

In practice CPR can greatly *reduce* the number of constraint checks required by backtracking or forward checking, in some cases even when searching for a single solution. In fact we have hopes that CPR will be helpful in searching for any solutions to "really hard problems" [Cheeseman, Kanefsky and Taylor 91], when many combinations of values almost succeed, but few fully succeed.

The second section discusses the experimental design we use to gather concrete constraint check and cpu time data. The third section presents a basic algorithm for adding CPR to backtrack search, and suggests directions for refinement. We establish the theoretical and practical advantages of the new algorithm. The fourth section discusses an algorithm for adding CPR to forward checking. The final section is a brief conclusion.

Experimental Design

We show in the next section that it is theoretically impossible for the addition of CPR to require more constraint checks from backtracking or forward checking, when seeking all solutions. (Between backtracking and forward checking neither is always superior in theory, though forward checking often is superior in practice.) However, several interesting questions remain, which experimental evidence can address:

* How many constraint checks can be saved?
* Can the savings be correlated with problem structure?
* Does overhead cost seriously affect the savings?
* Can we save when only seeking some solutions?

We performed tests with classical backtracking (BT) and with backtracking augmented by CPR (BT-CPR) to address these questions. (The CPR augmentation of forward checking has not yet been implemented; however, BT-CPR has been compared, favorably in some cases, with forward checking.)

Our test problems are binary CSPs: constraints involve two variables at a time. Random ten variable problems were generated with different specified "expected" values for three parameters: constraint density, constraint tightness and domain size. *Domain size* is a positive integer indicating the number of values per variable. *Constraint tightness* is a number between 0 and 1 indicating what fraction of the combinatorially possible pairs of values from two variable domains are not allowed by the constraint. (If the constraint between X, with values a and b, and Y, with values c and d, does not permit the pairs (a c) and (b c) and (b d), then the constraint tightness is .75.)

Constraint density is just a bit more involved. Binary CSPs can be represented as constraint graphs, with the vertices corresponding to variables and the edges to constraints. We want to deal with problems with connected constraint graphs, as unconnected components can be solved independently. A connected constraint graph with n vertices has, at a minimum, n-1 edges, and, at a maximum, $(n^2-n)/2$ edges. *Constraint density* is a number between 0 and 1 indicating what fraction of the possible constraints, beyond the minimum n-1, that the problem possesses. For example, a CSP with a complete constraint graph, where every vertex is connected to every other, has a constraint density of 1; a tree-structured constraint graph has a constraint density of 0.

The random problem generator allows us to specify "expected" values for density and, somewhat indirectly, tightness and domain size. For example, if we specify tightness is .5, every pair of values will have a 50/50 chance of being allowed.

Our main set of test problems consisted of five problems for each of fifteen combinations of tightness and density parameter values. The density parameter values tested were .1, .5 and .9; the tightness parameter values tested were .1, .3, .5, .7 and .9. The expected domain size was 5.

The method of generating problems has pros and cons that we do not have space to discuss fully here, but it should be noted that it permits some variation in actual results. For example, one variable domain size might actually be 3, another 7, rather than 5.

We measure performance in terms of constraint checks and cpu time. We also compute the ratio of the two, the number of checks per second. Cpu time measurements need to be viewed with even more than the usual skepticism as tests were not always conducted with the same machine, and an improvement was made to the algorithms during testing. None of this however affects the constraint check count, and the overall conclusions to be drawn from the timing results clearly coincide with the overall conclusions to be drawn from the constraint check results.

This set of problems includes many with no solutions and a few with millions of solutions. The ratio of the number of solutions to the number of possible value combinations ranges from 0 to about one in three. Constraint checks required by backtracking range from 8 to almost 78,000,000; time required by backtracking ranges from an unmeasurable .00 seconds (which we report as .01 to avoid problems in dividing by 0 in computing ratios) to almost 14,000 seconds, nearly 4 hours. For each set of five problems with specified parameter values we calculated, for each algorithm, the average constraint checks required, the average cpu time utilized and the average checks/second ratio. Effort can vary considerably among a set of five problems for a fixed set of tightness, density and domain size expected-value parameters. Therefore we do not wish to impute too much to individual averages over these sets. However, there are clearly strong, broad patterns in the averages viewed together over the range of parameter values.

We also compute ratios of BT and BT-CPR performance using constraint checks, cpu time and checks/second data. This permits a quick assessment of the relative performance of the two algorithms being compared. The larger the numbers the better the relative performance of BT-CPR. (Thus BT performance is in the numerator when comparing constraint checks and cpu time, and in the denominator when comparing checks/second.) Any number greater than 1 indicates superior performance by BT-CPR.

We were concerned that it could be misleading to compute, for example, constraint check ratios, for the two algorithms, by simply taking the average number of constraint checks used by one algorithm for a set of five problems and dividing it by the average number of constraint checks used by the other algorithm for the same five problems. One problem in the set of five might be so much larger than the others that it would unfairly dominate the result. Therefore we compute the ratio of constraint checks for each of the five problems separately and then average those ratios. The cpu time ratios, and the ratios of the checks/second ratios, were similarly computed.

Applying CPR to Backtracking

The cross product principle can be used to augment classical backtrack search. An algorithm, BT-CPR, is given in Figure 1. BT-CPR involves a "generate and merge" process. Given a cross product P, and a value, v, for the next variable to consider, V, we compute the subset, P', of the cross product, consistent with v, forming a new cross product P' X v. We do this for each value v of V. These new cross products are children of P in the search tree. Any children of P which are identical except for the value of V can be merged into a single child. For example, if a and b are both consistent with the same cross product set P', we can form the new cross product P' X {a b}.

```
Procedure BT-CPR
Push the set of values for the first variable
  onto CP-stack
While CP-stack is not empty
  Pop P from the CP-stack
  N-list <- empty list
  For each value v in the first variable V
        that is not represented in P
     P' <- P X {v}
     P' <- P' after removing values
        inconsistent with v
     If P' ≠ empty cross product
        then N-list <- N-list append P'
     Merge all P' in N-list differing only
        in the value of V
     If V is the last variable to consider
        then
             output N-list as solutions
        else
             push all elements in N-list
                 onto CP-stack
```

Figure 1. BT-CPR Algorithm.

The search process is basically depth-first; however, it generates all the children at a node at once, in order to allow merging among the children. There can be at most d children, where d is the maximum number of values for any variable, and the depth of search is limited by the number of variables, n. Thus there is an O(nd) upper bound on the number of cross products that the search needs to store at any point. There is clearly an O(nd) bound also on the number of values stored in any one cross product.

CPR could also be added to a breadth-first search for all solutions. This would offer greater opportunity for merging solution sets. In a breadth-first implementation this merging could take place among all "cousins" across an entire level of the tree, rather than just among siblings. However, there would be considerable additional overhead.

This algorithm uses a very simple merging procedure. More elaborate merging heuristics, or more generally, a more elaborate search for an optimal representation as a set of cross products, are worth pursuing. Here again there will be cost/benefit considerations.

In theory BT-CPR cannot do worse than classical backtracking in searching for all solutions to a CSP, and in practice it can do orders of magnitude better.

Theorem 1. Augmenting classical backtrack search with CPR will never increase the number of constraint checks required to search for all solutions to a CSP, or to determine that an unsolvable problem has no solution.

Proof: Consider that the worst case for backtrack augmented by CPR occurs when all the cross product sets are singletons, sets of one element. But in this case BT-CPR essentially reduces to classical backtracking. •

Note that this argument is independent of any ordering heuristics that may be applied to enhance backtrack search performance. Given a backtrack algorithm with good search ordering heuristics, adding CPR cannot increase the number of constraint checks required to find all solutions.

Ordering is something of a catch-22 proposition for CPR. On the one hand, the "fail first" principle [Haralick and Elliott 80] suggests that we narrow the top of the search tree. On the other hand, CPR thrives on handling multiple possibilities in a concise manner. (Some initial testing suggest that employing the opposite of a good standard CSP search heuristic might sometimes be helpful in the CPR context.)

Figure 2 shows the performance ratios, as explained in the previous section. BT-CPR never required more constraint checks than BT, of course; for all but the simplest cases it required less, up to three orders of magnitude less for the most weakly constrained set of problems. The constraints per second performance ratio is uniformly close to 1, suggesting that BT-CPR pays little overhead penalty. Indeed, BT-CPR has the advantage on three sets of problems.

Cpu time is, in fact, less for BT-CPR on all but some of the simpler problems. Again the performance ratio climbs up to three orders of magnitude on the most weakly constrained set of problems. On one problem with the tightness parameter set to .1 and the density parameter set to .1, BT-CPR required 25,854 constraint checks and 3.21 seconds while backtracking required 77,867,372 constraint checks and 13,986.59 seconds (almost four hours).

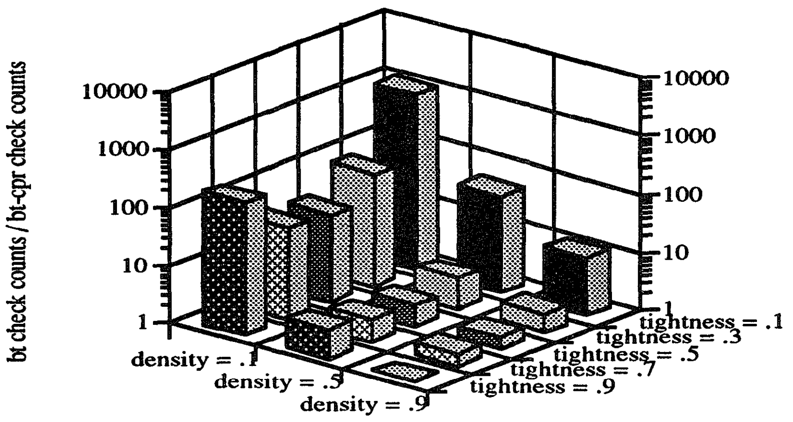
Clearly, the advantage of CPR increases as the problems become less tightly constrained. Weakly constrained problems are actually difficult problems for finding all solutions for the simple reason that there are a lot of solutions to find, and there is relatively little pruning of the search space. These problems also have a great many solutions, but some applications may need to sift through a great many solutions, if only to collect summary statistics, or in search of a candidate that will pass a further testing process. A simple theoretical analysis shows that in the degenerate case where virtually all possibilities are solutions backtracking is $O(n^2d^n)$ while backtracking with the cross product representation is $O(nd^2)$.

Roughly speaking BT-CPR does at least an order of magnitude better on problems with either low density or low tightness parameters. When both parameters are low the performance ratio climbs to three orders of magnitude.

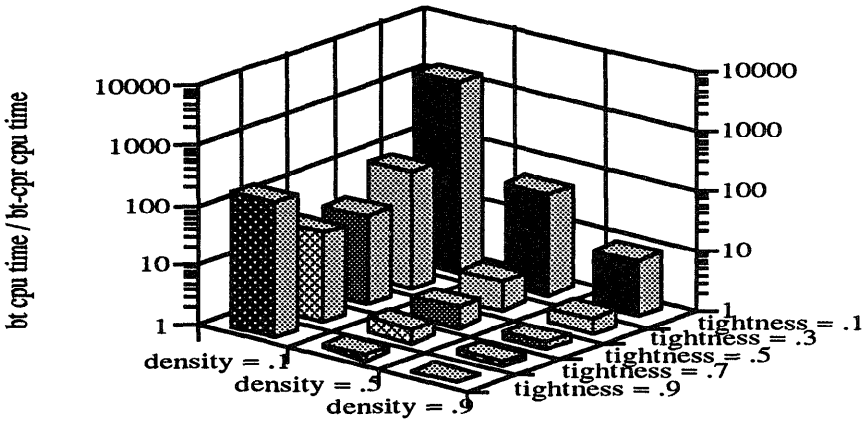
It is important to note that while our experiments have provided what might be called "heuristic sufficient conditions" for CPR to do especially well, these are not presented as necessary conditions. We expect CPR to be especially useful under other conditions, as well.

We expect CPR to help cope with what one might call "frustrating" problems, where many sets of value choices *almost* work, by merging many of the partial solutions into cross products. Note, for example, that we can take a test problem where CPR does extremely well because there are a lot of solutions, and transform it into another problem that allows no solutions, where CPR continues to do extremely well. Simply add an additional variable, all of whose values fail to be consistent with any of the solutions for the original variables.

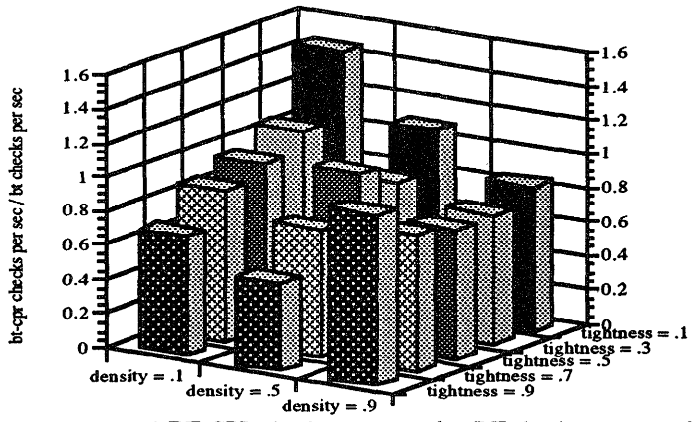
CPR can have an advantage even when we are only seeking *some* solutions. We may be looking for a fixed number or we may be taking solutions as needed, suspending search in a "lazy evaluation" mode. Figure 3 shows the constraint check effort required for BT and BT-CPR to supply x solutions, with x =1, 5, 10, 15, etc., for two sample problems with density parameter .5 and tightness parameter .3. (All the problems with density parameter .5 and tightness parameter greater than .3 failed to have any solutions.) Note that BT-CPR may return solutions in batches; for example, it may first find a cross



a) BT constraint checks to BT-CPR constraint checks



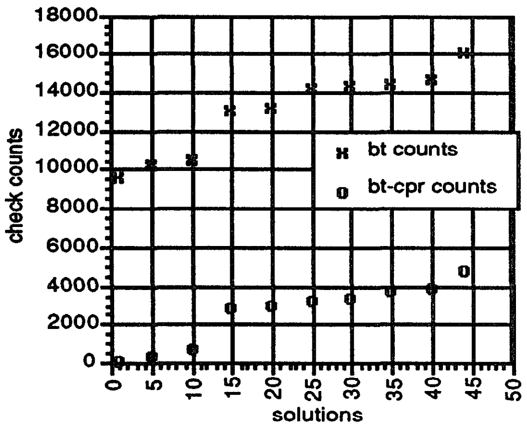
b) BT cpu time to BT-CPR cpu time



c) BT-CPR checks per second to BT checks per second

Figure 2. Performance ratios.

product containing 6 solutions, then a cross product with 15 more, and so on.



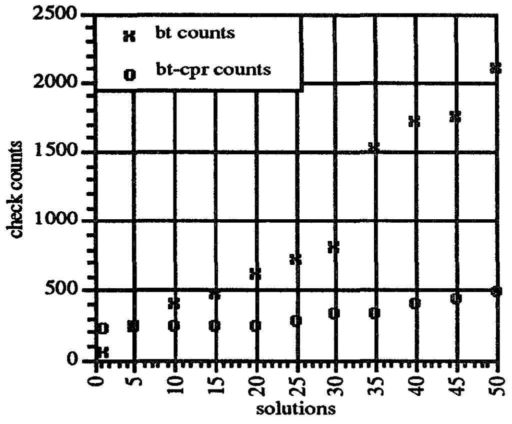


Figure 3. Asking for x solutions.

In the first case, a problem with 44 solutions out of 2,880,000 possibilities, BT-CPR is superior even when only a single solution is sought. In the second, a problem with 7,912 solutions out of 27,993,600 possibilites, BT-CPR requires more checks initially, but quickly becomes increasingly advantageous.

It is even possible for BT-CPR to require less effort than BT when the problem *has only* one solution. For one of our problems, which did happen to have only one solution, out of 405,000 possibilities, BT-CPR found that solution with 765 constraint checks while BT required 1819 constraint checks. (BT-CPR required 456 more constraint checks to go on and determine that there were no more solutions; BT required 1331 more.)

Applying CPR to Forward Checking

Forward checking works by "looking ahead". When a value is chosen for a variable we examine all the remaining uninstantiated variables and remove from their domains any values inconsistent with the new choice.

Again we add a "generate and merge" perspective, utilizing cross products. A node in the search tree will have associated with it a set of domains for uninstantiated variables, to be considered lower in the tree.

In expanding a given node in the search tree we construct children corresponding to each of the values in the domain of the next variable, and for each of these values we prune the domains of uninstantiated variables with a forward checking process. Then we merge values for which these sets of domains are identical.

For example, if values a and b for V are both consistent with {c d e} for W and {f g} for X, then a and b can be merged into a single child {a b}. The cross product involving {a b} and the sets at the nodes above {a b} in the tree will represent a set of partial solutions, and when we reach the bottom of the search tree, a set of solutions. (Values a and b are consistent with values above them in the search tree; this was assured by similar look ahead earlier in the search process.)

An algorithm augmenting forward checking with CPR, FC-CPR, is shown in Figure 4. As before the process is basically depth-first, though generating all the children at a node at once, in order to allow merging among the children.

The algorithm incorporates another improvement to forward checking, which utilizes a cross product representation. This is used here with FC-CPR, but could also be applied separately to forward checking. When search reaches a point where none of the remaining variables directly constrain each other (i.e. they form a "stable set", see [Freuder and Quinn 85]), there is no point in continuing. All the remaining values have been (forward) checked against all previous choices already; all combinations will work. Thus the cross product of the domains of the remaining variables can be added to the partial solution(s) represented at that point in the search tree, and these combinations can be reported as solutions.

As before, adding CPR to forward checking can not make things worse for us, and should in most cases improve matters.

Theorem 2. Augmenting forward checking with CPR will never increase the number of constraint checks required to search for all solutions to a CSP or to determine that an unsolvable problem has no solution.

Proof: Employ a similar reduction argument to that used for Theorem 1. •

While we have not yet implemented forward checking augmented by CPR, we found, in earlier experiments, that even BT-CPR can be orders of magnitude more efficient than forward checking for suitably weakly constrained problems. For example, on the sample problem cited above, where BT-CPR required 25,854 constraint checks and 3.21 seconds while backtracking required 77,867,372 constraint checks and 13,986.59 seconds, forward checking still took 33,789,729 checks and 2,029.5 seconds.

```
Procedure FC-CPR
Push onto Stack a list containing the empty
    cross procuct  and the cross product of all
    variable domains
While Stack is not empty
    Pop (Past-CP Future-CP) from Stack
    If there are no constraints among the variables
            represented by Future-CP (in particular,
            if there is only one variable represented)
        then
          return the solution represented by
              Past-CP X Future-CP
        else
          For each value, v, in the first variable, V,
                  represented in Future-CP
              form Future-CP-v by removing from
                  Future-CP the component corre-
                  sponding to V and removing all
                  values inconsistent with v
          For each maximal set of values, S, for
                  variable V, for which Future-CP-v
                  is identical for each v in S
              New-Past-CP <- Past-CP X S
              New-Future-CP <- Future-CP-v
              Push (New-Past-CP New-Future-CP)
                  onto Stack
```

Figure 4. Forward checking augmented by CPR.

Conclusion

This paper introduces a new representation for partial solutions to constraint satisfaction problems. This representation can be used with the standard CSP algorithms, backtracking and forward checking. When searching for all solutions or discovering that a problem is insoluble, CPR is guaranteed not to require any additional consistency checks.

Experiments on random problems demonstrate that CPR can in fact greatly reduce the number of constraint checks required in many cases, and cpu times demonstrate that the savings can be far more important than CPR overhead. Analysis of these experiments provides heuristic sufficient conditions for CPR to excel CPR may also be of help when searching for subsets of solutions or even a single solution.

Acknowledgments

Part of this work was done while the second author was a Visiting Scientist at the MIT Artificial Intelligence Laboratory. Richard J. Wallace and Karl Gevecker wrote the random problem generator.

References

[Cheeseman, Kanefsky and Taylor 91] Cheeseman, P., Kanefsky B. and Taylor, W., Where the really hard problems are, *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, 331-337, 1991.

[Freuder and Quinn 85] Freuder, E. and Quinn, M., Taking advantage of stable sets of variables in constraint satisfaction problems, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1985.

[Golumb and Baumert 65] Golomb, S. and Baumert, L., Backtrack programming, *JACM 12*, 516-524, 1965.

[Haralick and Elliott 80] Haralick R. and Elliott, G., Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence 14*, 263-313, 1980.

[Mackworth 87] Mackworth, A., Constraint satisfaction, in *Encyclopedia of Artificial Intelligence*, S. Shapiro, ed., vol. 1, John Wiley & Sons, New York, 205-211, 1987.

[Meseguer 89] Meseguer, P., Constraint satisfaction problems: an overview, *AI Communications 2*, 3-17, 1989.

[Nadel 89] Nadel, B., Constraint satisfaction algorithms, *Computational Intelligence*, Vol. 5, No. 4, 1989, pp. 188-224, 1989.

[Wu 91] Wu, T., Domain structure and the complexity of diagnostic problem solving, *Proceedings of the Ninth National Conference on Artificial Intelligence*, 855-861, 1991.