Neighborhood Inverse Consistency Preprocessing

Eugene C. Freuder and **Charles D. Elfe**

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

Abstract

Constraint satisfaction consistency preprocessing methods are used to reduce search effort. Time and especially space costs limit the amount of preprocessing that will be cost effective. A new form of consistency preprocessing, neighborhood inverse consistency, can achieve more problem pruning than the usual arc consistency preprocessing in a cost effective manner. There are two basic ideas: 1) Common forms of consistency enforcement basically operate by identifying and remembering solutions to subproblems for which a consistent value cannot be found for some additional problem variable. The space required for this memory can quickly become prohibitive. Inverse consistency basically operates by removing values for variables that are not consistent with any solution to some subproblem involving additional variables. The space requirement is at worst linear. 2) Typically consistency preprocessing achieves some level of consistency uniformly throughout the problem. A subproblem solution will be tested against each additional variable that constrains any subproblem variable. Neighborhood consistency focuses attention on the subproblem formed by the variables that are all constrained by the value in question. By targeting highly relevant subproblems we hope to "skim the cream", obtaining a high payoff for a limited cost.

Introduction

Key Ideas

Many problems in artificial intelligence can be represented as constraint satisfaction problems. Preprocessing the problem representation to achieve limited consistency is often used to reduce problem solving effort. The most common preprocessing achieves only very local consistency. Higher order consistency techniques can further reduce subsequent effort, but the preprocessing effort may not be cost effective. The preprocessing time may exceed the subsequent savings, or the cost of storing the results of the preprocessing may be excessive. The space problem in particular has been little studied, but can be crucial for large scale realistic problems.

Neighborhood inverse consistency is a new form of consistency that achieves higher order consistency while addressing these cost concerns, especially the space issue. There are two basic ideas:

Inverse Consistency: Common forms of consistency enforcement basically operate by identifying and remembering solutions to subproblems for which a consistent value cannot be found for some additional problem variable. The space required for this memory can quickly become prohibitive. Inverse consistency basically operates by removing values for variables that are not consistent with any solution to some subproblem involving additional variables. The space requirement is at worst linear; if the potential variable values are already represented explicitly, even high order inverse consistency processing can actually save space.

Neighborhood Consistency: Typically consistency preprocessing achieves some level of consistency uniformly throughout the problem. A subproblem solution will be tested against each additional variable that constrains any subproblem variable. Neighborhood consistency focuses attention on the subproblem formed by the variables that are all constrained by the value in question. By targeting highly relevant subproblems we hope to "skim the cream", obtaining a high payoff for a limited cost.

Combining these two ideas give us neighborhood inverse consistency. We present experimental evidence that suggests that for an interesting class of problems neighborhood inverse consistency preprocessing can be superior to conventional preprocessing methods.

In Section 2 we will present the basic idea of inverse consistency, including neighborhood inverse consistency as a special case. In Section 3 we present preprocessing algorithms that we tested. In Section 4 we describe experimental results with these algorithms. Section 5 presents conclusions and directions for further work.

Related Work

Freuder introduced, but did not implement, inverse consistency, as $(1, k-1)$-consistency in (Freuder 1985). In (Prosser 1993b) there is a form of "directed consistency" learning that might be viewed as acquiring some partial inverse consistency during the search process. Directed and adaptive consistency (Dechter & Pearl 1987) might be viewed as limited forms of partial neighborhood consistency preprocessing, employing standard, not inverse consistency. (Dechter & Meiri 1994) compares a variety of preprocessing methods experimentally.

Inverse Consistency

Constraint satisfaction problems (CSPs) involve finding values for problem variables subject to restrictions (*constraints*) on what combinations of values are allowed. A *solution* is an assignment of a value to each variable such that all the constraints are satisfied. Often, as here, we restrict our attention to *binary constraints* that involve two variables at a time. Binary CSPs can be represented by *constraint graphs*, where variables correspond to vertices, potential values are vertex labels, and constraints correspond to edges. Figure 1 is a sample CSP where the constraints are all the inequality constraint.

Most forms of consistency can be viewed as special cases of (i, j)-consistency (Freuder 1985). Basically, a problem is (i, j)-consistent if any solution to a subproblem of i variables can be extended to a solution including any j additional variables. When i is $k - 1$ and j is 1 we have k-consistency (Freuder 1978). If k is 2 we have arc consistency (AC) (Mackworth 1977); if k is 3 we have path consistency (PC) (Montanari 1974).

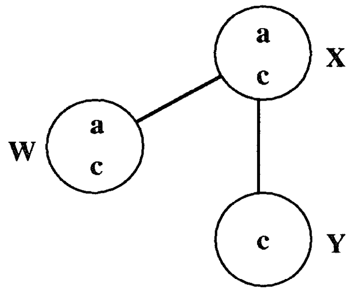

Figure 1. Path inverse consistency.

When i is 1 and j is $k - 1$ we have k inverse consistency. When k is 3 we have path inverse consistency (PIC). (Arc inverse consistency is no different from arc consistency.) In Figure 1, there is no solution for variables X and Y that is consistent with the choice of a for W. This implies that we can eliminate a as a potential value for W; we will say that we delete a from the domain of W.

The variables joined by an edge to a variable in a constraint graph are called its neighborhood. Basically, neighborhood inverse consistency (NIC) enforces, for each variable V, k inverse consistency for the k variables in the neighborhood of V.

The subproblem induced by a set of variables, S, corresponds to S and all the edges between two variables in S. A value v for variable V is consistent with a value u for variable U if those values satisfy, i.e. are allowed by, the constraint between V and U. A value v is consistent with a solution to a subproblem, if it is individually consistent with each of the values in the solution; in this case we also can say that the solution is consistent with v.

A constraint graph is neighborhood inverse consistent if, given any value v for any variable V, we can find a solution to the subproblem induced by the neighborhood of V that is consistent with v.

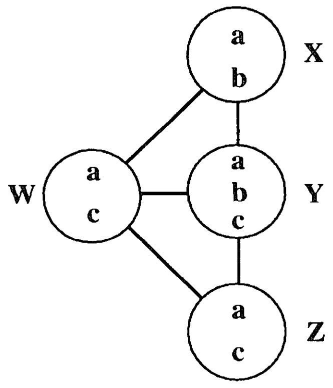

Figure 2. Neighborhood inverse consistency.

For example, In Figure 2, there is no solution for the neighborhood of variable W that is consistent with the choice of a for W. This implies that we can eliminate a as a potential value for W.

All these forms of consistency are defined locally, but enforcing this local consistency can propagate. Deleting a value because it is locally inconsistent can make some other value inconsistent that depended on the deleted value; that value in turn can be deleted. Since experience has shown that sometimes less consistency pruning is more cost effective, we also test a limited neighborhood inverse consistency algorithm that only processes each variable once for neighborhood consistency, and thus does not necessarily take into account all such propagation to fully achieve neighborhood inverse consistency. We call this ONIC, for one pass neighborhood inverse consistency.

The time complexity of k inverse consistency is comparable to that of k-consistency. In terms of space complexity, however, inverse consistency has a considerable advantage. In general, achieving k-consistency requires creating and storing constraints involving $k-1$ variables, which can require $O(d^{k-1})$ space, assuming d potential values for each variable. Achieving inverse consistency only requires specifying that values cannot be permitted for variables. At worst this requires linear space. If the potential values are already listed explicitly, inverse consistency can reduce space requirements by deleting some of these values.

The time complexity of k-consistency is polynomial with the exponent dependent on k. The time complexity of neighborhood inverse consistency is polynomial with the polynomial dependent on the maximum degree of the constraint graph. However, for the problems we consider experimentally below, neighborhood inverse consistency preprocessing is considerably faster than even path consistency preprocessing.

Neighborhood inverse consistency processing will delete a value, v, from a variable V that is not consistent with any value for an individual neighboring variable U. (If this is the case, clearly v is not consistent with any solution for the neighboring subproblem.) This deletion can propagate. Thus neighborhood consistency preprocessing leaves the problem arc consistent. Arc consistency is also achieved by path inverse consistency preprocessing since all nodes sharing edges with V are examined, and the deletions are allowed to propagate. Thus we have the following simple proposition:

Proposition: Neighborhood and path inverse consistency both imply arc consistency.

In particular, this means that NIC preprocessing is "stronger" than AC preprocessing. It deletes at least as many values.

Algorithms

We tested various old and new preprocessing schemes. In each case the search algorithm used after preprocessing was FC-CBJ-DMD, which is a combination of forward checking with conflict-directed backjumping and dynamic minimum domain size ordering. This is a good modern search algorithm. The code for FC-CBJ (Prosser 1993a) is from Peter van Beek's code library (available on the Internet via anonymous ftp at ftp.cs.ualberta.ca); the dynamic ordering was added for this paper. The preprocessing algorithms arc consistency and path consistency are also from van Beek's library. They are implementations of AC-3 and PC-2 algorithms, respectively, similar to those described by Tsang in (Tsang 1993), but make use of a stack to maintain the edges and paths, respectively, that need to be reexamined.

Path inverse consistency (PIC), neighborhood inverse consistency (NIC), and one pass neighborhood inverse consistency (ONIC) were all coded for this paper. All algorithms are coded in C.

Shown in Figure 3 is an algorithm for achieving neighborhood inverse consistency(NIC). An agenda is used to keep track of those variables that still need to be examined, and so initially, all variables are placed on the agenda. The algorithm proceeds by examining each value a in the domain of a variable v to see whether or not there is a solution among the variable's neighbors. If when a is assigned to v, no solution can be found for the variable's neighbors, i.e. the induced subproblem consisting of v and all neighbors of v, then a is removed from the domain of v. In our implementation, line 7 is performed by creating the subproblem induced by v (with the single value a) and $N(v)$, and then running FC-CBJ-DMD on the subproblem to see if there is a solution.

When a value is deleted from a domain of a variable, v, the deletion may affect all of the variables that share a constraint with v. As a result, all adjacent variables, which are precisely the neighbors, are inserted into the agenda. In this fashion, domain value deletions are allowed to propagate.

```
ALGORITHM NIC
1. insert each variable $v$ into agenda $A$, a queue
     without duplicates

2. while the agenda $A$ is not empty
3.   extract a variable $v$ from front of agenda $A$
4.   let the neighborhood $N(v)$ be the set of all
          nodes which share an edge with $v$
5.   set the flag deleted to false

6.   for each value $a$ in domain of $v$
7.     if there is no solution for $N(v)$ when $a$
          is assigned to $v$
8.       remove value $a$ from the domain of $v$
9.       change deleted to true
10.      if the domain of $v$ is empty
11.        return wipeout

12. if deleted is true
13.   insert all $x \in N(v)$ into agenda $A$, which
          are not already in $A$
14. return consistent
```

Figure 3: Algorithm for performing neighborhood inverse consistency.

The algorithm for ONIC is identical to that for NIC, except that each variable is examined only once; no further propagation is performed. In our implementation, the variables are examined in lexical order. Removal of lines 5, 9, 11, and 12 from Figure 3 results in an algorithm for ONIC, since it is these lines which are responsible for allowing propagation beyond one pass to take place.

PIC is similar to NIC, with the exception of line 7. Instead of selecting all neighbors of v and trying to find a solution among them, for each of the values in the domain of v, PIC generates all combinations of three distinct nodes which include v and verifies for each 3-tuple that a solution exists with a for v.

As a result of a few observations, it follows that not all 3-tuples need to be generated. For example, some are duplicates, like (x, y, z) and (x, z, y), which contain the same set of variables. Yet other 3-tuples, like those with no constraints are unable to force a domain value to be deleted.

In our implementation of PIC, only those sets of three nodes, where at least one node shares an edge with v are generated. If v is connected to neither of the two other nodes x and y, and they do not share an edge, no preclusion would have taken place, since there are no constraints. If v is connected to neither x nor y, but they do share an edge, then if any value for v could be precluded, it is clearly because there is no solution to the subproblem induced by x and y, regardless of the value of v. In this case there will be no solution to the problem as a whole. In our implementation, this situation is recognized at the point that PIC examines either x or y.

Experiments

Test problems were created using a random problem generator requiring four inputs: the number of variables, the number of values for each variable, the tightness, and the density. We chose 100 variables and 6 values to supply us with reasonably difficult problems.

Roughly speaking the *tightness* is a measure of how constraining the constraints are, and the *density* is a measure of how full of edges the constraint graph is. We ensure that our constraint graphs are connected.

Recent work on "really hard problems", e.g. (Cheeseman, Kanefsky, & Taylor 1991), suggests that most hard problems exist at the point where a phase transition from problems with solutions to those without occurs, and that for given size problem (variables and values) and a given tightness, we can find this transition area, or complexity peak, by varying the density.

We tested problems at a midrange tightness of .5. In other words, there is a 50% probability that an individual pair of values will be allowed by a constraint. Problems were generated and run around the transitional area. We took another "cut" through the "complexity ridge" at a lower tightness of .4. (At .6 the problems were very easy.)

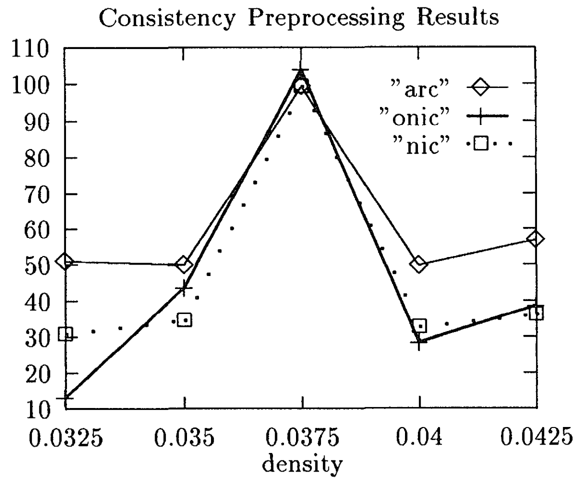

Figure 4. Running time in seconds for problems at tightness = 0.4.

We tested search alone (FC-CBJ-DMD). We also tested search preceded by each one of the preprocessing methods: AC, PC, PIC, NIC, ONIC. In Figures 4 and 5 we plot the results for the 3 best options: search preceeded by AC, NIC and ONIC. Figure 4 shows the average effort to solve sets of 25 problems at several different density values around the peak region for tightness of .4. Figure 5 does the same for tightness of .5.

Along the density axis each .01 of added density corresponds to roughly 50 more constraints. At 0 density we would still have roughly 100 constraints, forming a minimally connected, tree-structured constraint graph. At .02 density the average degree of a node in the constraint graph is roughly 4; i.e. each variable is involved in an average of roughly 4 constraints. This may seem sparsely constrained, but there is reason to believe that many practical problems are large and relatively sparse.

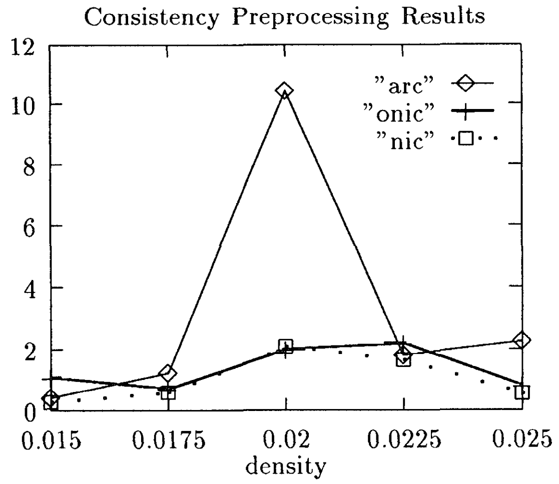

Figure 5. Running time in seconds for problems at tightness = 0.5.

Our new algorithms do well in general, doing better for the hardest problems at .5 than at .4. Note the y-axes are different in the two figures; the denser problems at .4 are considerably harder in general than the sparser ones at .5. Table 1 details the performance on the problems at tightness .5, density .02, where we see a peak in Figure 5. The time for preprocessing and the total time for preprocessing plus subsequent search are given for all the preprocessing methods (e.g. AC denoting AC preprocessing, and AC+S denoting AC preprocessing plus search), and we see the time for search without any preprocessing (S denoting FC-CBJ-DMD). Underlining indicates that preprocessing deleted all values from a variable domain, so no further search was required. Boldface identifies the minimum times. We indicate whether or not there is a solution. Notice that our new proprocessing can excel in either case.

Notice that the maximum effort for NIC preprocessing plus search is only about 22 seconds, while the maximum effort for AC preprocessing plus search is over 3 minutes. For the problem that requires over 3 minutes of AC plus search, NIC plus search finishes in about a tenth of a second. NIC is the outright "winner" on relatively few problems. However, the wins are more significant than the losses.

#	sol	S	AC	AC+S	PC	PC+S	PIC	PIC+S	ONIC	ONIC+S	NIC	NIC+S
1	yes	1.48	0.05	1.57	142.01	142.03	15.81	17.16	0.05	**0.30**	0.13	1.47
2	no	11.46	0.07	1.31	114.67	$\underline{114.67}$	19.01	19.92	0.06	**0.78**	0.15	1.18
3	no	17.87	0.05	1.68	86.12	$\underline{86.12}$	15.45	15.88	0.05	**0.49**	0.11	0.54
4	no	0.78	0.07	**0.16**	52.68	$\underline{52.68}$	21.31	21.32	0.06	0.18	0.21	0.22
5	yes	5.03	0.05	**0.12**	214.27	217.16	16.87	17.27	0.06	0.89	0.12	0.51
6	yes	0.70	0.06	**0.09**	133.70	133.72	16.84	16.87	0.05	0.09	0.11	0.14
7	yes	**0.06**	0.06	1.78	107.67	107.69	15.51	17.25	0.06	0.11	0.11	1.82
8	no	3.25	0.15	$\underline{\mathbf{0.15}}$	62.52	$\underline{62.52}$	30.75	$\underline{30.75}$	0.06	0.23	0.27	$\underline{0.27}$
9	no	86.77	0.05	29.78	271.35	$\underline{271.35}$	15.65	37.00	0.07	21.55	0.11	**21.45**
10	no	45.21	0.06	4.82	97.17	$\underline{97.17}$	18.09	18.33	0.06	0.41	0.13	**0.37**
11	no	32.95	0.08	**0.10**	106.45	$\underline{106.45}$	32.60	32.61	0.06	1.27	0.26	0.27
12	yes	**0.09**	0.06	6.58	255.04	255.06	16.90	24.02	0.06	7.51	0.13	7.34
13	no	14.56	0.06	0.38	79.92	$\underline{79.92}$	28.19	$\underline{28.19}$	0.06	**0.19**	0.34	$\underline{0.34}$
14	yes	**0.07**	0.05	1.12	291.25	291.27	15.34	16.32	0.06	5.03	0.11	1.08
15	no	49.53	0.07	0.99	75.91	$\underline{75.91}$	17.61	18.16	0.06	0.88	0.20	**0.73**
16	no	46.54	0.07	0.08	46.24	$\underline{46.24}$	14.14	$\underline{14.14}$	0.06	**0.07**	0.18	$\underline{0.18}$
17	no	15.10	0.07	0.10	47.13	$\underline{47.13}$	19.96	20.01	0.05	**0.07**	0.14	0.20
18	yes	**0.08**	0.05	191.41	131.45	131.48	14.15	14.17	0.05	1.29	0.11	0.13
19	yes	57.42	0.06	8.13	250.73	250.75	16.38	21.57	0.06	**0.61**	0.12	5.33
20	no	37.86	0.06	2.15	140.52	$\underline{140.52}$	18.95	22.23	0.06	**1.69**	0.13	3.41
21	no	212.15	0.05	1.18	104.57	$\underline{104.57}$	19.36	20.35	0.06	1.55	0.14	**1.13**
22	yes	1.55	0.06	3.29	148.57	148.59	22.20	23.07	0.06	1.04	0.15	**1.02**
23	yes	42.11	0.05	2.51	211.61	211.78	17.18	19.64	0.06	**1.47**	0.11	2.58
24	no	2.29	0.08	**0.08**	39.89	$\underline{39.89}$	9.44	$\underline{9.44}$	0.06	0.08	0.14	$\underline{0.14}$
25	no	135.70	0.08	1.31	53.30	$\underline{53.30}$	18.13	$\underline{18.13}$	0.06	2.07	0.25	$\underline{\mathbf{0.25}}$
avg	40%	32.82	0.06	10.43	130.59	130.72	18.63	20.55	0.06	**1.99**	0.16	2.08

Table 1. Running time in seconds for twenty-five random problems with 100 variables, a domain size of 6, tightness = 0.5, and density = 0.02.

Apparently the additional pruning done by NIC can sufficiently improve performance on hard problems to outweigh the rather small processing penalty involved in achieving NIC rather than AC. More efficient AC preprocessing, or partial AC preprocessing like directed arc consistency, could lower the AC preprocessing costs further, but would not address the real issue: the extreme search cost penalty that is sometimes paid for doing less preprocessing pruning than NIC. Finally, recall that NIC does not incur high space costs.

Figures 6 and 7 plots the results for the 10 problems in each problem set that are most difficult for AC preprocessing plus search, with the problems aligned on the horizontal axis in order of difficulty. These plots demonstrate how NIC is especially successful at keeping down the maximum effort required, and outperforming AC on the hardest problems. Such behavior would seem especially useful for real-time problems involving human interaction, where it is well known that large fluctuations in response time are especially annoying.

Conclusion

We have demonstrated that inverse consistency can effectively introduce higher order consistency without a significant space penalty. Neighborhood consistency limits the time commitment by targeting highly relevant subproblems. Neighborhood inverse consistency has shown some success at outperforming the standard arc consistency preprocessing by doing more pruning at small additional cost. It appears that this can ameliorate the risk of encountering unusually costly problems.

There are several opportunities for further research:

- The algorithms have been implemented in rather straightforward fashion; there is considerable room for improvement. In particular, some redundant processing of overlapping subproblems may be avoided.
- The information obtained during subproblem solution may be profitably retained, at some space cost.
- Inverse consistency can be interleaved with search in hybrid algorithms, analogous to forward checking.

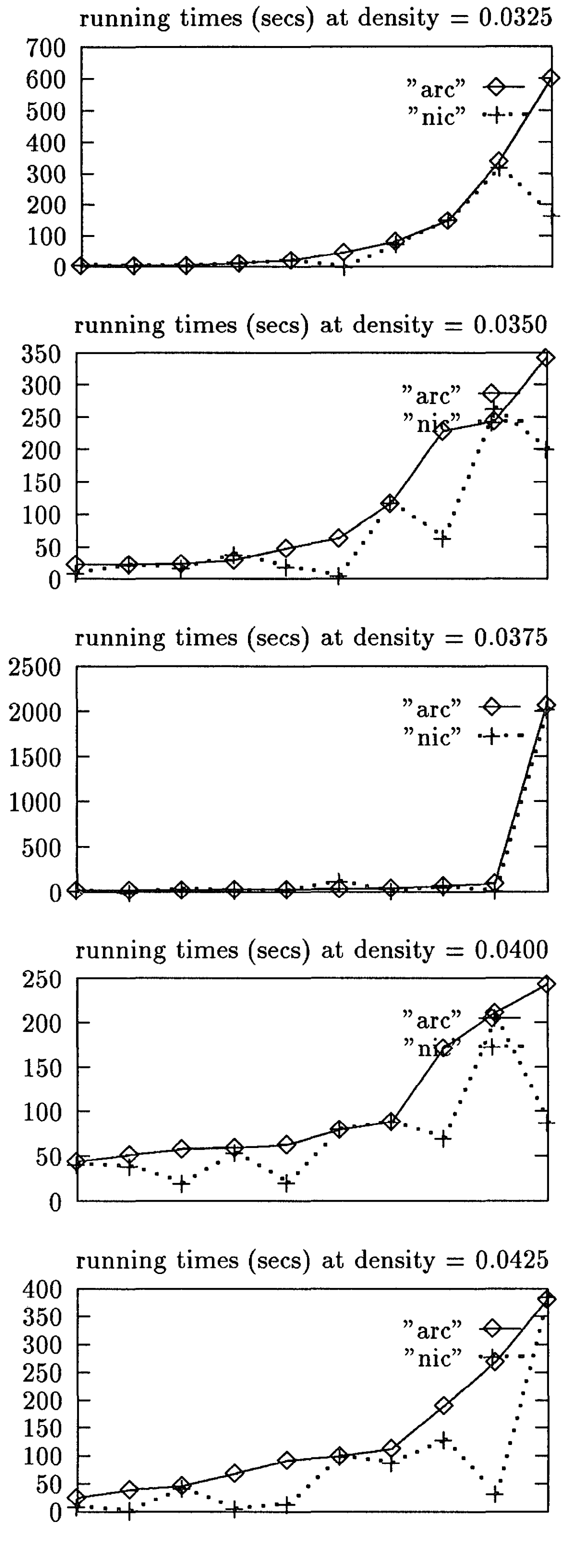

Figure 6. Most difficult ten problems for arc consistency ordered by difficulty for arc consistency at tightness = 0.4 and different densities.

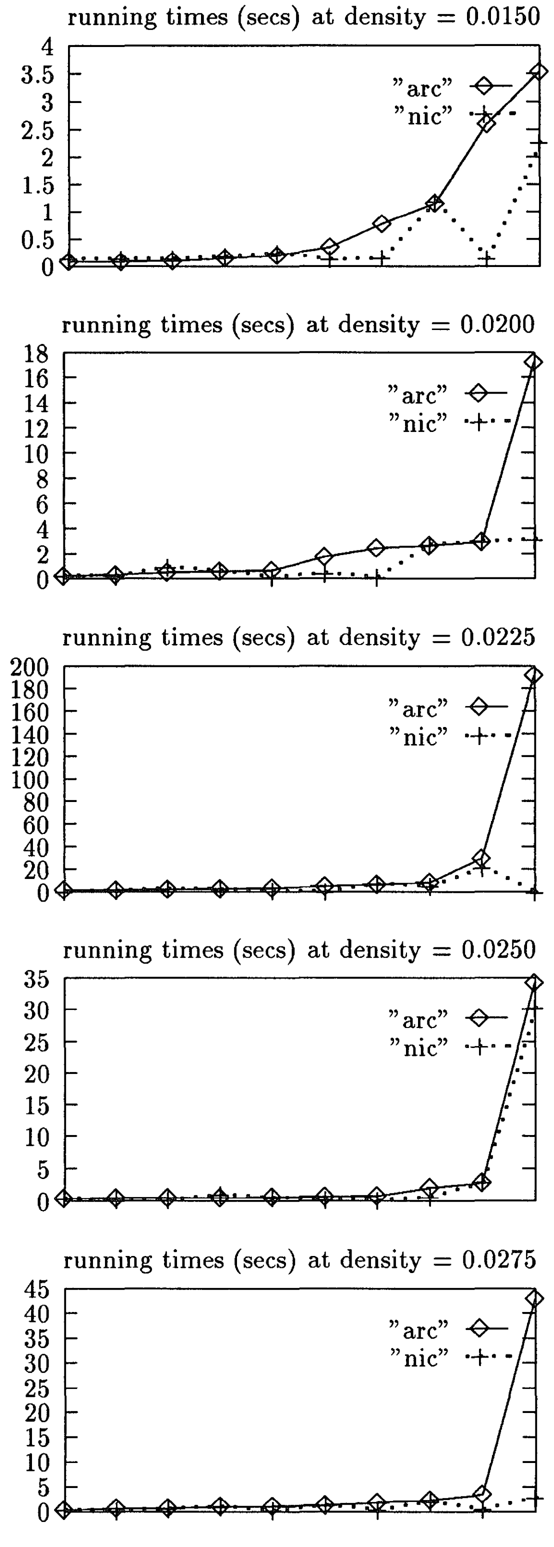

Figure 7. Most difficult ten problems for arc consistency ordered by difficulty for arc consistency at tightness = 0.5 and different densities.

- Neighborhood consistency, as opposed to neighborhood inverse consistency, should be considered.
- Higher order *k* inverse consistency should be studied. Inverse consistency can be combined with higher order k-consistency.
- Other forms of "targeted" consistency, as opposed to neighborhood consistency, based on syntactic or semantic understanding of individual problems, may prove useful.

Acknowledgements

This material is based on work supported by the National Science Foundation under Grant No. IRI-9207633 and Grant No. IRI-9504316. We thank Richard Wallace for his contributions to this work.

References

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

Dechter, R. and Meiri, I. 1994. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. *Artificial Intelligence*, 68(2), 211–241.

Dechter, R. and Pearl, J. 1987. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34(1), 1–38.

Freuder, E. 1978. Synthesizing Constraint Expresssions. *Communications of the ACM*, 21(11): 958–966.

Freuder, E. 1985. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4): 755–761.

Mackworth, A. 1977. Consistency in Networks of Relations, *Artificial Intelligence*, 8:99–118.

Montanari, U. 1974. Networks of constraints fundamental properties and applications to picture processing, *Information Sciences*, 7:95–132.

Prosser, P. 1993a. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299.

Prosser, P. 1993b. Domain filtering can degrade intelligent backtracking search. In Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, 262–267.

Tsang, E. 1993. *Foundations of Constraint Satisfaction.* Academic Press, London.