Inference-Based Constraint Satisfaction Supports Explanation

Mohammed H. Sqalli and Eugene C. Freuder
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
msqalli, ecf@cs.unh.edu

Abstract

Constraint satisfaction problems are typically solved using search, augmented by general purpose consistency inference methods. This paper proposes a paradigm shift in which inference is used as the primary problem solving method, and attention is focused on special purpose, domain specific inference methods. While we expect this approach to have computational advantages, we emphasize here the advantages of a solution method that is more congenial to human thought processes. Specifically we use inference-based constraint satisfaction to support explanations of the problem solving behavior that are considerably more meaningful than a trace of a search process would be. Logic puzzles are used as a case study. Inference-based constraint satisfaction proves surprisingly powerful and easily extensible in this domain. Problems drawn from commercial logic puzzle booklets are used for evaluation. Explanations are produced that compare well with the explanations provided by these booklets.

Introduction

Overview

Constraint satisfaction problems (*CSPs*) involve finding values for problems variables subject to restrictions on which combinations of values are acceptable. They are solved using search (e.g. backtrack) and inference (e.g. arc consistency) methods. Recently there has been considerable attention to "pure search" methods that use local search techniques (e.g. simulated annealing) without inference. In this paper we propose "pure inference" problem solving. This may also have efficiency advantages, but we focus here on the support that these methods provide for explanation.

In interacting with human users it may not be enough to simply supply a solution to a problem. The user may want an "explanation": how was the solution obtained or how is it justified? The computer may be functioning as a tutor or as a colleague. The user may want to consider alternative solutions, or need to relax constraints to permit a more complete solution. In these situations it is helpful if the computer can think more like a person, and people tend to use inference to avoid massive search.

Tracing through the "thought processes" of a backtrack search problem solver would result in an "explanation" of the form: "I tried this and then I tried that, and it didn't work so I backed up and tried the other ... " Following the thought processes of a simulated annealing local search problem solver is even less satisfying: "I tried this and then I tried that, and it didn't work, but I decided to use it anyway ... "

More satisfying are inference-based explanations that contain statements like "X had to be v because ...". We propose to obtain these by building problem solvers that combine established general purpose CSP inference methods, like arc consistency, with special purpose, domain specific inference methods. We illustrate this approach with a case study: logic puzzles.

Logic puzzles are a classic constraint solving testbed (Dechter 86). They are used in some form on the Graduate Record Examination and the Law School Aptitude Test. Booklets of logic puzzles are sold on newstands. Simple logic puzzles can be regarded as coloring problems. Logic puzzles are related to scheduling and temporal reasoning problems.

Much attention recently has been devoted to homogeneous random problems. However, real problems can have structure, which can be exploited. Logic puzzles provide us with a simple structure, and we propose specialized inference methods that exploit that structure.

We demonstrate how surprisingly powerful these inference methods can be. (Of course, such inference methods may not be complete, but unlike local search, the results of unsuccessful inference-based problem solving naturally provide useful preprocessing for subsequent search.) Then we obtain explanations from our inference-based problem solving that are not unlike those found in commercial logic puzzle booklets. We measure the success of our approach on puzzles taken from such booklets, as well as on the classic Zebra puzzle.

Relation to Previous Work

It has long been known (Freuder 78) that CSPs can be solved by pure inference, involving higher and higher order consistency processing. For some problem structures it has been proven that limited higher order consistency processing is sufficient to leave only backtrack-free search (Dechter & van Beek 95). However, the efficiency of obtaining even moderately high order consistency processing can be problematic in practice.

Our inference methods utilize very restricted forms of higher order consistency. (We also employ a general low order consistency technique: arc consistency.) We obtain partial higher order consistency in a manner targeted to take advantage of the semantic and syntactic structure of a specific class of problems. For example, (Smith 92) shows that path consistency preprocessing can greatly reduce search when solving logic puzzles. Our transitivity method implements a form of partial path consistency. Other methods we use can implement even higher order consistency in a restricted manner that limits their complexity.

Our approach is closely related to the use of specialized or redundant constraints, especially higher order constraints, and associated constraint propagation methods found in the constraint programming community (Beldiceanu & Coontejean 94; Baptiste & Le Pape 95). The work of Regin on "constraints of difference", (Regin 94) can be viewed as a generalization of our clique consistency method.

The connection between constraint solving and inference methods for theorem proving or truth maintenance is well known (de Kleer 89; Mackworth 92). We are not simply reinventing the wheel in this respect, but rather extending the CSP consistency propagation approach. Explanation based on linking inference methods and English phrases, is, of course, a classic AI technique. However, we are not aware of comparable work on explanation in a CSP context.

Example

Figure 1 shows an example of a logic puzzle, taken from the April 1995 issue of *Dell Logic Puzzles*. The puzzle is called The Flumm Four (devised by Ida J. Mason).

Figure 2 shows a constraint graph representation of this puzzle as a CSP. *Constraint graphs* represent variables as vertices, labeled with the *domain* of potential values, and *binary constraints* (on two variables) as edges. Here the variables are instruments and songs, the values are musicians (represented by their initials) and the binary constraints are all not-equal constraints. These constraints are derived from the puzzle statement.

We see the typical logic puzzle structure. All the domains are equal, except that some clues that correspond to *unary constraints* (on a single variable) have been implemented as domain reductions (shown here as crossouts). The variables are partitioned into equal sized *cliques* (in a clique all pairs of variables are joined by an edge), where the edges correspond to inequality constraints. Each variable appears in one and only one clique, and the number of variables in each clique (its *size*) is the same as the size of the domains (before unary constraint reductions). Here one clique corresponds to instruments (each musician plays a different instrument) and one to songs (each musician wrote a different song).

The Flumm Four is a new punk rock group from Liverpuddle. The four musicians (Furble, Icky, Muke, and Slyme) play bass, drums, guitar, and keyboard, not necessarily in that order. They've taken the music world by storm with four hit songs from their first album. Each of these songs - "Grossery Store," "Heartburn Hotel," "Love Is Garbage," and "2 Yuk 4 Words"- was written by a different member of the band. Can you match each musician with his instrument and his song?
1- Icky and Muke are, in one order or the other, the composer of "2 Yuk 4 Words" and the keyboardist.
2- Neither Slyme nor the composer of "Heartburn Hotel" plays guitar or bass.
3- Neither the bass player (who isn't Icky) nor the drummer wrote "Love Is garbage."

Figure 1. Statement of the Flumm Four puzzle.

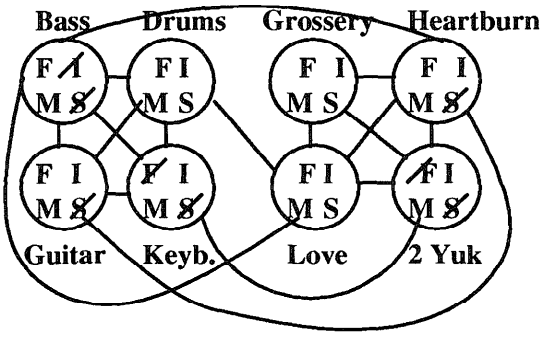

Figure 2. Constraint network for Flumm Four puzzle.

Slyme doesn't play keyboard, guitar, or bass, so he plays drums. His song isn't "2 Yuk 4 Words", "Heartburn Hotel" or "Love Is Garbage", so it's "Grossery Store.". The composer of "Heartburn Hotel" doesn't play bass or guitar, so he plays keyboard. The composer of "Love Is Garbage" doesn't play bass, so he plays guitar, and by elimination the bassist wrote "2 Yuk 4 Words." The bassist isn't Icky, so he's Muke, and Icky is the keyboardist. By elimination Furble is the guitarist.
In summary:
Furble, guitar, "Love Is Garbage"
Icky, keyboard, "Heartburn Hotel"
Muke, bass, "2 Yuk 4 Words"
Slyme, drums, "Grossery Store"

Figure 3. Explanation from the logic puzzle booklet.

Figure 3 reproduces the solution (what we call an *explanation*) from the back of the puzzle booklet. (The booklet also associates pieces of the explanation with specific clues.) Figure 4 contains the explanation produced by our program.

The program executes a series of inferences. These inferences are translated into pieces of the explanation. For example, if we look at the instrument clique, we see that Slyme has already been ruled out as bass player, guitarist and keyboard player. From that we can infer that Slyme must be the drummer. This inference corresponds to the bit of explanation: Drums player must be Slyme because the others are not. In the constraint graph we eliminate the other three domain values for the drummer (which may lead to further inferences).

Love composer must be either Furble or Icky or Muke because Love composer is either Guitar player or Keyboard player. Drums player must be Slyme because the others are not. Grossery composer must be Slyme because the others are not. Bass player must be 2 Yuk composer because Bass player can't be any of the others. They must be Muke because that's all that's left. Guitar player must be Love composer because Guitar player can't be any of the others. Keyboard player must be Heartburn composer because Keyboard player can't be any of the others. Guitar player must be Furble because the others are not. Keyboard player must be Icky because the others are not. Love composer must be Furble because the others are not. Heartburn composer can't be Muke because Bass player is. Heartburn composer must be Icky because that's all that's left.
Result:
Furble: Guitar player, Love composer,
Icky: Keyboard player, Heartburn composer,
Mukc: Bass player, 2 Yuk composer,
Slyme: Drums player, Grossery composer,

Figure 4. Explanation produced by our program.

Inference

Process

We use a standard general purpose consistency inference method of arc consistency (*AC*) and several special purpose methods that we developed for logic puzzles. Consistency inference methods essentially add redundant constraints; when these are unary constraints, values are eliminated. The methods we use are sound, values will not be eliminated that can appear in a solution.

All logic puzzles should have a solution, ideally a unique solution; we will assume that the makers of our puzzles have successfully designed them to have unique solutions. Given this assumption, only one value in each domain can participate in a solution, and if we finish processing with a single value in each domain then we have found a solution. (Our methods could still be useful when there are multiple solutions or no solutions.)

By studying the Flumm Four puzzle, we came up with two special purpose inference methods that were sufficient, when added to AC, to solve the puzzle, i.e. reduce each domain to a single value. These methods, clique consistency (CC), and neighbor clique consistency (NCC), both exploit the clique structure of logic puzzles. We were a bit surprised that these three inference methods alone, AC, CC and NCC, were sufficient to completely solve the puzzle. We were even more surprised to find that these three methods solved a number of other puzzles. (Indeed even AC alone solved some puzzles.) However, by examining the failures we derived two additional inference methods: a Generalized NCC (GNCC) to be used in place of NCC, and a Transitivity method (really a specialized form of path consistency, another standard general purpose consistency inference method).

In the next subsection we describe these methods in more detail. The methods are combined by assigning an order to them and applying one after the other in a round robin fashion. (We need to keep reapplying the methods as deletions made by one method may lead to additional deletions by another.) An exception is Transitivity, which is applied first, and then reapplied whenever a new equality constraint is added. We experimented with different orderings (but found none with a major advantage). More sophisticated control schemes are certainly possible.

We started working with simple logic puzzles where the constraints were all not-equal constraints (and the puzzles were thus special case coloring problems). However, our approach easily generalized to accomodate a wide variety of logic puzzle constraints. In addition to not-equal we encountered: $=, <, >, \leq, \geq$, greater by at least 2; exactly before, exactly after; next to; two times less/more; the difference between two values is at least 2, greater/less than 2, equals 2, is either 1 or 2. AC simply uses these constraints in the usual way. The inference methods exploiting the clique structure all rely on inequality, and any constraint that involves inequality, e.g $<$, will do as well as not-equal. Transitivity at present requires equality, in conjunction with any other constraint.

Methods

Arc Consistency is a basic constraint inference technique. Given any value, v, for any variable, V, and any other variable, U, there must be a value, u, in the domain of U consistent with v, i.e. u and v together satisfy the constraint (if any) between U and V. If not, v can be deleted.

With inequality constraints, AC comes into play when the domain of one of the variables involved has been reduced to a single value. When that happens, the other variable cannot take on that value. In processing the Flumm Four example, at one point the domain of the Bass player is reduced to Muke. At that point AC infers that the Heartburn composer can't be Muke, because the Bass player is.

```
Clique_Consistency (V,D)
    /* V: Set of variables, D: Set of domains */
1.  Begin
2.  For each variable x in V
3.    For each value v in D[x]
4.      | counter <- 0;
5.      | For each variable y in
        |      the same clique as x
6.      |      | If v is not in D[y]
7.      |      | Then increment counter;
8.      | EndFor
9.      | If (counter = vars_per_clique - 1)
10.     | Then D[x] <- {v};
11.     |      Exit For;
12.     | EndIf
13.   EndFor
14. EndFor
15. End
```

Figure 5. CC algorithm.

Clique Consistency is an inference method that exploits the intra-clique structure. It exploits the fact that each variable in the clique of inequality constraints must be assigned a different one of the problem values. If all but one of these variables do not have a value v in their domains, CC concludes that the remaining variable X must be assigned the value v. For example, if we observe the constraint network representation of Flumm Four in Figure 2, we see that unary clues have already told us that Slyme cannot be the Bass player, the Guitar player or the Keyboard player. From this CC can conclude that the only possible value for the Drums player is Slyme. Figure 5 contains an algorithm for Clique Consistency.

Neighbor Clique Consistency is an inference method that exploits the inter-clique structure by looking at one variable against one neighbor clique. The term *neighbor* refers to the fact that there is an explicit or implicit constraint between the variable and at least one of the clique variables. We say that there is an *implicit* inequality constraint between two variables if the intersection of their domains is empty and there is no explicit inequality constraint between them. NCC uses the fact that all the cliques have the same structure with the same domains and the same number of variables. Therefore, each variable of one clique has a matching variable in any other clique that must be assigned the same value. If all but one of the variables in the same clique are different from one variable X not in the clique, NCC concludes that the remaining variable Y in the clique must have the same values as X. Two variables are *different* if there is either an implicit or explicit inequality constraint between them. The NCC conclusion is reflected in the restriction of the domains of both X and Y to the intersection of their current domains, and the addition of an equality constraint between X and Y to the constraint graph.

Figure 6 shows an example of NCC application from the Flumm Four puzzle. Here explicit inequality constraints between Heartburn and Guitar and Heartburn and Bass, combined with an implicit inequality (dashed line) between Heartburn and Drums, leads to the addition of an equality constraint between Heartburn and Keyboard (dotted line), and the deletion of F from Heartburn (a backslash distinguishes this new deletion from previous ones).

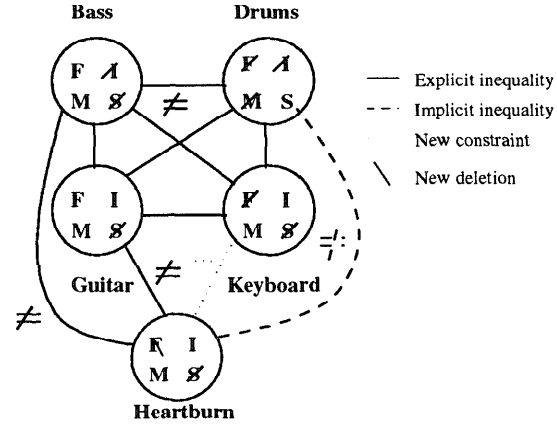

Figure 6. NCC example.

```
Generalized_Neighbor_Clique_Consistency (V,D,C)
    /* V: Set of variables, D: Set of domains */
    /* C: Set of Constraints */
1.  Begin
2.  For each variable x in V
3.   For each clique K not containing x
4.    counter <- 0;
5.    different_vars <- {};
6.    For each variable y in the clique K
7.     | If x and y are different
8.     | then increment counter;
9.     |      add y to different_vars;
10.    | else z <- y;
11.    | EndIf
12.   EndFor
13.   If (vars_per_clique - 1 = counter and no
          equality constraint between x and z)
14.   Then D[z] <- D[x] <-
                Intersection (D[x], D[z]);
15.        Cxz <- Czx <- {(i,i): i in D[x]};
                 /* Add new constraint to C */
16.   Else U[w] <- Union of D[w] such that:
                   w is not in different_vars;
17.        D[x] <- Intersection (D[x], U[w]);
18.   EndIf
19.  EndFor
20. EndFor
21. End
```

Figure 7. GNCC algorithm.

Generalized Neighborhood Clique Consistency generalizes NCC to exploit even partial information we get by looking at a neighboring clique. GNCC can infer statements such as: "This variable must match one of the following specified variables of another clique".

(CC might also be generalized to GCC which gives partial information about the same clique. GCC is not supported in our implementation.) GNCC states that if a variable X is different from m variables in a clique with k variables (and X is not in the clique), then it must have the same value as one of the other k-m variables in the clique. This means that the domain of X has to be a subset of the union of the domains of these k-m variables. Therefore, the domain of X is set to the intersection of the current domain with the union of the domains of the k-m variables. Figure 7 contains an algorithm for GNCC.

Transitivity inference concludes that if there is an equality constraint between X and Y and any other constraint, C, between Y and Z, and no constraint between X and Z, that we can add a constraint of type C between X and Z.

Explanation

Our initial motivation in this work was not simply to solve the logic puzzles, but to solve them in a manner that people would find natural to follow, and to explain how that solution was found. The solution section in the logic puzzle booklets contains an explanation of how to find the solution for each problem. These can be used by readers to provide hints when they are stuck in solving the puzzle, or to learn from, so they may do better on subsequent puzzles. Our goal was to present similar solutions.

It was immediately obvious that a trace of standard search techniques would not produce anything like a satisfactory explanation. This led to the inference-based approach, and the transformation of individual inferences into bits of explanation.

Explanation Templates

Each sentence in the explanation corresponds to a particular *inference event* produced by a particular inference method. Events are: deletion of a value, reduction of a variable domain, reduction of a domain to a single value, and adding an equality constraint between variables. Different combinations of method and event correspond to different *explanation templates*. Portions of these templates must be filled in to correspond to the specific variables, values or constraints involved in the specific inference event. We will now describe the explanation templates associated with each inference method, and provide a number of specific examples drawn from the Flumm Four explanation shown in Figure 4. (Flumm Four can be solved without GNCC and Transitivity, but they were used here.)

If application of AC reduces a domain to a single value, this calls up the template: *X must be v because that's all that's left.* In the Flumm Four explanation we see a specific instance of this template: Heartburn composer must be Icky because that's all that's left.

The other AC template is associated with AC simply deleting a value. The template is: *Y can't be v because X ....* The latter part of the template is filled in according to the type of constraint involved. For example, if we have an inequality constraint, the template becomes *Y can't be v because X is.* In Figure 4 we see: Heartburn composer can't be Muke because Bass player is. Another example: if the constraint is < the template becomes: *Y can't be v because X is before it.*

There is one template associated with CC. When CC concludes that a variable X must be assigned the value v, we say that: *X must be v because the others are not.* For example: *Drums player must be Slyme because the others are not.*

When NCC concludes that there is an "equal to" constraint between two variables X and Y, we instantiate the template: *X must be Y because it can't be any of the others.* for example: Bass player must be 2 Yuk composer because Bass player can't be any of the others. If only one value v is left in the domains after NCC updates them, then we say that: *They must be v because that's all that's left.* For example: They must be Muke because that's all that's left. If the "equal to" constraint already exists and only one value is left in one of the two variables, then NCC acts more like AC and we say that *X must be v because X is Y.*

In the case of GNCC: If a variable X equals Y or Z, causing its domain to be set to {v1 ... vk}, we invoke the template: *X must be either v1 ... or vk because it is either Y or Z.* In the Flumm Four puzzle, we have: *Love composer must be either Furble or Icky or Muke because Love composer is either Guitar player or Keyboard player.*

Refinements

NCC and GNCC did not initially add new equality constraints to the problems, but were only used to reduce domain sizes. Adding the new equality constraints to the problem can shorten explanations.

The order in which the inference methods are applied results in different explanations. For example, changing the order for the Zebra problem from AC, CC, GNCC to GNCC, CC, AC results in approximately a 10 per cent improvement in the size of the explanation.

More improvements are possible to get a shorter and better explanation. One improvement would be to use AC to get better explanations when some specific constraints such as: "=, less by 1, half of, ..." are used. This would be done by reducing the domain size of the two variables used by AC to the minimum of both. Fuller use of Transitivity may improve the explanations. Sentences might be profitably combined.

In setting up a logic puzzle problem, we need to represent fully the information given in the booklet as a constraint network. Some clues may be implicitly mentioned but not explicitly stated in a logic puzzle. A fuller explanation would take into account the work involved in representing these implicit clues.

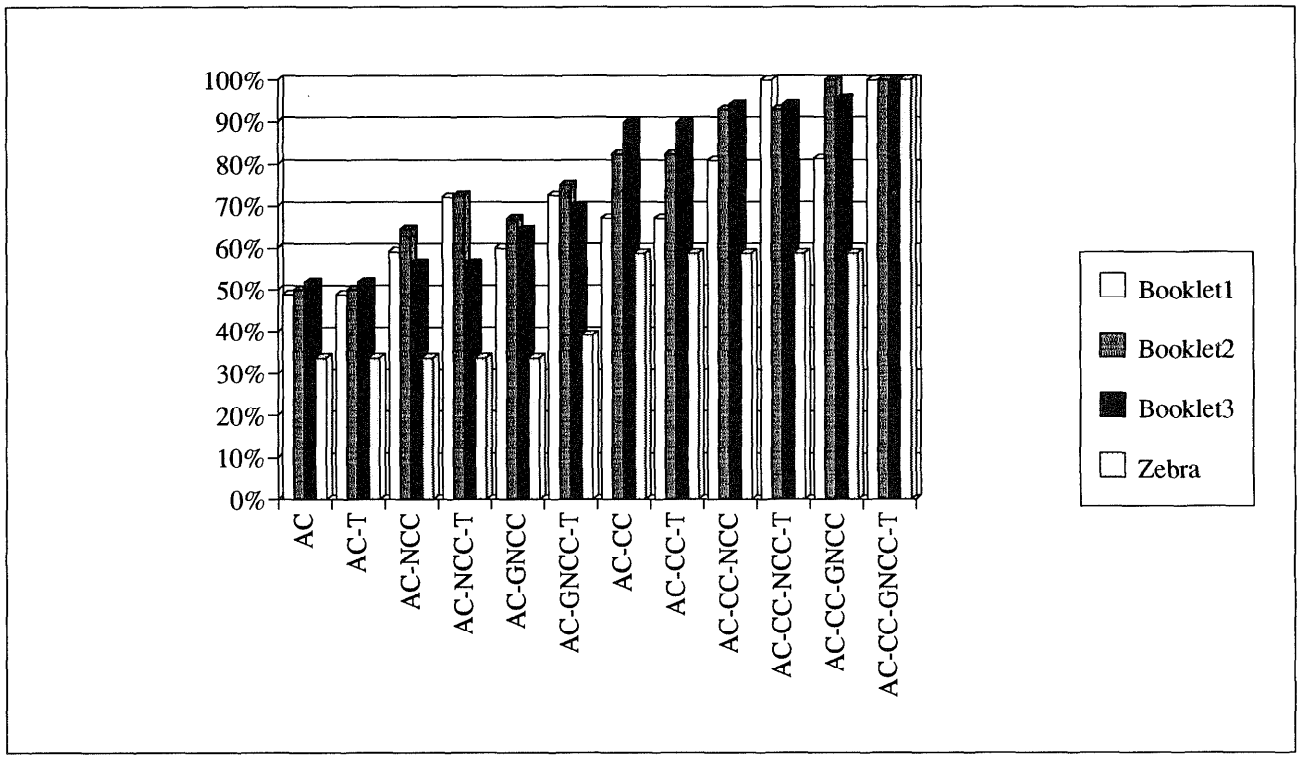

Figure 8. Average progress to solution.

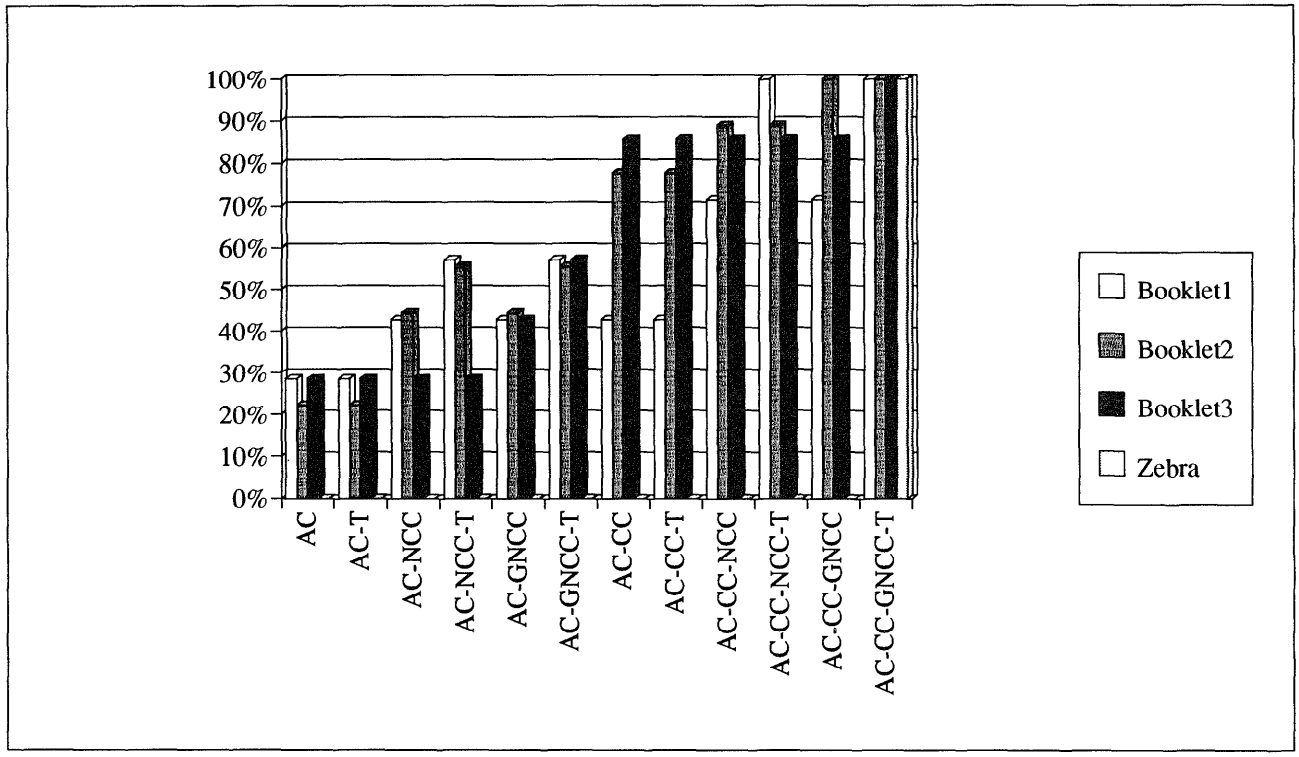

Figure 9. Problems solved.

Evaluation

Solvability

We used three Dell logic puzzle booklets purchased at newstands. These booklets rank the puzzles with one to five stars. We eventually tested our program on all the one and two star puzzles in each booklet that were naturally representable as binary CSPs. We had 7 problems from booklet 1, 9 from booklet 2, and 7 from booklet 3 (each booklet had a total of 10 puzzles with either one or two stars). We also tested on the classic Zebra puzzle, a fairly involved logic puzzle.

It is important to note that the AC, CC and NCC methods were chosen *before* looking at the second two booklets (indeed after just working with Flumm Four), and the other two methods, GNCC and Transitivity, were added *before* looking at booklet 3. Thus we were able to test the robustness of the methods on new puzzles for which they could not have been specifically biased.

The results are shown in Figures 8 and 9, which average results over each booklet (and report the zebra problem separately). The results are plotted for different combinations of methods (T indicates Transitivity). Figure 8 shows progress towards solution by measuring the percentage of deleted values; 0 per cent progress on an individual problem would mean no values deleted, 100 per cent progress would mean all values but one in each domain deleted. Figure 9 shows what percentage of problems were completely solved, e.g 33 and a third per cent for booklet 2 would mean 3 out of 9 problems completely solved.

We make the following observations:

- Even AC alone was able to solve some problems completely.
- The AC/CC/NCC combination developed for a single one star puzzle (Flumm Four) was able to solve a surprising number of puzzles, 80 per cent of the total; these methods alone deleted 80 per cent or more of the inconsistent values in each booklet.
- If a combination of methods fails to completely solve a problem, the progress made in the form of domain reductions might be taken advantage of by subsequent search.
- *A surprisingly small number of methods were sufficient to completely solve all the puzzles, without the need for any subsequent search.*
- The methods proved surprisingly robust and extensible; the methods developed to solve the puzzles in two booklets (and the Zebra puzzle) succeeded in solving all the puzzles from the third booklet, without any addition or alteration.
- Either one of the advanced methods alone, GNCC and Transitivity, could solve all the problems in the first two booklets; however, neither advanced method alone even made any progress toward solution of the zebra puzzle, while both together solved it.

Explanation

The most important thing to note here is that we were able to produce explanations qualitatively similar to the logic puzzle booklets, whereas a trace of the usual search solution methods would produce results very different in character, and very unlikely to be of interest to human puzzle solvers.

One possible measure of explanation quality is the size of the explanation. The booklets are more concise than our program. Comparing with the second booklet, for example, our explanations were roughly twice as long on average. We indicated above some possible improvements in this regard. We found that "redundant" inference methods can shorten explanations. For example, on one problem where AC alone was sufficient for solvability, adding CC, GNCC and Transitivity (placing AC last) cut the explanation size in half.

Complexity

AC is well known to be of quadratic complexity. Let c be the number of cliques in the CSP graph. Each clique is composed of s variables. This means also, according to the logic puzzle structure, that s is the maximum domain size of each variable. Let n be the total number of variables in the CSP graph. This means that: $n = c * s$.

Now, let us compute the complexity of CC (see steps in Figure 5). The complexity of step 2 is O(n). Step 3, in the worst case, will go through all the values and is O(s). Step 5 involves looking at all other variables in the same clique ($s-1$ variables) and is O(s). Therefore, the worst case complexity time for the CC algorithm is $O(ns^2)$.

The worst case complexity for NCC and GNCC is the same; we will examine the steps in Figure 7. Step 2 involves a complexity of O(n). For step 3, we need to go through all the other cliques ($c-1$). The complexity is then O(c). Step 6 involves looking at all the variables in the neighboring clique; thus the complexity is O(s). In step 7, we can determine if the two variables are different with worst case O(s) effort (assuming the domains are ordered). Therefore, the worst case complexity time for the NCC and GNCC algorithms is $O(ncs^2) = O(n^2 s)$.

The Transitivity method has a worst case complexity of $O(n^3)$. But, this complexity depends upon the number of equality constraints. If there are few, the complexity will be smaller.

Thus our methods are all of low order complexity, even though some of them can achieve partial consistency of a high order. E.g. CC achieves partial s-consistency. Full s-consistency would involve an algorithm with a complexity exponential to the power s

(Cooper 89). Here we *target* our effort, to take advantage of problem structure.

The CPU time involved in our tests was very small, for example, an average of 0.08 seconds for the puzzles in the second booklet. Even for the zebra problem, the CPU time was on the order of half a second. Thus our inference methods are certainly efficient enough for these problems. The simplicity of the problems was one reason we did not focus on efficiency measurements in these experiments.

Conclusion

There are a number of exciting directions for further work, among them:

- The inference-based approach should be applied to other domains. Specifically, scheduling problems tend to involve "clusters" or even cliques. Some of the methods developed for logic puzzles may carry over. Where the clusters are not complete cliques the challenge will be to develop methods that can still take some advantage of the looser structure.
- We can look to build an object oriented hierarchy of inference methods that cover a range of problem structures.
- The work of (Subramanian & Freuder 93) and (Freuder & Wallace 95) on learning rules or constraints based on problem solving experience provides some indication that the sort of specialized inferences we developed here by studying logic puzzles might someday by developed automatically by programs that compile specialized problem solvers.
- The quality of explanations can be improved; indeed the meaning of "quality" needs to be explored in this context.
- Explanations should be extended to deal with "what if" situations and to help debug incorrect knowledge bases (Huard & Freuder 93) and relax overconstrained problems. We have had some initial success with using arc consistency inference to answer questions of the form: "why can't variable X have value y?".
- The potential computational advantage of specialized inference should be further evaluated.

The work so far:

- Presents a new approach to CSP solving, emphasizing the primacy of inference rather than search.
- Addresses the complexity of general constraint satisfaction methods by taking advantage of the structure of specialized problem domains.
- Demonstrates the power and extensibility of these methods on the classic logic puzzle testbed.
- Uses inference to support explanation in a CSP context.

Acknowledgements

This material is based on work supported by the National Science Foundation under Grant No. IRI-9207633 and Grant No. IRI-9504316. The first author is a Fulbright grantee sponsored by the Moroccan-American Commission for Education and Cultural Exchange. This work profited from discussion with Mihaiela C. Sabin. We would like to thank Nancy Schuster and Dell Magazines for permission to reproduce material from *Dell Logic Puzzles.*

References

Baptiste, P. and Le Pape, C. 1995. A theoretical and experimental comparison of constraint propagation schemes for disjunctive scheduling. *Proceedings IJCAI-95*, 600–606.

Beldiceanu, N. and Contejean, E. 1994. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12), 97–123.

Cooper, M. 1989. An optimal k-consistency algorithm. *Artificial Intelligence*, 41, 89–95.

De Kleer, J. 1989. A comparison of ATMS and CSP techniques. *Proceedings IJCAI-89*, 290–296.

Dechter, R. 1986 Learning while searching in constraint-satisfaction problems. *Proceedings AAAI-86.*

Dechter, R. and van Beek, P. 1995. Local and global relational consistency. *Principles and Practice of Constraint Programming - CP '95*, Montanari and Rossi, eds. LNCS 976, Springer, 240–257.

Freuder, E. 1978. Synthesizing constraint expressions. *Communications of the ACM*, 21, 958–966.

Freuder, E. and Subramanian, S. 1993. Compiling rules from constraint-based experience. *International Journal of Expert Systems: Research and Applications*, 401–418.

Freuder, E. and Wallace, R. 1995. Generalizing inconsistency learning for constraint satisfaction. *Proceedings IJCAI-95*, 563–569.

Huard, S. and Freuder, E. 1993. A debugging assistant for incompletely specified constraint network knowledge bases. *International Journal of Expert Systems: Research and Applications*, 419–446.

Mackworth, A. 1992. The logic of constraint satisfaction. *Artificial Intelligence*, 58, 3–20.

Regin, J.-C. 1994. A filtering algorithm for constraints of difference in CSPs. *Proceedings AAAI-94*, 362–367.

Smith, B. 1992. How to solve the zebra problem, or path consistency the easy way. *Proceedings ECAI-92*, 36–37.