Constraint-based Program Plan Recognition in Legacy Code

Steven G. Woods and Qiang Yang*
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L–3G1

Keywords: AI and reverse engineering, program analysis and understanding, legacy systems.
Paper type: new research in the area of AI and SE.

1 Introduction

It is well-known that large legacy code sources present many challenges for software engineering. As a result of different groups of people making mostly local changes to these sources, they frequently contain redundant functionality, and knowledge about code functionality becomes dispersed, out of date, or non-existent. As a consequence, large source programs are often difficult to maintain, extend and debug; they may contain obsolete, unused pieces of code; their documentation may be outdated or even non-existent. All these factors contribute to the difficulties in understanding legacy code experienced by software maintainers.

We are interested in applying AI knowledge-based techniques to program code understanding. In the past, AI researchers have approached the understanding problem indirectly from the perspective of plan recognition [2, 1, 9] by constructing a mapping between hierarchies of actions and goals to the observed events. These plan recognition programs have been applied mostly to *toy domains* (such as the cooking domain), involving small knowledge bases and a small amount of search.

More recently, researchers have adopted a more direct approach to program understanding. In this direction, an explicit library of programming plan templates and concepts is constructed, and various top-down and bottom-up search strategies are utilized to implement the mapping process. Notable examples are Quilici[7], Kozaczynski and Ning[3], Rich and Waters[8] and Wills[10]. To some extent, all are aimed at improving the effectiveness of the mapping process through heuristic knowledge.

Our work stems directly from the aforementioned approaches. Our observation is that different pieces of the source code are related to each other by various constraints, and the program-plan recognition problem can be viewed as that of choosing among alternative plan-based explanations of the source code which satisfy these constraints. Under this view, the program plan recognition problem becomes a *Constraint Satisfaction Problem (CSP).* For a given legacy source code, the program components (explained later) are variables in the CSP. The domain values are the

*This work is supported by grants from the Natural Science and Engineering Council of Canada. E-mail: {sgwoods, qyang}@logos.uwaterloo.ca. Phone: 519-888-4567 (x3592, x4716). Fax: 519-885-1201

known program plans that may *explain* each component. The CSP constraints are either *knowledge constraints* which describe how program plans may fit together to form larger plans, or *structural constraints* which describe how program components are structurally related.

There are two advantages in our constraint-based approach. The first is its generality; most of the previous recognition methods can now be unified under the constraint-based view. Another advantage is heuristic adequacy; by casting program understanding as a CSP, the previously known constraint propagation and search algorithms could be easily adapted. We may now perform a systematic study of different search heuristics, including both top-down and bottom-up as well as many other hybrids, in order to discover their applicability to a particular source code.

Below, we present our approach by considering a C to C++ mapping example, and show how a constraint-based approach can help solve the understanding problem. We also show empirical examples demonstrating scalability and discuss the usability of the approach.

2 Program Understanding as a CSP

2.1 The Program Understanding Problem

Consider the C program outlined on the left hand side of Figure 1 (All figures are located at the end of the article). This example program contains declarations, initializations and an embedded print loop for *each* of three strings. As an illustration, strings are treated as a primitive data type by the programmer, with no shared functionality for printing.

To understand this program, one might exploit a library of program plans (see Figure 2) which represents previously compiled knowledge about program composition within a particular domain. Once this is accomplished one could translate the redundant source code to one with a single inclusion of the ADT, as shown in the C++ code on the right hand side of the figure.

The above understanding process might be executed in two steps. First, one identifies all instances of a particular abstract program plan in a source code. We refer to this problem as the *MAP-CSP* problem. Second, one relates some set of identified plan blocks (or program slices) to conform to the hierarchical structure in a given program-plan knowledge base. This is called *PU-CSP* problem. Below, we discuss them separately.

2.2 A CSP representation

A Constraint Satisfaction Problem[1] typically consists of three major components: A set of variables, a finite domain value set for each variable, and a set of constraints amongst the variables which restrict domain value assignments. A solution to a CSP is a set of domain value-to-variable assignments such that all inter-variable constraints are satisfied. For example, the map-coloring problem may be interpreted as a CSP in which every country on the map is a variable, every variable ranges over all available colors, and the problem definition that no two adjoining countries may be colored the same is represented as a constraint between each neighboring pair of countries. Depending on the problem domain, one may be interested in *any one* solution (any satisfying coloring), any solution satisfying some "goodness" measure (a coloring that meets some tasteful color combinations), or even *all solutions* (all possible colorings).

[1] See [4] for an accessible and detailed treatment of Constraint Satisfaction Problems.

As stated earlier, we formulate the program understanding problem in terms of two related CSPs, PU-CSP and MAP-CSP, which are explained in turn in the following sections.

2.3 Program plan identification as a CSP: MAP-CSP

A program template matching problem can be stated as follows: given a plan template with a number of elements and constraints among the elements, find all instances of the template in a source code. As an example, consider finding all instances of an abstract data type in a C program. Figure 3 is a **String** ADT plan template taken from a plan library. The ADT is described in terms of 5 features describing various key components of a string class. In addition, there are constraints among the different parts as well, such as the one that requires one component to go before another.

We could model this problem as a CSP. For the given plan template (or ADT), each feature is a variable in our MAP-CSP. The *domain range* consists of all possible source program statements. Variables here can have attributes such as (**print**,**for**) that may be seen as *constraints* on allowable assignment of program statements (values) to template features (variables). Other *constraints* are on the sharing of information among variables, and on the order in which template features or variable are expected to appear in legacy source. A solution to the MAP-CSP consists of the set of all assignments of plan template features by source code statements, where each assignment must satisfy all constraints. The solution to a MAP-CSP provides a mapping that *explains* the matched source statements as parts of an instance of the abstract program plan or ADT.

2.4 Program Understanding as CSP: PU-CSP

After the MAP-CSP problem is solved for a subset of abstract plans in the program plan library, each source code block is associated with a number of possible explanations by abstract plans. The next task is to select one explanation for each source code block such that the global explanation satisfies both the constraints imposed by the source code itself, and by the AND/OR relations inherent in the program library.

This task is formed as another CSP called PU-CSP. Suppose that an initial decomposition of the source code has been done by human or another system. Then each block of source code corresponds to a *variable* in the PU-CSP. The *Variable domain ranges* correspond to all possible explanations of an individual source code block.

As an example, consider the legacy code program statements of Figure 1 as the blocks. We take each block as a PU-CSP variable which ranges over all possible program plans of corresponding statement type, such initialization, print etc, in the plan library of Figure 2.

In a PU-CSP, the constraints among variables are of two types. The first, *Structural* constraints, are determined from the legacy code. They include such things as scope or called/calling relations, precedence relations, or shared information relations between component blocks. For instance, in Figure 1, the **print** statements appear within the scope of **for** statements, **declarations** precede their initial **assignment**, and print statements act upon array positions indexed by corresponding **for** statement indexes. The second type is *Knowledge* constraints, which are independent of the legacy code. The way program plans are restricted in their inter-relationships by the AND/OR structure given in the plan library yield knowledge constraints. AND constraints are for composing program plans into higher level plans, and OR's are for specializing an abstract plan in one of several ways. Assigning one program plan as an explanation of a particular PU-CSP variable thus constrains consistent assignments of other component variables.

A solution to the PU-CSP is an assignment to each variable by one program plan component in the plan library, such that no structural constraint from the source code, or knowledge constraint from the plan library is violated.

3 Search Algorithms and Empirical Results

3.1 Solving CSPs

We have explained program understanding as a form of CSP. Solving a CSP requires *constraint propagation* which attempts to systematically reduce variable domain ranges through the application of constraints between variables, or through heuristic backtracking algorithms which repeatedly select variables and instantiations and reduce remaining variable domains according to existing constraints. A thorough examination of these techniques is presented in [5, 4, 6]. Below we present a high-level description of the CSP search algorithm.

Generic CSP Search
V: variables in a CSP, $Dom(X)$: the domain values of X.

1. [**Initialization**] **for** each variable $X_i \in V$, find the set of domain values for X_i;
2. [**Initial Constraint Propagation**] Reduce $Dom(X)$ by constraint propagation.
3. Solution = NULL
4. [**Variable Selection**] Select and remove a variable X from V
5. [**Value Selection**] Select and remove a value of X from $Dom(X)$.
   The value must be consistent with all assignments in Solution.
6. [**In-search Propagation**] Apply a subset of constraints to V.
7. [**Backtrack Point Selection**] Backtrack if any $Dom(X)$ in V becomes empty.
8. [**Solution Evaluation**] If V is empty, exit with Solution (if all-solution, continue);

else, goto Step 4.

During a backtracking search, a variable is instantiated one at a time. In the case of MAP-CSP, this corresponds to identifying an additional instance of program plan fragment as part of the abstract program plan. In the case of PU-CSP, this instantiation is interpreted as extending the current understanding of a legacy program one step further. In this light, Quilici's work[7] can be seen as incrementally generating constraints from the hierarchical structure of program plans, and imposing these constraints on the partial solution to the CSP.

3.2 Empirical Results

We have completed our implementation of the solution to MAP-CSP in Common Lisp on a SPARC-server 470 workstation. Our system is applied to a recognition problem similar to the one in Figure 3, where there are five variables and a number of constraints on the plan components. In addition, we require that the **begin** block corresponding to the **for** statement be within a fixed number (15 for our experiment) of program lines, which represents a domain heuristic on the locality of program components. A test case is produced by instantiating 3 instances of the program template in a sample source code, and by adding some variable amount of additional program statements as "noise" around the template instances. An abstract plan thus generated is then mapped to legacy sources ranging in size from 50 source lines to 1000 source lines, in 50 source line increments. Each increment was tested with 10 different randomly permuted source code.

Figure 4 shows a straightforward application of the CSP search algorithm without using any heuristic knowledge. This algorithm exhibited very unstable performance for different problems of the same size. In addition, it is easily overwhelmed by the increasing complexity as code size increases.

To address this **scalability problem**, we then used a combination of three heuristics to guide the backtracking process. The first heuristic is known as *arc-consistency*[4], which compares every pair of CSP variable domains and discards any domain value which cannot be part of the final solution. The second heuristic is called *forward-checking*[4], which uses every new variable instantiation to prune future inconsistent values. The last one is called *dynamic rearrangement*[4], which reorders the remaining variables based on the size of their domains. The result of applying this algorithm to the same problems is shown in Figure 5. It is clear from this graph that the heuristic algorithm is not only stable in performance, but also scales up easily to 500 lines of code (well within 2 minutes of CPU time). The heuristic improvement is contrasted with standard backtracking in Figure 6.

4 Discussion

We summarize some of the advantages of our approach below.

Scalability Our empirical results demonstrated that the MAP-CSP problem can be scaled up for legacy code of useful sizes. This efficiency gain is achieved by viewing the recognition problem as constraint satisfaction, and applying known constraint satisfaction algorithms. In our experiment, we haven't utilized the full range of constraints inherent in a program source code, such as those derived from program parsing, a technique employed by Kozaczynski & Ning[3] and Wills[10]. More extensive consideration is given to the specific use of these constraints in [11]. We expect the empirical results to improve further with use of these constraints.

Usability We envision our system as one part of a programmer's assistant toolset. For the MAP-CSP problem, a programmer could use the system to identify abstract program plans in legacy programs up to around 500 lines of code in almost real-time, and can apply the system in batch-mode to much larger programs.

We are currently engaging in cooperation with a main telecommunications provider to investigate the applicability of this approach to extremely large source code in the telephony domain. Achieving partial automatic recognition of even 5% of the code would greatly benefit software maintainers.

We are currently implementing the search algorithm for PU-CSP. We expect to see similar effective results from constraining the search with hierarchical plan knowledge, particularly when this algorithm is fully integrated with the MAP-CSP solutions.

Acknowledgments

We thank Grant Weddell for many helpful discussions. This research has been carried out with the support of the National Science and Engineering Research Council of Canada.

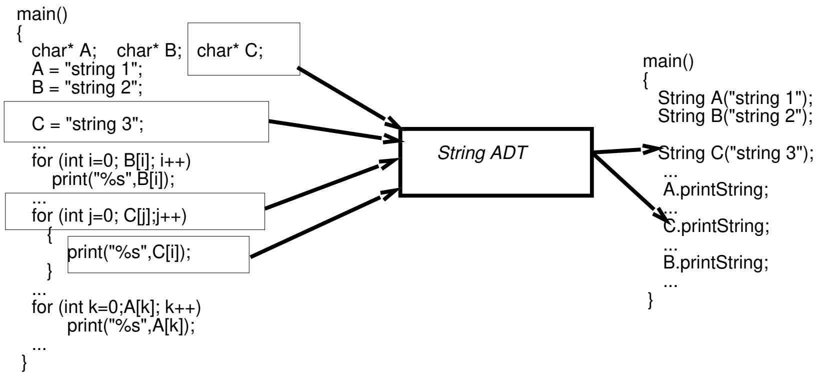

Figure 1: Desired C++ replacement of C code.

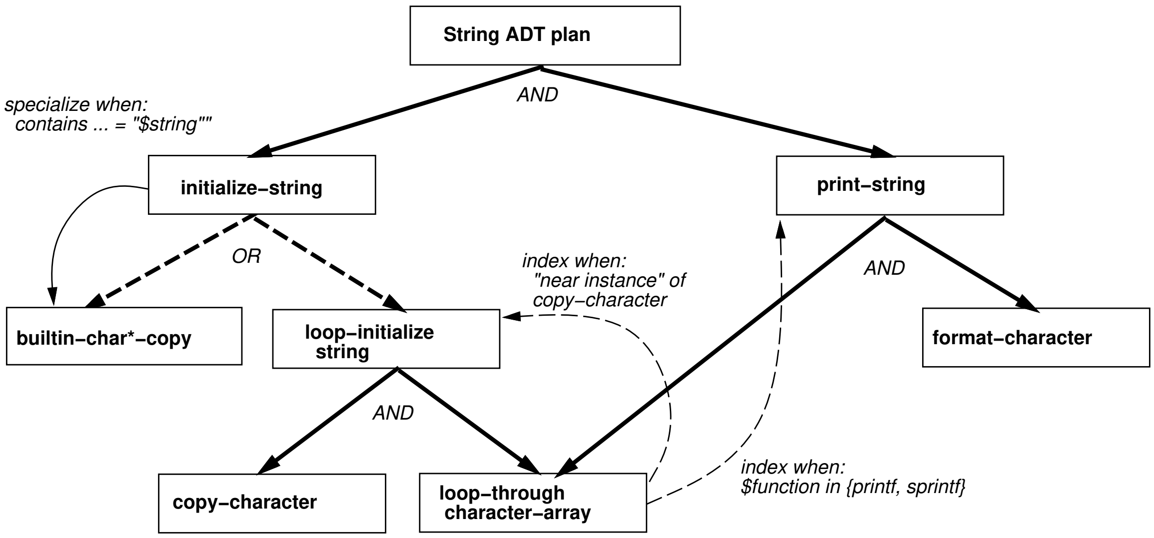

Figure 2: **String** ADT within a hierarchical program plan library.

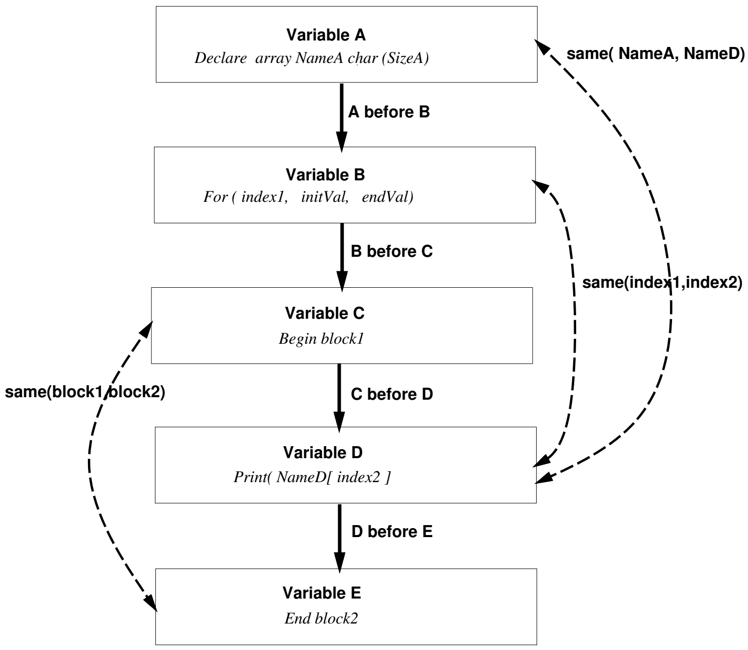

Figure 3: The **String** ADT in MAP-CSP.

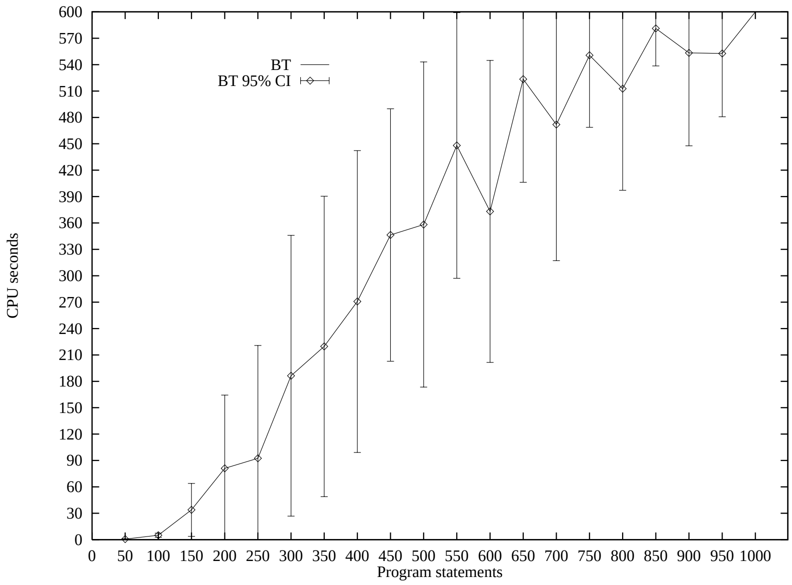

Figure 4: Standard BackTrack (95% confidence interval).

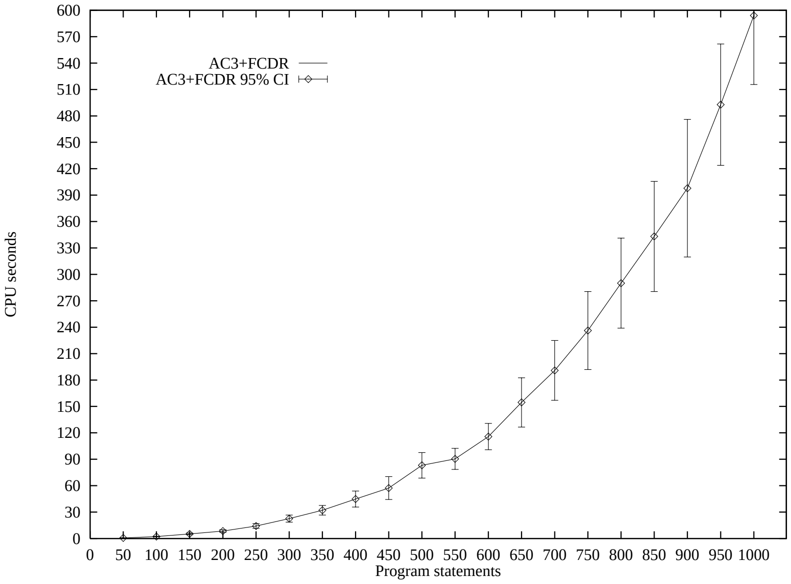

Figure 5: AC-3 with FC, DR (95% confidence interval).

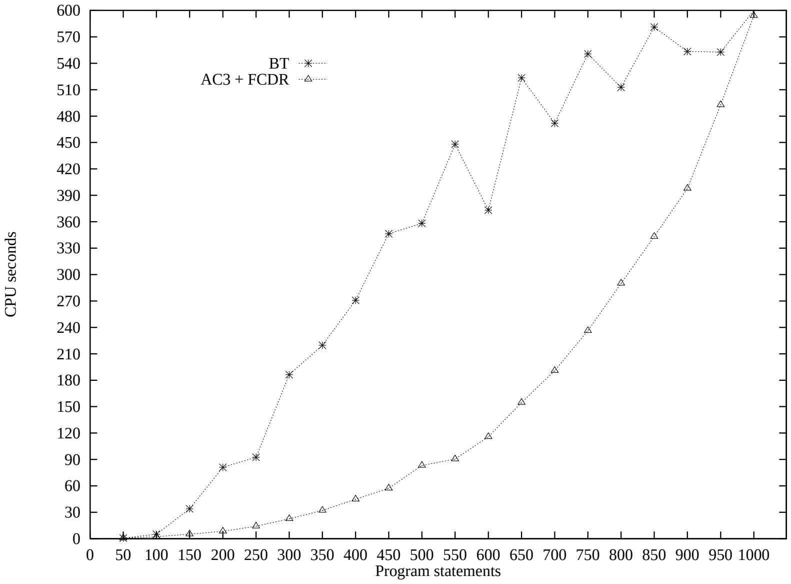

Figure 6: Standard BT vs AC-3 with FC, DR (cpu seconds).

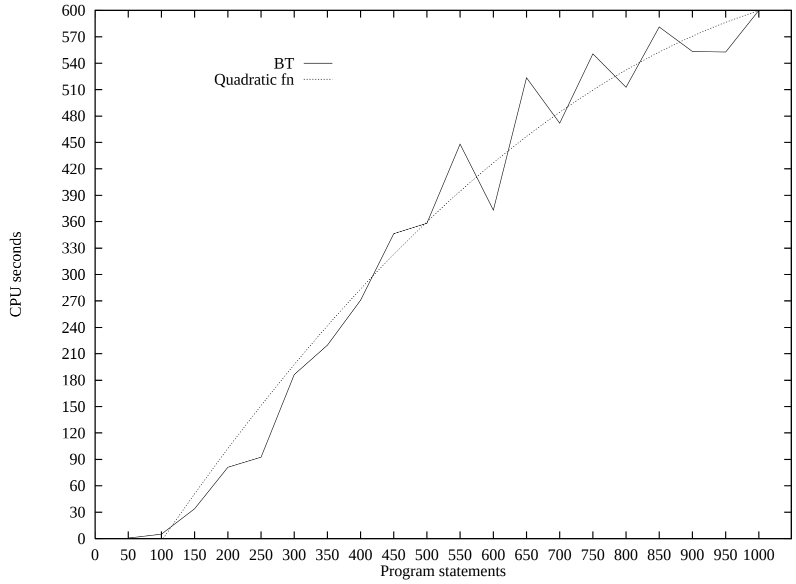

Figure 7: Quadratic fit to Standard BT Data.

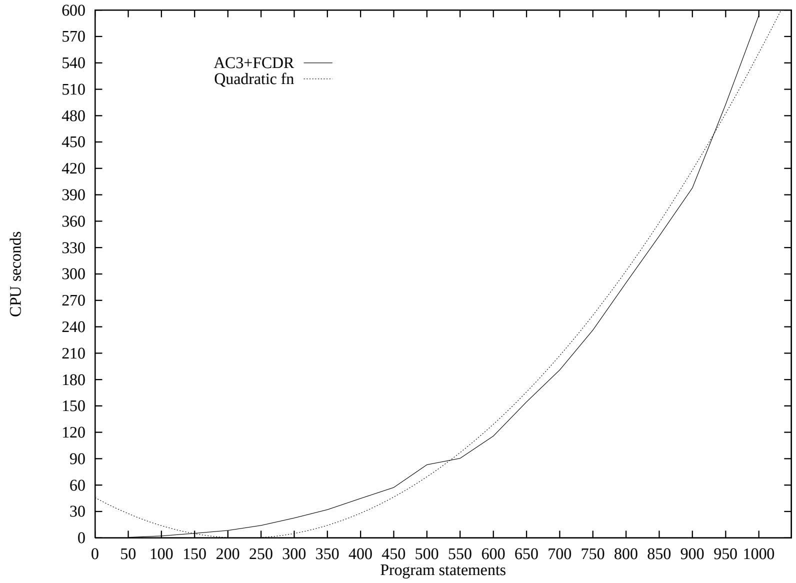

Figure 8: Quadratic fit to AC-3 with FC, DR.

References

[1] Sandra Carberry. Incorporating default inferences into plan recognition. *Proceedings of the 8th AAAI*, 1:471–478, 1990.

[2] Henry Kautz and James Allen. Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 32–37, Philadelphia, Pennsylvania, 1986.

[3] Wojtek Kozaczynski and Jim Q. Ning. Automated program understanding by concept recognition. *Automated Software Engineering*, 1:61–78, 1994.

[4] Vipin Kumar. Algorithms for constraint-satisfaction problems. *AI Magazine*, pages 32–44, Spring 1992.

[5] Bernard A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[6] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[7] Alex Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, May 1994.

[8] C. Rich and R.C. Waters. *The programmer's apprentice*. Addison-Wesley, Reading, Mass., 1990.

[9] Peter van Beek, Robin Cohen, and Ken Schmidt. From plan critiquing to clarification dialogue for cooperative response generation. advance version of journal paper, 1994.

[10] L. M. Wills. Automated program recognition by Graph Parsing. AI Laborary Technical Report 1358, MIT, 1992.

[11] Steven Woods. A constraint-based approach to program plan recognition in software reverse engineering. Ph.D. Thesis Proposal, University of Waterloo, February 1995.