

Applying Plan Recognition Algorithms to Program Understanding

Alex Quilici *
University of Hawaii
U.S.A.

Qiang Yang †
Simon Fraser University
Canada

Steven Woods ‡
University of Waterloo
Canada

Abstract

Program understanding is often viewed as the task of extracting plans and design goals from program source. As such, it is natural to try to apply standard AI plan recognition techniques to the program understanding problem. Yet program understanding researchers have quietly, but consistently, avoided the use of these plan recognition algorithms.

This paper shows that treating program understanding as plan recognition is too simplistic, and that traditional AI search algorithms for plan recognition are not suitable, as is, for program understanding. In particular, we show (1) that the program understanding task differs significantly from the typical general plan recognition task along several key dimensions, (2) that the program understanding task has particular properties that make it particularly amenable to constraint satisfaction techniques, and (3) that augmenting AI plan recognition algorithms with these techniques can lead to effective solutions for the program understanding problem.

*Department of Electrical Engineering, Holmes Building 483, Honolulu, Hawaii 96822. alex@wiliki.eng.hawaii.edu

†School of Computing Science, Burnaby, British Columbia V5A 1S6. qyang@cs.sfu.ca

‡Department of Computer Science, Logic Programming and Artificial Intelligence Group (LPAIG), Waterloo, ON N2L 3G1. sgwoods@logos.uwaterloo.ca

1 Introduction

Program understanding is often described as the process of recognizing program plans in source code [20, 14, 6, 19, 8]. In particular, most program understanding algorithms explicitly use a library of programming plans, along with various heuristic strategies to locate instances of these plans in the code. Because the program understanding task is so closely related to plan recognition, one would expect to see researchers directly apply well known plan recognition algorithms to the task [3, 4]. However, they have not, and have instead chosen to develop their own special purpose algorithms.

This paper is an attempt to understand and explain why. We examine the relationship between plan recognition and program understanding and study the assumptions underlying each task. As part of this analysis, we present an approach to program understanding in the spirit of typical plan recognition algorithms, and illustrate the inadequacy of this approach. We then demonstrate how a constraint satisfaction-based approach to plan recognition is particularly well suited to program understanding. Finally, we show how one existing AI plan recognition algorithm can be modified to take this into account.

Our motivation for this work is to help move program understanding from being an isolated subproblem of AI into the mainstream of AI research. This will allow results in AI involving plan recognition and constraint satisfaction to be quickly integrated into our program understanding algorithms, and will allow work in program understanding to influence the general AI community and perhaps benefit other AI application areas.

2 Plan Recognition

Plan recognition is the task of determining the *best*¹ unified *context* which causally explains a set of perceived events as they are observed. A context is essentially a hierarchical set of plans and goals that accounts for the observed actions. This process generally assumes a specific body of knowledge which describes and limits the types and combinations of events that may be expected to occur. This knowledge body is frequently represented as a specialization and decomposition structure of events and actions.

2.1 The AI Approach to Plan Recognition

Kautz and Allen [3, 4] formalized an approach to plan recognition that has served as a primary building block for many subsequent plan recognition methodologies, including [17, 18]. They provide a general algorithm by which “a set of observed or described actions is explained by constructing a plan that contains them”. In particular, as actions are observed, hypothetical explanations are proposed for them. This process involves uncertainty, as at any time there are a number of candidate explanations for an action, but only a portion of the actions within those candidates may have been observed. The process of arbitrating this uncertain selection process is the primary focus of the work of Kautz and Allen, and of plan recognition systems in general.

¹*Best* is a highly subjective term which changes definition depending on the intent of the particular plan recognition application.

Kautz and Allen’s approach² is based upon ordinary deductive inference. The *rules* for deduction are rooted in the exhaustive body of knowledge about actions in a particular domain encoded in the form of an **action hierarchy**. This action hierarchy describes all ways an action may be performed or used as a step in a more complex action. It does so by capturing two types of information (see Figure 1). The first is specialization relationships between actions, such as that Print-Value is a particular kind of Print-Temp-Result, which, in turn, is an Print-Sum-Plan, and so on. By having different specializations, this representation captures the notion that there are multiple ways to perform a given task. It also captures decomposition relationships, such as that Sum-Values contains a Sum-P-and-Q and a Return-Result, as well as constraints between these actions.

[NOTE: A picture containing the above-described Action Hierarchy goes here.]

[NOTE: We could use the same average-sum example for this purpose. How is the above?]

Their approach starts by turning this plan hierarchy into a set of axioms that captures the structure of the hierarchy and its underlying assumptions. The actual recognition process then undertakes a specialized forward chaining reasoning process over these axioms. In particular, as it observes each action, it chains up the action hierarchy until a top-level plan is reached, essentially using the action hierarchy as *control graph* which directs its inference process and limits its disjunctive reasoning. This step results in a set of possible paths from the observed action to top-level actions. These constitute an initial set of possible explanations (in terms of higher-level plans) of the action.

After more than one observation arrives, the system will have derived two or more sets of paths to top-level action instances (that is, it will have found a set of paths from each observed action, through the action hierarchy, to top-level actions). It then applies a “simplicity heuristic”, to unify the disjoint explanations. This heuristic is to prefer as few high-level actions as possible or, in other words, to reduce the explanation to the set of actions and the minimal set of higher-level plans that “cover” all of them. When this heuristic is applied, the result is a set of restrictive assertions about the functions of each observed actions. If this causes an inconsistency, the system backtracks up the explanation path to where the simplicity heuristic incorrectly merged the explanation paths.

For example, in the example shown in Figure 1, after a Print and a Sum action are observed, Kautz’s algorithm will recognize a Print-Sum-Plan. Furthermore, by applying the simplicity heuristic this hypothesis will be preferred over the set {Print-Sum-Plan, Print-Avg-Plan}.

[NOTE: Need to add a paragraph here that refers to a specific recognition example from the program understanding domain. It’s difficult to understand this algorithm without the example.]

[NOTE: How’s the above paragraph?]

This “simplicity” heuristic is key: by minimizing the number of hypotheses which account for all observations and accepting this event covering set as the current plan, we describe precisely how to recognize a plan from observation.

²The algorithms for this approach are detailed in [3].

3 Applying AI Plan Recognition To Program Understanding

The Kautz and Allen approach to plan recognition is elegant and the basis for much subsequent work in plan recognition. Given that program understanding appears to be a form of plan recognition, it's worth considering whether this approach is applicable to program understanding.

One key difference between plan recognition and program understanding is that plan recognition assumes *Open Perception* and program understanding assumes *Closed Perception*. That is, at any point in time, the plan recognition algorithm has an incomplete set of observed actions and, as a result, the plan recognizer is making a best guess as to what plan is present, and much of the work in forming this algorithm is in coming up with this best guess. In contrast, in program understanding exactly the opposite is true. The source program under consideration, together with any derived structural constraints, makes up all of the perceptual information that will ever be available. That is, it will never be the case that a program statement or part that was absent in the previously encountered functional specification will be perceived at a later time. Although the focus of program understanding may be only a sub-part of a larger program, the part in question is itself complete.

3.1 Incorrect Plan Recognition

As a consequence of this assumption and the simplicity heuristic used to deal with it, the Kautz and Allen approach can find an incorrect explanation, despite there being sufficient knowledge to eliminate it as a candidate. To illustrate, consider the following simple source code fragment:

```
c := a + b; print(c); c := c/2;
```

We can view this example as a simple series of observed actions: **Sum**, **Print**, and **Halve**. We ignore assignment and other structural constraints for this example. We wish to find plans for explaining this program fragment with response to the hierarchy below.

Consider what Kautz's approach would form as the plan explanation after encountering each of these observations:

- After the first observation of **Sum**, the expansion would be **Sum-P-And-Q** is-part-of **Sum-Values** is-part-of **Print-Sum-Plan** or (**Find-Avg-Plan** is-part-of **Print-Avg-Plan**).
- The second observation of **Print** would result in these independent explanations:
 1. **Print-Value** is-a **Print-Sum** is-part-of **Print-Sum-Plan**, or of
 2. **Print-Value** is-a **Print-Temp-Result** is-part-of **Sum-Values** is-part-of **Find-Avg** is-part-of **Print-Avg-Plan**, or of
 3. **Print-Value** is-a **Print-Temp-Result** is-part-of **Find-Avg** is-part-of **Print-Avg-Plan**.

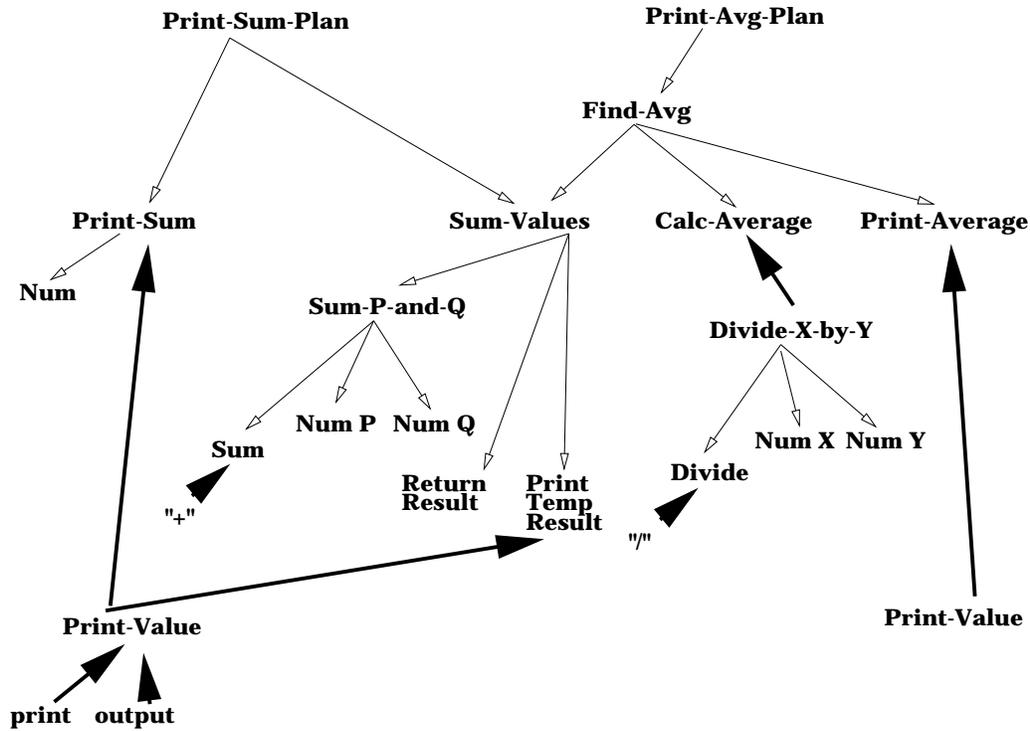


Figure 1: An Example Action Hierarchy

The reduced explanation resulting from this observation would be rooted in **Print-Average-Plan** (since it is the minimal set that covers both the **Sum** and **Values** events).

- The final observation is **Halve**, which results in an independent explanation of **Divide-X-By-Y** is-a **Calc-Average** is-part-of **Find-Avg** is-part-of **Print-Avg-Plan**. The reduced explanation from this set “explains” the **Sum** instance as an instance of **Sum-P-and-Q** in **Sum-Values**, **Print** as an instance of **Print-Temp-Result** in **Sum-Values**, and **Divide** as **Calc-Average** in **Find-Avg**.

The problem is that this explanation is actually wrong, given that we know no more actions relevant to these plans will appear in the program. Although this explanation is minimal in terms of top-level actions, it allows for the assumption that future actions will be encountered. In program understanding, it is inappropriate for the covering set to cover more actions than have already been encountered. Consequently, an exact covering set that is not necessarily minimal would give the correct explanation.

NOTE[The following is added. OK? (QY)]

To make the distinction more precise, let E be a set of observed events; in program understanding, E is the set of program statements. Let \mathcal{H} be the set of hypotheses where

the search is performed. These are the program plans at various levels of detail. The goal of both plan recognition and program understanding is simply to find one or more subsets of hypotheses H from \mathcal{H} such that H “covers” E .

The two problems differ in how H is defined. We can understand a hypothesis h as the set of events that it covers. Then Kautz’s principle can be understood as finding a smallest set of hypotheses H_K such that

$$H_K \subseteq E$$

In contrast, program understanding can be defined as finding a smallest set of hypotheses H_P such that

$$H_P = E$$

The difference is that in the latter, the cover must be exact. The hypotheses must explain all observed events, but no more. We call this latter principle the *of minimal exact-coverage principle*.

3.2 Inefficient Plan Recognition

Another problem with applying this approach directly is the *Combinatorial Problem* that occurs because any given action can be a component of a multitude of plans that can themselves be actions within a multitude of plans, and so on. The result is that number of possible explanations for a given set of observations can grow exponentially. To determine a minimal event cover of perceived actions from a plan hierarchy, it is necessary to generate potential covers and search to select the minimal one. This problem can be thought of as defining a search space of covers. Each action needs to be covered by some plan. Consider the analogical program understanding problem. Each perceived program statement needs to be covered by a program plan, in the order in which they appear in a source code. The Kautz method essentially imposes a single ordering on the domain values or program statements, resulting in a statically defined search order, hence a potentially very inefficient search tree.

This problem is especially relevant to program understanding since most programs involve thousands and thousands of actions (or more). Kautz explicitly notes this problem, and suggests that in some domains the combinatorial problem may be largely mediated through constraints on event types; however, he imagines that in realistically sized problems additional principles will be required.

4 Modifying AI Plan Recognition Approaches For Program Understanding

In some sense, program understanding has more knowledge available than is present in AI plan recognition. In particular, program understanders have the complete set of actions that are present in the program and many detailed data-flow and control-flow constraints between those actions. This allows program understanding to take a breadth-first approach to plan recognition, which avoids carrying along unconfirmed and possibly incorrect hypotheses.

One way to characterize AI plan recognition approaches is to say that they try to hypothesize complete explanation chains that cover each action, and that they use subsequent

actions to shrink the set of explanations (when the actions can be combined under some high-level action) or hypothesize additional explanations, (when they can't). At the end of a pass through all actions, the plan recognizer has a set of preferred hypothesized explanations for those actions.

In program plan recognition, we can immediately verify a portion of any hypothesized explanation chain, and that we can gradually construct explanation chains from verified pieces. In particular, given an action that is potentially part of a set of plans containing only actions (and not sub-plans), we can immediately verify whether that plan actually exists by locating the plan's other actions and verifying any constraints between them. That is, we can use each action in the AST (abstract syntax tree) as an index to the set of potential plans that might contain it, and then check whether each of those plans are present. Thus, at the end of a pass through all actions, the plan recognizer has located verified-single plan explanations for each of the actions.

One way to locate complete verified-explanation chains is to organize the plan library in layers, where the first layer is those plans that consist solely of events in a program's AST, the next layer is those plans that depend only on the events in the AST and plans in the first layer, and so on. After recognizing those plans in the initial layer, the plan recognizer runs through each of those plans and verifies whether the plans in the next layer that can contain them are actually present, creating a new set of verified recognized plans. This process is repeated until there are no newly recognized plans.

A question is how to perform this verification process. That is, given that an action suggests a set of possible plans that might explain it, how can we verify which of these plans are actually present? Given the presence of many constraints between the actions in any plan, this suggests using a constraint satisfaction approach.

A Constraint Satisfaction Problem³ typically consists of three major components: A set of variables, a finite domain value set for each variable, and a set of constraints among the variable which restrict domain value assignments. A solution of a CSP is a set of domain value to variable assignments such that all inter-variable constraints are satisfied. These mechanisms include global [5] and local search-based methods [16, 10, 22], constraint-propagation problem simplifications [11, 1, 13], hierarchical exploitation of problem structure [2], as well as hybrid combinations of these approaches.

In using a CSP for the task of verifying whether a single plan is present, the variables correspond to the actions in the plan, the domain values are the source statements (or sub-plans) with the same type within the program, and the constraints are reflexive type constraints on each variable, along with inter-variable constraints involving data and control-flow. Variables here can have attributes such as (**print,for**) that may be seen as *constraints* on allowable assignment of program statements (values) to plan features (variables). Other *constraints* are on the sharing of information among variables, and on the order in which plan components or variables are expected to appear in legacy source. Example plans using this representation can be found in [15] and [20].

A solution to the CSP consists of the set of all assignments of plan features by source code statements, where each assignment must satisfy all constraints. The solution to a CSP provides a mapping that *explains* the matched source statements as parts of an instance of

³See [7] for an accessible and detailed treatment of Constraint Satisfaction Problems.

the abstract program plan or ADT. When we start any given CSP for recognizing a particular plan, the variable represent the action that triggered this plan’s consideration is restricted to the single domain value corresponding to that action. (Thus, in some sense, each CSP is starting off partially solved.)

[NOTE: The following is added (QY)]

For the task of recognizing the overall plan of a program, we could devise another CSP representation [20]. Assuming that a program is sliced into several blocks. Each block can be represented as a variable in a CSP. The program plan components that can be used to explain the program block give rise to the values for the corresponding variable. The data flow and control flow may be seen as constraints among the variables. In this view, a solution to a CSP is an overall explanation for a program source code.

Applying ordinary plan recognition to program understanding imposes an ordering of the program statements—essentially they are considered in temporal order, top to bottom. Consider the simple case of attempting to recognize a single program plan in the CSP framework using the Kautz imposed order. A search space results in which the components of the CSP have domain ranges which include all program statements. A “cover” of the components that satisfies the existing component constraints is a potential solution. The domain ranges are ordered temporally (early program statements first), thus resulting in the generation of potential solutions with “earlier” combinations first, “later combinations” second, and an eventual generation of all combinations. Kautz’s insight that “additional principles” would be required to mediate the search can be at least partially satisfied for program understanding through the use of intelligent backtracking strategies during this process. In contrast, a constraint satisfaction algorithm relaxes the temporal ordering of domain ranges by dynamically re-arranging the domains (in the spirit of some types of forward checking algorithms), and reaping the benefits of improved search results through more effective constraint applications which reduce entire sub-parts of the search space.

5 Efficiency Implications

The previous section have shown how we can derive a new approach to program plan recognition by examining an existing AI plan recognition algorithm, studying its assumptions, determining how these assumptions differ from the program understanding problem, and then modifying this approach to take advantage of the differences. While it’s clear that our new plan recognition approach to program understanding addresses the correctness issue, and there’s clearly potential to address the efficiency issue, it’s necessary to carry out experiments to determine whether the efficiency issue is, in fact, addressed.

Using constraint satisfaction framework, we generated a set of test programs and applied the constraint satisfaction approach to checking whether the instances of a given plan are present in the source. That is, we are assuming that a plan has been suggested by an action in the program and then empirically verifying how efficient or inefficient it is to recognize *all* instances of the plans containing this action (or any similar action appearing later in the program).

The test program ranged in size from of 50 to 6000 lines in size, with 10 different programs at each size. Based on these 10 data points at each size level, we generate a 95% confidence

interval for the number of constraint checks occurring during the search. For each size, we generate programs according to an equal distribution of program statements, a “standard” distribution of statements that corresponds to what we’ve found in student programs, and a “random” distribution.

Our measure of efficiency is the number of constraint checks performed. This is reasonable, since that is where the dominant amount of work occurs in an attempt to recognize a program plan⁴.

Figure 2 shows the results of running our experiments. The plan instances we tested had an average of approximately 10-15 components and 20-25 constraints.

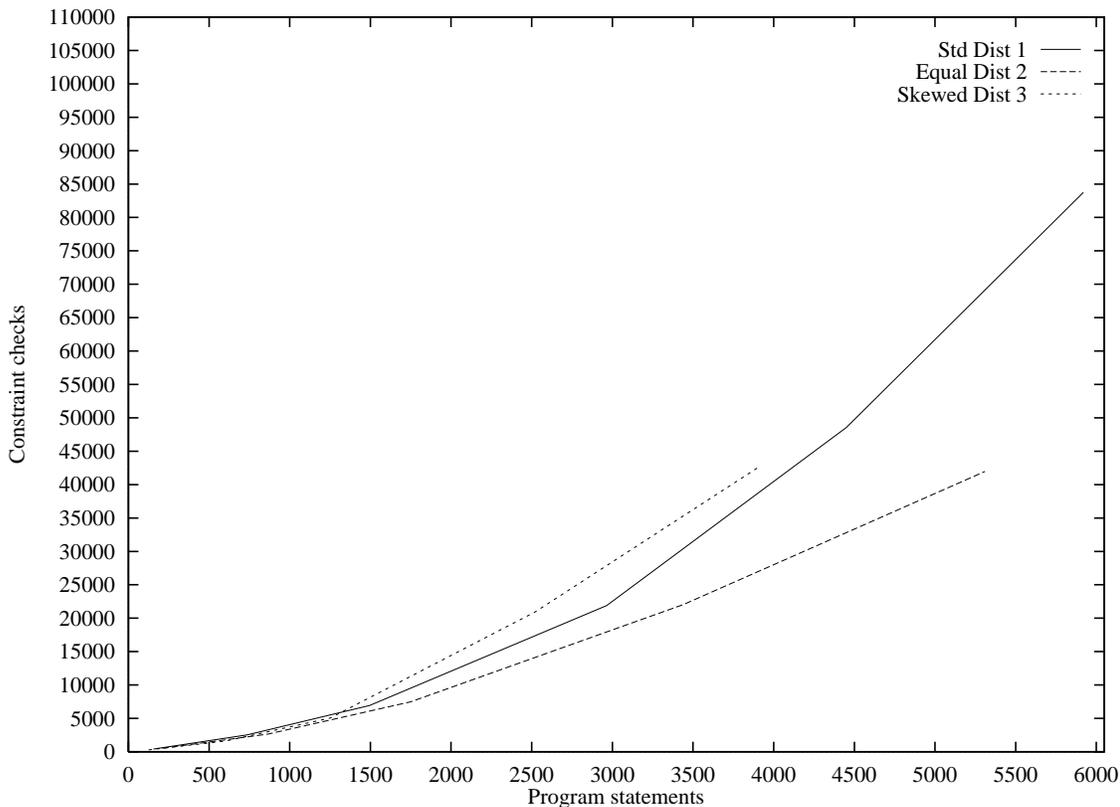


Figure 2: The results of our experiments.

Essentially, it shows a curve that using our standard distribution increases from 5000 constraint checks for 1000-line programs, up to 65000 constraint checks for 5000-line programs. While this curve appears exponential in nature, it’s heartening in several ways. Despite program understanding having been shown to be NP-hard in the worst case [21],

⁴In our experiments, implemented in Lisp on a Sparc10 workstation, the most efficient graphed strategy of Forward Checking with Dynamic Rearrangement (FCDR) is bounded by approximately 5 minutes of CPU time for instances of 6000 lines.

it's a demonstration that program plan recognition is *clearly* tractable for programs of up to 5000 lines in length with plans similar to those that we've tried. When this is combined with work done in semi-automatically modularizing Cobol programs, in which 850,000 Cobol programs were broken into modules in the 10,000-20,000 line range [12], it suggests that we are nearing the point where we can apply program plan recognition to modules of real-world legacy systems. Second, the steepness of the curve is at least partially an artifact of our particular method of representing programs. Our experiments rely primarily on the equivalent of control-flow constraints and not have data-flow constraints. Since data-flow constraints tend to be much more restrictive than control-flow constraints, they have the potential to reduce the steepness of the curve significantly and extend the size of the programs to which we can apply our plan recognition algorithm.

6 Conclusion

[Need: Expand, add more future discussion, add impact on AI of this work, add impact on PU of this work.]

We have seen that program understanding can be considered a task of understanding the plans inherent in a software code. Two main-stream methods are examined, of which the plan recognition method was found to be inadequate for the task at hand. The remaining method, constraint satisfaction was shown to be effective both theoretically and empirically. By exploiting proper domain knowledge, to discover the plan inherent in a program source code suggests that for programs in up to 5000-or so range, the recognition of program plans can be done efficiently.

Acknowledgment

Alex Quilici would like to acknowledge partial support for this work by the KBSA project, Air Force Rome Labs, under Air Force contract #F30602-93-C-0257. Steven Woods and Qiang Yang would like to acknowledge the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre (ITRC) for their support. Qiang Yang would also like to thank the following supports: MPR Teltech Ltd., Science Council of BC, BC Advanced Systems Institute, The Simon Fraser University, Ebco Industries Ltd., Epic Data International, MacDonald Dettwiler and Associates , Glenayre Electronics Inc., BC Telephone Company and Ontario Centers of Excellence (ITRC).

Thank you to Stephanie Ellis for useful comments on earlier drafts of this work.

References

- [1] R. Dechter, From local to global consistency. *Artificial Intelligence*, 55:87–107, 1992.
- [2] E. Freuder and J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

- [3] H. Kautz. *A Formal Theory of Plan Recognition*. PhD thesis, University of Rochester, Department of Computer Science, Rochester, New York, 1987.
- [4] H. Kautz and J. Allen. Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 32–37, Philadelphia, Pennsylvania, 1986.
- [5] G. Kondrak and P Van Beek. A theoretical evaluation of selected backtracking algorithms. *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pp. 541–547, 1995.
- [6] W. Kozaczynski and J. Q. Ning. Automated program understanding by concept recognition. *Automated Software Engineering*, 1:61–78, 1994.
- [7] V. Kumar. Algorithms for constraint-satisfaction problems. *AI Magazine*, pages 32–44, Spring 1992.
- [8] W.L. Johnson. *Intention Based Diagnosis of Novice Programming Errors*. Morgan Kaufman, Los Altos, CA, 1986.
- [9] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
- [10] S. Minton, M. Johnston, A. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.
- [11] Nadel, B. A., Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.
- [12] P. Newcomb and L. Markosian, Automating the Modularization of Large COBOL Programs: Application of an Enabling Technology for Reengineering. In *Proceedings of the Working Conference on Reverse Engineering*, Baltimore, MD, 222-230, 1993.
- [13] P. Prosser, P. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.
- [14] A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, May 1994.
- [15] A. Quilici and S. Woods. *Toward A Constraint-Satisfaction Framework for Program Understanding*, *Journal of Automated Software Engineering*, to appear, 1996.
- [16] R. Sosic and J. Gu. A polynomial time algorithm for the n-queens problem. *SIGART*, 1(3), 1990.
- [17] P. van Beek, R. Cohen, and K. Schmidt. From plan critiquing to clarification dialogue for cooperative response generation. *Computational Intelligence*, 9(3), 1993.

- [18] Fei Song and Robin Cohen. *Temporal Reasoning during Plan Recognition, Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI91)*, pages 247-252, July 1991.
- [19] L. M. Wills. Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45(2):113–172, February 1990.
- [20] S. Woods and Q. Yang. Program understanding as constraint satisfaction. In *Proceedings of the IEEE Seventh International Workshop on Computer-Aided Software Engineering (CASE-95)*, pages 318–327. IEEE Computer Society Press, July 1995. Also appears in the *Proceedings of the Second Working Conference on Reverse Engineering (WCRE)*, July 1995.
- [21] S. Woods and Q. Yang. The Program Understanding Problem: Analysis and A Heuristic Approach In *Proceedings of the 18th International Conference on Software Engineering (ICSE-96)*, pages 6–15Berlin, Germany.
- [22] Q. Yang and P. Fong. Solving partial constraint satisfaction problems using local search and abstraction. University of Waterloo, Technical Report CS-92-50, Waterloo, Ontario, 1992.