Applying Plan Recognition Algorithms To Program Understanding

Alex Quilici
Department of Electrical Engineering
University of Hawaii at Manoa
Honolulu, HI 96822

Qiang Yang
School of Computing Science
Simon Fraser University
Burnaby, British Columbia V5A 1S6

Steven Woods
Department of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1

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.

1 Introduction

Program understanding is often described as the process of recognizing program plans in source code [22, 15, 7, 21, 9]. 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 [4, 5]. 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 we 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, and show how one existing AI plan recognition algorithm can be modified to take this into account. Finally, we discuss the practical relevance of plan-based program understanding to real-world software engineers working to reverse engineer legacy systems.

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 it 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.

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

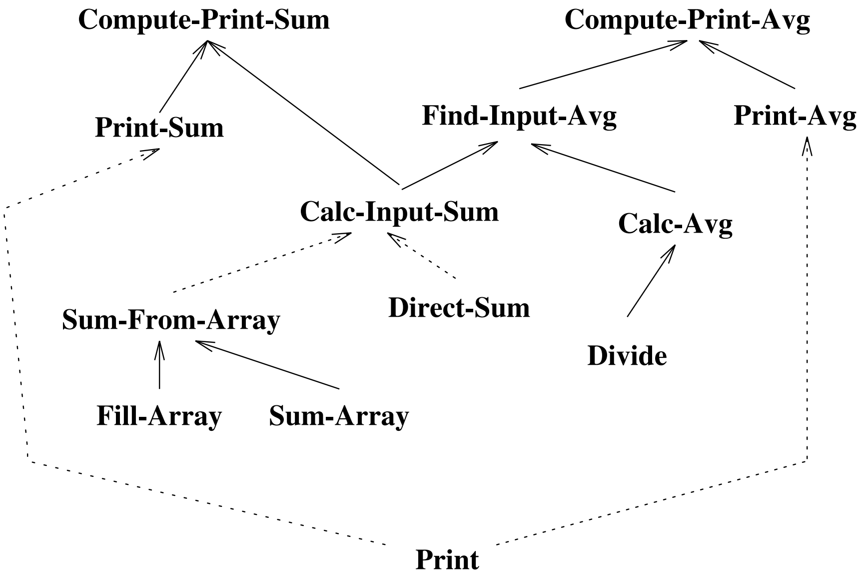

Figure 1: An Example Action Hierarchy

2.1 The AI Approach to Plan Recognition

Kautz and Allen [4, 5] formalized an approach to plan recognition that has served as a primary building block for many subsequent plan recognition methodologies, including [19, 20]. 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.

Kautz and Allen's approach[2] 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, as shown in Figure 1.

The first is specialization relationships between actions, which this representation uses to capture the notion that there are multiple ways to perform a given task and that a given action can be used to perform multiple tasks. For example, both **Sum-From-Array** and **Direct-Sum** are ways to accomplish **Calc-Input-Sum**. Similarly, **Print** is one way to accomplish both the **Print-Sum** and the **Print-Avg** actions. The other is decomposition relationships, which are used to represent that a plan requires a set of actions. For example, **Find-Input-Avg** requires both a **Calc-Input-Sum** and **Calc-Avg**. (Although not shown, the action hierarchy also captures constraints between these actions, such as temporal ordering.)

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 a *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

[2]The algorithms for this approach are detailed in [4].

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 observing a **Fill-Array**, there will be two possible explanations for it (**Compute-Print-Sum** and **Compute-Print-Avg**). After then observing a subsequent **Divide**, Kautz's algorithm will recognize **Compute-Print-Avg** as the plan. This is done by applying the simplicity heuristic, which recognizes that **Compute-Print-Avg** is the simplest way to cover both actions.

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.

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 line of C code:

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

We can view this example as a three observed actions: **Sum-Pair**, **Divide-Pair**, and **Assign**. 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.

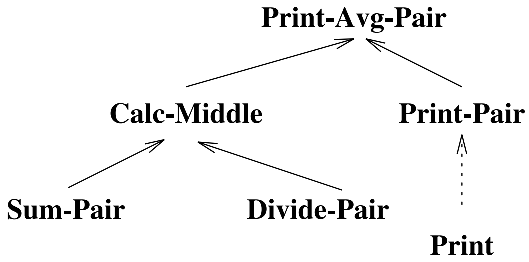

Figure 2: Another Example Action Hierarchy

Upon recognizing **Sum-Pair**, the explanation is **Sum-Pair** is-part-of **Calc-Middle** is-part-of **Print-Avg-Pair**. After recognizing **Divide-Pair**, the explanation is substantially the same, except that it now covers the **Divide-Pair** as well. However, if the **Print** never occurs, the problem is that this explanation cannot be the case. The explanation, in fact, should be limited to **Sum-Pair** and **Divide-Pair** being a part of **Calc-Middle**.

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.

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

$$H_K \supseteq E$$

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 the *principle of minimal exact-coverage*

Situations like the one in our example can occur frequently in program understanding because of *incomplete plan libraries*. It is unlikely that a plan library will contain all the plans necessary to understand a program [1, 16]. **Sum-Pair**, for example, has a myriad of uses, only some of which will be captured by any plan library. The result is that any algorithm we use must be capable of producing a forest of intermediate plans and not attempt to infer potentially incorrect high-level groupings.

It is reasonable to wonder whether Kautz's approach can easily be modified to address this problem. An initial modification is to change it so that, after all events have been processed, it runs through all of the explanation chains it forms and eliminates any that covers an event that did not occur. This guarantees that the algorithm never terminates with an incorrect explanation. However, since this algorithm does not carry along every candidate explanation, it may lose the correct portion of the overall explanation (**Calc-Middle**, in our example). To address this, we are forced to modify the algorithm to carry around all potential explanations and to use search to eliminate those that are missing events, a process which is likely to be inefficient.

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 To Support 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 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 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 action.

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 Problem [3] CSP typically con-

[3] See [8] for an accessible and detailed treatment of Constraint Satisfaction Problems.

sists 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 [6] and local search-based methods [18, 11, 24], constraint-propagation problem simplifications [12, 2, 14], and hierarchical exploitation of problem structure [3], 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 [17] and [22].

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 the abstract program plan or ADT. When we start any given CSP for recognizing a particular plan, the variable represents 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.)

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.

5.1 An Experiment

Using the 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 the 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.

Why have we chosen to work with artificially generated programs rather than real-world programs? Our primary motivation has been that we wanted to focus solely on the scalability of the recognition algorithm as programs with similar characteristics grow in size. In particular, we wanted to keep the distribution of AST components, the particular plans we were trying to locate, and the likelyhood of finding those plans constant across different program sizes. That's impossible to do with real-world programs. While we can certainly find real-world programs of varying sizes, the distribution of their components and the particular plans they contain will vary significantly.

Our approach runs the obvious danger of generating artificial programs that are far divorced from real-world programs. However, we have tried to mitigate this problem in several ways. First, we are generating programs containing plans that frequently appear in real-world programs, such as traversing arrays or strings, and by generating several different distributions. In addition, we view work with artificially generated programs as a baseline we can use in the future when we begin to work with real-world programs.

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 plan[4].

5.2 Results And Analysis

Figure 3 shows the results of running our experiments. The plan instances we tested had an average of approximately 10-15 components and 20-25 constraints.

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 [23], it's a demonstration that program plan recognition is 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 [13], it suggests that we may well be nearing the point where we can apply program plan recognition to modules of real-world legacy systems. If nothing else, there are many real-world *modules* of 5000 lines or less, and it appears to be worth applying our CSP-based plan recognition techniques on those modules.

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 do 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. In essence, we can view our initial results as showing the potential for the CSP-based approach with only minimal structural constraints in the programs being understood.

[4] In 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.

6 Future Work

There are several key areas for us to explore in the future. One is to compare the performance of this plan recognition algorithm to existing special-purpose algorithms for program plan recognition. We have done some initial work in this direction, which shows that this approach compares favorably with at least several of these algorithms[17].

Another is to verify that our initial performance figures are not artifacts of our experiments and our use of simulated programs with only control-flow-like constraints. Our feeling, however, is that working with real programs and data-flow constraints will further reduce the amount of work our plan recognizer does.

The last is to extend our plan recognizer to use constraint-satisfaction for all tasks, not just verifying whether a hypothesized plan is present. That is, we could devise an alternative CSP representation [22]. 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. We are currently constructing this plan recognizer and plan to compare its performance with the approach to plan recognition discussed in this paper.

7 Relevance To The Real-World

We have focused primarily on the scalability of algorithms for recognizing program plans. But suppose it turns out that recognizing instances of a given plan is sufficiently tractable that we can deploy our understander and apply it to real-world programs—then what?

One key problem is that the plan recognizer requires a library of program plans. Our simple example to illustrate the behavior of Kautz's algorithm showed that a relatively complex hierarchy is required to understand just a few lines of code. That seems to imply that a significantly more complex hierarchy will be required to understand 5000-line modules. It's clear that to apply plan-based understanding to real-world systems we will need a cost-effective way to create plan hierarchies.

We have begun exploring an approach for helping programmers construct a plan library. The idea is to provide

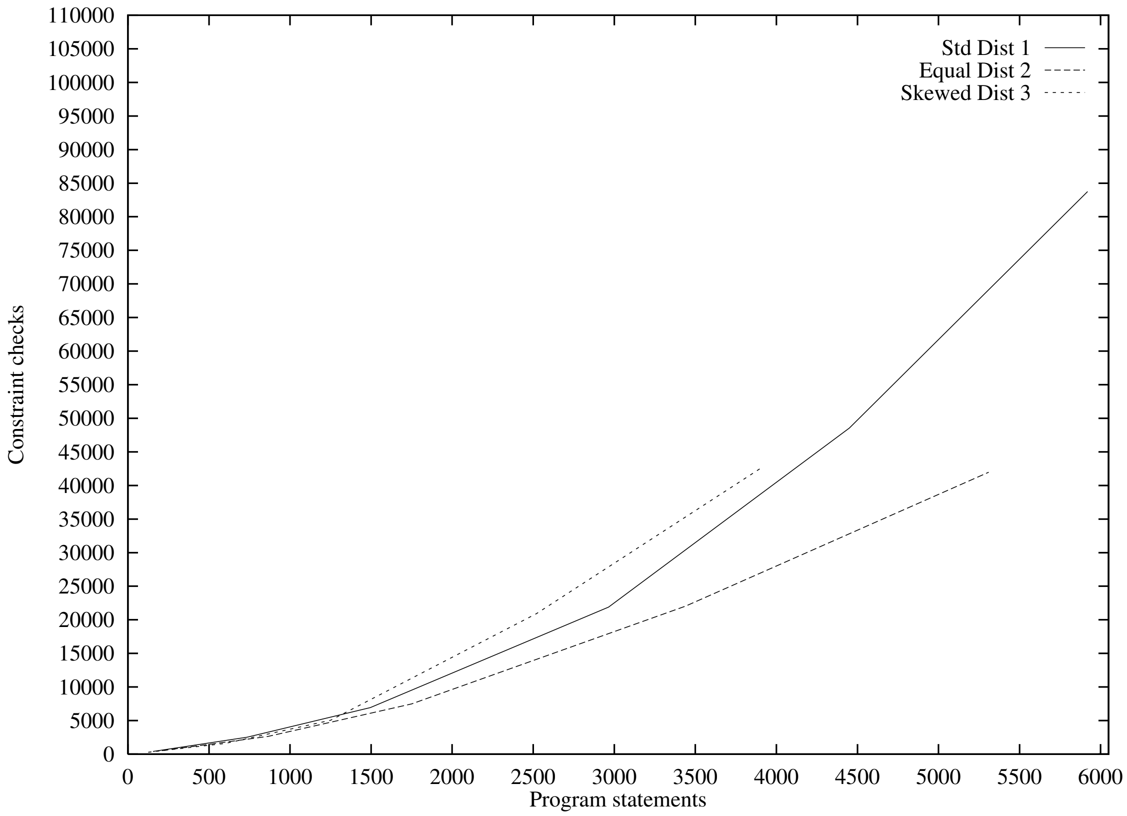

Figure 3: The results of our experiments.

programmers with a tool that allows them to provide plans by example. In particular, the approach is to let them highlight existing code as an instance of a plan, provide them with a detailed view of the components and constraints present in this instance, and allow them to delete and/or generalize constraints and components. The system can support this process by checking whether various combinations of the components/constraints present in this plan instance correspond to already-entered plans, and then automatically grouping and replacing them with previously-defined plans. The end result is a definition of the plan and links from it to other library entries. Given a sufficiently fast program understanding algorithm, the set of programs that may contain the user-provided plan can be immediately searched, and the user can adjust the plan's definition based on the results.

Besides the technical issues involved in constructing this tool, it's an open, emprical question whether such a tool can be used to cost-effectively provide plan libraries. However, it does suggest one possible path toward addressing the problem of how the necessary plans are provided to program understanding systems. Such a tool also suggests one near-term application of plan-based program understanding technology: letting users locate *conceptually* similar code fragments within a set of source files by using the tool to specify characteristics of the code to search for.

8 Conclusion

Program understanding is often viewed as a task of understanding the plans inherent in a software code. We have demonstrated that there are serious problems with the naive notion of simply applying AI plan recognition algorithms, and that these problems in some sense justify the rejection of this AI algorithm by researchers in program understanding. However, we have also demonstrated that by carefully analyzing the problems that arise in applying at least one existing AI plan recognition algorithms to program understanding, we can construct a variant of that algorithm that appears effective in efficiently recognizing certain classes of plans in real-world programs. In particular, we have demonstrated that by combining AI plan recognition techniques with AI CSP techniques, we can efficiently discover instances of plans in programs in up to 5000-or so line range.

Acknowledgments

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] D. Chin and A. Quilici. DECODE: A Cooperative Program Understanding Environment. *Journal of Software Maintenance*, 8(1), pp. 3-34, 1996.

[2] R. Dechter, From local to global consistency. *Artificial Intelligence*, 55:87–107, 1992.

[3] E. Freuder and J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[4] H. Kautz. *A Formal Theory of Plan Recognition*. PhD thesis, University of Rochester, Department of Computer Science, Rochester, New York, 1987.

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

[6] 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.

[7] W. Kozaczynski and J. Q. Ning. Automated program understanding by concept recognition. *Automated Software Engineering*, 1:61–78, 1994.

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

[9] W.L. Johnson. *Intention Based Diagnosis of Novice Programming Errors*. Morgan Kaufman, Los Altos, CA, 1986.

[10] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[11] 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.

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

[13] 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.

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

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

[16] A. Quilici. Reverse Engineering of Legacy Systems: A Path Toward Success. In *Proceedings of the 17th International Conference on Software Engineering*, pp. 333-336, Seattle, WA, April 1995.

[17] A. Quilici and S. Woods. *Toward A Constraint-Satisfaction Framework for Program Understanding*, *Journal of Automated Software Engineering*, to appear, 1996.

[18] R. Sosic and J. Gu. A polynomial time algorithm for the n-queens problem. *SIGART*, 1(3), 1990.

[19] P. van Beek, R. Cohen, and K. Schmidt. From plan critiquing to clarification dialogue for cooperative response generation. *Computational Intelligence*, 9(3), 1993.

[20] 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.

[21] L. M. Wills. Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45(2):113–172, February 1990.

[22] 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.

[23] 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.

[24] 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.