Some Experiments Toward Understanding How Program Plan Recognition Algorithms Scale

Steven Woods
Department of Computer Science
University of Waterloo
Waterloo, ON N2GL 3G1

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

Abstract

Over the past decade, researchers in program understanding have formulated many program understanding algorithms but have published few studies of their relative scalability. Consequently, it is difficult to understand the relative limitations of these algorithms and to determine whether the field of program understanding is making progress. This paper attempts to address this deficiency by formalizing the search strategies of several different program understanding algorithms as constraint satisfaction problems, and by presenting some preliminary empirical scalability results for these constraint-based implementations. These initial results suggest that, at least under certain conditions, constraint-based program understanding is close to being applicable to real-world programs.

1 Introduction

Program understanding can be viewed as a three-step process: parsing the program into an AST augmented with data and control flow information, canonicalizing this internal representation into a simpler form by applying transformations (such as transforming relational expressions so that they involve only the greater than operator and not the less than), and recognizing instances of program plans in this canonical internal representation (usually using a library of program plans). The underlying assumption is that by recognizing plan instances we can recover some of the programmer's abstract concepts and intentions [20, 8].

The heart of this approach is the "comparison" algorithm between plans (represented by schemas and knowledge constraints) and code (represented by annotated abstract syntax trees). However, this process is exponential in the worst case [24, 7, 6, 11] and proven to be NP-Hard [27]. In particular, recognizing instances of a particular program plan in a given source code is $O(S^A)$, where S is the size of the source and A is the number of plan actions.

To address this problem, most plan recognition algorithms use heuristic strategies [26, 17, 11, 7, 23, 9]. Unfortunately, the expected performance of these heuristic approaches is difficult to determine analytically, and there have been no detailed, empirical studies that demonstrate the resulting efficiency of these algorithms. Previous efforts have instead focused on issues such as plan library completeness for recognizing plan instances on a group of similarly sized programs for performing a particular task [7, 9]. Where studies have taken efficiency into account, it has been to identify what factors, such as constraint ordering and constraint strength, are critical to the performance of the recognizer [24, 11]. The actual performance of various plan recognizers, however, has been limited to anecdotal discussions. The result is that we are unable to draw conclusions about how well these plan recognition algorithms scale with program size.

This paper is a first step toward remedying this situation. It presents some results that compare the performance of constraint-satisfaction-based implementations of several previously published approaches to program plan recognition [1, 17, 11], and it presents an extended collection of experiments in the scalability of the most promising of these approaches [26].

The rest of this paper is organized as follows. Section 2 describes our initial forays in comparing a pair of plan recognition approaches. Section 3 describes a collection of new experiments in the scalability of a particular constraint-satisfaction-based plan recognition algorithm. Section 4 discusses what our initial results say about the program understanding problem as a whole. And Section 5 summarizes our work.

2 Comparing Plan Recognizers

It is difficult to compare systematically the performance of different program understanding algorithms, as they tend to use widely differing representations and a collection of specialized heuristic tricks. One proposal to address this problem is to view program understanding as a constraint-satisfaction problem (CSP) and to then map existing program understanding algorithms into this framework [26].[1]

The idea is that if a CSP framework is sufficiently general to unify existing approaches, we can then take advantage of it to compare their relative performance and to better understand the relative strengths and weaknesses of these algorithms. In addition, we can potentially achieve scalability of these approaches by augmenting them with the mechanisms developed for efficient heuristic solving of different classes of CSPs. These mechanisms include global [10] and local search-based methods [21, 13, 29], constraint-propagation problem simplifications [14, 2, 16], and hierarchical exploitation of problem structure [5], as well as hybrid combinations of these approaches.

Our assumption is that the CSP framework is adequate to this task. One reason for our making this assumption is that others have observed how *constraint* exploitation has played a pivotal role in almost all recognition strategies [8] and how structural and domain constraints help control the complexity of understanding in practice [24]. At a minimum, the described structure within library program plans provides constraints on which code fragments can possibly be plan instances.

Another reason is our initial success in mapping several different existing algorithms into this framework. In particular, with the CSP approach in mind, we took several algorithms (from the Concept Recognizer [11] and DECODE [1, 17]) and mapped them into a CSP framework [18]. We chose the Concept Recognizer because it appeared straightforward to transform its component and constraint plan representation into a constraint-satisfaction framework. We chose DECODE because its approach was based on the Concept Recognizer, but extended it to use a more complex recognition algorithm and plan indexing to try to address efficiency concerns. This made it an interesting test case of the CSP framework, as it was not immediately obvious how to map it into that framework. We call the resulting CSP implementation of the generic Concept Recognizer representation "MAP-CSP" and the resulting CSP implementation of DECODE "Memory-CSP".

2.1 Our Initial Experiments

In previous work [26], we introduced the constraint-based understanding approach and presented some early empirical results. The experiments reported in this work extend these results over much larger source programs, and over a range of plans of varying sizes, and in sources generated according to several different distributions of program structures.

[1] See [12] for an excellent overview of constraint satisfaction, or [22] for more detail.

```
define TRAVERSE-STRING(String) isa TRAVERSE-PLAN
define PRINT-STRING(String) isa PRINT-PLAN
define PRINT-CHAR(Char) isa PRINT-PLAN
define ZERO(Dest) isa ASSIGN-PLAN

plan TRAVERSE-STRING(String: ?a)
  components
    decl:     DECL-ARRAY(Name: ?s,
                         Items: ?max,
                         Type: char)
    init:     ZERO(Dest: ?i)
    loop:     LOOP(Test-Result: ?r, Body: ?body)
    access1:  ACCESS(Op1: ?s, Op2: ?i,
                     Res: ?val1)
    test:     BIN-OP(Op1: ?i, Op2: ?val1,
                     Op: !=, Res: ?r)
    access2:  ACCESS(Op1: ?s, Op2: ?i,
                     Res: ?val2)
    update:   INCREMENT(Op: ?i)
  constraints
    declbef:  ControlPath(decl, loop)
    initbef:  DataDep(test, init, ?i)
    acc1bef:  DataDep(test, access1, ?val1)
    testin:   DataDep(loop, test, ?r)
    acc2in:   ControlDep(access2, ?body)
    updaft:   DataDep(access2, update, ?i)
  index
    access2 WHEN accin

  implies PRINT-STRING(String: ?a)
    with
      dump:     PRINT-CHAR(Source: ?value)
    when
      dumpaft:  DataDep(dump, access2, ?v)
```

Figure 1: An example plan.

Our initial goal was to compare the performance of the Concept Recognizer and DECODE understanding algorithms in recognizing instances of program plans. Our initial experiments in comparing the performance of these algorithms focused on the specific task of recognizing instances of traversing a string in C programs. To give a feeling for the complexity of the plan being recognized, Figure 1 shows the DECODE plan for this task[2] and Figure 2 shows the resulting constraint-satisfaction based plan for the task.[3]

These experiments concentrated on comparing the performance of different algorithms for plan recognition in terms of the time taken to recognize all instances of a particular plan as programs increased in size. To keep the focus on scale issues alone, our desire was to have programs of varying sizes available where those programs contain correspondingly more instances of the plan as the programs increase in size, and to have programs with the same relative distribution of different program entities (loops, tests, and so on), regardless of size. As a result, our approach was to automatically generate test programs to be understood,

[2] The original Concept Recognizer plan is the DECODE plan, minus the indexing.

[3] The representation is a list of components followed by a list of constraints. The components are described by an internal name, the component's node type, and the node's slots. The constraints are described by a constraint name and the particular nodes or slots it constrains.

```
'( "quilici-t1"
   (
     (q1-c While       (ResultA (boolean)))
     (q1-d Begin       (Block1 (block)))
     (q1-g Assign      (NameC (array (char)))
                       (IndexC (int))
                       (ElemB (char)))
     (q1-e End         (Block2 (block)))
     (q1-i Increment   (IndexD (int)))
     (q1-a Decl        (NameA (array (char)
                                    (0 10000))))
     (q1-b Zero        (IndexA (int)))
     (q1-f Assign      (NameB (array (char)))
                       (IndexB (int))
                       (ElemA (char)))
     (q1-h Not-Equals  (ElemC (char))
                       (NULL (char))
                       (ResultB (boolean)))
   )
   (
     (before-p    (q1-c q1-d))
     (close-to-p  (q1-c q1-d) 10)
     (before-p    (q1-d q1-g))
     (same-name-p (q1-d q1-e) (Block1 Block2))
     (before-p    (q1-g q1-e))
     (before-p    (q1-b q1-c))
     (before-p    (q1-a q1-b))
     (before-p    (q1-b q1-h))
     (before-p    (q1-d q1-e))
     (before-p    (q1-f q1-h))
     (before-p    (q1-g q1-i))
     (before-p    (q1-d q1-i))
     (before-p    (q1-i q1-e))
     (same-name-p (q1-c q1-h) (ResultA ResultB))
     (same-name-p (q1-f q1-h) (ElemA ElemC))
     (same-name-p (q1-a q1-f) (NameA NameB))
     (same-name-p (q1-a q1-g) (NameA NameC))
     (same-name-p (q1-b q1-f) (IndexA IndexB))
     (same-name-p (q1-b q1-g) (IndexA IndexC))
     (same-name-p (q1-b q1-i) (IndexA IndexD))
  ))
```

Figure 2: Our CSP representation for the previous plan

starting with an instance of a given plan, adding program statements surrounding each instance and adding more instances until we constructed programs of the desired sizes. In particular, we added statements randomly according to a distribution that corresponded to our observed distribution against a cross-section of student C programs.[4]

We have chosen to work with artificially generated programs rather than real-world programs because we want to focus solely on the scalability of the recognition algorithm as programs with similar characteristics grow in size. In particular, we want to keep the distribution of AST components, the particular plans we were trying to locate, and the likelihood of finding those plans constant across different program sizes. This is impossible with real-world programs. While it's certainly possible to find real-world programs of varying sizes, the distribution of their components and the particular plans they contain vary significantly.

To have confidence in our results, we also want the ability to process multiple programs of the same size and component distribution. To do this with real-world programs, however, would require access to repository of programs large enough that by change some programs with the same size would have similar distributions.

Clearly, this approach runs the danger of generating artificial programs that are far divorced from real-world programs. However, we have tried to mitigate this problem by 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 this work with artificially generated programs as a baseline we can use in the future when we begin to work with real-world programs.[5].

Given this experimental framework, we generated programs of varying sizes at intervals of 50 from 50 to 1000 statements, with 10 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. The idea is that constraint checks are a reasonable proxy for performance, since across our methodologies the work performed to check each of the constraints we represent is the same. Others [24] have also described the performance in terms of the effectiveness of constraint evaluation. In addition, while certain methodologies require differing amounts of computation during the search, we have verified that the CPU-second graphs of these same experimental results yields comparable graphs. For comparison, examples requiring approximately 2500 constraint checks utilize roughly 6 CPU seconds in our examples. Since CPU usage is highly variable across implementations and platforms, constraint checks offer a more domain-independent reference point.

Figure 3 shows the results of our experiments. There were five tests based on typical CSP solution algorithms. Details of these algorithms may be found in [12].

1. MAP-CSP with simple backtracking. This is a straightforward, very basic approach that can be considered the CSP equivalent to the Concept Recognizer approach to plan recognition.

2. MAP-CSP with forward checking, dynamic rearrangement, and no variable sorting. This can be thought of as taking the CSP equivalent of the Concept Recognizer approach and improving it to take advantage of one common approach to solving constraint-satisfaction problems.

3. Memory-CSP with first phase backtracking and second phase forward checking with advanced variable

[4]Our initial assumption here is that small student programs suggest a reasonable distribution of components. Our later experiments at least partially address this potential problem by using several different distributions.

[5]We currently have begun work to integrate our CSP methodology with a richer constraint system capable of handling real C++ programs pre-processed using Devanbu's GENOA [3, 4]

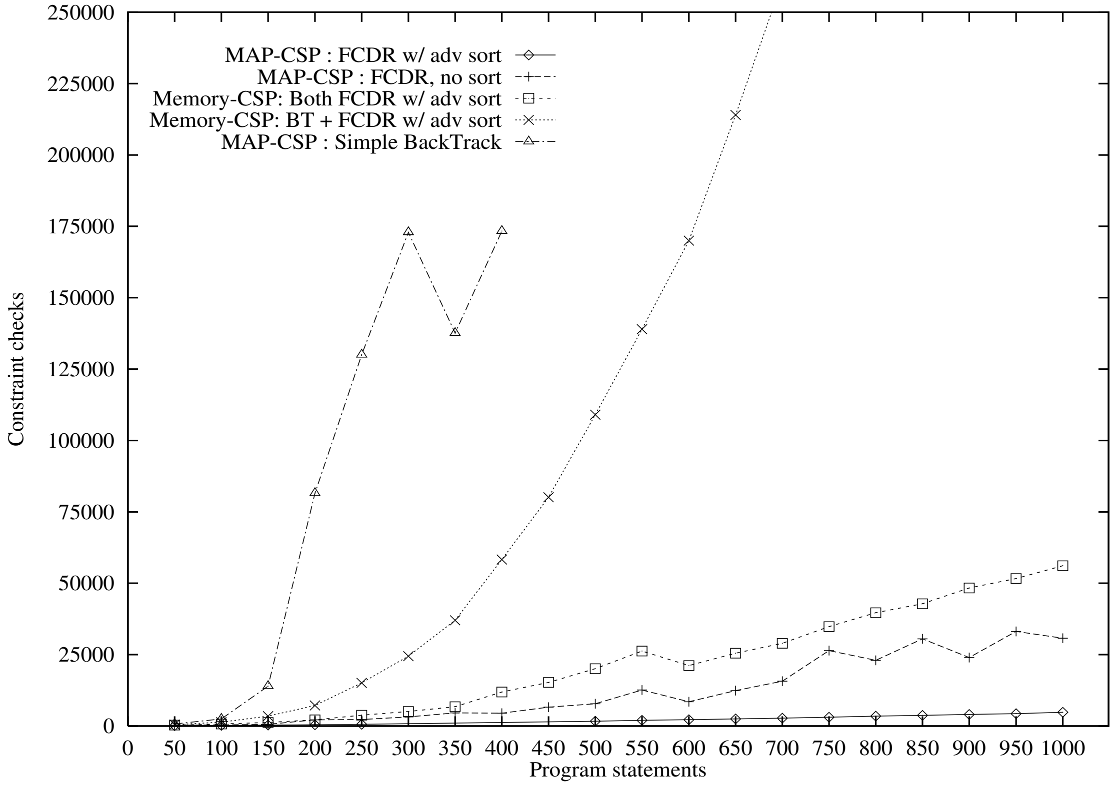

Figure 3: The results of comparing different program understanding approaches.

sorting. This is a straightforward, very basic CSP equivalent of DECODE's indexing approach to plan recognition.

4. Memory-CSP with forward checking with advanced variable sorting for both phases. This is an improved CSP version of DECODE's indexing approach.

5. MAP-CSP with forward checking, dynamic rearrangement, and advance variable sorting. This can be thought of as a Concept Recognizer approach that takes full advantage of another common approach to solving constraint-satisfaction problems.

All experiments were limited by a 600 CPU second time for any individual search. For 600 and more statements, there were an increasing number of aborted searches due to exceeding the pre-assigned time limit.

This experiment leads us to two conclusions:

1. Combining DECODE's approach with well-known CSP methods improves its overall performance.

2. However, combining well-known CSP methods with the Concept Recognizer's original plan representation outperform DECODE's more complex, indexed plan representation.

In hindsight, there is an obvious explanation for this result. It's been observed that constraint application should be ordered on the basis of constraint ordering heuristics derived from the predicted relative utility and cost of a given type of constraint application [24]. DECODE's indexing is by its very nature static. The assumption is that an expert can examine a plan and determine *in advance* which component is likely to indicate the presence of the entire plan, and which constraints are the most useful in rejecting incorrect potential plans, regardless what events are actually present in a given program. The ordering that results is set in advance, when the plan is added to the plan library, and never changes. In contrast, the heuristic constraint satisfaction approaches are dynamic, determining which constraints to satisfy based on properties of the particular program being examined. They automatically tune themselves to the particular properties of the program in which we are trying to find plan instances.

2.2 Some Problems With These Experiments

There are some problems with these initial experiments. One is that we used a locality heuristic to approximate control and data-flow constraints,[6] which means our generated test programs did not necessarily have the same structural properties that real-world programs might have. We are currently engaged in extending our results to include actual control-flow and data-flow based on a publicly available analyzer for C. Another problem is that we ran these initial sets of experiments identifying only a single plan rather than on a set of different plans. In particular, Figure 4 shows results in locating instances of the plan of Figure 2, and Figure 5 shows the results from locating instances of a more specialized version of this plan.

Because of these problems, there are limits to the conclusions we can draw from our initial experiments. It may well be that this experiment artificially limited the performance of DECODE's indexing approach, which was designed to exploit structural properties. It may also be that our selected plans have strange properties that make it more or less amenable to using constraint satisfaction techniques to recognize them. However, because of the well-constrained (near linear) appearance of the curve of constraints evaluated for the best of the CSP approaches, it seems reasonable to further explore how that approach scales in several different directions.

3 Evaluating the Scalability of the CSP-Based Approach

We performed two specific sets of experiments with our CSP approach[7]. The first set of tests involved expanding the size of the programs considered to various sizes between 1000 and 6000 (specifically, we generated programs at increments of 500 statements). For each of these sizes, we used three different distributions of program events or constructs: a "standard" distribution, an "equal" distribution, and a "randomly skewed" distribution. Table 1 shows our standard distribution.

In this distribution, declarations and tests appear four times as frequently as loops and array accesses, assignments appear three times as frequently, and so on. In an equal distribution, each construct appears approximately the same number of times. And in a random distribution, we randomly chose particular event types to appear much more frequently in the program that others.

[6]This heuristic is described in detail in [25]

[7]For a good explanation of the CSP representational scheme for partial local explanations, see [26, 25].

Statement Type	Frequency	Percentage
While	1/22	4.5
Zero	1/22	4.5
For	1/22	4.5
Block	2/22	9.0
Increment	2/22	9.0
Not-Equals	2/22	9.0
Print	2/22	9.0
Assign	3/22	13.5
Declare	4/22	18.0
Test	4/22	18.0

Table 1: Standard distribution of program statements.

3.1 Scaling With Program Size

Figure 4 shows the results of running a similar experiment using a plan template containing 9 components and 20 constraints. The line charted is the median of a 95% confidence interval generated for the set of programs tested at each size level.

One way to summarize this graph is by the ratio of constraints evaluated per statement. At 1000 statements, this ratio is approximately 4–1 for all of the approaches. At 3000 statements, the ratio varies between 10–1 for the random distribution to 5–1 for the equal distribution. At 5000 statements it is 12–1 for the standard distribution and 7–1 for the equal distribution. At 6000 statements this ratio is 14–1 for the standard distribution.

These numbers imply several things:

- The distribution of types of components within the program has a large effect on the performance of the CSP algorithm.
- The apparent linearity seen in our original experiments appears to be the flatter portion of a (relatively slowly rising) higher order curve.

It is clear that the algorithm performs significantly better when all components appear equally within the program as opposed to either a randomly skewed distribution or a fixed distribution. A possible explanation for this behavior is that in our fixed and randomly skewed distribution, entities that appear frequently as components of the plan tend to appear frequently in the program. This suggests to us that we need to generate a distribution similar to real-world C programs performing a variety of tasks in different domains, generate testing programs based on those distributions, and then examine the performance of the algorithm across a variety of distributions. It may well turn out that the CSP approach

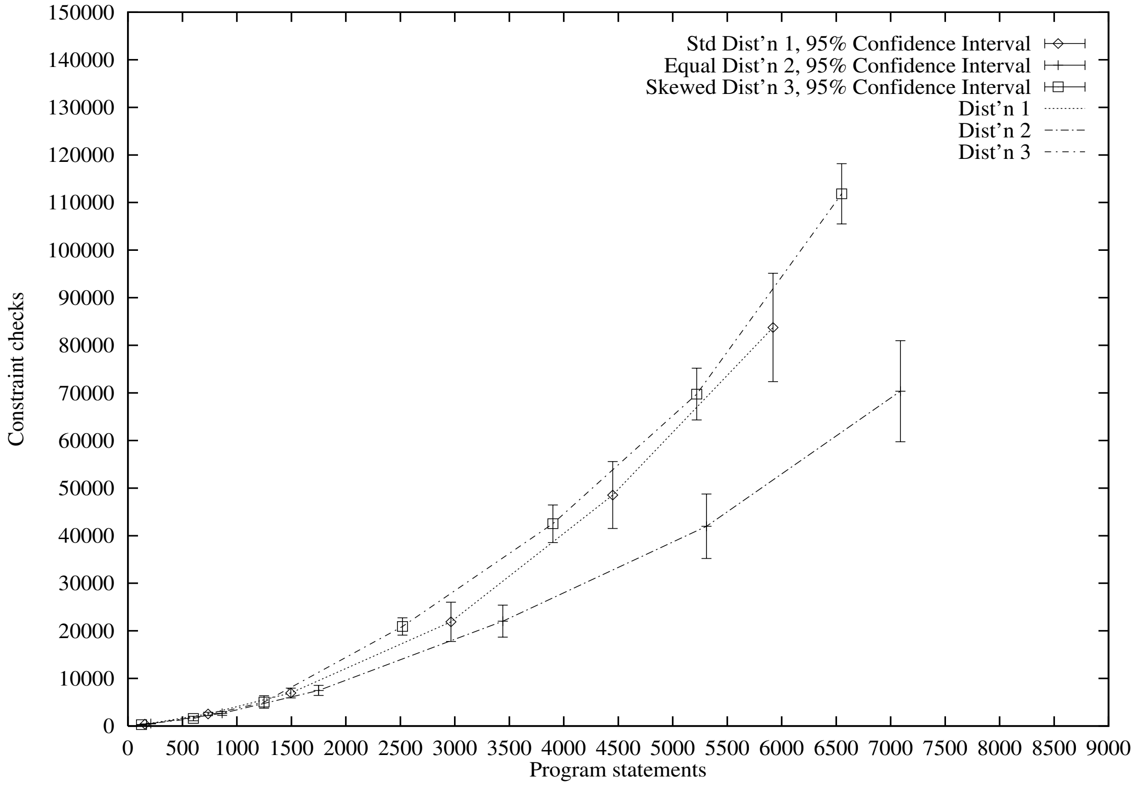

Figure 4: MAP-CSP, FCDR, standard plan over larger programs.

will perform noticeably better on some types of programs that others.

It is also clear the ratio of constraints evaluated to program statements is growing rapidly for our "standard distribution", from 5-1 at 2000 statements, to 8-1 at 4000 statements, to 14-1 at 6000 statements. This is problematic, but there are several reasons to remain optimistic. One is that our "simulated locality constraints" are much looser than the control-flow and data-flow constraints found in real programs. We therefore predict that this curve will flatten significantly when we run experiments using programs that preserve the structural properties of real-world programs. (This is because tighter constraints should reduce the size of the domain value sets, leading to a speedier solution to the CSP and fewer constraints to evaluate.) It is an open question, however, just how much it will flatten and how much that will slow the rate of growth of the number of constraints evaluated.

The other reason for hope is that, even if real-world structural constraints do not prevent the plan recognition algorithm from going exponential, it appears from our curves that instances of individual plans can be recognized in programs in the 5,000 statement range in a reasonable amount of time (the 60,000 or so constraints we need to evaluate takes less than 3 minutes or so on our workstations).[8] While 5000 statements is small compared to the sizes of real-world programs, it is within an order of magnitude of the size of modules in real-world COBOL programs or modules that have been created from legacy systems using semi-automatic techniques [15]. It's also far larger than the modules found in the large public-domain C programs we have examined.

3.2 Scaling With Plan Size

We have also run a second set of experiments designed to begin exploring the scaling properties of the CSP approach as the plans get larger in size. To do this, we explored plans that were considerably larger than our original plan (14 components and 29 constraints). Since the worst-case

[8] Doing a rough extrapolation of the current pattern of growth of this ratio, which appears to be doubling every 2000 statements or so, we get 26-1 at 8000 statements and 50-1 at 10000 statements, which is approximately 500,000 constraints, or about 20 minutes on our Sun workstations.

complexity of recognizing plan instances is exponential in the number of plan components, recognizing instances of these plans is theoretically much more difficult than recognizing instances of the original plan.

Figure 5 shows the results of running this set of experiments. We can summarize this graph using the same ratio of constraints evaluated per statement. At 3000 statements, the ratio varies between 20–1 for the standard distribution to 10–1 for the equal distribution. At 5000 statements, the ratio is 30–1 for the standard distribution and 12–1 for the equal distribution. As with the previous experiments, at that point the CSP algorithm starts to miss existing plans. Our conclusion here is that with smaller programs, at least, doubling the size of plan leads to approximately double the number of constraints that must be evaluated. This result is encouraging, but we need to see whether it holds up as we try larger and larger plans.

4 Implications for Program Understanding

This paper has explored some of the scalability of a constraint-based approach to recognizing all instances of a given plan. There are, however, other aspects to program plan recognition, that we have not addressed, including how to decide which plans to try to locate within a given program, and in which order to try those plans. Furthermore, we have focused on recognizing instances of single plans whose components are AST entries; we haven't examined the problem of recognizing higher-level plans, a task that is also a reasonable candidate for a constraint satisfaction approach [25]. Finally, we have ignored other aspects of program understanding, such the canonicalization problem, which involves difficult problems such as simplifying expressions.

Despite the narrow focus of our initial experiments, our initial results have several important implications. One is that it may well be necessary to have a modularization step that precedes the plan recognition process, where this step breaks the program into pieces of whatever size the program understanding algorithm can comfortably handle before the combinatorics become problematic. Some work on semi-automatic modularization of COBOL program has already been done that has demonstrated that large COBOL programs can be broken into modules of 25,000 or so statements [15]. This is only a factor of 5 larger that the point CSP approach can comfortably handle, which makes it appear worthwhile to determine whether those techniques can be extended to break programs down into even smaller modules.

In addition, even if we successfully recognize plans at the module level, there also needs to be a mechanism for combining this modular understanding that needs to follow the plan recognition process. It's an open question how we accomplish this task to come up with an understanding for a program as a whole, especially if the library is incomplete and we have only partial understanding of what a module does.

Finally, our at least partial success using CSPs in the understanding process suggests that perhaps they can be applied to other related tasks, such as selecting plans to attempt to recognize, or as part of the canonicalization process. However, it is an open and interesting research question how to do so.

5 Conclusions

Others [8, 24] have long recognized that the state of program understanding research will be improved not only by new plan representations and improvements in plan recognition algorithms, but also by empirical study. It has been suggested [24] that empirical studies will help us expand the size of the programs we can address, expand and refine libraries of clichés, and identify additional efficiency factors.

We have taken these suggestions to heart. The goal of our initial empirical studies has been to produce empirical results on the scalability of several existing plan recognition algorithms. We first mapped these algorithms into constraint satisfaction problems and compared their performance. These initial results suggested that a CSP implementation of a straightforward plan recognition algorithm can perform better than a CSP implementation of a more complex algorithm that relies on a more complex plan representation. We then took an initial look at scaling properties of the best-performing CSP implementation as the programs grow larger. It appears from our initial look that this CSP implementation takes a sharp polynomial increase in the amount of time to recognize plans as the programs grow in size, although the curve is bearable in the 5000-statement or less range. It also appears from an initial attempt to recognize larger plans that this CSP implementation does work that is roughly proportional to the size of the plans, likely as a result of the combined constraining effect of the kind of problems we are investigating.

Our results, however, should not be thought of as complete or definitive. They should instead be thought of as a few data points in a progress report on the state of the art of program understanding. In particular, the specific amount of work done by the CSP recognition algorithm can be reduced, perhaps significantly, by moving to real control-flow and data-flow constraints, an experiment we are in the process of setting up. This may well mean that significantly larger programs can be successfully understood. In addition, the relative amount of work done by

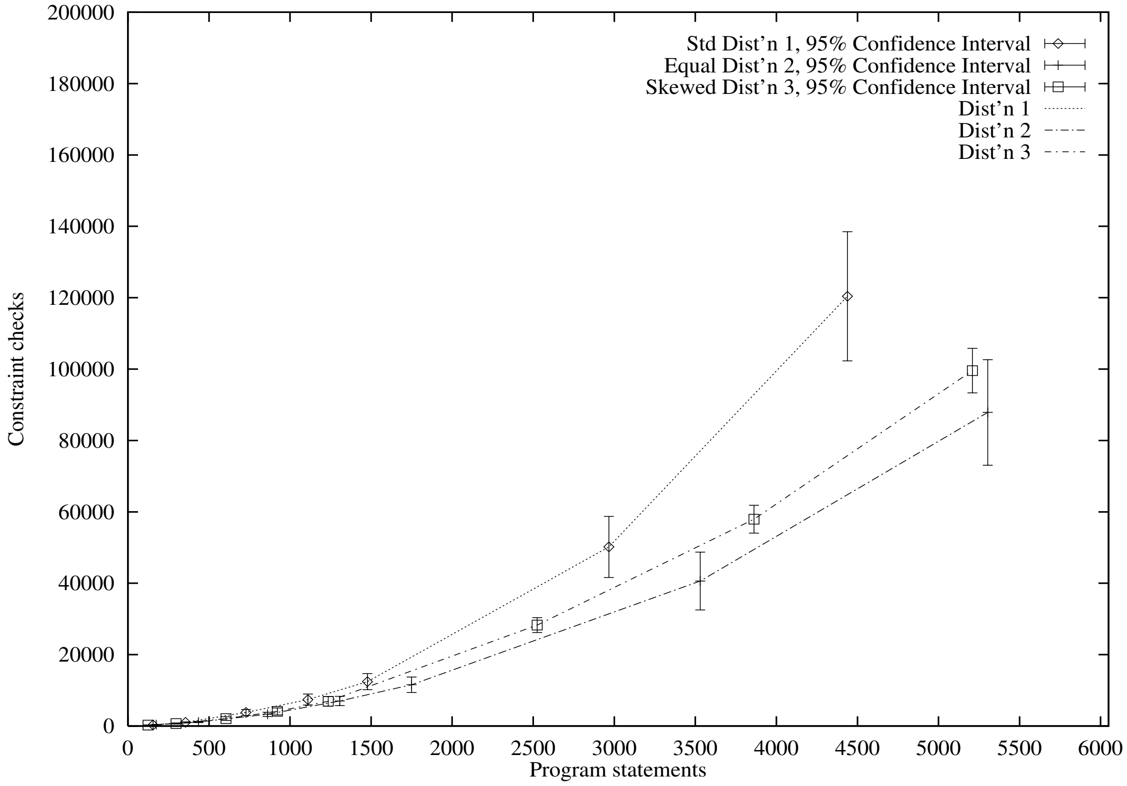

Figure 5: MAP-CSP, FCDR, using a larger plan.

the algorithm may increase rapidly as we move toward exploring larger and larger plans, rather than slowly as it had with our first few plans. This may mean that there is an practical upper bound on the size of individual plans that can be efficiently recognized in a program of a particular size. If our initial empirical results hold up, they suggest that automatically modularizing large programs and combining modular understanding are several important areas of future research.

Our hope is that this paper will spur others working in the area of program plan recognition to do one of two things: either map their understanding algorithms into the CSP framework so that we can easily compare their performance with the algorithms we have already mapped into the CSP approach, or to provide data on the performance of their program understanding algorithms as programs grow in size. Over time, this should allow us to arrive at a predicative model of performance for program plan recognition that will allow us to effectively offer limited understanding tools as valuable additions to reverse and re-engineering toolsets. This step is crucial to move beyond the understanding of "toy" programs and into the world of being a useful aid in the re-engineering of real legacy systems.

References

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

[2] Rina Dechter. From Local To Global Consistency. *Artificial Intelligence*, 55:87–107, 1992.

[3] P. Devanbu. GENOA/GENII — A Customizable, Language- And Front-End- Independent Code Analyzer. *Proceedings of the 14th International Conference on Software Engineering*, 1992.

[4] Prem Devanbu and Laura Eaves. Gen++ — An Analyzer Generator For C++ Programs. Technical report, AT & T Bell Labs, New Jersey, 1994.

[5] E. Freuder and J. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58:21–70, December 1992.

[6] J. Hartman. *Automatic Control Understanding For Natural Programs*. Research report ut-ai-tr-91-161, University of Texas at Austin, 1991.

[7] J. Hartman. Understanding Natural Programs Using Proper Decomposition. In *Proceedings of the International Conference on Software Engineering*, pages 62–73, Austin TX, 1991.

[8] J. Hartman. Technical Introduction To The First Workshop On AI And Automated Program Understanding. In *Workshop Notes, AAAI Workshop on AI and Automated Program Understanding*, Tenth National Conference on Artificial Intelligence, 1992.

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

[10] Grzegorz Kondrak and Peter van Beek. A Theoretical Evaluation Of Selected Backtracking Algorithms. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 541–547, 1995.

[11] Wojtek Kozaczynski and Jim Q. Ning. Automated Program Understanding By Concept Recognition. *Automated Software Engineering*, 1:61–78, 1994.

[12] Vipin Kumar. Algorithms For Constraint-Satisfaction Problems. *AI Magazine*, pages 32–44, Spring 1992.

[13] Steven Minton, Mark Johnston, Andrew Philips, and Philip Laird. Minimizing Conflicts: A Heuristic Repair Method For Constraint Satisfaction And Scheduling Problems. *Artificial Intelligence*, 58:161–205, 1992.

[14] Bernard A. Nadel. Constraint Satisfaction Algorithms. *Computational Intelligence*, 5:188–224, 1989.

[15] 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*, pages 222–230, 1993.

[16] Patrick Prosser. Hybrid Algorithms For The Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[17] Alex Quilici. A Memory-Based Approach To Recognizing Programming Plans. *Communications of the ACM*, 37(5):84–93, May 1994.

[18] Alex Quilici and Steven Woods. Toward A Constraint-Satisfaction Framework For Evaluating Program-Understanding Algorithms. *Journal of Automated Software Engineering*, 1996. To appear.

[19] Alex Quilici, Qiang Yang, and Steven Woods. Applying Plan Recognition Algorithms To Program Understanding. In *Proceedings of the Eleventh Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press, September 1996. To appear.

[20] Charles Rich. *Inspection Methods In Programming*. PhD thesis, Massachusets Institute of Technology, June 1987.

[21] R. Sosic and J. Gu. A Polynomial Time Algorithm For The N-Queens Problem. *SIGART*, 1(3), 1990.

[22] Edward Tsang. *Foundations Of Constraint Satisfaction*. Academic Press Limited, 24-28 Oval Road, London England, NW1 7DX, 1993.

[23] L. M. Wills. Automated Program Recognition: A Feasibility Demonstration. *Artificial Intelligence*, 45(2):113–172, February 1990.

[24] L. M. Wills. *Automated Program Recognition By Graph Parsing*. PhD thesis, MIT, July 1992.

[25] Steven Woods. *A Method of Program Understanding Using Constraint Satisfaction For Software Reverse Engineering*. PhD thesis, University of Waterloo, July 1996.

[26] Steven Woods and Qiang Yang. Program Understanding As Constraint Satisfaction. In *Proceedings of the IEEE Seventh International Workshop on Computer-Aided Software Engineering (CASE)*, pages 318–327. IEEE Computer Society Press, July 1995. Also appears in the *Proceedings of the 1995 Second Working Conference on Reverse Engineering (WCRE)*.

[27] Steven Woods and Qiang Yang. Approaching The Program Understanding Problem: Analysis And A Heuristic Solution. In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, 1996.

[28] Steven Woods and Qiang Yang. Program Understanding As Constraint Satisfaction: Representation and Reasoning Techniques. Under review, *Journal of Automated Software Engineering*, 1996.

[29] Qiang Yang and Philip Fong. Solving Partial Constraint Satisfaction Problems Using Local Search and Abstraction. Technical Report CS-92-50, University of Waterloo, 1992.