New Experiments With A Constraint-Based Approach To Program Plan Matching

Alex Quilici
Steven Woods
Yongjun Zhang

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

Abstract

In earlier work, we presented some preliminary empirical scalability results for a constraint-based program plan matching algorithm. Those initial experiments had several important shortcomings: they worked with a collection of artifically generated programs, and they applied a particular constraint satisfaction approach. This paper reports the results of a collection of new experiments that begin to address these deficiencies. In particular, we have begun experimenting with real-world C code, and we have begun exploring alternative approaches to solving constraint satisfaction problems. While not definitive, these new experiments provide further support for our earlier results, and they have led to a new approach that provides significant improvements in the scalability of our plan matching algorithm.

1 Introduction

A key part of any program understanding algorithm is the task of *plan matching*: locating all instances of a hypothesized program plan (such as a schema augmented with data and control constraints) in an internal representation of program source code (such as an AST augmented with data and control flow information). Unfortunately, this process is exponential in the worst case [2, 3, 6, 13] and has been proven to be NP-Hard [16]. 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. It is therefore an open question whether it is possible to develop efficient program understanding algorithms.

Most program understanding algorithms attack the combinatorial problems in plan matching by using heuristic strategies [1, 3, 6, 5, 8, 12, 17]. Unfortunately, the expected performance of these heuristic approaches is difficult to determine analytically. As a result, there is a need to perform empirical studies to better understand and predict how these program understanding algorithms will perform in practice.

In earlier work, we showed how program understanding could be viewed as a constraint satisfaction problem (CSP)[1], and we performed some initial studies on the efficiency of several constraint-based implementations of existing program understanding algorithms [9, 15]. These experiments involved generating a set of test programs, applying a particular constraint satisfaction approach to locate *all* instances of a given plan, and measuring the number of constraints evaluated in the process. Our experiments suggested that the effort required by these algorithms appeared to be suitable for recognizing smaller plans (containing 10 or so program entities) in source code of up to 5,000 lines in length, despite the amount of work (in terms of constraints being evaluated) being characterized by a steeply rising polynomial curve.

Although promising, these results were severely limited. First, they were dependent on artificial programs and locality constraints, not real-world programs and data- and control-flow constraints. And second, they focused on a single constraint satisfaction algorithm, Forward Checking with Dynamic Rearrangement (FCDR), rather than exploring a variety of different constraint satisfaction algorithms.

This paper is an attempt to remedy these defects. We report on a new series of experiments in which we begin working with real-world programs and we start considering different approaches to solving CSPs. It presents our experiments, their often surprising results, our explanations for those results, and the subsequent modifications to our understanding algorithm driven by those results. This paper is essentially organized along the path we took in exploring and trying to improve the performance of our program understanding algorithm.

[1] See [10] for an excellent overview of constraint satisfaction.

2 An Experiment With "Real" Programs

Our previous experiments were done with artificially generated programs and used locality constraints to capture relationships between plan components [9, 15]. Our prediction was that the performance of our constraint-satisfaction algorithm would improve when applied to real-world programs. The thought underlying this hypothesis is that the data-flow constraints present in real-world programs would be more effective than our locality constraints in shrinking the search space for program plans.

The obvious experiment is to take one of the plans used in those earlier experiments, search for it in a collection of real-world programs of various sizes, and measure the constraints evaluated in the process. However, this approach has several practical problems. One is that to have meaningful scaling results, it is necessary to vary the size of the programs but not the distribution of components, the plan being searched for, or the program's data-flow and control-flow complexity. Another is that our current home-built C data-flow and control-flow analyzer can only handle a subset of the language and can't compute data-flow relationships that cross function boundaries.

To address these problems, our initial real-world experiment works with variants of a single function that computes some basic statistics. That is, we have taken the core of the function's body and replicated it varying numbers of times to generate programs of different sizes from 50 lines to $5,000$ lines [18]. This experiment is then to search these programs for an array-averaging plan. As before, we use FCDR as our particular method of solving CSPs, and our measure of efficiency continues to be the number of constraint checks performed, as constraint checking is where the dominant amount of work occurs in an attempt to recognize a program plan.

Our original experiment with artificially generated programs actually searched for an array-traversal plan, not an array-averaging plan. However, our representation of plans is now simpler, as a result of the availability of data- and control-flow constraints and because of differences in the specific components available to represent program actions. As a result, the new representation of our array-traversal plan contains fewer components and fewer constraints than does the original. We decided to keep the number of components and constraints in the plan we're trying to locate constant across both experiments, rather than the specific task that the plan was being used to recognize. As a result, we now search for the array-averaging plan, which has the same number of components and constraints as our original array-traversal plan.

Figure 1 shows the results of running this new experiment with real-world programs, overlaid on our earlier results with the equivalent artificially-generated programs.

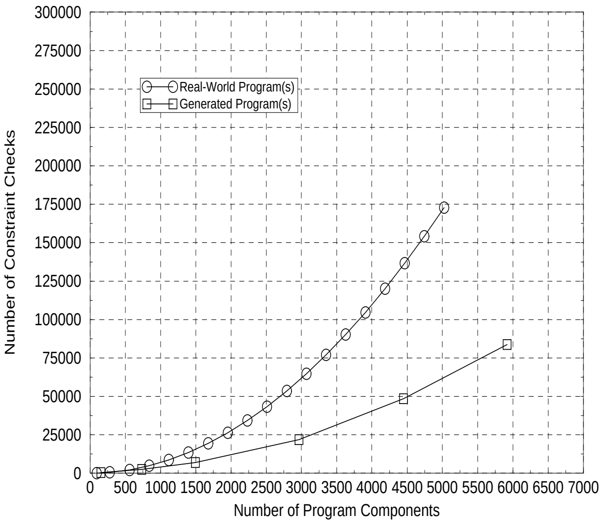

Figure 1: The results of our initial experiment with real-world programs.

Unfortunately, the results are exactly the opposite of what we predicted.

> *Applying our constraint-based plan matcher to real-world programs leads to a sharply rising curve of constraints evaluated.*

There are several potential explanations for this unexpected increase. One possibility is that while we are using similar plans, there is a substantially different component distribution between our artificial and real-world programs. For example, there were roughly twice as many assignments in the real-world program as in the artificial program, and one-fifth as many tests. Our earlier results have shown that the distribution of program components (equally distributed versus some components appear far more often than others) makes a significant difference in the performance of the recognizer [15]. Another possibility is that despite the array-averaging plan being the same size as the array traveral-plan, there are key differences in its particular combination of components and constraints that make it considerably more difficult to recognize. Still another possibility is that our particular constraint-satisfaction algorithm is not fully exploiting the tightness of real-world constraints. That is, there may be information that has been computed in the data-flow graph of the program that is not being used to reduce the size of the search space.

3 An Initial Experiment With A Two-Phase Constraint Satisfaction Algorithm

We decided to focus on the last possibility: namely, that our constraint-solving algorithm was not fully exploiting real-world constraints.

One key to solving CSPs quickly is to narrow down the sets of domain values for the nodes participating in any given constraint. That's because the constraint may be evaluated once for each pair of values in the cross product of the domain values of those nodes. The more we can reduce the size of the sets of domain values, the more we can reduce the complexity of the search space.

With this in mind, it appears that we can augment our initial plan recognition engine to exploit information that can be gleaned from static analysis. As part of the data-flow analysis of a program, we can extend the data-flow graph to group a node's successor and predecessor nodes by their type (e.g., assignment, addition, and so on). We can then use this information in a single preprocessing step to eliminate some of the domain values.

In particular, suppose we have a data-dependency constraint between two component types, A and B, such that B depends on A for some variable V. The domain values of A and B are all the instances $A_1, A_2, \ldots, A_m$ and $B_1, B_2, \ldots, B_n$ that appear in the source. For each node A_i, we can now in constant time determine whether there exists any value of type B in its successor set. If not, it means that there is no way A_i can have a data dependency on any instance of B, and we can eliminate A_i from A's domain values. Similarly, for each node B_j, we can in constant time determine whether there exists any value of type A in its predecessor set. If not, we eliminate B_j from B's domain values.

As a result of this insight, we modified our approach to have two phases after setting up the CSP. The first phase is domain-value pre-filtering, and the second is our usual use of FCDR to solve the CSP, but starting with the CSP that results from pre-filtering.[2] Figure 2 shows the results of our earlier experiment, redone using the two-phase approach.[3] These results were what we expected.

> *Domain-value pre-filtering significantly reduced the amount of work done during plan matching.*

In fact, pre-filtering reduces the number of constraints evaluated to the point where we can reasonably run ex-

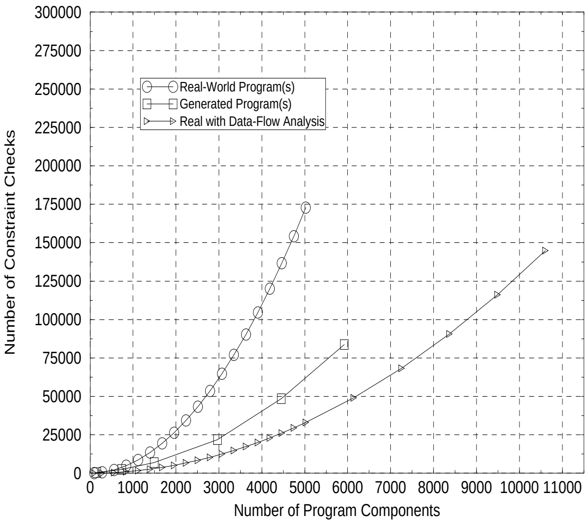

Figure 2: The results of an experiment with domain value pre-filtering.

periments on 10,000 line programs. It provides evidence for our earlier hypothesis that the data-flow information in real-world programs is in fact useful in constraining the amount of work done by the plan recognition engine and that our original constraint satisfaction engine performed poorly as a result of not sufficiently exploiting the available information.

4 Some Additional Experiments With The Two-Phase Approach

Our initial results with the two-phase, prefiltering algorithm were encouraging. As a result, we decided to perform several additional experiments to get a feel for how well this new algorithm scales with plan size. In particular, we repeated our earlier experiment with two new plans: one computes the sum of squares of an array's elements, and the other computes the statistical variance of an array's elements. The latter plan includes all of the components and constraints of the former and is approximately twice as large.

Figure 3 shows the results. Unfortunately, these results were disappointing.

> *Increasing the size and complexity of the plan leads to a dramatic increase in the steepness of the curve of constraints evaluated.*

While this experiment is merely one data point, it suggests that our current algorithm is impacted severely by plan size, and it tends to support the claim by past program

[2] In general, the cost of the pre-filtering step is a constant times the total number of domain values for all plan components.

[3] For this experiment, implemented in Lisp on a Sparc1000 workstation, without any attempt at serious optimization, it took less than 30 seconds of CPU time to recognize all instances of a particular plan in our 5,000 line programs and less than 90 seconds of CPU time in our largest 10,000 line program.

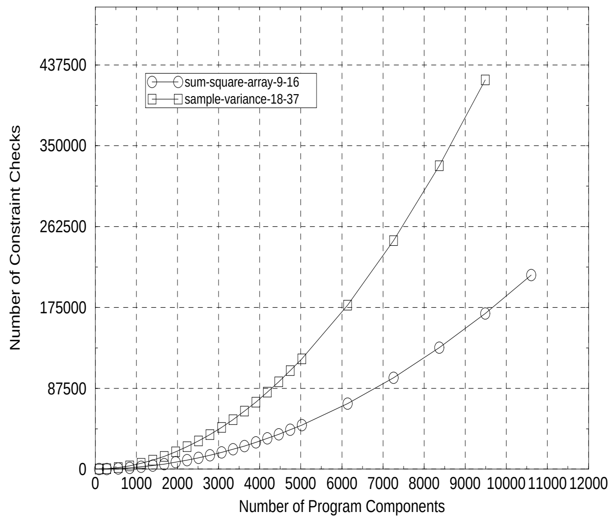

Figure 3: The results from an experiment with a larger plan and domain value pre-filtering.

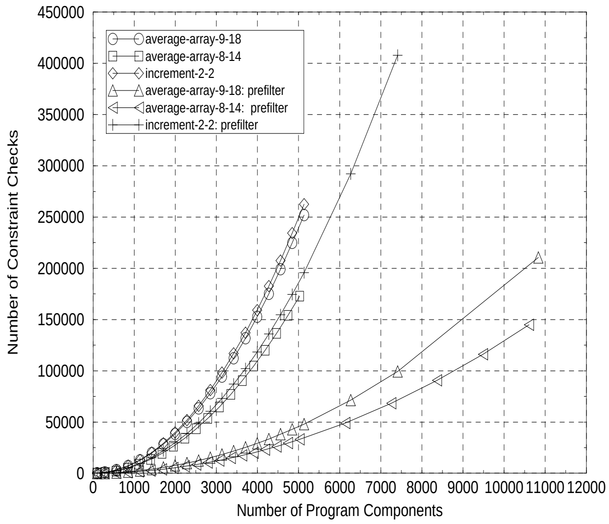

Figure 4: The results from an experiment involving hierarchical decomposition.

understanding researchers that performance suffers unless complex plans are hierarchically decomposed into smaller plans.

Given these results, we decided to perform an experiment to demonstrate how much improvement might be gained in breaking up a plan into subplans, recognizing those subplans, and then recognizing the combined entity. Our approach was to take our original array-averaging plan and do a simple decomposition of it into an increment plan (an assignment followed by an addition) and another, simpler array-averaging plan which is the original plan without the increment. We then determined the effort to recognize the increment, the effort to recognize this simpler plan, and the combined effort to recognize the original plan.

Figure 4 shows the results. This results were surprising.

> *More effort was spent recognizing the hierarchically organized plan structure than recognizing the original flattened plan.*

It turns out that recognizing the increment plan is more work for our constraint-based algorithm than recognizing the original array-averaging plan. This result is depressing, since the presumed way around the apparent combinatorial problems with large plans is to break them into pieces. However, not every hierarchical decomposition is going to be effective in reducing the search space and some may actually increase the amount of work being done.

Taking so much effort to recognize something so simple suggests that there is significant room for improvement in our constraint-based algorithm. Specifically, the increment problem is represented by two domain value sets, additions[4] $(P_1, \ldots, P_M)$ and assignments $(A_1, \ldots, A_N)$, with several constraints between them, including a data-dependency from P to A. Currently, in evaluating the data-dependency constraint between P and A, the constraint solver essentially forms the cross product of these sets and then applies the constraint to each of the resulting P_i, A_j pairs.[5] This effort, however, is almost entirely wasted, as the data-flow relationships have already been precomputed between various additions and assignments. For a given P_i, we already know exactly which of A's domain values depend upon it.

Unfortunately, it appears our current algorithm is spending an inordinate amount of time figuring out something it already knows. Given pre-computed data-dependencies, however, a more ideal plan matching algorithm would simply form the appropriate set of addition/assignment pairs that are related by this particular data-dependency.

5 An Experiment In Constraint Ordering

Our constraint-based plan matching algorithm uses FCDR, an algorithm which uses the size of the domain value sets to order the search. However, there are other techniques for ordering the search (e.g., ordering constraint evaluations by the probable effectiveness of the constraints). This led us to yet another experiment. This time

[4] Specifically, places in the program where we add one to some other value.

[5] We can view FCDR as a technique for trying to minimize the sizes of the sets of the cross products by forming cross products of the smaller domains first.

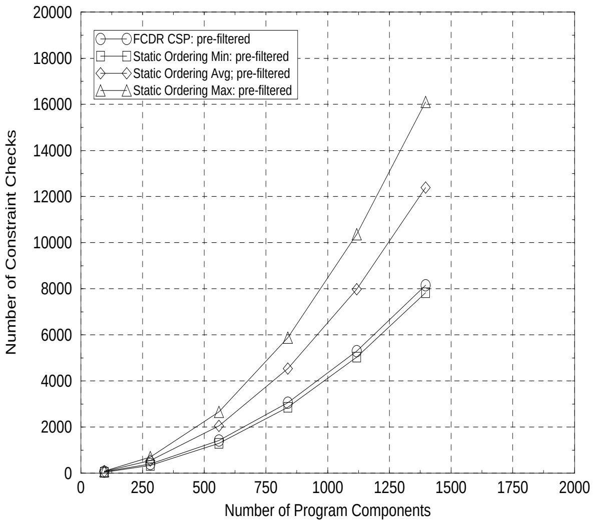

Figure 5: The results from trying all possible constraint orderings of an array traversal plan.

we took an array traversal plan containing five components and seven constraints, generated all possible orderings of the constraints, eliminated obviously inefficient constraint orderings, and then recorded the amount of work necessary to solve each of these problems [18].[6] The idea is to see if there exist constraint orderings that can perform better than the orderings dynamically determined by the FCDR approach.

Figure 5 shows the results. Because of the time-consuming nature of solving a large number of CSPs at each program size, we only considered programs of up to 1500 or so lines in length.

> *For plan matching, FCDR determines an ordering that is close to, but is not quite, the best possible ordering.*

Although the difference between FCDR's ordering and the best possible ordering appears to be small for these programs, it's increasing steadily as the programs get larger, so it is possible that there will be a significant difference for sizeable programs. In addition, this is merely a single data point, and it's possible that for some plans there could be a more significant difference. This suggests considering whether there are ways to improve the ordering in which the plan matcher evaluates constraints.

[6] Specifically, we generated 5616 different CSPs. For this plan, there are 7! (5040) possible constraint orderings, and for the first binary constraint, we can start by working through values in either of its domain value sets, doubling the number of different problems (to 10080). It then turns out that 4464 of these orderings lead to obviously inefficient (disjoint) search spaces.

6 An Improved Constraint-Based Algorithm And Some Experiments Using It

Our previous experiments suggest that our current two-phase constraint-based approach is not taking sufficient advantage of data-flow relationships and may not be quite as effective as it can be in terms of ordering constraints.

Constraint-based problems are solved using a combination of constraint-propagation and search. Constraint-propagation reduces the problem size by eliminating domain values that cannot possibly be part of an actual solution (e.g., our earlier pre-filtering heuristic). Once the domain sets have been reduced to some degree a search-based approach is typically used to identify the solutions.[7] During search, variables are instantiated (specific values are hypothesized for them), and the remainder of the uninstantiated variables can have their domains reduced according to partial application of constraints which exist between the instantiated and uninstantiated variables. Search algorithms try to minimize the number of constraints which must be checked to find one, some or all solutions. One approach is to perform perform limited or aggressive propagation or "lookahead" (e.g., in forward-checking, a just-instantiated value is used to reduce the range of any directly constrained uninstantiated variables). Given the additional information available as a result of static data-flow analysis, we can mimic this exact process, but with much greater effectiveness, and with no additional constraint checking.

Consider recognizing the increment plan. At some point during search, some particular addition within the program, P_i, is instantiated (or hypothesized) as a satisfying value for the variable P representing the plan's addition component. Suppose we then somehow determine that we are next interested in the variable A representing the assignment component. We can then directly observe from the data-flow analysis which specific assignments in the program depend upon or consume a value generated at P_i. This directly establishes A's domain, without checking the constraint against every domain value within A. Essentially, this reduces the branching factor of search by eliminating many "type A" values which could not have been part of a solution with Pi. Best of all, it requires no actual constraint checks during search, and the domain can now be further reduced through constraint propagation against other instantiated variable values.

Figure 6 provides the details of the algorithm. Essentially, it's a modified version of the standard forward-checking algorithm that obtains new candidate domain sets from the data-flow graph during search.

[7] Constraint-propagation may completely solve a problem, but this is usually not efficient and the more effective amount of pre-search reduction is believed to be problem-dependent.

1. **[Domain Initialization]** For each variable $X \in V$ (the set of CSP variables), find $Dom(X)$ (the set of domain values for X). *The domain values of a plan component X is the set all program components of the same type as X.*
2. **[Constraint Propagation]** Reduce $Dom(X)$ by constraint propagation. *Use pre-filtering to eliminate program components that could not possibly participate in a solution.*
3. **[Solution Initialization]** Set the solution set to empty.
4. **[Initial Variable Selection]** Select and remove a variable X from V. *Prefer X to be the variable with the smallest domain that is involved in a data-dependency constraint.*
5. **[Value Selection]** Select and remove a value of X from $Dom(X)$ and apply any applicable constraints involving already instantiated variables. *Instantiate a plan component with a program component and evaluate any applicable inter-component constraints.*
6. **[Backtrack Point Selection]** Backtrack if any $Dom(X)$ in V becomes empty.
7. **[Solution Evaluation]** If V is empty, output Solution. *Each set of instantiated, consistent variables is a recognized plan instance.*
8. **[Next Variable Selection]** Select and remove a variable Y from V. *Select a Y that has a data-dependency constraint with X where X is a previously instantiated variable and the current instantiated value of X (X_i) has a data-flow relationship with the fewest domain value candidates in Y.*
9. **[Next Domain Construction]** Determine the domain $Dom(Y)$ for Y using our modified form of forward checking. *Form $Dom(Y)$ by following the data-flow linkages from X_i in the data-flow graph to any consumers with Y's type. $Dom(Y)$ is set to those consumers.*
10. Goto Step 5.

Figure 6: Our new constraint-solving algorithm.

To determine how well this new constraint solving algorithm performs, we repeated some of our earlier experiments. Figure 7 shows the results of using this new algorithm to recognize instances of our original array averaging plan. These results are extremely encouraging.

For our initial plan, the revised algorithm has essentially linear performance on up to 10,000 line programs.

In fact, recognizing all instances of our example plan is only requiring the evaluation of approximately one constraint for every three to four lines of code, and this rate shows no signs of increasing significantly with program size.

We then turned our attention to scaling. Figure 8 shows the results of using this new algorithm to recognize instances of our larger statistical variance plan.

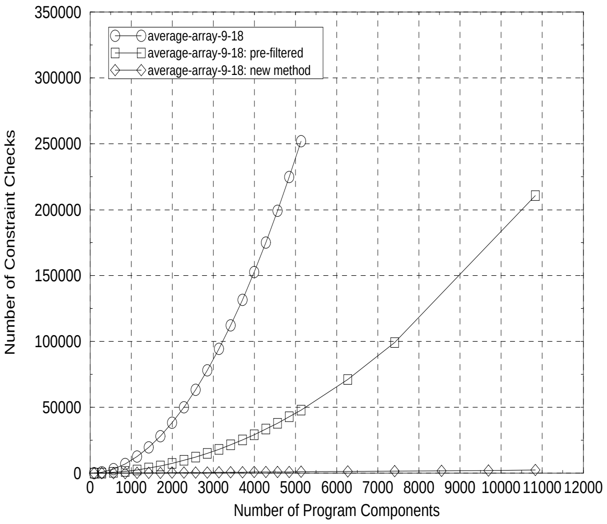

Figure 7: The results of applying our new algorithm on our original plan.

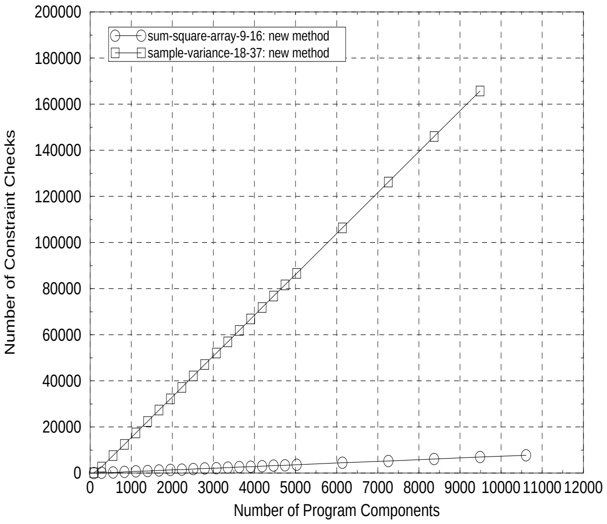

Figure 8: The results of using our new algorithm on a larger plan.

Even with our new algorithm, increasing the plan size still leads to a significant increase in the number of constraints evaluated.

The curve, however, is again essentially linear, although it has a significantly steeper slope. It appears as though doubling the plan size led to a roughly sixteen-fold increase in terms of the number of constraints evaluated. Still, despite this increase, it is doing significantly less work than our previous pre-filtered algorithm.

Fiinally, we wanted to confirm that our modified al-

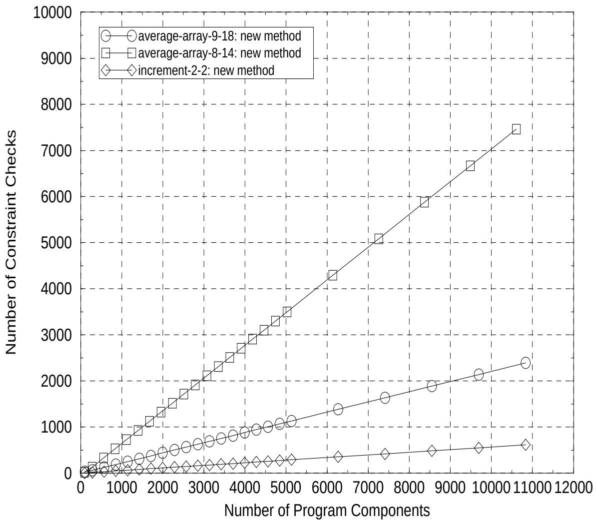

Figure 9: The results of using our new algorithm on the increment plan.

gorithm successfully addresses the problem we identified with how the original constraint-based algorithm handled the increment plan. Figure 9 shows the results of using it to repeat the experiment in Figure 4. There is a dramatic difference in performance as a result of eliminating unnecessary evaluations of data flow constraints.

Our new algorithm now expends significantly less work recognizing simple plans.

As expected, increment is now very cheap to recognize. However, perhaps surprisingly, the array-averaging plan that uses increment takes more effort to recognize than the array-averaging plan that uses assign and plus directly. This behavior is strange since it is a smaller and less complex plan, and it requires further experimentation to understand the cause. In any case, however, both of these plans can now be recognized in linear time and with far fewer constraint evaluations than before.

7 Our Future Experiments

Our initial results are extremely promising, however, they're based on a small set of experiments with a collection of artificially generated C programs and a small group of carefully constructed C programs. In addition, we have made a variety of simplifying assumptions in terms of the language constructs that appear in the programs we understand, the overall structure of these programs, and the type of control and data flow information that's available. It's an open question whether these assumptions, which have arisen from deficiencies in our home-grown control and data flow analysis tools, change the empirical performance of our algorithm. As a result, it's necessary for us to obtain more powerful analysis tools and use them to explore our constraint satisfaction engine's performance on searching for individual plans in real world C code, such as the X-Windows system or Mosaic.

In addition, our experiments have just begun exploring the scaling properties of our revised constraint-based understanding algorithm in terms of plan size. Program understanding has been shown to be worst-case exponential in terms of plan size [16], and there is a significant increase in the numbers of constraints evaluated in our initial scaling experiments plans, even with our new algorithm. However, the curve of constraints evaluated is still lineara and our experiments are preliminary. In addition, there is reason to believe that large plans may not be significantly harder to locate than small plans. In particular, the data flow constraints appear to be very effective in narrowing down the search space. One way to address this question is to continue searching our existing test programs for plans of increasing size and complexity.

Our initial experiments have also focused solely on plan matching. We haven't experimented with program understanding as a whole to determine the total cost to understand a non-trivial program, nor have we searched for a wide range of different plans within a single program. As a result, it's necessary for us to perform experiments using a complete hierarchically driven understanding algorithm (such as hierarchical MAP-CSP [9] or PU-CSP [14]) on real-world programs with a real-world plan library. Our likely experiment will be to take a hierarchical library of "Year 2000" plans and a set of real-world COBOL programs and explore the performance of the algorithm in recognizing these plans.

8 Related Work

This paper and our previous efforts [9, 15] are not the only empirical studies of program understanding algorithms. Many of the previous studies, however, have been devoted to non-efficiency issues, such as plan library completeness for plan instances on a group of similarly sized programs for performing a particular task [3, 5]. When earlier 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 [6, 13]. The actual performance of various plan recognizers, however, has peviously been limited to anecdotal discussions. Our work appears to be unique in its focus on experiments that enable us to draw conclusions about how well these plan recognition algorithms scale.

9 Conclusion

Despite program understanding having been shown to be NP-hard [16], our experimental results strongly suggests that plan matching is potentially quite tractable. For the small plans we've tried, we have demonstrated the scalability of our newly developed constraint-based algorithm in programs of up to 10,000 lines. When this is combined with work done in semi-automatically modularizing programs (in which source files as large as 850,000 lines were broken into modules in the 10,000 to 20,000 line range [7]), we are clearly at the point where we can apply our plan matcher to modules of real-world systems. In addition, the linear appearance of the curve of evaluated constraints leads us to believe that for small plans, at least, our algorithm is scalable to much larger programs.

While our results are not complete or definitive, they are certainly encouraging, and they provide additional support for the belief that program understanding is a worthwhile and not hopeless endeavor (as opposed to the opinions expressed in [11]). Rather than giving up, in fact, our results suggest that we are very near—if not at—the point where we can begin to apply program plan recognition to real-world legacy systems.

Others [4, 13] 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. Our empirically driven approach is proving them right. Not only has it led to significant improvements in our algorithm's performance, but it also provided some evidence that suggests we are successfully moving program understanding along the road from "toy" programs and to real legacy systems.

References

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

[2] J. Hartman. *Automatic Control Understanding For Natural Programs*. Research report UT-AI-TR-91-161, University of Texas at Austin, 1991.

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

[4] 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.

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

[6] W. Kozaczynski and J. Q. Ning. Automated Program Understanding By Concept Recognition. *Automated Software Engineering*, **1**:61–78.

[7] P. Newcomb and L. Markosian. Automating The Modularization Of Large Cobol Programs: Applications Of An Enabling Technology for Reengineering. In *Proceedings of the Working Conference on Reverse Engineering*, pp. 222–230, 1993.

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

[9] A. Quilici and S. Woods. Toward A Constraint-Satisfaction Framework for Evaluating Program Understanding Algorithms, *Journal of Automated Software Engineering*, **4**(3), 1997.

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

[11] B. Weide. Why Program Understanding Is Intractable. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, 1995.

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

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

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

[15] S. Woods and A. Quilici. Some Experiments Toward Understanding How Program Plan Recognition Algorithms Scale. In *Proceedings of the Third Working Conference on Reverse Engineering (WCRE)*, Monterey, CA, November 1996.

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

[17] S. Woods and Q. Yang. Program Understanding As Constraint Satisfaction: Representation and Reasoning Techniques. *Journal of Automated Software Engineering*, 1997 (to appear).

[18] Zhang, Y. 1997. *Scalability Experiments In Applying Constraint-Based Program Understanding Algorithms to Real-World Programs.* Masters Thesis, Department of Electrical Engineering, University of Hawaii at Manoa, Honolulu, Hawaii.