Using Graph Matching: Program Recognition of the Selection Sort Algorithm

Ronald Finkbine, Ph.D.
Indiana University Southeast
New Albany, Indiana 47250
rfinkbin@ius.edu

Abstract

The field of *program understanding* attempts to determine the function of a code segment without programmer intervention and for this to occur, it is necessary to have a model (*plan*) against which to attempt to match the code segment of interest. This paper traces in detail the pattern recognition of the selection sort algorithm.

Introduction

The purpose of this research is to develop a general-purpose algorithm recognition system, capable of recognizing any well-defined and well-written algorithm. This project uses *plans* (Wills 1992) to recognize common forms (code segments) within existing software in an attempt to gain knowledge about a legacy system (Sartipi 2003) from its source code (Biggerstaff 1990) by matching against a large defined set of common algorithms, rather than attempting to deduce what a code segment performs from its specific dataflow (Rugaber et. al. 1990). This research uses an intermediate representation of an abstract syntax tree (AST), standard in compiler toolsets. This AST representation is output as a flattened tree into a fact list, which is the operation underpinning of expert systems.

Targeted Problems

This research concentrates on design recovery from legacy software, written in older languages and with fewer techniques applicable to modern software development. This is because that recently written software is often written in a more modern language, but this leaves a large bulk of older, operational software, orphaned to endless software maintenance until it is rewritten.

Legacy software, in general, exhibits a number of the following problems: 1) parameter identification, 2) identifying code segments that are replaceable by calls to commercial libraries (such as IMSL), 3) removing duplicate code to user library, 4) separation of intertwined components, and 5) combining disparate codes into single equations. Each of these problems increases the difficulty in a software maintenance programming attempting to understand a software component. Graph matching is considered one of the most complex problems in computing (Bienenstock 1987).

The first problem, *parameter identification*, is the most simple. It involves searching the source code for variables that are assigned values within assignment statements (no reads) one time. Any usage, thereafter, is only on the right-hand side of assignment statements and is a reference to the variable, not a modification to the variable. Therefore, these types of variables, or constants, can be identified by the parameter statement which indicates their true usage.

The second problem, *plan recognition*, is comprised of identifying code segments that are replaceable by calls to commercial libraries (such as IMSL). This will involve detecting codes similar to those used within commercial libraries.

The third problem, *duplicate removal*, consists of detecting and removing duplicate code to the user's library. This allows the user to designate a section of code as common and to look through their remaining programs searching for codes that are copies of the target.

The fourth problem, *algorithm separation*, involves detection/separation of overlapping algorithms within the same section of code. In Figure 1 it can be seen that there are two initializations of arrays occurring within the same do-loop. This is good for optimizing computer resources, but not for optimizing the programmers' time for understanding and maintaining a program.

The fifth problem, *algorithm aggregation*, involves combining disparate codes into single equations. As displayed in Figure 2, an equation 1) can be coded in multiple ways. Though the computations are equivalent,

```
      DO 10 I = 1, N
        A(I) = 0
        B(I) = 0
10    CONTINUE
```

Figure 1: Intertwined Algorithms

Table 1: Selection Sort Algorithm Rule Flow

Firing set	Prefix	Rules
First	Defines	00
	General	00
	Expressions	00, 02, 07, 09, 10, 11, 17
	Structures	00
	Evaluations	01, 02
Second	Variables	00, 02, 04, 05, 06, 07, 08
	swaps	00, 01
Third	loop	00, 01, 02, 03
	min	00, 01
Complete	SSA	00, 01

the recognition of them must take these variations into account.

This paper describes a portion of the High-Level Algorithm Recognizer (HLAR) project (Finkbine1994), which recognizes three algorithms selection sort (SSA), quick sort and heap sort from four languages, C, Scheme, Postscript and COBOL. This research is unique in that it recognizes algorithms of significant size (currently 50 lines of code), and detects these algorithms directly from multiple third-generation programming languages instead of from one language or directly from an intermediate form (Ning 1989).

The first step in recognizing common algorithms is to compile the input source program into an intermediate representation (Seemann 1998). The bulk of the recognition efforts will be made by CLIPS, a rule-based forward-chaining expert system, therefore the source programs will be expressed as a facts list (a tree represented as a linked list). Once the CLIPS system is initialized and pattern recognition begins, the general flow of the pattern recognition process is listed in Table 1.

This first phase is the initial fact generation. In the following discussion, rule names are listed in separate phases (or firing sets). This is necessary since in a rule-based expert system, rules can fire at the time their conditional elements are satisfied. Within each phase, the rules can (and do) fire in an order determined within the expert system itself, not the order they are introduced into the expert system or the order in which they are listed within the system. A rule set once started will continue to fire until all rules have attempted to fire one final time with none successful. This procedure allows each rule to fire as many times as possible, only halting when all have been unsuccessful during the final pass.

SSA Recognition Trace

One of the algorithms currently recognized is the Selection Sort Algorithm (SSA). Figure 2 is a depiction of the component parts, also known as *plans* (or *sub-plans*), within the SSA. The *minimization plan* and the *swap plan* are contained, respectively, with the *ssort plan*. As well as proper containment and ordering of the *sub-plans* in this Figure, it is necessary that the *plans* have identifiers (variables) in common. For example, for this function to perform correctly, it is necessary that the indexing variable of the containing *for* loop be one of the positions of the data structure with the *swap plan*. These additional requirements are necessary for the proper execution of the algorithm and its subsequent recognition.

This section details the recognition of the SSA within HLAR. This algorithm was chosen because it is a common algorithm within computing literature and the computer programming community. It has a complex plan structure, providing enough challenge to be of interest within the program understanding/re-engineering community. Figure 2 depicts the general flow of the recognition process necessary for the SSA.

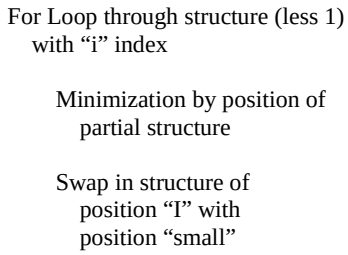

Figure 2: SSA Plans

Initial Facts

After a third generation source language program is translated into the intermediate form, a program will traverse it and generate a list of facts that are input into the HLAR system. The structure and purpose of these facts are further explained in the remainder of this section. In general, a statement will become a series of facts, roughly equivalent to tokens in traditional compiler technology.

Initial Rules

There are two rules that fire first due to their salience value regardless of the subject program being examined. Rule *gen_00* fires and establishes the number of the maximum *generalNode used.* Next, the *def_00* rule fires, establishing the *last-general-node* field of every *defineRoutineNode*, thus establishing the span of control of each routine. This is not possible in a one-pass translator, such as is used to generate the fact intermediate form from the standard intermediate form, which is necessary for input into a rule-based expert system. In the case of multiple routines within a program, the control of each routine extends to the beginning of the next routine. The control of the last routine extends to the last *generalNode*, the value calculated in the *gen_00* rule.

For the SSA, the annotated explanation of the recognition process appears in this section. Figure 4 contains the intermediate form code used for explanation of the recognition. In general, processing takes place from the lowest level (tokens and expressions) to higher levels (loops and if statements). Table 1 displays the general rule flow in the SSA recognition. To support the passing of information form rule within HLAR, it is necessary to have a set of abstract data types known as templates.

First Rule Firing Set

After the facts generated from the SSA program are input, the CLIPS system reviews these facts and attempts to satisfy the requirements for each of the rules within the HLAR system. This section describes the rules that are fired and the facts they modify by retracting, asserting or leaving them alone. Each of the statements referenced in this section is from Figure 3, and the rule firing is summarized in Table 2. In general, rules within a firing set can fire in any order; however, some rules in this first firing set generate data that fire other rules. This rule firing set recognizes three categories of rules: expressions, structures and evaluation clauses (used to determine the path of execution within an *if- statement* and the *exit* of a loop).

Figure 4 is included to display the intermediate code, produced by a C language parser, which is being recognized. The general layout of a program is a sequence of global variable definitions and assignments followed by sub-program definitions. In general, intermediate form statements use prefix notation and each statement is a function call followed by the parameters passed to or returned from the function.

```
[1]  (define-routine sort
[2]  (parameters (inout numbers) (in count)
[9]  (assign i  0)
[10] (loop
[11] (eval (gt I (minus count 1)))
[12] (assign big I)
[13] (assign j (plus i 1))
[14] (loop
[15] (eval (gt j count))
[16] (if
[17] (eval (gt (select numbers (key j) (field))
[18] (select numbers (key big) (field)))
[19] (assign big j))
[20] (assign j (plus j 1)))
[21] (assign temp (select numbers (key big) field)))
[22] (assign (select numbers (key big) (field))
[23] (select numbers (key I) (field))
[24] (assign (select numbers (key I) (field)) temp)
[25] (assign I (plus I 1))))
```

Figure 4: SSA Intermediate Form

Table 2: SSA Firing Phase One

Rule	Rule name	Statements
exp_00	*detect_exp_0*	9
exp_02	*detect_exp_plus_id_1*	13, 20, 25
exp_07	*detect_exp_gt_id_id*	15
exp_09	*detect_exp_gtlop_id_min_id_1*	11
exp_10	*detect_exp_id*	12, 17-19, 21-24
exp_11	*detect_exp_gt_strucid_strucid*	17, 18
exp_17	*detect_exp_minus_id_1*	11
struc_00	*detect_strucref_id*	17, 18, 21, 22, 23, 24
eval_01	*detect_gt_strucid_strucid*	17, 18
eval_02	*detect_gt_id_id*	15

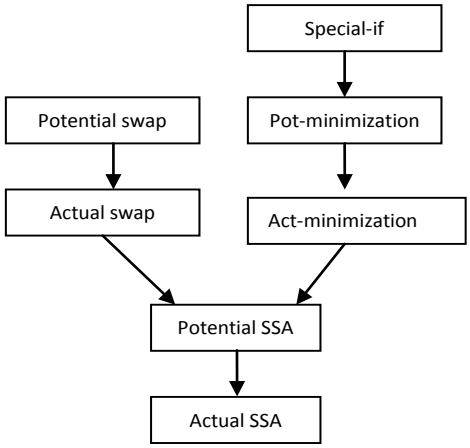

Figure 3: SSA CLIPS Template Flow

Expressions are recognized within the first firing set. They are the least-common denominator of program understanding and appear on the right-hand side of assignment statements and as the single operand of *evaluation* statements. Their variety (such as $x = x + 1$ versus $x = 1 + x$) lead to the increased complexity of pattern-matching algorithms.

The rule *exp_00* detects an expression of zero in statement 9. Rule *exp_02* detects an expression of an identifier plus one in statements 13, 20, and 25. Rule *exp_07* detects a Boolean expression of an identifier greater-than another identifier in statement 15. Rule *exp_17* fires on statement 11 followed by rule *exp_09*, which detects a complex expression in statement 11. Rule *eval_02* detects a comparison of simple identifiers in statement 15.

Rule *exp_10* detects an expression of a simple identifier in statements 12, 17, 18, 19, 21, 22, 23, and 24. This identification of simple identifiers (variable references) produces output that is part of the conditional input of rule *struc_00*, which detects an array referenced by a simple identifier in statements 17, 18, 21, 22, 23 and 24. Also, rule *eval_01* fires, recognizing the comparison of two positions of the same array with the greater-than operator.

Second Ruling Firing Set

Due to the extensive rule firing that occurs in this set, the rule execution is displayed in Table 3. Rule *var_00* detects an identifier occurring on both the left-hand- and right-hand-side of an incrementing assignment in statements 21 and 25. Rule *var_04* recognizes a somewhat similar $x = y + 1$ statement in 13. Rule *var_05* recognizes a simple save-value assignment statement in 12 and 19.

Rule *var_06* recognizes the save-value of an array position in statement 21. Rule *var_07* recognizes a save between two locations of the same array, and rule *var_08* recognizes an assignment from a simple identifier to an array position. These three assignment statement rules (*var_06, var_07* and *var_08*) produce input to rule *swap_00*. Swap rule *swap_00* fires on statements 21, 22, 23 and 24. This rule sets a controlling condition that has prevented the *not- rules* from firing. After the assertion, an interfering statement within the *swap* segment would be detected, if one existed. And since there are no interfering statements rule *swap_01* fires successfully.

Table 4: Algorithm rule firings

Algorithm	Rule Firings
Selection Sort	50
Quick Sort	75
Heap Sort	150

Third Rule Firing Set

Loop rule *loop_00* fires on statements 9, 10, 11, 5, recognizing the index variable initialization, increment, and testing. This is followed by rule *loop_01* firing on these same statements since there is no interference with the index variable within the *loop* statement and all statements are within the same routine. Loop rule *loop_02* fires on statements 13, 14, 15, 20, recognizing the index variable initialization by an expression, increment and testing. This is followed by loop rule *loop_03* which fires on statements 13, 14, 15, 20 since there is no interference of the index variable within the *loop* statement and all statements are within the same routine.

Minimum rule *min_00* fires on statements 13 through 20, recognizing the form of a degenerative minimization by position. This is followed by rule *min_01*, which fires on statements 13 through 20, which verifies the statements have the correct ordering, non-interference of variables and proper containment.

The variable *count* was established as the variable that contains the initialized length or the number of structure positions with actual values that are not undefined. This must be input by the user and not performed automatically by HLAR. In the future, it will be part of the system, but would involve recognition of additional algorithms that are not currently part of this research.

Completion of Rule Firing

SSA recognition rule *potential_ssort_00* recognizes the form of a *containing-loop*, a *contained-degenerative-minimization-plan* and a *swap plan* with all identifiers matching appropriately. This triggers a search for interfering statements that will hopefully find no reason to terminate the search. An example of which would be an intervening statement that sets the loop-indexing variable to zero (illegal in Pascal, legal in C).

The SSA recognition rule, *ssort_01* then will fire due to the correct statement ordering, proper containment and non-interference.

Summary

The HLAR system currently recognizes the three algorithms (written in the C programming language) in the number of rule firings listed in Table 4. In addition, it has recognized the SSA in the COBOL, Scheme and Postscript programming languages. Currently, this project is being redesigned which will involve a platform change in order to build a more appropriate GUI as well as to be able to distribute the recognition tasks across a network.

To limit the need for outside assistance from a programmer (Ning 1989), the HLAR system has been designed (and redesigned) to accept multiple forms of algorithms. Future work includes development of a subsystem to construct the *plans* by compiling from source and to not have the recognition rules written expressly by a programmer.

References

Biggerstaff, Ted, 1990. Design Recovery for Maintenance and Reuse, IEEE Computer: July.

Bauer, D., S. L. Hakimi, and E. Schmeichel, 1990. Recognizing Tough Graphs is NP-Hard: Discrete Applied Mathematics:28, 191.195.

Bienenstock, E and von der Malsburg, C.1987. A neural network for invariant pattern recognition, Europhysics Letters: 4 121-126.

Finkbine, Ronald 1994. B., Recognition of High-Level Algorithms, Ph.D. Dissertation, Department of Computer Science, New Mexico Institute of Mining and Technology.

Ning, Jim Qun 1989. A Knowledge-Based Approach to Program Analysis, Ph.D. Dissertation, Department of Computer Science, University of Illinois at Urbana-

Table 3: SSA Firing SetTwo

Rule	Rule name	Stmt
var_00	*detect_assign_sca_inc_1_self*	21, 25
var_02	*detect_assign_sca_array_base_c*	9
var_04	*detect_assign_sca_inc_1_other*	13
var_05	*detect_assign_sca_sca*	12, 19
var_06	*detect_assign_sca_struid*	21
var_07	*detect_assign_strucid_strucid*	22, 23
var_08	*detect_assign_struciddd_sca*	24
swap_00	*detect_potential_id_swap*	21-24
swap_01	*detect_actual_id_id_swap*	21-24

Champaign.

Rugaber, Spencer, Stephen B. Ornburn, and Richard LeBlanc, Jr., 1990. Recognizing Design Decision in Programs, IEEE Computer, July.

Sartipi, Kamran and Kontongiannis, Kostas 2003. On modeling software architechture recovery as graph matching. Proceedings of International Conference on Software Maintenance, September.

Seemann, Jochen and von Gudenberg, Jurgen 1998. Pattern-Based Design Recovery of Java Software, Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 10-16.

Wills, Linda 1992. Automated Program Recognition by Graph Parsing, Ph.D. Dissertation, Artificial Intelligence Laboratory MIT, July.