Sim: A Utility For Detecting Similarity in Computer Programs

David Gitchell & Nicholas Tran
Department of Computer Science
Wichita State University
Wichita, KS 67060-0083
{ddgitche,tran}@cs.twsu.edu

Abstract

We describe the design and implementation of a program called `sim` to measure similarity between two C computer programs. It is useful for detecting plagiarism among a large set of homework programs. This software is part of a project to construct tools to assist the teaching of computer science.

1 Introduction

Computer science education began with the establishment of the first CS Departments more than thirty years ago, and yet its practitioners have lagged behind their colleagues in other CS subfields in developing software tools for their trade. Today there is an impressive array of software systems at the disposal of software designers, circuit designers, network administrators, numerical analysts, and digital artists. In contrast, computer science educators are still relying on traditional tools and techniques: their main tool for communicating ideas, only recently supplemented by course webpages, is still chalk and blackboard, and their main tool for evaluation is still a human grader.

We believe that this dearth of software tools is not due to a lack of diligence from CS educators but rather to the absence of a mature theory in the field. Research on computational learning theory [15], which investigates the limitations and costs of different strategies of learning, began only in the mid 1980's. Similarly, research on program checking, which measures correctness of programs by examining their *behavior* [3, 4, 5], began only in the late 1980's. Both of these developing areas are generating deep and surprising results, which will certainly lead to improvements in the practice of computer science education in the near future.

SIGCSE '99 3/99 New Orleans, LA, USA

Yet at this point not all software development in CS education require further foundational work. As an example, look at the task of evaluation of assignments and exams in lower-level programming courses. Such courses tend to have high enrollment and a large number of programming assignments with simple solutions, which are to be evaluated on not only their correctness but also their style and uniqueness. A software tool for this task would need to examine the programs' *structure* and would benefit from the already well-developed theory of string algorithms.

In this paper we describe the design and implementation of a program called `sim` that measures structural similarity between two C computer programs, a task that underlies the evaluation of correctness, style, and uniqueness. This program is a direct application of string alignment techniques that were recently developed to detect similarity between DNA strings [6, 12] (see also [7, 13, 14]). Given two programs, `sim` first reduces them to their parse trees using a standard lexical analyzer. Viewing the parse trees as strings, `sim` then aligns them by inserting spaces in each to obtain a maximal common subsequence of tokens. A score between 0.0 and 1.0 is reported as the degree of similarity between the two programs. In its current form, `sim` runs in time $O(s^2)$, where s is the maximum size of the parse trees. The bulk of `sim` is implemented in C++, and its graphical user interface is implemented in Tcl/Tk.

As a first application, we used `sim` to detect plagiarism. Experiments on real data showed that `sim` is resistant to extensive name changes, reordering of statements and functions, and adding and removing white spaces and comments. `sim` is reasonably fast on small-sized programs: comparisons between all pairs of 56 programs with an average length of 3415 bytes take about three and a half minutes.

The rest of this paper is organized as follows. Section 2 explains the underlying string alignment algorithm used by `sim`, Section 3 describes the design and implementation of `sim`, Section 4 presents the experimental setup and results, and Section 5 discusses future improvements.

We conclude this section with a review of previous works on detecting structural similarity in programs. The Unix `diff` command [8, 9] also uses string alignment methods to detect similarity between two programs, but its basic textual unit is a line instead of a parse tree token as in `sim`; `diff` cannot detect systematic name changes or textual reordering of modules. Baker's dup program [2] reports all maximal exact

matches over a threshold length between two programs; it can detect systematic (parameterized) name changes and reordering of large modules, but it is ineffective when spurious statements are inserted, or when a small block of statements is reordered. Aiken's `moss` program [1], developed at UC Berkeley for plagiarism detection, also examines program structure, but its underlying algorithm for comparison has not been made public. Other tools designed specifically for plagiarism detection such as [10, 11] are based on heuristics that measure statistics on frequencies of words and symbols, and thus may have a high rate of false positives.

2 String Alignment

This section explains the string alignment algorithm used by `sim` to compare two programs. An alignment of two strings s and t (of possibly different lengths) is obtained by inserting spaces in the strings so that their lengths become the same. Note that there are many possible alignments. For example, two alignments of the strings "masters" and "stars" are

```
masters   masters
sta  rs     stars
```

Consider the pairs of characters given by the alignment: a match scores m, a mismatch scores d, and a gap scores g, where m, d, and g are some chosen values. The score of a block of pairs is the sum of the individual scores, and the score of a highest-scoring block of pairs gives the score of an alignment. For the above example, if $m = 1$, $d = -1$, and $g = -2$, then "rs/rs" and "sters/stars" are the highest scoring blocks in the first and second alignments respectively. The alignment scores are then 2 and 3. Alignment scores are related to edit distances and are used to measure similarity between two almost identical objects. Alignment is well suited for inexact matching and is used extensively in computational biology to detect relationships between DNA strands [6, 12]

The optimal alignment score between two strings is the maximum score among all alignments. This value can be computed using dynamic programming. Formally, given two strings s and t, define $D(i,j)$ to be the optimal alignment score between the two substrings $s[1..i]$ and $t[1..j]$. $\max_{1 \le i \le |s|, 1 \le j \le |t|} D(i,j)$ is the value we are looking for. Define $score(s[i], t[j]) = \begin{cases} m, \text{if } s[i] = t[j], \\ d, \text{otherwise.} \end{cases}$. The following recurrence relation gives us a method to compute the solution:

$$D(i,j) = \max \begin{cases} D(i-1, j-1) + score(s[i], t[j]), \\ D(i-1, j) + g, \\ D(i, j-1) + g, \\ 0 \end{cases}$$

The boundary conditions are given by $D(1,i) = i * g$ and $D(j,1) = j * g$. The elements of the matrix D can be computed by initializing the first row and column with the boundary conditions and then evaluating the elements from left to right and top to bottom. This is possible since the value of $D(i,j)$ depends only on $D(i-1, j-1)$, $D(i-1, j)$, and $D(i, j-1)$. The running time of evaluating the best alignment is $O(|s||t|)$. The space requirement is $O(\max(|s|, |t|))$, since only two rows are needed by the computation at any time.

3 Design and Implementation

`sim` was implemented in 1780 lines of C++ and 393 lines of Tcl/Tk. A block diagram of the design of `sim` appears in Figure 1.

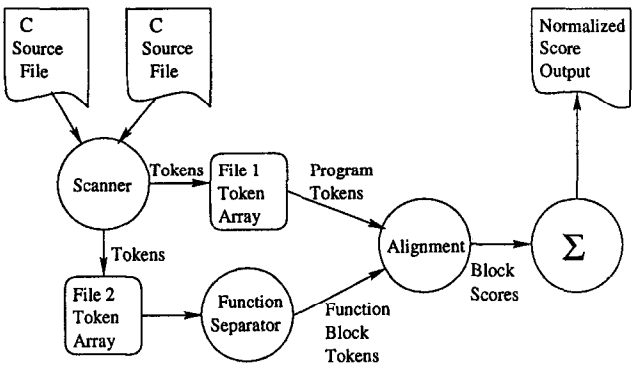

Figure 1: *Block diagram of* `sim`*'s design.*

Each input C program is first passed through a lexical analyzer to produce a compact form of its structure in terms of a stream of integers called *tokens*. The lexical analyzer is generated automatically by the Unix command `flex` given an appropriate subset of the C grammar, so `sim` can be easily modified to work with programs in other languages. Each token represents either an arithmetic or logical operation, a punctuation symbol, a C macro, a keyword, a numeric or string constant, a comment, or an identifier. For example, the statement

```
for (i = 0; i < max; i++)
```

would be replaced by the token stream

```
TKN_FOR TKN_LPAREN TKN_ID_I TKN_EQUALS TKN_ZERO ...
```

Token numbers for keywords and special symbols are predefined. Those for identifiers are assigned dynamically with the use of a symbol table (shared by both programs), such that two occurrences of a variable name are replaced by two occurrences of some integer. Each comment, no matter how long, is replaced by a fixed token, and white spaces are discarded. The purposes of this tokenizing process are to reduce a program to its parse tree, which is usually much shorter, and to remove inessential information such as white spaces and comments before the comparison is performed.

After the two source programs have been tokenized, the token stream of the second program is divided into sections, each representing a module of the original program. Each such module is then aligned with the token stream of the first program separately. This technique allows `sim` to detect

similarity even in the case when positions of the modules in a program are permuted. The actual alignment is scored using the following scheme:

- a match involving two identifier tokens scores 2; other matches score 1;
- a gap scores -2
- a mismatch involving two identifiers scores 0; other mismatches score -2.

The rationale for this scheme is clear: a match of identifiers is considered extraordinary and thus scores the highest, whereas a cosmetic mismatch between two identifiers is considered less significant than a structural mismatch between an identifier and an operator. The total alignment score is computed from the individual score for each block and then normalized to a number between 0.0 and 1.0 by dividing it by the sum of the alignment scores of the first and second program against themselves, i.e.

$$s = \frac{2 \times score(p_1, p_2)}{score(p_1, p_1) + score(p_2, p_2)}.$$

A Tk/Tcl graphical user interface is provided to allow comparison of a reference file against a collection of files and display and printing of the results in the form of a bar graph. Figures 2 and 3 show examples of an output screen and a file selection box. The bars are colored red, yellow, and green in decreasing similarity to the reference program according to thresholds specified by the user. A separate interface to display the results using GNUPlot is also provided.

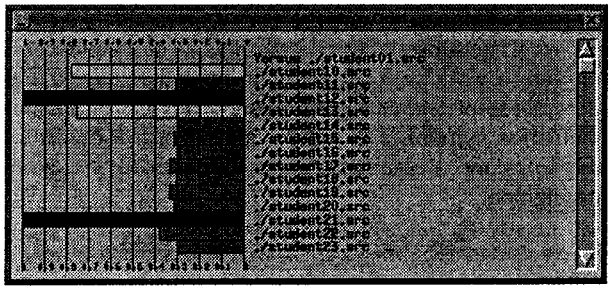

Figure 2: Displaying the results.

4 Experimental Setup and Result

As a simple experiment, we took a small C program and changed all variable and function names, removed comments, inverted the order of adjacent statements whenever possible, and permuted the order of the functions. The programs are shown in Figure 4 and 5. On this example, `sim` reported a score of 0.4, which is usually significant enough to warrant a closer examination by the grader.

To interpret the score values, we found it more reliable to look at the graphs of all the `sim` scores for a group of

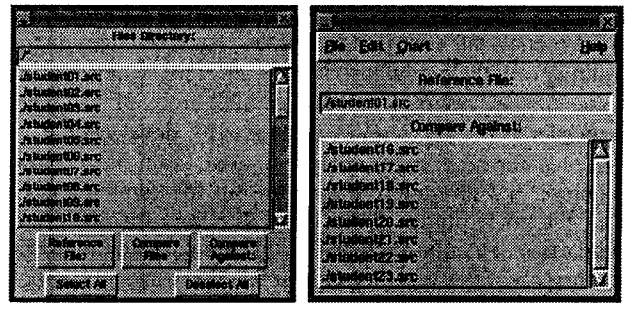

Figure 3: File selection.

programs than to use a set of fixed threshold values. To illustrate this point, we tested `sim` on a group of 36 actual homework programs submitted in a lower-level computer science course recently offered at our department. Each program was used as the reference program to be compared against the rest, and the scores were tabulated in a bar graph. The result from one such set of comparisons was shown in Figure 6.

In this graph, the leftmost bar denotes the score of the reference program against itself, which is 1.0. The two rightmost programs (37 and 38) were modifications to the reference program by one of the authors in an attempt to reduce the score by removing or moving comments, rearranging blocks, changing names, and adding redundant '{'-'}' pairs. As expected, the five programs that scored outstandingly had been determined earlier by the instructor to be modified copies of the reference program.

The statistics for the whole group concerning file sizes, token array sizes, and running time of the 630 comparisons in total are shown in Figure 7. Program sizes are in bytes, and times are in seconds. The platform used was a Pentium Pro 200Mhz workstation running Debian/GNU Linux 1.3.

We repeated the test for a larger group of homework programs collected from a different lower-level course, in which the instructor had found no plagiarism. A typical set of comparisons against one reference program is shown in Figure 8, and the statistics for the whole group is shown in Figure 9.

5 Discussion

We have designed and implemented a software tool for measuring similarity between two C programs, which can be used to detect plagiarism in programming homework in lower-level computer science courses. We have found that our tool was robust to common modifications such as name changes, reordering of statements and functions, and adding/removing comments and white spaces.

Much work remains to be done regarding building tools to assist in evaluating homework assignments. As

```c
/* Original copy */

#include <stdio.h>
#include <time.h>

#define SIZE 10

/* select a 'sample_size' random sample from 'size' elements */

void sample(int a[], int size, int sample_size)
{
        int i, j;

        if (sample_size > size || sample_size < 0)
                return;

        for (i = 0; i < size; ++i)
        {
                j = random() % (size -i);
                if (j < sample_size)
                {
                        a[i] = 1; --sample_size;
                }
                else
                        a[i] = 0;
        }
}

/* return a random permutation of [1..size] */

void shuffle(int a[], int size)
{
        int i, j;
        int temp;

        for (i = 0; i < size; ++i)
                a[i]  = i + 1;

        for (i = 0; i < size -1; ++i)
        {
                j = (random() % (size-i)) + i;
                temp = a[i]; a[i] = a[j]; a[j] = temp;
        }
}

/* print array */

void print_array(int a[], int size)
{
        int i;

        for (i = 0; i < size; ++i)
                printf("%d ", a[i]);
        printf("\n");
}

void main(void)
{
        int a[SIZE];
        int sample_size;

        srandom(time(NULL));

        printf("A random shuffle of %d elements:\n", SIZE);
        shuffle(a, SIZE);
        print_array(a, SIZE);

        sample_size = random() % SIZE + 1;
        printf("A random sample of %d out of %d elements:\n",
                sample_size, SIZE);
        sample(a, SIZE, sample_size);
        print_array(a, SIZE);
}
```

Figure 4: *A simple C program.*

```c
/* Modified copy */

#define N 10
#include <time.h>
#include <stdio.h>

void main(void)
{

        int f;
        int array[N];

        srandom(time(NULL));


        shuff(array, N);
        printf("A random shuffling:\n")
        print(array, N);

        f = random() % N + 1;
        samp(array, N, f);
        printf("A random sampling:\n");
        print(array, N);
}


void samp(int array[], int n, int m)
{
        int j, i;

        if (m > n || m < 0)
                return;

        for (i = 0; i < n; ++i)
        {
                j = random() % (n -i);
                if (j < m)
                {
                        array[i] = 1;
                        --m;
                }
                else
                        array[i] = 0;
        }
}
void print(int array[], int t)
{
        int i;

        for (i = 0; i < t; ++i)
                printf("%d ", array[i]);
        printf("\n");
}

void shuff(int array[], int s)
{

        int p;
        int n, m;

        for (m = 0; m < s; ++m)
                array[m]  = m + 1;

        for (m = 0; m <= s -1; ++m)
        {
                n = (random() % (s-m)) + m;
                p = array[m];
                array[m] = array[n];
                array[n] = p;
        }
}
```

Figure 5: *Modified by removing comments, changing names, moving statements and functions.*

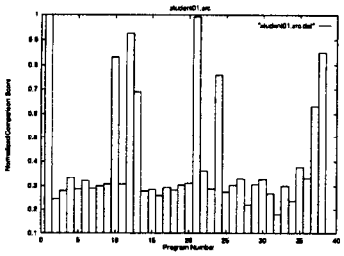

Figure 6: *One set of comparisons against a reference program in group I (36 programs).*

	Program Size	Token Array Size	Time
Ave.	7808	714	0.20
Max.	15027	1361	0.80
Total	296716	27157	124

Figure 7: *Statistics for group I.*

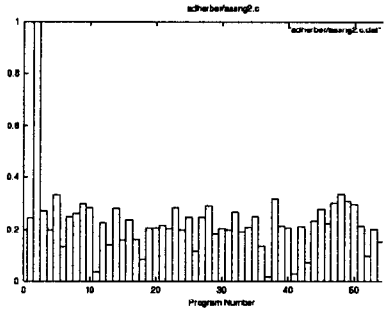

Figure 8: *One set of comparisons against a reference program in group II (56 programs).*

	Program Size	Token Array Size	Time
Ave.	3415	399	0.09
Max.	8050	835	0.48
Total	191243	22352	139

Figure 9: *Statistics for group II.*

we mentioned before, `sim` could be extended to evaluate not only uniqueness but also style and correctness of computer programs. Such a tool conceivably could be used for interactive testing in a first programming course.

There are known heuristics to reduce the quadratic running time of the underlying string alignment algorithm, which we plan to implement in the next version of `sim`. Also it appears not necessary to run `sim` on every possible pair of programs in order to detect plagiarism. Since the number of cheating incidents is usually small, it is often enough to find the highest-scoring pairs in a group of programs. If no cheating evidence can be found from these pairs, the instructor needs not examine the rest. We are currently working on sub-quadratic algorithms for finding highest-scoring pairs.

References

[1] A. AIKEN, *Measure of software similarity.* URL http://www.cs.berkeley.edu/~aiken/moss.html.

[2] B. S. BAKER, *Parameterized pattern matching: Algorithms and applications*, J. Comput. System Sci., 52 (1996), pp. 28–42.

[3] M. BLUM AND S. KANNAN, *Designing programs that check their work*, in Proceedings of the 21st Annual ACM Symposium on Theory of Computing, 1989, pp. 86–97.

[4] L. BABAI AND S. MORAN, *Arthur-Merlin games: a randomized proof system and hierarchy of complexity classes*, Journal of Computer and System Sciences, 36 (1988), pp. 254–276.

[5] S. GOLDWASSER, S. MICALI, AND C. RACKOFF, *The knowledge complexity of interactive proof systems*, SIAM Journal on Computing, 18 (1989), pp. 186–208.

[6] X. HUANG, R. C. HARDISON, AND W. MILLER, *A space-efficient algorithm for local similarities*, Computer Applications in the Biosciences, 6 (1990), pp. 373–381.

[7] D. HIRSCHBERG, *A linear space algorithm for computing maximal common subsequences*, Communications of the ACM, 18 (1975), pp. 341–343.

[8] J. W. HUNT AND M. D. MCILLROY, *An algorithm for differential file comparison*, Tech. Report 41, Bell Laboratories, June 1976.

[9] J. W. HUNT AND T. G. SZYMANSKI, *A fast algorithm for computing longest common subsequences*, Communications of the ACM, 20 (1977), pp. 350–353.

[10] H. T. JANKOWITZ, *Detecting plagiarism in student Pascal programs*, Computer Journal, 31 (1988), pp. 1–8.

[11] L. MALMI, M. HENRICHSON, T. KARRAS, J. SAARHELO, AND S. SAERKILAHTI, *Detecting plagiarism in Pascal and C programs*, tech. report, Helsinki University of Technology, 1992.

[12] E. W. MYERS AND W. MILLER, *Optimal alignments in linear space*, Computer Applications in the Biosciences, 4 (1988), pp. 11–17.

[13] S. B. NEEDLEMAN AND C. D. WUNSCH, *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, Journal of Molecular Biology, 48 (1970), pp. 443–453.

[14] T. F. SMITH AND M. S. WATERMAN, *Identification of common molecular subsequences*, Journal of Molecular Biology, 147 (1981), pp. 195–197.

[15] L. G. VALIANT, *A theory of the learnable*, in Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing, Washington, D.C., 1984, pp. 436–445.