Computer Science at Kent

# Naïve tools for studying compilation histories

Matthew C. Jadud, Sally A. Fincher

Technical Report No. 3-03
March 2003

# Naïve tools for studying compilation histories

Matthew C. Jadud, Sally A. Fincher
University of Kent Canterbury
Computing Laboratory
{mcj4, saf}@kent.ac.uk

March 14, 2003

## 1 Motivation

We are interested in assessing the impact of pedagogic programming environments in the teaching of programming to novices. As part of this work, we are interested in studying student *compilation histories*—a sequence of snapshots of a student's program taken each time they compile. Much like a sequence of moving pictures imply motion, we believe there is merit in studying the evolution of a student's program. A compilation history represents one aspect of a *solution trajectory*, a sequence of observable (external) and mental (internal) states that define a student's path from problem start to completion.

This work represents one step in a necessarily larger sequence of studies regarding the behaviour of novice programmers. The work is exploratory at this time, but nevertheless engages with significant and important questions:

**Why do we compile?** Tied up in this overarching question are many other related questions. Why do novices compile their programs? What is the significance of the act? At what point in their personal process do they "hit the compile button?"

**What is a $\Delta$?** There is nothing "between the frames" of a movie. In contrast, what takes place in the space between compiles is loaded with meaning. The question of what mental models the novice has attempted to map into code during the $\Delta$ (the "in-between times") is a reduction/smaller version of the larger question of how a novice (or any programmer, for that matter) maps concepts into code.[1]

**Is behaviour influenced by environment?** We define a *pedagogic programming environment* very broadly. BlueJ and DrScheme represent two examples of one class of pedagogic pro-

[1] *Key frame animation* may be a better analogy, as there is process that takes place between the key frames. Additionally, we wonder if there is a notion of a $\Delta$ as well as a $\delta$... we feel each means something slightly different. Because naming is such a powerful tool, we are hesitant at this time to say why it is a code $\Delta$, and not a code $\delta$, or why there is a difference.

gramming environment; they are quite traditional in nature.[2] By comparison, an evolving LEGO robot and visual (graphical) program together form a programming environment, with qualitatively different mechanisms for feedback with respect to syntactic and logical errors. Similarly, storytelling environments (Alice, ToonTalk) or simulation environments (StarLogo) are all pedagogic, all have students programming, and all provide an environment for that behaviour.[3] What behaviours do we see in common between solution trajectories in each of these environments? Which are different? Are those differences products or artifacts of the environment? Are "good" behaviours in one "bad" in another?

## 2 Naïve Tools for Exploring Compilation Histories

Compilation histories are compelling because they operate over data that is discrete, well-defined, and readily available. There are lots of students writing lots of programs in lots of places. Most importantly, the artifacts we wish to study are written in well defined, regular languages that lend themselves to automated analysis, supported by a body of literature and existing tools. We do not believe new tools are necessary for the work we propose.

Because we do not know what we will find in the $\Delta$s, we are interested in applying naïve metrics and examining what we find in many different ways. While we may need to assemble existing tools in complex ways, or even write new tools, to perform savvy analyses later on, we believe starting simply will reveal a great deal, and better inform later efforts. To this end, we wish to examine the $\Delta$s from three different perspectives: the textual, the structural, and the intersection of these two.

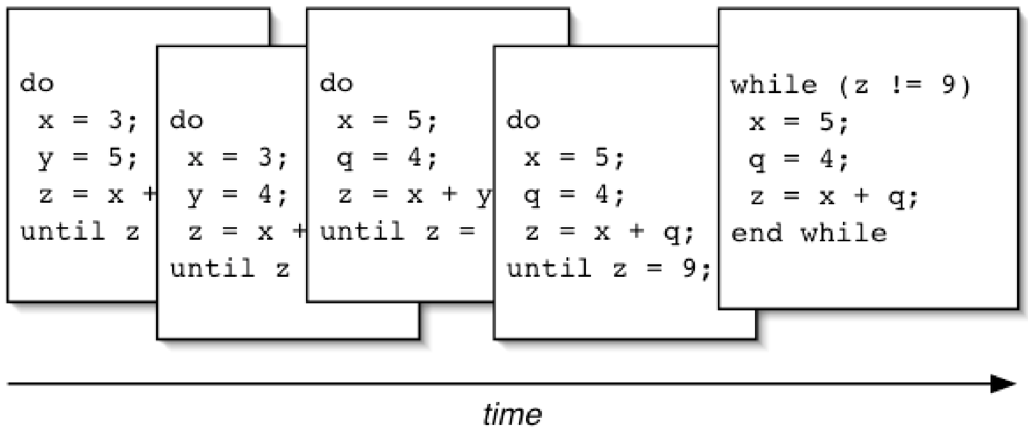
### 2.1 The Compilation History

A *solution trajectory* can be roughly defined as the evolution of a student's work towards a particular solution. A *compilation history* is one aspect of a solution trajectory: a sequence of time-stamped snapshots of code at various points between creation and completion. behaviourally, this is relatively impoverished—it tells us nothing of the programmer's mood, their thoughts, how tired they are, or any one of thousands of other aspects of their trajectory. But, it does give us something small and tangible to work with.

Figure 1 is a hypothetical compilation history we will use throughout the remainder of the text. We pretend that the student is attempting to figure out why their program is not terminating. The snapshots A$\rightarrow$E will be discussed throughout our presentation of some textual, structural, and combined tools for analyzing the $\Delta$.

[2]Both are available, as well as papers and more information, from http://www.bluej.org and http://www.drscheme.org, respectively.

[3]http://alice.cs.cmu.edu, http://www.toontalk.com, http://education.mit.edu/starlogo, respectively.



| A | B | C | D | E |
|---|---|---|---|---|
| do | do | do | do | while (z ! = 9) |
| x = 3; | x = 3; | x = 5; | x = 5; | x = 5; |
| y = 5; | y = 4; | q = 4; | q = 4; | q = 4; |
| z = x + y; | z = x + y; | z = x + y; | z = x + q; | z = x + q; |
| until z == 9; | until z == 9; | until z == 9; | until z == 9; | end while |

Figure 1: A compilation history, in picture and textual form.

### 2.1.1 Trivial analyses

In the name of simplicity, we begin with numbers and time. How many times did a given student submit their work? How much time elapsed between those submissions (mean, median, average, standard deviation, etc.). Based on this data alone, we might perform a crude cluster analysis, and see if we have students with similar compilation patterns based on the number of submissions they made, or the amount of time spent on each $\Delta$. This kind of information may help guide further analysis, or may be interesting in it's own right.

### 2.1.2 Textual analysis

At their simplest, textual comparisions involve applying an algorithmic derivative of STRING-EDIT or LEAST-COMMON-SUBSTRING (LCS) to the source code of any two adjacent compilations. STRING-EDIT is a relatively simple algorithm, often implemented and analyzed by computer science students.[WF74] STRING-EDIT compares two strings, and finds the smallest number of edit operations that need to be applied to convert the first string into the second. To convert one string to another, a combination of edit operations are applied in sequence; either we

**Insert** a character into the string, causing all characters to the right of the insertion point,

**Delete** a character, causing all characters to the right of the insertion point to shift left, or

**Update** the value of a character location.

So, for example, the STRING-EDIT distance between the words “fish” and “dig” is 3; one to update the ‘f’→‘d’, one to update the ‘s’→‘g’, and one to insert an ‘h’. So in calculating the STRING-EDIT cost, we are measuring the amount of text that has changed in a given Δ, regardless of the semantic content or structure of that text.

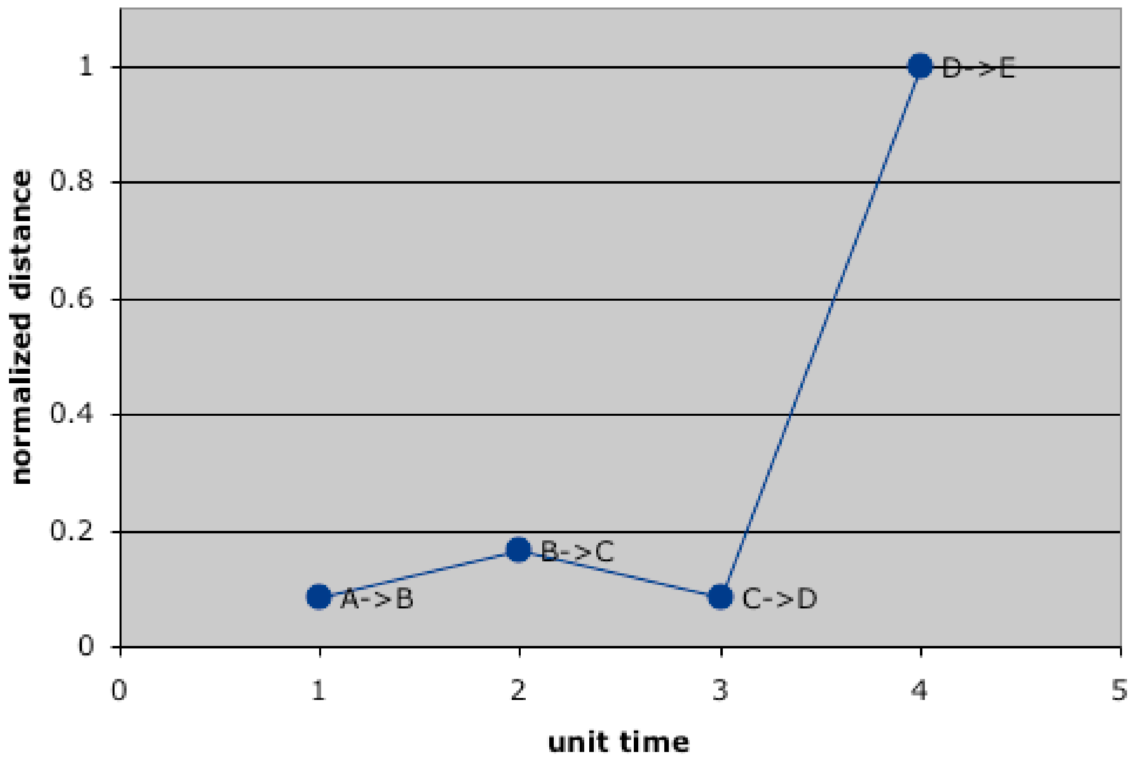


Figure 2: STRING-EDIT distance for each Δ in the history A→E

In our example trajectory, the volume of code changed in each Δ from A to D is very small—each edit is only one or two characters, and therefore the STRING-EDIT cost is small (Figure 2). The student is only making a very small change in each Δ, although we cannot easily say (without examining the code) how significant those changes are.[4]

The Δ from D→E is a bit more significant; here, the programmer changed their loop logic from a `do` loop to a `while` loop; the STRING-EDIT distance here is 12. This is interesting to us for a number of reasons; first, it is a significant change from previous edits. Second, we might look at how much time elapsed, and plot the change in time against the size of the edit—did all the small edits take (roughly) the same amount of time? Third, we will be interested in how this change tracks with other metrics; for example, did the structure of the code change significantly as well?[5]

[4] *What is going on in the programmer’s head?* Every time we make any statement in this document about the nature of a Δ, it must always be asked what that means *to the programmer*, and we cannot answer that question with the tools being discussed, nor can we assess the validity of our explorations without a number of triangulating studies. We would encourage the reader to ask this question over and over of themselves as you make your way through this document.

[5] This graph does not show another interesting possible point of exploration: how much code was written before the first compile?

STRING-EDIT and LCS are representative of a class of algorithms for comparing pieces of text irrespective of the content or structure of that text. To support our initial explorations, we implemented STRING-EDIT in Scheme for comparing S-expression based programs; in this case, we convert S-expressions into a list of characters (thus eliminating whitespace and indenting irregularities) so we can simply examine a given program in terms of the number of edits (insertions, deletions, and updates) that took place.

As well as implementing known algorithms, existing tools can potentially be used here as well. For example, a $\Delta$ can be completely described by `diff`, a powerful UNIX command line tool. Within a constant factor, the size (in bytes) of the output of `diff` could be used to measure the difference between two programs. Similarly, we could throw away *all* structure in a piece of code, sort it, and then run the $\Delta$ through `comm`. This can give us all the lines that are the same, all the lines that differ, and all the lines that are shared in common between the two programs. Combined with scripting languages like sed, awk, and Perl to operate on the output of these tools, we can gain many different and interesting perspectives on the nature of $\Delta$.

### 2.1.3 Structural analysis

Unfortunately, there are no readily available, off-the-shelf tools for performing a structural comparison between two documents that contain structure (eg. LaTeX files, XML documents, programs). However, there is a rich body of research surrounding the comparison of structure in documents; the recent explosion of XML as a textual, structured method for representing data in many different contexts has helped spur this research in the last 10 years.[CRGMW96, CJK$^{+}$01]

To remedy this problem, we implemented a tool to perform structural comparisons of code in Scheme. We felt Scheme was a natural choice for two reasons: first, we know of a large, existing set of programs written in Scheme (and the natural choice of language for processing and analyzing Scheme programs is Scheme); second, many parser generators (like ANTLR and others) are capable, with little or no modification, of generating parse trees in an S-expression-based syntax from languages like C, C++, Java, and other commonly used languages in introductory courses. As we initially wrote this tool to handle Scheme programs (which are expressed in a concrete, S-expression-based syntax), extending it to handle additional programs in an S-expression format will not be difficult.

The algorithm we chose was that suggested by Zhang, Shasha, and Wang.[ZSW94] It is very similar to STRING-EDIT, in that there are only three fundamental edit operations allowed on the tree:

**Insert** a node that takes one or more other nodes as it's children,

**Delete** a node, moving all the children of the removed node "up" in the tree, or

**Update** the label of a node.

While this algorithm was improved upon by ZSW and others in later work, we felt that these algorithms were too complex for our needs and might actually obscure some edits that take place in the Δ between compiles, which would be undesirable.[ZWS95, SWG02] As there is apparently no shared notational conventions among authors of this class of algorithms, as well as very few well documented implementations to compare and verify our work against, keeping our initial implementation simple helps us to explore with slightly more confidence.

Going back to our running example, the Δ between D and E introduced a significant structural change, as the student decided to use a `while` loop instead of the `do` loop they had been using previously.[6]

| D | E |
|---|---|
| do | while (z ! = 9) |
| x = 5; | x = 5; |
| q = 4; | q = 4; |
| z = x + q; | z = x + q; |
| until z == 9; | end while |

Figure 3: Snapshots D and E, a structural change.

If we imagine this being parsed into an S-expression based form, we might render these two programs (structurally) as the two trees in figure 4. For purposes of this document, we don't detail the algorithmic comparison; however, we can see at a glance that the lower-right hand portion of these trees are different in structure. Expressed as strings, there is a distance of 20 between the S-expressions representing these two programs; structurally, there are 18 edit operations required to convert the tree representing snapshot D into snapshot E.

If we plot the STRING-EDIT and TREE-EDIT distances of each Δ together, we find that they mirror each other well (Figure 5). At this time, we can imagine cases where small STRING-EDIT distances would result in large TREE-EDIT distances. For example, a copy-and-paste operation may be expressed succinctly by a command-line textual analysis tool, thus implying a low edit cost in the Δ (which, in some ways, it is). Structurally, however, the program would be very different. It is this kind of assumption regarding what is or is not "expensive" that makes us leery of using powerful tools too soon; the less we can assume about a programmer's thoughts and motives at this point— about what "costs" them a great deal and what does not—the better.

### 2.1.4 Textual and Structural comparions

There are two reasons we desire to keep our analysis simple:

- There are powerful techniques for analyzing even simple data that can illuminate much about the structure of the space that data defines, and
- There are tools available that perform complex analyses that we do not wish to replicate.

[6]Or, perhaps they became so tired of banging their head on the wall they simply copied someone else's solution, or...

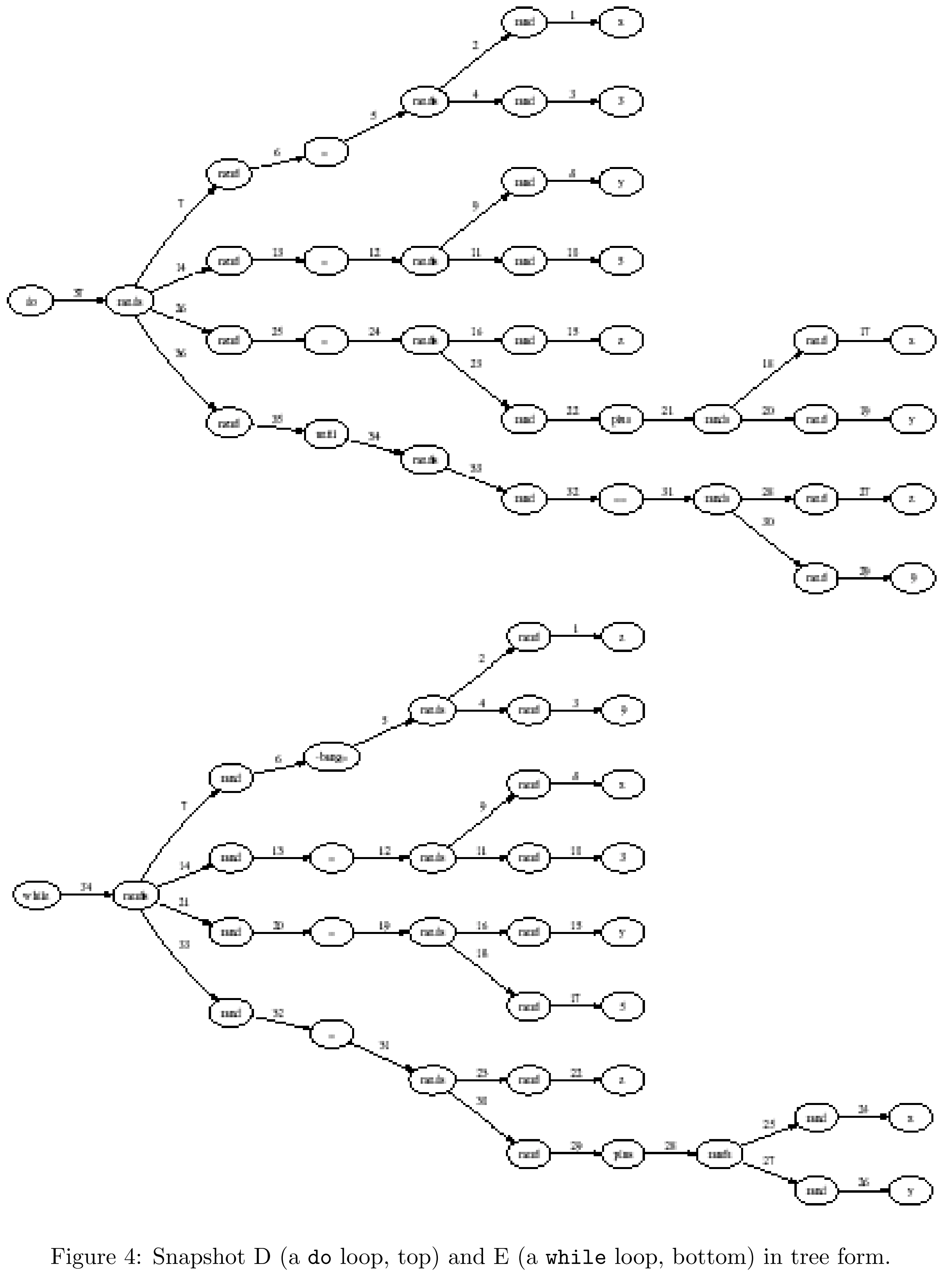


Figure 4: Snapshot D (a `do` loop, top) and E (a `while` loop, bottom) in tree form.

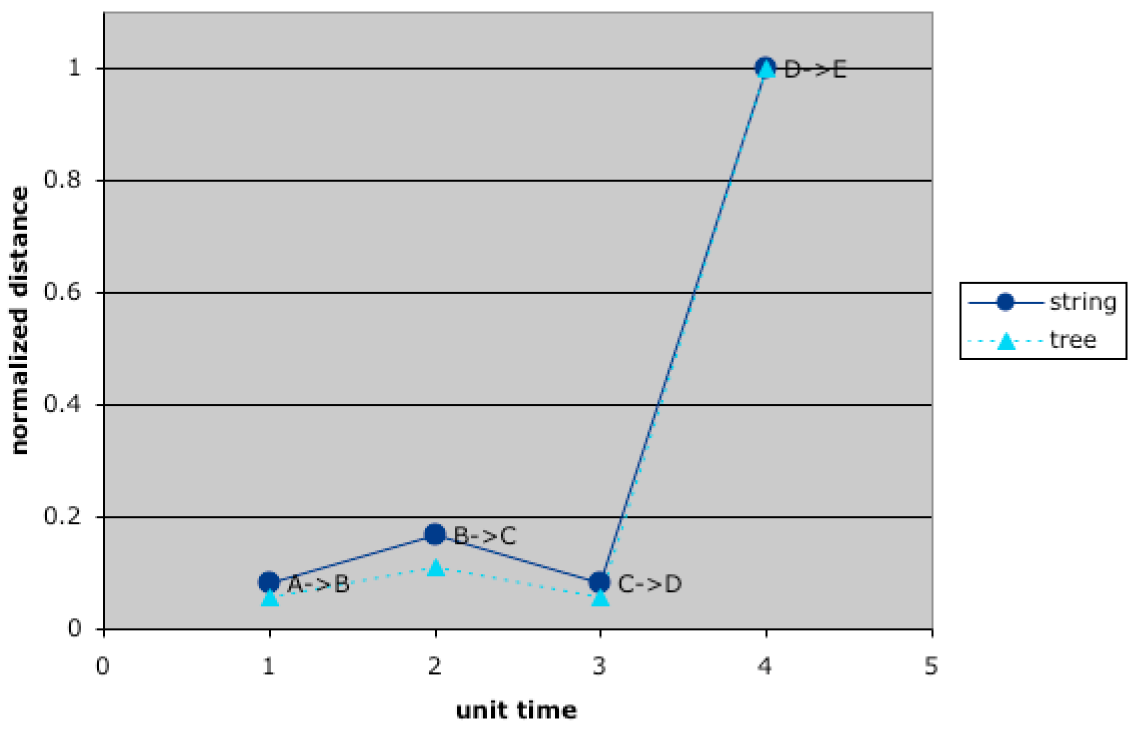


Figure 5: Normalized STRING- and TREE-EDIT distances for the history A→E.

As computers continue to become more powerful, so does the community's interest in automated plagiarism detection.[7] This is a very difficult problem; given that most student programming assignments involve writing small amounts of code, determining whether one student copied a four-line solution off another is a very difficult (if not impossible) task. While we do not believe that plagiarism (as a social problem) is best combatted through technical means, we are happy to utilize these tools in our own work.

Most plagiarism detection tools are designed to find documents that are remarkably similar. However, in doing so, they must explore the space of differences between two documents, and they often make use of a combination of algorithms like those we have described here. One particular technique, used by JPlag will be discussed as an example of how combining textual and structural differences can illuminate otherwise hidden information in a $\Delta$.[8]

One approach used by JPlag is a pseudo-structural analysis of a student's code. While a purely textual or structural analysis does tell you something about the code, it can be foiled by students simply by changing variable names and the like. To eliminate this kind of problem, JPlag goes through and renames all variables, procedures, and the like to whitewash the code into a bland, yet structurally and semantically relevant form. Perhaps it is better referred to as a "typewash," as the result is that every object is replaced with it's "type."

[7]http://www.comp.leeds.ac.uk/hannah/CandIT/plagiarism.html and http://www.dcs.shef.ac.uk/teaching/eproj/msc2001/abs/m0ra
are two good starting points for the interested reader.

[8]More information on JPlag can be found online at http://www.jplag.de.

After type-washing the code, we are left with a completely different compilation history.

| A | B | C | D | E |
|---|---|---|---|---|
| do | do | do | do | while (V $E_{op}$ C) |
| V $A_{op}$ C; | V $A_{op}$ C; | V $A_{op}$ C; | V $A_{op}$ C; | V $A_{op}$ C; |
| V $A_{op}$ C; | V $A_{op}$ C; | V $A_{op}$ C; | V $A_{op}$ C; | V $A_{op}$ C; |
| V $A_{op}$ V + V; | V $A_{op}$ V + V; | V $A_{op}$ V + V; | V $A_{op}$ V + V; | V $A_{op}$ V + V; |
| until V $E_{op}$ C; | until V $E_{op}$ C; | until V $E_{op}$ C; | until V $E_{op}$ C; | end while |

Figure 6: Typewashing the compilation history.

In figure 6 we have replaced all variables by $V$, all constants by $C$, all assignment operators by $A_{op}$, and all equality tests by $E_{op}$. If we then were to compare the analysis of this new compilation history ($A_{washed} \rightarrow E_{washed}$) against the original source, several interesting things come to light. Perhaps most interestingly, we see the STRING-EDIT cost of the Δs A→B, B→C, and C→D are zero. This tells us that the student is making changes entirely within the "type" of object they are editing; were there differences in both the original and typewashed code, we would know that they were making changes to the source that were somehow outside of the typing already in the code from the previous edit, and possibly structural as well.

## 3 Conclusion

Our hope is that with naïve tools and techniques, we will find interesting paths into a space that is relatively unexplored at the level of granularity we propose. We feel there is potentially a great deal to be learned in examining the Δ between compiles, and may be able to extend tools and techniques developed in this work to the larger question of analyzing solution trajectories in qualitatively different pedagogic programming environments.

## Acknowledgments

Thanks to Peter Linnington, Ian Utting, Dennis Shasha, Ralph Miarka, and Miguel Mendao for their assistance with our implementation of `EDIT()`.

## EDIT.ss

The core of Zhang, Shasha, and Wang's algorithm is commented and given below for reference. Procedures of particular importance are described or defined afterwards.

```
;;EDIT is the core of the algorithm. First, two S-expressions representing
;; trees are preprocessed and post-ordered, roots of subtrees
;; are calculated, and a matrix for storing the results of the subtree
;; calculations is defined. The minimal edit cost is left in the lower
;; right-hand corner of this matrix.

(define EDIT
  (lambda (T1 T2)
    (let* ([T1 (preprocess T1)]
           [T2 (preprocess T2)]
           [lrk1 (lrk T1)]
           [lrk2 (lrk T2)]
           [tdist (mcreate (length T1) (length T2) minit)])

      ;;Calculate the distance between each of the subtrees in the two trees.
      (foreach (i lrk1)
        (foreach (j lrk2)
          (treedist i j)))

      ;;RETURN
      (mref tdist (length T1) (length T2))
      )))
```

```
;;treedist does the majority of the work in the ZSW94 algorithm.
;; It is assumed here that treedist() is defined within the
;; lexical scope of EDIT().

(define treedist
  (lambda (i j)
    ;;Create the array that will be used to
    ;; temporarily store the edit-distance required
    ;; to convert the subtree rooted at i in T1 to
    ;; the subtree rooted at j in T2.
    (let ([TEMP (mcreate (length T1) (length T2) minit)]
          [theta-i (sub1 (l i))]
          [theta-j (sub1 (l j))])

      ;;Set the upper left-hand corner of the part of the
      ;; matrix we are interested in to zero.
      (mset! TEMP theta-i theta-j 0)

      ;;The worst possible cost we could incur would be to
      ;; delete all of the nodes from T1, and insert all the
      ;; nodes from T2. These two for loops initialize to the
      ;; worst case, which we then work to improve.
      (for (s  (l i)  (node-number i))
           (mset! TEMP s theta-j
                  (+ DELETE (mref TEMP (sub1 s) theta-j))))

      (for (t (l j) (node-number j))
           (mset! TEMP theta-i t
                  (+ INSERT (mref TEMP theta-i (sub1 t)))))

      ;;The critical part of the loop; visit every pair of nodes
      ;; in the subtrees rooted at i and j. If they are the same
      ;; node, use lemma 1; otherwise, as proved in ZSW94,
      ;; lemma 2 is appropriate.
      (for (s (l i) (node-number i))
           (for (t (l j) (node-number j))
                (if (and (= (l i) (l (node-lookup s T1)))
                         (= (l j) (l (node-lookup t T2))))
                    (forestdist1 s t T1 T2 TEMP)
                      (forestdist2 s t T1 T2 TEMP)
                  )))
      )))
```

```
;;forestdist1 and 2 represent two possible conditions for the application
;; of lemma 3 from the ZSW94 paper; we can only refer interested
;; readers to the paper for a thorough discussion and proof of their
;; applicability, and will not attempt to recreate that discussion here.

(define forestdist1
  (lambda (s t T1 T2 TEMP)
    ;;We do not assign any cost to the UPDATE operation if the labels
    ;; are already the same, and we assume all operations have unit cost.
    (let ([smallest
           (min (+ DELETE (mref TEMP (sub1 s) t))
                (+ INSERT (mref TEMP s (sub1 t)))
                (+ (mref TEMP (sub1 s) (sub1 t))

                   (if (equal?
                        (get-label (node-lookup s T1))
                        (get-label (node-lookup t T2)))
                       0
                     UPDATE))
               )])
      ;;Update the temporary matrix, and the global matrix;
      ;; this lemma is only applied when dealing with a pair
      ;; of keyroots (a pair of subtree roots).
      (mset! TEMP s t smallest)
      (mset! tdist s t (mref TEMP s t)))))


(define forestdist2
  (lambda (s t T1 T2 TEMP)
    ;;As per forestdist1, but we also need to examine the
    ;; running cost of editing this subtree.
    (let* ([ls (l (node-lookup s T1))]
           [lt (l (node-lookup t T2))]
           [smallest
            (min (+ DELETE (mref TEMP (sub1 s) t))
                 (+ INSERT (mref TEMP s (sub1 t)))
                 (+ (mref TEMP (sub1 ls) (sub1 lt))
                    (mref tdist
                          (number (node-lookup s T1))
                          (number (node-lookup t T2))))
                 )])
      (mset! TEMP s t smallest))))
```

## References

[CJK$^+$01] Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond T. Ng, and Divesh Srivastava. Counting twig matches in a tree. In *ICDE*, pages 595–604, 2001.

[CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Change detection in hierarchically structured information*, pages 493–504, 1996.

[SWG02] Dennis Shasha, Jason Tsong-Li Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Symposium on Principles of Database Systems*, pages 39–52, 2002.

[WF74] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[ZSW94] Kaizhong Zhang, Dennis Shasha, and Jason Tsong-Li Wang. Approximate tree matching in the presence of variable length don't cares. *J. Algorithms*, 16(1):33–66, 1994.

[ZWS95] K. Zhang, J. T. L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs and related problems. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, pages 395–407, Espoo, Finland, 1995. Springer-Verlag, Berlin.