DiffTree:
Inferring Phylogenies for Evolving Software

Christopher W. Fraser

Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA

August 2005

Technical Report

MSR-TR-2005-109

Abstract

DiffTree infers a parsimonious evolutionary tree from related computer programs. It can help programmers understand how to best remove copied code. It adapts techniques from computational biology that automatically infer evolutionary trees or phylogenies from gene sequences or other biological data.

Introduction

Software systems commonly include multiple versions of some elements, which arise from several practices. Programmers often copy and optionally edit code, resulting in identical copies or near matches. Static linking can also contribute variation when a system includes binaries linked with different versions of an evolving library.

Source code control systems [Rochkind] automatically maintain a version tree, which records the evolution of a code base at, typically, the file level. Check-ins often batch edits deemed too small too merit in an explicit node in a version tree, though genealogy extraction tools [Kim et al] nicely tease apart intra-release changes.

But there are, of course, organizations that share source but not source trees, and other organizations that share only binaries. They may still be interested in the relationships between different versions of software elements, even if they have only binaries or only source code without a source tree.

This paper describes the retrospective computation of version trees for a set of programs, without aid from source code control systems. An inference program accepts a set of codes, compares them with one another, and infers a parsimonious evolutionary tree for them. The tree proposes a likely descent from a common ancestor and thus suggests an economical strategy for refactoring [Fowler et al] or recombining them when fewer versions are wanted. It can also help to identify cases where a repair made to one version was missed in others.

Two implementations are described. The first applies a standard algorithm from computational biology for the inference of phylogenies. The second tailors a method for programmers who are eliminating duplication from code.

Computational phylogeny

Computational phylogeny infers an evolutionary tree or phylogeny from biological data [Felsenstein 2004]. A typical input is a DNA sequence from each of several species or a distance matrix that quantifies the difference between each pair of species. The output is a tree that includes all species and ideally minimizes the total number of mutations.

The tree is typically unrooted, as illustrated in Figure 1. The tree is unrooted because, for example, a two-by-two distance matrix for two species A and B cannot by itself specify whether A or B came first. That information must come from a source other than the sequence or distance matrix. When computing a phylogeny for evolving code,

one might presume that the shorter version came first, though this heuristic is not perfect.

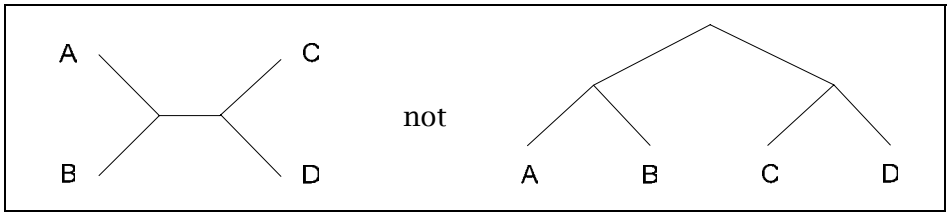

Figure 1. Unrooted tree.

Finding the optimal tree is NP-hard in general, but there are many useful heuristics and approximations. Some methods exploit domain-specific information from biology, but others apply equally well to non-biological sequences or distance matrices.

Phylogenies have also been inferred for non-biological data. For example, evolutionary trees have been inferred for chain letters [Bennett et al], which—like the computer programs studied here—also accumulate changes over time. Other examples include phylogenies for natural languages, music, and literature [Calabrasi and Vitanyi].

Distance matrices and the Neighbor-joining method

The initial implementation of phylogeny inference for evolving software versions applied a standard biological method, the Neighbor-joining method [Saitou and Nei]. It accepts a two-dimensional distance matrix. Each element of the matrix measures a distance between a pair of species or, in the current work, between two versions of some piece of code. The algorithm clusters the nearest neighbors, replaces their entries in the distance matrix with entries for an inferred ancestor, and repeats until only two leaves remain, for which there is only one unrooted tree.

Figure 2 shows a sample input and output. The input is a six by six matrix of distances. Each element is the edit or Levenshtein distance between two versions of a small program, that is, the number of lines that must be inserted, deleted, or changed to transform one version of the program into another. The distances are computed by a standard O(ND) dynamic-programming method [Myers and Miller]. The distance measure is symmetric, so a lower triangular matrix suffices.

The output is a phylogeny. It is computed by Neighbor-joining as implemented by Phylip [Felsenstein] and run as a service by Uppsala University [Uppsala]. The tree is drawn by the Phylodendron tree-drawing service at Indiana University [Gilbert].

	v1	v2	v3	v4	v5	v6
v1	0					
v2	71	0				
v3	405	402	0			
v4	483	492	242	0		
v5	410	397	25	249	0	
v6	405	402	2	242	25	0

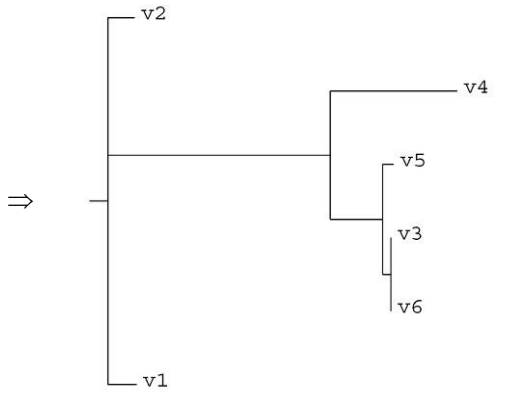

Figure 2. Sample distance matrix and resulting phylogeny.

The horizontal lines in the output tree are proportional in length to the edit distance between versions. For example, the horizontal line immediately to the left of the label v5 is short, which suggests that v5 changed little from its inferred ancestor. The horizontal lines leading to v3 and v6 are even shorter and effectively invisible; the edit distance is only two and thus dwarfed by the other distances.

The tree suggests a strategy for eliminating some or all of the duplication between the versions. One would start by merging v3 and v6, continue with v5 and then perhaps v4. Separately, it appears profitable to merge v1 and v2. The distance from {v1,v2} to the rest of the versions is, however, larger than the other distances. These two sets have diverged so much that refactoring might be unprofitable. Elements of this refactoring strategy can be read off the original table—for example, the table's lone single-digit entry calls attention to the fact that v3 and v6 are much closer than any other pair—but the phylogeny makes it easier to spot such features.

The original routines here are from disassembled binaries, though the method works equally well on source code or text files. The raw lines are expanded into tokens because programmers routinely make small changes within lines—such as renaming a variable or replacing literals [Baker]—and comparing tokens instead of lines is an inexpensive way to catch duplication within lines that are similar but not identical.

Biological applications use a variety of distance measures. Examples include divergence of gene frequencies in different populations and the edit distance between two DNA sequences. The current application uses edit distances, though it too could use other distances. For example, a set of systems too large for edit distances might be triaged with a distance matrix that compares the differing API calls between each pair of modules.

DiffTree

The phylogeny above is useful, but the distances computed for each new ancestor may not be transparent to programmers unfamiliar with phylogeny inference, because the distances computed for inferred ancestors do not correspond to the edit distance between any strings that are visible to or easily computed by the programmer.

DiffTree computes a version tree that may be easier for programmers to follow. It maintains at each step a transparent set of versions, and it mimics a process that programmers use themselves when confronted with a set of related versions: first run diff [Hunt and McIlroy] on the most similar inputs, then the next most similar, and so on. Diff computes the longest common sub-sequence of the inputs, and it emits an edit script for transforming one input into the other. For some inputs, programmers can replace the common parts with a subroutine and the differences with parameters to the subroutine calls.

More precisely, DiffTree proceeds as follows:

1. Accept a set S of versions, not a distance matrix computed from them.
2. Identify the two members of S with the smallest edit distance.
3. Replace these two members with their longest common sub-sequence. Augment the sub-sequence with a wildcard at each point where the two members differ, to account at least approximately for the fact that a parameter is needed at this point.
4. Repeat the process above until S has only two members, at which point only one (unrooted) phylogeny remains.

For example, when DiffTree is presented with the samples v1-v6 above, the result is:

```
1. Merge v6 and v3, which share 99.7%.
2. Merge #1 and v5, which share 96.8%.
3. Merge v1 and v2, which share 85.6%.
4. Merge v4 and #3, which share 33.6%.
5. Merge #2 and #4, which share 37.2%.
```

The output represents a phylogeny in postfix. DiffTree's output clearly flags the similar versions, suggests an ordering in which to tackle refactoring, and helps the programmer identify the point of diminishing returns, here probably at the pay-off cliff between steps 3 and 4. The proposed ordering is, in general, not optimal, but its greedy heuristic is common and easily understood by programmers.

As an option, DiffTree produces HTML with hyperlinks for the file names (such as “v6”) and references to merged results (such as “1.” and “#1” above). Selecting these links displays the text of the merged version, namely the longest common sub-sequence with wildcards marking differences. On disassembled code, this output is useful mainly for a high-level scan of the number and clustering of the wildcards. It ought to be more useful on source code.

For the sample input above, DiffTree and Neighbor-joining compute trees with the same shape, though they do not behave identically on all inputs. The distances are computed differently; for example, the percentages above are consistent with the relative lengths of Figure 2’s horizontal lines for closely related versions, but the correspondence fades as the edits accumulate. Nevertheless, DiffTree’s percentages have a basis that is directly useful to the programmer: they represent the fraction of the input pair that is redundant according to diff.

Discussion

DiffTree is a first cut at tree inference for evolving software, but there are many other methods to infer phylogenies [Felsenstein 2004] and to display trees [van Wijk et al]. The large number of alternatives presents an opportunity and a problem: better combinations probably exist, but it is hard to know which of the many alternatives to try next.

Other distance measures also merit study. The “similarity metric” [Li et al], which is based on LZ compression [Ziv and Lempel] instead of edit distance, has been successfully applied to detect program plagiarism [Chen et al]. Plagiarism is generally assumed to involve simpler edits, and detection does not require constructing a tree of copies, but a distance measure that helps detect plagiarism might also help with evolving software versions.

Domain-specific edit distances might also merit study. Programmers commonly rename identifiers in copied code, resulting in “parameterized duplication” [Baker]. DiffTree uses a traditional edit distance, which incurs a cost for each appearance of the identifier. The programmer, however, might regard such a change as a single edit and, indeed, might have made the change with a single editor command. A domain-specific definition for edit distances—or, equivalently, domain-specific compression of edit scripts—might give results that better match the programmer’s model.

References

B. S. Baker. On finding duplication and near-duplication in large software systems. *Proceedings of the Second Working Conference on Reverse Engineering*:86-95, 1995.

C. H. Bennett, M. Li, and B. Ma. Chain letters and evolutionary histories. *Scientific American 288*(6):76-81, 6/2003.

X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory 51*(4):1545-1551, 7/2004.

R. Cilibrasi and P. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory 51*(4):1523-1545, 4/2005.

J. Felsenstein. *Inferring Phylogenies*. Sinauer, 2004.

J. Felsenstein. PHYLIP (Phylogeny Inference Package. Department of Genetics, University of Washington, Seattle. http://evolution.genetics.washington.edu/phylip.html.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

D. G. Gilbert. Phylodendron Phylogenetic tree printer. Department of Biology, Indiana University. http://iubio.bio.indiana.edu/treeapp.

J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. CSTR #41, Bell Telephone Laboratories, 1976.

M. Kim, V. Sazawall, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. To appear in *ESEC/FSE'05*, 9/2005.

M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi. The similarity metric. *IEEE Transactions on Information Theory 50*(12):3250-3264, 12/2004.

W. Miller and E. W. Myers. A file comparison program. *Software—Practice and Experience 15*(11):1025-1040, 11/1985.

M. J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering SE-1*(4):364-370, 12/1975.

N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution 27*:261-273, 1987.

Uppsala University. Department of Molecular Evolution, Uppsala University. http://artedi.ebc.uu.se/programs/neighbor.html.

J. J. van Wijk, F. van Ham, and H. van de Wetering. Rendering hierarchical data. *Communications of the ACM 46*(9):257-262, 9/2003.

J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*(3):337-343, 5/1977.