Incremental Tree-edit Distance

Kieran O'Brien

Master in Computer Science (MCS)

University of Dublin, Trinity College

Supervisor: Stephen Barrett

Submitted to the University of Dublin, Trinity College May 2014

Declaration

Kieran O'Brien

Date: May 29, 2014

Summary

This research started with exploring programming gamification by investigating metrics which can be obtained during program development. This metric would be used to influence a users behaviour as changes are made to the code. The metric decided upon was code distance, a measure of how different the structure of one program is from another. Performing this calculation is too slow to be responsive in a gamification context. Instead, it is proposed that the distance could be updated as changes are made, using the information in the diff between versions. Two Strategies are designed and tested for accomplishing this. They are tested on examples of varying sizes for accuracy and efficiency.

Related topics are explored. E-learning is examined because it would be the proposed home for these approaches. Tree data structures are explained, because they are the representation programs take when compared. Program syntax is defined, because it is what is fundamentally being compared. Diffs are also given an overview. Dynamic program is explored in detail because the concepts are integral to understanding how the code comparison is done, and string edit distance is used as a simpler introductory example.

Previous Research is examined in the areas of gamification, code assessment, Integrated Development Environments (IDEs) (as a modification to one of these would be required to test the suitability of the metric for gamification) and code variability. Tree edit-distance algorithms are given the most thorough overview because they are used to compare programs and the work undertaken by this dissertation modifies one of these algorithms.

The design chapter describes the two strategies proposed for updating the tree edit distance using the diff. One is Distance Update and the other is Distance Inference. Both of these require a shortest edit sequence (to change one tree into another) to be generated. Distance Update reuses previously calculated results as much as it can (making decisions based on the edit sequence) and Distance Inference compares edit sequences and looks for evidence that programs are diverging or converging.

These methods are tested on a variety of input data of different sizes. Accuracy and efficiency are recorded. Distance Inference was found to be mostly accurate for very small trees with occasional spikes of error. It was unreliable on large trees with error between fifty and one hundred percent. Distance Update was able to make modest savings on both sizes of tree but the overhead involved in building the edit sequence caused it to take longer than the standard algorithm.

It is concluded that further work needs to go into making Distance Inference more reliable, in which case it will prove useful in the gamification context in which it was originally imagined. The overhead involved in building the edit sequence needs to be dealt with for Distance Update to become useful, but even so the modest savings make it ill suited to the gamification context.

Future work involves improving both approaches and testing the effects of code difference as a gamification metric by modifying an IDE.

Acknowledgements

Thanks to Stephen Barrett and Brendan Cody-Kenny for the advice, feedback and support they gave me over the course of doing this dissertation. I would also like to thank my family and friends for their support.

Kieran O'Brien
University of Dublin, Trinity College
May 2014

Contents

List of Figures

Abbreviations

AST Abstract Syntax Tree. 3, 8, 9, 23, 24, 32, 35

DAG Directed Acyclic Graph. 10–12

DP Dynamic Programming. 3

IDE Integrated Development Environment. iv, 5, 36, 38

LMD Leftmost Descendant. 8, 23

LOC Lines of Code. 30, 31, 36, 37, 41, 44, 45, 47, 49

MOOC Massive Open Online Course. 5, 6

TED Tree-edit Distance. iv, 3, 22–24, 29

Chapter 1

Introduction

This chapter contains a brief overview of the background behind the work, the methodology for exploring this topic and an outline of the structure of the rest of the document.

1.1 Background

The metrics gathered from analysis of code changes in real-time could be used to gamify the development process, yielding pedagogical benefits. Gamification techniques could be used to encourage students to be creative and to approach tasks in unique ways, potentially incentivising them to attempt their own solutions instead of relying on their peers. Real time assessment could be leveraged to grant students points and awards for desirable behaviour.

The initial inspiration for this work was a visualization created by [Nguyen et al., 2014]. It shows the "distances" between different pieces of source code. The distance is a measure of how different one piece of code is from another. 40,000 programs were compared, each a submission to a webcourse in Machine Learning provided by the website Coursera. Each node represents a submission, and the colour represents the functionality (red being submissions that passed all of the unit tests and were therefore correct). Edges are drawn between submissions which contain similar syntax, and if programs are both

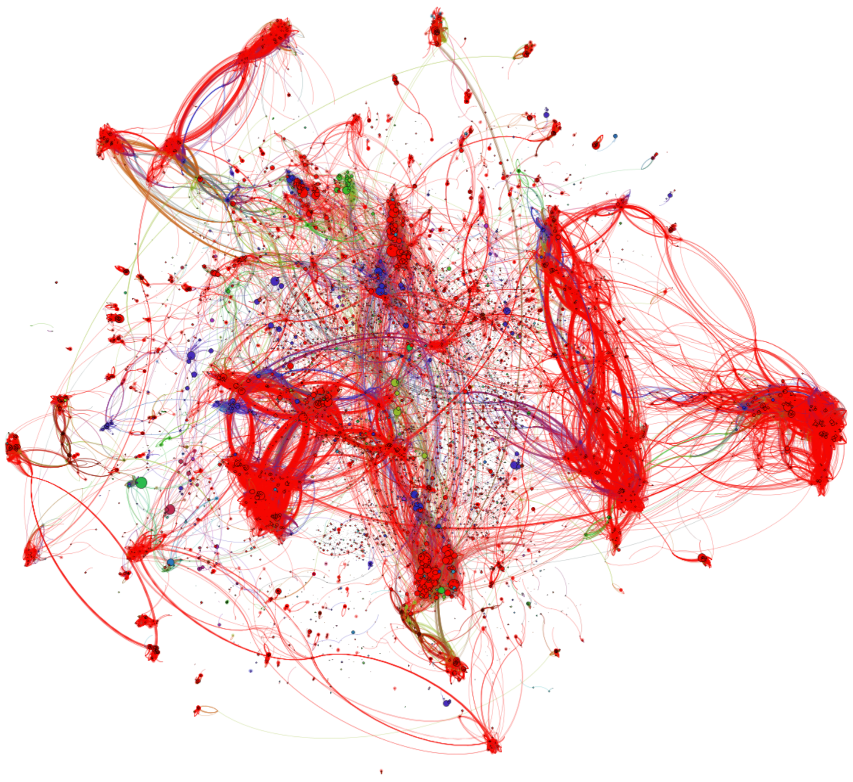


Fig. 1.1: Comparison between 40,000 programs

syntactically *and* functionally similar, they are clustered together.

It was proposed that using this distance metric in a gamification context could prove useful. Gamification is the application of techniques which incentivise and encourage desired behaviour through rewards and competition. Providing users with an indication of how unique their own code is could potentially influence them to explore more diverse and interesting solutions to problems, while discouraging sharing solutions. To achieve this, these distances would need to be calculated in close to real-time, so that the system feels interactive and user can see the effects of code changes immediately.

1.2 Objective

Code distances are calculated using a Tree-edit Distance (TED) algorithm, but unfortunately these tend to have very high time complexities. To find the distance between one program and another, the AST needs to be abstracted from both. The question posed by this research is: If changes are made to the first program, can the knowledge of which changes have been made be used to find the new distance faster than using the standard tree edit distance algorithm again?

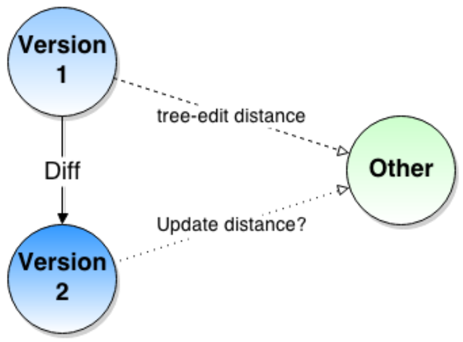


Fig. 1.2: Visualization of research question

Two different strategies are designed for accomplishing this. One reuses data generated from comparing the original two programs. The diff informs the algorithm which data needs to be recomputed. The other strategy infers a new distance based on the changes that have been made and examining the diff. Test data has been generated both by hand and by extracting an ASTs from programs. The different approaches are timed and for each the accuracy and efficiency is measured.

1.3 Outline of dissertation

This introduction is followed by the Background chapter explains some of the concepts essential to understanding the research carried out. These include e-learning, tree data structures, program syntax, code diffs, Dynamic Programming (DP) and finding string edit distance. The Related Work chapter examines existing research in the areas of e-learning, gamification, code analysis, syntactic variability and tree edit distance algorithms.

The Design chapter details the proposed strategies for updating the tree edit distance using the diff. The Methodology chapter describes the test data and the tests that will be carried out. The findings are listed in the Results chapter and the Conclusions chapter reasons about what the results mean and where the research can go next.

Chapter 2

Background

This chapter provides an introduction to many of the important concepts related to the work undertaken. These topics include e-learning, tree data structures, code syntax and the information contained in a diff (a representation of code changes that have been made). Some of these topics are explored in more depth in the Related Work section.

2.1 E-Learning

2.1.1 Overview

E-learning is the use of information technology and electronic media in the field of education. It may supplement traditional learning approaches or replace them. It may or may not be synchronous (all paricipants interacting at the same time).

This work was motivated by Massive Open Online Courses (MOOCs) in particular. Two prominent e-learning resources are Coursera and Khan Academy. These courses may have tens of thousands of people taking part in them. They aim to have unlimited participation and facilitate asynchonous participation. Content can take the form of video, audio and interactive web elements. IDEs allow programmers to write real code and compile it without needing to install anything locally or configure their own personal machines. The code they write can be assessed by the course and feedback can be given.

2.1.2 Advantages

MOOCs provide a number of advantages when compared to traditional education approaches. Multimedia content can be sped up, slowed down, paused and replayed. Often one lecturer is presenting material to thousands of students. Forums for discussion can contain thousands of users. These forums can be moderated to prevent students from sharing hints which are too detailed. Auto-grading can be used to give almost instantaneous feedback on submitted assignments. Detailed statistics can be collected and gamification techniques, such as awarding badges and points, can be employed.

2.1.3 Disadvantages

MOOCs also introduce difficulties. Participants may feel more isolated as they won't have as much contact with their peers. This could lead to a lack of motivation in keeping up with the course. It could also diminish competion if measures aren't taken to inform participants what the class standard is. The scale of these courses make generating feedback and assessing performance very difficult. Teaching assistants and lecturers are vastly outnumbered and cannot spend time helping or assessing the work of students.

This is somewhat alleviated by the presence of forums, but other measures must be taken so that students can be assessed on an individual level. Autograding techniques exist to assess the correctness of code artifacts. The computation required for autograding scales well because the metrics can be gathered and analysed independently in parallel.

2.2 Trees

2.2.1 Overview

A tree a hierarchical structure which is used in fields such as graph theory and computer science. It resembles an upside-down tree when visualised, with the “root” at the top and the “leaves” at the bottom. Trees are made up of nodes connected together. A node can have other nodes connected to it as children, but there are no loops. The ancestors of a node can be found by moving up from parent to parent

2.2.2 Post-order Traversal

To visit the nodes of a tree in post-order, the following steps are needed at each node visited, beginning with the root: For each child node from left to right, recursively descend before giving the node the next number in the sequence. Each node will have a lower number than those to the right of it or above.

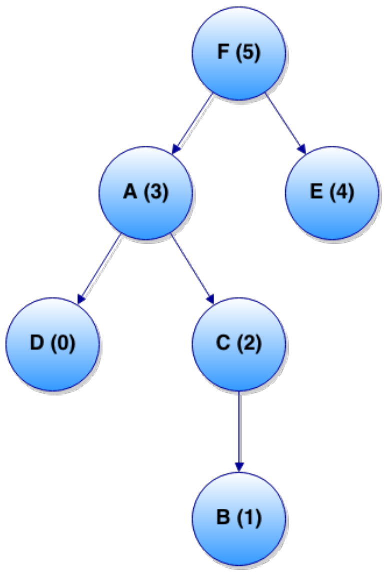


Fig. 2.1: A tree annotated in post order

2.2.3 Leftmost Descendants

The Leftmost Descendant (LMD) of a node can be found by repeatedly descending left from that node until you reach a node with no children, which is the LMD. The LMD of a node with no children is the node itself.

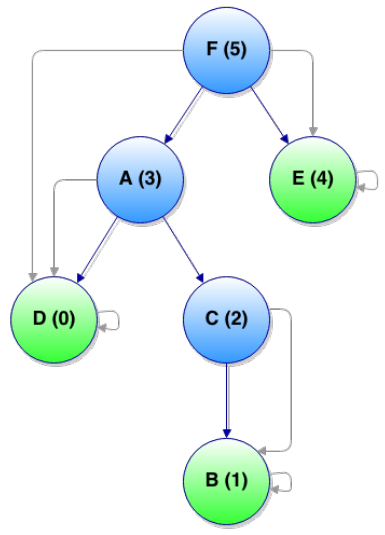


Fig. 2.2: Left most descendant relationships

2.2.4 Keyroots

Keyroots are nodes for which there exists no greater node (in post-order notation) which shares a LMD ().

2.3 Program Syntax

The syntax of a programming language is the set of rules which describe how it can be correctly constructed. Programming languages are made up of symbols and words. These must be used in the correct sequence for a program to compile. For example, an addition symbol typically requires that there be numbers on either side of it. These

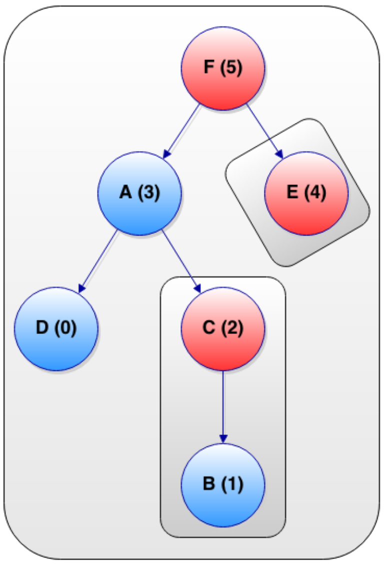


Fig. 2.3: Keyroots and respective subtrees

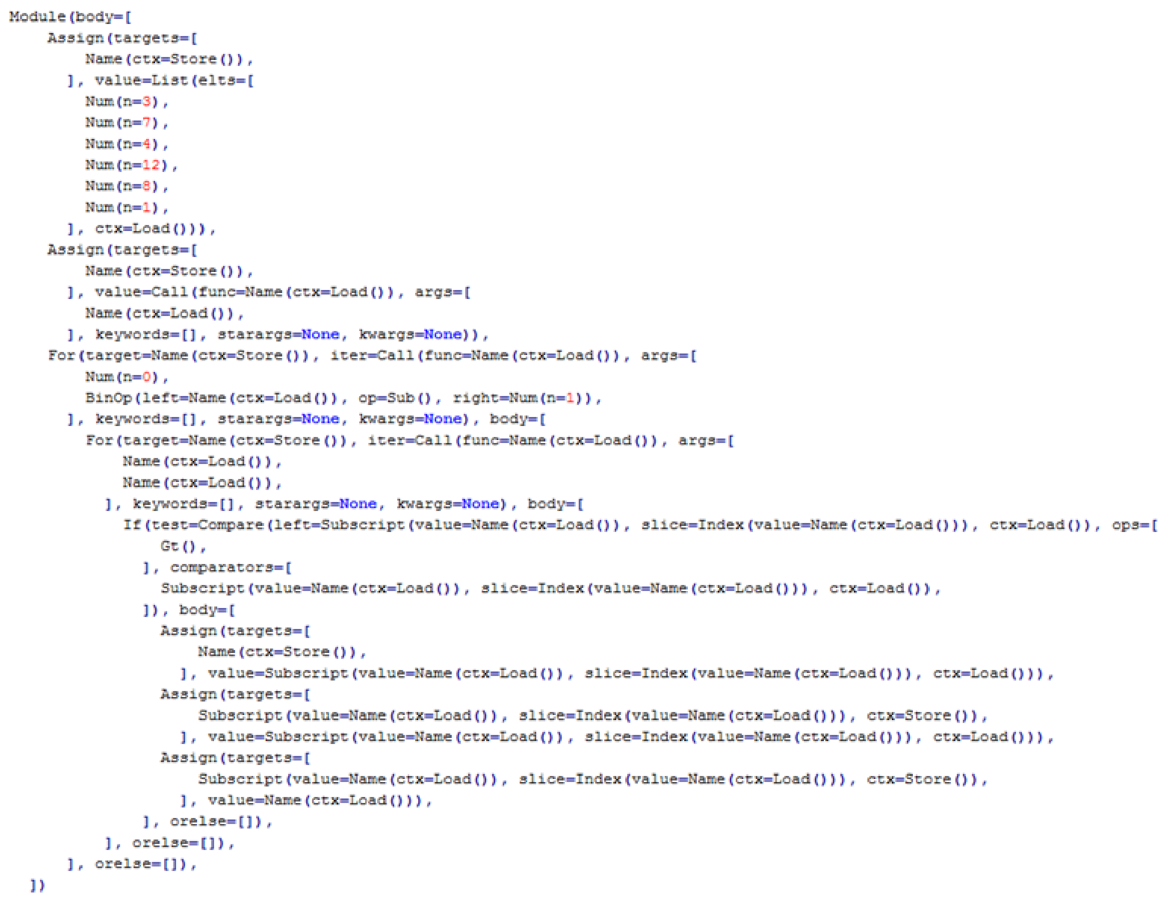
numbers can be literals or variable names which refer to numbers. Keywords are also defined, such as *for* and *if*. The rules which specify how programs must be constructed are the grammar for the language.

This syntax can be represented using a tree structure. This is called an AST. It is considered *abstract* because it does not show every detail, for example parentheses. Each element in the code is a node. Nodes can have children. For example, a for loop has the statements contained inside as children. Those statements may have children of their own.

The AST in Figure 3.2 has been extracted from the bubble sort program in the appendix (A.2.1). Ten lines of code have become a hundred nodes.

2.4 Diffs

Diff is a file comparison utility. The output is also referred to as a diff. There are various formats for this output, including "normal", "context" and "unified". The information



```
Module(body=[
    Assign(targets=[
        Name(ctx=Store()),
      ], value=List(elts=[
        Num(n=3),
        Num(n=7),
        Num(n=4),
        Num(n=12),
        Num(n=8),
        Num(n=1),
      ], ctx=Load())),
    Assign(targets=[
        Name(ctx=Store()),
      ], value=Call(func=Name(ctx=Load()), args=[
        Name(ctx=Load()),
      ], keywords=[], starargs=None, kwargs=None)),
    For(target=Name(ctx=Store()), iter=Call(func=Name(ctx=Load()), args=[
        Num(n=0),
        BinOp(left=Name(ctx=Load()), op=Sub(), right=Num(n=1)),
      ], keywords=[], starargs=None, kwargs=None), body=[
        For(target=Name(ctx=Store()), iter=Call(func=Name(ctx=Load()), args=[
            Name(ctx=Load()),
            Name(ctx=Load()),
          ], keywords=[], starargs=None, kwargs=None), body=[
            If(test=Compare(left=Subscript(value=Name(ctx=Load()), slice=Index(value=Name(ctx=Load())), ctx=Load()), ops=[
                Gt(),
              ], comparators=[
                Subscript(value=Name(ctx=Load()), slice=Index(value=Name(ctx=Load())), ctx=Load()),
              ]), body=[
                Assign(targets=[
                    Name(ctx=Store()),
                  ], value=Subscript(value=Name(ctx=Load()), slice=Index(value=Name(ctx=Load())), ctx=Load())),
                Assign(targets=[
                    Subscript(value=Name(ctx=Load()), slice=Index(value=Name(ctx=Load())), ctx=Store()),
                  ], value=Subscript(value=Name(ctx=Load()), slice=Index(value=Name(ctx=Load())), ctx=Load())),
                Assign(targets=[
                    Subscript(value=Name(ctx=Load()), slice=Index(value=Name(ctx=Load())), ctx=Store()),
                  ], value=Name(ctx=Load())),
              ], orelse=[]),
          ], orelse=[]),
      ], orelse=[]),
  ])
```

Fig. 2.4: AST extracted from bubble sort program

returned generally includes line numbers of the lines changed from one version to the other and the lines changed, inserted or deleted themselves. There is a different section for each file that has been changed, containing sets of changes called “hunks”. The surrounding context of changes may also be provided.

2.5 Dynamic Programming

DP is an algorithmic paradigm for solving problems which can be modelled as Directed Acyclic Graphs (DAGs) in this way. A large problem is broken into smaller ones. The smallest problems are solved first, each result bringing us closer to the solution. The nodes are the subproblems and the edges are dependencies between them. For example; to solve a subproblem X, we need to solve subproblem Y. This can be thought of as an edge between X and Y, and Y will be a smaller problem than X. Problems solvable by dynamic program must have the following properties:

- The problem can be divided into subproblems.
- The subproblems can be ordered.
- There exists a relation between the subproblems so that the answers to smaller subproblems (which are lower in the ordering) can be used to solve larger subproblems.

[Dasgupta et al., 2006]

2.5.0.1 DAGs

A collection of nodes and edges, such that edges connect one node to another in one direction only and that there is no loop (from any position, there is no sequence of edges that will take you back to where you started. This property means that directed acyclic graphs can be linearized (they can be arranged in a line so that all edges go left to right. [Dasgupta et al., 2006]

2.5.0.2 Shortest path

The shortest path to a node X is the minimum of the shortest paths to the nodes preceding it added the cost of moving from the shortest preceding path to X. This method can be used to determine the shortest path to any node. By computing the shortest paths for each node from left to right, the information required for the next computation has always been already computed earlier.

The algorithm solves a series of subproblems, beginning with the smallest and using each result to solve increasingly larger subproblems until the answer is reached. [Dasgupta et al., 2006]

2.5.0.3 String Edit Distance

The following explanation of string edit distance was informed by [Dasgupta et al., 2006].

DP can be used to find the edit distance between two strings. The edit distance is the smallest number of edits required to turn one string into the other. An edit can be an insertion, deletion or substitution. The edits required depend on how the strings have been aligned. There are many possible alignments, but DP can increase the efficiency of the search.

As mentioned previously, to solve a problem with DP, it must satisfy several properties. Suitable subproblems must be found. An example of a subproblem is calculating the edit distance between prefixes of the two strings. The edit cost for a subproblem is the minimum cost of the three edit operations. The answers to each subproblem is entered into a two-dimensional table. There is an underlying DAG which can be traced from the smallest subproblem in the upper left of the table (comparing the smallest prefixes) to the bottom right (comparing both entire strings). Each step along the DAG represents an edit operation.

Chapter 3

Related Work

This chapter discusses previous research related to this dissertation. The search for a good gamification metric is what lead to the current work. Various metrics were considered before a code similarity metric was selected. Analysis of current algorithms for comparing abstract syntax trees was carried out. Automating these processes is required for massively open online courses (MOOCs). Development environments will be key to gathering the data required for analysis.

3.1 Gamification

It has been observed that students became addicted to improving a *rules compliance score* [Dubois and Tamburrelli, 2013]. Badges have been shown to have very positive effects on student engagement [Denny, 2013] [Anderson et al., 2013]. Similar experimental approaches include showing students "skillometers" [Malacria et al., 2013], measuring engagement and completion speed [Lee et al., 2013] and making tasks more game-like through narrative and metaphors [Decker and Lawley, 2013] [O'Donovan et al., 2013]. Feedback from peers can lead to improved ideation [Toubia et al., 2006].

3.2 Code Assessment and Analysis

Code Assessor breaks programs up into blocks which are attempted one after the other by students. Code submissions provide instant feedback but each student may only submit code a limited number of times. In the case of failure, students are shown a working solution so that they still learn from the experience. [Zanden et al., 2012]

A system for testing student adherence to test-driven development has been explored [Buffardi and Edwards, 2012]. This needs to be done during the development process to ensure that students aren't writing the tests afterwards.

Code churn and its relationship to bugs being introduced to code is well understood at this point [Giger et al., 2011] [Munson and Elbaum, 1998]. There has also been experimentation into assessment tools that can instruct students on how to fix broken code through use of a knowledge base and leveraging the fact that student assignments typically use a specific subset of a programming language for each assignment [Singh et al., 2013].

3.3 Integrated Development Environments

Nooblab [Neve et al., 2012] attempts to fill in for the student/tutor dynamic which is usually missing in an online education environment. Tutors look over code during development and give feedback, assisting students who are stuck and including others to write better code. It is an IDE that provides a "learning area" beside the text editor. It logs the state of the code when the "run" button is pressed. It also makes a record of navigation commands through the learning area. The learning area is reactive to what the student is doing, much like a tutor. For example, if the student is stuck in a cycle of making small changes and running the program, while the error messages do not change, the learning area may prompt the user to investigate error meanings and debugging techniques.

3.4 Code Variability

Source code can differ in different ways, including syntactic differences and functional differences.

3.4.1 Functional Variability

Two programs are functionally similar when they achieve the same results, i.e. compute the same results. Unit testing can be used to determine how functionally similar two pieces of code are.

3.4.2 Syntactic Variability

There are different ways to measure how programs differ syntactically. It is suggested that string alignmnent algorithm can be used to compare strings contained in the source code. [Gitchell and Tran, 1999]

Sequences of API calls may give a good indication of the functionality of a program. Entire programs and be tokenized and the euclidian distance can be found between histograms based on word frequency. Abstract Syntax Trees are a very accurate representation of programs although comparisons between them are relatively slow. [Piech et al., 2012]

[Hartmann et al., 2010] tokenizes the code and uses lexical analysis to detect structures.

3.5 Tree Edit Distance

3.5.1 Overview

Comparing programs and giving the user a "uniqueness score" could be effective for gamification purposes. Programs represented in abstract syntax tree form retain syntactic structure. They can then be compared to other programs by calculating the tree edit distance between two trees. This metric is a number representing the cost of transforming one tree into another. Operations used in the transformation are insertion, deletion and re-labelling. Each of these may have a different cost associated with them.

3.5.2 History

Algorithms for tree edit distance have poor time complexity but they have been improving over time.

- An early attempt achieved both time and space complexity of $O(m^3n^3)$. This was the first non-exponential solution. [Tai, 1979]
- Improvements were made reaching time complexity of $O(n^2m^2)$ and space complexity of $O(mn)$. They always use the left decomposition rule, and observe that some subproblems can be disgarded. Partitioning subproblems leads to better space complexity. The efficiency depends on the tree shapes (it works well with balanced trees, for example). [Shasha et al., 1994]
- There were further advances with both time and space complexity of $O(n^2m \log m)$. The algorithm follows heavy paths in one of the trees. [Klein, 1998]
- A solution was found with time complexity $O(n^2m(1+\log \frac{m}{n})$ and space complexity $O(nm)$. Heavy paths are used in both trees. The space complexity is obtained by partitioning subproblems. It is worst case optimal. [Demaine et al., 2009]
- Time complexity of $O(n^3)$ (where $n > m$) and space complexity $O(mn)$ was later achieved. It manages to be shape-independent by precalculating the best decomposition strategies (between left, right and heavy). [Pawlik and Augsten, 2011]

3.5.3 Algorithm

3.5.4 Overview

To avoid complexity added by later algorithms, the version described here will in the 1998 paper [Shasha et al., 1994]. It concerns itself with the comparison of unordered trees (unordered meaning that the order of the siblings does not matter, but ancestor-descendant relationships are still important). The distance between trees is measured as the sum of the costs of all insertion, deletion and relabel operations on tree nodes.

In addition to being unordered, trees in the paper are considered to be rooted and labeled. Deleting a node X means giving its children to its parent and removing it, while inserting a node Y as a child of X will give a subset of the children of X to Y. A sequence of edit operations are needed to transform one tree into another. A cost function determines the cost of each edit operation. Edit operations can be represented as a mapping specifying a sequence of operations to be applied to nodes.

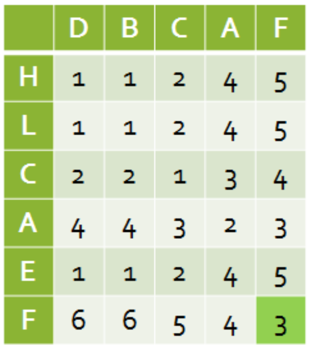
3.6 Tree Data

Before the tree edit distance can be calculated, three pieces of information are needed about each tree: A list of the nodes, they key roots and a dictionary mapping each node to its leftmost descendant.

3.6.1 Finding the edit distance

Create a table of integers with dimensions x and y, where x is the number of nodes in the first tree and y is the number of nodes in the second. Initialise all values to zero. The algorithm will update the values in this table and when it is finished, the result (the distance between the two trees) will be stored in position (x,y).

Processing is done on each pair of root nodes between the two trees. Each key root is considered the root of a subtree. Each subtree of the two trees is compared pairwise,



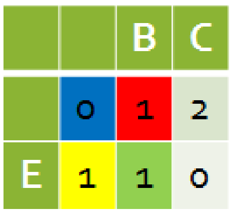
	D	B	C	A	F
H	1	1	2	4	5
L	1	1	2	4	5
C	2	2	1	3	4
A	4	4	3	2	3
E	1	1	2	4	5
F	6	6	5	4	3

Fig. 3.1: Table of costs between subtrees

updating the master table.

A new temporary table is used to compute the distance between each pair of subtrees. The dimensions of this table exceed the size of the subtrees by one, giving one extra row and column. The extra row and column are initialised to values ascending from zero.

Every pair of nodes between the two subtrees is then compared. The cost for an insertion, deletion or substitution are considered. If either node under consideration does not share a left most descendant with its respective roots, instead of the cost of substitution, the master table is consulted as a cost has been previously computed. Otherwise, the master table is updated.



		B	C
	0	1	2
E	1	1	0

Fig. 3.2: Inheriting nearby costs

The cost of performing an insertion, deletion or substitution is typically one, but if both nodes are the same the cost of the "substitution" is zero. These costs are added to the cost inherited from an a neighbouring cell in the table. Insert operations inherit the cost from above, deletes from the left. Substitute operations inherit from the upper left cell. The commented implementation can be seen in the appendix (B.1).

Chapter 4

Design

4.1 Incremental update strategies

Two strategies were designed for updating the distance between two trees after changes are made to one of them. In the following discussion, three trees will be referred to. $V1$ (version 1) is the original tree, and $V2$ (version 2) is a tree obtained after applying a sequence of changes to the first (these changes are the diff). O is a third tree which $V1$ is originally compared to using a standard tree edit distance algorithm. The goal is to find a way to then compute the distance between $V2$ and O in a more efficient way than using the standard technique, using the information obtained from the diff between $v1$ and $V2$.

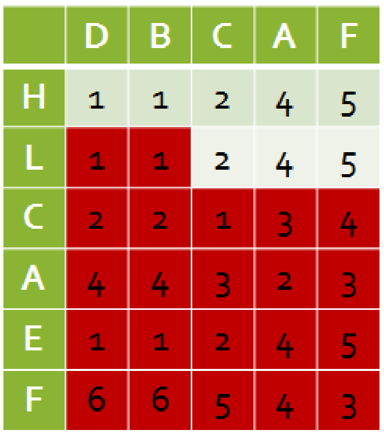
4.2 Building the edit sequence

The edit sequence is the sequence of changes that need to be made to change one tree into another. The original algorithm returns the shortest cost, a result that is reached by building a table of costs for subproblems and then inheriting them. This was changed so that each cell contained not only the cost, but the sequence of edits involved in reaching it. This can be applied to the diff, generating a list of the nodes that have been changed.

As before, insert operations inherit cost from above on the table, delete operations inherit from above and substitutions inherit from the upper left. With this new approach, the sequence is also inherited from the neighbour, and the chosen (minimum cost) edit is appended.

4.3 Distance Update

The idea behind this strategy is to reuse some of the previously done computation when finding the new distance. This is done by returning the main table instead of just the final cost. The goal is to take cells from this table if the values won't change, instead of recomputing them. The edit sequence between v1 and o is used to determine which cells can be reused and which need to be recomputed.



	D	B	C	A	F
H	1	1	2	4	5
L	1	1	2	4	5
C	2	2	1	3	4
A	4	4	3	2	3
E	1	1	2	4	5
F	6	6	5	4	3

Fig. 4.1: Invalidated cells

Before comparing two subtrees, a check happens to see if either subtree contains nodes which have been changed. The edit sequence can be inspected to find this information. If they do, the values must be recomputed. Also, inserting or deleting nodes change the numbering of all nodes which come after, so if either subtrees have been affected by this the values need to be recomputed as well. Otherwise, the result can be taken from the previous table.

4.4 Distance Inference

4.4.1 Strategy

The following is a proposed strategy for inferring changes in edit distance based on comparisons between edit sequences. This can be done by performing set operations between the edit sequences.

4.4.1.1 Diverging Changes

A basic inference strategy would be to assume that all edits between v1 and v2 are diverging further from o, and increasing the previously calculated distance. It would be an improvement if we could determine which edits are increasing the distance and which are making it smaller.

4.4.1.2 Converging changes

It is possible that the changes being made are making the programs more similar than they were before. This can be seen in Figure 4.2. v2 has alread had operation x applied to it, so the only remaining operation required is y, decreasing the distance. This result can naively be obtained by getting the symmetric difference between the sets.

4.4.1.3 Synonymous Sequences

It is possible for certain pairs of edits to to completely interchangeable with others. For example, consider the following edits:

$$\begin{matrix} k \to d \\ h \to None \end{matrix} \iff \begin{matrix} k \to None \\ h \to d \end{matrix}$$

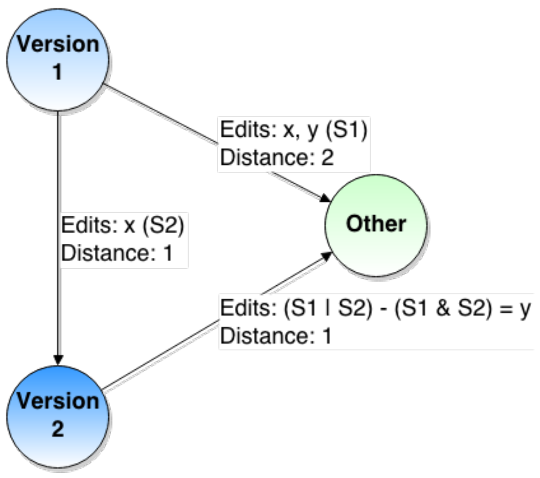


Fig. 4.2: Edit sequence inference

The net result of either of these sequences is that k and h are gone but there is a d. It needs to be recognised that edits like these mean programs are still converging. This can be done by replacing substitution operations with pairs of insert and delete. $k \to d$ becomes $k \to None$ and $None \to d$. If we expand the previous example in this way we get this:

$$\begin{array}{ccc} k \to None & & k \to None \\ None \to d & \Longleftrightarrow & h \to None \\ h \to None & & None \to d \end{array}$$

The final solution is to get the length of the union of both sequences and subtract the length of the common elements (after we have replaced substitutes). A possible concern with this approach is the time it will take to extract a sequence of edits from a diff. We have already seen how long it can take to calculate TED on large trees.

Chapter 5

Methodology

The goal of the research is to design methods of updating TED using the diff generated from code changes and to experiment on the effectiveness of those methods.

5.1 Algorithms tested

There are three algorithms to test. Each algorithm is implemented in Python and the code listing can be found in the appendix (B.1) The first is the TED algorithm described by [Shasha et al., 1994]. The other two are Distance Update and Distance Inference, which are described in the Design chapter. These two make use of a diff between two versions of a program. Comparisons will be made between the speed of the standard algorithm and that of Distance Update. The percentage of previous result reuse of Distance Update will also be measured. The accuracy of Distance Inference will be investigated.

5.2 Input Data

The algorithms will be tested on both trees and diffs of varying magnitude. Two types of tree are used. Trees of both types were used to investigate the time complexity of TED implementations. Timings were compared between the original implementation and the Distance Update version.

5.2.1 Types of tree

5.2.1.1 Simple data set

Instances of the first type of tree are constructed from a Node class. A node contains a label and a list of child nodes (which will be empty if the node is a leaf). The implementation of this class can be found in the appendix. Six of these trees were created, containing between five and seven nodes each. Labels are single letters. A listing of these trees can be found in the appendix.

To test both Distance Update and Distance Inference, the trees were combined into sets of (v1, v2, o), where v1 and v2 are two versions of the same program and o is the other program they are being compared to. The percentage error of the Distance Inference strategy was recorded for each combination, as was the percentage of computation saved with the Distance Update strategy.

5.2.1.2 Real-world data set

Instances of the second type of tree are ASTs generated by the ast module in Python. Each of the ASTs were parsed from Python source code stored in individual Python files. Nodes subclass the AST class, and have both individual children and lists of children as attributes. Three different scenarios were devised for testing. Each scenario has v1, v2 and other.

The first is a comparison between sorting algorithms (bubble and selection sort). The second compares two functions for extracting metadata from trees (the lists of nodes and keyroots and the mapping between nodes and LMDs). The third compares two functions for finding the TED between two trees. All of these pieces of code can be viewed in the appendix, along with supplementary information such as the number of lines of code, the number of nodes and the size and nature of the diff between versions.

The percentage error of the Distance Inference strategy was recorded for each scenario, as was the percentage of computation saved with the Distance Update strategy. The details of each program can be found in the appendix, along with the number of lines of code and the number of nodes in the AST once extracted.

5.2.1.3 Handling both types of tree

The implementation of the TED algorithm doesn't need to be drastically changed to handle both of these types of tree. During the metadata collection stage, a function can be passed in as an argument which gets a list of all of the children of a node. A node comparison function should also be provided, as custom nodes are compared by label while ast nodes are compared by class name.

5.3 Hardware Configuration

All experiments carried out on a machine with the following specifications:

- CPU: Intel Core 2 Duo SU7300 (1.3GHz@1.6GHz, 800 MHz, 3 MB)
- RAM: 4GB
- Motherboard: Mobile Intel GS45 Chipset
- Operating system: Lubuntu 12.04

Chapter 6

Results

The combinations and scenarios discussed here can be found in the appendix, along with supplementary information such as the number of nodes trees have and the number of nodes changed from one version to the next.

6.1 Timing

The standard algorithm was given examples of each of the problem sizes available to investigate its performance on the test machine, and to get a clear idea of how computation time scaled as the size of the input increased. The resulting graph behaved as expected, given the time complexity of the algorithm ($O(n^2m^2)$). See Figure 6.1.

It is evident that comparing trees with less than ten nodes is trivial. The time taken was acually 2 milliseconds. Programs that contain about ten Lines of Code (LOC) (100 nodes) are still quite fast to compare, at only three seconds. Five hundred nodes (20 LOC) takes twenty seconds to compare and a thousand nodes (only forty LOC!) can take over four minutes.

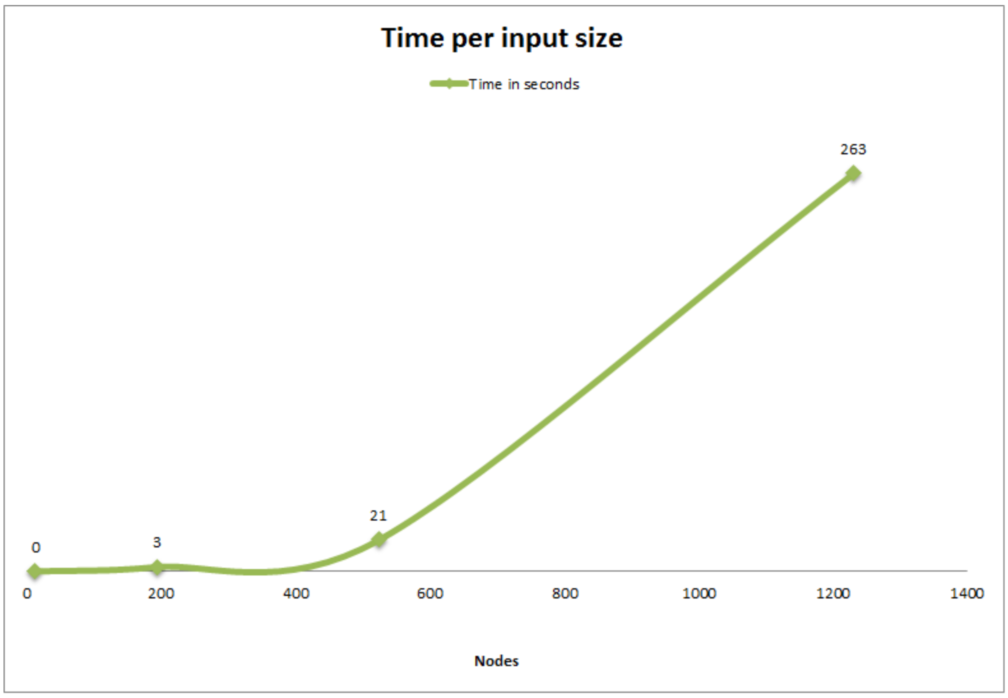


Fig. 6.1: Tree Edit Distance performance

6.2 Building the edit sequence

Unfortunately, the overhead involved in building the sequence of edits is quite severe. This has an effect on both the Distance Update and Distance Inference strategies. Figure 6.2 shows how the time taken by the modified algorithm compares to the original.

The time taken to process small programs of ten LOC goes from three seconds to eleven. The most staggering jump can be seen for programs with LOC.

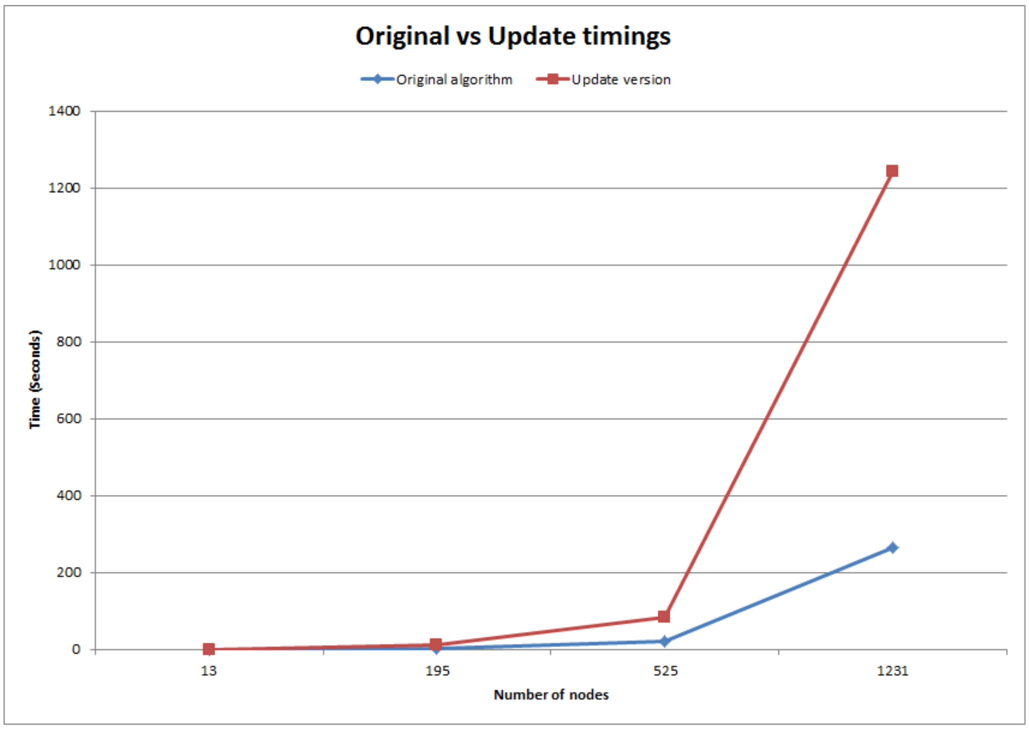


Fig. 6.2: Overhead of building the edit sequence

6.3 Incremental update strategies

6.3.1 Distance Update

The accuracy of this algorithm was tested by comparing the results to those generated by the standard implementation. Not only was the final cost examined, but ever cell generated in the distance table was compared. The Python assert statement was used to ensure that perfect accuracy was maintained.

Unfortunately, in most of the combinations of simple trees tested, there was no computation saved. In a few cases there were modest savings of about ten and thirty percent. This can be seen in Figure 6.3.

When applied to the larger ASTs, the savings were always present but even smaller.

This can be seen in Figure 6.4.

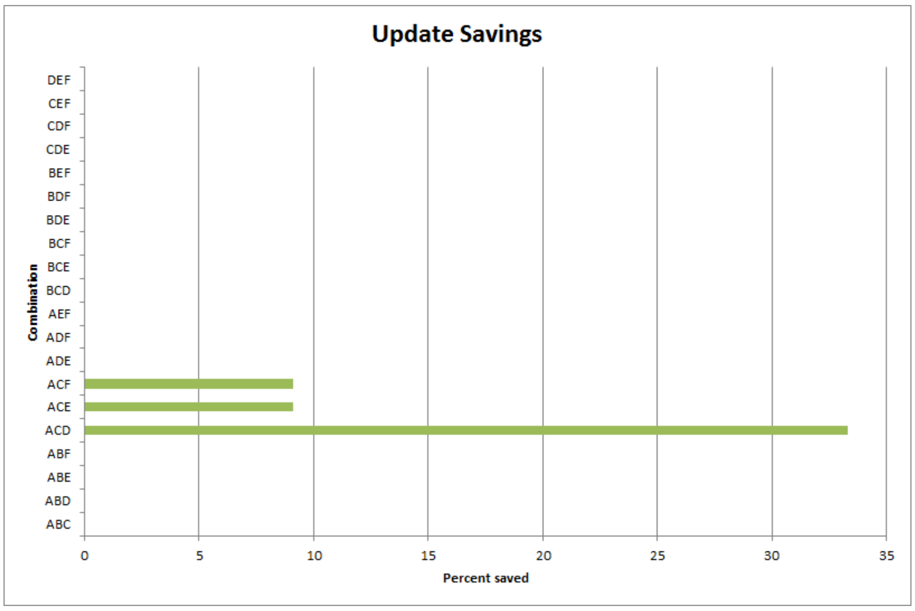


Fig. 6.3: Computation saved for different combinations

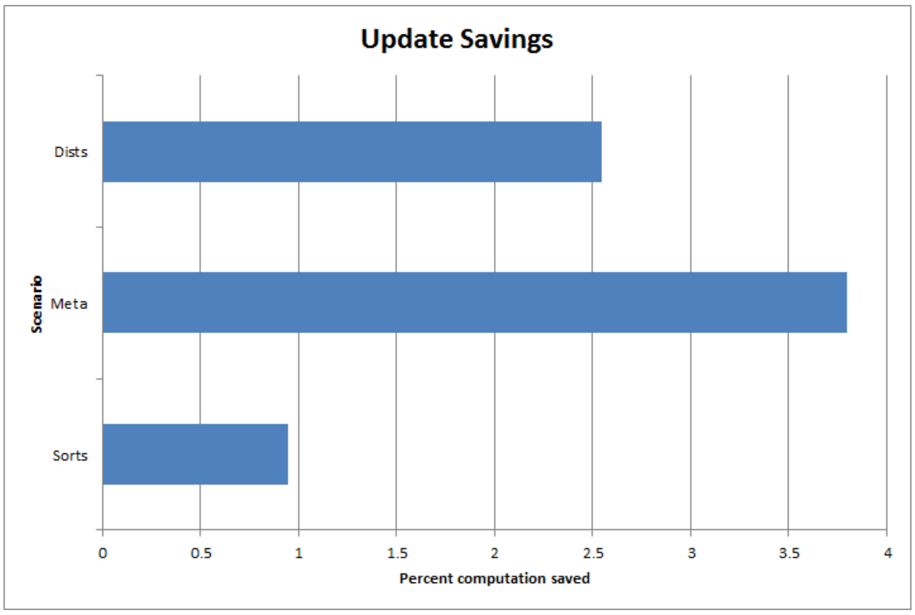


Fig. 6.4: Computation saved in different scenarios

6.3.2 Edit Sequence Inference

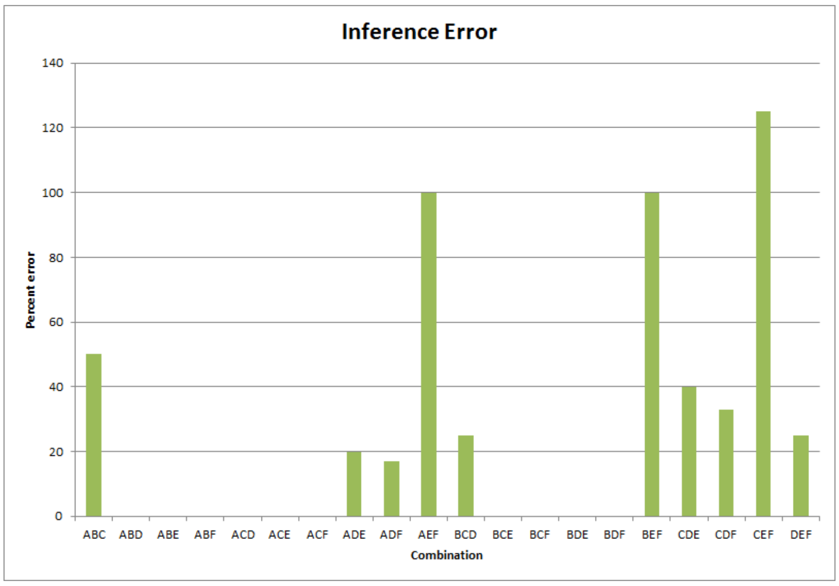


Fig. 6.5: Percentage error for different combinations

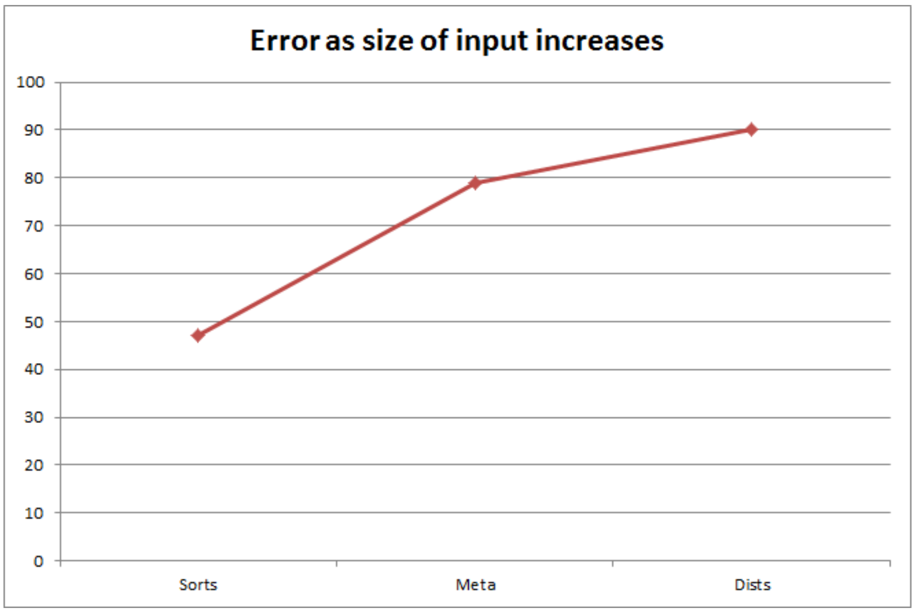


Fig. 6.6: Percentage error in different scenarios

When applied to the simple trees, it can be seen that in many cases the inferred distance was absolutely accurate. Some of the errors were quite large, even going over one hundred percent error in one case. This can be seen in Figure 6.5.

When tested on the larger ASTs, the margin of error increased along with the problem size, and the percentage of error was not really acceptable. It fell between about fifty and a hundred percent. This can be seen in Figure 6.6.

Chapter 7

Conclusion

7.1 Timing considerations

7.1.1 Original algorithm

The initial idea was to be able to update distances between different programs in real time, as changes are made in an IDE. There was an assumption that it would need to reach a certain level of responsiveness or reactiveness to successfully motivate users to alter behaviour.

The timing results show that programs with ten LOC can be compared in about three seconds, using the standard implementation. Programs with twenty LOC can take up to twenty seconds. This would probably be the latest permissable time to wait for a result. Programs of a greater size than this take much too long to compare.

7.1.2 Building the edit sequence

The time taken to generate the edit sequence poses a significant problem for the Distance Update strategy, and has the potential to cause problems for the Distance Inference problem.

7.1.2.1 Distance Update

Comparing small programs with ten LOC can take ten seconds, which is of debatable responsiveness. Bigger programs cannot be compared quickly enough in this way. These problems mean that for Distance Update strategy, the percentage of computation saved needs to be high enough to offset this overhead, and preferably surpass it.

7.1.2.2 Distance Inference

Distance Inference relies on building the edit sequence for both programs once, and then for the diff thereafter. This means that when starting to use this strategy, the startup cost could be very expensive. This wouldn't be a problem if the strategy is employed from the very beginning, when the programs being compared are still small. If the application is still to be responsive, it is important that the diff does not contain a lot of nodes. The time taken to infer a new distance is negligable because it consists of a couple of set operations, but if there have been ten lines of code changed, building that edit sequence could take ten seconds. To mitigate this, changes should be detected automatically and often. Preferably the distance would be updated once every time a line is changed, deleted or removed.

7.2 Distance Inference Accuracy

In the context of gamification, accuracy doesn't need to be perfect. However, the better it is, the more believable users will find it. Users are unlikely to be influenced by a metric which is clearly wrong. Keeping this in mind, the results are less than ideal. In many cases with the small trees, accuracy is perfect. There are a few cases in which the error may be tolerable (between twenty and forty percent). Often, at least the direction is correctly deduced (programs becoming more or less similar as a result of changes). However, there are a few severe spikes which could break the suspension of disbelief of users. When tested on larger problem sizes, it appears that there is a trend towards larger problem sizes and decreased accuracy.

In light of these observations, it is advised that improvements still need to be made to the intelligence of the Distance Inference strategy. In its current form, it could still be used to investigate the use of a code difference metric for gamification.

7.3 Distance Update Savings

Unfortunately the percentage of compututation saved is very low. Combined with the previously mentioned overhead, there is currently no reason to use this approach. However, if the overhead could be reduced to acceptable levels, even the small savings discovered could be useful in certain domains. It is recommended that it should be abandoned for the gamification use case. Even if 50% of computation could be saved, the problem still wouldnt scale. Large programs would still take too long to compare.

7.4 Future Work

It still remains to investigate the effects of using a code metric for gamification purposes. To facilitate this, it is recommended that an online IDE is modified to send code or code changes away regularly so that the distance can be computed or updated and to dsiplay the result as raw data or as a visualization. Once this has been carried out, the results of using this metric for gamification can be examined. There is also an open question as to the scalability of using calculating this metric regularly for large numbers of participants.

There are two immediate suggestions for improving the Distance Update strategy. First of all, The overhead of discovering which nodes have been changed needs to be reduced. It may be possible to extract this information from the lists of nodes instead of building the edit sequence, which would be a huge improvement.

Secondly, changes on the left of the tree have a more severe effect on the reuse available because the tree is annotated from left to right. It is possible that reacting to the position of changes by changing the order of nodes could be beneficial.

Bibliography

[Anderson et al., 2013] Anderson, A., Huttenlocher, D., Kleinberg, J., and Leskovec, J. (2013). Steering user behavior with badges. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 95–106, Republic and Canton of Geneva, Switzerland. International World Wide Web Conferences Steering Committee.

[Buffardi and Edwards, 2012] Buffardi, K. and Edwards, S. H. (2012). Exploring influences on student adherence to test-driven development. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 105–110, New York, NY, USA. ACM.

[Dasgupta et al., 2006] Dasgupta, S., Papadimitriou, C., and Vazirani, U. (2006). Dynamic programming. In *Algorithms*. McGraw-Hill.

[Decker and Lawley, 2013] Decker, A. and Lawley, E. L. (2013). Life's a game and the game of life: How making a game out of it can change student behavior. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 233–238, New York, NY, USA. ACM.

[Demaine et al., 2009] Demaine, E. D., Mozes, S., Rossman, B., and Weimann, O. (2009). An optimal decomposition algorithm for tree edit distance. *ACM Trans. Algorithms*, 6(1):2:1–2:19.

[Denny, 2013] Denny, P. (2013). The effect of virtual achievements on student engagement. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 763–772, New York, NY, USA. ACM.

[Dubois and Tamburrelli, 2013] Dubois, D. J. and Tamburrelli, G. (2013). Understanding gamification mechanisms for software development. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 659–662, New York, NY, USA. ACM.

[Giger et al., 2011] Giger, E., Pinzger, M., and Gall, H. C. (2011). Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 83–92, New York, NY, USA. ACM.

[Gitchell and Tran, 1999] Gitchell, D. and Tran, N. (1999). Sim: A utility for detecting similarity in computer programs. *SIGCSE Bull.*, 31(1):266–270.

[Hartmann et al., 2010] Hartmann, B., MacDougall, D., Brandt, J., and Klemmer, S. R. (2010). What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028, New York, NY, USA. ACM.

[Henderson and Johnson, 2013] Henderson, T. and Johnson, S. (2013). Zhang-shasha: Tree edit distance in python. `https://github.com/timtadh/zhang-shasha`.

[Klein, 1998] Klein, P. N. (1998). Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, pages 91–102, London, UK, UK. Springer-Verlag.

[Lee et al., 2013] Lee, M. J., Ko, A. J., and Kwan, I. (2013). In-game assessments increase novice programmers' engagement and level completion speed. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 153–160, New York, NY, USA. ACM.

[Malacria et al., 2013] Malacria, S., Scarr, J., Cockburn, A., Gutwin, C., and Grossman, T. (2013). Skillometers: Reflective widgets that motivate and help users to improve performance. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 321–330, New York, NY, USA. ACM.

[Munson and Elbaum, 1998] Munson, J. C. and Elbaum, S. G. (1998). Code churn: A measure for estimating the impact of code change. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 24–, Washington, DC, USA. IEEE Computer Society.

[Neve et al., 2012] Neve, P., Hunter, G., Livingston, D., and Orwell, J. (2012). Nooblab: An intelligent learning environment for teaching programming. In *Proceedings of the The 2012 IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technology - Volume 03*, WI-IAT '12, pages 357–361, Washington, DC, USA. IEEE Computer Society.

[Nguyen et al., 2014] Nguyen, A., Piech, C., Huang, J., and Guibas, L. (2014). Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 491–502, Republic and Canton of Geneva, Switzerland. International World Wide Web Conferences Steering Committee.

[O'Donovan et al., 2013] O'Donovan, S., Gain, J., and Marais, P. (2013). A case study in the gamification of a university-level games development course. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '13, pages 242–251, New York, NY, USA. ACM.

[Pawlik and Augsten, 2011] Pawlik, M. and Augsten, N. (2011). Rted: A robust algorithm for the tree edit distance. *Proc. VLDB Endow.*, 5(4):334–345.

[Piech et al., 2012] Piech, C., Sahami, M., Koller, D., Cooper, S., and Blikstein, P. (2012). Modeling how students learn to program. In *Proceedings of the 43rd ACM*

Technical Symposium on Computer Science Education, SIGCSE '12, pages 153–160, New York, NY, USA. ACM.

[Shasha et al., 1994] Shasha, D., Wang, J. T.-L., Zhang, K., and Shih, F. Y. (1994). Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):668–678.

[Singh et al., 2013] Singh, R., Gulwani, S., and Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA. ACM.

[Tai, 1979] Tai, K.-C. (1979). The tree-to-tree correction problem. *J. ACM*, 26(3):422–433.

[Toubia et al., 2006] Toubia, O., Frederick, S., Prelec, D., Ariely, D., Wernerfelt, B., Gibbs, B., and Weaver, R. (2006). Idea generation, creativity, and incentives. *Marketing Science*, pages 411–425.

[Zanden et al., 2012] Zanden, B. V., Anderson, D., Taylor, C., Davis, W., and Berry, M. W. (2012). Codeassessor: An interactive, web-based tool for introductory programming. *J. Comput. Sci. Coll.*, 28(2):73–80.

Appendix A

Input

A.1 Simple trees

```
A = (
    Node("f")
        .add_child(Node("a")
            .add_child(Node("h"))
            .add_child(Node("c")
                .add_child(Node("l"))))
        .add_child(Node("e"))
    )

B = (
    Node("f")
        .add_child(Node("a")
            .add_child(Node("h")
                .add_child(Node("k")))
            .add_child(Node("c")
                .add_child(Node("l"))))
        .add_child(Node("e"))
    )
```

```
C = (
    Node("k")
        .add_child(Node("a")
            .add_child(Node("h"))
            .add_child(Node("c")
                .add_child(Node("l"))))
        .add_child(Node("e"))
    )

D = (
    Node("f")
        .add_child(Node("a")
            .add_child(Node("d"))
            .add_child(Node("c")
                .add_child(Node("b"))))
    )

E = (
    Node("f")
        .add_child(Node("a")
            .add_child(Node("h"))
            .add_child(Node("c"))
            .add_child(Node("w")
                .add_child(Node("l"))))
        .add_child(Node("d"))
    )

F = (
    Node("f")
        .add_child(Node("a")
            .add_child(Node("h"))
                .add_child(Node("l")))
        .add_child(Node("d"))
            .add_child(Node("c"))
            .add_child(Node("w"))
    )
```

A.2 Sort programs

A.2.1 Bubble sort

- LOCs: 10
- Nodes: 94

```
print "A sorting algorithm"

xs = [3,7,4,12,8,1]
n = len(xs)

for i in xrange(0, n-1):
    for j in xrange(i, n):
        if xs[i] > xs[j]:
            temp = xs[i]
            xs[i] = xs[j]
            xs[j] = temp

print xs
```

A.2.2 Selection sort

- LOCs: 12
- Nodes: 101

```
print "A sorting algorithm"

xs = [3,7,4,12,8,1]
n = len(xs)
```

```
for i in xrange(0, n):
    for j in xrange(i, n):
        m = i
        if xs[j] < xs[m]:
            m = j

    temp = xs[i]
    xs[i] = xs[m]
    xs[m] = temp

print xs
```

A.2.2.1 Diff

```
--- bubble.py   2014-05-11 12:58:10.716113716 +0100
+++ bubble2.py  2014-05-14 23:58:16.784466634 +0100
 -1,13 +1,14
 print "A sorting algorithm"

 xs = [3,7,4,12,8,1]
-n = len(xs)
+n = xs[2]

-for i in xrange(0, n-1):
+for i in xrange(0, n):
     for j in xrange(i, n):
         if xs[i] > xs[j]:
             temp = xs[i]
-            xs[i] = xs[j]
+            temp+=2
             xs[j] = temp
+            xs[i] = temp-1
```

```
print xs
```

A.3 Tree metadata-extraction programs

A.3.1 Implementation A

[Henderson and Johnson, 2013]

- LOCs: 43
- Nodes: 331

```
class AnnotatedTree(object):

    def __init__(self, root, get_children):
        self.get_children = get_children

        def setid(n, _id):
            setattr(n, "_id", _id)
            return n

        self.root = root
        self.nodes = list()  # a pre-order enumeration of the nodes in
             the tree
        self.lmds = list()  # left most descendents
        self.keyroots = None

        stack = list()
        pstack = list()
        stack.append((root, collections.deque()))
        j = 0
        while len(stack) > 0:
            n, anc = stack.pop()
            setid(n, j)
            for c in self.get_children(n):
                a = collections.deque(anc)
                a.appendleft(n._id)
                stack.append((c, a))
            pstack.append((n, anc))
            j += 1
        lmds = dict()
        keyroots = dict()
        i = 0
        while len(pstack) > 0:
            n, anc = pstack.pop()
            self.nodes.append(n)
            if not self.get_children(n):
                lmd = i
                for a in anc:
                    if a not in lmds: lmds[a] = i
                    else: break
            else:
                try: lmd = lmds[n._id]
                except:
                    import pdb
                    pdb.set_trace()
            self.lmds.append(lmd)
            keyroots[lmd] = i
            i += 1
        self.keyroots = sorted(keyroots.values())
```

A.3.2 Implementation B

- LOCs: 20
- Nodes: 194

```
class TreeData:
    def __init__(self, root):
        self.nodes = list()
        self.keyroots = list()
        self.lmds = dict()
        self.build_lists(root)
        self.keyroots.append(len(self.nodes)-1)

    #Generate postorder lists of nodes and keyroots, determine lmd of
        each node
    def build_lists(self, node):
        #The postorder enumeration can be found by inspecting the
            length of the list of nodes.
        def id(): return len(self.nodes)

        children = list(iter_child_nodes(node))

        #Leaf node case
        if not children:
            self.lmds[id()] = id()
        else:
            self.build_lists(children[0])
            #Descend left to find lmd, pass result back up to top
            lmd = self.lmds[id()-1] #id-1 is currently the index of
                the left most child

            #Process the other child nodes
            for i in xrange(1, len(children)):
                self.build_lists(children[i])
                self.keyroots.append(id()-1) #id-1 index of current
                    child

            self.lmds[id()] = lmd
        self.nodes.append(node)
```

A.3.3 Diff

```
--- mydata.py   2014-05-13 12:19:36.574185135 +0100
+++ mydata2.py  2014-05-15 11:26:17.212533127 +0100
 -23,9 +23,9
             #id-1 is currently the index of the left most child

             #Process the other child nodes
-            for i in xrange(1, len(children)):
-                self.build_lists(children[i])
-                self.keyroots.append(id()-1)
+            for i, e in enumerate(1, len(children)):
+                self.build_lists(e)
+                self.keyroots.append(id()-1)
                 #id-1 index of current child

             self.lmds[id()] = lmd
```

A.4 Tree metadata-extraction programs

A.4.1 Implementation A

[Henderson and Johnson, 2013]

- LOCs: 40
- Nodes: 674

```
def distance(A, B, get_children, insert_cost, remove_cost, update_cost
   ):
    A, B = AnnotatedTree(A, get_children), AnnotatedTree(B,
        get_children)
    treedists = zeros((len(A.nodes), len(B.nodes)), int)

    def treedist(i, j):
```

```
Al = A.lmds
Bl = B.lmds
An = A.nodes
Bn = B.nodes

m = i - Al[i] + 2
n = j - Bl[j] + 2
fd = zeros((m,n), int)

ioff = Al[i] - 1
joff = Bl[j] - 1

for x in xrange(1, m):  #   (l(i1)..i,    ) =   (l(1i)..1-1,
   ) +   (v        )
    fd[x][0] = fd[x-1][0] + remove_cost(An[x-1])
for y in xrange(1, n): #   (   , l(j1)..j) =   (   , l(j1)..j-1)
    +   (        w)
    fd[0][y] = fd[0][y-1] + insert_cost(Bn[y-1])

for x in xrange(1, m): ## the plus one is for the xrange impl
    for y in xrange(1, n):
        # only need to check if x is an ancestor of i
        # and y is an ancestor of j
        if Al[i] == Al[x+ioff] and Bl[j] == Bl[y+joff]:
            #                   +-
            #                   |   (l(i1)..i-1, l(j1)..j) +
                 (v        )
            #   (F1 , F2 ) = min-+   (l(i1)..i , l(j1)..j-1) +
                  (        w)
            #                   |   (l(i1)..i-1, l(j1)..j-1) +
                  (v      w)
            #                   +-
            fd[x][y] = min(
                fd[x-1][y] + remove_cost(An[x+ioff]),
                        fd[x][y-1] + insert_cost(Bn[y+joff]),
                        fd[x-1][y-1] + update_cost(An[x+ioff], Bn[y+
                            joff]),
                    )
                    treedists[x+ioff][y+joff] = fd[x][y]
                else:
                    #                     +-
                    #                     |    (l(i1)..i-1, l(j1)..j) +
                         (v        )
                    #    (F1  , F2  ) = min-+    (l(i1)..i  , l(j1)..j-1) +
                          (        w)
                    #                     |    (l(i1)..l(i)-1, l(j1)..l(j
                        )-1)
                    #                     |                          +
                        treedist(i1,j1)
                    #                     +-
                    p = Al[x+ioff]-1-ioff
                    q = Bl[y+joff]-1-joff
                    #print (p, q), (len(fd), len(fd[0]))
                    fd[x][y] = min(
                        fd[x-1][y] + remove_cost(An[x+ioff]),
                        fd[x][y-1] + insert_cost(Bn[y+joff]),
                        fd[p][q] + treedists[x+ioff][y+joff]
                    )

    print "Keyroots:"
    print A.keyroots
    print B.keyroots


    for i in A.keyroots:
        for j in B.keyroots:
            treedist(i,j)
    return treedists[-1][-1]
```

A.4.2 Implementation B

- LOCs: 29
- Nodes: 557

```
#return tree edit distance between two trees, given the roots
def tredit(a, b):
    #Update distances based on subtrees with roots x and y
    def subtree_distance(x, y):
        #Sizes of subtrees p and q with roots x and y
        size_p = x - a.lmds[x] + 1
        size_q = y - b.lmds[y] + 1

        #sub-table for subtree distances (Need extra row and column).
        sd = numpy.zeros((size_p+1, size_q+1), int)

        #Initialize extra row and column
        for i in xrange(1, size_p+1):
            sd[i][0] = sd[i-1][0] + 1
        for i in xrange(1, size_q+1):
            sd[0][i] = sd[0][i-1] + 1

        #For each pair of nodes between p and q
        for np_index, np in enumerate(xrange(a.lmds[x], x+1)):
            i = np_index + 1 #skip extra row
            for nq_index, nq in enumerate(xrange(b.lmds[y], y+1)):
                j = nq_index + 1 #skip extra column

                delete = sd[i-1][j] + 1
                insert = sd[i][j-1] + 1
                #if substitute cost has yet to be computed
                if a.lmds[np] == a.lmds[x] and b.lmds[nq] == b.lmds[y]:
                    substitute = sd[i-1][j-1] + substitute_cost(a.nodes[np], b.nodes[nq])
                    sd[i][j] = min(insert, delete, substitute)
                    treedists[np][nq] = sd[i][j]

                #lookup treedistance table for previous result
                else:
                    m = a.lmds[np]-a.lmds[x]
                    n = b.lmds[nq]-b.lmds[y]
                    substitute = sd[m][n] + treedists[np][nq]
                    sd[i][j] = min(insert, delete, substitute)

    treedists = numpy.zeros((len(a.nodes), len(b.nodes)), int)

    #for each keyroot pair between a and b, update treedists
    for i in a.keyroots:
        for j in b.keyroots:
            subtree_distance(i, j)

    return treedists[-1][-1]
```

A.4.3 Diff

```
--- mydist.py   2014-05-13 12:11:40.467470071 +0100
+++ mydist2.py  2014-05-15 11:28:25.586077684 +0100
 -23,13 +23,10
                 j = nq_index + 1
                 #skip extra column

-                delete = sd[i-1][j] + 1
-                insert = sd[i][j-1] + 1
```

```
-
                    #if substitute cost has yet to be computed
                    if a.lmds[np] == a.lmds[x] and b.lmds[nq] == b.lmds[y
                        ]:
                        substitute = sd[i-1][j-1] + substitute_cost(a.
                            nodes[np], b.nodes[nq])
-                       sd[i][j] = min(insert, delete, substitute)
+                       sd[i][j] = min(sd[i][j-1] + 1, sd[i-1][j] + 1,
    substitute)
                        treedists[np][nq] = sd[i][j]

                    #lookup treedistance table for previous result
 -37,7 +34,7
                        m = a.lmds[np]-a.lmds[x]
                        n = b.lmds[nq]-b.lmds[y]
                        substitute = sd[m][n] + treedists[np][nq]
-                       sd[i][j] = min(insert, delete, substitute)
+                       sd[i][j] = min(sd[i][j-1] + 1, sd[i-1][j] + 1,
    substitute)

      treedists = numpy.zeros((len(a.nodes), len(b.nodes)), int)
```

Appendix B

Code Listing

B.1 Tree-edit Distance implementations

B.1.1 Implementation of standard tree edit distance

```python
#return tree edit distance between two trees, given the roots
def tredit(a, b, getlab):
    #Update distances based on subtrees with roots x and y
    def subtree_distance(x, y):
        #Sizes of subtrees p and q with roots x and y
        size_p = x - a.lmds[x] + 1
        size_q = y - b.lmds[y] + 1

        #sub-table for subtree distances (Need extra row and column).
        sd = numpy.zeros((size_p+1, size_q+1), int)

        #Initialize extra row and column
        for i in xrange(1, size_p+1):
            sd[i][0] = sd[i-1][0] + 1
        for i in xrange(1, size_q+1):
            sd[0][i] = sd[0][i-1] + 1
        #For each pair of nodes between p and q
        for np_index, np in enumerate(xrange(a.lmds[x], x+1)):
            i = np_index + 1 #skip extra row
            for nq_index, nq in enumerate(xrange(b.lmds[y], y+1)):
                j = nq_index + 1 #skip extra column

                delete = sd[i-1][j] + 1
                insert = sd[i][j-1] + 1

                #if substitute cost has yet to be computed
                if a.lmds[np] == a.lmds[x] and b.lmds[nq] == b.lmds[y
                    ]:
                    substitute = sd[i-1][j-1] + substitute_cost(a.
                        nodes[np], b.nodes[nq], getlab)
                    sd[i][j] = min(insert, delete, substitute)
                    treedists[np][nq] = sd[i][j]

                #lookup treedistance table for previous result
                else:
                    m = a.lmds[np]-a.lmds[x]
                    n = b.lmds[nq]-b.lmds[y]
                    substitute = sd[m][n] + treedists[np][nq]
                    sd[i][j] = min(insert, delete, substitute)

    treedists = numpy.zeros((len(a.nodes), len(b.nodes)), int)

    #for each keyroot pair between a and b, update treedists
    for i in a.keyroots:
        for j in b.keyroots:
            subtree_distance(i, j)

    return treedists[-1][-1]
```

B.1.2 Distance Update

(also building the edit sequence)

```
#return tree edit distance between two trees, given the roots
def tredit_update(a, b, getlab, affected=None, prev_results=None, ):
    def nodes_affected(i, j):
        #If numbering changed by insertion or deletion
        insert_cond = i >= ch.first_insert if ch.first_insert != None
            else False
        delete_cond = j >= ch.first_delete if ch.first_delete != None
            else False
        #insert_cond = i >= len(b.nodes)
        #delete_cond = j > len(a.nodes)

        #Or node affected by relabeling
        set1_affected = Set(ch.a_data) & Set(xrange(a.lmds[i], i+1))
        set2_affected = Set(ch.b_data) & Set(xrange(b.lmds[j], j+1))

        return set1_affected or set2_affected or insert_cond or
            delete_cond

    #Update distances based on subtrees with roots x and y
    def subtree_distance(x, y):
        #Sizes of subtrees p and q with roots x and y
        size_p = x - a.lmds[x] + 1
        size_q = y - b.lmds[y] + 1

        #sub-table for subtree distances (Need extra row and column).
        tp = numpy.empty((size_p+1, size_q+1), dtype=object)
        tp[:][:] = Path()



        #Initialize extra row and column
        for i in xrange(1, size_p+1):
```

```
        tp[i][0] = Path(tp[i-1][0], del_op(i-1))
    for i in xrange(1, size_q+1):
        tp[0][i] = Path(tp[0][i-1], ins_op(i-1))

    #print tabulate(costTable(tp))

    #For each pair of nodes between p and q
    for np_index, np in enumerate(xrange(a.lmds[x], x+1)):
        i = np_index + 1 #skip extra row
        for nq_index, nq in enumerate(xrange(b.lmds[y], y+1)):
            j = nq_index + 1 #skip extra column

            del_cost = Path(tp[i-1][j], del_op(np))
            ins_cost = Path(tp[i][j-1], ins_op(nq))

            #if substitute cost has yet to be computed
            if a.lmds[np] == a.lmds[x] and b.lmds[nq] == b.lmds[y
                ]:
                sub_cost = Path(tp[i-1][j-1], sub_op(np, nq, a, b,
                     getlab))
                tp[i][j] = min(ins_cost, del_cost, sub_cost, key=
                    attrgetter('cost'))
                treepaths[np][nq] = tp[i][j]

            #lookup treedistance table for previous result
            else:
                m = a.lmds[np]-a.lmds[x]
                n = b.lmds[nq]-b.lmds[y]
                sub_cost = Path(tp[m][n], sub_op(np, nq, a, b,
                    getlab), treepaths[np][nq].cost)
                if sub_op(np, nq, a, b, getlab): sub_cost.add_edit
                    (sub_op(m, n, a, b, getlab))
                else: sub_cost.seq.extend(treepaths[np][nq].seq)
                tp[i][j] = min(ins_cost, del_cost, sub_cost, key=
                    attrgetter('cost'))

update_mode = affected != None and prev_results != None

#Table for new results
treepaths = numpy.empty((len(a.nodes), len(b.nodes)), dtype=object
    )

if update_mode:
    taken = 0.0
    skipped = 0.0

    #Initialise new table
    for i in xrange(0, min(treepaths.shape[0], prev_results.shape
        [0])):
        for j in xrange(0, min(treepaths.shape[1], prev_results.
            shape[1])):
            treepaths[i][j] = prev_results[i][j]

    ch = ChangedData(affected)

#for each keyroot pair between a and b, update treepaths
for i in a.keyroots:
    for j in b.keyroots:

        if update_mode:
            if nodes_affected(i, j):
                subtree_distance(i, j)
                taken+=(i-a.lmds[i] + 1)*(j-b.lmds[j] + 1)
            else:
                skipped+=(i-a.lmds[i] + 1)*(j-b.lmds[j] + 1)
        else:
            subtree_distance(i, j)
if update_mode:
    print taken
    print skipped
    treepaths[-1][-1].saved = 100 - ((taken/(taken+skipped))*100)
    print treepaths[-1][-1].saved

treepaths[-1][-1].seq = list(OrderedDict.fromkeys(treepaths
    [-1][-1].seq))
return treepaths
```

B.1.3 Distance Inference

```
def tredit_infer(v1, v2, o, edits, diff, getlab):
    #TODO: Make this cleverer
    edit_labels = [(getlabel(v1, s, getlab), getlabel(o, d, getlab))
        for (s,d) in edits]
    diff_labels = [(getlabel(v1, s, getlab), getlabel(v2, d, getlab))
        for (s,d) in diff]
    syn_edits = Set(concatMap(subsubs, edit_labels))
    syn_diff = Set(concatMap(subsubs, diff_labels))

    #cost =  len(syn_edits ^ syn_diff)
    cost = len(Set(edit_labels) | Set(diff_labels)) - len(Set(
        edit_labels) & Set(diff_labels))
    cost = len(Set(edit_labels) | Set(diff_labels)) - len(Set(
        syn_edits) & Set(syn_diff))

    return cost
```

B.2 Extracting tree metadata

```
class TreeData:
    def __init__(self, root):
        self.nodes = list()
        self.keyroots = list()
        self.lmds = dict()
        self.build_lists(root)
        self.keyroots.append(len(self.nodes)-1)

    #Generate postorder lists of nodes and keyroots, determine lmd of
        each node
    def build_lists(self, node):
        #The postorder enumeration can be found by inspecting the
            length of the list of nodes.
        def id(): return len(self.nodes)

        children = list(iter_child_nodes(node))

        #Leaf node case
        if not children:
            self.lmds[id()] = id()
        else:
            self.build_lists(children[0])    #Descend left to find lmd
                , pass result back up to top
            lmd = self.lmds[id()-1]          #id-1 is currently the
                index of the left most child

            #Process the other child nodes
            for i in xrange(1, len(children)):
                self.build_lists(children[i])
                self.keyroots.append(id()-1) #id-1 index of current
                    child

            self.lmds[id()] = lmd
        self.nodes.append(node)
```

B.3 Changed data

```
class ChangedData:
    def __init__ (self, edited):
        self.a_data = list()
        self.b_data = list()

        self.first_insert = None
        self.first_delete = None

        for edit in edited:
            if edit[0] != None: self.a_data.append(edit[0])
            elif self.first_insert == None: self.first_insert = edit
                [1]

            if edit[1] != None: self.b_data.append(edit[1])
            elif self.first_delete == None: self.first_delete = edit
                [0]
```

B.4 Node class

```
class Node:
    def __init__(self, label, children=None):
        self._label = label
        self._children = children or list()

    def label(self):
        return self._label

    def label(self, l):
        self._label = l

    def children(self):
        return self._children
    def children(self, c):
        self._children = c

    def add_child(self, child):
        self.children.append(child)
        return self

    def __str__(self, level=0):
        result = "\t"*level + self._label + "\n"
        for child in self.children:
            result += child.__str__(level+1)
        return result

    def __eq__(self, other):
        return self.label == other.label
```