Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Informatik
Computer Science

Diff library in Eiffel
(Diffeif)

Bachelor Thesis

Rafael Wampfler
ETH Zurich
wrafael@student.ethz.ch

Supervised by: Yu Pei
Prof. Dr. Bertrand Meyer
Chair of Software Engineering

January 9, 2013

Abstract

The *Diffeif* library is a library which provides functionality for computing the difference of two code or text chunks and visualize it in a GUI. The library is implemented in and for Eiffel.

Finding the difference of two code or text pieces is a common problem. There exists a bunch of algorithms for computing such a difference. The most widely used algorithms are line-based, which can easily be adapted to word- or character-based. Another class of algorithms can be applied to trees created from the corresponding input, so called tree-based algorithms.

This documentation gives an overview of some existing line-based and tree-based algorithms, explains the implemented algorithms in detail accompanying with a performance analysis and provides a short overview of the implementation and the usage of the library.

Acknowledgements

I would like to thank my supervisor Yu Pei for the helpful suggestions and valuable comments and corrections received throughout the project. I would also like to extend my thanks to Prof. Bertrand Meyer for allowing me to work on this interesting topic. Further thanks go to my parents for supporting me during the whole time.

Contents

1 Introduction **1**
1.1 Focus of this Work 4
1.2 Related Work 5
1.3 Thesis Organization 5
1.4 Definitions 5

2 Flat-Based Diff Algorithms **9**
2.1 Simple Algorithm 13
2.2 Basic Dynamic Programming Algorithm 14
2.3 Miller & Myers' Algorithm 16
2.4 Hunt & McIlroy's Algorithm (Original GNU *Diff*) 22
2.5 Heckel's Algorithm 23

3 Tree-Based Diff Algorithms **27**
3.1 Tai's Algorithm 33
3.2 Zhang & Shasha's Algorithm 35
3.3 X-Diff Algorithm 37
3.4 LaDiff Algorithm 39
3.5 Miscellaneous Algorithms 40

4 Implementation **43**
4.1 Internal Data Representation 43
4.2 Diffeif Library 45
4.3 GUI 46

5 User Guide **49**

6 Performance Analysis **53**
6.1 Miller & Myers' Algorithm (flat-based) 54
6.2 Zhang & Shasha's Algorithm (tree-based) 56

6.3 Miller & Myers' Algorithm vs. Zhang & Shasha's Algorithm . . . 59

7 Conclusion 61

8 Future Work 63

A Line-Based vs. Word-Based vs. Character-Based Diff 65

B Listings 67

B.1 Miller & Myers Algorithm 67

B.2 Zhang & Shasha's Algorithm 69

C Internal Data Representation 71

List of Figures

1.1 Example code calculating the factorial of a number. 4

2.1 Prefix of the example code. 10
2.2 A singular deletion. 11
2.3 Dynamic Programming Example. 15
2.4 Example of Miller & Myers Algorithm. 19
2.5 Example of Heckel's Algorithm. 24

3.1 An example of a tree. 28
3.2 Preorder and postorder traversal. 29
3.3 An example of an edit sequence on trees. 31
3.4 A sample mapping. 32
3.5 An illegal mapping. 34
3.6 An example of a forest. 35

4.1 Class diagram for the *Diffeif* library. 45
4.2 Class diagram for the GUI. 46

5.1 A screenshot of the GUI. 50

6.1 Performance of the flat-based algorithm. 56
6.2 Performance of the tree-based algorithm. 58
6.3 Performance of tree-based diff vs. flat-based diff. 60

A.1 Line-based diff. 65
A.2 Word-based diff. 66
A.3 Character-based diff. 66

C.1 A DOM-tree generated from the XML file in Listing C.1. 73

List of Tables

5.1 Features of a `DIFFEIF` object. 51

Listings

4.1 Example code in Eiffel. 44
B.1 Code for Miller & Myers Algorithm. 67
B.2 Code for Zhang & Shasha's Algorithm. 69
C.1 XML file representing the AST of the code in Listing 4.1. 71

Chapter 1

Introduction

Computing and visualizing differences has a wide range of applicability.

- *Evolution of code.* Explore how versions of text files or source programs differ. Visualizing the difference will give an overview of the evolution of the text or code.

- *File merging.* A file can be merged with the difference to other files. There is no need to replace the whole file.

- *Backup programs and file storage.* For a new version of a backup the difference to the previous version can be stored instead of a whole new copy of the corresponding files. This will safe space. The same method can be used for file storage. If a file is evolved from another, it is a good idea to store only the difference to the parent file instead of storing a new file. Especially in the case of networking it will safe a lot of bandwidth.[1] This method is generally known as *delta encoding*.[2]

- *HTTP Servers.* They usually send the difference to clients if a new version of a web page is available. This will safe bandwidth.

- *Biology.* The difference of molecules or amino acid sequences is computed.

- *Video processing.* Difference algorithms play an important role. Only the difference between two video frames is transferred. The new frame can be

[1]The capacity of disks have evolved much faster than the bandwidth of (non-LAN) networking did. Therefore, the bottleneck shifted from disks towards the bandwidth of networking.

[2]See `http://en.wikipedia.org/wiki/Delta_encoding` for further details. Retrieved 2012-12-28.

created by merging the old frame with the difference. In video processing the difference is often called a *movement vector*.

There are a lot of file comparison tool available for every conceivable operating system. The most famous one is probably the GNU *diff* utility.[3] Two major classes of difference [4] algorithms can be distinguished:

- *Flat-based diff algorithms.* Algorithms working on a string or array of n symbols. The symbols can be in the form of lines, words, characters or bytes[5]. Throughout the document this class of algorithms will be referred to as either flat-based diff algorithms or as line-based algorithms, unless otherwise stated.

- *Tree-based diff algorithms.* Algorithms working on a tree, so called tree-based algorithms. For code differencing, it is most often an abstract syntax tree describing the syntactic structure of the code. Before applying this class of algorithms, an appropriate input, like a file, has to be parsed into a tree. This is described in Chapter 3.

Most diff algorithms compute an *edit script* which will turn the source file and the source tree into the destination file and destination tree, respectively. Because an edit script alters one file or tree into the other, it is common to name them the source and destination. An edit script consists of a sequence of *edit operations*. Each edit operation is one operation performed on the source file or source tree and has an assigned *cost*[6]. There exists a wide range of edit operations. The most commons are add, delete, change and move. From the add and delete operations almost all other operations can be built. Therefore, standard algorithms only compute the operations delete and add. In this thesis an additional copy operation[7] is introduced.

If each edit operation is assigned a cost of one then the length of an edit script, or in other words the number of edit operations, is the so called *edit distance*. If the edit operations are weighted in a different way, then the edit distance is the sum of the costs of the operations in the edit script. The edit distance is a measure for

[3]Other programs are e.g. *WinMerge* and *Microsoft XML Diff and Patch Tool*.

[4]Referred to as diff throughout the document. It is explicitly stated and written in italic if *diff* designates the GNU utility.

[5]Byte-diff is used for comparing binary files. This is not covered in this thesis.

[6]The term *cost* refers to an integer greater or equal zero. It reflects the weighting of the edit operation compared to other edit operations, i.e. an edit operation with a smaller cost is more important than one with a higher cost and thus gets more likely used.

[7]Also called twin or clone operation

the distance between two files or trees. If the edit distance is long, i.e. there exists a lot of edit operations in the edit script in the unit cost case, the files or trees are rather different.

Instead of creating an edit script, it is also common to compute the *longest common subsequence* (LCS), which is defined to be the longest sequence contained in two files or the biggest subtree contained in two trees[8]. In a more formal way the longest common subsequence of two strings is defined as follows: let $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$ be two strings. $C = c_1c_2...c_i$ is a common subsequence of A and B if deleting $n - i$ characters from A and $m - i$ characters from B yields C. C is the longest common subsequence if i is maximal.

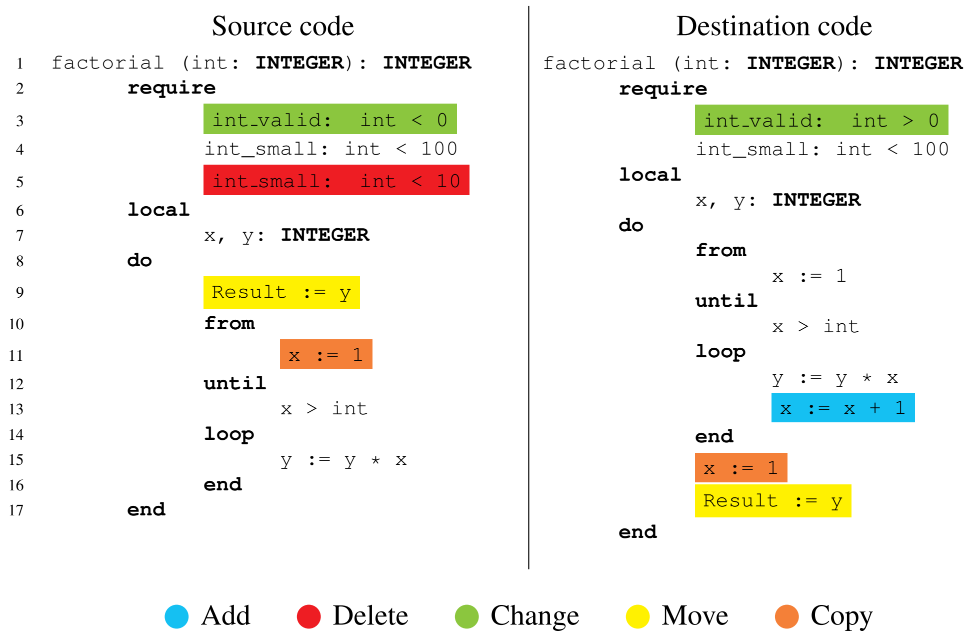
As stated in [14], finding the longest common subsequence and computing the edit script are dual problems. If we know the LCS, it is easy to get the edit script: each item in the source file and destination file, which is not present in the LCS, must be deleted or added from the source file and destination file, respectively. Vice versa, the LCS can be computed from an edit script by including all items from the source and destination file which are not deleted or added. Because all other edit operations can be computed from the delete and add operation as stated above, the LCS and an edit script are similar. Therefore, only edit scripts are examined throughout this document, unless otherwise stated.

In Figure 1.1 an example is provided for introducing the edit operations and showing how a difference can look like.[9] Line-based diff is used in the example. The different colors indicate the edit operations. Line 3 contains a change, which is represented by green color. Line 5 is deleted from the source code. The used color for this operation is red. The statement in line 9 of the source code is moved to line 17 of the destination code. Yellow is used for this operation. In line 14 a statement is added to the destination code, this corresponds to blue color. Finally, the statement *i := 1* in line 11 of the source code is copied and inserted at line 16 in the destination code. This is shown in orange color. The edit distance in the example is five because there exists five edit operations. For a word- or character-based diff, the result would look slightly different. This is demonstrated in Chapter 5.

The main issue of the diff algorithms is to find the optimal edit script, which is the shortest edit script if all operations are assigned a cost of one or otherwise, the edit script of minimum cost, i.e. minimizing the edit distance. It is possible that there exists several optimal edit scripts. In such a situation just one script is taken

[8]This is also called the *largest common subtree*

[9]The code snippets could be shortened and simplified, but for demonstration purposes it is chosen as it is.



Source code

```
factorial (int: INTEGER): INTEGER
        require
                int_valid:  int < 0
                int_small: int < 100
                int_small:  int < 10
        local
                x, y: INTEGER
        do
                Result := y
                from
                        x := 1
                until
                        x > int
                loop
                        y := y * x
                end
        end
```

Destination code

```
factorial (int: INTEGER): INTEGER
        require
                int_valid:  int > 0
                int_small: int < 100
        local
                x, y: INTEGER
        do
                from
                        x := 1
                until
                        x > int
                loop
                        y := y * x
                        x := x + 1
                end
                x := 1
                Result := y
        end
```

🔵 Add 🔴 Delete 🟢 Change 🟡 Move 🟠 Copy

Figure 1.1: Example code calculating the factorial of a number.

or the algorithm uses semantic analysis to choose the proper edit script.[10]

1.1 Focus of this Work

This report provides an overview of existing tree-based and flat-based diff algorithms, whereas one algorithm of each category is implemented. The implementation is given as a library for ease of reuse. Moreover, a GUI is provided for visualizing the differences graphically. Primarily, the library can be used for code differencing, although the flat-based diff algorithm is applicable to every other text document. As stated in [9], there are four major fields of application for code differencing.

- *Editing aid.* Verification of modifications and detection of unintended edits.
- *Debugging aid.* Finding the difference between a working and a non-working piece of code.
- *Program maintenance aid.* Searching and joining changes in the same code introduced by several different people.

[10]Semantic analysis is beyond the scope of this thesis and therefore not covered.

- *Quality control.* Examining the difference between two versions of a program.

The focus of the implemented algorithms lies on correctness, i.e. detecting all differences and returning an edit script of minimal length, and not on performance, even though the algorithms were chosen in a way so that the runtime should not be too bad. At the end of this report, an overview of the performance of the implemented algorithms is given.

1.2 Related Work

A good overview over flat-based diff algorithms is provided by Neil Fraser[6] and Ward Cunningham [5]. Besides an introduction to diff algorithms, Neil Fraser also introduced some pre- and postprocessing steps, which can be applied before and after the diff algorithm, respectively. Preprocessing can consist, for example, of detecting common prefixes and suffixes and singular insertions and deletions. Postprocessing covers syntactic and semantic changes to the edit script. This will be explained in more detail in Chapter 2.

A survey of tree-based diff algorithms was written by Philip Bille[1] and by Daniel Hottinger and Franziska Meyer in their semester thesis about XML-diff algorithms[10].

1.3 Thesis Organization

Chapter 2 presents different flat-based diff algorithms, applicable to words, character and lines. Moreover, postprocessing and preprocessing steps are introduced. Chapter 3 deals with tree-based diff algorithms. In Chapter 4 the implementation details of the library and the GUI are explained. A user guide for the library and the GUI is presented in Chapter 5. A performance analysis of the implemented flat- and tree-based diff algorithms can be found in chapter 6. Finally, chapter 7 and 8 contains the conclusion and future work, respectively.

1.4 Definitions

In this section some crucial definitions are given, which are used in one ore more chapters. The terms are sorted alphabetically.

- *Abstract Syntax Tree (AST):* a tree representing the syntactic structure of a piece of code.

- *Damerau-Levenshtein Distance:* distance metric obtained by allowing add, delete and change edit operations and the transposition of two neighbouring characters.

- *Document Object Model (DOM):* a specification of an API for the access of XML and HTML documents defined by the World Wide Web Consortium (W3C)[11]. It is up to the developers to implement the API. An implementation of the API allows to change the content, structure and layout of the corresponding document.

- *Dynamic Programming:* a tabular computation method for solving of optimisation problems. The problem is divided into subproblems, which can be solved in an optimal way.

- *Edit Distance:* a measure of the distance between two strings or trees. In the case of strings it is often used to refer specifically to the Levenshtein distance. The edit distance is equal to the sum of the cost of each edit operation in the edit script or, if unit costs are used, to the length of the edit script.

- *Edit Operation:* an operation performed on files or trees with an assigned cost. The term *cost* refers to an integer greater or equal zero. It reflects the weighting of the edit operation compared to other edit operations, i.e. an edit operation with a smaller cost is more important than one with a higher cost and thus gets more likely used. Most common edit operations are add, delete, change and move.

- *Edit Script:* a sequence of edit operations which turns a source file and a source tree into a destination file and destination tree, respectively.

- *Extensible Markup Language (XML):* a markup language for the representation of hierarchical structured data in the form of text files. It is defined, among others, in the XML 1.0 Specification by W3C.[12]

- *Hamming Distance:* distance metric obtained by allowing only change edit operations. Therefore, only strings of the same length can be compared.

- *Levenshtein Distance:* distance metric obtained by allowing add, delete and change edit operations.

[11] `http://www.w3.org/`. Retrieved 2012-12-08.

[12] `http://www.w3.org/TR/REC-xml/`. Retrieved 2012-12-08.

- *Longest Common Subsequence (LCS):* distance metric obtained by allowing add and delete edit operations. A *LCS* is the longest sequence contained in two strings.
- *NP-hard Problem:* for NP-hard (non-deterministic polynomial-time hard) problems it is only possible to verify that a solution of the problem is correct, but it is impossible to compute a solution in polynomial time.
- *Ordered Tree:* the order of the children of a node is important. The opposite of an *unordered* tree.
- *Preorder Traversal:* depth-first traversal method, the root of a subtree is visited first, followed by visiting the left subtree and the right subtree recursively.
- *Postorder Traversal:* depth-first traversal method, the root node of a subtree is visited last, i.e. first, the left subtree followed by the right subtree is traversed recursively, and at last the root of the subtree is visited.
- *Rooted Tree:* a tree with a distinguished node, the root node. The opposite of an *unrooted* tree.
- *Tree:* a set of hierarchical structured nodes. There exists three types of nodes: a *root node* is the root of the tree, it has zero or more children but no parent. An *inner node* has exactly one parent and at least one children. A *leaf node* has one parent but no children.

Chapter 2

Flat-Based Diff Algorithms

This chapter covers some flat-based diff algorithms. First, preprocessing and post-processing steps are introduced, which can be applied before and after running the diff algorithm, respectively, followed by a general introduction and some definitions of diff algorithms. Finally, some diff algorithms are explained in detail. In Section 2.1 a straightforward algorithm is presented, followed by the basic *dynamic programming* algorithm in Section 2.2. Section 2.3 covers the Miller & Myers' algorithm which is implemented in the *Diffeif* library. The Hunt & McIlroy's Algorithm is introduced in Section 2.4. The chapter is concluded by Heckel's algorithm in Section 2.5. For simplicity, we will consider throughout this chapter an example of calculating the difference of two strings on character basis. A file can easily be parsed in such a string and for word- and line-based diff, the pre- and postprocessing steps as well as the underlying diff algorithm stay the same.

Before running a diff algorithm, it is a good idea to consider some preprocessing steps as introduced by Neil Fraser[6]. This will help detecting special cases for which the diff algorithm has not to run at all, and therefore it will safe runtime. Furthermore, preprocessing simplifies the two strings lowering the runtime of the diff algorithm.

A first step in preprocessing is to search for common prefixes and suffixes of the two strings. A prefix and a suffix is a common substring at the beginning and at the end of both strings, respectively. Because most of the time the two sequences compared are similar to a certain extent, it is likely to find some common prefixes and/or suffixes. In our running example on page 4, the first and second line of the source and destination code form the prefix (see Figure 2.1).

The computation of the prefix and suffix can be done by scanning through both strings in $O(n)$ time or by a binary search taking $O(log(n))$ time[1] where

[1]Assuming that the equality operation for two strings takes $O(1)$ time

Source code

```
factorial (int:  INTEGER): INTEGER
      require
            int_valid: int < 0
            int_small: int < 100
            int_small: int < 10
      local
            x, y: INTEGER
      do
            Result := y
            from
                  x := 1
            until
                  x > int
            loop
                  y := y * x
            end
      end
```

Destination code

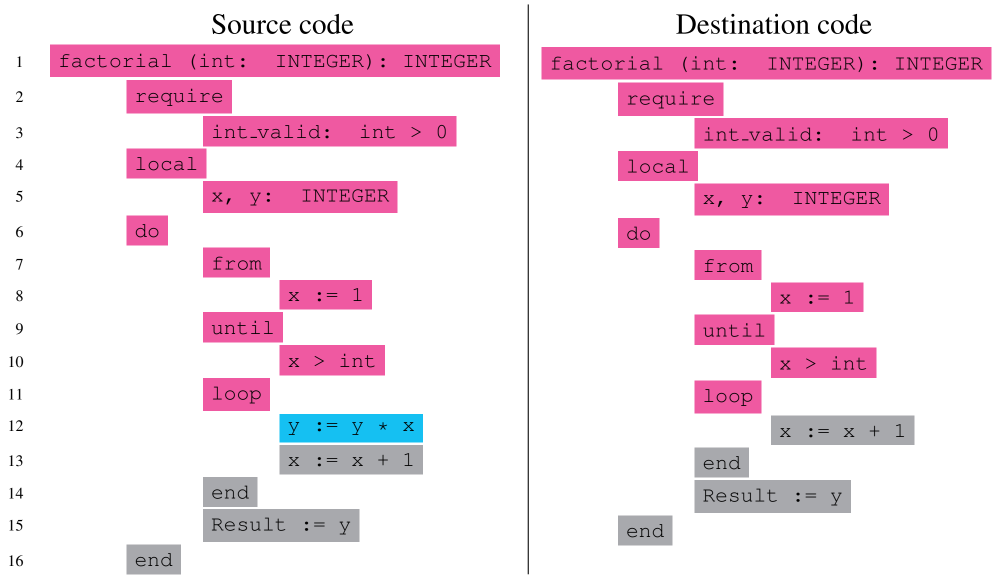
```
factorial (int:  INTEGER): INTEGER
      require
            int_valid: int > 0
            int_small: int < 100
      local
            x, y: INTEGER
      do
            from
                  x := 1
            until
                  x > int
            loop
                  y := y * x
                  x := x + 1
            end
            x := x + 1
            Result := y
      end
```

Figure 2.1: Prefix of the example code.

$n = min(length(source\_string), length(destination\_string))$. The prefix and suffix can safely be removed. The diff algorithm is applied to the rest of the string. Once the prefix and suffix are calculated, it is also easy to detect the equality of both strings: the two sequences are equal if the prefix and suffix span the whole string.

The second step in preprocessing is to search for singular insertions and deletions as described in [6]. They can be detected by deleting the prefix and suffix from both strings. If only one element remains in one string then it is a singular insertion or deletion. If such a common case is detected, the diff algorithm does not need to run at all. Figure 2.2 shows an example of a singular deletion. The prefix is marked in magenta and the suffix in grey. If the prefix and suffix are deleted, line 12 in the source code remains, and hence it is an insert (marked in blue). Similar if a line would remain in the destination code, it would be a singular deletion. Of course, the preprocessing could be extended to detect longer insertions and deletions, but this will cost more runtime and, at some point, running the diff algorithm without the preprocessing step is cheaper.

After the preprocessing is done, the diff algorithm can run. The flat-based diff algorithms are almost always working on an array. The elements of the array are lines, words or characters, depending on how the string is split. The algorithms produce a minimal edit script, i.e. an edit script with minimal edit distance. The



```
factorial (int: INTEGER): INTEGER
        require
                int_valid: int > 0
        local
                x, y: INTEGER
        do
                from
                        x := 1
                until
                        x > int
                loop
                        y := y * x
                        x := x + 1
                end
                Result := y
        end
```

```
factorial (int: INTEGER): INTEGER
        require
                int_valid: int > 0
        local
                x, y: INTEGER
        do
                from
                        x := 1
                until
                        x > int
                loop
                        x := x + 1
                end
                Result := y
        end
```

Figure 2.2: A singular deletion.

edit distance can be defined in many ways. Often it refers to the longest common subsequence which allows add and delete edit operations. Another famous distance metric is the Levenshtein distance using add, delete and change operations. Other distance metrics are the Hamming distance, allowing only the change operation[2], and the Damerau-Levenshtein distance, granting add, delete, change operations and the transposition of two neighbouring characters. Usually a unit cost of 1 is assigned to each edit operation, and therefore the minimal edit distance is reduced to the minimal number of edit operations.

The edit script E for the example in Figure 1.1 on page 4 looks as follows: $E = [3, 3c3, 3; 5, 5d4, 4; 9, 9m17, 17; 15, 15a14, 14; 11, 11t16, 16]$. The edit script produced by the *Diffeif* library has the same style, it is inspired by the output of the GNU *diff* tool. An edit operation is defined as $O = s1, s2Xd1, d2$ with $X \in \{a, c, d, m, t\}$, *s1 = source start index*[3], *s2 = source end index*, *d1 = destination start index* and *d2 = destination end index*. Now, the different types of edit operations are explained in more detail.

1. $X = a \rightarrow s1 = s2$. The index elements *d1* to *d2* of the destination string were *added* after index element *s2* of the source string.

[2]As a consequence, only strings of the same length can be compared.
[3]The index can refer to lines, words or characters.

2. $X = c$. The index elements *s1* to *s2* of the source string were *changed* to the index elements *d1* to *d2* of the destination string. Note that it is possible for a change operation to cover unequal block sizes, i.e. a block of $s2 - s1$ elements in the source can be changed to $d2 - d1$ elements in the destination, where $s2 - s1 \neq d2 - d1$.

3. $X = d \rightarrow d1 = d2$. The index elements *s1* to *s2* were *deleted* from the source string. The index element *d1* (or *d2*) represents the position in the destination string where the index elements *s1* to *s2* would be added if the delete operation would be changed into an add operation.[4]

4. $X = m$.

   (a) $s1 = s2 \rightarrow d1 = d2$. The index element *s1* (or *s2*) of the source string was *moved* to the index element *d1* (or *d2*) of the destination string.

   (b) $s1 \neq s2 \rightarrow s2 - s1 = d2 - d1$. The index elements *s1* to *s2* of the source string were *moved* to the index elements *d1* to *d2* of the destination string.

5. $X = t$.

   (a) $s1 = s2 \rightarrow d1 = d2$. The index element *d1* (or *d2*) of the destination string was *copied (cloned)* from the index element *s1* (or *s2*) of the source string.

   (b) $s1 \neq s2 \rightarrow s2 - s1 = d2 - d1$. The index elements *d1* to *d2* of the destination string were *copied (cloned)* from the index elements *s1* to *s2* of the source string.

If the flat-based diff algorithm is finished, the postprocessing steps adjust the edit script. The flat-based diff algorithm implemented in the *Diffeif* library produces an edit script containing add and delete operations. Other edit operations can be built from these two basic operations in the following way: a change operation can be derived from a delete operation followed by an add operation affecting the same elements in the source and destination. If the same element is removed from one place in the source and added to another place in the destination, it is a move operation. If an element is added to the destination and another equal element exists already in the source, a copy (or clone/twin) operation is detected. All this can be done with scanning over the edit script. This does not introduce big

[4]Most of the time *d1* and *d2* are useless information for a delete operation, nevertheless they are given for consistency.

additional runtime costs because edit scripts are usually not that large. Neil Fraser describes some more syntactic and semantic postprocessing steps.[6]

Before we will start with the flat-diff algorithms, the difference between line-based, word-based and character-based diff should be shortly discussed. In appendix A some examples are shown.[5] In these examples the differences between line-based, word-based and character-based diff are not big (only line 3 differs slightly) but usually there exists remarkable variances. Character-based diff produces the finest-grained result but takes the longest to execute because each character must be compared. Line-based and word-based diff are faster and produce less individual edits but the length of the particular edits is larger. Which level to use depends on the application. Source code is usually compared with line-based diff, whereas for a text word-based diff is applied. Character-based diff is often used for binary data. Besides that, some algorithms are more efficient when applied to bigger chunks of data, such as lines, others are faster at handling smaller chunks of data, such as characters. That is because different lines appears infinitely often, whereas characters are limited to a number of tokens, which appears more or less frequently.[6]

2.1 Simple Algorithm

A straightforward technique for computing the diff is shortly covered in [9, 11, 13]. A first approach is to scan through both strings or files comparing each corresponding element of the source and destination. An optimisation could be to require that several consecutive elements have to match. Nevertheless, the produced edit script is often much longer than the optimal one. A second approach is to compare not only corresponding elements but also to examine shifted elements. If a mismatch is encountered, the kth element of each file or string is compared against the k elements after the mismatch in the other file or string for $k = 1, 2..., n$, where n is the position of a matching element.

Both methods are unscalable[6] with the length of the strings or files and have a time complexity of O(n*m), where n is the length of the source string or file and m is the length of the destination string or file.

[5]Note that the *Diffeif* library would produce slightly different but still correct results for word-diff and char-diff of the examples shown in the figures. The examples are only shown in this way for easier understanding.

[6]Scalability is the ability of the algorithm to handle a growing amount of data (lines, words or characters).

2.2 Basic Dynamic Programming Algorithm

A good introduction to *dynamic programming* can be found in [7]. Most algorithms in this and the next chapter are based on dynamic programming. In this section, first, a formal definition of dynamic programming is given, followed by an example showing a sample computation of a difference. Dynamic Programming is a tabular computation method for solving of optimisation problems. The problem in question is divided into subproblems, which can be solved in an optimal way. If this is applied recursively reaching smaller and smaller subproblems, elementary subproblems, which can not be divided further, are obtained at the end. From optimal solutions of these subproblems, an optimal solution of the problem can be constructed in a bottom-up fashion.

As a distance metric the *Levenshtein distance* is used, allowing for deletions, insertions and changes. The *add* and *delete* operations introduce a cost of 1, whereas the *change* operation is assigned a cost of 2.[7] With the cost of 2 for a change operation it is more likely to get add and delete operations. This reflects the diff problem in a good way. Let us assume that we have two strings $A = a_1, a_2, ..., a_m$ and $B = b_1, b_2, ..., b_n$, where a_i and b_j for $i = 1, ..., m$ and $j = 1, ..., n$ represent characters, words or lines. For an optimal solution there are only three possible cases for the last elements: (1) a_m and b_n match or mismatch $\rightarrow$ for $a_1, ..., a_{m-1}$ and $b_1, ..., b_{n-1}$ an optimal solution can be computed. (2) a_m is deleted $\rightarrow$ for $a_1, ..., a_{m-1}$ and $b_1, ..., b_n$ an optimal solution can be computed. (3) b_n is deleted[8] $\rightarrow$ for $a_1, ..., a_m$ and $b_1, ..., b_{n-1}$ a optimal solution can be computed. Therefore, we arrive at the following recursion for computing the optimal edit distance between string *A* and *B*:

$$D_{m,n} = min(D_{m-1,n-1} + x_{m,n}, D_{m,n-1} + 1, D_{m-1,n} + 1)$$

$$x_{m,n} = \begin{cases} 2 & \text{if } a_m \neq b_n \\ 0 & \text{if } a_m = b_n. \end{cases}$$

The base conditions are $D_{0,0} = 0$, $D_{i,0} = i$ and $D_{0,j} = j$ for $i = 1, ..., m$ and $j = 1, ..., n$.

The recursion can be computed with the help of a table *D*. The table contains *m + 1* rows with a_i, $i = 1, ..., m$ as the row labels and *n + 1* columns with b_j, $j = 1, ..., n$ as the column labels. One extra row and column is needed for the base conditions $D_{i,0}$ and $D_{0,j}$ for $i = 1, ..., m$ and $j = 1, ..., n$. In our case, the

[7]The chosen cost of 2 for a change operation is due to the fact that the algorithm in the next section is based on this assumption. Of course, it is also possible to assign other costs for the operations.

[8]This corresponds to an insert of b_n after the last position of string A.

table can be filled row by row or column by column. There are two phases when applying a dynamic programming algorithm. (1) In the forward computation phase each entry of the table is calculated, in our case corresponding to the edit distance. The entries can be computed from the entries in the west (left), north (above) and northwest (diagonal) as shown in the example. The optimal solution is located at the entry *D(m,n)*. (2) In the backward computation phase a optimal solution path is found, i.e. an optimal edit script. Of course, it is possible to calculate the corresponding edit scripts already in the forward phase and thus making the backward step unnecessary, which will safe computation time.

The time complexity of the described algorithm is $O((m+1)*(n+1))$ because every element of the table has to be computed. The space requirement is *O(min(m + 1, n + 1))*. According to [7], the algorithm can be improved by a divide and conquer approach, not covered here, which reduces the cost of the computation.

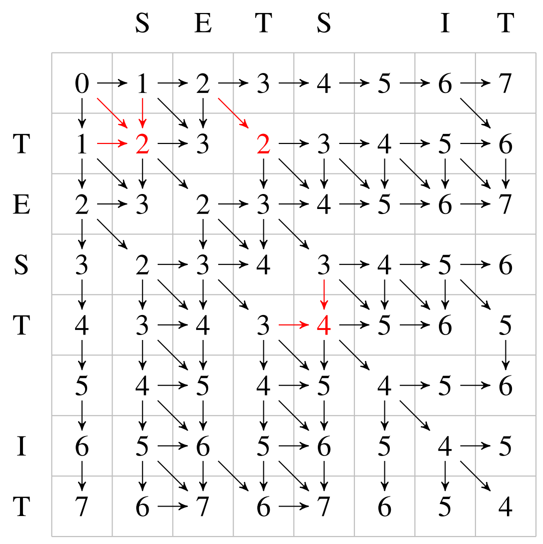


Figure 2.3: Dynamic Programming Example.

Let us look at an example which is shown in Figure 2.3. There are two strings *A* = *TEST IT* and *B* = *SETS IT*. A character-based diff is searched. The table *D* is filled row by row. Each field contains the edit distance for the corresponding substrings and the arrows indicates from where the values come from. Note that the table starts at position *D(0,0)*. Filling the first row and column is simple: for the difference between an empty substring of *A* and string *B*, every element of string *B* has to be removed introducing a cost of 1 for each delete operation. The same holds for the difference between an empty substring of *B* and string *A*. In the following, three entries of the table *D* are explained in more detail.

(1) For computing *D(1,1)* we can come either from *D(0,0)*, *D(1,0)* or *D(0,1)*. *D(0,0)* holds a distance of 0 and because $a_1 = T \neq S = b_1$, a cost of 2 for a change would be introduced leading to a distance of 2 in *D(1,1)*. *D(1,0)* contains a distance of 1 revealing *D(1,1) = 2* because $b_1 = S$ is added after $a_1 = T$. The same holds for *D(0,1)* except that $a_1 = T$ is deleted. Therefore, the minimum achievable distance is *D(1,1) = 2* coming from *D(0,0)*, *D(1,0)* or *D(0,1)*.

(2) For filling *D(1,3)* we can come from *D(0,2)*, *D(0,3)* or *D(1,2)*. Going from *D(0,2)* to *D(1,3)* would introduce a cost of 0 because $a_1 = b_3$ revealing *D(1,3) = 2*. This is already the minimum since *D(0,3) = 3* $\rightarrow$ *D(1,3) = 4* (deleting $a_1 = T$) and *D(1,2) = 3* $\rightarrow$ *D(1,3) = 4* (adding $b_3 = T$ after $a_1 = T$).

(3) *D(4,4) = 4* and this can be achieved by coming from *D(3,3)*, *D(4,3)* or *D(3,4)* because *D(3,3) = 4* $\rightarrow$ *D(4,4) = 6* (changing $a_4 = T$ to $b_4 = S$), *D(4,3) = 3* $\rightarrow$ *D(4,4) = 4* (adding $b_4 = S$ after $a_4 = T$) and *D(3,4) = 3* $\rightarrow$ *D(4,4) = 4* (deleting $a_4 = T$).

The minimum edit distance is *D(m,n) = D(7,7) = 4*. Due to the form of the example, there exists eight different minimal edit scripts, obtained by backtracking, with an edit distance of 4.[9] Three of these edit scripts[10] are $E_1 = [0a1, 0a2, 2d3, 4d4]$, $E_2 = [1c1, 2a3, 4d4]$ and $E_3 = [0a1, 1d1, 2a3, 4d4]$. Note that in this case, the length of the edit script is not equal to number of edit operations because not all edit operations have the same assigned cost.

2.3 Miller & Myers' Algorithm

This section is based on the paper from Web Miller and Eugene W. Myers.[13] The algorithm presented here is implemented in the *Diffeif* library, hence it is explained in detail including code and an example. The basis is the *dynamic programming* algorithm from the previous section but with the big advantage that there is no need to compute every entry in the table and therefore it is faster. The distance metric used is the *Longest Common Subsequence* allowing *add* and *delete* edit operations.

A table *D* with *m + 1* rows and *n + 1* columns with a_i, $i = 1, ..., m$ as the row labels and b_j, $j = 1, ..., n$ as the column labels is used. Every entry *D(i,j)* in the table holds the edit distance which corresponds to the number of edit operations

[9]Usually far less edit scripts of the same length exists.

[10]Instead of source/destination start and end index only one index is shown.

needed to transform *A[1:i]* to *B[1:j].* Accompanying with the edit distance the corresponding edit script is stored for every entry in the table. The big difference to the basic dynamic programming algorithm from the previous section is that, instead of filling the table row by row or column by column, the algorithm determines all entries of the table *D* in ascending order: first, all 0s are filled in the table, followed by all 1s, 2s and so on until the south-east value *D[m,n]* is reached. As we can see in Figure 2.3, the *d* entries of the table lies only on the diagonals *-d, -d+2,...,d-2,d*[11]. For example, for *d = 3* we only have to consider the diagonals -3, -1, 1 and 3. For efficiency reason, it is enough to compute and store only the position of the last element on each diagonal. This information is held in the array *last_d[k].* For each diagonal *k*, *last_d[k]* holds the row number of the last entry.

Now let us look how an entry on diagonal *k* is calculated. For computing the *d* entry on diagonal *k*, we only have to look at the diagonals *k+1* and *k-1*, which contain the *(d-1)* entry at their last position. There are only two possibilities for getting to diagonal *k*: first, we can move down from the last entry on diagonal *k+1* which will lead to a value *last_d[k] = last_d[k+1] + 1* because we moved one row downwards from diagonal *k+1*, and to an according edit operation of deleting the *last_d[k+1] + 1* symbol in string *A*. Second, we can move right from the last entry on diagonal *k-1* which will give us last_d[k] = last_d[k-1] because we stay on the same row, and an edit operation of appending the *last_d[k-1] + k* symbol of *B* after the last_d[k-1] symbol of *A*. Since we just want the farthest point on diagonal *k* we can check if $last\_d[k+1] \geq last\_d[k-1]$. If this is true, we can just move down from diagonal *k+1*, otherwise we would move right from diagonal *k-1*. In addition, there are two special cases: if *k = d*, we can only move right from diagonal *k-1* because the diagonal *k+1* is not yet filled with entries. If *k = -d*, it is only possible to move down due to the fact that diagonal *k-1* is empty.

Finally, when a position on diagonal *k* is found, we can move down the diagonal as far as possible until we encounter two different string elements, i.e. until $a_{i+1} \neq b_{j+1}$ for $i = row, ..., m$ and $j = row + k, ..., n$, where *row* indicates the position found above ($row = max(last\_d[k+1]+1, last\_d[k-1])$). *Row+k* is the column where the row intersects diagonal *k*. We continue with every relevant diagonal *k* = *-d, -d+2,...,d-2,d* in the same way which will give us last_d[k] values for the *d* entry of the table. We proceed for every *d* entry until we arrive at the south-east corner D(m,n) of the table. Accompanying with every *last_d[k]* value an edit script of length *d* is stored in *script[k].*

In Appendix B.1 Eiffel code for the Miller & Myers Algorithm can be found.

[11]The zeroth diagonal is the main diagonal, all diagonals above the main diagonal are numbered in ascending order (1,2,...), all diagonals below the main diagonal are numbered in descending order (-1,-2,...)

In the following, the code is explained in detail. *Lower* and *upper* are a pair of bounds indicating the range of diagonals to examine for each *d* entry of the table. Lines 5–22 initialize the data structures including the *last_d* array, the edit script and the array of edit scripts.[12] In line 18 and 19, *row* and *col* are initialized to the number of prefix items of both strings, which were computed in a preprocessing phase. The prefix represents the 0 entries of the table (line 20), which lie on the main diagonal. In the lines 24–34 the special case of touching the bottom or right border of the table is handled. If the bottom of the table is reached, the source string is a proper prefix of the destination string and vice versa if the right border of the table is reached. If both strings are identical, *m* = *n* must hold and hence *lower* is set to 1 (line 25) and *upper* to -1. This implies that $lower > upper$ and two identical strings are detected (line 36).

There are two nested loops, an outer and an inner one. The inner loop iterates over all edit distances (*d* values) until the south-east corner *D(m,n)* of the table is hit (the result is found), or the edit distance is greater than *max_length* ($d > max\_length$) implying that no difference is calculated. This is desirable if too large to be useful differences should be avoided. If *max_length* is set to $m + n$ the difference is calculated regardless of its size. After each iteration of the outer loop, the lower bound is decremented by one and the upper bound is incremented by one. This is because for each additional edit distance, one more upper and lower diagonal has to be considered due to *k = -d, -d+2,...,d-2,d*.

The inner loop iterates over all relevant diagonals. Line 55 catches the case that the inspected diagonal is the lowest one or $last\_d[k + 1] \geq last\_d[k - 1]$. In this instances, it is only possible to move down as described above. If we are at the topmost diagonal or $last\_d[k + 1] < last\_d[k - 1]$, only a right move is applicable (lines 64–68). Afterwards, we move down the diagonal until we hit the bottom or right border of the table or $a_{i+1} \neq b_{j+1}$ for $i = row, ..., m$ and $j = row + k, ..., n$. If we hit the south-east corner, a result is found. This is detected by the lines 84–88. Finally, the lines 90–93 and 95–98 handle two special cases: if we hit the bottom of the table, say on diagonal *k*, it is useless to look to the left of diagonal *k*. Therefore, the lower bound is incremented by one. In fact, on line 92 the lower bound is incremented by two but with the decrementation by one on line 101, the result is an incrementation by one. The same holds in the case that the right border is hit, except that it is pointless to look above the diagonal and the upper bound is therefore decremented by one.

An example of Miller & Myers algorithm is given in Figure 2.4. It calculates the minimum edit script for the same strings as in the example of the previous

[12]The class EDIT_OPERATION is not shown here but it is pretty obvious that it represents edit operations.

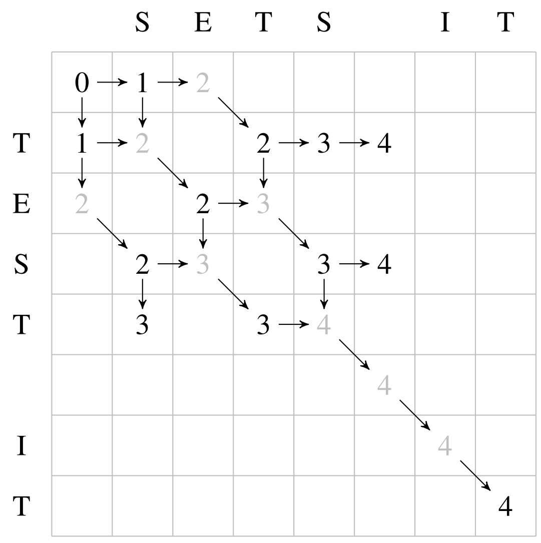


Figure 2.4: Example of Miller & Myers Algorithm.

section. The entries in light grey are not calculated and stored explicitly, they are only shown here for demonstration purposes. As we can see, compared to the standard dynamic programming algorithm (Figure 2.3) much less entries of the table have to be computed and stored, hence the algorithm shows a considerably better space and time performance, especially for a small number of differences. In the following, the calculation of every edit distance entry *d*, for *d = 1,...,4*, is shown.

(1) For computing the *d = 0* values, only diagonal 0 has to be considered. It is not possible to slide down the diagonal because $a_1 = T \neq S = b_1$ and therefore only at *D(0,0)* a 0 is filled in. This corresponds to *last_d[0] = 0* and *script[0]* = {}.

(2) For calculating the *d = 1* entries, the diagonals {*-d,d+2,...,d-2,d*} = {*-1,1*} have to be inspected.

   (a) *Diagonal 1:* it is only possible to move right from diagonal 0. Sliding further down diagonal 1 is not admissible because $a_1 = T \neq E = b_2$. Therefore, *last_d[1] = 0* (last entry on diagonal 1 is located on row 0) and *script[1]* = {*[0a1]*} (insert $b_1 = S$ before $a_1 = T$).

   (b) *Diagonal -1:* it is only possible to move down from diagonal 0. Since $a_2 = E \neq S = b_1$, sliding down diagonal -1 is not applicable, and hence *last_d[-1] = 1* and script[-1] = {[1d0]} (delete $a_1 = T$).

(3) For filling the *d = 2* values, the diagonals {*-d,-d+2,...,d-2,d*} = {*-2,0,2*} are the interesting ones.

   (a) *Diagonal 2: last_d[2] = 1* (*D(1,3) = 1*) and *script[2] = script[1] + 0a2* = {*[0a1,0a2]*} (insert $b_2 = E$ before $a_1 = T$) because only a move right from diagonal 1 is applicable, leading to the field *D(0,2)*, afterwards moving down the diagonal one step due to $a_1 = T = b_3$.

   (b) *Diagonal 0:* either moving down from diagonal 1 or moving right from diagonal -1, followed by moving down on diagonal 0, leading to last_d[0] = 2. This means that the last entry on diagonal 0 is now located on row 2 (The zero value from step (1) is no more needed). In addition, *script[0] = [script[1] + 1d0, script[-1] + 0a1]* = {*[0a1,1d0],[1d0,0a1]*}.[13]

   (c) *Diagonal -2: last_d[-2] = 2* and *script[-2] = script[-1] + 2d0* = {*[1d0,2d0]*} due to first moving down from diagonal -1 and then sliding down on diagonal -2.

(4) For computing the edit distances with value *d = 3*, the diagonals -3,-1,1,3 has to be examined.

   (a) *Diagonal 3:* Moving right from the last entry on diagonal 2, i.e. from *last_d[2]*. Therefore, *last_d[3] = last_d[2] = 1* (same row) and *script[3] = script[2] + 1a4* = {*[0a1,0a2,1a4]*}. Moving down on diagonal 3 is not possible.

   (b) *Diagonal 1:* Either moving down from the last entry on diagonal 2 (*last_d[2] = 1* $\rightarrow$ *last_d[1] = 2*) or moving right from the last entry on diagonal 0 (*last_d[0] = 2* $\rightarrow$ *last_d[1] = 2*), then sliding down on diagonal 1, yielding *last_d[1] = 3* and *script[1] = [script[2] + 2d3, script[0] + 2a3]* = {*[0a1,0a2,2a3],[0a1,1d0,2a3],[1d0,0a1,2a3]*}.

   (c) *Diagonal -1:* Moving down from diagonal 0 or moving right from diagonal -2 introducing *last_d[1] = 4* and *script[-1] = [script[0] + 3d2, script[-2] + 3a2]* =
   {*[0a1,1d0,3d2],[1d0,0a1,3d2],[1d0,2d0,3a2]*}.

   (d) *Diagonal -3:* Moving down from diagonal -2, sliding down diagonal -3 is not possible, leading to *last_d[-3] = 4* and *script[-3] = script[-2] + 4d1* = {*[1d0,2d0,4d1]*}.

[13]Note that {*[0a1,1d0],[1d0,0a1]*} contains two edit scripts: *0a1,1d0* and *1d0,0a1*.

(5) For the d = 4 entries of the table, the diagonals -4,-2,0,2,4 have to be considered.

   (a) *Diagonal 4:* Only moving right from diagonal 3 is applicable, sliding down diagonal 4 is not feasible. Therefore, *last_d[4] = 1* and *script[4] = script[3] + 1a5 =* $\{$*[0a1,0a2,1a4,1a5]*$\}$.

   (b) *Diagonal 2:* Moving right from diagonal 1, yielding *last_d[2] = 3* and *script[2] = script[1] + 3a5 =* $\{$*[0a1,0a2,2a3,3a5],[0a1,1d0,2a3,3a5],[1d0,0a1,2a3,3a5]*$\}$.

   (c) *Diagonal 0:* Either moving down from the last entry on diagonal 1 (*last_d[1] = 3* $\rightarrow$ *last_d[0] = 4*) or moving right from the last entry on diagonal -1 (*last_d[-1] = 4* $\rightarrow$ *last_d[0] = 4*), leading to position *D(4,4)*. Afterwards, sliding down diagonal 0 until *last_d[0] = 7* is possible because $a_5 = b_5$, $a_6 = I = b_6$ and $a_7 = T = b_7$. At *D(7,7)* the south-east corner is hit (*row = m = 7* and *col = n = 7*), and hence the algorithm terminates. Moreover, *script[0] = [script[1] + 4d4, script[-1] + 4a4] =* $\{$*[0a1,0a2,2a3,4d4],[0a1,1d0,2a3,4d4],[1d0,0a1,2a3,4d4], [0a1,1d0,3d2,4a4],[1d0,0a1,3d2,4a4],[1d0,2d0,3a2,4a4]*$\}$.

The minimum edit distance is *D(m,n) = D(7,7) = d = 4* and the corresponding edit scripts are hold in script[0]. There exists six different minimal edit scripts: [0a1,0a2,2a3,4d4], [0a1,1d0,2a3,4d4], [1d0,0a1,2a3,4d4],[0a1,1d0,3d2,4a4], [1d0,0a1,3d2,4a4] and [1d0,2d0,3a2,4a4]. Note that, in contrast to the standard dynamic programming algorithm, there are no change operations, hence the number of minimal edit scripts is less than in Section 2.2. As described earlier, the change operation can be easily derived from the add and delete operations.

A big advantage of the algorithm is that it always produces a shortest edit script, in contrast to other algorithms. According to [13], the worst case runtime is in $O((2d+1)min(m,n))$ and the expected runtime is $O(min(m,n)+d^2)$ where $d = D(m,n)$. This reveals that the algorithm is especially efficient when the differences between two files or strings are small compared to the their lengths *m* and *n*, i.e. $d \ll m$ and $d \ll n$, as a consequence the runtime complexity is reduced to $O(min(m,n))$. In fact, the algorithm is often four times faster than the GNU *diff*[14] tool, as stated in the paper [13]. A short exposition of the correctness of the algorithm can also be found in this paper.

A variation of the algorithm from Miller & Myers is given in [14] revealing a time and space complexity of $O(ND)$ and a expected runtime of $O(N+D^2)$ with

[14]The first version of the GNU *diff* tool was implemented according to Hunt & McIlroys algorithm[11].

$N = m + n$ (sum of the lengths of both strings) and $D = |edit\ script|$ (number of edit operations in the edit script). The algorithm is based on finding a path in the edit graph with the smallest number of non-diagonal edges and performs best for small number of differences. A further refinement presented in the paper shows a linear space complexity of only $O(N)$ at an expense of a poorer runtime performance.

2.4 Hunt & McIlroy's Algorithm (Original GNU *Diff*)

The Hunt & McIlroys algorithm was the basis for the first implementation of the GNU *diff* tool. As stated in the previous section, it is less efficient than Miller & Myers algorithm. Nevertheless, for the sake of completeness it is shortly discussed here. This section is based on the paper presenting the Hunt & McIlroys algorithm.[11] Furthermore, the standard *diff* output, also used in the implementation of the *Diffeif* library, was introduced in this paper.

The available edit operations are *add*, *change* and *delete*, hence the used distance metric is the *Levenshtein distance*. As in the algorithms from the previous sections, *dynamic programming* is used again. The dynamic programming table *D* is annotated with the elements of the first string *A* as the rows and with the elements of the second string *B* as the columns, as a result the table is again of size *m + n*, where *m* is the size of the first string and *n* is the size of the second string. The table entry *D(i,j)*, for *i = 1,...,m* and *j = 1,...,n*, stores the length of the longest common subsequence of *A(1..i)* and *B(1..j)* instead of the length or cost of the edit script as in the previous algorithms.[15] Rather than filling the whole table, only the so called *k-candidates* are computed and filled in the table, where *k* is the corresponding length of the longest common subsequence. An entry of the table is a k-candidate if and only if $A_i = B_j$ and $D(i,j) > max(D(i-1,j), D(i,j-1))$. First, all k-candidates are calculated. This is done by sorting the second string and then putting the equal elements into the same equivalence class[16]. Afterwards, every element of the first string is joined with an equivalence class. This produces the candidates[17] for each column. Then we generate the k-candidates by iterating through the columns left to right. A candidate (i_2, j_2) on a column is a k-candidate if $i_2 < i_1$ and $j_2 > j_1$ for (i_1, j_1) being a k-candidate. Now, the longest common subsequence can be determined. A common subsequence is a set of k-candidates

[15]Certainly it is also possible to adapt the algorithm to hold the length or costs of the edit scripts as the table entries.

[16]An equivalence class is a container holding equivalent elements regarding an equivalence relation.

[17]Note that this are not yet the k-candidates.

lying on a strictly increasing path, i.e. $i_2 > i_1$ and $j_2 > j_1$ for (i_1, j_1) and (i_2, j_2) being k-candidates k_1 and k_2 and $k_2 > k_1$. A longest common subsequence is a longest path from a set of paths, meaning a path containing the most k-candidates.

The algorithm can be tuned by hashing each string element. This especially helps for comparison of big files at the expense of stating unequal elements to be equal if they fall in the same bucket. The worst case time complexity is *O(mn log(m))* and the worst case space complexity is *O(mn)*, but in practice the algorithm performs better requiring only *O(m(m + n))* time and linear space.

2.5 Heckel's Algorithm

The flat-diff algorithms are completed by the presentation of Heckel's algorithm[9]. A prominent feature of this algorithm is the detection of *move* operations. Beside this, it also supports *delete* and *add* operations. First, an overview of the algorithm is given, followed by an example.

It is assumed to have two strings or files *A* and *B*[18] as in the previous sections. The elements (or symbols) of *A* and *B* could either be lines, words or characters. There are three data structures used by the algorithm: a *symbol table* (can be implemented as an array) and arrays *A* and *B* representing the strings. The symbol table contains the elements (or symbols) appearing in both strings, i.e. the characters, words or lines. Moreover, the symbol table stores for each entry the number of occurrences in both strings, called *OA* and *OB*, and a link to the corresponding entry in the array *A*. The symbol table can be built in a first step by scanning through both strings, putting each element into the symbol table if it is not yet present and incrementing a counter.

The second step consists of connecting the elements in array *A* and array *B* having *OA = OB = 1* (the elements occurring only once in both strings). These are unchanged, perhaps moved elements. The third step is to scan upwards from each mapped element until two not similar elements in both strings are found. This means if *A(i)* and *B(j)* are paired in the second step and *A(i+1)* and *B(j+1)* are equal, the corresponding elements are assumed to belong together (even if they occur more than once in both strings). In the fourth step the same thing is applied downwards the matching elements, i.e. compare *A(i-1)* with *B(i-1)*, *A(i-2)* with *B(i-2)*, and so on until a non matching pair is found. After this step, everything is done and the edit script can be created by scanning again through both strings. If a element in array *A* (the source array) is not matched with an element in array *B* (the destination array), a delete is found. Otherwise, if a unmatched element in array *B*

[18]In the following it is assumed to have two strings, but it is similar for two files.

is encountered, an insert is found. The matched elements are moved if they do not appear at the same position in both strings.

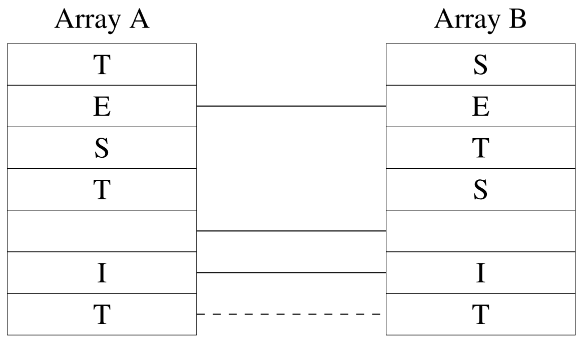


Figure 2.5: Example of Heckel's Algorithm.

In Figure 2.5 an example is shown, helping to better understand the algorithm. The strings are the same as in the examples of the previous algorithms, *A = TEST IT* and *B = SETS IT*. The algorithm is applied on character basis, and therefore array *A* and array *B* contains the characters of the strings. In the first step the symbol table is built, resulting in the entries *(T,3,2)*, *(E,1,1)*, *(S,1,2)*, *(Gap,1,1)* and *(I,1,1)*. The first value of each entry is the corresponding element and the second and third value is the number of occurrences of the element in string *A (= OA)* and string *B (= OB)*. In the second step the symbols with *OA = OC = 1* are connected, this are the elements *E*, *I* and *Gap* (solid lines). Afterwards, in the third and fourth step, we scan downwards and upwards from the matchings found in the previous step. Searching upwards yields no additional results but scanning downwards from element *A(6) = B(6) = I* reveals the element *A(7) = B(7) = T* because both next elements in string *A* and string *B* are equal (dashed line). Now it is time to build the edit script. The elements in string *A* not touched by a line introduce deletions, this applies to *A(1)*, *A(3)* and *A(4)*. Elements in string *B* not affected by a line yield insertions, this is true for *B(1)*, *B(3)* and *B(4)*. The paired elements (solid and dashed lines) can introduce moves. In this example this is not the case because the lines are solely horizontal. For non horizontal lines the corresponding elements would introduce moves.

The algorithm is easy to understand and has a good time and space performance. Its runtime and space complexities are both linear in the file length. It performs well on large files too. The algorithm can even be speeded up by using hash codes for the elements of the strings. A major drawback is the possibility of detecting wrong differences. If there are much duplicate elements, the algorithm provides bad results. Another algorithm providing move (and add) operations was introduced by Tichy [18]. Its runtime performance is *O(mn)* and the space complexity is in *O(m+n)* with *m* and *n* the length of string *A* and string *B*.

Chapter 3

Tree-Based Diff Algorithms

In this chapter the most important tree-based diff algorithms are discussed. The basic definitions used by the algorithms are similar to those of the flat-based case. First, this definitions are adapted for using them with tree-based algorithms, followed by the introduction of the notion of *mappings*, which is the most important term for tree-based algorithms. Second, several different algorithms will be presented. In Section 3.1 the algorithm due to Tai is presented. Section 3.2 introduces the algorithm by Zhang & Shasha. The *X-Diff* algorithm is covered in Section 3.3, followed by the *LaDiff* algorithm in Section 3.4. The chapter is concluded by miscellaneous algorithms in Section 3.5.

As its name implies, a tree-based algorithm needs two trees as input, in the following called source tree T_1 and destination tree T_2. A tree *T* is a set of hierarchical structured nodes. There exists three types of nodes: a *root node* is the root of the tree. It has zero or more children[1] but no parent. An *inner node* has exactly one parent and at least one children. A *leaf node* has one parent but no children. If the root and every inner node has exactly one child, the tree degenerates into a list. An example of a (binary) tree is shown in Figure 3.1.

A set of trees is called a forest *F*. $|T|$ and *Nodes(T)* denote the size and *Nodes(T)* the number of nodes of the tree *T*. *T[i]* is the *i*th node of tree *T*, *T[i..j]* stands for the nodes numbered i to j and *T(v)* is the subtree rooted at node *v*. *F(v)* is the forest retrieved by deleting *v* from *T(v)*. *Parent(v)* and *P(v)* denote the distinguished parent of node *v* and *l(v)* is the leftmost leaf descendant of node *v*. The empty tree is denoted by $\emptyset$.

In the *Diffeif* library the tree is obtained by first translating the two code snippets, of which we want to calculate the difference, to XML representing the abstract

[1]In a binary tree every node has at most two children. For most tree-based diff algorithms the number of children does not matter.

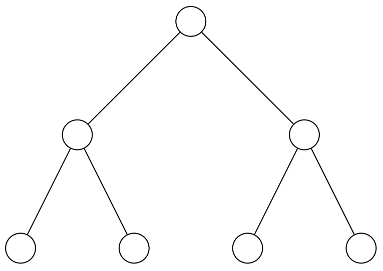


Figure 3.1: An example of a tree.

syntax tree structure of the code and then parsing each XML file into a DOM-tree, on which the tree-diff algorithm can operate. A detailed description of this steps can be found in Chapter 4. Of course, the tree-based algorithms are not restricted to trees representing the abstract syntax tree of a piece of code or to a DOM-tree, and therefore a general tree is assumed throughout this chapter. There are two major tree characteristics which distinguish the different algorithms: *ordered* and *unordered* trees. In an ordered tree the order of the children of a node is important, whereas for a unordered tree it does not play a role. For unordered trees the calculation of the difference is NP-hard[2], i.e. the difference is not computable unless for special cases (e.g. constant degree trees) or if some restrictions are applied (e.g. a structure preserving restriction allowing only mappings of disjoint subtrees in the source tree to disjoint subtrees in the destination tree).[1, 23, 19] Most algorithms operate on rooted, ordered trees.

The algorithms either use *preorder* or *postorder* traversal to number the nodes of a tree. Both are depth-first traversal methods. In preorder traversal the root of a subtree is visited first, followed by visiting the left subtree and then the right subtree recursively. In a postorder numbering the root node of a subtree is visited last, i.e. first, the left subtree followed by the right subtree is traversed recursively and at last the root of the subtree is visited. Which traversal method the different algorithms are using is always stated at the beginning of the corresponding section. In Figure 3.2 an example for preorder and postorder traversal is shown.

The tree-based diff algorithms calculate an edit script which turns the source tree into the destination tree. Remember that an edit script is a sequence of edit operations. Available edit operations of the tree-based diff algorithm in the *Diffeif* library are *delete*, *add* (*insert*), *change*, *move* and *copy*. Note that the distance metrics introduced for the flat-based diff algorithms, such as the Levenshtein distance or the Hamming distance, cannot directly be used for tree-based diff. A very

[2]For NP-hard (non-deterministic polynomial-time hard) problems it is only possible to verify that a solution of the problem is correct, but it is not possible to compute a solution in polynomial time.

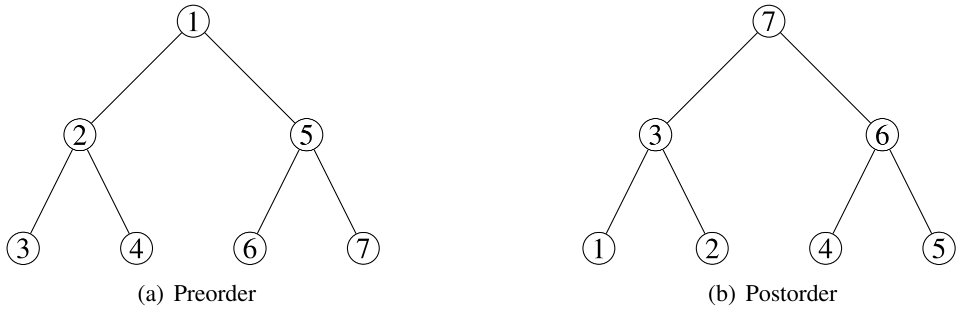


Figure 3.2: Preorder and postorder traversal.

important difference to the flat-based diff algorithms is the fact that detection of moves (for ordered and unordered trees) is NP-hard.[8, 12, 10] There exists some algorithms, e.g. [2], which are able to detect moves but they can only approximate the result, and hence the resulting edit script could either be non minimal or visually not appealing. Of course, it is possible to build the move (and copy) operation from the add and delete operations in a postprocessing step, but this can also lead to visually not appealing results. In the *Diffeif* library move (and copy) operations are optionally derived in such a postprocessing step. The different edit operations have a slightly different semantic than in the flat-based case but the appearance is similar. An edit operation is defined in the *Diffeif* library in the same way as $O = s1, s2Xd1, d2$ with $X \in \{a, c, d, m, t\}$, *s1 = source start index*[3], *s2 = source end index*, *d1 = destination start index* and *d2 = destination end index*.

- $Insert(x, y, k, s, t)$ appends node *y* from the destination tree as the *k*th child to node *x* in the source tree, making the children *s* to *t* of node *x* the children of node *y*. This can be simplified to *Insert(x,y)* or $s1 = s2 = x$, $X = a$ and $d1 = d2 = y$.[4] Note that *x*, *y*, *s* and *t* refer to the node number and not the node label.

- $Delete(x)$ or $s1 = s2 = x$, $X = d$ and $d1 = d2 = \lambda$[5] deletes node *x* from the source tree. The children of node *x* are appended to the parent of node *x*.

- $Change(x, y)$ or $s1 = s2 = x$, $X = c$ and $d1 = d2 = y$ changes node *x* in the source tree to node *y* in the destination tree.

[3]The index refers to the node numbers.

[4]In most cases (e.g. for visualizing the difference) it is not necessary to know at which position in the source tree node *x* was inserted

[5]λ denotes the empty string.

- $Move(x, y)$ or $s1 = s2 = x$, $X = m$ and $d1 = d2 = y$ moves node *x* in the source tree to node *y* in the destination tree.
- $Copy(x, y)$ or $s1 = s2 = x$, $X = t$ and $d1 = d2 = y$ inserts node *y* in the destination tree as a copy of node *x* in the source tree.

Figure 3.3 provides an example of an edit sequence containing the edit operations introduced above.

As for flat-based diff algorithms the minimal edit script is searched, i.e. the edit script with minimal cost.[6] For this purpose a cost function $\gamma : (x \rightarrow y) \longrightarrow \Re_0^+$ with $x \in Nodes(T_1) \cup \lambda$, $y \in Nodes(T_2) \cup \lambda$, λ a special blank symbol and T_1 and T_2 the source and destination tree, is introduced.[10] To each operation $a \rightarrow b$ a nonnegative real number $\gamma(a, b)$ is assigned. $\gamma(a \rightarrow \lambda)$ is the assigned cost of deleting node *a*, $\gamma(\lambda \rightarrow b)$ is the cost for the deletion of node *b* and $\gamma(a \rightarrow b)$ is the cost for changing node *a* into node *b*. If $a = b$, it holds that $\gamma(a \rightarrow b) = 0$. In contrast to the flat-based diff algorithms, it is possible to assign a cost to each edit operation depending on the node instead of a global cost. For example, a deletion of a node labelled x could introduce a higher cost than deleting a node labelled y. Therefore, the costs of the edit operations have to be chosen carefully depending on the tree structure (for each tree separately). If an operation is assigned a very high or infinite cost, the edit operation is most likely not used.

The notion of a *mapping* is crucial for tree-based diff algorithms.[1, 17, 20] A mapping is a connection of the source and destination tree which maps corresponding nodes to each other. It describes indirectly the edit operations used for transforming a tree into another tree but ignoring the order of the edit operations. Formally, a mapping is defined as a set of tuples (a, b), where $a \in Nodes(T_1)$ and $b \in Nodes(T_2)$, satisfying for any pair (a_1, b_1) and (a_2, b_2) in the mapping[1, 17, 20]:

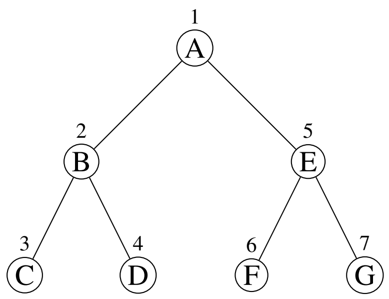
(1) $a_1 = a_2 \Longleftrightarrow b_1 = b_2$ (one-to-one mapping)

(2) a_1 is to the left of $a_2 \Longleftrightarrow b_1$ is to the left of b_2 (preserve sibling order)

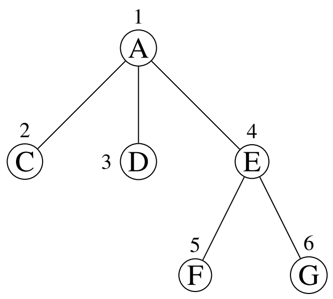
(3) a_1 is an ancestor of $a_2 \Longleftrightarrow b_1$ is an ancestor of b_2 (preserve ancestor order)

These structural preserving conditions imply that two nodes *a* and *b* are in the mapping if and only if both nodes are equal or the corresponding edit script contains an edit operation *Change(a,b)*, and the above constraints hold for every other pair of nodes in the mapping. The edit script can easily be derived from

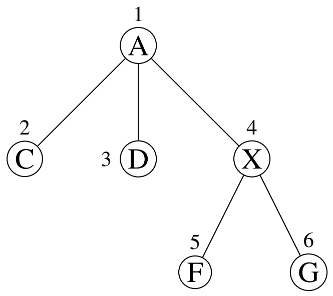
[6] If unit costs of one are used for each edit operation, the minimal edit script corresponds to the minimum length edit script, i.e. the edit script with the smallest number of edit operations.



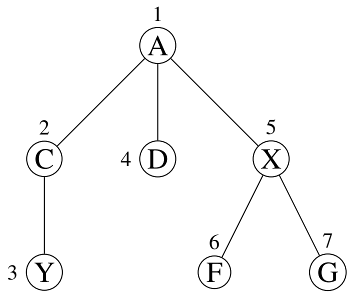
(a) The original tree



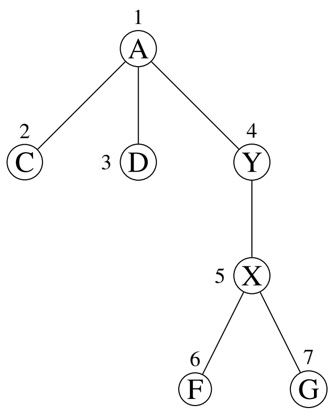
(b) Delete(2)



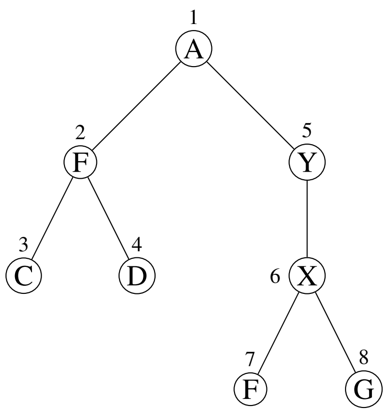
(c) Change(4,4)



(d) Insert(2,3)



(e) Move(3,4)



(f) Copy(6,2)

Figure 3.3: An example of an edit sequence on trees.

a mapping by creating a delete operation for every node of the source tree not contained in the mapping, generate an insert operation for every not mapped node in the destination tree and introducing a change operation for every two non equal nodes in the mapping. Similarly, it is easy to get a mapping from an edit script.

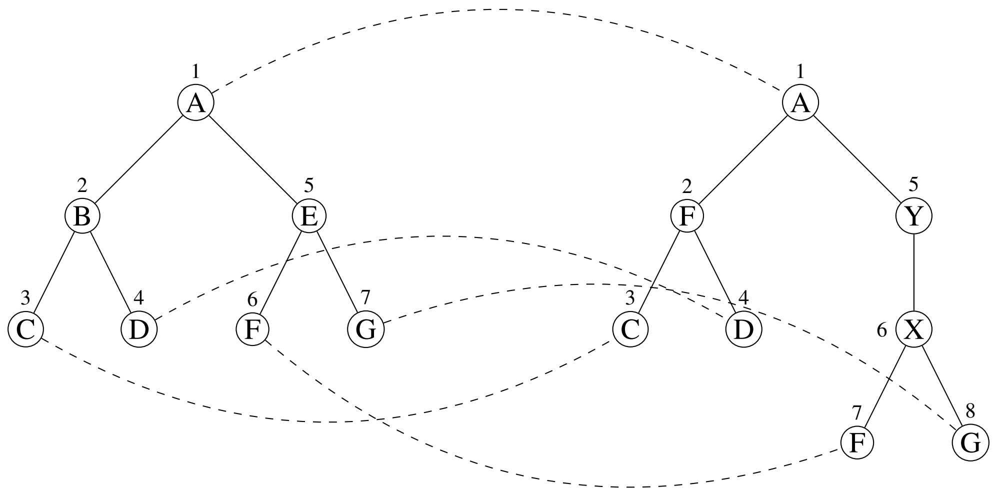


Figure 3.4: A sample mapping.

Figure 3.4 shows a mapping of the tree in Figure 3.3 (a) as the source tree T_1 and the tree in Figure 3.3 (f) as the destination tree T_2. The corresponding edit script[7] is $[Delete(2), Delete(5), Insert(1,2), Insert(1,5), Insert(5,6)] = [2d, 5d, 1a2, 1a5, 5a6]$. There is an infinite number of possible edit scripts which turn the source tree into the destination tree. Thus, it is impossible to enumerate all edit scripts and pick the one with the minimum costs (or the shortest one if the same costs for all edit operations are used). That is why mappings are important because with mappings it is possible to compute the minimal edit script in polynomial time.[17]

[7]Instead of source/destination start and end index only one index is shown.

3.1 Tai's Algorithm

In this section the algorithm introduced by Tai[17] for solving the *tree-to-tree correction problem*[8] on *ordered* tree is presented. It has not a good time and space complexity and is impractical to implement due to the complexity of the algorithm. Nevertheless, it is a basis of the algorithm shown in the next section. Tai's algorithm is a generalization of the *string-to-string correction problem* introduced in the previous chapter. In fact, each string can be represented as a tree of depth two with a virtual root node. The algorithm provides *change*, *insert* and *delete* operations. It uses a preorder traversal to number the nodes. This has the advantage that the nodes *T[1]* to *T[i]* of a tree *T* form a subtree rooted at *T[1]*. *T[i]* denotes the node with number *i* of tree *T*. It is assumed that the root of the source tree T_1 and of the destination tree T_2 are equal and remains unchanged.

As for the flat-based algorithms, a *dynamic programming* approach is used with the following recursion[17]:

$$\begin{aligned} D_{m,n} = \min(&D_{m-1,n} + \gamma(T[m] \to \lambda), \\ &D_{m,n-1} + \gamma(\lambda \to T[n]), \\ &\text{MIN\_}M(m,n)) \end{aligned}$$

The base conditions are $D_{1,1} = 0$, $D_{i,1} = \sum_{k=2}^{i} \gamma(T_1[k] \to \lambda)$ and $D_{1,j} = \sum_{k=2}^{j} \gamma(\lambda \to T_2[k])$ for $1 < i \leq |T_1|$ and $1 < j \leq |T_2|$. $D_{m,n}$ is the edit distance[9] from the nodes $T_1[1], ..., T_1[m]$, also denoted as $T_1(1 : m)$, to the nodes $T_2[1], ..., T_2[n]$, referred to as $T_2(1 : n)$. The resulting edit distance containing all nodes is obtained by setting m to the number of nodes in tree T_1 and n to the number of nodes in tree T_2.

As can be seen in the formula above, the value of the edit distance $D_{m,n}$ can be obtained by solving a finite number of subproblems (mappings) of smaller size, hence the edit distance can be computed by solving these smaller subproblems (mappings) in advance and then compute the final distance by composing the subproblems (mappings). Suppose M is a minimum cost mapping (also called an optimal mapping) from $T_1(1 : m)$ to $T_2(1 : n)$. For an optimal mapping there are only three possible cases for the last element of each tree:

(1) $T_1[m]$ is not in the mapping. The mapping M' from $T_1(1 : m-1)$ to $T_2(1 : n)$ must be minimal and $Cost(M) = Cost(M') + \gamma(T_1[m] \to \lambda) = D_{m-1,n} + \gamma(T_1[m] \to \lambda)$.

[8]The tree-to-tree correction problem is just another name for finding the minimal cost edit script.
[9]The corresponding edit script can be computed in parallel.

(2) $T_2[n]$ is not in the mapping. The mapping M' from $T_1(1:m)$ to $T_2(1:n-1)$ must be minimal and $Cost(M) = Cost(M') + \gamma(\lambda \rightarrow T_2[n]) = D_{m,n-1} + \gamma(\lambda \rightarrow T_2[n])$.

(3) $T_1[m]$ and $T_2[n]$ are in the mapping. The mapping M' from $T_1(1:m-1)$ to $T_2(1:n-1)$ must be minimal. This is a special case because it is possible under some circumstances that $Cost(M) = Cost(M') + \gamma(T_1[m] \rightarrow T_2[n]) > D_{m-1,n-1} + \gamma(T_1[m] \rightarrow T_2[n])$, i.e. it is not allowed to add the nodes $T_1[m]$ and $T_2[n]$ to the mapping M'.

Case (3) needs a deeper examination. It is not possible to add $T_1[m]$ and $T_2[n]$ to every matching from $T_1(1:m-1)$ to $T_2(1:n-1)$ because constraint (3) of the matching definition on page 30 could be violated, i.e. the ancestor order would be no longer preserved. Figure 3.5 shows an example of such a mapping.

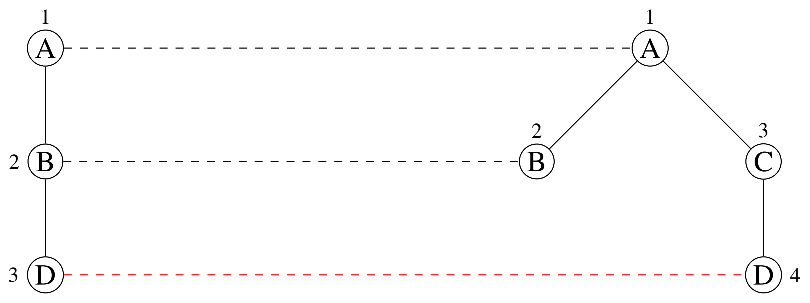


Figure 3.5: An illegal mapping.

The minimum mapping from $T_1(1:2)$ to $T_2(1:3)$ is $\{(1,1),(2,2)\}$.[10] If the nodes $T_1[3]$ and $T_2[4]$ are added to the mapping (red dashed line), the resulting mapping $\{(1,1),(2,2),(3,4)\}$ is not a legal mapping from $T_1(1:3)$ to $T_2(1:4)$ because $T_1[2]$ is an ancestor of $T_1[3]$ but $T_2[2]$ is not an ancestor of $T_2[4]$. Thus, the minimum cost mapping from $T_1(1:3)$ to $T_2(1:4)$ is $\{(1,1),(2,2)\}$.

$MIN\_M(m,n)$ was introduced in the dynamic programming formula to ensure that only legal mappings are achieved. $MIN\_M(m,n)$ is the minimum cost mapping from $T_1(1:m)$ to $T_2(1:n)$ with $T_1[m]$ and $T_2[n]$ included in the mapping.[11]

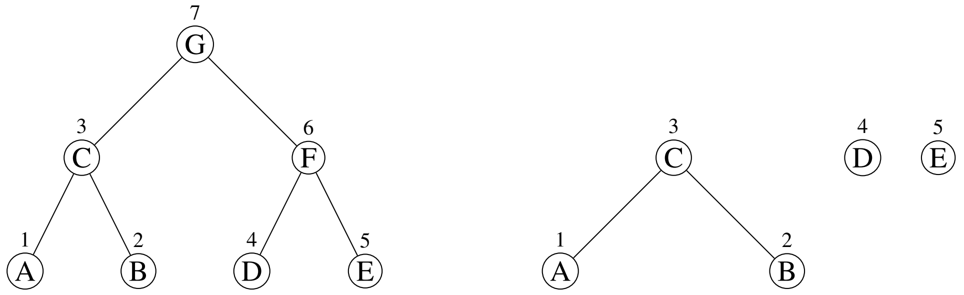
As stated in [17], the time and space complexity of the algorithm is in $O(V_1 * V_2 * L_1^2 * L_2^2)$ with V_1 and V_2 the number of nodes in tree T_1 and T_2 and L_2 and L_2 the maximum depth of T_1 and T_2, respectively.

[10]Same costs for all edit operations are assumed.

[11]The formal definition of $MIN\_M(m,n)$ is not given here. It can be looked up in [17].

3.2 Zhang & Shasha's Algorithm

The algorithm due to Zhang & Shasha[20] is an improvement of Tai's algorithm presented in the last section. It has a better time and space complexity and, moreover, it is simpler. The algorithm can only be used for *ordered* trees and provides *change*, *insert* and *delete* operations. In intermediate steps, the algorithm calculates the distance between two ordered forests. $forestdist(T_1[i_1..j_1], T_2[i_2..j_2])$ designates the distance between $T_1[i_1..j_1]$ and $T_2[i_2..j_2]$. The final result is then composed of these intermediate results. Therefore, the algorithm uses a postorder numbering of the nodes. This has the advantage that $T[i..j]$ (the nodes numbered *i* to *j*) denotes an ordered subforest of *T*. In Figure 3.6 an example is shown of a forest induced by the nodes T[1..5].



(a) The tree (b) The forest induced by *T[1..5]*

Figure 3.6: An example of a forest.

Moreover, mappings for forests are defined in the same way as for trees and $i > j$ implies $T[i..j] = \emptyset$. Zhang & Shasha's algorithm is implemented in the *Diffeif* library, and therefore it is explained in detail by giving a formal definition of the algorithm, followed by Eiffel pseudocode.

First, the set of *keyroots* for the source tree T_1 and the destination tree T_2 has to be computed. The keyroots are the nodes which have a left sibling, i.e. $l(k) \neq L(p(k))$ for all $k \in Nodes(T)$, plus the root of the tree. Remember that $P(k)$ denotes the parent of node k and $l(k)$ is the leftmost leaf descendant of node k. The keyroots of the tree in Figure 3.6 (a) are $\{2, 5, 6, 7\}$. For example, node 6 (labelled with F) has the leftmost leaf descendant $l(6) = 4$ but $l(p(6)) = l(7) = 1$ and thus node 6 is a keyroot. On the other hand, $l(3) = 1$ and $l(p(3)) = l(7) = 1$ and therefore node 3 is not a keyroot. Note that the number of keyroots is always equal to the number of leaves in the tree.[4] In addition to the keyroots, the leftmost leaf descendant for each node of the tree must also be precomputed. For the tree in Figure 3.6 (a) the leftmost leaf descendants are $[1, 2, 1, 4, 5, 4, 1]$. For example, the leftmost leaf of node 6 is node 4 and therefore a 4 is stored in the array at position 6. The keyroots and the leftmost leaf descendants are stored in arrays. Their computation can be done in linear time.

To calculate the minimum cost mapping from a node in the source tree to a node in the destination tree (stored in a permanent tree distance array), the children of the corresponding nodes have to be mapped, i.e. all keyroots plus the leftmost child (remember that the leftmost child is not part of the keyroots). The keyroots represent the nodes which need separate computation. To obtain the minimum cost mapping of a keyroot, the minimum cost mapping of its subforests, stored in a temporary forest distance array, has to be computed. For each node, which is not a keyroot (the leftmost child of each parent node), the mapping can directly be read out from the tree distance array.[12]

Tai's algorithm computed the solution top down, and therefore it was necessary to record several solutions for later backtracking if an illegal mapping was detected on the lower levels. Zhang & Shasha's algorithm does the computation bottom up, as a consequence only one solution to a subproblem has to be remembered because the minimum cost mapping of all descendants of each node have been calculated before the node is visited.

For computing the minimum cost mapping between two forests $T_1[l(i_1)..i]$ and $T_2[l(j_1)..j]$, where $i \in \mathit{keyroots}(T_1)$, $j \in \mathit{keyroots}(T_2)$, $i_1 \in [l(i), ..., i]$, $j_1 \in [l(j), ..., j]$ and i_1 and j_1 are ancestors of i and j, respectively, the following dynamic programming recursion can be used[20]:

$$\begin{aligned}
&\mathit{forestdist}(T_1[l(i_1)..i], T_2[l(j_1)..j]) = \min(\\
&\qquad \mathit{forestdist}(T_1[l(i_1)..i-1], T_2[l(j_1)..j]) + \gamma(T_1[i] \to \lambda), \\
&\qquad \mathit{forestdist}(T_1[l(i_1)..i], T_2[l(j_1)..j-1]) + \gamma(\lambda \to T_2[j]), \\
&\qquad \mathit{forestdist}(T_1[l(i_1)..l(i)-1], T_2[l(j_1)..l(j)-1]) \\
&\qquad + \mathit{forestdist}(T_1[l(i)..i-1], T_2[l(j)..j-1) + \gamma(T_1[i] \to T_2[j]))
\end{aligned}$$

The border conditions are:

(i) $\mathit{forestdist}(\emptyset, \emptyset) = 0$

(ii) $\mathit{forestdist}(T_1[l(i_1)..i], \emptyset) = \sum_{k=l(i_1)}^{i} \gamma(T_1[k] \to \lambda)$

(iii) $\mathit{forestdist}(\emptyset, T_2[l(j_1)..j]) = \sum_{k=l(j_1)}^{j} \gamma(\lambda \to T_2[k])$.

[12]Note that this reduces the time complexity because, instead of computing the mapping of the subforests for all nodes, it must only be done for the *keyroots*.

In the following, the intuition for the above recursion is presented. There are three ways of computing the minimum cost mapping between two forests $T_1[l(i_1)..i]$ and $T_2[l(j_1)..j]$:

(1) $T_1[i]$ is not in the mapping. Therefore, the cost of deleting $T_1[i]$ and the forest distance of the remaining nodes have to be summed up.

(2) $T_2[j]$ is not in the mapping. Therefore, the cost of inserting $T_2[j]$ and the forest distance of the remaining nodes have to be summed up.

(3) If $T_1[i]$ and $T_2[j]$ are both in the mapping, the cost of changing $T_1[i]$ in $T_2[j]$, the forest distance of the remaining nodes and a special term for guaranteeing the mapping restrictions have to be summed up.

Eiffel pseudocode for Zhang & Shasha's algorithm can be found in appendix B.2.[13] The computation of the leftmost leaf arrays (lml_T1 and lml_T2) and the keyroots arrays ($keyroots\_T1$ and $keyroots\_T2$) is not shown because the calculation is straightforward. The two outermost loops iterate over the keyroots of T_1 and T_2. In the lines 24–40 the border conditions (ii) and (iii) are applied. In the two innermost loops (lines 42–76) the forest distances are computed according to the recursive dynamic programming algorithm. The final distance between the two trees is available in *treedist(m,n)*, where m and n are the number of nodes in tree T_1 and T_2, $m = Nodes(T_1)$ and $n = Nodes(T_2)$. It is easy to extend the pseudocode to compute the corresponding minimal cost edit script in parallel to the calculation of the minimal edit distance (the minimal cost mapping).

The time complexity of the algorithm is in $O(|T_1| * |T_2| * min(height(T_1), leaves(T_1)) * min(height(T_2), leaves(T_2)))$ and the space complexity is in $O(|T_1| * |T_2|)$, where $|T|$ is the number of nodes in tree T, $leaves(T)$ is the number of leaf nodes in tree T and $heigth(T)$ is the height of tree T. In addition, the algorithm can be parallelized, leading to a time complexity of $O(|T_1| * |T_2|)$.[20] Due to the good time and space complexity, the algorithm is currently the reference for tree-based diff algorithms.[4]

3.3 X-Diff Algorithm

The *X-Diff* algorithm due to Wang, DeWitt and Cai[19] provides *insert*, *delete* and *change* edit operations. The algorithm is an instance of the class of diff algorithms for *unordered* trees, i.e. trees in which only ancestor but not sibling

[13]Note that it is only pseudocode. For the real code take a look at the *Diffeif* library.

relationships are relevant. As mentioned at the beginning of this chapter, the problem of computing the difference of unordered trees is NP-hard.[1, 23, 19] Hence, strong assumptions have to be made to solve it in polynomial time.[10] Due to these assumptions, the algorithms are efficient but provide not exact results. *X-Diff* also uses the notion of a *mapping*[14]. It is defined in the same way as for Zhang & Shasha's algorithm except that children are only allowed to map if their ancestors are mapped. Furthermore, each node is assigned a node *signature* which is used for comparing two nodes (Two nodes are equal if they have the same signature). This reduces the mapping space and accelerates the algorithm. The *X-Diff* algorithm is not usable for the *Diffeif* library because in our case it is important to maintain the order of child nodes. Nevertheless, in the following *X-Diff* is presented shortly for giving an idea of tree-based diff algorithms for unordered trees.

There are three steps in *X-Diff*. The first step is a preprocessing step. In this step every node in the source and destination tree gets annotated with a hash value representing the entire subtree rooted at that node. This hash value corresponds to the signature of the node mentioned above. Note that for two isomorphic trees[15] all nodes of the source tree have the same hash value as those in the destination tree because the algorithm operates on unordered trees.

The second step consists of creating a minimum cost mapping between the nodes of both trees. The algorithm starts at the bottom of the tree (the highest level) and proceeds bottom up from level to level until the top of the tree (the lowest level) is reached. At each level, first, all equivalent subtrees are filtered out. This is done by looking at the corresponding hash values. If the hash value of a node in the source tree is equal to the hash value of a node in the destination tree, the corresponding subtrees are most likely equal and they can be filtered out (remember that a hash value of a node represents the whole subtree rooted at that node). Because most of the time the difference is rather small, there are often subtrees to filter out and thus the algorithm is speeded up. Second, the mapping for the remaining subtrees at the corresponding level is computed recursively with dynamic programming as done in the previous algorithms. Very important is the fact that the mapping must only be computed between nodes having the same signature, as stated in [19]. This reduces the mapping space and transfers the NP-hard problem to a polynomial time problem.

In the third and last step, the minimum cost edit script is generated from the minimum cost mapping. This is done by recursively traversing all nodes in both trees bottom up.

The time complexity of *X-Diff* is in $O(|T_1| * |T_2| * \max(deg(T_1), deg(T_2)) *$

[14]In the *X-Diff* paper it is called a matching.

[15]Trees are isomorphic if they have the same shape.

$log_2(\max(deg(T_1), deg(T_2)))$, where $deg(T)$ is the maximum out-degree of a tree T.

3.4 LaDiff Algorithm

The Tool *LaDiff* and the corresponding algorithm are described in the paper of Chawathe et al[2]. Although *LaDiff* is a tool for comparing Latex documents, it is possible to use the algorithm in other cases. The *LaDiff* algorithm is an instance of the class of diff algorithms for *ordered* trees. Beside *insert*, *delete* and *change* operations, a *move* edit operation is provided. Thus, the definition of the edit operations on page 29 have to be extended by the move operation $Move(x, y, k)$ which moves the subtree rooted at node x in the source tree, so that x gets the kth child of node y in the destination tree. There are only few algorithms able to detect moves[10] due to the fact that the detection of moves is NP-hard in the tree-diff case[8, 12, 10] and therefore some restrictions have to be introduced to make the problem solvable. These restrictions lead to a faster runtime of the *LaDiff* algorithm compared with the previously presented algorithms at the expense of the possibility of producing non minimal cost edit scripts. The creation of a minimum cost edit script can be guaranteed if there are not too many duplicate nodes.[16] Of course, it is also possible to construct moves in a postprocessing phase from insert and delete operations. If one is interested in an exact result (as for the *Diffeif* library), this is the way to go.

The algorithm performs two steps. First, the minimal cost *mapping*[17] between the nodes of both trees is computed. Second, the minimum cost edit script is generated from the mapping. The mapping constraints introduced on page 30 are extended by two additional restrictions: (1) Two nodes which are too different, must not be mapped to each other[18] and (2) a pair of internal nodes are only allowed to be in the mapping if they have a certain number of common descendants. The mapping is obtained by scanning through the tree levels bottom-up. At each level k, an initial mapping between the nodes of level k is acquired by computing the longest common subsequence (LCS) between the nodes of both trees at level k and add all nodes in the LCS to the mapping. Afterwards, the initial mapping is extended by scanning through the still unmapped nodes and compare them with every node in the other tree at level k.

The generation of the minimal cost edit script from the minimal cost mapping

[16] It is not further explained what too many is.

[17] In the paper [2] it is called a matching but the meaning is the same.

[18] This can be achieved by using a compare function returning a real number in a specific interval. If the value is above a certain threshold, the nodes must not map to each other.

involves five phases. In the first phase, called the update phase, for each pair of dissimilar nodes in the mapping, a change operation is added to the edit script. The second phase is the align phase. A pair of internal nodes *x* and *y* in the mapping has misaligned children, if node *u* and *v* are children of node *x* in the source tree T_1 and *u* is to the left of *v* but the partner of *u* in the destination tree T_2 is to the right of the partner of *v*. If a pair of internal nodes in the mapping have misaligned children, move operations, so called *intra-parent moves*, are added to the edit script to align the children (bring them to the right order). In the third phase, for every node in the destination tree T_2 which is not in the mapping, an insert operation is added to the edit script. In the fourth phase, the *move phase*, every node *x* in the source tree T_1 is moved if the parent of *x* does not map to the parent of the partner of *x* in the destination tree T_2. The last phase is the *delete phase*. For all nodes in the source tree T_1 which are not in the mapping, a delete operation is added to the edit script. For the first four phases a breadth-first scan is applied and for the last phase a postorder traversal is used. It is also easy to see that the first, third and fifth phase are similar to the translation of a mapping to an edit script described in the previous sections, only the second and fourth steps are new.

The time complexity of the algorithm is in $O(ne + e^2)$, where n is the number of leaves in the trees and e is the (weighted) edit distance, i.e. the runtime is proportional to the number of nodes multiplied with the number of differences. Because most of the time the number of differences, and thus the edit distance, is small compared with the size of the tree, it holds that e $\ll$ n and therefore the algorithm is fast.

3.5 Miscellaneous Algorithms

In this section, the chapter about tree-based diff algorithms is concluded by the listing of a selection of miscellaneous algorithms together with their runtime. The algorithms are not explained in detail, only an overview of additional tree-based diff algorithms is given. Short descriptions of the algorithms can be found in [1]. All algorithms provide *insert*, *delete* and *change* edit operations.

In [12] Klein introduced an algorithm for *unrooted ordered* trees based on dynamic programming similar to the one due to Zhang & Shasha in Section 3.2. The algorithm requires fewer subproblems to be solved in the worst case and thus it is faster. The time complexity of the algorithm is in $O(|T_1|^2\,|T_2|\log|T_2|)$ and the space complexity is in $O(|T_1|\,|T_2|)$.

Chen [3] proposed another more advanced algorithm, using *fast matrix multiplicaton*, for *ordered* trees. The runtime is in $O(|T_1|\,|T_2| + L_1^2\,|T_2| + L_1^{2.5}L_2)$ and the space usage is in $O((|T_1| + L_1^2)\min(L_2, D_2) + |T_2|)$, where L is the number of leaves and D is the depth of a tree.

In [22, 21] Zhang introduced two algorithms for the *constraint* edit distance between two trees for the *unordered* and the *ordered* case. The restrictions introduced is that disjoint subtrees must be mapped to disjoint subtrees only. The time complexity in the *unordered* case is $O(|T_1|\,|T_2|\,(I_1 + I_2)\log(I_1 + I_2))$, where I is the maximum degree of a tree, and for *ordered* trees $O(|T_1|\,|T_2|)$ time is needed. Both algorithms use $O(|T_1|\,|T_2|)$ space.

Another algorithm due to Zhang & Shasha[16] for *rooted ordered* trees uses the unit cost model, i.e. each edit operation is assigned the same cost. Thus, the edit distance is equal to the number of edit operations. The time complexity of the algorithm is $O(c^2 \min(|T_1|\,, |T_2|) \min(L_1, L_2))$, where c is the edit distance between T_1 and T_2. The space complexity is $O(|T_1|\,|T_2|)$.

The last algorithm mentioned here is the one from Selkow[15]. Insert and delete operations are confined to the leaves of the trees, and thus the algorithm is simple but does not always provide the most intuitive edit script. For example, if an internal node is removed, the algorithm removes all descendants of that node instead of only removing the affected node. The algorithms needs $O(|T_1|\,|T_2|)$ time and space.

Chapter 4

Implementation

In this chapter the architectural design details of the *Diffeif* library and the GUI are revealed. Throughout this chapter it is assumed that the input consists of two code snippets, which corresponds to the intended usage of the library and the GUI, although other types of input are also possible. As seen in Chapter 3, the tree-based diff algorithm works on trees. Therefore, the code snippets have to be converted into an intermediate data format representing the hierarchical structure of the code which then can be parsed into a DOM-tree. This is discussed in Section 4.1. In the Sections 4.2 and 4.3 the class diagrams of the *Diffeif* library and the corresponding GUI are given and explained in detail. The implementation details of the classes are not given here due to the large amount and the size of the classes. With the comments in the code and the explanations of the algorithms in the previous chapters, it should not be a problem to understand the implementation of the classes.

4.1 Internal Data Representation

The flat-based diff algorithm operates directly on the code snippets and thus no conversion into another format is needed. Of course, any other textual input can be provided to the flat-based diff algorithm. Remember that flat-based diff is an umbrella term that encompasses line-based, word-based and character-based diff, but the underlying diff algorithm is the same for all three types of diff.

The tree-based diff algorithm implemented in the *Diffeif* library needs as input two XML files, hence the code snippets have to be converted into such files beforehand. In the case of code snippets, the XML file represents the abstract syntax tree of the code. Note that every code snippet has an implicit structure visible to humans but not to machines, and therefore a conversion to another format, e.g. XML, on which machines can derive the hierarchical structure, is required. Of course, the tree-based diff algorithm is not only restricted to XML files representing code. In fact, it can be used for a wide range of XML files. In Listing 4.1 a small example code snippet is shown.

Listing 4.1: Example code in Eiffel.

```
prune_first (n: INTEGER)
        do
                prune (n, 1)
        end
```

The corresponding XML file representing the abstract syntax tree of that code snippet is available in the appendix in Listing C.1 on page 71.[1] The XML file is shown in a prettified format providing better human readability. The first line of the XML file contains the XML *declaration* stating the used XML version and the encoding. The second line defines a XML *namespace* identifying the unique vocabulary of the XML document. The rest of the XML File consists of three constructs: *elements*, *attributes* and *texts*.[2] An element either begins with a *start tag*, e.g. $< start\_tag >$, and ends with an *end tag*, e.g. $< end\_tag >$, or it is made up of an *empty element tag*, e.g. $< empty\_element\_tag\ / >$. Attributes consists of a name/value pair and are located inside a start tag or an empty element tag, for example $< start\_tag\ name = "value" >$. Texts appear inside an element, i.e. between a start and an end tag. Note that in the example new line symbols (%N) are explicitly shown in the text elements. The XML file can easily be retrieved from a code file by using *EVE*, the *Eiffel Verification Environment*[3].

The XML files are internally parsed into separate DOM-trees using some libraries of *EVE*. Afterwards, the tree-based diff algorithm runs on these DOM-trees. The DOM-tree retrieved from the XML file in Listing C.1 is shown in the appendix in Figure C.1 on page 73. There are two types of nodes: *element nodes* and *text nodes*. Element nodes are ordered internal nodes with one label, the name. Text nodes are ordered leaf nodes with one label, the value. They are shown as a dashed rectangle. According to the Document Object Model specification by the *World Wide Web Consortium*[4], the attributes should form separate unordered leaf nodes with two labels, name and value, but for the *Diffeif* library this is not necessary, and therefore the attributes are annotated at the corresponding element node. As can be seen in Figure C.1, the small code snippet introduces a lot of nodes, and

[1]The XML file was provided by my supervisor Yu Pei.

[2]In general, there are much more constructs available for a XML file but in our case three are enough.

[3]`http://se.inf.ethz.ch/research/eve/`. Retrieved 2012-12-14.

[4]`http://www.w3.org/TR/REC-xml/`. Retrieved 2012-12-14.

hence the tree-based diff algorithm is slowed down. As a consequence, the *Diffeif* library offers additional functionality for skipping unnecessary nodes, i.e. internal element nodes with only one child not being a text node. Further speedup offered by the *Diffeif* library is the merging of element and text nodes into one single node. In the majority of cases, the tree-based diff algorithm returns accurate results using this optimisation because most element nodes have either none or only one text node as a child. Let us assume that the keywords *if* and *end* form two text nodes belonging to the same parent element node. If the conditional branch is removed from the code, both keywords disappear from the DOM-tree, and thus there is no disadvantage in this case if both keywords are located in the same element.

4.2 Diffeif Library

The class diagram of the *Diffeif* library is shown in Figure 4.1. A red arrow from a class *A* to a class *B* symbolizes the inheritance of class *A* from class *B*. A green arrow from a class *A* to a class *B* signifies that class *A* is a client of class *B* and class *B* is a supplier of class *A*, i.e. class *A* holds an object of the type of class *B* at runtime. Classes annotated with a star (*) are deferred and classes annotated with a plus (+) are effective.

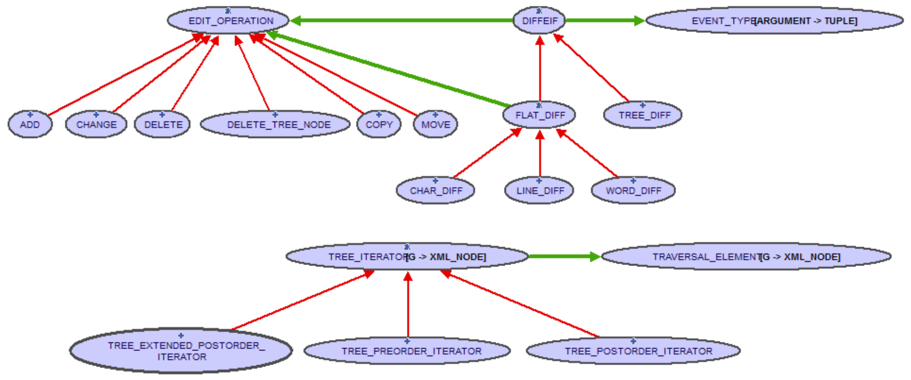


Figure 4.1: Class diagram for the *Diffeif* library.

`DIFFEIF` is a deferred class encapsulating common features for tree-based diff and flat-based diff. `FLAT_DIFF` is a deferred superclass containing the most functionality for `CHAR_DIFF`, `WORD_DIFF` and `LINE_DIFF` which are effective classes inheriting from the superclass `FLAT_DIFF`. The class `TREE_DIFF` inherits directly from `DIFFEIF`. The classes `DIFFEIF` and `FLAT_DIFF` are clients of the deferred class `EDIT_OPERATION` which provides the edit operations used in the edit script, more precisely it contains attributes denoting the start and end index in the source and destination of the corresponding edit operation. The effective edit operations are subclasses of the EDIT_OPERATION class and just contain a character identifier symbolizing the type of the edit operation. The class DELETE_TREE_NODE stands for a special edit operation only available for tree-based diff. Moreover, DIFFEIF is a client of the EVENT_TYPE class which implements the *publisher/subscriber* design pattern. The EVENT_TYPE class is used in the library to notify registered subscribers (publish events) about certain events. Clients of the *Diffeif* library are able to register agents to the corresponding EVENT_TYPE. Once routines ensure that the EVENT_TYPE objects in the *Diffeif* library are create once and only once at the first call to the object, either through publishing an event or subscribing an agent, similar to the *singleton* design pattern. This is also called *lazy creation*.

Last but not least, the classes TREE_ITERATOR, TREE_EXTENDED_POSTORDER_ITERATOR, TREE_POSTORDER_ITERATOR, TREE_PREORDER_ITERATOR and TRAVERSAL_ELEMENT implements the *iterator* design pattern. These classes are used to traverse trees in preorder and postorder. The class TREE_PREORDER_ITERATOR provides the iteration over a tree in preorder by visiting element nodes, text nodes and attribute nodes of the tree. The class TREE_POSTORDER_ITERATOR supports the iteration over the tree in postorder by visiting only element nodes. The class TREE_EXTENDED_POSTORDER_ITERATOR enables to iterate in postorder not only over element nodes but also over text nodes.

4.3 GUI

The class diagram of the GUI is shown in Figure 4.2.

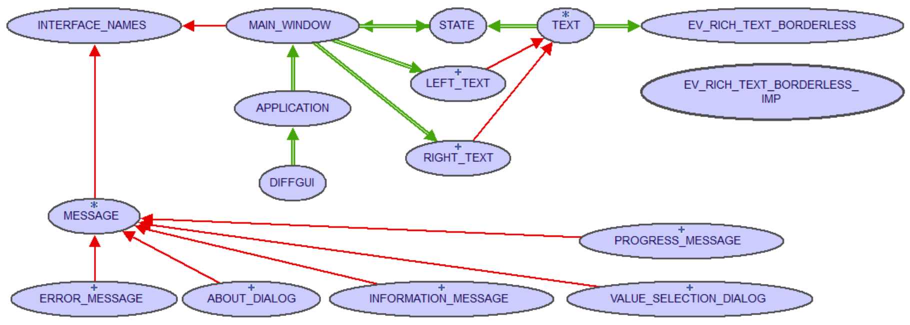


Figure 4.2: Class diagram for the GUI.

DIFFGUI is the main class of the system containing the user interface, i.e. the user must only care about this class and can use the features of the class after creating an object of the type of this class. DIFFGUI is a client of APPLICATION, the root class of the GUI, which is a client of MAIN_WINDOW. MAIN_WINDOW is the responsible class for building the GUI with the menu bar, status bar and all its subwindows. The main window contains two text boxes, a LEFT_TEXT and a RIGHT_TEXT, both inheriting from the deferred class TEXT. The classes EV_RICH_TEXT_BORDERLESS and EV_RICH_TEXT_BORDERLESS_IMP are minor redefinitions of EV_RICH_TEXT and EV_RICH_TEXT_IMP, respectively, allowing text boxes with smoother borders. The STATE class encapsulates the actual state of the GUI. This class is used by MAIN_WINDOW and TEXT and its descendants, i.e. it is a supplier to them. Furthermore, the class STATE must also know the MAIN_WINDOW class, i.e. STATE is also a client of MAIN_WINDOW.

MESSAGE is a deferred class representing different types of messages used by the GUI, e.g. error messages, information messages, value selection dialogues etc. The last remaining class is named INTERFACE_NAMES. It contains many string constants utilized by the main window and the messages.

Chapter 5

User Guide

Overall, the usage of the GUI and the library is straightforward[1]. Nevertheless, in this chapter there are some hints given, explaining the usage in more detail. Do not forget to add the *Diffeif* library to your project.

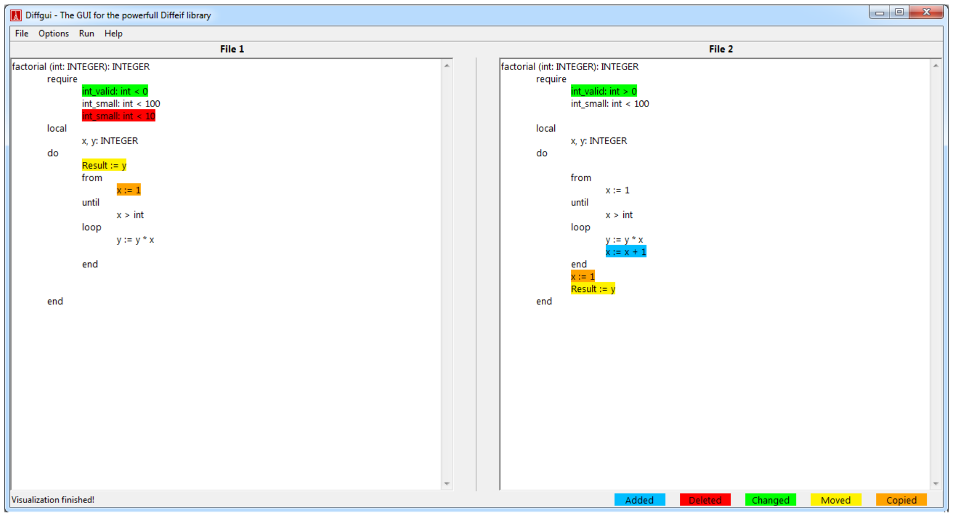
First, the user must create an object of the corresponding type of diff he want to use, either `LINE_DIFF`, `WORD_DIFF`, `CHAR_DIFF` or `TREE_DIFF`. The creation procedures allow to create an object with a source and destination string (for flat-based diff) or file set (*set_string* and *set_file*) or with additional options set right at the creation time of the object (*set_file_option* and *set_string_option*). These features can also be used to set new files on an existing object. On the created object, the user can set different options, some of them are listed in Table 5.1.

Once the options are set, the diff algorithm can be called with the *diff* feature. The calculated difference is stored in the attribute *difference:* `LINKED_LIST [EDIT_OPERATION]`, and the attribute *difference_set:* `BOOLEAN` is set to true. If both files or strings are equal, the attribute *equality:*`BOOLEAN` is set to true. **Note that it is possible for a change operation to cover unequal block sizes, i.e. a block of n elements in the source can be changed to m elements in the destination, where $n \neq m$.**

For using the GUI, the *Diffgui* library has to be included in the project. First, an instance of the class `DIFFGUI` must be created. This can be done in three ways: by creating the instance with default settings (*make*), by setting a source file and a destination file (*set_file*), or by setting a source string and a destination string (*set_string*). On this newly created object it is possible to set different options like in the Diffeif library (see Table 5.1). Afterwards, there are two possibilities to launch the GUI: by calling *launch* on the `DIFFGUI` object, or by calling *launch_and_calculate* on the `DIFFGUI` object, which not only launches the GUI

[1]Read the comments of the different features in the *Diffeif* library.

but also calculates and visualizes the initial difference of the files or strings which are set previously. A screenshot of the GUI is shown in Figure 5.1.[2]



```
factorial (int: INTEGER): INTEGER
        require
                int_valid: int < 0
                int_small: int < 100
                int_small: int < 10
        local
                x, y: INTEGER
        do
                Result := y
                from
                        x := 1
                until
                        x > int
                loop
                        y := y * x

                end


        end
```

```
factorial (int: INTEGER): INTEGER
        require
                int_valid: int > 0
                int_small: int < 100

        local
                x, y: INTEGER
        do

                from
                        x := 1
                until
                        x > int
                loop
                        y := y * x
                        x := x + 1
                end
                x := 1
                Result := y
        end
```

Figure 5.1: A screenshot of the GUI.

[2]Note that blank lines are added to synchronize the source and destination.

Table 5.1: Features of a DIFFEIF object.

Feature	**Tree-diff only**	**Description**
enable_move (bool: BOOLEAN)		Enables/Disables move operations.
enable_copy (bool: BOOLEAN)		Enables/Disables copy operations.
enable_write_file (path: STRING)		Difference will be written to a file. File will be created at *path*.
disable_write_file		Difference will not be written to a file.
set_max_length (length: INTEGER)		Set the maximum length of the edit script (number of differences). If the number of differences is larger than *max_length*, the difference will not be calculated.
reset_max_length		Resets the maximum allowed length of an edit script to the default value -1 (edit scripts of arbitrary size are computed).
enable_ignore_whitespace (bool: BOOLEAN)		Enables/Disables ignoring of leading and trailing whitespace of text for diff computation.
transform_xml_to_code (directory_path: STRING)	x	Transforms a bunch of XML documents to code (.e files). All subfolders and files in *directory_path* are traversed recursively.
set_add_cost (cost: INTEGER)	x	Set the assigned cost for an add operation.
set_change_cost (cost: INTEGER)	x	Set the assigned cost for a change operation.
set_delete_cost (cost: INTEGER)	x	Set the assigned cost for a delete operation.
enable_speedup (bool: BOOLEAN)	x	Enables/Disables speeding up the computation of the diff.
enable_extended_computation (bool: BOOLEAN)	x	Enables/Disables a more exhaustive computation, leading to the most accurate result at the expense of a higher computation time. The *speedup* option can also be used with the extended computation.

Chapter 6

Performance Analysis

In this chapter the runtime performance of the flat-based diff and the tree-based diff algorithm implemented in the *Diffeif* library is empirically studied. Miller & Myers' algorithm from Section 2.3 and Zhang & Shasha's algorithm from Section 3.2 are used in the *Diffeif* library. The analytical bound of the performance can be found in the corresponding sections. The experiments were performed on an Intel® Core™2 Duo 2.20 GHz PC with 4 GB memory. The operating system is Microsoft Windows® 7 x64.

Two data sets were used for the performance measurement. The first data set, used for testing the flat-based diff algorithm, consists of 5,566 plain text documents[1] containing code snippets representing a feature or a class.[2] From these text files, 222 pairs were built by combining different files, resulting in a varying number of differences (from 2 to 1,200). The number of lines of the text files varies between 0 and 1,200. On these pairs the line-based diff algorithm were run and the runtime was measured according to some criteria explained in Section 6.1. The second data set, used for testing the tree-based diff algorithm, contains 5,566 XML documents representing the abstract syntax tree of the code snippets of the first data set.[3] 48 pairs were built from these documents (with and without the speedup option enabled for tree-diff).[4] The number of nodes of the pairs of XML files varies between 42 and 800 and therefore the height of the trees and the number of leaves

[1]The comma is used as a delimiter for numbers, i.e. $5,566 = 5566$.

[2]Because the main purpose of the *Diffeif* library is to support the calculation of differences between codes snippets, files in the size of common code snippets are chosen as test files. Of course, the algorithm can also be applied to general text files. In this case the measured execution time for code files can be scaled up.

[3]The XML data set was provided by my supervisor Yu Pei.

[4]The amount of test data files for tree-based diff is smaller than for flat-based diff due to the shape of the data and the fact that running the tree-based diff algorithm consumes much more time.

in the trees is varying too. On these pairs of XML files the tree-based diff algorithm were run logging the execution time. Note that in the figures similar data points are merged into a single data point for better visibility and that there were more measurements gathered for small values due to the shape of the available test data and the complexity of the algorithms.

In Section 6.1 the performance of Miller & Myers' algorithm for flat-based diff is analysed, followed by an overview of the runtime behaviour of Zhang & Shasha's algorithm for tree-based diff in Section 6.2. Last but not least, in Section 6.3 the performance of the Miller & Myers' algorithm and Zhang & Shasha's algorithm are related to each other.

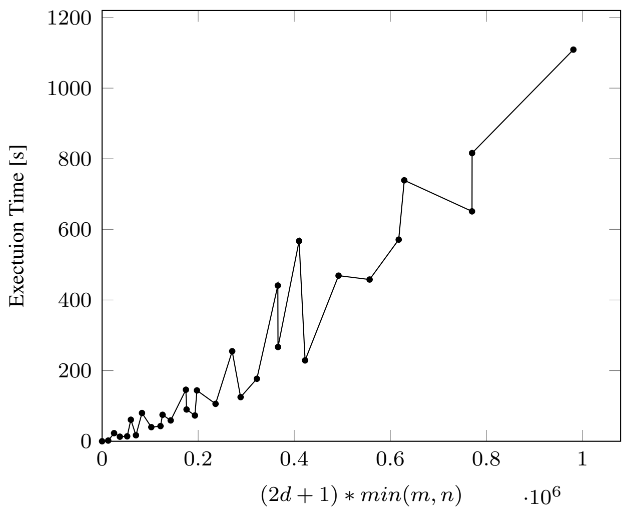
6.1 Miller & Myers' Algorithm (flat-based)

This section covers the performance analysis of the flat-based diff algorithm used in the *Diffeif* library. The detail of the algorithm can be looked up in Section 2.3. Remember that word-based diff and character-based diff use the same algorithm as line-based diff. Word-based diff and character-based diff provide only a finer grained partition of the input text (into words or characters) than line-based diff does. This implies that the execution time for character-based diff is slightly higher than for word-based diff, which itself has a somewhat higher runtime than line-based diff. In the following, only the performance of line-based diff is studied.

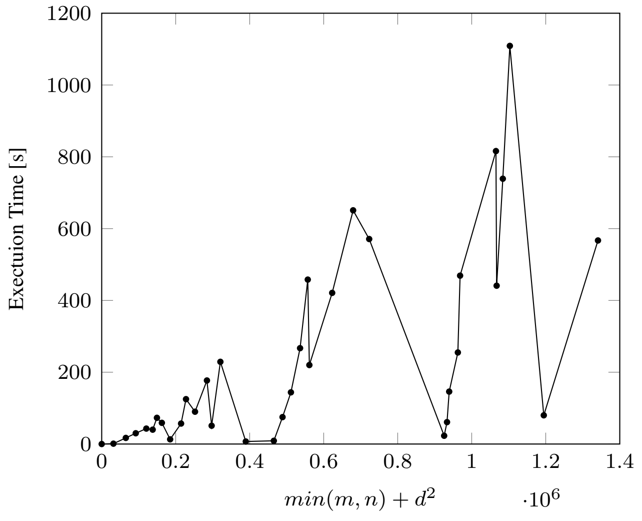
As seen in Section 2.3, the worst case runtime complexity is in $O((2d + 1)min(m, n))$ and the expected time complexity is in $O(min(m, n) + d^2)$, where m and n are the number of lines in the source and destination, respectively, and d is the number of differences between the two files. This implies that the algorithm performs well when d is small compared to m and n, i.e. it is preferable to use the algorithm on not totally different files. Figure 6.1 shows the relation of both complexities to the execution time. The worst case complexity ranges from 25 to 981,167 and the expected time complexity is in the interval from 9 to 1,341,141. The executions time lies between less than one second and 1,109 seconds.

In Figure 6.1(a) the worst case time complexity is plotted against the execution time. The execution time is not strictly increasing as might be reasonably expected. This is due to the preprocessing step of the algorithm, where prefixes and suffixes are eliminated. The shape of the differences has an influence too, i.e. if the differences span a consecutive block, the execution time needed might be smaller than for scattered differences. These two things, the preprocessing step and the shape of the differences, lead in some cases to a much smaller runtime as one might expect. In general, it can be said that the runtime increases if the worst case time complexity increases, i.e. if the size of the files and/or the number of differences grows. For example, for $O((2d + 1)min(m, n)) = O(12, 831)$ with $m = 39$, $n = 177$ and $d = 164$ the runtime is 2 seconds, whereas for the computation of two pairs of files with the worst case runtime complexity $O((2d + 1)min(m, n)) = O(422, 609)$ with $m = 373$, $n = 467$ and $d = 566$, 229 seconds are needed. On the other hand, if the number of differences for both examples is reduced to a small number, say $d \in [1, 20]$, the runtime is less than a second. As already mentioned above, the execution time needed is mainly limited by the number of differences. If we have two files, both containing 1,000 lines, and a huge difference between them, say $d = 1,000$, then $O((2d + 1)min(m, n)) = O((2 * 1,000 + 1) * 1,000) = O(2,001,000)$. For getting the same runtime complexity with a small number of differences, say $d = 30$, the files must contain at least 32,803 lines. Indeed, the expected time complexity $O(min(m, n) + d^2)$ reveals that the execution time often depends on the number of differences if the variance of the sizes of the files is not too big.

In Figure 6.1(b) the execution time in relation to the expected time complexity is plotted. Most of the time the execution time does not fit well the expected execution time, i.e. in most cases the algorithm is in fact faster. This is also due to the preprocessing steps and the shape of the differences.



(a) Execution time vs. worst case time complexity



(b) Execution time vs. expected time complexity

Figure 6.1: Performance of the flat-based algorithm.

6.2 Zhang & Shasha's Algorithm (tree-based)

The performance of the tree-based algorithm used in the *Diffeif* library is studied in this section. The algorithm can be looked up in Section 3.2. The worst case time complexity of the algorithm is in $O(|T_1| * |T_2| * min(height(T_1), leaves(T_1)) * min(height(T_2), leaves(T_2))) = Time\,Complexity$, where $|T|$ is the number of nodes in tree T, $leaves(T)$ is the number of leaf nodes in tree T and $heigth(T)$ is the height of tree T. As can be seen from this formula, the execution time mainly depends on the size and the shape of the trees but not directly on the number of differences as it is the case for flat-based diff. This makes the tree-based algorithm interesting for files with big differences.

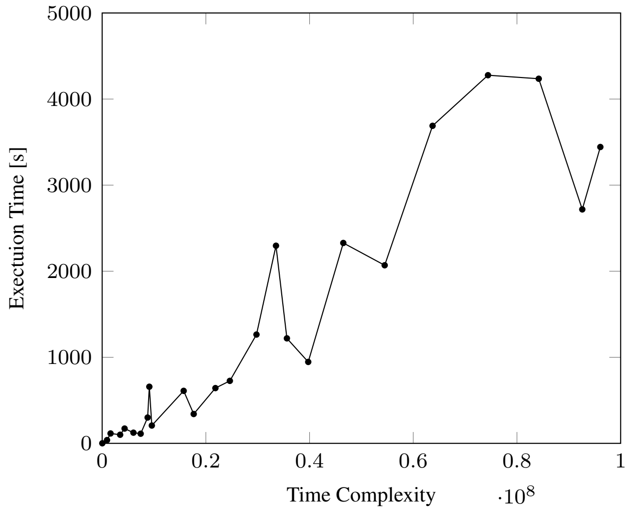
Figure 6.2 shows the execution time of the tree-based diff algorithm based on the time complexity and the sum of the number of nodes of the source and destination tree. The time complexity spans the interval from 26,730 to 96,083,456. The number of nodes ranges from 42 to 800. The execution time lies between 1 second and 4,277 seconds $\approx$ 71 minutes.

Figure 6.2(a) relates the execution time of the tree-based diff algorithm to the analytical time complexity. The execution time is not strictly increasing as might be expected. Outlier data points can arise from the shape of the tree, i.e. if the tree has a special shape, the number of *keyroots* can change, and thus the runtime is affected.[5] Furthermore, the number of differences can have an indirect influence (if two trees are very similar, they are equally shaped). In general, it can be assumed that the runtime of the algorithm increases when the time complexity grows. The experiments have shown that the runtime gets too big if the time complexity is greater than $0.4*10^8$ or if there are more than approximately 400 nodes in the trees. In this case the algorithm needs more than half an hour for computing the result. If the time complexity is bigger than 10^8 or if the amount of nodes in the trees exceed 1,000, the computation time even raises to more than one hour. Unfortunately, the XML test data are parsed into really big trees, making the tree-based diff algorithm very slow. Therefore, it is almost always a must to use the speedup option to shrink the size of the trees, their height and also the number of leaves, resulting in a much faster runtime (See next section). Of course, in some rare cases the speedup option can produce suboptimal results.

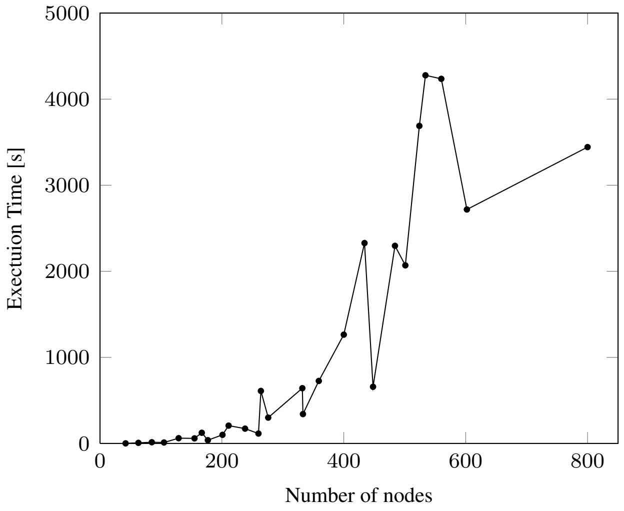
It is not surprising that the biggest influence on the runtime comes from the number of nodes in the trees. Figure 6.2(b) shows the execution time according to the sum of the number of nodes in the source tree and the destination tree. Again, the curve is not strictly increasing. This is due to the influence of the height and of the number of leaves of the trees (see the time complexity formula). As a general rule, it can be assumed that if the number of nodes is increasing, the execution time most likely increases for normal shaped trees too.[6]

[5]To relate the execution time of the algorithm to the shape of tree would be very difficult and is omitted here.

[6]It is not explained here what a normal shape of a tree is.



(a) Execution time vs. time complexity.



(b) Execution time vs. number of nodes

Figure 6.2: Performance of the tree-based algorithm.

6.3 Miller & Myers' Algorithm vs. Zhang & Shasha's Algorithm

In this section the tree-based diff algorithm and the flat-based diff algorithm used in the *Diffeif* library are compared on the same set of files, see Figure 6.3. 28 pairs of files were built out of the test data set. For line-based diff and tree-based diff the same pairs were used, i.e. for tree-based diff each text file was converted to a XML file representing the abstract syntax tree of the code contained in the text file, and thus the runtime of the two algorithms can be compared although the tree-based diff algorithm works on tree nodes and the line-based diff algorithm operates on lines. In Figure 6.3 these pairs of files are plotted against the execution time needed to compute the difference between the two files. The pairs are arranged in increasing order according to the value of $(2d+1)min(m,n)$, i.e. $Pair\ (i+1) > Pair\ (i) \iff (2d_{i+1}+1)min(m_{i+1},n_{i+1}) > (2d_i+1)min(m_i,n_i)$.[7]

In Figure 6.3 it can be seen that tree-based diff with the speedup option enabled performs much better than the tree-based diff without using the speedup option. In all cases the speeded up tree-based diff algorithm is at least 5 times faster. In some cases it is up to 11 times faster and on average it is 9 times faster. Moreover, the variance of the amount of speedup is small, i.e. the computation is most of the time accelerated by a factor of 9. In addition, the variance of the runtime is much lower for the tree-based diff with speedup than it is when the speedup option is not used. For all pairs the runtime of the line-based diff algorithm is less than 1.[8] Most of the time, the tree-based diff algorithm without speedup enabled can not compete with the line-based algorithm in terms of the runtime, whereas the tree-based diff algorithm with speedup enabled can keep up with the line-based algorithm in almost all cases.

Altogether, it has been shown that the flat-based diff algorithm is fast. Of course, tree-based diff sometimes provides better results but at the expense of a much higher runtime. Therefore, in most cases and especially if time matters it is better to use flat-based diff. If the tree-based diff algorithm is used, it is recommended to enable the speedup option, reducing the execution time noticeably at the expense of a small chance of getting not an optimal result. As a consequence, if the result has to be optimal regardless of the computation time, the tree-based diff algorithm without the speedup option enabled should be used.

The diff algorithms implemented in the *Diffeif* library were not compared with

[7]Also other sorting criteria can be used, e.g $m+n+d$ or $min(m,n)+d^2$. This would only change the shape of the curves, but the implications would be the same.

[8]It was only possible to use small text files resulting in a runtime for the line-based diff algorithm of less than 1 second because otherwise the obtained XML files would have been too big.

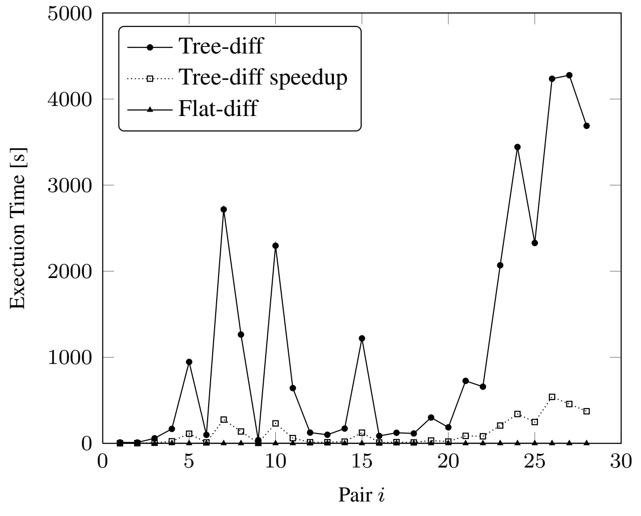


Figure 6.3: Performance of tree-based diff vs. flat-based diff.

algorithms from other diff tools because it is either not clear which diff algorithm is used by the tools or the implementation details of the algorithms are hidden. Moreover, it is in general very difficult to compare two different programs due to the fact that most of the time performance depends on the used test data and on the programming details.[13] Therefore, to compare the diff algorithms with other tools it would have been necessary to implement other algorithms in the same programming language and the same spirit as it was done in the *Diffeif* library. The gentle reader can try out other tools himself. There are lot of diff tools available for every possible platform. Some famous tools are, among others, the GNU *diff* tool, *WinMerge* and *Microsoft XML Diff and Patch Tool*.[9]

[9]A list of available diff tools can be found at `http://en.wikipedia.org/wiki/Comparison_of_file_comparison_tools`. Retrieved 2012-12-28. Most of them are line-based diff tools.

Chapter 7

Conclusion

The aim of this thesis was to investigate different non-hierarchical and hierarchical algorithms for computing the difference between two files or two strings, to select one hierarchical and one non-hierarchical algorithm for the implementation as a library and to visualize the differences in a GUI.

The non-hierarchical computation of a difference can be done line-based, word-based or character-based. For all three variants the underlying algorithm is the same, only the partitioning of the files or strings is different. A majority of flat-based tools offers only line-based diff, but in the *Diffeif* library all three variants are provided. Most diff algorithms are of non-hierarchical nature, so called flat-based diff algorithms. Many of these algorithms are based on *dynamic programming*, which was presented in detail at the beginning of this report. Moreover, different distance metrics were introduced as a prerequisite for the flat-based diff algorithms. Miller & Myers algorithm was chosen for the implementation. The experiments revealed that the algorithm performs well, especially for files with small differences. The algorithm performs even faster in some cases because prefixes, suffixes and singular insertions and deletions are detected before running the algorithm. This is due to the preprocessing step added to the algorithm in the *Diffeif* library. A postprocessing step, also added to the algorithm, even detects change, move and copy operations, which the original algorithm was not able to detect.

Tree-based diff algorithms are much more complex and in general slower than flat-based diff algorithms. There is still a lot of ongoing research in this field. The algorithms can be categorized into two different classes: algorithms operating on *ordered* trees and algorithms operating on *unordered* trees. For unordered trees, the computation of the difference is always NP-hard. Moreover, the computation of moves is also NP-hard. This is an important difference to flat-based diff algorithms. Algorithms operating on unordered trees or calculating move operations can only approximate the result, leading to non optimal edit scripts. As a prerequisite for many algorithms, the notion of a *mapping* was introduced in detail. An edit script can be derived from such a mapping. Algorithms of both categories, operating on ordered and unordered trees, were investigated in this report. One algorithm presented is also able to detect move operations. Furthermore, some of the algorithms build on dynamic programming. For the implementation in the *Diffeif* library Zhang & Shasha's algorithm was chosen. The algorithm was extended by a postprocessing step for detecting move and copy operations, which the original algorithms was not able to detect.

The experiments have shown that the tree-based diff algorithm is in general slow compared with the flat-based diff algorithm. Therefore, a speedup option for tree-based diff was introduced leading to an acceleration of the computation by a factor of 9 on average and in some cases even by a factor of 11. In addition, the flat-based diff algorithm showed a much better scalability[1] than the tree-based diff algorithm. Last but not least, it was shown that the performance of the flat-based diff algorithm mainly depends on the size of the files and the number of differences and for the tree-based diff algorithm the number of nodes in the trees is the most important term regarding the performance.

[1]Remember that scalability is the ability of the algorithm to handle a growing amount of data (lines, words, characters or nodes).

Chapter 8

Future Work

In this chapter some directions for future work are revealed. There exists still a wide field of interesting tasks. In the following, some of these tasks are listed.

- The diff algorithms implemented in the *Diffeif* library can be compared to other tools, for example the GNU *diff* tool, *WinMerge* or *Microsoft XML Diff and Patch Tool*, regarding the performance or other quantitative or qualitative measurements.

- A faster tree-based diff algorithm can be developed from scratch and added to the *Diffeif* library. For example, the optimality of the edit script can be sacrificed in favour of a better runtime performance. Furthermore, Zhang & Shasha[20] described a parallelized version of their algorithm reducing the execution time.

- The *Diffeif* library can be extended by a diff algorithm which uses semantic methods.

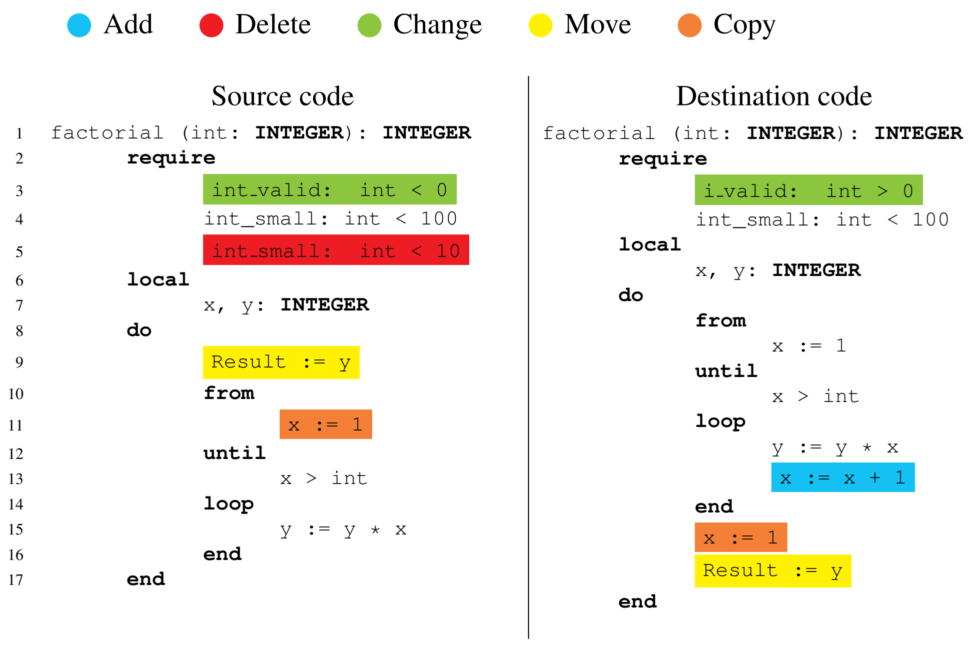
- The edit script can be overlayed onto the data using node annotations, i.e. every node knows by which edit operation it is affected if any. This corresponds to the notion of a delta tree.[2] It would lead to an easy processing (the trees just have to be traversed, reading out the edit operation at the nodes). On the other hand, the edit script would get very big because it is always of the size of the trees, leading to a higher space consumption. Usually the edit scripts are not that big, and thus it is questionable if it is an advantage to blow up the edit script by using node annotations.

- Other display formats of the edit script can be developed.

- Allowing to set or calculate the costs for each edit operation of the tree-based diff algorithm for each node separately, e.g. depending on the number of children of the node.

Appendix A

Line-Based vs. Word-Based vs. Character-Based Diff



```
factorial (int: INTEGER): INTEGER
        require
                int_valid:  int  < 0
                int_small: int < 100
                int_small:  int  < 10
        local
                x, y: INTEGER
        do
                Result  := y
                from
                        x  := 1
                until
                        x > int
                loop
                        y := y * x
                end
        end
```

```
factorial (int: INTEGER): INTEGER
        require
                i_valid:  int > 0
                int_small: int < 100
        local
                x, y: INTEGER
        do
                from
                        x := 1
                until
                        x > int
                loop
                        y := y * x
                        x := x + 1
                end
                x := 1
                Result  := y
        end
```

Figure A.1: Line-based diff.

Source code

```
factorial (int: INTEGER): INTEGER
    require
        int_valid: int < 0
        int_small: int < 100
        int_small: int < 10
    local
        x, y: INTEGER
    do
        Result := y
        from
            x := 1
        until
            x > int
        loop
            y := y * x
        end
    end
```

Destination code

```
factorial (int: INTEGER): INTEGER
    require
        int_valid: int > 0
        int_small: int < 100
    local
        x, y: INTEGER
    do
        from
            x := 1
        until
            x > int
        loop
            y := y * x
            x := x + 1
        end
        x := 1
        Result := y
    end
```

Figure A.2: Word-based diff.

Source code

```
factorial (int: INTEGER): INTEGER
    require
        int_valid: int < 0
        int_small: int < 100
        int_small: int < 10
    local
        x, y: INTEGER
    do
        Result := y
        from
            x := 1
        until
            x > int
        loop
            y := y * x
        end
    end
```

Destination code

```
factorial (int: INTEGER): INTEGER
    require
        int_valid: int > 0
        int_small: int < 100
    local
        x, y: INTEGER
    do
        from
            x := 1
        until
            x > int
        loop
            y := y * x
            x := x + 1
        end
        x := 1
        Result := y
    end
```

Figure A.3: Character-based diff.

Appendix B

Listings

B.1 Miller & Myers Algorithm

Listing B.1: Code for Miller & Myers Algorithm.

```
-- Algorithm is described in the paper of Webb Miller, Eugene, and W. Myers.
-- A file comparison program. Software: Practice and Experience,
-- 15:1025-1040, 1985.

m, n, lower, upper, d, k, row, col: INTEGER
last_d: ARRAY [INTEGER]
result_found: BOOLEAN
arr: ARRAY [LINKED_LIST [EDIT_OPERATION]]
edit_script: LINKED_LIST [EDIT_OPERATION]
edit_command: EDIT_OPERATION

result_found := False
create edit_script.make
create last_d.make_filled (0,0,0)
create arr.make_filled (edit_script,0,0)
m := source.count
n := destination.count
row := pref -- Identical prefixes, precomputed in preprocessing phase.
col := row
last_d.force (row,0)
  -- 0 entries in table D lie on the main diagonal (last_d[0]).
  -- They indicate identical prefixes.

if row = m then
  lower := 1
else
  lower := -1
end

if row = n then
  upper := -1
else
```

```
  upper := 1
end

if lower > upper then
    -- Files are identical.
  result_found := True
end

  -- Loop over all edit distances.
from
  d := 1
until
  d > max_length or
  result_found
loop
    -- Loop over all relevant diagonals.
  from
    k := lower
  until
    k > upper or
    result_found
  loop
    if (k = -d or (k /= d and last_d[k+1] >= last_d[k-1])) then
        -- Move down from diagonal k + 1.
      row := last_d[k+1] + 1
      edit_script := arr[k+1].twin
      create {DELETE} edit_command.make (row, row, row + k, row + k)
      edit_script.force (edit_command.twin)
      arr.force (edit_script.twin,k)
    else
        -- Move right from diagonal k - 1.
      row := last_d[k-1]
      edit_script := arr[k-1].twin
      create {ADD} edit_command.make (row, row, row + k, row + k)
      edit_script.force (edit_command.twin)
      arr.force (edit_script.twin,k)
    end
    col := row + k

      -- Move down on diagonal k.
    from
    until
      row >= m or
      col >= n or
      not source[row+1].is_equal (destination[col+1])
    loop
      row := row + 1
      col := col + 1
    end
    last_d.force (row,k)

    if row = m and col = n then
        -- Arrived at southeast corner (m,n). Result found.
      edit_script := arr[k]
      result_found := True
    end

    if row = m then
        -- Arrived at last row. Don't look to the left.
      lower := k + 2
    end

    if col = n then
        -- Arrived at last column. Don't look up.
      upper := k - 2
    end
    k := k + 2
  end
  lower := lower - 1
  upper := upper + 1
  d := d + 1
end
```

B.2 Zhang & Shasha's Algorithm

Listing B.2: Code for Zhang & Shasha's Algorithm.

```
-- Algorithm is described in the paper of K. Zhang and D. Shasha.
-- Simple fast algorithms for the editing distance between trees
-- and related problems. SIAM J. Comput., 18:1245-1262, 1989.

-- Compute lml_T1, lml_T2, keyroots_T1, keyroots_T2
-- lml: the leftmost leaf array holding for each node the corresponding
--  leftmost leaf.
-- keyroots: array holding the keyroots in increasing order.

from
  i' := 1
until
  i' > keyroots_T1.count
loop
  from
    j' := 1
  until
    j' > keyroots_T2.count
  loop
    i := keyroots_T1[i']
    j := keyroots_T2[j']
    forestdist(∅,∅) := 0

    from
      i1 := lml_T1[i]
    until
      i1 > i
    loop
      forestdist(T1[lml_T1[i]..i1],∅) = $\sum_{k=lml_T1[i]}^{i1}$ γ(T1[k] → λ)
      i1 := i1 + 1
    end

    from
      j1 := lml_T2[j]
    until
      j1 > j
    loop
      forestdist($\emptyset$,T2[lml_T2[j]..j1]) = $\sum_{k=lml_T2[j]}^{j1}$ $\gamma$($\lambda \rightarrow$ T2[k])
      j1 := j1 + 1
    end

    from
      i1 := lml_T1[i]
    until
      i1 > i
    loop
      from
        j1 := lml_T2[j]
      until
        j1 > j
      loop
        if lml_T1(i1) = lml_T1(i) and lml_T2(j1) = lml_T2(j) then
          forestdist(T1[lml_T1(i)..i1],T2[lml_T2(j)..j1]) := min(
            forestdist(T1[lml_T1(i)..i1-1],T2[lml_T2(j)..j1])
              + $\gamma$(T1[i1] $\rightarrow \lambda$),
            forestdist(T1[lml_T1(i)..i1],T2[lml_T2(j)..j1-1])
              + $\gamma$($\lambda \rightarrow$ T2[j1]),
            forestdist(T1[lml_T1(i)..i1-1],T2[lml_T2(j)..j1-1])
              + $\gamma$(T1[i1]$\rightarrow$ T2[j1]))

            -- Put in permanent tree distance array.
          treedist(i1,j1) := forestdist(T1[lml_T1(i)..i1],
                                  T2[lml_T2(j)..j1])
        else
          forestdist(T1[lml_T1(i)..i1],T2[lml_T2(j)..j1]) := min(
            forestdist(T1[lml_T1(i)..i1-1],T2[lml_T2(j)..j1])
              + $\gamma$(T1[i1] $\rightarrow \lambda$),
            forestdist(T1[lml_T1(i)..i1],T2[lml_T2(j)..j1-1])
              + $\gamma$($\lambda \rightarrow$ T2[j1]),
            forestdist(T1[lml_T1(i)..lml_T1(i1)-1],
                   T2[lml_T2(j)..lml_T2(j1)-1]) + treedist(i1,j1))
        end
        j1 := j1 + 1
      end
      i1 := i1 + 1
    end
    j' := j' + 1
  end
  i' := i' + 1
end
```

Appendix C

Internal Data Representation

Listing C.1: XML file representing the AST of the code in Listing 4.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<ast:eiffel xmlns:ast="http://se.inf.ethz.ch/east/xsd/core">
  <ast:feature_as>
    <ast:feature_names>
      <ast:eiffel_list_of_feature_name>
        <ast:feat_name_id_as>
          <ast:feature_name>
            <ast:id_as name="prune_first">prune_first</ast:id_as>
          </ast:feature_name>
        </ast:feat_name_id_as>
      </ast:eiffel_list_of_feature_name>
    </ast:feature_names>
    <ast:body>
      <ast:body_as>
        (
        <ast:arguments>
          <ast:type_dec_list_as>
            <ast:type_dec_as>
              n:
              <ast:type>
                <ast:class_type_as>
                  <ast:class_name>
                    <ast:id_as name="INTEGER">INTEGER</ast:id_as>
                  </ast:class_name>
                </ast:class_type_as>
              </ast:type>
            </ast:type_dec_as>
          </ast:type_dec_list_as>
        </ast:arguments>
        )
        <ast:content>
          <ast:routine_as>
            <ast:routine_body>
              <ast:do_as>
                do%N
                <ast:compound>
```

```
         <ast:eiffel_list_of_instruction_as>
          <ast:instr_call_as>
           <ast:call>
            <ast:access_id_as name="prune">
             prune (
             <ast:eiffel_list_of_expr_as>
              <ast:expr_call_as>
               <ast:call>
                <ast:access_id_as name="n">n
                </ast:access_id_as>
               </ast:call>
              </ast:expr_call_as>
              ,
               <ast:integer_constant>1
               </ast:integer_constant>
             </ast:eiffel_list_of_expr_as>
             )%N
            </ast:access_id_as>
           </ast:call>
          %N
          </ast:instr_call_as>
         </ast:eiffel_list_of_instruction_as>
        </ast:compound>
       </ast:do_as>
      </ast:routine_body>
      end%N
     </ast:routine_as>
    </ast:content>
   </ast:body_as>
  </ast:body>
 </ast:feature_as>
</ast:eiffel>
```

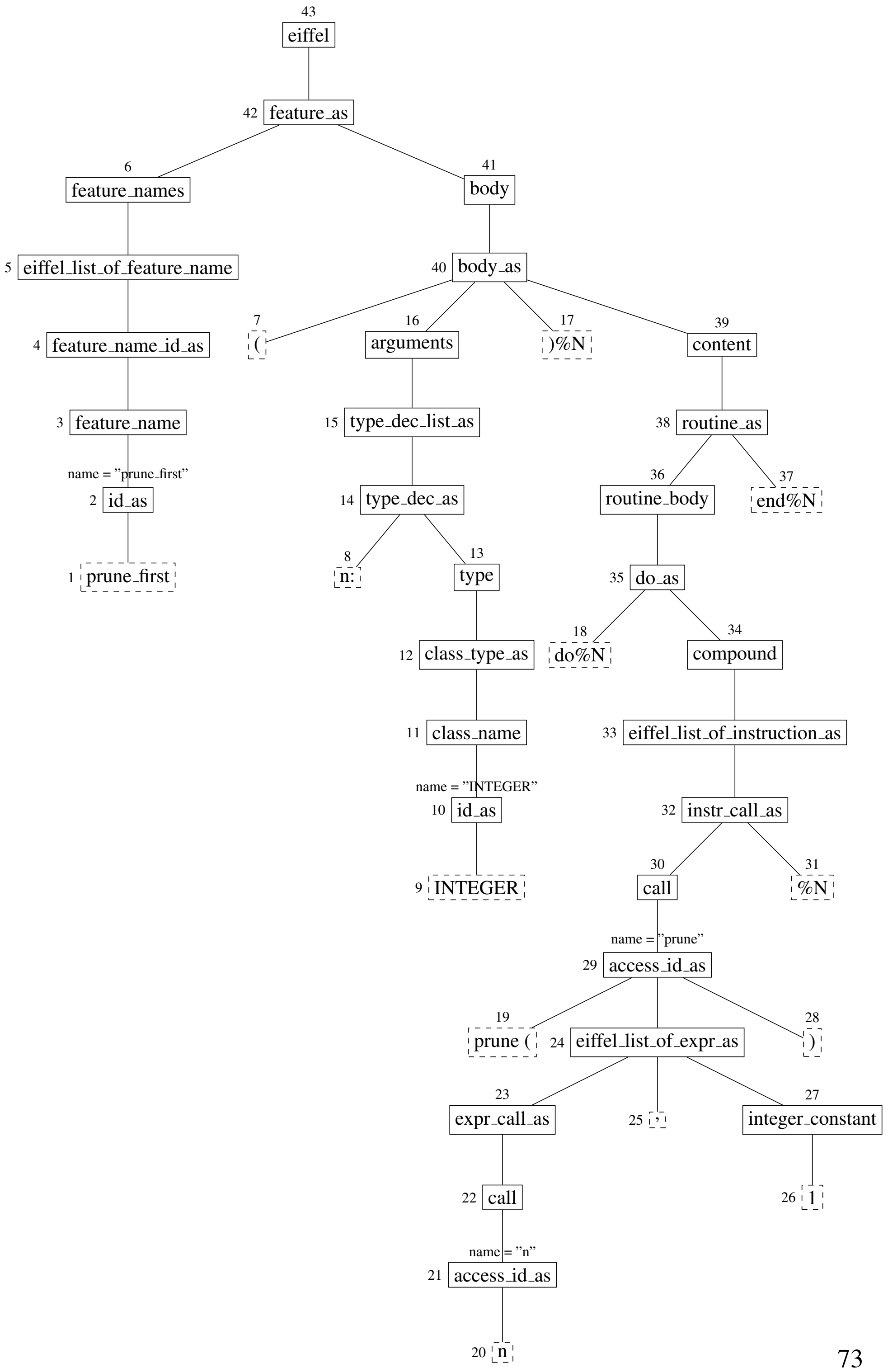


Figure C.1: A DOM-tree generated from the XML file in Listing C.1.

Bibliography

[1] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci*, 337:217–239, 2005.

[2] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504, 1996.

[3] Weimin Chen. New algorithm for ordered tree-to-tree correction problem. *J. Algorithms*, 40:135–158, 2001.

[4] T. Cranen. Comparison of questionnaires at the cbs. Master thesis, University of Maastricht, 2005.

[5] Inc. Cunningham & Cunningham. Diff algorithm, January 2012. `http://c2.com/cgi/wiki?DiffAlgorithm`, Last visited on 2012.10.18.

[6] Neil Fraser. Diff strategies, April 2006. `http://neil.fraser.name/writing/diff/`, Last visited on 2012.10.18.

[7] Gaston H. Gonnet and Ralf Scholl. *Scientific Computation*, chapter 7.6 and 7.7, pages 127–138. Cambridge University Press, 2009.

[8] Masatomo Hashimoto and Akira Mori. Diff/ts: A tool for fine-grained structural change analysis. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 279–288, 2008.

[9] Paul Heckel. A technique for isolating differences between files. *Commun. ACM*, 21:264–268, 1978.

[10] Daniel Hottinger and Franziska Meyer. Xml-diff-algorithmen. Semester thesis, ETH Zurich, 2005.

[11] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, July 1976.

[12] Philip N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th Annual European Symposium on Algorithms*, pages 91–102, 1998.

[13] Webb Miller, Eugene, and W. Myers. A file comparison program. *Software: Practice and Experience*, 15:1025–1040, 1985.

[14] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[15] Stanley M. Selkow. The tree-to-tree editing problem. *Inf. Process. Lett*, 6:184–186, 1977.

[16] Dennis Shasha and Kaizhong Zhang. Fast algorithms for the unit cost editing distance between trees. *J. Algorithms*, 11:581–621, 1990.

[17] Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, 26:422–433, 1979.

[18] Walter F. Tichy. The string-to-string correction problem with block moves. *ACM Trans. Comput. Syst.*, 2:309–321, 1984.

[19] Yuan Wang, David J. DeWitt, and Jin yi Cai. X-diff: An effective change detection algorithm for xml documents. In *ICDE*, pages 519–530, 2003.

[20] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18:1245–1262, 1989.

[21] Kaizhong Zhang. Algorithms for the constrained editing problem between ordered labeled trees and related problems. *Pattern Recognition*, 28:463–474, 1995.

[22] Kaizhong Zhang. A constrained edit distance between unordered labeled trees. *Algorithmica*, 15:205–222, 1996.

[23] Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Inf. Process. Lett.*, 42:133–139, 1992.