FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Arkadiusz Martin Antoniewicz

Tokenization-aware Diff and Patch

Department of Software Engineering

Supervisor of the bachelor thesis:	RNDr. Miroslav Kratochvíl
Study programme:	Computer Science
Study branch:	Programming and Software Development

Prague 2020

First and foremost, I would like to thank my supervisor Miroslav Kratochvíl for his patient guidance and professional advice. Next, I would like to express gratitude to Simona Zálešáková and my family for boundless support and faith in me. Last but not least, I am very grateful to Martina Matuláková and Jaromír Šimonek for their help with scientific English.

Title: Tokenization-aware Diff and Patch

Author: Arkadiusz Martin Antoniewicz

Department: Department of Software Engineering

Supervisor: RNDr. Miroslav Kratochvíl, Department of Software Engineering

Abstract: File comparison algorithms and utilities 'diff', 'patch' and 'diff3' are widely used in programming for the purpose of code comparison, and in many version control systems. Despite the usefulness, the differences and patches produced by the tools are strictly line-oriented, which complicates processing of differently formatted data, such as free flowing text, markup, and various other formats where line breaks are not crucial. This thesis describes and implements a customizable version of these tools, which allows the user to specify an arbitrary tokenization of the input, thus allowing easy diffing, patching and change-merging of content not supported by the traditional diff. Additionally, the thesis describes a newly appearing challenge of managing the whitespace in the patches, and demonstrates the functionality on a practical use-case that can not be performed with the current diff utilities.

Keywords: editing distance, three-way merge, text algorithms, version control

Contents

Introduction

Text comparison is an essential part of working with computers. Not only programmers but also many other professions use text comparison tools on a daily basis. There exist many different file and text comparison tools [2]. The text comparison tools include finding and showing differences, as well as revisiting, modifying and applying changes, comparing and showing differences in three files, merging three files together and many other cases of usage.

There is a problem with all common existing solutions. They do not allow a user to choose the unit of comparison, whether it may be a section, sentence, word, cells in a table or anything else. By considering a text with multiple changes in a long section we can present why common existing solutions fail to provide a convenient way of working with them. GNU Diff [9] (shown in Figure 1) simply shows that the lines are different but does not point to a place in the section where the difference is. Longer sections would make diff inapplicable. GNU Wdiff [5] (shown in Figure 2) demonstrates the difference in a more profound matter then the diff, however it lacks tools to patch and merge. Other solutions such as Beyond Compare [1] or the git diff [4] with word-diff=color option (shown in Figure 3) are capable of patching and showing the difference well. However, there is still a problem with merging. Neither of those tools would be able to three way merge if one file had changes at the beginning of a section and the other file had changes at the end of the same section.

The aim of this thesis is to design and implement tools that can work with various file formats, print readable differences and apply them. To be capable of working with many different formats, the user needs to be able to divide the text into sections of their own accord, which then they compare to each other. The process of text division is called tokenization and the results are called tokens.

The implemented utilities should be able to tokenize texts using rules defined by the user, work with the tokenized text effectively and show readable differences between them. This results in a multipurpose tool of comparing any text file format.

Notably, the custom tokenization creates a problem not present in other diff implementations. When a part of the text is left untokenized, it is considered

```
*** t1       2020-07-12 11:26:02.268930863 +0200
--- t2       2020-07-12 11:26:01.728660862 +0200
***************
*** 1,2 ****
! In mathematical theory, linguistics and computer science, the Levenshtein
distance is a string metric for measuring the difference between two sequences.
Informally, the Levenshtein distance between two words is the minimum number of
single-character edits (insertions, deletions or substitutions) required to
change one word into the other. It is named after the Russian mathematician
Vladimir Levenshtein, who considered this distance in 1965.
! Levenshtein distance may also be referred as edit distance, although that
term may also denote a larger family of distance metrics known collectively as
edit distance. It is not closely related to pairwise string alignments.
--- 1,2 ----
! In information theory, linguistics and computer science, the Levenshtein
distance is a string metric for measuring the difference between two sequences.
Informally, the Levenshtein distance between two words is the minimum number of
single-character edits (insertions, deletions or substitutions) required to
change one word into the other. It is named after the Soviet mathematician
Vladimir Levenshtein, who considered this distance in 1965.
! Levenshtein distance may also be referred to as edit distance, although that
term may also denote a larger family of distance metrics known collectively as
edit distance. It is closely related to pairwise string alignments.
```

Figure 1 GNU diff used on a text with multiple changes in the same section.

as a whitespace which is not significant for comparing. However, a whitespace around the tokens may sometimes carry information that is relevant for the result, and thus needs to be handled separately.

Layout of this Thesis

This thesis is structured as follows: the first chapter provides a detailed overview on handling text differences. There is described how to compare text, how to apply patches and how to compare and merge three files. The second chapter outlines the proposed solution for user-specified tokenization, how to implement it and what the lexers are in general. Whitespace handling is also addressed in the second chapter. The attention of the third chapter is focused on the program structure, tokenizer itself, patch format specification and the performance of the implemented tools.

```
In [-mathematical-] +information+ theory, linguistics and computer science, the
Levenshtein distance is a string metric for measuring the difference between
two sequences. Informally, the Levenshtein distance between two words is the
minimum number of single-character edits (insertions, deletions or
substitutions) required to change one word into the other. It is named after
the [-Russian-] +Soviet+ mathematician Vladimir Levenshtein, who considered
this distance in 1965.
Levenshtein distance may also be referred +to+ as edit distance, although that
term may also denote a larger family of distance metrics known collectively as
edit distance. It is [-not-] closely related to pairwise string alignments.
```

Figure 2 GNU wdiff used on a text with multiple changes in the same section.

```
@@ -1,2 +1,2 @@
In mathematicalinformation theory, linguistics and computer science, the
Levenshtein distance is a string metric for measuring the difference between
two sequences. Informally, the Levenshtein distance between two words is the
minimum number of single-character edits (insertions, deletions or
substitutions) required to change one word into the other. It is named after
the RussianSoviet mathematician Vladimir Levenshtein, who considered this
distance in 1965.
Levenshtein distance may also be referred to as edit distance, although that
term may also denote a larger family of distance metrics known collectively as
edit distance. It isnot closely related to pairwise string alignments.
```

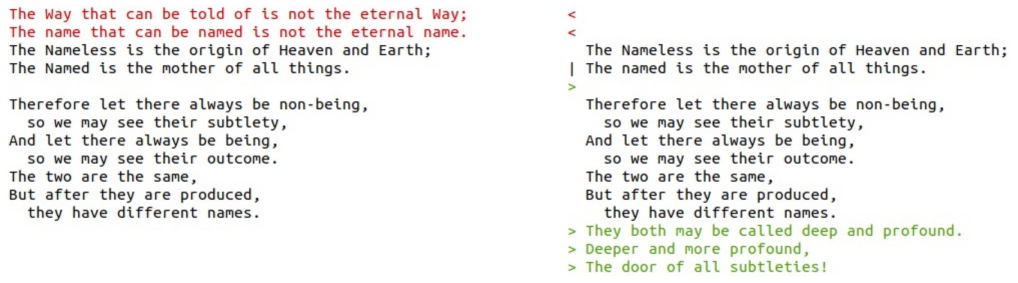
Figure 3 GIT diff with enabled word diff color option used on a text with multiple changes in the same section.

Chapter 1

Algorithms for comparing text

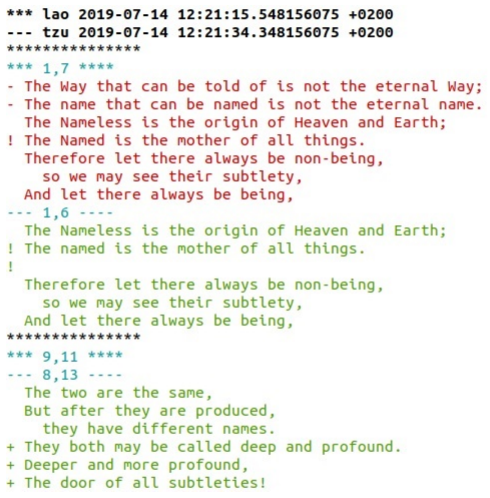
Utilities for comparing texts are used by programmers on a daily basis. Probably the biggest application of these utilities are version control systems. For the proper development of large projects, it is important to store all versions of previous projects as a form of communication among the programmers. Every additional version of a project is called a revision. Every revision, except for the first one, originates in the previous one. When the revision needs to be checked — what has changed — the differences between current revision and the one it originated from need to be shown.

The three main tools used in comparing text are diff, patch and merge. The diff serves as a data comparison tool that calculates and displays the differences between two files. The changes made in a standard format, so that both humans and machines can understand them, are displayed by the diff. An example of how colored side-by-side comparison of two files looks like can be seen in Figure 1.1. The patch utility takes a comparison output produced by the diff and applies the differences to a copy of the original file, producing a patched version. Diff3 is used when two people make changes to copies of the same file. It can produce a merged output that contains both sets of changes and warnings about conflicts.



```
The Way that can be told of is not the eternal Way;           <
The name that can be named is not the eternal name.           <
The Nameless is the origin of Heaven and Earth;                 The Nameless is the origin of Heaven and Earth;
The Named is the mother of all things.                        | The named is the mother of all things.
                                                              >
Therefore let there always be non-being,                        Therefore let there always be non-being,
  so we may see their subtlety,                                   so we may see their subtlety,
And let there always be being,                                  And let there always be being,
  so we may see their outcome.                                    so we may see their outcome.
The two are the same,                                           The two are the same,
But after they are produced,                                    But after they are produced,
  they have different names.                                      they have different names.
                                                              > They both may be called deep and profound.
                                                              > Deeper and more profound,
                                                              > The door of all subtleties!
```

Figure 1.1 The colored diff in a side-by-side format.



```
*** lao 2019-07-14 12:21:15.548156075 +0200
--- tzu 2019-07-14 12:21:34.348156075 +0200
***************
*** 1,7 ****
- The Way that can be told of is not the eternal Way;
- The name that can be named is not the eternal name.
  The Nameless is the origin of Heaven and Earth;
! The Named is the mother of all things.
  Therefore let there always be non-being,
    so we may see their subtlety,
  And let there always be being,
--- 1,6 ----
  The Nameless is the origin of Heaven and Earth;
! The named is the mother of all things.
!
  Therefore let there always be non-being,
    so we may see their subtlety,
  And let there always be being,
***************
*** 9,11 ****
--- 8,13 ----
  The two are the same,
  But after they are produced,
    they have different names.
+ They both may be called deep and profound.
+ Deeper and more profound,
+ The door of all subtleties!
```

Figure 1.2 The colored diff in a context format.

1.1 Diff implementation

The diff produces differences between two files. One way to achieve this is by computing the edit distance (as seen in Section 1.1.2) using the Wagner–Fischer algorithm (as seen in Section 1.1.3) and using the output of the Wagner-Fischer to find a sequence of insertions, substitutions and deletions to get from one text to another (as seen in Section 1.1.4).

1.1.1 Tokenization

Before we describe algorithms, we need to explain what tokenization and tokens are. Tokenization is a process of demarcating sections of a string of input characters. Tokenizers are usually designed to use a regular grammar (although it usually can not be achieved). An output is a list of tokens. Unlike parsing, which is usually a context-free grammar, the output is an abstract syntax tree. The parsing results in obtaining more information about the input and it is, certainly, more complex. We are going to analyze the tokenization more thoroughly in the Chapter 2.

Parts which are compared in the text are tokens. In edit distance, each character is a single token. In diff, tokenization is done by splitting text with delimiters being newlines. Each line is a single token. Tokens are comparable — it is possible to determine whether they are equal or not.

1.1.2 Edit distance

Let us consider these three editing operations:

- Changing one character to another single character
- Deleting one character from a given string
- Inserting a single character into the given string

Using only these three editing operations we get the most common metric for edit distance. It is called Levenshtein distance [8]. The term Levenshtein distance is often used interchangeably with the term edit distance.

Definition 1 (Notation). *Let A be a finite string (or sequence) of characters (or symbols). $A\langle i\rangle$ is the i-th character of string A. $A\langle i : j\rangle$ is the i-th through j-th characters (inclusive) of A. If $i > j$, $A\langle i : j\rangle = \Lambda$, the null string. $|A|$ denotes the length (number of characters) of string A. [12]*

So $A\langle i : j\rangle = A\langle i\rangle, A\langle i + 1\rangle \dots A\langle j\rangle)$

Definition 2 (String edit). *An edit operation is a pair $(a, b) \neq (\Lambda, \Lambda)$ of strings of length less than or equal to 1 and is usually written as $a \to b$ [12]*

Definition 3 (Editing operations). *String B results from the application of the operation $a \to b$ to string A, written $A \Rightarrow B$ via $a \to b$, if $A = \sigma a \tau$ and $B = \sigma b \tau$ for some strings σ and τ. We call $a \to b$ a* change or substitution *operation if $a \neq \Lambda$ and $b \neq \Lambda$; a* delete *operation if $b = \Lambda$; and an* insert *operation if $a = \Lambda$ [12].*

Let S be a sequence $s_l, s_2, \dots, s_m$ of edit operations (or edit sequence for short). An S-derivation from A to B is a sequence of strings $A_0, A_1, \dots, A_m$ where $A = A_0, B = A_m$, and $A_{i-1} \Rightarrow A_i$ via s_i for $1 \leq i \leq m$. We say S takes A to B if there is some S-derivation from A to B [12].

Now let γ be an arbitrary cost function which is assigned to each edit operation $a \to b$ a nonnegative real number $\gamma(a \to b)$. Extend γ to a sequence of edit operations $S = s_l, s_2, \dots, s_m$ by letting $\gamma(S) = \sum_{i=1}^{m} \gamma(s_i)$. (If $m = 0$, we define $\gamma(S) = 0$) [12].

Definition 4 (Edit distance). *Let $\gamma(A, B)$ from the string A to the string B be the minimum cost of all sequences of edit operations which transform A into B. Formally, $\gamma(A, B) = min\{\gamma(S)|S$ is an edit sequence taking A to $B\}$ [12].*

1.1.3 Wagner and Fischer algorithm

The Wagner–Fischer algorithm [12] is an algorithm used for finding edit distance. There are two strings as an input (can be applied to any two lists of items that can be compared). The computing is based on the following observation. If we reserve a matrix to hold edit distances between all the prefixes of the first string and all the prefixes of the second one, then the values in the matrix can be computed by flood filling the matrix, and thus the distance between the two full strings can be determined as the last value computed. An example of such implementation can be observed in Algorithm 1.

Definition 5 (Notation)**.** *Let A and B be arrays of tokens and a and b be the tokens. Define $A(i) = A\langle 1 : i\rangle$, $B(j) = B\langle 1 : j\rangle$, and $D(i,j) = \gamma(A(i), B(j)), 0 \leq i \leq |A|, 0 \leq j \leq |B|$.*

$\gamma(a \to b)$ is 0 if a equals b otherwise is 1

$$\begin{aligned} D(i,j) = min\{&D(i-1,j-1) + \gamma(A\langle i\rangle \to B\langle j\rangle),\\ &D(i-1,j) + \gamma(A\langle i\rangle \to \Lambda),\\ &D(i,j-1) + \gamma(\Lambda \to B\langle j\rangle)\} \end{aligned}$$

for all $i, j, 1 \leq i \leq |A|, 1 \leq j \leq |B|$.

$$D(0,0) = 0; D(i,0) = \sum_{r=1}^{i} \gamma(A\langle r\rangle \to \Lambda); D(0,j) = \sum_{r=1}^{j} \gamma(\Lambda \to B\langle r\rangle)$$

1.1.4 Backtrack algorithm to find edit operations in matrix

We can apply edit distance matrix of substrings filled by the Wagner Fischer algorithm to find edit operations. An example of such implementation can be seen in Algorithm 1 as a backtrack function. The algorithm starts at the right bottom cell of the matrix (edit distance between the two full strings). It finds a path in which the last cell was taken to fill. The path is always nondecreasing and ambiguous. As an example solution for strings ’ac’ and ’b’ are deletion ’a’ and substitution ’b’ for ’c’. The second possible solution is substitution ’b’ for ’a’ and deletion ’c’. Both solutions are of length 2 and are correct.

Algorithm 1 Wagner Fischer algorithm to fill a matrix with edit distances of substrings. The backtrack algorithm to determine edit operations.

```
1: procedure WAGNER AND FISCHER($D$)
2:     $D[0,0] \leftarrow 0$
3:     for $i \leftarrow 1, |A|$ do
4:         $D[i,0] \leftarrow D[i-1,0] + \gamma(A\langle i\rangle \rightarrow \Lambda)$
5:     end for
6:     for $j \leftarrow 1, |B|$ do
7:         $D[0,j] \leftarrow D[0,j-1] + \gamma(\Lambda \rightarrow B\langle j\rangle)$
8:     end for
9:     for $i \leftarrow 1, |A|$ do
10:        for $j \leftarrow 1, |B|$ do
11:            $m_1 \leftarrow D[i-1,j-1] + \gamma(A\langle i\rangle \rightarrow B\langle j\rangle)$
12:            $m_2 \leftarrow D[i-1,j] + \gamma(A\langle i\rangle \rightarrow \Lambda)$
13:            $m_3 \leftarrow D[i,j-1] + \gamma(\Lambda \rightarrow B\langle j\rangle)$
14:            $D[i,j] \leftarrow min(m_1, m_2, m_3)$
15:        end for
16:    end for
17: end procedure
18: procedure BACKTRACK($D$)
19:    $i \leftarrow |A|$
20:    $j \leftarrow |B|$
21:    while $i \neq 0$ and $j \neq 0$ do
22:        if $D[i,j] = D[i-1,j] + \gamma(A\langle i\rangle \rightarrow \Lambda)$ then
23:            $i \leftarrow i-1$
24:            print($''Addition: '', A\langle i\rangle$)
25:        else if $D[i,j] = D[i,j-1] + \gamma(\Lambda \rightarrow B\langle j\rangle)$ then
26:            $j \leftarrow j-1$
27:            print($''Deletion: '', B\langle j\rangle$)
28:        else
29:            $i \leftarrow i-1$
30:            $j \leftarrow j-1$
31:            if $A\langle i\rangle \neq B\langle j\rangle$ then
32:                print($''Substitution: '', A\langle i\rangle, B\langle j\rangle$)
33:            end if
34:        end if
35:    end while
36: end procedure
```

1.2 Merging and applying changes

1.2.1 Patch

The output of the diff is not only for people but also for other programs as well. A patch is a program that takes an output of a diff and applies it. The utility that updates text files according to instructions is called a patch. The instructions are produced by the diff. This may seem to have no use. Comparing files and then applying changes to the first file, results in forming of the second file. The use of the patch is to be able to review the output of the diff, adjust or remove some of the changes and apply them afterwards.

The GNU Patch manual [9] describes the patch algorithm as follows:

> "The patch reads instructions and applies them to the file (as seen in Figure 1.3). As for context diffs, patch can detect when the line numbers mentioned in the patch are incorrect, and it attempts to find the correct place to apply each hunk of the patch. A hunk is a sequence of lines common to both files, interspersed with groups of differing lines. As a first guess, it takes the line number mentioned in the hunk, plus or minus any offset used in applying the previous hunk. If that is not the correct place, the patch makes a forward and a backward scan for a set of lines to match the context given in the hunk.
>
> At first, the patch looks for a place where all lines of the context match. If it cannot find such place, and it reads a context or a unified diff and the maximum fuzz factor is set to 1 or more, then the patch makes another scan, ignoring the first and the last line of the context. If that fails, and the maximum fuzz factor is set to 2 or more, it makes a scan again, ignoring the first two and the last two lines of context. It behaves similarly if the maximum fuzz factor is larger.
>
> If the patch cannot find a place to install a hunk of the patch, it writes the hunk out to a reject file. The line numbers on the hunks in the reject file may be different from those in the patch file: they show the approximate location where the patch thinks the failed hunks belong in the new file rather than in the old one.
>
> The patch usually produces correct results, even when it makes many guesses. However, the results are guaranteed only when the patch is applied to an exact copy of the file that the patch was generated from."

Algorithm 2 Patch pseudocode.

```
1:  $hunks \leftarrow parseContext()$
2:  $file \leftarrow readFile()$
3:  for all $h \leftarrow hunks$ do
4:      $p \leftarrow position(h)$
5:      if $h.match(file.atPosition(p))$ then
6:          $applyContext(h, file)$
7:      else if $h.match(neighborhood(file, p))$ then
8:          $applyContext(h, file)$
9:      else
10:         $saveRejectedHunk(h)$
11:     end if
12: end for
```

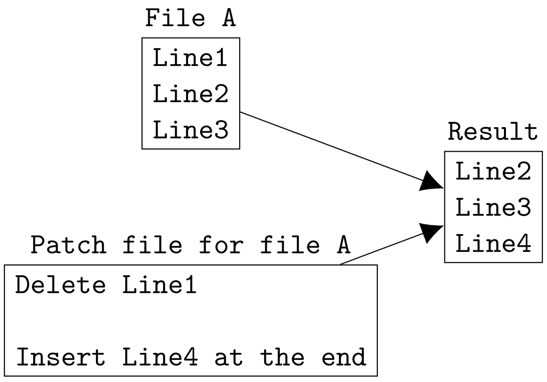


Figure 1.3 Simplified patching.

1.2.2 Merging patches

A more interesting phenomenon than patching is merging, which is a fundamental operation that reconciles multiple changes made to a version-controlled collection of files. Most often, it is necessary when a file is modified to two independent branches and is subsequently merged. The result is a single collection of files containing both sets of changes. In some cases, the merge can be performed automatically. Multiple changes made on the same place are called a conflict. Merging conflicts cannot be performed automatically, as a program does not know what the result should look like.

1.2.3 Three-way merge

One possible way of merging is using a three-way merge algorithm. Two files that merge and their common ancestor (base) are considered. All three files are compared and the result consists of:

Definition 6 (Notation)**.** *We assume given some set of* `atoms` $\mathcal{A}$*. (In practice, these might be lines of text, as in GNU diff3, or they could be words, characters, tokens etc.). We write* $\mathcal{A}^*$ *for the set of lists with elements drawn from* $\mathcal{A}$ *and use variables J, K, L, O, A, B and C to stand for elements of* $\mathcal{A}^*$*. If* L *is a list and* $k \in 1, \ldots, |L|$*, then* $L[k]$ *denotes the* k*-th element of* L*. A* `span` *in a list* L *is a pair of indices* $[i..j]$ *with* $1 \leq i, j \leq |L|$ *[7].*

Definition 7 (Configuration)**.** *A configuration is a triple* $(A, O, B) \in \mathcal{A}^* \times \mathcal{A}^* \times \mathcal{A}^*$*. We usually write configurations in the more suggestive notation* $(A \leftarrow O \rightarrow B)$ *to emphasize that* O *is the archive from which* A *and* B *have been derived [7].*

The first step of `diff3` is to call a two-way comparison subroutine on (O, A) and (O, B) to compute a non-crossing matching M_A between the indices of O and A–that is, a boolean function on pairs of indices from O and A such that if $M_A[i, j]$ = true then (a) $O[i] = A[j]$, (b) $M_A[i', j] = false$ and $M_A[i, j'] = false$ whenever $i' \neq i$ and $j' \neq j$, and (c) $M_A[i', j'] = false$ whenever either $i' < i$ and $j' > j$ or $i' > i$ and $j' < j$–and a non-crossing matching M_B between the indices of O and B. We treat this algorithm as a black box, simply assuming (a) that it is deterministic, and (b) that it always yields maximum matchings [7].

Definition 8 (Chunk)**.** *A chunk (from A, O, and B) is a triple* $H = ([a_i..a_j], [o_i..o_j], [b_i..b_j])$ *of a span in A, a span in O, and a span in B so that at least one of the three is non-empty. The size of a chunk is the sum of the lengths of all three spans. Write* $A[H]$ *for* $A[a_i..a_j] \in \mathcal{A}^*$*, and similarly* $O[H] = O[o_i..o_j]$ *and* $B[H] = B[b_i..b_j]$ *[7].*

Definition 9 (Stable chunk)**.** *A stable chunk is a chunk in which all three spans have the same length and corresponding indices are matched in all three — i.e., a hunk* $([a..a+k-1],[o..o+k-1],[b..b+k-1])$ *for some* $k>0$*, with* $M_A[o+i,a+i]=M_B[o+i,b+i]=true$ *for each* $0\leq i<k$*. That is, a table chunk corresponds to a span in* O *that is matched in both* M_A *and* M_B *[7].*

Definition 10 (Unstable chunk)**.** *An unstable chunk is one that is not stable. An unstable chunk H is classified as follows: [7]*

H *is changed in* A	*if* $O[H]=B[H]\neq A[H]$
H *is changed in* B	*if* $O[H]=A[H]\neq B[H]$
H *is falsely conflicting*	*if* $O[H]\neq A[H]=B[H]$
H *is conflicting*	*if* $O[H]\neq A[H]\neq B[H]$

An example with all types of chunks can be seen in Figure 1.4

Algorithm 3 Three-way merge algorithm [7].

```
1. Initialize $l_O = l_A = l_B = 0$. Find matching $M_A$ and $M_B$.

2. Find the least positive integer $i$ so that either $M_A[l_O+i, l_A+i] = false$ or
   $M_B[l_O+i, l_B+i] = false$. If $i$ does not exist, then skip to step 3 to output
   a final stable chunk.

   (a) If $i = 1$, then find the least integer $o > l_O$ such that there exist indices
       $a, b$ with $M_A[o,a] = M_B[o,b] = true$. If $o$ does not exist, then skip
       to step 3 to output a final unstable chunk. Otherwise, output the
       (unstable) chunk. $C = ([l_A+1..a-1], [l_O+1..o-1], [l_B+1..b-1])$

   (b) If $i > 1$, output the (stable) chunk $C = ([l_A+1..l_A+i-1], [l_O+1..l_O+
       i-1], [l_B+1..l_B+i-1])$, Set $l_O = l_O+i-1, l_A = l_A+i-1, l_B =
       l_B+i-1$, and repeat step 2.

3. If $(l_O < |O| or l_A < |A| or l_B < |B|)$, output a final chunk $C = ([l_A+
   1..|A|], [l_O+1..|O|], [l_B+1..|B|])$.
```

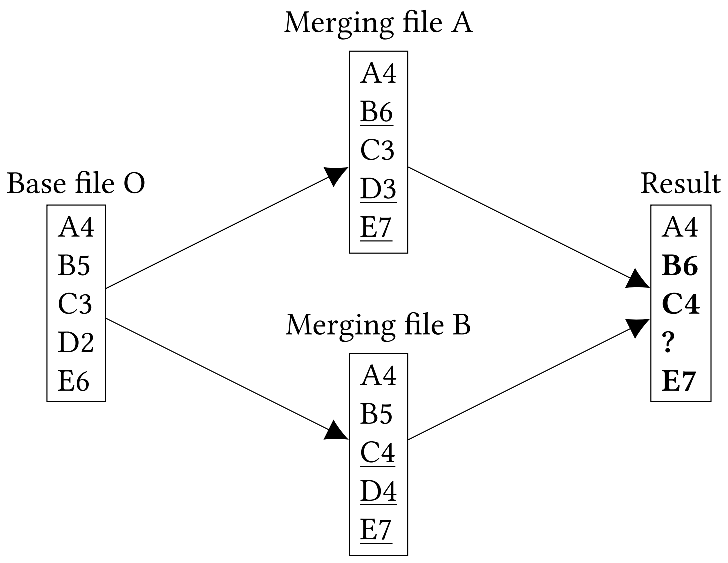


Figure 1.4 The three-way merge with the base file O and two files A and B. The result is composed of: A4, a stable chunk. B6 changed in A. C4 changed in B. Next, there is a truly conflicting chunk. E7 is a falsely conflicting chunk.

Chapter 2

Custom tokenization support

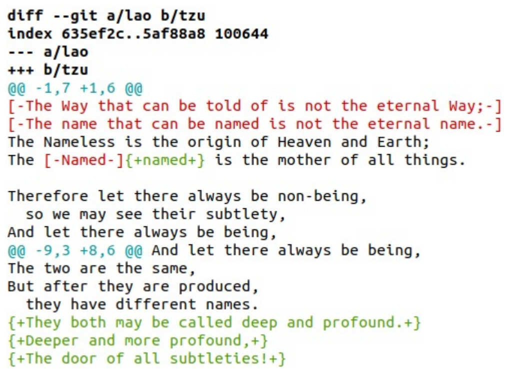
There are many different versions of diffs. They differ in the application and the way of displaying changes. Some of them are designed for finding differences in specific file formats. HTML diff tries to compare not only the source codes but also the appearance of the final webpage. XML diffs compare the hierarchical structure of XML documents. There is a word comparing option in the Gits implementation of the diff (as seen in Figure 2.1) but it lacks the patch. None of the tools mentioned above allow the user to specify the tokenization. The user specifiable tokenization has an advantage in its wide variety of applications. It can result in a universal diff that could then handle any programming language and text format.

There are many possible existing solutions for the user-specifiable implementation of the tokenization process: regular expressions with capture groups, lexical analyzer generators such as the Lex and the Flex. In this thesis we are going to design and implement our own solution.

Terms such as automaton, regular grammar, (non)deterministic finite state machine etc. are used in this section. Their definitions can be found in a book by Hopcroft, Motwani, and Ullman [6].

2.1 Lexing specification

Most tokenizers are designed to use a regular grammar. Tokenizers are sometimes referred to as lexers. Although they share very similar properties, the difference between them is that a lexer usually attaches an extra context to the tokens. We are going to consider regular expression [6] and try to simplify the defining of a more complex tokenization using a deterministic finite state machine with regular expression as edges. Let us call it Regex Edge Deterministic Finite Automaton (REDFA). The definition of REDFA is similar to the definition



```
diff --git a/lao b/tzu
index 635ef2c..5af88a8 100644
--- a/lao
+++ b/tzu
@@ -1,7 +1,6 @@
[-The Way that can be told of is not the eternal Way;-]
[-The name that can be named is not the eternal name.-]
The Nameless is the origin of Heaven and Earth;
The [-Named-]{+named+} is the mother of all things.

Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
@@ -9,3 +8,6 @@ And let there always be being,
The two are the same,
But after they are produced,
  they have different names.
{+They both may be called deep and profound.+}
{+Deeper and more profound,+}
{+The door of all subtleties!+}
```

Figure 2.1 The git diff with enabled word option.

of the deterministic finite automaton but with an elaborate transition function. An example of REDFA can be seen in Figure 2.2.

Definition 11. *Definition of Regex Edge Deterministic Finite Automaton (REDFA) M is a 5-tuple, $(Q, \Sigma, R, \gamma, q_0, F)$, consisting of:*

- *a finite set of states Q*
- *a finite set of input symbols called the alphabet Σ*
- *a finite set of regular expressions, search patterns over alphabet R*
- *a finite set of tuples $(q_1, r, 1_2)$ γ, where $q_1, q_2 \in Q$ and $r \in R$*
- *an initial state $q_0 \in Q$*
- *a set of accept states $F \subseteq Q$*

Let $w, a_1, a_2, \ldots, a_n$ be strings over the alphabet Σ. $w = a_1 a_2 \ldots a_n$. The automaton M accepts the string w if a sequence of states, $s_0, s_1, \ldots, s_n$, exists in Q with the following conditions:

1. *$r_0 = q_0$*
2. *$r \in R : r$ accepts (is matching) $a_i \wedge (a_i, r, a_{i+1}) \in \gamma$*
3. *$r_n \in F$*

Theorem 1. *Any language accepted by REDFA is a regular language.*

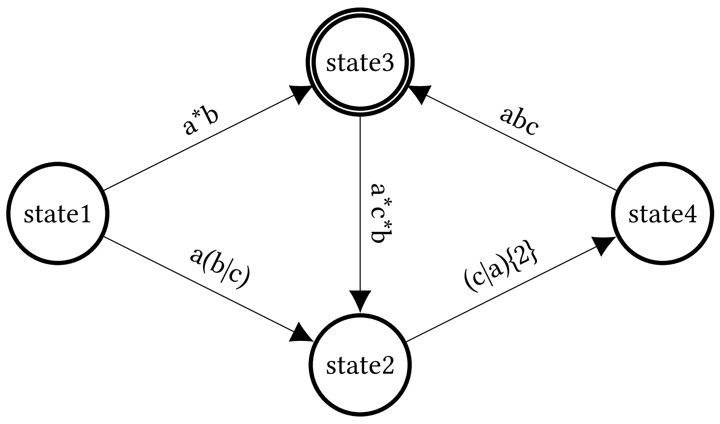


Figure 2.2 The automaton where edges are regular expressions.

Proof. At first, let us prove that a finite state machine with the edges as regular expression still fulfills the criteria for being a finite state machine.

We start with converting the regular expression to a NFA (nondeterministic finite automaton). This is called the Thompson algorithm [10]. The algorithm works recursively by splitting an expression into its constituent subexpressions, from which the NFA will be constructed using a set of rules. The constants and operations, which define a basis for the construction of the regular expression, are going to be used. The elementary constants are an empty expression ϵ and an expression with one symbol of alphabet. Operations are a union expression |, a concatenation expression and The Kleene star expression *. Firstly we convert elementary constants (as seen in Figure 2.3) and expand the constants with regular expression operations (as seen in Figure 2.4). Now we have an NFA instead of a regular expression. We can replace all regular expression edges in our REDFA with the NFA. Then we create an epsilon edge (empty expression ϵ) from the starting node of REDFA edge to the starting node of NFA created from the regular expression and also create epsilon edges starting in all the ending nodes of the created NFA to the ending node of the REDFA edge.

As for the next step, ordered edges are transformed to be a part of a nondeterministic finite-state machine, starting from the highest priority to the lowest priority edge for each node, unioning the complement with all edges with lower priority. The complement of automaton is done by reversing accepting and non accepting states. The complement of a regular expression a(b|c) is shown in Figure 2.6. $\square$

A regular finite state machine with ordered edges and edges as regular expressions (REDFA) is proven to accept regular language.

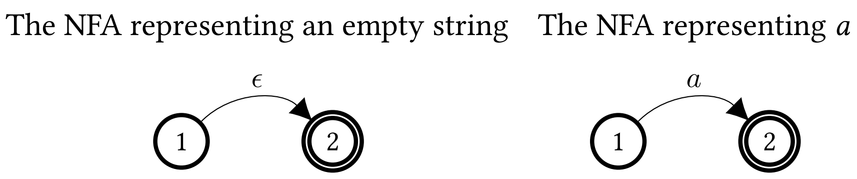


Figure 2.3 Elementary constants — empty and one character long.

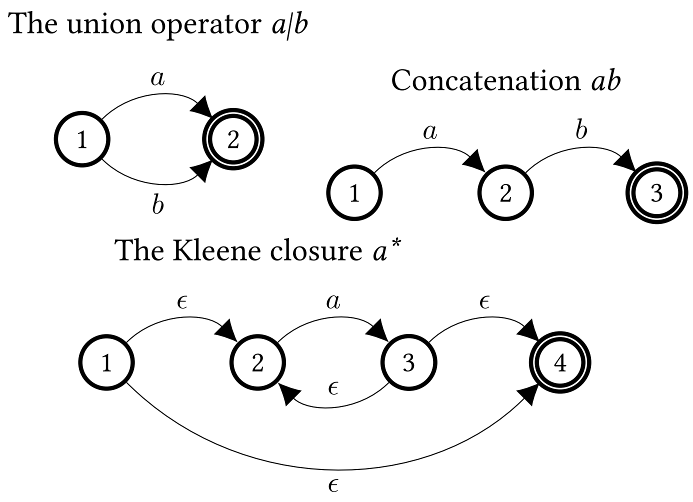


Figure 2.4 Converting a basic regular expression operator into NFA.

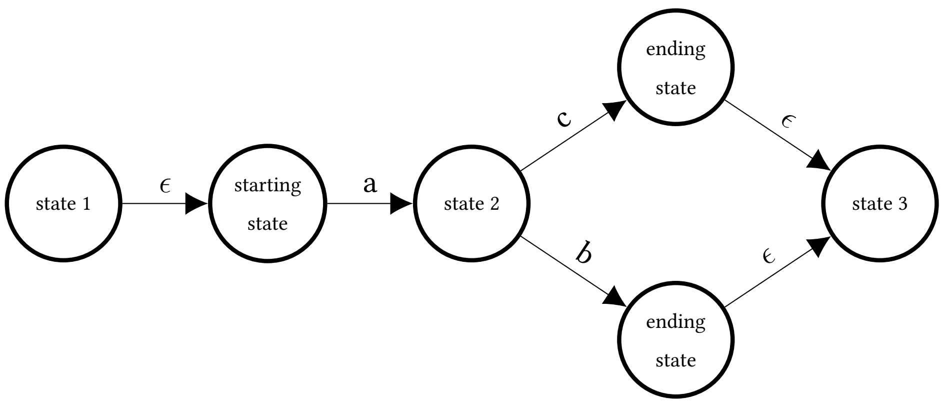


Figure 2.5 Inserting a regular expression *a(b|c)* into an automaton.

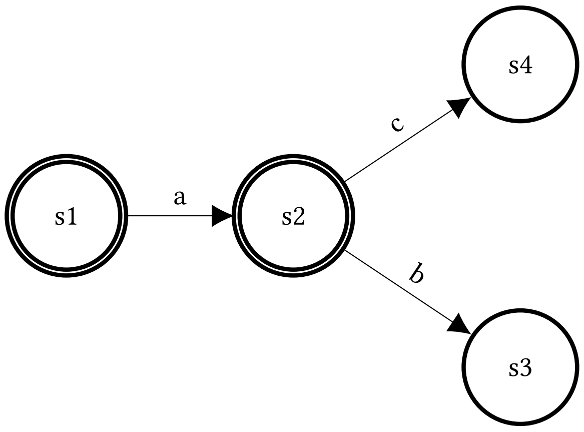


Figure 2.6 The complement of a regular expression a(b|c) converted into an NFA.

2.1.1 Specifying REDFA using strings

Let us show how REDFA can be used to make user-specifiable tokenization easier. The usage is similar to using a simple regular expression. Capture groups are used to define tokens and anything that is not in any capture group is considered a whitespace.

Before showing the differences of a simple regular expression and REDFA we need to decide how to define REDFA. We can easily define REDFA by specifying a set of 3-tuples γ, 3-tuples consisting of (starting node, regular expression, ending node). This is sufficient to define REDFA:

- Q — All nodes in rules
- Alphabet Σ — same as the alphabet regular expression use
- R — All regular expressions in rules
- γ — It is the same as rules
- q_0 — Starting node of the first rule
- F — All nodes ($F = Q$)

The example of inputs used as a simple regular expression and REDFA can be seen in Table 2.1. Simple regular expression is shorter but it becomes unreadable in more complicated rules. On the other hand, REDFA definitions are easily readable and extendable. It is possible to use REDFA as a simple regular expression — an automaton with only one node and an edge going into itself.

Automaton with regular expression as edges	Regular expression
Each word as token	
word,(\s*),whitespace whitespace,\S*,word	(\s*)\S*
Every other word in every other line	
tword,(\s*),word word,\S*\n,emptyline word,\S*,tword emptyline,.*\n,tword	(?:(\s*)\S*\s*\S*)*\n.*\n

Table 2.1 Comparison of the definitions of Regular expression and Regex Edge Deterministic Finite Automaton (REDFA).

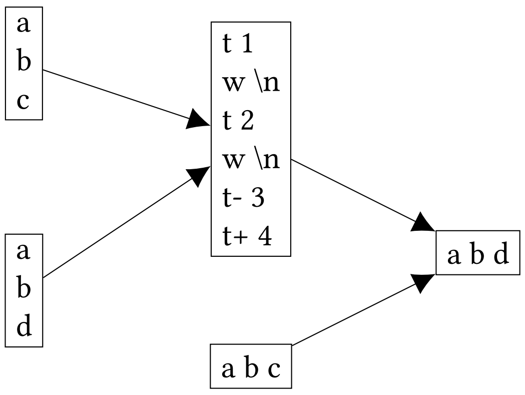


Figure 2.7 A simple example that the patching file with modified whitespaces is successful.

2.2 Whitespace handling

Whitespaces in this context are everything that is not compared in the text. The whitespaces in line-oriented diff are new lines, in word diff the whitespaces are the actual whitespaces (spaces, tabs, newlines etc.). In the user-defined tokenization the whitespaces are parts of the text that are between the tokens. In the line-oriented diff, whitespaces do not need to be handled because all the whitespaces are always the same. In the user-specified tokenization whitespaces can be anything, thus they need to be handled. A simple example why whitespaces needs to be handled can be seen in Figure 2.7.

Whitespace changes between tokens which are not changed nor shown within the context are not found by the diff and the patch. The whitespace from the source file is going to be used as shown in Figure 2.8 and the change is not

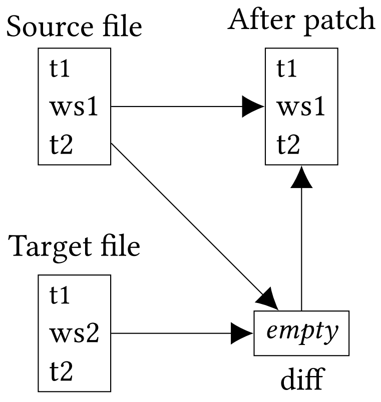


Figure 2.8 The change of a whitespace between non changing tokens. The diff is not able to find such change and the patch cannot patch it.

detected. Only whitespace changes that are around the token (in the context) changes are found. When inserting a token, the whitespaces around the token being inserted, are inserted as well (as seen in Figure 2.9). When deleting a token, the whitespaces around the token, that is being deleted are deleted too and the whitespace from the target file is inserted as shown in Figure 2.10. During tokenizing, this needs to be considered. It is advised not to leave crucial parts of the text as whitespace because the program is not able to determine which whitespaces are to be used or deleted.

When the whitespace change is shown in a context but it is not directly located next to a token change, it is considered the same way as whitespace changes on a different place where they are not a part of the patch file. Therefore, the change is not applied. An example is shown in Figure 2.12.

2.2.1 Resolving whitespace conflicts

A conflict occurs when a whitespace in a patch file does not match the whitespace of the file that patch is applied to. When tokens in context match but whitespaces do not, the hunk can not be rejected as whitespaces are not significant. The whitespaces in patch files are used in the result. This example can be seen in Figure 2.11 — for deletion case when the diff is running, the source file has ws3 between t1 and t2 and ws4 between t2 and t3. So that the information about the change from ws1 to ws2 in not lost, the whitespaces are saved into a separate whitespace file, for insertion case when the diff is running, the source file has ws4 between t1 and t3, ws4 was changed to ws1 before the patch was running, so ws1 is saved into a whitespace file not to lose the information about having it. If the whitespaces do not change, the information about their deletion is saved

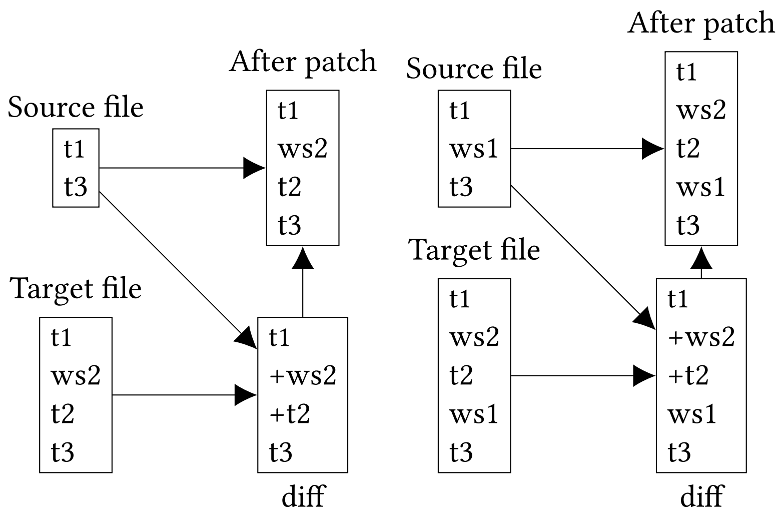


Figure 2.9 Insertion of token t2.

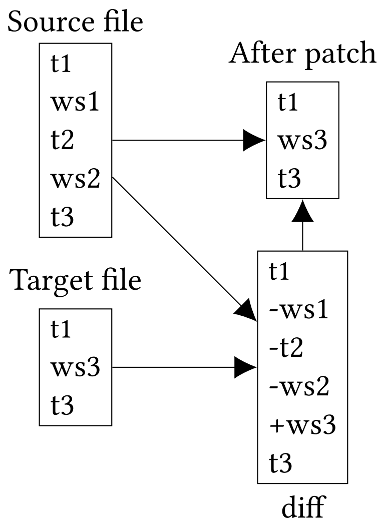


Figure 2.10 Deletion of token t2.

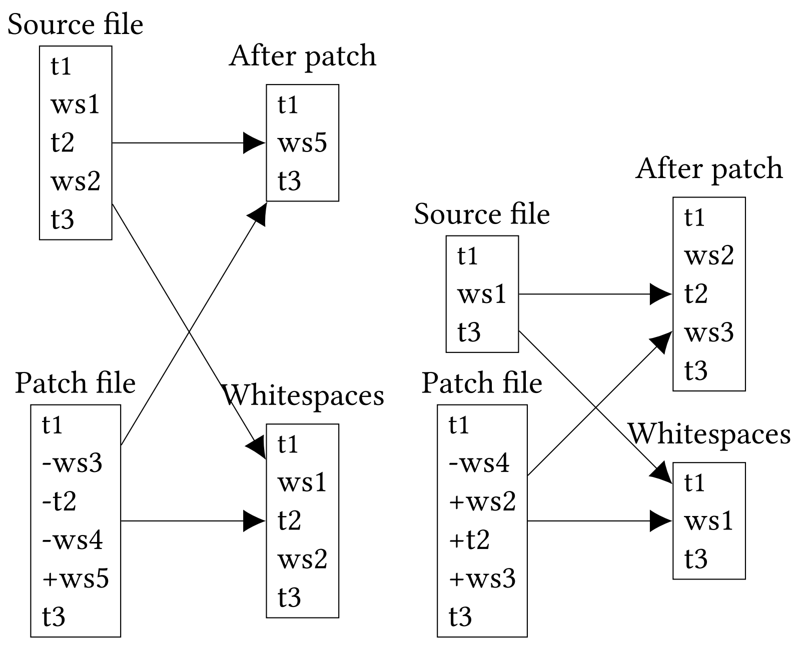


Figure 2.11 Deletion and insertion with different whitespaces in the patch file.

in the patch file, and thus there is no need for them to be stored again.

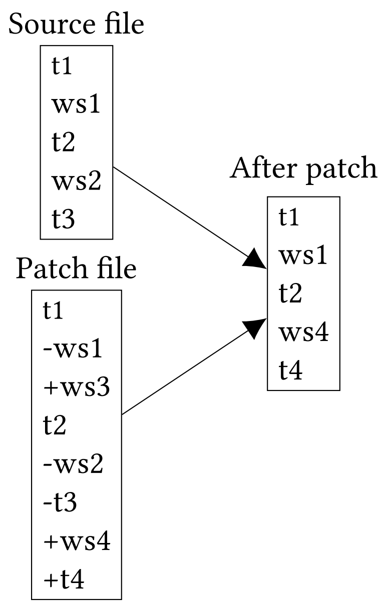


Figure 2.12 Different whitespaces in the context but not directly next to the token change. ws1 is not changed into ws3 despite the fact that the patch knows about the change.

Chapter 3

Implementation and results

Three utilities, TDiff, TPatch and TDiff3, have been implemented. They were programmed using the C++ programming language and the C++17 standard. In this chapter, the structure of the program and its implementation details are going to be introduced. After that we are going to show the results of a benchmark to see if it can handle larger files in reasonable time. At the end, we are going to show many different aplications where having a user specifiable tokenization in the diff is superior to almost all the other diff utilities.

3.1 Program structure

The program is divided into 3 standalone executables and one library which is used by all the projects. In every project there is a Main file and an InputOutput file. The Main is used for parsing arguments and calling appropriate methods from the InputOutput file. In the InputOutput file there is a logic of the program and it is calling the shared library methods. The program structure can be seen in Figure 3.1.

3.1.1 Shared library

The shared library contains 3 header files and their implementation.

- Tokenizer

  The Tokenizer is used for parsing the text into tokens. There are classes used for the definition of automaton and its edges. It also contains the definition of the parsed text which consists of the tokens and the text that was tokenized. Methods for building the automaton from the rules and for tokenizing the text with the automaton are implemented as well.

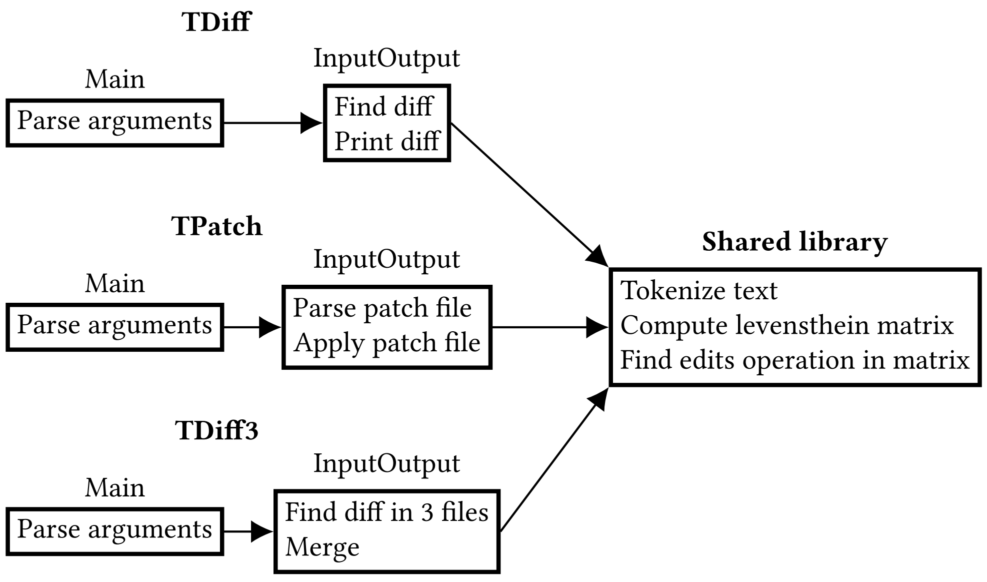


Figure 3.1 The program structure.

- Differentiate

  The Differentiate is used for finding the differences in two tokenized texts. It contains the definition of edit distance matrix, file differences and the difference. The file difference contains information about differentiating two files, paths to those files, their contents, the tokenized content and differences between them. One difference contains its type and indexes of tokens that are compared. There are methods for the Wagner and Fischer algorithm to create a matrix from the tokens and to backtrack the matrix to find the differences.

- Utils

  The Utils contains other methods and classes — a method that takes two paths to files and an automaton definition, creating an automaton, reading files, comparing files and returning them filled into the file difference class, a method for replacing whitespaces (tab, newline) for printable characters and vice versa, a method for splitting one argument which contains all the rules of automaton into separate rules and a method to group the diffs that are near each other within the context.

3.1.2 TDiff

TDiff is a program which compares two files or directories according to the given rules and prints the differences to a standard output. The Main parses arguments and calls the methods from InputOutput file. The InputOutput contains a method for comparing folders and files and for printing differences.

The format for executing the diff is `tdiff from to [options]`. From and to must be paths to either files or directories.

Below, there is a summary of all the options that the Token-aware diff accepts. Standard long and short arguments can be passed to the diff. The Getopt [3] is used to parse the input arguments.

–automaton,-a AT	Use AT to build an automaton to tokenize the input. The rules are in a format starting state, regular expression, ending state. The rules are separated by a semicolon. The semicolon in the rules needs to be escaped with a backslash.
–context,-C NUM	Output NUM (default 3) lines of context.
–debug, -d	Tokenize file and print the tokens.
–help, -h	Display help.
–file-automaton, -f PATH	Read the rules from PATH. The rules are delimited by newlines.

3.1.3 TPatch

TPatch is a program which applies patches on a file. The input is a patch file which contains information about which file is going to be patched and what changes are going to be applied. It applies the changes, rejected changes and whitespaces that were deleted are then saved to separate files. The Main parses arguments and calls methods from the InputOutput file. The InputOutput contains a method to parse the patch file and apply the changes.

The format for running the patch is `tpatch patch-file [options]`. The patch file must be in the format described further below.

–ignorews,-i	Ignore whitespaces during patching. Only the tokens are inserted and deleted.
–output, -o PATH	Write output to PATH instead of the path specified in the patch.
–help, -h	Display help.

3.1.4 TDiff3

TDiff3 is a program that reads 3 files and merges them together. The Main parses arguments and calls the method from the InputOutput file. The InputOutput contains a method of comparing three files, merging them and to printing the result.

The format for running the diff3 is `tdiff3 mine base yours [options]`. Mine, base and yours are the paths to three files.

–automaton,-a AT	Use AT to build an automaton to tokenize the input. The rules are in a format starting state, regular expression, ending state. The rules are separated by a semicolon. The emicolon in the rules needs to be escaped with a backslash.
–file-automaton, -f PATH	Read the rules from PATH. The rules are delimited by newlines.
–help, -h	Display help.

3.2 Data formats

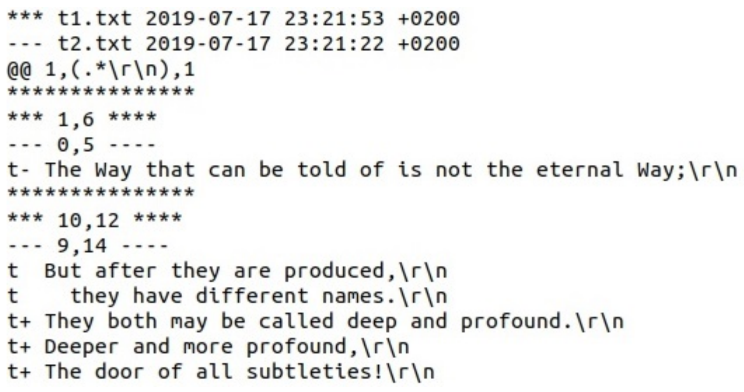
3.2.1 Tokenizer specification

To define the REDFA automaton we use something that has already been mentioned in Section 2.1.1. The automaton is going to be defined as an ordered set of rules (edges). Each rule has a starting state, regular expression and an ending state. To tokenize the input, capture groups are used. It is possible for a capture group not to end in one edge. The token starts in one edge and ends in another one. This is achieved by allowing the regular expression to have an incomplete group structure. As an example we consider rules `1,a(bc,2;2,d)aa,1` and input `abcdaa`. The tokenization is going to be successful with the application of these rules and the input will result in one token `bcd`.

3.2.2 Patch file format

The output of the diff can be seen in Figure 3.2. On the first two lines there is a path to the source and the target files. On the third line there is a definition of the automaton delimited by newlines. After that there are hunks. All hunks start with a line consiting of asterisks. It is followed by two lines with indexes of tokens of a source and a target used in the hunk. The hunk is similar to the hunk in GNU patch.

When a line should be deleted there is a minus at the beginning of the line. The same is applied with a plus and addition of a line. Space means that everything is on a place where it is supposed to be and it remains as it is. In the token-aware patch there are tokens and whitespaces instead of the lines but the notions are the same in both patches. To differentiate between the whitespaces and the tokens we use letters `'w'` and `'t'`. After `'w'` or `'t'` we put `'+'`, `'-'` or



```
*** t1.txt 2019-07-17 23:21:53 +0200
--- t2.txt 2019-07-17 23:21:22 +0200
@@ 1,(.*\r\n),1
***************
*** 1,6 ****
--- 0,5 ----
t- The Way that can be told of is not the eternal Way;\r\n
***************
*** 10,12 ****
--- 9,14 ----
t  But after they are produced,\r\n
t    they have different names.\r\n
t+ They both may be called deep and profound.\r\n
t+ Deeper and more profound,\r\n
t+ The door of all subtleties!\r\n
```

Figure 3.2 Example of a token-aware diff tokenized into lines.

’ ’ as a second character to determine the type of the operation in the patch. All these can be seen in Figure 3.2.

3.3 Performance and use cases

3.3.1 Tokenization performance

The benchmarks are done using Ubuntu 18.04.4 on the virtual machine with the host running Windows 10 1903. The tool for measurements is linux `time` [11] utility. Presented results are always mean time of ten runs with the slowest run being discarded. We are going to redirect the output to /dev/null as writing output can add overhead. The text will be generated lorem ipsum.

To measure the tokenization time we are going to use the diff debug option. With this option only tokenization is going to run. The results can be seen in Table 3.1. From the results we can conclude three things:

- For the same automaton, execution time depends linearly on the length of the text.
- For almost the same automaton (only capture group changed), the execution time depends on the number of tokens.
- When an incorrect automaton is used, it can heavily affect the performance.

To sum up, the tokenization runs reasonably quickly even on larger files, however, not optimal definition of REDFA (both regular expression itself as well as DFA) can slow down a great deal of the execution time.

File size	Automata definition	Mean time
200000 words~1.4MB	1,(\S*\s*),1	0.125s
2000000 words~13.5MB		1.274s
200000 words~1.4MB	1,(\S*\s*),2;	0.088s
2000000 words~13.5MB	2,\S*\s*,1	0.864s
200000 words~1.4MB	1,(\S+),1;	0.190s
2000000 words~13.5MB	1,\s*,1	1.901s

Table 3.1 The tokenization benchmark.

```
In [-mathematical-]
+information+ theory,
linguistics and computer
science, the Levenshtein
distance is a string
metric for measuring the
difference between two
sequences. Informally,
the Levenshtein distance
between two words is the
minimum number of
single-character edits
(insertions, deletions or
substitutions) required
to change one word into
the other. It is named
after the [-Russian-]
+Soviet+ mathematician
Vladimir Levenshtein, who
considered this distance
in 1965.
```

```
*** 1,4 ****
--- 1,4 ----
t  In
t- information
t+ mathematical
t  theory,
t  linguistics
***************
*** 51,55 ****
--- 51,55 ----
t  after
t  the
t- Soviet
t+ Russian
t  mathematician
t  Vladimir
```

Figure 3.3 GNU wdiff and tdiff.

3.3.2 Use case

First, let us compare the long section we have already mentioned in the intro. As we can see in Figure 3.3, both GNU diff and tdiff produce precise and readable output, however only tdiff is capable of running tpatch in this format.

Next, let us show a simple case where patching using the GNU utilities fails, but tdiff and tpatch are able to handle it. Firstly, let us consider two simple c files with small changes (as can be seen in Figure 3.4). Then we run diff between this two files to produce the patching file. The outputs of diffs can be seen in Figure 3.5. As we can observe, the tdiff output is more verbose. Let us see what happens when we change the formatting of the source file (the file we are applying the patch to). The changed file and the tpatch result can be seen in Figure 3.6. The GNU patch fails to apply anything to the changed file, on the other hand, the tpatch handles it correctly.

Another example we can offer is a three-way merge. Considering two files

```
#include <stdio.h>

int add(int a,int b)
{
  return a + b;
}

int main()
{
  int a = 5;
  int b = 4;

  printf("Hello, World!");
  printf("%d",add(a, b));
  return 0;
}
```

```
#include <stdio.h>

int add(int a,int b,int c)
{
  return a + b + c;
}

int main()
{
  int a = 5;
  int b = 4;
  int c = 6;

  printf("Hello, World!");
  printf("%d",add(a,b,c));
  return 0;
}
```

Figure 3.4 Two C codes with small changes.

already mentioned earlier in Figure 3.4, we add the third file to those 2 with changed `Hello world` to `Hi`. The GNU Diff3 fails to produce merged output of these 3 files. However, the tdiff3 is capable of doing a correct merge as can be seen in Figure 3.7.

```
***************
*** 9,10 ****
--- 9,13 ----
w
t  b
t+ ,
w+
t+ int
w+
t+ c
t  )
***************
*** 15,16 ****
--- 18,21 ----
w
t  b
w+
t+ +
w+
t+ c
t  ;
***************
*** 32,33 ****
--- 37,43 ----
t  ;
w+ \n
t+ int
w+
t+ c
w+
t+ =
w+
t+ 6
t+ ;
w  \n\n
t  printf
***************
*** 44,45 ****
--- 54,57 ----
w
t  b
t+ ,
w+
t+ c
t  )
```

```
@@ -2,5 +2,5 @@

-int add(int a, int b)
+int add(int a, int b, int c)
 {
-    return a + b;
+    return a + b + c;
 }
@@ -11,5 +11,6 @@
     int b = 4;
+    int c = 6;

     printf("Hello, World!");
-    printf("%d",add(5, 4));
+    printf("%d",add(5, 4, 6));
     return 0;
```

Figure 3.5 The token aware diff and the GNU diff output.

```
#include<stdio.h>

int add(int a, int b){
  return a+b;
}

int main(){
  int a=5;
  int b=4;
  printf("Hello, World!");
  printf("%d", add (a, b) );
  return 0;
}
```

```
#include<stdio.h>

int add(int a, int b, int c){
  return a+b + c;
}

int main(){
  int a=5;
  int b=4;
  int c = 6;
  printf("Hello, World!");
  printf("%d", add (a, b, c) );
  return 0;
}
```

Figure 3.6 tdiff is able to apply patches even to reformatted code. Left: Code from Figure 3.4 with changed coding style. Right: Token-aware patching is able to apply the patches from Figure 3.5 even in the reformatted code.

```
#include <stdio.h>

int add(int a,int b)
{
  return a + b;
}

int main()
{
  int a = 5;
  int b = 4;
  printf("HI!");
  printf("%d\n",add(a, b));
  return 0;
}
```

```
int add(int a,int b,int c)
{
  return a + b + c;
}

int main()
{
  int a = 5;
  int b = 4;
  int c = 6;
  printf("HI!");
  printf("%d\n",add(a,b,c));
  return 0;
}
```

Figure 3.7 Different changes in the original file from Figure 3.4 can be merged with the other patches using tdiff3. Left: The new modification. Right: Merged patches applied to the file.

Conclusion

In this thesis, we have designed the user-specifiable tokenization and implemented tools for differentiating text files using the user-specifiable tokenization.

In the Chapter 1 we have discussed the text difference handling. We have described the Wagner and Fischer algorithm for text comparing, algorithms for patching and merging three files.

In the Chapter 2 we have designed a solution for generic tokenization. We have created our own form of lexer. We also proposed a way of handling whitespace conflicts when working with generic tokenization.

In the Chapter 3 we have described the implementation of a program, the specification of defining the tokenizer and the tdiff output format to be able to consider whitespaces and be readable at the same time. We have also measured the performance of tdiff, verifying its sufficient fastness, and the performance scales as predicted by the asymptotic complexities of the used algorithms.

Bibliography

[1] *Beyond Compare.* Web page. 2020. URL: https://www.scootersoftware.com/.

[2] *Comparison of file comparison tools.* Web page. 2020. URL: https://en.wikipedia.org/wiki/Comparison_of_file_comparison_tools.

[3] *Getopt manual page.* Web page. 2017. URL: https://www.gnu.org/software/libc/manual/html_node/Getopt.html.

[4] *Git Diff.* Web page. 2020. URL: https://git-scm.com/docs/git-diff/.

[5] *GNU Wdiff.* Web page. 2020. URL: https://www.gnu.org/software/wdiff/.

[6] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321455363.

[7] Sanjeev Khanna, Keshav Kunal, and Benjamin Pierce. "A Formal Investigation of Diff3". In: Dec. 2007, pp. 485–496. DOI: 10.1007/978-3-540-77050-3_40.

[8] Vladimir I Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady.* Vol. 10. 8. 1966, pp. 707–710.

[9] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files with Gnu Diff and Patch.* Network Theory, 2003. ISBN: 9780954161750. URL: https://books.google.cz/books?id=oIINAAAACAAJ.

[10] Ken Thompson. "Programming Techniques: Regular Expression Search Algorithm". In: *Commun. ACM* 11.6 (June 1968), 419–422. ISSN: 0001-0782. DOI: 10.1145/363347.363387. URL: https://doi.org/10.1145/363347.363387.

[11] *time - Linux manual page.* Web page. 2020. URL: https://man7.org/linux/man-pages/man1/time.1.html.

[12] Robert A Wagner and Michael J Fischer. “The string-to-string correction problem”. In: *Journal of the ACM (JACM)* 21.1 (1974), pp. 168–173.

Appendix A

Using tdiff

To compile and run the software, you need:

1. C++ compiler with version at least 8.1 (filesystem)
2. POSIX mmap `#include <sys/mman.h>`
3. POSIX getopt `#include <getopt.h>`
4. External library RE2 (contained in debian package `libre2-dev`). On Debian-based Linux systems (such as Ubuntu), you may install this dependency with:

```
git clone https://code.googlesource.com/re2
cd re2
make
make test
make install
make testinstall
```

To unpack and compile the software, proceed as follows:

```
unzip tdiff.zip
cd tdiff/tdiff
make
```

The following example shows the usage of tdiff and tpatch with tokens being printable characters delimited by whitespace characters and automaton specified in a command line:

```
tdiff file1 file2 -a '1,(\S*)\s*,1' > diffoutput
tpatch diffoutput
```

An example below presents the usage of tdiff3 with automaton specified in the file automatondef is:

```
tdiff3 mine old your -f automatondef
```

With automatondef being the file with following content:

```
1,(\S*),2
2,(\s*),1
```

A different example displays automaton definition for simple c files (note that comments are not supported and for diff3 it is necessary to put the first rule into the capture group):

```
ws,[ \r\n\t]*,wend
wend,(#include [^ \r\n\t]*),ws
wend,([^ \r\n\t,;(){}+*=&%\-!|^<>~\[\]\/\\"]+),ws
wend,([,;(){}+*=&%\-!|^<>~\[\]\/\\]),ws
wend,("(?:[^"\\\n]|\\.|\\\n)*"),ws
```