Technology
Arts Sciences
TH Köln

Technische Hochschule Köln
Fakultät für Informations-, Medien- und Elektrotechnik (F07)

B A C H E L O R A R B E I T

Entwicklung einer Software zum Vergleich textbasierter Dateien mit graphischer Anzeige der Unterschiede und besonderem Fokus auf XML und JSON

Vorgelegt an der TH Köln
Campus Deutz
im Studiengang
Technische Informatik B.Sc.

ausgearbeitet von:
Allen Kletinitch
(Matrikelnummer: 11124870)

Erster Prüfer: Prof. Dr. Rainer Bartz
Zweiter Prüfer: Prof. Dr. René Wörzberger

Köln, im Juli 2021

Titel

Entwicklung einer Software zum Vergleich textbasierter Dateien mit graphischer Anzeige der Unterschiede und besonderem Fokus auf XML und JSON

Zusammenfassung

Wachsende Datenmengen sind ein stetiger Trend in der Informatik. Häufig können Anwendungsfälle gefunden werden bei denen es interessant ist die Unterschiede zwischen diesen Daten, bspw. bei der Versionierung, zu untersuchen. Im Rahmen dieser Bachelorarbeit wird eine Software erstellt, die beliebig viele textbasierte Dateien miteinander vergleicht, ihre Ähnlichkeit quantifiziert und die Möglichkeit bietet, Unterschiede für bis zu drei Dateien graphisch im Detail nachzuvollziehen. Weiterhin werden besondere Operationen für die gängigen Dateiformate XML und JSON implementiert um deren Inhalte sinnvoll vergleichbar zu machen. Zusätzlich werden für diese Dateiformate spezialisierte Vergleichsalgorithmen auf Basis ihrer Dokumentbäume entworfen und implementiert.

Stichwörter: Textvergleich, Diff, XML, JSON

Title

Development of a software to compare text-based files with graphical display of differences and special focus on XML and JSON

Abstract

Growing amounts of data are a constant trend in computer science. Frequently, use cases can be found where it is interesting to look at the differences between parts of this data, for example in the case of versioning. In the context of this bachelor thesis a software is created, which compares arbitrarily many text-based files with each other, quantifies their similarity and offers the possibility to trace differences graphically in detail for up to three files. Furthermore, special operations for the common file formats XML and JSON are implemented to make their contents meaningfully comparable. Additionally, specialized comparison algorithms are designed and implemented for these file formats based on their document trees.

Keywords: text comparison, diff, XML, JSON

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Diff	Difference
DOM	Document Object Model
EDT	Event Dispatch Thread
IO	Input/Output
JAR	Java Archive
JDK	Java Develepment Kit
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
L&F	Look-and-Feel
LCS	Longest Common Subsequence
LLCS	Length of a Longest Common Subsequence
UI	User Interface
UUID	Universally Unique Identifier
XML	Extensible Markup Language

1 Einleitung

Bereits seit der Industrialisierung sind Menschen bemüht ihre Aufgaben zu automatisieren. Diese Entwicklung hat auch in den letzten Jahren nicht stagniert. Besonders in der Informatik finden sich viele Themengebiete, die versuchen dem Menschen Arbeit durch Automatisierung abzunehmen. Sie existieren bspw. im Feld des Machine Learnings beim autonomen Fahren. Durch den Einsatz komplexer Algorithmen soll es der Person hinter dem Lenkrad ermöglicht werden ohne eigenen Einsatz an ihr Ziel zu kommen.

Auch im Cloud Computing findet Automatisierung Platz. Dort wird die Skalierbarkeit von cloudbasierten Software-Systemen immer seltener von Mitarbeitenden manuell eingestellt, um die Verfügbarkeit der Systeme zu versichern. Für solche Zwecke werden Orchestrierungstools wie Kubernetes eingesetzt, die die Systeme anhand von Konfigurationsdateien automatisch auf die verfügbare Hardware-Infrastruktur verteilen. Allerdings sind diese Dateien selbst nicht automatisch generiert, sondern müssen manuell erstellt werden.

Ein Beispiel für automatisch generierte Dateien findet sich ebenfalls beim Machine Learning im Bereich der Objekterkennung. Für ein System, welches Objekte auf Bildern erkennen soll, werden große Mengen an Bilddateien für das Training des Systems bereitgestellt. Beim Testen wertet das System anschließend Bilder aus auf denen es nicht trainiert wurde, und kann automatisch für jedes dieser Bilder eine Datei erstellen, die unter anderem die Klassifikation gefundener Objekte und deren Lokalisierung mit Objektmasken oder Objektbegrenzungen dokumentiert [1][2].

1.1 Motivation

Gerade bei dem Beispiel der Objekterkennung können riesige Dateimengen entstehen. Dabei ist es oft interessant zu betrachten, wie sich das System im Laufe der Entwicklung verändert und ob sich die Genauigkeit der Erkennung verbessert. Es kann nämlich sein, dass das System in früheren Durchläufen ähnliche, aber falsche Objekte erkennt und sich über den Trainingszeitraum dahingehend verbessert, dass die Objekte nun richtig erkannt werden.

Um diese Verbesserung nachzuvollziehen, müssten zunächst die Dateien gesucht werden, die mit dem gleichen Bild korrespondieren. Diese haben zum Beispiel einen ähnlichen Namen und liegen dann nach Durchlauf getrennt in verschiedenen Verzeichnissen. Nachdem diese Dateien gefunden wurden, müssen sie in einem Texteditor geöffnet und nach Unterschieden durchsucht werden. Für wenige Dateien ist das zwar möglich, aber selbst dann kann es schon Zeitaufwändig werden.

Diese Arbeit fokussiert sich deshalb auf die Weiterentwicklung einer Software, die dazu dient beliebig viele Dateien aus dem Dateibaum einzulesen, ihre Ähnlichkeit zu quantifizieren und mögliche Unterschiede graphisch darzustellen.

1.2 Ausgangssituation

Im Vorfeld dieser Arbeit ist bereits im Rahmen eines Projekts im Studiengang Technische Informatik an der Technischen Hochschule Köln eine erste Version dieser Software entstanden. Es handelt sich dabei um die Software MultiTextCompare; eine Desktop-Anwendung für Windows Betriebssysteme, die bereits grundlegende Implementierungen für einige der notwendigen Funktionen bereitstellt. Es ist möglich, Dateien des selben Dateityps nach Name oder Namensmuster, wie etwa im Windows-Explorer, automatisch einzulesen. Dafür muss lediglich ein Wurzelverzeichnis angegeben werden, unter dem alle Unterverzeichnisse nach passenden Dateien durchsucht werden. Im nächsten Schritt hat der Benutzer die Wahl, ob die ausgewählten Dateien nach bestimmten Parametern normiert werden sollen oder nicht. Aktuell werden die Dateiendungen txt, xml und json unterstützt und für jeden dieser Dateitypen stehen andere Parameteroptionen zur Verfügung. Für reguläre Textdokumente können Operationen wie das Entfernen von Leerzeichen, Leerzeilen, Satzzeichen oder Groß- und Kleinschreibung durchgeführt werden. Für die Formate XML und JSON gibt es zusätzliche Operationen auf Basis von Parsern, die u. a. die Sortierung der Dokumentbäume ermöglichen.

Nach dieser Normierung findet der eigentliche Vergleich statt. Dem Benutzer werden zwei unterschiedliche Vergleichsmodi angeboten. Zum einen können die Dateien Wort für Wort verglichen werden, zum anderen können auch die gesamten Zeichenmengen auf Ähnlichkeit überprüft werden. Auf die genaue Funktionsweise der beidem Modi wird später in der Arbeit noch im Detail eingegangen. Nachdem alle Dateien miteinander verglichen wurden, wird eine kolorierte Matrix mit allen Vergleichsergebnissen angezeigt (siehe Abb. 1). Ähnliche Dateien werden grün dargestellt, stark verschiedene rot. Bei der Ähnlichkeit handelt es sich des weiteren um einen prozentualen Wert, d. h. dass identische Dateien eine Ähnlichkeit von 1 haben, und der Minimalwert 0 ist.

Zuletzt kann der Benutzer dann auf die einzelnen Zellen der Matrix klicken, um sich die Unterschiede für zwei oder drei Dateien gleichzeitig farbig markieren zu lassen. Dieser Stand der Software wird fortan als Version 1.0 referenziert.

Abbildung 1: Ähnlichkeitsmatrix in Version 1.0

1.3 Zielsetzung

Ziel dieser Arbeit ist es, die Software MultiTextCompare an den nötigen Stellen zu verbessern. Dazu zählen hauptsächlich die Benutzerfreundlichkeit, Laufzeit und die Genauigkeit der Vergleichsalgorithmen.

Dabei gliedert sich die Arbeit in 6 Kapitel: Die theoretischen Grundlagen, die Konzeption der Software, die Implementierung der Verbesserungen, deren Verifikation sowie die Zusammenfassung mit Ausblick.

Die theoretischen Grundlagen sollen zunächst in das Thema des Textvergleichs einleiten. Dafür wird der Begriff der Ähnlichkeit von Zeichenketten diskutiert und anhand von zwei verschiedenen Textvergleichsalgorithmen dargestellt. Darauf folgt eine Übersicht zu den Dateiformaten XML und JSON und eine Vorstellung der speziellen Aspekte in der Programmierung der Software. Dort geht es dann beispielsweise um die Funktionsweise des UI-Toolkits Java Swing und die Implementierung von Nebenläufigkeit in Java.

Das Kapitel zur Konzeption zieht anschließend einen Vergleich zu bereits bestehender Software, deren Vorteilen und ihren Limitationen. Danach werden kurz die besonderen Aspekte für die Entwicklung dargestellt. Darunter liegen bspw. die zu verwendenden Bibliotheken und Tools und die Limitationen der Systeme auf denen die Software lauffähig sein soll. Zuletzt gibt es einen kurzen Überblick über die bestehende Architektur der Software und der Codebase.

In Kapitel 4 geht es hauptsächlich um die durchzuführenden Aufgaben dieser Arbeit. Es wird also konkret die Vorgehensweise bei der Entwicklung erklärt. Dabei soll auch gezeigt werden, durch welche Probleme eine Änderung der Funktionalität notwendig ist,

und wie diese Probleme gelöst werden können. Für die Dateiformate XML und JSON sollen zudem eigene Vergleichsalgorithmen entworfen werden, die nun nicht mehr auf reiner Textbasis arbeiten, sondern sich explizit mit den Baumstrukturen der Dokumente auseinandersetzen, um dort einen spezialisierten Vergleich durchführen zu können.

Das fünfte Kapitel beschäftigt sich dann ausführlich mit der Verifikation der in Kapitel 4 durchgeführten Laufzeitoptimierungen und den neuen Vergleichsalgorithmen. Dabei werden die Testsysteme vorgestellt, die Methodik erklärt und die Ergebnisse kritisch evaluiert.

Im letzten Kapitel befindet sich abschließend eine Zusammenfassung der durchgeführten Entwicklungen und der gewonnenen Erkentnisse und ein Ausblick, wie die Software in Zukunft noch weiter verbessert werden könnte.

2 Grundlagen

Im folgenden Kapitel werden die theoretischen und technischen Grundlagen erklärt, die für das Verständnis der Funktionsweise von MultiTextCompare notwendig sind.

2.1 String similarity Algorithmen

Für eine Software, die dazu dient textbasierte Dateien miteinander zu vergleichen ist es essenziell eine Metrik zu finden, die die Ähnlichkeit von Zeichenketten (engl. Strings) bewerten kann. Allerdings ist der Begriff der Ähnlichkeit im allgemeinen Sprachgebrauch ein subjektives Konstrukt. Der Begriff der Gleichheit scheint hingegen auf den ersten Blick etwas greifbarer. Strings sind gleich, wenn ihre Länge, ihre Zeichen und die Reihenfolge dieser Zeichen gleich sind. In der Sprache existieren allerdings auch Wörter, die bei gleicher Schreibweise unterschiedliche Bedeutungen haben. Wörter sind hierbei im strukturellen Sinne auch nur durch Leerzeichen voneinander getrennte Strings mit Zeichen des zugehörigen Alphabets. Im Deutschen kann z.B. das Wort „einstellen“ je nach Kontext u.a. die Bedeutung „mit einer Tätigkeit [...] aufhören“, „etwas regulieren“ und „(jemanden) in ein Arbeitsverhältnis nehmen “ haben [3], obwohl sich die Schreibweise nicht ändert. Andersherum gibt es Wörter, die anders geschrieben werden und dennoch die gleiche Bedeutung haben. Je nach Kontext können z.B. die Wörter Software, Anwendung und System auf den gleichen Begriff angewandt werden.

Diese Inkonsistenzen sind allerdings schwer messbar und treten zudem bei automatisiert generierten Dateien selten auf, weshalb bei MultiTextCompare auf einen rein strukturellen Vergleich von Strings gesetzt wird. Im Folgenden werden zwei verbreitete Algorithmen für den Vergleich von Strings vorgestellt.

2.1.1 Longest common subsequence

Die Longest Common Subsequence (LCS) für zwei Strings ist die größte Menge aller Zeichen, die in beiden Strings in gleicher Reihenfolge vorkommen. Diese Menge wird durch Löschung einzelner Zeichen der beiden Strings errechnet, sodass beide Strings nun dieser Menge entsprechen [4]. Dadurch dass die Strings nicht gleich lang sein müssen, können Zeichen der Ergebnismenge nicht für beide Strings an den gleichen Indices stehen.

Das folgende Beispiel zeigt zwei unterschiedliche Strings S_1 und S_2 für die die LCS berechnet werden soll, wobei unterschiedliche Zeichen in rot markiert sind.

$$S_1 = Dies\ ist\ ein\ Text$$

$$S_2 = Das\ ist\ ein\ anderer\ Text$$

Die zugehörige Ergebnismenge M_{LCS} ist $\{Ds\ ist\ ein\ Text\}$. Der LCS Algorithmus findet Verwendung bei der Erstellung der Difference (Diff), einem Verfahren bei dem die Unterschiede zwischen zwei Dateien ermittelt werden. Die Versionskontroll-Software Git, ermittelt bspw. die Diff für verschiedene Versionen der gleichen Datei. Dafür wird grundsätzlich der LCS-Algorithmus verwendet [5].

Die Implementierung der Diff in der Software MultiTextCompare beruht für zwei Dateien ebenfalls auf dem gleichen Prinzip. Dort entsteht für zwei Dateien A und B genau ein Vergleichsdurchlauf $D_2 = \{A, B\}$, für den alle Zeilen von A und B mittels der LCS miteinander verglichen werden. Zeichen, die sich in der Ergebnismenge M_{LCS} befinden, werden im User Interface (UI) als unverändert (weiß) dargestellt, gelöschte und eingefügte Zeichen werden entsprechend rot und grün markiert. Für drei Dateien A, B, C wird weiterhin die LCS verwendet, allerdings entstehen dort 3 Vergleichsdurchläufe $D_3 = \{\{A, B\}, \{A, C\}, \{B, C\}\}$. Dieses Verhalten wird im Vergleich zu alternativer Software in Kapitel 3.1 genauer beleuchtet.

Eine weitere interessante Metrik ist die Länge der LCS (LLCS). Diese entspricht für das Beispiel oben $|M_{LCS}| = 15$ und gilt als sinnvolle Quantifizierung der Ähnlichkeit von zwei oder mehr Strings. Als Metrik ist sie auch verwandt zur *Edit Distance*, der minimalen Anzahl an Einfügungen, Löschungen und Änderungen von Zeichen um einen String in einen anderen zu überführen [6]. Im Gegensatz zur Edit Distance beschreibt die LLCS allerdings die Anzahl der *gemeinsamen* Zeichen. Die beiden Begriffe sind also zu einem gewissen Grad komplementär.

Auch die LLCS wird innerhalb von MultiTextCompare verwendet. Durch sie wird im Diff-Prozess festgestellt ob zwei Strings eine Mindestähnlichkeit erfüllen um zu entscheiden, ob gleiche Zeichen noch als gleich geblieben angezeigt werden sollen oder die zwei Strings eine komplette Zeilenlöschung bzw. Zeileneinfügung repräsentieren.

Damit ähnliche, aber verschobene Zeilen miteinander für die Diff verglichen werden können, existiert ein Matcher, der innerhalb der zu vergleichenden Dateien nach ähnlichen Zeilen sucht um diese an den selben Zeilenindex zu schreiben. Für die jeweiligen Strings

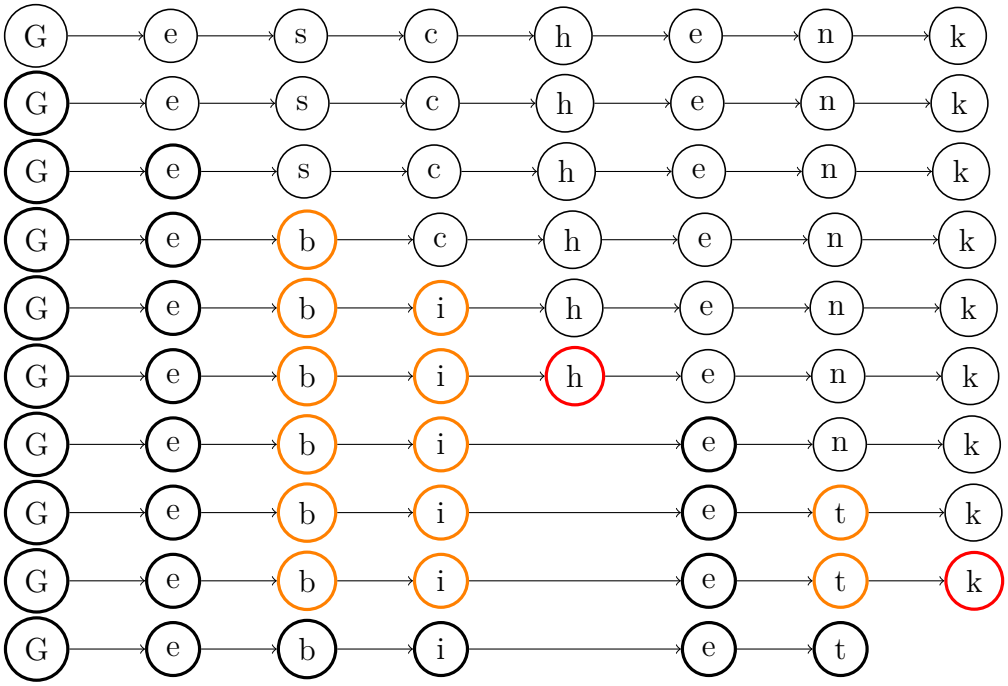

Abbildung 2: Beispiel der String-Transformation

S_1, S_2 zweier nichtleerer Zeilen Z_1, Z_2 existiert genau dann ein Match M wenn

$$LLCS(S_1, S_2) \geq (max(|S_1|, |S_2|) \cdot x), \ mit\ 0 \leq x \leq 1 \tag{1}$$

gilt, wobei x ein vom Benutzer einstellbarer Ähnlichkeitsfaktor ist.

2.1.2 Levenshtein-Distanz

Die Levenshtein-Distanz, benannt nach dem russischen Mathematiker Vladimir Levenshtein, ist eine weitere verbreitete Form der Edit Distance. Im Gegensatz zur LLCS beschreibt sie nun tatsächlich die minimale Anzahl an Operationen um Strings ineinander zu überführen und nicht mehr wie ähnlich sich zwei Strings sind. Als kleines Beispiel wird in Abb. 2 die Transformation des Wortes „Geschenk“ in „Gebiet“ vorgeführt. Ein orangener Knoten symbolisiert eine Änderung und ein roter Knoten steht für eine Löschung und ein dicker, schwarzer Rand kennzeichnet ein unverändertes Zeichen. Dabei ergibt sich für die beiden Strings eine Edit Distance von 5 und ist damit gleich der Anzahl der unterschiedlichen, farbigen Knoten.

Allerdings kann diese Distanz für zwei Strings S_1 und S_2 auch in ein Ähnlichkeitsmaß

überführt werden:

$$\ddot{A}hnlichkeit(S_1, S_2) = 1 - \frac{LevenshteinDist(S_1, S_2)}{max(|S_1, S_2|)} \tag{2}$$

Für das Beispiel in Abb. 2 ergibt sich dann eine Ähnlichkeit von $0,375\%$.

Diese Funktion wird im MultiTextCompare eingesetzt um die Ähnlichkeit für Dateien prozentual zu quantifizieren. Im *Line Compare*-Modus werden für zwei Dateien A und B Zeile für Zeile die Ähnlichkeiten nach Gleichung (2) errechnet und mit einem Gewicht multipliziert.

$$Zeilengewicht(A, B) = \frac{1}{max(|A|, |B|)} \tag{3}$$

Das Gewicht wird wie in Gleichung 3 errechnet, wobei hier die Länge einer Datei ihrer Zeilenanzahl entspricht. Die Gesamtähnlichkeit der beiden Dateien ergibt sich dabei aus der Summe ihrer Teilähnlichkeiten.

Der zweite Vergleichsmodus *Character compare* funktioniert ebenfalls nach Formel (2), wobei diesmal die zwei Input-Strings die alphabetisch sortierte Konkatenation der Zeichen aller Zeilen aus der jeweiligen Datei sind. Dies hat den Vorteil, dass Dateien mit gleichem, aber verschieden sortiertem oder verschobenem Inhalt immer noch als gleich erkannt werden. Der Nachteil ist, dass es nur für Dateitypen sinnvoll funktioniert, die eine gewisse Dateistruktur aufweisen und ohnehin ähnlich sind. Dadurch, dass alle Zeichen außerhalb ihrer eigentlichen Reihenfolge betrachtet werden, erhöht sich zudem das Risiko für *false positives*, also Dateien die als ähnlich anerkannt werden obwohl ihre Inhalte semantisch klar verschieden sind.

2.2 Extensible Markup Language

Extensible Markup Language (XML) ist eine vom *World Wide Web Consortium* (W3C) definierte Auszeichnungssprache, die dazu dient hierarchisch angeordnete Daten strukturiert darzustellen. Diese Strukturierung erfolgt in Form eines Baumes. Jedes XML-Dokument enthält genau einen Wurzelknoten, welcher Kindknoten enthalten kann. Diese Kindknoten können ebenfalls beliebig viele Kindknoten aufweisen [7, Abs. 2].

Die Knoten innerhalb des Baumes können außerdem verschiedene Typen aufweisen. Der Wurzelknoten ist vom Typ Element und schließt weitere Elementknoten ein. Neben Elementknoten existieren auch Attribut- und Textknoten, welche allerdings stets im Kontext

eines Elements existieren. Diese Zusammenhänge sind beispielhaft in Abb. 3 dargestellt, in welcher eine XML-Datei mit einem zugehörigen Baumgraphen abgebildet ist.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Song>
        <Artist>Deep Purple</Artist>
        <Name album="Machine Head">Smoke on the Water</Name>
        <Year>1972</Year>
        <Genre type="Rock">
                <Subgenre>Hard Rock</Subgenre>
        </Genre>
</Song>
```

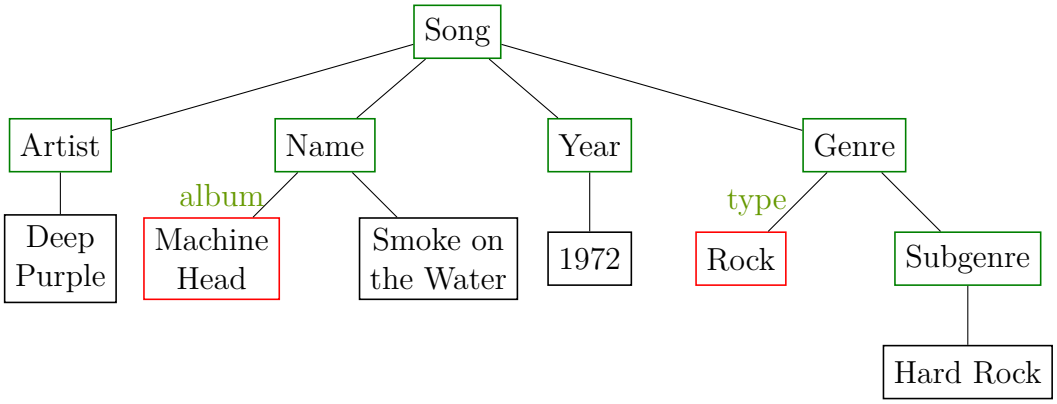

Abbildung 3: XML-Datei und zugehöriger Baum

Darüber hinaus existieren noch weitere Knotentypen wie Kommentare oder CDATA-Abschnitte, wobei letztere eine Sonderform des Textknotens sind und auch sonst für Markup reservierte Zeichen erlauben [7, Abs. 2.7].

Zuletzt sollen noch kurz die Begriffe der *Wohlgeformtheit* und der *Validität* eingeführt werden. Ein XML-Dokument ist dann wohlgeformt, wenn alle Regeln gemäß der Definition der W3C in dem Dokument eingehalten werden. Diese beinhalten u. a., dass es wie zuvor erwähnt nur ein Wurzelelement gibt oder dass jedes geöffnete Tag <*Elementname*> auch ein schließendes Tag </*Elementname*> besitzt [7, Abs. 2.1].

Ein valides XML-Dokument muss wohlgeformt sein, einer Grammatik zugehörig sein und diese auch einhalten. Mögliche Formen dieser Grammatiken sind Dokumenttypdefinitionen (DTD) und XML-Schemadefinitionen (XSD)[7, Abs 2.8][8, Abs 2.1].

2.3 JavaScript Object Notation

JavaScript Object Notation (JSON) ist ein weiteres textbasiertes Datenformat, welches auf einer hierarchischen Datenstruktur basiert und dadurch als Baumstruktur betrachtet

werden kann. Es ist dadurch in bestimmten Aspekten ähnlich zu XML, wobei es aber distinkte Unterschiede gibt.

Jedes JSON-Dokument beginnt mit einem Wurzelknoten, welches anders als in XML keinen eigenen Namen hat und damit ausschließlich Informationen zu den eigenen Kindknoten beinhaltet. Für diese Kindknoten existieren im Wesentlichen 3 verschidene Knoten- bzw. Elementtypen:

Value-Knoten Die einfachste Form in JSON Daten zu speichern, ist in einem einzigen Key-Value Paar. Der Key ist dabei ein auf der aktuellen Ebene eindeutiger Identifier, welcher innerhalb von doppelten Anführungszeichen steht und von einem Doppelpunkt gefolgt wird. Das Value kann ein Integer-, Boolean-, String- oder Null-Wert sein [9]. Für Strings besteht die Besonderheit, dass dort das Value auch in doppelten Anführungszeichen stehen muss.

Array-Knoten Ein JSON-Array ist eine geordnete Liste an beliebigen Elementtypen, die jeweils durch Kommata getrennt sind [9]. Ein Array hat auch genau einen für die Ebene eindeutigen Key und die Value-Elemente werden über ihren Index im Array adressiert. Diese Elementliste beginnt mit einem „[“ und endet mit einem „] “.

Objekt-Knoten Ein JSON-Objekt wird ebenfalls über einen eindeutigen Key adressiert. Es beinhaltet eine ungeordnete Liste an Elementen, welche wie beim Array durch Kommata getrennt sind. Anders als beim Array ist jedes Element innerhalb eines Objekts ein Key-Value Paar, und wird über den Key adressiert. Die Elemente eines JSON-Objekts sind durch „{ “ bzw. „} “ begrenzt.

Im Vergleich zu XML fehlt also die Unterstützung für Attribute und Kommentare als Knotentyp. Häufig lassen sich dennoch Informationen aus einer XML-Darstellung in JSON übersetzen. Dies ist hier beispielhaft für die XML-Datei aus Abb. 3 geschehen.

Analog zu XML existieren die Begriffe der Wohlgeformtheit und Validität auch für JSON. Für valide JSON-Dateien existieren JSON-basierte Schemata gegen die diese Dateien geprüft werden. Außerdem muss eine valide JSON-Datei wohlgeformt sein und darf dadurch keine strukturellen Fehler beinhalten.

```
{
    "Artist": "Deep Purple",
    "Name": {
       "Album": "Machine Head",
       "Name": "Smoke on the Water"
    },
    "Year": 1972,
    "Genre": {
       "Type": "Rock",
       "Subgenre": ["Hard Rock"]
    }
 }
```

Abbildung 4: JSON-Datei analog zu Abb. 3

2.4 Java

Java ist eine vielseitige, objektorientierte und weit verbreitete Programmiersprache. Laut [10] gehört Java auch aktuell noch zu den 3 beliebtesten Programmiersprachen nach C und Python. Im Folgenden werden Java-spezifische Aspekte, die für das Projekt relevant sind, kurz aufgeführt.

2.4.1 Verwendete Libraries

Für die Entwicklung einer Software mit mehreren Funktionen ist es häufig von Vorteil, bereits geschriebenen Code von reputablen Quellen wiederzuverwenden. Im Kontext von Java ist es möglich, fremden Code in Form von Libraries zu verwenden. Zu den zugehörigen Vorteilen gehört zum einen die Zeitersparnis, da der Code nicht neu entwickelt werden muss und zum anderen die Qualität des Codes. Häufig sind Libraries Open-Source, wodurch jeder Entwickler die Möglichkeit hat Fehler zu finden und zu melden oder diese direkt zu beheben.

Bei der Entwicklung von MultiTextCompare wurden deshalb mehrere Libraries verwendet.

JDOM

JDOM bietet eine breit aufgestellte API zur Verarbeitung von XML-Dateien. In MultiTextCompare wird JDOM verwendet um XML-Dateien auf Wohlgeformtheit und Validität zu prüfen, um sie bspw. durch Sortierung zu manipulieren und um ihre XML-Strukturbäume für den Vergleich zu nutzen. Für die Validierung enthält JDOM einen Parser, der die Dateien zunächst nur auf Wohlgeformtheit prüft. Je nach Konfiguration

ist es auch möglich, die Dateien über DTDs und XSDs zu validieren. Für das Traversieren der Dokumentbäume bzw. des Document Object Model (DOM) liefert JDOM die Klassen *Element* und *Attribute*. Dabei ist es für jedes Element möglich, dessen Kindelemente und Attribute zu erfragen.

Jackson

Analog zu JDOM und XML bietet Jackson eine API für die Verarbeitung von JSON-Dateien. Die Aufgaben der API im Projekt sind nahezu identisch, wobei hier die Validierung wegfällt. Auch über Jackson lässt sich das DOM erzeugen und navigieren. Dafür existiert die Klasse *JsonNode* mit welcher sich die Kindknoten eines Elements auflisten lassen.

Apache Commons

Apache Commons ist eine Bibliothekssammlung von frei verfügbaren Java-Klassen. Für das Projekt wird die Implementierung der Levenshtein-Distanz, LCS-Länge und Diff innerhalb von *apache.commons.text* verwendet, sowie einige dateibezogene Utility-Methoden aus der Bibliothek *apache.commons.io*.

2.4.2 Nebenläufigkeit

Java bietet Entwicklern die Möglichkeit, die Ausführung ihrer Software zu parallelisieren. Konkret passiert das über das Konzept von Threads, also die mögliche Aufteilung von Aufgaben eines zugehörigen Prozesses auf kleinere Teilstücke. Falls keine Parallelisierung stattfindet, existiert pro Prozess genau ein Thread. Prozesse und Threads werden vom Betriebssystem bzw. vom Scheduler des Betriebssystems verwaltet, wobei Java über die Klasse *Thread* Methoden bereitstellt, um Programme aufzuteilen und auf mehrere Threads zu verteilen. Das sog. *Multithreading* hat besonders für Mehrkernprozessoren Vorteile, da die ausgeführte Java-Anwendung so mehrere Prozessorkerne gleichzeitig belasten kann, anstatt alle Aufgaben sequentiell auf einem Prozessorkern abzuarbeiten. Dies kann zu einer erheblichen Verbesserung der Bearbeitungszeit führen.

2.4.3 Java Swing

Java Swing ist ein UI-Toolkit, welches gängige UI-Komponenten in Form von Java Klassen realisiert und für die Entwicklung zur Verfügung stellt. Dazu gehören u. a. Buttons, Textfelder und Labels, die in nahezu jeder Anwendung mit Benutzerinteraktion vorhanden sind.

Swing ist durch die Vielseitigkeit und Modularität der Komponenten zwar auf der Oberfläche relativ simpel aufgebaut, allerdings gibt es ein paar Besonderheiten im Zusammenhang mit Threads. Für das Event-Handling existiert in Swing der Event Dispatch Thread (EDT). Wird bspw. ein Button vom Benutzer gedrückt, so wird die dem Button zugehörige Handler-Methode aufgerufen und auf dem EDT ausgeführt [11]. Für die Ausführungsdauer dieser Handler-Methode ist der EDT dann blockiert und es kann nicht auf weitere Benutzereingaben reagiert werden. Um dennoch Methoden mit langer Ausführdauer aus der UI aufrufen zu können, existiert die Java Klasse *SwingWorker*, die zwei überladene Methoden *doInBackground()* und *done()* zur Verfügung stellt. Für *doInBackground()* wird ein separater Worker-Thread erstellt, auf dem dann die lang andauernden Methoden laufen während der EDT weiterhin auf Benutzereingaben reagiert. Erst sobald der Worker-Thread beendet ist, wird die *done()*-Methode ausgeführt [12]. Ein weiteres, wichtiges Detail ist, dass UI-Komponenten in Swing nicht thread-safe sind. Falls also zwei Threads gleichzeitig auf eine Komponente zugreifen, führt dies zu undefiniertem Verhalten. Dadurch gilt es als *best practice* UI-Komponenten nur über den EDT zu manipulieren [11]. Im Kontext von SwingWorker, sollten jegliche UI-Zugriffe nur über die *done()*-Methode ausgeführt werden, da diese auf dem EDT ausgeführt wird.

3 Konzeption

3.1 Vergleich zu bestehender Software

Im Wesentlichen hat MultiTextCompare zwei zentrale Funktionen. Zum einen den Vergleich beliebig vieler textbasierter Dateien, und zum anderen die Anzeige der Diff für bis zu drei Dateien. Es gibt Anwendungen, die eine prozentuale Ähnlichkeit für zwei Dateien anzeigen können. Jedoch liegt dort der Fokus auf dem semantischen Textvergleich im Sinne der Plagiatserkennung. Wie bereits in 2.1 erwähnt, arbeitet MultiTextCompare rein zeichenbasiert und reagiert daher nicht auf sprachliche Ähnlichkeiten bei der Ähnlichkeitsbewertung.

Im Bereich der Diff-Tools existieren häufig Programme, die die Diff für maximal zwei Dateien anzeigen können. Eine Ausnahme ist hierbei die Anwendung *KDiff3*, welche einen Vergleich für drei Dateien zulässt. Um eine Diff für drei Dateien zu erzeugen, muss eine der Dateien als Referenz markiert werden. Für die drei Inputdateien A,B,C mit A als Referenz entstehen dann die Vergleiche $\{A, B\}, \{A, C\}$, während MultiTextCompare keine Datei als Referenz festlegt und jede Datei mit jeder vergleicht.

Um Unterschiede farbkodiert darzustellen, ist ein Ansatz mit nur zwei Vergleichen zwar für den Benutzer einfacher nachzuvollziehen und zudem performanter als alle drei Vergleiche auszuführen, aber dafür gehen Informationen verloren. Gerade dann, wenn 3 ähnliche Dateien für die Diff ausgewählt werden, kann es interessant sein, die Unterschiede zwischen allen Dateien gleichermaßen zu betrachten.

3.2 Anforderungen

Für die Entwicklung der Software MultiTextCompare sind einige funktionale und nicht-funktionale Anforderungen entstanden. Die funktionalen Anforderung wurden bereits größtenteils innerhalb des Einleitungskapitels beschrieben. Zu den nicht-funktionalen Anforderungen gehört der Aspekt, dass die Software speziell für die Laborrechner der Technischen Hochschule Köln entworfen werden soll. Dafür soll Java in der Version 1.7.0_75 für 32 Bit Systeme verwendet werden. Als IDE soll Eclipse (Release Luna SR2) verwendet werden, um die Code-Basis auch für die Installationen im Labor nahtlos lauffähig zu machen und um nicht auf generierte Java Archive (JAR) angewiesen zu sein. Zudem soll die Software primär mit Windows 7 kompatibel sein.

Eine weitere Anforderung ist die einfache Benutzbarkeit. Um dies zu gewährleisten sollen

alle notwendigen Funktionen auf den ersten Blick erkennbar, und jegliche Auswahlmöglichkeiten annotiert sein, um dem Benutzer Hilfestellung zu leisten.

3.3 Architektur

Als Architektur wurde auf eine 3-Schichten-Architektur zurückgegriffen. Diese ist grob in Abb. 5 als UML-Komponentendiagramm dargestellt. Die Komponenten sind hier als logische Zusammenfassungen von Java Klassen gemeint, wobei jede Komponente aus mindestens einer Java Klasse besteht und genau ein Aufgabengebiet erfüllt. Zusätzlich handelt es sich bei den verwendeten Pfeilen um *«uses»* Relationen, wobei die Annotationen zwecks einer besseren Lesbarkeit weggelassen wurden. Aus dem gleichen Grund wurde auf die Darstellung von Interfaces und detailierten Relationen zwischen den Schichten verzichtet.

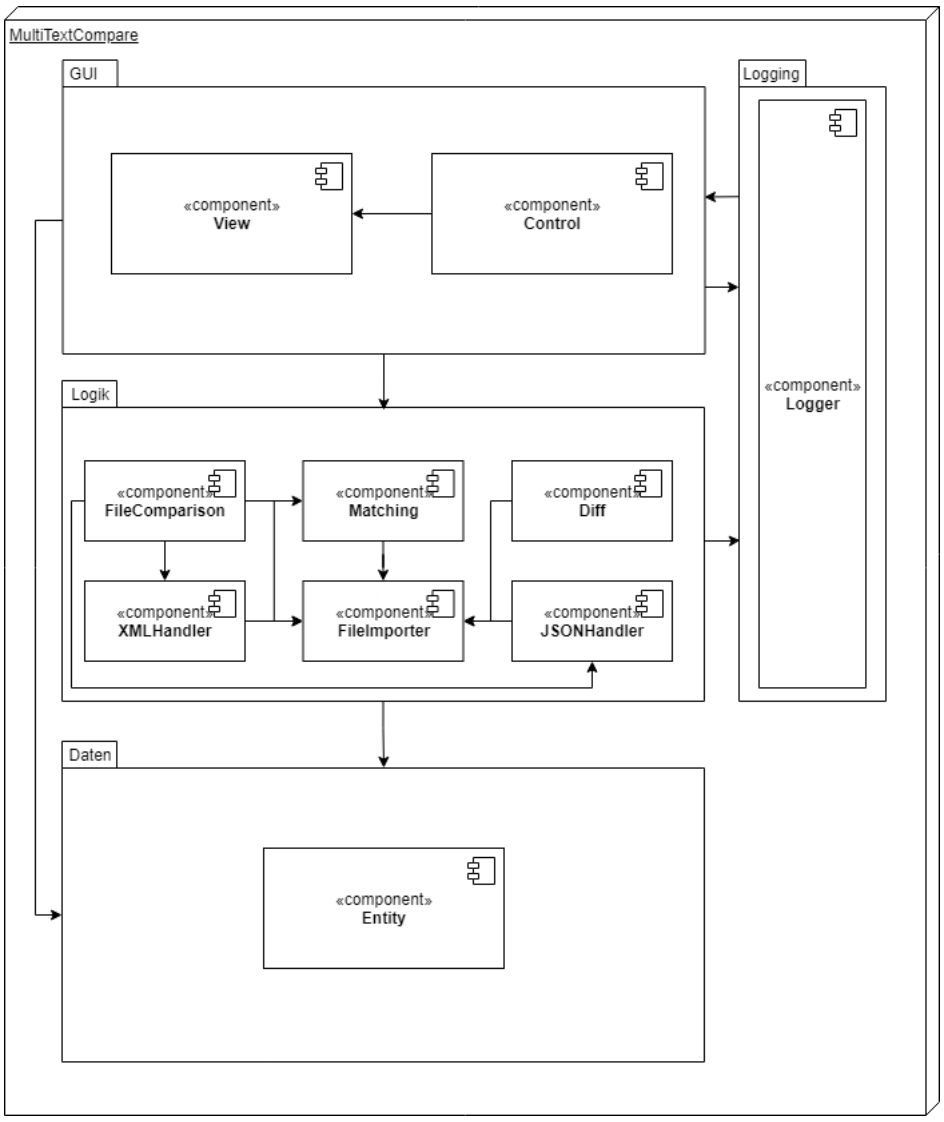

Abbildung 5: Übersicht der Architektur

In der obersten Schicht liegen alle Java-Klassen, die mit der UI in Verbindung stehen. Dort existiert eine Teilung zwischen Klassen der View-Komponente und denen der Control-Komponente. View-Klassen initialisieren bestimmte UI-Elemente und bestimmen deren Aussehen. Jede View-Klasse, die dynamische Inhalte anzeigen kann, hat eine zugehörige Controller-Klasse für das Event-Handling und Verändern der UI.

In der Logikschicht befinden sich die Komponenten, die für die Algorithmen und das File-Handling verantwortlich sind. Jede öffentliche Klasse dieser Schicht bietet dabei ein Interface an die GUI-Schicht an. Die Komponente *FileImporter* ist u. a. dafür zuständig, die vom Benutzer ausgewählten Dateien zu finden, sie als temporäre Dateien zu klonen und auf die Originaldateien zu mappen. Alle anderen Komponenten greifen auf diese Map zu, um die Dateien weiter zu verarbeiten. Die *FileComparison*-Komponente beherbergt die Vergleichsalgorithmen, die die Werte für die Matrix berechnen. Für Operationen an XML und JSON wie die Filterung oder Sortierung sind die Komponenten XMLHandler bzw. JSONHandler zuständig.

Die Datenschicht ist letztlich eine rein logische Trennung von den anderen Schichten. Da keine Datenbank existiert, gibt es keine Notwendigkeit für CRUD-Komponenten. Die im Diagramm gezeigte Entity-Komponente beherbergt also nur Java Objekte, die für die Speicherung von Daten zur Laufzeit notwendig sind. Dazu gehört bspw. die Klasse *IConfigImpl*, die ein Klassenattribut für jede konfigurierbare Einstellung enthält und zur Laufzeit mit dem Inhalt der aktiven Konfigurationsdatei befüllt wird. Jede Entitätsklasse besitzt ebenfalls ein zugehöriges Interface, welches die *Getter* und *Setter* an höhere Schichten übergibt.

Die Logging-Komponente existierte zwar in Version 1.0 noch nicht, da sie aber die einzige architektonische Änderung während der Bearbeitung dieser Arbeit ist, wird sie hier bereits aufgeführt. Sowohl Komponenten aus der Logik-Schicht als auch aus der GUI-Schicht, können Fehler loggen. Diese Fehler werden in der UI dargestellt, falls das entsprechende Log-Level vom Benutzer gesetzt ist. Die genauere Funktionsweise wird in Kapitel 4.4.5 beschrieben. Informationen zu den einzelnen Klassen, Interfaces und Methoden können als Javadoc direkt über MultiTextCompare im Browser geöffnet werden.

4 Implementierung

In diesem Kapitel werden die in 1.3 geplanten Verbesserungen an MultiTextCompare durchgeführt und dargestellt. Angegebene Codestücke sind zwecks Lesbarkeit teilweise auf die relevantesten Zeilen gekürzt.

4.1 Verbesserung des Zeilenmatchings

Textbasierte Dateien können inhaltlich identisch sein, ohne dass sie exakt gleich aussehen. Sind zwei Dateien inhaltlich identisch aber um eine Zeile verschoben, würden sie von der Software als sehr unterschiedlich bewertet werden. Daher existiert die Matching-Komponente, die sicherstellt dass gleiche oder ähnliche Zeilen erkannt werden, und in der Diff-Anzeige im gleichen Zeilenindex stehen.

Diese Eigenschaft ist daher unerlässlich für den Diff-Prozess innerhalb der Software. Im Kontext der Laufzeit, ist sie aber sehr kostspielig weshalb Maßnahmen notwendig sind, um diese zu senken. Dafür wird hier zunächst die ursprüngliche Funktionsweise in Version 1.0 zusammengefasst dargestellt.

4.1.1 Optimierung des Matching-Algorithmus

Das Matching kann in zwei Teilprobleme zerlegt werden. Auf der einen Seite existiert die Matcherkennung, bei der festgelegt wird ob zwei Zeilen überhaupt gematcht werden können. Auf der anderen Seite gibt es noch das Alignment, also die Sicherstellung davon, dass alle Zeilen nach dem Matching korrekt dargestellt werden können. Im Folgenden wird nur auf den ersten Aspekt eingegangen, da dieser die hauptsächliche Funktionalität darstellt.

Um für zwei Dateien A und B alle Matches zu ermitteln, wird zunächst die Datei mit weniger Zeilen als Referenzdatei festgelegt. Dies sei ab jetzt Datei A. In Version 1.0 wurde für jede Zeile von A jede Zeile von B nach einem Match durchsucht. Die Kriterien für ein Match waren dabei, dass sich die Zeilen nach Gleichung (1) ähnlich genug sind und dass keine der infrage kommenden Zeilen bereits gematcht wurde. Zudem sind Matches „über Kreuz“ ausgeschlossen. Das heißt, dass die Matchindices für ein potenzielles Match immer größer sein mussten, als die des letzten erfassten Match. Der zugehörige Code ist in Codeabschnitt 1 dargestellt.

Besonders ausschlaggebend für die Laufzeit ist dabei die Feststellung der Ähnlichkeit.

```
1  String ref = "", comp = "";
2  int lastMatchedIndex = 0;
3  // Schaue für jede Zeile der linken Datei
4  for (int i = 0; i < lineCountLeft; i++) {
5          reference = leftFile.get(i);
6          // ob es in der rechten Datei ein Match gibt
7          for (int j = 0; j < lineCountRight; j++) {
8                  comp = rightFile.get(j);
9                  if (matches(reference, comp) // falls Zeilen ähnlich genug
10                                 && j >= lastMatchedIndex // falls Match nicht über Kreuz
11                                 && notMatchedYet(i, j)) { // falls noch nicht gematcht
12                         lastMatchedIndex = j;
13                         // Speichere Match
14                         matches.add(new IMatchImpl(i, j, reference, comp));
15                         // Gehe in die nächste Zeile der Referenzdatei
16                         break;
17                 }
18
19         }
20 }
```

Quellcode 1: Matching in V1.0

Für eine Optimierung sollte die Anzahl dieser Feststellungen also, wo möglich, reduziert werden.

Für den Fall, dass in beiden Dateien kein einziges Match existiert, würde jede Zeile von A mit jeder Zeile von B verglichen werden. Dieses Szenario enthält die größtmögliche Anzahl an Vergleichen und stellt daher den worst case dar. Um hier die Laufzeit zu verbessern, wird ein *Lookahead*-Limit eingeführt. Das bedeutet, dass für jede Zeile der Referenzdatei A_i eine maximale Suchreichweite abhängig vom aktuellen Zeilenindex i festgelegt wird. Als Beispiel würde eine Zeile A_5 mit einem Lookahead-Wert von 10 mit jeder Zeile $B_{0...15}$ verglichen werden sofern kein Match für $B_{0...15}$ existiert und Datei B mindestens 15 Zeilen lang ist. Ist Datei B kürzer, so werden alle Zeilen bis zum Dateiende verglichen.

Auch wenn Matches existieren, kann ein solches Lookehead-Limit Vorteile haben. Für größere Dateien kann es passieren, dass ein Match erst 50-100 Zeilen nach dem letzten gefunden wird. Dies führt dann zu genauso vielen Leerzeilen im Alignment-Prozess, was die Lesbarkeit für den Benutzer beeinträchtigen kann. Da die Eingabedateien vor dem Vergleichen nicht bekannt sind, wird dem Benutzer die Möglichkeit gegeben, den Schwellwert selbst nach Bedarf anzupassen.

Eine weitere Auffälligkeit im Quellcode-Abschnitt 1 ist, dass die Indexvariable j, unabhän-

gig davon ob ein Match gefunden wird oder nicht, immer bei 0 anfängt. Dabei würde es genügen, wenn nach einem Match $M_{i,j}$ der Vergleich erst bei $i = i+1$ und $j = j+1$ fortgesetzt werden würde. So müsste auch nicht mehr geprüft werden ob ein Match über Kreuz ist oder ob es bereits ein Match mit den aktuellen Indices gibt. Diese Optimierungen sind im Quellcode 2 dargestellt

```
1  String ref = "", comp = "";
2  int lowestPossibleIndex = 0;
3  // Schaue für jede Zeile der linken Datei
4  for (int i = 0; i < lineCountLeft; i++) {
5          reference = leftFile.get(i);
6          int maxSearchIndex = i + LOOKAHEAD + 1;
7          if (maxSearchIndex > lineCountRight) {
8                  maxSearchIndex = lineCountRight;
9          }
10         // ob es in der rechten Datei ein Match gibt
11         for (int j = lowestPossibleIndex; j < maxSearchIndex; j++) {
12                 comp = rightFile.get(j);
13                 if (matches(reference, comp)) { // falls Zeilen ähnlich genug
14                         lowestPossibleIndex = j + 1;
15                         // Speichere Match
16                         matches.add(new IMatchImpl(i, j, reference, comp));
17                         // Gehe in die nächste Zeile der Referenzdatei
18                         break;
19                 }
20         }
21 }
```

Quellcode 2: Aktualisiertes Matching mit Lookahead

4.1.2 Verbesserung der Funktionsweise

Trotz Optimierung gibt es noch einen weiteren Aspekt der verbessert werden kann. Aktuell wird die Suche nach Matches nach dem ersten Match beendet. So entstehen Situationen, bei denen mehrere ähnliche Zeilen nah beieinander existieren und ein Match der ersten möglichen Zeile nicht die beste Option wäre.

In Abb. 6 sind zwei JSON-Dateien abgebildet, bei denen dieses Szenario zutrifft. In der Standardeinstellung würden, bis auf die schließenden Klammern, keine Zeilen der zweiten Datei verschoben werden obwohl in Datei 1 ein identischer Eintrag existiert. Zwar könnte der Wert für x aus Gleichung (1) auf 1 erhöht werden, allerdings würden dadurch nicht-identische aber ähnliche Zeilen für das Matching ignoriert werden falls die Dateien weitere Einträge hätten.

```
{
  "phones" : [ {
          "brand": "Apple",
          "name": "IPhone 12"
  },
  {
      "brand": "Apple",
          "name": "IPhone 13"
  } ]
}
```

(a) Beispiel-JSON 1

```
{
  "phones" : [ {
          "brand": "Apple",
          "name": "IPhone 13"
  } ]
}
```

(b) Beispiel-JSON 2

Abbildung 6: Zwei ähnliche JSON-Dateien

Die Lösung für dieses Problems wird durch einen *Best-Match* Algorithmus gelöst. Innerhalb des Lookahead-Limits werden alle potenziellen Matches in einer zusätzlichen Liste gesammelt. Mit jedem potenziellen Match wird zusätzlich die LLCS mitgespeichert. Nachdem alle nötigen Zeilen verglichen wurden, wird die Kandidatenliste nach der durch die maximale Länge der verglichenen Strings geteilten LLCS sortiert und das Match verwendet, welches dort den höchsten Wert besitzt. Diese Normierung führt u. a. dazu, dass ähnlichere Matches trotz gleicher LLCS bevorzugt werden. Ein Beispiel dafür wäre der Vergleich der Zeilenpaare (*Kunde*, *Kunden*) und (*Kunde*, *Kundennummer*), bei denen die LCS für beide Paare 5 ist, aber die Edit-Distance für das zweite Paar mit 7 deutlich größer ausfällt als die des ersten Paares mit Distanz 1. Dadurch ergäbe sich für Paar 1 ein normierter Ähnlichkeitswert von ca. 0,83 und für Paar 2 ein Wert von nur 0,42, weswegen Paar 1 als Match bevorzugt werden würde. Sollte dieser Wert für zwei potenzielle Matches gleich sein, wird danach entschieden ob eins der Matches direkt auf ein vorangegangenes Match folgt, um möglichst nach verschobenen Blöcken zu suchen. Existiert kein direkter Vorgänger und der Ähnlichkeitswert ist gleich, wird das Match mit geringerem Index verwendet. Das Ergebnis innerhalb der Software ist in Abb. 7 dargestellt. Nun werden alle verfügbaren Zeilen der rechten Datei auch in der linken Datei als existent dargestellt. Das Ergebnis zeigt jedoch auch eine Limitation des Verfahrens. Da es nicht auf die Syntax der unterliegenden Dateiformate eingeht, kann nicht darauf geachtet werden ob ganze Objekte oder Array-Einträge verschoben wurden. Für den hier beschriebenen Fall führt diese Limitation wiederum zu keinen Abzügen in der Genauigkeit bei der Bewertung durch den *Line Compare*-Vergleichsmodus.

Abbildung 7: Resultat des Best-Match Verfahrens

4.2 Überarbeitung der Vergleichsalgorithmen

Das ausschlaggebende Feature der Software ist ihre Ähnlichkeitsmatrix. Diese wurde bereits in der Einleitung kurz erwähnt. Auch die Funktionsweisen der beiden Vergleichsalgorithmen *Line Compare* und *Character Compare* wurden bereits oberflächlich behandelt. Dieses Unterkapitel behandelt nun die Überarbeitung des Modus Line Compare und die Einführung zweier neuer Vergleichsmodi, die sich in ihrer Funktionsweise stark von den bisherigen Algorithmen unterscheiden.

4.2.1 Zeilenweiser Vergleich regulärer Textdateien

Line Compare war der erste Vergleichsmodus der Software. Da zum Zeitpunkt der Entwicklung noch keine Matching-Komponente existierte, arbeitete der Vergleichsmodus nicht mehr optimal sobald ähnliche Zeilen leicht verschoben waren. Das Ziel ist nun die Einbettung der Matching-Komponente in den regulären Textvergleich um Dateien, die dem Benutzer in der Diff-Anzeige als ähnlich angezeigt werden würden, auch mit einer höheren Genauigkeit für den Vergleich bewerten zu können.

Alle Vergleiche basieren auf den von der Komponente *FileImporter* erstellten temporären Dateien. Diese sind notwendig um die vom Benutzer ausgewählten Originaldateien bei einer Normierung, also zum Beispiel einer Entfernung aller Leerzeichen, nicht zu verändern. Des Weiteren wurde explizit für den Line Compare Modus ein weiteres Set an temporären Dateien erstellt, bei denen nach jedem Wort ein Zeilenumbruch eingefügt wurde. Als Wort

galt eine beliebige Zeichenkette gefolgt von einem Leerzeichen.

Dieser Prozess sollte eigentlich dazu dienen, um die Genauigkeit des Vergleichs leicht zu verbessern, wobei dies nur für Dateien funktioniert, die ohnehin ähnliche Zeileneinträge in der gleichen Reihenfolge besitzen.

Um die Matching-Komponente in den Vergleich einzubinden, muss zunächst ihre Schnittstelle angepasst werden. Aktuell hat die Matching-Methode *matchLines()* nämlich noch den Rückgabetyp void, denn ihre Erzeugnisse waren bisher immer neue temporäre Dateien im Dateisystem auf die andere Komponenten zugreifen konnten. Um IO-Zugriffe zwecks Laufzeit zu minimieren, erstellt die Methode diese temporären Dateien nicht falls sie für den Vergleich aufgerufen wird. Stattdessen werden die gematchten Zeilen als ein *Object*-Array zurückgegeben. So kann der Rest des Vergleichsalgorithmus unverändert bleiben. Das heißt, dass Zeilen des gleichen Index nach dem in Abschnitt 2.1.2 beschriebenen Verfahren bewertet werden. Sind die beiden zu vergleichenden Dateien nicht gleich lang, wird die kürzere Datei mit Leerzeilen aufgefüllt. Das Ergebnis ist ein zeilenbasierter Vergleichsalgorithmus, der nun verschobene Zeilen erkennen und genauer bewerten kann. Gleichzeitig profitiert er von den in Kapitel 4.1 durchgeführen Verbesserungen.

4.2.2 Vergleich von XML-Dateien

Erweiterung der Element-Sortierung

In Version 1.0 besteht bereits die Möglichkeit, die Strukturbäume von XML-Dateien nach ihren Element-Namen und Attribut-Namen zu sortieren. Die Sortierung der Attribute ist insofern eindeutig, als dass ein Attributname pro Element immer einzigartig sein muss. Für Elemente gibt es allerdings keine solche Regel. Elemente, die den gleichen Namen haben und auf der gleichen Ebene innerhalb des Baumgraphen liegen, können als Liste interpretiert werden.

Vergleichsoperationen wie Line Compare und die Berechnung der Diff würden davon profitieren, wenn XML-Dateien eindeutig sortiert werden könnten, da sie zeilenbasiert operieren. Wenn ähnliche Zeilen in der gleichen Reihenfolge vorliegen, können sie zusätzlich durch das Zeilenmatching erkannt werden. Da weder durch die XML-Spezifikation noch durch die JDOM-API eine Sortierreihenfolge vorgeschrieben wird, wird im Folgenden eine eigene Reihenfolge festgelegt. Diese Priorisierung bezieht sich auf den Vergleich von zwei Elementen:

1. Elementname
2. Attributname
3. Attributwert
4. Textinhalt

Die Implementierung erfolgt über die Schnittstelle *java.util.Comparator* und das Überschreiben ihrer *compare()*-Methode. Diese gibt einen Integer-Wert zurück mit Hilfe dessen zwei Strings lexikographisch als kleiner, größer oder gleich bewertet werden können. Diese Bewertung wird durch die von JDOM breitgestellte Methode *sortChildren()* ausgewertet, um die Elemente strukturerhaltend zu sortieren.

Sind die Elementnamen identisch, werden die Attribute untersucht. Da Elemente beliebig viele Attribute haben können, werden zunächst nur so viele Attribute verglichen, wie in der kürzesten Attributliste vorhanden sind. Dabei werden erst die Attributnamen verglichen und dann die Werte. Dafür ist vorausgesetzt, dass die Attribute vorher bereits sortiert wurden. Sind die bisher betrachteten Attribute sowohl im Namen als auch im Wert identisch, entscheidet die Anzahl der Attribute. Mehr Attribute werden dabei als größer eingestuft. Bei gleicher Attributzahl werden letztlich noch die Textknoten der Elemente miteinander verglichen. Falls alle dieser Kriterien für die beiden Elemente identisch sind, wird das Verfahren in gleicher Reihenfolge rekursiv für alle Kindknoten der beiden infrage stehenden Elemente durchgeführt, bis ein Unterschied festgestellt wird. Bei Gleichheit der Elemente wird nichts verändert.

Struktureller Vergleich von XML-Dateien

Im Vorfeld wurden bereits Limitationen der vorhandenen Vergleichsalgorithmen diskutiert. Weder der zeilen- noch der zeichenbasierte Vergleichsansatz kann für Dateien, die einer gewissen Struktur unterliegen zuverlässig für akkurate Ergebnisse sorgen. Aus diesem Grund wird im Folgenden ein Vergleichsalgorithmus entworfen und implementiert, der XML-Dateien auf Basis der Baumstruktur vergleicht.

Verglichen werden sollen dabei die Element-, Attribut-, Text-, CDATA- und Kommentarknoten, wobei letztere optional sind und durch einen Konfigurationsparameter vom Vergleich ausgeschlossen werden können. Ausschlaggebend für den Vergleich sind allerdings die Elementknoten, da jeder der zuvor genannten Knotentypen ein zugehöriges Elternelement hat. Die einzige Ausnahme ist der Wurzelknoten, wobei dieser vom Vergleich ausgenommen ist. Um *false positives* zu vermeiden werden zwei Elemente nur miteinander verglichen werden, wenn ihre Namen identisch sind.

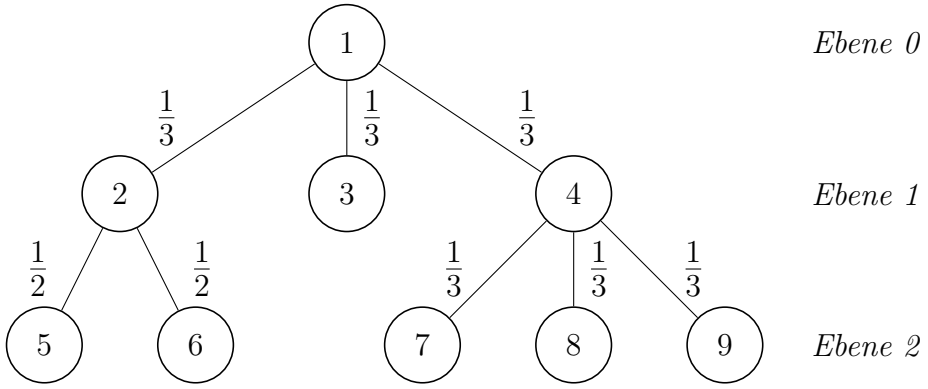

Abbildung 8: Gewichtung von Elementknoten für strukturellen Vergleich

Durch diesen verschiedenen Ansatz, kann das alte Bewertungssystem für die Ähnlichkeit über die LCS oder die Levenshtein-Distanz hier nicht verwendet werden. Stattdessen wird für den strukturellen Vergleich ein Ähnlichkeitsmaß auf Basis der Baumstruktur eingeführt. In Abb. 8 ist ein Beispielbaum dargestellt, für den die Gewichte der Knoten an der Kante zum Elternelement annotiert sind. Für eine sinnvolle Bewertung wird hierbei davon ausgegangen, dass innerhalb der zu vergleichenden XML-Dateien wichtige Informationen nah am Wurzelelement liegen. Zunächst wird geschaut, wieviele Elemente n innerhalb der ersten Ebene liegen. Jedes dieser Elemente geht mit einem Gewicht von $\frac{1}{n}$ in den Gesamtwert der Ähnlichkeit ein. Gleichzeitig wirkt dieses Gewicht als ein Budget für dessen Kindelemente. Im Beispiel hat Knoten 2 ein Gewicht von $\frac{1}{3}$, welches gleichmäßig auf die Kindknoten verteilt wird. Die beiden Kindknoten 5 und 6 haben also im Kontext von Knoten 2 ein Gewicht von $\frac{1}{2}$. Das Gewicht eines Knotens für die Gesamtähnlichkeit wird berechnet aus dem Produkt aller Kantengewichte $f(V_1, V_2)$ auf dem Weg zwischen der Wurzel und genau diesem Knoten. Dadurch ergibt sich für Knoten 5 bspw. ein Gesamtgewicht von

$$f(1,2) \cdot f(2,5) = \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{6}$$

Um diese Gewichte einzusetzen, muss noch definiert werden wodurch zwei Knoten ähnlich sind. Dafür wird bei den Kindknoten der Elemente zwischen textuellem Inhalt, Attributen und Kommentaren unterschieden. Falls Kommentare ignoriert werden sollen oder keine Kommentare präsent sind, sind einzig der Text und die Attribute zu bewerten. Sollten auch keine Attribute existieren, zählt nur der Textinhalt bzw. sollte kein Textinhalt existieren, zählen nur die Attribute. Der Textinhalt schließt reine Textknoten und CDATA-Knoten ein, da JDOM hier auch keine explizite Unterscheidung macht und sie häufig den gleichen Zweck erfüllen. Falls Elemente keinerlei Kindknoten haben, sind sie identisch. Wie Abb. 9 zeigt, wird jeder dieser Knoten, sofern er existiert, gleichwertig gewichtet.

Letztlich muss noch sichergestellt werden, dass das Verfahren symmetrisch ist, also die

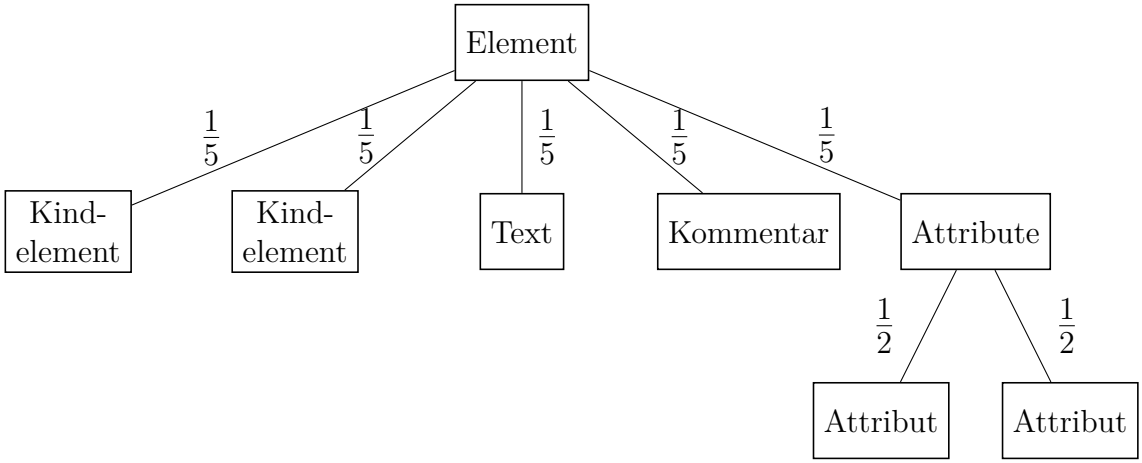

Abbildung 9: Beispielhafte Gewichtung eines Elementinhalts für XML

Reihenfolge der verglichenen Bäume nicht das Ergebnis beeinflusst. Ein Beispiel wäre, wenn der Baum aus Abb. 8 mit einem Baum verglichen werden würde, der auf Ebene 1 vier Knoten besitzt. In diesem Fall würde die maximale Anzahl an Knoten auf der Ebene entscheidend sein, also läge die maximale Ähnlichkeit der Graphen unabhängig vom Inhalt bei $\frac{3}{4}$. In jeder Situation bei der unterschiedliche Anzahlen an Einträgen existieren können, wird die gemeinsame Anzahl an gleichen Einträgen durch die maximal verfügbare Anzahl geteilt.

Implementiert wird dieses Konzept indem zunächst die Wurzelknoten der beiden zu vergleichenden Dateien eingelesen werden, wobei einer der Knoten als Referenzknoten festgelegt wird. Dabei ist nicht wichtig welcher der Knoten die Referenz ist, da in XML das Tauschen von Knoten mit gleichem Elternknoten erlaubt ist. Um diesen Vergleichsmodus anwenden zu können, müssen die jeweiligen XML-Dateien wohlgeformt sein, da ansonsten ein Traversieren der Dokumentbäume nicht möglich ist.

Bei vorliegender Wohlgeformtheit wird zunächst für alle Kindknoten der Referenzwurzel, wie in Quellcode 3 gezeigt, der aktuelle Knotenname in einer Liste gespeichert. Beim ersten Vorkommen dieses Namens werden alle Elemente mit diesem Namen in einer jeweiligen Liste für Referenzknoten (matchingRef) und Vergleichsknoten (matchingComp) gespeichert. Die Methode *getElementCount()* zählt wie oft ein Elementname in einer Liste vorkommt und sorgt dadurch dafür, dass Elementlisten nicht doppelt eingefügt werden.

```
1         List<Element> matchingRef = new ArrayList<Element>();
2         List<Element> matchingComp = new ArrayList<Element>();
3         List<Element> refFirstLevelChildren = rootRef.getChildren();
4         List<String> refElementNames = new ArrayList<String>();
5         for (int i = 0; i < refFirstLevelChildren.size(); i++) {
6                 Element currentRef = refFirstLevelChildren.get(i);
7                 String currentRefName = currentRef.getName();
8                 refElementNames.add(currentRefName);
9                 if (getElementCount(refElementNames, currentRefName) == 1) {
10                        matchingRef.addAll(rootRef
11                                        .getChildren(currentRefName));
12                        matchingComp.addAll(rootComp
13                                        .getChildren(currentRefName));
14                }
15        }
```

Quellcode 3: Suche nach gleichnamigen Knoten auf gleicher Ebene

Da die Matching-Komponente für den strukturellen Vergleich nicht eingesetzt werden kann, muss an dieser Stelle noch ein Mechanismus eingeführt werden um gleiche Listeneinträge zu finden und zu vergleichen. Restliche Listeneinträge können dann noch nach ihrem Index geordnet verglichen werden. Allerdings ist es mit JDOM nicht direkt möglich festzustellen ob zwei Elemente tatsächlich gleich sind ohne alle ihre Kindknoten zu betrachten. Die *equals*()-Methode für die Klasse *Element* prüft nicht auf die inhaltliche Ähnlichkeit sondern ob die Referenz zweier Objekte gleich ist. Dadurch, dass eine rekursive Prüfung aller Kindelemente sehr aufwändig sein kann, wird, wie in Quellcode 4 gezeigt, mit der Klasse *XMLOutputter* von JDOM gearbeitet. Mit ihr ist es nämlich möglich ein Element als String darzustellen. Da die *equals*()-Methode für Strings *true* zurück gibt sobald die verglichenen Zeichensequenzen identisch sind, kann man auf diese Weise die Gleichheit zweier Objekte feststellen. Um identische Elemente vom rekursiven Vergleich auszuschließen, werden diese durch den Wert *null* ersetzt um sie als bereits bewertet zu kennzeichnen. Nachdem alle verfügbaren Elementknoten miteinander auf Gleichheit untersucht wurden, werden gefundenen null-Werte durch eine Hilfsmethode *clearNullValues*() aus den Listen gestrichen.

Für die restlichen Elementknoten werden noch die Inhalte verglichen. Der Vergleich von Textknoten, Attributwerten und Kommentaren basiert auf der Levenshtein-Distanz bzw. der Gleichung (2), da es sich hierbei um reguläre Texte handelt. Bei Attributen werden, wie bei Elementen, nur Attribute mit identischem Namen verglichen. Kommentare werden, falls sie zu mehreren für ein Element existieren, in ihrer originalen Reihenfolge betrachtet.

Alle aus Quellcode 4 übrig gebliebenen Knoten in der Liste *matchingRef* bzw. *matching-*

```
1         for (int i = 0; i < matchingRef.size(); i++) {
2                 Element currentRef = matchingRef.get(i);
3                 for (int j = 0; j < matchingComp.size(); j++) {
4                         Element currentComp = matchingComp.get(j);
5                         if (currentComp == null) {
6                             continue;
7                         }
8                         XMLOutputter xmlOut = new XMLOutputter();
9                         String refString = xmlOut.outputString(currentRef);
10                        String compString = xmlOut.outputString(currentComp);
11                        boolean equals = refString.equals(compString);
12                        if (equals) {
13                            similarity.add(currentLevelWeight);
14                            matchingRef.set(i, null);
15                            matchingComp.set(j, null);
16                            break;
17                        }
18                }
19        }
20        matchingRef = clearNullValues(matchingRef);
21        matchingComp = clearNullValues(matchingComp);
```

Quellcode 4: Suche nach identischen Knoten auf gleicher Ebene

Comp werden in ihrer bestehenden Reihenfolge rekursiv miteinander verglichen. Falls die Listen unterschiedlich lang sind, werden so viele Elemente verglichen wie sie in der kürzeren Liste vorhanden sind, da die restlichen Knoten ohnehin nicht in beiden Bäumen vorhanden sein können.

4.2.3 Vergleich von JSON-Dateien

Erweiterung der Sortierung

Anders als bei XML, existiert für JSON die Möglichkeit Elemente nach ihren Namen eindeutig zu sortieren da die Namen bzw. Keys innerhalb einer Ebene eindeutig sind. Innerhalb der Container-Elemente Array und Object existiert ebenfalls eine eindeutige Zuweisung, da Arrays laut JSON-Spezifikation geordnete Listen sind und innerhalb von Objekten auch nach Keys sortiert werden kann.

Falls kein semantischer Kontext für die Reihenfolge der Array-Elemente vorliegt, kann sich eine alternative, konsistente Sortierung, aus den gleichen Gründen wie in 4.2.2 beschrieben, lohnen. Auch hier muss zunächst eine Ordnungsreihenfolge festgelegt werden, da Arrays beliebige Elementtypen enthalten können. Inhaltlich werden hierbei nur Knoten des gleichen Typs miteinander verglichen, ansonsten wird folgende Reihenfolge verwendet.

1. Valueknoten
2. Arrayknoten
3. Objectknoten

Zwei Valueknoten innerhalb des Arrays werden, unabhägig vom Datentyp, über den Textinhalt ihrer Values als String verglichen. Für zwei Arrays verläuft der Vergleich zunächst über die Anzahl der Einträge, wobei mehr Einträge als größer bewertet werden. Sind die Arrays gleich groß, entscheidet der Inhalt der Knoten. Es wird so lange über die Arrayknoten iteriert, bis zwei Einträge ungleich sind. Bei Objekten ist das Verfahren ähnlich wie bei Arrays, nur dass bei gleicher Anzahl an Elementen zunächst die Keys und dann die Values miteinander verglichen werden.

Die in Quellcode 5 gezeigte Implementierung dieser Vergleiche verläuft wie bei XML über einen selbst erstellten Comparator, wobei diesmal für Jackson keine Methode existiert, die den Strukturbaum mit den sortierten Elementen aktualisiert. Stattdessen wird zunächst mittels einer Tiefensuche rekursiv nach Arrays innerhalb des Strukturbaums gesucht. Bei einem Fund, wird das Array sortiert und an die Ursprungsstelle eingefügt. Dafür sorgt die Methode *set*(*int* index, *JsonNode* value) der Klasse *ArrayNode*, die den Knoten am übergebenen Index ersetzt.

```
1  private void sortArraysRecursively(JsonNode currentRoot) {
2          if (currentRoot.isArray()) {
3              ArrayList<JsonNode> arrayNodes = new ArrayList<JsonNode>();
4              for (JsonNode node : currentRoot) {
5                  if (node.isValueNode()) {
6                    arrayNodes.add(node);
7                  } else if (node.isArray()) {
8                    sortArraysRecursively(node);
9                    arrayNodes.add(node);
10                 } else if (node.isObject()) {
11                   sortArraysRecursively(node);
12                   arrayNodes.add(node);
13                 }
14             }
15             Collections.sort(arrayNodes, new JsonArrayComparator());
16             for (int i = 0; i < arrayNodes.size(); i++) {
17               ((ArrayNode) currentRoot).set(i, arrayNodes.get(i));
18             }
19         } else if (currentRoot.isObject()) {
20             for (JsonNode node : currentRoot) {
21                 if (node.isValueNode()) {
22                   //besitzt keine Kindelemente
23                 } else if (node.isArray()) {
24                   sortArraysRecursively(node);
25                 } else if (node.isObject()) {
26                   sortArraysRecursively(node);
27                 }
28             }
29         }
30 }
```

Quellcode 5: Rekursive Sortierung aller Arrayknoten

Struktureller Vergleich von JSON-Dateien

Das zuvor vorgestellte Konzept für den strukturellen Vergleich baumbasierter Dateiformate soll nun auch für JSON angepasst und angewendet werden. Auf der einen Seite wird es dadurch vereinfacht, dass es in JSON weniger mögliche Knotentypen gibt. Auf der anderen Seite fällt die Konstante weg, dass jegliche Knotentypen immer einen bestimmten Elternknotentyp, wie im Falle von Elementen bei XML, haben. In JSON können, wie in Abb. 10 gezeigt, sowohl Arrays als auch Objekte Einträge eines beliebigen Typs haben, wodurch einige Sonderregeln eingeführt werden müssen. Um einen strukturellen Vergleich für JSON anwenden zu können, müssen auch hier die verglichenen Dateien wohlgeformt sein.

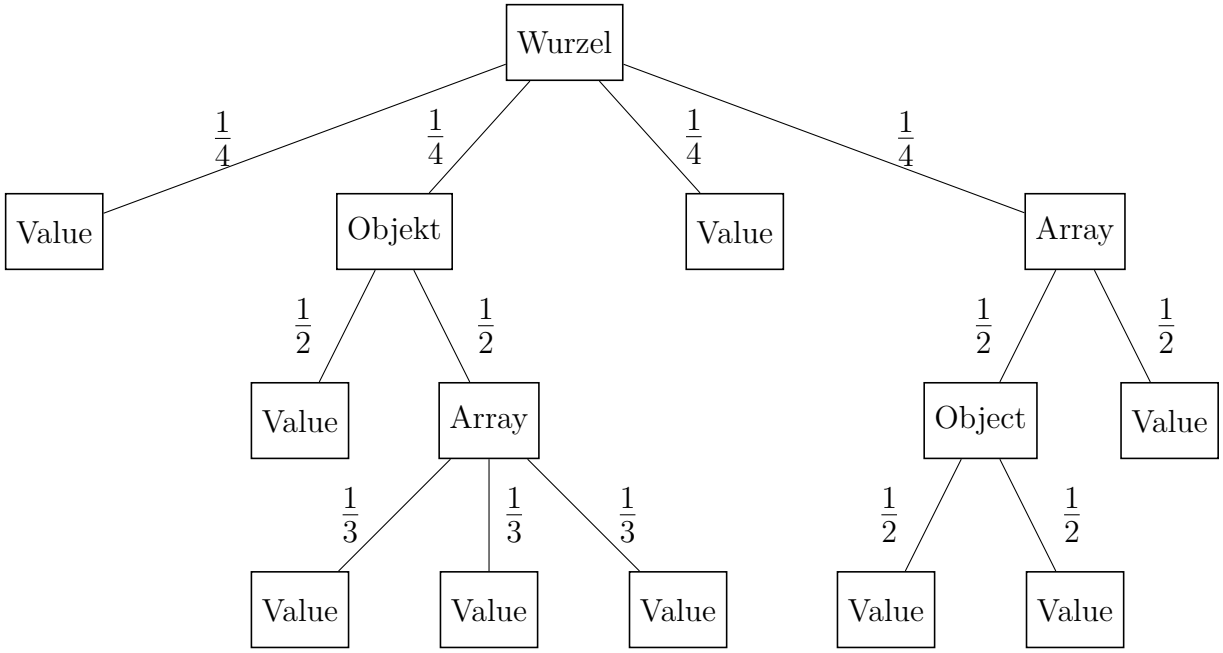

Abbildung 10: Beispielbaum für JSON

Zunächst werden Knoten über ihre Namen, also ihre Keys gematcht. Sind beide gematchen Knoten vom Typ Value, werden sie nach ihrem Inhalt nach Gleichung (2) verglichen. Für die Paarung Value- und Arrayknoten wird im Array nach einem Eintrag mit dem Value gesucht. Da das Ziel eine geringe Fehlerquote ist, wird hierbei nur auf exakte Übereinstimmungen geachtet. Andererseits kann hier auch argumentiert werden, dass für eine höchstmögliche Genauigkeit nur Knoten des gleichen Typs verglichen werden sollten. Diese Entscheidung hängt allerdings von der Struktur der Inputdateien ab und ist deshalb nicht eindeutig, weshalb hier auf die Suche im Array gesetzt wird. Value- und Objektknoten werden wiederum nicht miteinander verglichen, da innerhalb des Objekts jedes Element einen eigenen Key hat und es somit keine Möglichkeit gibt, das Value exakt einem Objektelement zuzuordnen. Die einzige Ausnahme wäre, wenn es innerhalb des Objekts ein Element gäbe, welches den gleichen Key hat wie das Elternobjekt. Dieses Element könnte aber auch einen der drei Knotentypen haben weswegen hier auf einen Vergleich verzichtet wird.

Wenn zwei Arrays miteinander verglichen werden, wird es etwas schwieriger da Arrayelemente weder Keys haben um sie eindeutig zu matchen, noch auf einen bestimmten Elementtyp festgelegt sind. Analog zu den Elementlisten für XML, werden hier zunächst identische Arrayelemente festgestellt und bewertet. Danach folgt die Auswertung der Elementtypen innerhalb der Arrays. Falls Valueknoten innerhalb der Arrays existieren, gelten die gleichen Regeln wie zuvor genannt. Existieren Array-Elemente innerhalb der Arrays, werden diese ebenfalls zunächst durch das Matching ihrer Einträge und dann durch die

Fallunterscheidung verglichen. Einträge die nicht gematcht werden konnten, werden in ihrer Reihenfolge verglichen. Zudem werden Arrays und Objekte aufgrund der fehlenden Keys der Arrays nicht miteinander verglichen.

Sollten zwei Objekte innerhalb von Arrays miteinander verglichen oder durch ihre Keys gematcht werden, findet der Vergleich auf Basis der Kindelemente statt. Diese werden über ihre Keys gematcht und je nach Typ mit einer der zuvor genannten Methoden bewertet.

Die Implementierung basiert im Wesentlichen auf drei Methoden: *compareValues*(), *compareArrays*() und *compareObjects*(). Diese Methoden bekommen als Übergabeparameter die beiden Knoten, die Verglichen werden sollen und liefern ihre Ähnlichkeit zurück. Sie sind so aufgebaut, dass sie sich problemlos gegenseitig aufrufen können, z. B. für den Fall dass beim Vergleich von zwei Arrays, zwei Objekteinträge verglichen werden sollen. Zudem funktionieren die beiden Methoden für Arrays und Objekte auch rekursiv, da diese Knotentypen auch Einträge des gleichen Typs beherbergen können.

Das Matching von Arrayeinträgen funktioniert zwar genauso wie das Matching von XML-Elementlisten, die Implementierung weicht aber dadurch ab, dass identische Knoten nicht mehr durch *null* ersetzt werden können, da dies ein gültiger Typ von Valueknoten ist. Stattdessen wird zu Beginn des Vergleichs ein zufälliger Universally Unique Identifier (UUID) generiert um einen Wert zu erzeugen, dessen Vorkommen innerhalb einer Datei maximal unwahrscheinlich ist. Um gleiche Arrayeinträge zu finden werden sie mittels der *toString*()-Methode als String dargestellt und über die *equals*()-Methode gematcht. Identische Knoten werden dann im Baum durch einen neuen Arrayknoten ersetzt, welcher lediglich einen Eintrag mit der UUID beinhaltet. Nach dem Matching werden diese Knoten dann durch die Methode *clearUUIDFields*() entfernt und für den restlichen Vergleich freigegeben. Dafür werden die beiden zu vergleichenden Knoten *fieldValueRef* bzw. *fieldValueComp* mit den veränderten Arrays überschrieben. Dieses Verfahren ist in Quellcode 6 zu sehen.

```
1         ArrayNode refArray = (ArrayNode) fieldValueRef;
2         ArrayNode compArray = (ArrayNode) fieldValueComp;
3         for (int i = 0; i < refArray.size(); i++) {
4             String refString = refArray.get(i).toString();
5             for (int j = 0; j < compArray.size(); j++) {
6                   String compString = compArray.get(j).toString();
7                   if (compString.equals("[\"" + uuid + "\"]")) {
8                       continue;
9                   }
10                  if (refString.equals(compString)) {
11                      similarity += currentLevelWeight
12                                  * calcLevelWeight(fieldValueRef, fieldValueComp);
13                      refArray.set(i,objectMapper.createArrayNode()
14                                          .add(uuid));
15                      compArray.set(j,objectMapper.createArrayNode()
16                            .add(uuid));
17                      break;
18                  }
19            }
20        }
21        refArray = clearUUIDFields(refArray, uuid);
22        compArray = clearUUIDFields(compArray, uuid);
23        fieldValueRef = refArray;
24        fieldValueComp = compArray;
```

Quellcode 6: Array-Matching für JSON

4.3 Batch-Processing für Vergleiche

Dem Benutzer ist es möglich, beliebig viele Dateien für den Vergleich miteinander auszuwählen. Da die Software in der Version 1.0 für die Vergleiche single-threaded war, konnte dies je nach Dateimengen- und größen zu langen Laufzeiten führen. Die in 2.1.2 beschriebene Levenshtein-Distanz hat in ihrer Berechnung für zwei Strings S_1, S_2 mit der Länge m und n in der verwendeten Implementierung von Apache Commons eine Laufzeit von $\mathcal{O}(m \cdot n)$ [13], was die Laufzeit für lange Zeileneinträge drastisch erhöht.

$$N = \sum_{k=2}^{n} k - 1 = \sum_{k=1}^{n-1} k = \frac{n^2 - n}{2} \text{ , mit } n \in \mathbb{N}_{\geq 2} \tag{4}$$

In Gleichung (4) ist beschrieben, wie viele Vergleiche N für n vom Benutzer ausgewählte Dateien entstehen. Das Minimum an Dateien ist hierbei 2, da es in dem Kontext der Software keinen Mehrwert hat eine Datei mit sich selbst zu vergleichen. Was jedoch auffällt,

ist wie stark die Anzahl der zu berechnenden Vergleiche mit der Anzahl der importierten Dateien wächst.

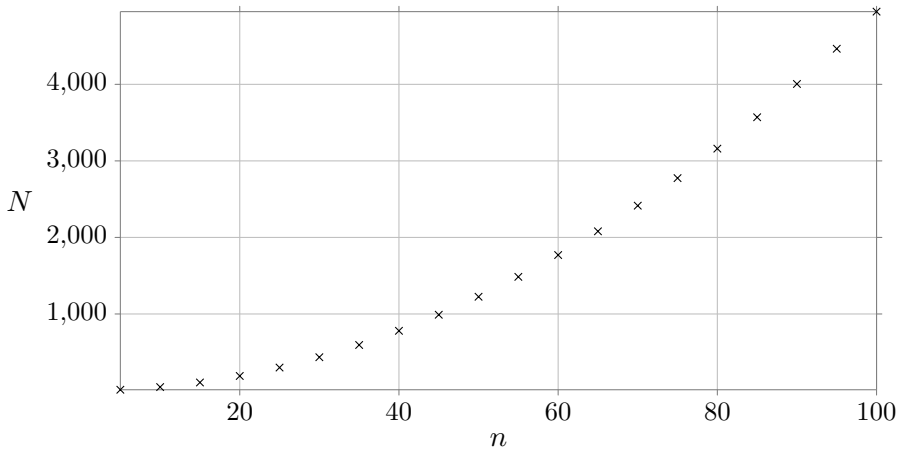

Abbildung 11: Skalierung der Vergleichsanzahl

Dies ist für relativ geringe Anzahlen an Dateien in Abb. 11 dargestellt. Dadurch entsteht die Frage, wie die Laufzeit der Vergleiche reduziert werden kann, ohne die Genauigkeit der Ergebnisse zu verschlechtern.

Eine Möglichkeit um die Laufzeit drastisch zu verringern, wäre eine Parallelisierung der Vergleiche. So bekäme jeder Prozessorkern (oder jeder Thread bei Mutlithread-Prozessoren) einen Batch (Stapel) an Vergleichen zugewiesen, den er bearbeiten kann. Zusätzlich sollte sicher gestellt werden, dass jeder Thread ungefähr gleich stark belastet wird um die CPU möglichst gleichmäßig auszulasten.

Für die Umsetzung wird zunächst die Java-Klasse *IComparison* aus der *Entity*-Komponente erweitert. Objekte dieser Klasse speichern Referenzen auf die verglichenen Dateien und den Ähnlichkeitswert eines Vergleichs. Dazu kommt nun ein Feld für die ID und ein Feld für das Gewicht des Vergleichs. Die ID ist notwendig, da die Objekte nach ihrem Gewicht auf die verfügbaren Threads verteilt werden und nach dem Vergleich aller Dateien wieder in die ursprünglichen Reihenfolge zusammengefügt werden müssen. Das Gewicht wird hierbei durch die Gesamtzahl der Zeichen dargestellt.

Zwar beachtet dieses Gewicht nicht, ob die Zeichen gleichmäßig auf die Zeilen verteilt sind, jedoch approximiert es die mittlere Berechnungsdauer. Sollten Zeilen mit besonders vielen Zeichen in den verglichenen Dateien existieren, besteht die Möglichkeit für den Benutzer eine Obergrenze für die Edit-Distance anzugeben, was die Laufzeit in diesem Fall deutlich verringert.

Um die gewichteten Vergleiche nun auf die verfügbaren Threads zu verteilen, werden sie zunächst absteigend nach ihrem Gewicht sortiert. Danach wird mittels der Codezeile

```
Runtime.getRuntime().availableProcessors();
```

die Anzahl der Threads und damit die Anzahl der zu befüllenden Batches ermittelt. Um die Vergleiche auf die Batches zu verteilen wird das *Round-Robin*-Verfahren verwendet, bei dem alle Batches nacheinander einen Vergleich zugewiesen bekommen bis alle Vergleiche verteilt sind.

Um nun jeden Batch auf einem eigenen Thread zu berechnen, wird sobald der Benutzer den Vergleich startet im jeweiligen Event-Handler ein SwingWorker erstellt. Innerhalb dessen *doInBackground()*-Methode wird wiederum ein *ExecutorService* angelegt. Ein ExecutorService liefert Methoden, mit denen Threads und Threadgruppen synchronisiert gestartet bzw. gestoppt werden können [14]. Für das Interface ExecutorService existieren mehrere Implementierungen wobei hier die Implementierung *newFixedThreadPool*(*int* nThreads) verwendet wird. Der Übergabeparameter *nThreads* steht dabei für die Anzahl an Threads, die innerhalb des ExecutorService verwendet werden können. Diese Anzahl wird ebenfalls auf die Anzahl der verfügbaren Prozessorthreads festgesetzt um jedem Thread genau einen Batch zuzuweisen.

Die *shutdown()*-Funktion des ExecutorServices lässt alle zuvor erstellten Threads zu Ende laufen und baut diese anschließend ab. In der *done()*-Methode des SwingWorkers werden dann alle Batches wieder in eine einzige, nach IDs sortierte, Liste vereinigt und für die Anzeige freigegeben.

4.4 Verbesserung der UI und der Bedienbarkeit

Neben der Funktionalität, spielen das Aussehen und die Bedienbarkeit einer Software eine große Rolle. Eine Software sollte möglich intuitiv bedienbar sein und dem Benutzer bei der Bedienung durch Hinweise und Erklärungen Hilfestellung leisten können.

4.4.1 Kosmetische Änderungen

Ob eine Software mit Hilfe von Java Swing erstellt wurde, kann häufig am Aussehen festgestellt werden. Das liegt am von Swing als Standard eingestellten *Look-and-Feel (L&F)*, dem sog. „Metal“-Theme. Das L&F verändert, wie der Name schon sagt, das Aussehen und Verhalten von Swing-Komponenten. Der Vorteil des Metal-L&Fs ist, dass es cross-platform ist und daher unabhängig vom zugrundeliegenden Betriebssystem eingesetzt werden kann. Der Nachteil ist das teils veraltete Aussehen und die stellenweise schwierige Bedienung. Als Beispiel ist in Abb. 12 eine Dateiauswahl mit dem Metal-L&F

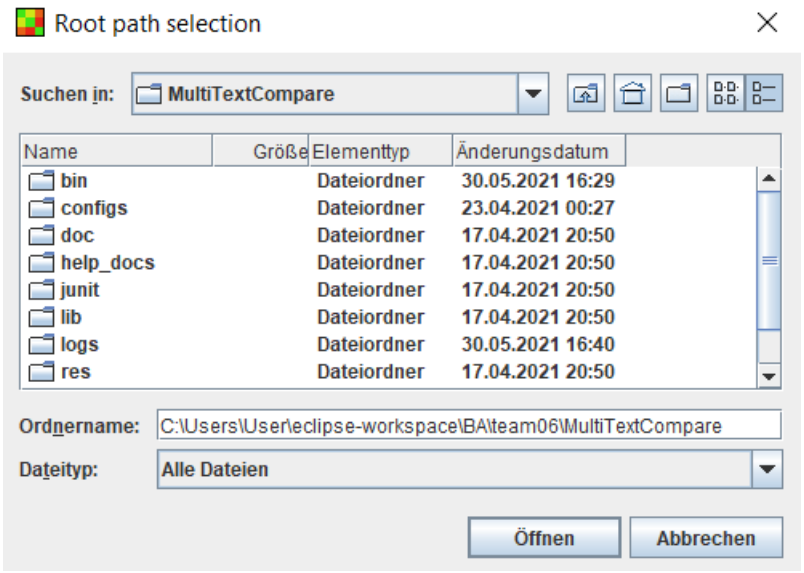

Abbildung 12: Dateiauswahl unter Metal-L&F

dargestellt.

Als Alternative existieren unter Swing noch weitere, vom Betriebssystem abhängige L&Fs. Diese können mit der Zeile

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

automatisch beim Programmstart geladen und angewandt werden. Zwar ist MultiText-Compare zunächst nur als Windows-Software geplant, allerdings wurde sie bisher immer so entwickelt, dass sie auch auf anderen Betriebssystemen lauffähig ist. Deswegen wird immer das dem Betriebssystem zugehörige L&F geladen. Die in Abb. 13 gezeigte Dateiauswahl wirkt dank dem Windows-L&F nun verständlicher und etwas moderner. Unter Linux würde, sofern es installiert ist, bspw. dann das GTK+ L&F geladen werden [15].

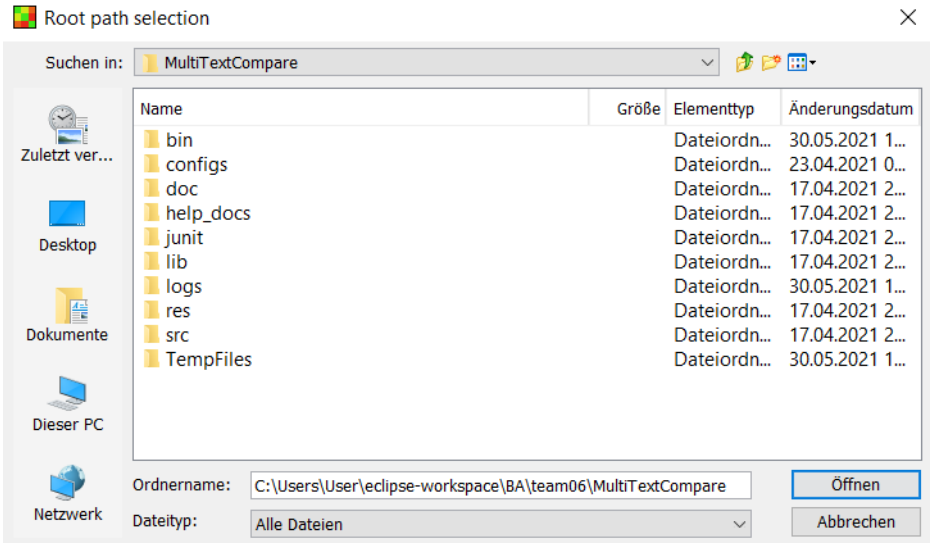

Abbildung 13: Dateiauswahl unter Windows-L&F

4.4.2 Bedienung der Ähnlichkeitsmatrix

Eine der Anforderungen ist, dass die Software einfach zu bedienen sein soll, jedoch ist die Auswahl der Dateien zur Anzeige der Diff in Version 1.0 in dieser Hinsicht noch nicht optimal. Ein Klick auf eine Zelle innerhalb der Ähnlichkeitsmatrix öffnete eine Diff-Anzeige für die beiden Dateien in der jeweiligen Zeile bzw. Spalte. Um drei Dateien auszuwählen, musste der Benutzer die Spaltenköpfe der Matrix anklicken, was mehrere Probleme beinhaltet. Zum einen musste für jede Datei einzeln die Ähnlichkeit zu anderen Dateien nachgeschaut werden, da die Matrix nicht automatisch die zugehörigen Zellen der ausgewählen Spaltenköpfe gesondert anzeigt. Zum anderen war es nicht möglich nach dem Fund zweier ähnlicher Dateien nach einer dritten ähnlichen Datei zu suchen ohne die ersten beiden immer wieder auszuwählen. Aus diesen Gründen soll das Auswahlsystem komplett überarbeitet werden.

Zunächst sei grundlegend zu erwähnen, dass die Ähnlichkeitsmatrix technisch gesehen eigentlich keine Matrix ist, da diese nicht als Swing-Komponenten existieren. Stattdessen wird eine JTable verwendet, die als erste Spalte Zellen besitzt, die das Aussehen und Verhalten der Spaltenköpfe aus der JTable imitieren. Dies wird durch eine angepasste Version der Klasse *RowNumberTable* von Rob Carrick umgesetzt [16]. Jegliche Funktionen um Klicks des Benutzers innerhalb der Matrix zu bearbeiten basieren also auf den vorgegebenen Funktionen der Klasse JTable. Das Event-Handling dieser Klicks passiert in einer zusätzlichen Handler-Klasse namens *MouseAdapterMatrix*. Diese erbt von der Swing-nativen Klasse *MouseAdapter* und implementiert die Handler-Methode *mousePressed*(), welche das zu handlende Event als Übergabeparameter erhält.

Praktischerweise lassen sich aus diesem Event die Zeile und Spalte des Klicks auslesen.

Da die RowNumberTable die JTable der Matrix einschließt und kein tatsächlicher Teil der klickbaren Matrix ist, verändern sich auch die Indices nicht.

Um eine Diff-Anzeige zu öffnen soll nun ein Doppelklick notwendig sein, da ein einzelner Klick einfacher zu Fehlklicks führen kann und diese je nach Dateigröße auch eine Wartezeit nach sich ziehen können. Um einen Doppelklick zu erkennen lässt sich die Methode *getClickCount*() auf dem Eventobjekt ausführen. Um drei Dateien auszuwählen, soll nun ein Zwischenschritt eingeführt werden. Zunächst soll es möglich sein eine Zelle als Referenz festzulegen. Dabei werden alle Zellen, die nicht innerhalb der gleichen Zeile und Spalte oder auf der Hauptdiagonale liegen, ausgegraut und sollen damit nicht klickbar sein. Dies führt dazu, dass jeder mögliche Vergleich übersichtlich dargestellt wird und immer genau drei Dateien ausgewählt werden können. Diese Auswahl geschieht dann mit einem weiteren Klick auf eine der farbigen Zelle und führt nicht dazu, dass die Referenzzelle zurückgesetzt wird. Dafür muss der Renderer der JTable wissen, welche Zelle gerade die Referenzzelle ist.

Für solche Fälle existiert eine zentrale Klasse *Management*, die nach dem Singleton-Entwurfsmuster Objekte für sämtliche Klassen instanziiert, von denen genau eine Instanz existieren soll. Zudem können dort Variablen gespeicht werden, die an verschiedenen Stellen im Code gebraucht werden und ebenfalls nur einmal existieren sollten. Beispiele dafür sind die aktuelle Konfiguration der Software oder die eben erwähnte aktuelle Referenzzelle. Für den sog. *TableCellRenderer* wurde die *prepareRenderer*()-Methode so überschrieben, dass sie prüft ob ein boolsches Flag durch die MouseAdapterMatrix im Management gesetzt wurde und dann entweder die Matrix komplett oder nur die Zellen nach den o. g. Kriterien koloriert. Das Resultat ist in Abb. 14 zu sehen.

4.4.3 Konfiguration

Im Laufe dieser Arbeit wurden bereits diverse konfigurierbare Parameter besprochen. Darunter fällt u. a. die Auswahl der Vergleichsmodi, die Funktionsweise des Matchings, Parameter zum Sortieren oder Normieren der Inputdateien. Zwar erhöht eine größere Anzahl an Konfigurationsparametern die Komplexität der Software für Benutzer und Entwickler, zeitgleich liefert sie aber auch die Möglichkeit die allgemeinen Funktion des Textvergleichs für die zu vergleichenden Dateien anzupassen. Schließlich sind die vorgestellten Vergleichsalgorithmen lediglich Approximationen der Ähnlichkeit von Dateien nach bestimmten Kritierien und bilden daher keine objektive Bewertung für alle möglichen Arten von Wissensrepräsentation innerhalb der unterstützten Dateiformate. Falls der Benutzer Dateien vergleichen möchte, die nicht ideal mit den aktuellen Parametern verglichen werden würden, wäre es sinnvoll mehrere Konfigurationsdateien erstellen zu können und diese einfach

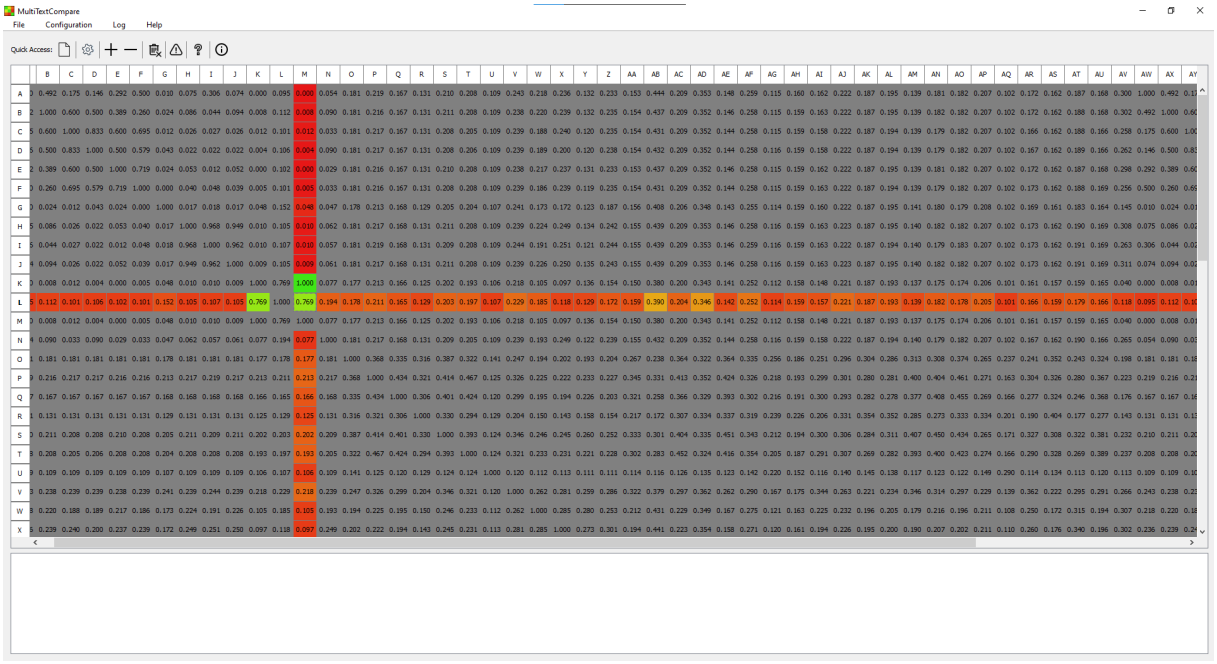

Abbildung 14: Ausgrauen der Matrix

auszutauschen anstatt immer wieder die aktuelle Konfiguration zurückzusetzen oder zu überschreiben.

MutliTextCompare erstellt, falls noch nicht vorhanden, eine Standardkonfiguration basierend auf Key-Value Paaren für jeden Parameter. Um eine alternative Konfiguration zu verwenden, wird in der Standardkonfiguration ein weiteres Feld für den Pfad der aktuellen Konfigurationsdatei hinterlegt. Ist dieses nicht leer und die Datei existiert, wird diese Konfiguration verwendet, ansonsten wird die Standardkonfiguration verwendet. Der Benutzer hat dadurch die Möglichkeit die aktuelle Konfiguration über die UI zu überschreiben oder als neue Datei unter Angabe des Dateinamens zu erstellen.

4.4.4 Persistenz von Vergleichen

Trotz der bisher durchgeführten Optimierungen für die Vergleiche und das Matching, können Vergleiche mit vielen, großen Dateien durchaus noch einige Minuten dauern. Solche langen Wartezeiten können sehr unangenehm sein, besonders wenn die gleichen Vergleichsergebnisse zu einem späteren Zeitpunkt wieder gebraucht werden. Es ist allerdings in Java bspw. durch Serialisierung möglich, bestimmte Programmteile persistent zu speichern.

Nach Erstellung und Anzeige der Ähnlichkeitsmatrix wird die Diff für Dateien erst berechnet, wenn sie vom Benutzer ausgewählt wurden. Falls vergleichsbezogene Parameter geändert werden, muss die Matrix neu berechnet werden, um diese zu sehen. Das heißt, dass es ausreicht genau den Programmzustand wiederherzustellen, bei dem die Matrix

gerade erzeugt wurde. Innerhalb des Codes wird dafür die Liste mit den berechneten Vergleichen der Matrix benötigt. Zusätzlich muss die Map der *FileImporter*-Komponente gespeichert werden, die die erstellten temporären Dateien auf die Originaldateien mappt. Letztlich wird eine geordnete Liste der ausgewählten Dateien bzw. ihrer Dateipfade mit zugehörigem Index gespeichtert. Zwar könnte diese Liste aus der Map wiederhergestellt werden, allerdings müssten dafür einige Methoden zentralisiert oder publik gemacht werden, was die Abhängigkeiten der Klassen und damit die Komplexität der Code-Base weiter erhöht.

Für die persistente Speicherung über Serialisierung muss sichergestellt werden, dass alle zu speichernden Objekte das Interface *Serializable* implementieren. Dies muss nur für die selbst erstellte Matrix bzw. die Klasse *ICompareImpl* nachgeholt werden. Um die Objekte zu serialisieren wird die native Java Klasse *ObjectOutputStream* verwendet und mit einem *FileOutputStream* in eine Datei geschrieben. Der Dateiname ist vom Benutzer wählbar und besitzt immer die Dateiendung *.mtc*. Damit die Objekte korrekt ein- und ausgelesen werden können, müssen diese in der gleichen Reihenfolge geschrieben und gelesen werden.

Da die Map des FileImporters nur Referenzen auf die Dateien enthält, muss noch sichergestellt werden dass die temporären Dateien eines Vergleichs mitgespeichert werden um die an den Dateien vorgenommenen Normierungen und Sortierungen korrekt anzuzeigen. Dafür wird für jeden gespeicherten Vergleich ein Verzeichnis erstellt, indem sowohl die mtc-Datei mit den gespeicherten Objekten, als auch ein Verzeichnis mit den zugehörigen temporären Dateien vorliegt. Diese temporären Dateien werden dann in das *TempFiles*-Verzeichnis der Software kopiert sobald der jeweilige Vergleich geladen wurde.

4.4.5 Benutzerfeedback und Logging

Beim Öffnen von MultiTextCompare erscheint in der unteren Hälfte des Panels ein Log. Dieser dient dazu, dem Benutzer Feedback zu seinen Aktionen zu liefern, bspw. wenn eine Referenzzelle bei der Auswahl von Dateien für die Diff-Anzeige selektiert wird. Zusätzlich werden so Fehler beim Parsen oder interne Fehler während der Laufzeit an den Benutzer weitergegeben. Bisher gab es jedoch keine Unterscheidung nach der Schwere des Fehlers oder der Wichtigkeit der Nachricht, was durch eine interne Logging-Komponente geändert werden soll. Wie bereits im Kapitel zu Architektur in 3.3 gezeigt wurde, funktioniert diese Logging-Komponente außerhalb der eigentlichen 3-Schichten-Architektur.

Es existiert zwar im Paket *java.util.logging* bereits eine native Logging-Lösung, aber sie ist hier fast schon zu umfangreich und benötige mehr Arbeit um sie in die Code-Base einzupflegen. In der eigenen Logger-Komponenete werden nur 3 Log-Level unterschieden:

Info Beinhaltet das Feedback zur User-Interaktion und Benachrichtigungen über Berechnungen wie den Beginn des Vergleichs und das Ende mit der Berechnungsdauer. Hier handelt es sich um optionale Ausgaben, die besonders dann signifikant sind, wenn eine Berechnung länger dauert oder sich der Benutzer bei der Bedienung der Software noch nicht sicher ist. Diese Ausgaben werden als regulärer Text im Log hinterlegt.

Warning Bei diesem Level werden Fehler aufgeführt die entweder durch falsche Benutzung der Software auftreten oder durch fehlerhafte Inputdateien. Ein Beispiel für falsche Benutzung wäre z. B. der Versuch einen Vergleich persistent zu speichern bevor überhaupt eine Matrix erzeugt wurde. Warnings führen nicht zu Fehlverhalten und werden mit roter Schrift auf gelbem Hintergrund im Log angezeigt.

Error Errors sind schwerwiegende Fehler, die dazu führen dass das Programm nicht ordnungsgemäß funktionieren kann. Der wahrscheinlichste Fehler auf diesem Level ist eine *FileNotFoundException*, die geworfen wird falls Dateien unerwartet gelöscht, verschoben oder umbenannt wurden. Diese Fehler werden im Log mit rotem Hintergrund und weißer Schrift angezeigt.

Zusätzlich zur Anzeige im Log werden diese Ausgaben auch persistent in Logdateien gespeichert. Um riesige Logdateien und damit einen Verlust der Übersicht zu vermeiden, werden die Logs nach Datum getrennt. Für jede Logausgabe wird dafür geprüft ob eine aktuelle Logdatei existiert, ansonsten wird eine neue Datei erstellt. Zwar werden alle Logausgaben unabhängig vom Level in die Logdateien geschrieben, dem Benutzer ist es aber möglich einzelne Level für die Anzeige im Log ein- und auszuschalten.

5 Test und Auswertung

Das letzte Kapitel des Hauptteils soll nun die wichtigsten Verbesserungen aus Kapitel 4 im Anwendungskontext zeigen und evaluieren.

5.1 Übersicht der Testsysteme

Zunächst sollen die Testsysteme vorgestellt werden. Es geht bei der Analyse der Laufzeit hierbei nicht darum, die Hardwarekonfigurationen untereinander zu vergleichen, sondern um die Darstellung der durch das Batch-Processing eingeführten Skalierbarkeit. PC1 ist dabei der schnellste PC, gefolgt von PC2 und PC3, wobei letzterer der Konfiguration der Laborrechner am meisten ähnelt. Das in der Tabelle angegebene Jahr bezieht sich auf das Einführungsdatum der jeweiligen CPU, die den größten Einfluss auf die Berechnungsdauer hat.

Auf allen Computern ist Windows 10 in einer 64 Bit Version installiert. Außerdem wird Java sowohl beim JDK, als auch bei der JRE in der 32 Bit Variante unter Version 1.7.0_75 verwendet. Desweiteren wird MultiTextCompare immer über Eclipse selbst und nicht über eine ausführbare JAR gestartet. Um eine vollständige Auslastung der CPU sicherzustellen werden die Run Commands für das Programm so angepasst, dass bis zu 1 GB RAM verwendet werden kann. Zusätzlich werden die Vergleiche direkt nach dem Start der PCs ausgeführt ohne dass weitere Programme geöffnet werden.

Hardwarekomponente	PC1	PC2	PC3
CPU	Intel i9 10850k (10c/20t)	Intel i7 7700HQ (4c/8t)	Intel i5 4570 (4c/4t)
RAM	32GB DDR4 @ 2933 MHz	8GB DDR4 @ 2400 MHz	16GB DDR3 @ 1600 MHz
Festplatte	NVMe SSD	NVMe SSD	SATA SSD
Jahr [17]	2020	2017	2013

Tabelle 1: Hardwarekonfigurationen der Testsysteme

5.2 Laufzeitoptimierung durch Multithreading

Als erstes soll das Laufzeitverhalten beim Aufbau der Matrix untersucht werden. Dadurch dass MultiTextCompare aber für beliebige textbasierte Dateien konzipiert ist, lässt sich hier allerdings kein „typischer“ Fall herstellen, da die Laufzeit sehr stark von den verglichenen Dateien und den Konfigurationsparametern abhängt. Deshalb soll hier nur erfasst werden, wie groß die Zeitersparnis im *best case* ist.

Für den ersten Durchlauf werden 80 reguläre Textdateien erstellt, deren Inhalt identisch ist. Beim Inhalt wird der Platzhaltertext *Lorem ipsum* verwendet, welcher über den Line Compare-Vergleichsmodus ohne Zeilenmatching verglichen wird. Die Dateien sind 196KB groß, besitzen 611 Zeilen und die Zeichen pro Zeile variieren ca. zwischen 100 und 1700. Gemessen wird die Dauer des Vergleichs mit und ohne Multithreading und der Overhead, der durch das Verteilen und Zusammenfügen der gewichteten Vergleiche zusätzlich aufkommt. Dafür wird im Code mittels

```
System.nanoTime()
```

der Start- bzw. Endzeitpunkt der Messung festgehalten und die Differenz der beiden Werte als Ergebnis notiert. Da IO-Zugriffe und das Scheduling des Betriebssystems Varianzen in die Laufzeit einbringen, werden die Ergebnisse bei einer Dauer von unter fünf Minuten über fünf Durchläufe gemittelt.

Testlauf	PC1	PC2	PC3
Ohne Multithreading	43min 11s	66min 29s	68min 37s
Mit Multithreading	4min 26s	20min 49s	18min 27s
Overhead	2,355s	6,241s	2,440s

Tabelle 2: Messergebnisse für Multithreading

Die Messergebnisse sind in Tabelle 2 dargestellt und zeigen wie erwartet, dass die Vergleiche ohne Multithreading auf PC1 am schnellsten und auf PC3 am langsamsten durchlaufen. Mit Multithreading ist PC3 aber schneller als PC2, was wahrscheinlich der besseren Kühlleistung des Systems geschuldet ist. Der Unterschied zwischen Vergleichen die multi- bzw. single-threaded sind ist zwar erheblich, allerdings korrespondiert dieser nicht unbedingt genau mit der Anzahl der verfügbaren Prozessorkerne. Bei PC1 mit zehn Prozessorkernen, ist eine Verbesserung um ungefähr den gleichen Faktor erkennbar. PC2 hat mit 4 physischen Kernen aber nur knapp eine Verbesserung um den Faktor 3,3. Da sich die einzelnen Prozessoren auch in der Leistung, z.B. der Taktrate oder der Instruktionen pro Takt, unterscheiden, sind erhebliche Unterschiede ohne Multithreading erkennbar.

Auch der entstandene Overhead durch die Verteilung und das Zusammenführen der Batches variiert zwischen den Systemen merkbar. Dieser Wert ist aber wenig signifikant wenn die mögliche Zeitersparnis durch das Multithreading betrachtet wird. Am meisten Zeit verbraucht dabei die Gewichtung der Vergleiche, für die alle Dateien der Vergleiche eingelesen werden müssen. Faktoren für die Abweichungen zwischen den Systemen können hier auch die Leistung der CPU, der Festplatten und das Scheduling des Betriebssystems sein, da keins der Systeme identische Komponenten beinhaltet.

Mit Multithreading entsteht also eine große Zeitersparnis, wenn die Systeme die nötige Hardware bieten. Hier gibt es aber auch noch weitere Möglichkeiten um die Wartezeit für den Benutzer zu verringern. Für zwei Zeilen mit vielen Zeichen entsteht bspw. das Problem, dass die Berechnung der Levenshtein-Distanz dort relativ teuer wird. Dort könnte es sinnvoll sein, die Strings in kleinere Sequenzen zu zerlegen und diese zu vergleichen. Ansonsten kann aber auch konfiguriert werden, dass Zeilen nur bis zu einer bestimmten Edit Distance verglichen werden und bei einer Überschreitung mit der Ähnlichkeit 0 bewertet werden.

5.3 Strukturbasierte Vergleichsmodi

Die beiden neu eingeführten Vergleichsmodi für XML und JSON können für bestimmte Anwendungsfälle die Ähnlichkeitsbewertung von Dateien deutlich hinsichtlich ihrer Präzision verbessern. Da Knoten nur verglichen werden falls ihre eindeutigen Identifier übereinstimmen, sind false positives deutlich unwahrscheinlicher als bspw. beim Modus Line Compare, bei dem nicht gematchte Zeilen trotzdem miteinander verglichen werden und eine zufällige, wenn auch geringe, Ähnlichkeit haben können. Dadurch werden weniger, aber dafür tatsächlich strukturell ähnlichere Dateien in der Matrix als Ähnlich dargestellt und vor einer möglichen Auswahl der Diff gefiltert. Ein weiterer Vorteil kann im Bereich der Laufzeit entstehen, da ganze Teilbäume aufgrund von unterschiedlichen Bezeichnern evtl. nicht verglichen werden müssen.

Im Folgenden soll der strukturbasierte Vergleich für Dateien des Typs JSON angewandt und mit den Ergebnissen des Modus Line Compare verglichen werden. Dafür werden drei Beispielkonfigurationen einer fiktiven Rest-Schnittstelle verwendet, bei denen es sich um unterschiedliche Versionen handelt. Diese Konfigurationsdateien sind in Abb.15 dargestellt und zwecks Übersicht relativ einfach gehalten. Datei V0 enthält die wenigsten Funktionalitäten, denn es fehlt im Vergleich zu V1 bspw. die Authentifizierung des zugreifenden Akteurs oder die Validierung der übergebenen JSON- oder XML-Inhalte. Der Unterschied von V1 zu V2 ist dafür deutlich kleiner, es ändert sich lediglich der Host von einer IP-Adresse zu einem tatsächlichen Domain-Namen und es kommt ein weiteres

Dateiformat in Form von YAML für schreibende Anfragen dazu.

```
{
    "api_version" : "v0",
    "endpoint" : {
        "host" : "192.168.0.165",
        "port": 5000
    },
    "accept_format": [
        "xml",
        "json"
    ],
    "request_types": [
        "GET",
        "POST",
        "PUT",
        "DELETE"
    ]
}
```

(a) Konfiguration V0

```
{
    "api_version" : "v1",
    "endpoint" : {
        "host" : "201.177.6.52",
        "port": 5000
    },
    "accept_format": [
        "xml",
        "json"
    ],
    "authentication": {
        "user": "$username",
        "pass": "$password"
    },
    "enable_validation": true,
    "request_types": [
        "GET",
        "POST",
        "PUT",
        "DELETE"
    ]
}
```

(b) Konfiguration V1

```
{
  "api_version": "v2",
  "endpoint": {
    "host": "test@server.de",
    "port": 5000
  },
  "accept_format": ["xml", "json", "yaml"],
  "authentication": {
    "user": "$username",
    "pass": "$password"
  },
  "enable_validation": true,
  "request_types": [
        "GET",
        "POST",
        "PUT",
        "DELETE"
    ]
}
```

(c) Konfiguration V2

Abbildung 15: Testdateien für strukturbasierten Vergleich

Beim strukturellen Vergleich werden nun zunächst die entsprechenden Dokumentbäume aufgestellt. Die maximale Anzahl an Knoten auf Ebene 1 beträgt sechs. Jeder dieser Knoten geht also für seinen Inhalt mit einem Gewicht von $\frac{1}{6}$ in die Gesamtähnlichkeit ein. Beim zeilenbasierten Vergleich ist hingegen die maximale Anzahl an Zeilen für das Gewicht entscheidend. Aus Platzgründen wurde die Darstellung der *request_types* in Abb. 15 mit einem Array-Eintrag pro Zeile gewählt, der JSON-Parser von Jackson schreibt allerdings das gesamte Array in eine Zeile weswegen sich eine maximale Zeilenanzahl von 14 und damit ein Gewicht von $\frac{1}{14}$ pro Zeile ergibt. Das heißt hier für die Bewertung, dass beim Vergleich von V0 und V1 die fehlende Authentifizierung im strukturellen Vergleich $\frac{1}{6}$ und im zeilenbasierten Vergleich $\frac{4}{14}$ der Gesamtähnlichkeit kostet. Dieser Effekt entsteht, da der zeilenbasierte Modus nicht auf die Struktur von JSON bzw. auch auf die Darstellung von JSON-Dateien durch den Parser achten kann. Die Ergebnisse aller Vergleiche sind in Tabelle 3 dargestellt.

Ähnlichkeit	V0	V1	V2
V0	1,000	0,526	0,444
V1	0,526	1,000	0,784
V2	0,444	0,784	1,000

(a) struktureller Vergleich

Ähnlichkeit	V0	V1	V2
V0	1,000	0,618	0,594
V1	0,618	1,000	0,954
V2	0,594	0,954	1,000

(b) zeilenbasierter Vergleich

Tabelle 3: Vergleichsergebnisse für die Dateien aus Abb. 15

Aus den Ergebnissen wird deutlich, dass die Vergleichsmodi die Ähnlichkeit zwar unterschiedlich, dafür aber ohne Widersprüche bewerten. Der Vergleich von V1 und V2 ist bei beiden Modi der ähnlichste, es folgen V1 und V0 und zuletzt kommen V0 und V2. Letzteres ist dadurch erklärbar, dass in V0 und V1 im Feld *host* noch IP-Adressen stehen, die durch die Trennpunkte und die in beiden Adressen vorkommenden Zahlen eine höhere Ähnlichkeit haben als eine Adresse und ein Domain-Name. Der Hauptunterschied liegt also weniger in der Richtung der Bewertung, also ob der Wert näher an 0 oder näher an 1 ist, sondern in den absoluten Werten. Besonders die hohe Ähnlichkeit beim Vergleich von V1 und V2 ist beim zeilenbasierten Vergleich dadurch entstanden, dass die identischen Attributschlüssel eine Grundähnlichkeit für die Zeilen herstellen in denen sie vorkommen. Ebenfalls erhöht die identische Struktur die Ähnlichkeit, da Markup wie Klammern und Kommata in den gleichen Zeilen liegen und somit die Ähnlichkeit um bis zu $\frac{1}{14}$ anheben. Hier kommen also die Vorteile des strukturbasierten Modus zum tragen, bei dem Markup nicht für die Ähnlichkeit berücksichtigt wird und ausschließlich der Inhalt der Dateien bewertet wird.

In Tabelle 4 sind die Abzüge in der Ähnlichkeit für den Vergleich von V1 zu V2 dargestellt. Drei der Key-Value-Paare sind identisch und werden deshalb nicht aufgelistet. Für den

Attributschlüssel	Abzug
api_version	$\frac{1}{6} \cdot \frac{1}{2} = \frac{1}{12}$
endpoint	$\frac{1}{6} \cdot 0,464 \approx 0,077$
accept_format	$\frac{1}{6} \cdot \frac{1}{3} = \frac{1}{18}$

Tabelle 4: Bewertung der inhaltlichen Unterschiede für V1 und V2

Schlüssel *api_version* existieren zwei Zeichen und eins davon unterscheidet sich, wodurch ein erster Abzug von $\frac{1}{12}$ entsteht. Das Objekt *endpoint* hat ein identisches Paar für den Port, nur der Hostname unterscheidet sich. Da die Hostnamen mit dem Punkt nur ein gemeinsames Zeichen haben, kommt hier ein großer Unterschied zu stande. Die 0,464 aus der Tabelle wird durch

$$\frac{1}{2} - \left(\frac{1}{2} \cdot \left(1 - \frac{13}{14}\right)\right) = 0,464$$

errechnet und beschreibt den Unterschied der *endpoint*-Objekte zwischen den beiden Dateien. In der Formel wird letztendlich nur die gewichtete Ähnlichkeit der Hostnamen von der Ähnlichkeit der Ports subtrahiert. Im Array *accept_format* fehlt in V1 noch der Eintrag für YAML und führt zu einem Abzug von $\frac{1}{3}$ im Kontext des Feldes bzw. $\frac{1}{18}$ im Kontext der Gesamtähnlichkeit. Werden nun alle Abzüge von der maximalen Ähnlichkeit 1 subtrahiert, entsteht der Wert aus Tabelle 3(a) für den Vergleich von Datei V1 und V2.

Es sei allerdings noch abschließend zu erwähnen, dass die hier gezeigten Dateien nur einen Anwendungsfall abdecken. Die Anwendung ist zwar nicht für einen bestimmten Anwendungsfall konzipiert, es kann im Rahmen dieser Arbeit aber nicht jeder mögliche Fall dargestellt und analysiert werden.

6 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine bereits existierende Software zum Vergleich von beliebig vielen textbasierten Dateien überarbeitet und erweitert. Besonders im Fokus standen die Aspekte der Genauigkeit von Algorithmen, der Laufzeit, sowie der Benutzbarkeit und Verständlichkeit der Anwendung. Dafür wurde zunächst das Zeilenmatching in der bestehenden Funktionsweise untersucht, wobei nennenswerte funktionale und zeitliche Verbesserungen bspw. durch das neue *Best Match*-Verfahren eingeführt werden konnten. Ähnliches galt auch für den zeilenweisen Textvergleich, dessen Funktionsweise durch die Integration der Matching-Komponente hinsichtlich der Genauigkeit verbessert wurden. Zusätzlich profitiert der Vergleichsmodus nun von den zuvor genannten Verbesserungen des Zeilenmatchings.

Ein weiterer großer Teil der Arbeit bestand daraus, zwei neue Vergleichsalgorithmen für XML und JSON zu konzipieren und zu implementieren. Diese arbeiten primär auf einer strukturellen Basis mit Hilfe von Dokumentbäumen und sind in der Lage Markup zu ignorieren und daher strikt inhaltlich zu vergleichen. Da die Algorithmen Elemente größtenteils anhand von eindeutigen Identifiern matchen und vergleichen, sind sowohl semantisch genauere als auch insgesamt schnellere Vergleiche möglich. Letzteres entsteht dadurch, dass ganze Teilbäume vom Vergleich ausgeschlossen werden können, wenn der zugehörige Identifier nur in einer Datei existiert. Zusätzlich eliminieren die Algorithmen durch dieses Vorgehen die Möglichkeit von zufälligen, falschen Bewertungen.

In Hinblick auf die Laufzeit wurde ein Verfahren entwickelt, womit die Anwendung verfügbare Hardware effektiv nutzt. Vergleiche werden nun gleichmäßig auf verfügbare CPU-Threads verteilt, wodurch für jeden Vergleichsmodus je nach Hardwarekonfiguration proportionale Verbesserungen messbar sind. Sowohl die Laufzeitverbesserungen als auch die Funktionsweise der strukturbasierten Algorithmen wurden zudem in Kapitel 5 anhand von Beispieldateien präsentiert.

Auch das User Interface erhielt zahlreiche Änderungen, die sowohl kosmetisch als auch funktional sind. Neben einer Änderung des Java-spezifischen *Look-and-Feels*, welches nun abhängig vom unterliegenden Betriebssystem ist, wurden weitere Funktionen erstellt um die Benutzung der Software für den Benutzer angenehmer zu machen. Die Auswahl von Dateien für eine graphische Anzeige der Unterschiede gibt dem Benutzer nun durch das Ausgrauen der Matrix zusätzliche visuelle Informationen dazu, welche Vergleiche sinnvoll auszuwerten wären. Zudem können Vergleiche nun persistent gespeichert werden, die Funktionsweise kann durch mehr Parameter für den vorliegenden Anwendungsfall angepasst werden und Aktionen und Fehler werden nun automatisch über ein eigenes Log-System festgehalten.

Insgesamt lässt sich sagen, dass die Anwendung MultiTextCompare sich nun nicht mehr nur durch die Anzahl der zu vergleichenden Dateien von bestehender Software abhebt, sondern zusätzlich Möglichkeiten bietet um Dateien vom Typ JSON und XML spezialisiert inhaltlich zu vergleichen.

Trotzdem gibt es noch Aspekte, die verbessert werden können. Dazu zählt z.B. eine weitere Anpassung im Matching, die nicht mehr bloß zeilenbasiert nach Matches sucht, sondern tatsächlich versucht verschobene Textblöcke zu erstellen. Desweiteren könnte die Matching-Komponente auch auf XML und JSON angepasst werden, sodass diese auch Markup erkennt und diesen beim Feststellen von Matches besonders beachtet. Dies ist auch etwas was in Abb. 7 zu sehen war, wo Markup-Elemente gleichermaßen zur Ähnlichkeit von Zeilen beigetragen haben wie der eigentliche Inhalt der Dateien. Das Best-Match Verfahren bringt zwar bessere Ergebnisse als wenn nach dem erstem Match die Suche abgebrochen wird, aber die Darstellung könnte durch eine Spezialisierung auf bestimmte Dateiformate weiter verbessert werden. Als letzten Punkt ließe sich noch die Einbindung des Dateiformats YAML nennen. Dies ist besonders im in der Einleitung genannten Cloud-Umfeld als Konfigurationsformat weit verbreitet und ist auch durch die Jackson Bibliothek unterstützt. Da JSON eine direkte Untermenge von YAML ist [18], sollte auch der strukturelle Vergleichsalgorithmus nicht neu entwickelt, sondern bloß angepasst werden müssen.

7 Quellenverzeichnis

[1] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, P. Dollár und et al., „Microsoft COCO: Common Objects in Context," 2015, S. 1. Adresse: https://arxiv.org/abs/1405.0312v3 (besucht am 29. 04. 2021).

[2] *Results Format*, Microsoft COCO. Adresse: https://cocodataset.org/#format-results (besucht am 29. 04. 2021).

[3] Dudenredaktion, *einstellen*. Adresse: https://www.duden.de/node/136591/revision/136627 (besucht am 30. 04. 2021).

[4] N. Nakatsu, Y. Kambayashi und S. Yajima, „A longest common subsequence algorithm suitable for similar text strings," *Acta Informatica*, S. 1, Adresse: https://link.springer.com/article/10.1007/BF00264437 (besucht am 01. 05. 2021).

[5] Y. S. Nugroho, H. Hata und K. Matsumoto, „How different are different diff algorithms in Git?" *Empirical Software Engineering*, S. 1–4, 2019. Adresse: https://link.springer.com/article/10.1007/s10664-019-09772-z (besucht am 01. 05. 2021).

[6] M. Paterson und V. Dančík, „Longest common subsequences," in *Mathematical Foundations of Computer Science 1994*, I. Prívara, B. Rovan und P. Ruzička, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, S. 127, ISBN: 978-3-540-48663-3.

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler und F. Yergeau, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 2008. Adresse: https://www.w3.org/TR/xml/ (besucht am 20. 05. 2021).

[8] S. Gao, C. M. Sperberg-McQueen und H. S. Thompson, *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*, 2012. Adresse: https://www.w3.org/TR/xmlschema11-1/ (besucht am 14. 06. 2021).

[9] T. Bray, „The JavaScript Object Notation (JSON) Data Interchange Format," Nr. 8259, S. 2 –6, Dez. 2017. DOI: 10.17487/RFC8259. Adresse: https://rfc-editor.org/rfc/rfc8259.txt (besucht am 20. 05. 2021).

[10] *TIOBE Index for May 2021*. Adresse: https://www.tiobe.com/tiobe-index/ (besucht am 17. 05. 2021).

[11] *The Event Dispatch Thread*, Oracle. Adresse: https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html (besucht am 03. 06. 2021).

[12] *SwingWorker*, Oracle. Adresse: https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html (besucht am 03. 06. 2021).

[13] *Source code LevenshteinDistance implementation*, Apache Software Foundation. Adresse: https://commons.apache.org/proper/commons-text/apidocs/src-html/org/apache/commons/text/similarity/LevenshteinDistance.html (besucht am 02. 05. 2021).

[14] *ExecutorService*, Oracle. Adresse: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html (besucht am 03. 06. 2021).

[15] *How to Set the Look and Feel*, Oracle. Adresse: https://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html (besucht am 03. 06. 2021).

[16] R. Carrick, *Row Number Table*. Adresse: https://tips4java.wordpress.com/2008/11/18/row-number-table/ (besucht am 04. 06. 2021).

[17] *Produktspezifikationen*, Intel. Adresse: https://ark.intel.com/content/www/de/de/ark.html#@PanelLabel122139 (besucht am 29. 06. 2021).

[18] O. Ben-Kiki, C. Evans und I. d. Net, *YAML Ain't Markup Language(YAML™) Version 1.2*, 2009. Adresse: https://yaml.org/spec/1.2/spec.html#id2759572 (besucht am 05. 06. 2021).

Erklärung über die selbständige Abfassung der Arbeit

Ich erkläre an Eides statt, dass ich die vorgelegte Abschlussarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

(Ort, Datum, Unterschrift)