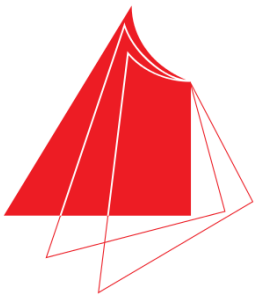


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Textmergealgorithmen

Seminararbeit in Informatik an der Hochschule Karlsruhe für Technik und Wirtschaft

Tim Roes

Karlsruhe, im Sommersemester 2012
geändert am 18. November 2012

Betreuer: Prof. Dr. Heiko Körner

Vorwort zum geänderten Version

Leider hatte sich in der ersten Version der Ausarbeitung ein gravierender Fehler in Abschnitt 3.2.4 versteckt. Der in Abbildung 3.11 dargestellte Algorithmus hatte in Zeile 12 einen Fehler, auf Grund dessen er für die meisten Beispiele nicht funktioniert hat.

Damit der Algorithmus ohne Beschränkungen funktioniert, muss bei der Suche der beiden Abstände o_A und o_B (in Zeile 12) noch geprüft werden, dass die beiden gefunden Elemente auch gleich sind, welches in der ersten Version dieser Ausarbeitung nicht geprüft wurde.

Die Ausarbeitung ist nun entsprechend angepasst und ich möchte mich recht herzlich bei Susanne Schulze bedanken, die mich auf den Fehler aufmerksam gemacht hat und mich mit ausreichend Beispielen versorgt hat, welche die Problematik gezeigt haben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Gerade in der Softwareindustrie wurden Projekte im Laufe der Zeit immer komplexer und größer. Somit wurden auch immer mehr Entwickler benötigt und die Teamgrößen stiegen an. Zusätzlich zu der Anzahl der Entwickler, wuchs im Laufe der Zeit auch deren Entfernung zueinander. So ist es heutzutage nicht mehr unüblich, dass Entwickler eines Projektes quer über die Welt verteilt arbeiten.

Solch eine Entwicklung kann aber nur stattfinden, wenn Systeme vorhanden sind, die eine solche dezentrale Teamarbeit unterstützen. An erste Stelle solcher Hilfswerkzeuge stehen in der Softwareentwicklung sicherlich Versionsverwaltungssysteme, die das gemeinsame Arbeiten an einem Projekt erleichtern. Das System kümmert sich um die Versionierung und erlaubt mehreren Entwicklern, parallel an einer Datei zu arbeiten. Anschließend versucht es alle vorgenommen Änderungen bestmöglich zusammenzuführen.

Zum Zusammenführen der Änderungen werden entsprechende Algorithmen benötigt. Die vorliegende Arbeit erläutert die Funktionsweise solcher Algorithmen, die Mehrfachänderungen an Textdateien (insbesondere auch Quellcode) zusammenführen können. Dazu wird nach einer allgemeinen Einführung speziell die Funktionsweise eines bestimmten Textmergealgorithmus detailliert erläutert.

Kapitel 2

Grundlagen

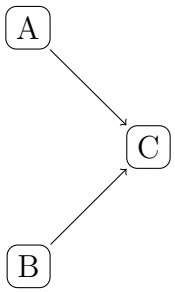
2.1 Arten von Mergealgorithmen

Bei Mergealgorithmen kann zwischen zwei grundlegenden Sorten von Algorithmen unterschieden werden: dem Zwei-Wege-Merge und dem Drei-Wege-Merge. Im folgenden werden die beiden Varianten erläutert.

2.1.1 Zwei-Wege-Merge

Beim Zwei-Wege-Merge existieren als Eingabe zwei getrennte Dateien. Diese Dateien sollen zu einer neuen Datei zusammen geführt werden. So werden in Abbildung 2.1 Datei A und Datei B zu einer neuen Datei C zusammengeführt.

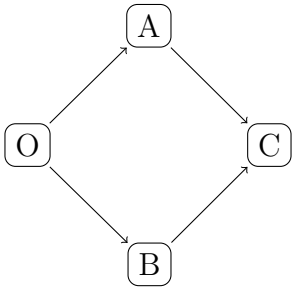
Abbildung 2.1: Zwei-Wege-Merge



2.1.2 Drei-Wege-Merge

Im Gegensatz zum Zwei-Wege-Merge existiert beim Drei-Wege-Merge eine gemeinsame Ausgangsdatei. Wie in Abbildung 2.2 zu sehen ist, sind Datei A und Datei B geänderte Versionen der Datei O. Diese beiden Änderungen sollen wieder in einer Datei C zusammengeführt werden.

Abbildung 2.2: Drei-Wege-Merge



2.2 Einsatzgebiete

Es gibt eine Reihe von Verwendungen für Mergealgorithmen. Der vermutlich häufigste Anwendungsfall von Textmergealgorithmen ist die Versionsverwaltung.

2.2.1 Versionsverwaltung

Aufgabe der Versionsverwaltung ist es, Dokumente jeglicher Art in verschiedenen Versionen zu verwalten. Die Versionsverwaltung soll dabei sicherstellen, dass mehrere Personen gleichzeitig auf den Dokumenten arbeiten können sowie dass Änderungen protokolliert und archiviert werden. Besonders häufig wird Versionsverwaltung bei der Entwicklung von Anwendungen verwendet. So werden unter anderem der Quelltext, aber auch die Dokumentation und weitere benötigte Dokumente und Anwendungen (zum Beispiel Compiler) in einem Versionsverwaltungssystem abgelegt.

Bei Versionsverwaltungssystemen kommt in der Regel der Drei-Wege-Merge zum Einsatz. Eine Ausgangsdatei wird von mehreren Personen parallel bearbeitet und diese Änderungen müssen anschließend zusammengeführt werden, so dass alle Änderungen bestmöglich übernommen werden.

Einige bekannte Systeme für die Versionsverwaltung von Quellcode sind zum Beispiel SVN[Svn], git[Git] oder Mercurial[Mer]. Ein weiteres Teilgebiet der Versions-

verwaltung ist der Einsatz von Wikisystemen. Diese werten im nächsten Abschnitt erläutert.

2.2.2 Wikis

Wikis sind auch eine Form von Versionsverwaltungsystemen. Es handelt sich bei diesen überwiegend um Webanwendungen, welche es Benutzern erlauben, zusammen an Inhalten im Internet zu arbeiten. Das bekannteste Wikisystem ist sicherlich MediaWiki[Med], welches für das Projekt Wikipedia verwendet wird. Andere Vertreter dieser Kategorie sind zum Beispiel DokuWiki[Dok] oder TWiki[Twi].

Bei Wikisystemen kommt ebenso wie bei der Quellcodeversionierung fast ausschließlich der Drei-Wege-Merge zum Einsatz. Auch hier existiert meist eine Ausgangsdatei, an der mehrere Änderungen parallel vorgenommen werden, die anschließend zu einer Ausgabe zusammengeführt werden sollen. Allerdings unterstüzt Nicht jedes Wikisystem unterstützt diese Möglichkeit. Einige Wikis verwenden Sperren, um das parallele Bearbeiten einer Seite zu verhindern. In diesem Fall müssen keine Änderungen gemergt werden. Jedoch leidet die Benutzbarkeit unter Umständen, wenn viele Seiten lange blockiert sind.

Kapitel 3

Algorithmen

Nach der Erläuterung der Grundlagen von Textmergealgorithmen, soll im Rest der Arbeit deren Funktionsprinzip genauer erläutert werden.

Die Suche nach Literatur zu verschiedenen Textmergealgorithmen, erweist sich als recht schwierig. Es ist kaum Literatur über Analyse und Beschreibung verschiedener Textmergealgorithmen zu finden. Ein besonders im Bereich der UNIX-Systeme etabliertes Mergewerkzeug ist das Programm *diff3*. In der Arbeit *A Formal Investigation of Diff3*[KKP07] wurde die Funktionsweise dieses Werkzeuges und des darin verwendeten Algorithmus genauer untersucht und Schwachstellen aufgezeigt. Im Folgenden wird die Funktionsweise von Textmergealgorithmen am Beispiel des *diff3*-Algorithmus erläutert. Die Erklärungen stützen sich dabei, soweit nicht anders angegeben, auf [KKP07].

3.1 Geschichte von *diff3*

Das Werkzeug *diff3* wurde 1988 von Randy Smith entwickelt und beinhaltet zwischenzeitlich viele Änderungen von Paul Eggert.[Egga; Eggb] Es wird zusammen mit anderen Werkzeugen (zum Beispiel dem *diff*-Werkzeug) in den *GNU-Diffutils*[Dif] vertrieben. Auf UNIX-Systemen ist *diff3* auch durch seine Verwendung in Versionsverwaltungssystemen wie Subversion weit verbreitet.[PCSF04, S. 245]

Wird in den folgenden Erklärungen von Textmergealgorithmen gesprochen, ist damit spezifisch die Funktionsweise von *diff3* gemeint.

3.2 Funktionsweise von *diff3*

Im Folgenden wird zunächst ein Überblick über die Funktionsweise von *diff3* gegeben. Anschließend werden die einzelnen Schritte genauer erläutert.

Als Dateinamen werden im Folgenden einheitlich O für die Originaldatei, A und B für die beiden veränderten Versionen, sowie C für die Ausgabedatei verwendet.

3.2.1 Grundlegende Funktionsweise

Zunächst müssen die Eingabedateien O, A und B eingelesen und in eine Liste von kleinsten Elementen umgewandelt werden. Die Beschreibung dieses Schrittes folgt in Abschnitt 3.2.2.

Anschließend werden die längsten gemeinsamen Teilfolgen der Datei O und der Datei A, beziehungsweise der beiden Dateien O und B ermittelt. Mit Hilfe der längsten gemeinsamen Teilfolgen können anschließend die Unterschiede und Gemeinsamkeiten der Dateien ermittelt werden. Eine genauere Erläuterung dazu erfolgt in Abschnitt 3.2.3.

Mit Hilfe der im vorherigen Schritt ermittelten längsten Teilfolgen von O und A beziehungsweise O und B, wird anschließend eine Liste aus instabilen und stabilen Sequenzen erstellt. Dieser Schritt stellt das Kernstück des *diff3*-Algorithmus dar und wird in Abschnitt 3.2.4 genauer beschrieben.

Im letzten Schritt wird für jede im vorherigen Schritte erstellten Sequenzen geprüft, welche Änderung in die Ausgabedatei übernommen werden muss, beziehungsweise ob ein Konflikt zwischen den beiden Änderungen vorliegt. Die so erstelle Ausgabeliste kann anschließend zum Beispiel als Datei (falls Konflikte auftraten mit entsprechenden Konfliktmarkierungen) wieder ausgegeben werden. Dies wird in Abschnitt 3.2.5 erläutert.

In Abbildung 3.1 sind die Schritte des Algorithmus zusammengefasst.

3.2.2 Einlesen der Dateien

Zunächst müssen die Originaldatei O, sowie ihre beiden veränderten Versionen A und B eingelesen werden. Beim Einlesen der Datei wird diese in eine Liste aus kleinsten Elementen zerteilt. Im Folgenden wird eine Datei x als Konfiguration $K(x)$

Abbildung 3.1: Schritte des *diff3*-Algorithmus

1. Lese Datei O,A und B ein
2. Unterschiede zwischen O und A, sowie O und B ermitteln
3. Liste der instabilen und stabilen Sequenzen erstellen
4. Finale Ausgabeliste erstellen

dargestellt. Die Konfiguration ist dabei eine Liste kleinster Elemente, dargestellt durch Zahlen. Die Zahlen repräsentieren jeweils eines der kleinsten Elemente. Im folgenden Beispiel sind zwei Dateien A und B gegeben, welche sich nur in ihrem zweiten Element unterscheiden.

$$K(A) = [1, 2, 3, 4]$$
$$K(B) = [1, 5, 3, 4]$$

Im Folgenden sei auch ein Indexzugriff, beginnend bei dem Index 1, auf Konfigurationen erlaubt. So ist im obigen Fall $K(B)[2] = 5$.

Neben dem Indexzugriffs ist ein Zugriff auf einen Bereich einer Konfiguration möglich. Dieser wird als $K(X)[i..k]$ geschrieben, wobei dies den Bereich des i-ten bis einschließlich k-ten Elements bezeichnet. Im Beispiel oben gilt also $K(A)[2..3] = [2, 3]$ und $K(B)[1..3] = [1, 5, 3]$.

Wahl des kleinsten Elementes

Eine weit verbreitete Wahl des kleinsten Elements ist das einer Zeile innerhalb der Datei. Das Werkzeug *diff3* verwendet immer Zeilen als kleinste Elemente. Sollen statt Quellcodedateien zum Beispiel Fließtexte, wie im Falle eines Wikisystems, zusammengeführt werden, ist eventuell eine andere Granularität sinnvoll. So kann eine Unterteilung in Sätze anstatt Zeilen in einigen Fällen bessere Ergebnisse erzielen.

In Abbildung 3.2 ist das Problem des zeilenweisen Zusammenfügens von Fließtext anhand eines Beispiels verdeutlicht.

In dem Beispiel wurde in Datei A nur die erste Zeile geändert, in Datei B nur die dritte Zeile. In diesem Fall würde der Algorithmus, wie er im Weiteren erläutert

Abbildung 3.2: Beispiel zum zeilenweise Zusammenführen von Fließtext

Datei A	**Datei O**	**Datei B**
Dieses Haus würde ich wirklich sehr gerne kaufen.	Diese Villa würde ich wirklich sehr gerne kaufen.	Diese Villa würde ich wirklich sehr gerne kaufen, denn sie gefällt mir.

wird, problemlos beide Änderungen zu dem Satz "Dieses Haus würde ich wirklich sehr gerne kaufen, denn sie gefällt mir." zusammen führen. Dieser Satz beinhaltet nun einen grammatikalischen Fehler, denn das Wort "Haus" ist im Deutschen ein Neutrum und der Satz müsse somit korrekt "Dieses Haus würde ich wirklich sehr gerne kaufen, denn *es* gefällt mir." lauten. Würden als kleinste Elemente Sätze verwendet, träte dieser Fehler nicht, da dann an einem Element zwei unterschiedliche Änderungen vorgenommen würden, was zu einem Konflikt führt.

In den meisten Fällen werden Zeilen eines Fließtextes länger sein als im obigen Beispiel. Dann wird es häufiger vorkommen, dass unnötige Konflikte erzeugt werden, wenn zwei komplett getrennte Sätze, die in einer Zeile standen, bearbeitet wurden.

Die Wahl von Sätzen als kleinstes Element kann somit bei Fließtexten einige grammatikalische Fehler verhindern, die beim zeilenweise Zusammenführen auftreten würden. Es können jedoch weiterhin semantische Fehler auftreten, wie zum Beispiel, dass ein Satz eingefügt wird, der sich auf eine vorangehende Textstelle bezieht, die jedoch von einem anderen Benutzer herausgelöscht wurde.

3.2.3 Unterschiede zwischen Original und bearbeiteten Versionen bestimmen

Zwischen den im ersten Schritt erzeugten Konfigurationen $K(O)$ und $K(A)$, sowie $K(O)$ und $K(B)$ werden anschließend die Unterschiede ermittelt. Dies entspricht jeweils einem Zwei-Wege-Merge der beiden Dateien. Dazu muss zunächst die längste gemeinsame Teilfolge der beiden Konfigurationen bestimmt werden.

Längste gemeinsame Teilfolge

Eine gemeinsame Teilfolge (GT) zweier Konfigurationen $K(X_1)$ und $K(X_2)$ ist erneut eine Konfiguration, welche durch Wegstreichen von Elementen aus $K(X_1)$ sowie $K(X_2)$ entsteht. Elemente werden so gestrichen, dass beide Konfigurationen danach dieselben Elemente in derselben Reihenfolge beinhalten (diese nennt man dann ei-

ne gemeinsame Teilfolge). Die Anzahl und Position der gestrichenen Zeichen kann dabei für $K(X_1)$ und $K(X_2)$ unterschiedlich sein.

Das Ganze soll an folgendem Beispiel verdeutlicht werde: Gegeben sind zwei Konfigurationen $K(X_1)$ sowie $K(X_2)$.

$$\begin{aligned} K(X_1) &= [1, 2, 3, 4, 5, 6] \\ K(X_2) &= [1, 4, 5, 7, 2, 3] \end{aligned}$$

Werden aus $K(X_1)$ die Elemente 1, 2, 3 und 6 gestrichen, bleibt die Konfiguration $[4, 5]$. Werden aus $K(X_2)$ die Elemente 1, 7, 2 und 3 gestrichen, bleibt ebenso die Konfiguration $[4, 5]$ übrig. Diese ist somit eine gemeinsame Teilfolge von $K(X_1)$ und $K(X_2)$.

$$\begin{aligned} K(X_1) &= [\not{1}, \not{2}, \not{3}, 4, 5, \not{6}] \\ K(X_2) &= [\not{1}, 4, 5, \not{7}, \not{2}, \not{3}] \\ GT(X_1, X_2) &= [4, 5] \end{aligned}$$

In diesem Beispiel gibt es noch einige andere gemeinsame Teilfolgen, wie zum Beispiel die Teilfolge $[1, 2, 3]$:

$$\begin{aligned} K(X_1) &= [1, 2, 3, \not{4}, \not{5}, \not{6}] \\ K(X_2) &= [1, \not{4}, \not{5}, \not{7}, 2, 3] \\ GT(X_1, X_2) &= [1, 2, 3] \end{aligned}$$

Eine längste gemeinsame Teilfolge (LGT) ist eine gemeinsame Teilfolge maximaler Länge. Im obigen Beispiel ist das unter anderem die Teilfolge $[1, 2, 3]$, da sich keine gemeinsame Teilfolge finden lässt, die mehr als drei Elemente beinhaltet.

Die längste gemeinsame Teilfolge muss aber nicht eindeutig sein. So gibt es im obigen Beispiel neben $LGT(X_1, X_2) = [1, 2, 3]$ noch eine andere längste gemeinsame Teilfolge:

$$\begin{aligned} K(X_1) &= [1, \not{2}, \not{3}, 4, 5, \not{6}] \\ K(X_2) &= [1, 4, 5, \not{7}, \not{2}, \not{3}] \\ LGT_2(X_1, X_2) &= [1, 4, 5] \end{aligned}$$

Um eine längste gemeinsame Teilfolge zu bestimmten, verwendet *diff3* einen Algorithmus, der ursprünglich in [Mye86] und [Ukk85] erläutert wird. Die folgende Erläuterung zum Ermitteln einer längsten gemeinsamen Teilfolge stützt sich auf die Beschreibungen in [Cor+09].

Zur besseren Lesbarkeit wird im Folgenden $K(X_1)$ durch X_1 und $K(X_2)$ durch X_2 abgekürzt.

Der Algorithmus macht sich die folgenden drei Eigenschaften des Problems der längsten gemeinsamen Teilfolgen zunutze, für deren Beweis auf [Cor+09, S. 392] verwiesen sei:

1. Wenn das letzte Element von X_1 und X_2 übereinstimmt, ist dies auch das letzte Element einer längsten gemeinsamen Teilfolge. Nimmt man das letzte Zeichen dieser längsten gemeinsamen Teilfolge weg, ist die resultierende Folge wieder eine längste gemeinsame Teilfolge von $X_1[1..len(X_1) - 1]$ und $X_2[1..len(X_2) - 1]$, wobei die Funktion $len()$ die Länge der entsprechenden Konfiguration bezeichnet.

2. Sind die letzten beiden Elemente von X_1 und X_2 unterschiedlich und das letzte Element einer längsten gemeinsamen Teilfolge ungleich dem letzten Element von X_1, impliziert dies, dass diese längste gemeinsame Teilfolge auch eine längste gemeinsame Teilfolge von $X_1[1..len(X_1) - 1]$ und X_2 ist.

3. Sind die letzten beiden Elemente von X_1 und X_2 unterschiedlich und das letzte Element einer längsten gemeinsamen Teilfolge ungleich dem letzten Element von X_2, impliziert dies, dass diese längste gemeinsame Teilfolge auch eine längste gemeinsame Teilfolge von X_1 und $X_2[1..len(X_2) - 1]$ ist.

Mit Hilfe dieser drei Eigenschaften lässt sich das Problem der längsten gemeinsamen Teilfolge rekursiv lösen. Die Grundidee hinter dem Algorithmus ist im Folgenden beschrieben.

Als Startwerte der Rekursion sollen $A = X_1$ und $B = X_2$ gewählt werden. Diese Startwerte beschreiben unser eigentliches Problem, eine längste gemeinsame Teilfolge von X_1 und X_2 zu finden. In den folgenden Schritten werden A und B stets

verkleinert und das Problem somit rekursiv auf mehrere kleinere Probleme aufgeteilt.

Werden die beiden Konfigurationen A und B betrachtet, können dabei die folgenden drei Fälle auftreten:

(a) Ist das letzte Element von A und B gleich, ist dies wie in der ersten Eigenschaft beschrieben auch das letzte Element einer längsten gemeinsamen Teilfolge. Der Rest dieser längsten gemeinsamen Teilfolge muss anschließend aus $A[1..len(A)-1]$ und $B[1..len(B)-1]$ ermittelt werden.

(b) Sind die letzten Elemente von A und B verschieden, gilt entweder die zweite oder dritte Eigenschaft. Da die längste gemeinsame Teilfolge nicht bekannt ist, wird rekursiv in sowohl $A = A[1..len(A)-1]$ und $B = B$ als auch in $A = A$ und $B = B[1..len(B)-1]$ nach einer längsten gemeinsamen Teilfolge gesucht. Dies bedeutet zwei rekursive Abstiege, wobei am Ende die längere der beiden Teilfolgen aus den rekursiven Abstiegen verwendet wird.

(c) Ist entweder A oder B leer, wird die Rekursion beendet.

Das Ganze soll am Beispiel $X_1 = [1, 2, 3]$ und $X_2 = [4, 1, 3]$ veranschaulicht werden. Die im Folgenden beschriebenen Rekursionspfade sind in Abbildung 3.3 als Graph dargestellt.

Zu Beginn der Rekursion ist $A = X_1$ und $B = X_2$. Das letzte Element ist in beiden Konfigurationen die 3 und ist somit (Fall a) ein Bestandteil einer längsten gemeinsamen Teilfolge. Die restlichen Elemente dieser längsten gemeinsamen Teilfolge müssen nun ermittelt werden, indem $A = [1, 2]$ und $B = [4, 1]$ betrachtet wird.

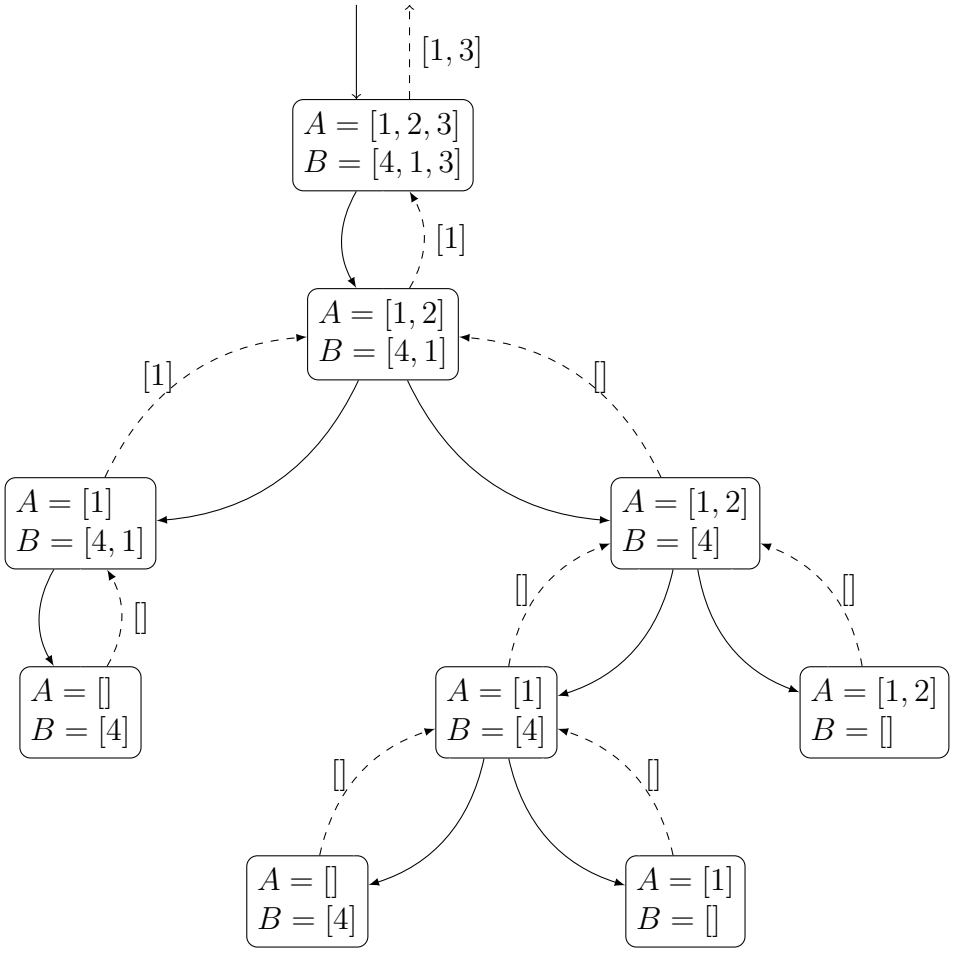
Da in diesem Schritt die beiden letzten Elemente unterschiedlich sind (Fall b), gibt es zwei Rekursionspfade. Einer sucht eine längste gemeinsame Teilfolge in $A = [1]$ und $B = [4, 1]$ der andere Pfad sucht in $A = [1, 2]$ und $B = [4]$. Die längere der von beiden Pfaden zurückgelieferten Folgen wird dann vor die bisher gefundenen Elemente gestellt.

Zunächst soll der Pfad, welcher $A = [1, 2]$ und $B = [4]$ überprüft, betrachtet werden. Dies ist wieder Fall b, da die beiden letzten Elemente unterschiedlich sind. Also wird zu $A = [1]$ und $B = [4]$ sowie $A = [1, 2]$ und $B = []$ verzweigt. Im zweiten Pfad bricht die Rekursion ab, da B leer ist. Im ersten Fall würden nochmals zwei Rekursionspfade erstellt, die aber beide abbrechen, da entweder A oder B leer ist. Der gesamte Pfad mit $A = [1, 2]$ und $B = [4]$ konnte somit keine längste gemeinsame Teilfolge ermitteln.

Im Pfad $A = [1]$ und $B = [4, 1]$ tritt wieder Fall a auf, da die beiden letzten Elemente gleich sind. Somit wird für $A = []$ und $B = [4]$ weiter geprüft. Da A leer ist, bricht die Rekursion sofort ab. Dieser Pfad gibt folglich [1] als längste gemeinsame Teilfolge zurück.

Somit konnte einer der Pfade keine längste gemeinsame Teilfolge ermitteln, der andere [1]. Die längere der beiden Teilfolgen (in diesem Falle [1]) wird anschließend an den ursprünglichen Aufruf der Rekursion zurückgegeben. Der ursprüngliche Aufruf hatte schon [3] als Ende der längsten gemeinsamen Teilfolge ermittelt und gibt nun insgesamt $[1, 3]$ als längste gemeinsame Teilfolge zurück. In diesem Beispiel ist dies auch die einzige längste gemeinsame Teilfolge.

Abbildung 3.3: Rekursive Aufrufe zur Bestimmung der längsten gemeinsamen Teilfolge



Für die Implementierung kann die Rekursion aufgelöst werden. Dafür wird eine Matrix C erstellt, welche an der Position $C[i, j]$ die Länge einer längsten gemeinsamen Teilfolge von $X_1[1..i]$ und $X_2[1..j]$ enthält. Die Matrix lässt sich ohne Rekursion

füllen und reicht aus, um anschließend eine längste gemeinsame Teilfolge aus ihr abzulesen.

Die Matrix besitzt $len(X_1)+1$ Zeilen, $len(X_2)+1$ Spalten und ihre Indizes beginnen bei 0. Der Wert eines bestimmten Elements der Matrix ist dabei wie folgt definiert:

$$C[i,j] = \begin{cases} 0 & \text{falls } i = 0 \text{ oder } j = 0 \\ C[i-1, j-1] + 1 & \text{falls } i,j > 0 \text{ und } X_1[i] = X_2[j] \\ max(C[i, j-1], C[i-1, j]) & \text{falls } i,j > 0 \text{ und } X_1[i] \neq X_2[j] \end{cases} \tag{3.1}$$

Diese drei Fälle spiegeln, wie weiter unten ausführlich erläutert, die drei Fälle der Rekursion von oben wieder.

Neben der Länge der gemeinsamen Teilfolge wird in der Matrix noch eine Zusatzinformation gespeichert. Diese Zusatzinformation kodiert die verschiedenen oben erwähnten Rekursionsfälle. Tritt der Fall auf, dass die beiden letzten Zeichen ungleich sind, muss außerdem gespeichert werden, welcher der beiden Rekursionspfade die längere gemeinsame Teilfolge zurück liefert. Somit ist die Zusatzinformation für ein Element der Matriz wie folgt definiert:

$$C[i,j] = \begin{cases} \text{keine} & \text{falls } i = 0 \text{ oder } j = 0 \\ \nwarrow & \text{falls } i,j > 0 \text{ und } X_1[i] = X_2[j] \\ \uparrow & \text{falls } i,j > 0 \text{ und } X_1[i] \neq X_2[j] \text{ und } C[i-1,j] \geq C[i,j-1] \\ \leftarrow & \text{falls } i,j > 0 \text{ und } X_1[i] \neq X_2[j] \text{ und } C[i-1,j] < C[i,j-1] \end{cases} \tag{3.2}$$

Falls $i = 0$ oder $j = 0$, entspricht dies einem Abbruch der Rekursion. Eine gemeinsame Teilfolge mit einer leeren Konfiguration existiert nicht, was bedeutet, dass die Länge 0 in der Matrix gespeichert wird und keine Zusatzinformation gespeichert werden muss.

Falls $i,j > 0$ und $X_1[i] = X_2[j]$ würden sowohl von A als auch B das letzte Zeichen gestrichen und in den verbleibenden Konfigurationen nach einer längsten gemeinsamen Teilfolge gesucht werden. Die Länge der dort gefunden längsten Teilfolge, lässt sich in der Matrix in $C[i-1, j-1]$ ablesen. Als zusätzliche Information wird somit das Zeichen "$\nwarrow$" hinterlegt, welches die Richtung angibt, in der der nächste Rekursionsschritt in der Matrix zu finden ist.

Falls $i, j > 0$ und $X_1[i] \neq X_2[j]$ steigt die Rekursion in zwei verschiedenen Pfaden ab. Die Länge der gefundenen längsten gemeinsamen Teilfolgen lassen sich in den benachbarten Elementen $C[i, j-1]$ beziehungsweise $C[i-1, j]$ ablesen. Als zusätzliche Information wird ein "$\leftarrow$" oder "$\uparrow$" hinterlegt, abhängig davon, in welchem der beiden benachbarten Elemente die längere gemeinsame Teilfolge gefunden werden konnte.

Eine Implementierung des Algorithmus muss eine Matrix anlegen, deren erste Zeile und Spalte mit 0 füllen und anschließend die Matrix gemäß der Formel 3.1 füllen. Je nach Datenstruktur werden die zusätzlichen Informationen in der gleichen Matrix abgelegt, oder es wird eine zusätzliche Matrix für sie erstellt. Eine mögliche Implementierung in Pseudocode ist in Abbildung 3.4 gezeigt.

Für das Beispiel $X_1 = [1, 2, 3]$ und $X_2 = [4, 1, 3]$ würde die Matrix anschließend wie in Abbildung 3.5 aussehen.

Aus der Matrix lässt sich leicht eine längste gemeinsame Teilfolge ablesen. Dazu startet man beim Element $C[len(X_1), len(X_2)]$ (in der obigen Darstellung unten rechts in der Matrix). Von dort an folgt man den Markierungen durch die Matrix, bis die erste Zeile oder Spalte erreicht ist. Die entsprechenden Elemente sind im Beispiel oben grün hinterlegt. Jedes Mal, wenn eine Zelle mit einer "$\nwarrow$" Markierung passiert wird, ist das entsprechende Element von X_1 und X_2 ein Element der längsten gemeinsamen Teilfolge. Auf diese Weise wird eine längste gemeinsame Teilfolge rückwärts aufgebaut.

Im obigen Beispiel wird eine "$\nwarrow$" Markierung bei $C[3, 3]$ erreicht. Das Element $X_1[3] = X_2[3] = 3$ ist also das letzte Element einer längsten gemeinsamen Teilfolge. Die nächste "$\nwarrow$" Markierung wird bei $C[1, 2]$ erreicht. $X_1[1] = X_2[2] = 1$ ist somit das vorletzte Element dieser längsten gemeinsamen Teilfolge. Anschließend wird keine weitere Markierung mehr gefunden und somit $[1, 3]$ als eine längste gemeinsame Teilfolge zurückgegeben.

Wie im Beispiel oben zu sehen ist (grau hinterlegte Elemente), gibt es Fälle, in denen beide benachbarten Zellen den gleichen Wert aufweisen. Dies ist immer dann der Fall, wenn auf beiden Rekursionspfaden eine gemeinsame Teilfolge gleicher Länge gefunden werden konnte. Gemäß der Definition 3.2 wird in solchen Fällen die "$\uparrow$" Markierung verwendet, da $C[i-1, j] \geq C[i, j-1]$. Wird die Definition so abgewandelt, dass für diese Markierung $C[i-1, j] > C[i, j-1]$ gelten muss und für die "$\leftarrow$" Markierung dementsprechend $C[i-1, j] \leq C[i, j-1]$, würde in solchen Fällen die "$\leftarrow$" Markierung gewählt werden. Von der Wahl der Definition hängt es also ab, welche längste gemeinsame Teilfolge zurückgegeben wird, falls mehrere existieren.

Abbildung 3.4: Algorithmus zum Füllen der Rekursionsmatrix

```
1   m = len(X_1)
2   n = len(X_2)
3   C = new Matrix[0..m, 0..n]
4   for i = 0 to m {
5       C[i, 0].length = 0
6   }
7   for j = 0 to n {
8       C[0, j].length = 0
9   }
10  for i = 1 to m {
11      for j = 1 to n {
12          if X_1[i] == X_2[j] {
13              C[i, j].length = C[i − 1, j − 1] + 1
14              C[i, j].info = ↖
15          } elseif C[i − 1, j] ≥ C[i, j − 1] {
16              C[i, j].length = C[i − 1, j]
17              C[i, j].info = ↑
18          } else {
19              C[i, j].length = C[i, j − 1]
20              C[i, j].info = ←
21          }
22      }
23  }
```

Abbildung 3.5: Matrix für $X_1 = [1, 2, 3]$ und $X_2 = [4, 1, 3]$

	j	0	1	2	3
i		$X_2[j]$	4	1	3
0	$X_1[i]$	0	0	0	0
1	1	0	0 ↑	1 ↖	1 ←
2	2	0	0 ↑	1 ↑	1 ↑
3	3	0	0 ↑	1 ↑	2 ↖

Der Algorithmus, wie er hier beschrieben wird, dient in *diff3* als Grundlage. Es existieren verschiedene Optimierungen, besonders im Hinblick auf Speicherkomplexität. Einige davon werden in [Cor+09, S. 396] erläutert.

Unterschiede und Gemeinsamkeiten zweir Konfigurationen ermitteln

Mit Hilfe einer längsten gemeinsamen Teilfolge zweier Konfigurationen lassen sich deren Unterschiede beziehungsweise Gemeinsamkeiten bestimmen. In Abbildung 3.6 wird ein Algorithmus gezeigt, der die Unterschiede und Gemeinsamkeiten ermittelt. Als Eingabe nimmt er die Konfigurationen $K(X_1)$, $K(X_2)$, sowie eine längste gemeinsame Teilfolge $LGT(X_1, X_2)$ entgegen. Als Ausgabe produziert er zwei Ausgabekonfigurationen $A(X_1)$ und $A(X_2)$. Auf die Ausgabekonfigurationen kann ebenso wie auf die Eingabekonfiguration per Indizes zugegriffen werden (siehe auch 3.2.2). Zur besseren Lesbarkeit werden LGT statt $LGT(X_1, X_2)$ und A_1, A_2 anstelle von $A(X_1)$ beziehungsweise $A(X_2)$ verwendet.

In Zeile 1 werden zunächst die Zeiger a, e_1 und e_2 mit 1 initialisiert. Diese Zähler zeigen an, welches Element der Ausgabekonfigurationen beziehungsweise von $K(X_1)$ und $K(X_2)$ gerade geschrieben oder gelesen wird.

Anschließend läuft die Schleife in Zeile 2 bis 20, so lange, bis alle Elemente von $K(X_1)$ oder $K(X_2)$ verarbeitet wurden.

In jedem Schleifendurchlauf wird eine von vier Möglichkeiten gewählt und anschließend in Zeile 19 der Ausgabezeiger um eins erhöht. So wird in jedem Schleifendurchlauf in das jeweils nächste Element der beiden Ausgabekonfigurationen geschrieben.

Die erste Möglichkeit (Zeile 3) ist, dass sowohl das aktuell betrachtete Element in $K(X_1)$ als auch $K(X_2)$ das nächste Element der längsten gemeinsamen Teilfolge

Abbildung 3.6: Unterschiede und Gemeinsamkeiten zwischen zwei Dateien ermitteln

```
1    a := e1 := e2 := 1
2    while e1 <= Länge von X1 oder e2 <= Länge von X2 {
3       if X1[e1] und X2[e2] nächstes Element in LGT {
4          A1[a] := A2[a] := X1[e1]
5          e1 := e1 + 1
6          e2 := e2 + 1
7       } elseif nur X1[e1] nächstes Element in LGT {
8          A2[a] := X2[e2]
9          e2 := e2 + 1
10      } elseif nur X2[e2] nächstes Element in LGT {
11         A1[a] := X1[e1]
12         e1 := e1 + 1
13      } elseif weder X1[e1] noch X2[e2] nächstes Element in LGT {
14         A1[a] := X1[e1]
15         A2[a] := X2[e2]
16         e1 := e1 + 1
17         e2 := e2 + 1
18      }
19      a := a + 1
20   }
```

sind. Bei beiden Ausgabekonfigurationen wird dieses Symbol geschrieben und anschließend die beiden Konfigurationszeiger weiter gerückt.

Bei der zweite Möglichkeit (Zeile 7) ist nur das Element aus $K(X_1)$ ein Element der längsten gemeinsamen Teilfolge. In diesem Fall wird nur das Element von $K(X_2)$ in dessen Ausgabekonfiguration kopiert und dessen Zeiger um eins erhöht. In die Ausgabekonfiguration $A(X_1)$ wird nichts geschrieben. Der Zeiger e_1 wird nicht erhöht, sondern wartet, bis in $K(X_2)$ auch ein Element der längsten gemeinsamen Teilfolge gefunden wurde. Dies ist der Fall, wenn in $K(X_2)$ im Vergleich zu $K(X_1)$ Elemente eingefügt wurden.

Die dritte Möglichkeit (Zeile 10) ist analog zur zweiten Möglichkeit, wenn nur das aktuelle Element von $K(X_2)$ ein Element der längsten gemeinsamen Teilfolge ist.

Bei der letzten Möglichkeit in Zeile 13 sind weder das Element in $K(X_1)$ noch das in $K(X_2)$ ein Element der längsten gemeinsamen Teilfolge. In diesem Fall werden beide Elemente in die Ausgabekonfigurationen geschrieben und beide Zeiger weiter gerückt. Dies ist der Fall, wenn eine der Konfigurationen in diesem Element im Vergleich zur anderen Konfiguration bearbeitet wurde.

Bei Betrachtung des Algorithmus lässt sich festellen, dass die Zeiger auch über das Ende der Konfiguration hinauslaufen können. Wird in so einem Fall von dem Zeiger gelesen, soll ein leeres Element geschrieben werden.

Das folgende Beispiel soll den Algorithmus veranschaulichen. Dazu werden die Konfigurationen $K(X_1) = [1, 2, 3, 4, 5, 6]$ und $K(X_2) = [1, 4, 5, 7, 2, 3]$ aus dem vorherigen Beispiel verwendet. Als längste gemeinsame Teilfolge wurde $LGT(X_1, X_2) = [1, 4, 5]$ ermittelt.

In Abbildung 3.7 werden die einzelnen Schleifendurchläufe Schritt für Schritt gezeigt. Dabei sind die Elemente, die gerade gelesen beziehungsweise geschrieben werden, farbig hinterlegt. Ist kein Element der Konfiguration farbig hinterlegt, steht der Zeiger außerhalb der Konfiguration und es wird, wie oben beschrieben, ein leeres Element kopiert.

Aus den so erzeugten Ausgabekonfigurationen lassen sich die Unterschiede und Gemeinsamkeiten der Konfigurationen ablesen. Im obigen Beispiel wurden die Elemente $[2, 3]$ von $K(X_1)$ nach $K(X_2)$ ans Ende verschoben, sowie das Element 6 zum Element 7 bearbeitet.

Wird $LGT(X_1, X_2) = [1, 2, 3]$ als längste gemeinsame Teilfolge verwendet, entsteht die in Abbildung 3.8 gezeigte Ausgabe.

Die so erhaltenen Ausgabekonfigurationen sind das Ergebnis eines Zwei-Wege-Merges,

Abbildung 3.7: Ermitteln der Unterschiede und Gemeinsamkeiten von K_1 und K_2

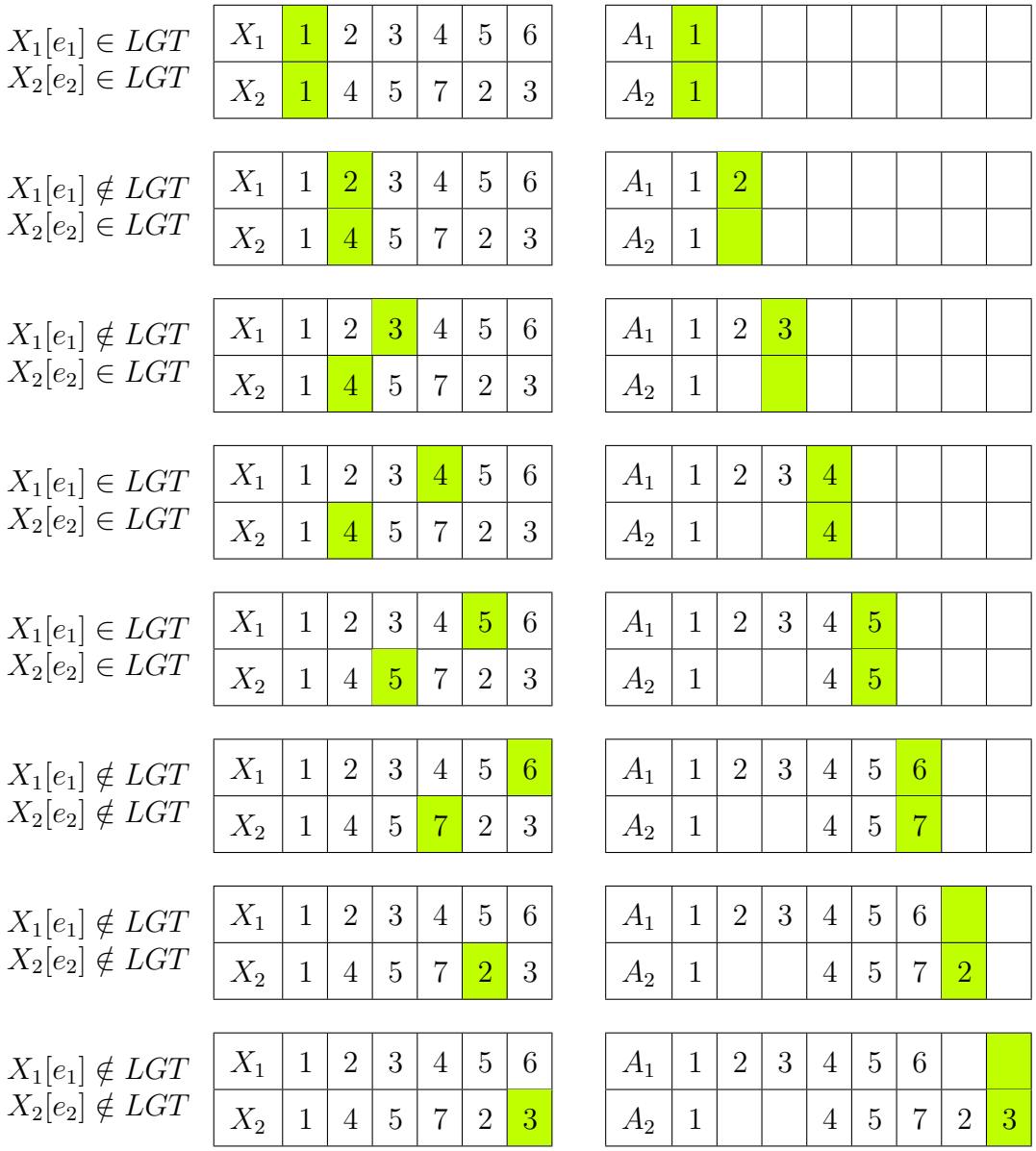


Abbildung 3.8: Ausgabe bei Verwendung von $LGT_2(X_1, X_2)$

$A(X_1)$	1				2	3	4	5	6
$A(X_2)$	1	4	5	7	2	3			

wie er auch vom UNIX Programm *Diff* erzeugt wird.[1] Die ermittelte Ausgabekonfiguration zweier Dateien X_1 und X_2 wird im Folgenden als $diff(X_1, X_2)$ bezeichnet.

Das Werkzeug *diff3* ruft intern das ebenfalls in den *GNU-Diffutils* enthaltene Programm *Diff* auf, um zu diesem Ergebnis zu gelangen. Wie es anschließend von $diff(O, A)$ und $diff(O, B)$ zu einem Drei-Wege-Merge dieser Dateien kommt, wird im folgenden Abschnitt erläutert.

3.2.4 Liste stabiler und instabiler Sequenzen erstellen

Die beiden Ausgaben $diff(O, A)$ und $diff(O, B)$ werden im eigentlichen Schritt des *diff3*-Algorithmus in eine Liste aus stabilen und instabilen Sequenzen unterteilt. Eine Sequenz besteht aus einem oder mehreren aufeinanderfolgenden Elementen von $K(O)$, $K(A)$ und $K(B)$. Eine stabile Sequenz bezeichnet eine Sequenz, in der alle Elemente (aus $K(O)$, $K(A)$ und $K(B)$) übereinstimmen. Weicht eine der Konfigurationen von den anderen ab, befindet sich das entsprechende Element in einer instabilen Sequenz. Diese werden so gewählt, dass ihre Größe maximal ist, was bedeutet, dass alle Änderungen bis zur nächsten stabilen Sequenz zu einer instabilen Sequenz zusammen gefasst werden. Somit erzeugt dieser Schritt eine alternierende Liste von stabilen und instabilen Sequenzen.

Um diese Sequenzen an einem Beispiel zu verdeutlichen, werden im Folgenden $K(O) = [1, 2, 3, 4, 5, 6]$, $K(A) = [1, 4, 5, 7, 2, 3]$ (dies entspricht den beiden Konfigurationen aus dem vorherigen Beispiel) sowie $K(B) = [1, 2, 3, 4, 7, 8]$ verwendet. In Abbildung 3.9 sind $diff(O, A)$, bei der Verwendung von $LGT(O, A) = [1, 2, 3]$, sowie $diff(O, B)$ gezeigt.

Abbildung 3.9: $diff(O, A)$ und $diff(O, B)$

$A(O)$	1				2	3	4	5	6
$A(A)$	1	4	5	7	2	3			

$A(O)$	1	2	3	4	5	6
$A(B)$	1	2	3	4	7	8

Die Liste der abwechselnd stabilen und instabilen Sequenzen würde in diesem Beispiel wie in Abbildung 3.10 dargestellt aussehen. Die stabilen Sequenzen sind die erste Sequenz [1] und die dritte Sequenz $[2, 3]$. Die zweite und die vierte Sequenz sind instabil, da sie sich in den drei Dateien voneinander unterscheiden. Für sie muss der *diff3*-Algorithmus eine Entscheidung treffen, welche Elemente in die Ausgabekonfiguration übernommen werden. Dies wird in Abschnitt 3.2.5 erläutert.

[1] Um ein Ergebnis zu bekommen, das den Ausgabetabellen im Beispiel ähnelt, kann der Befehl `diff -y DATEI_1 DATEI_2` verwendet werden.

Abbildung 3.10: Ausgabeliste des *diff3*-Algorithmus

$A(A)$	1	4,5,7	2,3	
$A(O)$	1		2,3	4,5,6
$A(B)$	1		2,3	4,7,8

Im Folgenden wird der Algorithmus erläutert, welcher die Ausgabeliste wie in Abbildung 3.10 erzeugt. Für diesen Algorithmus werden weitere Definitionen benötigt.

Zunächst wird eine Funktion $equal_{X,Y}(i)$ definiert. Diese arbeitet auf den bisher erstellten *diff* Ausgaben. Sie überprüft, ob in $diff(X, Y)$ an der Stelle i beide Elemente (das von $A(X)$ und das von $A(Y)$) übereinstimmen oder voneinander abweichen. Der Rückgabewert der Funktion ist $true$, falls beide Elemente gleich sind, andernfalls $false$. So gilt für das hier verwendete Beispiel $equal_{O,A}(1) = equal_{O,A}(5) = equal_{O,A}(6) = true$. Bei allen anderen Indizes außer 1, 5 und 6 von $diff(O, A)$ unterscheiden sich die Elemente, und die Funktion $equal_{O,A}$ gibt somit $false$ zurück. Für $diff(O, B)$ gilt $equal_{O,B}(5) = equal_{O,B}(6) = false$. Für alle anderen Indizes hat $equal_{O,B}$ den Rückgabewert $true$.

Neben der *equal*-Funktion wird der Zugriff auf einen Bereich einer *diff*-Ausgabe benötigt. Dies wird im Folgenden geschrieben als $\Delta_{X,Y}(X)[i...k]$ und bezeichnet das i-te bis k-te Element von $A(X)$ der $diff(X, Y)$-Ausgabe. Leere Elemente werden dabei weggelassen. In Abbildung 3.9 gilt zum Beispiel $\Delta_{O,A}(O)[3..5] = [2]$, $\Delta_{O,A}(A)[1..4] = [1, 4, 5, 7]$ und $\Delta_{O,B}(B)[4..5] = [4, 7]$. Wird k weggelassen, ist der Rest der Elemente bis zum Ende der *diff*-Ausgabe gemeint. $\Delta_{O,A}(A)[4..]$ ist somit $[7, 2, 3]$. Für die Fälle, in denen $k < i$, wird eine leere Sequenz zurückgegeben. Diese verkürzte Schreibweise wird verwendet, um die Lesbarkeit des Algorithmus zu erhöhen.

Die Funktionen `sseq` (stabile Sequenz) und `iseq` (instabile Sequenz) erstellen eine neue stabile beziehungsweise instabile Sequenz und hängen diese hinten an die Ausgabeliste an. `sseq` hat nur einen Parameter, der den Inhalt aller drei Teile (des O-, A- und B-Teiles) der Sequenz angibt. Die Funktion `iseq` nimmt drei Parameter entgegen. Der erste gibt den A-Teil der Sequenz an, der zweite den O-Teil und der letzte den B-Teil.

In Zeile 1 des Algorithmus werden zunächst zwei Zähler initialisiert. Der Zähler l_A wird verwendet, um das letzte Element der zuletzt erstellten Sequenz in $diff(O, A)$ zu markieren. Der Zähler l_B markiert entsprechend das letzte Element der zuletzt erstellten Sequenz in $diff(O, B)$. Zu Beginn sind beide mit 0 initialisiert, da bisher

Abbildung 3.11: *diff3*-Algorithmus

1 $l_A := l_B := 0$
2 `Wiederhole bis` $ENDE$ `eintritt:`
3 `Finde kleinstes positives` i, `so dass entweder` $equal_{O,A}(l_A + i) = false$ `oder` $equal_{O,B}(l_B + i) = false$. `Falls:`
4 `(a) es kein solches` i `gibt:`
5 `sseq(`$\Delta_{O,A}(O)[l_A + 1..]$`)`
6 $ENDE$
7 `(b)` $i > 1$`:`
8 `sseq(`$\Delta_{O,A}(O)[l_A + 1..l_A + i - 1]$`)`
9 $l_A := l_A + i - 1$
10 $l_B := l_B + i - 1$
11 `(c)` $i = 1$`:`
12 `Finde kleinste positive` o_A `und` o_B, `so dass` $equal_{O,A}(l_A + o_A) = true$ `und` $equal_{O,B}(l_B + o_B) = true$ `und` $\Delta_{O,A}(l_A + o_A) = \Delta_{O,B}(l_B + o_B)$. `Falls:`
13 `(i) es keine passenden` o_A `und` o_B `gibt:`
14 `iseq(`$\Delta_{O,A}(A)[l_A + 1..]$`,`$\Delta_{O,A}(O)[l_A + 1..]$`,`$\Delta_{O,B}(B)[l_B + 1..]$`)`
15 $ENDE$
16 `(ii) sonst:`
17 `iseq(`$\Delta_{O,A}(A)[l_A + 1..l_A + o_A - 1]$`,`
$\Delta_{O,A}(O)[l_A + 1..l_A + o_A - 1]$`,`
$\Delta_{O,B}(B)[l_B + 1..l_B + o_B - 1]$`)`
18 $l_A := l_A + o_A - 1$
19 $l_B := l_B + o_B - 1$

noch keine Sequenz erstellt wurde. Sie zeigen also vor die beiden $diff$-Ausgaben, da das erste Element den Index 1 hat.

Anschließend wird der Algorithmus so lange wiederholt, bis alle Elemente in Sequenzen aufgeteilt sind. Sobald dies geschehen ist, befindet sich der Algorithmus in einem der beiden $ENDE$ Fälle (Zeile 6 oder 15).

Für jede Sequenz wird versucht, ein minimales i zu finden, so dass $equal_{O,A}(l_A+i) = false$ oder $equal_{O,B}(l_B + i) = false$ gilt. Dies bedeutet, es wird der Anfang der nächsten instabilen Sequenz gesucht. Solange nämlich beide *equal*-Funktionen noch $true$ liefern, sind alle Elemente jeweils gleich und somit Bestandteile einer stabilen Sequenz. Die nächste instabile Sequenz beginnt dort, wo eine der beiden *equal*-Funktionen $false$ zurück liefert.

Existiert kein solches i (Zeile 4) sind alle drei Konfigurationen bis zum Ende hin identisch. Deshalb wird eine letzte stabile Sequenz ausgegeben und der Algorithmus terminiert anschließend. In Zeile 5 wird $\Delta_{O,A}(O)[l_A + 1..]$ für die Ausgabe der Sequenz verwendet. Dass die Elemente aus $A(O)$ von $diff(O, A)$ kopiert werden, muss nicht unbedingt sein. Ebenso könnte $A(A)$ von $diff(O, A)$ oder entweder $A(O)$ oder $A(B)$ von $diff(O, B)$ verwendet werden. Da dies eine stabile Sequenz ist, sind die Elemente in allen drei Konfigurationen in dem entsprechenden Bereich gleich. Als Beginnindex für den Zugriff wird l_A+1 verwendet, da l_A auf das letzte Element, der vorherigen Sequenz in $diff(O, A)$ zeigt, und die neue Sequenz somit ein Element später beginnt. Dies gilt für alle in dem Algorithmus verwendeten Beginnindizes. Analog wird $l_B + 1$ bei Zugriffen auf $diff(O, B)$ verwendet.

Existiert ein i, bei dem die nächste instabile Sequenz beginnt, können zwei Fälle auftreten. Im einfacheren Fall, ist $i > 1$ (Zeile 7). Dies bedeutet, dass zwischen der letzten Sequenz und der nächsten instabilen Sequenz mindestens ein anderes Element liegt. Alle Elemente, die bis zum Beginn der nächsten instabilen Sequenz liegen, ergeben somit eine stabile Sequenz, welche in Zeile 8 erstellt wird. Da $l_A + i$ beziehungsweise $l_B + i$ das erste Element der nächsten stabilen Sequenz sind, wird alles bis zu $l_A + i - 1$ in die stabile Sequenz gepackt. In Zeile 9 und 10 werden anschließend die Zähler so erhöht, dass sie auf das letzte Element der soeben angelegten Sequenz zeigen. Anschließend beginnt der Algorithmus, die nächste Sequenz zu suchen.

Ist $i = 1$ beginnt die nächsten instabilen Sequenz mit dem nächsten Element. Um diese zu erstellen, muss die nächste stabile Sequenz gefunden werden. Dazu werden ein minimales o_A und o_B gesucht, so dass $equal_{O,A}(l_A + o_A) = true$, $equal_{O,B}(l_B + o_B) = true$ und $\Delta_{O,A}(l_A+o_A) = \Delta_{O,B}(l_B+o_B)$. Im Gegensatz zu dem oben gesuchten

i müssen hier zwei verschiedene Werte ermittelt werden, da sich unterschiedlich viele Elemente aus den einzelnen Konfigurationen in einer instabilen Sequenz befinden können.

Lässt sich kein o_A und o_B finden (Zeile 13), existiert keine weitere stabile Sequenz mehr. In diesem Fall wird in Zeile 14 analog zu Zeile 4 eine letzte instabile Sequenz aus den restlichen Elementen erstellt. Anschließend terminiert der Algorithmus.

Existiert ein o_A und o_B dann gibt es noch mindestens eine stabile Sequenz hinter der aktuellen instabilen Sequenz. In diesem Fall (Zeile 16) wird eine instabile Sequenz bis zum Beginn der nächsten stabilen Sequenz ($l_A + o_A - 1$ bzw. $l_B + o_B - 1$) erstellt und anschließend die Zähler auf das letzte Element dieser instabilen Sequenz gesetzt.

Sobald der Algorithmus terminiert, wird eine alternierende Liste aus stabilen und instabilen Sequenzen ausgegeben.

Im Folgenden wird der Algorithmus am Beispiel aus Abbildung 3.9 gezeigt.

Zu Beginn sind l_A und l_B 0. Als i wird 2 ermittelt, da $equal_{O,A}(0+2) = false$. Es existiert kein kleineres i, bei dem entweder $equal_{O,A}$ oder $equal_{O,B}$ den Wert $false$ liefert. Somit tritt Fall (b) ein und es wird eine stabile Sequenz mit dem Inhalt $\Delta_{O,A}(O)[0+1..0+2-1] = \Delta_{O,A}(O)[1..1] = [1]$ erstellt. Anschließend werden $l_A = 0+2-1 = 1$ und $l_B = 0+2-1 = 1$ gesetzt.

Im nächsten Schritt wird $i = 1$, da $equal_{O,A}(1+1) = false$. Somit tritt Fall (c) ein, und es müssen ein o_A und o_B ermittelt werden. o_A bekommt den Wert 4, da $equal_{O,A}$ an der Stelle $5 = 1+4 = l_A + o_A$ das nächste Mal den Wert $true$ liefert. Analog dazu gilt $o_B = 1$, da $equal_{O,B}(1+1) = true$. In Zeile 17 wird folglich die instabile Sequenz $iseq(\Delta_{O,A}(A)[2..4], \Delta_{O,A}(O)[2..4], \Delta_{O,B}(B)[2..1]) = iseq([4,5,7],[],[])$ erstellt. Anschließend werden $l_A = 4$ und $l_B = 1$ gesetzt.

Im nächsten Schritt wird wieder ein i für die nächste instabile Sequenz gesucht. Durch $equal_{O,A}(l_A+i) = equal_{O,A}(4+3) = false$ und $equal_{O,B}(l_B+i) = equal_{O,B}(1+3) = false$ wird $i = 3$ gewählt. Es wird also im Fall (b) eine stabile Sequenz mit $\Delta_{O,A}(O)[5..6] = [2,3]$ erstellt. Anschließend gilt $l_A = 4+3-1 = 6$ und $l_B = 1+3-1 = 3$.

Im letzten Schritt ist $i = 1$. Wieder müssen ein o_A und o_B gefunden werden. Zwar gäbe es noch ein o_B mit 1, jedoch gibt es kein o_A mehr. Deshalb wird eine letzte instabile Sequenz $iseq([],[4,5,6],[4,7,8])$ erstellt. Anschließend terminiert der Algorithmus.

Somit werden genau die in Abbildung 3.10 gezeigten Ausgabesequenzen erzeugt.

Im letzten Schritt des *diff3*-Algorithmus wird für jede instabile Sequenz eine Entscheidung getroffen, welche Elemente in die Ausgabe übernommen werden. Dies wird im nächsten Kapitel erläutert.

3.2.5 Finale Ausgabeliste erstellen

Als letzter Schritt des Algorithmus, wird eine finale Ausgabeliste erstellt, in der die Änderungen der Dateien A und B, soweit wie möglich zusammengeführt und gegebenenfalls Konfliktstellen markiert werden. Dabei wird die Ausgabeliste des vorherigen Schrittes Sequenz für Sequenz abgearbeitet. Stabile Sequenzen können dabei problemlos übernommen werden, da diese in den beiden geänderten Dateien A und B genauso enthalten sind. Für alle instabile Sequenzen muss eine Entscheidung getroffen werden. Hier gibt es verschiedene Fälle, die auftreten können und je nach Situation wird entweder eine Ausgabe gewählt oder die entsprechende Sequenz als Konflikt markiert.

In Abbildung 3.12 sind alle Fälle aufgelistet.[Hof08, S. 446] Das Symbol $\emptyset$ bedeutet, dass der entsprechende O-, A- oder B-Teil einer Sequenz leer war. O ist der unveränderte Inhalt aus der Datei O. Wurde dieser verändert so wird dies als O' beziehungsweise O'' geschrieben. Das Symbol $\lightning$ bei der Ausgabe bedeutet, dass in dieser Sequenz ein Mergekonflikt vorliegt.

Bei den Fällen 1 bis 4 handelt es sich um das Einfügen von neuen Elementen. Im ersten und zweiten Fall wurde in A beziehungsweise B jeweils etwas hinzugefügt. Dies wird in die Ausgabe übernommen, da die Stelle in der anderen abgeänderten Datei nicht verändert wurde. Im dritten Fall wurde in beiden geänderten Versionen, das Gleiche hinzugefügt. Dieses kann somit in die Ausgabe übernommen werden. Im vierten Fall wurde in beiden Dateien Unterschiedliches an der gleichen Stelle hinzugefügt. Hier kann nicht automatisch bestimmt werden, welche Änderung in die Ausgabe übernommen werden soll, und es entsteht ein Konflikt.

Die Fälle 5 bis 8 beschreiben, was bei Änderungen geschieht. Im fünften und sechsten Fall wurde die entsprechende Stelle nur in einer Datei abgeändert. Die Änderung wird somit in die Ausgabe übernommen. Im siebten Fall, wurde die gleiche Änderung in Datei A und B vorgenommen und kann somit ebenfalls in die Ausgabe übernommen werden. Werden in beiden Dateien unterschiedliche Änderungen vorgenommen (Fall 8), führt dies ähnlich wie beim Hinzufügen auch zu einem Konflikt, da nicht automatisch entschieden werden kann, welche der Änderungen übernommen werden soll.

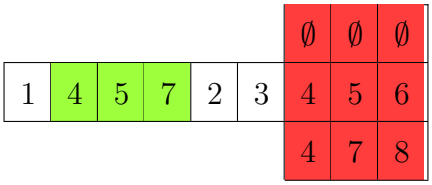
Abbildung 3.12: Fälle zur Auswahl der Ausgabesequenz

	$A(A)$	$A(O)$	$A(B)$	Ausgabe
1. Fall	$\emptyset$	$\emptyset$	O'	O'
2. Fall	O'	$\emptyset$	$\emptyset$	O'
3. Fall	O'	$\emptyset$	O'	O'
4. Fall	O'	$\emptyset$	O''	$\lightning$
5. Fall	O'	O	O	O'
6. Fall	O	O	O'	O'
7. Fall	O'	O	O'	O'
8. Fall	O'	O	O''	$\lightning$
9. Fall	O	O	$\emptyset$	$\emptyset$
10. Fall	$\emptyset$	O	O	$\emptyset$
11. Fall	$\emptyset$	O	$\emptyset$	$\emptyset$
12. Fall	$\emptyset$	O	O'	$\lightning$
13. Fall	O'	O	$\emptyset$	$\lightning$

In den Fällen 9 bis 13 geht es um das Löschen von Elementen. Wird nur in einer der geänderten Versionen eine bestimmte Stelle (Fall 9 und 10) oder in beiden dieselbe Stelle gelöscht (Fall 11), ist diese in der Ausgabe auch gelöscht. Die letzten beiden Fälle 12 und 13 treten auf, falls in einer Datei ein Abschnitt entfernt wurde, der in der anderen Datei bearbeitet wurde. Dies löst einen Konflikt aus.

Betrachtet man die Ausgabeliste aus Abbildung 3.10, tritt für die zweite Sequenz Fall 2 ein. Es werden also die Elemente $[4, 5, 7]$ in die Ausgabe übernommen. Bei der vierten Sequenz liegt Fall 13 vor und somit ein Konflikt. Die Ausgabe sieht dann wie in Abbildung 3.13 gezeigt aus.

Abbildung 3.13: Ergebnis der Zusammenführung



Diese Ausgabe ist das finale Ergebnis des *diff3*-Werkzeuges, welches bei Benutzung des Programmes in verschiedenen Ausgabeformaten angezeigt werden kann. In Abbildung 3.14 ist die Ausgabe für den Parameter `-m` zu sehen, welcher das zusammengeführte Ergebnis anzeigt. Für Konflikte werden dabei Konfliktmarker verwendet, welche den Inhalt der drei Dateien in der Konfliktregion aufzeigen. Für weitere Informationen zu Ausgabeformaten sei auf die Dokumentation der *GNU-diffutils*[Fou] oder die entsprechenden Manpages von *diff3*verwiesen.

3.3 Laufzeit des *diff3*-Algorithmus

Im Folgenden soll die Laufzeit des *diff3*-Algorithmus, wie er im vorherigen Kapitel beschrieben wurde, analysiert werden. Dazu wird der Zeitaufwand der einzelnen Schritte betrachtet.

Der Zeitaufwand soll dabei in Abhängigkeit von n angegeben werden, wobei n die Anzahl der Elemente einer der drei Konfigurationen ist. Für die Laufzeitanalyse gehen wir davon aus, dass alle drei Konfigurationen gleich viele Elemente haben und somit keine Unterscheidung zwischen den Konfigurationen gemacht werden muss. Dies ist ohne Beschränkung der Allgemeinheit gültig, wenn man davon ausgeht, dass n die Anzahl der Elemente der längsten Konfiguration repräsentiert. Zwar können

Abbildung 3.14: Ausgabe von `diff3 -m A O B`

```
1
4
5
7
2
3
<<<<<<< A
4
7
8
||||||| O
4
5
6
=======
>>>>>>> B
```

die anderen Konfigurationen weniger Elemente haben, jedoch spielt das für eine Laufzeitanalyse im Sinne des O-Kalküls keine Rolle, da eine obere Schranke der Laufzeit durch die längste Konfiguration gegeben ist.

Im ersten Schritt werden alle drei Dateien eingelesen und als Konfigurationen $K(A)$, $K(O)$ und $K(B)$ dargestellt. Da jedes Element lediglich einmal gelesen (und gespeichert) wird, entspricht der Aufwand des Einlesens einer Konfiguration $O(n)$. Da drei Konfigurationen zu verarbeiten sind, ist der Aufwand $O(3n) = O(n)$.

Im zweiten Schritt werden die Unterschiede und Gemeinsamkeiten von $K(O)$ zu $K(A)$ und $K(O)$ zu $K(B)$ ermittelt. Dazu wird die längste gemeinsame Teilfolge bestimmt, für die der Algorithmus 3.4 eine Matrix in $O(n^2)$ erstellt.[Cor+09, S. 394] Das Ablesen einer längsten gemeinsamen Teilfolge benötigt $O(2n)$.[Cor+09, S. 395] Wurde eine längste gemeinsame Teilfolge bestimmt, wird der Algorithmus in Abbildung 3.6 ausgeführt. Dieser hat einen Zeitaufwand von $O(2n)$, da er so lange läuft, bis entweder e_1 oder e_2 die Länge einer Konfiguration, also n, erreicht haben. Da in jedem Schritt mindestens einer der Zähler e_1 und e_2 um eins erhöht wird, und der Zählerstand niemals verringert wird, braucht der Algorithmus maximal $n+n$ Schritte. Somit liegt der Aufwand für den *diff* zweier Konfigurationen bei $O(n^2+2n+2n) = O(n^2)$. Da $diff(O, A)$ und $diff(O, B)$ bestimmt werden müssen, liegt der gesamt Aufwand des zweiten Schrittes bei $O(2n^2) = O(n^2)$.

Im dritten Schritt werden die instabilen und stabilen Sequenzen durch den Algorithmus aus Abbildung 3.11 erstellt. Da in jedem Schleifenschritt genau eine Sequenz

erstellt wird, macht der Algorithmus so viele Schleifendurchläufe, wie es Sequenzen gibt. Die Anzahl der Sequenzen ist durch die Anzahl der Elemente aller Konfigurationen begrenzt, also durch $n + n + n$. Die Anzahl der Durchläufe ist somit linear $O(n)$. In jedem Schritt muss dabei ein kleinstes positives i gefunden werden, was ebenso mit linearem Aufwand möglich ist. Falls $i = 1$ ist, müssen noch die beiden Werte o_A und o_B ermittelt werden. Diese zwei Werte lassen sich in $O(n + n)$ ermittelt. Da das Erstellen einer Sequenz über direkten Indexzugriff funktioniert, geschieht dies in konstanter Zeit, womit sich für den gesamten dritten Schritt ein Aufwand von $O(n^2)$ ergibt.

Im letzten Schritt muss die komplette Ausgabeliste durchgegangen und für alle instabilen Sequenzen eine Entscheidung getroffen werden. Dafür müssen die einzelnen Teile einer instabilen Sequenz miteinander verglichen werden. Im schlimmsten Fall befinden sich alle Elemente aller drei Konfigurationen in einer instabilen Sequenz (dies ist genau dann der Fall, wenn es nur eine instabile Sequenz gibt), und es müssen somit $n + n + n$ Elemente verglichen werden. Der Aufwand dieses letzten Schrittes ist somit $O(3n) = O(n)$.

Der gesamte Zeitaufwand eines Drei-Wege-Merges mit dem hier erläuterten Algorithmus liegt also bei $O(n + n^2 + n^2 + n) = O(n^2)$.

Der in [KKP07] beschriebene Algorithmus, auf den sich ein Großteil der Erläuterung in dieser Arbeit stützt, benötigt $O(n^3)$, da der dritte Schritt ohne die im zweiten Schritt erstellten *diff*-Ausgabelisten arbeitet. Die eigentliche Implementierung hat laut Angabe der Entwickler vermutlich die Komplexität $O(n^2)$.[Eggc]

Kapitel 4

Ausblicke

Zum Abschluss dieser Einführung in Textmergealgorithmen gibt es einige Verweise auf weiterführende Literatur, welche sich mit Problemen und Verbesserungsvorschlägen rund um das Thema Textmergealgorithmen befasst.

Bill Ritcher hat unter [Rit] einige Probleme typischer Drei-Wege-Mergealgorithmen aufgezeigt, um damit für den eigenen Algorithmus zu werben, der diese Probleme umgehen kann. Testet man *diff3* mit diesen Testfällen, schlagen die meisten fehl. Das beudetet, es werden unnötige Konflikte markiert, Änderungen verworfen oder Änderungen fehlerhaft zusammengeführt.

Auch in [KKP07] werden einige Schwächen des Algorithmus aufgezeigt. Ebenso werden einige intuitive Annahmen zum Verhalten von *diff3* als falsch aufgezeigt. So zum Beispiel die Vermutung, dass eine Änderung am Anfang einer Datei und eine am Ende mit genügend Abstand dazwischen sich in jedem Fall problemlos zusammenführen lassen.

Einen anderen Ansatz, um einen Drei-Wege-Merge auszuführen, bietet "Operational Transforming". Dabei werden die verschiedenen Änderungen als Transaktionen auf den Dateien aufgefasst. Beim Zusammenführen wird versucht, diese in eine konfliktfreie Reihenfolge zu bringen. Für eine Erläuterung dieser Technik sei auf [SE98] verwiesen. Mit Hilfe dieses Ansatzes lassen sich Algorithmen realisieren, die mehrfache Änderungen an einer Datei in $O(n)$ zusammen führen können.[SLG10]

Literatur

[Cor+09] Thomas H. Cormen u. a. In: *Introduction to Algorithms*. 2009. Kap. Longest common subsequence, S. 390 –397.

[Dif] *Diffutils*. URL: `http://www.gnu.org/software/diffutils` (besucht am 14. 05. 2012).

[Dok] *DokuWiki*. URL: `http://www.dokuwiki.org` (besucht am 14. 05. 2012).

[Egga] Paul Eggert. *First commit to diff3*. URL: `http://git.savannah.gnu.org/cgit/diffutils.git/commit/?id=f9d66e60` (besucht am 26. 04. 2012).

[Eggb] Paul Eggert. *Re: [bug-diffutils] time complexity of diff3*. URL: `http://lists.gnu.org/archive/html/bug-diffutils/2012-05/msg00012.html` (besucht am 29. 05. 2012).

[Eggc] Paul Eggert. *Re: [bug-diffutils] time complexity of diff3*. URL: `http://lists.gnu.org/archive/html/bug-diffutils/2012-05/msg00010.html` (besucht am 29. 05. 2012).

[Fou] Free Software Foundation. *GNU Diffutils - Comparing and Merging Files*. URL: `http://www.gnu.org/software/diffutils/manual/` (besucht am 11. 05. 2012).

[Git] *git*. URL: `http://git-scm.com` (besucht am 14. 05. 2012).

[Hof08] Dirk W. Hoffmann. *Software-Qualität*. Springer Verlag, 2008.

[KKP07] Sanjeev Khanna, Keshav Kunal und Benjamin C. Pierce. *A Formal Investigation of Diff3*. Techn. Ber. University of Pennsylvania, 2007.

[Med] *Mediawiki*. URL: `http://www.mediawiki.org` (besucht am 14. 05. 2012).

[Mer] *Mercurial*. URL: `http://mercurial.selenic.com` (besucht am 14. 05. 2012).

[Mye86] Eugene W. Myers. “An O(ND) difference algorithm and its variations”. In: *Algorithmica* 1.2 (1986), S. 251 –266.

[PCSF04] Michael Pilato, Ben Collins-Sussman und Brian W. Fitzpatrick. *Version Control With Subversion*. O'Reilly & Associates, Inc., 2004.

[Rit] Bill Ritcher. *Guiffy SureMerge - A Trustworthy 3-Way Merge*. URL: `http://www.guiffy.com/SureMergeWP.html` (besucht am 29. 05. 2012).

[SE98] Chengzheng Sun und Clarence Ellis. "Operational transformation in real-time group editors: issues, algorithms, and achievements". In: *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. CSCW '98. ACM, 1998, S. 59–68.

[SLG10] Bin Shao, Du Li und Ning Gu. "A Fast Operational Transformation Algorithm for Mobile and Asynchronous Collaboration". In: *IEEE Trans. Parallel Distrib. Syst.* 21.12 (Dez. 2010), S. 1707–1720.

[Svn] *Subversion*. URL: `http://subversion.apache.org` (besucht am 14. 05. 2012).

[Twi] *TWiki*. URL: `http://www.twiki.org` (besucht am 14. 05. 2012).

[Ukk85] Esko Ukkonen. "Algorithms for approcimate string matching". In: *Information and Control* 64 (1985), S. 110 –118.