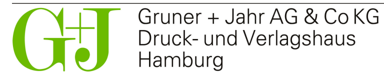


FH NORDAKADEMIE
Fachbereich Wirtschaftsinformatik
Köllner Chaussee 11
25337 Elmshorn

Gruner+Jahr AG & Co KG
CPS IS/IT ASD
Am Baumwall 11
20459 Hamburg

Diplomarbeit Wirtschaftsinformatik

Unscharfe Suche in großen Adressbeständen

Jochen Schulz
Matrikelnr.: 1882

29. August 2006

Erstgutachter: Prof. Dr.-Ing. Johannes Brauer
Zweitgutachter: Helmut Guttenberg
Betrieblicher Betreuer: Ricardo Nebot

Inhaltsverzeichnis

1. Einleitung **1**

2. Motivation **3**

2.1. Adressverwendung . . . 3
2.1.1. Mailings . . . 3
2.1.2. Abgleich mit Negativlisten . . . 5
2.1.3. Gewinnspiele und Fragebögen . . . 5
2.2. Beschreibung der Adressdaten . . . 6
2.2.1. Struktur . . . 6
2.2.2. Qualität . . . 7
2.2.3. Vorliegende Testdaten . . . 8
2.2.4. Fehlerklassen- und ursachen . . . 8
2.3. Aktuelle Lösung . . . 9
2.3.1. Vorbereitungen . . . 10
2.3.2. Regelsystem . . . 10
2.3.3. Ergebnisse . . . 11
2.4. Anforderungen an Eigenentwicklung . . . 11

3. Lösungsansätze **13**

3.1. Ursprünge . . . 13
3.2. Distanzfunktionen . . . 15
3.2.1. Levenshtein . . . 16
3.2.2. Damerau-Levenshtein . . . 19
3.2.3. Hamming . . . 19
3.3. Metrische Räume . . . 20
3.3.1. Definition . . . 20
3.3.2. Burkhard-Keller-Baum . . . 22
3.3.3. Vantage-Point-Baum . . . 24
3.3.4. Bisector-Baum . . . 26
3.3.5. Weitere Ansätze und Überblick . . . 28

4. Implementierung **30**
4.1. Überblick . . . 30
4.2. Schnittstellen . . . 31
4.2.1. Distanzfunktionen . . . 31
4.2.2. Metrische Räume . . . 31
4.3. Implementierungsdetails . . . 33
4.3.1. Distanzfunktionen . . . 33
4.3.2. Metrische Räume . . . 34
4.4. Anwendung . . . 35
4.5. Abgleiche . . . 36
4.5.1. Parameter des Vergleichs . . . 37
4.5.2. In-Sich-Abgleich . . . 38
4.5.3. Abgleich zweier Adressmengen . . . 40

5. Auswertungen **42**
5.1. Laufzeitverhalten . . . 42
5.1.1. Testumgebung . . . 42
5.1.2. Distanzfunktionen . . . 42
5.1.3. Metrische Räume . . . 44
5.2. Abgleiche . . . 49
5.2.1. Vorbereitungen . . . 49
5.2.2. Geeignete Schranken . . . 50
5.2.3. Optimale Prüfreihenfolge . . . 53
5.3. Vergleich mit ClickIt . . . 54
5.3.1. Laufzeit . . . 54
5.3.2. Qualität . . . 55

6. Zusammenfassung **58**
6.1. Bewertung . . . 58
6.2. Optimierungspotentiale . . . 59

Literaturverzeichnis **iv**

A. Quellcode **viii**
A.1. com.guj.searching.metric.distance . . . viii
A.1.1. MetricDistance.java . . . viii
A.1.2. SimpleLevenshtein.java . . . x
A.1.3. DamerauLevenshtein.java . . . xii
A.1.4. Hamming.java . . . xv
A.2. com.guj.searching.metric.tree . . . xvi

A.2.1. MetricTree.java xvi
A.2.2. BKTree.java xxiii
A.2.3. VPTree.java xxvii
A.2.4. BSTree.java xxxii
A.3. com.guj.searching.address xxxvii
A.3.1. Address.java xxxvii
A.3.2. AddressDistance.java xli
A.3.3. FullnameDistance.java xliii
A.3.4. PostalAddressDistance.java xliii
A.3.5. ComparisonPlan.java xliv

Darstellungsverzeichnis

Abbildungen und Diagramme

3.1. Levenshtein-Matrix 17
3.2. Beispiel eines Burkhard-Keller-Baums 23
3.3. Beispiel eines Vantage-Point-Baums 26
3.4. Beispiel eines Bisector-Baums 27

5.1. Laufzeitverhalten der Distanzfunktionen 43
5.2. Laufzeit der Baumkonstruktion pro Knoten 45
5.3. Verhalten bei der Suche abhängig von k 48

Tabellen

2.1. Adressbestandteile und ihre Längen 7
2.2. Antizipierte Fehlerklassen 9

3.1. Eigenschaften der vorgestellten Bäume 29

5.1. Eigenschaften der Suche nach Adressmerkmalen 54
5.2. Vergleich In-Sich-Abgleiche 56

Codelistings

4.1. Die Klasse `MetricDistance<T>` und eine Unterklasse . . 32
4.2. Ausschnitt aus dem Interface der Klasse `MetricTree<T>` 32
4.3. Eine metrische Distanzfunktion für `Address`-Objekte . . . 36
4.4. Ausschnitt aus der Klasse `ComparisonPlan` 38

1. Einleitung

Diese Arbeit befasst sich mit dem Problem des Abgleichs von postalischen Adressen aus unterschiedlichen Datenquellen. Es wird ein System vorgestellt, das in der Lage ist, zwei große Mengen von Adressen derart zusammenzuführen, dass anschließend möglichst wenige Adressen mehrfach vorhanden sind. Das System wird weiterhin auf seine Eignung hin überprüft, einzelne Ad-hoc-Abfragen auf der Basis von beliebigen Adressfragmenten schnell mit einer Liste möglicher Treffer beantworten zu können.

Die Schwierigkeit dieser Aufgabe liegt darin, dass die vorliegenden Daten zwar ein ähnliches (oder leicht ineinander überführbares) Format haben, aber nicht davon ausgegangen werden kann, dass übereinstimmende Datensätze immer exakt identisch sind. Vielmehr ist mit einer erheblichen Anzahl von Abweichungen aufgrund unterschiedlicher Fehlerquellen zu rechnen. Es existieren am Markt viele Softwarelösungen für dieses Problem, jedoch sind diese allesamt proprietär und das zugrunde liegende Modell liegt nicht offen. Hinzu kommt, dass kommerzielle Software umfangreiches domänenspezifisches Wissen über die Struktur und den Inhalt von Adressen nutzt.

Die zur Problemlösung herangezogenen Methoden entstammen daher dem allgemein gehaltenen Forschungsgebiet der unscharfen Suche in langen Texten oder großen Datenbanken. Auf diesem Gebiet, das fast genau so alt ist wie der Begriff „Informatik“ selbst und dennoch weiterhin äußerst aktiv bearbeitet wird, kann auf reichhaltiges, frei zugängliches Forschungsmaterial zurückgegriffen werden. Untersucht wird die Anwendbarkeit metrischer Abstandsfunktionen und die Möglichkeiten der Beschleunigung der Suche durch metrische Indexstrukturen.

Die Arbeit ist wie folgt gegliedert:

Abschnitt 2 schildert die fachliche Motivation dieser Arbeit. Er beschreibt konkrete Anwendungsfälle und nennt technische und andere relevante Rahmenbedingungen. Die Eigenschaften und Fähigkeiten der aktuell eingesetzten Lösung und die Anforderungen an die Eigenentwicklung werden dargestellt.

Abschnitt 3 stellt metrische Distanzfunktionen und Indexstrukturen vor, deren Eignung zum Adressabgleich untersucht wird.

Abschnitt 4 stellt die in Java implementierte Lösung mit besonderem Bezug auf das Zusammenspiel der dargestellten Lösungsansätze vor. Es wird gezeigt, wie die Implementierung zum Abgleich von Adressbeständen benutzt werden kann.

Abschnitt 5 analysiert das Verhalten der Implementierung in Bezug auf Laufzeit und Ergebnisqualität anhand von experimentellen Messdaten. Problematische Fehlerklassen werden identifiziert und quantifiziert. Es wird aufgezeigt, inwieweit die dargestellten Ansätze in der Lage sind, die real auftretenden Fehlerklassen auszuschließen.

Abschnitt 6 fasst schließlich die Ergebnisse zusammen und bewertet diese. Verbesserungsmöglichkeiten und offene Fragen werden diskutiert und ein Vorschlag für zukünftige Arbeiten entworfen.

2. Motivation

2.1. Adressverwendung

Die Gruner + Jahr AG & Co KG (im Folgenden: Gruner + Jahr) ist einer der größten Verlage für Publikumszeitschriften in Europa. Circa 2,9 Millionen Abonnenten werden regelmäßig mit aktuellen Ausgaben der rund 285 Titel beliefert. Aus dieser Tätigkeit ergibt sich ein ständiger Umgang mit einer großen Zahl von Adressen, an die ein hoher Qualitätsanspruch gestellt wird. Das wichtigste Qualitätsmerkmal einer Adresse ist die *Zustellfähigkeit*. Sie beantwortet die Frage, ob die Post oder ein anderer Botendienst in der Lage ist, auf der Basis der gemachten Angaben ein Schriftstück erfolgreich dem beabsichtigten Empfänger zuzustellen. Die Zustellung der Mehrheit der Magazine von Gruner + Jahr wird von der Deutschen Post AG erledigt.

Neben der Inrechnungstellung und Zustellung der Zeitschriften werden die Adressdaten von Abonnenten zu Marketingzwecken verwendet. Der Zeitschriftenvertrieb selbst, aber auch absatzunterstützende Maßnahmen, werden seit dem Frühjahr 2006 von der neu gegründeten, einhundertprozentigen Gruner + Jahr-Tochter Deutscher Pressevertrieb (DPV) geleistet. Dazu zählt auch das sogenannte Direktmarketing, also das persönliche Bewerben einer Vielzahl von Personen über verschiedene Kanäle. Die meisten Kontakte finden schriftlich auf dem Postweg statt.

2.1.1. Mailings

Zum Zwecke der Kundenaquise veranstaltet der DPV regelmäßig sogenannte *Mailings*, die allein im letzten Jahr 6,4 Millionen Haushalte erreichten – Tendenz steigend. Ein Mailing ist der einmalige postalische Versand einer (im Gegensatz zur Postwurfsendung) *personalisierten* Werbebotschaft an

mehrere Empfänger mit der Absicht

- neue Kunden zu gewinnen,
- Bestandskunden ein weiteres Abonnement zu verkaufen,
- Bestandskunden zu überzeugen, ihre Abonnements noch nicht zu kündigen,
- Kunden, die ihr Abonnement kürzlich gekündigt haben, zurückzugewinnen (sogenannte „Kündigerrückgewinnung"), oder
- die Kundenbindung zu stärken.

Für diese Zwecke greift Gruner + Jahr sowohl auf Stammdaten aktueller und ehemaliger Kunden zurück, als auch auf von externen Dienstleistern (sogenannte Adress- oder Listbroker) gemietete[1] oder gekaufte Adressen. Die Verlagsleitungen[2] bestimmen die Anzahl und das Profil der anzuschreibenden Personen oder Haushalte. Anhand der in der Kundendatenbank verfügbaren Kriterien ermittelt die Abteilung „Database Marketing" des DPV mit Hilfe verschiedener Verfahren diejenigen Adressen, die dem geforderten Profil möglichst genau entsprechen. Ist die Anzahl der auf diese Weise gefundenen Adressen kleiner als das von der Verlagsleitung geforderte Minimum, werden für dieses Mailing zusätzlich Adressen eingekauft oder gemietet.

Aus Gründen des Datenschutzes und nicht zuletzt auch, um die vertragswidrige Weiterbenutzung gemieteter Adressen auszuschließen, werden gemietete Adressen nicht direkt an werbende Unternehmen wie Gruner + Jahr weitergegeben. Stattdessen erledigt ein weiterer Dienstleister, der sogenannte „Lettershop", den Abgleich mit dem eigenen Bestand. Währenddessen müssen hinzugekaufte Adressen innerhalb des Hauses zum Kundenstamm

[1] Eine Adresse wird als „gemietet" bezeichnet, wenn das Verwendungsrecht für eine begrenzte Zahl Werbesendungen vom Besitzer beziehungsweise vom als Mittler auftretenden Adressbroker gewährt wurde.

[2] Als Verlagsleiter werden diejenigen Personen bezeichnet, die für eine *Familie* von Zeitschriften verantwortlich sind, etwa die stern-Gruppe oder die Food-Titel. Gemeint ist nicht die Leitung des Verlags*hauses*.

hinzugefügt werden. Dabei ist es wichtig zu prüfen, ob die neu hinzukommenden Adressen bereits im Bestand sind, oder nicht, denn doppelte Anschreiben verursachen unnötige Kosten und verärgern den Empfänger.

2.1.2. Abgleich mit Negativlisten

Verbraucher haben die Möglichkeit, ihre Daten in der sogenannten „Robinsonliste“ eintragen zu lassen. Diese Liste wird vom Deutschen Direktmarketing Verband e.V. (DDV) gepflegt und enthält die Kontaktdaten von Verbrauchern, die keine Werbung per Post, Telefon oder E-Mail erhalten möchten. Die Mitglieder des DDV, darunter Gruner + Jahr, haben sich verpflichtet, diesem Wunsch zu folgen. Dementsprechend muss vor dem Versand ein sogenannter *Negativabgleich* mit der Robinsonliste durchgeführt werden. Das bedeutet, vor einem Mailing muss sichergestellt werden, dass *keine* der Adressen in der Robinsonliste angeschrieben wird.

Wie in allen vorangegangenen Fällen, ist auch hier darauf zu achten, dass die abzugleichende Liste nicht unbedingt exakte Übereinstimmungen mit dem eigenen Bestand aufweisen wird.

2.1.3. Gewinnspiele und Fragebögen

Alle Magazine des Hauses veranstalten in mehr oder weniger regelmäßigen Abständen Gewinnspiele. Gelegentlich werden auch Umfragen veranstaltet, in denen Leser nach ihren Meinungen oder Einschätzungen befragt werden. Einer der Hauptgründe für diese und ähnliche Aktionen ist die Akquise neuer Adressdaten.

Die auf diese Weise generierten Adressen müssen innerhalb des Hauses mit dem Bestand zusammengeführt werden. Gerade Adressen aus derartigen Quellen müssen besonders gründlich überprüft werden, da ihre Qualität oftmals fragwürdig ist und einige Personen mehrfach an ein und demselben Gewinnspiel teilnehmen.

2.2. Beschreibung der Adressdaten

2.2.1. Struktur

Da die zu vergleichenden Adressdaten aus verschiedenen Quellen stammen, sind die Adressen nicht einheitlich strukturiert. Tabelle 2.1 enthält den kleinsten gemeinsamen Nenner der verwendeten Felder und einige Informationen über ihre Länge.[3] Weggelassen wurden folgenden Felder:

Adressart Unterscheidet Privat- und Geschäftsadressen. Diese Information ist zwar in den Gruner + Jahr-Datenbanken enthalten, jedoch nicht immer in den hinzukommenden Daten.

Anrede Enthält Bezeichnungen wie „Herr“, „Frau“oder „Firma“. Dieses Feld ist ebenfalls in den Gruner + Jahr-Datenbanken enthalten, aber nicht zwingend in Fremddaten.

Titel Gibt – sofern vorhanden – akademische oder Adelstitel an. Dieses Feld wird nur selten separat gespeichert. Stattdessen sind diese Informationen meist in den Namensfeldern enthalten.

Abteilung Spezifiert eine Abteilung innerhalb einer Firma. Dieses Feld ist in eigenen und externen Datensätzen oft enthalten. Jedoch ist es extrem selten gefüllt und eine Durchsicht ergab inhaltlich stark divergierende Nutzungen dieses Feldes. Ein Vergleich von Adressen auf Basis dieses Feldes erscheint somit nicht sinnvoll.

Namens- und Straßenzeilen In diesen und ähnlichen Feldern sind oft im Voraus aus anderen Feldern berechnete Werte enthalten. Meist sind sie – wie im Falle der Namenszeile – eine einfache Verkettung von Anrede, Vor- und Nachname und eventuell der Firma. Da diese Felder nur redundante Informationen enthalten, werden sie nicht weiter betrachtet.

[3] Alle Zahlen in dieser Tabelle beziehen sich auf die in Abschnitt 2.2.3 beschriebenen Testdaten.

Feld	Maximallänge	∅ Länge
Vorname	30	6,09
Nachname	30	7,11
Namensanhang	30	0,67
Firma	30	0,32
Straße (inkl. Hausnr.)	39	15,33
Postleitzahl	10	5,00
Ort	38	8,58
Land (Kürzel)	11	1,01
Voller Name (1-4)	79	13,94
Anschrift (5-8)	57	27,64
Gesamt	102	41,58

Tabelle 2.1.: Adressbestandteile und ihre Längen

2.2.2. Qualität

Für die Beurteilung der Qualität von Adressdaten ist ihre Herkunft entscheidend. Der Großteil der Gruner + Jahr-eigenen Daten ist naturgemäß guter Qualität, da die Erbringung der Dienstleistung Zeitschriftenversand ohne zustellfähige Adressen nicht möglich ist. Allerdings ist hier zwischen den Adressen aktiver und inaktiver Kunden zu unterscheiden. Aktive Kunden sind Inhaber eines laufenden Abonnements, inaktive Kunden hatten zu einem Zeitpunkt in der Vergangenheit ein Abonnement, haben aber inzwischen gekündigt. Ihre gespeicherten Adressen können daher unbemerkt unzustellbar werden. Die Adressen inaktiver Kunden werden daher in regelmäßigen Bereinigungsläufen aus dem Kundenstamm gelöscht.

Zudem ist die mehrfache Erfassung derselben Person möglich. Zwar hat jeder Abonnent eine eindeutige Nummer, die auch auf seinen Rechnungen angegeben ist, jedoch gibt ein aktiver oder inzwischen inaktiver Kunde diese bei Abschluss eines neuen Abonnements nicht unbedingt an. Macht er bei Vertragsschluss Angaben mit Abweichungen von den bisher gespeicherten, kommt es vor, dass die Daten ein und derselben Person mehrfach gespeichert werden.

Von regelmäßig schlechter Qualität sind aus Gewinnspielen und Umfragen akquirierte Daten. Hier kommt es häufig zu Mehrfachteilnahmen, die Angaben sind unvollständig oder fehlerhaft. Offensichtlich falsche Daten werden bereits in der Erfassungsphase gefiltert, aber nicht immer sind falsche Angaben eindeutig zu erkennen.

2.2.3. Vorliegende Testdaten

Untersuchungsgegenstand sind zwei Adressdateien, wie sie für ein Mailing tatsächlich verwendet wurden. Etwa 350.000 Adressen stammen aus dem eigenen Bestand und weitere 80.000 entstammen externen Quellen. Alle Dateien liegen im CSV-Format[4] vor, da praktisch alle Datenbanksysteme dieses Format erzeugen und die gängigen Büroprogramme (Microsoft Excel, Openoffice Math etc.) es verarbeiten können. Zudem macht es das CSV-Format einfach, Unterschiede in der Strukturierung der Daten auszugleichen.

Zusätzlich zu den Rohdaten liegt das Ergebnis eines von mit Hilfe der aktuell von Gruner + Jahr eingesetzten Softwarelösung (s.u.) durchgeführten Abgleichlaufs vor. Das Ergebnis setzt sich aus den jeweils in-sich abgeglichenen Dateien und den Schnittmengen beider Dateien zusammen.

2.2.4. Fehlerklassen- und ursachen

Ein Problem dieser Arbeit liegt in der Tatsache, dass die genaue Art und Anzahl der Abweichungen zwischen identischen Adressen nicht im Voraus bekannt ist. Eine Analyse der auftretenden Fehlerklassen ist wiederum ohne ein System, das in den Testdaten tatsächlich auftretende Abweichungen zwischen äquivalenten Adressen finden kann, kaum zu leisten.

Aus diesem Grund bleibt an dieser Stelle nur, möglicherweise auftretende Fehlerklassen zu antizipieren. Tabelle 2.2 enthält eine Übersicht derjenigen Fehler, für die in den folgenden Kapiteln Erkennungsmöglichkeiten gefunden werden sollen. Einige davon sind leicht durch entsprechende Vor-

[4] CSV steht für *comma-separated values* und ist ein beliebtes, in [Sha05] definiertes Datenaustauschformat. In der Praxis werden allerdings häufig andere Trennzeichen zwischen den Datenfeldern (Semikola, Tabulatoren etc.) verwendet.

Feld(er)	Fehlerart	Beispiel
alle	Groß- / Kleinschreibung	Hamburg vs. HAmburg
	Buchstabendreher	Schmidt vs. Schmitd
	Hörfehler	Hamburg vs. Harburg
	Schreibfehler	Hammburg vs Hamburg
	Leerzeichen	Poststr. 11 vs Poststr.11
Straße	Abkürzungen	Str. vs. Straße
	Hausnummer	11 vs. 11a
Name	Rufname unvollständig	P. Schmidt vs. Peter Schmidt
	Mittelinitial	Jan P. Aue vs. Jan Aue
	Kurzformen	Steffi vs. Stefanie
	Schreibweise	Stefanie vs. Stephanie

Tabelle 2.2.: Antizipierte Fehlerklassen

bereitung der Daten von vornherein zu vermeiden. Beispielsweise können Unterschiede in der Großschreibung dadurch ignoriert werden, dass vor dem Vergleich alle Adressbestandteile in entweder Groß- oder Kleinschreibung überführt werden. Leerzeichen können entfernt und gängige Abkürzungen durch Ersetzungen vereinheitlicht werden.

Gegen andere Fehler kann dagegen nicht präventiv vorgegangen werden. Insbesondere Buchstabendreher, Schreib- und Hörfehler können nicht ohne Weiteres beseitigt werden.

2.3. Aktuelle Lösung

Für die erläuterten Marketingaktivitäten wird aktuell das Produkt *ClickIt* der Firma Uniserv[5] eingesetzt. ClickIt ist eine Client-Server-Anwendung. Ein dediziertes Serversystem (Dell Poweredge, 2×3,6 GHz Intel CPU, 8 GB RAM) leistet die eigentliche Arbeit, wird aber von einem Clientprogramm auf dem Arbeitsplatz-PC der Benutzer gesteuert.

[5] http://www.uniserv.de

ClickIt speichert die hauseigenen Kundenstammdaten in einer eigenen Datenbank auf dem Serversystem. Externe Daten können auf dieses System hochgeladen werden und stehen damit für Abgleiche zur Verfügung.

2.3.1. Vorbereitungen

Vor einem Abgleich einer oder mehrerer Dateien muss ClickIt erfahren, in welchen Feldern der Dateien welche Adressbestandteile gespeichert sind. Diese Zuordnung kann ClickIt durch eine Inhaltsanalyse weitgehend selbst vornehmen. Es kann beispielsweise erkennen, ob ein Feld Vor- oder Nachnamen oder beides gleichzeitig enthält. Genauso verhält es sich mit postalischen Anschriften in Bezug auf Straße, Hausnummer, Postleitzahl und Ort. Auf diese Weise können auch einzelne Teile der Adressen separat behandelt werden, auch wenn sie in der Quelldatei gemeinsam in einem Feld gespeichert wurden. Die automatisch vorgenommen Zuordnungen können natürlich auch von Hand korrigiert werden.

Alle Zuordnungen gelten für alle Adressen einer Datei. ClickIt führt keine Analyse eines jeden einzelnen Datensatzes durch. Dementsprechend müssen alle Datensätze innerhalb einer Datei einheitlich formatiert sein.

2.3.2. Regelsystem

Die Erkennung von Dubletten basiert in ClickIt auf einem punktebasierten Regelsystem. Für jeden Adressbestandteil kann angegeben werden, wieviele Punkte für eine genaue Übereinstimmung, ungefähre Übereinstimmung oder einen „Überhang“[6] vergeben werden und welche Gewichtung er hat. Die Gesamtpunktzahl einer Adresse wird durch den gewichteten Mittelwert aller Bestandteile ermittelt. Schließlich wird eine Minimalpunktzahl angegeben, ab der eine Adresse als Dublette einer anderen gewertet wird.

Die genaue Definition der „ungefähren Übereinstimmung“ von Adressbestandteilen in ClickIt ist ein Betriebsgeheimnis von Uniserv. Hier wird aller Wahrscheinlichkeit nach eine beträchtliche Menge an domänenspezifischem

[6] Ein Überhang liegt vor, wenn zwei Zeichenketten nicht gleich lang, aber in den ersten gemeinsamen Zeichen identisch sind.

Wissen eingeflossen sein. Neben phonetischen und optischen Charakteristika spielen eventuell auch Listen ähnlicher Wörter, Synonyme, gebräuchliche Abkürzungen und Ähnliches eine Rolle.

Uniserv stellt eine Reihe vordefinierter Regelsätze für verschiedene Anwendungszwecke zur Verfügung und erlaubt das Anlegen und Speichern eigener Regelsätze. So können Privatadressen auf der Ebene „Einzelperson“, aber auch „Haushalt“ abgeglichen werden. Dahinter steht der Wunsch, mehrere in einem Haushalt zusammenlebende Personen nicht mehrfach anzuschreiben. Firmenadressen können auf Firmenebene, aber auch auf der Basis einer Firma und aller bekannten Ansprechpartner angeschrieben werden. Ob eine Firma nur einmalig, oder jeder einzelne bekannte Ansprechpartner angeschrieben werden soll, hängt vom Inhalt des Mailings ab und ist eine Vorgabe der Verlagsleitungen.

ClickIt erlaubt auch, Einflussnahme auf die Wahl eines bestimmten Repräsentanten aus als Dubletten erkannten Adresstupeln zu nehmen. Hier besteht die Wahl zwischen der als neuer bekannten Adresse, der vollständigeren Adresse oder es wird einfach der Adresse aus einer bestimmten Datei immer der Vorzug gegeben.

2.3.3. Ergebnisse

Schließlich ermöglicht ClickIt, exakt die gewünschten Ausgaben zu spezifizieren. Das bedeutet, für jede Zweierkombination aller Eingabedateien können beliebige Ergebnisse angefordert werden:

- Schnittmenge beider Dateien
- Erste Datei ohne Adressen in zweiter Datei und umgekehrt (Negativabgleich)
- In-sich abgeglichene Dateien

2.4. Anforderungen an Eigenentwicklung

Im Rahmen einer solchen Arbeit läßt sich ein ausgereiftes System wie ClickIt nicht vollständig nachbilden. Daher müssen einige Einschränkungen gemacht

werden. Die schwerwiegendste Einschränkung bezieht sich auf die Wahl der Erkennungsmethoden übereinstimmender Adressbestandteile. Hier wird sich auf Funktionen konzentriert, die alle Eigenschaften einer Metrik erfüllen. Ein Grund dafür ist, dass für solche Funktionen effiziente Indexstrukturen existieren, die die Suche in großen Datenbeständen in annehmbarer Zeit erst möglich machen. Ein weiterer Vorteil dieser Funktionen ist, dass sie keinerlei Wissen über die Natur der Daten voraussetzen. Die phonetische Suche, also die Suche nach gleich klingenden Wörtern, findet deswegen keine Betrachtung. Jeder Algorithmus, der dies leistet, muss auf eine bestimmte Sprache angepaßt werden und die gängigen, auch in vielen Datenbanksystemen vorhandenen Funktionen, sind meist auf die englische Sprache zugeschnitten. Hinzu kommt, dass auch im deutschen Sprachraum Namen aus anderen Regionen der Welt immer häufiger werden. In diesen Fällen würde eine phonetische Suche prinzipbedingt versagen

Weitere Einschnitte sind in der Funktionalität der Software zu machen. Ziel ist es nicht, eine fertige Anwendung bereitzustellen, die alle Arbeitsschritte von der Vorbereitung der Daten über das Konfigurieren eines Regelsystems bis hin zur Unterstützung aller möglichen Formen von Abgleichen bietet. Stattdessen erfolgt eine Konzentration auf die wesentlichen Kernfunktionalitäten, auf deren Basis später ein System zum automatischen oder interaktiven Abgleich und der Einzelsuche aufgebaut werden kann.

Im Ergebnis bedeutet dies, dass lediglich eine Bibliothek entwickelt wird, die Distanzfunktionen, metrische Indexstrukturen und Methoden zum Abgleichen zweier Adressdateien enthält. Die Vorbereitung der Daten wird an den Testdaten einmalig händisch erledigt, die Definition angemessener Abgleichsregeln für jeden Anwendungsfall wird weitestgehend von der Implementation offen gelassen. Auf eine geschickte Wahl eines besten Repräsentanten unter äquivalenten Adressen wird verzichtet und die Präsentation der Funktionalitäten mit Hilfe einer geeigneten Benutzerschnittstelle ausgeklammert.

Die einzige Beschränkung für die maximale Laufzeit soll sein, dass ein Abgleich von einigen hunderttausend bis zu zwei Millionen Adressen innerhalb weniger Stunden, etwa einer Nacht, erfolgen soll.

3. Lösungsansätze

3.1. Ursprünge

Das Problem effizienter Suche gehört zu den fundamentalen und am besten untersuchten Problemen der Informatik. Traditionell werden hauptsächlich strukturierte Daten nach *exakten* Übereinstimmungen mit einem gegebenen Suchbegriff durchsucht. Dies ist beispielsweise in Datenbanken der Fall, wenn ein Tupel mit einem bestimmten Primärschlüssel gesucht wird.

Zu den bekanntesten Suchalgorithmen gehören KMP [CR03, Kapitel 2.1] (benannt nach den Entdeckern Knuth, Morris und Pratt) und der vom UNIX-Standardtool `grep` verwendete Boyer-Moore-Algorithmus [NR02, Kapitel 2.3.1] aus dem Jahre 1977, die jeweils einen (potentiell langen) unstrukturierten Text nach einer beliebigen Zeichenkette durchsuchen und die Position von exakten Übereinstimmungen ermitteln. Die Laufzeit der Suche hängt bei beiden Algorithmen in der asymptotischen Betrachtung linear von der Länge des zu durchsuchenden Textes ab.

Eine sublineare Laufzeit der Suche kann durch vorheriges Sortieren der Daten erreicht werden (bspw. durch Suche in sortierten Binärbäumen, siehe [Knu98, Kapitel 6.2.2]). Das Sortieren selbst benötigt allerdings eine Laufzeit von $\mathrm{O}(n \log(n))$.[1] Dieser Aufwand wäre im vorliegenden Fall aufgrund der häufig durchgeführten Suchen wahrscheinlich zu rechtfertigen. Jedoch scheitert schon die Sortierung an dem Fehlen einer totalen Ordnung über alle möglichen Zeichenketten, die „ähnlich“ aussehende Zeichenketten nah beieinander sortiert.

Ein anderer Ansatz sind die u.A. in [Knu98, Kapitel 6.4] beschriebenen

[1] Hier und im Folgenden wird die gebräuchliche Landau-Notation (auch: Groß-O-Notation) verwendet, um algorithmische Komplexitäten anzugeben. Wenn nicht anders angegeben, bezieht sich die genannte Komplexität auf die Laufzeit des Algorithmus.

Hashtables, auch Dictionaries genannt, die mittels einer Hashfunktion, angewandt auf ein zu suchendes Element, mit konstantem Zeitaufwand die Position des gesuchten Suchbegriffs im Speicher ermitteln. Allerdings sind Hashfunktionen gerade darauf ausgelegt, auch bei nur kleinen Variationen des Eingabeparameters möglichst unterschiedliche Ergebnisse zu liefern. Daher ist auch der Einsatz von Hashtables für unscharfe Suche ungeeignet.

Die unscharfe Suche (im Zusammenhang mit Zeichenketten auch *fuzzy search*), also die Suche nach einem Element, das eine gewisse *Ähnlichkeit* zu einem gegebenen Suchbegriff hat, spielt seit den 1960er Jahren eine zunehmend wichtige Rolle in verschiedenen Anwendungsgebieten. Neben dem zu dieser Arbeit artverwandten Problem der unscharfen Suche von Zeichenketten in großen Texten sind nach [Nav01, Kapitel 2] besonders die Felder Molekularbiologie und Signalverarbeitung hervorzuheben.

In der Molekularbiologie werden DNA- und Proteinsequenzen untersucht, die sich selten exakt gleichen. Diese Sequenzen können als Texte über einem bestimmten Alphabet ausgedrückt werden. Eine fachspezifische Definition der Distanz und ein Algorithmus, der diese berechnet, ermöglichen eine Schätzung der evolutionären „Entfernung“ zweier Spezies. Die Signalverarbeitung wiederum beschäftigt sich seit Langem mit der Fehlerkorrektur physikalisch übertragener Information, da derartige Übertragungen – abhängig vom Medium – von schlechter Qualität sein können. Die maschinelle Transformation gesprochener Sprache in Text fällt ebenfalls in dieses Gebiet. Dieses Problem ist auch trotz aktueller Fortschritte noch nicht zufriedenstellend gelöst.

In den letzten zwanzig Jahren ist das Interesse an Lösungen für derartige Probleme drastisch gestiegen. Der Grund liegt vor Allem in der Allgegenwärtigkeit von informationsverarbeitenden Maschinen. Die Menge verfügbarer Texte ist unüberschaubar gewachsen, gleichzeitig sind die Datenbestände in Sprache und Struktur heterogener und fehlerbehafteter geworden. Als Beispiel sind hier Systeme zur optischen Zeichenerkennung (OCR) oder das World Wide Web zu nennen. Eine exakte Suche schließt in solchen Beständen oft einen nichtvertretbaren Anteil wertvoller Ergebnisse aus. Suchmaschinen für das WWW bieten daher beispielsweise inzwischen ei-

ne Art Rechtschreibkorrektur an, wenn der eingegebene Suchbegriff einem häufiger vorkommenden Begriff sehr ähnlich ist (etwa „Did you mean“ bei Google). Die Suchergebnisse basieren aber weiterhin auf exakten Treffern.

3.2. Distanzfunktionen

Distanzfunktionen sind Funktionen, die den Grad der Unterschiedlichkeit zweier Zeichenketten ausdrücken. Allgemein läßt sich eine Distanzfunktion d wie folgt definieren:

$$d : \Sigma^* \times \Sigma^* \to \mathbb{R}$$

Dabei ist Σ ein beliebiges, endliches Alphabet und Σ^* bezeichnet alle möglichen Wörter (Zeichenketten) über diesem Alphabet. Eine Distanzfunktion bildet also jeweils zwei Zeichenketten auf eine reelle Zahl ab. Diese Zahl beschreibt, wie stark sich diese Zeichenketten voneinander unterscheiden.

In dieser Arbeit werden ausschließlich Distanzfunktionen betrachtet, die alle Eigenschaften einer *Metrik* erfüllen. Dies sind:

$$\forall x, y \in \mathbb{X} : d(x, y) \geq 0 \qquad \text{(Positivität)}$$

$$\forall x, y \in \mathbb{X} : d(x, y) = 0 \Leftrightarrow x = y \qquad \text{(Strenge Positivität)}$$

$$\forall x \in \mathbb{X} : d(x, x) = 0 \qquad \text{(Reflexivität)}$$

$$\forall x, y \in \mathbb{X} : d(x, y) = d(y, x) \qquad \text{(Symmetrie)}$$

$$\forall x, y, z \in \mathbb{X} : d(x, z) \leq d(x, y) + \mathrm{d}(y, z) \qquad \text{(Dreiecksungleichung)}$$

Analog zu [Nav01, Kapitel 3.1] werden hier außerdem nur Distanzfunktionen betrachtet, die sich als ein Satz von Ersetzungesregeln oder -operationen formulieren lassen. Jede Ersetzungsregel überführt dabei eine Teilzeichenkette der einen in eine Teilzeichenkette der anderen Zeichenkette. Mit jeder Regel sind bestimmte Kosten assoziiert. Im einfachen Fall haben alle Regeln die gleichen Kosten. Die Distanzfunktion gibt in diesem Modell nun die minimalen Kosten zurück, die durch Anwendung von Regeln entstehen, die

die zwei Zeichenketten ineinander überführen.

3.2.1. Levenshtein

Die nach ihrem Entdecker benannte *Levenshteindistanz* (auch: *edit distance*), erstmals beschrieben in [Lev66], ist der De-facto-Standard um die gegenseitige „Entfernung“ zweier Zeichenketten auszudrücken. Sie ist informell definiert als die minimale Anzahl von Operationen auf einzelnen Zeichen, die nötig ist, um eine der beiden Zeichenketten in die andere zu überführen. Dabei sind nach Levenshtein drei Operationen erlaubt:

- Einfügen eines Zeichens
- Löschen eines Zeichens
- Ersetzen eines Zeichens durch ein anderes Zeichen

Berechnung

Der erste Algorithmus zur Berechnung der Levenshteindistanz basiert auf dem Prinzip der dynamischen Programmierung. Die grundlegende Idee lautet nach [Nav01, Kapitel 5] wie folgt: seien x und y zwei Zeichenketten der Länge $|x|$ und $|y|$. Soll die Distanz $\mathrm{ed}(x, y)$ berechnet werden, wird eine Matrix $C_{0..|x|,0..|y|}$ wie in Abbildung 3.1 gefüllt, deren Elemente $C_{i,j}$ die minimale Anzahl der benötigten Operationen enthält, die $x_{1..i}$ nach $y_{1..j}$ überführen. Die Berechnungsvorschrift ist rekursiv definiert:

$$
\begin{aligned}
C_{i,0} &= i \\
C_{0,j} &= j \\
C_{i,j} &= \text{wenn } (x_i = y_j) \text{ dann } C_{i-1,j-1} \\
&\qquad \text{sonst } 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})
\end{aligned}
$$

Die Zeile $C_{0..|x|,0}$ und die Spalte $C_{0,0..|y|}$ enthalten jeweils den Abstand der leeren Zeichenkette ε zum entsprechenden Präfix des anderen Wortes. Ist die

		M	E	I	E	R	S
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
A	2	1	1	2	3	4	5
Y	3	2	2	2	3	4	5
E	4	3	2	3	2	3	4
R	5	4	3	3	3	2	**3**

Abbildung 3.1.: Levenshtein-Matrix

Matrix vollständig gefüllt, steht das Ergebnis für $\mathrm{ed}(x, y)$ in $C_{|x|,|y|}$, also der rechten unteren Ecke (in der Darstellung durch Fettdruck hervorgehoben).

Diese Herangehensweise ist jedoch unter Umständen problematisch, denn für die Matrix C ergibt sich ein Speicherbedarf von $\mathrm{O}(|x||y|)$, die Laufzeit beträgt ebenfalls $\mathrm{O}(|x||y|)$, was unter der Annahme gleichlanger Zeichenketten einem quadratischen Wachstum entspricht. Immerhin läßt sich der Speicherbedarf leicht auf $\mathrm{O}(\min(|x|,|y|))$ reduzieren, indem die Matrix spalten- oder zeilenweise berechnet wird. Schließlich werden zur Berechnung jedes Elements $C_{i,j}$ nur die Nachbarelemente $C_{i,j-1}$ (darüber), $C_{i-1,j}$ (links daneben) und $C_{i-1,j-1}$ (links oben) benötigt, also muss C nicht vollständig im Speicher gehalten werden. Zudem sind die zu vergleichenden Adressbestandteile im Durchschnitt relativ kurz (siehe Abschnitt 2.2.1). Die quadratische Laufzeit muss daher nicht zwingend zu inakzeptablen Laufzeiten führen.

Für das Problem, Fundstellen eines gegebenen Suchworts mit einer maximalen Levenshteindistanz von k in einem langen Text zu finden, existiert eine große Zahl teilweise sehr effizienter Lösungen, die auf unterschiedlichen Ansätzen beruhen. Die hervorragende Zusammenfassung [Nav01] bietet zu diesem Thema einen umfassenden Überblick. Hervorzuheben ist hier insbesondere der erstmals in [NR98] beschriebene BNDM-Algorithmus (*Backward Nondeterministic Dawg Matching Algorithm*), der eine Vereinigung vorangegangener Ideen darstellt und nach Angabe der Autoren sowohl effizient als auch leicht zu implementieren ist. Wegweisend ist ebenfalls [WM91], was die Grundlage für die bekannte Erweiterung `agrep` („approximate grep“)

des UNIX-Standardtools `grep` ist.

Zur Berechnung der exakten Distanz zweier vollständiger Zeichenketten existieren ähnliche Ansätze (beispielsweise [Hyy03b]), allerdings eignen sich diese bisher nach Angabe der Autoren nur dazu, zu entscheiden, *ob* zwei Zeichenketten innerhalb einer gegebenen Entfernung zueinander liegen, oder nicht. Die tatsächliche Distanz können sie nicht ermitteln.

Eigenschaften

Die obere Schranke für die Levenshteindistanz ed() zweier beliebiger Zeichenketten x und y läßt sich angeben als $\max(|x|, |y|)$, also die Länge der längeren der beiden Zeichenketten. Diese Schranke wird erreicht, wenn x und y keine gemeinsamen Zeichen haben, also jedes Zeichen ersetzt werden muss. Sind die Zeichenketten nicht gleich lang, müssen zusätzlich Einfüge- oder Löschoperationen durchgeführt werden, um diesen Unterschied auszugleichen.

Die untere Schranke beträgt im allgemeinen Fall $|(|x| - |y|)|$. Dieser Fall tritt ein, wenn eine der beiden Zeichenketten eine Teilzeichenkette der Anderen ist. Die Differenz der Längen muss wieder durch Einfügen oder Löschen von Zeichen ausgeglichen werden. Gilt $|x| = |y|$, ist die untere Schranke Null. Levenshtein ergibt genau dann eine Distanz von Null, wenn die zu vergleichenden Zeichenketten identisch sind.

Approximation

Seit Kurzem existieren auch Ansätze, die Levenshteindistanz nicht genau zu berechnen, sondern nur näherungsweise. Der in [BEK+03] beschriebene Algorithmus erlaubt eine Abschätzung der Distanz in sublinearer Zeit durch Anwendung des *divide-and-conquer*-Prinzips und Berechnung der Distanz für zufällig ausgewählte Teilzeichenketten. Der vorgestellte Algorithmus beantwortet allerdings nur die Frage, ob zwei gegebene Zeichenketten „nah“ oder „fern“ voneinander liegen, wobei als Grad der Entfernung die Form n^α verwandt wird. Beide Zeichenketten haben die Länge n, α ist ein beliebiger Wert zwischen 0 und 1. Nach Angabe der Autoren ist der vorgestellte Algo-

rithmus besonders effizient für Werte von $\alpha \leq \frac{2}{3}$. Er kann also sehr schnell entscheiden, ob zwei Zeichenketten einander überhaupt ähnlich sind, oder nicht. Die Autoren machen leider keine Angaben über die Wahrscheinlichkeit der Korrektheit der Resultate.

3.2.2. Damerau-Levenshtein

Eine andere Möglichkeit der Variation der Levenshteindistanz ist das Erlauben von weniger oder mehr Operationen auf den Zeichenketten. Die *Damerau-Levenshtein-Distanz* [Dam64] erlaubt beispielsweise als zusätzliche Operation das Vertauschen von zwei benachbarten Zeichen zum gleichen Preis wie die übrigen Operationen. Nach [Dam64] erfassen die vier erlaubten Operationen 80% aller Schreibfehler, die von Menschen gewöhnlich gemacht werden.

Eine Erweiterung der bisher erfolgreichsten Algorithmen für die Levenshteinsuche um Berücksichtigung der Tauschoperation wird in [Hyy03a] vorgeschlagen und mit den ursprünglichen Algorithmen verglichen. Es zeigt sich, dass diese Modifikation asymptotisch keinen Einfluss auf die Laufzeit der Suche haben muss. Für die genaue Ermittlung der Distanz zweier Zeichenketten beschreibt [WL75] einen Algorithmus mit der gleichen Laufzeitkomplexität wie der Levenshteinalgorithmus ($O(nm)$), allerdings mit einem größeren konstanten Faktor.

3.2.3. Hamming

Ein weiteres bekanntes Maß für die Entfernung zweier Zeichenketten ist die *Hammingdistanz*. Sie ist nur für Zeichenketten gleicher Länge definiert und erlaubt ausschließlich Ersetzungen einzelner Zeichen, was diese Distanz in ihrer Einsatzfähigkeit stark beschränkt. Ihr Vorteil ist aber, dass sie äußerst effizient berechenbar ist. Liegen die Daten in Binärdarstellung vor, genügt die Anwendung der XOR-Funktion und das Abzählen der resultierenden Einsen, um sie zu ermitteln. Hier wird allerdings deutlich, dass die Ergebnisse aller vorgestellten Distanzmaße vom verwendeten Alphabet abhängig sind. Sollen ASCII-Zeichenketten verglichen werden, liefert die eben beschriebene

Berechnungsmethode keine sinnvollen Ergebnisse, denn zwei verschiedene ASCII-codierte Zeichen können sich in einem, aber auch mehreren Bits unterscheiden. In unserem Fall müsste also auf einen paarweisen Vergleich ganzer ASCII-Zeichen zurückgegriffen werden. Unabhängig vom verwendeten Alphabet bleibt die Laufzeit der Berechnung der Hammingdistanz aber linear abhängig von der Länge der zu vergleichenden Zeichenketten.

3.3. Metrische Räume

In vielen Fällen sind metrische Distanzfunktionen äußerst aufwändig zu berechnen. Als Beispiel kann hier die besprochene Levenshteindistanz mit ihrer quadratischen Laufzeitkomplexität dienen. Sind die zu durchsuchenden Texte oder Datenbanken sehr umfangreich, ist es nicht mehr praktikabel, jedes Wort oder jeden Datensatz mit einem gegebenen Suchbegriff zu vergleichen. Im Falle des batch-artigen Vergleichs zweier großer Datenbanken führt dieses Vorgehen zu nicht mehr akzeptablen Laufzeiten. Benötigt beispielsweise der Vergleich zweier Zeichenketten, die eine Adresse repräsentieren, nur zwanzig Mikrosekunden,[2] ergibt sich bei der vorhandenen Anzahl Adressen eine Laufzeit von $20 \times 10^{-6} \times 80.000 \times 350.000 \div 3600 \approx 155$ Stunden oder circa $6\frac{1}{2}$ Tagen.

Der Zweck metrischer Räume ist es, die zu durchsuchenden Objekte so anzuordnen, dass bei einer Suche möglichst wenig Distanzberechnungen ausgeführt werden müssen. Die in der Literatur angegebenen Laufzeitkomplexitäten geben daher meist nicht direkt die tatsächliche Laufzeit dieser Operationen an, sondern die Anzahl der Distanzberechnungen in Abhängigkeit von der Eingabemenge. Dies gilt auch für alle Angaben in den folgenden Abschnitten.

3.3.1. Definition

Ein Metrischer Raum ist ein Paar aus einer möglicherweise unendlichen Menge $\mathbb{X}$ und einer Metrik d, oder kurz: $(\mathbb{X}, d)$. Zur Erinnerung: eine Metrik

[2] Genauere Meßergebnisse folgen in Abschnitt 5.1.2.

ist eine Distanzfunktion mit bestimmten Eigenschaften, darunter die für die Indexierung eines metrischen Raums wichtigste Eigenschaft, die Erfüllung der Dreiecksungleichung (siehe Seite 15). Sie erlaubt, die Entfernung zweier Elemente aus $\mathbb{X}$ nach oben abzugrenzen, wenn die Entfernung beider Elemente zu einem Dritten bekannt ist. Dadurch kann bei einer Suche unter Umständen ein großer Teil des Suchraums ausgeschlossen werden, ohne dass die darin enthaltenen Elemente mit dem zu suchenden Element mit Hilfe der aufwändigen Distanzfunktion verglichen werden müssen. Diese Eigenschaft einer Funktion läßt sich sehr einfach beschreiben: seien x, y und z drei Punkte in einer Ebene (also einem zweidimensionalen euklidischen Raum) und liefere d den geometrischen Abstand zweier Punkte. Solange nicht alle drei Punkte auf einer Geraden liegen, ergibt sich ein Dreieck. Der Abstand zwischen Punkt x und Punkt z kann nicht länger sein, als der „Umweg“ von x nach z über den Punkt y. Nur für den Spezialfall, dass alle Punkte auf einer Geraden liegen, sind beide Abstände gleich lang. Auch die anderen Eigenschaften einer Metrik stimmen mit dem intuitiven Verständnis des Verhaltens einer Distanzfunktion überein.

Euklidische Räume mit n Dimensionen (auch $\mathbb{R}^n$, im genannten Beispiel galt $n = 2$) sind ein Spezialfall metrischer Räume. Für sie existieren viele Lösungen zur effizienten Indexierung. Zu den bekanntesten zählen kd-Bäume, beziehungsweise deren Verallgemeinerung, die BSP-Verfahren (Binary Space Partitioning). Die Effizienz dieser Verfahren hängt jedoch stark von der Anzahl der Dimensionen ab, woraus der Begriff vom „Fluch der Dimensionalität“ (engl.: *the curse of dimensionality*, [CNBYM01, Kapitel 4.0]) entstanden ist.

Zudem lassen sich nicht alle metrischen Räume auf einen n-dimensionalen Raum abbilden. Die Menge aller Zeichenketten über einem beliebigen endlichen Alphabet bildet mit einer geeigneten Funktion solch einen metrischen Raum. Trotzdem kann durch geschickte Anordnung der Zeichenketten in einem Index die Dreiecksungleichung benutzt werden, um eine erschöpfende Suche (engl.: *exhaustive search*) des vollständigen Suchraums zu vermeiden. Bedingung ist nur, dass für die Suche die gleiche Distanzfunktion benutzt wird, wie für den Aufbau des Index. Alle vorgestellten Distanzfunktionen

erfüllen die Bedingungen für einen Einsatz in metrischen Indexstrukturen.

Im Folgenden werden drei Indexstrukturen beschrieben, die eine Suche nach Elementen unterstützen, die ähnlich zu einem gegebenen Suchelement sind. Sie werden in Bezug auf die Indexierung von Zeichenketten beschrieben, sind aber auf alle denkbaren metrischen Räume anwendbar. Die vorgestellten Strukturen wurden auf der Basis ihrer Bekanntheit und der Häufigkeit der Erwähnungen in der gesichteten Literatur ausgewählt. Sie stellen alle Repräsentanten unterschiedlicher Lösungsansätze dar. Alternativen werden im darauf folgenden Abschnitt 3.3.5 aufgezeigt.

3.3.2. Burkhard-Keller-Baum

Die ersten Ideen zur Indexierung allgemeiner metrischer Räume stammen aus [BK73]. Dort werden drei Strukturen vorgeschlagen, darunter der anschließend nach den Autoren benannte *Burkhard-Keller-Baum* (kurz: BK-Baum). In der beschriebenen Form setzt der BK-Baum eine Distanzfunktion voraus, die diskrete Werte zurückliefert, beispielsweise nur natürliche Zahlen.

Die Konstruktionsvorschrift lautet folgendermaßen: sei $\mathbb{S} \subseteq \mathbb{X}$ eine endliche Menge aller zu indexierenden Elemente. Man wähle ein beliebiges Element $x' \in \mathbb{S}$. Die Restmenge $\mathbb{S} \setminus \{x'\}$ wird nun in disjunkte Teilmengen $\mathbb{S}^0$, $\mathbb{S}^1$, ..., $\mathbb{S}^m$ aufgespalten, so dass für alle $i = 0, 1, \ldots, m$ und $x \in \mathbb{S}^i$ gilt: $d(x, x') = i$. Dieser Vorgang wird rekursiv für alle $\mathbb{S}^i$ wiederholt, bis eine beliebig gewählte Mächtigkeit $|\mathbb{S}^i|$ unterschritten wird. Daraus bestimmt sich die maximale Anzahl von Elementen in den Blättern (sogenannte *bucket size*). Jeder Knoten kann also viele Kindknoten haben. Alle Elemente eines Teilbaums haben den gleichen Abstand zum Vater der Wurzel des Teilbaums, wie der Vater selbst.

Abbildung[3] 3.2 auf der nächsten Seite zeigt ein Beispiel eines BK-Baums. Die Distanzen der Kinder zum übergeordneten Knoten sind an den zugehörigen Kanten notiert. Für die Konstruktion wurde die ungewichtete Levenshteindistanz verwendet. Alle Knoten (insbesondere die Blätter) tra-

[3] Diese und die folgenden Abbildungen von Bäumen wurden mit Graphviz [GN00] erstellt.

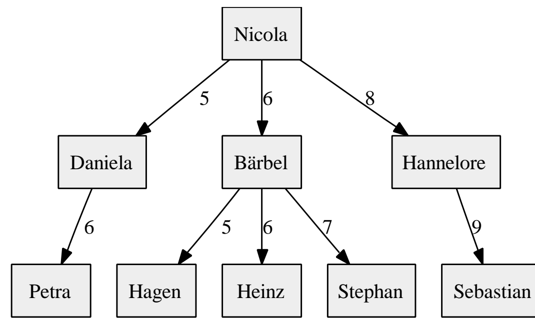


Abbildung 3.2.: Beispiel eines Burkhard-Keller-Baums

gen hier nur einen Wert. Dass alle Blätter auf der gleichen Höhe liegen, ist Zufall. Der BK-Baum garantiert *keine* Balancierung seiner Teilbäume. Die Balanceeigenschaft eines BK-Baums hängt im Wesentlichen von der Varianz der berechneten Distanzen ab. Die Wahl der Wurzel jedes Teilbaums spielt hierfür nur eine untergeordnete Rolle. Die Literatur enthält auch keine Vorschläge, die eine bessere Balancierung der Teilbäume garantieren.

Die Suche nach allen Elementen innerhalb eines (Teil-)Baums, die einen bestimmten maximalen Abstand von einem gegebenen Suchelement haben, nutzt die vorberechneten Distanzen und die Dreicksungleichung, um Teilbäume aus der Liste der möglichen Treffer auszuschließen. Seien $q \in \mathbb{X}$[4] das gesuchte Element, $k \in \mathbb{N}_0$ die maximal erlaubte Abweichung im Sinne der verwendeten Distanzfunktion und $x \in \mathbb{S}$ der Wurzelknoten. Nun können nach der Dreiecksungleichung nur noch Kindknoten mit einem Abstand i zur Wurzel Treffer enthalten, für den gilt: $d(x, q) - k \leq i \leq d(x, q) + k$ [BYN98, Kapitel 2.3]. Diese Regel wird rekursiv angewendet, bis ein Blatt erreicht ist, beziehungsweise keine Kindknoten zur weiteren Suche in Frage kommen. Innerhalb eines jeden Knotens werden alle enthaltenen Elemente x_j zur Menge der Suchtreffer hinzugefügt für die gilt: $d(q, x_j) \leq k$.

Ein Vorteil des BK-Baums ist, dass zu jeder Zeit neue Elemente hinzugefügt werden können, da das jeweilige Wurzelelement eines Teilbaums beliebig gewählt werden kann. Das Löschen von Elementen ist jedoch nicht mit ver-

[4] Es gilt nicht zwingend $q \in \mathbb{S}$.

tretbarem Aufwand möglich, da dessen gesamter Teilbaum neu aufgebaut werden müsste.[5] Eine Löschoperation ist allerdings für die vorliegenden Zwecke auch nicht notwendig. Des Weiteren ist die Einschränkung des BK-Baums auf diskrete Distanzfunktionen hier unproblematisch, da die ungewichtete Levenshteindistanz – wie auch im Beispiel ersichtlich – von vornherein diskrete Werte liefert.

Die erwartete Laufzeit der Konstruktion eines BK-Baums mit n Elementen beträgt $O(n \log(n))$, da jedes Element im Durchschnitt die Hälfte der Höhe heruntergereicht wird, bevor es an seinem endgültigen Platz angelangt ist. Die Laufzeit der Suche wird in [CNBYM01, Kapitel 5.1.1] mit $O(n^{\alpha})$ angegeben, wobei $0 < \alpha < 1$ gilt. Keine Beachtung findet in dieser Angabe der Suchparameter k, obwohl er zweifelsohne Einfluß auf die tatsächliche Laufzeit einer konkreten Suche hat. Dessen konkrete Auswirkung ist aber nicht ohne genauere Betrachtung der Daten im Baum zu beurteilen.

Es sei noch daran erinnert, dass die genannten Komplexitäten die Anzahl der Distanzberechnungen angeben. Dementsprechend ist bei längeren Zeichenketten mit möglicherweise signifikant erhöhten realen Laufzeiten zu rechnen. Gleichzeitig ermöglichen längere Zeichenketten auch größere Distanzen, was sich wiederum positiv auf die Höhe des Baums auswirken und damit die Suche beschleunigen kann.

3.3.3. Vantage-Point-Baum

Der *Vantage-Point-Baum* (kurz: VP-Baum) ist ein binärer Suchbaum, dessen Konstruktionsvorschrift annähernde Balance sicherstellen soll. Er wurde unabhängig voneinander sowohl in [Uhl91] als auch in [Yia93] beschrieben. Die Bezeichnung für diese Datenstruktur wurde aus dem letztgenannten Dokument entnommen, da sie sich in der nachfolgenden Literatur durchgesetzt hat und deutlich deskriptiver als der von Uhlmann vorgeschlagene Name

[5] Man könnte zwar ein beliebiges Element aus dem Teilbaum des zu löschenden Elements an dessen Stelle setzen, aber dann müssten die Distanzen aller „Brüder“ dieses Elements und deren Kinder neu berechnet werden. Die Laufzeit der Löschoperation hinge also von der Höhe des Teilbaums des zu löschenden Elements ab. Im schlimmsten Fall (wenn die Wurzel gelöscht wird) würde der komplette Baum neu aufgebaut.

metric tree ist.

Die Konstruktionsvorschrift: man wähle zunächst ein geeignetes Element $x \in \mathbb{S}$. Dieses Element wird *Vantage Point* genannt. Nun werden die Distanzen aller übrigen Elemente zum Vantage Point berechnet und der Median m aller Distanzen bestimmt. Der Median wird im aktuellen Knoten vermerkt. Alle Elemente y mit $d(x, y) < m$ werden dem linken (oder auch „inneren") Teilbaum zugeordnet, alle Elemente mit $d(x, y) > m$ werden dem rechten (oder „äußeren") Teilbaum zugeordnet. Elemente mit einer Distanz $d(x, y) = m$ können links oder rechts zugeordnet werden, die Wahl muss aber natürlich bei der späteren Suche berücksichtigt werden. Uhlmann bezeichnete diesen Vorgang als *ball decomposition*, da der Raum in graphischer Darstellung in zwei konzentrische Kugeln um den Vantage Point herum aufgeteilt wird. „Innen" liegen die näheren Elemente zum Knoten, „außen" diejenigen, die weiter entfernt sind. Wie beim BK-Baum wird der Vorgang rekursiv angewendet, bis die gewünschte Maximalanzahl von Elementen in den Blättern erreicht ist.

Abbildung 3.3 auf der nächsten Seite zeigt ein Beispiel eines VP-Baums, bei dem Elemente mit einer Distanz zum Vaterelement gleich dem Median nach rechts sortiert wurden. Daraus resultiert in diesem Fall auch das Übergewicht auf der rechten Seite. Der Vantage Point wurde jeweils zufällig gewählt. Yianilos schlägt in [Yia93] vor, statistische Eigenschaften mehrerer zufälliger Kandidaten für den Vantage Point zu untersuchen und den Besten auszusuchen.

Die unscharfe Suche verläuft folgendermaßen: in jedem Knoten x wird zunächst die Distanz $d(x, q)$ zum Suchelement q bestimmt. Ist sie nicht größer als die maximal erlaubte Distanz k, wird der aktuelle Knotenwert zur Liste der Suchtreffer hinzugefügt. Gilt $d(x, q) - k < m$, wird anschließend der linke Teilbaum rekursiv durchsucht, gilt $d(x, q) + k \geq m$, wird ebenso der rechte Teilbaum durchsucht. Wie beim BK-Baum können mehrere (beziehungsweise hier: beide) Teilbäume durchsucht werden.

Ein Vorteil des VP-Baums gegenüber dem BK-Baum besteht in der Tatsache, dass er keine diskrete Distanzfunktion erfordert. Für den vorliegenden Anwendungsfall könnte sich diese Eigenschaft aber auch als Nachteil er-

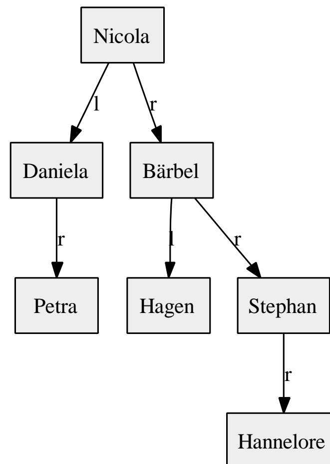


Abbildung 3.3.: Beispiel eines Vantage-Point-Baums

weisen, da eine diskrete Distanzfunktion häufiger Distanzen gleich dem Median liefert und deswegen zu einem Ungleichgewicht des Baums führen wird. Asymptotisch wird in [CNBYM01, Kapitel 5.1.1] für den Aufbau eine Laufzeitkomplexität von $O(n \log(n))$ und für die Suche $O(\log(n))$ angegeben.

Ein nachträgliches Einfügen oder Löschen von Elementen in einen VP-Baum ist nicht vorgesehen. Beides wäre im Prinzip mit vertretbarem Aufwand möglich, doch kann der Baum dadurch übermäßig stark außer Balance geraten, denn die bei der Konstruktion gewählten Mediane verlören ihre Eigenschaft, die jeweilige Teilmenge wirklich in der Mitte zu spalten.

3.3.4. Bisector-Baum

Ein weiterer Index für metrische Räume heißt*Bisector-Baum* (kurz: BS-Baum) und wurde erstmals in [KM83] beschrieben. BS-Bäume sind genau wie VP-Bäume Binärbäume, jeder Knoten hat also maximal zwei Kindknoten. Ein Knoten in einem BS-Baum speichert allerdings *zwei* Werte, c_1 und c_2, die „Zentren“ genannt werden. Ein Zentrum ist mit jeweils einem Kindknoten verknüpft. Zusätzlich wird für jedes Zentrum dessen „Radius“ (r_1 bzw. r_2) vermerkt.

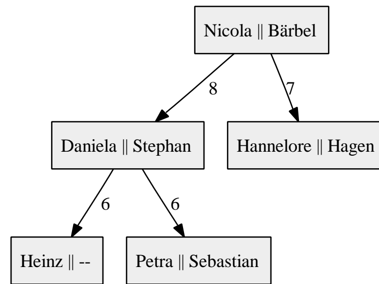


Abbildung 3.4.: Beispiel eines Bisector-Baums

Die Konstruktion verläuft wie folgt: zunächst werden zwei geeignete Elemente $c_1, c_2 \in \mathbb{S}$ ausgewählt. Nun wird die Distanz eines jeden anderen Elements $x \in \mathbb{S} \setminus \{c_1, c_2\}$ zu beiden Zentren berechnet. Jedes Element x wird anschließend rekursiv in denjenigen Teilbaum einsortiert, zu dessen Zentrum er eine geringere Distanz hat. Gleichzeitig wird in jedem Knoten für die beiden Zentren die bisher größte Distanz eines Wertes zum jeweiligen Zentrum gespeichert. Diese maximale Distanz wird als Radius bezeichnet.

Abbildung 3.4 enthält eine Darstellung eines BS-Baums. An den Kanten sind die Radii des übergeordneten Knotens vermerkt, die Zentren sind durch „||“ voneinander getrennt. Leere Zentren sind durch „- -“ gekennzeichnet. Auffällig gegenüber den bisher vorgestellten Bäumen ist vor Allem die geringere Knotenzahl, die natürlich daraus resultiert, dass in jedem Knoten zwei Elemente enthalten sein können. Trotz der geringen Knotenzahl ist zu erahnen, dass der BS-Baum nicht ausbalanciert ist. Auch das verwundert nicht, trifft die Konstruktionsvorschrift doch keine Maßnahmen, die Balance des Baumes sicherzustellen.

Die Suche in einem BS-Baum nutzt die gespeicherten Radii in jedem Knoten, um mit Hilfe der Dreiecksungleichung Teilbäume von der Suche auszuschließen. Seien $q \in \mathbb{X}$ ein zu suchendes Element, k die maximal erlaubte Distanz und x der Wurzelknoten des Baums mit seinen beiden Zentren c_i und den dazugehörigen Radii r_i. Nun wird jeder Teilbaum rekursiv durchsucht, für dessen Zentrum c_i gilt: $\text{ed}(q, c_i) \leq r_i + k$. In jedem erreichten

Knoten werden diejenigen Zentren zur Liste der Treffer hinzugefügt, für die gilt $\mathrm{ed}(q, c_i) \leq k$.

Die Laufzeitkomplexität für die Konstruktion eines BS-Baums wird in [CNBYM01, Kapitel 5.1.1] mit $\mathrm{O}(n \log(n))$ angegeben. Leider fehlt eine Angabe zur Komplexität der Suche. Es ist jedoch zu vermuten, dass sie asymptotisch nicht schlechter ist, als die der Suche in einem BK-Baum. Eine Optimierungsmöglichkeit wäre, Zentren mit einem möglichst großen Abstand voneinander zu wählen. Dies würde bewirken, dass bei einer Suche möglichst selten beide Teilbäume eines Knotens durchsucht werden müssen. Bei der Konstruktion des abgebildeten Baums wurden die Zentren zufällig gewählt.

Für den BS-Baum ist kein Algorithmus angegeben, einzelne Elemente nach der Konstruktion einzufügen. Dies ließe sich jedoch relativ einfach bewerkstelligen, sofern auf die eben erwähnte Optimierung keinen Wert gelegt wird. Da diese Operation aber nicht für den untersuchten Anwendungsfall gefordert wird, wird diese Möglichkeit im Weiteren ausgeklammert.

3.3.5. Weitere Ansätze und Überblick

Neben den vorgestellen Strukturen zur Beschleunigung der Suche in metrischen Räumen existiert eine Reihe weiterer Lösungen. Die Arbeit von Hjaltason und Samet [HS03] bietet darüber einen umfassenden Überblick.

In [BO97] wird etwa eine Verallgemeinerung des VP-Baums vorgeschlagen, der *Multi Vantage Point Tree*. Die zugrundeliegende Idee ist, den Raum in jedem Knoten in mehr als zwei gleich große Teile aufzuspalten. Eine Variante der BK-Bäume sind *Fixed-Query-Trees* [BCMW94] beziehungsweise deren alternative Speicherform, *FQ-Arrays*. Sie basieren auf der Idee, auf jeder Ebene des Baumes das gleiche Vergleichselement zu benutzen, um die Anzahl der teuren Distanzberechnungen zu minimieren. BS-Bäume sind wiederum eine Weiterentwicklung der von Uhlmann in [Uhl91] entwickelten *Generalized-Hyperplane-Trees*.

Wie bei den Distanzfunktionen auch, existieren Lösungen, die relevante Suchergebnisse nur mit einer gewissen Wahrscheinlichkeit liefern, dafür aber

	BK-Baum	**VP-Baum**	**BS-Baum**
Balanciert	nein	ja	nein
Laufzeit Suche	$O(n^\alpha)$	$O(n \log(n))$	?
Laufzeit Konstruktion	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$
Speicherbedarf	$O(n)$	$O(n)$	$O(n)$

Tabelle 3.1.: Eigenschaften der vorgestellten Bäume

äußerst effizient arbeiten. Ein Vertreter davon ist in [FCNP06] beschrieben.

Eine Sonderstellung nimmt der *M-Tree* [CPZ97] ein, der – ähnlich wie ein B-Baum – auf Speicherung auf langsamen Speichermedien wie Festplatten optimiert ist und auch das Hinzufügen weiterer Elemente unter Beibehaltung seiner positiven Laufzeiteigenschaften unterstützt. Die zugrundeliegenden Algorithmen zum Spalten mehrwertiger Knoten sind allerdings auch entsprechend komplex.

Auch allgemeine Vorschläge zur Verbesserung der Suchlaufzeiten unabhängig von der verwendeten Datenstruktur werden diskutiert. So enthält [Fre05] einen Vorschlag, wie durch nur eine zusätzliche Distanzberechnung zur Suche – nach Angabe der Autoren – 10–50% der Suchlaufzeit eingespart werden können.

Schließlich enthält Tabelle 3.1 eine Übersicht über die Eigenschaften der detailliert vorgestellten Strukturen.

4. Implementierung

4.1. Überblick

Soweit keine frei[1] verfügbaren Implementierungen der im vorigen Abschnitt vorgestellten Algorithmen verfügbar waren, wurden sie in der Programmiersprache Java Version 5 implementiert.

Alle Klassen der erstellten Bibliothek liegen entsprechend den Javakonventionen unterhalb von `com.guj.searching`.[2] Den Kern der Bibliothek bildet das Paket `metric` mit zwei Unterpaketen `distance` und `tree`. Diese enthalten die Implementierungen der metrischen Distanzfunktionen und der Bäume. Das Paket `analysis` enthält einige Klassen, die Benchmarks, sowie die statistische Analyse der Adressdaten und die Analyse von Bäumen unterstützen. Auf diesen Teil der Bibliothek wird im Weiteren nicht weiter eingegangen, weil er nicht direkt zur Lösung des Problems beiträgt. Lediglich die Ergebnisse der mit seiner Hilfe durchgeführten Messungen sind Gegenstand des Abschnitts 5. Zudem wurden er benutzt, um die Abbildungen von Baumstrukturen im vorigen Abschnitt zu erstellen. Das Paket `address` enthält alle Adress-spezifischen Programmteile, die das Laden und Vergleichen der CSV-Dateien unterstützen.

Der vollständige Sourcecode der implementierten Lösung, abzüglich des Pakets `analysis`, ist in Anhang A enthalten.

[1] „Freiheit“ ist hier nicht unbedingt wie im Begriff „Freie Software“ zu verstehen. Es wurde bei der Auswahl lediglich darauf geachtet, dass die Nutzung zu privaten, kommerziellen oder Forschungszwecken nicht grundsätzlich ausgeschlossen und die Quellen einsehbar sind, so dass die entscheidenden Programmteile leicht in die eigene Lösung integrierbar sind.

[2] Hier und im Folgenden wird `nichtproportionale Schrift` verwendet, wenn Paket- oder Klassenbezeichner oder Quellcode im Allgemeinen wiedergegeben werden.

4.2. Schnittstellen

4.2.1. Distanzfunktionen

Implementierungen des dynamischen Algorithmus zur Berechnung der Levenshteindistanz sind in großer Zahl und in verschiedenen Programmiersprachen im Internet zu finden. Verwendet wird hier die Version aus dem Apache-Jakarta-Projekt,[3] da sie das beste Laufzeitverhalten einer Implementierung in Java zeigt. Die Damerau-Levenshteindistanz findet sich seltener. Die vom LingPipe-Projekt[4] entwickelte Javabibliothek zur Analyse von Texten in natürlicher Sprache enthält aber auch diese Distanzfunktion. Lediglich die Hammingdistanz wurde vom Autor selbst implementiert.

Da Methoden in Java keine Objekte sind, die herumgereicht werden können, wurde für jede Distanzfunktion eine Klasse erstellt. Alle diese Klassen erben von der abstrakten Klasse `MetricDistance<T>`, deren interessantestes Merkmal die Methode `call(T, T)` ist, die die eigentliche Implementierung enthält. Der Typparameter `T` der Klasse legt also den Typ derjenigen Objekte fest, für die Instanzen der Klasse eine metrische Distanz berechnen können. Dementsprechend sind die Klassen `SimpleLevenshtein`, `DamerauLevenshtein` und `Hamming` auf Objekte vom Typ `String` festgelegt. Listing 4.1 enthält einen Ausschnitt der Klasse `MetricDistance<T>` und die Signatur der Implementierung der Hammingdistanz, um diesen Zusammenhang zu demonstrieren.

4.2.2. Metrische Räume

Die vorgestellten Algorithmen für Burkhard-Keller-, Vantage-Point- und Bisector-Bäume wurden vollständig vom Autor selbst implementiert. Die Klassen `BKTree`, `VPTree` und `BSTree` erben alle von der abstrakten Oberklasse `MetricTree<T>`. Der Typparameter `T` zeigt hier den Typ der in den Baum einzufügenden Objekte an. Jede Instanz von `MetricTree<T>` hat außerdem ein Attribut `distance` vom Typ `MetricDistance<T>`. Dadurch,

[3] http://jakarta.apache.org/commons/lang/
[4] http://www.alias-i.com/lingpipe/

```java
public abstract class MetricDistance<T> {
  public abstract int call(T first, T second);
}

public class Hamming extends MetricDistance<String> {
  @Override
  public int call(String first, String second) { [...] }
}
```

Listing 4.1: Die Klasse `MetricDistance<T>` und eine Unterklasse

```java
public abstract class MetricTree<T> {
  protected final MetricDistance<T> distance;
  protected Node rootNode;
  public Set<T> search(T query, int k) {
    return this.rootNode.search(query, k);
  }

  public abstract class Node {
    protected abstract Set<Match> search(T query, int k);
  }
}
```

Listing 4.2: Ausschnitt aus dem Interface der Klasse `MetricTree<T>`

dass der gleiche Typparameter für den Bauminhalt und die Distanzfunktion verwendet wird, wird sichergestellt, dass die im Baum enthaltenen Objekte von der zu benutzenden Distanzfunktion verarbeitet werden können.

Da sich alle implementierten Bäume leicht als rekursive Datenstrukturen beschreiben lassen, benutzen sie eine klassenlokale innere Klasse, die einen einzelnen Knoten vom jeweiligen Typ beschreibt. Die rekursiven Algorithmen für die Baumkonstruktion und die Suche wurden möglichst eng an ihrer Beschreibung in den zitierten Quellen implementiert. Die von außen sichtbaren Baumklassen stellen fast ausschließlich ein Interface für die Funktionalitäten ihres Wurzelknotens dar. Listing 4.2 zeigt einen Ausschnitt aus der Signatur der abstrakten Klassen `MetricTree<T>` und `Node`. In der Methode `MetricTree<T>.search` ist zu sehen, wie der Aufruf an die entsprechende Methode des Wurzelknotens delegiert wird.

Zudem benutzen die Knoten das finale Attribut `distance` des Baums, zu dem sie gehören, für alle durchzuführenden Distanzberechnungen. Auf diese Weise wird sichergestellt, dass alle Knoten eines Baumes für alle Operationen die gleiche Distanzfunktion benutzen. Die zu benutzende Distanzfunktion muss im Konstruktor der Bäume übergeben werden und kann anschließend nicht mehr ausgetauscht werden.

Da außer dem BK-Baum keine der Datenstrukturen eine Manipulation ihres Inhalts nach der Konstruktion erlaubt, sind entsprechende Methoden nicht Teil der abstrakten Oberklasse `MetricTree<T>`. Gewöhnlich werden alle in den Baum einzufügenden Objekte dem Konstruktur der Baumklassen übergeben. Gelegentlich (beispielweise in Abschnitt 4.5.1) kann es jedoch auch nützlich sein, leere Bäume zu erzeugen, die erst zu einem späteren Zeitpunkt befüllt werden sollen. Für diesen Zweck existiert eine Methode mit dem Namen `construct`, mit deren Hilfe ein Baum auch nach seiner Instanzierung mit neuen Objekten aufgebaut werden kann. Wird diese Methode aufgerufen, obwohl der Baum bereits Objekte enthält, werden die alten Bauminhalte verworfen.

4.3. Implementierungsdetails

4.3.1. Distanzfunktionen

Die Implementierungen der Levenshtein- und Damerau-Levenshteindistanz orientieren sich zwar an dem vorgestellten Modell aus der dynamischen Programmierung, sind aber nicht rekursiv formuliert, sondern iterativ. Beide Algorithmen sind dahingehend optimiert, dass die zu füllende Matrix (vgl. Abbildung 3.1 auf Seite 17) nicht vollständig im Speicher gehalten wird. Für die Levenshteindistanz werden zu jedem Zeitpunkt nur zwei Spalten gespeichert, für die Damerau-Levenshteindistanz derer drei. Die Matrix wird in beiden Fällen so konstruiert, dass die jeweils kürzere Zeichenkette die Spalten bildet (vgl. erneut Abbildung 3.1). Dadurch wird eine weitere Speicherersparnis erzielt, wenngleich diese sich in der asymptotischen Betrachtung nicht auszahlt.

Der Hamming-Algorithmus ist äußerst einfach. Er vergleicht die einzelnen Zeichen der übergebenen Zeichenketten paarweise und zählt, wie oft die Zeichen nicht übereinstimmen. Zusätzlich wurde der Algorithmus um die Fähigkeit erweitert, auch Distanzen von Zeichenketten ungleicher Länge berechnen zu können. Hierzu wird die Differenz der Längen beider Zeichenketten dem ursprünglichen Ergebnis hinzuaddiert. Dieses Vorgehen ist äquivalent zum Auffüllen der kürzeren Zeichenkette mit Zeichen, die nie mit dem korrespondierenden Zeichen in der anderen Zeichenkette übereinstimmen.

4.3.2. Metrische Räume

Alle implementierten Baumstrukturen wurden dahingehend modifiziert, dass in jedem Knoten mehrere Elemente in einer Liste gespeichert werden können, sofern ihre Distanz zueinander gleich Null ist. Diese Maßnahme vermeidet Distanzberechnungen zu Elementen, deren Distanz bereits ermittelt wurde. Schließlich sind Elemente mit einer Distanz von Null nach der Regel der strengen Positivität identisch (siehe Seite 15). Zudem folgt aus der Dreickesungleichung und der Positivität der metrischen Funktion d, dass gilt $d(x, y) = 0 \wedge d(x, z) = 0 \Rightarrow d(y, z) = 0$.[5] Sind also bereits mehrere Elemente (x und y) mit einer Distanz von Null zueinander in einem Knoten gespeichert und kommt ein Weiteres (z) hinzu, reicht die Distanzermittlung zu *einem* der vorhandenen Elemente (beispielweise x) um festzustellen, ob das neue Element eine Distanz von Null zu *allen* Elementen in dem Knoten hat. Ruft ein Baumknoten die Distanzfunktion seines Baumes auf, übergibt er dieser jeweils einen (beliebig gewählten) Repräsentanten der in ihm gespeicherten Werte. Im BS-Baum wirkt sich diese Modifikation so aus, dass pro Knoten zwei Listen gespeichert werden – pro Zentrum eine Liste mit jeweils identischen Elementen.

[5]Beweis: seien x und y zwei Elemente mit $d(x, y) = 0$ in einem Knoten und z ein neu hinzukommendes. Gilt $d(x, z) = 0$, so folgt $d(y, z) \leq \mathrm{d}(x, y) + d(x, z) = 0 + 0$. Da die Distanz nicht kleiner als Null werden kann gilt $d(y, z) = 0$. □

4.4. Anwendung

Das eingangs erwähnte Paket `address` enthält eine Klasse `Address`, die einzelne Adressdatensätze beschreibt. Ihre wichtigste Eigenschaften sind diverse Instanzattribute für die einzelnen Adressbestandteile. Die Klasse `AddressFile` ist mit Hilfe der freien Bibliothek OstermillerUtils in der Lage, CSV-Dateien mit Adressen zu laden und aus jedem Datensatz eine neue `Address`-Instanz zu erzeugen. Da sie das Interface `Iterable<Address>` implementiert, kann über ihre Instanzen beliebig oft iteriert werden. Jedes Mal, wenn ein neuer Iterator über ein `AddressFile` verlangt wird, wird die gewünschte Datei neu geöffnet und nur jeweils soweit gelesen, wie es der Aufrufer benötigt.

Eine Instanz von `AddressFile` kann direkt dem Konstruktor einer der Unterklassen von `MetricTree<T>` übergeben werden. Dadurch wird der Typ des Bauminhalts auf `Address` festgelegt, was zur Folge hat, dass die übergebene Distanzfunktion vom Typ `MetricDistance<Address>` sein muss. Da alle Attribute von `Address`-Instanzen vom Typ `String` sind,[6] kann nun leicht eine Klasse erstellt werden, deren Instanzen eine der bereits definierten metrischen Funktionen für Zeichenketten benutzen, aber von außen betrachtet `Address`-Objekte vergleicht. Im Paket `address` sind zur Veranschaulichung zwei Möglichkeiten implementiert. Listing 4.3 zeigt ihre gemeinsame Basisklasse `AddressDistance`, die das Prinzip demonstriert.

Der Konstruktor der Klasse nimmt eine metrische Distanzfunktion für Zeichenketten entgegen, die in der `call`-Methode aufgerufen wird. Welche Teile einer Adresse dieser Distanzfunktion übergeben werden, entscheidet sich in der Methode `extractComponent`. In der Darstellung wird die komplette String-Repräsentation einer Adresse benutzt, Unterklassen können diese Methode jedoch überschreiben und beispielsweise entscheiden, an dieser Stelle nur den Namen, die postalische Adresse oder andere (eventuell vorberechnete) Adressbestandteile zum Vergleich auszuwählen. Auf diese Weise kann ein Baum ganze `Address`-Objekte speichern und als Suchergebnisse zurückgeben, während er nur einzelne Bestandteile für Vergleiche

[6] Ausnahme: eine Identifikationsnummer, die nicht für Vergleiche herangezogen wird.

```
public class AddressDistance extends
    MetricDistance<Address> {

  public final MetricDistance<String> strategy;

  public AddressDistance(
      MetricDistance<String> strategy) {
    this.strategy = strategy;
  }

  @Override
  public int call(Address s, Address t) {
    return this.strategy.call(
      this.extractComponent(s),
      this.extractComponent(t));
    }

  public String extractComponent(Address a) {
    return a.toString();
  }
}
```

Listing 4.3: Eine metrische Distanzfunktion für `Address`-Objekte

berücksichtigt.

Zwar wäre es auch möglich, einen Baum nur aus einzelnen Addressbestandteilen als einfachen Zeichenketten zu konstruieren. Die Zusammenführung verschiedener Suchergebnisse, etwa auf Basis eines separaten Vergleichs von postalischen Adressen und Namen, würde dadurch aber erschwert.

4.5. Abgleiche

Die bisher vorgestellten Implementierungen ermöglichen die schnelle Beantwortung einzelner Suchanfragen auf der Basis einzelner Adressbestandteile. Um aber vollständige Adressen in großer Zahl vergleichen zu können, werden zwei Dinge benötigt: eine Definition von „Gleichheit" zweier Adressen sowie ein System, das diese Definition abbilden kann.

Ein genauer Vorschlag für eine angemessene Definition wird erst in Ab-

schnitt 5.2 erarbeitet. Mit den beschriebenen und implementierten Werkzeugen läßt sich dennoch eine Schablone für eine passende Definition angeben, die anschließend nur noch parametrisiert werden muß. Die Implementierung dieser Schablone und deren Nutzung für verschiedene Arten von Abgleichen ist Gegenstand der folgenden Abschnitte.

4.5.1. Parameter des Vergleichs

Ein Vergleich mehrerer Adressen mit Hilfe der in Abschnitt 3 dargestellten Mittel kann aus einer Menge beliebiger Kombinationen folgender Faktoren bestehen:

- Adressbestandteil(e)
- Distanzfunktion
- maximal erlaubte Entfernung
- zu verwendende Indexstruktur

In der Klasse `AddressDistance` und ihren Unterklassen werden bereits die ersten beiden Punkte zusammengefasst. Eine Vereinigung aller vier Punkte stellt die Klasse `ComparisonPlan` dar, die auszugsweise in Listing 4.4, abgebildet ist. Mit ihrer Hilfe lassen sich Bedingungen formulieren wie „Zwei Adressen sind dann gleich, wenn ihre Vornamen eine Levenshteindistanz von zwei nicht überschreitet“. Die Angabe einer Indexstruktur ist dabei nur eine Hilfestellung zur Ermittlung der Treffer.

Ein Verwender einer oder mehrerer Instanzen dieser Klasse kann nun alle möglichen Paare von Adressen darauf überprüfen, ob die durch diesen „Plan“ formulierten Bedingungen zutreffen, oder nicht. Zur Beschleunigung dieser Vergleiche kann eine (anfangs leere) Indexstruktur verwendet werden. Dies ist allerdings nicht zwingend notwendig.

Jeder der nachfolgend dargestellten Algorithmen benutzt eine oder mehrere Instanzen dieser Klasse, um Adresslisten gegeneinander abzugleichen.

```
1 public class ComparisonPlan {
      public final AddressDistance distance;
      public final int maxDistance;
      public final MetricTree<Address> tree;
5 }
```

Listing 4.4: Ausschnitt aus der Klasse `ComparisonPlan`

4.5.2. In-Sich-Abgleich

Das Ziel eines In-Sich-Abgleichs ist, Dubletten innerhalb einer einzelnen Datei zu finden und zu eliminieren. Das bedeutet, eine in-sich abgeglichene Datei enthält keine Adressen mehr, die entsprechend einer bestimmten Definition (angegeben durch mehrere Instanzen von `ComparisonPlan`) gleich sind.

Um dies zu erreichen, sind mehrere Strategien denkbar. Eine mögliche Vorgehensweise läßt sich folgendermaßen beschreiben:

1. Konstruiere für jeden Plan einen Baum mit allen Adressen.
2. Für jede Adresse:
   a) Ist die Adresse bereits als Dublette erkannt, nimm die nächste und gehe zurück zu 2.
   b) Ermittle Übereinstimmungen durch Suche in dem Baum des ersten Plans mit dem dazugehörigen Maximalabstand.
   c) Entferne die aktuelle Adresse aus den Suchergebnissen. Alle übrigen Adressen sind *Kandidaten*, Dubletten dieser Adresse zu sein.
   d) Für jeden übrigen Plan:
      i. Suche in dessen Baum nach Übereinstimmungen für die aktuelle Adresse mit dem durch den Plan vorgegebenen maximalen Abstand.
      ii. Die Schnittmenge aus der Menge der Kandidaten und dem letzten Suchergebnis bildet die neue Kandidatenmenge.
      iii. Sind keine Kandidaten mehr übrig, gehe zurück zu 2.
   e) Füge alle Kandidaten zur Menge der Dubletten hinzu.

3. Gib die ursprüngliche Liste von Adressen *ohne* diejenigen Adressen zurück, die in der Liste von Dubletten vorkommen.

Schritt 2a ist aufgrund der Symmetrieeigenschaft (siehe Seite 15) der Distanzfunktionen notwendig. Andernfalls würde zu jeder Dublette D einer Adresse A wiederum A als Dublette von D gefunden werden. Zudem könnten für D eine oder mehrere Dubletten Z existieren, die allerdings nicht als Dubletten von A betrachtet werden. Die korrekte Art der Behandlung dieser Dubletten-von-Dubletten ist nicht allgemein vorherzusehen und äußerst fehlerträchtig. Man kann leicht einen pathologischen Fall konstruieren, in dem alle Adressen eine Kette von Dubletten bilden, so dass am Ende nur eine einzige übrig bleibt. Daher wird hier und im Folgenden auf deren Erkennung und gesonderte Behandlung verzichtet. An dieser Stelle wird auch deutlich, dass keine Anstrengungen unternommen werden, den „besten“ der als äquivalent betrachteten Datensätze zu erkennen und zu bevorzugen. Die Reihenfolge der Adressen entscheidet darüber, welche entfernt wird und welche übrig bleibt.

Diese Herangehensweise birgt jedoch auch mehrere lösbare Probleme. Zum Einen müssen mehrere Bäume konstruiert werden. Dies kostet Zeit und vor Allem belegen die Bäume viel Speicher für die gesamte Dauer des Abgleichs. Zum Anderen wird in der innersten Schleife jeweils der *komplette* Baum nach Übereinstimmungen durchsucht. Dabei ist schon beim zweiten Durchlauf klar, dass nur die Ergebnisse des ersten Suchlaufs überhaupt in Frage kommen, alle Bedingungen zu erfüllen. Ein weiterer Punkt ist, dass in dieser Variante der erlaubte Maximalabstand *jedes* Plans eine gewichtige Rolle für die Laufzeit spielt. (Dieser Zusammenhang wird in Abschnitt 5.1.3 genauer untersucht.) Die Berücksichtigung aller genannten Überlegungen ergibt eine schnellere Variante:

1. Konstruiere für *einen* der Pläne einen Baum mit allen Adressen.
2. Für jede Adresse:

    a) Ist sie als Dublette bekannt, gehe zurück zu 2 und fahre mit der nächsten Adresse fort.

   b) Suche in dem Baum nach Übereinstimmungen für die aktuelle Adresse mit dem durch dessen Plan vorgegebenen maximalen Abstand.
   c) Entferne die aktuelle Adresse selbst aus den Suchergebnissen. Die restlichen Adressen sind *Kandidaten* für Dubletten.
   d) Für jeden Kandidaten:
      i. Ermittle den Abstand des Kandidaten zur aktuellen Adresse mit Hilfe der Distanzfunktionen aller übrigen Pläne. Brich ab und gehe zum nächsten Kandidaten über, sobald der erlaubte Maximalabstand bei einem Plan überschritten wird.
      ii. Besteht der Kandidat alle Tests, gilt er als *verifiziert* und wird zur Liste der Dubletten hinzugefügt.

3. Gib die ursprüngliche Liste von Adressen *ohne* diejenigen Adressen zurück, die in der Liste von Dubletten vorkommen.

Hier wird auch deutlich, wieso `ComparisonPlan` sowohl das Attribut `distance` als auch `tree` hat, obwohl doch ein Baum allein schon eine metrische Distanzfunktion enthält. Der Baum wird nicht in jedem Fall benutzt und kann daher auch mit `null` initialisiert werden.

Beide Abgleichalgorithmen sind in der Klasse `ComparisonPlan` als statische Methoden mit den Namen `getUniquesWithTrees` und `getUniques` implementiert, die jeweils eine Adressdatei und mehrere Instanzen von `ComparisonPlan` übergeben bekommen.

4.5.3. Abgleich zweier Adressmengen

Zwei oder mehrere Adressmengen können mit unterschiedlichen Zielsetzungen abgeglichen werden. Werden die Adressen, wie in Abschnitt 2.1.1 beschrieben, für Marketingaktionen verwendet, treten zwei zu unterscheidende Fälle auf: im ersten Fall gilt es, zwei Adressmengen so zusammenzuführen, dass keine Adressen doppelt vorkommen. Im zweiten Fall, dem sogenannten Negativabgleich, geht es darum, alle Adressen, die in einer der beiden Dateien vorkommen, aus der anderen zu entfernen.

Zusammenführung

Die vorgestellte Lösung zum In-Sich-Abgleich kann direkt für die Zusammenführung mehrerer Adressmengen verwendet werden, wenn die Adressmengen vorher (ohne Beachtung von Übereinstimmungen) in einer Liste zusammengefasst werden.

Eine andere Möglichkeit besteht darin, zuerst beide Mengen einzeln in-sich abzugleichen und aus einer der beiden einen Baum zu konstruieren. Alle Adressen der anderen Menge werden anschließend in diesem Baum gesucht und die Suchergebnisse mit Hilfe der angegebenen Pläne mit der gesuchten Adresse verglichen. Wurden keine übereinstimmenden Adressen gefunden, kann die erfolglos gesuchte Adresse zur ersten Menge (mit der der Baum konstruiert wurde) hinzugefügt werden. Nach Prüfung aller Adressen enthält diese Menge das Ergebnis. Diese Variante der Zusammenführung zweier Adressen ist ebenfalls in der Klasse `ComparisonPlan` als statische Methode mit dem Namen `mergeFiles` implementiert.

Beide Varianten liefern im Test die gleichen Ergebnisse. Dabei war die Methode `mergeFiles` geringfügig schneller als die Benutzung von `getUniques`.

Negativabgleich

Ein Negativabgleich kann fast genau so vollzogen werden, wie die Zusammenführung zweier Dateien. Aus der Positivliste wird ein Baum konstruiert und die Adressen der Negativliste werden darin gesucht. Die Suchergebnisse werden wieder mit den übrigen Vergleichsplänen verglichen. Ergeben sich allerdings Treffer, werden diese aus der Positivliste (aus der der Baum konstruiert wurde) entfernt. Diese Arbeit wird von der statischen Methode `getAllWithout` der Klasse `ComparisonPlan` geleistet.

5. Auswertungen

5.1. Laufzeitverhalten

5.1.1. Testumgebung

Um direkte Vergleichbarkeit von Zeitmessungen zu gewährleisten, wurden alle Tests unter möglichst gleichen Bedingungen ausgeführt. Zur Verfügung stand ein i386-kompatibler PC mit einer 2,8 GHz Pentium 4 CPU sowie einem Gigabyte RAM. Das verwendete Betriebssystem ist Microsoft Windows XP (SP 2). Dies entspricht der Standardausstattung eines PCs bei Gruner + Jahr.

Es wurden der Compiler und die virtuelle Maschine (JVM) aus dem Sun J2SE Development Kit in der Version 1.5.0_05 verwendet. Die Vielzahl der verfügbaren Optionen der JVM in Bezug auf Speicherverwaltung wurden bei den Standardeinstellungen belassen. Lediglich der maximal verfügbare Arbeitsspeicher musste mit der Option `-Xmx` auf einige hundert Megabyte (Standard: 64MB) erhöht werden. Außerdem bietet das Development Kit von Sun die Auswahl zwischen zwei verschiedenen virtuellen Maschinen, der sogenannten „Client VM“ und der „Server VM“. In den nachfolgend beschriebenen Tests zeigten sich Performancevorteile von bis zu fünfzig Prozent zugunsten der Server VM, daher wurde diese durch Angabe des Parameters `-server` beim Start der JVM ausgewählt.

5.1.2. Distanzfunktionen

Abbildung 5.1 zeigt die Laufzeiten der implementierten Distanzfunktionen in Abhängigkeit von der Länge der zu vergleichenden Zeichenketten. Da die Laufzeit einzelner Berechnungen zu kurz für zuverlässige Messungen ist, wurde jeweils die Zeit für zehn Millionen direkt aufeinanderfolgende

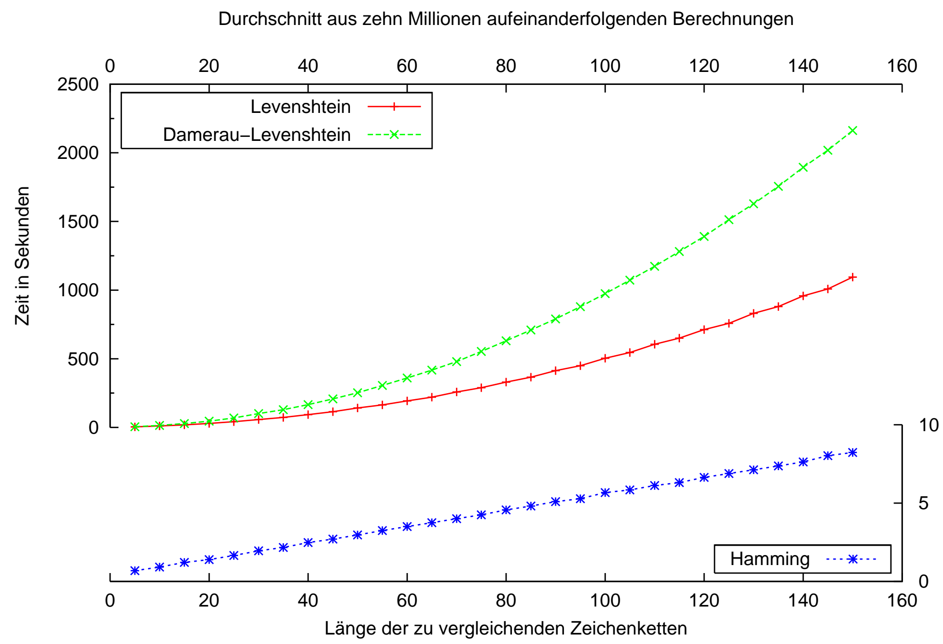


Abbildung 5.1.: Laufzeitverhalten der Distanzfunktionen

Berechnungen gemessen und der Durchschnitt dargestellt.[1] Da die Laufzeiten der Hammingdistanz deutlich geringer sind, als die der anderen beiden Funktionen, sind diese auf einer eigenen y-Achse auf der rechten Seite abzulesen.

Klar zu erkennen ist, dass die Hammingdistanz am schnellsten berechnet werden kann. Hinzu kommt, dass die erwartete lineare Abhängigkeit der Laufzeit von der Länge der Zeichenketten experimentell bestätigt wird. Ebenso erwartungsgemäß ist die Berechnung der Damerau-Levenshteindistanz langsamer als die der reinen Levenshteindistanz, wobei beide Funktionen polynomielles Laufzeitverhalten aufweisen. Das Verhältnis der Laufzeiten dieser Funktionen liegt bei etwa 2:1.

Obwohl die Damerau-Levenshteindistanz die langsamste Funktion unter den drei Untersuchten ist, erscheint sie am geeignetsten für die weitere Verwendung. Dadurch, dass sie in einigen Fällen durch die Berücksichtigung vertauschter Buchstaben eine geringere Distanz liefert, lässt sich eventuell später sogar die Gesamtlaufzeit einer Suche senken (siehe Abschnitt 5.1.3 auf Seite 47 und insbesondere die Abbildung 5.3).

5.1.3. Metrische Räume

Konstruktion

Da die zugrundeliegenden Daten im Prinzip statisch sind, ist die zur Konstruktion eines Baums benötigte Zeit nur von untergeordneter Bedeutung. Allerdings ist die Abhängigkeit der Laufzeit von der Anzahl der enthaltenen Elemente unter Umständen interessant. Schließlich wächst nach den Angaben in Tabelle 3.1 auf Seite 29 die Laufzeit überproportional zur Anzahl der Elemente.

Die Kurven in Abbildung 5.2 stellen diesen Zusammenhang dar. Sie geben allerdings die Laufzeit der Konstruktion *pro Knoten* an, da so das Wachstum

[1] Alle Zeitmessungen wurden durch Aufrufe von `System.currentTimeMillis()` vorgenommen. Diese Methode gibt die aktuelle Zeit in Millisekunden seit dem 01. Jan. 1970 00:00 Uhr (UTC) zurück. Mit welcher Präzision die virtuelle Maschine die Zeit angeben kann, hängt allerdings vom unterliegenden Betriebssystem ab. Für Windows werden Werte zwischen 10 und 20 Millisekunden angegeben.

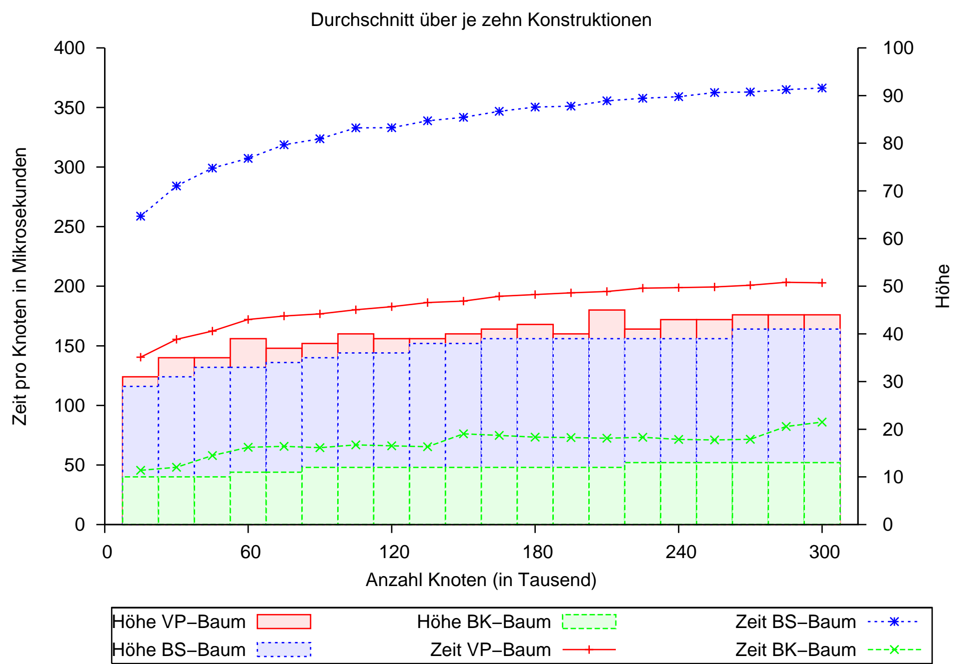


Abbildung 5.2.: Laufzeit der Baumkonstruktion pro Knoten

der Laufzeit deutlicher in Erscheinung tritt. Die verwendete Distanzfunktion vergleicht die Anschriften der einzufügenden Adressen mit Hilfe der Damerau-Levenshtein-Distanz. Zusätzlich zeigt die Abbildung mit den eingefärbten Flächen die Höhe der konstruierten Bäume. Für jede Messung wurde die jeweils vorige Menge Adressen um 15.000 weitere Adressen erweitert.

Die Abbildung zeigt, dass das Wachstum der Laufzeit pro Knoten bei steigender Knotenzahl in allen Fällen logarhitmisch mit einem äußerst moderaten konstanten Faktor wächst. Die in der asymptotischen Betrachtung vorhergesehene Abhängigkeit dieser Laufzeit von der Höhe des betreffenden Baumes ist klar zu erkennen. Allerdings ergeben sich deutliche Unterschiede in den konstanten Faktoren, die die reale Laufzeit entscheidend beeinflussen. Während das Verhältnis von Laufzeit und Höhe bei dem BK- und dem VP-Baum annähernd gleich ist, ist es für den BS-Baum deutlich größer. Damit ist der BS-Baum der langsamste der drei Bäume. Mit seiner Konstruktionszeit von etwa zwei Minuten für 300.000 Elemente wäre er aber immer noch verwendbar. Mit dreißig Sekunden Konstruktionszeit für ebensoviele Elemente ist der BK-Baum am schnellsten. Der VP-Baum liegt mit etwa einer Minute in der Mitte. Diese Zeitangaben stellen allerdings nur ungefähre Richtwerte dar. Man bedenke, dass der durchgeführte Testlauf pro Baumkonstruktion eine sehr große Anzahl von Objekten instanziert und danach sofort wieder verwirft. Einen nicht unerheblichen Teil der Laufzeit verbringt die virtuelle Maschine daher in der Garbage Collection. Durch Einsatz eines Profilers wurde ein Anteil von ungefähr 15% ermittelt.

Bei der Betrachtung der Baumhöhen fallen mehrere Dinge auf: dass der BK-Baum die geringste Höhe hat, sollte nicht überraschen. Schließlich ist er der einzige der untersuchten Bäume, der pro Knoten mehr als zwei Kinder erlaubt. Der geringe Höhenunterschied zwischen BS- und VP-Baum läßt sich dadurch erklären, dass der BS-Baum pro Knoten zwei Elemente speichern kann, also nur etwa die halbe Anzahl Knoten wie der VP-Baum enthält. Da beides Binärbäume sind, sollte sich die halbe Knotenzahl in einem BS-Baum durch einen Höhenunterschied von ungefähr eins bemerkbar machen. Bemerkenswert ist allerdings die Tatsache, dass die Höhe des VP-Baums mit steigender Knotenzahl nicht monoton steigt. Die Ursache hierfür wird sein,

dass die jeweils neu hinzukommenden Elemente Einfluss auf die Ermittlung des Median nehmen. Und dieser bestimmt wiederum die Balance und damit die Höhe des VP-Baums.

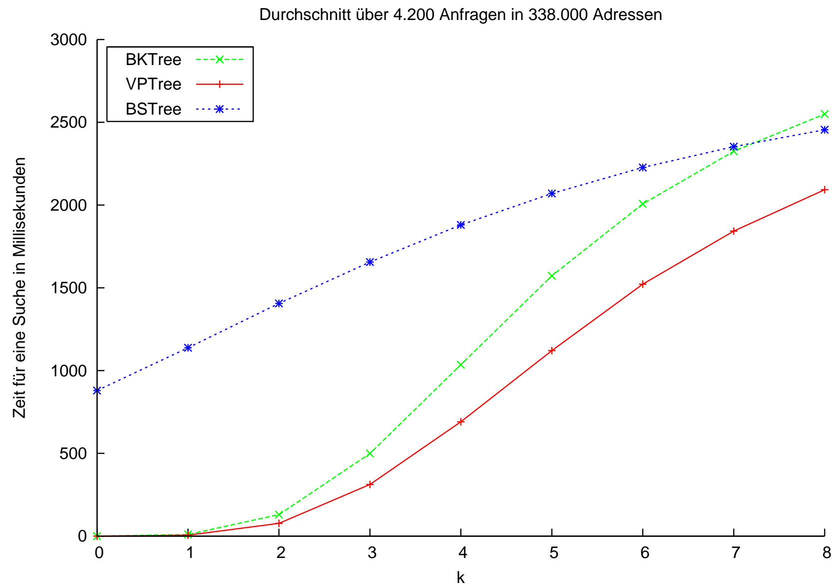
Suche

Die wichtigste Eigenschaft der Baumstrukturen ist ihre Fähigkeit, bei einer Suche möglichst schnell einen großen Teil des Suchraums von der Suche auszuschließen (in [CTTFF02] „pruning ability“ oder kurz „prunability“ genannt). Nur so läßt sich die Anzahl der zu berechnenden Distanzen und damit die Laufzeit der Suche minimieren. Die Abbildungen 5.3(a) und 5.3(b) illustrieren die starke Abhängigkeit der Suchlaufzeit von der Anzahl notwendiger Distanzberechnungen. Dargestellt sind die Laufzeit (beziehungsweise die Anzahl Distanzberechnungen) einer Suche in Abhängigkeit von der Fehlertoleranz, also der maximal erlaubten Distanz k zum Suchobjekt. Da in keiner der drei Strukturen bei der Suche die Distanz eines Elements zum Suchobjekt mehr als einmal berechnet wird, ist die Anzahl der Distanzberechnungen äquivalent zur Anzahl der mit dem Suchobjekt verglichenen Elemente.

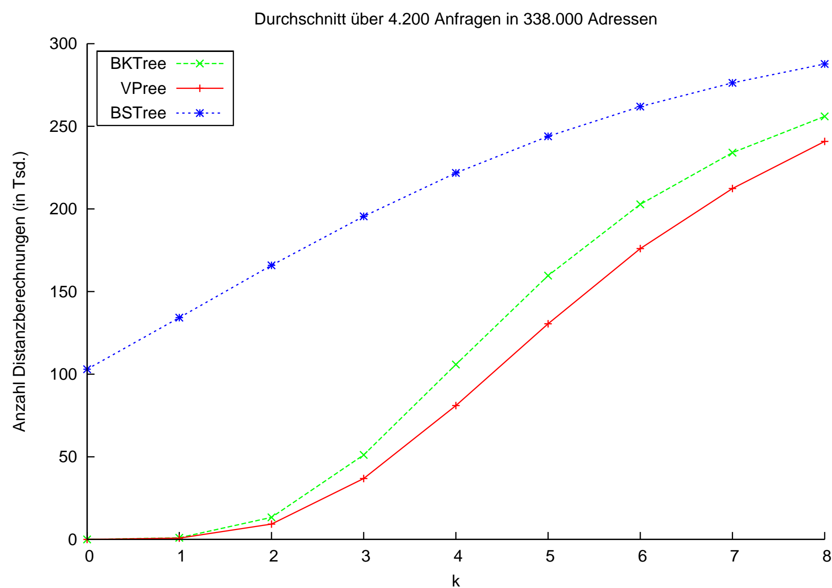
Die aufgetragenen Werte sind Durchschnittswerte über mehr als viertausend Suchanfragen. Die Messung mehrerer Suchanfragen ist zum einen nötig, um sehr kurze Zeiträume möglichst genau zu messen. Zum Anderen hat auch das ausgewählte Suchobjekt Einfluss darauf, wie aufwändig die Suche ist. Daher muss eine möglichst repräsentative Auswahl der Suchbegriffe vorgenommen werden.

Inhalt der Bäume sind `Address`-Objekte. Das verwendete Distanzmaß vergleicht die Anschriften, also eine Verkettung von Straße, Postleitzahl, Ort und Land, unter Zuhilfename der Damerau-Levenshteindistanz.

Klar zu erkennen ist, dass der BS-Baum für kleine Werte von k völlig unbrauchbare Laufzeiten liefert. Schon bei $k = 0$, also bei einer Suche ohne Fehlertoleranz, muss der BS-Baum fast ein Drittel der in ihm enthaltenen Elemente untersuchen, um alle Treffer zu liefern. Der BK-Baum und der VP-Baum verhalten sich dagegen für kleine Werte von k sehr angenehm. Da



(a) Laufzeit pro Suche



(b) Anzahl Distanzberechnungen pro Suche

Abbildung 5.3.: Verhalten bei der Suche abhängig von k

aber eine größere Fehlertoleranz den durchsuchten Teil der Bäume sozusagen „verbreitert“, ist eine überproportionale Abhängigkeit der Laufzeit von k zu erkennen. Ab $k = 5$ kehrt sich dieser Trend jedoch um. Der Grund hierfür wird sein, dass die äußeren Ränder der Bäume erreicht sind und sich die Suche in diese Richtung nicht weiter ausweiten kann. In unendlich großen Bäumen wäre dieser Effekt nicht zu beobachten.

Unterschiede in konstanten Faktoren, die die Laufzeit beeinflussen, kommen erst bei großen Werten für k zum Tragen. Bei $k = 8$ ist sogar zu beobachten, dass die Suche in einem BK-Baum langsamer ist als in einem BS-Baum. Und das, obwohl der BK-Baum auch dort weniger Distanzberechnungen durchführt. Führt man diesen Test mit kürzeren Zeichenketten durch, zeigt sich, dass der Wert von k kleiner wird, ab dem der BS-Baum schneller ist, als der BK-Baum. Dieser Effekt lässt sich dadurch erklären, dass die Distanzberechnungen eine geringeren Anteil an der Gesamtlaufzeit ausmachen, weil die Zeichenketten kürzer sind. Benutzt man statt der Damerau-Levenshteindistanz eine schneller berechenbare Distanzfunktion, wird dieser Effekt ebenfalls früher deutlich.

Für das Problem des Adressabgleichs ist eine derart hohe Fehlertoleranz allerdings nicht mehr interessant. Nach Tabelle 2.1 auf Seite 7 sind die hier verglichenen Anschriften durchschnittlich etwa 27 Zeichen lang. Für $k = 8$ ergibt sich somit eine Fehlerrate von fast einem Drittel. Anschriften mit solch großen Unterschieden sind kaum als identisch zu betrachten.

Da der VP-Baum unabhängig von k immer die kürzesten Suchlaufzeiten liefert, wird im Weiteren ausschließlich dieser Baum benutzt.

5.2. Abgleiche

5.2.1. Vorbereitungen

Die in einem CSV-ähnlichen Format vorliegenden Adressdateien lagen ursprünglich in unterschiedlichen Strukturen vor. Um die automatische Verarbeitung der Daten nicht unnötig zu verkomplizieren, wurden die Formate einander manuell angeglichen. Die Änderungen beschränkten sich auf das

Umordnen und Löschen ganzer Spalten, da teilweise Daten redundant vorgehalten wurden. Beispielsweise enthielt eine der beiden Dateien eine Spalte „Namenszeile“, die eine Verkettung von Anrede und Vor- und Familiennamen enthielt. Derartige Anpassungen korrespondieren mit dem Arbeitsablauf bei der bisher eingesetzten Lösung (siehe Abschnitt 2.3). Der Unterschied liegt nur darin, dass diese Vorgänge durch ClickIt vereinfacht werden und teilweise automatisiert sind. Außerdem wurde jeder Datensatz mit einer eindeutigen Kennziffer versehen, um die Auswertung der Abgleichresultate zu vereinfachen.

Die Klasse `Address` enthält eine Möglichkeit, die Inhalte aller Adressbestandteile zum Zeitpunkt der Instanzierung einer Normierung zu unterziehen. Hier werden die in Abschnitt 2.2.4 Maßnahmen umgesetzt, das heißt: Leer- und Satzzeichen werden entfernt, die Zeichenketten werden in Großbuchstaben umgewandelt und als Beispiel für eine Ersetzungsregel werden „Strasse“ und „Straße“ zu „Str“ abgekürzt. Diese Maßnahmen haben zwei positive Nebeneffekte: die Zeichenketten werden kürzer, was die Distanzberechnungen beschleunigt, und sie senken den erwarteten maximalen Abstand zwischen zwei als gleichwertig zu betrachtenden Adressbestandteilen, weil einige Fehlerarten von vornherein ausgeschlossen werden können.

5.2.2. Geeignete Schranken

Damit zwei Adressen als äquivalent betrachtet werden können, müssen alle ihre Bestandteile wenigstens ungefähr übereinstimmen. Es folgt ein Versuch, für einige Bestandteile obere Schranken zu ermitteln, bei denen noch mit einer gewissen Sicherheit von einer Übereinstimmung ausgegangen werden kann. Natürlich wird es für dieses Problem keine eindeutige und in allen Fällen richtige Lösung geben. Allerdings kann durch die Verknüpfung mehrerer derartig formulierter Bedingungen mit einer recht hohen Wahrscheinlichkeit auch dann von einer Dublette ausgegangen werden, wenn bei einzelnen Adressbestandteilen eine gewisse Restunsicherheit bleibt.

Nachgebildet werden soll ein ClickIt-Vergleich auf Personenebene.

Name

Namen sind die problematischsten Bestandteile von Adressen. Sie sind meist kurz, viele Namen kommen sehr häufig vor und noch mehr Namen existieren in verschiedenen Varianten, die sich nur in wenigen Zeichen (und im Klang manchmal gar nicht) unterscheiden. Das gilt sowohl für Vor- als auch für Nachnamen. Bei Vornamen kommt noch erschwerend hinzu, dass schon ein einziger Buchstabe über das Geschlecht entscheidet, wie bei „Andrea“ und „Andreas“. Daraus ließe sich nun schließen, dass hier ein eher geringer Maximalabstand gewählt werden sollte. Allerdings kann genau so gut argumentiert werden, dass eben gerade aufgrund der geringen Unterscheidbarkeit vieler Namen ein großzügiger Abstand gewählt werden muss. Ist „Stephanie Müller“ die gleiche Person wie „Stefanie Möller“?

Eine Sichtung der Suchergebnisse für Vornamen mit einem Abstand von höchstens eins bringt bereits viele Treffer, die mit einiger Sicherheit als falsch positiv eingeordnet werden müssen, wie beispielsweise „Ute“ und „Uwe“. Die Mehrzahl der Treffer sind aber alternative Schreibweisen wie „Silvia“ und „Sylvia“ oder „Thorsten“ und „Torsten“. Bei einem Abstand von zwei ergeben sich schon viele grobe Abweichungen wie „Peter“ und „Dieter“ oder „Ulrich“ und „Ulrike“. Alternative Schreibweisen eines Namens mit einem Abstand von zwei sind selten, kommen aber durchaus vor, etwa „Helmut“ und „Hellmuth“. Für Vornamen empfiehlt sich also ein Abstand von maximal zwei. Gleiches gilt aus ähnlichen Gründen für Nachnamen. Um damit Treffer zu vermeiden, die in beiden Namensteilen einen Abstand von zwei haben, soll zusätzlich eine Regel sicherstellen, dass eine Verkettung von Vor- und Nachnamen einen Abstand größer als zwei hat. Damit würden „Stephanie Müller“ und „Stefanie Müller“ sowie „Stephanie Möller“ und „Stephanie Müller“ als gleichwertig erkannt, nicht aber „Stephanie Möller“ und „Stephanie Müller“, was recht glaubwürdig erscheint.

Die Implementierung ermöglicht solche Verknüpfungen durch Angabe entsprechender Pläne mit passenden Distanzfunktionen. Allerdings kann sie die Laufzeit des Abgleichs nicht dadurch optimieren, dass sie sich die Abstände einzelner Bestandteile merkt und einen Maximalabstand aus der

Summe voriger Distanzberechnungen betrachtet. Der Abstand zwischen der Verknüpfung von Vor- und Nachname muss neu berechnet werden.

Da die Felder „Namensanhang“ und „Firma“ nur äußerst selten gefüllt sind, lohnt eine isolierte Betrachtung dieser Felder nicht. Sie werden aber in den Vergleich voller Namen mit einbezogen.

Anschrift

Die Anschrift setzt sich zusammen aus der Straße mit der Hausnummer, der Postleitzahl, dem Ort und einem Länderkürzel. Wie in Tabelle 2.1 auf Seite 7 schon zu erahnen ist, ist das Feld Länderkürzel allerdings nicht einheitlich nur mit Kürzeln gefüllt, sondern enthält in manchen Fällen ganze Ländernamen. Daher wird dieses Feld nicht isoliert für Vergleiche verwendet.

Der Großteil der Abweichungen, die bei Straßen vorkommen können, werden bereits durch das Entfernen von Leerzeichen und Punkten und der Abkürzung von „Straße“ durch „Str“ verhindert. Analysiert man die übrigen Abweichungen, beziehen diese sich fast ausschließlich auf die enthaltene Hausnummer. Daher wurde in der Klasse `Address` dafür gesorgt, dass die Hausnummer wenn möglich mit Hilfe eines regulären Ausdrucks aus der Straße herausgelöst und separat gespeichert wird. So reicht für den Abgleich der Straßen schon eine Distanz von eins um einige Schreibfehler („Haupstr“ und „HauptStr“) oder geringe Abweichungen („Am Teich“, „Am Deich“) zu erkennen. Die Hausnummer separat zu vergleichen, ergibt keine brauchbaren Ergebnisse, weil sie sehr kurz ist und daher schon bei sehr kurzen Abständen zu vielen Treffern führt.

Für den Vergleich von Städten wird ebenfalls eine Distanz von eins vorgeschlagen. Bei einer Sichtung der Suchergebnisse bei einem Abstand von zwei werden fast auschließich falsche Treffer gefunden, etwa „Hamburg“ und „Marburg“ oder „München“ und „Büchen“.

Der Vergleich von Postleitzahlen mit Hilfe der Distanzfunktionen ist schwierig. Eine Sichtung der Suchergebnisse ergibt keine interpretierbaren Ergebnisse, weil sie isoliert betrachtet nicht auf Fehler untersucht werden können. Auf der einen Seite dürfen sie, um die Zustellfähigkeit sicherzu-

stellen, möglichst gar nicht voneinander abweichen. Auf der anderen Seite sind auch hier Schreib- oder Lesefehler denkbar. Daher wird auch hier ein Maximalabsstand von eins vorgeschlagen.

Für den Gesamtabgleich einer Anschrift als Verkettung von Straße, Hausnummer, Postleitzahl, Ort und Länderkürzel wird ein Maximalabstand von zwei vorgeschlagen.

5.2.3. Optimale Prüfreihenfolge

Die Reihenfolge, in der alle formulierten Bedingungen geprüft werden, hat keinen Einfluss auf das Ergebnis eines Abgleichlaufs. Jedoch ergeben sich bei unterschiedlichen Anordnungen der Prüfungen erhebliche Unterschiede in der Laufzeit, wenn der zweite in Abschnitt 4.5.2 vorgestellte Algorithmus verwendet wird.

Die Betrachtung des Laufzeitverhaltens der Suche in den Baumstrukturen hat ergeben, dass diese stark von der Laufzeit der verwendeten Distanzfunktion und dem erlaubten Maximalabstand der Ergebnisse vom zu suchenden Objekt abhängt. Die Laufzeit der bereits in Abschnitt 5.1.3 vorgestellten Algorithmen zum In-Sich-Abgleich, der Zusammenführung und dem Negativabgleich von Adressdateien hängt wiederum von den folgenden drei Faktoren ab (absteigend nach Relevanz sortiert):

1. Der Laufzeit der Suche im Baum, weil diese für jede zu suchende Adresse ausgeführt wird.

2. Der Anzahl der Suchergebnisse im Baum, weil die übrigen Pläne für sie geprüft werden müssen.

3. Der Reihenfolge der übrigen durch die Pläne, weil ihre Prüfung sofort abgebrochen werden kann, wenn für einen Plan keine Übereinstimmungen gefunden werden.

Das bedeutet, für den ersten Filterschritt sollte eine Funktion verwendet werden, die möglichst schnell zu berechnen ist, nur eine geringe Fehlertoleranz zuläßt und gleichzeitig möglichst wenige Suchtreffer bei dieser

Feld	Abstand	∅ Laufzeit (ms)	∅ Treffer
Vorname	2	11,7	3512,9
Nachname	2	49,7	1096,5
Straße	1	3,9	157,7
Postleitzahl	1	1,3	1510,7
Ort	1	2,4	438,8
Voller Name	2	86,9	5,5
Anschrift	2	75,9	1,3

Tabelle 5.1.: Eigenschaften der Suche nach Adressmerkmalen

Fehlertoleranz liefert. Tabelle 5.1 gibt diese Informationen für die zu vergleichenden Felder mit dem im vorigen Abschnitt ermittelten Abstand wieder.[2]

Als Kandidaten für das erste Filterkriterium kommen nach diesen Angaben Straße, Postleitzahl und der Ort in Frage. Danach sollten die Namen und zuletzt die aus mehreren Feldern zusammengesetzten Attribute geprüft werden.

5.3. Vergleich mit ClickIt

5.3.1. Laufzeit

Genaue Laufzeitmessungen von ClickIt konnten leider nicht durchgeführt werden. Diese Zeiten wären ohnehin nicht vergleichbar gewesen, da ClickIt auf einem dedizierten System mit zwei CPUs und acht Gigabyte RAM läuft, während die Laufzeittests auf einem normalen Arbeitsplatz-PC von Gruner + Jahr durchgeführt wurden.

Um dennoch ungefähre Richtgrößen anzugeben: ClickIt konnte die zwei Dateien mit zusammen 430.000 Testadressen in ungefähr fünf Minuten in-sich und jeweils gegeneinander (in Form eines Negativabgleichs) abgleichen.

[2]Die angegebenen Trefferzahlen und Laufzeiten sind Durchschnittswerte, die durch eine Suche von 5% der Adressen der kleineren Adressdatei in einem VP-Baum mit allen Adressen der größeren Datei ermittelt wurden.

Die Eigenimplementierung schaffte diese Aufgaben (unter Berücksichtigung der ermittelten optimalen Prüfreihenfolge) auf dem Gruner + Jahr-PC in etwa fünfzig Minuten. Das selbstgesteckte Laufzeitziel, auch größe Dateien innerhalb einer Nacht abgleichen zu können, wurde damit erreicht.

5.3.2. Qualität

Exemplarisch für die verschiedenen Vergleichstypen wurden die zwei vorhandenen Adressdateien jeweils in-sich abgeglichen und mit dem Resultat des In-Sich-Abgleichs auf Personenebene von ClickIt verglichen. Tabelle 5.2 stellt die verschiedenen Ergebnisse gegenüber.

Zunächst werden für beide Programme die Gesamtzahl der erkannten Dubletten und die Anzahl derjenigen Dubletten angegeben, die erkannt wurden, obwohl sie nicht vollkommen identisch sind. Die Differenz beider Zahlen ergibt also die Zahl exakt übereinstimmender Dubletten an. Diese Differenz ist – wenig überraschend – für beide Dateien gleich.

Darauf folgen die Mengen der Adressen, die von beiden Programmen übereinstimmend als Dubletten oder eindeutig identifiziert wurden. Für die Dubletten wird wieder separat die Anzahl der der nur ungefähr übereinstimmenden Adressen angegeben. Hier wurden auch diejenigen Dubletten berücksichtigt, für die ClickIt einen anderen Repräsentanten gewählt hat, als die Eigenentwicklung. Diese Zuordnung war nur möglich, weil das eigene Programm protokollieren kann, welche Adresse es trotz einer Übereinstimmung mit einer anderen Adresse in der Ergebnisdatei behalten hat.

Schließlich sind die von beiden Programmen unterschiedlich behandelten Adressen angegeben. „Falsch positiv“ sind diejenigen Adressen, die von der Eigenentwicklung als Dubletten erkannt wurden, aber nicht von ClickIt. Bei „Falsch negativ“ gilt das selbe, nur umgekehrt.

Dennoch lassen die Angaben in der Tabelle einige Aussagen über die Qualität des eigenen Abgleichs zu. Es wäre nun verführerisch, den Anteil der Übereinstimmungen des eigenen Resultats mit dem von ClickIt auf der Basis *aller* gleich behandelten Adressen zu berechnen. So ergäbe sich im Falle von Datei 2 eine Übereinstimmung von über 99%. Doch für eine

		Datei 1	Datei 2
ClickIt	Erkannte Dubletten	3.136	819
	Davon unscharf	249	394
Eigene Lösung	Erkannte Dubletten	3.012	526
	Davon unscharf	125	101
Übereinstimmungen	Dubletten	2.998	498
	Davon unscharf	111	73
	Eindeutige	79.199	347.245
Abweichungen	Falsch Positive	14	28
	Falsch Negative	138	321

Tabelle 5.2.: Vergleich In-Sich-Abgleiche

seriöse Beurteilung der Leistungsfähigkeit eines solchen Systems ist nur interessant, wie gut es gerade die nicht exakt übereinstimmenden Adressen erkennt. Damit ergibt sich für Datei 1 eine Erkennungsrate von 44% ($\frac{111}{249}$) und für Datei 2 lediglich 18% ($\frac{73}{394}$). Positiv herauszuheben sind die geringen Anteile der fälschlich als Dubletten erkannten Adressen in beiden Dateien. Die Hauptursache für die Abweichungen liegt also in der großen Zahl der nicht erkannten Dubletten.

Eine stichprobenartige Durchsicht der falsch erkannten Adressen ergab, dass etwa drei Viertel der nicht erkannten Dubletten auf Abweichungen im Vornamen zurückzuführen sind. Die häufigste Form der nicht erkannten Abweichungen bei Vornamen sind Kurzformen, wie „Lilo“ für „Liselotte“, „Dieter“ für „Hans-Dietrich“, oder weggelassene Teile von Doppelnamen wie „Hans“ für „Hans-Peter“. Hier bestätigt sich die eingangs geäußerte Vermutung, dass ClickIt domänenspezifisches Wissen einsetzt. Wurden nur Initialen verwendet oder der Vorname insgesamt weggelassen, konnte eine Übereinstimmung zwischen Adressen ebenfalls nicht erkannt werden. Manche der von ClickIt gefundenen Übereinstimmungen sind allerdings auch fragwürdig. So wurden auch zwei Datensätze als Dublette erkannt, wenn der Vorname einmal „Katja“ und einmal „Katharina“ lautete und die übrigen Felder gleich waren. In einem anderen Fall wurde Adresse mit

dem Vornamen „Ruth“ entfernt, obwohl es nur einen „Werner“ und eine „Christa“ mit dem gleichen Nachnamen unter der gleichen Adresse gab. In einem umgekehrten Fall hat ClickIt eine Dublette nicht erkannt, weil der erste Buchstabe des Vornamen fehlte („ichael“ statt „Michael“). In seltenen Fällen führt also auch die Vorgehensweise von ClickIt zu falschen Dubletten.

Das nächste problematische Feld (wenn auch zahlenmäßig weit weniger als der Vorname) sind Nachnamen. Hier führen besonders ausgelassene Teile von Doppelnamen zu nicht erkannten Dubletten. Ein einmal leeres und einmal gefülltes Firmenfeld führte zu etwa genau so vielen nicht erkannten Dubletten. Das Feld Namensanhang führte ebenfalls zu vereinzelten falsch negativ erkannten Adressen. In diesen Fällen sind dort Angaben über die Etage in einem Mehrfamilienhaus oder ähnliche Informationen enthalten.

6. Zusammenfassung

6.1. Bewertung

Die Ergebnisse der Abgleiche von Adressen ausschließlich auf der Basis metrischer Distanzfunktionen sind insgesamt betrachtet schon recht zufriedenstellend. Etwas erstaunlich ist, dass der In-Sich-Abgleich der eigentlich qualitativ hochwertigeren hauseigenen Adressdatei schlechter funktioniert hat, als der Abgleich der schlechteren, externen Datei. Hier zeigt sich, dass die Qualität einer Adressdatei nicht eindimensional messbar ist. Adressen aus verschiedenen Quellen bergen spezifische Probleme, die die Eigenentwicklung nicht in jedem Fall in der Lage zu erkennen im Stande ist. Zudem ist nur eine grobe Parametrisierung der Abgleiche möglich. Ein feiner granuliertes Punktesystem wäre hier wünschenswert, und wenn es nur die Angabe eines Verhältnisses der Fehlerzahl zur Länge der Zeichenketten statt einer statischen maximalen Fehlerzahl erlaubt. Insbesondere bei Vornamen versagt die einfache Anwendung der eingesetzten Distanzfunktion.

Die Laufzeit der Abgleiche ist innerhalb des gesteckten Rahmens geblieben. Ein direkter Vergleich mit ClickIt war wie erwähnt leider nicht möglich, allerdings ist auch hier zu vermuten, dass die Eigenentwicklung nicht voll konkurrenzfähig ist.

Die Differenzen in Laufzeit und Qualität erscheinen allerdings in Anbetracht der Reife von ClickIt und der kurzen Zeit für die Eigenentwicklung jedoch durchaus ermutigend. Hinzu kommt, dass die benutzten Distanzfunktionen und Indexstrukturen sehr flexibel in verschiedenen Anwendungsdomänen einsetzbar sind. Ein handfester Vorteil gegenüber ClickIt ist weiterhin, dasss die eigene Lösung auch für interaktive Einzelsuchen benutzt werden kann. Diese Fähigkeit kann beispielsweise bei der Datenerfassung dazu genutzt werden, die Datenqualität von vornherein zu erhöhen.

6.2. Optimierungspotentiale

Der wohl einfachste Weg, die Laufzeit von Abgleichen zu verkürzen, wäre die Parallelisierung verschiedener Aufgaben. Auf diese Weise könnten die selbst in Standard-PCs immer häufiger anzutreffenden doppelten CPU-Kerne voll ausgelastet werden. Sollen beispielsweise zwei Dateien gegeneinander abgeglichen werden, müssen diese zuerst in-sich abgeglichen werden. Das passiert in der aktuellen Lösung sequentiell, kann aber ohne große Synchronisationsprobleme in zwei parallelen Threads geschehen. Genauso könnte die Suche nach Kandidaten für Dubletten getrennt von der Verifikation der ermittelten Kandidaten getrennt ablaufen. Eventuell ist auch der eigentlich verworfene erste Algorithmus zur Suche nach Dubletten bei konsequenter Parallelisierung schneller. Ist genug Speicher vorhanden, könnten für alle Adressbestandteile Bäume konstruiert werden, die gleichzeitig durchsucht werden. Die Ermittlung von Dubletten wäre dann gerade so schnell, wie die langsamste Suche in einem der Bäume (und die anschließende Bildung der Schnittmenge aller Suchergebnisse). Ob sich derartige Hardwareaufwendungen allerdings wirklich auszahlen, erscheint zweifelhaft.

Eine andere Möglichkeit der Laufzeitoptimierung kann sein, noch andere als die vorgestellten Indexstrukturen zu implementieren und mit den Vorhandenen zu vergleichen. Außerdem bergen der VP-Baum und der BS-Baum selbst noch Optimierungspotential bei der Auswahl der Vantage Points, beziehungsweise günstiger Zentren. Dadurch sind aber nur kleinere Gewinne in der Laufzeit zu erwarten.

Die Erhöhung der Qualität ist ein schwierigeres Problem. Eine Lösung könnte darin bestehen, die metrischen Distanzfunktionen ausschließlich dazu zu benutzen, geeignete Kandidaten zu ermitteln. Diese Kandidaten könnten dann eingehend unter Zuhilfename verschiedener Funktionen getestet werden, die nicht zwingend metrische Eigenschaften aufweisen müssen. Auf diese Weise wäre es auch möglich, ein weniger grobes Punktesystem zu erstellen, dass Aufschluss über die Sicherheit oder Unsicherheit über einen Treffer ausdrücken kann. Ein flexibleres System zur Definition der Gleichheit zweier Adressen kann sich außerdem auch positiv auf die Laufzeit der Abgleiche

auswirken, wenn es nämlich in der Lage ist, Schranken für Kombinationen einzelner Bestandteile auszudrücken, ohne dass diese insgesamt neu berechnet werden müssen.

Interessant wären auch Abgleiche weiterer Adressdateien aus verschiedenen Quellen und auf unterschiedlicher Basis (Haushalte, Firmen). Hier wäre insbesondere zu überprüfen, ob sich die Relevanz der erkannten Schwäche der Lösung in Bezug auf Vornamen bestätigt, oder ob sogar bessere Ergebnisse erzielt werden können.

Angesichts der bisher erreichten durchaus respektablen Qualität, der annehmbaren Laufzeit und der beschriebenen Verbesserungsmöglichkeiten erscheint eine weitere Verfolgung des gewählten Ansatzes lohnenswert.

Literaturverzeichnis

[BCMW94] Baeza-Yates, R. A. ; Cunto, W. ; Manber, U. ; Wu, S.: Proximity Matching Using Fixed-Queries Trees. In: Crochemore, M. (Hrsg.) ; Gusfield, D. (Hrsg.): *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*. Asilomar, CA : Springer-Verlag, Berlin, 1994, S. 198–212

[BEK+03] Batu, T. ; Ergün, F. ; Kilian, J. ; Magen, A. ; Raskhodnikova, S. ; Rubinfeld, R. ; Sami, R.: A sublinear algorithm for weakly approximating edit distance. In: *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, ACM Press, 2003. – ISBN 1–58113–674–9, S. 316–324

[BK73] Burkhard, W. A. ; Keller, R. M.: Some approaches to best-match file searching. In: *Commun. ACM* 16 (1973), Nr. 4, S. 230–236. – ISSN 0001–0782

[BO97] Bozkaya, Tolga ; Ozsoyoglu, Meral: Distance-based indexing for high-dimensional metric spaces. In: *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. New York, NY, USA : ACM Press, 1997. – ISBN 0–89791–911–4, S. 357–368

[BYN98] Baeza-Yates, R. ; Navarro, G.: Fast Approximate String Matching in a Dictionary. In: *Proceedings of the 5th International Symposium on String Processing and Information Retrieval (SPIRE'98)*, IEEE CS Press, 1998, S. 14–22

[CNBYM01] Chávez, E. ; Navarro, G. ; Baeza-Yates, G. ; Marroquín, J.: Searching in Metric Spaces. New York, NY, USA : ACM Press, 2001 (3). – Forschungsbericht. – 273–321 S. http://citeseer.ist.psu.edu/avez99searching.html. – ISSN 0360–0300

[CPZ97] Ciaccia, Paolo ; Patella, Marco ; Zezula, Pavel: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In: *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, Morgan Kaufmann, 1997. – ISBN 1–55860–470–7, S. 426–435

[CR03] Crochemore, Maxime ; Rytter, Wojciech: *Jewels of Stringology.* 5 Toh Tuck Link, Singapore 596224 : Worlds Scientific Publishing Co. Pte. Ltd., 2003

[CTTFF02] Caetano Traina, Jr. ; Traina, Agma ; Filho, Roberto S. ; Faloutsos, Christos: How to improve the pruning ability of dynamic metric access methods. In: *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management.* New York, NY, USA : ACM Press, 2002. – ISBN 1–58113–492–4, S. 219–226

[Dam64] Damerau, Fred J.: A technique for computer detection and correction of spelling errors. In: *Commun. ACM* 7 (1964), Nr. 3, S. 171–176. – ISSN 0001–0782

[FCNP06] Figueroa, Karina ; Chávez, Edgar ; Navarro, Gonzalo ; Paredes, Rodrigo: On the Least Cost For Proximity Searching in Metric Spaces. In: *Proc. 5th Workshop on Efficient and Experimental Algorithms (WEA'06)*, 2006 (LNCS 4007), S. 279–290

[Fre05] Fredriksson, Kimmo: Exploiting distance coherence to speed up range queries in metric indexes. In: *Inf. Process. Lett.* 95 (2005), Nr. 1, S. 287–292. – ISSN 0020–0190

[GN00] GANSNER, Emden R. ; NORTH, Stephen C.: An open graph visualization system and its applications to software engineering. In: *Software — Practice and Experience* 30 (2000), Nr. 11, S. 1203–1233

[HS03] HJALTASON, Gisli R. ; SAMET, Hanan: Index-driven similarity search in metric spaces. In: *ACM Trans. Database Syst.* 28 (2003), Nr. 4, S. 517–580. – ISSN 0362–5915

[Hyy03a] HYYRÖ, Heikki: Bit-Parallel Approximate String Matching Algorithms with Transposition. In: *Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE'03)*, 2003

[Hyy03b] HYYRÖ, Heikki: A bit-vector algorithm for computing Levenshtein and Damerau edit distances. In: *Nordic J. of Computing* 10 (2003), Nr. 1, S. 29–39. – ISSN 1236–6064

[KM83] KALANTARI, Iraj ; MCDONALD, Gerard: A Data Structure and an Algorithm for the Nearest Point Problem. In: *IEEE Trans. Software Eng.* 9 (1983), Nr. 5, S. 631–634

[Knu98] KNUTH, Donald E.: *The Art of Computer Programming: Sorting and Searching (Volume 3)*. 2nd ed. Addison-Wesley, 1998

[Lev66] LEVENSHTEIN, Vladimir: Binary codes capable of correcting deletions, insertions, and reversals. In: *Soviet Physics Doklady* 10 (1966), Nr. 8, S. 707–710. – Original auf Russisch erschienen in *Doklady Akademii Nauk SSSR, 163(4): 845–848, 1965.*

[Nav01] NAVARRO, Gonzalo: A guided tour to approximate string matching. In: *ACM Comput. Surv.* 33 (2001), Nr. 1, S. 31–88. – ISSN 0360–0300

[NR98] NAVARRO, Gonzalo ; RAFFINOT, Mathieu: Fast and Flexible String Matching by Combining Bit-parallelism and Suffix Automata. 1998 (TR/DC–98–4). – Forschungsbericht

[NR02] NAVARRO, Gonzalo ; RAFFINOT, Mathieu: *Flexible Pattern Matching in Strings.* The Edinburgh Building, Cambridge CB2 2RU, UK : Cambridge University Press, 2002

[Sha05] SHAFRANOVICH, Y. *Common Format and MIME Type for Comma-Separated Values (CSV) Files.* RFC 4180 (Informational). Oktober 2005

[Uhl91] UHLMANN, Jeffrey K.: Satisfying General Proximity/Similarity Queries with Metric Trees. In: *Inf. Process. Lett.* 40 (1991), Nr. 4, S. 175–179

[WL75] WAGNER, Robert A. ; LOWRANCE, Roy: An Extension of the String-to-String Correction Problem. In: *J. ACM* 22 (1975), Nr. 2, S. 177–183. – ISSN 0004–5411

[WM91] WU, S. ; MANBER, U.: Fast text searching with errors / Dept. of Computer Science, Univ. of Arizona. 1991 (TR-91-11). – Forschungsbericht. http://citeseer.ist.psu.edu/wu91fast.html

[Yia93] YIANILOS, Peter: Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In: *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1993

A. Quellcode

A.1. com.guj.searching.metric.distance

A.1.1. MetricDistance.java

```java
package com.guj.searching.metric.distance;

/**
 * Common base class for all classes that can calculate a metric distance
 * between two objects of a certain type.
 *
 * <p>
 * A function is considered metric when it takes two parameters of the same type
 * and fulfills the following conditions:
 * </p>
 *
 * <ul>
 * <li>Symmetry: d(x, y) = d(y, x)
 * <li>Positivity: d(x, y) >= 0
 * <li>Reflexitivity: d(x, x) = 0
 * <li>Triangular Inequality: d(x, z) <= d(x, y) + d(y, z)
 * </ul>
 *
 * Examples for metric distances are euclidian distances in n-dimensional space,
 * Manhattan distance (google for "Taxicab geometry") and -- in its most simple
 * form -- the discrete metric described by:
 *
 * <p>
 * If x = y then d(x,y) = 0. Otherwise, d(x,y) = 1.
 * </p>
 *
 * <p>
 * Of course, the latter metric does not help at all with fuzzy searching and if
 * such a metric is used in instances of <code>MetricTree</code>, these trees
 * may degenerate into linked lists or other data structures which are
 * inefficient for searching.
 * </p>
 *
 * <p>
```

```
35   * Metric functions may not only be applied to numbers. Non-numeric
       examples for
     * metric functions are Levenshtein (or: edit) distance and Hamming
       distance.
     * </p>
     *
     * <p>
40   * An important restriction that this class poses upon you is that it only
     * allows for discrete metric functions to be implemented (you have to
       return an
     * int).
     * </p>
     *
45   * @param <T>
     *            The type of objects which this distance function applies to.
     *
     * @see com.guj.searching.metric.tree.MetricTree
     */
50  public abstract class MetricDistance<T> {

        /**
         * Counter for the number of runs of the <code>call</code> method.
             Mainly
         * used for debugging and benchmarking. Implementors have to make sure
55       * themselves that this number is updated correctly.
         */
        protected int numCalls = 0;

        /**
60       * Calculate a metric distance between the two given objects. The
             order of
         * objects you pass does not matter because of the symmetry property
             of
         * metric distances.
         *
         * @param first
65       *            The first object.
         * @param second
         *            The second object.
         *
         * @return A discrete distance between the two given objects.
70       */
        public abstract int call(T first, T second);

        /**
         * Returns the number of calls of the distance methods of this
             instance.
75       *
         * @return The number of calls that have been made to the
             <code>call</code>
         *         method since instantiation or the last run of
         *         <code>resetCallCounter</code>.
         */
80      public int numCalls() {
            return this.numCalls;
        }
```

```
        /**
85       * Reset the call counter to zero.
         */
        public void resetCallCounter() {
            this.numCalls = 0;
        }
90  }
```

A.1.2. SimpleLevenshtein.java

```
 1  package com.guj.searching.metric.distance;

    /*
     * Jakarta implementation of the Levenshtein sitance.
 5   * <p>
     * Taken from <http://www.merriampark.com/ldjava.htm>.
     */

    /**
10   * Levenshtein distance (also called "edit distance") is a metric distance
     * function for Strings. It is defined as the minimum number of operations
     * needed to transform one String into another.
     *
     * <p>
15   * Allowed operations are
     *
     * <ul>
     * <li>Deletion of one character
     * <li>Replacement of one charater
20   * <li>Insertion of one character
     * </ul>
     * </p>
     *
     * <p>
25   * For example, lev("ab", "abc") = lev("abc", "ab") = lev("abc", "aac") =
         1. In
     * the first case, there is one insertion of a single character, in the
         second
     * case there is one deletion of a single character and in the last case
         there
     * is one replacement of a single character.
     * </p>
30   *
     * <p>
     * There may be cases where the same number of differing operations lead
         to the
     * transformation of one String into another. This implementation does not
     * favour any specific operation over another one, since all operations
         have the
35   * same cost (one). It is generally possible to assign different costs do
     * different operations, but care has to be taken not to destroy the
         metric
     * property of the original Levenshtein distance so as not to violate the
```

```
     * contract of <code>MetricDistance</code>.
     * </p>
40   *
     * <p>
     * This implementation runs in O(min(n, m)) space and O(n*m) time, where n
         and m
     * are the lengths of the two Strings.
     * </p>
45   */
    public class SimpleLevenshtein extends MetricDistance<String> {

        /**
         * Computes Levenshtein distance between two Strings.
50       * @param first The first String
         * @param second The second String
         * @return The Levenshtein distance between the two given Strings.
         */
        @Override
55      public int call(String first, String second) {
            this.numCalls++;
            if (first == null || second == null) {
                throw new IllegalArgumentException("Strings must not be
                    null");
            }
60
            int n = first.length(); // length of s
            int m = second.length(); // length of t

            if (n == 0) {
65              return m;
            } else if (m == 0) {
                return n;
            }

70          int p[] = new int[n + 1]; // 'previous' cost array, horizontally
            int d[] = new int[n + 1]; // cost array, horizontally
            int _d[]; // placeholder to assist in swapping p and d

            // indexes into strings s and t
75          int i; // iterates through s
            int j; // iterates through t

            char t_j; // jth character of t

80          int cost; // cost

            for (i = 0; i <= n; i++) {
                p[i] = i;
            }
85
            for (j = 1; j <= m; j++) {
                t_j = second.charAt(j - 1);
                d[0] = j;

90              for (i = 1; i <= n; i++) {
                    cost = first.charAt(i - 1) == t_j ? 0 : 1;
```

```
                // minimum of cell to the left+1, to the top+1, diagonally
                     left
                // and up +cost
                d[i] = Math.min(d[i-1]+1, Math.min(p[i]+1, p[i-1]+cost));
95              //d[i] = CollectionUtils.min(
                //        d[i - 1] + 1,
                //        p[i] + 1,
                //        p[i - 1] + cost);
            }
100
            // copy current distance counts to 'previous row' distance
                counts
            _d = p;
            p = d;
            d = _d;
105     }

        // our last action in the above loop was to switch d and p, so p
            now
        // actually has the most recent cost counts
        return p[n];
110 }

}
```

A.1.3. DamerauLevenshtein.java

```
1   package com.guj.searching.metric.distance;

3   /*
     * LingPipe v. 2.0
     * Copyright (C) 2003-5 Alias-i
     *
     * This program is licensed under the Alias-i Royalty Free License
8    * Version 1 WITHOUT ANY WARRANTY, without even the implied warranty of
     * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the Alias-i
     * Royalty Free License Version 1 for more details.
     *
     * You should have received a copy of the Alias-i Royalty Free License
13   * Version 1 along with this program; if not, visit
     * http://www.alias-i.com/lingpipe/licenseV1.txt or contact
     * Alias-i, Inc. at 181 North 11th Street, Suite 401, Brooklyn, NY 11211,
     * +1 (718) 290-9170.
     */
18
    /**
     * Calculates the Damerau-Levenshtein distance between two Strings.
     *
     * <p>
23   * Damerau-Levenshtein is a modification of the original Levenshtein
         distance in
     * that it allows one additional operation. Besides insertion, deletion
         and
     * replacement of single characters, it allows the transposition of two
```

```
 * adjacent characters at the same cost as the other operations.
 * </p>
28 *
 * <p>
 * For example, dam("ba", "ab") = 1.
 * </p>
 *
33 * <p>
 * Note that applying Levenshtein to these two Strings would yield a
     distance
 * of two (one for deletion, one for insertion).
 * </p>
 *
38 * @author Alias-i
 *
 * @see com.guj.searching.metric.distance.SimpleLevenshtein
 */
public class DamerauLevenshtein extends MetricDistance<String> {
43
    /**
     * Calculates the Damerau-Levenshtein distance between two Strings.
     * @param first The first String
     * @param second The second String
48   * @return The Damerau-Levenshtein distance between the two given
         Strings.
     */
    @Override
    public int call(String first, String second) {

53      this.numCalls++;

        if (first.length() == 0) return second.length();
        if (second.length() == 0) return first.length();

58      // switch for min sized lattice slices so that
        // tLen >= sLen
        if (first.length() < second.length()) {
            String temp = first;
            first = second;
63          second = temp;
        }

        // cSeq1.length >= cSeq2.length > 1
        int xsLength = first.length() + 1; // > ysLength
68      int ysLength = second.length() + 1; // > 2


        int[] twoLastSlice = new int[ysLength];
        int[] lastSlice = new int[ysLength];
73      int[] currentSlice = new int[ysLength];

        // x=0: first slice is just inserts
        for (int y = 0; y < ysLength; ++y)
            lastSlice[y] = y; // y inserts down first column of lattice
78
        // x=1:second slice no transpose
```

```
        currentSlice[0] = 1; // insert x[0]
        char cX = first.charAt(0);
        for (int y = 1; y < ysLength; ++y) {
83          int yMinus1 = y - 1;
            currentSlice[y] = Math.min(
                    cX == second.charAt(yMinus1) ? lastSlice[yMinus1] //
                        match
                            : 1 + lastSlice[yMinus1], // subst
                    1 + Math.min(lastSlice[y], // delelte
88                          currentSlice[yMinus1])); // insert
        }

        char cYZero = second.charAt(0);

93      // x>1:transpose after first element
        for (int x = 2; x < xsLength; ++x) {
            char cXMinus1 = cX;
            cX = first.charAt(x - 1);

98          // rotate slices
            int[] tmpSlice = twoLastSlice;
            twoLastSlice = lastSlice;
            lastSlice = currentSlice;
            currentSlice = tmpSlice;
103
            currentSlice[0] = x; // x deletes across first row of lattice

            // y=1: no transpose here
            currentSlice[1] = Math.min(cX == cYZero ? lastSlice[0] //
                match
108                 : 1 + lastSlice[0], // subst
                    1 + Math.min(lastSlice[1], // delelte
                            currentSlice[0])); // insert

            // y > 1: transpose
113         char cY = cYZero;
            for (int y = 2; y < ysLength; ++y) {
                int yMinus1 = y - 1;
                char cYMinus1 = cY;
                cY = second.charAt(yMinus1);
118             currentSlice[y] = Math.min(cX == cY ? lastSlice[yMinus1]
                    // match
                        : 1 + lastSlice[yMinus1], // subst
                        1 + Math.min(lastSlice[y], // delelte
                                currentSlice[yMinus1])); // insert
                if (cX == cYMinus1 && cY == cXMinus1)
123                 currentSlice[y] = Math.min(currentSlice[y],
                            1 + twoLastSlice[y - 2]);
            }
        }
        return currentSlice[currentSlice.length - 1];
128 }

}
```

A.1.4. Hamming.java

```java
package com.guj.searching.metric.distance;

/**
 * Calculates a variation of the Hamming distance between two Strings.
 *
 * <p>
 * Hamming distance is usually defined as the number of characters of two
 * Strings with equal length that are at the same position but not equal.
    This
 * could be viewed as a simplified Levenshtein distance where only
    replacement
 * operations on a single character are allowed.
 * </p>
 *
 * <p>
 * For example, ham("aaa", "aba") = 1.
 * </p>
 *
 * <p>
 * This implementation removes the restriction of the two Strings being of
     equal
 * length by pretending to fill the shorter String with never-matching
 * characters. That means if the first String s1 is of length n and the
    second
 * String s2 is of length n+x, the Hamming distance is defined as the
    distance
 * between the first n characters of both Strings plus x.
 * </p>
 *
 * <p>
 * Example: ham("aaa", "ab") = 2. The replacement of the second character
    costs
 * one and the missing character from the  second String adds an
    additional cost
 * of one.
 * </p>
 *
 * @author schul3
 *
 * @see com.guj.searching.metric.distance.SimpleLevenshtein
 */
public class Hamming extends MetricDistance<String> {

    /**
     * Calculates a variation of the Hamming distance.
     * @param first The first String
     * @param second The second String
     * @return The modified Hamming distance between the two given
         Strings.
     */
    @Override
    public int call(String first, String second) {
        this.numCalls++;
        int distance = 0;
```

```
        int lenS = first.length();
        int lenT = second.length();
        for (int i=0; i < Math.min(lenS, lenT); i++)
50          if (first.charAt(i) != second.charAt(i))
                distance++;
        distance += Math.abs(lenS - lenT);
        return distance;
    }
55
}
```

A.2. com.guj.searching.metric.tree

A.2.1. MetricTree.java

```
1   package com.guj.searching.metric.tree;

    import java.util.ArrayList;
4   import java.util.Collection;
    import java.util.HashSet;
    import java.util.List;
    import java.util.Set;
    import java.util.SortedMap;
9   import java.util.TreeMap;

    import com.guj.searching.metric.distance.MetricDistance;

    /**
14   * Common base class for trees which support fuzzy (inexact) search using
        a
     * metric distance function.
     *
     * A metric distance d() fulfills the following conditions:
     * <ul>
19   * <li>Symmetry: d(x, y) = d(y, x)
     * <li>Positivity: d(x, y) >= 0
     * <li>Reflexitivity: d(x, x) = 0
     * <li>Triangular Inequality: d(x, z) <= d(x, y) + d(y, z)
     * </ul>
24   *
     * At construction time you have to pass an object of the type
     * <code>MetricDistance</code> whose <code>call()</code> method must
        fulfill
     * these properties. Otherwise, search results will be bogus and useless.
     *
29   * This class provides two constructors which implementors are strongly
     * encouraged to implement as well (by using the appropriate super()
        call). One
     * constructor only receives a metric distance function, the other one
        takes an
     * additional Iterable which returns the objects to insert into the tree.
        If the
```

```
     * first constructor is used, tree construction is deferred until the
         create
34   * method is called with an Iterable returning the objects to insert into
         the
     * tree.
     *
     * Please note that instances of this class are practically immutable
         since they
     * do not offer any methods to add or remove specific values from the
         tree. You
39   * can only change the contents of MetricTree instances by calling their
     * construct method which discards all old tree elements and constructs a
         new
     * tree with the values returned by the given Iterable.
     *
     * Implementing classes may of course offer ways to manipulate the
         contents of
44   * the tree but since only few algorithms to build metric trees allow
         this,
     * these methods are not part of this interface.
     *
     * @author schul3
     *
49   * @param <T>
     *            The type of objects contained in this tree. This parameter
         has to
     *            correspond with the type parameter of the distance function
         you
     *            pass at construction time.
     *
54   * @see com.guj.searching.metric.distance.MetricDistance
     * @see com.guj.searching.analysis.TreeGraph
     * @see com.guj.searching.analysis.TreeStatistics
     */
    public abstract class MetricTree<T> {
59
        /**
         * The number of values (objects of type <code>T</code>) stored in
             this
         * tree. Implementors have to make sure themselves that this number is
         * updated correctly.
64       */
        protected int numValues = 0;

        /**
         * The number of nodes in this tree. This may differ from
69       * <code>numValues</code> since a node may store multiple values in
             one
         * node. Implementors have to make sure themselves that this number is
         * updated correctly.
         */
        protected int numNodes = 0;
74
        /**
         * The root node of this tree.
         */
```

```
        protected Node rootNode;
79
        /**
         * The object used for calculation of distances between the values.
             Note
         * that its type parameter has to be equal to the type parameter of
             this
         * tree.
84       */
        protected final MetricDistance<T> distance;

        protected MetricTree(MetricDistance<T> distance) {
            this.distance = distance;
89      }

        protected MetricTree(MetricDistance<T> distance, Iterable<T> values) {
            this(distance);
            this.construct(values);
94      }

        /**
         * (Re-)Constructs the tree with the given values.
         *
99       * If this instance has been created before (by calling the
             constructor with
         * an Iterable or by calling this method explicitly), all old elements
              will
         * be discarded in favor of the new values passed to this method.
         *
         * @param values -
104      *            An Iterable returning objects that shall be inserted
             into this
         *            tree.
         */
        public abstract void construct(Iterable<T> values);

109     /**
         * Answers whether this tree has been already constructed before.
         *
         * @return - Whether this tree has been already constructed before.
         */
114     public boolean isConstructed() {
            return (this.rootNode != null);
        }

        /**
119      * Returns the height of this tree.
         * @return The height if this tree.
         */
        public int getHeight() {
            try {
124             return this.rootNode.height();
            } catch (NullPointerException e) {
                return 0;
            }
        }
```

```
129
        /**
         * Returns the number of nodes in this tree.
         * @return The number of nodes in this tree.
         */
134     public int getNumNodes() {
            return this.numNodes;
        }

        /**
139      * Returns the number of values stored in this tree. A tree may have
             more
         * values than nodes because a node may store more than one value.
         *
         * @return The number of values stored in this tree.
         */
144     public int getNumValues() {
            return this.numValues;
        }

        /**
149      * Takes any number of Match objects and returns a sorted map with all
         * matching values sorted by their distance to the query object. This
             might
         * make handling of search results easier.
         *
         * @param matches - A Collection of Match objects. This is compatible
             to
154      *                 the return type of this class' search()method.
         * @return - A Map with distances as keys and Collections of
         *          objects which have been found to be at this distance from
         *          a previous query object.
         */
159     public SortedMap<Integer, Collection<T>> aggregateMatches(
                Collection<Match> matches) {
            SortedMap<Integer, Collection<T>> aggregation =
                new TreeMap<Integer, Collection<T>>();
            for (Match match: matches) {
164             int distance = match.distance;
                if (!aggregation.containsKey(distance))
                    aggregation.put(distance, new ArrayList<T>());
                aggregation.get(distance).addAll(match.values);
            }
169         return aggregation;
        }


        /**
174      * Searches for objects in this tree which have a distance of at most
         * <code>k</code> to the query object. If you want the search results
             to
         * be sorted by their distance, you can pass the result of this method
              to
         * <code>aggregateMatches</code>.
         *
179      * @param query - An object to search for.
```

```
         * @param k - The maximum distance which defines a search hit.
         * @return A Set of Match objects which save matching objects and
             their
         *         distance to the query object.
         */
184     public Set<Match> search(T query, int k) {
            try {
                return this.rootNode.search(query, k);
            } catch (NullPointerException e) {
                return new HashSet<Match>();
189         }
        }

        /**
         * Returns the root node of this tree.
194      * @return The root node of this tree.
         */
        public Node getRootNode() {
            return this.rootNode;
        }
199
        /**
         * @return The <code>MetricDistance</code> object used for building
         * and searching the tree.
         */
204     public MetricDistance getDistanceStrategy() {
            return this.distance;
        }

        /**
209      * Describes common attributes of all Nodes in a tree.
         *
         * A Node is a recursive data structure defined by one or more values
             that
         * with a distance of zero to each other and zero to many children
             which
         * are nodes themselves. Depending on the actual implementation, every
214      * node may be a completely valid tree.
         *
         * The storage of values and children may differ among implementing
             classes
         * and consequentially has to be defined there.
         */
219     public abstract class Node {

            /**
             * Search this node and all its children recursively for all
                 objects
             * with a maximum difference of <code>k</code> to the query
                 object.
224          * @param query - The object to search for.
             * @param k - The maximum distance allowed for hits.
             * @return A Set of Match objects with all objects in this tree
                 that
             * have a distance of at most k to the query object.
             */
```

```
229         protected abstract Set<Match> search(T query, int k);

            /**
             * Returns the height this subtree.
             * @return The height this subtree.
234          */
            protected abstract int height();

            /**
             * Returns a list of the outgoing edges of this node. May be an
                 emtpy
239          * list if this node is a leaf.
             * @return A list of the outgoing edges of this node.
             */
            public abstract List<Edge> getEdges();

244         /**
             * Returns the direct child nodes of this node.
             * @return The direct child nodes of this node.
             */
            public abstract Collection<Node> getChildNodes();
249
            /**
             * @return A String representation of this node.
             */
            @Override
254         public abstract String toString();
        }


        /**
259      * Small helper class to bundle child nodes with their edge labels.
             It's
         * primary use is plotting trees with TreeGraph objects.
         */
        public class Edge {
            /** The child node connected to this edge. */
264         public final Node childNode;
            /** The label  of this edge. May be empty. */
            public final String label;
            /**
             *  Creates a new Edge object with a given node and label.
269          *
             * @param childNode - The child node that this Edge leads to.
             * @param label - A label attached to this Edge. May be an empty
                 String
             *                 or null.
             */
274         protected Edge(Node childNode, String label) {
                this.childNode = childNode;
                this.label = label;
            }
        }
279
        /**
         * A <code>Match</code> object describes a partial search result. All
```

```java
 * <code>values</code> were found to be in the distance
 * <code>distance</code> to the query object given at search time.
 */
public class Match {

    /**
     * The values that were found during search.
     */
    public final Collection<T> values;

    /**
     * The distance of all <code>values</code> to the query object.
     */
    public final int distance;

    /**
     * Creates a new Match object with given values and distance.
     *
     * @param values - A Collection of objects which were found to be
     *                 of the same distance to a certain other object
     *                 (possibly a search query).
     * @param distance - The distance of all values to a certain other
     *                   object.
     */
    protected Match(Collection<T> values, int distance) {
        this.values = values;
        this.distance = distance;
    }

    /**
     * @return A String representation of this Match object.
     */
    @Override
    public String toString() {
        return Integer.toString(this.distance) +
            this.values.toString();
    }

    /**
     * @param o - The object to compare with this object.
     * @return - True, if the other object contains the same values
         and the
     *         same distance, otherwise false.
     */
    @Override
    public boolean equals(Object o) {
        try {
            Match other = (Match) o;
            return (this.distance == other.distance
                    && this.values.equals(other.values));
        } catch (ClassCastException e) {
            return false;
        }
    }
}
```

```
}
```

A.2.2. BKTree.java

```
package com.guj.searching.metric.tree;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

import com.guj.searching.metric.distance.MetricDistance;

/**
 * This class implements a metric search tree as it was first proposed
 * by Burkhard & Keller in 1973.
 *
 * A special ability of this type of this tree is its ability to add
    elements
 * without reconstruction.
 *
 * This class is merely a wrapper for its root node.
 *
 * @author jrschulz
 *
 * @param <T> The type of the values stored in the
    <code>NodeValues</code>.
 *
 */
public class BKTree<T> extends MetricTree<T>  {

    /**
     * This number is used to determine the initial size of the
     * map of children.
     */
    private static int expectedNumChildren = 20;

    public BKTree(MetricDistance<T> distance) {
        super(distance);
    }

    public BKTree(MetricDistance<T> distance, Iterable<T> values) {
        super(distance, values);
    }

    @Override
    public void construct(Iterable<T> values) {
        this.numNodes = 0;
        this.numValues = 0;
        this.rootNode = this.new BKNode(values);
```

```
49      }

        /**
         * This class implements a slight variation of (the first) one of the
         * algorithms described by Burkhard and Keller in their paper "Some
54       * Approaches for Best-Match File Searching" from 1973.
         *
         * See comments for BKNode#add and BKNode#search for an explanation of
         * the algorithms.
         *
59       * A special property of this type of metric tree is that it allows
             for
         * addition of single or multiple elements to the tree without having
             to
         * reconstruct the complete tree again. In fact, adding an element is
         * of O(log(n)) complexity (in other words, runtime depends on the
             tree's
         * height).
64       *
         * @author schul3
         *
         */
        private class BKNode extends Node {
69
            /**
             * The list of values that this node stores. All values in this
                 list
             * have a distance of zero to each other.
             */
74          private final List<T> values = new ArrayList<T>();

            /**
             * A representative for all values stored in this node.
             *
79           * Since a <code>BKNode</code> will only store different objects
                 in a
             * single node as long as their distance to each other is zero, an
             * arbitrary representative is picked from the list of
             * <code>values</code> and used for distance calculations.
             */
84          private T representative;

            /**
             * A mapping containing the child nodes keyed by their distance to
             * the current node.
89           */
            protected Map<Integer, BKNode> children;

            /**
             * Create a new BKNode with the given properties.
94           *
             * @param value The value to store in this node.
             */
            private BKNode(T value) {
                BKTree.this.numNodes++;
99              this.values.add(value);
```

```
            this.representative = value;
            this.children = new HashMap<Integer,
                BKNode>(expectedNumChildren);
        }

104     /**
         * Creates a BKTree with all values returned by the given
             Iterable.
         *
         * Building of the tree is done at instantiation time (i.e. when
             you
         * call this constructor). If <code>values</code> wraps a lot of
109      * objects and the distance function of the underlying
         * <code>BKTree</code> is expensive, <b>this may take a lot of
             time</b>.
         *
         * @param values - A collection of values to be inserted into the
             tree.
         */
114     private BKNode(Iterable<T> values) {
            BKTree.this.numNodes++;
            this.children = new HashMap<Integer,
                BKNode>(expectedNumChildren);
            this.addAll(values);
        }
119
        /**
         * Adds a single value to to the tree.
         *
         * This is done by determining the distance of <code>value</code>
             to
124      * the <code>representative</code> of this node. There are four
         * possible cases:
         *
         * <ol>
         * <li> This node doesn't store any values yet. In this case,
             store this
129      * one and pick it as the representative of this node.
         * <li> The distance from the new <code>value</code> and this
             node's
         * <code>representative</code> is zero. Then <code>value</code> is
         * just added to this node's list of values.
         * <li> If the distance is not zero, the action depends on whether
              we
134      * already have a child with the same distance to our
         * <code>representative</code>. If not, we create a new child node
         * and store it in our <code>children</code> map. Otherwise, we
             add
         * <code>value</code> to the child node with the same distance to
             our
         * <code>representative</code> as <code>value</code>.
139      * </ol>
         *
         * This is a tail-recursive method.
         *
         * @param value -
```

```
144          *            The object to insert into this Node.
             */
            private void add(T value) {
                BKTree.this.numValues++;
                if (this.values.isEmpty()) {
149                 // 1st case
                    this.values.add(value);
                    this.representative = value;
                    return;
                }
154             int distance = BKTree.this.distance.call(
                        this.representative, value);
                if (distance == 0) {
                    // 2nd case
                    this.values.add(value);
159                 return;
                }
                BKNode child = this.children.get(distance);
                if (child == null) {
                    // 3a
164                 child = new BKNode(value);
                    this.children.put(distance, child);
                } else {
                    // 3b case
                    child.add(value);
169             }
            }

            /**
             * Adds any number of values into this Node.
174          * @param values - The values to add to this Node.
             */
            private void addAll(Iterable<T> values) {
                for(T value: values) {
                    this.add(value);
179             }
            }

            @Override
            protected Set<Match> search(T query, int k) {
184             Set<Match> matches = new HashSet<Match>();
                Integer fromPtoQ = BKTree.this.distance.call(
                        this.representative, query);
                if (fromPtoQ <= k) {
                    matches.add(new Match(this.values, fromPtoQ));
189             }
                for (Integer fromPtoI: this.children.keySet()) {
                    if (fromPtoQ - k <= fromPtoI && fromPtoI <= fromPtoQ + k)
                         {
                        BKNode child = this.children.get(fromPtoI);
                        matches.addAll(child.search(query, k));
194                 }
                }
                return matches;
            }
```

```
199         @Override
            public  Collection<Node> getChildNodes() {
                List<Node> childNodes = new
                    ArrayList<Node>(expectedNumChildren);
                childNodes.addAll(this.children.values());
                return childNodes;
204         }

            @Override
            public int height() {
                Collection<Integer> childHeights =
209                 new ArrayList<Integer>(this.children.size());
                childHeights.add(0);
                for (BKNode child: this.children.values()) {
                    childHeights.add(child.height());
                }
214             int subtreeMax = 0;
                subtreeMax = Collections.max(childHeights);
                return 1 + subtreeMax;
            }

219         @Override
            public String toString() {
                if (this.values.isEmpty()) {
                    return "";
                } else {
224                 return this.representative.toString();
                }
            }

            @Override
229         public List<Edge> getEdges() {
                List<Edge> edges = new ArrayList<Edge>();
                List<Integer> distances = new
                    ArrayList<Integer>(this.children.keySet());
                Collections.sort(distances);
                for (int distance: distances) {
234                 String label = Integer.toString(distance);
                    Node child = this.children.get(distance);
                    edges.add(new Edge(child, label));
                }
                return edges;
239         }

        }

    }
```

A.2.3. VPTree.java

```
1   package com.guj.searching.metric.tree;
2
    import java.util.ArrayList;
    import java.util.Collection;
```

```
    import java.util.HashSet;
    import java.util.Iterator;
7   import java.util.List;
    import java.util.Set;
    import java.util.SortedMap;
    import java.util.TreeMap;

12  import com.guj.searching.metric.distance.MetricDistance;

    /**
     * This class represents a metric search tree as it was first proposed by
     * Yianilos in his paper "Data structures and algorithms for nearest
        neighbour
17   * search in general metric spaces" (1993). VP stands for "vantage point".
     *
     * This tree is built by picking a random element from to the total list
        of
     * elements to add. This element is called "vantage point" and is stored
        as the
     * value of the current node. The distance from every other element of the
         list
22   * to the vantage point is calculated. Then the list is split in
        (approximately)
     * two halves by determining the median. All elements which have a
        distance to
     * the vantage point smaller than the median are recursively inserted into
         the
     * left child node. All values with a distance larger than (or equal to)
        the
     * median are inserted into the right child node. This process is called
        "ball
27   * decomposition" by the author above.
     *
     * @author schul3
     *
     * @param <T> The type of objects that are to be stored in the tree.
32   */
    public class VPTree<T> extends MetricTree<T> {

        /**
         * @param <T> - The type of objects which are in the values if the map
             to be
37       *            passed.
         * @param dValues - A Map with objects and their distances to a
            specific
         *                other object.
         * @return The median distance of all previously passed dValues.
         */
42      private static <T> int determineMedianDistance(
                SortedMap<Integer, List<T>> dValues) {
            int medianDistance = 0;
            int totalNumValues = 0;
            for (List v: dValues.values()) {
47              totalNumValues += v.size();
            }
            int sumUpTocurrent = 0;
```

```
            for (int d: dValues.keySet()) {
                sumUpTocurrent += dValues.get(d).size();
52              if (sumUpTocurrent <= totalNumValues / 2) {
                    medianDistance =  d;
                } else {
                    break;
                }
57          }
//          System.out.println(medianDistance);
            return medianDistance;
        }

62      public VPTree(MetricDistance<T> distance, Iterable<T> values) {
            super(distance, values);
        }

        public VPTree(MetricDistance<T> distance) {
67          super(distance);
        }

        @Override
        public void construct(Iterable<T> values) {
72          this.numNodes = 0;
            this.numValues = 0;
            this.rootNode = this.new VPNode(values);
        }

77      /**
         *
         * @author schul3
         *
         */
82      private class VPNode extends Node {

            private T representative;

            private final List<T> values = new ArrayList<T>();
87
            private VPNode left;

            private VPNode right;

92          private int medianDistance = -1;

            /**
             * Creates a new VPTree using all values returned by the given
                 Iterable.
             *
97           * @param values -
             *            An Iterable returning values to insert into the
                 tree.
             */
            private VPNode(Iterable<T> values) {
                VPTree.this.numNodes++;
102             List<List<T>> childValues = this.decompose(values);
                this.distribute(childValues);
```

```
            }

            /**
107          * @param values - Objects that are to be inserted into this node.
             */
            private VPNode(List<T> values) {
                VPTree.this.numNodes++;
                List<List<T>> childValues = this.decompose(values);
112             this.distribute(childValues);
            }

            private List<List<T>> decompose(Iterable<T> values) {
                // these values will be inserted into the left side of the
                    tree
117             List<T> leftValues = new ArrayList<T>();
                // the same for the right side
                List<T> rightValues = new ArrayList<T>();
                // this list of list will be returned after filling
                // leftValues and rightvalues.
122             List<List<T>> childValues = new ArrayList<List<T>>(3);
                childValues.add(leftValues);
                childValues.add(rightValues);
                Iterator<T> iter = values.iterator();
                if (iter.hasNext()) {
127                 // The vantage point may be chosen arbitrarily, so we
                        simply
                    // pick the first one.
                    this.addValueHere(iter.next());
                } else {
                    return childValues;
132             }
                SortedMap<Integer, List<T>> dValues =
                    new TreeMap<Integer, List<T>>();
                while (iter.hasNext()) {
                    T nextValue = iter.next();
137                 int curDistance = VPTree.this.distance.call(
                            this.representative, nextValue);
                    if (curDistance == 0) {
                        this.addValueHere(nextValue);
                    }
142                 else {
                        List<T> atCurDistance = dValues.get(curDistance);
                        if (atCurDistance == null) {
                            atCurDistance = new ArrayList<T>();
                        }
147                     atCurDistance.add(nextValue);
                        dValues.put(curDistance, atCurDistance);
                    }
                }
                this.medianDistance = VPTree.determineMedianDistance(dValues);
152             for (int distance : dValues.keySet())
                    if (distance < this.medianDistance)
                        leftValues.addAll(dValues.get(distance));
                    else
                        rightValues.addAll(dValues.get(distance));
157             return childValues;
```

```
        }

        private void distribute(List<List<T>> childValues) {
162         List<T> left = childValues.remove(0);
            if (!left.isEmpty())
                this.left = new VPNode(left);
            List<T> right = childValues.remove(0);
            if (!right.isEmpty())
167             this.right = new VPNode(right);
        }

        @Override
        protected Set<Match> search(T query, int k) {
172         Set<Match> matches = new HashSet<Match>();
            if (this.values.isEmpty())
                return matches;
            Integer distanceToThis = VPTree.this.distance.call(
                    this.representative, query);
177         if (distanceToThis <= k) {
                matches.add(new Match(this.values, distanceToThis));
            }
            if (this.medianDistance == -1)
                // we have no children so we don't need to search further
182             return matches;
            if (distanceToThis - k < this.medianDistance && this.left !=
                null) {
                matches.addAll(this.left.search(query, k));
            }
            if (distanceToThis + k >= this.medianDistance && this.right !=
                 null) {
187             matches.addAll(this.right.search(query, k));
            }
            return matches;
        }

192     private void addValueHere(T value) {
            this.values.add(value);
            VPTree.this.numValues++;
            this.representative = value;
        }
197
        @Override
        protected int height() {
            int leftHeight;
            int rightHeight;
202         if (this.left == null) {
                leftHeight = 0;
            } else {
                leftHeight = this.left.height();
            }
207         if (this.right == null) {
                rightHeight = 0;
            } else {
                rightHeight = this.right.height();
            }
```

```
212             return 1 + Math.max(leftHeight, rightHeight);
            }

            @Override
            public List<Edge> getEdges() {
217             List<Edge> edges = new ArrayList<Edge>(3);
                String label = "\"<\"";
                Node child = this.left;
                if (child != null)
                    edges.add(new Edge(child, label));
222             label = "\">=\"";
                child = this.right;
                if (child != null)
                    edges.add(new Edge(child, label));
                return edges;
227         }

            @Override
            public Collection<Node> getChildNodes() {
                Collection<Node> children = new ArrayList<Node>(3);
232             children.add(this.left);
                children.add(this.right);
                return children;
            }

237         @Override
            public String toString() {
                return this.representative.toString()
                    + " {" + Integer.toString(this.medianDistance) + "}";
            }
242
        }

    }
```

A.2.4. BSTree.java

```
 1  package com.guj.searching.metric.tree;

    import java.util.ArrayList;
    import java.util.Collection;
 5  import java.util.HashSet;
    import java.util.Iterator;
    import java.util.List;
    import java.util.Set;

10  import com.guj.searching.metric.distance.MetricDistance;

    /**
     * This class implements a type of metric search tree called "Bisector
         Tree".
     *
15   *
     * @author schul3
```

```
 *
 * @param <T>
 */
20 public class BSTree<T> extends MetricTree<T> {

    public BSTree(MetricDistance<T> distance, Iterable<T> values) {
        super(distance, values);
    }
25
    public BSTree(MetricDistance<T> distance) {
        super(distance);
    }

30  @Override
    public void construct(Iterable<T> values) {
        this.numNodes = 0;
        this.numValues = 0;
        this.rootNode = this.new BSNode(values);
35  }


    /**
     * This class describes nodes of a BSTree.
40   *
     * Every node has two Lists of "center" values. One is denoted to be
         the
     * left center, the other one is the right center. The distance
         between all
     * objects in a center is zero (in other words: they are considered to
          be
     * equal by the distance function in use).
45   *
     * For each center there is a representative, which is an arbitrary
         element
     * from the respective center. The representative is used for distance
     * evaluations. The radius variables for each center denote the
         maximum
     * distance of one item in the corresponding subtree.
50   *
     * @author schul3
     *
     */
    private class BSNode extends Node {
55
        /**
         * Left child node.
         */
        private BSNode left;
60
        /**
         * Left center values.
         */
        private List<T> leftCenter = new ArrayList<T>();
65
        /**
         * Representative of left center values.
```

```
         */
        private T leftRepresentative;
70
        /**
         * Maximum distance of children in the left subtree to this node's
              left
         * center.
         */
75      private int leftRadius = 0;

        /**
         * Right child node.
         */
80      private BSNode right;

        /**
         * Right center values.
         */
85      private List<T> rightCenter = new ArrayList<T>();

        /**
         * Representative of right center values.
         */
90      private T rightRepresentative;

        /**
         * Maximum distance of children in the left subtree to this node's
              right
         * center.
95       */
        private int rightRadius = 0;

        /**
         * Recursively create a new BSNode with all objects returned by
             the
100      * Iterable values.
         *
         * @param values -
         *            The values to insert into this BSNode.
         */
105     private BSNode(Iterable<T> values) {
            BSTree.this.numNodes++;
            List<List<T>> childValues = this.decompose(values);
            this.distribute(childValues);
        }
110
        private List<List<T>> decompose(Iterable<T> values) {
            List<T> closerToLeft = new ArrayList<T>();
            List<T> closerToRight = new ArrayList<T>();
            List<List<T>> childValues = new ArrayList<List<T>>();
115         childValues.add(closerToLeft);
            childValues.add(closerToRight);
            Iterator<T> iter = values.iterator();
            if (iter.hasNext()) {
                T value = iter.next();
120             this.leftCenter.add(value);
```

```
                BSTree.this.numValues++;
                this.leftRepresentative = value;
            } else {
                return childValues;
125         }
            while (iter.hasNext()) {
                T value = iter.next();
                int distToLeft = BSTree.this.distance.call(
                        this.leftRepresentative, value);
130             if (distToLeft == 0) {
                    this.leftCenter.add(value);
                    BSTree.this.numValues++;
                    continue;
                }
135             this.rightCenter.add(value);
                BSTree.this.numValues++;
                this.rightRepresentative = value;
                break;
            }
140         while (iter.hasNext()) {
                T nextValue = iter.next();
                int distanceToLeft = BSTree.this.distance.call(
                        this.leftRepresentative, nextValue);
                int distanceToRight = BSTree.this.distance.call(
145                     this.rightRepresentative, nextValue);
                if (distanceToLeft == 0) {
                    this.leftCenter.add(nextValue);
                    BSTree.this.numValues++;
                    this.leftRepresentative = nextValue;
150             } else if (distanceToRight == 0) {
                    this.rightCenter.add(nextValue);
                    BSTree.this.numValues++;
                    this.rightRepresentative = nextValue;
                } else if (distanceToLeft <= distanceToRight) {
155                 closerToLeft.add(nextValue);
                    if (distanceToLeft > this.leftRadius)
                        this.leftRadius = distanceToLeft;
                } else {
                    closerToRight.add(nextValue);
160                 if (distanceToRight > this.rightRadius)
                        this.rightRadius = distanceToRight;
                }
            }
            return childValues;
165     }

        private void distribute(List<List<T>> childValues) {
            List<T> left = childValues.remove(0);
            if (!left.isEmpty()) {
170             this.left = new BSNode(left);
            }
            List<T> right = childValues.remove(0);
            if (!right.isEmpty()) {
                this.right = new BSNode(right);
175         }
        }
```

```
    @Override
    protected Set<Match> search(T query, int k) {
180     Set<Match> matches = new HashSet<Match>();
        if (!this.leftCenter.isEmpty()) {
            int distanceToLeft = BSTree.this.distance.call(
                    this.leftRepresentative, query);
            if (distanceToLeft <= k)
185             matches.add(new Match(this.leftCenter,
                    distanceToLeft));
            if (this.left != null && this.leftRadius >= distanceToLeft
                 - k)
                matches.addAll(this.left.search(query, k));
        }
        if (!this.rightCenter.isEmpty()) {
190         int distanceToRight = BSTree.this.distance.call(
                    this.rightRepresentative, query);
            if (distanceToRight <= k)
                matches.add(new Match(this.rightCenter,
                    distanceToRight));
            if (this.right != null
195                 && this.rightRadius >= distanceToRight - k)
                matches.addAll(this.right.search(query, k));
        }
        return matches;
    }
200
    @Override
    protected int height() {
        int leftHeight;
        int rightHeight;
205     if (this.left == null) {
            leftHeight = 0;
        } else {
            leftHeight = this.left.height();
        }
210     if (this.right == null) {
            rightHeight = 0;
        } else {
            rightHeight = this.right.height();
        }
215     return 1 + Math.max(leftHeight, rightHeight);
    }

    @Override
    public List<Edge> getEdges() {
220     List<Edge> edges = new ArrayList<Edge>(3);
        String label = Integer.toString(this.leftRadius);
        Node child = this.left;
        edges.add(new Edge(child, label));
        label = Integer.toString(this.rightRadius);
225     child = this.right;
        edges.add(new Edge(child, label));
        return edges;
    }
```

```
230         @Override
            public Collection<Node> getChildNodes() {
                Collection<Node> children = new ArrayList<Node>(3);
                children.add(this.left);
                children.add(this.right);
235             return children;
            }

            @Override
            public String toString() {
240             String left = "--";
                String right = "--";
                if (!this.leftCenter.isEmpty())
                    left = this.leftCenter.get(0).toString();
                if (!this.rightCenter.isEmpty())
245                 right = this.rightCenter.get(0).toString();
                return left + " <|> " + right;
            }

        }
250
    }
```

A.3. com.guj.searching.address

A.3.1. Address.java

```
 1  package com.guj.searching.address;

    import java.util.regex.Matcher;
 4  import java.util.regex.Pattern;


    /**
     * A simple class to save relevant details of an address.
 9   */
    public class Address {

        public final long ID;
        public final String firstname;
14      public final String surname;
        public final String nameAppendix;
        public final String company;
        public final String street;
        public final String housenumber;
19      public final String zip;
        public final String city;
        public final String country;

        /**
24       * This field will be precomputed during initialization.
         *
         * @see Address#makeFullname()
```

```
         */
        public final String fullname;
29      /**
         * This field will be precomputed during initialization.
         *
         * @see Address#makePostalAddress()
         */
34      public final String postalAddress;

        /**
         * A table which contains replacement rules used by
         * {@link Address#prepareString(String)}. Each entry contains an array
39       * of two String items. The first item is to be replaced by the second
              item.
         *
         *  @see Address#prepareString(String)
         */
        public static final String[][] replacementTable = {
44              { " ", "" }, { ".", "" }, {"-", ""}, {"/", ""},
                { "STRASSE", "STR"}, { "STRAßE", "STR"}
        };

        /**
49       * Makes String uppercase and applies all rules in
         * {@link Address#replacementTable}.
         *
         * @param s -
         *            The String to process.
54       * @return - A standardized String.
         */
        public static String prepareString(String s) {
            s = s.toUpperCase();
            for (String[] rule: replacementTable) {
59              s = s.replace(rule[0], rule[1]);
            }
            return s;
        }

64      /**
         * The regular expression used to split street name and house number.
         */
        public final static Pattern streetPattern =
            Pattern.compile("([^0-9]+)([0-9]+.*)");
69
        /**
         * Expects a String containing a name of a street with a house number
             and
         * tries to split them. The success does not depend on the parameter s
              being
         * preprocessed.
74       *
         * @param s -
         *            A street, possibly with house number.
         * @return - A String array with two Strings in it. The first one is
             the
```

```
         *         street name, the second one is the housenumber. Both
              Strings have
79       *         leading and trailing whitespaces trimmed. If splitting did
              not
         *         succeed, housenumber is the empty string and street equals
              s.
         */
        public static String[] extractHousenumber(String s) {
            String street;
84          String housenumber;
            Matcher m = streetPattern.matcher(s);
            if (m.matches() && m.groupCount() == 2) {
                street = m.group(1).trim();
                housenumber = m.group(2).trim();
89          } else {
                street = s;
                housenumber = "";
            }
            String[] result = { street, housenumber } ;
94          return result;
        }


        /**
99       * Creats a new Address from the given details.
         * <p>
         * @param ID Unique identifier for this address
         * @param firstname the first name of a person/entity
         * @param surname the surname of a person/entity
104      * @param nameAppendix whatever may follow a usual name
         * @param company the name of the company, if any
         * @param street street name
         * @param zip ZIP code
         * @param city city name
109      * @param country country abbreviation (ISO)
         */
        public Address(String ID, String firstname, String surname,
                String nameAppendix, String company, String street, String
                    zip,
                String city, String country) {
114         this(true, ID, firstname, surname, nameAppendix, company, street,
                zip,
                    city, country);
        }

        private Address(boolean doPreprocessing, String ID, String firstname,
119             String surname, String nameAppendix, String company, String
                    street,
                String city, String zip, String country) {
            this.ID = new Long(ID);
            String [] streetAndNr = Address.extractHousenumber(street);
            if (doPreprocessing) {
124             this.firstname = Address.prepareString(firstname);
                this.surname = Address.prepareString(surname);
                this.nameAppendix = Address.prepareString(nameAppendix);
                this.company = Address.prepareString(company);
```

```
                this.street = Address.prepareString(streetAndNr[0]);
129             this.housenumber = Address.prepareString(streetAndNr[1]);
                this.country = Address.prepareString(country);
                this.zip = Address.prepareString(zip);
                this.city = Address.prepareString(city);
            } else {
134             this.firstname = firstname;
                this.surname = surname;
                this.nameAppendix = nameAppendix;
                this.company = company;
                this.street = streetAndNr[0];
139             this.housenumber = streetAndNr[1];
                this.country = country;
                this.zip = zip;
                this.city = city;
            }
144         this.fullname = this.makeFullname();
            this.postalAddress = this.makePostalAddress();
        }


149     /**
         * Creates a new Address from the details in a String-Array.
         * <p>
         * @param line a String-Array with all eight address details in the
         *          same order you had to pass them to the other constructor.
154      *
         */
        public Address(String line[]) {
            this(true, line[0], line[1], line[2], line[3], line[4], line[5],
                    line[7], line[8], line[6]);
159     }

        private String makeFullname() {
            StringBuilder buf = new StringBuilder(20);
            buf.append(this.firstname);
164         buf.append(this.surname);
            buf.append(this.nameAppendix);
            buf.append(this.company);
            return buf.toString();
        }
169
        private String makePostalAddress() {
            StringBuilder buf = new StringBuilder(30);
            buf.append(this.street);
            buf.append(this.housenumber);
174         buf.append(this.city);
            buf.append(this.zip);
            buf.append(this.country);
            return buf.toString();
        }
179
        @Override
        public String toString() {
            StringBuilder buf = new StringBuilder(50);
            buf.append(String.valueOf(this.ID));
```

```
184         buf.append(this.fullname);
            buf.append(this.postalAddress);
            return buf.toString();
        }

189     public String toCSVRecord(char separator) {
            StringBuilder buf = new StringBuilder(50);
            buf.append(this.ID);
            buf.append(separator);
            buf.append(this.firstname);
194         buf.append(separator);
            buf.append(this.surname);
            buf.append(separator);
            buf.append(this.nameAppendix);
            buf.append(separator);
199         buf.append(this.company);
            buf.append(separator);
            buf.append(this.street);
            buf.append(this.housenumber);
            buf.append(separator);
204         buf.append(this.country);
            buf.append(separator);
            buf.append(this.zip);
            buf.append(separator);
            buf.append(this.city);
209         buf.append(System.getProperty("line.separator"));
            return buf.toString();
        }

        @Override
214     public boolean equals(Object other) {
            try {
                return this.ID == ((Address ) other).ID;
            } catch (ClassCastException e) {
                return false;
219         }
        }

        @Override
        public int hashCode() {
224        return (int) this.ID;
        }

    }
```

A.3.2. AddressDistance.java

```
1   package com.guj.searching.address;

3   import com.guj.searching.address.Address;
    import com.guj.searching.metric.distance.MetricDistance;

    /**
     * <p>
```

```
8    * A class wrapping a metric distance for Strings to be able to calculate
        the
     * distance between two addresses. The idea is that subclasses may decide
        which
     * of the various fields an Address instance has should be used for
        distance
     * calculation. The content of the chosen field is supposed to be returned
        by
     * the <code>extractComponent</code> method defined in this class.
13   * </p>
     *
     * <p>
     * In this class, the return value of Address#toString is returned by
     * <code>extractComponent</code>. If you want to subclass this class, you
18   * generally only need to implement this method.
     * </p>
     *
     * @author schul3
     *
23   */
    public class AddressDistance extends MetricDistance<Address> {

        /**
         * The metric distance strategy object to be used to calculate the
            distance
28       * between two address components.
         */
        public final MetricDistance<String> strategy;

        /**
33       * Create a new instance using the given object for distance
            calculations.
         * @param strategy The <code>MetricDistance</code> object to use.
         */
        public AddressDistance(MetricDistance<String> strategy) {
            super();
38          this.strategy = strategy;
        }

        @Override
        public int call(Address s, Address t) {
43          return this.strategy.call(
                    this.extractComponent(s), this.extractComponent(t));
        }

        public String extractComponent(Address a) {
48          return a.toString();
        }

        @Override
        public int numCalls() {
53          return this.strategy.numCalls();
        }

        @Override
        public void resetCallCounter() {
```

```
58          this.strategy.resetCallCounter();
        }


    }
```

A.3.3. FullnameDistance.java

```
 1  package com.guj.searching.address;

 3  import com.guj.searching.address.Address;
    import com.guj.searching.metric.distance.MetricDistance;

    /**
     * A metric distance for Addresses which uses the full name of an Address
 8   * instance for distance calculations.
     *
     * @author schul3
     *
     */
13  public class FullnameDistance extends AddressDistance {

        public FullnameDistance(MetricDistance<String> strategy) {
            super(strategy);
        }
18
        @Override
        public String extractComponent(Address a) {
            return a.fullname;
        }
23
        @Override
        public String toString() {
            return "fullname";
        }
28
    }
```

A.3.4. PostalAddressDistance.java

```
 1  package com.guj.searching.address;

    import com.guj.searching.address.Address;
    import com.guj.searching.metric.distance.MetricDistance;

 6  /**
     * A metric distance for Addresses which uses the postal address of an
     * Address instance for distance calculations.
     *
     * @author schul3
11   *
```

```
 */
public class PostalAddressDistance extends AddressDistance {

    public PostalAddressDistance(MetricDistance<String> strategy) {
        super(strategy);
    }

    @Override
    public String extractComponent(Address a) {
        return a.postalAddress;
    }

    @Override
    public String toString() {
        return "postal";
    }

}
```

A.3.5. ComparisonPlan.java

```
package com.guj.searching.address;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import com.guj.searching.address.Address;
import com.guj.searching.address.AddressDistance;
import com.guj.searching.metric.distance.DamerauLevenshtein;
import com.guj.searching.metric.distance.MetricDistance;
import com.guj.searching.metric.tree.MetricTree;
import com.guj.searching.metric.tree.VPTree;

public class ComparisonPlan {

    private static String logfilename = "output.log";
    private static PrintStream log;
    static {
        try {
            log = new PrintStream(new File(logfilename));
        } catch (FileNotFoundException e) {
            log = System.out;
        }
    }

    public final AddressDistance distance;
```

```
        public final MetricTree<Address> tree;
        public final int maxDistance;
36
        public ComparisonPlan(AddressDistance distance, int maxDistance,
                MetricTree<Address> tree) {
            this.distance = distance;
            this.tree = tree;
41          this.maxDistance = maxDistance;
        }

        public ComparisonPlan(AddressDistance distance, int maxDistance) {
            this.distance = distance;
46          this.maxDistance = maxDistance;
            this.tree = null;
        }

        /**
51       * Returns a Set of Address objects in which there are no duplicates.
         *
         * What a duplicate is is determined by the ComparisonPlans you
             pass.The
         * primaryPlan is used to build up a tree and determine candidate
             matches
         * for each Address. The candidates are then verified by the
56       * verificationPlans. Only Addresses which pass all plans are
             considered
         * dupes. If two addresses are found to be duplicates of each other,
             the
         * first one that was encountered will be kept, the other one will be
         * purged.
         *
61       * The primaryPlan's distance function and maximumDistance should
             ideally be
         * fast to compute and result in only a few candidates at the same
             time. A
         * good candidate for this is the ZIP code (compared by some distance
         * function for Strings), provided that not all of your addresses are
             from
         * the same area.
66       *
         * The verificationPlans' distance function should be ordered in
             ascending
         * order of runtime of their distance functions. Generally this means
             that
         * the first one should compare the shortest address field and the
             last one
         * the longest one.
71       *
         * @param file -
         *            The file to process
         * @param primaryPlan -
         *            A plan which will be used to determine possible dupes
76       * @param verificationPlans -
         *            An arbitrary number of additional plans. Two adresses
             are only
         *            consideres dupes, if they pass the primaryPlan and *all*
```

```
         *           verifications plans
         * @return - A Set of all Addresses in file without duplicates.
81       */
        public static Set<Address> getUniques(Iterable<Address> file,
                ComparisonPlan primaryPlan,
                    ComparisonPlan...verificationPlans) {
            long start = System.currentTimeMillis();
            Set<Address> allAddresses = ComparisonPlan.setFromIterable(file);
86          log.println("Addresses read.");
            Set<Address> dupes = new HashSet<Address>();
            primaryPlan.tree.construct(allAddresses);
            log.println("Tree built.");
            log.printf("Time: %3.1f\n",
91                  (1.0 * (System.currentTimeMillis() - start) / 1000));
            log.println("total # of values: " +
                primaryPlan.tree.getNumValues());
            log.println("# of distinct values: " +
                primaryPlan.tree.getNumNodes());
            int numSkipped = 0;
            int numDone = 0;
96          for (Address a: allAddresses) {
                numDone++;
                if (dupes.contains(a)) {
                    // this Address has already been recognized as a dupe. If
                        we
                    // searched for similar Addresses for it now, we would
                        find the
101                 // Address of which it was found to be a dupe in the first
                    // place, too.
                    numSkipped++;
                    continue;
                }
106             Set<MetricTree<Address>.Match> firstMatches =
                    primaryPlan.tree.search(a, primaryPlan.maxDistance);
                Set<Address> candidates = new HashSet<Address>();
                for (MetricTree<Address>.Match match: firstMatches) {
                    candidates.addAll(match.values);
111             }
                Set<Address> verified = ComparisonPlan.verifyCandidates(
                        a, candidates, verificationPlans);
                dupes.addAll(verified);
                if (numDone % 5000 == 0) {
116               double time = (System.currentTimeMillis() - start) / 1000.0;
                  log.printf("Done: %8s; Skipped %6s in %5.1f; Dupes: %6s\n",
                          numDone, numSkipped, time, dupes.size());
                }
            }
121         allAddresses.removeAll(dupes);
            log.println("# dupes: " + dupes.size());
            log.println("# remaining: " + allAddresses.size());
            return allAddresses;
        }
126
        /**
         * Merges two files into one so that the resulting file does not
             contain any
```

```
         * duplicates.
         *
131      * Duplicates are determined by the primaryPlan and an arbitrary
             number of
         * verificationPlans. All comments of
         * {@link ComparisonPlan#getUniques(Iterable, ComparisonPlan,
             ComparisonPlan[])}
         * regarding these plans apply here as well.
         *
136      * Note that the ordering of the files you pass has two consequences.
             First,
         * the first file (called "large") will be used to build a tree, since
              it is
         * generally faster to build a tree out of the larger file. Second, if
         * duplicate is found (an address which is in large and in small as
             well),
         * the address from the large file will be retained while the other
             one will
141      * be discarded.
         *
         * @param large -
         *            The larger one of the two files.
         * @param small -
146      *            The smaller one of the two files.
         * @param alreadyUnique -
         *            Whether duplicates in each passed file have already been
         *            removed. A plan which will be used to determine possible
              dupes
         * @param verificationPlans -
151      *            An arbitrary number of additional plans. Two adresses
             are only
         *            consideres dupes, if they pass the primaryPlan and *all*
         *            verifications plans
         * @return - A Set of all Addresses in the two given files without
         *         duplicates.
156      */
        public static Set<Address> mergeFiles(Iterable<Address> large,
                Iterable<Address> small, boolean alreadyUnique,
                ComparisonPlan primaryPlan,
                    ComparisonPlan...verificationPlans) {
            Set<Address> forTree;
161         Set<Address> toTest;
            int numDupes = 0;
            if (alreadyUnique) {
                forTree = ComparisonPlan.setFromIterable(large);
                toTest = ComparisonPlan.setFromIterable(small);
166         } else {
                forTree = ComparisonPlan.getUniques(
                        large, primaryPlan, verificationPlans);
                log.println("Made file unique.");
                toTest = ComparisonPlan.getUniques(
171                     small, primaryPlan, verificationPlans);
                log.println("Made file unique.");
            }
            primaryPlan.tree.construct(large);
            for (Address a: toTest) {
```

```
176             Set<Address> candidates = new HashSet<Address>();
                Set<MetricTree<Address>.Match> matches =
                    primaryPlan.tree.search(a, primaryPlan.maxDistance);
                for (MetricTree<Address>.Match match: matches) {
                    candidates.addAll(match.values);
181             }
                Set<Address> verified = ComparisonPlan.verifyCandidates(
                        a, candidates, verificationPlans);
                if (verified.size() < 1) {
                    forTree.add(a);
186             } else {
                    numDupes++;
                }
            }
            log.println("dupes: " + numDupes);
191         log.println("rest: " + forTree.size());
            return forTree;
        }

        /**
196      * Removes all Addresses in file which also occur in toRemove.
         *
         * @see ComparisonPlan#getUniques(Iterable, ComparisonPlan,
         *    ComparisonPlan[])
         * @see ComparisonPlan#mergeFiles(Iterable, Iterable, boolean,
         *    ComparisonPlan, ComparisonPlan[])
         *
201      * @param file
         * @param toRemove
         * @param primaryPlan
         * @param verificationPlans
         * @return
206      */
        public static Set<Address> getAllWithout(Iterable<Address> file,
                Iterable<Address> toRemove, ComparisonPlan primaryPlan,
                ComparisonPlan...verificationPlans) {
            Set<Address> forTree = ComparisonPlan.setFromIterable(file);
211         primaryPlan.tree.construct(file);
            for (Address a: toRemove) {
                Set<Address> candidates = new HashSet<Address>();
                Set<MetricTree<Address>.Match> matches =
                    primaryPlan.tree.search(a, primaryPlan.maxDistance);
216             for (MetricTree<Address>.Match match: matches) {
                    candidates.addAll(match.values);
                }
                Set<Address> verified = ComparisonPlan.verifyCandidates(
                        a, candidates, verificationPlans);
221             for (Address dupe: verified) {
                    forTree.remove(dupe);
                    log.printf("Found [%s] to be a dupe of [%s]:\n", dupe.ID,
                        a.ID);
                    log.printf(dupe.toCSVRecord('\t'));
                    log.printf(a.toCSVRecord('\t'));
226             }
            }
            return forTree;
```

```
    }

231 private static Set<Address> verifyCandidates(Address
        objectOfComparison,
            Set<Address> candidates, ComparisonPlan...plans) {
        Set<Address> verified = new HashSet<Address>();
        for (Address candidate: candidates) {
            boolean isVerified = true;
236         for (ComparisonPlan plan: plans) {
                int dist = plan.distance.call(objectOfComparison,
                    candidate);
                if (objectOfComparison.equals(candidate)
                        || dist > plan.maxDistance) {
                    isVerified = false;
241                 break;
                }
            }
            if (isVerified) {
                verified.add(candidate);
246             log.printf("Found [%s] to be a dupe of [%s]\n",
                        objectOfComparison.ID, candidate.ID);
                log.println(objectOfComparison.toCSVRecord('\t'));
                log.println(candidate.toCSVRecord('\t'));
            }
251     }
        return verified;
    }


256 public static Set<Address> getUniquesUsingTrees(Iterable<Address>
        file,
            ComparisonPlan...plans) {
        long start = System.currentTimeMillis();
        Set<Address> allAddresses = ComparisonPlan.setFromIterable(file);
        log.println("Addresses read.");
261     Set<Address> dupes = new HashSet<Address>();
        for (ComparisonPlan plan: plans) {
            plan.tree.construct(file);
            log.println("Tree built.");
            log.printf("Time: %3.1f\n",
266                 (1.0 * (System.currentTimeMillis() - start) / 1000));
        }
        log.println("Starting to search.");

        int numSkipped = 0;
271     int numDone = 0;
        for (Address a: allAddresses) {
            numDone++;
            if (dupes.contains(a)) {
                // this Address has already been recognized as a dupe. If
                    we
276             // searched for similar Addresses for it now, we would
                    find the
                // Address of which it was found to be a dupe in the first
                // place, too.
                numSkipped++;
```

```
                continue;
281             }
                // initialized with zero to able to recognize first loop
                    execution
                Set<Address> candidates = null;
                for (ComparisonPlan plan : plans) {
                    Set<Address> planResult = new HashSet<Address>();
286                 for (MetricTree<Address>.Match match : plan.tree.search(a,
                            plan.maxDistance)) {
                        planResult.addAll(match.values);
                    }
                    // we will always find the Address we are searching for to
                         be
291                 // similar to itself. Since this is not really a
                        duplicate, we
                    // discard it.
                    planResult.remove(a);
                    if (candidates == null) {
                        // Apparently this is the first loop execution, so the
                             whole
296                     // search result is a list of candidates.
                        candidates = planResult;
                    } else {
                        // Keep only the intersection of the old candidate
                            list
                        // and the current search results.
301                     candidates.retainAll(planResult);
                    }
                    if (candidates.size() < 1) {
                        break;
                    }
306             }
                if (candidates != null) {
                    dupes.addAll(candidates);
                }
                if (numDone % 5000 == 0) {
311               double time = (System.currentTimeMillis() - start) / 1000.0;
                  log.printf("Done: %8s; Skipped %6s in %5.1f; Dupes: %6s\n",
                          numDone, numSkipped, time, dupes.size());
                }
            }
316 //        log.println("total: " + primaryPlan.tree.getNumValues());
            log.println("Dupes: " + dupes.size());
            log.println("Skips: " + numSkipped);
            allAddresses.removeAll(dupes);
            return allAddresses;
321     }

        public static void writeToCSV(Collection<Address> data, PrintStream
            file) {
            for (Address a: data) {
326             file.print(a.toCSVRecord('\t'));
            }
        }
```

```
        public static Set<Address> setFromIterable(Iterable<Address> file) {
331         Set<Address> addresses = new HashSet<Address>();
            try {
                addresses = (Set<Address>) file;
            } catch (ClassCastException e) {
                addresses = new HashSet<Address>();
336             for (Address a: file) {
                    addresses.add(a);
                }
            }
            return addresses;
341     }

        public static void main(String[] args) throws IOException {
            Iterable<Address> large = DataLoader.getLargeSet();

346         MetricDistance<String> dam = new DamerauLevenshtein();

            int maxFirstnameDist = 2;
            int maxSurnameDist = 2;
            int maxStreetDist = 1;
351         int maxCityDist = 1;
            int maxZipDist = 1;
            int maxPostalDist = 2;
            int maxFullnameDist = 2;

356         AddressDistance postal = new PostalAddressDistance(dam);
            AddressDistance fullname = new FullnameDistance(dam);

            AddressDistance firstname = new AddressDistance(dam) {
                @Override
361             public String extractComponent(Address a) {
                    return a.firstname;
                }
                @Override
                public String toString() { return "firstname"; }
366         };

            AddressDistance surname = new AddressDistance(dam) {
                @Override
                public String extractComponent(Address a) {
371                 return a.surname;
                }
                @Override
                public String toString() { return "surname"; }
            };
376
            AddressDistance street = new AddressDistance(dam) {
                @Override
                public String extractComponent(Address a) {
                    return a.street;
381             }
                @Override
                public String toString() { return "street"; }
            };
```

```
386         AddressDistance city = new AddressDistance(dam) {
                @Override
                public String extractComponent(Address a) {
                    return a.city;
                }
391             @Override
                public String toString() { return "city"; }
            };

            AddressDistance zip = new AddressDistance(dam) {
396             @Override
                public String extractComponent(Address a) {
                    return a.zip;
                }
                @Override
401             public String toString() { return "zip"; }
            };

            ComparisonPlan firstnameVP = new ComparisonPlan(
                firstname, maxFirstnameDist, null);
406         ComparisonPlan surnameVP = new ComparisonPlan(
                surname, maxSurnameDist, null);
            ComparisonPlan streetVP = new ComparisonPlan(
                street, maxStreetDist, new VPTree<Address>(street));
            ComparisonPlan cityVP = new ComparisonPlan(
411             city, maxCityDist, new null);
            ComparisonPlan zipVP = new ComparisonPlan(
                zip, maxZipDist, new null);
            ComparisonPlan postalVP = new ComparisonPlan(
                postal, maxPostalDist, null);
416         ComparisonPlan fullnameVP = new ComparisonPlan(
                fullname, maxFullnameDist, null);

            ComparisonPlan primaryPlan = streetVP;
            ComparisonPlan[] verificationPlans=  {
421                 zipVP, cityVP, firstnameVP, surnameVP, fullnameVP,
                        postalVP };

            Set<Address> uniqueB = ComparisonPlan.getUniques(large,
                    primaryPlan, verificationPlans);
        }
426
    }
```

Erklärung der Selbständigkeit

Ich erkläre hiermit, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe. Wörtlich oder sinngemäß aus anderen Werken entnommene Stellen habe ich unter Angabe der jeweiligen Quellen kenntlich gemacht.

Jochen Schulz