Universität Bielefeld
Fakultät für Linguistik und Literaturwissenschaft

Masterarbeit im Studiengang Linguistik:
Kommunikation, Kognition und Sprachtechnologie

Approximativer Stringvergleich in selbstorganisierenden Trie-Strukturen

vorgelegt von Nils Diewald

Erstgutachter: JProf. Dr. Alexander Mehler (Universität Bielefeld)
Zweitgutachter: Dr. Andreas Witt (Universität Tübingen)
Bielefeld, den 12. Juli 2008

Inhaltsverzeichnis

1 Über diese Arbeit **4**

2 Trie-Strukturen **7**
2.1 Ansatz nach Fredkin 10
2.2 Ansatz nach de la Briandais 14
2.3 Ansatz nach Clampett 15

3 Selbstorganisierende Trie-Strukturen **19**
3.1 Listen 21
3.2 Listen-Tries 27
3.3 Binäre Suchbäume 29
3.4 Ternäre Suchbäume 37

4 Approximativer Stringvergleich **40**
4.1 Ähnlichkeitsschlüssel 41
4.2 *n*-Gramm-Vergleiche 43
4.3 Hamming-Distanz 44
4.4 Levenshtein-Distanz 46
4.4.1 Levenshtein-Damerau-Distanz 50
4.5 Weitere Ansätze 52

5 Approximativer Stringvergleich in Lexika **55**
5.1 In Trie-Strukturen 56
5.2 In selbstorganisierenden Trie-Strukturen 60
5.2.1 Bewertung 65
5.3 Weitere Ansätze 66

6 Zusammenfassung **68**

A Korpora **70**

B Anmerkungen **71**

C Algorithmen **72**

Abbildungsverzeichnis **83**

Algorithmenverzeichnis **84**

Literaturverzeichnis **85**

1 Über diese Arbeit

Seit seinen Anfängen gehört das Verfassen und Bearbeiten von Texten zu den zentralen Aufgaben bei der Arbeit an einem Personal Computer. Daher wurden schon früh Methoden entwickelt, den Schreibenden bei diesen Tätigkeiten zu unterstützen. Insbesondere Rechtschreibkorrekturprogramme helfen hierbei, indem sie orthographische Fehler rechtzeitig erkennen und Vorschläge machen, wie mögliche Fehler zu korrigieren sind.

Solche Systeme finden sich heute nicht mehr nur in Textverarbeitungsprogrammen – auch beim Verfassen von E-Mails, Chatnachrichten oder dem Ausfüllen von Formulareingaben auf Webseiten können sie aktiviert werden.

Zu diesem Zweck analysieren sie einen Text, indem sie ihn segmentieren und Wort für Wort mit Einträgen in einem Systemlexikon abgleichen – ist ein Wort nicht im Lexikon enthalten, markiert das Programm das *Nichtwort* als möglichen Fehler und unterbreitet dem Autor Vorschläge für eine Korrektur.[1]

Die Geschwindigkeit solcher Programme ist dabei abhängig von der benötigten Zeit zur Überprüfung jeden Wortes im Lexikon, bei der mögliche Fehler detektiert werden, und von der Zeit, die für die Generierung einer Korrekturkandidatenliste für jedes Wort benötigt wird, das beim ersten Suchlauf nicht gefunden werden konnte. Es handelt sich hierbei um zwei verschiedene Formen der Suche: Dem exakten Nachschlagen im Lexikon, das sehr oft – für jedes Wort im Text – durchgeführt werden muss, und der inexakten, approximativen Suche zur Findung möglichst ähnlicher Wörter zu jedem unbekannten Wort.

Die Güte auf der anderen Seite wird zum einen vom Umfang der im Lexikon enthaltenen Wörter bestimmt und zum anderen von der Art, mit der die ähnlichsten Wörter im Lexikon bei der unscharfen Suche nach Kandidaten für die richtige Schreibung ermittelt werden.

In dieser Arbeit wird hierbei eine bekannte Eigenschaft des zu analysierenden Ge-

[1] Nur diese Art der Rechtschreibfehlerkorrektur wird in dieser Arbeit behandelt. Systeme, die den Kontext oder die grammatikalische Verwendung eines Wortes untersuchen, werden nicht thematisiert. (Vgl. Fliedner, 2004)

genstandes „Text“ ausgenutzt: Die schiefe Verteilung der Häufigkeiten, mit denen Wörter im Text auftreten. Diese Eigenschaft ist wohl bekannt und auch wenn bestimmte Gesetzmäßigkeiten, wie das Zipf-Mandelbrot-Gesetz, nur für komplette, kohärente Einzeltexte ihre Gültigkeit besitzen, die deutlich seltener Gegenstand von Rechtschreibprüfungen sein dürften als etwa Textfragmente, gilt auch hier eine schiefe Verteilung, mit der erwartet werden kann, dass einige Wörter im Text bei der Analyse häufiger an das Lexikon angefragt werden als andere. Diese Wörter sind zum einen Funktionswörter, die unabhängig vom Thema des Textes oft und über viele Texte einer Sprache hinaus auftreten. Andere Wörter, wie in diesem Absatz das Wort „Wörter“, tauchen nicht in allen Texten mit gleicher Frequenz auf – hier fünf Mal.

Die Idee dieser Arbeit ist schnell skizziert: Anstelle der Betrachtung atomarer Anfragen an das Lexikon wird die Überprüfung eines Textes oder Textfragments als eine Anfragesequenz betrachtet, in der einige Wörter häufiger als andere angefragt werden – viele der im Lexikon vorhandenen Wörter gar nicht. Diese schiefe Verteilung soll eine höhere Geschwindigkeit bei der exakten Suche und eine höhere Güte bei der inexakten Suche fördern, indem wahrscheinlichere Kandidaten für ein korrektes Wort in der Liste möglicher Korrekturkandidaten für ein *Nichtwort* höher positioniert werden. Würde sich in diesen Satz versehentlich ein „Würter“ einschleichen, sollte die Rechtschreibkorrektur „Wörter“ in der Kandidatenliste höher gewichten als „Wärter“, da es bereits Element der Anfragesequenz war.

Diese Gewichtung sollte ohne Weltwissen und im besten Fall ohne ein „Merken“ der vergangenen Anfragen geschehen. Zentral hierfür ist die Form der elektronischen Repräsentation des Lexikons. In Kapitel 2 wird mit der Datenstruktur des Tries eine der verbreitetsten Datenstrukturen für den Zweck der Lexikonorganisation vorgestellt, dessen unterschiedliche Implementationsvarianten unter den Aspekten der Geschwindigkeit und des Speicherbedarfs untersucht werden. Darauf aufbauend wird in Kapitel 3 eine Verbesserung der Organisation der Wörter innerhalb der Datenstruktur durch selbstorganisierende Heuristiken diskutiert, wodurch oben erwähnte Verbesserung in Bezug auf die Geschwindigkeit exakter sowie die Güte bei inexakter Suche eintreten soll.

Um in den Bereich dieser unscharfen Suche einzuleiten, werden in Kapitel 4 Varianten des unscharfen Zeichenkettenvergleichs thematisiert, von denen die Levenshtein-Damerau-Distanz im Folgekapitel 5 für die inexakte Suche in selbstorganisierenden Trie-Strukturen angewendet wird.

In den Kapiteln 3 und 5 werden Experimente vorgestellt, mit denen die Effizienz

der Methoden untersucht werden soll. Als Text wurde hierfür der Roman „Little Brother“ von Cory Doctorow verwendet (*Cory*), als Vergleichskorpus dient der Roman „Moby Dick“ von Herman Melville (*Moby*). Einige Experimente nutzen zudem ein Korpus bestehend aus 50000 Wörtern der deutschsprachigen Wikipedia (*Wiki*).[2]

[2] Eine ausführliche Beschreibung der Korpora findet sich in Anhang A.

2 Trie-Strukturen

Ein Lexikon ist ein Inventar von Einträgen, die durch Schlüsselwörter indiziert sind. In Buchform sind Lexika meist alphabetisch nach ihren Schlüsselwörtern sortiert – dies erleichtert dem menschlichen Sucher das Auffinden eines Eintrags. Schlägt dieser das Buch an beliebiger Stelle auf und findet dort den gesuchten Begriff nicht, dient der Buchstabenbereich der aufgeschlagenen Seiten zur Orientierung. Je nachdem ob der zu suchende Begriff alphabetisch hinter oder vor dem Wortbereich der aktuellen Seiten einzuordnen ist, blättert er vor oder zurück.

Für die elektronische Repräsentation eines Lexikons gelten ähnliche Regeln. Ein Lexikon L, das eine Menge von Wörtern $\{w_1 \cdots w_n\}$ enthält, muss in einer Weise sortiert sein, die es dem Computer erlaubt, ein angefragtes Wort leicht zu finden und die Textstelle, die damit abgelegt wurde, zurückzugeben. Dies sollte schnell und vor allem unabhängig vom Umfang des Lexikons geschehen, denn Lexika können in elektronischer Form je nach Anwendungsgebiet sehr groß werden, zum Beispiel „Vollformenlexika von Sprachen mit produktiver Morphologie, Wörterbücher mit enthaltenen Komposita oder mehrsprachige Wörterbücher etwa für die Korrektur von OCR-Dokumenten.“[3]

Eine geeignete Repräsentation muss zudem drei Basisoperationen anbieten:

Access: Lokalisiere den gesuchten Eintrag x in Lexikon L.

Insert: Füge den Eintrag x dem Lexikon L hinzu: $L \cup \{x\}$.

Delete: Lösche den Eintrag x aus Lexikon L: $L \backslash \{x\}$.

In Buchlexika bestehen Einträge aus Schlüsselwörtern und den auf diese Weise indizierten Textstellen. In dieser Arbeit wird die Komponente der Textstelle nicht berücksichtigt und der Begriff *Eintrag* synonym zu *Schlüsselwort des Eintrags* verwendet –

[3] Reffle (2005), S. 16.

nichtsdestotrotz werden nur Repräsentationsformen betrachtet, die durch Schlüsselwörter auf weitergehende Informationen referenzieren können.[4]

Da die Suche die häufigste Verwendung eines Lexikons darstellt, sollen den drei Operationen unterschiedliche Priorität beigemessen werden: Der *Access* wird als häufigste Operation betrachtet, *Insert* und *Delete* werden als seltener angesehen. Dies bedeutet, dass eine Implementation der Datenstruktur eine effiziente Suche fokussieren sollte und weiteren Operationen geringere Priorität, d. h. größere Berechnungskomplexität einräumen darf.

Eine oft verwendete Form der Lexikonorganisation, die alle obigen Kriterien erfüllt, ist das *Hashing*.[5] Hierbei wird eine Funktion $f(x)$ ermittelt, die für den Eintrag x die Stelle in einer Tabelle zurückgibt, in der der betreffende Eintrag zu finden ist. *Hashing* bietet im besten Fall eine sehr hohe Geschwindigkeit bezogen auf den Zugriff, das Einfügen und das Löschen.[6] Das Verfahren hat leider den Nachteil, einen exakten Schlüssel x zu erwarten, um eine Adresse zurückgeben zu können. Für das eingangs formulierte Szenario bedeutet dies, *Hashing* ist geeignet für die exakte Suche im Lexikon – für die unscharfe Suche müsste jedoch eine weitere Datenstruktur implementiert werden, was im besten Falle zu vermeiden ist.

Daher ist eine oft gewählte Form der elektronischen Repräsentation eines Lexikons der *Trie* (auch als *lexicographic* oder *prefix tree* bekannt). Die Bezeichnung *Trie* kommt von Re*tri*eval, wird jedoch wie „try“ ausgesprochen, um es vom allgemeineren Begriff „tree“ zu trennen. Es handelt sich dabei um eine gerichtete Baumstruktur mit einer Wurzel,[7] die Wörter als Zeichenketten (*Strings*) repräsentiert, deren einzelne Symbole den Kantenbezeichnungen der Äste im Baum entsprechen (s. Abb. 2.1).

Zur besseren formalen Beschreibung kann ein Trie als deterministischer, azyklischer, endlicher Automat in Form eines Quintupel $T = \langle \Phi, \Sigma, \delta, S, F \rangle$ angegeben werden:[8]

[4] In der eingangs vorgestellten Anwendung der Rechtschreibkorrektur ist dies nicht zwingend notwendig. Hier werden seit einigen Jahren Lexika eingesetzt, die effektiv morphologische Prozesse durch *Finite State Transducern* modellieren können und deren Ausgaben nicht wort- sondern zeichenbasiert sind.

[5] Vgl. Knuth (1998), S. 513ff.

[6] Im Fall einer kollisionsfreien Zuordnung $O(1)$.

[7] Crochemore und Rytter (2002), S. 45f.

[8] Klabunde (2001), S. 66.

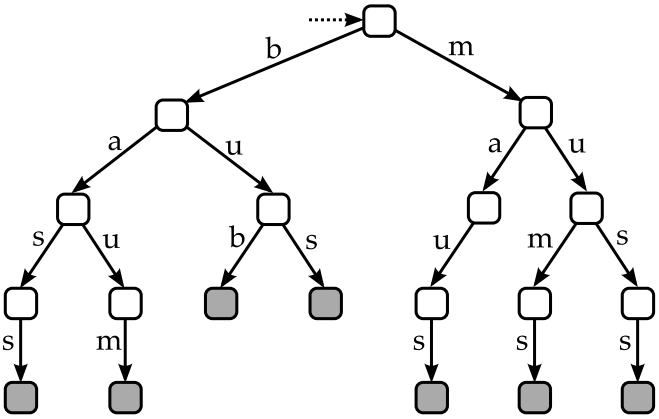


Abbildung 2.1: Trie als abstrakte Datenstruktur mit $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{m}, \texttt{s}, \texttt{u}\}$ und $L(T) = \{\texttt{bass}, \texttt{baum}, \texttt{bub}, \texttt{bus}, \texttt{maus}, \texttt{mums}, \texttt{muss}\}$.

Φ	Die Menge aller Zustände/Knoten im Trie
Σ	Das Alphabet, einschließlich des leeren Zeichens ε
δ	Die Übergangsrelation, $\delta : \Phi \times \Sigma \to \Phi$
S	Der Startzustand/Wurzelknoten des Tries, $S \in \Phi$
F	Die Menge aller finalen Zustände/Knoten, $F \subseteq \Phi$

Σ^* ist die Menge aller Wörter über Σ. Die Übergangsfunktion δ gibt den erreichten Zustand nach dem Lesen eines Zeichens zurück und die erweiterte Übergangsfunktion $\delta^* : \Phi \times \Sigma^* \to \Phi$ den erreichten Zustand nach dem Lesen einer Zeichenkette über Σ. Die Sprache, die der Automat T akzeptiert, ist $L(T) = \{x \in \Sigma^* | \delta^*(S, x) \in F\}$. Ein Eintrag x im Trie entspricht somit der Konkatenation aller Labels eines Pfades vom Startzustand S zu einem finalen Knoten $\in F$. Die Länge des Eintrags x wird mit $|x|$ angegeben, x_i bezeichnet das i-te Zeichen des Strings x, x ist somit die Konkatenation aller Zeichen $x_1 \cdots x_{|x|}$. Eine Teilzeichenkette von x des i-ten bis j-ten Zeichens wird mit $x_{i \ldots j}$ angegeben. Die Ebene (oder das Level) l des Tries bezeichnet den Abstand in Kantentraversierungen eines Knotens zum Wurzelknoten. Der Wurzelknoten ist dabei auf der niedrigsten Ebene $l = 0$, alle vom Wurzelknoten ausgehenden Kanten führen zu Knoten der Ebene $l = 1$ und so weiter. Die Höhe h eines Tries gibt die höchste Ebene des Baumes wieder, entspricht also der maximalen Distanz eines Knotens zur Wurzel und dementsprechend der Länge des längsten gespeicherten Wortes.

Das Alphabet Σ lässt sich in einer Form ordnen, so dass gilt $x, y \in \Sigma$: $x \neq y \to x < y$ oder $x > y$. Als Ordnungsprinzip wird in dieser Arbeit der Unicode Skalar Wert ge-

mäß ISO/IEC 10646 angenommen.[9] Die finalen Zustände werden als Blätter betrachtet, d. h. es gibt keine Übergangsfunktion $\delta(f, w)$ mit $f \in F$ und $w \neq \varepsilon$.[10] Zudem besitzt jeder Lexikon-Eintrag genau einen finalen Zustand, den er sich mit keinem anderen Eintrag teilt. Im Vergleich zu einem minimalen endlichen Automaten, bei dem nicht nur gemeinsame Präfixe sondern auch Suffixe zusammengefasst werden, bedeutet dies, dass sich in einem Trie auch weitere Informationen zu einem Wort abspeichern lassen, indem diese in den terminalen Knoten abgelegt werden. Damit ist $|F| = |L(T)|$.

Die Herausforderung eine geeignete Implementation für einen Trie zu finden, liegt darin, eine möglichst geschickte Codierung von $\delta(\phi, *), \phi \in \Phi$ zu wählen.[11] Nach Clément et al. (1998) existieren hierfür grundsätzlich drei Varianten:

Array-Trie: Ein Array von Zeigern wird für die Adressierung der Subtries verwendet.

List-Trie: Eine verkettete Liste wird für die Adressierung der Subtries verwendet.

BST-Trie: Ein binärer Suchbaum wird für die Adressierung der Subtries verwendet.

In den folgenden Abschnitten werden diese drei Varianten der Implementation von δ diskutiert.

2.1 Ansatz nach Fredkin

Fredkin ist Namensgeber der Trie-Struktur. Anstelle einer abstrakten Datenstruktur, wie wir ihn hier verwenden, bezeichnet der Begriff in Fredkin (1960) eine Implementation des Tries mittels *Registern* bzw. vollständigen Σ-Listen zur Organisation der Kanten – die Liste jedes Knotens ist ein $|\Sigma|$-dimensionaler Vektor der Referenzen zu den jeweiligen Kindknoten oder einen leeren Zeiger θ enthält.[12] Da das Alphabet

[9] Entsprechend werden zwei Zeichen x und y als identisch betrachtet, wenn ihr Unicode Skalar Wert identisch ist. Unicode gestattet zwar keine Mehrfachbelegung eines Zeichens in der Codetabelle, es erlaubt jedoch die Konstruktion identischer Grapheme auf unterschiedliche Art. So lässt sich das Symbol *é* (*e* mit einem Akut Akzent) in der Codierung UTF-8 als `U+00E9` speichern, eine identische Ausgabe ergibt sich durch die Sequenz `U+0065 U+0301` (ein *e* gefolgt von einem kombinierenden Akut Akzent; vgl. Yergeau, 2003, S. 10). Diese Sequenz würde als Folge zweier unabhängiger Zeichen gesehen.

[10] Dies wird erreicht durch Wortendmarken.

[11] Navarro und Raffinot (2002), S. 45.

[12] Um Speicherplatz zu sparen werden bei Fredkin nur Zeichen auf diese Weise organisiert, die Teil eines von mehreren Einträgen genutzten Präfixes sind – nach Sussenguth, Jr. (1963) eine der zentralen

sortiert ist und somit jedem Zeichen eine eindeutige Position im Vektor zugeordnet werden kann, wird auf eine Speicherung des Zeichenwertes im Lexikon verzichtet.

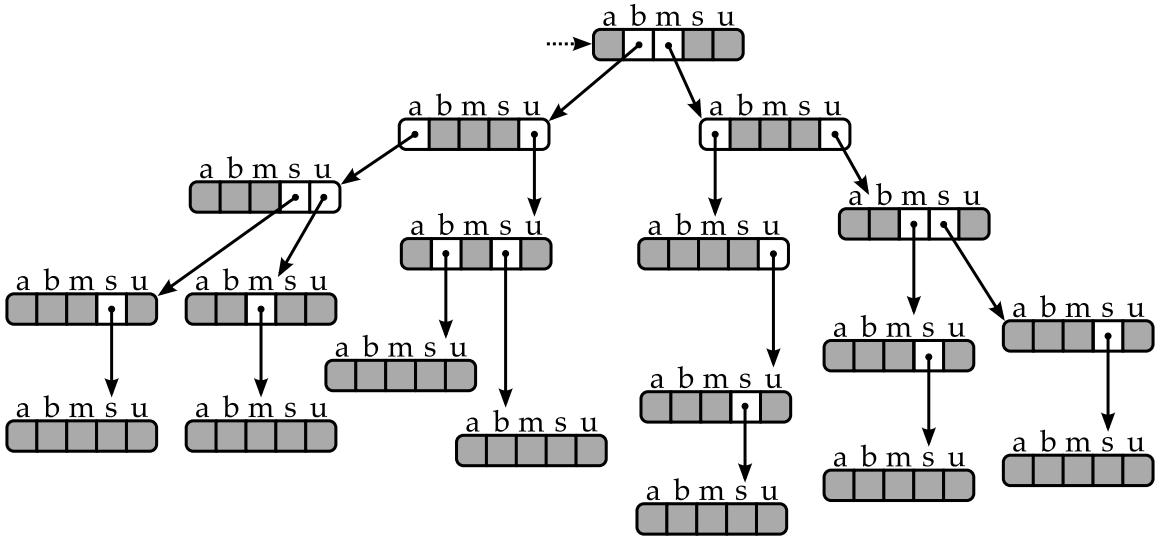


Abbildung 2.2: Fredkin-Trie mit $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{m}, \texttt{s}, \texttt{u}\}$ für $L(T) = \{\texttt{bass}, \texttt{baum}, \texttt{bub}, \texttt{bus}, \texttt{maus}, \texttt{mums}, \texttt{muss}\}$

Ein Beispiel. Ein Lexikon über $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{m}, \texttt{s}, \texttt{u}\}$ mit der Sortierung $(\texttt{a}, \texttt{b}, \texttt{m}, \texttt{s}, \texttt{u})$ repräsentiert die Sprache $L = \{\texttt{bass}, \texttt{baum}, \texttt{bub}, \texttt{bus}, \texttt{maus}, \texttt{mums}, \texttt{muss}\}$ (s. Abb. 2.2). Das Wort $x = \texttt{mums}$ wird angefragt. Zunächst wird für das erste Zeichen $x_1 = \texttt{m}$ entsprechend der Alphabetssortierung die dritte Zelle des Registers des Wurzelknotens S angefragt, $\delta(S, \texttt{m})$ gibt daraufhin den Kindknoten für das Zeichen `m` zurück, an den das zweite Zeichen `u` angefragt wird. Dies setzt sich fort, bis $x_{|x|}$ erreicht und ein Wortendknoten gefunden wurde[13] oder, wäre `mums` nicht im Lexikon, eine Traversierung endet, d.h. ein Eintrag mit leerem Folgezeiger θ wurde erreicht, ohne dass das gesuchte Zeichen gefunden werden konnte.

Auf diese Weise ist die *Access*-Operation mit $O(|x|)$ Zeit sehr schnell. Ähnlich verhält es sich mit *Insert*- und *Delete*-Operationen. Auf der anderen Seite führt der Ansatz, für jeden Knoten ein Register mit $|\Sigma|$ Zellen zu reservieren, zu einem immensen Speicheraufwand, sofern die Kardinalität von Σ nicht sehr klein ist. Dies kann bei der Speicherung von DNA-Sequenzen der Fall sein (mit $|\Sigma| = 4$ für die Anzahl der DNA-Basen `A`, `G`, `C` und `T`) oder der Speicherung binärer Daten ($|\Sigma| = 2$).[14] Im Fall von sehr

Eigenschaften des Fredkin-Trie. Diese Möglichkeit gilt für alle folgenden Trie-Implementationen, wird jedoch unterschlagen, um die Algorithmen in Kapitel 3 und 5 einfach zu halten.

[13] In Abbildung 2.2 entsprechend ein Knoten mit ausschließlich grauen Feldern.

[14] Fredkin schlägt daher auch explizit die Kodierung jedes größeren Alphabets Σ in einen binären

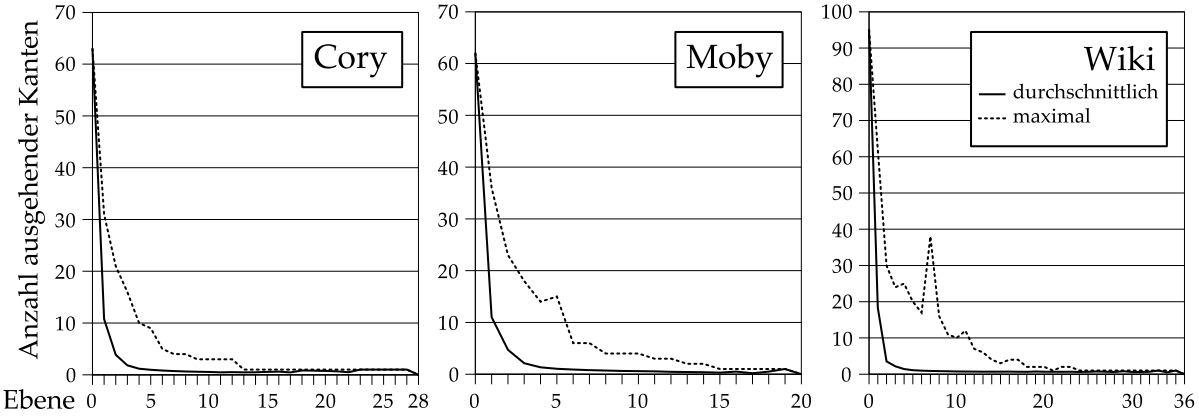


Abbildung 2.3: Maximale und durchschnittliche Anzahl ausgehender Kanten pro Ebene in den drei Korpora

großen $|\Sigma|$-Werten jedoch, wie sie im Szenario dieser Arbeit auftreten können,[15] ist dieser Ansatz höchst ineffizient, insbesondere bei geringer Auslastung der Register.

Abbildung 2.3 zeigt die Anzahl ausgehender Kanten pro Ebene in den Tries der verwendeten Korpora. Wie zu sehen, nimmt die durchschnittliche Anzahl nach der Wurzelebene rapide ab, was zu einer geringen Registerauslastung schon bei sehr niedriger Ebene führen würde.

Erwähnenswert sind jedoch Ansätze, die die variable Auslastung von Registern in hybriden Datenstrukturen nutzen und nur in Fällen von guter Auslastung Knoten durch Σ-Listen repräsentieren. Heinz et al. (2002) schlagen hierfür *Burst Tries* vor, eine Kombination aus Fredkins Ansatz und weiteren Datenstrukturen. Hierbei werden Einträge zunächst in *Containern* abgespeichert, die eine, je nach Anwendung, speichereffiziente Datenstruktur für eine geringe Menge an Einträgen beherbergen, wie einen BST oder eine sequentiell zu durchsuchende Liste. Hat ein Container so viele Einträge aufgenommen, dass die Suche ineffizient würde, wird der Container g*eburst*et, das heißt gegen einen Fredkin-Trie-Knoten ausgetauscht, dessen Zeiger auf Kindcontainer verweisen, die Einträge des ursprünglichen Containers enthalten, abzüglich des Zeichens für den Trie-Knoten. *Burst Tries* haben den Vorteil, die Geschwindigkeit eines Fredkin-Tries zu nutzen, wenn der Speicheraufwand in einer anderen Struktur nicht weit besser wäre. Sie benötigen eine Reihe von Parametern, die den Typ des Containers je nach Eintragsart bestimmen und den Grad der Ineffizienz,

Trie vor. In manchen Büchern wird daher unter einem Trie explizit ein binärer Trie verstanden (z. B. Sedgewick, 1998, S. 629).

[15] Zum Beispiel bei der Unterstützung von Unicode. Die aktuelle Unicode-Spezifikation 5.1 enthält ein Alphabet $|\Sigma| > 100000$.

ab dem ge*burst*et werden soll.

Doch es gibt auch homogene Ansätze, δ so zu implementieren, dass es selbst bei variabler Auslastung nicht zu unnötigem Speicherverbrauch kommt.

Intuitiv könnte hierfür wieder das *Hashing* in Erwägung gezogen werden – denn im Fall der Kantentraversierung ist die Unterstützung inexakter Vergleiche unnötig. Doch der Speicherbedarf steht unter Umständen dem des Fredkin-Tries in nichts nach – bei geringerer Geschwindigkeit.

Hashing ist von der Qualität der Funktion f und der Größe der von ihr verwalteten Tabelle abhängig. Eine Funktion $f(x)$ ist nicht zwingend injektiv, das heißt, es können Funktionen existieren, die für zwei unterschiedliche Schlüssel x und y eine identische Position $f(x) = f(y)$ in der Tabelle ermitteln – in diesen Fällen kommt es zu *Kollisionen* von Einträgen, die aufgelöst werden, indem durch verschiedene Mechanismen die hinterlegten Einträge iteriert werden um den richtigen Eintrag zu finden. Ist eine Hashfunktion ungünstig in Bezug auf eine kollisionsarme Verteilung der Daten gewählt oder die verwaltete Tabelle zu klein, kann es zu einer *Entartung* der Hashfunktion kommen, bei der die Geschwindigkeitsvorteile schnell abnehmen da aufgrund der Kollisionsauflösung zu oft linear gesucht werden muss. Um diese Art der *Entartung* aufzufangen muss bei einer ungünstigen Befüllung der Tabelle ein *Rehashing* auf eine größere Tabelle mit angepasster Funktion f durchgeführt werden.[16] Dieses *Rehashing* kann sehr zeitaufwändig sein.

Wird eine Hashtabelle für einen Knoten zu klein gewählt und kommt es in Folge zu häufigem *Rehashing*,[17] führt dies zu einer deutlichen Geschwindigkeitsabnahme, denn ein *Rehashing* könnte für jeden Knoten entlang des Pfades eines neu eingefügten Wortes erforderlich sein. Wird sie hingegen zu groß gewählt, ist der Speicherplatzvorteil gegenüber dem Fredkin-Trie schnell verloren und etwaige Kollisionen würden demgegenüber zu einem Geschwindigkeitsverlust führen. Dadurch ist eine Implementation von δ als Hashfunktion nur selten zu empfehlen.[18]

[16] Auf komplexere Verfahren wie das *dynamische Hashing* (Saake und Sattler, 2006, S. 410ff), welches das stetige Anwachsen der Tabelle ohne ein beim *Rehashing* notwendiges Neusortieren erlaubt, wird nicht eingegangen.

[17] Die Maximalkurve des *Wiki*-Korpus in Abb. 2.3 zeigt, dass auch in höheren Ebenen des Tries Knoten vorkommen können, die viele ausgehende Kanten besitzen – so lässt sich nicht allein anhand der Ebene des Knotens entscheiden, wie groß die erforderliche Tabelle sein sollte.

[18] Bachelard et al. (2006) vergleichen Java-Implementationen von Tries mit Hashes und Arrays und kommen zu dem selben Ergebnis – ihre Resultate sind jedoch mit Vorsicht zu genießen, da sie ausschließlich die Standard-Größen der Java-`HashMap` nutzen, die erwartbar schlechte Ergebnisse liefert (s. Ullenboom, 2006, Kap. 11.6.7).

2.2 Ansatz nach de la Briandais

Fredkin bemerkt, dass sein Ansatz in erster Linie auf Geschwindigkeitseffizienz abzielt und bezeichnet de la Briandais' Arbeit, die laut Fredkin unabhängig entstand, als Beispiel für eine speichereffiziente Implementation mit geringerer Geschwindigkeit. De la Briandais (1959) implementiert die Kanten des Tries in Form einer verketteten Liste (Abb. 2.4) anstelle eines vollständigen Σ-Registers. Dadurch ist der Speicherbedarf auch bei großer Kardinalität von Σ noch praktikabel und der Trie sehr viel universeller einsetzbar, obwohl zum Speichern eines Zeichens statt einer Zelle für einen Zeiger drei Zellen benötigt werden: Für den Zeichenwert und für die Zeiger zum nächsten Listeneintrag sowie zum Kindknoten.

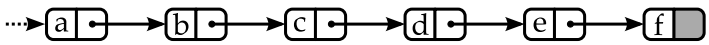


Abbildung 2.4: Verkettete Liste

Allerdings führt diese Implementation zur Notwendigkeit der sequentiellen Suche in Listen bei jedem Knotenzugriff, was bei einer ungünstigen Platzierung des zu suchenden Zeichens am Ende einer langen Liste einen erheblichen Zeitaufwand bedeuten kann. Durchschnittlich werden zum Auffinden eines Eintrags $\frac{n+1}{2}$ Schlüsselvergleiche in einer Liste der Länge n benötigt.

Ein Beispiel. Ein Lexikon über $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{m}, \texttt{s}, \texttt{u}\}$ repräsentiert die Sprache $L = \{\texttt{bass}, \texttt{baum}, \texttt{bub}, \texttt{bus}, \texttt{maus}, \texttt{mums}, \texttt{muss}\}$. Bei der Generierung des Lexikons wurden die Einträge in der Reihenfolge $(\texttt{bass}, \texttt{baum}, \texttt{bub}, \texttt{bus}, \texttt{maus}, \texttt{mums}, \texttt{muss})$ abgelegt und entsprechend auf allen Ebenen an die verketteten Listen angehängt (s. Abb. 2.5). Das

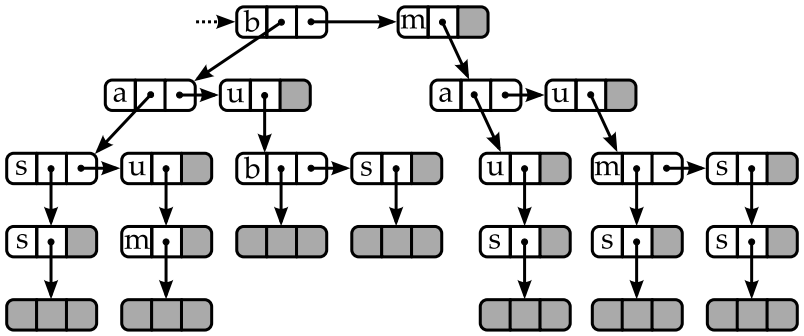


Abbildung 2.5: Briandais-Trie mit $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{m}, \texttt{s}, \texttt{u}\}$ für $L(T) = \{\texttt{bass}, \texttt{baum}, \texttt{bub}, \texttt{bus}, \texttt{maus}, \texttt{mums}, \texttt{muss}\}$

Wort $x = mums$ wird angefragt. Zunächst wird der erste Knoten der Liste des Wurzelknotens mit dem ersten Zeichen $x_1 = m$ verglichen. Da $m \neq b$ wird über den Zeiger in den nächsten Knoten der Liste gewechselt und mit dem dort abgelegten Zeichen verglichen. Da $m = m$, wird über den Zeiger auf den Listenkopf des Kindknotens übergegangen und mit diesem verglichen. Wieder stimmen die Zeichen nicht überein, $u \neq a$, und der nächste Listeneintrag wird überprüft. Dies setzt sich fort, bis $x_{|x|}$ erreicht und ein Wortendknoten gefunden wurde[19] oder, wäre die Suche erfolglos, eine Listentraversierung endet, d.h. ein Eintrag mit leerem Folgezeiger θ wird erreicht ohne dass das gesuchte Zeichen gefunden werden konnte.

Das Beispiel zeigt eine negative Eigenschaft des Listen-Tries: Die Geschwindigkeit, mit der ein Wort gefunden wird, ist abhängig von der Position des Wortes im Trie. Dies war bei Fredkin unerheblich. Kommt ein Buchstabe in einer Liste nicht vor, und kann dementsprechend das Wort nicht gefunden werden, sind $n + 1$ Schritte in einer Liste mit n Einträgen erforderlich um dies festzustellen – dies stellt den *worst case* dar.

Ein effizienteres Verfahren als die sequentielle Suche ist die *binäre Suche*, die im wesentlichen der menschlichen Suche in einem alphabetisch sortierten Lexikon entspricht, wie sie zu Beginn des Kapitels skizziert wurde. Sie lässt sich in so genannten *binären Suchbäumen* durchführen.

2.3 Ansatz nach Clampett

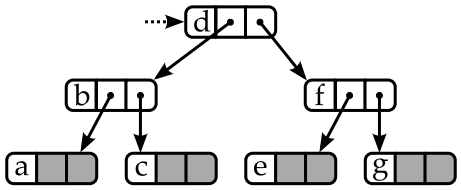


Abbildung 2.6: Binärer Suchbaum

Werden die Kanten des Tries in Form eines solchen binären Suchbaums implementiert, entsteht ein ternärer Suchbaum (*Ternary Search Tree* oder auch *Lexicographic Search Tree*).[20] Die Knoten eines binären Suchbaums enthalten neben einem Eintrag zwei Zeiger auf Kindknoten: Einen linken, der auf die Wurzel eines Teilbaums zeigt, der ausschließlich Einträge enthält, die nach einem Ordnungsprinzip kleiner sind als

[19] In Abbildung 2.5 entsprechend ein Knoten mit 3 grauen Feldern.
[20] Bentley und Saxe (1979).

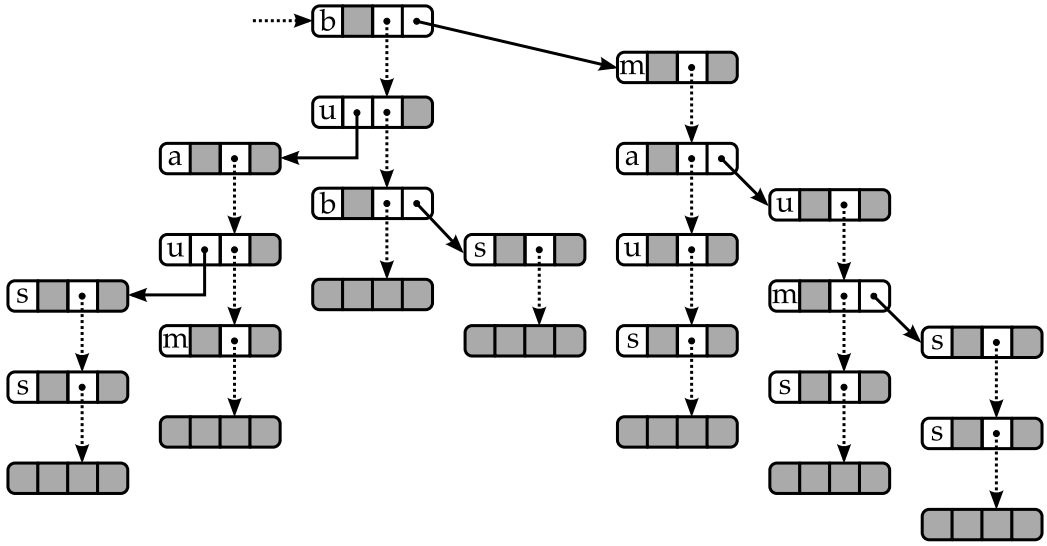


Abbildung 2.7: Ternary Search Tree mit $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{m}, \texttt{s}, \texttt{u}\}$ für $L(T) = \{\texttt{bass}, \texttt{baum}, \texttt{bub}, \texttt{bus}, \texttt{maus}, \texttt{mums}, \texttt{muss}\}$

der eigene Eintrag und einen rechten Zeiger auf die Wurzel eines Teilbaums mit größeren Einträgen (s. Abb. 2.6). Der Suchaufwand in einem binären Suchbaum hängt von der Organisation des Baumes ab. Im *worst case* benötigt eine Suche nach einem Eintrag $O(h)$ Schritte (h ist die Höhe, also die maximale Ebene des Baumes). Ist der Baum nur einseitig verzweigend entspricht die Höhe des Baumes der Anzahl seiner Knoten n – in diesem Fall verhält sich der binäre Suchbaum wie eine einfach verkettete Liste. In einem *balancierten* binären Suchbaum unterscheidet sich die Höhe des linken Teilbaumes jedes Knotens nicht mehr als $\pm$ 1 von der Höhe seines rechten Teilbaums.[21] Ein balancierter binärer Suchbaum benötigt niemals mehr als $\log n$ Schritte zum Auffinden eines Eintrags.[22]

Bereits Clampett (1964) schlug binäre Bäume zur Kantenrepräsentation eines Tries vor, wenn dieser alphabetische oder alphanumerische Schlüssel verwalten soll. Laut Bentley und Sedgewick (1998) wurde diese Struktur jedoch lange Zeit als rein theoretisches Konstrukt ohne praktischen Nutzen abgetan. Sie stellten die Struktur in einer einfachen aber effizienten Implementierung in Dr. Dobb's Journal als Alternative zum *Hashing* vor und machten ternäre Suchbäume auf diese Weise auch in der Praxis populär.[23] Ein Knoten in einem TST speichert neben genau einem Zeichen drei Zeiger:

[21] Knuth (1998), S. 459.
[22] Adel'son-Vel'skiĭ und Landis (1962).
[23] Die in dieser Arbeit verwendete Implementation entspricht im Wesentlichen der Implementation von Bentley und Sedgewick (1997, 1998).

Einen linken auf die Wurzel eines Teilbaums mit Zeichen, die kleiner als dem des Knotens sind, einen rechten auf die Wurzel eines Teilbaums mit Zeichen, die größer dem des Knotens sind, und einen mittleren, der auf die Wurzel des Teilbaums zeigt, der alle Suffixe des aktuellen Knotens enthält (also auf die nächste Ebene des Tries verweist).

Ein Beispiel. Ein Lexikon über $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{m}, \texttt{s}, \texttt{u}\}$ repräsentiert die Sprache $L = \{\texttt{bass}, \texttt{baum}, \texttt{bub}, \texttt{bus}, \texttt{maus}, \texttt{mums}, \texttt{muss}\}$. Bei der Generierung des Lexikons wurden die Einträge in der Reihenfolge $(\texttt{bub}, \texttt{bus}, \texttt{baum}, \texttt{bass}, \texttt{maus}, \texttt{mums}, \texttt{muss})$ abgelegt (s. Abb. 2.7). Das Wort $x = \texttt{mums}$ wird angefragt. Zunächst wird das Zeichen des Wurzelknotens des Baumes mit dem ersten Zeichen $x_1 = \texttt{m}$ verglichen. Da $\texttt{b} < \texttt{m}$ wird dem rechten Zeiger des Knotens gefolgt und wieder das erste Zeichen mit dem Wert des Zielknotens verglichen. Da $\texttt{m} = \texttt{m}$ wird dem mittleren Zeiger gefolgt und das Zeichen des Zielknotens mit dem zweiten Zeichen des Strings verglichen. Da $x_2 = \texttt{u} > \texttt{a}$ wird wieder dem rechten Zeiger gefolgt und so weiter. Ist ein gefundenes Zeichen größer als das gesuchte Zeichen im String, wird dem linken Zeiger gefolgt. Dies setzt sich fort, bis $x_{|x|}$ erreicht und ein terminaler Knoten gefunden wurde. Erfolglos ist eine Suche, wenn ein Knoten erreicht wird, dessen Zeichen größer dem gesuchten Zeichen x_i ist, der linke Zeiger jedoch auf θ verweist, oder ein Knoten erreicht wird, dessen Zeichen kleiner x_i ist und dessen rechter Zeiger auf θ verweist.

Ternäre Suchbäume sind auf diese Weise sehr schnell und gleichzeitig platzsparend. „Ternary search trees combine the best of two worlds: the low space overhead of binary search trees and the character-based time efficiency of digital search tries.“[24] Doch wiederum ist, wie bei de la Briandais, die Sortierung im Trie entscheidend für die Suchgeschwindigkeit. Anders als verkettete Listen können die binären Suchbäume allerdings bezogen auf den *worst case* durch Balancierung optimiert werden.[25]

[24] Bentley und Sedgewick (1998).

[25] Da für die Auswertung der Algorithmen in Kapitel 3 und 5 auf eine Online-Balancierung (d.h. eine Rebalancierung bei jedem *Insert* und *Delete*) verzichtet werden kann, ist es möglich, auf eine einfache Methode zurückzugreifen:

> You can build a completely balanced tree by inserting the median element of the input set, then recursively inserting all lesser elements and greater elements. A simpler approach first sorts the input set. The recursive *build* function inserts the middle string of its subarray, then recursively builds the left and right subarrays. (...) it is fast and produces fairly well-balanced trees.
>
> *- Bentley und Sedgewick (1998), S. 22*

In den meisten Fällen entstehen jedoch auch bei zufälliger Sortierung der Eingabedaten ausreichend gut balancierte Bäume (vgl. Knuth, 1998, S. 458). In der Evaluation werden beide Varianten verwendet.

Clément et al. haben experimentell die hier vorgestellten Trie-Varianten untersucht. Ihr Fazit lautet:

> Ternary search tries are an efficient data structure from the information theoretic point of view since a search costs typically $\log n$ comparisons on real-life textual data. List-tries require about 3 times as many comparisons as ternary search tries that implement bst-tries. For an alphabet of cardinality 26, the storage cost of ternary search tries is about 9 times smaller than standard array-tries.
>
> *- Clément et al. (1998), S. 539*

Wie gezeigt, hat die Ordnung, in welcher die Zeichen in Listen- und BST-Tries abgelegt werden, Auswirkungen auf die Geschwindigkeit der Suche nach Einträgen. Im folgenden Kapitel werden Mechanismen vorgestellt, die Sortierung der Einträge im Lexikon dem Szenario der Anfragesequenz anzupassen um so nicht die Anzahl der Zeichenvergleiche im *worst case* zu senken sondern die durchschnittliche Anzahl von Zeichenvergleichen pro Anfrage.

3 Selbstorganisierende Trie-Strukturen

„Die Suche ist der zeitaufwändigste Teil vieler Programme, und die Ersetzung einer schlechten Suchmethode gegen eine gute führt oftmals zu einer substantiellen Verbesserung der Geschwindigkeit“.[26] Effiziente Suchalgorithmen und die für die Suche günstige Organisation von Datenstrukturen gehören daher seit vielen Jahrzehnten zu den wichtigsten Bereichen der Informatik. Dabei muss, wie im vorangegangenen Kapitel gezeigt, oft zwischen Speicherbedarf und Geschwindigkeit der Suche abgewogen werden. Die Trie-Implementation nach Fredkin ist unter dem Aspekt der Suchgeschwindigkeit den Implementationsvorschlägen von de la Briandais und Clampett zwar überlegen, doch ist sie nur bedingt praxistauglich für die Speicherung einer beliebigen Anzahl von Strings über ein beliebig großes Alphabet Σ. Die Implementation eines BST-Tries wiederum ist der eines Listen-Tries überlegen – doch für beide Strukturen lässt sich diskutieren, inwieweit sie für das in der Einleitung beschriebene Szenario optimiert werden können.

Für den ternären Suchbaum ist eine Optimierung mittels Balancierung zu erreichen, durch welche die Geschwindigkeit einer Suche im *worst case* reduziert werden kann. Allerdings bedeutet die Senkung der schlimmsten anzunehmenden Zugriffszeit nicht automatisch immer auch eine Senkung der durchschnittlichen Zugriffszeit einer Sequenz von Anfragen – und diese ist in den meisten zeitkritischen Anwendungen weitaus entscheidender.

Hier kann die im Eingangskapitel genannte, nicht uniform verteilte Häufigkeit der Wörter in den Anfragesequenzen ausgenutzt werden, indem oft angefragte Einträge in der Datenstruktur besser positioniert werden als Einträge, die selten oder nie gefordert werden.

Formal hat jeder Eintrag x_i im Lexikon eine Wahrscheinlichkeit p_i, nach der x_i angefragt wird. Sind die Wahrscheinlichkeiten $p_o \cdots p_n$ *a priori* für eine Anfragesequenz Q bekannt und ist die Datenstruktur anhand dieser Verteilung sortiert, wird von einer *optimalen Sortierung* gesprochen. Im einfachen Fall einer Listensortierung wird hier-

[26] Knuth (1998), S. 393; übersetzt.

bei der Eintrag mit der höchsten Wahrscheinlichkeit am Listenanfang positioniert, alle weiteren nach absteigender Wahrscheinlichkeit.[27]

Doch selten sind Wahrscheinlichkeiten für die Anfrage von Einträgen im Vorhinein bekannt. Selbstorganisierende Datenstrukturen versuchen die durchschnittlichen Kosten einer Operation innerhalb einer Anfragesequenz zu senken, indem sie nach bestimmten Anfragen und bestimmten Regeln die Daten so restrukturieren, dass zukünftige Operationen sich aufgrund des Wissens über aktuelle Anfragen effizienter gestalten. Dabei kann sich die Datenstruktur zu Beginn einer Anfragesequenz in einem willkürlichen Zustand befinden, das heißt mit beliebiger Ordnung der Einträge. Der Zustand einer Datenstruktur zu einem Zeitpunkt t wird als *Konfiguration* C_t bezeichnet. In Bezug auf eine Sequenz von Operationen wird der Ausgangszustand vor der ersten Operation als *initiale Konfiguration* C_0 bezeichnet, der Zustand nach der letzten Operation als *finale Konfiguration* C_n.[28]

Selbstorganisierende Datenstrukturen haben verschiedene Vorteile gegenüber statischen oder regelbasierten Datenstrukturen:[29]

1. Die amortisierten Zeiten[30] der *Access*- und *Insert*-Operationen sind niemals schlechter als die entsprechenden Zeiten in regelbasiert organisierten Datenstrukturen, im Fall einer günstigen Eingabesequenz mit schiefer Verteilung können sie sogar sehr viel effizienter sein.

2. Selbstorganisierende Regeln benötigen keinen zusätzlichen Speicher, da keine Balance- oder andere Regelinformationen benötigt werden.[31]

3. Selbstorganisierende Regeln sind in den meisten Fällen leicht zu implementieren.

Auf der anderen Seite gibt es zwei mögliche Nachteile:

1. Sie erfordern einen höheren Restrukturierungsaufwand, insbesondere bei *Access*-Operationen (während Balancierungsregeln nur bei *Insert*- oder *Delete*-Operationen restrukturieren).

[27] McCabe (1965).
[28] Sherk (1995).
[29] Sleator und Tarjan (1985b), S. 653 und Albers und Westbrook (1998), S. 14. In Bezug auf statische Datenstrukturen sind zum Beispiel durch Annahmen über die Wahrscheinlichkeiten sortierte Strukturen gemeint. Unter regelbasierten Datenstrukturen (*Constrained Structures*) werden etwa balancierte Bäume verstanden – Regeln zur Selbstorganisation sind nicht gemeint.
[30] Die durchschnittliche Zeit pro Operation einer *worst case* Sequenz von Operationen wird als *amortisierte Zeit* bezeichnet. (Sleator und Tarjan, 1985a, S. 202)
[31] Dies gilt nur für „gedächtnislose“ Heuristiken.

2. Individuelle Anfragen einer Sequenz können höhere Kosten verursachen.

Die Annahme, dass eine schiefe Verteilung wie sie im vorliegenden Szenario auftritt, besonders günstig für selbstorganisierende Strukturen ist, hat in Bezug auf Tries eine doppelte Bedeutung. Nicht nur kann die Sequenz von Anfragen an den Trie einer *schiefen Verteilung* unterliegen, auch die Zeichensequenzen eines Requests sind nicht uniform verteilt:

> For example we normally expect "U" to follow "Q" and one of the vowels or the letters "H," "L," "R," or "Y" to follow the letter "C." Thus we find we are able to adjust favorably the averages we determined previously due to bunching, a phenomenon which usually leads to decreased efficiency in other methods.
>
> *- de la Briandais (1959), S. 298*

Bei einer Sortierung – und selbst bei einer Balancierung – der Einträge, könnte es in oben erwähntem Fall vorkommen, dass der Landesname „QATAR" vor dem Wort „QUICK" in den Trie einsortiert wurde und alle Anfragen, die mit einem „Q" beginnen und ein „U" an zweiter Stelle besitzen, zunächst auf den Knoten mit dem Zeichen „A" zugreifen müssen um erst in einem weiteren Schritt den Knoten mit dem gesuchten Zeichen „U" zu erreichen. Der Effekt einer schiefen Verteilung der Anfragesequenz findet entsprechend auf mehreren Ebenen statt. Ziel selbstorganisierender Regeln ist es, solche ungünstigen Zugriffe zu vermindern, indem häufig angefragte Einträge günstiger positioniert werden.

Die ältesten Regeln dieser Art entwickelte McCabe (1965) für selbstorganisierende Suchlisten. Im Folgenden werden Beispiele für solche Heuristiken zunächst anhand von verketteten Listen und binären Suchbäumen vorgestellt, die in einem nächsten Schritt auf die Implementation in List-Tries und TSTs angewendet werden.

3.1 Listen

Die geringe Effizienz einer sequentiellen Suche in einer Liste wurde bereits in den Kapiteln 2.2 und 2.3 thematisiert. Dennoch gibt es Bereiche, in denen sich der Einsatz lohnt – vorausgesetzt, die Listen können effizient sortiert werden:

> · When n is small (say, at most several dozen), the greater constant factors in the run times of other strategies may make linear search competitive. This occurs, for example, when linked lists are used to resolve collisions in a hash table.
>
> · When space is severely limited, sophisticated data structures may be too space-expensive to use.
>
> · If the performance of linear search is almost (but not quite) good enough, a self-organizing heuristic may make it effective for the application at hand without adding more than a few lines of code.
>
> *- Bentley und McGeoch (1985), S. 410*

Im Fall der Implementation nach de la Briandais ist der erste Punkt, die geringe Anzahl an Einträgen, in vielen Lexika schon ab einer niedrigen Ebene des Tries gegeben (vgl. Abb. 2.3). Der zweite Punkt trifft zumindest im Fall großer Lexika mit großer Kardinalität von Σ zu. Um die Performanz zu steigern, schlagen Bentley und McGeoch selbstorganisierende Heuristiken vor. McCabe (1965) führte die noch heute gebräuchlichsten Regeln *Move To Front* (Abb. 3.1) und *Transpose* (Abb. 3.2) ein (s. Algorithmen S. 72).

Move To Front: Nach einer *Access*- oder *Insert*-Operation wird der betreffende Eintrag E_i an den Anfang der Liste gesetzt.[32]
$E_1, E_2 \cdots E_i \cdots E_{m-1}, E_m \rightarrow E_i, E_1, E_2 \cdots E_{m-1}, E_m$

Transpose: Nach einer *Access*- oder *Insert*-Operation wird der betreffende Eintrag E_i mit dem Eintrag E_{i-1} vertauscht, es sei denn E_i steht bereits am Anfang der Liste.
$E_1, E_2 \cdots E_{i-1}, E_i \cdots E_{m-1}, E_m \rightarrow E_1, E_2 \cdots E_i, E_{i-1} \cdots E_{m-1}, E_m$

[32] In Arrays bedeutet diese Permutation einen erheblichen Aufwand, da alle verschobenen Einträge $E_{1\ldots i}$ kopiert werden müssen. Im Folgenden wird für diese Regeln von einer Implementation als doppelt verkettete Liste ausgegangen.

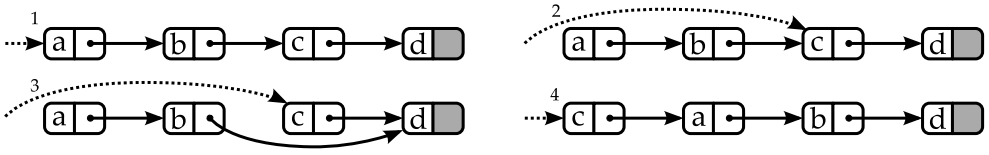


Abbildung 3.1: *Move To Front* in verketteter Liste. *c* wird an den Listenanfang bewegt.

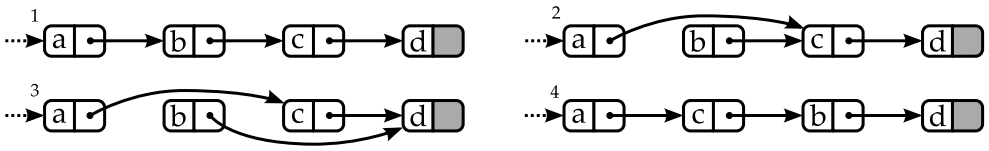


Abbildung 3.2: *Transpose* in verketteter Liste. *c* wird eine Position zum Listenanfang bewegt.

Ein Beispiel. Ein Listen-Lexikon mit den Einträgen $L = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ wird mit der Anfragesequenz $Q = (\mathtt{a}, \mathtt{a}, \mathtt{b}, \mathtt{b}, \mathtt{c})$ befragt. Nach jeder Anfrage wird die Liste der *Move To Front* Heuristik folgend restrukturiert. Für die initiale Konfiguration C_0 von L wird die alphabetische Sortierung angenommen. Angegeben wird die aktuelle Konfiguration der Liste, der anzufragende Eintrag und die benötigten Knotenzugriffe um den gesuchten Eintrag zu finden:

$$\begin{array}{l|l} C_0 = (\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}) & q_1 = \mathtt{a} \rightarrow 1 \\ C_1 = (\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}) & q_2 = \mathtt{a} \rightarrow 1 \\ C_2 = (\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}) & q_3 = \mathtt{b} \rightarrow 2 \\ C_3 = (\mathtt{b}, \mathtt{a}, \mathtt{c}, \mathtt{d}, \mathtt{e}) & q_4 = \mathtt{b} \rightarrow 1 \\ C_4 = (\mathtt{b}, \mathtt{a}, \mathtt{c}, \mathtt{d}, \mathtt{e}) & q_5 = \mathtt{c} \rightarrow 3 \\ C_5 = (\mathtt{c}, \mathtt{b}, \mathtt{a}, \mathtt{d}, \mathtt{e}) & \end{array}$$

Für die Bearbeitung der Anfragesequenz Q sind insgesamt 8 Eintragsvergleiche erforderlich. Ohne Restrukturierung wären 9 Vergleiche benötigt worden.[33] Die *Transpose* Heuristik hätte für die selbe Anfrage bei gleicher initialer Konfiguration ebenfalls 8 Vergleiche benötigt.

Eine weitere grundlegende Heuristik zur Selbstorganisation von Listen basiert auf der Idee einer inkrementellen optimalen Ordnung der Einträge. *Frequency Count* (s. Algorithmus S. 73) ist, anders als *Move To Front* und *Transpose*, keine „gedächtnislose“[34]

[33] 2×1 für a, 2×2 für b und 1×3 für c. Dabei entspricht $C_{initial}$ sogar $C_{optimal}$ für Q. (Weitere Möglichkeiten für $C_{optimal}$ wären $(\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}, \mathtt{d})$, $(\mathtt{b}, \mathtt{a}, \mathtt{c}, \mathtt{d}, \mathtt{e})$ oder $(\mathtt{b}, \mathtt{a}, \mathtt{c}, \mathtt{e}, \mathtt{d})$.)

[34] Albers und Westbrook (1998), S. 14.

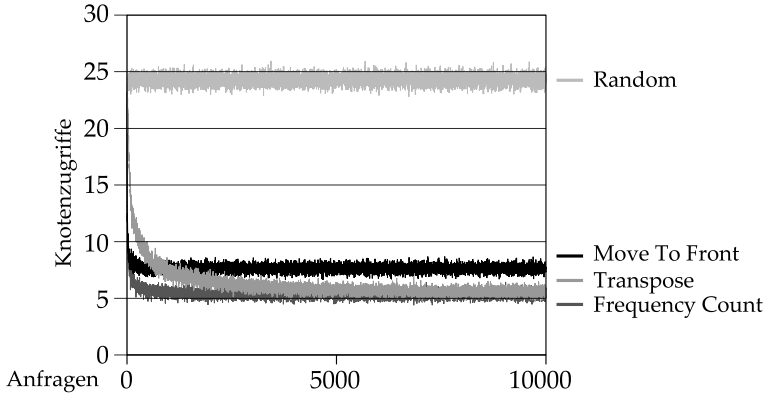


Abbildung 3.3: Experiment A – Vergleich von *Move To Front*, *Transpose* und *Frequency Count* in einer Listenumgebung. Als Vergleich ist die nicht restrukturierende Liste mit zufälliger Konfiguration angegeben. (*Wiki*)

Heuristik – sie benötigt zusätzlichen Speicherplatz pro Eintrag um die Anzahl der Anfragen an diesen zu speichern.

Frequency Count: Nach einer *Access*- oder *Insert*-Operation wird die Zählervariable eines Eintrags $E_{i,a}$ inkrementiert. Der Eintrag wird so lange mit dem vorstehenden Eintrag getauscht, bis dieser eine Zählervariable besitzt, die größer der Zählervariable von $E_{i,a+1}$ ist.

$$E_{1,c>a}, E_{2,c>a} \cdots E_{n-2,c>(a+2)}, E_{i-1,c=a}, E_{i,c=a} \cdots E_{m-1,c<a}, E_{m,c<a}$$
$$\xrightarrow{a' = a+1}$$
$$E_{1,c>a'}, E_{2,c>a'} \cdots E_{i-2,c>(a'+1)}, E_{i,c=a'}, E_{i-1,c<a'} \cdots E_{m-1,c<a'}, E_{m,c<a'}$$

Um die Vorteile verschiedener selbstorganisierender Regeln miteinander vergleichen zu können und gegenüber statischen zu verdeutlichen, sind längere Anfragesequenzen als die im Beispiel gezeigten erforderlich, deren Anfragen einer schiefen Verteilung unterliegen. Dabei ist zu erkennen, dass selbstorganisierende Strukturen mit der Zeit einen stabilen Zustand (*Steady State*) erreichen, bei dem erwartet werden kann, das zukünftige Permutationen die Suchzeit nicht mehr signifikant verbessern oder verschlechtern.[35]

[35] Hester und Hirschberg (1985), S. 298.

Experiment A

Verglichen wird die Anzahl notwendiger Knotenvergleiche in einer mit *Move To Front*, *Transpose* und *Frequency Count* restrukturierten und einer undynamischen Liste (*Random*). Die Liste enthält die ersten 50 Einträge des *Wiki*-Korpus. Angegeben sind die durch 1000 Durchläufe gemittelten Werte einer Anfragesequenz $|Q| = 10000$. Die Wahrscheinlichkeitsverteilung der Anfrage wurde künstlich generiert.[36] Die initiale Konfiguration wurde zu Beginn jeden Durchlaufs neu randomisiert.[37]

Random	$\approx 24{,}3$
Move To Front	$\approx 7{,}624$
Transpose	$\approx 5{,}593$
Frequency Count	$\approx 5{,}315$

Abbildung 3.4: Experiment A – Durchschnittliche Anzahl von Knotenzugriffen in einem angenommenen *Steady State* $Q_{7500 \cdots 10000}$

Wie zu sehen (Abb. 3.3), haben alle selbstorganisierenden Listen eine deutlich geringere Suchzeit als *Random*. Dabei erreicht *Move To Front* früher seinen *Steady State*, konvergiert also schneller an seine Asymptote, die jedoch erheblich höher liegt als bei *Transpose*. Tabelle 3.4 zeigt die durchschnittliche Anzahl von Knotenzugriffen im *Steady State*: Die mit *Transpose* restrukturierte Liste benötigt deutlich weniger Knotenzugriffe als *Move To Front* und nur unwesentlich mehr als die „merkende" Heuristik *Frequency Count*. Rivest (1976) zufolge gilt dies für alle möglichen Verteilungen in Q, weshalb er *Transpose* als die immer überlegene Regel bezeichnet.[38] Geht man jedoch nicht von den geringeren asymptotischen Kosten sondern der schnelleren Konvergenz zur Asymptote aus, ist *Move To Front* überlegen.[39] Daher schlägt Bitner (1979) eine hybride Regel vor, bei der jeder Eintrag initial m Male nach *Move To Front* restruk-

[36] Devroye (1986) führt hierzu einen einfachen, einparametrigen Zipf-Generator an (s. Algorithmus S.82; Devroye, 1986, S. 550f). Für den Parameter a wurde der Wert 1,1 gewählt. In den folgenden Experimenten entsprechen die Wahrscheinlichkeitsverteilungen den Wortfrequenzverteilungen der Texte.

[37] Die erhobenen Daten aller in dieser Arbeit aufgeführten Experimente befinden sich auf der beigefügten DVD.

[38] Rivest (1976), S. 67. Rivest gibt sogar an, *Transpose* sei die optimale Permutation aller möglichen Permutationen der Liste bei $|Q| \to \infty$. Zwar widerlegen Anderson et al. (1982) dies mit einem Gegenbeispiel durch eine von den hier vorgestellten Heuristiken stark abweichende Permutation, lassen jedoch offen, ob *Transpose* die optimale „gedächtnislose" Regel nach dem *Move ahead k*-Prinzip (s. folgender Absatz) mit konstantem k ist.

[39] Bitner (1979), S. 108. Damit ist *Move To Front* auch besonders in Umgebungen mit geringer Anzahl von Anfragen geeignet.

turiert wird bevor für alle weiteren Anfragen die *Transpose*-Regel eingesetzt wird.

Rivest (1976) stellt eine Verallgemeinerung der beiden Heuristiken mittels der *Move ahead k* Regel auf, bei der durch k die Anzahl Positionen angegeben wird, die der Eintrag E_i nach erfolgreichem Zugriff zum Listenanfang verschoben wird. *Transpose* entspricht dabei *Move ahead 1*, *Move To Front* ist *Move ahead i*. Auf diese Weise lassen sich kostengünstiges asymptotisches Verhalten und schnelle Konvergenz abwägen. Hester und Hirschberg (1985) generalisieren die Regel weiter und erlauben für k auch prozentuale Werte[40] und Funktionen mit Parametern wie der Listenlänge oder der Anzahl vergangener Zugriffe auf die gesamte Liste.

Die bisher angeführten Annahmen gehen von einer Wahrscheinlichkeitsverteilung über Q aus – werden Einträge aber in einer Teilsequenz von Q häufiger im Verhältnis zur gesamten Sequenz Q angefragt, kommt ein weiterer Aspekt hinzu, der für selbstorganisierende Datenstrukturen eine Rolle spielt und sie selbst im Vergleich zu optimalen Sortierungen überlegen machen kann: Die *Lokalität*.[41] Im angeführten Beispiel zur *Move To Front*-Permutation lassen sich die Teilsequenzen (a, a) und (b, b) als lokale Aufkommen verstehen. *Move To Front* reagiert dabei schneller auf diese lokalen Konzentrationen als *Transpose*, besitzt damit einen höheren Grad an *lokaler Adaptivität*, was in verschiedenen Anwendungen vorteilhaft genutzt werden kann.[42]

Auch *Frequency Count*, trotz bestem asymptotischen Verhalten (s. Tab. 3.4), besitzt einen geringeren Grad an lokaler Adaptivität. Sein größter Nachteil ist jedoch der zusätzlich benötigte Speicherplatz, der sehr groß sein kann, da der Zähler gegen unendlich geht. Um einen unbegrenzt wachsenden Zähler zu verhindern und dennoch eine „merkende“ Heuristik zu erhalten, schlägt Bitner (1979) anstelle einer kontextunabhängigen Inkrementierung eine „Limited Difference Rule“ vor, bei der Eintrag $E_{i,c}$ inkrementiert wird und $E_{i-1,c}$ dekrementiert, gleichzeitig ein oberer, definierter Grenzwert nicht überschritten werden darf, der wohl aber wieder dekrementiert werden kann. Hofri und Shachnai (1991) schlagen die ausschließliche Verwendung dieses Grenzwertes für ein „Limited Counter Scheme“ vor. Sobald der Grenzwert eines Eintrags erreicht ist, werden für diesen Eintrag keine Permutationen mehr durchge-

[40] Und verallgemeinern damit auch die *Move-Fraction(k)* Heuristik, nach der bei jedem Zugriff der Eintrag E_i um $\frac{i}{k}$ - 1 Positionen zum Listenanfang bewegt wird. (Sleator und Tarjan, 1985a, S. 204)

[41] *Locality*. Hester und Hirschberg (1985), S. 298.

[42] In den hier vorgestellten Experimenten spielt Lokalität keine Rolle, da die Anfragen randomisiert sind. In Texten können aber durchaus lokale Konzentrationen vorkommen, insbesondere im skizzierten Szenario, wenn die Rechtschreibprüfung mehrsprachig ist und die Sprache der analysierten Texte wechselt. Bentley und McGeoch (1985) nennen als Beispiel für diesen Effekt das Einlesen von Programmcode, in dem Variablenbezeichnungen eine lokale Konzentration aufweisen können.

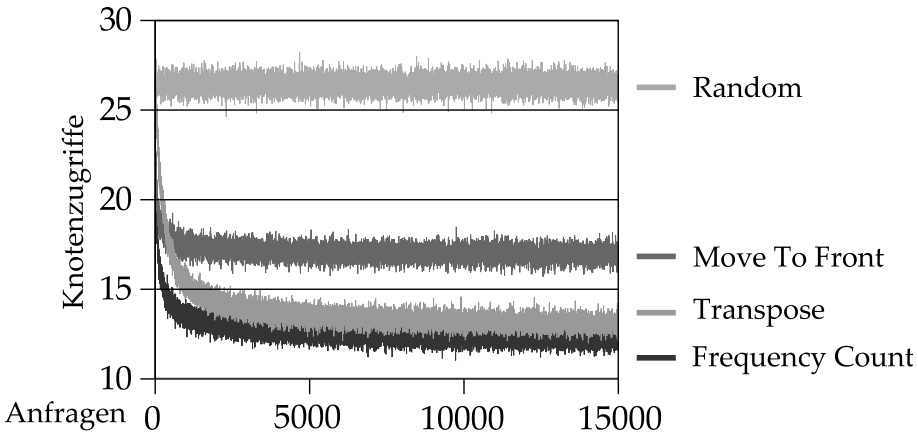


Abbildung 3.5: Experiment B – Vergleich von *Move To Front*, *Transpose* und *Frequency Count* in einem Listen-Trie. (*Cory*)

führt. Auf diese Weise wird mit steigender Anzahl an Zugriffen ein immer längerer Listenkopf *eingefroren*.[43]

Bachrach und El-Yaniv (1997) zeigen neben den hier vorgestellten drei Basisheuristiken in einer Übersicht 13 weitere[44] und vergleichen diese experimentell.[45] Auch wenn ihre Ergebnisse zeigen, dass die Basisheuristiken selten optimal für reale Daten sind, beschränken sich die folgenden Experimente auf diese drei.

3.2 Listen-Tries

Die vorgestellten Heuristiken lassen sich auf die Trie-Struktur nach de la Briandais übertragen, indem bei einem *Access* alle Listen, die angefragte Knoten enthalten, der gewählten Heuristik entsprechend permutiert werden.[46]

[43] Hofri und Shachnai (1991), S. 548.

[44] In den meisten Fällen *move ahead k* konforme Permutationen.

[45] Der Vergleich bezieht sich hierbei nicht nur auf Suchlistenzugriffe im Sinne eines Lexikons sondern auch auf ein von Bentley et al. (1986) vorgestelltes Datenkompressionsverfahren, das die *Move To Front* Heuristik nutzt und in erster Linie in Kompressionssystemen eingesetzt wird, die in der Vorverarbeitung eine Burrows-Wheeler-Transformation (Burrows und Wheeler, 1994) durchführen. (Vgl. Salomon, 2007, S. 37f)

[46] Es werden ausschließlich erfolgreiche *Access*-Operationen durchgeführt – weder *Insert* noch *Delete* führen zur Permutation. So orientiert sich die initiale Konfiguration an der Einfüge-Reihenfolge, da neue Knoten in Listen auf allen Ebenen an das Listenende gehängt werden. Im Folgenden werden für alle Ebenen identische Heuristiken eingesetzt, auch wenn eine Kombination denkbar wäre.

Experiment B

Verglichen wird die Anzahl notwendiger Knotenvergleiche in einem mit *Move To Front*, *Transpose* und *Frequency Count* restrukturierten und einem undynamischen Listen-Trie. Der Trie enthält alle Wörter des *Cory*-Korpus. Angegeben sind die durch 1000 Durchläufe gemittelten Werte einer Anfragesequenz $|Q| = 15000$. Die Wahrscheinlichkeitsverteilung entspricht der Wortfrequenzverteilung im Text. Die initiale Konfiguration wurde zu Beginn jeden Durchlaufs neu randomisiert.

In allen Trie-Experimenten werden nur die „nicht-optimalen" Knotenzugriffe gezählt, d.h. jeder Vergleich mit dem Listenkopf ist kostenfrei, wie auch die Traversierungen der mittleren Kanten im ternären Suchbaum (s. Kap. 3.4).[47]

	Insgesamt	Ohne Ebene 0
Random	$\approx$ 26,439	$\approx$ 9,593
Move To Front	$\approx$ 16,955	$\approx$ 4,304
Transpose	$\approx$ 13,144	$\approx$ 3,783
Frequency Count	$\approx$ 12,253	$\approx$ 3,255

Abbildung 3.6: Experiment B – Durchschnittliche Anzahl von Knotenzugriffen in einem angenommenen *Steady State* $Q_{10000 \cdots 15000}$.

Die selbstorganisierenden Heuristiken zeigen ein deutlich besseres asymptotisches Verhalten als der undynamische Trie (s. Tab. 3.6). *Frequency Count* ist dabei der *Transpose*- und der *Move To Front*-Heuristik überlegen. Auch ist zu sehen, dass *Move To Front* erwartbar schneller konvergiert als *Transpose* (s. Abb. 3.5).

Abbildung 3.7 zeigt das Verhalten der einzelnen Ebenen des Tries. Wie nach Abbildung 2.3 auf Seite 12 zu erwarten, verursacht der Vergleich auf der ersten Ebene die größten Kosten, da sich dort die längste Liste befindet. Würde diese gegen ein Fredkin-Register getauscht, ähnlich der genannten *Burst*-Idee, würden weniger als ein Drittel der ursprünglichen Knotenzugriffe im selbstorganisierten Fall benötigt (s. Tab. 3.6).

Es ist zu erkennen, dass ein selbstorganisierender de la Briandais-Trie effizienter auf Anfragen mit schiefer Verteilung, wie sie in einem Text vorliegt, reagieren kann als seine undynamische Variante. Und dies bei nur geringem komputationellen Aufwand: Wie an den Statistiken zum Ebenenvergleich zu sehen, sind bei einem *Access* von x selten $|x|$ Listen zu restrukturieren, da sich nur auf wenigen Ebenen der ange-

[47] Auf diese Weise ist die Länge eindeutiger Präfixe nicht von Bedeutung. Siehe Fußnote 12, S. 10.

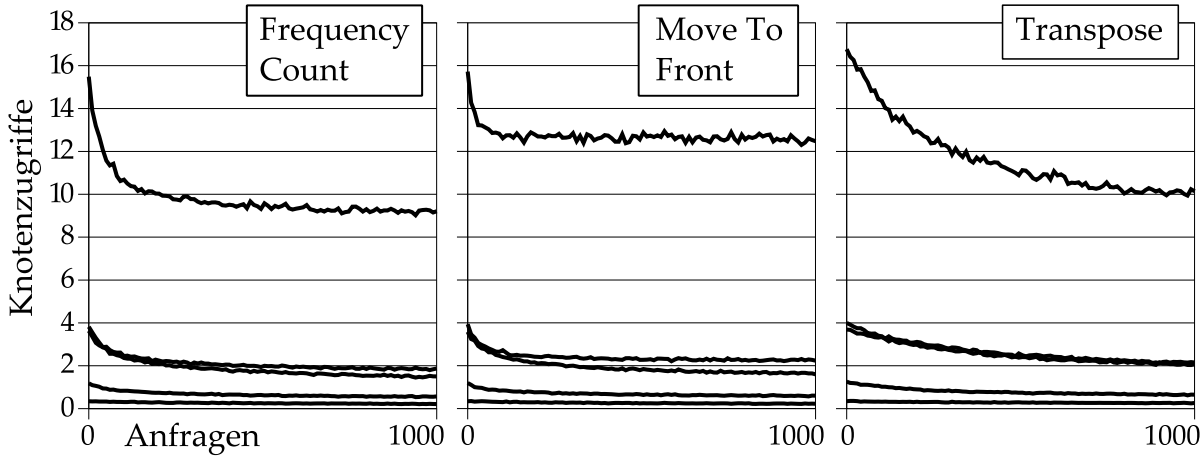


Abbildung 3.7: Experiment B – Ebenen-Vergleich ($l = 0 \cdots 4$) von *Frequency Count* (links), *Move To Front* (Mitte) und *Transpose* (rechts). (*Cory*)

fragte Eintrag hinter dem Listenanfang befindet.

Der komputationelle Aufwand selbstorganisierender verlinkter Listen ist dabei sehr viel geringer als bei selbstorganisierenden binären Suchbäumen.

3.3 Binäre Suchbäume

Die ersten Vorschläge für selbstorganisierende Heuristiken in binären Suchbäumen wurden von Allen und Munro (1978) gemacht und orientierten sich an den entsprechenden Listen-Heuristiken von McCabe: Die *Move To Root* Regel bewegt, ähnlich wie *Move To Front*, einen angefragten Eintrag an die Spitze der Datenstruktur – an die Wurzel des binären Suchbaums. *Simple Exchange* sorgt hingegen für einen einfachen Wechsel mit dem nächsten Eintrag, der näher der Strukturspitze ist – im Fall des binären Suchbaumes also dem Elternknoten – ganz ähnlich der *Transpose*-Heuristik für Listen (s. Abb. 3.8; Algorithmen Seite 73). Ziel war es, ein Restrukturierungssystem für Bäume zu finden, das in amortisierter Zeit mit der Balancierung vergleichbar ist, ohne zusätzliche Balance-Informationen in den Knoten speichern zu müssen[48] und sich bei schiefer Verteilung effektiver verhält.

Im Gegensatz zu ihren Gegenstücken für Listen lieferten *Move To Root* und *Simple*

[48] Daher werden hier ausschließlich „gedächtnislose“ mit balancierten Bäumen verglichen. Beispiele für selbstorganisierende Heuristiken mit Zählern sind *Dynamic Monotonic Trees* (Bitner, 1979, S. 96ff) und *Conditional Rotations* (Cheetham et al., 1993). Im Folgenden wird für die Bäume ein zusätzlicher Zeiger zum Elternknoten angenommen. Ist der Knoten y die Wurzel, zeigt $p(y)$ auf θ.

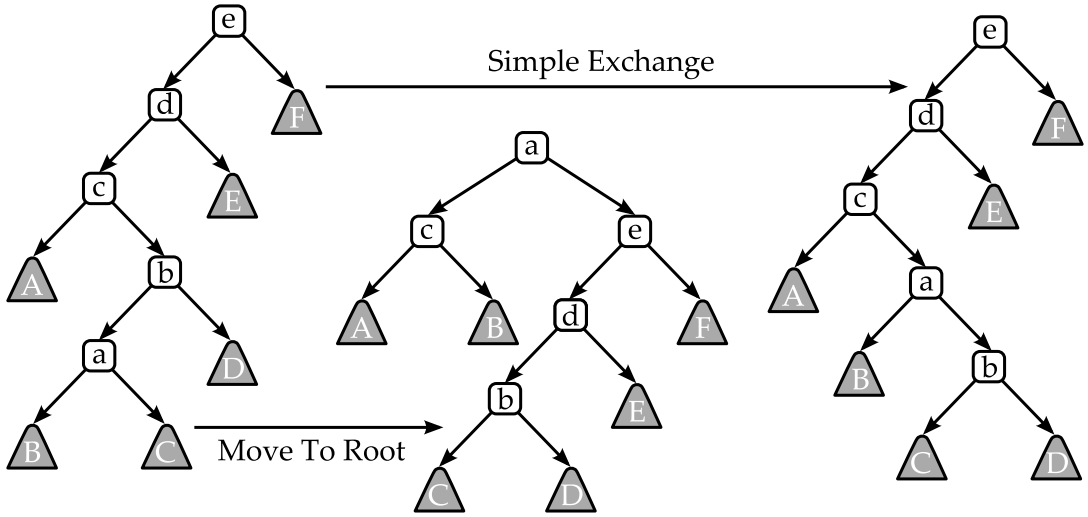


Abbildung 3.8: *Simple Exchange* und *Move To Root*. Der Zugriffsknoten ist *a*.

Exchange allerdings keine überzeugenden Ergebnisse:

> First we recommend that the simple exchange never be used. Even when the distribution of requests is known to be uneven (but, of course not known), the scheme flirts with disaster. Second, the move to root heuristic, and indeed an optimal binary search tree, should not be used if a large number of probes (...) cannot be tolerated for a single retrieval. This probably eliminates many real time applications. Third, the adaptive techniques are inappropriate if the occurence of a request makes a subsequent request unlikely in the near future.
>
> *- Allen und Munro (1978), S. 534*

Zudem kann es zu *worst case* Szenarien kommen, in denen für die Anfrage eines Knotens in einem Baum mit n Einträgen n Schritte benötigt werden[49] – und beliebig lange Sequenzen geben, die eine amortisierte Zeit von $O(n)$ haben.[50] Allerdings besitzt *Move To Root* entsprechend zu *Move To Front* einen starken Grad an lokaler Adaptivität, was diese Heuristik in Anwendungen mit starker Lokalität effektiv machen kann. Der Aufwand der *Move To Root* Heuristik ist allerdings vergleichbar mit der von *Move To*

[49] Was dem *worst case* Szenario einer Liste entspricht.

[50] Sleator und Tarjan (1985b), S. 655. Dies tritt bei einer Konfiguration $C_{initial}$ mit einseitiger Verzweigung (z. B. nicht rebalancierende alphabetische Einfügung der Einträge; s. Abb. 3.15 auf S. 36) und alternierender Anfrage an den kleinsten und größten Eintrag auf.

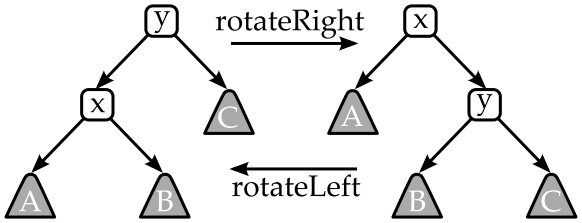


Abbildung 3.9: Rotation der Kante zwischen x und y

Front in Arrays: Statt wie in verketteten Listen wenige Zeigerwechsel durchführen zu müssen, wird jeder Knoten bis zur Wurzel restrukturiert. Die Basisoperation hierfür ist die Rotation[51] (s. Abb. 3.9), die auch Basis der meisten Balancierungsoperationen ist.

Simple Exchange: Nach einer *Access*- oder *Insert*-Operation auf den Knoten y

1. Beende die Restrukturierung, wenn y die Wurzel des Baumes ist.
2. Rotiere die Kante zwischen y und dem Elternknoten $p(y)$ nach rechts, wenn y das linke Kind ist.
3. Rotiere die Kante zwischen y und dem Elternknoten $p(y)$ nach links, wenn y das rechte Kind ist.

Move To Root: Nach einer *Access*- oder *Insert*-Operation auf den Knoten y wird folgende Restrukturierung rekursiv aufgerufen:

1. Beende die Restrukturierung, wenn y die Wurzel des Baumes ist.
2. Rotiere die Kante zwischen y und dem Elternknoten $p(y)$ nach rechts, wenn y das linke Kind ist. Führe *Move To Root* für y aus.
3. Rotiere die Kante zwischen y und dem Elternknoten $p(y)$ nach links, wenn y das rechte Kind ist. Führe *Move To Root* für y aus.

Im Gegensatz zu *Move To Root* und *Transpose* ist das *Splaying* (Sleator und Tarjan, 1983, 1985b, s. Algorithmus S. 75) ein Verfahren zur Selbststrukturierung von Bäumen, das sich in vielen Anwendungen durchgesetzt hat – da es verglichen mit *Move To Root* auch bei weniger schiefer Verteilung effizient ist und eine amortisierte Grenze von $O(\log n)$ besitzt.[52] Dabei ähnelt das *Splaying* der *Move To Root* Regel insofern,

[51] Adel'son-Vel'skiĭ und Landis (1962).

[52] Tarjan (1983), S. 53. Das bedeutet, ein Zugriff kann zwar initial n Knotenvergleiche benötigen (s. Fußnote 50), es gibt in der Folge aber keine beliebig lange Sequenz die in amortisierter Zeit $O(n)$ kostet (s. Abb. 3.15).

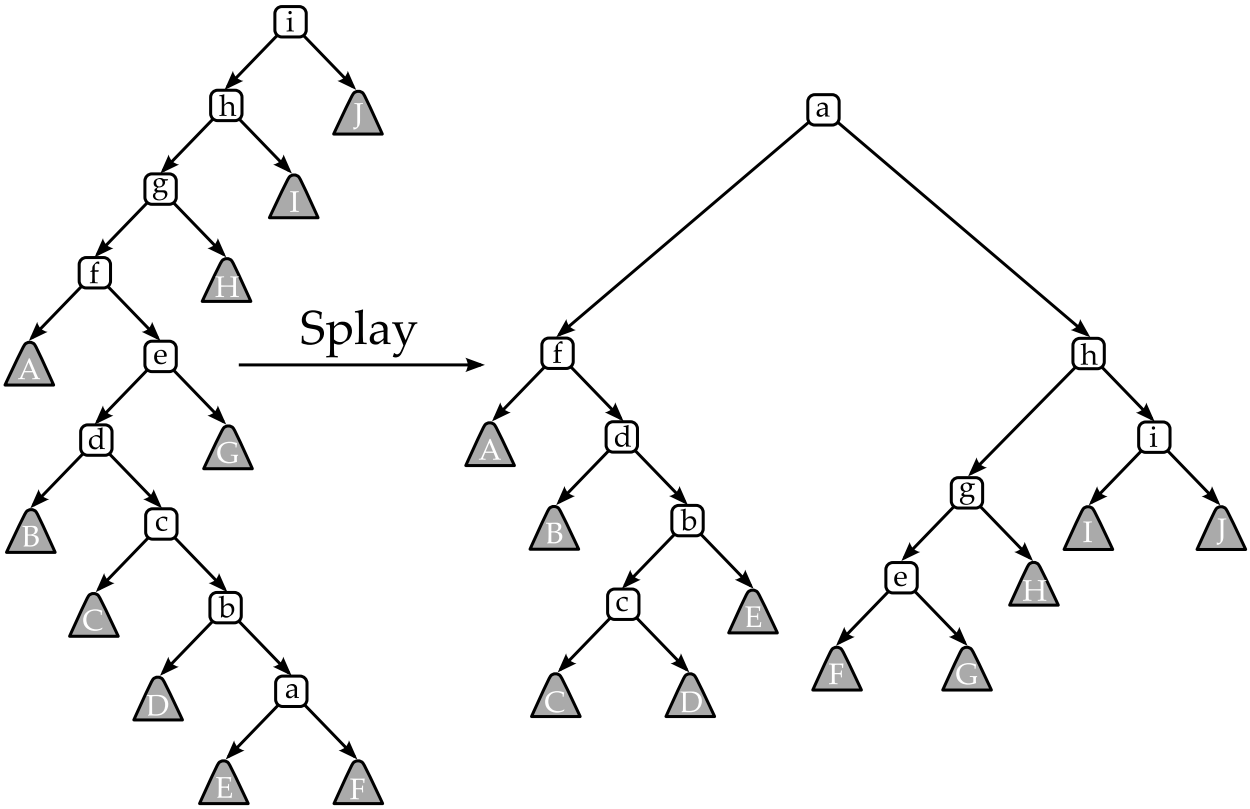


Abbildung 3.10: *Splaying* am Knoten *a*

als es den angefragten Knoten immer an die Wurzel des Baumes verschiebt. Allerdings werden hierfür nicht nur einzelne Rotationen mit den jeweiligen Elternknoten durchgeführt – *Splaying* rotiert paarweise entlang des Pfades zur Wurzel.[53]

Splay: Nach einer *Access*- oder *Insert*-Operation auf den Knoten y:[54]

Zig: Ist $p(y)$ der Wurzelknoten, rotiere die Kante zwischen y und $p(y)$. Beende das *Splaying*.

Zig-Zig: Ist $p(y)$ nicht die Wurzel und sowohl y als auch $p(y)$ entweder linker oder rechter Kindknoten ihrer Elternknoten, rotiere die Kante zwischen $p(y)$ und $p(p(y))$ und rotiere danach die Kante zwischen y und $p(y)$.

Zig-Zag: Ist $p(y)$ nicht die Wurzel und y ist linkes und $p(y)$ rechtes Kind seines Elternknotens (oder umgekehrt), rotiere die Kante zwischen y und $p(y)$

[53] Dies ist im hier vorgestellten *bottom-up* Verfahren der Fall. Sleator und Tarjan (1985b) stellen auch ein *top-down Splaying* vor (S. 667ff).
[54] Sleator und Tarjan (1985b), S. 655.

und rotiere danach die Kante zwischen y und seinem neuen Elternknoten $p(y)$.

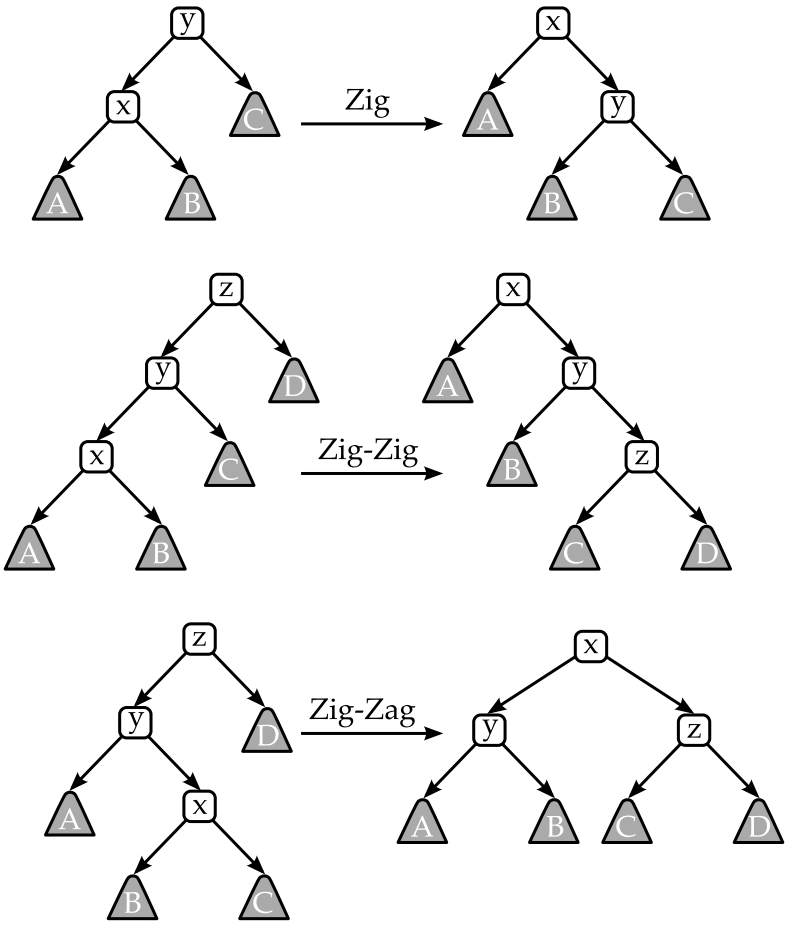


Abbildung 3.11: *Splaying*-Operationen. Der Zugriffsknoten ist x, die symmetrischen Varianten sind nicht aufgeführt.

Die Tiefe jedes Knotens entlang des Zugriffspfades wird durch die Anwendung der Paarrotationen *Zig-Zig* und *Zig-Zag* (s. Abb. 3.11) auf diese Weise grob halbiert. Diese Eigenschaft besitzt *Move To Root* nicht. Komputationell sind beide Heuristiken aber vergleichbar aufwändig. Sleator und Tarjan (1985b) stellen zwei Varianten des Splayings (*Semisplaying* und *Conditional Splaying*) vor, die in erster Linie den Aufwand der Restrukturierung reduzieren sollen, aber auch die Wirkung der Selbstorganisation (besonders in Bezug auf die lokale Adaptivität) mindern.[55]

[55] In dieser Arbeit wird dies insofern genutzt, als dass die approximative Suche nicht zu einem *Splaying* führt, also die Analyse auf einem *Splay*-TST in einem angenommenen *Steady State* operiert. Dies ist eine Form des *Conditional Splayings*. (Sleator und Tarjan, 1985b, S. 673)

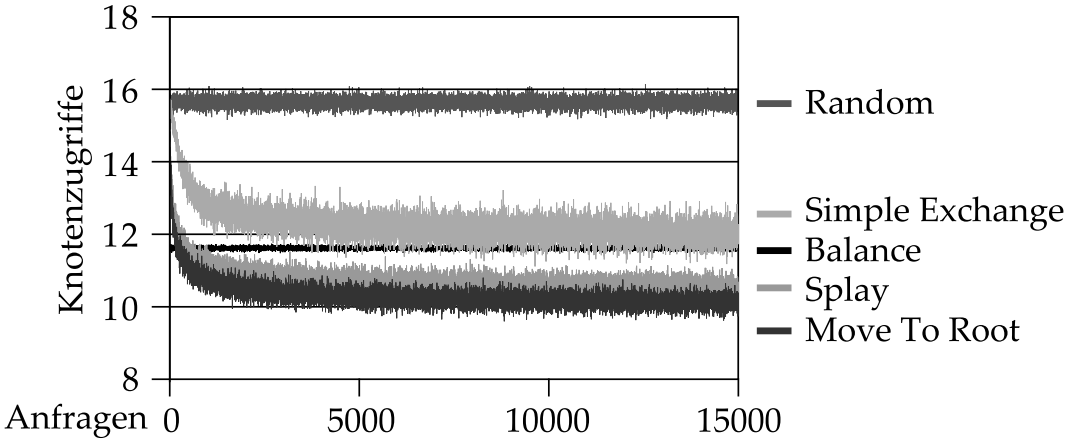


Abbildung 3.12: Experiment C – Vergleich von *Move To Root*, *Simple Exchange* und *Splay* in einem binären Suchbaum. Für den undynamischen Vergleich ist eine zufällige Konfiguration und der balancierte Baum angegeben. (*Cory*)

Experiment C / C2

Verglichen wird die Anzahl notwendiger Knotenvergleiche in einem mit *Move To Root*, *Simple Exchange* und *Splay* restrukturierten binären Suchbaum. Als undynamische Vergleichsstrukturen dienen ein Baum mit zufälliger Konfiguration und ein balancierter Baum. Der Baum enthält alle Einträge des *Cory*-Korpus. Angegeben sind die durch 1000 Durchläufe gemittelten Werte einer Anfragesequenz $|Q| = 15000$. Die Wahrscheinlichkeitsverteilung entspricht der Wortfrequenzverteilung im Text. Die initiale Konfiguration wurde zu Beginn jeden Durchlaufs neu randomisiert mit Ausnahme des balancierten Baumes, dessen Konfiguration immer gleich blieb. In einem Vergleichsexperiment (C2) wurde der selbe Versuch mit dem *Moby*-Korpus in 100 Läufen durchgeführt.

	Cory	Moby
Random	$\approx$ 15,638	$\approx$ 16,801
Balance	$\approx$ 11,618	$\approx$ 12,372
Move To Root	$\approx$ 10,152	$\approx$ 10,59
Simple Exchange	$\approx$ 12,067	$\approx$ 12,37
Splay	$\approx$ 10,602	$\approx$ 11,067

Abbildung 3.13: Experiment C/C2 – Durchschnittliche Anzahl von Knotenzugriffen in einem angenommenen *Steady State* $Q_{10000 \cdots 15000}$

Wie Tabelle 3.13 zeigt, sind sowohl *Move To Root* als auch *Splay* dem balancierten

Baum in diesem Szenario überlegen. Mit dem randomisierten Baum kann auch *Simple Exchange* konkurrieren (s. Abb. 3.12).

Die Überlegenheit der *Move To Root* Heuristik gegenüber dem *Splaying* ist auf die gute Verteilung der Anfrage zurückzuführen. **Experiment D** (s. Abb. 3.14) zeigt das Verhalten der beiden Heuristiken im *worst case* (s. Abb. 3.15). (Verglichen werden *Move To Root* und *Splay* in einem binären Suchbaum bei initialer alphabetischer Sortierung. Der Baum enthält alle Einträge des *Cory*-Korpus. Angegeben sind die durch 100 Durchläufe gemittelten Werte einer Anfragesequenz $|Q| = 15000$. Die Wahrscheinlichkeitsverteilung entspricht der Wortfrequenzverteilung im Text.) Dabei zeigt die *Splay*-Heuristik ihre Fähigkeit zu einer echten, baumartigen Restrukturierung, was zu einer schnelleren Konvergenz führt, während sich *Move To Root* entsprechend der *Move To Front* Listen-Heuristik verhält und nicht in der Lage ist, die Entartung des Baumes aufzuheben.

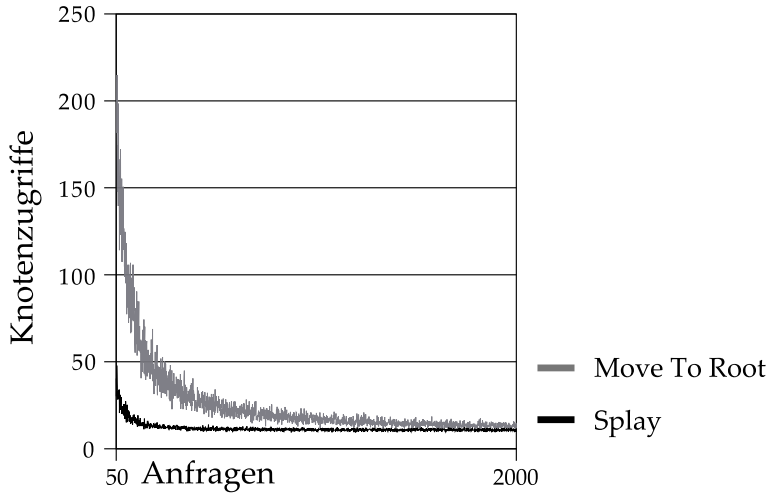


Abbildung 3.14: Experiment D – *Move To Root* und *Splay* in einem binären Suchbaum dessen initiale Konfiguration dem *worst case* einer alphabetischen Sortierung entspricht. (*Cory*)

Neben binären *Splay*-Bäumen gibt es auch weitere baumbasierte selbstorganisierende Datenstrukturen, die das *Splaying* einsetzen, etwa *k-Splay Trees* (Sherk, 1995) oder selbstorganisierende *k-Forests* (Martel, 1991). Für die Trie-Implementierung wird im Folgenden aber nur die Übertragung der Heuristiken für binäre auf ternäre Suchbäume vorgestellt.

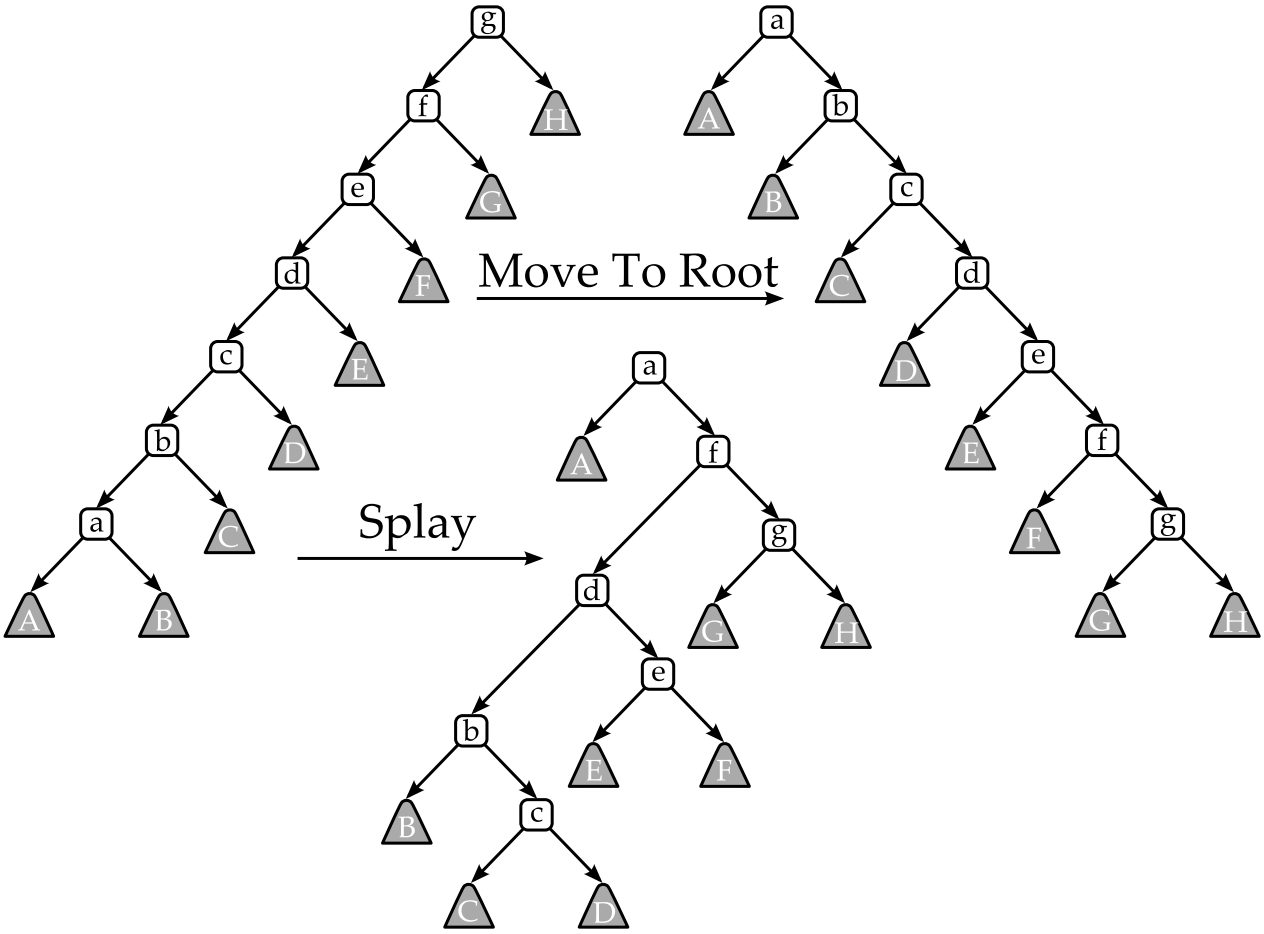


Abbildung 3.15: *Splaying* und *Move To Root* am Knoten *a* bei einer *Worst Case* Konfiguration

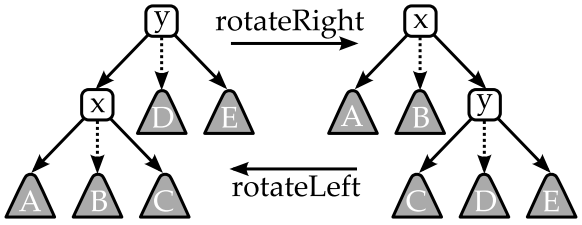


Abbildung 3.16: Rotation der Kante zwischen x und y in einem TST

3.4 Ternäre Suchbäume

Sleator und Tarjan (1985b) selbst stellten die einfache Möglichkeit der Adaption des *Splayings* für ternäre Suchbäume vor. Da TSTs eine Hierarchie binärer Suchbäume abbilden (s. Abb. 2.7 auf Seite 16: Knoten die mit durchgezogenen Pfeilen verbunden sind, entsprechen einem binären Suchbaum, gestrichelte Pfeile einem Hierarchiewechsel), kann, wie beim Listen-Trie, eine Restrukturierung nur pro Ebene stattfinden. Um einen Knoten y zu *splayen*, werden *bottom-up* entlang des Zugriffspfades von y zur Wurzel die *Splaying*-Operationen durchgeführt, mit der zusätzlichen Bedingung, dass, wenn ein Knoten y ein mittleres Kind ist, das Splaying mit $p(y)$ fortgesetzt wird anstatt mit y (s. Algorithmus auf Seite 78).[56]

Ebenso lassen sich die Heuristiken *Move To Root* und *Simple Exchange* auf ternäre Suchbäume übertragen (s. Algorithmen auf Seite 76). Nach dem *Splayen* oder einem *Move To Root* von y besteht entsprechend der Abbildung auf Seite 16 der Pfad von der Wurzel zu y ausschließlich aus gestrichelten Kanten – y würde bei einem unmittelbaren erneuten *Access* $|y|$ Kantenvergleiche benötigen. Auf diese Weise ist auch die lokale Adaptivität im TST gegeben.

Experiment E / E2

Verglichen wird die Anzahl notwendiger Knotenvergleiche in einem mit *Move To Root*, *Simple Exchange* und *Splay* restrukturierten ternären Suchbaum. Als undynamische Vergleichsstrukturen dienen ein Baum mit zufälliger Konfiguration und ein balancierter Baum. Der Baum enthält alle Einträge des *Cory*-Korpus. Angegeben sind die durch 1000 Durchläufe gemittelten Werte einer Anfragesequenz $|Q| = 15000$. Die Wahrscheinlichkeitsverteilung entspricht der Wortfrequenzverteilung im Text. Die

[56] Für einen ausführlichen Vergleich selbstorganisierender ternärer Suchbäume, siehe Badr und Oommen (2005).

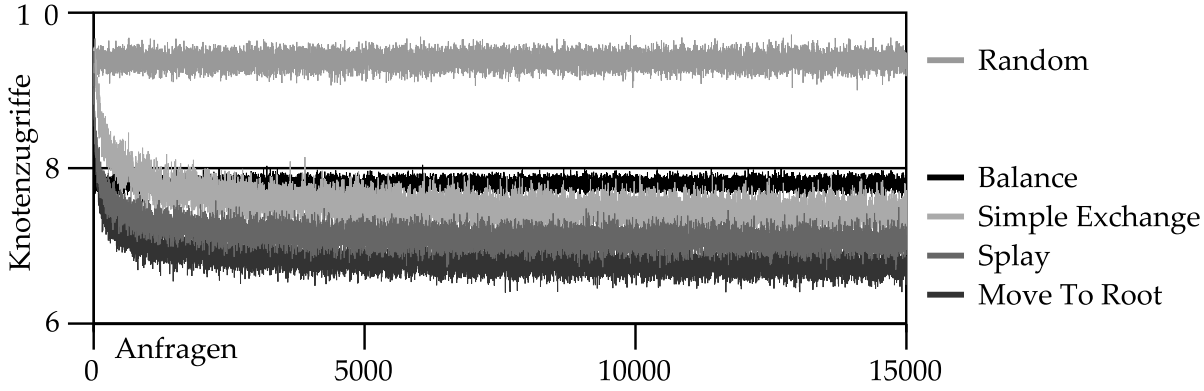


Abbildung 3.17: Experiment E – *Move To Root*, *Simple Exchange* und *Splay* in einem ternären Suchbaum. Für den undynamischen Vergleich ist eine zufällige Sortierung und der balancierte Baum angegeben. (*Cory*)

initiale Konfiguration wurde zu Beginn jeden Durchlaufs neu randomisiert mit Ausnahme des balancierten Baumes. In einem Vergleichsexperiment (E2) wurde der selbe Versuch mit dem *Moby*-Korpus in 100 Läufen durchgeführt.

	Cory	Moby
Random	$\approx$ 9,383	$\approx$ 9,684
Balance	$\approx$ 7,797	$\approx$ 8,288
Move To Root	$\approx$ 6,752	$\approx$ 6,88
Simple Exchange	$\approx$ 7,427	$\approx$ 7,417
Splay	$\approx$ 7,069	$\approx$ 7,205

Abbildung 3.18: Experiment E/E2 – Durchschnittliche Anzahl von Knotenzugriffen in einem angenommenen *Steady State* $Q_{10000 \cdots 15000}$

Anders als im binären Suchbaum sind alle selbstorganisierenden Heuristiken der balancierten Konfiguration überlegen – besonders deutlich wird dies im Vergleichskorpus (s. Tab. 3.18; Abb. 3.17). Wieder ist *Move To Root* im gegebenen Szenario die beste Heuristik – allerdings verhält sich *Simple Exchange* kaum schlechter und könnte in einer praktischen Implementation *Move To Root* und *Splay* insofern überlegen sein, als es den geringsten komputationellen Aufwand bedeutet, da pro Ebene maximal eine Rotation durchgeführt werden muss.

Das untersuchte Lexikon enthält allerdings ausschließlich Wörter der Anfragesequenz. Es wäre zu erwarten, dass *Simple Exchange* bei größerem Lexikon und kürzerer Anfragesequenz ein deutlich schlechteres Verhalten zeigt als die anderen Heuristiken mit größerem Grad an lokaler Adaptivität. **Experiment F** wiederholt Experiment E

mit einem Lexikon, das alle Einträge von *Cory*, *Moby* und *Wiki* vereint, und dessen Anfragen aus *Cory* stammen. Die Wahrscheinlichkeitsverteilung entspricht der Wortfrequenzverteilung im Text. Angegeben sind die durch 100 Durchläufe gemittelten Werte einer Anfragesequenz $|Q| = 15000$.

Random	≈ 10.772
Balance	≈ 8.614
Move To Root	≈ 6.837
Simple Exchange	≈ 7.55
Splay	≈ 7.176

Abbildung 3.19: Experiment F – Durchschnittliche Anzahl von Knotenzugriffen in einem angenommenen *Steady State* $Q_{10000 \cdots 15000}$

Im Vergleich der Tabellen 3.18 und 3.19 ist zu erkennen, dass sich zwar die absoluten Werte bei größerem Lexikonumfang leicht erhöhen, das Verhältnis der Heuristiken zueinander aber bestehen bleibt und alle dem balancierten Baum überlegen sind.

Experiment E ist die Übertragung des Listen-Trie-Experiments B (S. 28) auf den ternären Suchbaum. Der Vorteil der binären Suche innerhalb der Knotenübergänge ist deutlich – erforderte die effizienteste Heuristik im Briandais-Trie, *Frequency Count*, noch durchschnittlich $\approx 12,253$ Knotenzugriffe, ist im TST selbst der randomisierte Baum mit $\approx 9,383$ Knotenzugriffen dem überlegen.

Damit ist ein ternärer Suchbaum eine adäquate elektronische Repräsentation eines Lexikons. Die Zugriffsgeschwindigkeit kann – gemessen in Knotenzugriffen – bei erwartet schiefer Verteilung durch selbstorganisierende Heuristiken optimiert werden, wobei alle vorgestellten Regeln unterschiedliche Vorteile in Bezug auf das beschriebene Szenario haben: *Splaying* zeigt ein besseres Verhalten im *worst case*, *Move To Root* hat in den vorgestellten Experimenten mit realistischen Daten den niedrigsten asymptotischen Wert, dafür ist *Simple Exchange* die komputationell bescheidenste Variante.

4 Approximativer Stringvergleich

Durch den exakten Vergleich von Zeichenketten in Lexika können Rechtschreibkorrekturprogramme den Autor eines Textes auf dem System unbekannte Wörter aufmerksam machen. Dies kann bereits eine Hilfe sein, denn so wird der Schreibende diese Wörter nocheinmal genauer betrachten. Eine größere Hilfe wäre dann gegeben, wenn das System dem Autor Vorschläge unterbreiten könnte, wie das Wort richtig zu schreiben ist.

Gesetz dem Fall, das durch die exakte Suche nicht gefundene Wort enthält tatsächlich einen oder mehrere Fehler – wie ist es dem Computerprogramm möglich, diese Fehler nicht nur zu erkennen sondern auch insofern zu korrigieren, dass das falsche Wort auf ein korrekt geschriebenes im Lexikon passt?

Hierzu muss dem System bekannt sein, wann zwei Wörter als ähnlich betrachtet werden können – mehr noch: Es muss bestimmen können, inwieweit zwei Zeichenketten x und y sich ähnlicher oder unähnlicher sind als zwei Zeichenketten x und z. Für die Ähnlichkeit muss also ein Maß definiert werden, dass den Grad an Ähnlichkeit zweier Zeichenketten angibt. Damit ist in dieser Arbeit eine Form der *physikalischen Ähnlichkeit* gemeint – die etwa von der Ähnlichkeit von Wortbedeutungen oder der Ähnlichkeit durch gleichen etymologischen Ursprung zu trennen ist. Faulk (1964) definiert drei mögliche Funktionen zur Messung physikalischer Ähnlichkeit von Zeichenkettenpaaren:

Materiell: In welchem Maße enthalten zwei Zeichenketten identische Zeichen?

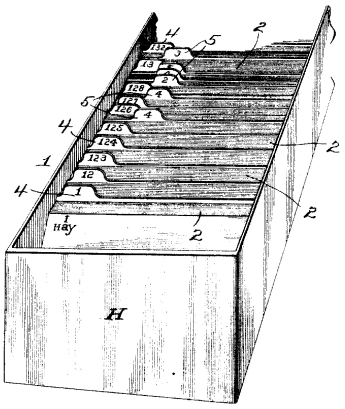
Ordinal: In welchem Maße stimmt die Reihenfolge dieser Zeichen in beiden Zeichenketten überein?

Positional: In welchem Maße stimmen die Positionen der Zeichen in beiden Zeichenketten überein?

Letztlich formuliert er die *Total Similarity Function*, die das Maß wiedergibt, mit welchem zwei Zeichenketten aus übereinstimmenden Zeichen bestehen, die in selber Reihenfolge und an selber Position in beiden Zeichenketten vorkommen.

Angell et al. (1983) führen als Beispiel für ein Ähnlichkeitsmaß das rein den materiellen Aspekt Faulks wiedergibt, die n-Gramm-Distanz zweier Zeichenketten an, für ordinale Ähnlichkeit Schlüsselsysteme wie SOUNDEX und für positionale Maße die Hamming-Distanz.[57] Im Folgenden wird auf diese Maße eingegangen um letztlich auf die Levenshtein-Metrik überzugehen, die alle drei Ähnlichkeitsfunktionen vereint und damit der *Total Similarity Function* Faulks am nächsten kommt.

4.1 Ähnlichkeitsschlüssel



Schon früh wurden Verfahren des Suchens nach ähnlichen Wörtern benötigt – noch bevor Computer hierfür zum Einsatz kamen. Beispielsweise mussten auf Ämtern Register nach Namen durchsucht werden, ohne die genaue Schreibung zu kennen, da dem Suchenden der betreffende Name nur mündlich mitgeteilt wurde. Ein erster und bis heute in vielen Datenbanksystemen zu findender Ansatz ist das SOUNDEX-Verfahren, welches bereits 1918 von Margaret K. Odell und Robert C. Russel für Buchindizes und Karteikartensysteme entwickelt und patentiert wurde.[58] Mit diesem Verfahren ist es möglich, Namen in einen Code zu transformieren, der lediglich ihre wichtigsten phonetischen Merkmale enthält.

0	1	2	3	4	5	6
a, e, i, o, u, h, w, y	b, f, p, v	c, g, j, k, q, s, x, z	d, t	l	m, n	r

Abbildung 4.1: SOUNDEX-Zeichenklassen

Dabei werden alle Zeichenketten, gleich welcher Länge, auf einen vierstelligen „SOUNDEX-Code“ reduziert, bestehend aus einem Buchstaben und drei Zahlen. Die

[57] Angell et al. (1983) nennen ein System nach Glantz (1957), das in wesentlichen Punkten aber mit der Hamming-Distanz für alphanumerische Werte übereinstimmt.

[58] Russel (1918, 1922). Die Bezeichnung SOUNDEX wurde dem Verfahren erst später gegeben.

Transformation einer Zeichenkette $x = \texttt{BILLERBECK}$ in einen SOUNDEX-Code x' geschieht wie folgt:[59]

1. Der Anfangsbuchstabe des Codes entspricht dem ersten Zeichen des zu transformierenden Wortes: $x' = \texttt{B}$.

2. Jeder Folgebuchstabe wird gegen eine Zahl entsprechend einer Code-Tabelle (s. Abb. 4.1) ausgetauscht, die Buchstaben, die im Englischen ähnlich klingenden Lauten entsprechen, zu Zeichenklassen zusammenfasst: $x' = \texttt{B044061022}$.

3. Alle Vorkommen der Null, d.h. alle ehemaligen Vokale und die Buchstaben h, w und y, werden aus dem Code entfernt: $x' = \texttt{B446122}$.

4. Alle Sequenzen identischer Zahlen werden auf ein Vorkommen reduziert: $x' = \texttt{B4612}$.

5. x' wird auf die Länge 4 gekürzt: $x' = \texttt{B461}$. Ist der resultierende Code kürzer als vier Stellen, werden entsprechend Nullen angehängt.

Auf diese Weise sollen gleiche oder ähnlich ausgesprochene Wörter einen identischen Code erhalten und damit an selber Stelle im Register abgelegt werden.[60] Allerdings erhalten zuweilen auch sehr unähnliche Wörter identische SOUNDEX-Codes ($Soundex(\texttt{Knuth}) = Soundex(\texttt{Kant}) = \texttt{K530}$) während zugleich ähnlich ausgesprochene Wörter in verschiedene Codes transformiert werden ($Soundex(\texttt{Rogers}) = \texttt{R262} \neq Soundex(\texttt{Rodgers}) = \texttt{R326}$).[61] Dies führte in Folge, insbesondere in Bezug auf Computer-Anwendungen, zu Weiterentwicklungen des Verfahrens und Übertragungen in andere Sprachen.

[59] Véronis (1988), S. 51. Das vorgestellte Verfahren ist eine modifizierte, vereinfachte Form des durch Russel patentierten Systems und weicht in wenigen Punkten ab.

[60] Im Falle der approximativen Suche in Lexika bedeutet dies eine Generierung eines Index über SOUNDEX-Codes der Lexikon-Schlüssel. Bei exakter Anfrage von $Soundex(x)$ werden alle Einträge mit dem gleichen Code zurückgegeben.

[61] Beispiele aus Knuth (1998), S. 395.

4.2 n-Gramm-Vergleiche

n-Gramm-Vergleiche gehören ebenfalls zu den älteren Verfahren des unscharfen Stringvergleichs und werden in zahlreichen Distanzmaßen verwendet.

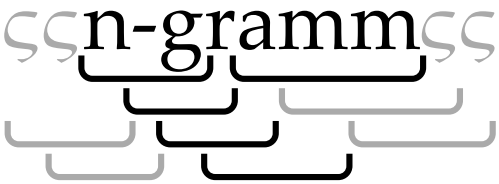


Abbildung 4.2: n-Gramm

Ein n-Gramm ist eine Zeichenkette $v = v_0 \cdots v_{n-1}$ über Σ. Σ^n ist die Menge aller Zeichenketten der Länge n über Σ, also gilt $v \in \Sigma^n$. Die n-Gramme eines Strings $x = x_0 \cdots x_{m-1}$ über Σ sind alle Teilzeichenketten der Länge n in x, also $G(x) = \{x_i \cdots x_{i+n-1} \mid 0 \leq i \leq m-n\}$ (s. Abb. 4.2).[62] Ukkonen (1992) führt eine vektorbasierte n-Gramm-Distanz D_{Ukk} ein: $G(x)[v]$ bezeichnet die Anzahl der Vorkommen von v in x. Das n-Gramm-Profil von x ist der Vektor $G_n(x) = (G(x)[v]), v \in \Sigma^n$. Die Distanz-Funktion D_{Ukk} berechnet die Differenz der n-Gramm-Profile zweier Zeichenketten x und y ($n > 0$) in $O(|x| + |y|)$:

$$D_{Ukk,n}(x,y) = \sum_{v \in \Sigma^n} |G(x)[v] - G(y)[v]|$$

Es zeigt sich, dass D_{Ukk} eine Distanz von 0 selbst bei ungleichen Zeichenketten x und y erlaubt, solange sie identische n-Gramm-Profile besitzen. Ein Beispiel hierfür wäre $D_{Ukk,n=3}(\texttt{beadubbe}, \texttt{dubbeadu}) = 0$.[63]

Die vektorbasierte n-Gramm-Distanz ist einigen auf n-Grammen beruhenden Maßen insofern überlegen, als sie mehrfach vorkommende n-Gramme in einem Wort berücksichtigt. Der *Dice Koeffizient*[64] etwa ignoriert die Anzahl gleicher n-Gramme in

[62] Je nach Definition wird x auch um $a \times \varsigma$, $a \leq (n-1)$, sowohl prä- als auch suffigiert, so dass sich für die n-Gramme von x ergibt: $G(x) = \{x_i \cdots x_{i+n-1} \mid 0 - n + 1 \leq i \leq m - n + a\}$. Auf diese Weise sind auch n-Gramm-Profilierungen für Zeichenketten mit einer Länge $< n$ möglich. Dadurch gewinnt das Maß zusätzlich an positionalem Charakter, da für $2 \times a$ n-Gramme die Position im String feststeht. (ς ist hierbei ein beliebiges Zeichen $\varsigma \notin \Sigma$. Die Menge aller n-Gramme über Σ ist damit $(\Sigma \cup \{\varsigma\})^n$.)

[63] Ukkonen (1992) stellt zwei Verfahren zur Generierung derartiger Zeichenkettenpaare mit identischer n-Gramm-Profilierung vor.

[64] Siehe beispielsweise Faulk (1964).

einem Wort und berücksichtigt ausschließlich die Menge.[65]

$$D_{Dice,n}(x,y) = \frac{2|x^n \cap y^n|}{|x^n| + |y^n|}$$

Auf diese Weise ist $D_{Dice,n=2}(\texttt{ananas},\texttt{nase}) = D_{Dice,n=2}(\texttt{anas},\texttt{nase}) = \frac{2}{3}$, während $D_{Ukk,n=2}(\texttt{ananas},\texttt{nase}) = 4 > D_{Ukk,n=2}(\texttt{anas},\texttt{nase}) = 2$.

Je nach Anwendung werden für Zeichenkettenvergleiche häufig n-Gramme der Längen $n = 2$ (Bigramme) oder $n = 3$ (Trigramme) herangezogen. n-Gramme der Länge $n = 1$ werden Monogramme genannt und nur eingesetzt, wenn es um die einfache Materialidentität geht.[66]

n-Gramm-Distanzen geben, anders als das SOUNDEX-System, ein Maß des Ähnlichkeitsgrades wieder, was für die Bestimmung eines ähnlichsten Wortes im Lexikon unabdingbar ist. Allerdings haben sie eine ähnliche Schwäche wie Ähnlichkeitsschlüsselsysteme: Sie geben zuweilen zwei sehr verschiedene Zeichenketten als identisch, also mit Abstand 0, zurück. Echte metrische Ähnlichkeitsmaße wie die Folgenden tun dies nicht.

4.3 Hamming-Distanz

Als besonders populär, da einfach zu berechnen, kann die *Hamming*-Distanz (oder Signal-Distanz) gelten.[67] Sie gibt die Anzahl unterschiedlicher Zeichen zweier gleichlanger Zeichenfolgen x und y ($|x| = |y|$) an.[68] Es gilt:

$$D_{Ham}(x,y) = \sum_{i=1}^{|x|} \gamma(x_i, y_i)$$

$$\gamma(x_i, y_i) = \begin{cases} 0, \textit{ wenn } x_i = y_i \\ 1, \textit{ wenn } x_i \neq y_i \end{cases}$$

Ursprünglich wurde unter D_{Ham} ausschließlich die Distanz binärer Zeichenketten mit $\Sigma = \{0,1\}$ verstanden, die der Erstellung von *Hamming-Codes* zur Fehler-Detektion

[65] x^n ist die Menge aller n-Gramme in x, y^n ist die Menge aller n-Gramme in y.
[66] Zwei Zeichenketten x und y mit $D_{Ukk,n=1}(x,y) = 0$ sind dementsprechend Anagramme.
[67] Hamming (1980), S. 44f.
[68] ATIS-T1.523-2001 H.

0	0	1	1	0	1	
0	0	1	0	0	1	
0	0	0	1	0	0	= 1

o	r	g	a	n	e	
o	r	a	n	g	e	
0	0	1	1	1	0	= 3

Abbildung 4.3: Hamming-Distanzen für binäre und alphanumerische Zeichenfolgen

und -Korrektur digitaler Signale dienen,[69] doch auch alphanumerische Zeichenfolgen lassen sich auf diese Weise vergleichen (s. Abb. 4.3).

Bei der Hamming-Distanz handelt es sich um eine so genannte Metrik. Eine Distanz gilt als metrisch, wenn für beliebige Objekte einer Domäne (in diesem Fall Zeichenketten $\in \Sigma^*$) folgende Eigenschaften zutreffen:[70]

Nicht-Negativität: $\forall x, y \in \Sigma^*,\ d(x,y) \geq 0$

Symmerie: $\forall x, y \in \Sigma^*,\ d(x,y) = d(y,x)$

Reflexivität: $\forall x, y \in \Sigma^*,\ d(x,x) = 0$

Positivität: $\forall x, y \in \Sigma^*,\ x \neq y \Rightarrow d(x,y) > 0$

Dreiecksungleichung: $\forall x, y \in \Sigma^*, d(x,z) \leq d(x,y) + d(y,z)$

Ist die Eigenschaft der Positivität nicht erfüllt (alle anderen Eigenschaften treffen aber zu), existieren also zwei ungleiche Strings x und y in der Domäne die einen Nullabstand zueinander haben, wird von einer *Pseudometrik* gesprochen. In diese Klasse fällt, wie gesehen, die zuvor genannte n-Gramm-Distanz nach Ukkonen.

Die Berechnung der Hamming-Metrik lässt sich für binäre Strings mittels eines $|x|$-dimensionalen Würfels darstellen (s. Abb. 4.3 für die String-Länge 3), indem alle möglichen Bitfolgen der Länge $|x|$ als Koordinaten dienen. Die Hamming-Distanz entspricht dabei der Mindestanzahl an Kantentraversierungen von der Koordinate x zur Koordinate y.[71]

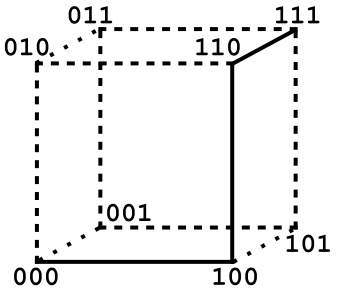


Abbildung 4.4

Insbesondere in Bereichen der optischen Zeichenerfassung (*Optical Character Recognition; OCR*) findet die Hamming-Distanz Verwendung,

[69] Hamming (1950).
[70] Zezula et al. (2006), S. 8.
[71] Hamming (1950), S. 154ff.

wo zwar falsche Zeichen erfasst werden, bei guter Segmentierung deren Anzahl jedoch korrekt ist.[72]

Die größte Einschränkung bei der Hamming-Distanz ist die Erfordernis gleichlanger Strings mit identischer Alignierung. Daher wird sie nur selten in Applikationen der Rechtschreibkorrektur verwendet.[73] Ein Vorteil ist hingegen die lineare Berechnung in $O(|x|)$.

Die Hamming-Distanz kann als so genannte *Editier*-Distanz betrachtet werden, da sie die Anzahl von Editieroperationen zählt, die auf einen String x angewendet werden müssen, um ihn in einen String y zu transformieren. Die einzig mögliche Editieroperation hierbei ist die **Ersetzung**, d.h. die Überführung eines Zeichens in ein anderes:

Ersetzung / Substitution: Symbol a_i wird zu Symbol b ergibt $a_1 \cdots a_{i-1} b a_{i+1} \cdots a_m$.

Für die Überführung einer Zeichenkette x in die Zeichenkette y bei $|x| \neq |y|$ sind neben der Substitution jedoch zwei weitere Editieroperationen notwendig: Die **Einfügung** und die symmetrische Operation der **Löschung**. Eine Editierdistanz, die diese Operationen berücksichtigt, ist die *Levenshtein-Distanz*.

4.4 Levenshtein-Distanz

Damerau (1964) unternahm die erste Untersuchung von Fehlerklassen bei Tastatureingaben und stellte fest, dass 80% der Eingabefehler in Wörtern auf vier einfache Fehlertypen zurückgeführt werden können:

- Ein Zeichen ist falsch.
- Ein Zeichen fehlt.
- Ein Zeichen wurde hinzugefügt.
- Zwei aufeinanderfolgende Zeichen wurden vertauscht.

[72] Ausnahmen stellen hier die Segmentierungsfehler *Split* (z.B. `m` → `rn`) und *Merge* (z.B. `cl` → `d`) dar, s. S. 53.

[73] Ein Beispiel hierfür zeigen Gollapudi und Panigrahy (2006), die Strings so in gleichlange Bitvektoren transformieren, dass ähnliche Strings in ähnlichen Bitvektoren mit geringer Hamming-Distanz resultieren.

Eine Beobachtung, die in der Folge vielfach bestätigt werden konnte.[74] Unabhängig hiervon formulierte Levenshtein (1965) ein Distanzmaß beruhend auf der minimal notwendigen Anzahl von Editieroperationen um einen beliebigen String x in einen beliebigen String y zu überführen.[75] Die Transformation lässt sich als Korrektur der ersten drei Basisfehler von Damerau bezeichnen, die zusätzlich in Kombination auftreten können:[76] Ein falsches Symbol erfordert die schon von der *Hamming-Distanz* bekannte **Ersetzung**, ein fehlendes Symbol eine **Einfügung** und ein hinzugefügtes Zeichensymbol eine **Löschung**.

Einfügung / Insertion: Symbol b wird hinter Position i in einen String $a_1 a_2 \cdots a_m$ eingefügt ergibt $a_1 \cdots a_i b a_{i+1} \cdots a_m$.

Löschung / Deletion: Symbol a_i wird gelöscht ergibt $a_1 \cdots a_{i-1} a_{i+1} \cdots a_m$.

Mit diesen drei Operationen ist es möglich, auch Zeichenketten ungleicher Länge zu transformieren und den Aufwand anhand der minimalen Anzahl dieser Operationen zu messen.[77] Dieses Maß wird heute als *Levenshtein-* oder *minimale Editier-Distanz* bezeichnet.[78]

Die drei Basisoperationen lassen sich mithilfe von *Editier-Transkripten* darstellen.[79] Ein Editier-Transkript ist ein String $T_{x \to y}$ über $\Sigma = \{D, I, S, M\}$ mit $T_{x \to y} \in \Sigma^*$, der die Transformation eines Strings x in einen String y beschreibt. Jedes Zeichen des Transkripts steht dabei für eine der oben beschriebenen Operationen.[80]

Für die Umwandlung eines Strings in einen anderen existieren zahlreiche mögliche Editier-Transkripte (s. Abb. 4.5). Als *optimal* werden jene bezeichnet, die eine minimale Anzahl an Editieroperationen vom Typ D,I oder S wiedergeben. Auch hier können

[74] Kukich (1992) führt Studien an, die zwischen 69% und 94% einfache Fehler (*single-error misspellings*) in Nichtwort-Fehler-Korpora angeben, die übrigen 6% bis 31% sind Mehrfachfehler (*multi-error misspellings*; S. 388).

[75] Wie bei Hamming handelte es sich zunächst nur um eine Betrachtung binärer Strings, ebenfalls zur Korrektur digitaler Signale.

[76] Damerau (1964) berücksichtigte nur das einfache Vorkommen eines Fehlers während Levenshtein diese als atomar betrachtete.

[77] Dameraus Vertauschung lässt sich hierbei durch eine Kombination der Basisfehler umformulieren, beispielsweise durch eine Löschung von a_i mit einer Einfügung des Zeichens $b = a_i$ hinter Position $i + 1$. Für ein Maß welches diese Editierung als eine Operation auffasst, s. Kap. 4.4.1.

[78] Um Verwechslungen mit der *Hamming-* oder der in einem späteren Abschnitt vorgestellten *Levenshtein-Damerau-Distanz* zu vermeiden, wird auf diese Bezeichnung im Folgenden verzichtet und *Editier-Distanz* als Klassenbezeichnung verschiedener Distanzmaße verwendet.

[79] Gusfield (1997), S. 215f.

[80] I steht für eine Einfügung (*Insertion*), D für eine Löschung (*Deletion*), S für eine Ersetzung (*Substitution*) und M für die Nichtoperation einer Übereinstimmung (*Match*).

o	r	g	a	n	e	_	_
_	g	r	a	_	n	i	t
D	S	S	M	D	S	I	I

o	r	_	g	a	n	_	e
g	_	r	a	_	n	i	t
S	I	D	S	D	M	I	S

o	r	g	a	n	e	_
g	r	_	a	n	i	t
S	M	D	M	M	S	I

Abbildung 4.5: Editier-Transkripte für die alignierten Strings „organe “→ „granit “. Das rechte Transkript ist optimal.

mehrere Transkripte existieren, die als *kooptimal* bezeichnet werden. Die Berechnung der *Levenshtein-Distanz* zweier Strings x und y lässt sich somit als Berechnung der Editieroperationen in einem optimalen Transkript $T_{x \to y}$ formulieren.[81] Levenshtein blieb eine elegante Form der Berechnung solcher optimaler Transkripte jedoch schuldig – diese lieferten mit einem *Dynamic-Programming*-Ansatz Wagner und Fischer (1974) in $O(|x||y|)$. Sie definieren die Berechnung des Abstandes rekursiv:

$$d(i,j) = \min \left\{ \begin{array}{l} d(i-1,j) + \gamma(\varepsilon \to y_j) \\ d(i,j-1) + \gamma(x_i \to \varepsilon) \\ d(i-1,j-1) + \gamma(x_i \to y_j) \end{array} \right\}$$

$$\gamma(\varepsilon \to y_j) = 1$$

$$\gamma(x_i \to \varepsilon) = 1$$

$$\gamma(x_i \to y_j) = \begin{cases} 0, \; wenn \; x_i = y_j \\ 1, \; wenn \; x_i \neq y_j \end{cases}$$

Die Funktion $\gamma(a \to b)$ gibt die Kosten an, ein Zeichen a in ein Zeichen b zu überführen. Hierbei werden drei Fälle unterschieden: Ist a das leere Zeichen ε und $b \in \Sigma \setminus \{\varepsilon\}$ handelt es sich um eine Einfügung mit den Kosten von 1. Umgekehrt, falls $a \in \Sigma \setminus \{\varepsilon\}$ und $b = \varepsilon$ handelt es sich um eine Löschung, die ebenfalls 1 kostet. Sind $a, b \in \Sigma \setminus \{\varepsilon\}$ und $a \neq b$ ist eine Ersetzung erforderlich und wiederum entstehen Kosten von 1. Lediglich der Fall einer Übereinstimmung $a = b$ ist kostenlos.

Um mit dieser Methode der Kostenbestimmung einzelner Zeichentransformationen die Gesamtkosten einer Stringtransformation zu ermitteln, wird das Problem $D_{Lev}(x,y)$, was den minimalen Gesamtkosten aller Zeicheneditierungen entspricht, in die Berechnung aller Zeicheneditierungen $d(i,j)$ für alle $\{i \mid 0 \leq i \leq |x|\}$ und

[81] Oder in $T_{x \leftarrow y}$. Das symmetrische Editier-Transkript ergibt sich durch die Ersetzung von D durch I und umgekehrt, da Löschung und Einfügung symmetrische Operationen sind.

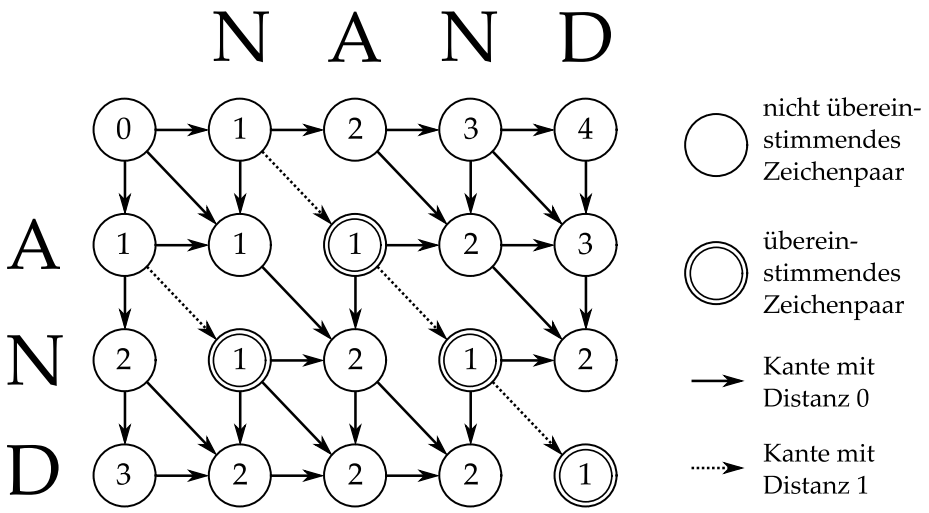


Abbildung 4.6: Berechnung der Levenshtein-Distanz nach dem *Dynamic-Programming*-Ansatz von Wagner und Fischer (1974)

$\{j \mid 0 \leq j \leq |y|\}$ umformuliert. Jeder Wert einer Zeicheneditierung $d(i,j)$ mit $i,j > 0$ steht dabei in einer Rekurrenzrelation zu allen Werten von Zeicheneditierungen $d(i',j')$ mit $i' < i$ und $j' < j$. Sind i oder $j = 0$, also gibt es keine kleineren Indizes i', j', wird auf so genannte Basiskonditionen für $d(i,j)$ zurückgegriffen.[82] Für den Fall der Levenshtein-Distanz gilt $d(i,0) = i$ und $d(0,j) = j$.[83] Für den Fall positiver Index-Werte i und j gilt obige rekursive Berechnung. Hierbei wird eine Matrix $|x| \times |y|$[84] (s. Abb. 4.6)[85] erstellt, in welcher die Werte für $d(i,j)$ abgelegt und für die Berechnungen mit steigenden Index-Werten abgefragt werden können. Der *Levenshtein-Abstand* der Strings x und y entspricht dabei der letzten Berechnung $d(i,j)$ mit $d(|x|,|y|)$ (s. Algorithmus S. 80). Somit gelten die Grenzen $0 \leq D_{Lev}(x,y) \leq max(|x|,|y|)$.[86] Die Levenshtein-Distanz ist bei obiger Definition von γ eine echte Metrik.

[82] Gusfield (1997), S. 218.

[83] Diese Konditionen ergeben sich, da zur Transformation der i Zeichen von x in das nullte Zeichen von y nur die Operation der Löschung (mit $\gamma(x_i \rightarrow \varepsilon) = 1$) infrage kommt, respektive für die Transformation des nullten Zeichens von x in die j Zeichen von y nur die Operation der Einfügung (mit $\gamma(\varepsilon \rightarrow y_j) = 1$).

[84] Exklusive der nullten Spalten und Zeilen entsprechend der Basiskondition.

[85] Abb. 4.6 und Abb. 4.8 orientieren sich an der Darstellung in Du und Chang (1992), S. 292.

[86] Damit entspricht die minimal zu erwartende Distanz dem Betrag der Differenz der Längen beider Strings. Dies wird in verschiedenen Verfahren des approximativen Matchings ausgenutzt, wenn es um die einfache Überprüfung auf Ähnlichkeit zweier Strings x und y geht, deren Längendifferenz $||x| - |y||$ die akzeptierte Anzahl notwendiger Editieroperationen, d.h. die Anzahl, die als so gering erachtet wird, dass beide Strings als ähnlich bezeichnet werden, übersteigt. In diesem Fall lassen sich die Strings ohne Berechnung der Matrix als unähnlich zurückweisen.

		o	r	g	a	n	e
	0	1	2	3	4	5	6
o	1	0	1	2	3	4	5
r	2	1	0	1	2	3	4
a	3	2	1	1	1	2	3
n	4	3	2	2	2	1	2
g	5	4	3	2	3	2	2
e	6	5	4	3	3	3	2

Abbildung 4.7: *Ukkonens Cut-Off* mit $k = 2$

Die Sequenzen der Editieroperationen und damit die optimalen Editiertranskripte lassen sich ermitteln, indem die minimalen Pfade von $d(0,0)$ zu $d(|x|,|y|)$ verfolgt werden. Auf diese Weise entsteht ein so genannter *Dependency Graph*. Ukkonen (1985a) bemerkte, dass für den Fall einer vorher gegebenen maximal erlaubten Fehlergrenze k nur die Zellen der Matrix Teil des Graphen sein können, die einen Wert nicht größer als k besitzen. In der Matrix wären demnach alle Einträge $\leq k$ innerhalb eines diagonalen Streifens von $d(0,0)$ zu $d(|x|,|y|)$, dessen Breite sich nach der Größe von k richtet. Um den Wert von $d(|x|,|y|)$ bei gegebenem k zu ermitteln, müssten entsprechend nur die Werte $d(i,j)$ ermittelt werden, die sich innerhalb des Streifens befinden (s. Abb. 4.7).

4.4.1 Levenshtein-Damerau-Distanz

Zwar ist es durch die nach Levenshtein vorgegebenen Operationen möglich, jeden beliebigen String x in jeden beliebigen String y zu überführen, allerdings sind die Kosten für die Reperatur des nach Damerau oft anzutreffenden Fehlers der Zeichenvertauschung bei Tastatureingaben doppelt so hoch wie die Korrektur eines Fehlers der drei anderen Typen. Daher ist es naheliegend, die Transposition als vierte Editieroperation mit den Kosten einer atomaren Editieroperation zu definieren.

Vertauschung / Transposition: Symbol a_i wird mit Symbol a_{i+1} vertauscht ergibt $a_1 \cdots a_{i+1} a_i \cdots a_m$.

Ein Abstandsmaß, das diese vier Operationen berücksichtigt, wird *Levenshtein-Damerau-Distanz* genannt. Bereits ein Jahr nach der Entwicklung des *Dynamic-Programming-*

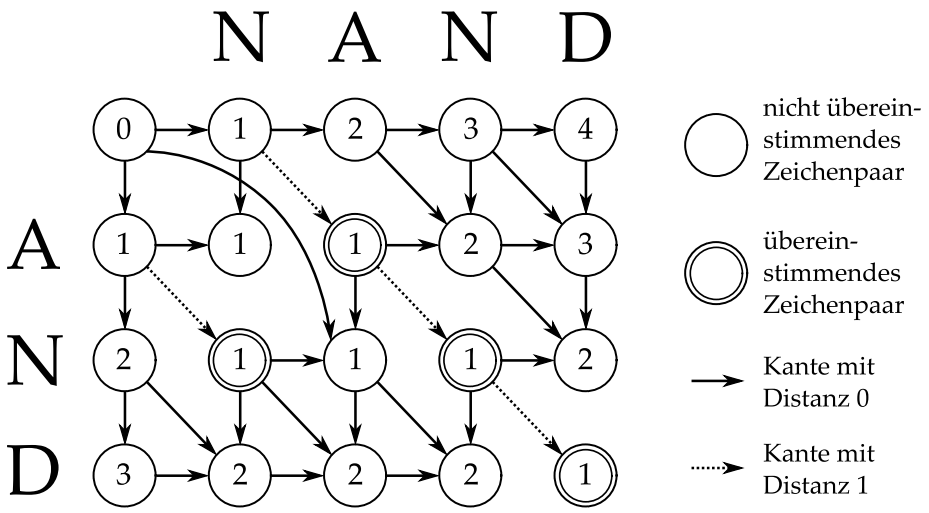


Abbildung 4.8: Berechnung der Levenshtein-Damerau-Distanz nach dem *Dynamic-Programming*-Ansatz von Lowrance und Wagner (1975)

Ansatzes zur Berechnung der Levenshtein-Distanz stellten Lowrance und Wagner (1975) eine Weiterentwicklung des Verfahrens vor, das die Transposition unterstützt.[87]

$$d(i,j) = min \left\{ \begin{array}{l} d(i-1,j) + \gamma(\varepsilon \rightarrow y_j) \\ d(i,j-1) + \gamma(x_i \rightarrow \varepsilon) \\ d(i-1,j-1) + \gamma(x_i \rightarrow y_j) \\ d(i-2,j-2) + \left\{ \begin{array}{l} \gamma(x_i \rightarrow y_j),\ wenn\ x_i = y_{j-1}\ und\ y_j = x_{i-1} \\ andernfalls\ \infty \end{array} \right\} \end{array} \right\}$$

$$\gamma(\varepsilon \rightarrow y_j) = 1$$

$$\gamma(x_i \rightarrow \varepsilon) = 1$$

$$\gamma(x_i \rightarrow y_j) = \left\{ \begin{array}{l} 0,\ wenn\ x_i = y_j \\ 1,\ wenn\ x_i \neq y_j \end{array} \right\}$$

Die Distanz $D_{LevDam}(x,y)$ entspricht auch hier $d(|x|,|y|)$. In Abbildung 4.8 ist zu sehen, wie sich die Transposition auf die Berechnung der Matrix auswirkt. Für die Berechnung der Ähnlichkeit zweier Strings wird in dieser Arbeit die *Levenshtein-Damerau-Distanz* verwendet (s. Algorithmus S. 80).

[87] Algorithmus nach Navarro (2001), S. 46.

4.5 Weitere Ansätze

In den vorangegangenen Abschnitten wurde die Kostenfunktion γ eingeführt, die bei jeder Zeichenoperation Kosten von $\gamma(a \to b) = 1$ zurückgibt, wenn $a \neq b$, um die Anzahl notwendiger Editieroperationen zu zählen. Diese Kostenfunktion lässt sich jedoch variabel gestalten. Distanzmaße, die eine andere als die binäre Kostenfunktion für die Levenshtein-Distanz nutzen, heißen *gewichtete* oder *generalisierte Levenshtein-Distanz*. Gewichtete Levenshtein-Distanzen folgen nicht in allen Fällen metrischen Regeln, wenn $\gamma(a \to b) \neq \gamma(a \leftarrow b)$ und somit die Symmetrie verletzt wird.

1	2	3	4	5	6	7	8	9	10	*
a, e, i, o, y	b, p	c, k, q	d, t	l, r	m, n	g, j	f, p, v	s, x, z	c, s, z	h, w

Abbildung 4.9: EDITEX-Zeichenklassen

Ein einfaches Verfahren zur phonetischen Gewichtung der Kostenfunktion ist EDITEX (Zobel und Dart, 1996). Es kombiniert die Berechnung der Editierdistanzen mit einem phonetischen Ähnlichkeitsschlüsselsystem, das SOUNDEX ähnelt. Dabei gilt $\gamma(a \to b) = 0$, wenn $a = b$ und $\gamma(a \to b) = 1$, wenn a und b der selben EDITEX-Zeichenklasse (s. Abb. 4.9) angehören oder $a = \varepsilon$ und $b \in \{h, w\}$ (Zeichen, die im Englischen oft stumm sind und damit bei einer Löschung eine geringere Rolle spielen sollen), andernfalls $\gamma(a \to b) = 2$. Daher ist $D_{Editex}(\texttt{saum}, \texttt{zaum}) = 1$, aber $D_{Editex}(\texttt{saum}, \texttt{baum}) = 2$.

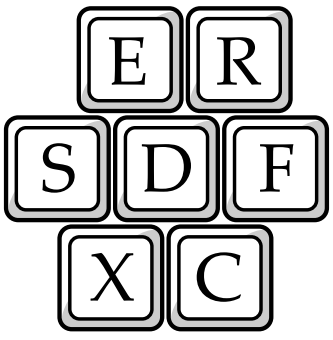


Abbildung 4.10

Insbesondere bei Fehlern durch Tastatureingaben können auch physikalische Gründe für eine Gewichtung vorliegen. Durch das Abrutschen eines Fingers auf der Tastatur versehentlich getätigte Eingaben eines falschen Symbols in einen String („Vertipper“) lassen sich entsprechend der Nähe zweier Tasten auf der Computertastatur gewichten.[88] Für die Substitution werden in solchen Fällen zweidimensionale Matrizen genutzt, die alle Kombinationen $\Sigma \times \Sigma$ vorberechnen.[89] Handelt es sich bei der Distanz um eine

[88] Véronis (1988), S. 47.

[89] Oommen und Kashyap (1998) stellen eine entsprechende Tabelle (*confusion matrix*) für Substitutionsgewichte vor, die die Tastennähe im QWERTY-Layout berücksicht. (Verfügbar unter `http://www.scs.carleton.ca/~oommen/papers/QWERTY.doc`.)

Metrik, sind die Einträge achsensymmetrisch entlang der Diagonalen.

Häufiger als diese Form der physikalischen ist jedoch die der graphemischen Gewichtung in Umgebungen der OCR anzutreffen. Dabei verursachen Substitutionen von optisch ähnlichen Zeichen geringere Kosten als optisch unähnliche, so dass beispielsweise gilt $D_{wLev,OCR}(\texttt{Bibel},\texttt{Bibe1}) < D_{wLev,OCR}(\texttt{Bibel},\texttt{Babel})$. Zudem werden in OCR-Applikationen zwei weitere atomare Editieroperationen unterstützt: **Split** und **Merge**. Ähnlich der Transposition werden hierbei Zeichenketten statt einzelner Zeichen betrachtet. Splits und Merges entstehen bei falscher Segmentierung im ersten Schritt der Zeichenerfassung. Bei einem Split wird dabei in einem String x' ein einfaches Zeichen erkannt das im Originalstring mehreren separaten Zeichen entsprach, zur Korrektur also aufgesplittet werden muss. Da es für diesen Fall unterschiedlich wahrscheinliche Kombinationen gibt, werden Kostengewichte $\gamma(a \rightarrow b_1 \cdots b_i)$ definiert, mit üblicherweise $i = 2$. Umgekehrt ist ein Merge, also die Verschmelzung aufeinanderfolgender Zeichen erforderlich, wenn eine Segmentierung in x' angenommen wurde, die in x nicht vorkam. Zur Reparatur wird eine Operation mit der Kostenfunktion $\gamma(a_1 \cdots a_i \rightarrow b)$ definiert, mit in den meisten Fällen $i = 2$. Zwei Beispiele sind $D_{wLev,OCR_{split}}(\texttt{zom},\texttt{zorn}) = 1$ bei $\gamma(\texttt{m} \rightarrow \texttt{rn}) = 1$ und $D_{wLev,OCR_{merge}}(\texttt{baclen},\texttt{baden}) = 1$ bei $\gamma(\texttt{cl} \rightarrow \texttt{d}) = 1$. Die Gewichtung von einfachen Substitutionen und so genannten Multisubstitutionen ist von den Vorverarbeitungsschritten der OCR abhängig, die von vielen Faktoren, beispielsweise der zu erkennenden Schriftart, abhängt.[90]

Ein weiterer Fehler, der in der Vorverarbeitung der OCR entstehen kann, ist der so genannte *Reject*.[91] Dabei werden Zeichen mit so geringer Wahrscheinlichkeit erkannt, dass sie nicht in die weiteren Verarbeitungsschritte einfließen sollen und gegen *Wildcards* ersetzt werden, für die jede Ersetzung gleiche Kosten verursacht, und da sie nicht Teil des Alphabets sind, es zu keiner Übereinstimmung kommen kann.

Diese Form des unscharfen Stringvergleichs (*partial-match* oder *crossword puzzle search*) wurde auch in frühen statistischen Verfahren der Rechtschreibfehlerkorrektur mit Lexika verwendet: Cornew (1968) suchte innerhalb eines im Lexikon nicht gefundenen Wortes die statistisch unwahrscheinlichsten Übergänge zweier aufeinanderfolgenden Zeichen (mittels einer Bigramm-Statistik) und ersetzte das unwahrscheinlichste Zeichen gegen eine *Wildcard*.[92]

[90] Für Levenshtein-Gewichte in der OCR, siehe Ringlstetter (2003).

[91] Ullmann (1977).

[92] Das Verfahren von Cornew (1968) erlaubte zwar alle Ersetzungen, gewichtete diese aber, indem zunächst das wahrscheinlichste Zeichen im Kontext ersetzt wurde und der neue String im Lexikon nachgeschlagen wurde. Bei wiederholtem Fehler folgte das zweitwahrscheinlichste Zeichen etc.

Weitere Formen der Levenshtein-Gewichtung berücksichtigen den Kontext innerhalb der Zeichenkette, wie das *Tapering* (Zobel und Dart, 1996), welches die Position des Fehlers im String betrachtet und Ersetzungen und Löschungen am Wortanfang höher bewertet als am Wortende. Tanaka (1988) formuliert ein weitergehendes System mit zeichenkontextabhängigen Kostenfunktionen. Dabei sind bei einer Editieroperation nicht mehr nur die Symbole x_i und y_j zu berücksichtigen, sondern auch x_{i-1}, x_{i+1}, y_{j-1} und y_{j+1}.

All diese Maße haben in erster Linie das Ziel, Ähnlichkeiten zwischen zwei Strings möglichst genau zu bestimmen. Die Bedeutung solcher Ähnlichkeitsmaße geht dabei über die Verwendung in Rechtschreib- oder OCR-Korrekturprogrammen weit hinaus. So dienen sie in der Bioinformatik beispielsweise der Auswertung und dem Vergleich von DNA-Sequenzen. Doch auch weniger naheliegende Bereiche nutzen Verfahren des approximativen Stringvergleichs um etwa Bild- oder Tondaten miteinander zu vergleichen.[93]

Da ein Maß zur Bestimmung von Ähnlichkeiten zweier Zeichenketten auch als Maß von Unähnlichkeit verwendet werden kann, dienen die hier vorgestellten Methoden in der Pharmazie zum Finden möglichst unähnlicher Produktbezeichnungen für Medikamente sowohl in Bezug auf die Schreibung als auch die Aussprache,[94] um auf diese Weise Verwechslungen in der Medikamentengabe zu minimieren.[95]

Wurden alle Kombinationen erfolglos getestet, wurde das zweitunwahrscheinlichste Zeichen im String gegen eine *Wildcard* ersetzt und die Ersetzungsprozedur begann erneut.

[93] So stellen Bradley und Bradley (1983) zum Beispiel ein System zum Vergleich und damit zur Klassifikation von Vogelgesängen vor.

[94] *LA/SA Problem* (look-alike/sound-alike).

[95] Nahezu jeder vierte dokumentierte Medikationsfehler in den USA war einer Studie von 1999 zufolge auf Namensverwechslungen von Medikamenten zurückzuführen. (Lambert et al., 1999)

5 Approximativer Stringvergleich in Lexika

Die direkte Anwendung solcher Ähnlichkeitsmaße zur unscharfen Suche in elektronischen Lexika, wie sie in Kapitel 2 dargestellt wurden, wird erst seit wenig mehr als zehn Jahren praktiziert. Zuvor dominierte für diese Zwecke ein 2-Phasen-Verfahren:[96]

> **First stage.** From the dictionary words select a subset which contains all words of which the entry word may be a misspelling.[97]
>
> **Second stage.** To each word of the above subset, apply a pairwise comparison algorithm which will determine whether or not the entry word is a misspelling of that dictionary.
>
> *- Morgan (1970), S. 91*

Zobel und Dart (1995) untersuchen Verfahren sowohl für die erste Stufe der Kandidatensuche als auch die zweite Stufe des genaueren Ähnlichkeitsabgleichs. Sie bezeichnen die Stufen als *grobe* und *feine* Suche. Dabei stellen sie fest, dass die Verwendung von Ähnlichkeitsschlüssel-Indizes für die *grobe* Suche überraschend schlechte Ergebnisse liefert, n-Gramm-Indizes hingegen sehr gute. Hierfür wird die Vereinigung aller Einträge, die n-Gramme des Suchwortes enthalten, als Kandidatenmenge betrachtet, die entsprechend der Anzahl der gemeinsamen n-Gramme $|x^n \cap y^n|$ sortiert werden. Die 50 Kandidaten mit der größten Anzahl gemeinsamer n-Gramme werden der zweiten Phase zugeführt. Dieser Index lässt sich auch für die exakte Suche nutzen, indem für alle n-Gramme des Suchwortes die Mengen aller Einträge erfragt werden,

[96] In der Praxis ist dies noch heute oft der Fall.

[97] Morgan bezieht sich hierbei auf Programmiersprachen und die fehlertolerante Verarbeitung durch den Compiler. Der Compiler-Automat gibt in diesem Fall also die Teilmenge erwarteter Wörter zurück.

die das betreffende *n*-Gramm enthalten. Die Schnittmenge aller Mengen gilt als Kandidatenliste für den exakten Vergleich. Auch für die *feine* Suche lässt sich der Index nutzen, indem Ukkonens *n*-Gramm-Distanzvergleich (Kap. 4.2) auf die Kandidaten angewendet wird. Zobel und Dart kommen zu dem Schluss, „string similarity based on *n*-grams is the approximate string matching technique of choice."[98]

Für die *feine* Suche lässt sich auch das wesentlich exaktere Maß der Levenshtein-Damerau-Distanz anwenden, das jedoch größere Kosten als D_{Ukk} verursacht, wenn die Kandidatenliste sehr umfangreich ist. Ein Vorteil wäre die Unterstützung sehr viel genauerer Analysen durch die Erweiterung der Distanz um die Gewichtung von „Vertippern" oder phonetischer Ähnlichkeit in Bezug auf das Szenario der Rechtschreibprüfung.

Einige dieser Fehler könnten sogar dazu führen, dass betreffende korrekte Schreibweisen nicht in der Kandidatenliste der *groben n*-Gramm-Suche auftauschen – die Verwendung der Levenshtein-Damerau-Distanz in einem einstufigen Ansatz wäre damit wünschenswert. Während Zobel und Dart dies als „out of the question"[99] zurückweisen, stellen Shang und Merrettal (1996) sowie Oflazer (1996) automatenbasierte Verfahren vor, mit denen dies möglich ist.

5.1 In Trie-Strukturen

Shang und Merrettal entwickelten einen Algorithmus, der es erlaubt, einen Trie mittels der *Dynamic-Programming*-Berechnung des Levenshtein-Damerau-Abstands nach einem String x zu durchsuchen und alle Zeichenketten zurückzugeben, die zu x maximal einen Abstand von k Editieroperationen aufweisen (diese Form der Suche wird als *Similarity Range Query*[100] bezeichnet). Die Haupterkenntnis ihres Verfahrens ist, dass die Repräsentation des Tries die erforderlichen Berechnungen in der Levenshtein-Damerau-Matrix erheblich reduzieren kann. Statt für jeden String im Lexikon einen neuen Vergleich zu starten, wird ausgenutzt, dass der Trie gemeinsame Präfixe zusammenfasst und sich die Suffixe des Subtries eines Knotens im approximativen Vergleich die zu berechnenden Reihen in der D_{LevDam}-Matrix teilen.

Shang und Merrettal (1996) formulierten ihre Beobachtung wie folgt:

[98] Zobel und Dart (1995), S. 344.
[99] ebd.
[100] Zezula et al. (2006), S.15.

> **Observation I.** Each trie path is a prefix shared by all sistrings in the subtrie. When evaluating DP[101] tables for these sistrings, we will have identical columns up to the prefix. Therefore, these columns need to be evaluated only *once*.
>
> *- Shang und Merrettal (1996), S. 542*

Gleichzeitig sorgt die Prämisse einer zuvor erlaubten Anzahl von k Fehlern für die Möglichkeit, die Berechnung einer Tabelle abzubrechen, sobald feststeht, dass ein Präfix zu dem gesuchten String so unähnlich ist, dass kein String mit diesem Präfix dem Suchstring in einem Abstand $\leq k$ ähnlich sein kann.

> **Observation II.** If all entries of a column are $> k$, no word with the same prefix can have a distance $\leq k$. Therefore, we can stop searching down the subtrie.
>
> *- Shang und Merrettal (1996), S. 543*

Diese Beobachtung ähnelt *Ukkonens Cut-Off* (s. S. 50) und wird als *Oflazers Cut-Off* bezeichnet, da Oflazer (1996) diese Beobachtung unabhängig von Shang und Merrettal ebenfalls machte[102] und eine *cut-off edit distance* formulierte, die die minimale Editierdistanz zwischen dem Präfix eines gesuchten, inkorrekten Wortes x und dem Präfix eines möglicherweise korrekten Kandidatenstrings y bezeichnet:

$$
\begin{aligned}
D_{cuted}(x, y) = \min_{p \leq i \leq u} D_{LevDam}(x_{1 \cdots i}, y) \\
p = \max(1, |y| - k) \\
u = \min(|x|, |y| + k)
\end{aligned}
$$

[101] *Dynamic Programming.*
[102] Ein ähnliches Abbruchkriterium formulierten bereits Du und Chang (1992).

Ein Beispiel mit $k = 2$:[103]

$$D_{cuted}(\texttt{reprter},\texttt{repo}) = \min \left\{ \begin{array}{l} D_{LevDam}(\texttt{re},\texttt{repo}) = 2 \\ D_{LevDam}(\texttt{rep},\texttt{repo}) = 1 \\ D_{LevDam}(\texttt{repr},\texttt{repo}) = 1 \\ D_{LevDam}(\texttt{reprt},\texttt{repo}) = 2 \\ D_{LevDam}(\texttt{reprte},\texttt{repo}) = 3 \end{array} \right\} = 1$$

Die Traversierung des Tries folgt einer rekursiven Tiefensuche (s. Algorithmus S. 81). Bei jedem Kantenübergang[104] wird die Reihe l (erreichte Ebene im Trie) der Levenshtein-Damerau-Matrix berechnet. Ist ein Blattknoten erreicht und $D_{LevDam} \leq k$, wird der gefundene String der Menge der Kandidatenstrings hinzugefügt. Bei jedem anderen Knoten wird auf D_{cuted} getestet.[105] Ist $D_{cuted} > k$ wird die Suche am aktuellen Knoten und damit für den gesamten Subtrie abgebrochen und entsprechend die Tiefensuche fortgesetzt, bis alle möglichen Knoten erreicht wurden. Shang und Merrettal geben für den *worst case* der Kandidatensuche mit k Unterschieden $O(k|\Sigma|^k)$ an.

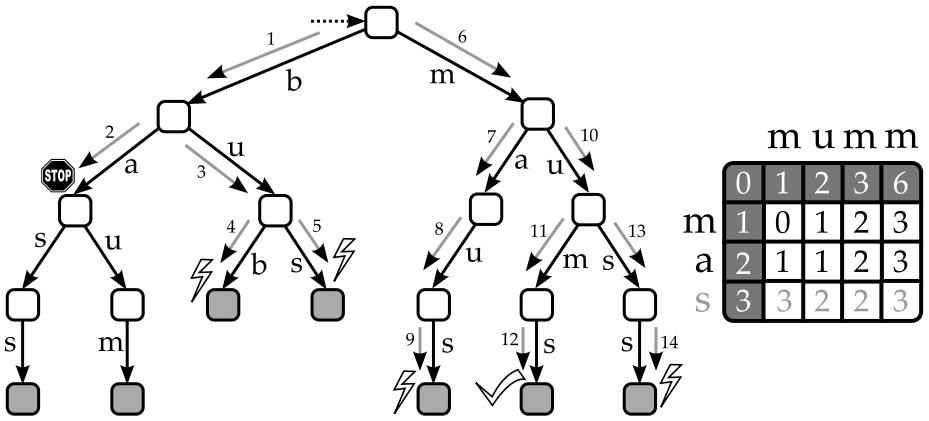


Abbildung 5.1: Abstrakter Trie mit $\Sigma = \{\texttt{a},\texttt{b},\texttt{m},\texttt{s},\texttt{u}\}$ für $L(T) = \{$`bass`, `baum`, `bub`, `bus`, `maus`, `mums`, `muss`$\}$ wird nach dem Wort $x =$ `mumm` mit $k = 1$ durchsucht. Die Levenshtein-Damerau-Matrix gibt den Zustand nach Schritt 7 wieder.

Ein Beispiel. Ein Lexikon über $\Sigma = \{\texttt{a},\texttt{b},\texttt{m},\texttt{s},\texttt{u}\}$ mit der Sortierung $(\texttt{a},\texttt{b},\texttt{m},\texttt{s},\texttt{u})$ repräsentiert die Sprache $L = \{\texttt{bass},\texttt{baum},\texttt{bub},\texttt{bus},\texttt{maus},\texttt{mums},\texttt{muss}\}$ (s. Abb. 5.1).

[103] Aus Oflazer (1996), S. 76.

[104] Gemeint sind hier Kanten entsprechend dem abstrakten Datentyp des Tries, also einem Übergang von einem Level zum nächsthöheren.

[105] Shang und Merrettal nutzen *Ukkonens Cut-Off* zur Reduzierung der Zellenberechnung und damit auch für D_{cuted}. Der Algorithmus auf Seite 82 entspricht dagegen *Oflazers Cut-Off*.

Alle Wörter, die sich maximal $k = 1$ von dem Wort $x =$ `mumm` unterscheiden, werden gesucht. Zunächst wird die erste Kante des Tries traversiert (1) und für das Zeichen `b` die erste Reihe der Levenshtein-Damerau-Matrix befüllt. D_{cuted} gibt 1 zurück. Der Tiefensuche folgend wird die erste Kante des Subtries mit dem Zeichen `a` traversiert (2) und die zweite Zeile der Matrix befüllt. D_{cuted} gibt 2 zurück – die Traversierung des Subtries wird abgebrochen, da kein String im Abstand von $k \leq 1$ mehr erreicht werden kann. Die Suche wird am letzten verzweigenden Elternknoten fortgesetzt und die Kante mit dem Zeichen `u` traversiert (3) und in die Matrix eingetragen. D_{cuted} gibt 1 zurück, also wird fortgesetzt mit dem nächsten Symbol, `b` (4), und die dritte Zeile der Matrix befüllt. Da ein Wortende erreicht ist, wird die letzte Zelle in der dritten Reihe der Matrix überprüft, die die Levenshtein-Damerau-Distanz zurückgibt. Da $D_{LevDam}(\texttt{mumm}, \texttt{bub}) = 3$ wird der String verworfen und die Tiefensuche fortgesetzt. Erst in Schritt 12 wird mit `mums` ein Wort im Abstand $k = 1$ gefunden, das der Kandidatenmenge hinzugefügt wird. Nach Schritt 14 ist die Suche beendet. Die Kandidatenliste enthält nur das Wort `mums`.

Die Suche nach der Menge der ähnlichsten Strings eines gegebenen Strings x wird als *Nearest Neighbor Query*[106] oder auch *Best Match* bezeichnet. Shang und Merrettal (1996) formulierten auch hierzu ein Verfahren. Dabei wird mit jedem Erreichen eines Blattknotens während der Traversierung $k' = D_{LevDam}(x, s)$ berechnet, wobei x das zu suchende Wort ist und s das Wort, dessen Ende durch das Erreichen des Blattknotens markiert ist. Ist $k < k'$ wird $k = k'$ gesetzt und die Menge aller bisher gefundenen Strings durch den neuen String ersetzt. Damit sinkt k während der *Best Match* Suche monoton. Initial wird beim *Nearest Neighbor* Verfahren der Grenzwert $k = \infty$ gesetzt.[107]

Ein intuitiveres Verfahren als bei Shang und Merrettal (1996) wäre eine zunächst exakte Suche nach jedem Query und erst bei einer erfolglosen Zeichenverarbeitung ein Wechsel in die approximative Suche – zunächst *depth-first* ab dem letzten erfolgreich traversierten Knoten, danach ab jedem Elternknoten bis zur Wurzel (mit Aussparung der schon untersuchten Subtries). Diese Form der Suche würde im *Nearest Neighbor* Verfahren in den meisten Fällen zu schneller sinkendem k führen, jedoch wäre eine Evaluation schwieriger, da ein weiterer Parameter zu berücksichtigen wäre: Die Position des Fehlers. Insbesondere bei Fehlern in höheren Leveln würde die Verarbeitung beschleunigt werden, während es bei Fehlern am Wortanfang zu keiner Verbesserung

[106] Zezula et al. (2006), S. 16.

[107] Für die Evaluation wurde initial $k = 100$ gewählt. Entscheidend ist $k > h$ des Tries, damit bei der ersten Pfad-Traversierung kein Abbruch durch k ausgelöst wird und $k < k'$.

des Verfahrens käme. Um die Fehlerposition hier nicht berücksichtigen zu müssen, wird das Verfahren nach Shang und Merrettal evaluiert.

Da hierbei der Trie in interner Ordnung traversiert wird, ist ähnlich der exakten Suche die Sortierung der Einträge entscheidend. Wie in Kapitel 3 gezeigt, können selbstorganisierende Datenstrukturen zu einem Geschwindigkeitsgewinn bei exakter Suche und schiefer Verteilung der Anfragen führen. Untersucht werden soll nun, inwieweit sich eine Selbstorganisation auch auf die Güte und Geschwindigkeit einer nicht-exakten Suche auswirken kann.

5.2 In selbstorganisierenden Trie-Strukturen

Im Folgenden werden einige Experimente in Bezug auf die approximative Suche gemäß Shang und Merrettal (1996) vorgestellt, die auf ternären Suchbäumen arbeiten, welche zuvor mit den in Kapitel 3 vorgestellten Heuristiken „trainiert“ wurden.[108] Untersucht werden folgende Varianten:

Similarity Range Query: Alle Kandidaten, die in einem Abstand k zu dem gesuchten Nichtwort stehen, werden entsprechend der Reihenfolge ihres Auffindens in die Kandidatenliste geschrieben. Diese ist demnach der Selbstorganisation folgend sortiert.

Best Match: Wie bei der *Similarity Range Query* werden alle gefundenen Kandidaten in eine Liste geschrieben, in der Reihenfolge ihrer Traversierung. Sinkt k, wird die Kandidatenliste gelöscht und neu angelegt.

In beiden Fällen kann die Güte des Verfahrens durch die Position des korrekten Wortes in der Kandidatenliste gemessen werden und im *Best Match*-Verfahren die Geschwindigkeit in Bezug auf die erforderlichen Knotenzugriffe – im Fall der *Similarity Range Query* unterscheidet sich die Anzahl der Knotenzugriffe nicht.

Zur Evaluation wurden für die Korpora *noisy files* erstellt. Dabei wurde eine Anfragesequenz gemäß der Wortfrequenz der Korpora generiert und jeder String der Sequenz mit K Editieroperationen zerstört.[109] Die Zerstörungen wurden zufällig im String positioniert und mit gleichmäßiger Wahrscheinlichkeit aus *Ersetzung*, *Löschung*, *Einfügung* und *Transposition* gewählt. Bei der *Einfügung* wurde ein beliebiges Zeichen

[108] Die Annahme des *Steady State* entspricht den dort präsentierten Ergebnissen.
[109] K gibt den Grad der Zerstörung in den *noisy files* an, k den Grenzwert der Suchmethode.

$\in \Sigma \setminus \{\varepsilon\}$ gewählt. Die Zerstörung wurde wiederholt bis der gewünschte Grad erreicht wurde.[110] Für das originale Anfragewort existierten keine Einschränkungen, das zerstörte Wort musste mindestens 2 Zeichen lang sein. Für die Experimente wurden alle Strukturen mit den selben *noisy files* angefragt.

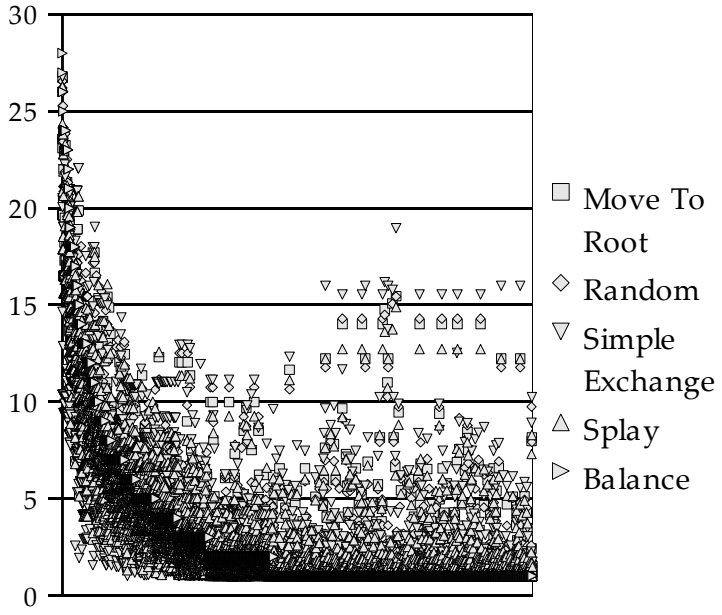


Abbildung 5.2: Experiment G – Positionierung des korrekten Begriffs in der Kandidatenliste bei einer *Similarity Range Query* mit $k = 1$ für *Splay*, *Move To Root* und *Simple Exchange* sowie die undynamische Datenstruktur mit balancierter und zufälliger Konfiguration. Die X-Achse gibt 15000 Anfragen sortiert nach absteigender Position im balancierten Baum wieder, die Y-Achse gibt die Position an. (*Cory*)

Experiment G

Eine *Similarity Range Query* mit $k = 1$ und einem *noisy file* mit $K = 1$ wird auf einem ternären Suchbaum durchgeführt. Der Baum enthält alle Einträge des *Cory*-Korpus. Die initiale Konfiguration wird zu Beginn jeden Durchlaufs neu randomisiert mit Ausnahme des balancierten Baumes. Danach wird mit $|Q| = 15000$ der Baum durch *Move To Root*, *Simple Exchange* oder *Splay* trainiert. Zum Vergleich ist das Verhalten

[110] Dies ist notwendig, da sich beispielsweise eine Einfügung gefolgt von einer Löschung an selber Stelle wieder aufheben kann. Für ein komplexeres Modell, das auch Levenshtein-Gewichte berücksichtigt, s. Oommen und Kashyap (1998).

in den untrainierten Strukturen eines balancierten und eines randomisierten Baumes dargestellt. Angegeben sind die durch 100 Durchläufe und 15000 Anfragen gemittelten Werte. Die Wahrscheinlichkeitsverteilung im Training wie in der approximativen Suche entspricht der Wortfrequenzverteilung im Text. Im Fall des balancierten Baumes wird nur ein Lauf durchgeführt. In einem Vergleichsexperiment (G2) wird der selbe Versuch mit dem *Moby*-Korpus in 10 Durchläufen durchgeführt.

	Cory	Moby
Balance	$\approx$ 3,216	$\approx$ 3,928
Random	$\approx$ 3,205	$\approx$ 3,878
Move To Root	$\approx$ 2,868	$\approx$ 3,749
Splay	$\approx$ 2,83	$\approx$ 3,858
Simple Exchange	$\approx$ 2,632	$\approx$ 3,899

Abbildung 5.3: Experiment G/G2 – Durchschnittliche Positionierung des korrekten Begriffs in der Kandidatenliste bei einer *Similarity Range Query* mit $k = 1$

Im Fall einer *Similarity Range Query* ist die Geschwindigkeit aller Datenstrukturen in Bezug auf die durchschnittliche Anzahl Knotenzugriffe identisch.[111] Die Güte, die in der durchschnittlichen Positionierung des korrekten Wortes in den Kandidatenlisten gemessen wird (s. Abb. 5.3), spricht hingegen für die selbstorganisierenden Datenstrukturen.

Während der balancierte wie der zufällige Baum eine deutlich schlechtere Positionierung des gesuchten Wortes in der Kandidatenliste aufweist, ist die Positionierung im *Splay*-Baum wie im mit *Move To Root* restrukturierten Baum leicht besser. Eine weitere Verbesserung der Positionierung gibt *Simple Exchange* zurück. Während in einigen Szenarien der höhere Grad an lokaler Adaptivität für *Splay* und *Move To Root* spricht, zeigt sich hier der Vorteil des *Simple Exchange*: Die Datenstruktur ist asymptotisch der optimalen Ordnung näher, da eine Anfrage nicht die gesamte Konfiguration verändert, wie dies beim *Splaying* und *Move To Root* der Fall ist. Allerdings sind die Werte im Vergleichsexperiment G2 weit weniger eindeutig. Zwar sorgen die selbstorganisierten Datenstrukturen für eine leicht bessere Positionierung in der Kandidatenliste als der balancierte Baum, der randomisierte Baum hingegen zeigt ein Verhalten, das dem „Sieger“ des Hauptexperiments *Simple Exchange* überlegen ist.

[111] Durchschnittlich waren bei allen Datenstrukturen für *Cory* $\approx$ 155,224 Knotenvergleiche für ein zerstörtes Wort erforderlich – im Vergleichskorpus *Moby* $\approx$ 164,357.

Experiment H

Eine *Similarity Range Query* mit $k = 2$ und einem *noisy file* mit $K = 2$ wird auf einem ternären Suchbaum durchgeführt. Der Baum enthält alle Einträge des *Cory*-Korpus. Die initiale Konfiguration wird zu Beginn jeden Durchlaufs neu randomisiert mit Ausnahme des balancierten Baumes. Danach wird mit $|Q| = 15000$ der Baum durch *Move To Root*, *Simple Exchange* oder *Splay* trainiert. Zum Vergleich ist das Verhalten in den untrainierten Strukturen eines balancierten und eines randomisierten Baumes dargestellt. Angegeben sind die durch 10 Durchläufe und 15000 Anfragen gemittelten Werte. Die Wahrscheinlichkeitsverteilung im Training wie in der approximativen Suche entspricht der Wortfrequenzverteilung im Text. Im Fall des balancierten Baumes wird nur ein Lauf durchgeführt.

Balance	$\approx$ 48,136
Random	$\approx$ 46,505
Move To Root	$\approx$ 41,788
Splay	$\approx$ 36,485
Simple Exchange	$\approx$ 38,851

Abbildung 5.4: Experiment H – Durchschnittliche Positionierung des korrekten Begriffs in der Kandidatenliste bei einer *Similarity Range Query* mit $k = 2$

Auch bei einem stärker zerstörten Datum sind die selbstorganisierenden Strukturen den undynamischen Strukturen überlegen. Doch wie zu sehen ist (s. Abb. 5.4), sind *Similarity Range Queries* mit $k > 1$ nur in wenigen Anwendungsbereichen nützlich, da die Kandidatenliste sehr umfangreich ist[112] und somit eine Ergebnismenge generiert wird, die allenfalls für eine weitere *feine* Suche geeignet ist und bei der die leicht bessere Positionierung durch die selbstorganisierenden Heuristiken kaum ins Gewicht fällt. Durchschnittlich waren für alle Datenstrukturen $\approx$ 1484,761 Knotenzugriffe erforderlich. Die Suche verlangsamte sich also um den Faktor 10 gegenüber der Suche mit $k = 1$.

Anstelle einer *Similarity Range Query* lässt sich ein *Best Match* Verfahren auch mit größerem initialen k durchführen.

[112] Im Fall dieser Untersuchung hat das Wort „ae“ (ein zweifach zerstörtes „the“) beispielsweise 1303 Wörter im Lexikon mit dem Abstand $k \leq 2$.

Experiment I

Eine *Best Match* Suche mit $k = 100$ und einem *noisy file* mit $K = 1$ wird auf einem ternären Suchbaum durchgeführt. Der Baum enthält alle Einträge des *Cory*-Korpus. Zunächst wird der Baum mit $|Q| = 15000$ durch *Move To Root*, *Simple Exchange* oder *Splay* trainiert. Zum Vergleich ist das Verhalten in einem untrainierten, balancierten Baum dargestellt. Die initiale Konfiguration wird zu Beginn jeden Durchlaufs neu randomisiert mit Ausnahme des balancierten Baumes. Angegeben sind die durch 10 Durchläufe und 15000 Anfragen gemittelten Werte. Die Wahrscheinlichkeitsverteilung im Training wie in der approximativen Suche entspricht der Wortfrequenzverteilung im Text. Im Fall des balancierten Baumes wird nur ein Lauf durchgeführt.

	Position	Knotenzugriffe
Balance	$\approx$ 2,746	$\approx$ 2289,658
Move To Root	$\approx$ 2,447	$\approx$ 2042,407
Splay	$\approx$ 2,412	$\approx$ 2062,309
Simple Exchange	$\approx$ 2,26	$\approx$ 2028,997

Abbildung 5.5: Experiment I – Durchschnittliche Positionierung des korrekten Begriffs in der Kandidatenliste und durchschnittliche Anzahl der notwendigen Knotenzugriffe bei einer *Best Match* Suche mit $K = 1$ und $k = 100$

Wie in Tabelle 5.5 zu sehen, sind auch im Fall der *Best Match* Suche die selbstorganisierenden Strukturen in Bezug auf die Güte der undynamischen Struktur überlegen, insbesondere *Simple Exchange*, wenn die Werte auch nicht so eindeutig sind wie im Fall der *Similarity Range Query* in Experiment G.[113] Auch in Bezug auf die Geschwindigkeit ist ein leichter, positiver Effekt festzustellen, der darauf hindeutet, dass das schnellere Finden des korrekten Wortes für ein früheres Sinken von k sorgt.

Experiment J

Eine *Best Match* Suche mit $k = 100$ und einem *noisy file* mit $K = 2$ wird auf einem ternären Suchbaum durchgeführt. Der Baum enthält alle Einträge des *Cory*-Korpus. Zunächst wird der Baum mit $|Q| = 15000$ durch *Move To Root*, *Simple Exchange* oder *Splay* trainiert. Zum Vergleich ist das Verhalten in einem untrainierten, balancierten

[113] In allen Bewertungen wurden nur die Ergebnisse berücksichtigt, in denen das korrekte Wort Teil der Ergebnisliste war. Dies war in allen Datenstrukturen 964 Mal nicht der Fall, d. h. in diesen Fällen war das zerstörte Wort Teil des Lexikons. Aus diesem Grund ist das Verhalten von *Balance* auch nicht mit dem Verhalten in Experiment G identisch.

Baum dargestellt. Die initiale Konfiguration wird zu Beginn jeden Durchlaufs neu randomisiert mit Ausnahme des balancierten Baumes. Angegeben sind die durch 10 Durchläufe und 15000 Anfragen gemittelten Werte. Die Wahrscheinlichkeitsverteilung im Training wie in der approximativen Suche entspricht der Wortfrequenzverteilung im Text. Im Fall des balancierten Baumes wird nur ein Lauf durchgeführt.

	Position	Knotenzugriffe
Balance	$\approx$ 6,944	$\approx$ 4661.426
Move To Root	$\approx$ 6,123	$\approx$ 4446.947
Splay	$\approx$ 6,226	$\approx$ 4450.551
Simple Exchange	$\approx$ 5,736	$\approx$ 4338.398

Abbildung 5.6: Experiment J – Durchschnittliche Positionierung des korrekten Begriffs in der Kandidatenliste und durchschnittliche Anzahl der notwendigen Knotenzugriffe bei einer *Best Match* Suche mit $K = 2$ und $k = 100$

Erneut sind die selbstorganisierenden Datenstrukturen in Bezug auf die Güte dem balancierten Baum überlegen – und erneut liefert *Simple Exchange* das beste Ergebnis.[114] Die durchschnittliche Anzahl der Knotenzugriffe ist beim balancierten Baum geringfügig höher.

5.2.1 Bewertung

Selbstorganisierende Heuristiken können, wie gezeigt, einen positiven Einfluß auf die Güte der approximativen Suche in Lexika haben, da sie die Datenstruktur nach Wahrscheinlichkeiten von Anfragen sortieren. In Bezug auf die Geschwindigkeit sorgt das frühere Finden des gesuchten Wortes im *Best Match* Verfahren ebenfalls für leicht bessere Ergebnisse gegenüber der statischen Datenstruktur.

Der positive Effekt auf die Güte scheint abhängig von den analysierten Daten, wie Experiment G2 zeigt. Für aussagekräftigere Ergebnisse diesbezüglich müssten die beschriebenen Experimente mit weiteren, unterschiedlichen Korpora durchgeführt werden. Zudem müssten alle Experimente mit größeren zerstörten Daten gemäß der Wortfrequenz in den Texten angefragt werden. Insbesondere bei größerem K in der *Best Match* Suche ist aufgrund des reduzierten Vorkommens der korrekten Schreibweise in den Ergebnislisten eine geringere Aussagekraft für diese Form der Güte-

[114] Die Datengrundlage ist in diesem Fall jedoch stark geschmälert, da 6205 Ergebnisse nicht in die Bewertung einflossen – das korrekte Wort fand sich nicht in der Ergebnisliste, d. h. es existierte mindestens ein Wort im Lexikon, das zu dem zerstörten String einen Abstand ≤ 1 besaß.

bewertung gegeben. Aufgrund des erheblichen Evaluationsaufwands war dies im Rahmen der hier vorgestellten Arbeit nicht möglich.

5.3 Weitere Ansätze

Ein Nachteil des Verfahrens nach Shang und Merrettal (1996) ist, dass bei jedem neuen Knotenzugriff eine Reihe der Levenshtein-Matrix berechnet werden muss. Schulz und Mihov (2002) verzichten auf diese Berechnungen, indem statt der Nutzung des *Dynamic-Programming*-Ansatzes alle möglichen Editieroperationen auf den Anfragestring x in Form einer Konstruktion eines deterministischen endlichen Automaten nach Ukkonen (1985b) durchgeführt werden,[115] dessen Sprache alle Strings $y \in \Sigma^*$ enthält, die im Abstand zu maximal k Editieroperationen zu x existieren, also $D_{Lev}(x, y) \leq k$. Diese Vorverarbeitung ist besonders bei häufigem Mustervergleich von Vorteil, wie er bei der Suche in Lexika vorkommt. Zur Berechnung der Kandidatenliste wird dabei eine Paralleltraversierung des Lexikon- und des Levenshtein-Automaten durchgeführt.

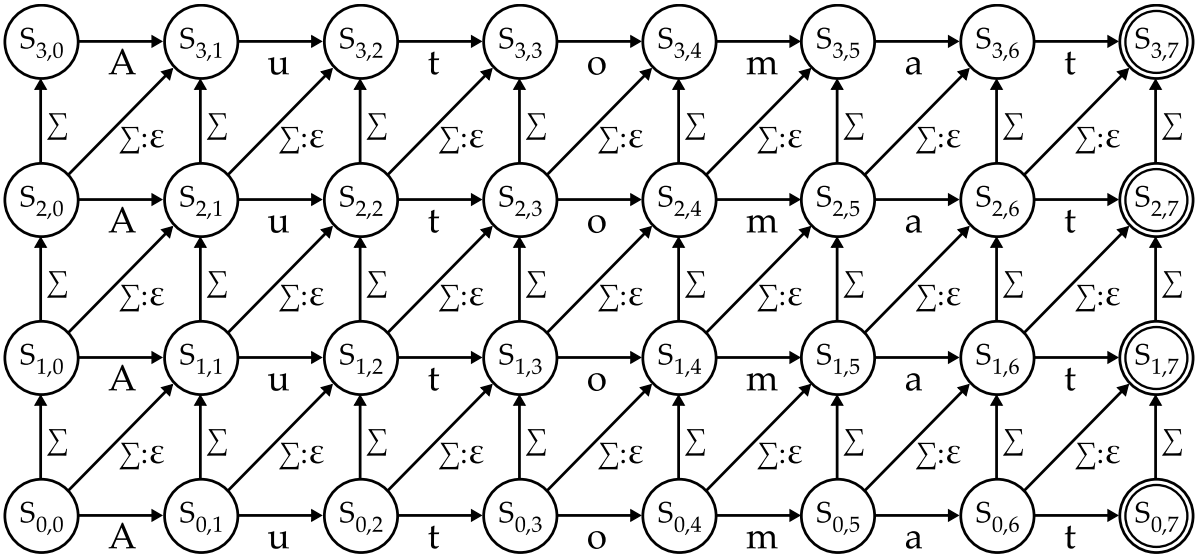


Abbildung 5.7: Levenshtein-Automat für die Erkennung von Zeichenketten im Abstand $k \leq 3$ zu dem String „Automat“

Der Automat wird in Stockwerken organisiert, wobei jeder Aufstieg in eine höhere Etage einer der drei Levenshtein-Editieroperationen entspricht. Die Anzahl der

[115] Berechnet werden kann dadurch D_{Lev} – Transpositionen, wie in D_{LevDam}, werden nicht unterstützt.

konstruierten Etagen bedingt also die mögliche maximale Anzahl an Editieroperationen,[116] weshalb Levenshtein-Automaten in erster Linie in Anwendungen mit vorgegebenem maximalen k eingesetzt werden, also für *Similarity Range Queries*.

Bunke (1995) stellt eine andere Form eines Levenshtein-Automaten vor,[117] der keine Vorverarbeitung des Suchbegriffes erfordert. Dabei werden alle Wörter des Lexikons gemäß obigem Ansatz vorbereitet und miteinander zu einem Automaten unifiziert. Auf diese Weise ist die Zeitkomplexität für die Suche nach den nächsten Strings eines Anfragewortes linear der Länge des Suchbegriffes – Der Speicherbedarf des Lexikons ist allerdings exponentiell in Bezug auf sowohl die Anzahl und die Länge der Lexikoneinträge als auch den Umfang des verwendeten Alphabets. Zudem sind die Operationen zur Einfügung und Löschung von Einträgen sehr aufwändig. Daher bezeichnet Bunke die Motivation für die Entwicklung der Datenstruktur und des Suchalgorithmus als rein theoretischer Natur.[118]

[116] $k + 1$ Stockwerke entspricht maximal k Editieroperationen. Somit ist die Begrenzung nach oben gleichbedeutend mit *Oflazers Cut-Off* im *Dynamic-Programming*-Ansatz.
[117] *Nearest neighbour string edit distance computation finite state automaton*, NN-SED-Automat.
[118] Bunke (1995), S. 87.

6 Zusammenfassung

In den vorangegangenen Kapiteln wurde der approximative Stringvergleich in Tries unter der Verwendung selbstorganisierender Heuristiken betrachtet. Motivation war die Anwendung in Rechtschreibkorrekturprogrammen, die zum einen schnelle, exakte Suche in Lexika zur Fehlerdetektion erfordern und zum anderen eine hohe Güte bei der Generierung von Vorschlagslisten für Nichtwörter. Ausgenutzt wurde dabei der Charakter einer Anfragesequenz über einen Text mit schiefer Wortfrequenzverteilung.

In Kapitel 2 wurden verschiedene Varianten für die elektronische Repräsentation solcher Lexika begutachtet und in Kapitel 3 im Vergleich einer Selbstorganisation bei schiefer Anfragesequenz unterzogen. Hierbei erwies sich der ternäre Suchbaum als überlegen und eine Selbstorganisation bei schiefer Verteilung als lohnend.

In Kapitel 4 wurden Varianten des unscharfen Stringvergleichs vorgestellt, von der die Levenshtein-Damerau-Metrik für die Versuche zur approximativen Suche in Lexika ausgewählt wurde. Diese Experimente in Kapitel 5 zeigten, dass selbstorganisierende Lexika einen positiven Effekt sowohl auf die Güte als auch die Geschwindigkeit der approximativen Suche haben können.

Offen bleibt, inwieweit sich diese Ergebnisse auf einen praktischen Einsatz übertragen lassen. Da sich die Bewertung der Geschwindigkeit auf die Anzahl benötigter Knotenzugriffe in den Datenstrukturen beschränkt, wurde der komputationelle Aufwand der Selbstorganisation ignoriert, der bei einigen Heuristiken erheblich sein kann – und absolut gemessen zu einer Geschwindigkeitseinbuße führen könnte.

Außerdem sind die positiven Effekte abhängig von zwei Voraussetzungen: Die Anfragesequenzen müssen lang sein und einer schiefen Verteilung unterliegen. Dies ist nicht in allen Einsatzgebieten der Rechtschreibkorrektur der Fall.

In weiteren Experimenten wäre daher zu untersuchen, ob sich das vorgestellte Verfahren in ungünstigeren Szenarien deutlich schlechter verhält als die Suche in statischen Datenstrukturen: Bei größeren Lexika, kürzeren Anfragesequenzen und weniger schiefer Verteilung.

Die generelle Konkurrenzfähigkeit dieses Verfahrens gegenüber traditioneller, zweistufiger Ansätze würde zudem von einer effektiven Implementierung abhängen. Systeme, die Levenshtein-Automaten einsetzen, haben in den vergangenen Jahren in diesem Bereich gute Ergebnisse erzielt und wären in der Lage, auch auf den hier vorgestellten, selbstorganisierenden Datenstrukturen zu operieren.

A Korpora

Cory: Der Roman „Little Brother“ (2008) von Cory Doctorow steht unter einer Creative Commons[119] Lizenz zum freien Download[120] bereit.

Moby: „Moby Dick, or, The Whale“ (1851) von Herman Melville wurde im Rahmen des „Project Gutenberg“[121] digitalisiert und steht zum freien Download zur Verfügung.[122]

Wiki: „Wiki“ enthält 50000 Token die gleichmäßig verteilt einer frequenzsortierten Liste aller Token der deutschen Wikipedia[123] vom 27.7.2007 entnommen sind.[124]

	Cory	**Moby**	**Wiki**
Wörter im Text:	113354	212548	—
Wörter im Lexikon:	10853	19242	50000
Minimale Wortlänge:	1	1	1
Maximale Wortlänge:[125]	28	20	35
Durchschnittliche Wortlänge im Text:	$\approx$ 4,197	$\approx$ 4,417	—
Durchschnittliche Wortlänge im Lexikon:	$\approx$ 6,706	$\approx$ 7,245	$\approx$ 8,241
Alphabetsumfang ($\lvert\Sigma\rvert$):	64	64	382

[119] Attribution-Noncommercial-Share Alike 3.0 Unported; `http://creativecommons.org/licenses/by-nc-sa/3.0/`.

[120] `http://craphound.com/littlebrother/download/`; zuletzt abgerufen: 21.6.2008.

[121] `http://www.gutenberg.org/`.

[122] `http://www.gutenberg.org/etext/2701`; zuletzt abgerufen: 4.7.2008.

[123] `http://de.wikipedia.org/`.

[124] Die Daten wurden freundlicherweise von Dr. Alexander Mehler und seinen Mitarbeitern zur Verfügung gestellt. Eine lemmatisierte Form der Daten ist frei verfügbar unter: `http://ariadne.coli.uni-bielefeld.de/indogram/component/option,com_vfm/Itemid,33/do,view/file,Corpora|Wikipedia|wikipedia_de_2007-07-27_lemma.txt/`; zuletzt abgerufen: 4.7.2008.

[125] Die längsten Wörter in den Korpora sind: „Little_Brother_Readers_Guide“ (*Cory*), „uninterpenetratingly“ (*Moby*) und „Dreifurchenpollen-Zweikeimblättrige“ (*Wiki*).

B Anmerkungen

Die für die Evaluation erforderlichen Programme wurden prototypisch in Perl implementiert. Da die Auswertung der absoluten Geschwindigkeit kein Evaluationsziel darstellte, wurde auf eine Übertragung zeitkritischer Programmteile in effizientere Sprachen verzichtet. Diese Programme, sowie alle Korpus- und Evaluationsdaten sind auf der beiliegenden DVD zu finden.

Die Abbildung auf Seite 41 ist Russel (1918) entnommen.

C Algorithmen

Prozedur `moveToFrontList(`*y*,*h*`)`

Input : y ist ein Listeneintrag; h ist der Listenkopf

```
1  prev(θ), next(θ) := θ, θ
2  p := prev(y)
3  n := next(y)
4  if p ≠ θ then
5      next(p) := n
6      if n ≠ θ then
7          prev(n) := p
8      prev(h) := y
9      next(y) := h
10     prev(y) := θ
11     h := y
```

Prozedur `transposeList(`y,h`)`

Input : y ist ein Listeneintrag; h ist der Listenkopf

```
1  prev(θ), next(θ) := θ, θ
2  p := prev(y)
3  n := next(y)
4  if p ≠ θ then
5      next(p) := n
6      if n ≠ θ then
7          prev(n) := p
8      next(y) := p
9      prev(y) := prev(p)
10     if prev(p) ≠ θ then
11         next(prev(p)) := y
12         prev(p) := y
13     else
14         prev(y) := θ
15         h:= y
16         prev(p) := h
```

Prozedur `frequencyCountList(`y,h`)`

Input : y ist ein Listeneintrag; h ist der Listenkopf

```
1  prev(θ), next(θ), count(θ) := θ, θ, θ
2  count(y) := count(y) + 1
3  p := prev(y)
4  n := next(y)
5  if p ≠ θ then
6      while p ≠ θ and count(p) ≤ count(y) do
7          transposeList(y,h)
8          p := prev(y)
9          n := next(y)
```

Prozedur `moveToRoot(`y`)`

Input : y ist ein Baumknoten

```
1  while p(y) ≠ θ do
2      simpleRotate(y)
```

Prozedur `simpleExchange(`y`)`

Input : y ist ein Baumknoten

```
1  simpleRotate(y)
```

Prozedur `simpleRotate(`y`)`

Input : y ist ein Baumknoten

```
1  z := p(y)
2  x := g(y)
3  if z ≠ θ then
4      if y = left(z) then
5          c := right(y)
6          right(y) := z
7      else if y = right(z) then
8          c := left(y)
9          left(y) := z
10     g(y) := y
11     if y = left(z) then
12         left(z) := c
13     else if y = right(z) then
14         right(z) := c
15     p(c) := z
16 if x ≠ θ then
17     if z = left(x) then
18         left(x) := y
19     else if z = right(x) then
20         right(x) := y
```

Prozedur `splay(`x`)`

```
   Input : x ist ein Baumknoten
 1 p(θ), left(θ), right(θ) := θ, θ, θ
 2 while p(x) ≠ θ do
 3     if x = left(p(x)) then
 4         if g(x) = θ;                                   // Zig
 5         then
 6             rotateRight(p(x))
 7         else if p(x) = left(g(x));                     // Zig-Zig
 8         then
 9             rotateRight(g(x))
10             rotateRight(p(x))
11         else if p(x) = right(g(x));                    // Zig-Zag
12         then
13             rotateRight(p(x))
14             rotateLeft(p(x))
15     else if x = right(p(x));                           // Zig
16     then
17         if g(x) = θ then
18             rotateLeft(p(x))
19         else if p(x) = right(g(x));                    // Zig-Zig
20         then
21             rotateLeft(g(x))
22             rotateLeft(p(x))
23         else if p(x) = left(g(x));                     // Zig-Zag
24         then
25             rotateLeft(p(x))
26             rotateRight(p(x))
```

Prozedur `rotateLeft(`y`)`

Input : y ist ein Baumknoten

```
1  x := right(y)
2  z := p(y)
3  if left(z) = y then
4      left(z) := x
5  else if right(z) = y then
6      right(z) := x
7  left(x), right(y) := y, left(x)
8  p(x), p(y) := z, x
9  if right(y) ≠ θ then
10     p(right(y)) := y
```

Prozedur `rotateRight(`y`)`

Input : y ist ein Baumknoten

```
1  x := left(y)
2  z := p(y)
3  if right(z) = y then
4      right(z) := x
5  else if left(z) = y then
6      left(z) := x
7  right(x), left(y) := y, right(x)
8  p(x), p(y) := z, x
9  if left(y) ≠ θ then
10     p(left(y)) := y
```

Prozedur `g(`x`)`

Input : x ist ein Baumknoten
Output : g ist ein Baumknoten

```
1  g := p(p(x))
```

Prozedur `moveToRootTST(`y`)`

Input : y ist ein Baumknoten

```
1  while p(y) ≠ θ do
2      if y = down(p(y)) then
3          y := p(y)
4      else
5          simpleRotateTST(y);
```

Prozedur `simpleExchangeTST(`y`)`

```
   Input : y ist ein Baumknoten
1  x := 1
2  while p(y) ≠ θ do
3      if y = down(p(y)) then
4          y := p(y)
5          x := 1
6      else if x ≠ 0 then
7          simpleRotateTST(y); x := 0
8      else
9          y := p(y)
```

Prozedur `simpleRotateTST(`y`)`

```
   Input : y ist ein Baumknoten
1  z := p(y)
2  x := g(y)
3  if z ≠ θ then
4      if y = left(z) then
5          c := right(y)
6          right(y) := z
7      else if y = right(z) then
8          c := left(y)
9          left(y) := z
10     g(y) := y
11     if y = left(z) then
12         left(z) := c
13     else if y = right(z) then
14         right(z) := c
15     p(c) := z
16 if x ≠ θ then
17     if ) then z = left(x
18         left(x) := y
19     else if z = right(x) then
20         right(x) := y
21     else if z = down(x) then
22         down(x) := y
```

Prozedur `splayTST(x)`

```
while p(x) ≠ θ do
    if x = down(p(x)) then
        x := p(x)
    else if x = left(p(x)); // Zig
    then
        if g(x) = θ then
            rotateRightTST(p(x))
        else if p(x) = left(g(x)); // Zig-Zig
        then
            rotateRightTST(g(x))
            rotateRightTST(p(x))
        else if p(x) = right(g(x)); // Zig-Zag
        then
            rotateRightTST(p(x))
            rotateLeftTST(p(x))
        else if p(x) = down(g(x)); // Zig
        then
            rotateRightTST(p(x))
            x := p(x)
    else if x = right(p(x)); // Zig
    then
        if g(x) = θ then
            rotateLeftTST(p(x))
        else if p(x) = right(g(x)); // Zig-Zig
        then
            rotateLeftTST(g(x))
            rotateLeftTST(p(x))
        else if p(x) = left(g(x)); // Zig-Zag
        then
            rotateLeftTST(p(x))
            rotateRightTST(p(x))
        else if p(x) = down(g(x)); // Zig
        then
            rotateLeftTST(p(x))
            x := p(x)
```

Prozedur `rotateLeftTST(`y`)`

```
   Input : y ist ein Baumknoten
 1 x := right(y)
 2 z := p(y)
 3 if left(z) = y then
 4 |   left(z) := x
 5 else if right(z) = y then
 6 |   right(z) := x
 7 else if down(z) = y then
 8 |   down(z) := x
 9 left(x), right(y) := y, left(x)
10 p(x), p(y) := z, x
11 if right(y) ≠ θ then
12 |   p(right(y)) := y
```

Prozedur `rotateRightTST(`y`)`

```
   Input : y ist ein Baumknoten
 1 x := left(y)
 2 z := p(y)
 3 if right(z) = y then
 4 |   right(z) := x
 5 else if left(z) = y then
 6 |   left(z) := x
 7 else if down(z) = y then
 8 |   down(z) := x
 9 right(x), left(y) := y, right(x)
10 p(x), p(y) := z, x
11 if left(y) ≠ θ then
12 |   p(left(y)) := y
```

Prozedur `levenshtein(x,y)`

Input : x ist ein Zeichenkette
Input : y ist ein Zeichenkette
Output : z ist die Levenshtein-Distanz von x und y

```
1  d_{0,0} := 0
2  for i := 0 to |x| do
3      d_{i,0} := d_{i-1,0} + 1
4  for j := 0 to |y| do
5      d_{0,j} := d_{0,j-1} + 1
6  for i := 1 to |x| do
7      for j := 1 to |y| do
8          c := 1
9          if x_{i-1} = y_{j-1} then
10             c := 0
11         d_{i,j} = min(d_{i-1,j} + 1, d_{i,j-1} + 1, d_{i-1,j-1} + c)
12 z := d_{|x|,|y|}
```

Prozedur `levenshteinDamerau(x,y)`

Input : x ist ein Zeichenkette
Input : y ist ein Zeichenkette
Output : z ist die Levenshtein-Damerau-Distanz von x und y

```
1  d_{0,0} := 0
2  for i := 0 to |x| do
3      d_{i,0} := d_{i-1,0} + γ(x_i → ε)
4  for j := 0 to |y| do
5      d_{0,j} := d_{0,j-1} + γ(ε → y_j)
6  for i := 1 to |x| do
7      for j := 1 to |y| do
8          c := 1
9          if x_{i-1} = y_{j-1} then
10             c := 0
11         d_{i,j} = min(d_{i-1,j} + 1, d_{i,j-1} + 1, d_{i-1,j-1} + c)
12         if i > 1 and j > 1 and x_{i-1} = y_{j-2} and x_{i-2} = y_{j-1} then
13             d_{i,j} = min(d_{i,j}, d_{i-2,j-2}+ c)
14 z := d_{|x|,|y|}
```

Prozedur `DFSearchTST(`y`)`

Data : d ist die Matrix, x ist der Eingabestring, z ist der Ausgabestring, k ist der Grenzwert

Input : y ist ein Baumknoten

```
1  if cuted(y) ≤ k then
2      if down(y) = θ then
3          if distance(y) ≤ k then
4              out z_{1···(level(y)−1)}
5      else
6          DFSearch(down(y))
7  if left(y) ≠ θ then
8      DFSearch(left(y))
9  if right(y) ≠ θ then
10     DFSearch(right(y))
```

Prozedur `cuted(`y`)`

Data : d ist die Matrix, x ist der Eingabestring, z ist der Ausgabestring, k ist der Grenzwert
Input : y ist ein Baumknoten
Output : ce ist die *Cut-off*-Distanz

```
1  l := level(y)
2  d_{l,0} := l
3  z_{l-1} := char(y)
4  for i := 1 to |x| + 1 do
5      c := 1
6      if x_{i-1} = char(y) then
7          c := 0
8      d_{l,i} = min(d_{l-1,i} + 1, d_{l,i-1} + 1, d_{l-1,i-1} + c)
9      if i > 1 and l > 1 and x_{i-1} = z_{l-2} and x_{i-2} = z_{l-1} then
10         d_{l,i} = min(d_{l,i}, d_{l-2,i-2} + c)
11 ce := min(d_{l,max(1,l-k)} ... min(|x|,l+k))
```

Prozedur `distance(`y`)`

Data : d ist die Matrix, x ist der Eingabestring
Input : y ist ein Baumknoten
Output : ed ist die Editierdistanz

```
1  ed := d_{level(y),|x|+1}
```

Prozedur `genZipf(`a`)`

Input : a ist ein Parameter, der > 1 sein muss
Output : x ist die generierte Zufallsvariable

```
1  b := 2^{a-1}
2  repeat
3      U := rand(0,1)
4      V := rand(0,1)
5      X := ⌊U^{-1/(a-1)}⌋
6      T := 1 + 1/X
7  until V · X · (T-1)/(b-1) ≤ T/b
```

Abbildungsverzeichnis

2.1 Trie als abstrakte Datenstruktur 9
2.2 Fredkin-Trie 11
2.3 Anzahl augehender Kanten 12
2.4 Verkettete Liste 14
2.5 Briandais-Trie 14
2.6 Binärer Suchbaum 15
2.7 Ternary Search Tree 16

3.1 Move To Front 23
3.2 Transpose 23
3.3 Experiment A 24
3.4 Experiment A – Werte 25
3.5 Experiment B – 1 27
3.6 Experiment B – Werte 28
3.7 Experiment B – 2 29
3.8 Simple Exchange und Move To Root 30
3.9 Rotate 31
3.10 Splaying 32
3.11 Zig, Zig-Zig und Zig-Zag 33
3.12 Experiment C 34
3.13 Experiment C/C2 – Werte 34
3.14 Experiment D 35
3.15 Splaying 36
3.16 RotateTST 37
3.17 Experiment E 38
3.18 Experiment E/E2 – Werte 38
3.19 Experiment F – Werte 39

4.1 SOUNDEX-Zeichenklassen 41

4.2 *n*-Gramm 43
4.3 Hamming-Distanz 45
4.4 Hamming-Würfel 45
4.5 Editier-Transkripte 48
4.6 Matrix zur Berechnung der Levenshtein-Distanz 49
4.7 Ukkonens Cut-Off 50
4.8 Matrix zur Berechnung der Levenshtein-Damerau-Distanz 51
4.9 EDITEX-Zeichenklassen 52
4.10 Tastennähe in QWERTZ-Layout 52

5.1 Approximative Suche im Trie 58
5.2 Experiment G 61
5.3 Experiment G/G2 – Werte 62
5.4 Experiment H – Werte 63
5.5 Experiment I – Werte 64
5.6 Experiment J – Werte 65
5.7 Levenshtein-Automat 66

Liste der Algorithmen

1 moveToFrontList(y,h) . . . 72
2 transposeList(y,h) . . . 73
3 frequencyCountList(y,h) . . . 73
4 moveToRoot(y) . . . 73
5 simpleExchange(y) . . . 73
6 simpleRotate(y) . . . 74
7 splay(x) . . . 75
8 rotateLeft(y) . . . 76
9 rotateRight(y) . . . 76
10 g(x) . . . 76
11 moveToRootTST(y) . . . 76
12 simpleExchangeTST(y) . . . 77
13 simpleRotateTST(y) . . . 77
14 splayTST(x) . . . 78
15 rotateLeftTST(y) . . . 79
16 rotateRightTST(y) . . . 79
17 levenshtein(x,y) . . . 80
18 levenshteinDamerau(x,y) . . . 80
19 DFSearchTST(y) . . . 81
20 cuted(y) . . . 82
21 distance(y) . . . 82
22 genZipf(a) . . . 82

Literaturverzeichnis

Adel'son-Vel'skiĭ, Georgii Maximovich und Yevgeniy Mikhailovich Landis (1962): *An algorithm for the organization of information*. In: *Proceedings of the USSR Academy of Sciences*, Ausgabe 146. S. 263–266. Englisch in: Soviet Mathematics Doklady, 3, S. 1259–1263 (übersetzt von Myron J. Ricci).

Albers, Susanne und Jefferey Westbrook (1998): *Self-Organizing Data Structures*. In: Amos Fiat und Gerhard J. Woeginger (Hrsg.), *Online Algorithms: The State of the Art*, Kap. 2. Springer.

Allen, Brian und Ian Munro (1978): *Self-Organizing Binary Search Trees*. In: *Journal of the Association for Computing Machinery*, 25, 4, S. 526–535.

Anderson, Edward James, P. Nash und Richard R. Weber (1982): *A Counterexample To A Conjecture On Optimal List Ordering*. In: *Journal of Applied Probability*, 19, S. 730–732.

Angell, Richard C., George E. Freund und Peter Willett (1983): *Automatic Spelling Correction Using a Trigram Similarity Measure*. In: *Information Processing & Management*, 19, 4, S. 255–261.

ATIS-T1.523-2001 (2001): *ATIS Telecom Glossary T1.523-2001*. Alliance for Telecommunications Industry Solutions (ATIS) Committee. Nachfolger von: Federal Standard 1037C (1997), U.S. Department of Commerce, National Telecommunications and Information Administration, Institute for Telecommunications Services.

Bachelard, Simon, Olivier Raynaud und Yoan Renaud (2006): *Implementing collection of sets with trie: a stepping stone for performances? Forschungsbericht*, LIMOS – Université Blaise Pascal, Campus universitaire des Cézeaux, Clermont-Ferrand, France.

Bachrach, Ran und Ran El-Yaniv (1997): *Online list Accessing Algorithms and Their Applications: Recent Empirical Evidence*. In: *Proceedings of the 8th annual ACM-SIAM symposium on Discrete algorithms (SODA)*. S. 53–62.

Badr, Ghada Hany und Basantkumar John Oommen (2005): *Self-Adjusting of Ternary Search Tries Using Conditional Rotations and Randomized Heuristics*. In: *The Computer Journal*, 48, 2, S. 200–219.

Bentley, Jon Louis und James B. Saxe (1979): *Algorithms on Vector Sets*. In: *News of the ACM Special Interest Group on Algorithms and Computation Theory (SIGACT News)*, 11, 2.

Bentley, Jon Louis und Robert Sedgewick (1997): *Fast Algorithms for Sorting and Searching Strings*. In: *Proceedings of the 8th annual ACM-SIAM symposium on Discrete algorithms (SODA)*. New Orleans, S. 360–369.

Bentley, Jon Louis und Robert Sedgewick (1998): *Ternary Search Trees*. In: *Dr. Dobb's Journal*, S. 20–25.

Bentley, Jon Louis, Daniel Dominic Kaplan Sleator, Robert Endre Tarjan und Victor Keh-Wei Wei (1986): *A Locally Adaptive Data Compression Scheme*. In: *Communications of the Association for Computing Machinery*, 29, 4, S. 320–330.

Bentley, Jon Louise und Catherine C. McGeoch (1985): *Amortized Analyses of Self-Organizing Sequential Search Heurisics*. In: *Communications of the Association for Computing Machinery*, 28, 4, S. 404–411.

Bitner, James R. (1979): *Heuristics that Dynamically Organize Data Structures*. In: *SIAM Journal on Computing*, 8, 1, S. 82–109.

Bradley, David W. und Richard A. Bradley (1983): *Application of Sequence Comparison to the Study of Bird Songs*. In: David Sankoff und Joseph B. Kruskal (Hrsg.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Kap. 6. Addison-Wesley, Reading, Massachusetts, USA, S. 189–209.

de la Briandais, Rene (1959): *File searching using variable length keys*. In: *Western Joint Computer Conference*, Ausgabe 15. Montvale, NJ, USA, S. 295–298.

Bunke, Horst (1995): *Fast Approximate Matching of Words Against a Dictionary*. In: *Computing*, 55, S. 75–89.

Burrows, Michael und David J. Wheeler (1994): *A Block-sorting Lossless Data Compression Algorithm*. *Forschungsbericht 124*, Digital Equipment Corporation.

Cheetham, Robert P., Basantkumar John Oomen und David T. H. Ng (1993): *Adaptive Structuring of Binary Search Trees Using Conditional Rotations*. In: *IEEE Transactions on Knowledge and Data Engineering*, 5, 4, S. 695–704.

Clampett, Harry A. (1964): *Randomized Binary Searching With Tree Structures*. In: *Communications of the Association for Computing Machinery*, 7, 3, S. 163–165.

Clément, Julien, Philippe Flajolet und Brigitte Vallée (1998): *The Analysis of Hybrid Trie Structures*. In: *Proceedings of the 9th annual ACM-SIAM symposium on Discrete algorithms (SODA)*. Philadelphia, S. 531–539.

Cornew, Ronald W. (1968): *A Statistical Method of Spelling Correction*. In: *Information and Control*, 12, S. 79–93.

Crochemore, Maxime und Wojciech Rytter (2002): *Jewels of Stringology. Text Algorithms*. World Scientific.

Damerau, Fred J. (1964): *A Technique for Computer Detection and Correction of Spelling Errors*. In: *Communications of the Association for Computing Machinery*, 7, 3, S. 171–176.

Devroye, Luc (1986): *Non-Uniform Random Variate Generation*. Springer, New York. Errata unter: `http://cg.scs.carleton.ca/~luc/errors.pdf`.

Doctorow, Cory (2008): *Little Brother*. Tor Teen.

Du, M. W. und S. C. Chang (1992): *A model and a fast algorithm for multiple errors spelling correction*. In: *Acta Informatica*, 29, 3, S. 281–302.

Faulk, Ramon D. (1964): *An Inductive Approach to Language Translation*. In: *Communications of the Association for Computing Machinery*, 7, 11, S. 647–653.

Fliedner, Gerhard (2004): *Korrekturprogramme*. In: Kai-Uwe Carstensen, Christian Ebert, Cornelia Endriss, Susanne Jekat, Ralf Klabunde und Hagen Langer (Hrsg.), *Computerlinguistik und Sprachtechnologie – Eine Einführung. 2., überarbeitete und erweiterte Auflage*, Kap. 5.1. Spektrum Akademischer Verlag, S. 463–470.

Fredkin, Edward (1960): *Trie Memory*. In: *Communications of the Association for Computing Machinery*, 9, 3, S. 490–499.

Glantz, Herbert T. (1957): *On the Recognition of Information With a Digital Computer*. In: *Communications of the Association for Computing Machinery*, 4, 2, S. 178–188.

Gollapudi, Sreenivas und Rina Panigrahy (2006): *A Dictionary for Approximate String Search and Longest Prefix Search*. In: *Proceedings of the 15th ACM International Conference on Information and Knowledge Management (CIKM)*. Arlington, Virginia, USA, S. 768–775.

Gusfield, Dan (1997): *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York.

Hamming, Richard W. (1950): *Error Detecting and Error Correcting Codes*. In: *The Bell System Technical Journal*, 26, 2, S. 147–160.

Hamming, Richard W. (1980): *Coding and Information Theory*. Prentice-Hall, Inc., Englewood Cliffs, N.J., USA.

Heinz, Steffen, Justin Zobel und Hugh E. Williams (2002): *Burst Tries: A Fast, Efficient Data Structure for String Keys*. In: *ACM Transactions on Information Systems*, 20, 2, S. 192–223.

Hester, James H. und Daniel S. Hirschberg (1985): *Self-Organizing Linear Search*. In: *ACM Computing Surveys*, 17, 3, S. 295–311.

Hofri, Micha und Hadas Shachnai (1991): *Self-Organizing Lists And Independent References – A Statistical Synergy*. In: *Journal of Algorithms*, 12, 4, S. 533–555. Überarbeitete Version: April 1997.

Klabunde, Ralf (2001): *Automatentheorie und Formale Sprachen*. In: Kai-Uwe Carstensen, Christian Ebert, Cornelia Endriss, Susanne Jekat, Ralf Klabunde und Hagen Langer (Hrsg.), *Computerlinguistik und Sprachtechnologie – Eine Einführung*, Kap. 2.2. Spektrum Akademischer Verlag, S. 59–86.

Knuth, Donald Ervin (1998): *Sorting and Searching*, Ausgabe 3 von *The Art of Computer Programming*. Addison-Wesley, Zweite Auflage.

Kukich, Karen (1992): *Techniques for Automatically Correcting Words in Text*. In: *ACM Computing Surveys*, 24, 4, S. 377–439.

Lambert, Bruce L., Swu-Jane Lin, Ken-Yu Chang und Sanjay K. Gandhi (1999): *Similarity As a Risk Factor in Drug-Name Confusing Errors. The Look-Alike (Orthographic) and Sound-Alike (Phonetic) Model*. In: *Medical Care*, 37, 12, S. 1214–1225.

Levenshtein, Vladimir I. (1965): *Binary codes capable of correcting deletions, insertions, and reversals*. In: *Doklady Akademii Nauk SSSR*, 163, 4, S. 845–848. Englisch in: Soviet Physics Doklady (1966), Volume 10, Nummer 8, S. 707-710.

Lowrance, Roy und Robert A. Wagner (1975): *An Extension of the String-to-String Correction Problem*. In: *Journal of the Association for Computing Machinery*, 22, 2, S. 177–183.

Martel, Charles (1991): *Self-adjusting multi-way search trees*. In: *Information Processing Letters*, 38, S. 135–141.

McCabe, John (1965): *On Serial Files with Relocatable Records*. In: *Operations Research*, S. 609–618.

Morgan, Howard L. (1970): *Spelling correction in systems programs*. In: *Communications of the Association for Computing Machinery*, 13, 2, S. 90–94.

Navarro, Gonzalo (2001): *A Guided Tour to Approximate String Matching*. In: *ACM Computing Surveys*, 33, 1, S. 31–88.

Navarro, Gonzalo und Mathieu Raffinot (2002): *Flexible Pattern Matching in Strings – Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press.

Oflazer, Kemal (1996): *Error-tolerant Finite-State Recognition with Applications to Morphological Analysis and Spelling Correction*. In: *Computational Linguistics*, 22, 1, S. 73–89.

Oommen, Basantkumar John und Rangasami Kashyap (1998): *A Formal Theory for Optimal and Information Theoretic Syntactic Pattern Recognition*. In: *Pattern Recognition*, 31, 8, S. 1159–1177.

Reffle, Uli (2005): *Varianten approximativen Matchings*. Magisterarbeit, Centrum für Informations- und Sprachwissenschaft, Ludwig-Maximilian-Universität, München.

Ringlstetter, Christoph (2003): *OCR-Korrektur und Bestimmung von Levenshtein-Gewichten*. Magisterarbeit, Centrum für Informations- und Sprachwissenschaft, Ludwig-Maximilian-Universität, München.

Rivest, Ronald (1976): *On Self-Organizing Sequential Search Heuristics*. In: *Communications of the Association for Computing Machinery*, 19, 2, S. 63–67.

Russel, Robert C. (1918): *Index*. U.S. Patent 1,261,167.

Russel, Robert C. (1922): *Index*. U.S. Patent 1,435,663.

Saake, Guenther und Kai-Uwe Sattler (2006): *Algorithmen und Datenstrukturen*. dpunkt.Verlag, Heidelberg, Dritte Auflage.

Salomon, David (2007): *Data Compression – The Complete Reference*, Kap. 1.5. Springer-Verlag, Vierte Auflage, S. 37–40.

Schulz, Klaus U. und Stoyan Mihov (2002): *Fast string correction with Levenshtein-Automata*. In: *International Journal of Document Analysis and Recognition*, 5, 1, S. 67–85.

Sedgewick, Robert (1998): *Algorithms in C++*. Addison-Wesley, Dritte Auflage.

Shang, Heping und T. H. Merrettal (1996): *Tries for Approximate String Matching*. In: *IEEE Transactions on Knowledge and Data Engineering*, 8, 4, S. 540–547.

Sherk, Murray (1995): *Self-Adjusting k-ary Search Trees*. In: *Journal of Algorithms*, 19, S. 25–44.

Sleator, Daniel Dominic Kaplan und Robert Endre Tarjan (1983): *Self-Adjusting Binary Trees*. In: *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*. Boston, Massachusetts, S. 235–245.

Sleator, Daniel Dominic Kaplan und Robert Endre Tarjan (1985a): *Amortized Efficiency of List Update and Paging Rules*. In: *Communications of the Association for Computing Machinery*, 28, 2, S. 202–208.

Sleator, Daniel Dominic Kaplan und Robert Endre Tarjan (1985b): *Self-adjusting binary search trees*. In: *Journal of the Association for Computing Machinery*, 32, 3, S. 652–686.

Sussenguth, Edward H., Jr. (1963): *Use of tree structures for processing files*. In: *Communications of the Association for Computing Machinery*, 6, 5, S. 272–279.

Tanaka, Eiichiro (1988): *A String Correction Method based on the Context-Dependent Similarity*. In: Gabriel Ferraté, Theo Pavlidis, Alberto Sanfeliu und Horst Bunke (Hrsg.), *Syntactic and Structural Pattern Recognition*, NATO ASI, Kap. 1. Springer, S. 3–17.

Tarjan, Robert Endre (1983): *Data Structures and Network Algorithms*. Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, Vierte Auflage.

Ukkonen, Esko (1985a): *Algorithms for Approximate String Matching*. In: *Information and Control*, 64, S. 100–118.

Ukkonen, Esko (1985b): *Finding approximate patterns in strings*. In: *Journal of Algorithms*, 6, 1, S. 132–137.

Ukkonen, Esko (1992): *Approximate string-matching with q-grams and maximal matches*. In: *Theoretical Computer Science*, 92, 1, S. 191–211.

Ullenboom, Christian (2006): *Java ist auch eine Insel – Programmieren für die Java 2-Plattform in der Version 5*. Galileo Press. EBook unter: `http://www.galileodesign.de/openbook/javainsel5/`; zuletzt abgerufen: 2.7.2008.

Ullmann, Julian R. (1977): *A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words*. In: *The Computer Journal*, 20, 2, S. 141–147.

Véronis, Jean (1988): *Computerized Correction of Phonographic Errors*. In: *Computers and the Humanities*, 22, S. 43–56.

Wagner, Robert A. und Michael J. Fischer (1974): *The String-to-String Correction Problem*. In: *Journal of the Association for Computing Machinery*, 21, 1, S. 168–173.

Yergeau, Francois (2003): *RFC 3629: UTF-8, a transformation format of ISO 10646*. *Forschungsbericht*, Alis Technologies.
URL `http://www.rfc-archive.org/getrfc.php?rfc=3629`

Zezula, Pavel, Giuseppe Amato, Vlastislav Dohnal und Michal Batko (2006): *Similarity Search – The Metric Space Approach*. Advances in Database Systems. Springer.

Zobel, Justin und Philip W. Dart (1995): *Finding Approximate Matches in Large Lexicons*. In: *Software – Practice and Experience*, 25, 3, S. 331–345.

Zobel, Justin und Philip W. Dart (1996): *Phonetic String Matching: Lessons from Information Retrieval*. In: *Proceedings of the 19th International Conference on Research and Development in Information Retrieval*. S. 166–172.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit selbständig verfasst und gelieferte Datensätze, Zeichnungen, Skizzen und graphische Darstellungen selbständig erstellt habe. Ich habe keine anderen Quellen als die angegebenen benutzt und habe die Stellen der Arbeit, die anderen Werken entnommen sind – einschl. verwendeter Tabellen und Abbildungen – in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht.

Bielefeld, den