Köln, den 12. März 2006

Studiengang Informationsverarbeitung
WS 2005/2006
Sprachliche Informationsverarbeitung
Hauptseminar: “Stringverarbeitung”
bei Prof. Dr. Jürgen Rolshoven

Suffixbäume und Ähnlichkeitsmaße

vorgelegt von

Fabian Steeg
Matrikelnummer 3598900
steeg@netcologne.de
Liebigstr. 43
50823 Köln

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Suffixbäume in der maschinellen Sprachverarbeitung

Gegenstand dieser Arbeit ist die Untersuchung der Anwendbarkeit von Suffixbäumen zur Ermittlung von Ähnlichkeiten im Rahmen einer Strukturanalyse in der maschinellen Sprachverarbeitung.

1.1 Suffixbäume: Eine vielseitige Datenstruktur

Suffixbäume sind eine vielseitige Datenstruktur mit vielen effizienten Anwendungsmöglichkeiten in der Stringverarbeitung.

Ein Suffixbaum T für einen String S mit m Symbolen ist ein gerichteter Baum mit m Blättern. Jede Kante ist mit einem Teilstring von S beschriftet. Jeder innere Knoten von T hat mindestens zwei Kinder, deren Kantenbeschriftungen nie mit dem gleichen Symbol beginnen. Für jedes Blatt i in T ergeben die Beschriftungen der Kanten auf dem Pfad von der Wurzel zu i aneinander gehangen das Suffix von S, das an Index i beginnt.

Somit enthält T alle Suffixe von S, wobei mehrfach auftretende Teilstrings nur einmal in T enthalten sind (siehe Abbildung 1, in dieser und allen weiteren Abbildungen von Suffixbäumen in dieser Arbeit sind die Knoten der Bäume *depth-first* durchnummeriert). Dadurch ermöglicht ein Suffixbaum die Lösung zahlreicher Probleme im Bereich der Stringverarbeitung in linear wachsender oder sogar konstanter Laufzeit. Für eine ausführliche Darstellung von Suffixbäumen siehe Gusfield ([2]1999:89ff.).

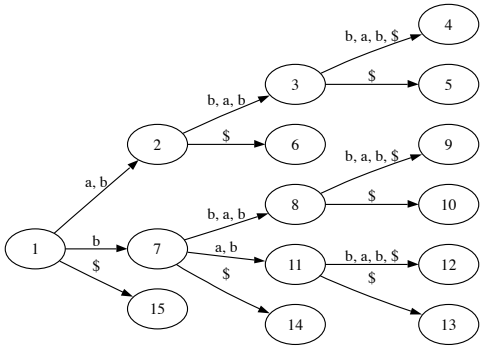

Abbildung 1: Suffixbaum für *abbabbab*

Damit erscheinen Suffixbäume durch zwei Eigenschaften interessant:

1. Suffixbäume ermöglichen die effiziente Analyse großer Korpora durch verschiedene effiziente Algorithmen.

2. Suffixbäume kodieren Struktur, der Suffixbaum fasst wiederholte Strukturen zusammen und ist damit eine Generalisierung des Eingabetextes.[1]

Das Haupteinsatzgebiet für Suffixbäume liegt traditionellerweise in der Bioinformatik (vgl. Gusfield [2]1999, Böckenhauer & Bongartz 2003), wo große Datenmengen manipuliert und analysiert werden. Anwendungen im Bereich der Verarbeitung natürlicher Sprache sind bislang deutlich weniger zu finden. Ein Gegenbeispiel ist Geertzen (2003), siehe dazu Abschnitt 3 auf Seite 6.

1.2 Morphologie: Suffixbäume für Buchstaben und Wörter

Suffixbäume zur morphologischen Analyse enthalten als Symbole die Buchstaben des entsprechenden natürlichsprachlichen Alphabets und ähneln damit den Suffixbäumen aus der Bioinformatik. Jedoch sind bei der Darstellung mehrerer Wörter Suffixe, die über Wortgrenzen hinausgehen, nicht von Belang. Hier würde daher ein *generalisierter* Suffixbaum (vgl. Gusfield [2]1999:116) für alle Types im Korpus erstellt (siehe Abbildung 2 auf Seite 3).

1.3 Syntax: Suffixbäume für Wörter und Sätze

Für die syntaktische Analyse mit Suffixbäumen müssen diese als Symbole nicht einzelne Buchstaben, sondern Wörter enthalten. Zum Vergleich von Sätzen kann dann ein wortbasierter, über mehrere Sätze generalisierter Suffixbaum erstellt werden. Ein generalisierter, wortbasierter Suffixbaum enthält Informationen über die Konstituentenstruktur der im Baum enthaltenen Sätze, denn aufgrund der Struktur des Suffixbaumes bilden die Beschriftungen der ausgehenden Kanten jedes inneren Knotens Paradigmen (siehe Abbildung 3 auf Seite 3, mehr dazu in Abschnitt 3 auf Seite 6).

[1] Ein Verfahren zum Lernen von Grammatiken, das auf diesem Zusammenhang zwischen Kompression und Generalisierung sowie bayesscher Schätzung basiert, ist das Prinzip der *minimum description length* (MDL). Geertzen (2003:20f.) umreißt das Verfahren und gibt weitergehende Literaturhinweise.

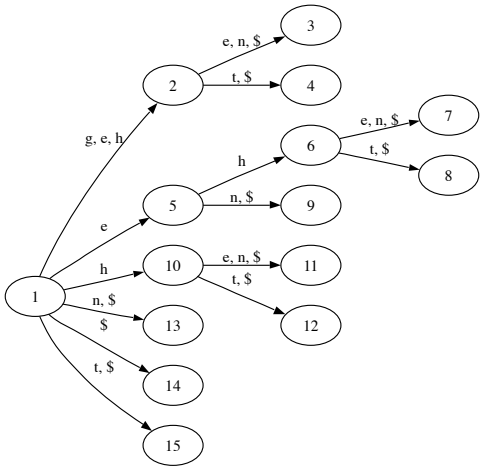

Abbildung 2: Generalisierter, zeichenbasierter Suffixbaum für *gehen geht*

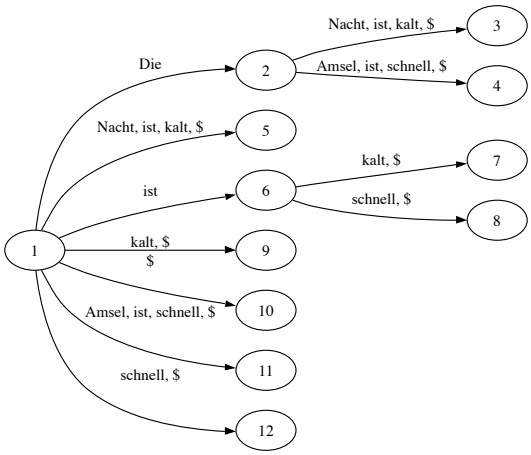

Abbildung 3: Generalisierter, wortbasierter Suffixbaum für *Die Nacht ist kalt. Die Amsel ist schnell.*

2 Laufzeit und Speicherplatzbedarf

Traditionelle Algorithmen zur Erstellung von Suffixbäumen basieren auf der Tatsache, dass *alle* Suffixe des Eingabestrings eingefügt werden und nutzen aus, dass das Eingabealphabet sehr klein ist (Andersson et al. 1999).

Das Einfügen aller Suffixe ist jedoch für eine Anwendung zur syntaktischen Analyse weder nötig noch erwünscht, da lediglich solche Suffixe eingefügt werden sollen, die an Wortgrenzen beginnen. Zudem ist hier das Alphabet viel größer, es umfasst alle unterschiedlichen Wörter *m'* (Types) in der Menge aller Wörter m (Token). Das bedeutet zum einen, dass die Effizienz der traditionellen Algorithmen nicht ohne weiteres übertragbar ist auf die Erstellung von wortbasierten Suffixbäumen, und zum anderen sollten wortbasierte Suffixbäume mit weniger Speicherplatzbedarf konstruierbar sein, da die Anzahl von Wörtern m deutlich kleiner ist als die Anzahl der Buchstaben n (siehe Tabelle 1 auf Seite 6).

Ziel ist also die Erstellung eines wortbasierten Suffixbaumes für einen String S mit n Buchstaben und m Wörtern mit einer Laufzeitkomplexität von *O(n)* und einer Speicherplatzkomplexität von *O(m)*.

2.1 Naiver Algorithmus

Wie ein traditioneller Suffixbaum kann auch ein wortbasierter Suffixbaum mit einem naiven Algorithmus (siehe etwa Gusfield [2]1999:93ff oder Böckenhauer & Bongartz 2003:56ff.) erstellt werden. Die Speicherplatzkomplexität beträgt dabei von Anfang an wie angestrebt *O(m)*, da der Eingabestring vor der Konstruktion des Baumes in Wörter zerlegt werden kann und so nur Wörter eingefügt werden. Die Laufzeitkomplexität beträgt jedoch beim naiven Algorithmus aufrund der mit Größe des Baumes immer weiter zunehmenden Zeit zum Auffinden der Einfügestelle $O(m^2)$.

2.2 Modifikation traditioneller Algorithmen

Eine Möglichkeit der Nutzung traditioneller Algorithmen zur Erstellung von wortbasierten Suffixbäumen ist in Geertzen (2003:39ff.) beschrieben. Dort wird im Wesentlichen der Algorithmus von Ukkonen (siehe Gusfield [2]1999:94ff.) verwendet, wobei Problemen durch das wesentlich größere Alphabet durch effiziente Speicherverwaltung begegnet

wird. Geertzen macht keine expliziten Angaben zu Laufzeit und Speicherplatzbedarf des angepassten Algorithmus.

Ein weiterer Ansatz, unter Nutzung traditioneller Algorithmen einen wortbasierten Suffixbaum zu erstellen, besteht in der Erstellung eines traditionellen Suffixbaumes mit einer Laufzeitkomplexität und einer Speicherplatzkomplexität von *O(n)*. Anschließend wird der Baum auf die Größe m komprimiert. Dieser Ansatz hat zwar wie angestrebt eine Laufzeitkomplexität von *O(n)*, jedoch auch eine Speicherplatzkomplexität von *O(n)* und nicht, wie angestrebt, *O(m)* (Andersson et al. 1999).

2.3 Effiziente Konstruktion von wortbasierten Suffixbäumen

In Andersson et al. (1999:5ff.) wird ein Algorithmus vorgestellt, der die Erstellung eines wortbasierten Suffixbaumes mit einer Laufzeitkomplexität von *O(n)* und einer Speicherplatzkomplexität von *O(m)* ermöglicht.[2] Der Algorithmus besteht aus den folgenden Schritten:

1. Konstruktion eines Trie für die Menge der unterschiedlichen Wörter, der Types *m'* im Eingabetext *S*. Laufzeitkomplexität: *O(n)*, Speicherplatzkomplexität: *O(m')*.
2. Wörter im Trie werden durchnummeriert. Laufzeit- und Speicherplatzkomplexität: *O(m')*.
3. Jedem Wort im Originalstring wird sein Zahlenwert aus dem Trie zugeordnet. So entsteht ein String aus Zahlen. Laufzeit- und Speicherplatzkomplexität: *O(m)*.
4. Erstellung eine lexikographischen Suffixbaumes mit den Zahlen als Symbole mithilfe eines traditionellen Algorithmus. Laufzeit- und Speicherplatzkomplexität: *O(m)*.
5. Mithilfe des Tries den lexikographischen Suffixbaum zu einen nicht-lexikographischen, wortbasierten Suffixbaum expandieren. Laufzeit- und Speicherplatzkomplexität: *O(m)*.

So hat das effiziente Verfahren einen deutlichen Vorteil in Bezug auf den Speicherplatzbedarf, da nicht zwischenzeitlich ein Suffixbaum für alle Buchstaben des Eingabetextes erstellt werden muss (vgl. Tabelle 1 auf Seite 6).

[2] Für kleine Alphabete stellen Andersson et al. (1999) zudem ein sublineares Verfahren zur Erstellung von wortbasierten Suffixbäumen vor.

Autor	Titel	Zeichen (n)	Token (m)	Types (m')
J. W. v. Goethe	Faust, erster Teil	200.957	46.028	7.386
Mark Twain	Tom Sawyer	387.922	71.457	7.389
August Strindberg	Röda rummet	539.473	91.771	13.425
Leo Tolstoi	Krieg und Frieden	3.217.395	692.328	19.512

Tabelle 1: Anzahl von Buchstaben, Token und Types in natürlichsprachlichem Text (aus: Andersson et al. 1999, ergänzt)

3 Syntaxanalyse

Geertzen (2003) beschreibt mit dem von van Zaanen (2002) entwickelten *Alignment-Based Learning* (ABL) ein Verfahren zur Ermittlung von Konstituenten. Konstituenten sind Teile von Sätzen, die gegen andere austauschbar sind. Konstituenten, die im gleichen Kontext stehen können (die also die gleiche Distribution aufweisen), bilden ein Paradigma. Zu paradigmatischen und syntagmatischen Relationen siehe etwa Lyons ([13]2001:70)[3].

ABL verwendet ursprünglich die Ermittlung der gewichteten Levenshtein-Distanz aller Sätze (*all-against-all* Vergleich) über *dynamic programming* zur Erkennung der Konstituenten (siehe Abschnitt 3.2 auf Seite 7), doch dieses Verfahren ist zu ineffizient, um größere Korpora zu analysieren. Geertzen nennt als Grenze 100.000 Sätze, er bricht bei der Evaluation des Verfahrens jedoch bereits bei einer Anzahl von 38.009 Sätzen ab (siehe Abschnitt 3.3.3 auf Seite 9). Zur Anwendung des Verfahrens auf größere Korpora untersucht Geertzen daher Suffixbäume.

[3] Die Begrifflichkeit der paradigmatischen und syntagmatischen Relationen sowie die zugrunde liegende synchrone Sprachbeschreibung gehen zurück auf den Begründer der modernen Sprachwissenschaft, Ferdinand de Saussure, der damit zu Beginn des 20. Jahrhunderts an eine Tradition anknüpft, die bis zum altindischen Grammatiker Panini zurückgeführt werden kann, der bereits um 500 v. Chr. die Grammatik des Sanskrit in einer Form beschrieb, die in ihrer Mächtigkeit der in den späten 50er Jahren des 20. Jahrhunderts entwickelten Backus-Naur-Form oder Backus-Normalform (die daher auch Panini-Backus-Form genannt wird) entspricht. Eine aktuelle Betrachtung von Paninis Grammatik findet sich in Kiparsky (2002).

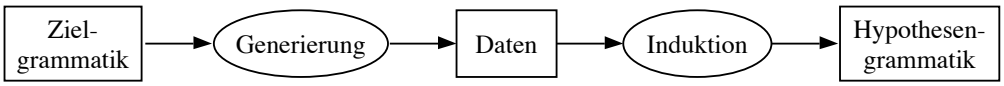

Abbildung 4: Grundgedanke von ABL (aus: Geertzen 2003)

3.1 Grundgedanke

Der Gedanke, der ABL zugrunde liegt, ist dass aus einem Korpus die Grammatik induziert werden kann, die das Korpus ursprünglich generiert hatte (siehe Abbildung 4 und Abbildung 8 auf Seite 10).

Zur Evaluation des Verfahrens wird die generierte Hypothesengrammatik mit der Zielgrammatik verglichen. Dies setzt die Kenntnis der Zielgrammatik voraus, etwa in Form einer Baumbank. Im besten Fall ist die Hypothesengrammatik vollständig, d.h. sie erzeugt alle Sätze der Zielgrammatik, sowie konsistent, d.h. sie erzeugt keine Sätze, die nicht in der Baumbank vorhanden sind.

Der Unterschied zwischen der Analyse beim ABL und einem Parsing-Vorgang besteht darin, dass im Gegensatz zum Parsing bei der ABL-Analyse die Struktur bei der Verarbeitung der Eingabe nicht bekannt ist. Dies wird auch als *structure bootrapping* bezeichnet und stellt eine Form von maschinellem Lernen syntaktischer Struktur dar.

Das Verfahren ist ein unüberwachtes Lernen, da dem Lerner, d.h. dem System, die korrekte Struktur nicht bekannt ist, die Baumbank wird lediglich zur Evaluierung genutzt, das Lernen des syntaktischen Wissens geschieht aus nicht annotierten Korpora und ohne Eingriff durch den Menschen.

3.2 Vergleich von Sätzen

Konstituenten werden beim ABL über paarweisen Vergleich aller im Korpus enthaltenen Sätze ermittelt. Dies kann über die Levenshtein-Distanz erfolgen, so wäre etwa die (wortbasierte) Levenshtein-Distanz zwischen *Ich kaufe ein Auto* und *Ich kaufe ein Boot* 1, das geänderte Wort (*Haus* oder *Boot*) erscheint im gleichen Kontext (nämlich nach *Ich kaufe ein*) und ist damit eine Konstituente des Satzes. Die beiden Wörter, die im gleichen Kontext auftauchen (*Auto* und *Boot*) bilden gemeinsam ein Paradigma. In ähnlicher Weise ist diese Information in einem Suffixbaum gespeichert (siehe Abbildung 3 auf Seite 3).

3.3 Alignment-Based Learning (ABL)

Ziel von ABL ist das Ermitteln einer Hypothesengrammatik durch das oben beschriebene Verfahren. Es werden verschiedene Hypothesen pro Satz aufgestellt und im Anschluss kontrolliert. Kriterium bei dieser Kontrolle ist, dass geänderte Sätze valide sein müssen, d.h. im Korpus vorhanden sind. Wenn Sätze valide sind, werden sie in den Hypothesenraum aufgenommen. Für das Beispiel im vorigen Abschnitt wären die Hypothesen für beide Sätze etwa:

Ich kaufe ein [Auto]
Ich kaufe ein [Boot]

3.3.1 Über die Levenshtein-Distanz

Durch Erstellung eines *edit transcript* (siehe Gusfield [2]1999:215ff.) für die zu vergleichenden Sätze können Gemeinsamkeiten ermittelt werden: Gleiche Wörter bilden Wortcluster, ungleiche Teile sind dann im gleichen Kontext austauschbar und damit Konstituenten-Hypothesen.

3.3.2 Über Suffixbäume

Im Suffixbaum sind alle Wortcluster bereits in der Struktur gespeichert: Alle Wortcluster beginnen am Wurzelknoten und enden in einem inneren Knoten. Von inneren Knoten ausgehende Verzweigungen markieren so stets den Beginn von Konstituenten und bilden Paradigmen. Jedoch ist im Suffixbaum an sich das Ende von Konstituenten nicht markiert. Geertzen evaluiert verschiedene Strategien zur Ermittlung des Endes von Konstituenten:

1. Konstituenten beginnen an inneren Knoten und enden am Satzende. So können Konstituenten am Ende von Sätzen erkannt werden (siehe Abbildung 5 auf Seite 9).

2. Wird der Suffixbaum zusätzlich andersrum aufgebaut können auch Konstituenten am Anfang von Sätzen erkannt werden (siehe Abbildung 6 auf Seite 10).

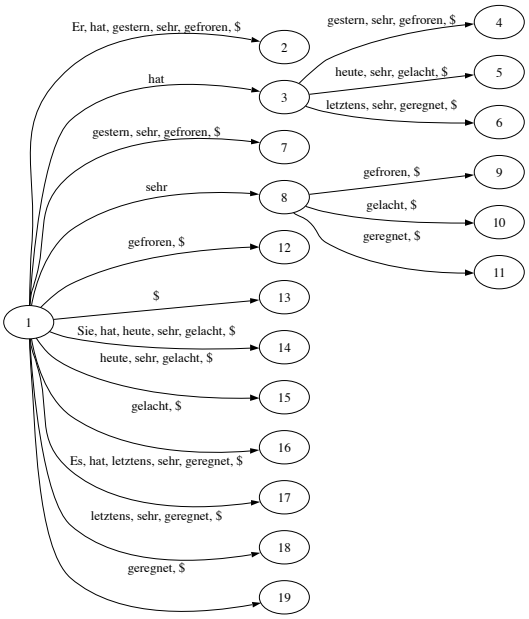

Abbildung 5: Generalisierter, wortbasierter Suffixbaum für *Er hat gestern sehr gefroren. Sie hat heute sehr gelacht. Es hat letztens sehr geregnet.*

3. Durch Kombination der öffnenden und schließenden Klammern der ersten beiden Schritte können auch Konstituenten in der Satzmitte erkannt werden (siehe Abbildung 7 auf Seite 10)

Letzteres, kombiniertes Verfahren lieferte Geertzen die besten Ergebnisse, daher werden im Folgenden lediglich diese wiedergegeben.

3.3.3 Ergebnisse

Die Evaluation von ABL erfolgt über die im Information Retrieval üblichen Werte Precision und Recall, die im Kontext von ABL das Folgende ausdrücken:

- Recall: Anteil der Konstituenten in der ursprünglichen Baumbank, die auch in der gelernten Baumbank vorhanden sind.
- Precision: Anteil der korrekten Konstituenten an den ermittelten Konstituenten. Geertzen nimmt eine perfekte Selektion von korrekten Hypothesen an, daher be-

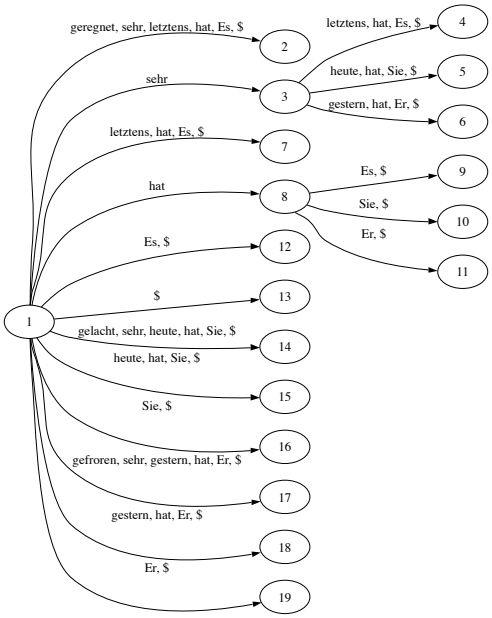

Abbildung 6: Generalisierter, umgekehrter, wortbasierter Suffixbaum für *Er hat gestern sehr gefroren. Sie hat heute sehr gelacht. Es hat letztens sehr geregnet.*

$$[_2[_2\text{Er}]_2 \text{ hat } [_1\text{gestern}]_2 \text{ sehr } [_1\text{gefroren}]_1]_1$$

Abbildung 7: Ermittelte Konstituenten beim kombinierten Verfahren (Abschnitt 3.3.2 auf Seite 8): Mit einer 1 gekennzeichnete Klammern wurden durch den ersten, vorwärts aufgebauten Baum ermittelt, mit einer 2 gekennzeichnete Klammern durch den zweiten, rückwärts aufgebauten Baum. Konstituenten in der Mitte von Sätzen können durch die Kombination der so ermittelten Klammerung bestimmt werden, hier *gestern*.

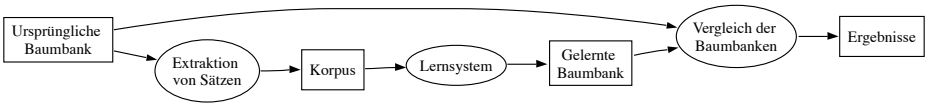

Abbildung 8: Vorgehen bei der Evaluierung von ABL (nach Geertzen 2003:58, vgl. Abb. 4 auf Seite 7)

Korpus	ATIS	OVIS	WSJ 1	WSJ 2
Beschaffenheit				
Sätze	568	10000	1904	38009
Satzlänge (Wörter pro Satz)	7,5	3,5	>20	>20
Recall				
Levenshtein	48.8	94.22	53.26	–
Suffixbaum	38.35	59.18	52.05	37.27

Tabelle 2: Beschaffenheit der drei Korpora und Ergebnisse

trägt die Precision immer 100% für diese Auswertung, im Folgenden werden daher nur die Recall-Werte wiedergegeben.

Geertzen verwendet drei Baumbanken zur Evaluation des Verfahrens, dies sind im Einzelnen ATIS (*Air Traffic Information System*, aus der Penn-Baumbank[4]), OVIS (*Openbaar Vervoer Systeem*, eine niederländische Baumbank mit Texten aus einem Verkehrs-Informationssystem), sowie zwei Korpora verschiedener Größe aus dem *Wall Street Journal*. Die Korpora unterscheiden sich deutlich in Umfang und Beschaffenheit (siehe Tabelle 2).

Aus Tabelle 2 wird deutlich, dass die gewichtete Levenshtein-Distanz[5] für alle Korpora die besseren Ergebnisse liefert, außer bei einer Korpusgröße, die mit dem Verfahren nicht mehr zu bewältigen ist. Jedoch sind die Ergebnisse bei dem Korpus mit den längsten Sätzen nahezu identisch. Zudem ist zu bemerken, dass die Qualität der Ergebnisse des Suffixbaumes auf dem gleichen Korpus mit steigender Korpusgröße abnimmt. Möglicherweise ist der Grund hierfür jedoch, dass in dem von Geertzen untersuchten Korpora die Anzahl unterschiedlicher syntaktischer Konstruktionen mit einer Zunahme der Korpusgröße zunächst ebenso zunahm. Interessant könnte in diesem Zusammenhang eine Evaluation mit mehr als zwei Korpusgrößen gleicher Beschaffenheit und größeren Umfangs sein. Das größte von Geertzen untersuchte Korpus hatte in der Anzahl der Sätze etwa den Umfang von *Krieg und Frieden* von Leo Tolstoi. Denkbar und aufgrund der in Abschnitt 2 auf Seite 4 beschriebenen Eigenschaften von Suffixbäumen auch realisierbar wäre ein Analyse mit vielfach größeren Korpora.

[4]The Penn Treebank Project: <http://www.cis.upenn.edu/~treebank/>

[5]Die Gewichtung für das Hinzufügen, Löschen und Ersetzen von Symbolen beträgt 1,1,2.

Es ist nicht klar, wie die Qualität des Levenshtein-basierten Verfahrens bei größeren Korpora ausfallen würde. Für kleinere Korpora wäre jedoch eine Verbesserung der Ergebnisse des Suffixbaum-basierten Verfahrens und damit eine Annäherung an die Ergebnisse des Levenshtein-basierten Verfahrens wünschenswert. Dies könnte etwa durch eine Annäherung an die Vorgehensweise des Levenshtein-basierten Verfahrens geschehen, eventuell durch Verfahren wie Matching mit *k-mismatch* oder Matching mit *wildcards* (siehe Abschnitte 4.3 und 4.2 auf Seite 14 und 15).

Ein anderer Ansatz bestünde darin, die Levenshtein-basierte Variante von ABL mithilfe von hybridem *dynamic programming* und Suffixbäumen (siehe Abschnitt 4.4 auf Seite 15) effizienter zu machen und so auch größere Korpora mit diesem Verfahren zu analysieren.

4 Ähnlichkeitsmaße

4.1 Suffixbäume zur Optimierung im Hintergrund

4.1.1 *Lowest Common Ancestor*

Der *lowest common ancestor* (LCA) von zwei Knoten in einem Baum ist der tiefste Knoten im Baum, der gleichzeitig Vorfahr beider Knoten ist (siehe Abbildung 9 auf Seite 13).

Der LCA kann in konstanter Zeit ermittelt werden. Das in Gusfield ([2]1999:184ff.) beschriebene Vorgehen basiert auf der Darstellung der Baumstruktur als Zahlen in binärer Form. Binärzahlen an den Knoten des Baumes repräsentieren den Pfad zum entsprechenden Knoten: Eine 0 steht für *links*, eine 1 für *rechts*, daran angehangen wird eine 1, anschließend wird die Zahl bis zur Länge der Baumtiefe + 1 mit Nullen aufgefüllt (siehe Abbildung 10 auf Seite 13). Durch Vergleiche der einzelnen Pfadnummern ist eine Ermittlung des *LCA* nach Erweiterungen auch für nicht-binäre Bäume möglich (Gusfield [2]1999:184ff).

4.1.2 *Longest Common Extension*

Die Ermittlung der längsten gemeinsamen Erweiterung (*longest common extension*, LCE) von zwei Strings ab zwei Positionen i in String 1 und j in String 2 ist eine Teilaufgabe von vielen Algorithmen in der Stringverarbeitung, insbesondere im Bereich des wei-

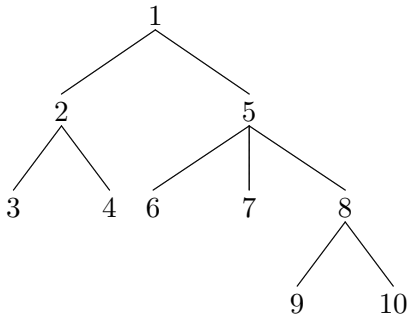

Abbildung 9: Ein Baum mit *depth-first* Nummerierung. Der LCA der Knoten 10 und 7 etwa wäre Knoten 5, der LCA von Knoten 3 und 5 wäre Knoten 1.

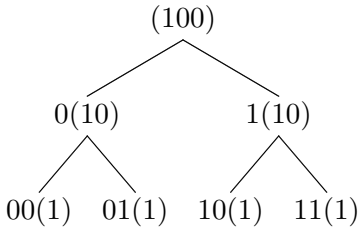

Abbildung 10: Ein Binärbaum mit Pfadnummern im Binärformat. Eine 1 bedeutet 'rechts', eine 0 'links', die Zahlen in Klammern sind Ergänzungen (siehe Abschnitt 4.1.1 auf Seite 12).

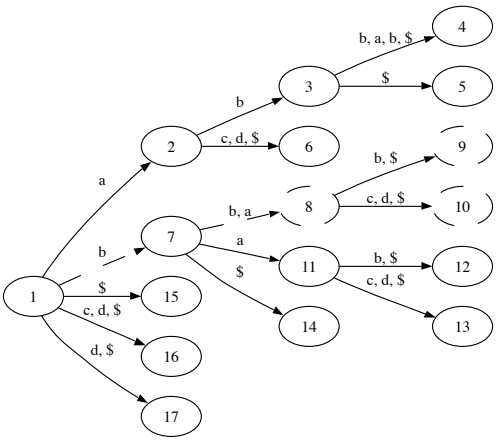

Abbildung 11: Ermittlung der *longest common extension* (LCE) über den *lowest common ancestor* (LCA): Der LCA der Knoten 9 und 10 ist Knoten 8. Der Pfad von der Wurzel zu Knoten 8 ist beschriftet mit der gesuchten LCE der Strings *abbab* ab Position 2 sowie *bbacd* ab Position 1. Die Blätter 9 und 10 korrespondieren mit den Teilstrings beginnend an Position 2 respektive 1 (Indizes sind 1-basiert). Die gesuchte LCE ist hier *bba*.

chen String-Matching, z.B. für Matching mit *wildcards* oder *k-mismatch* (siehe Gusfield [2]1999:196ff.).

Die Ermittlung der LCE ist mit Suffixbäumen in konstanter Zeit möglich, über die Ermittlung des *lowest common ancestor* (LCA, siehe Abschnitt 4.1.1 auf Seite 12), denn die Stringtiefe des LCA der zwei Blätter, die den Positionen i und j in den beiden Strings entsprechen ist die Länge der LCE (siehe Abbildung 11 sowie Gusfield [2]1999:196f. und 181ff.).

4.2 Matching mit Slots: *wildcards*

Matching mit *wildcards*, d.h. mit Positionen, an denen ein beliebiges Symbol stehen kann (hier symbolisiert durch '*') könnte etwa eingesetzt werden zur Ermittlung aller Sätze, die folgendem Ausdruck entsprechen:

Er kauft ein *

Auf diese Weise könnte eine effiziente Ermittlung von ähnlichen Sätzen und damit von Konstituenten und Paradigmen möglich sein. Zur Ermittlung längerer Konstituenten könnte mit mehreren *wildcards* gesucht werden. Das Verfahren hat eine Laufzeitkomplexität von *O(k*m)* zur Ermittlung aller Treffer, bei einer Zahl k von wildcards in einem Text der Länge m (Gusfield [2]1999:199).

4.3 Matching mit Fehlern: *k-mismatch*

Ein mögliches Vorgehen beim Matching mit einer bestimmten Anzahl von erlaubten Fehlern k (*k-mismatch*) wäre etwa die Suche nach allen Vorkommen von *Er kauft ein Haus* mit einem Fehler (k=1). Eine solche Suche könnte Ergebnisse wie *Er kauft ein Pferd, Sie kauft ein Haus, Er hat ein Haus* liefern. Diese Sätze könnten zur Ermittlung von Paradigmen miteinander verglichen werden. Das Verfahren hat eine Laufzeitkomplexität von *O(n+m)* zur Ermittlung aller Treffer eines Suchmusters der Länge n in einem Text der Länge m (Gusfield [2]1999:200).

4.4 Levenshtein-Distanz über hybrides *dynamic programming* mit Suffixbäumen

Eine *dynamic programming* Lösung zur Ermittlung der gewichteten Levenshtein-Distanz beim *p-against-all* und *all-against-all* Matching mit quadratisch wachsender Laufzeit kann durch den Einsatz von Suffixbäumen in seiner Effizienz verbessert werden. Der Grundgedanke ist dabei, redundante Teilstrings nicht jedes mal neu, sondern nur einmal miteinander zu vergleichen. Dazu wird die im Rahmen des *dynamic programming* aufgebaute *edit table* beim Traversieren des Suffixbaumes statt beim Verarbeiten des Textes aufgebaut (Gusfield [2]1999:279ff.).

Eine Effizienzsteigerung des Verfahrens ist dadurch davon anhängig, wie viele redundante Strukturen im Text enthalten sind. Ein Maß für den Umfang redundanter Strukturen ist die Länge des Baumes:[6] Je kürzer der Baum, desto höher die Redundanz. Gusfield ([2]1999:286) berichtet von einer Effizienzsteigerung um den Faktor 100 bei menschlicher DNA als Text. Da auch in natürlicher Sprache redundante Strukturen vorhanden sind,[7] wäre mit diesem Verfahren eine Anwendung des Levenshtein-basierten ABL (Abschnitt

[6] Die Länge eines Baumes ist die Summe aller Symbole aller Kanten (siehe Gusfield [2]1999:282)

[7] Das Ausmaß an redundanten Strukturen in natürlichsprachlichem Text ist aufgrund des größeren Alphabets (insbesondere bei Wörtern als Symbole) vermutlich erheblich geringer als in der Bioinformatik, wo mit einem Alphabet aus nur 4 Symbolen gearbeitet wird.

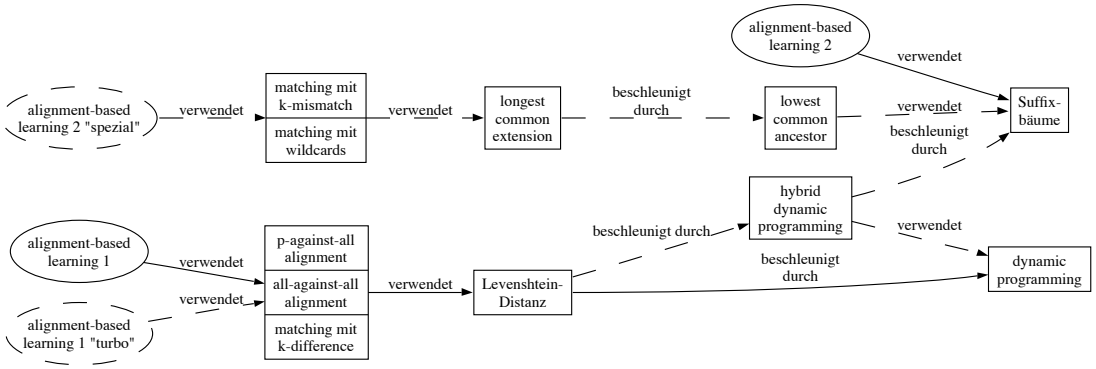

Abbildung 12: Übersicht der beschriebenen Verfahren. Durchgezogene Linien beschreiben die vorgestellten Verfahren des *Alignment-Based Learning* aus Geertzen (2003), gestrichelte Linien beschreiben mögliche Erweiterungen durch die beschriebenen Verfahren aus Gusfield ([2]1999).

3.3.1 auf Seite 8) mithilfe von Suffixbäumen auch für größere Korpora möglich, als durch *dynamic programming* alleine.

5 Fazit

Suffixbäume sind eine effizient nutzbare, vielseitige Datenstruktur für die Stringverarbeitung. Sie können auch für die Verarbeitung natürlicher Sprache verwendet werden, sowohl auf Wort- als auch auf Satzebene, da auch für diese Bereiche effiziente Algorithmen zur Verfügung stehen (siehe Andersson et al. 1999 und Abschnitt 2.3 auf Seite 5). Es gibt erste Anwendungen im Bereich der Erkennung syntaktischer Strukturen mit auf Suffixbäumen basierenden Verfahren mit linear wachsender Laufzeit (siehe Geertzen 2003 und Abschnitt 3 auf Seite 6). Solche Ansätze sind möglicherweise ausbaubar, einerseits durch eine Beschleunigung bestehender Verfahren mit quadratisch wachsender Laufzeit (etwa *all-against-all* Matching) durch den Einsatz von Suffixbäumen (hybrides *dynamic programming*, siehe Gusfield [2]1999:279ff. und Abschnitt 4.4 auf Seite 15), andererseits durch Anwendung von Verfahren mit linear wachsender Laufzeit wie Matching mit *wildcards* oder *k-mismatch* (siehe Gusfield [2]1999:196ff. und Abschnitt 4.1 auf Seite 12). Eine Übersicht der beschriebenen Verfahren findet sich in Abbildung 12.

A Implementation

Die Darstellungen von Suffixbäumen in der vorliegenden Arbeit wurden mit der im Rahmen dieser Arbeit entstandenen Software erstellt. Die Software ermöglicht eine flexible Erstellung und Darstellung von Suffixbäumen aus unmittelbar eingegebenem oder in einer Datei gespeichertem Text. Darüber hinaus kann mit zwei Matching-Algorithmen auf dem eingegebenen Text experimentiert werden. Neben der graphische Oberfläche zur Erstellung von Abbildungen kann die Software als API genutzt werden. Die Software ist im WWW verfügbar[8] und enthält weitergehende Dokumentation.

A.1 Programmierschnittstelle

Über die Programmierschnittstelle (*application programming interface*, API) können mit linear wachsender Laufzeit (basierend auf dem Algorithmus von Andersson et al. (1999) und und einer angepassten Version der Klasse *UkkonenSuffixTree* der BioJava API[9]) wortbasierte Suffixbäume konstruiert, traversiert und als *dot* (siehe folgender Abschnitt) exportiert werden. Für diese Bäume ist ein LCA-Retrieval in konstanter Zeit implementiert (basierend auf einer Implementation von A.G. McDowell[10]). Es bestehen Implementationen von Matching mit *k-mismatch* (Gusfield [2]1999:200) und *wildcards* (Gusfield [2]1999:199) sowie eine experimentelle Implementation der Ermittlung der LCE über den LCA (siehe Gusfield [2]1999:196f.). Darüber hinaus können kleine Suffixbäume naiv in quadratischer Laufzeit sowohl buchstaben- sowie wortbasiert konstruiert und im *dot* Format exportiert werden.

A.2 Graphische Oberfläche

Die graphische Darstellung der Suffixbäume basiert auf einem in Java implementierten naiven Algorithmus zur Konstruktion von Suffixbäumen (siehe Böckenhauer & Bongartz 2003:56ff.), optional mit Wörtern als Symbole, generalisiert für mehrere Sätze und umkehrbar. Die graphische Darstellung erfolgt mithilfe der Open-Source Software GraphViz,[11] das generierte *dot* Format kann abgespeichert werden, um daraus andere

[8] Suffixbäume und Ähnlichkeitsmaße: <http://www.quui.de/fsteeg/papers/spinfo/stringverarbeitung>
[9] BioJava API: <http://www.biojava.org>
[10] Software von A. G. McDowell: <http://www.mcdowella.demon.co.uk/programs.html>
[11] GraphViz Graph Visualization Software: <http://www.graphviz.org>

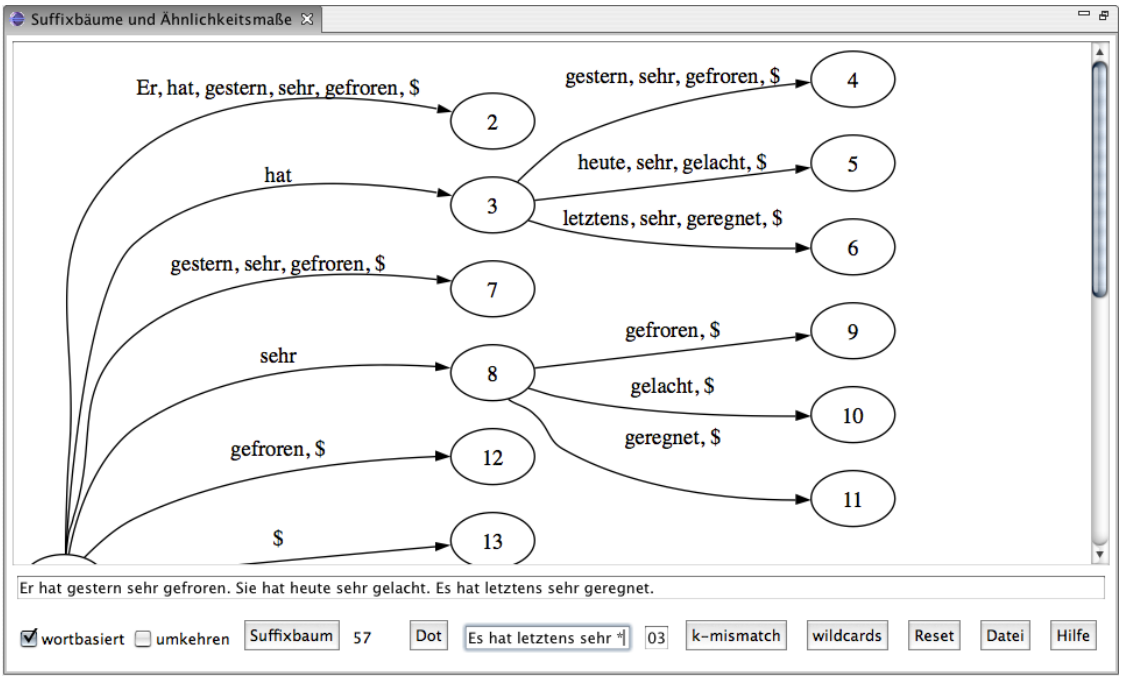

Abbildung 13: Screenshot der im Rahmen dieser Arbeit entstandenen Software zur Visualisierung von Suffixbäumen und zwei der vorgestellten Matching-Algorithmen.

Bildformate zu erstellen.

Größere Texte können aus einer Datei eingelesen und als wortbasierter Suffixbaum im *dot* Textformat gespeichert werden. Der hier verwendete Suffixbaum ist eine experimentelle Java-Implementation auf Grundlage des in Andersson et al. (1999) beschriebenen Verfahrens mit linear wachsender Laufzeit (siehe Abschnitt 2.3 auf Seite 5). Die graphische Oberfläche (*graphical user interface*, GUI) ist als Eclipse Plug-in[12] implementiert.

In der Software sind darüber hinaus Java-Implementationen und Visualisierungen der Algorithmen zum Matching mit *k-mismatch* (Gusfield [2]1999:200) und *wildcards* (Gusfield [2]1999:199) enthalten, sowie angepasste Versionen für wort- bzw. satzbasiertes Matching. Abbildung 13 zeigt einen Screenshot der Anwendung.

[12]Eclipse Platform: <http://www.eclipse.org>

Literatur

Andersson, Arne; Larsson, Jesper & Kurt Swanson (1999) "Suffix Trees on Words". *Algorithmica* 23. 24. 1. 2006 <http://www.larsson.dogma.net/words-alg.pdf>.

Geertzen, Jeroen (2003) *String Alignment in Grammatical Inference: What Suffix Trees can do.* Magisterarbeit, Universität Tilburg, Die Niederlande. 24. 1. 2006 <http://www.cosmion.net/jeroen/publications/docs/geertzen03mthesis.pdf>.

Gusfield, Dan ([2]1999 [[1]1997]) *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge.

Kiparsky, Paul (2002) *On the Architecture of Panini's Grammar.* Three lectures delivered at the Hyderabad Conference on the Architecture of Grammar, Jan. 2002. 14. 2. 2006 <http://www.stanford.edu/~kiparsky/Papers/hyderabad.pdf>.

Lyons, John ([13]2001 [[1]1968]) *Introduction to Theoretical Linguistics.* Cambridge University Press. Cambridge.

van Zaanen, Menno (2002) *Bootstrapping Structure into Language: Alignment-Based Learning.* Dissertation, University of Leeds, Leeds, UK. 13. 2. 2006 <http://arxiv.org/abs/cs.LG/0205025>.