Über die Vermeidung redundanter Betrachtungen beim Approximate String Matching

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik
von

Christoph Jan Richter

Dortmund

2004

Über die Vermeidung redundanter Betrachtungen beim Approximate String Matching

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik
von

Christoph Jan Richter

Dortmund

2004

Tag der mündlichen Prüfung: 11. Februar 2005

Dekan / Dekanin: Prof. Dr. Bernhard Steffen
Gutachter: Prof. Dr. Wolfgang Banzhaf
Prof. Dr. Peter Padawitz

Bewertungsskala: ausgezeichnet, sehr gut, gut, genügend

Inhaltsverzeichnis

Danksagung

An erster Stelle gebührt mein Dank Herrn Prof. Dr. Wolfgang Banzhaf für seine wohlwollende Anleitung während der Zeit der gemeinsamen Arbeit und seine vertrauensvolle Unterstützung bei der Erstellung des vorliegenden Werkes.

Als ungemein wichtig hat sich auch die freundliche und kreative Arbeitsatmosphäre am Lehrstuhl für Systemanalyse des Fachbereichs Informatik an der Universität Dortmund erwiesen. So möchte ich allen Mitarbeitern dieses Lehrstuhls für die Begleitung während meiner Zeit dort danken. Insbesondere hervorheben möchte ich dabei Dr. André Leier, Dipl.-Inform. Michael Emmerich, Dr. Jens Busch und Dipl.-Inform. Wolfgang Kantschik. Ohne ihre Hilfe in Form von zahlreichen Diskussionen, programmiertechnischen Tipps, der Bereitstellung von Daten und des Korrekturlesens hätte dieses Dokument wohl schwerlich die vorliegende Form erreicht.

Ein Teil dieser Arbeit entstand während eines Aufenthalts als Gaststudent am Fachbereich Informatik der Memorial University of Newfoundland in St. John's, Kanada. Meine Dankbarkeit gilt allen Mitarbeitern dort für die herzliche Gastfreundschaft und die vielen anregenden Gespräche. Ganz besonders möchte ich dabei Dr. Todd Wareham für seine Unterstützung mit vielfältigen Informationen im Kontext der Bioinformatik danken.

Zu großem Dank bin ich auch Dr. William B. Langdon vom University College London verpflichtet, der zeitgleich mit mir Gast an der Memorial University war. Die sich aus vielen Diskussionen ergebenden Impulse waren für mich sehr wertvoll.

Mein Dank gilt ferner Prof. Dr. Peter Padawitz, Prof. Dr. Gisbert Dittrich und Dr. Stefan Droste, für Ihre Bereitschaft, sich mit dieser Arbeit auseinander zu setzen.

1 Einleitung

Die Weisheit läuft dir
nach. . . doch du bist schneller!

(M. Krischer)

A string of alpha/numeric characters needs to be parsed to become data;
Data needs to be interpreted to become information;
Information needs to be applied to become knowledge;
Knowledge needs to be integrated into experience to be meaningful;
And experience needs to be tested against the context of truth to become wisdom.
J. Thom Mickelson, 2001[1]

Im *Informationszeitalter* — wie die heutige Zeit oft genannt wird — liegen fast beliebig viele Daten und Informationen (meist in Form von durch Einordnung in eine Themendatenbank interpretierten Daten) digital gespeichert vor. Die Grundlage für den Weg zur Weisheit ist somit nach der von Mickelson beschriebenen Daten-Hierarchie gesichert. Jedoch erweist sich gerade in Anbetracht der großen Menge an Daten und Informationen oft der nächste Schritt in obiger Hierarchie, also die Interpretation von Daten oder die Anwendung von Informationen, als schwierig. Als essenzielle Werkzeuge haben sich dabei Suchverfahren herausgestellt, die Informationen in Datenbanken auffinden oder Daten vergleichend neben andere Daten stellen. Allgemein werden Daten in Form von Zeichenketten, auch Strings genannt, gespeichert. Daher basieren die meisten Suchverfahren wie auch alle Betrachtungen innerhalb dieser Arbeit auf Zeichenketten.

Für das Problem der exakten Suche gibt es schon länger einige sehr schnelle Algorithmen. Spätestens jedoch seit dem Erscheinen des „Handbook of Exact String Matching Algorithms“ von Christian Charras und Thierry Lecroq [23], welches neben einem gelungenen Überblick auch die jeweilige Implementierung demonstriert, stellt das Problem der exakten Suche in der Praxis kein Problem mehr dar.

Eine viel größere Bedeutung als die exakte Suche hat in den letzten Jahren die fehlertolerierende Suche gewonnen, die üblicherweise *Approximate String Matching* genannt wird. Die fehlertolerierende Suche ist nicht nur für die offensichtliche Anwendung des Auffindens von Tippfehlern und die Spracherkennung verwendbar, sondern es existiert

[1] http://exalter.net/resource/axiom.html

ein breites Spektrum an Anwendungsgebieten. Beispielsweise ist die Genetik ein wichtiges Anwendungsfeld. Insbesondere ist hier das durch die ausführliche Berichterstattung in den Medien bekannte *Human Genome Project* hervorzuheben, welches sich mit der Gewinnung der Sequenzdaten des menschlichen Genoms befasste. Die dabei gewonnenen Sequenzdaten alleine reichen nicht aus, um Informationen über die Funktionalität einzelner Gene zu erhalten, die für medizinische Anwendungen unbedingt notwendig sind. Deswegen werden neu sequenzierte Gene zum Zwecke einer möglichen Funktionsvorhersage unter Nutzung fehlertolerierender Suche auf Ähnlichkeiten mit bereits bekannten Genen überprüft [51, 71, 78].

Die Vielzahl von Anwendungsgebieten für das Approximate String Matching führte zu der Entwicklung verschiedener Algorithmen. Trotz der Vielfalt an Algorithmen gilt dieses Gebiet jedoch nicht als vollständig erschlossen [85].

Das Approximate String Matching ist durch die Beachtung möglicher Fehler deutlich aufwendiger als die exakte Suche, weshalb insbesondere bei großen Datenmengen schnelle Algorithmen gefordert sind.

Das Ziel dieser Arbeit ist es daher, einen schnellen Algorithmus für das Approximate String Matching zu erreichen. Die zentrale Idee dabei ist es, durch die Ausnutzung von Redundanzen eine Beschleunigung zu erzielen. Dazu werden hier zwei verschiedene Ansätze vorgestellt, die *Patchwork-Verifikation* und *GraI* („Grammatikbasierter Index“) genannt werden. Beide Ansätze gehören zur Klasse der Filteralgorithmen, die zweiphasig arbeiten und dabei die eigentliche Suche und die Betrachtung der möglichen Fehler trennen. Dabei dient die erste Phase dazu, mögliche Lösungsstellen zu identifizieren, während es der zweiten, aufwendigeren Phase obliegt, diese Stellen zu überprüfen.

In der zweiten Phase kann es zur mehrfachen - und damit redundanten - Betrachtung mancher Bereiche im zu durchsuchenden Text kommen. Mit dem ersten hier vorgestellten Ansatz, der *Patchwork-Verifikation*, wird ein allgemeines Verfahren demonstriert, diese redundanten Betrachtungen zu vermeiden und dadurch Berechnungszeit zu sparen. Anders als bei anderen Verfahren, die eine Mischung der beiden Phasen (Suche und Verifikation) propagieren, werden bei der Patchwork-Verifikation keine Eingriffe in die grundsätzliche Algorithmusstruktur vorgenommen. Prinzipiell kann daher bei einem gegebenen, auf dem Filterprinzip basierenden Algorithmus einfach die Verifikationsroutine ausgetauscht werden.

Im Gegensatz zur Patchwork-Verifikation betrachtet der zweite Ansatz, *GraI*, keine algorithmischen Redundanzen. Vielmehr besteht die Idee darin, im zu durchsuchenden Text enthaltene redundante Bereiche zu erkennen und durch Vermeidung unnötiger Betrachtungen einen beschleunigten Algorithmus zu erhalten. Zur Identifizierung redundanter Textbereiche wird eine Grammatik verwendet, bei der sich aus jeweils identischen Textstellen eine Regel ableitet. Die Grammatik liefert mit ihrer Regelstruktur damit eine Möglichkeit, sowohl während der Suchphase als auch in der Verifikationsphase mehrfach vorhandene Textbereiche nur einmal zu betrachten und so eine Beschleunigung beim Approximate String Matching zu erreichen.

Diese Arbeit strukturiert sich wie folgt:

In Kapitel 2 werden für das Verständnis der Arbeit notwendige Erkenntnisse über das Approximate String Matching vermittelt. Zudem wird dort auch das eng verwandte Problem des *Compressed Approximate String Matching* in den Kontext dieser Problemstellung eingeordnet.

Kapitel 3 stellt den prinzipiellen Lösungsansatz für das Problem des Approximate String Matching dar und geht kurz auf weitere, direkt darauf aufbauende Lösungsalgorithmen sowie auf automatenbasierte Lösungsalgorithmen ein.

Die Klasse der Filteralgorithmen, die für die in dieser Arbeit vorgestellten Ansätze relevant ist, wird in Kapitel 4 eingehend betrachtet. Dabei werden verschiedene Filteralgorithmen auf der Grundlage einer so erstmalig durchgeführten Klassifikation vorgestellt.

Kapitel 5 präsentiert den neuen Ansatz der Patchwork-Verifikation. Neben einer vollständigen Beschreibung findet sich eine Analyse der Auswirkungen dieses Ansatzes.

Der zweite hier in der Arbeit vorgeschlagene Ansatz, GraI, stellt bestimmte Anforderungen an die zu verwendende Grammatik. Diese Anforderungen werden in Kapitel 6 diskutiert. Zudem werden verschiedene Methoden zur Erzeugung von Grammatiken vorgestellt und unter Berücksichtigung der gewünschten Anwendung bewertet.

Der Algorithmus GraI, der dann eine der Grammatiken nutzt, wird in Kapitel 7 entwickelt. Zudem werden dort auch kleinere Variationen angesprochen, bevor in Kapitel 8 eine umfassende Analyse vorgenommen wird.

In Kapitel 9 werden die wichtigsten Ergebnisse dieser Arbeit zusammengefasst. Abschließend werden dort auch in einem Ausblick mögliche Zielrichtungen für an diese Arbeit anknüpfende Forschungen aufgezeigt.

2 Approximate String Matching

Wenn etwas sicher ist, dann ist
es nur die Unsicherheit.

*(Beatrix, Königin der Niederlande, *1938)*

Approximate String Matching ist die Suche in einem Datensatz nach einem bestimmten Datum, wobei als Antworten auch hinreichend ähnliche Daten akzeptiert werden.

Diese sehr allgemeine und weit gefasste Problemstellung tritt in den vielfältigsten Anwendungsgebieten auf. Ausführliche Benennungen der verschiedenen Anwendungsfelder finden sich in verschiedenen Arbeiten wie z. B. [63, 85, 104], während an dieser Stelle nur einige wenige dieser Gebiete exemplarisch genannt werden sollen, um die Praxisrelevanz und die Bandbreite des Problems zu verdeutlichen.

Text Retrieval:
Die Korrektur von falsch geschriebenen Wörtern ist vielleicht das älteste mögliche Anwendungsgebiet für das Approximate String Matching. Eine große Bedeutung hat diese Art der Fehler beim *Information Retrieval (IR)*, also dem Finden relevanter Informationen in großen Text-Sammlungen. Beispielsweise ist es ohne das dort mittlerweile fundamentale Werkzeug des Approximate String Matching unmöglich, ein fehlerhaft in eine Datenbank (oder auch ins Internet) eingegebenes Wort wieder zu finden. Auch andere Anwendungen, die sich mit der Verarbeitung von Texten beschäftigen, sind ohne Approximate String Matching undenkbar. Z. B. sind dabei Rechtschreibprüfungen, Schnittstellen zur Spracheingabe, computergestütztes Lernen (z.B. Sprachlernsysteme) und die optische Zeichenerkennung (OCR) [36, 107] zu nennen. Detailliertere Informationen über Approximate String Matching und Text Retrieval finden sich beispielsweise in [11, 30, 63, 129, 142, 143].

Computational Biology:
Viele der in der Molekularbiologie und Genetik wichtigen Probleme befassen sich mit DNS- und Proteinsequenzen. Diese können als Texte über einem speziellen Alphabet aufgefasst werden (am bekanntesten ist wohl das Alphabet der DNS bestehend aus den Basen Adenin (A), Guanin (G), Cytosin (C) und Thymin (T)). Die Suche von spezifischen Sequenzen in diesen Texten hat sich als eine wesentli-

che Operation bei der Lösung der häufigsten Probleme entwickelt. Probleme dieser Art sind z. B. die Suche nach bestimmten Eigenschaften in DNS-Strängen, die Bestimmung der Unterschiedlichkeit zweier genetischer Sequenzen (wichtig für den Aufbau eines Evolutions- oder Abstammungsbaumes (Phylogenetic Tree)), das Zusammensetzen von DNS-Fragmenten zu einem DNS-Strang oder auch die Suche nach ähnlichen Proteinen zur Funktionsvorhersage bei neu entdeckten Proteinen.

Aufgrund leicht ungenauer experimenteller Daten, real aufgetretener Mutationen oder aufgrund der Problemstellung (Ähnlichkeitsbestimmung) ist eine Suche notwendig, die Fehler erlaubt.

Weitere, detailliertere Informationen über die Anwendung des Approximate String Matching in der Computational Biology finden sich beispielsweise in [4, 48, 82, 94, 104, 108, 130].

Datenkompression:
Bei manchen (meist speziellen) Alphabeten kann eine gute Datenkompression durch das Betrachten von Wiederholungen ähnlicher Teilstrings erreicht werden. Allison et al. [2, 3] beispielsweise nutzen ähnliche Wiederholungen in Kombination mit einem geeigneten Transformationsmodell, um eine gute Symbolvorhersage bei der Kompression von DNS-Strings zu erreichen. Auch bei der verlustbehafteten Bildkompression kann die Identifikation von ähnlichen Strings nützlich sein [10, 72, 73].

Aufgrund dieses breiten Spektrums von Anwendungsgebieten, wurde der für das generelle Problem des Approximate String Matching grundsätzliche Lösungsansatz mehrfach unabhängig voneinander entwickelt (vgl. Kapitel 3.1) und auch später entstanden verschiedenste verbesserte Lösungen.

Dieses Kapitel befasst sich mit den grundlegenden Dingen zum Problem des Approximate String Matching und gibt dabei zum Verständnis der folgenden Kapitel notwendige Informationen.

Im Folgenden werden zuerst in Kapitel 2.1 die wichtigen Bezeichnungen genannt, bevor in Kapitel 2.2 das Problem formal definiert wird. Dem folgend werden in den Kapiteln 2.3 bis 2.5 die wichtigsten Begriffe und Annahmen erläutert, die die gemeinsame Basis für die in dieser Arbeit betrachteten Algorithmen bilden. Ebenso wird der Gedanke der Klassifikation erklärt (Kapitel 2.6), die in dieser Arbeit in den Kapiteln 3 und 4 verwendet wird. Nach einem Abschnitt zur Statistik des Problems (Kapitel 2.7) befasst sich Kapitel 2.8 in aller Kürze mit einer Variante des Approximate String Matching, die auf komprimiert vorliegenden Texten die Suche durchführt.

2.1 Wichtige Bezeichnungen

Mit *Approximate String Matching* (oder manchmal auch *Approximate Pattern Matching* genannt) wird das Problem der ungenauen Textsuche bezeichnet. Dabei werden alle Positionen gesucht, an denen ein kurzes Textstück, *Pattern* genannt, in einem (langen) *Text* auftritt, wobei jedoch eine gewisse Ungenauigkeit zugelassen ist. Diese Ungenauigkeit wird über einen Fehlerwert definiert, der den maximal zugelassenen Unterschied zwischen dem Pattern und der Textstelle angibt.

Die Betrachtung des Unterschieds hat zur Folge, dass es prinzipiell egal ist, ob Fehler im Text oder im Pattern oder in beiden auftreten. Liegen real Fehler sowohl im Pattern als auch im Text vor, so hat dies in der Praxis die Folge, dass nur einer der beiden Teile Fehler enthält (und alle Fehler auf sich akkumuliert), da die fehlerfreie Version des Textes oder des Patterns einfach nicht bekannt ist.

Im Folgenden sollen in Kürze die wichtigsten Begriffe und Bezeichnungen genannt und definiert werden.

> Σ sei ein endliches Alphabet.
>
> Die Größe von Σ sei $|\Sigma| =: \sigma$.
>
> Die Elemente von Σ werden *Buchstaben*, *Symbole*, *Zeichen* oder auch (bei Grammatiken, vgl. Kapitel 6) *Terminale* genannt. Der Begriff des *Nicht-Terminals* kennzeichnet dann explizit ein Symbol, welches nicht zu dem gegebenen Alphabet gehört.
>
> Ein *String* s bezeichnet eine Reihe von Buchstaben eines Alphabets: $s \in \Sigma^*$.
>
> Der Operator $|.| : \Sigma^* \to \mathbb{N}$ gibt die Länge eines Strings an[1]. Konkret ist dabei $|s| = n$ für den String $s = s_1 s_2 \cdots s_n$.
>
> Der *leere String* ϵ ist ein String der Länge 0.
>
> Ein *Substring* (oder auch *Teilstring* oder *factor*[2]) eines Strings $s = s_1 s_2 \cdots s_n$ ist definiert als $s_{i..j} = s_i s_{i+1} \cdots s_j, i, j \in \{1, \ldots, n\}$. Für $i > j$ ergibt sich dabei der leere String.
>
> Ein *Präfix* ist ein Substring am Stringanfang, also $s_{1..j} = s_1 s_2 \cdots s_j, j \in \{1, \ldots, n\}$.
>
> Ein *Suffix* ist ein Substring am Stringende, also $s_{i..n} = s_i s_{i+1} \cdots s_n, i \in \{1, \ldots, n\}$.

2.2 Problemdefinition

Basierend auf einem endlichen Alphabet Σ der Größe σ kann das Problem des Approximate String Matching wie folgt definiert werden:

[1] Die Null wird hier zu den natürlichen Zahlen gezählt.

[2] In englischsprachiger Literatur ist *factor* durchaus üblich, während in deutschen Texten der Begriff *Faktor* keine Verwendung findet.

Definition 2.2.1 (Approximate String Matching (ASM))
Gegeben seien zwei Strings T (ein *Text*) und P (ein *Pattern*) mit den Längen $n = |T|$ und $m = |P|$. $k \in \mathbb{R}$ sei der maximale erlaubte Abstand und ferner sei $d : \Sigma^* \times \Sigma^* \to \mathbb{R}$ eine *Abstandsfunktion* (auch *Distanzfunktion* genannt).

Das Problem des *Approximate String Matching* besteht in der Bestimmung der Menge $\{|x\bar{P}|, T = x\bar{P}y \wedge d(P, \bar{P}) \leq k\}$.

Genau betracht wird zur Bestimmung der Lösungsmenge der Text T in drei Substrings x (Präfix), $\bar{P}$ und y (Suffix) gegliedert. Hat der Substring $\bar{P}$ zum Pattern P einen Abstand (gegeben durch die Abstandsfunktion), der nicht größer ist als k, so zählt die Endposition (generiert durch den Längenoperator) von $\bar{P}$ in T zur Lösungsmenge.

Zur Vereinfachung ist es an dieser Stelle sinnvoll, den Begriff des *Approximate Matching* zu definieren:

Definition 2.2.2 (Approximate Matching)
Ein *Approximate Matching* (eines Patterns P in einem Text T) ist ein Substring $\bar{P}$ von T, für den gilt $d(P, \bar{P}) \leq k$.

In der Lösungsmenge des Problems des Approximate String Matching befinden sich also die Endpositionen aller Approximate Matchings von P in T. Um anstelle der Endpositionen Anfangspositionen von Approximate Matchings zu erhalten, müssen nur die beteiligten Strings umgedreht werden. Insgesamt werden nur Positionen gesammelt, um eine lineare Größe der Ausgabe (im Verhältnis zur Eingabe) gewährleisten zu können.

Es sei noch bemerkt, dass die Voraussetzung eines endlichen Alphabetes keine echte Beschränkung für die Lösung des ASM darstellt. Da das Pattern P immer von endlicher Länge m ist, hat es auch nur maximal m verschiedene Symbole. Alle anderen Symbole können auf ein einziges Symbol abgebildet werden. Jedes Zeichen des Alphabetes kann also mittels einer Suchstruktur auf eines von $m + 1$ verschiedenen Symbolen abgebildet werden. Für einen Algorithmus zur Lösung des ASM bedeutet dies dann, dass bei einem unendlichen Alphabet ein Mehraufwand von $O(\log m)$ für die Suchstruktur zu leisten ist [85].

2.3 Die Abstandsfunktion

Für Algorithmen zur Lösung des ASM-Problems ist die genaue Abstandsfunktion von essenzieller Bedeutung. Allgemein definiert die Abstandsfunktion $d(s, t)$ nur die Kosten für die Transformation des Strings s in den String t, wobei die Beschränkung auf minimale Kosten eine Problemlösung erst sinnvoll macht. Durch die Abstandsfunktion kann der Raum der Strings zu einem *metrischen Raum* werden. Dazu muss die Abstandsfunktion für Strings s, t und u die Bedingungen einer *Metrik* erfüllen:

1. $d(s,t) = 0 \Leftrightarrow s = t$ (Definitheit)
2. $d(s,t) = d(t,s)$ (Symmetrie)
3. $d(s,t) \leq d(s,u) + d(u,t)$ (Dreiecksungleichung)

Die Abstandsfunktion muss weiter präzisiert werden, nicht nur, um die Metrikeigenschaften einer Abstandsfunktion zu überprüfen, sondern um überhaupt praktisch an das Problem herangehen zu können. Je nach Anwendung kommen verschiedene Abstandsfunktionen zum Tragen, wie zum Beispiel beim *Record Linkage* die Jaro-Winkler-Metrik [55, 56, 133] oder andere wie in [26] beschrieben.

Meist wird jedoch die Abstandsfunktion im Sinne eines *Edit-Distanz-Modells* (welches auch Grundlage dieser Arbeit ist) weiter eingeschränkt.

Dazu wird die Transformation aufgefasst als eine Reihe nacheinander angewandter Operatoren der Art $\delta : \Sigma^* \times \Sigma^* \rightarrow \mathbb{R}^0_+$. Jeder dieser endlich vielen Operatoren gibt die Kosten für die Transformation eines Substrings in einen anderen an. Wichtig ist dabei, dass jeder Substring nur einmal mit einer Operation verwendet werden darf, da ansonsten allgemeine *Termersetzungssysteme*, auch *Rewriting Systems* genannt, ermöglicht würden und so im Allgemeinen die Distanz zweier Strings nicht berechenbar wäre. Die Kosten der Abstandsfunktion ergeben sich nun einfach als Summe der einzelnen angewandten Operatoren.

Eine derartige Abstandsfunktion erfüllt offensichtlich immer die Dreiecksungleichung und die Definitheit. Damit die Abstandsfunktion eine Metrik auf dem Raum der Strings darstellt, muss nur noch die Symmetrieforderung erfüllt sein. Die Symmetrieeigenschaft der Abstandsfunktion läßt sich direkt auf die Operatoren zurückführen. Sind die Operatoren also von der Art $\delta(u,v) = \delta(v,u)$, so stellt die Abstandsfunktion eine Metrik dar.

Die konkrete Wahl der Operatoren ist nicht ganz unabhängig von der jeweiligen Anwendung, und so wird beispielsweise eine Form der allgemeinen Substringersetzung bei der Korrektur phonetischer Fehler eingesetzt [143]. In den meisten Anwendungen werden jedoch die Operatoren auf Substrings der Maximallänge 1 eingeschränkt (also auf einzelne Buchstaben und den leeren String). Dabei entsteht folgender Satz möglicher Operatoren:

- Einfügen $\delta(\epsilon, a)$: Einfügen des Symbols a.
- Löschen $\delta(a, \epsilon)$: Löschen des Symbols a.
- Ersetzung $\delta(a,b)$ für $a \neq b$: Ersetzen des Symbols a durch das Symbol b.

Manchmal wird explizit noch ein weiterer Operator definiert, der benachbarte Buchstabenvertauschungen, typischerweise für Tippfehler interessant, in Betracht zieht (anders als die oben genannten, wirkt dieser Operator allerdings auf Substrings der Länge 2):

- Transposition $\delta(ab, ba)$ für $a \neq b$: Austauschen der benachbarten Symbole a und b.

Auf diesen Operatoren (die den von Damerau [30] identifizierten vier Klassen der häufigsten Schreibfehler entsprechen) basieren die am häufigsten genutzten Abstandsfunktionen, die im Folgenden kurz erläutert werden.

> *Episode-Distanz*: Die Episode-Distanz erlaubt nur das Einfügen (mit konstanten Kosten von üblicherweise 1) und ist damit nicht symmetrisch. Sie dient dazu, in einem Datensatz von Ereignissen (Text) innerhalb eines „Zeitfensters“ (erlaubte Fehler k) eine Sequenz (Pattern) zu suchen (beispielsweise also um Problemsequenzen in einer Reihe von Alarmmeldungen in einem Kommunikationsnetzwerk zu identifizieren). Die sich daraus ergebende Variante des ASM-Problems wird auch *Episode Matching* genannt [31].

> *Longest-Common-Subsequence-Distanz*: Diese erlaubt neben dem Einfügen noch das Löschen, beides mit konstanten Kosten von 1, und ist damit symmetrisch. Sie dient der Bestimmung der längsten Teilsequenz, die zugleich in Pattern und Text enthalten ist (die Buchstabenreihenfolge bleibt erhalten). Dabei wird die Anzahl der Stellen als Distanzmaß geliefert, an denen in Text oder Pattern kein zur Teilsequenz gehöriger Buchstabe steht. Die sich daraus ergebende Variante des ASM-Problems wird auch das *Longest-Common-Subsequence-Problem* genannt [8, 52].

> *Hamming-Distanz*: Diese Abstandsfunktion erlaubt nur Ersetzungen mit konstanten Kosten von 1 und ist dadurch symmetrisch. Zudem führt das Fehlen von Einfüge- oder Löschoperatoren dazu, dass dieses Maß nur für Strings gleicher Länge wohldefiniert ist. Die sich daraus ergebende Variante des ASM-Problems wird auch *string matching with k mismatches* genannt [104].

> *Edit-Distanz*: Die Edit-Distanz erlaubt Einfüge-, Lösch- und Ersetzungsoperatoren. Normalerweise sind die Kosten für diese Operatoren konstant 1 (sofern nicht explizit andere Kosten angegeben werden) und in diesem Fall wird die Abstandsfunktion auch *Levenshtein-Distanz* genannt [70]. Mit konstanten Kosten ist diese Abstandsfunktion in jedem Fall symmetrisch, andernfalls hängt dies von den genauen Kosten ab. Die sich (bei Kosten 1) ergebende Variante des ASM-Problems wird auch *string matching with k differences* genannt.

Neben diesen üblichen Abstandsfunktionen sind auch andere möglich. So können auch die Operatoren $\delta(.,.)$ einfach über eine Kostenmatrix mit Einträgen für jede Buchstabenkombination definiert werden. Beispielsweise wird dies bei der Bestimmung von evolutionären Distanzen zwischen Proteinen mit Hilfe der PAM-Matrizen für Aminosäuren gemacht [32, 33]. Allerdings werden in diesem konkreten Fall nicht die (minimalen) Kosten, sondern eine (maximale) Ähnlichkeit bestimmt, wobei das Problem aber das Gleiche bleibt.

2.4 Der Fehlerlevel

Die Betrachtungen in dieser Arbeit beschränken sich (sofern nicht explizit anders vermerkt) auf die Verwendung der Levenshtein-Distanz als Abstandsfunktion. Für den Fehlerparameter k beim ASM ergibt sich mit dieser Abstandsfunktion die Einschränkung $0 < k < m$. Der Fall $k = 0$ entspricht der exakten Suche des Pattern im Text und fällt aus dem Rahmen dieser Arbeit. Der Fall $k \geq m$ ist nicht sinnvoll, weil mit m Transformationsoperationen in jedem Fall das ganze Pattern vollständig zu einer beliebigen Textstelle derselben Länge transformiert werden kann. Mit dieser Einschränkung für k kann ein *Fehlerlevel* definiert werden:

Definition 2.4.1 (Fehlerlevel (error level))
Für das Problem des Approximate String Matching unter Verwendung der Levenshteindistanz ist der *Fehlerlevel* α definiert als

$$\alpha = \frac{k}{m}.$$

α beschreibt das Verhältnis zwischen erlaubten Fehlern und exakten Übereinstimmungen im gesuchten Approximate Matching und es gilt $0 < \alpha < 1$.

2.5 Problemparameter in der Praxis

In der Praxis ist ein Fehlerlevel von beinahe 1 relativ uninteressant. Wirklich sinnvolle Parameter hängen von der jeweiligen Anwendung ab, dennoch soll im Folgenden ein Rahmen gesteckt werden, für das, was hier unter „praktisch“ verstanden wird:

- Die Größe σ des zugrunde liegenden Alphabets bewegt sich in der Regel zwischen 4 (z. B. bei DNS-Daten) und 256 (z. B. volle Bytegröße bei Datenkompression). Jedoch sind auch leicht größere Alphabete durchaus noch praxisrelevant (z. B. asiatische Alphabete).

- Die Länge n des Textes bewegt sich im Bereich von ein paar wenigen Tausend (wie es in der Computational Biology der Fall ist) bis hin zu mehreren Millionen oder gar Milliarden Symbolen (wie es eher im Text Retrieval der Fall ist).

- Die Länge m des Pattern schwankt in der Praxis zwischen sehr kurz (ab ungefähr 5, was beim Text Retrieval durchaus vorkommt) und recht lang (in der Größenordnung von einigen Hundert Symbolen, wie z. B. in der Computational Biology).

- Die Anzahl zugelassener Fehler k überschreitet in der Regel nicht die halbe Patternlänge (Suchanfragen mit $k > m/2$ machen recht wenig Sinn). Der Fehlerlevel ist damit eher gering bis moderat.

2.6 Klassifikation von Lösungsalgorithmen

Gewissermaßen aus der Anwendbarkeit in der Praxis definiert sich eines der wichtigsten Kriterien zur Kategorisierung von Algorithmen zum Approximate String Matching. So wird in der Regel zwischen *Online*- und *Offline*-Algorithmen unterschieden. Besitzt ein Algorithmus die Online-Fähigkeit, so bedeutet dies, dass er eine Suche sofort durchführen kann, ohne Vorberechnungen auf dem Text durchführen zu müssen. Benötigt ein Algorithmus hingegen Vorberechnungen auf dem Text (die bei großen Datenmengen auch lange dauern können), so handelt es sich um einen Offline-Algorithms. Durch die Durchführung von vorbereitenden Operationen auf dem Text, versuchen Offline-Algorithmen eine schnellere Suchphase zu erreichen, so dass sich der Einsatz eines solchen Algorithmus bei einem großen Text oder häufigen Suchanfragen lohnen kann.

Das Merkmal der Online-Fähigkeit wird in dieser Arbeit nicht zur Klassifikation der Algorithmen verwendet, jedoch wird bei den Algorithmen dieses Merkmal vermerkt.

Ein weiteres übliches Klassifikationsmerkmal ist das Grundprinzip des Algorithmus. Üblicherweise werden hier die Prinzipien *DP-basiert* (auf einer Matrix der Dynamischen Programmierung basierend), *Automaten* und *Filter* unterschieden. Daneben hat sich mittlerweile noch die Klasse der *bitparallelen* Algorithmen etabliert. Diese findet jedoch in dieser Arbeit so keine Beachtung, da es sich bei diesen Algorithmen um geschickt programmierte andere Algorithmen handelt, wobei durch das Ausnutzen der Wortverarbeitung des Prozessors eine teilweise parallele Bearbeitung erreicht werden kann.

In dieser Arbeit wird eine Grundklassifikation nach den genannten Merkmalen DP-basiert (vgl. Kapitel 3.1), automatenbasiert (vgl. Kapitel 3.2) und filterbasiert (vgl. Kapitel 4.1) vorgenommen. Jegliche Art von parallelem Algorithmus wird hingegen nicht behandelt.

Eine ganz eigene Klasse an Algorithmen für das Approximate String Matching Problem soll hier auch nur erwähnt, aber dann nicht weiter betrachtet werden: die Klasse der heuristischen oder approximativen Algorithmen. Die Algorithmen dieser Klasse garantieren nicht, dass jede Lösung gefunden wird, sind dafür in der Regel aber sehr schnell und finden so in der Praxis oft Verwendung [4, 5, 19, 40, 98, 99].

2.7 Zur Statistik des Problems

Eine ausführliche Darstellung der wichtigen theoretischen Ergebnisse zur Statistik des Approximate String Matching findet sich in [85]. An dieser Stelle sollen nur die für diese Arbeit interessanten Ergebnisse zusammengefasst werden.

Die theoretischen Resultate basieren für die Modellierung des Problems auf zwei Voraussetzungen:

- Für den Entstehungsprozess der Strings wird ein einheitliches Bernoulli-Modell angenommen. Das bedeutet, dass an jeder Stelle des Strings die Wahrscheinlichkeit für jeden Buchstaben des Alphabets gleich, also $1/\sigma$, ist.

- Zur Bestimmung des Abstands zwischen zwei Strings findet die Edit-Distanz Verwendung.

Trotz dieser vereinfachenden Annahmen, erwiesen sich die folgenden Ergebnisse als recht zuverlässig in der Praxis.

Gegeben seien zwei Strings S_1 und S_2 der Länge m. Für die durchschnittliche Edit-Distanz dieser beiden Strings, $ed(S_1, S_2)$, konnte gezeigt werden, dass

$$m(1 - \frac{e}{\sqrt{\sigma}}) < ed(S_1, S_2) < 2m(1 - \frac{1}{\sqrt{\sigma}})$$

gilt. Es gibt zwar keinen Beweis, aber es besteht die starke Vermutung, dass der wahre Wert der durchschnittlichen Edit-Distanz $m(1 - 1/\sqrt{\sigma})$ beträgt.

Eine andere wichtige Aussage betrifft die Wahrscheinlichkeit eines Approximate Matching $f(m, k)$, also die Wahrscheinlichkeit dafür, dass ein zufälliges Pattern der Länge m an einer Stelle im gegebenen Text mit höchstens k Fehlern auftritt (auch hier kennzeichnet die Stelle wieder das Ende des Approximate Matching).

Es wurde gezeigt, dass es einen maximalen Fehlerlevel α^* gibt, bis zu dem die Wahrscheinlichkeitsfunktion $f(m, k)$ für wachsendes m exponentiell abnimmt, dass also

$$f(m, k) = O(\gamma^m) \text{ für } \alpha < \alpha^* \text{ mit einem } \gamma < 1$$

gilt.

Dies ist insbesondere für Algorithmen von Bedeutung, die das Filterprinzip nutzen (siehe Kapitel 4.1). Derartige Algorithmen haben eine *Verifikationsphase*, in der mögliche Approximate Matchings überprüft werden. Die Überprüfung hat polynomielle (typischerweise quadratische) Kosten in m. Wenn also diese Überprüfung nur mit einer Wahrscheinlichkeit von $O(\gamma^m)$ für ein $\gamma < 1$ aufgerufen wird, so belaufen sich die Gesamtkosten der Verifikation auf $O(m^2\gamma^m) = o(1)$ und sind von daher vernachlässigbar.

Der maximale Fehlerlevel α^* konnte bisher nicht exakt berechnet werden, jedoch existieren für ihn eine untere und eine obere Schranke:

$$1 - \frac{e}{\sqrt{\sigma}} < \alpha^* \leq 1 - \frac{1}{\sigma}$$

Im Rahmen von experimentellen Überprüfungen konnte ein maximaler Fehlerlevel festgestellt werden, der immer sehr nahe bei $1 - 1/\sqrt{\sigma}$ liegt, was dem für den maximalen Fehlerlevel vermuteten Wert von Sankoff und Mainville [105] entspricht.

2.8 Die „compressed“-Variante des Problems

Eine Variante des Approximate String Matching ist das *Compressed Approximate String Matching*. Im Gegensatz zum Approximate String Matching wird hier als Eingabe ein komprimierter Text erwartet. Wird die Kompression als spezieller Vorverarbeitungsschritt betrachtet, so kann Compressed Approximate String Matching als ein Spezialfall der Offline-Algorithmen zum Approximate String Matching gesehen werden.

Es gibt eine große Vielfalt an Kompressionsmethoden. Von Kida et al. [60] wurde ein allgemeines Collage-System eingeführt, das Wörterbuch-basierte Kompressionsmethoden wie beispielsweise die verschiedenen Lempel-Ziv Varianten [132, 139, 140], BPE (byte pair encoding) [37], Sequitur [97], Re-Pair (recursive pairing) [69] und Lauflängenkodierung in einem gemeinsamen Modell vereint. Dieselben Autoren zeigten zugleich einen allgemeinen Algorithmus für die exakte Suche in komprimierten Texten (innerhalb dieses Modells) auf, jedoch wurde das Problem des (Compressed) Approximate String Matching nicht aufgegriffen.

Kärkkäinen et al. [62] präsentierten den ersten Lösungsalgorithmus für dieses Problem. Ihr Algorithmus arbeitet auf LZ78 [140] und LZW [132] komprimierten Texten verwendet einen Ansatz mit Dynamischer Programmierung. Es wird eine Laufzeit von $O(mk\bar{n} + r)$ erreicht, wobei $\bar{n}$ die Länge des komprimierten Textes und r die Anzahl der vorhandenen Approximate Matchings ist.

Auf denselben Kompressionsschemen basiert auch der Algorithmus von Matsumoto et al. [77]. Mit der Verwendung bitparalleler Techniken wird eine Laufzeit von $O(k^2\bar{n} + km)$ erreicht.

Mit einem Filter-Ansatz erzielen Navarro et al. [91] ein besseres durchschnittliches Verhalten. Dazu werden Stücke des Patterns exakt im komprimierten Text gesucht und immer dann, wenn eine genauere Überprüfung notwendig erscheint, der Text zur Verifikation lokal dekomprimiert.

Speziell für das Kompressionsschema der Lauflängenkodierung wurde von Mäkinen et al. [79] ein Algorithmus entworfen. Dieser Algorithmus ist im Gegensatz zu allen anderen genannten nicht an die Levenshtein-Distanz gebunden und hat dabei eine Laufzeit von $O(m\bar{n}\bar{m})$, wobei $\bar{m}$ die Länge des komprimierten Patterns ist.

Für andere Kompressionsschemen ist das Problem des Compressed Approximate String Matching noch nicht gelöst.

3 Approximate String Matching - direkte Lösungen

Der Weg ist das Ziel.

(Konfuzius, Chinesischer Philosoph, 551-479 v. Chr.)

Der in Kapitel 2.6 gegebenen Klassifikationsstruktur folgend, zeigen dieses und das folgende Kapitel verschiedene Lösungsprinzipien und die wichtigsten Lösungen für das Problem des Approximate String Matching auf.

Dieses Kapitel beschränkt sich dabei auf Prinzipien, die das Problem des Approximate String Matching direkt angehen. Kapitel 3.1 stellt den allgemeinen auf Dynamischer Programmierung basierenden Lösungsansatz und direkt darauf aufbauende Varianten vor. In Kapitel 3.2 wird dann die Klasse der Automaten-Algorithmen näher betrachtet.

Weiterführende Informationen über die in diesem Kapitel diskutierten Algorithmen können selbstverständlich den Original-Artikeln entnommen werden. Insbesondere eigenen sich dazu aber auch verschiedene Bücher (wie z. B. [35, 48, 104, 130]) und Übersichtsartikel (wie z. B. [38, 40, 41, 49, 58, 80, 82, 85, 90, 98, 122, 123]), die bis zu einem gewissen Grad sogar recht gut die historische Entwicklung dieses Arbeitsgebietes widerspiegeln.

3.1 Lösungsalgorithmen basierend auf Dynamischer Programmierung

Das Prinzip der *Dynamischen Programmierung (DP)*, 1957 von Richard Bellman vorgestellt [17, 34], besteht darin, ein Problem in Teilprobleme zu zerlegen und diese wieder geeignet zusammenzufügen. Dabei ist es notwendig, dass eine optimale Lösung des Problems einer bestimmten Größe aus optimalen Lösungen von Teilproblemen geringerer Größe zusammengesetzt werden kann (dies wird oft auch *Bellmannsches Optimalitätsprinzip* genannt).

Üblicherweise werden die Lösungen aller zu einem Problem der Größe n gehörenden Teilprobleme der Größen $< n$ in einer Matrix gespeichert (*DP-Matrix*). Die Berechnung der Lösung für das Problem der Größe n wird dann direkt aus den in der Matrix verfügbaren Lösungen konstruiert und ebenso in der Matrix abgelegt, wodurch dann eine einfache iterative Konstruktion einer Lösung für das Problem jeder Größe möglich ist.

Im Folgenden werden verschiedene auf diesem Prinzip basierende Lösungen für das ASM vorgestellt.

3.1.1 Das allgemeine DP-Lösungsverfahren

Die Wurzeln der grundsätzlichen Lösung des ASM-Problems liegen beim *Sequence Alignment*, einem dem ASM sehr nahe verwandten Problem. Das Sequence Alignment beschäftigt sich mit der Anordnung zweier Sequenzen zueinander, wobei mit Anordnung gemeint ist, dass gleiche Symbole gegenübergestellt werden und ungleiche durch eine der Operationen Einfügen, Löschen und Ersetzen entsprechend angepasst werden. Ein *Alignment* entspricht also einer Folge von Operationen, die einen String in einen anderen überführt. In gewisser Hinsicht betrachtet das Sequence Alignment also auch direkt die Ähnlichkeit zweier Zeichenketten.

Der dazu einfachste Ansatz, üblicherweise *Dot-Matrix* oder auch *Dot Plot-Matrix* genannt, wurde von Gibbs and McIntyre [43] beschrieben und arbeitet rein graphisch (siehe Abbildung 3.1), ist aber vor allem in der einfachsten Variante eine Vorstufe der Lösung mit Dynamischer Programmierung. Die beiden Achsen einer Matrix repräsentieren die beiden Sequenzen $T = t_1 t_2 \cdots t_n$ and $P = p_1 p_2 \cdots p_m$. Eine Matrix-Position (i, j) wird immer mit einem Punkt markiert, wenn die zwei Substrings $t_i \cdots t_{i+k-1}$ and $p_j \cdots p_{j+k-1}$ in mindestens c aufeinander folgenden Symbolen identisch sind (d. h. Transformationsoperationen werden nicht erfasst). Der Parameter $k \geq 1$ gibt eine Wort- oder Fensterbreite vor, während c, $1 < c \leq k$, einen Schwellenwert für die graphische Darstellung angibt.

	x	y	x	y	z	z	x	y
z					●	●		
x	●		●				●	
y		●		●				●
z					●	●		
x	●		●				●	
y		●		●				●

Abbildung 3.1: Dot Plot-Matrix für die Strings `xyxyzzxy` und `zxyzxy` und $k = 1, c = 1$.

Nicht graphisch, sondern auf der Basis von Operatorkosten und Dynamischer Programmierung arbeitet das Verfahren zur Bestimmung eines *globalen Alignments*, dessen ursprüngliche Bestimmung es war, eine konkrete Sequenz an Transformationsoperationen zu finden, die einen String in einen anderen überführt. Dieses Verfahren selbst berechnet tatsächlich nur den Wert einer solchen Anordnung, d. h. die Kosten der Sequenz von Transformationsoperatoren. Die genaue Sequenz für die Transformation wird dann, sofern nötig, in einem separaten Schritt berechnet.

Der Algorithmus zur Bestimmung eines globalen Alignments wurde mehrfach unabhängig voneinander entwickelt, am bekanntesten jedoch ist die Variante von Needleman und Wunsch [94]. Diese ursprüngliche Variante benötigte noch eine Laufzeit von $O(nm^2)$, während in verbesserten Varianten eine Laufzeit von $O(nm)$ bei sogar allgemeineren Metriken erreicht wurde [46, 108, 131].

Im Gegensatz zu globalen Alignments betrachten *lokale Alignments* nur lokale Ähnlichkeiten zwischen zwei Strings, es wird also versucht, die beiden Substrings mit der größten Ähnlichkeit zu bestimmen. Bei der generellen hierzu von Smith und Waterman [112] vorgestellten Lösung handelt es sich eine Variation des Algorithmus für globales Alignment.

Sowohl für das lokale als auch das globale Alignment wird davon ausgegangen, dass beide Strings in etwa die gleiche Länge haben. Sellers [109] zeigte eine Variation dieser Algorithmen, die eine Such-Variante darstellt. Dabei wird davon ausgegangen (aber nicht zwingend erwartet), dass der zu suchende String (das Pattern) deutlich kürzer ist als der Textstring. Der Algorithmus bestimmt die Ähnlichkeit des Patterns an jeder Textstelle.

Im Folgenden wird anhand der Such-Variante (als Grundlage für das Approximate String Matching) der prinzipielle Algorithmus erläutert. Als Distanzmaß soll dabei eine allgemeine Funktion $d(a, b)$ gelten, die die Kosten für die Ersetzung des Symbols a durch das Symbol b angibt. Dabei ist das Einfügen eines Symboles a mit den Kosten $d(\epsilon, a)$ und das Löschen mit $d(a, \epsilon)$ verbunden.

Die Idee hinter der Lösung ist die Zerlegung des Problems in kleinere Teilinstanzen, die zu größeren Instanzen zusammengesetzt werden (Dynamische Programmierung). Hierzu wird das ASM-Problem erst ein wenig erweitert, indem der Restriktionswert k vernachlässigt wird (d. h. $k = \infty$). Für das so resultierende Problem wird für jede Stelle in T der Wert des besten Approximate Matching bestimmt, das dort endet. Für die Zerlegung in Teilprobleme werden die beteiligten Strings in der Länge eingeschränkt. Dazu sei $D(i, j)$ definiert als die minimale Distanz (unter einer gegebenen Abstandsfunktion d) zwischen $P_j = p_1 \cdots p_j$ und einem Approximate Matching, das in t_i endet (betrachtet wird also $T_i = t_1 \cdots t_i$). Für die Berechnung dieser Distanz sind drei Fälle von Bedeutung:

- $D(i-1, j-1)$ und die Angleichung von t_i und p_j
- $D(i-1, j)$ und die Angleichung von t_i mit dem leeren Wort (d. h. Löschen von t_i)
- $D(i, j-1)$ und die Angleichung des leeren Worts mit p_j (d. h. Einfügen von p_j)

Da an jeder Stelle das Approximate Matching mit dem minimalen Abstand gefunden werden soll, ergibt sich daraus auch sofort eine Vorschrift für das Zusammenfügen der Teilprobleme. Mit zusätzlicher Betrachtung der initialen Teilprobleme bei leerem T und P ergibt sich eine Rekursionsgleichung, die jede Probleminstanz löst ($i = 0, \ldots, n, j = 0, \ldots, m$):

$$D(i,0) = 0 \tag{3.1.1}$$

$$D(0,j) = \sum_{k=1}^{j} d(\epsilon, p_k) \tag{3.1.2}$$

$$D(i,j) = \min \begin{cases} D(i-1,j) + d(t_i,\epsilon) \\ D(i-1,j-1) + d(t_i,p_j) \\ D(i,j-1) + d(\epsilon,p_j) \end{cases} \tag{3.1.3}$$

Die über diese Rekursionsgleichung gewonnen Ergebnisse werden üblicherweise in der DP-Matrix dargestellt. Abbildung 3.2 verdeutlicht allgemein die derartige DP-Matrix für diese Rekursionsgleichung.

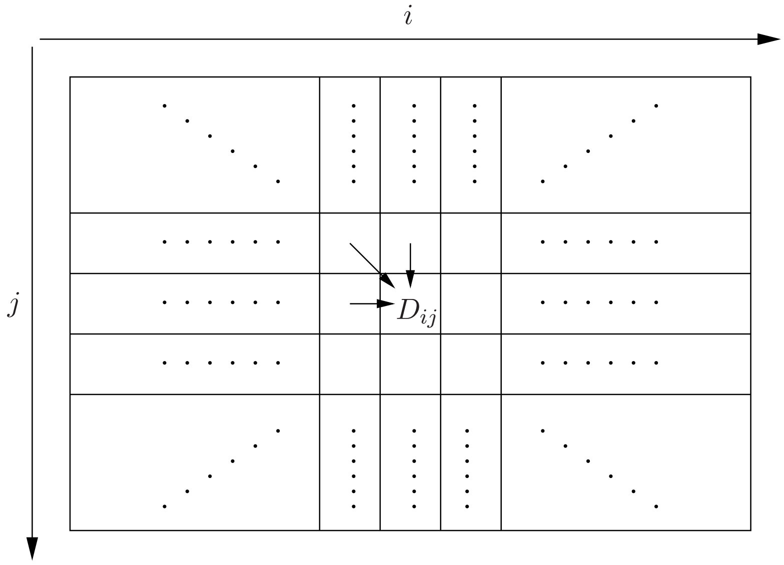


Abbildung 3.2: Die allgemeine DP-Matrix. Die Abhängigkeiten bei der Berechnung des Wertes einer Zelle $D_{ij} = D(i,j)$ sind durch Pfeile gekennzeichnet.

Abbildung 3.3 zeigt beispielhaft eine unter der Levenshtein-Distanz ($d(a,a) = 0$, $d(a,b) = 1$ für $a \neq b$) berechnete DP-Matrix. Der dort exemplarisch aufgezeigte Berechnungsweg zeigt zugleich auch das Approximate Matching, welches mit den Kosten 2 an der vorletzten Stelle von T endet. Eine genaue Betrachtung des Weges zeigt eine anfängliche Auftrennung, was besagt, dass es genau genommen zwei verschiedene Approximate Matchings mit dem gleichen Wert gibt, die an dieser Stelle enden. Das erste dieser beiden Approximate Matchings beginnt hinter dem `b` und ersetzt dort das `e` durch ein `h`.

Die folgenden `e` und `r` werden beibehalten, ein `d` eingefügt und abschließend wieder das `e` beibehalten. Das zweite Approximate Matching beginnt ein Zeichen später mit dem Einfügen eines `h`, während alle folgenden Operationen denen des ersten Approximate Matching entsprechen.

		e	r	d	b	e	e	r	e	n
	0	0	0	0	0	0	0	0	0	0
h	1	1	1	1	1	1	1	1	1	1
e	2	1	2	2	2	1	1	2	1	2
r	3	2	1	2	3	2	2	1	2	2
d	4	3	2	1	2	3	3	2	2	3
e	5	4	3	2	2	2	3	3	2	3

Abbildung 3.3: DP-Matrix für T = `erdbeeren` und P = `herde`. Die Initialzellen (Gleichungen 3.1.1 und 3.1.2) sind dunkelgrau unterlegt. Die hellgrau unterlegten Zellen zeigen die Ergebniszeile. Die Werte dort geben die Kosten eines Approximate Matching an, das an der entsprechenden Stelle in T endet. Exemplarisch ist zudem der Berechnungsweg für einen dieser Werte aufgezeigt (punktiert).

Die Lösung des ASM-Problems mittels dieser DP-Matrix Methode hat eine Laufzeit von $O(nm)$ und einen Platzaufwand von $O(m)$, da immer nur zwei Spalten der Matrix zugleich im Speicher behalten werden müssen, um die Endpositionen der Approximate Matchings mit Kosten $< k$ auszugeben.

Der Vollständigkeit halber sollen an dieser Stelle kurz die genauen Unterschiede der oben erwähnten Problemvarianten der Bestimmung von globalen und lokalen Alignments genau genannt werden. Ursprünglich wurde bei beiden Problemvarianten eine Ähnlichkeit anstelle von Kosten betrachtet, was eine duale Problembeschreibung ist. Dabei bedeutet ein höherer Wert (d. h. die Kostenfunktion bewertet anders) dann eine größere Ähnlichkeit, es muss also maximiert statt minimiert werden. Im Folgenden sind diese Problemvarianten jedoch auf das bekannte Kostenmodell transferiert.

Für die Variante des globalen Alignments wird eine andere Initialisierung vorgenommen, da in jedem Fall immer der ganze Text in die Bestimmung des Approximate Matching mit einfließen muss. Gleichung 3.1.1 wird also ersetzt durch:

$$D(i,0) = \sum_{k=1}^{i} d(t_i, \epsilon)$$

Zudem wird nur ein globales Ergebnis gesucht, welches sich dann beim Wert $D(n, m)$ ablesen lässt.

Für die Variante des lokalen Alignments werden die beiden Initialisierungsgleichungen 3.1.1 und 3.1.2 konstant 0, da ein lokales Alignment schließlich an jeder beliebigen Stelle in T und P beginnen darf. Wichtig ist, dass die Kostenfunktion negative Kosten für wirklich gute Anordnungen und positive Kosten für nicht zu einander passende Symbole vorsehen muss. Zusätzlich wird die Rekursionsgleichung 3.1.3 um die Möglichkeit erweitert, jederzeit ein neues lokales Alignment zu beginnen:

$$D(i,j) = \min \begin{cases} 0 \\ D(i-1,j) + d(t_i, \epsilon) \\ D(i-1,j-1) + d(t_i, p_j) \\ D(i,j-1) + d(\epsilon, p_j) \end{cases}$$

Um das beste lokale Alignment zu erhalten, muss dann noch das Minimum aller Teilergebnisse in der DP-Matrix gefunden werden.

3.1.2 Die „Four Russians"-Technik

Eine der bekanntesten Verbesserungen des allgemeinen Ansatzes mit Dynamischer Programmierung stammt von Masek und Paterson [74, 75] und basiert auf der unter dem Namen „Four Russian"-Technik bekannt gewordenen Idee von Arlazarov et al. [9] [1]. Dieser Ansatz wurde ursprünglich zur Berechnung der Levenshtein-Distanz zwischen zwei Strings konzipiert (was den Kosten eines globalen Alignments unter Verwendung der Levenshtein-Distanz entspricht), kann jedoch leicht über eine veränderte Initialisierung (siehe Kapitel 3.1.1) zur Lösung des ASM-Problems verwendet werden.

Im Wesentlichen wird die DP-Matrix in quadratische Blöcke mit der Seitenlänge r, *r-Blöcke* genannt, unterteilt. Auf der Matrix der r-Blöcke wird dann mittels Dynamischer Programmierung das Problem gelöst.

Jeder r-Block hängt von Abschnitten der Länge r in T und P ab (siehe Abbildung 3.4). Aufgrund jeder beliebigen Kombinationsmöglichkeit in den Strings sind σ^{2r} verschiedene r-Blöcke möglich.

Zudem bestehen weitere Abhängigkeiten von den Endzellen bestimmter Nachbarblöcke (siehe Abbildung 3.4). Die Feststellung, dass unter der Levenshtein-Distanz benachbarte Zellen maximal um 1 variieren, ermöglicht eine Darstellung der Endzellen als Differenzen und somit nur die drei möglichen Werte $0, 1$ und -1 pro Zelle. Hieraus resultieren 3^{2r} Möglichkeiten für verschiedene r-Blöcke.

[1] Die Benennung der Technik als „Four Russians"-Technik ist dennoch aus verschiedenen Gründen falsch:

1. Nur einer der vier Autoren ist Russe.
2. Nach üblichen akademischen Standards erfolgt die Ehre der Benennung nach dem Autorennamen.
3. Der Artikel ordnet eindeutig das grundlegende Zerlegungslemma allein Kronrod zu.

Insgesamt gibt es also $(3\sigma)^{2r}$ mögliche verschiedene r-Blöcke, die bei einer Wahl von $r = \frac{1}{2}\log_{3\sigma} n$ vorberechnet und mit $O(n)$ zusätzlichem Platz gespeichert werden können. Jeder Block kann dann in konstanter Zeit nachgeschlagen werden und somit ergibt sich eine Laufzeit von $O(nm/r^2) = O(nm/\log_\sigma^2 n)$.

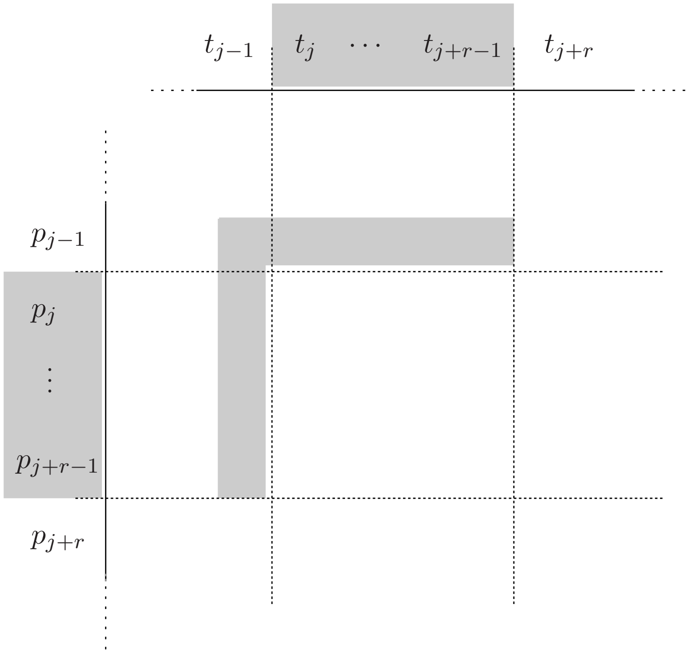


Abbildung 3.4: Abhängigkeiten eines r-Blocks der DP-Matrix. Jeder r-Block ist sowohl von den benachbarten linken und oberen Matrixeinträgen abhängig, als auch von den entsprechenden Abschnitten von T und P (grau unterlegt).

3.1.3 Diagonal-Transition-Algorithmen

Diagonal-Transition-Algorithmen basieren auf der Erkenntnis, dass die Werte auf den Diagonalen der DP-Matrix (unter der Edit-Distanz) linear wachsen. Die daraus folgende Grundidee ist die Berechnung der relevanten Teile der DP-Matrix nur durch Bestimmung der Positionen an der die Diagonalen ihre Werte verändern. Betrachtet werden dabei die *e-Strokes*, d. h. Bereiche auf einer Diagonalen mit dem gleichen Wert e.

3.1.3.1 Das Grundprinzip nach Ukkonen

Der erste Diagonal-Transition-Algorithmus stammt von Ukkonen [122]. Dieser Algorithmus dient der Bestimmung der Edit-Distanz zweier Strings und berechnet dazu die Endspalten der notwendigen Strokes. Der Wert des Strokes in der Diagonalen $n - m$ mit der Endspalte n ist die gesuchte Distanz.

Für die Berechnung der minimal für einen Stroke in Frage kommenden Endspalte wird auf zuvor berechnete Informationen (siehe Abbildung 3.5) und die beiden Strings (siehe Abbildung 3.6) zurückgegriffen.

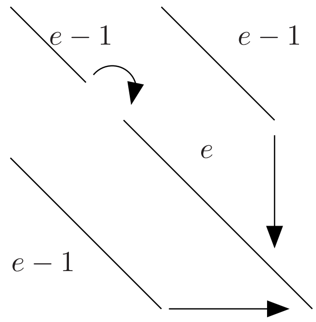


Abbildung 3.5: Berechnung des Startpunkts eines Strokes. Jeder neue Stroke (mit e Fehlern) beginnt automatisch immer direkt am Ende des alten Strokes (mit $e-1$ Fehlern). Unter Verwendung der Informationen aller benachbarten $(e-1)$-Strokes kann jedoch (sofern vorhanden) ein späterer Einstiegspunkt in den e-Stroke berechnet werden. Erst ab diesem Einstiegspunkt muss der e-Stroke näher betrachtet werden.

Durch eine geschickte Auswahl der notwendigerweise zu berechnenden Strokes wird eine Laufzeit von $O(\min(n,m)D(n,m))$ bei ebensolchem Platzbedarf erreicht.

3.1.3.2 Das Prinzip von Landau und Vishkin

Der Ansatz von Landau und Vishkin [66, 67] verbessert den Diagonal-Transition-Algorithmus von Ukkonen (Kapitel 3.1.3.1). Dieser Ansatz ist (wie auch alle folgenden) auf die Lösung des ASM-Problems ausgelegt und betrachtet nicht mehr nur den Abstand zwischen zwei Strings.

Die zentrale Idee ist, dass bei der Berechnung von Approximate Matchings eine Diagonale der DP-Matrix nicht weiter betrachtet werden muss, sobald der k-te Stroke auf dieser Diagonalen endet. Mit dem Beginn des $(k+1)$-Strokes ist es ausgeschlossen, dass ein Approximate Matching mit höchstens k Fehlern auf dieser Diagonalen endet. Zur Berechnung findet eine Rekursionsgleichung Verwendung, die die bekannte DP-Matrix nicht nach Zeilen und Spalten parametrisiert, sondern anhand der Diagonalen und der Fehlerzahl e.

Zudem wird das Teilproblem der Bestimmung der Länge eines Strokes auf die Berechnung der *Matching Statistics* zurückgeführt. Das Berechnen der Matching Statistics

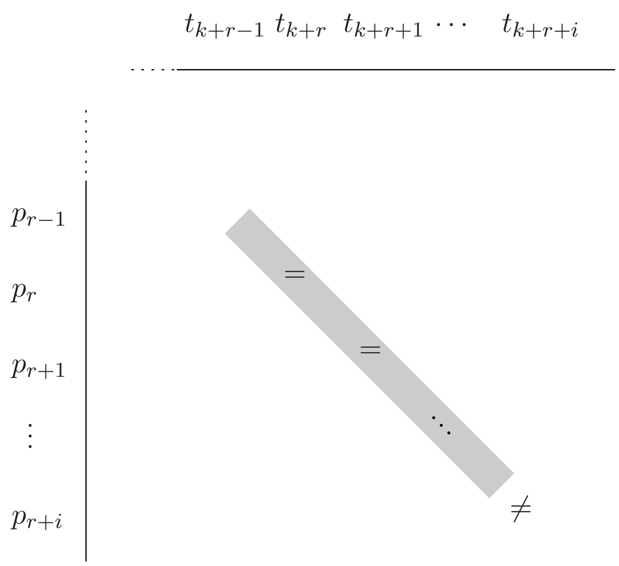


Abbildung 3.6: Berechnung der verbleibenden Länge eines Strokes. r markiert den Startpunkt des Strokes (vgl. Abbildung 3.5), während k einen Verschiebungsoffset darstellt, der vom jeweils gerade akut berechneten Stroke abhängt. Der Stroke wird solange fortgeführt, wie aufeinander folgende Symbole p_{r+j} und t_{k+r+j} $(j \geq 0)$ übereinstimmen.

bedeutet das Bestimmen des längsten gemeinsamen Präfixes der Substrings $P_{j \ldots m}$ und $T_{i \ldots n}$. (Die zuvor bei Ukkonens Algorithmus in Kapitel 3.1.3.1 in Abbildung 3.6 aufgezeigte Berechnung der Länge eines Strokes entspricht auch einer Berechnung der Matching Statistics.)

Landau und Vishkin verwenden zur Berechnung der Matching Statistics einen Suffix-Tree von $T\hat{\$}P\$$, wobei $\hat{\$}$ and \$ eindeutige Trennsymbole darstellen, die für einen einwandfreien Suffix-Tree notwendig sind. Das längste gemeinsame Präfix von Substrings von T und P wird durch einen Knoten im Suffix-Tree repräsentiert, der *Lowest Common Ancestor (LCA)* genannt wird. Der LCA-Knoten ist der letzte gemeinsame Knoten der beiden von der Wurzel des Suffix-Trees ausgehenden Substring-Pfade. Zur Bestimmung des LCA-Knotens verwenden Landau und Vishkin den Algorithmus von Harel und Tarjan [50], der eine LCA-Anfrage in konstanter Zeit beantworten kann.

Somit kann die Länge jedes Strokes in konstanter Zeit berechnet werden und bei maximal k Strokes auf jeder der n Diagonalen ergibt sich eine Laufzeit von $O(kn)$. Der Platzbedarf liegt bei $O(n)$.

Varianten

Durch die Verwendung eines schnelleren LCA-Algorithmus [106], konnten Chang und Lawler den Algorithmus von Landau und Viskin verbessern [21].

Der Algorithmus von Myers [65, 81] verwendet die gleichen Grundlagen wie Landau und Vishkin, jedoch bestimmt er bei einem zusätzlichen Speicheraufwand von $O(n)$ nicht nur die Endpositionen der Approximate Matchings, sondern das Approximate Matching selbst (d. h. auch die Folge der Transformationsoperationen).

Galil und Giancarlo [38] reduzierten den Platzbedarf auf $O(m)$, indem sie die Matching Statistics nur auf Grundlage eines Suffix-Trees des Patterns berechnen. Auch wenn die prinzipielle Laufzeitkomplexität erhalten bleibt, so ist dieser Ansatz in der Praxis deutlich langsamer als der von Landau und Vishkin.

Ausgehend von Ideen von Sahinalp und Vishkin [103] konstruierten Cole und Hariharan [27] eine weitere Variante des Diagonal-Transition-Algorithmus von Landau und Vishkin. Diese Variante nutzt Filter-Ideen (vgl. Kapitel 4.1), um die Anzahl der zu berechnenden Diagonalen zu reduzieren. Der Algorithmus selbst unterscheidet zwischen zwei Fällen und erreicht eine Laufzeit von $O(\frac{nk^3}{m} + n + m)$, wenn das Pattern „größtenteils aperiodsich" ist, und $O(\frac{nk^4}{m} + n + m)$ sonst. In einem Vorberechnungsschritt werden neben der Matching Statistics auch noch so genannte *Breaks*, bestimmte disjunkte und aperiodische Substrings des Patterns, berechnet. Diese Breaks werden im folgenden Schritt verwendet, um Textbereiche festzustellen, in denen Approximate Matchings beginnen können. Im letzten Schritt wird der Ansatz von Landau und Vishkin auf die so identifizierten Textbereiche angewendet.

3.1.3.3 Das Prinzip von Galil und Park

Der Diagonal-Transition-Algorithmus von Galil und Park [39] nutzt einen Suffix-Tree des Patterns P und einen Algorithmus zur Bestimmung des LCA-Knotens, um die längsten übereinstimmenden Präfixe zwischen allen Patternsubstrings $p_k \cdots p_m$ und $p_l \cdots p_m$ (für $1 \leq k, l \leq m$) im Voraus zu berechnen.

Nach dieser Vorberechnung berechnet der Algorithmus nacheinander alle relevanten Strokes der DP-Matrix (relevant sind nur e-Strokes für $e \leq k$). Der Ablauf dieser Berechnung ist schematisch in Abbildung 3.7 dargestellt. Die Strokes in der DP-Matrix werden von links nach rechts gehend berechnet. Dabei werden immer alle Strokes für alle möglichen Fehler berechnet, d. h. den Anfang bildet der am weitesten links liegende 0-Stroke, gefolgt vom 1-Stroke bis hin zum k-Stroke, um dann mit dem nächsten rechts benachbarten 0-Stroke wieder zu beginnen. In einer Liste werden die aktuellen, d. h. zuletzt berechneten, Strokes für jede Fehlerzahl gespeichert (in Abbildung 3.7 durch dicke Linien dargestellt) und zudem wird auch von jedem Stroke die Endspalte festgehalten.

Zur Berechnung der Endspalte eines neuen e-Strokes (in der Abbildung ist dies der 1-Stroke in der Diagonalen d), werden zuerst die benachbarten $(e-1)$-Strokes herangezogen, wie dies schon bei Ukkonens Diagonal-Transition-Algorithmus erläutert wurde (vgl. Kapitel 3.1.3.1). Die so erhaltene Spalte $d + j$ ist garantiert kleiner oder gleich der noch zu bestimmenden Endspalte (irgendwo auf der gestrichelten Linie in der Abbildung). Dann werden die Endspalten der aktuellen Strokes daraufhin überprüft, ob zumindest eine größer ist als die bisher erhaltene Spalte. Ist dies nicht der Fall, so werden Text und Pattern ab dieser Stelle, also ab t_{d+j} und p_j, direkt verglichen. Andernfalls, wenn also eine größere Endspalte vorhanden ist (in der Abbildung mit einem Pfeil gekennzeichnet), repräsentiert dieser Stroke mit der höheren Endspalte ein übereinstimmendes Präfix zwischen $t_{d+j} \cdots t_n$ und $p_i \cdots p_m$. Unter Ausnutzung der vorberechneten Daten (längstes gemeinsames Präfix zwischen $p_i \cdots p_m$ und $p_j \cdots p_m$) kann nun das längste gemeinsame Präfix zwischen $t_{d+j} \cdots t_n$ und $p_j \cdots p_m$, und somit die Endspalte des Strokes, direkt berechnet werden.

Die Endpositionen der Approximate Matching können dann wieder leicht aus der letzten Zeile der DP-Matrix bestimmt werden (bzw. aus den Strokes, die diese Zeile erreichen).

Der Algorithmus besitzt eine Laufzeitkomplexität von $O(kn)$ (für die Vorberechnung ist zuzüglich $O(m \log \min\{m, \sigma\})$ zu zählen) und kommt mit einem Speicherplatz von $O(m)$ aus.

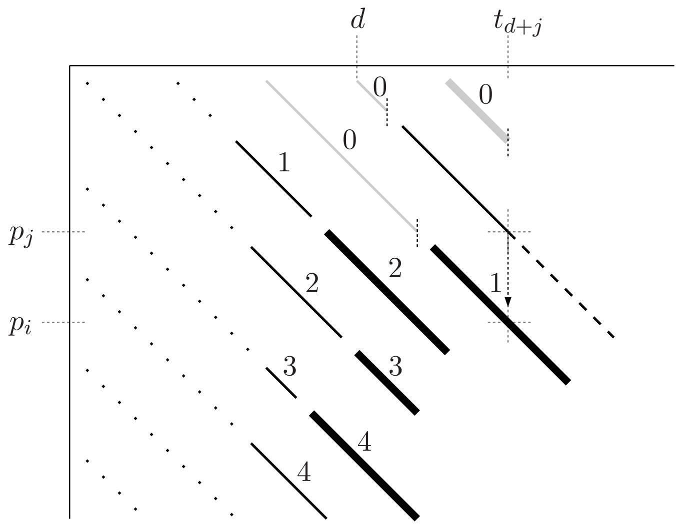


Abbildung 3.7: Der Algorithmus von Galil und Park. Die Länge des gestrichelt dargestellten Strokes wird berechnet. Dabei werden die Informationen der aktuellen Strokes (dicke Linien) verwendet.

Varianten

Ukkonen und Wood präsentierten eine Variante dieses Algorithmus mit veränderter Vorberechnung [127]. Die dort gezeigte Nutzung eines Suffix-Automaten ändert jedoch nichts an der Laufzeit- und Platzkomplexität des Algorithmus.

3.1.4 Cut-Off-Heuristik

Ukkonen [123] erreichte durch Anwendung einer einfachen Heuristik eine Verbesserung der durchschnittlichen Laufzeit beim grundsätzlichen DP-Algorithmus. Während die Laufzeitkomplexität im schlechtesten Fall (worst case) mit $O(nm)$ erhalten bleibt, reduziert sie sich im Durchschnitt (average case) auf $O(kn)$ bei unverändertem Platzbedarf.

Diese so genannte *Cut-Off-Heuristik* nutzt die Eigenschaft der Levenshtein-Distanz, dass in der DP-Matrix bei Einträgen mit Werten $> k$ der genaue Wert (und alle in der Spalte folgenden) keinen weiteren Einfluss auf das Ergebnis hat. Durch Nachhalten des jeweils letzten Spalteneintrags j mit einem Wert $\leq k$ kann oft die Berechnung einer Spalte frühzeitig abgebrochen werden. Dazu wird in der nächsten Spalte der Eintrag $j+1$ überprüft (bis zu diesem muss also berechnet werden). Ist der Wert dort größer als k, so wird oberhalb dieser Matrixzelle der erste Eintrag dieser Spalte mit Wert $\leq k$ gespeichert und Rest der Spalte nicht weiter beachtet. Ist der an der Position $j+1$ der Wert $\leq k$, so ist ab dieser Stelle der Einfluss der vorherigen Spalte nicht mehr vorhanden und der letzte Eintrag mit einem Wert $\leq k$ kann in $O(1)$ direkt berechnet werden. Die Funktionsweise der Cut-Off-Heuristik ist exemplarisch in Abbildung 3.8 gezeigt.

3.1.5 Column Partitioning

Der *Column Partitioning* Algorithmus von Chang und Lampe [20] berechnet die DP-Matrix spaltenweise. Dazu wird der Begriff des *Laufs* definiert, der eine Reihe aufeinander folgender Zellen in einer Spalte bezeichnet, deren Werte von Zelle zu Zelle jeweils um 1 ansteigen (Levensthein-Distanz).

Zunächst wird für jede Patternposition und jeden Buchstaben des Alphabets die Position der nächsten Übereinstimmung im Pattern vorberechnet. Es ergibt sich die Tabelle *loc* mit den Einträgen

$$loc(a, j) = \min\{k | k \geq j \wedge p_k = a\} \quad \forall a \in \Sigma, j = 1, \ldots, m\,.$$

Damit ist es möglich, einen vollständigen Lauf in $O(1)$ zu bestimmen. In der DP-Matrix gilt für $D(i, j)$, dass der Lauf dieser Stelle entweder bei $loc(t_i, j)$ endet, oder dann, wenn ein Lauf in der DP-Matrix an einer der Stellen $(i-1, j-1)$ oder $(i-1, j)$ endete. Abbildung 3.9 zeigt das Column Partitioning an einem Beispiel.

Die durchschnittliche Länge eines Laufes ist $O(\sqrt{\sigma})$. Durch die zusätzliche Verwendung der Cut-Off-Heuristik (vgl. Kapitel 3.1.4) wird eine durchschnittliche Laufzeit (average case) von $O(kn/\sqrt{\sigma})$ bei einem Platzaufwand von $O(m\sigma)$ erreicht.

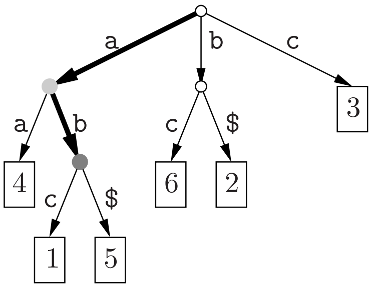
		q	a	c	d	b	d	a
	0	0	0	0	0	0	0	0
q	1	0	1	1	1	1	1	1
a	2	1	0	1	2	2	2	1
w		2	1	1	2	3	3	2
x			2	2	2	3	4	
b				3	3	2	3	

Abbildung 3.8: Anwendung der Cut-Off Heuristik. Bei der Suche von $P =$ `qawxb` in $T =$ `qacdbda` mit maximal $k = 2$ Fehlern (unter der Levenshtein-Distanz) wird für jede Spalte die erste Stelle mit einem Wert $= k$ (hellgrau unterlegt) gespeichert. Zellen, deren Wert sofort als $\geq k$ identifiziert werden kann, brauchen nicht berechnet zu werden (dunkelgrau unterlegt).

loc	a	b	c	d	q	w	x
1	2	5	∞	∞	1	3	4
2	2	5	∞	∞	∞	3	4
3	∞	5	∞	∞	∞	3	4
4	∞	5	∞	∞	∞	∞	4
5	∞	5	∞	∞	∞	∞	∞

		q	a	c	d	b	d	a
	0	0	0	0	0	0	0	0
q	1	0	1	1	1	1	1	1
a	2	1	0	1	2	2	2	1
w	3	2	1	1	2	3	3	2
x	4	3	2	2	2	3	4	3
b	5	4	3	3	3	2	3	4

Abbildung 3.9: Anwendung des Column-Partitionings. Gesucht wird $P =$ `qawxb` in $T =$ `qacdbda`. Die Tabelle *loc* gibt für jede Patternposition und jeden Buchstaben die Position der nächsten Übereinstimmung im Pattern, anhand welcher die Läufe berechnet werden. Gültige erkannte Laufanfänge sind hellgrau unterlegt. Dunkelgrau unterlegte Zellen werden dann nicht mehr berechnet.



		a	b
	0	0	0
a	1	0	1
d	2	1	1
b	3	2	1

Abbildung 3.10: DP-Matrix Berechnung entlang des Suffix-Trees. Gesucht wird $P =$ `adb` in $T =$ `abcaab$` mit der maximalen Fehlerzahl $k = 1$. Links ist der Suffix-Tree für T dargestellt. Die Blätter geben die Position im Text an, an der ein Suffix-Tree-Durchlauf begonnen hat, wenn er in diesem Blatt endet. Die bei der Eingabe von `ab` besuchten Knoten sind gefüllt dargestellt. Jeder Knoten wird assoziiert mit einer Spalte einer DP-Matrix. In der rechten Bildhälfte ist die zur Eingabe `ab` passende Matrix dargestellt. Die zu den im Suffix-Tree grau gefärbten Knoten assoziierten Spalten der DP-Matrix sind entsprechend der Knotenfarbe grau unterlegt.

3.1.6 DP-Matrix Berechnung am Suffix-Tree

Von Ukkonen [125] stammt ein Algorithmus, dessen Grundidee auf den ersten Blick nicht die Berechnung der DP-Matrix involviert. Die Idee ist die Bestimmung einer k-Nachbarschaft des Patterns P, also die Bestimmung aller Strings $\bar{P}$, die zu P einen Abstand $\leq k$ haben, und weiter die Suche aller dieser Strings im Text. Die Größe dieser Nachbarschaft ist exponentiell in k (genauer $O(m^k \sigma^k)$) [83, 123], was diesen Ansatz prinzipiell nur für kleinere Fehlerzahlen und kürzere Pattern interessant sein lässt.

Ukkonens Algorithmus berechnet jedoch nicht die k-Nachbarschaft des Patterns explizit, sondern berechnet diese, soweit notwendig, direkt anhand des Textes. Dazu wird zuerst ein Suffix-Tree des Textes (versehen mit der dafür notwendigen eindeutigen Endmarkierung `$`) konstruiert. Dieser wird dann mit einer Tiefensuche und Backtracking vollständig durchlaufen und dabei wird zu jedem Knoten die entsprechende Spalte einer DP-Matrix (in Bezug auf das Pattern) berechnet (vgl. Abbildung 3.10). Ist der unterste Eintrag einer zu einem Knoten assoziierten Spalte $\leq k$, so entspricht jedes Blatt des hier beginnenden Teilbaumes einem Approximate Matching. Ist jeder Eintrag in der Spalte größer als k, so kann der Teilbaum komplett verworfen werden. In allen anderen Fällen wird der Baum weiter normal durchlaufen.

Der Algorithus von Ukkonen erreicht eine Laufzeit von $O(m^2 \min\{n, m^{k+1}\sigma^k\})$. In [25, 59] und [126] werden verschiedene Verbesserungsvorschläge unterbreitet, die das Nichtbeachten von redundanten Knoten zum Inhalt haben.

3.2 Lösungsalgorithmen basierend auf Automaten

Ein anderer Ansatz, das ASM-Problem zu lösen, besteht darin, die Möglichkeiten eines Approximate Matching mit einem Automaten zu modellieren. Die Symbole des Textes dienen dann als Eingabe-Symbole für den Automaten. Die Ausgabe des Automaten ist dann immer der Kostenwert eines Approximate Matching, welches an der Eingabetextstelle endet.

Im Folgenden werden kurz grundsätzliche automatenbasierte Lösungsalgorithmen vorgestellt.

3.2.1 DP-Matrix-Spalten Automat

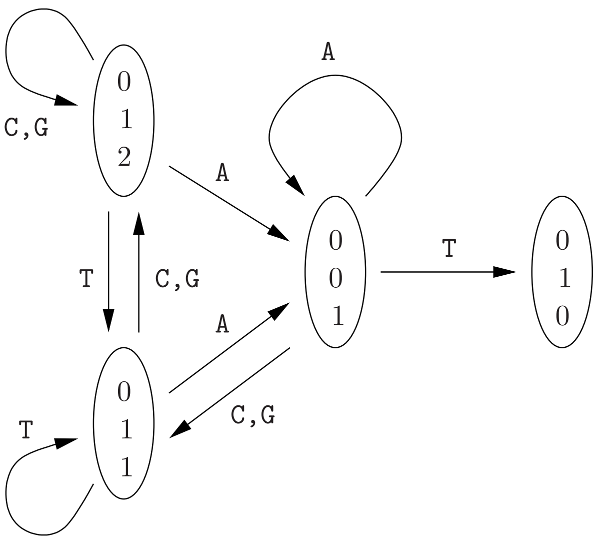
Ukkonens Ansatz [123] ist eine direkte Umsetzung der DP-Matrix (unter der Levenshtein-Distanz) in einen endlichen, deterministischen Automaten (deterministic finite automaton, DFA). Dazu wird für jedes Symbol des Alphabets die Spalte einer DP-Matrix (bzgl. P) vorberechnet. Die Zustände des Automaten werden dann von diesen Spalten gebildet. Der unterste Wert eines solchen Zustands (Spalte) gibt die aktuelle Fehlerzahl an, wenn mittels des Automaten P in T gesucht wird. Das Beispiel in Abbildung 3.11 verdeutlicht das Prinzip eines solchen Automaten.

Tatsächlich finden bei Ukkonen leicht andere Zustände Verwendung. Mittels der Cut-Off-Heuristik (vgl. Kapitel 3.1.4) wird die Anzahl der Zustände reduziert, indem alle Werte auf $k+1$ gesetzt werden, die größer sind als $k+1$. Zudem werden für eine beschleunigte Berechnung und einen geringeren Speicherplatzverbrauch die Zustände in Form von Differenzen benachbarter Einträge gespeichert. So wird aus dem Zustand $(0,1,0)$ beispielsweise der Zustand $(0,1,-1)$.

Insgesamt erreicht der Algorithmus für die Vorberechnungen eine Laufzeitkomplexität von $O(m\sigma \min(3^m, 2^k\sigma^k m^{k+1}))$ und für das Suchen $O(n)$. Der Platzbedarf ergibt sich aus der Vorberechnung und beträgt $O((m+\sigma)\min(3^m, 2^k\sigma^k m^{k+1}))$.

3.2.2 „Four Russians“-Automat

Wu, Manber und Myers [138] verwenden den „Four Russians“-Ansatz (vgl. Kapitel 3.1.2), um Zustände des Automaten und somit Platz zu sparen. Dazu werden die Spalten, die bei Ukkonens Automaten (vgl. Kapitel 3.2.1) durch Zustände repräsentiert sind, in Abschnitte der Größe r unterteilt. Diese Abschnitte bilden dann die Zustände eines Automaten. Obwohl durch diese Modifikation mehr Abhängigkeiten (ein Zustand kann von drei anderen abhängen) pro Zustand zu berechnen sind, wird mit $O(n)$ deutlich weniger Platz während der Vorberechnungen benötigt. Die durchschnittliche Laufzeit (average case) ist $O(kn/\log n)$, während im schlechtesten Fall (worst case) noch mit $O(mn/\log n)$ zu rechnen ist.



		A	G
	0	0	0
A	1	0	1
T	2	1	1

Abbildung 3.11: DFA einer DP-Matrix. Auf der linken Seite ist der DFA dargestellt, dessen Zustände durch Spalten einer DP-Matrix (Symbole des Alphabets $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ gegenüber zu $P = \mathtt{AT}$) gebildet werden. Der Zustand links oben ist der Startzustand des Automaten. Auf der rechten Seite ist die DP-Matrix für P und $T = \mathtt{AG}$ zu sehen, anhand derer der Zusammenhang mit dem Automaten leicht zu sehen ist.

3.2.3 „Lazy Evaluation"-Automat

Kurtz [64] verwendet die so genannte *Lazy Evaluation*, d. h. die verzögerte Berechnung der Zustände eines Automaten. Dazu wird initial nur der Startzustand berechnet und ausgehend von diesem werden jeweils genau die Zustände und Transitionen des Automaten erzeugt, die zur Berechnung der Problemlösung notwendig sind. Diese Technik wurde bereits von Baeza-Yates und Gonnet angewendet [13], jedoch beschränkte ihr Automat sich auf die Hamming-Distanz, während Kurtz die Levenshtein-Distanz betrachtet.

Mit dem Ansatz der Lazy Evaluation werden im Vergleich zu anderen Automaten-Algorithmen einige Zustände niemals erzeugt und somit wird auch entsprechend Platz gespart. Der Algorithmus benötigt insgesamt maximal $O(mn)$ Platz und hat eine Laufzeit von $O(nm + m)$.

4 Approximate String Matching - Lösungen mit dem Filterprinzip

Warum die Anderen ändern,
wenn sie doch schon anders sind?

*(Jeannine Luczak, schweizer. Literaturwissenschaftlerin und Aphoristikerin, *1938)*

In Kapitel 3 wurden bereits verschiedene Lösungen zum Problem des Approximate String Matching vorgestellt. Dieses Kapitel folgt weiter der Klassifikationsstruktur aus Kapitel 2.6 und beleuchtet zunächst in Kapitel 4.1 eingehend das Prinzip des Filterns als Lösungsstrategie und stellt dann Algorithmen vor, die sich dieses Prinzip zunutze machen. Schließlich befaßt sich Kapitel 4.2 mit allgemeinen Methoden, Filteralgorithmen zu verbessern.

4.1 Lösungsalgorithmen basierend auf dem Filterprinzip

Die Erkenntnis, dass es oft einfacher ist für eine Textposition zu entscheiden, dass diese nicht zu einem Approximate Matching gehört, als diese eindeutig einem Approximate Matching zuzuordnen, bildet die Grundlage für das Konzept des Filters.

Algorithmen, die das Filterprinzip nutzen, bestehen immer aus zwei Phasen:

1. In der *Filter-* oder *Suchphase* werden die Textbereiche identifiziert, in denen möglicherweise ein Approximate Matching auftreten kann. Dies kann auch dadurch geschehen, dass Textbereiche verworfen werden, in denen ein Approximate Matching nicht auftreten kann.

2. In der *Prüf-* oder *Verifikationsphase* werden die in der Filter-Phase identifizierten Bereiche mit einem der bekannten Algorithmen zur Lösung des ASM-Problems untersucht.

Die Filter-Phase ist prinzipiell mit deutlich geringerer Laufzeit realisierbar als die Verifikationsphase. Damit ist die Gesamtlaufzeit eines Filter-Algorithmus abhängig von der

Verifikationsphase, welche prinzipiell eine quadratische Laufzeit hat (siehe Kapitel 3.1 und 3.2). Es ist also wichtig, dass die Verifikationsphase nicht dominiert, also insgesamt (während der vollständigen Suche über den ganzen Text) nur $O(n)$ benötigt. In diesem Fall erreicht ein Filter-Algorithmus also die geringe Laufzeit des Suchalgorithmus. Natürlich ist aber der Gesamtaufwand der Verifikationsphase stark abhängig vom Fehlerlevel $\alpha = k/m$, denn offensichtlich sind prinzipiell für höhere Fehlerlevel mehr mögliche Bereiche zu prüfen.

Es ist zu beachten, dass ein Filter-Algorithmus im schlechtesten Fall (worst case) zumindest immer das Verhalten des Verifikationsalgorithmus alleine dadurch erreicht, dass dann jeder Bereich, also der ganze Text, geprüft werden muss. Filter-Algorithmen können also nur den Durchschnittsfall (average case) verbessern.

Im Folgenden wird zuerst eine Klassifikationsmöglichkeit für Filter-Algorithmen erläutert. Anschließend werden wichtige Algorithmen der verschiedenen Klassen kurz vorgestellt.

4.1.1 Klassifikation von Filter-Algorithmen

Einige Filter-Algorithmen verwenden die in Kapitel 2.6 angesprochene Möglichkeit der Durchführung vorbereitender Operationen auf dem Text (Offline-Algorithmen).

Dennoch bietet sich gerade für Filter-Algorithmen eine andere, feinere Möglichkeit der Klassifikation an, indem nach der Art der Filters, also der *Filter-Methode*, unterschieden wird.

Die meisten Filter-Methoden können als Anwendung des folgenden Lemmas (nach den Lemmata 1 und 2 in [90]) gesehen werden, welches verschiedene Filter-Methoden auseinander hält (und dabei zugleich auf die üblicherweise betrachtete Levenshtein-Distanz einschränkt).

Lemma 4.1.1
Für ein beliebiges $j \in \mathbb{N}$ sei A ein String der Form $A = A_1X_1A_2 \ldots X_{j-1}A_j$, mit (nichtleeren) Substrings A_i $(i = 1, \ldots, j)$ und (möglicherweise leeren) Substrings X_i $(i = 1, \ldots, j-1)$. Weiterhin sei B ein String mit $d(A, B) \leq k$. Dann gilt

1. für $j < k+1$: Sei $\{k_1, \ldots, k_j\}$ eine Menge beliebiger, nicht negativer ganzer Zahlen mit $\sum_{i=1}^{j} k_i \geq k - j + 1$. Dann ist zumindest ein String A_i mit höchstens k_i Fehlern in B enthalten.

2. für $j \geq k+1$:

   a) Mindestens $j-k$ Substrings $A_{i_1} \ldots, A_{i_{j-k}}$ sind in B enthalten.

   b) Die relativen Abstände dieser $j-k$ Substrings innerhalb von B können sich nicht um mehr als k von den relativen Abständen in A unterscheiden.

Der Beweis dieses Lemmas lässt sich schnell führen:

Beweis

1. Wenn keiner der Substrings A_i mit höchstens k_i Fehlern in B enthalten ist, dann sind für jeden der j Substrings A_i also mindestens k_i+1 Korrekturen notwendig. Somit kann dann der Gesamtabstand $d(A,B)$ nicht weniger als $\sum_{i=1}^{j}(k_i+1) = \sum_{i=1}^{j} k_i + j \geq (k-j+1)+j = k+1$ betragen, was aber größer als der vorausgesetzte Abstand ist.

2. a) Da $d(A,B) \leq k$ ist, kann eine Sequenz von höchstens k Operationen A nach B konvertieren. Jede Operation kann jedoch nur maximal einen Substring A_i verändern. Damit folgt direkt, dass s Substrings unverändert bleiben.

   b) Die Aussage über die relative Distanz folgt direkt aus der Überlegung, dass k Operationen einen Substring um nicht mehr als k Stellen verschieben können. ∎

Für das Approximate String Matching hat der erste Teil des Lemmas nun folgende Bedeutung. Werden j nicht überlappende Substrings A_i aus dem String A genommen, und werden dann höchstens $k-j+1$ „mögliche“ Fehler über diese Substrings irgendwie beliebig verteilt, so existiert immer ein Substring A_l, der sich mit weniger Operationen als der ihm zugeordneten Fehlerzahl in einen Teilstring von B konvertieren lässt. Mittels dieses ersten Teils des Lemmas kann das ASM-Problem in kleinere Probleminstanzen zerlegt werden, d. h. statt nach einem Pattern P mit maximal k Fehlern zu suchen, können auch Subpattern von P mit weniger als k Fehlern (aber insbesondere mit Fehlern) gesucht werden. So kann beispielsweise auch die gleichförmige Verteilung $k_i = \lfloor k/j \rfloor$ gewählt, und jedes der j Patternstücke dann mit $\lfloor k/j \rfloor$ Fehlern gesucht werden.

Aus dem zweiten Teils des Lemmas ergibt sich eine mögliche Zerlegung des ASM-Problems in Teilprobleme, die auf exakter Suche basieren.

Um also nach der Filter-Methode zu unterscheiden, ergeben sich aus obigem Lemma direkt die Klassen

- der *Aufteilung in kleinere Probleminstanzen* (für $j < k+1$, siehe Lemma Teil 1) und

- der Aufteilung in auf exakter Suche basierende Teilprobleme (für $j \geq k+1$, siehe Lemma Teil 2), im Folgenden kurz *Aufteilung in exakte Suche* genannt.

Eine weitere Unterteilung dieser Klassen ergibt sich durch die beiden verschiedenen Interpretationsmöglichkeiten des Strings A des Lemmas. Es ist für die Algorithmen ein Unterschied, ob

- der String A ein Teilstring des Textes T ist (d. h. es wird angenommen, dass die *Fehler im Text* sind) oder, ob

- der String A direkt dem Pattern P entspricht (d. h. es wird angenommen, dass die *Fehler im Pattern* sind).

In Abbildung 4.1 ist die Aufteilung verschiedener Filter-Algorithmen in diese genannten Klassen gezeigt.

	kleinere Instanzen	exakte Suche
Fehler im Pattern	[14], [83]*, [88]*	[15, 16],[21],[58],[86]*,[111]*,[121],[137]
Fehler im Text	[22], [93]*	[59]*,[118]*,[119]*,[120]*

Abbildung 4.1: Klassifikation von Filter-Algorithmen. Die Algorithmen sind hier in Form von Literaturreferenzen auf die Originalarbeiten gegeben. Offline-Algorithmen sind dabei mit einem * gekennzeichnet.

In den folgenden vier Kapiteln 4.1.2 bis 4.1.5 werden Algorithmen dieser Klassen vorgestellt. Das darauf folgende Kapitel 4.1.6 behandelt kurz Filter-Algorithmen, die von diesem Lemma nicht abgedeckt werden.

4.1.2 Aufteilung in exakte Suche - Fehler im Pattern

Bei der Aufteilung in exakte Suche unter Berücksichtigung von Fehlern im Pattern werden mindestens $k + 1$ nicht überlappende Substrings des Patterns P ohne Fehler im Text gesucht (bzgl. dieser Aufteilung sei auf das Lemma in Kapitel 4.1.1 verwiesen). Für die exakte Suche kann im Prinzip ein beliebiges exaktes Suchverfahren verwendet werden, wie beispielsweise Knuth–Morris–Pratt [61], Boyer–Moore [18], Horspool [54] oder Sunday [115, 116].

Im Folgenden werden kurz die wichtigsten Algorithmen vorgestellt, die auf der Aufteilung in exakte Suche bei gleichzeitiger Annahme der Fehler im Pattern basieren.

4.1.2.1 Filter nach Tarhio und Ukkonen

Tarhio und Ukkonen [121] (mit Korrekturen in [58]) entwarfen den ersten auf dem Filterprinzip basierenden Algorithmus überhaupt. Der Algorithmus teilt das Pattern in m Substrings auf, was jeden Buchstaben zu einem Substring macht. Nach dem Lemma gilt, dass mindestens $m - k$ dieser Substrings in einem Approximate Matching enthalten sein müssen, oder andersherum, dass kein Approximate Matching vorhanden sein kann, wenn mehr als k dieser Substrings nicht enthalten sind.

Die Filter-Phase dieses Algorithmus vergleicht ausgehend vom Textende immer *Textfenster*, d. h. Textabschnitte einer bestimmten Breite, mit dem Pattern. Beginnt dieses Fenster also an der Position $j + 1$, wird die Stelle t_{j+i} mit p_i verglichen ($1 \geq i \geq m$). t_{j+i} wird dann „schlecht“ genannt, wenn $\min_c\{|i - c|\} > k$ mit $c \in \{1, \ldots, m | p_c = t_{j+i}\}$, wenn also das Textzeichen t_{j+i} weder mit dem Patternbuchstaben p_i übereinstimmt, noch mit einem anderen Patternbuchstaben, der nicht weiter als k Stellen entfernt ist.

Anhand des Patterns kann für alle Symbole a des Alphabets eine entsprechende Tabelle „schlechter“ Zeichen vorberechnet werden. Das Textfenster wird dann von rechts ausgehend betrachtet. Der Durchlauf wird gestoppt, sobald $k+1$ „schlechte“ Stellen gefunden wurden oder das ganze Fenster betrachtet wurde. In letztgenanntem Fall wird die Stelle j für die Überprüfung durch einen ASM Algorithmus in der zweiten Phase markiert. Nach der Bearbeitung wird das Betrachtungsfenster nach links weitergeschoben (basierend auf den Ideen der Algorithmen von Boyer–Moore [18] und Horspool [54]), bis einer der letzten $k+1$ Buchstaben von P mit einem Buchstaben in T übereinstimmt. Dieser Verschiebungswert wird während des Durchlaufs durch das Fenster berechnet.

Die Suchphase dieses Algorithmus hat eine durchschnittliche Laufzeit von $O(k^2n/\sigma + kn/(m-k))$. In [85] wurde gezeigt, dass für $\alpha < e^{-(2k+1)/\sigma}$ (für $m >> \sigma > k$) der Anteil der Verifikationsphase vernachlässigbar ist.

4.1.2.2 Filter nach Jokinen, Tarhio und Ukkonen

Jokinen, Tarhio und Ukkonen [58] adaptierten einen Algorithmus von Grossi und Lucio [47] von der Nutzung der Hamming-Distanz auf die Edit-Distanz. Auch dieser Algorithmus teilt das Pattern in m Substrings auf, und betrachtet so also die einzelnen Buchstaben. Nach dem Lemma (vgl. Kapitel 4.1.1) müssen dann mindestens $m-k$ Buchstaben in einem Approximate Matching enthalten sein.

Der Algorithmus zählt initial die Häufigkeit jedes Buchstabens im Pattern und betrachtet dann ein Fenster, das über den Text geschoben wird. Für jeden Buchstaben des Patterns wird dann kontinuierlich die Differenz der Häufigkeiten in Textfenster und Pattern nachgehalten. Ist die Summe dieser Differenzen höchstens k, so wird das aktuelle Textfenster als Kernbereich einer späteren Verifikation vorgemerkt.

Der Algorithmus wurde in [85] auf eine feste Fenstergröße beschränkt und analysiert. Demnach wird eine durchschnittliche Laufzeit von $O(n)$ für $\alpha < e^{-m/\sigma}$ erreicht.

4.1.2.3 Filter nach Wu und Manber

Der Algorithmus von Wu und Manber [137] teilt das Pattern lückenlos in $k+1$ nicht überlappende Stücke. Nach dem Lemma (vgl. Kapitel 4.1.1) muss jedes Approximate Matching mindestens eines dieser Stücke vollständig beinhalten (dieser spezielle Fall des Lemmas wurde bereits von Rivest in [102] explizit beschrieben).

Die $k+1$ Patternstücke werden mit einer Erweiterung des Shift-Or-Algorithmus [12] alle zugleich im Text gesucht. Wird eines dieser Stücke gefunden, so wird der Fundort als Kernbereich für die spätere Verifikation markiert, die für jeden Fundort einen Bereich der Größe $2k+1$ überprüfen muss.

Varianten

Um die Zeitkomplexität für die exakte Suche auf $O(n)$ zu fixieren, schlugen Baeza-Yates und Perleberg [15, 16] die Verwendung anderer Such-Algorithmen wie beispielsweise des Algorithmus von Aho und Corasick [1] vor.

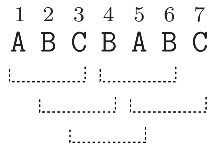
Baeza-Yates und Navarro [14] verwenden ein für die gleichzeitige Suche mehrerer Pattern angepasste Variante des Such-Algorithmus von Sunday [115, 116]. Der so erhaltene Algorithmus erwies sich, gepaart mit *Hierarchischer Verifikation* (siehe Kapitel 4.2.1), als äußerst schnell in der Praxis [85].

Insgesamt gibt es mit $O(kn \log_\sigma(m)/\sigma)$ für die durchschnittliche Laufzeit der Suche (bei Verwendung des Sunday-Algorithmus) nur eine Näherung von Navarro [85], doch es ist klar, dass die Suchphase für $\alpha < 1/(3 \log_\sigma m)$ dominiert [14, 16].

Shi [111] nutzt für die Suche einen auf einem Suffix-Tree des Textes basierenden Index und teilt das Pattern lückenlos in $k + s, s \geq 1$ nicht überlappende Stücke auf. Für diese allgemeine Variante mit s Teilpattern existiert keine Analyse.

4.1.2.4 Filter mit q-Gramm-Index auf dem Text nach Navarro und Baeza-Yates

Navarro und Baeza-Yates [86] erweiterten das Prinzip des Filters von Wu und Manber (vgl. Kapitel 4.1.2.3) um einen q-Gramm-Index des Textes für eine beschleunigte Suche. Ein *q-Gramm* (eines Strings) ist ein Substring der Länge q. Ein *q-Gramm-Index* eines Strings beinhaltet zu jedem q-Gramm des Strings die Position, an der dieses q-Gramm im String existiert. Abbildung 4.2 illustriert das Prinzip eines solchen Indexes.



ABC	1, 5
BAB	4
BCB	2
CBA	3

Abbildung 4.2: q-Gramm-Index für T = `ABCBABC` mit $q = 3$. Die verschiedenen q-Gramme sind in der linken Bildhälfte durch Unterklammerung des Textes angedeutet. In der rechten Bildhälfte ist der Index mit der Zuordnung der q-Gramme zu Positionen zu sehen.

Der Algorithmus generiert zuerst den q-Gramm Index des Textes und teilt das Pattern lückenlos in $k+1$ nicht überlappende Stücke. Jedes der Patternstücke wird dann im q-Gramm-Index gesucht, und jede Fundstelle wird zur Überprüfung in der Verifikationsphase gesichert.

Der Aufbau des Indexes ist mit einem Aufwand von $O(n)$ möglich. Die Gesamtkosten für die Verifikation betragen im Durchschnitt $O(m^2 kn/\sigma^{m/(k+1)})$, was für den Fehlerlevel $\alpha < 1/(3 \log_\sigma m)$ dann im Durchschnitt zu linearen Gesamtkosten führt.

4.1.2.5 Filter nach Chang und Lawler

Von Chang und Lawler [21] stammt ein Algorithmus, der auch auf der Übereinstimmung von Buchstaben zwischen Pattern und Text basiert (das entspricht einer Aufteilung von

P in m Teile im Lemma in Kapitel 4.1.1). In der Filter-Phase werden mindestens $m - k$ zwischen Text und Pattern übereinstimmende Buchstaben gesucht, die zugleich innerhalb eines Bereichs liegen, in dem die garantierte (durch Nicht-Übereinstimmung erkannte) Anzahl an Fehlern nicht größer als k ist.

Der Algorithmus berechnet initial einen Suffix-Tree des Patterns. In der Suchphase wird der Text durchlaufen und mittels des Suffix-Trees wird die längste Übereinstimmung (Stroke) zwischen P und T bestimmt. Ist eine Übereinstimmung maximal, so wird ab dem darauf folgenden Textzeichen der nächste Stroke bestimmt. In der abschließenden Verifikationsphase werden alle Bereiche überprüft, in denen k aufeinander folgende Strokes mindestens $m - k$ Textzeichen abdecken.

Neben diesem obigen Algorithmus, *LET* (linear expected time) genannt, stellten Chang und Lawler noch eine zweite Variante namens *SET* (sublinear expected time) vor. Sie unterscheidet sich darin, dass die Suche nach Strokes an bestimmten Blockgrenzen anfängt, wenn bereits k Strokes in einem Block gefunden wurden. Die Größe dieser Blöcke, in die der Text unterteilt wird, ist mit $(m-k)/2$ so gewählt, dass in jedem Approximate Matching mindestens ein solcher Block vollständig enthalten sein muss.

Bei LET hat die Filter-Phase eine Laufzeit von $O(n)$. Mit der Verwedung des Algorithmus von Landau und Vishkin (siehe Kapitel 3.1.3.2) zur Verifikation wird eine durchschnittliche Laufzeitkomplexität von $O(kn)$ erreicht. Für $\alpha < 1/\log_\sigma m + O(1)$ (in der Praxis ist die Konstante bedeutungsvoll; siehe [85]) ist die Filter-Phase dominant und somit eine lineare Laufzeit möglich. Die durchschnittliche Gesamtlaufzeit bei SET ist für obiges α $O(\alpha n \log_\sigma m)$. Für die noch stärkere Einschränkung von $\alpha = o(\log_\sigma m)$ wird eine durchschnittliche Laufzeit von $o(n)$ erreicht.

Der von beiden Algorithmusvarianten erzeugte Verwaltungsaufwand sorgt dafür, dass sie in der Praxis erst für längere Pattern brauchbar werden.

4.1.3 Aufteilung in exakte Suche - Fehler im Text

Bei der Aufteilung in exakte Suche unter Berücksichtigung von Fehlern im Text, werden mindestens $k+1$ nicht überlappende Substrings des Textes gesucht, die mit Substrings des Pattern übereinstimmen und gegebenenfalls zugleich zueinander einen relativen Abstand haben, der kleiner als k ist. In Bezug auf das Lemma in Kapitel 4.1.1 entspricht also B dem Pattern und die A_i entsprechen den nicht überlappenden Substrings des Textes. Lücken zwischen diesen Substrings entsprechen den Strings X_i.

Für die Suche wird üblicherweise ein Index des Textes verwendet, in dem die zum Pattern passenden Textstücke direkt gefunden werden können.

Im Folgenden werden kurz die wichtigsten Algorithmen vorgestellt, die auf der Aufteilung in exakte Suche bei gleichzeitiger Annahme der Fehler im Text basieren.

4.1.3.1 Filter nach Takaoka

Die Idee des Algorithmus von Takaoka [120] ist es, q-*Samples* (das sind nicht überlappende q-Gramme) des Textes im Pattern zu suchen und dort Umgebungen zu überprüfen, wo ein q-Sample gefunden werden konnte.

Der Algorithmus konstruiert zuerst einen Suffix-Tree des Patterns (bis zur Tiefe q). In der Suchphase werden dem Text q-Samples im Abstand von $h = \lfloor (m-k-q+1)/(k+1) \rfloor$ entnommen und im Pattern (unter Nutzung des Suffix-Trees) gesucht. In der Verifikationsphase wird eine Umgebung um alle gefundenen q-Samples überprüft.

Die durchschnittliche Laufzeit des Algorithmus ist $O(\alpha n \log_\sigma m)$. Für $\alpha < O(1/\log_\sigma m)$ dominiert die Suchphase.

4.1.3.2 Filter nach Sutinen und Tarhio

Der von Sutinen und Tarhio in [119] vorgestellte Algorithmus basiert auf Vorarbeiten in [59, 118] und dem Algorithmus von Takaoka (siehe Kapitel 4.1.3.1). Anders als Takaoka, dessen Algorithmus auf einer $k+1$ Aufteilung basiert, legen Sutinen und Tarhio eine allgemeine $k+s, s \geq 1$ Aufteilung zugrunde.

Initial konstruiert der Algorithmus einen Index aus q-Samplen, die dem Text im Abstand von $h = \lfloor (m-k-q+1)/(k+s) \rfloor$ entnommen werden. In der Suchphase werden alle $m-q+1$ q-Gramme des Patterns im Index gesucht. Diese Suche kann auch als Suche der q-Samples im Pattern interpretiert werden. Immer dann, wenn von $k+s$ aufeinander folgenden q-Samplen des Textes s im Pattern gefunden wurden, wird der Textbereich überprüft, der diese $k+s$ q-Samples umfasst.

Die durchschnittliche Laufzeit des Algorithmus ist $O(\alpha n \log_\sigma m)$ bei einem optimal gewählten $q = \Theta(\log_\sigma m)$. Für $\alpha < 1/\log_\sigma m$ dominiert die Suchphase.

4.1.4 Aufteilung in kleinere Instanzen - Fehler im Pattern

Bei der Aufteilung in kleinere Probleminstanzen unter Berücksichtigung von Fehlern im Pattern werden höchstens k nicht überlappende Substrings des Patterns P in T gesucht. Dabei wird dann jeder der Substrings mit der für ihn gültigen Fehlerzahl betrachtet. In Bezug auf das Lemma in Kapitel 4.1.1 entsprechen also die A_i den Substrings des Patterns und B dem Text (bzw. jeweils dem entsprechendem Textteilstring).

Bei der Aufteilung in kleinere Probleminstanzen wird üblicherweise jedem Substring des Patterns die gleiche gültige Fehlerzahl (k_i im Lemma) zugewiesen, um so eine Problemgröße zu erhalten, die sich mit speziellen Algorithmen gut handhaben lässt.

Im Folgenden werden kurz die wichtigsten Algorithmen vorgestellt, die auf der Aufteilung in kleinere Probleminstanzen bei gleichzeitiger Annahme der Fehler im Pattern basieren.

4.1.4.1 Filter nach Baeza-Yates und Navarro

Baeza-Yates und Navarro zeigen in [14] (neben vielen anderen Ideen) einen Algorithmus, der das Pattern in $j \leq k$ Subpattern aufteilt. Die dadurch entstehenden Teilprobleme erfordern eine Suche der Subpattern mit jeweils $\lfloor k/j \rfloor$ Fehlern. Insgesamt ist j so gewählt, dass die entstehenden Teilprobleme in ein Computer-Wort passen und dann mit einem speziellen Algorithmus schnell gelöst werden können. Baeza-Yates und Navarro verwenden dazu einen bitparallel implementierten Automaten, der die entstandenen Subprobleme löst.

Bei der Computer-Wortbreite w wird eine durchschnittliche Laufzeit von $O(n\sqrt{mk/w})$ erreicht. Die Verifikationskosten sind nicht signifikant für $\alpha \leq 1 - m^{1/\sqrt{w}}/\sqrt{\sigma}$.

4.1.4.2 Filter mit Nachbarschaft für Subpattern

Sowohl Myers [83], als auch Navarro und Baeza-Yates [88] verwenden im Prinzip die gleiche Idee der Aufteilung des Patterns in j $(j \leq k)$ Subpattern und der Nutzung der (k/j)-*Nachbarschaft* bei der Suche. Unter der e-Nachbarschaft eines Strings werden alle Strings verstanden, die unter der Edit-Distanz maximal den Abstand e zum gegebenen String haben.

Initial wird vom Algorithmus ein Index des Textes berechnet. Myers arbeitet hier mit einem q-Gramm-Index, während Navarro und Baeza-Yates auf einen Suffix-Tree als Indexstruktur zurückgreifen. Das Pattern wird in j Subpattern zerlegt, wobei die Zerlegung von Myers auf Zweierpotenzen basiert und zum q-Gramm-Index passt und bei Navarro und Baeza-Yates auf dieses j nur im Hinblick auf die Suchzeit optimiert wird. Für jedes der Subpattern werden alle Strings der (k/j)-Nachbarschaft konstruiert, die dann unter Ausnutzung des Indexes im Text gesucht werden. In der Verifikationsphase wird dann für jede Fundstelle ein entsprechender Bereich überprüft.

Die Laufzeitkomplexität dieses Algorithmus beträgt insgesamt $O(n^{\lambda})$ für $0 \leq \lambda \leq 1$ bei einem optimal gewählten $j = \Theta(m/log_{\sigma}n)$ [88]. Für $\alpha < 1 - e/\sqrt{\sigma}$ ist $\lambda < 1$ und somit wird Sublinearität erreicht.

4.1.5 Aufteilung in kleinere Instanzen - Fehler im Text

Bei der Aufteilung in kleinere Probleminstanzen unter Berücksichtigung von Fehlern im Text, werden höchstens k nicht überlappende Substrings des Textes im Pattern gesucht. Dabei wird dann jeder der Substrings mit der für ihn gültigen Fehlerzahl betrachtet. In Bezug auf das Lemma in Kapitel 4.1.1 entsprechen also die A_i den Substrings des Textes und B dem Pattern.

Im Folgenden werden kurz die wichtigsten Algorithmen vorgestellt, die auf der Aufteilung in kleinere Probleminstanzen bei gleichzeitiger Annahme der Fehler im Pattern basieren.

4.1.5.1 Filter nach Chang und Marr

Der Algorithmus von Chang und Marr [22] basiert auf dem Algorithmus SET von Chang und Lawler (siehe Kapitel 4.1.2.5). Im Gegensatz zu den meisten Algorithmen mit dem Prinzip der Zerlegung in kleinere Instanzen wird bei diesem Algorithmus keine konkrete Zuordnung einer maximalen Fehlerzahl für jedes Teilproblem vorgenommen, sondern stattdessen nur die Summe der Fehler in den Teilproblemen betrachtet (vgl. Lemma in Kapitel 4.1.1).

Für den Algorithmusablauf wird der Text lückenlos in benachbarte Substrings der Länge $l = t \log_\sigma m$ (für eine von σ abhängige Konstante t) aufgeteilt. In einem Vorberechnungsschritt wird für jeden String der Länge l das beste Approximate Matching im Pattern bestimmt (wobei keine konkrete maximale Fehlerzahl zugeordnet wird (s. o.), aber k weiterhin ein zu betrachtendes Maximum darstellt). In der Suchphase wird der Text blockweise (bei Blöcken der Größe $(m-k)/2$) bearbeitet. Dazu werden ab jedem Blockanfang die aufeinander folgenden Substrings der Länge l im Pattern gesucht, bis die Gesamtfehlerzahl entweder k überschreitet, oder bis $m-k$ Buchstaben des Textes durch die Substrings abgedeckt wurden. Im ersten Fall bedarf der Block keiner weiteren Beachtung, im zweiten Fall ist der Block für die Verifikationsphase vorzumerken.

Chang und Marr konnten zeigen, dass dieser Filter-Algorithmus im Durchschnitt optimal ist (in der Praxis aber allenfalls für sehr lange Pattern nützlich [85]) und eine Laufzeit von $O(\frac{k+\log_\sigma m}{m} n)$ hat für $\alpha < \rho_\sigma$. Dabei gilt $\rho_\sigma \to 1 - e/\sqrt{\sigma}$ für $\sigma \to \infty$.

4.1.5.2 Filter nach Navarro, Sutinen, Tarhio und Tanninen

Ausgehend von Ideen in [119] zeigten Navarro et al. [93] einen Algorithmus, der j aufeinander folgende q-Samples des Textes im Pattern sucht und zudem deren Reihenfolge dort beachtet.

Es wird ein q-Sample-Index des Textes aufgebaut, wobei die q-Samples dem Text jeweils im Abstand $h = \lfloor (m-k-q+1)/j \rfloor$ entnommen werden. Das Pattern wird in überlappende Blöcke $Q_i = p_{hi-k} \cdots p_{hi+q-1+k}$ unterteilt. Für jeden Patternblock wird eine e-Nachbarschaft erzeugt (für $q > e \geq \lfloor k/j \rfloor$). In der Suchphase wird fehlerbehaftet (durch die Nachbarschaft) jeder Patternblock Q_i im q-Sample-Index des Textes gesucht. Die Suche wird dabei so vorgenommen, dass das Resultat wiedergibt, welche der q-Samples in den Patternblöcken enthalten sind. Anschließend wird immer dort eine Überprüfung durchgeführt, wo j aufeinander folgende q-Samples mit insgesamt weniger als k Fehlern in aufeinander folgenden Patternblöcken gefunden wurden. In Abbildung 4.3 ist die Gesamtsituation exemplarisch verdeutlicht.

In [93] wird die Wahl der Parameter diskutiert, jedoch keine Laufzeitanalyse gegeben.

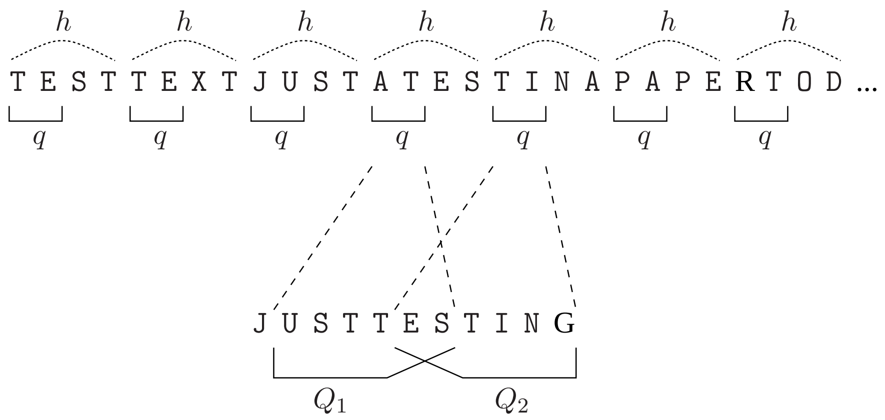


Abbildung 4.3: q-Samples des Textes und Blöcke des Patterns. In dem gezeigten Fall für $k = 2, q = 2, m = 11, j = 2$ werden die q-Samples mit einer Schrittweite von $h = 4$ aus dem Text genommen. Q_1 und Q_2 repräsentieren Blöcke des Patterns. Für einen Kandidaten eines Approximate Matching müssen j aufeinander folgende q-Samples mit insgesamt nicht mehr als k Fehlern in aufeinander folgenden Patternblöcken vorhanden sein.

4.1.6 Algorithmen außerhalb des Lemmas - überlappende Teilstrings

Neben den Algorithmen der in den Kapiteln 4.1.2 bis 4.1.5 beschriebenen Klassen gibt es zudem auch noch einige Filter-Algorithmen, die sich nicht nach dem Lemma (aus Kapitel 4.1.1) zuordnen lassen. Diese Algorithmen unterscheiden sich zu dem bisher genannten Filter-Algorithmen darin, dass sie in irgendeiner Form auf überlappenden Teilstrings basieren. Die Betrachtung überlappender Teilstrings wird im Lemma aus Kapitel 4.1.1 nicht in Erwägung gezogen und fand deswegen bei der Klassifikation nach dem Lemma auch keine Beachtung.

Im Folgenden werden kurz wichtige, in der Klassifikation außerhalb des Lemmas stehende Filter-Algorithmen vorgestellt.

4.1.6.1 Filter nach Ukkonen

Von Ukkonen [124] stammt ein Algorithmus, der im Prinzip eine Aufteilung in exakte Suche bei der Verwendung von überlappenden Teilstrings (q-Grammen) des Patterns darstellt. Generell besteht das Pattern aus $(m - q + 1)$ q-Grammen. Jeder Fehler kann nur maximal q q-Gramme beeinflussen, also müssen in jedem Approximate Matching mindestens $(m - q + 1 - kq)$ q-Gramme enthalten sein.

Der Algorithmus beginnt mit der Konstruktion eines Suffix-Trees des Patterns. Da dieser nur alle möglichen q-Gramme des Patterns repräsentieren muss, wird er nur bis zu der Tiefe q berechnet. Beim Textdurchlauf wird dann im Suffix-Tree an den relevanten Knoten genau die Anzahl der auftauchenden q-Gramme des Patterns nachgehalten. Sind zu einer Zeit in einem Fenster der Größe m mindestens $(m - q + 1 - kq)$ q-Gramme gefunden worden, so wird die Stelle zur Verifikation vermerkt.

Es gibt keine genaue Berechnung der Laufzeit, jedoch benötigt die Filter-Phase $O(n)$ und in [85] wurde gezeigt, dass die Verifikationskosten für $\alpha < o(1/log_\sigma m)$ vernachlässigbar sind. Die optimale Größe der q-Gramme ist $q = \log_\sigma m$. Für $q = 1$ entspricht dieser Algorithmus dem Gedanken der auch von Jokinen et al. formuliert wurde (vgl. Kapitel 4.1.2.2).

4.1.6.2 Filter nach Navarro und Raffinot

Von Navarro und Raffinot [92] stammt ein Filter-Algorithmus, der auf dem exakten Suchalgorithmus BDM von Crochemore et al. [28] basiert. Die Grundidee hinter dem Algorithmus von Navarro und Raffinot ist die Überlegung, dass in jedem Approximate Matching mit nicht mehr als k Fehlern immer ein Teilstring des Patterns der Größe $m-k$ mit höchstens k Fehlern enthalten sein muss. Also werden mit Hilfe eines Automaten alle diese Teilstrings zugleich unter Zulassung von Fehlern gesucht. Dann wird immer dort genauer auf ein Approximate Matching geprüft, wo ein Teilstring mit höchstens k Fehlern gefunden wurde.

Etwas ausführlicher beschrieben hat der Algorithmus also folgenden Ablauf. Initial wird ein nichtdeterministischer endlicher Automat (non-deterministic finite automaton, NFA) konstruiert, der das Pattern (genauer gesagt, das umgedrehte Pattern) bis zu einem Vorhandensein von maximal k Fehlern identifizieren kann (vgl. Abbildung 4.4). Der Automat wird nun angewendet auf ein Textfenster der Größe $m - k$ (die Größe der gesuchten Teilstrings des Patterns), welches vorne beginnend am Text entlang geschoben wird. An jeder neuen Fensterposition werden alle Zustände des Automaten auf aktiv geschaltet und dann wird der Text betrachtet. Bei der gelesenen Eingabe nicht erreichbare Zustände werden inaktiv gesetzt (siehe Abbildung 4.4). Sobald der Automat keine aktiven Zustände mehr hat, ist auch kein Approximate Matching im dem betrachteten Fenster mehr möglich, und das Fenster wird verschoben, bis zu der Position, an der zuletzt noch ein aktiver Zustand erkannt wurde. Wird das Ende eines Textfensters mit noch aktiven Zuständen erreicht, so ist dieses Fenster (erweitert um $2k$ Stellen nach rechts) nun noch auf ein Approximate Matching zu überprüfen.

Die durchschnittliche Laufzeit des Algorithmus beträgt $O(n(\alpha + \alpha^* \log_\sigma(m)/m)/((1-\alpha)\alpha^* - \alpha))$ wobei für α^* gilt: $1 - e/\sqrt{\sigma} < \alpha^* \leq 1 - 1/\sigma$ (siehe Kapitel 2.7). Die Kosten für die Verifikation sind für $\alpha < (1 - e/\sqrt{\sigma})/(2 - e/\sqrt{\sigma})$ vernachlässigbar, was für große σ also etwa $\alpha < 1/2$ bedeutet.

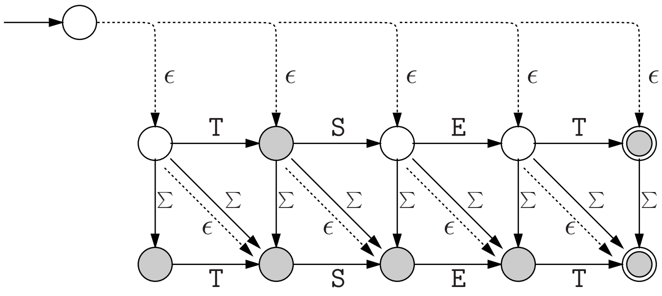


Abbildung 4.4: NFA für das umgekehrte Pattern `TEST`. Die Zustände, die nach dem Einlesen von `T` noch aktiv sind, sind grau unterlegt.

4.2 Verbesserung von Filter-Algorithmen

Das Hauptproblem beim Prinzip des Filterns (siehe Kapitel 4.1) liegt in den teuren Kosten der Verifikationsphase im Vergleich zu denen der Suchphase. Konkret ist die Suchphase oft in linearer Zeit durchführbar, wohingegen für die Verifikationsphase die Kosten prinzipiell quadratisch in der Länge des Patterns sind (siehe Kapitel 3.1).

Wie schon in Kapitel 4.1 erläutert, ist es aus diesem Grund für die Robustheit eines Filter-Algorithmus bezüglich der durchschnittlichen Laufzeit wichtig, den Anteil der Verifikationsphase an den Gesamtkosten so niedrig wie möglich zu halten.

In diesem Kapitel werden bestehende Ansätze kurz vorgestellt, die sich damit beschäftigen, die Verifikationsphase so anzupassen, dass sich ihr Anteil an den Gesamtkosten verringert. Zu beachten ist, dass die in den Kapiteln 4.2.2 und 4.2.3 vorgestellten Ansätze die Trennung der beiden Phasen der Filter-Algorithmen aufheben.

4.2.1 Hierarchische Verifikation

Von Myers und später verfeinernd von Navarro und Baeza-Yates stammt das Prinzip der *Hierarchischen Verifikation* [83, 84, 87, 89]. Grundlage für dieses Prinzip bildet das Lemma aus Kapitel 4.1.1.

Ausgangspunkt für die Hierarchische Verifikation bildet die fortgesetzt halbierende Zerlegung des Patterns in Teilstrings, bis diese Subpattern klein genug sind, um sie mit $\lfloor k/j \rfloor$ Fehlern im Text zu finden (dabei ist $j = 2^i$ mit der Anzahl i an erfolgten Halbierungen). Für die Hierarchische Verifikation ist es also unerheblich, ob eine Zerlegung des Problems in kleinere Teilinstanzen oder in exakte Suche erfolgt ist, einzig notwendig ist die Initiierung einer Verifikation durch ein Subpattern, das mit höchstens $\lfloor k/j \rfloor$ Fehlern im Text gefunden wird.

Die Verifikation revidiert dann schrittweise die Aufteilung des Patterns. Das gefundene Subpattern wird mit dem benachbarten Subpattern zusammen überprüft, ob für diese ein

Approximate Matching mit maximal $2\lfloor k/j \rfloor$ Fehlern existiert. Existiert ein solches Approximate Matching nicht, kann die Verifikation abgebrochen werden. Andernfalls wird die Zusammenfassung von Subpattern auf einer höheren Ebene wiederholt. Allgemein werden gemäß der ursprünglichen Patternaufteilung also solange immer zwei benachbarte Subpattern zusammengefasst und mit verdoppelter Fehlerzahl überprüft, bis entweder eine dieser Überprüfungen zu dem Ergebnis führt, dass kein Approximate Matching möglich ist, oder aber das gesamte Pattern überprüft wurde. In Abbildung 4.5 ist dieser Vorgang verdeutlicht.

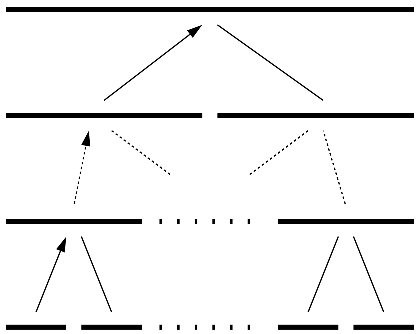


Abbildung 4.5: Hierarchische Verifikation. Unter der Annahme, dass der am weitesten links liegende Teilstring des Patterns zuerst überprüft wird, deuten die Pfeile die Abfolge der Verifikationsschritte an.

Es wurde gezeigt [84, 87], dass bei der Verwendung von exakter Suche und der Zerlegung in $k+1$ Subpattern (vgl. Kapitel 4.1.2.3) die Verifikationsphase in der Laufzeit vernachlässigbar ist für $\alpha < 1/\log_\sigma m$.

4.2.2 Phasenmischung

In [14] wird explizit ein einfaches Prinzip erläutert, das zur verbesserten Verifikation die Trennung der beiden Phasen eines Filter-Algorithmus aufhebt.

Das Prinzip sieht eine simple Verzahnung der Such- und der Verifikationsphase vor, um die mehrfache Überprüfung sich überlappender Bereiche zu vermeiden. Wird bei der Suche ein mögliches Approximate Matching gefunden, so wird dieser Bereich erst verifiziert, bevor genau anschließend an dem verifizierten Bereich die Suche fortgesetzt wird.

Eine Analyse der Auswirkungen dieses Ansatzes wird nicht gegeben.

4.2.3 „Dynamic Filtering“

Giegerich et al. [44, 45] zeigten ein allgemeines Prinzip, welches die beiden Phasen (Suche und Verifikation) noch stärker vermischt, als dies bei der Phasenmischung in Kapitel 4.2.2

der Fall ist. Das Prinzip wurde von den Autoren *Dynamic Filtering* genannt, weil es nicht mehr nur auf der „statischen“ Information der Suchphase, dass ein Bereich überprüft werden muss, basiert, sondern „dynamisch“ auf weitere Informationen zurückgreift.

Abhängig von dem konkret verwendetem Filter ist dann, wenn die Notwendigkeit der vollständigen Überprüfung eines Bereichs erkannt wurde, auch häufig noch mehr über die zur Identifizierung dieses Bereichs bereits benötigte Fehlerzahl bekannt (wie es sich beispielsweise durch die Aussage 2.b) im Lemma in Kapitel 4.1.1 ergibt). Diese Information wird während der Verifikation mit dem aktuellen Verifikationszustand abgeglichen und sobald ein mögliches Approximate Matching bereits mehr Fehler enthält, als es nach der Suchphase in diesem Moment noch möglich wäre, wird die Verifikation des Bereichs vorzeitig abgebrochen.

Dieses Prinzip wurde von Giegerich et al. anhand der Filter-Algorithmen von Sutinen und Tarhio (Kapitel 4.1.3.2) und von Chang und Lampe (Kapitel 3.1.5) demonstriert. Zudem zeigten sie (ohne genaue Analyse), dass bei Anwendung dieses Prinzips praktisch der Fehlerlevel α erhöht wird, zu dem die Suchphase noch dominant ist.

5 Patchwork-Verifikation

Um klar zu sehen, genügt oft ein Wechsel der Blickrichtung.

(Antoine de Saint-Exupery, frz. Schriftsteller, 1900-1944)

Bei einem Filter-Algorithmus (vgl. Kapitel 4.1) wird immer dann, wenn ein mögliches Approximate Matching identifiziert werden konnte, der Verifikationsalgorithmus aufgerufen. Bei der Aufteilung in exakte Suche (vgl. Kapitel 4.1.2 und 4.1.3) wird so ein Bereich des Textes um ein oder mehrere exakt gefundene Subpattern herum auf ein Approximate Matching hin überprüft. Liegen diese vom Filter identifizierten Bereiche nahe beieinander, so kann es zu Überschneidungen dieser Bereiche kommen. Das im Folgenden vorgestellte Verfahren der *Patchwork-Verifikation* (siehe auch [101]) stellt einen Ansatz dar, einen zur Verifikation verwendeten Algorithmus so zu erweitern, dass dieser geschickter mit solchen Überschneidungen umzugehen vermag. Zudem wird bei diesem Verfahren keinerlei Abhängigkeit zwischen Verifikationsphase und Suchphase geschaffen und die Suche verläuft weiterhin vollständig unabhängig.

Allgemeine Ansätze, die ebenso versuchen die Verifikationsphase von Filteralgorithmen zu verbessern, wurden in Kapitel 4.2 vorgestellt. Im selben Maße unabhängig von der Suchphase wie die Patchwork-Verifikation ist dort jedoch nur die Hierarchische Verifikation.

Das hier gezeigte Verfahren und seine Analyse basieren konkret auf dem Filteransatz der Zerlegung des Patterns in $k+1$ Teilstücke, die jeweils exakt gesucht werden (vgl. Kapitel 4.1.2.3). Die genaue Funktionsweise der Patchwork-Verifikation wird in Kapitel 5.1 erläutert. Dem folgt in Kapitel 5.2 eine Betrachtung der technischen Details bezüglich der Implementierung und der durchgeführten Experimente. In Kapitel 5.3 wird die Patchwork-Verifikation analysiert. Dabei werden die Grenzen der Anwendbarkeit erschlossen und die Nützlichkeit des Verfahrens bei längeren Pattern und kleineren Alphabeten bei gleichzeitig hohen Fehlerleveln (die ja automatisch zu einer größeren Anzahl an Überlappungen von Verifikationsbereichen führen) aufgezeigt. Schließlich fasst Kapitel 5.4 die wesentlichen Ergebnisse kurz zusammen.

5.1 Das Prinzip der Patchwork-Verifikation

Die grundsätzliche Idee hinter der Patchwork-Verifikation besteht darin, sich den Endzustand der letzten Verifikation zu merken und im Falle zu eng beieinander liegender Bereiche der Verifikation diesen Endzustand zu nutzen, um nahtlos mit der Verifikation fortfahren zu können. Zudem werden auch Positionen von positiv beantworteten Verifikationen kopiert, sofern dies möglich ist. Dadurch wird erreicht, dass – gesichert innerhalb des betrachteten Bereichs – auch alle Positionen, an denen die Verifikation eine positive Antwort liefert, als Ausgabe geliefert werden.

Wie dazu genau vorzugehen ist, ergibt sich aus der nun folgenden Analyse der Situation, wenn ein Subpattern exakt gefunden wurde und also eine Verifikation initiiert wird.

Das Subpattern P_i des Patterns P sei nun exakt an der Position t im Text T gefunden worden. Ohne weiteres zusätzliches Wissen kann der zu verifizierende Textbereich $[pos_b, pos_e]$ nur auf eine Art bestimmt werden. Für die Anfangsposition pos_b dieses Bereichs ist zu bedenken, dass P_i im Extremfall der letzte Buchstabe von P sein kann und somit alle anderen $m-1$ Buchstaben des Patterns und die k möglichen Fehler noch davor liegen können, also

$$pos_b = t - k - (m-1)\,. \tag{5.1.1}$$

Andererseits könnte P_i auch ganz am Anfang von P liegen und ein Präfix bilden. In diesem Fall würden also die anderen $m-1$ Buchstaben des Patterns und die k Fehler möglicherweise noch folgen. Das Ende pos_e des zu betrachtenden Verifikationsbereichs wird dadurch also bestimmt zu

$$pos_e = t + k + m - 1\,. \tag{5.1.2}$$

Wird als zusätzliches Wissen noch die Länge $|P_i|$ des exakt gefundenen Subpatterns betrachtet, so kann der Anfang des Verifikationsbereichs leicht verschoben werden zu

$$pos_b = t - k - (m - |P_i|)\,, \tag{5.1.3}$$

da zumindest die Buchstaben von P_i noch ab der Position t folgen.

Wird nun bei jedem Aufruf des Verifikationsalgorithmus immer das vollständige Intervall $[pos_b, pos_e]$ verifiziert, so kann dies zu den bereits angesprochenen Bereichsüberlappungen und somit Mehrfachbetrachtungen führen. Um dies zu beheben, wird am Ende der Verifikation nun der Status des Verifikationsalgorithmus gesichert und zudem auch die Ergebnisse dieser Verifikation. Wie diese Informationen bei dem nächsten Aufruf des Verifikationsalgorithmus verwendet werden können, zeigt die folgende Betrachtung einer Überlappung zweier Verifikationsbereiche.

Sei nun $[oldpos_b, oldpos_e]$ der zuletzt vor der Betrachtung von $[pos_b, pos_e]$ verifizierte Textbereich. Bei der Berechnung eines Approximate Matching mit einem Pattern der Länge m hat ein Berechnungszustand noch maximal Einfluss auf die folgenden $m-1$

Positionen (betrachte dazu das Berechnungsschema der DP-Matrix in Kapitel 3.1). Aufgrund dieses Einflussfaktors von vorherigen Positionen können nur dann Ergebnisse der vorangehenden Verifikation kopiert werden, wenn eine betrachtete Position in identischer Weise von Positionen aus beiden Verifikationsbereichen abhängt. Dies ist der Fall, wenn $pos_b \leq oldpos_e - (m-1)$ gilt, denn dann ist ab der Position $pos_b + (m-1)$ der Einfluss vorangehender Positionen im alten und neuen Verifikationsbereich gleich. Es sei zudem angemerkt, dass mit demselben Argument diese Bedingung zugleich eine „nahtlose" Fortsetzung der Verifikation im neuen Verifikationsbereich mit Hilfe des Endzustandes der letzten Verifikation erlaubt. Es ergibt sich somit für die Möglichkeit, Verifikationsergebnisse im Falle der Überlappung zu kopieren, die Bedingung

$$(oldpos_b \leq pos_b) \wedge (pos_b \leq oldpos_e - m + 1) \wedge (oldpos_e < pos_e)\,. \tag{5.1.4}$$

Eine nähere Betrachtung dieser Situation zeigt vier verschiedene Bereiche (siehe auch Abbildung 5.1):

1. $[oldpos_b, pos_b - 1]$:
   Dieser Bereich liegt außerhalb des neuen Verifikationsbereichs und ist somit nicht von weiterem Interesse.

2. $[pos_b, pos_b + m - 2]$:
   Wurde in diesem Bereich während der letzten Verifikation kein Approximate Matching gefunden, so bedeutet dies, dass dort keinesfalls ein Approximate Matching endet und nichts weiter getan werden muss. Wurde dort zuvor jedoch die Position eines Approximate Matching lokalisiert, so kann nicht mit Sicherheit gesagt werden, ob dieses durch den Einfluss vorheriger Positionen entstanden ist, oder ob dieses Approximate Matching auch in dem aktuell betrachteten Bereich bei einer Verifikation zu Tage treten würde. Aus diesem Grund, muss in diesem Fall dieser Bereich noch einmal einer getrennten Verifikation unterzogen werden (deren Endzustand nicht gespeichert wird).

3. $[pos_b + m - 1, oldpos_e]$:
   Wurden in diesem Bereich während der letzten Verifikation Approximate Matchings identifiziert, so können diese positiven Ergebnisse jetzt für die neue Verifikation übernommen werden.

4. $[oldpos_e + 1, pos_e]$:
   Über diesen Bereich ist nichts bekannt, also kann die Verifikation hier unter Verwendung des Endzustands der letzten Verifikation fortgesetzt werden.

Das Verfahren der Patchwork-Verifikation betrachtet nun immer den letzten Verifikationsbereich. Überlappt sich dieser nicht mit dem aktuellen Verifikationsbereich, so wird wie üblich eine Überprüfung durchgeführt. Wird jedoch eine Überlappung festgestellt, so

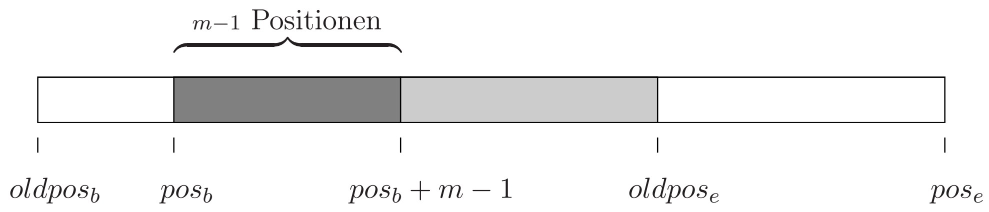


Abbildung 5.1: Patchwork-Verifikation. Die Einflussbereiche der vorangehenden Verifikation sind grau eingefärbt. In dem hellgrauen Bereich können die positiven Verifikationsergebnisse für den neuen Verifikationsbereich kopiert werden. Wurden im dunkelgrauen Bereich zuvor Approximate Matchings identifiziert, so muss dieser Bereich getrennt auch noch überprüft werden.

werden die oben angegebenen vier Bereiche unterschieden und entsprechend agiert. Es ist anzumerken, dass Bedingung 5.1.4 nur notwendig dafür ist, dass Verifikationsergebnisse der letzten Verifikation kopiert werden können. Ein leerer dritter Bereich (d. h. eine einfache Überlappung) hingegen stört nicht, denn im zweiten Bereich wird schon Verifikationsarbeit gespart, sofern dort keine Verifikation durchgeführt werden muss. Genaueres dazu zeigt auch die Analyse in Kapitel 5.3.

5.2 Praktische Aspekte

An dieser Stelle sollen nun praktische Aspekte beleuchtet werden, die bei der Implementierung oder Ausführung der Patchwork-Verifikation oder eines Vergleichsalgorithmus von Bedeutung sind. Der erste Unterabschnitt befasst sich konkret mit Aspekten implementierungstechnischer Natur, während sich der zweite Unterabschnitt auf die durchgeführten Experimente bezieht.

5.2.1 Implementierungstechnische Details

Zum Test und zum Vergleich der Patchwork-Verifikation ist das Gesamtszenario des Approximate String Matching Problems implementiert. Als genereller Lösungsansatz wurde der bei dem Entwurf der Patchwork-Verifikation zugrunde gelegte Ansatz der Aufteilung in exakte Suche bei Zerlegung des Patterns in $k+1$ Teilpattern verwendet (siehe Kapitel 4.1.2.3). Das Pattern wird vollständig gleichmäßig aufgeteilt, so dass von den entstehenden Subpattern s von der Größe $\lceil m/(k+1) \rceil$ und $k+1-s$ von der Größe $\lfloor m/(k+1) \rfloor$ sind, wobei $s = m \bmod (k+1)$ gilt (wobei mod die Modulo-Operation darstellt). Für die exakte Suche dieser Subpattern wird eine wie in [14] und [87] beschriebene, zur gleichzeitigen Suche mehrerer Pattern angepasste Variante des Algorithmus von Sunday [115, 116] verwendet.

Als einfacher und grundsätzlicher Verifikationsalgorithmus findet der Algorithmus von Chang und Lampe mit Cut-Off-Heuristik Verwendung (siehe Kapitel 3.1.5).

Selbstverständlich ist das Verfahren der Patchwork-Verifikation auf Basis der in Kapitel 5.1 aufgezeigten Unterscheidung von Teilbereichen des gesamten Verifikationsbereichs implementiert. Um dabei nahtlos an die letzte Verifikation anknüpfen zu können, ist es notwendig, am Ende der Verifikation eines Bereichs den Zustand der Verifikation zu sichern. Der Zustand einer Verifikation definiert sich über die wichtigen Bearbeitungsvariablen:

- Bei der spaltenweise erfolgenden Berechnung der DP-Matrix reicht es aus, immer nur die zuletzt berechnete und die aktuelle Spalte verfügbar zu halten. Auch bei der Betrachtung der Läufe (die sich ja auch nur innerhalb einer Spalte bewegen), ist nur die Verfügbarkeit zweier Spalten notwendig. Die Verwendung immer der gleichen zweispaltigen Matrix ermöglicht es, auf einfache Weise immer den letzen Matrixzustand verfügbar zu halten. Greift das Kriterium (Bedingung 5.1.4) für die Patchwork-Verifikation einmal nicht, so ist an dieser Stelle unbedingt eine Neuinitialisierung der Initialzellen dieser Matrix nötig.

- Um überhaupt Überlappungen der Verifikationsbereiche feststellen zu können, müssen die Start- und Endposition des Verifikationsbereichs jeweils gesichert werden.

- Der Cut-off-Wert der letzten Spalte muss zur Sicherstellung einer funktionsfähigen Cut-Off-Heuristik ebenso gesichert werden.

- Die Sicherung der Verifikationsergebnisse eines Bereichs ist erforderlich im Hinblick auf eine vollständige Ausgabe bei jeder einzelnen Verifikation. Dazu wird der Anfang der Liste mit den Stellen gefundener Approximate Matchings und zudem die Länge dieser Liste (also die Anzahl der gefundenen Stellen) gespeichert. Über die Listenlänge ist es dann weiterhin möglich, auf die Ergebnisse zuzugreifen, auch wenn die Liste selbst in späteren Schritten noch erweitert wird. Wird diese Ergebnislisteliste (wie es hier geschieht) in eine sortierte Gesamtergebnisliste integriert, so muss zudem dafür Sorge getragen werden, dass bei dem Vorgang der Verschmelzung der Ergebnislisten auf jeden Fall das gesicherte Listenanfangselement auch weiterhin gültig bleibt.

Mit Hilfe dieses Zustands und der Unterscheidung der genannten Teilbereiche des Verifikationsbereichs ist das Verfahren der Patchwork-Verifikation also implementiert.

Neben der einfachen Verifikation nach Chang und Lampe dient als weiterer Vergleichsalgorithmus das Verfahren der Hierarchischen Verifikation (siehe Kapitel 4.2.1). Entscheidend für die Auswahl der Hierarchischen Verifikation ist die direkte Vergleichbarkeit zur Patchwork-Verifikation, die sich aus der Unabhängigkeit des Verfahrens ergibt: sowohl die Hierarchische Verifikation als auch die Patchwork-Verifikation überprüfen genau den gegebenen Bereich, liefern dafür die vollständige Ergebnismenge und erwarten keine weiteren Eingaben. Insbesondere sind diese Verfahren völlig unabhängig von der Suchphase und prinzipiell beliebig gegeneinander austauschbar.

Die hier vorgenommene Implementierung der Hierarchischen Verifikation weicht minimal von der originalen Version ab, die das Pattern immer solange halbierend unterteilt, bis eine für die Suchphase brauchbare Größe entsteht. Hier ist jedoch durch die $k+1$ Aufteilung des Patterns die Größe der kleinsten Subpattern bereits gegeben. Die für den Prozess notwendige hierarchische Zerlegung des Patterns ergibt sich nun durch eine Umkehrung der Halbierung: es werden sukzessive immer Subpattern miteinander verschmolzen bis das vollständige Pattern erreicht wird. Zur Bestimmung der Fehlerzahl, mit der die verschmolzenen Subpattern dann betrachtet werden müssen, ist der erste Teil des Lemmas aus Kapitel 4.1.1 hilfreich. Im Falle der Aufteilung des Patterns in weniger als $k+1$ Stücke (wie es sich bei der Verschmelzung ergibt), erlaubt es das Lemma insbesondere, auch die Fehler gleichmäßig auf die Buchstaben des Patterns aufzuteilen, und so ein Subpattern S mit $\lfloor |S|k/m \rfloor$ Fehlern zu überprüfen.

Die Anzahl der Stufen bei der Hierarchischen Verifikation ist $\log_2(k+1)$ und auf jeder dieser Stufen werden immer so viele der $k+1$ grundsätzlichen Subpattern miteinander verschmolzen, dass die neu entstehenden Teilpattern S alle aus möglichst der gleichen Anzahl der $k+1$ Grundpattern gebildet werden.

Ein in der Praxis zu beachtendes Problem besteht in der möglichen Mehrdeutigkeit der anfänglichen $k+1$ Subpattern. Wenn eines dieser Subpattern mehr als einmal im Pattern vorkommt, so kann die Hierarchische Verifikation erst dann gestoppt werden, wenn jedes der möglichen Vorkommen dieses Subpattern geprüft wurde. Wurde also zum Beispiel das Pattern `aaxxaaaa` in 4 Subpattern ($k = 3$) aufgeteilt und das Subpattern `aa` wurde in der Suchphase ohne Fehler gefunden, so reicht es auf der hierarchisch höheren Ebene nicht aus, alleine `aaxx` mit $\lfloor 4 * 3/8 \rfloor = 1$ Fehlern zu überprüfen, sondern vielmehr muss auch noch `aaaa` überprüft werden.

5.2.2 Erläuterungen zu den Experimenten

Wie auch alle anderen Programme dieser Arbeit sind die zur Bewertung der Patchwork-Verifikation notwendigen Routinen in Java implementiert. Sowohl die Patchwork-Verifikation, als auch die Hierarchische Verifikation verwenden zur einfachen Verfikation dieselbe Routine, die den Algorithmus von Chang und Lampe mit Verwendung der Cut-Off-Heuritik durchführt. Bei den verwendeten Algorithmen findet zudem dasselbe Rahmenprogramm zur Realisierung des konkreten Filters zum Approximate String Matching Verwendung. Insgesamt wurde versucht, die höchstmögliche Vergleichbarkeit zu erhalten.

Alle in diesem Kapitel durchgeführten Berechnungen wurden auf einem 1.5 GHz Pentium M Windows PC ausgeführt. Bei den durchgeführten Vergleichen kommt es nicht auf eine absolute Systemlaufzeit an (sonst wäre zudem bei dieser Anwendung eine Implementierung in C unumgänglich). Vielmehr ist eine qualitative Betrachtung der Ergebnisse insbesondere im Hinblick auf die in Kapitel 5.3 durchgeführte Analyse von Bedeutung. Wichtig sind also die direkten Vergleiche und die Veränderung des Laufzeitverhaltens in Abhängigkeit von der Anzahl der Fehler.

Die Experimente sind, sofern nicht wie in Einzelfällen sehr hohe Rechenzeiten absehbar waren, in mindestens 20-facher Wiederholung ausgeführt worden. In den Abbildungen ist dann entsprechend ein Balken eingetragen, der die Standardabweichung aufzeigt.

Der Text, in dem gesucht wird, ist immer von einer der drei folgenden Arten:

- Zufälliger Text. Aus einem Alphabet einer beliebigen Größe σ werden über eine zufällige Gleichverteilung Texte einer beliebigen Länge n generiert. Zur Erzeugung der Zufallswerte findet der Algorithmus des *Mersenne Twisters* [76] Verwendung.
- Englischer Text. Als Text der englischen Sprache findet hier die King-James-Bibel Verwendung. Alle Buchstaben sind in Großbuchstaben konvertiert, und alle Trennzeichen bis auf die Zeilenumbrüche wurden auf das Leerzeichen abgebildet. Das sich so ergebende Alphabet umfasst 28 Symbole. Die Textlänge wird entsprechend der Eingabe auf n Zeichen beschränkt.
- DNS. Das vollständige Genom des Bakteriums Haemophilus Influenzae Rd findet als biologische DNS Sequenz hier Verwendung. Der Datensatz ist in Zeilen von jeweils 70 Zeichen eingeteilt und besteht abgesehen von wenigen Ausnahmen (wie beispielsweise ein N für eine unbekannte Base oder ein P für Pyrin, was entweder C oder T entspricht) aus Symbolen des vier-buchstabigen DNS-Alphabets (A,C,G,T). Der Anteil der Ausnahmen liegt bei unter 0.01% im gesamten DNS-Strang, jedoch hat hier das zugrunde liegende (und verwendete) Alphabet dadurch insgesamt eine Größe von 12. Der vollständige Datensatz hat eine Länge von etwa 1.77 MB.

Die im Text gesuchten Pattern wurden durchweg zufällig dem Text entnommen.

5.3 Analyse

An dieser Stelle soll der Ansatz der Patchwork-Verifikation bewertet werden. Zuerst werden Abschätzungen für die Bedingungen hergeleitet, unter denen bei der Patchwork-Verifikation gegenüber der einfachen Verifikation (d. h. vollständigen Überprüfung des Verifikationsbereichs, wann immer dies gefordert ist) ein verbessertes Verhalten zu erwarten ist. Zudem wird auch insgesamt die Leistungsfähigkeit der Patchwork-Verifikation analysiert. Die Verlässlichkeit der sich ergebenden Resultate wird anhand von Experimenten aufgezeigt. Weiterhin wird Patchwork-Verifikation in direkten Vergleich zur Hierarchischen Verifikation (vgl. Kapitel 4.2.1) gesetzt, da das Verfahren der Hierarchischen Verifikation von den äußeren Bedingungen her als gleichwertig betrachtet werden kann. Unter einer Gleichwertigkeit bezüglich der äußeren Bedingungen ist in diesem Fall gemeint, dass sich diese beiden Verfahren beliebig austauschen lassen, da beide nur die Verifikationsphase verändern, die Suchphase jedoch unangetastet lassen.

5.3.1 Vergleich mit einfacher Verifikation

Wird ein Textbereich $[pos_b, pos_e]$ verifiziert, so kann konzeptionell gesehen die Patchwork-Verifikation gegenüber der einfachen Verifikation nur dann Vorteile haben, wenn Überlappungen mit dem zuletzt verifizierten Textbereich $[oldpos_b, oldpos_e]$ auftreten. Unter Verwendung der in den Gleichungen 5.1.1 und 5.1.2 aufgezeigten Bereichsgrenzen ergibt sich also die Bedingung:

$$t_{new} \leq t_{old} + 2m + 2k - 2\,. \tag{5.3.1}$$

Prinzipiell ist es für eine Ersparnis gegenüber der einfachen Verifikation schon ausreichend, wenn die Patchwork-Verifikation einmal auf zwei aufeinander folgenden Bereichen ausgeführt wird, die die Bedingung 5.3.1 erfüllen. Fällt jedoch der durchschnittliche Abstand zweier Verifikationsbereiche unter die durch die Bedingung vorgegebene Grenze von $2m + 2k - 2$, ist es im Schnitt auch bei jedem Aufruf der Patchwork-Verifikation möglich, von der Überlappung zu profitieren.

Es sei nun angenommen, dass die Buchstaben des Textes gleichverteilt sind, also ein beliebiger Buchstabe an jeder beliebigen Textposition mit der Wahrscheinlichkeit $1/\sigma$ vertreten ist. Zudem sei das Pattern (unter Vernachlässigung nicht restlos durch $k+1$ teilbarer Patternlängen m) gleichmäßig in $k{+}1$ Subpattern der Länge $m/(k{+}1)$ aufgeteilt.

Mit diesen Annahmen ist der durchschnittliche Abstand zweier Verifikationsbereiche (siehe dazu auch Anhang A)

$$t_{new} - t_{old} = \frac{\sigma^{\frac{m}{k+1}}}{k+1}\,. \tag{5.3.2}$$

Damit kann Bedingung 5.3.1 zu

$$\frac{\sigma^{\frac{m}{k+1}}}{k+1} \leq 2m + 2k - 2 \tag{5.3.3}$$

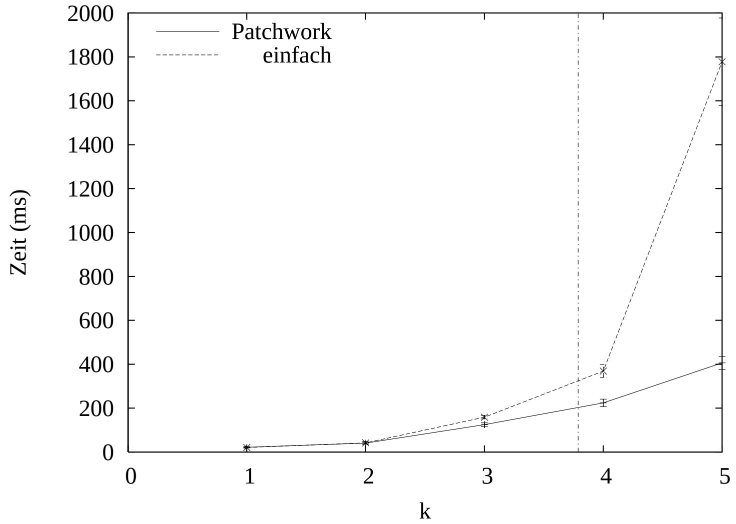
umgeformt werden.

Abbildung 5.2 zeigt zur Illustration dieser Bedingung Beispiele mit zufällig erzeugten Texten. Es ist leicht zu erkennen, dass das Verhalten der Patchwork-Verifikation ab dem durch Bedingung 5.3.3 gegebenen Limit für die maximale Fehlerzahl k (in der Tat sogar schon etwas vorher) besser ist als das der einfachen Verifikation.

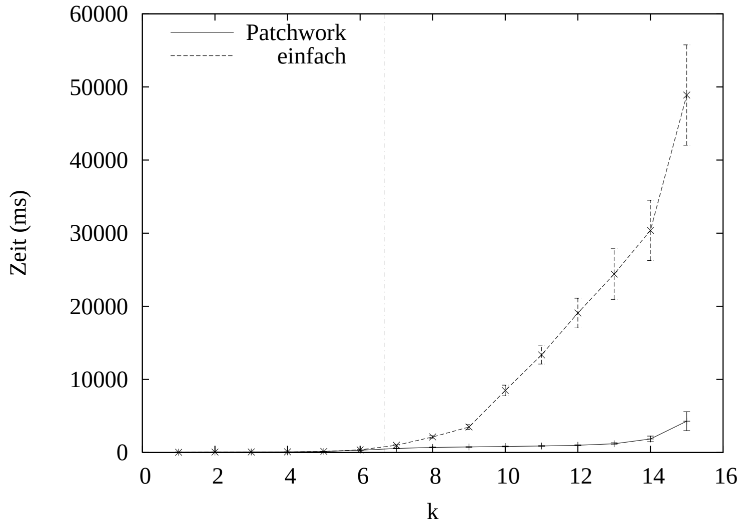
Sind zwei aufeinanderfolgende Verifikationsbereiche $[oldpos_b, oldpos_e]$ und $[pos_b, pos_e]$ sehr nahe beieinander, so sind, wie in Kapitel 5.1 dargelegt, einige Verifikationsergebnisse des ersten Verifikationsbereichs auf den zweiten übertragbar. Wie aus Bedingung 5.1.4 hervorgeht, ist dies der Fall für

$$pos_b + m - 1 \leq oldpos_e\,. \tag{5.3.4}$$

Unter Verwendung der in den Gleichungen 5.1.1 und 5.1.2 fixierten Verifikationsbereichsgrenzen und des durchschnittlichen Abstands zweier Verifikationsbereiche nach Gleichung 5.3.2 ergibt sich nun



(a) $\sigma = 10, n = 1000000, m = 10$



(b) $\sigma = 5, n = 1000000, m = 30$

Abbildung 5.2: Die Bedingung 5.3.3 bei der Suche in Zufallstexten. In (a) ist die Bedingung für $k \geq 3.789$ erfüllt, in (b) für $k \geq 6.66$.

$$\frac{\sigma^{\frac{m}{k+1}}}{k+1} \leq m + 2k - 1 \tag{5.3.5}$$

als Bedingung für das Kopieren von Ergebnissen des ersten Verifikationsbereichs.

Die Grenze, die die Patchwork-Verifikation am stärksten beschränkt, wird erreicht, wenn die Verifikationsbereiche immer näher zusammenrücken und schließlich das Intervall $[pos_b, pos_b + m - 1]$ des Verifikationsbereichs immer überprüft werden muss. Dieses Intervall wird nur dann separat geprüft, wenn dort bereits während der vorangegangenen Verifikation ein Approximate Matching (bzw. das Ende eines solchen) lokalisiert werden konnte, andernfalls wird es nicht wieder betrachtet (vgl. Kapitel 5.1). Es muss also während der vorangegangenen Verifikation das Pattern mit höchstens k Fehlern im Bereich $[oldpos_b, pos_b + m - 1]$ gefunden worden sein. Die Größe dieses Bereichs kann näherungsweise durch m abgeschätzt werden, wobei der Abschätzungsfehler für steigende Fehlerlevel schnell sinkt (dies ergibt sich aus dem in Gleichung 5.3.2 angegebenen durchschnittlichen Abstand zweier Verifikationsbereiche). Die durchschnittliche Edit-Distanz zwischen zwei Pattern der Länge m beträgt nun nach Kapitel 2.7 $m(1-1/\sqrt{\sigma})$ und somit ist in dem genannten Intervall ein Approximate Matching dann vorhanden, wenn

$$m(1 - \frac{1}{\sqrt{\sigma}}) \leq k \tag{5.3.6}$$

gilt.

Abbildung 5.3 zeigt exemplarisch das allgemeine Laufzeitverhalten der Patchwork-Verifikation und setzt die oben berechneten Grenzwerte zugleich hierzu in Relation. Offensichtlich ist ein Einfluss der Bedingung 5.3.5 nicht wahrnehmbar. Der Grund dafür liegt darin, dass diese Bedingung nur beschreibt, wann Informationen über Approximate Matchings im betrachteten Bereich kopiert werden. Doch genau diese Anzahl kopierter Approximate Matchings wächst mit dem Fehlerlevel (also mit k bei festem m), und so kann sich auch kein deutlicher Einfluss an einer bestimmten Stelle zeigen.

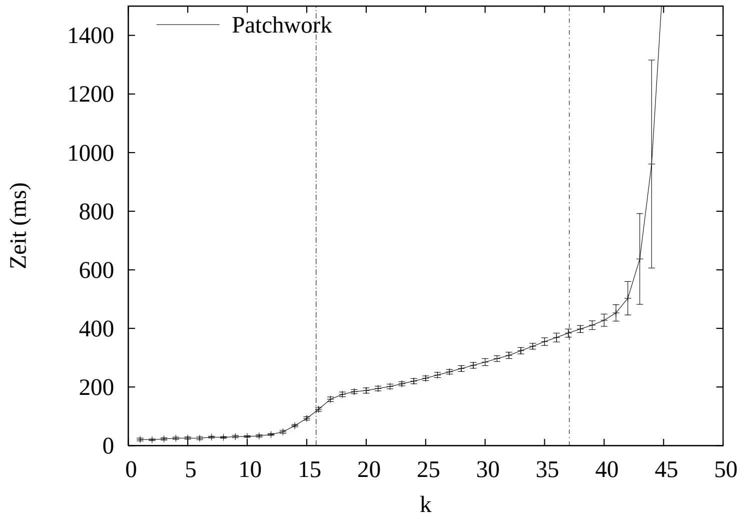
5.3.2 Bereiche dominanter Suchphase

Damit ein Filter-Algorithmus insgesamt die durchschnittliche Laufzeit der Suchphase erreichen kann, ist es wichtig, dass die Verifikationsphase nicht dominiert, d. h. dass über den gesamten Text gesehen die Verifikationskosten $O(n)$ betragen.

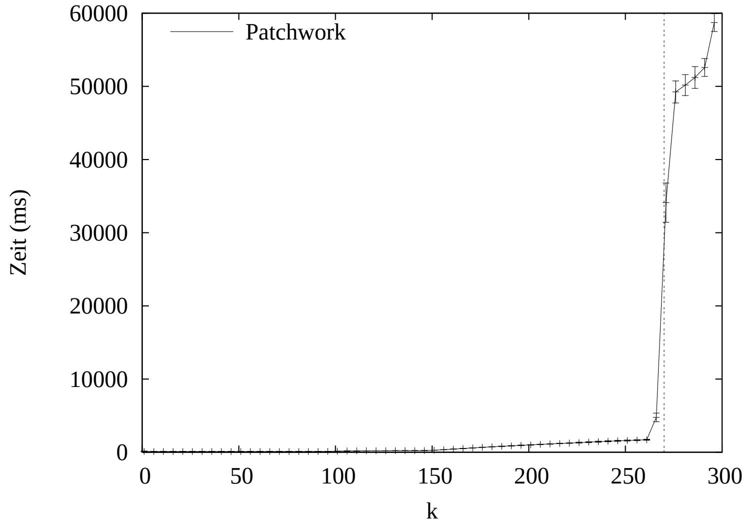
Die Patchwork-Verifikation wird immer dann aufgerufen, wenn eines der $k+1$ Subpattern der Länge $\frac{m}{k+1}$ exakt im Text gefunden wird. Die Wahrscheinlichkeit p dafür, dass dies der Fall ist (siehe auch Anhang A), ist an jeder der n Textstellen

$$p = \frac{k+1}{\sigma^{\frac{m}{k+1}}} .$$

Für die weitere Analyse sei nun davon ausgegangen, dass die Patchwork-Verifikation auch wirklich verwendet wird, also Bedingung 5.3.3 erfüllt ist. Weiterhin sei vorausgesetzt, dass



(a) $\sigma = 4, n = 100000, m = 100$



(b) $\sigma = 100, n = 100000, m = 300$

Abbildung 5.3: Die linke senkrechte Linie in (a) markiert den Grenzwert, für den k die Bedingung 5.3.3 erfüllt, wohingegen die rechte Senkrechte die Grenze für die Bedingung 5.3.5 darstellt. Die Bedingung 5.3.6 ist hier für $k \geq 50$ erfüllt und wird so durch den rechten Rand der Grafik aufgezeigt. In (b) ist nur der Grenzwert der Bedingung 5.3.6 eingetragen und zeigt klar die allgemeine Übereinstimmung des berechneten mit dem gemessenen Wert.

das Intervall $[pos_b, pos_b+m-1]$ des Verifikationsbereichs keiner Überprüfung bedarf, also Bedingung 5.3.6 erfüllt ist. Ferner seien die Kosten für das Kopieren der Positionen von Approximate Matchings im Bereich $[pos_b + m - 1, oldpos_e]$ nicht betrachtet. Damit sind nun die Kosten pro Aufruf der Patchwork-Verifikation quadratisch in der Länge des verifizierten Bereichs $[oldpos_e, pos_e]$, was unter Annahme der Verifikationsbereichsgrenzen aus den Gleichungen 5.1.1 und 5.1.2 genau dem Abstand zwischen zwei exakten gefundenen Subpattern (siehe Gleichung 5.3.2) entspricht. Somit sind als Verifikationskosten also jeweils $(\sigma^{\frac{m}{k+1}}/(k+1))^2$ anzusetzen und es ergibt sich als Gesamtforderung für die Linearität

$$n\frac{k+1}{\sigma^{\frac{m}{k+1}}}\Big(\frac{\sigma^{\frac{m}{k+1}}}{k+1}\Big)^2 = O(n)\,, \tag{5.3.7}$$

was für eine Konstante c äquivalent ist mit

$$\frac{\sigma^{\frac{m}{k+1}}}{k+1} \leq c\,. \tag{5.3.8}$$

Mit $\frac{m}{k+1} \approx \frac{1}{\alpha}$ ergibt sich

$$\frac{\sigma^{\frac{1}{\alpha}}}{k+1} \leq c\,. \tag{5.3.9}$$

Um diese Ungleichung nun nach dem Fehlerlevel α auflösen zu können, wird weiter die folgende, durch Ersetzung von k durch $m-1$ abgeschwächte Ungleichung verwendet:

$$\frac{\sigma^{\frac{1}{\alpha}}}{m} \leq c\,. \tag{5.3.10}$$

Diese Ungleichung kann umgeformt werden zu

$$\frac{1}{\alpha} \leq \log_\sigma cm\,. \tag{5.3.11}$$

Sei nun zudem der Fall betrachtet, in dem der Textbereich $[pos_b, pos_b + m - 1]$ nicht immer überprüft werden muss, also nach Bedingung 5.3.6 für

$$\alpha < 1 - \frac{1}{\sqrt{\sigma}}\,,$$

so ergibt dies zusammen mit Ungleichung 5.3.11 dann die Bedingung

$$\frac{1}{\log_\sigma cm} \leq \alpha < 1 - \frac{1}{\sqrt{\sigma}}\,, \tag{5.3.12}$$

die eine Näherung für den Bereich des Fehlerlevels angibt, in dem die Verifikation insgesamt nur einen linearen Anteil hat. Zusätzlich gilt im Fall, dass die Patchwork-Verifikation nicht wirklich verwendet wird, also wenn zwei aufeinander folgende Verifikationsbereiche zu weit auseinander liegen, immer noch der Grenzwert für die Linearität der allgemeinen Filter, die auf einer Aufteilung des Patterns in $k+1$ Subpattern basieren (siehe Kapitel 4.1.2.3). Somit ist also dem durch Gleichung 5.3.12 charakterisierten Bereich für den Fehlerlevel noch zusätzlich der durch

$$\alpha < \frac{1}{3\log_\sigma m} \tag{5.3.13}$$

beschriebene Bereich hinzuzufügen.

Aus der Bedingung 5.3.12 geht klar hervor, dass für die Patchwork-Verifikation vernünftige Fehlerlevel α bei kleinen Alphabeten und längeren Pattern erreicht werden. Andernfalls kann es sogar sein, dass dieser *Nutzbereich* für α gar nicht existiert. Mit dem Ziel, diese Feststellung qualitativ zu erfassen, soll der durch Bedingung 5.3.12 gegebene Nutzbereich der Patchwork-Verifikation im Folgenden genauer untersucht werden.

Die Größe dieses Nutzbereichs ergibt sich aus der Bedingung 5.3.12 und kann (bei Festlegung von $c=1$) durch die Funktion $f(\sigma, m)$ beschrieben werden mit

$$\begin{aligned} f(\sigma, m) &= 1 - \frac{1}{\sqrt{\sigma}} - \frac{1}{\log_\sigma m} \\ &= 1 - \frac{1}{\sqrt{\sigma}} - \frac{\ln\sigma}{\ln m}. \end{aligned} \tag{5.3.14}$$

Damit der Nutzbereich überhaupt existiert, muss also $f(\sigma, m) > 0$ sein:

$$\begin{aligned} & f(\sigma, m) &>& \; 0 \\ \iff & \tfrac{\sqrt{\sigma}-1}{\sqrt{\sigma}} &>& \; \frac{\ln\sigma}{\ln m} \\ \iff & m &>& \; e^{\frac{\ln\sigma\sqrt{\sigma}}{\sqrt{\sigma}-1}} \\ \iff & m &>& \; \sigma^{\frac{\sqrt{\sigma}}{\sqrt{\sigma}-1}} \end{aligned} \tag{5.3.15}$$

Mit Bedingung 5.3.15 ergibt sich somit ein Kriterium dafür, wie groß das Pattern mindestens sein muss, damit sich der Einsatz der Patchwork-Verifikation bei Fehlerleveln im Nutzbereich lohnen kann. Die Funktion $\sigma^{\frac{\sqrt{\sigma}}{\sqrt{\sigma}-1}}$ entspricht näherungsweise einer Geraden und für große Alphabete kann so dieses Kriterium auch grob zu $m > \sigma$ vereinfacht werden. In Abbildung 5.4 ist die Funktion $f(\sigma, m)$ (eingeschränkt auf den Bereich eines existierenden Nutzbereichs, also auf $f(\sigma, m) > 0$) dargestellt. Der Verlauf der näherungsweise linearen Grenzkurve ist klar ersichtlich. Zudem hat f für jedes m, bzw. jedes σ ein deutliches Maximum, welches sich auch berechnen lässt.

Um für eine gegebene Alphabetgröße σ das m zu berechnen, welches einen maximal großen Nutzbereich erzeugt, wird bei fixiertem m nun von f die Ableitung f' nach σ berechnet:

$$f'_m(\sigma) = \frac{1}{2}\sigma^{-\frac{3}{2}} - \frac{1}{\ln m}\sigma^{-1}. \tag{5.3.16}$$

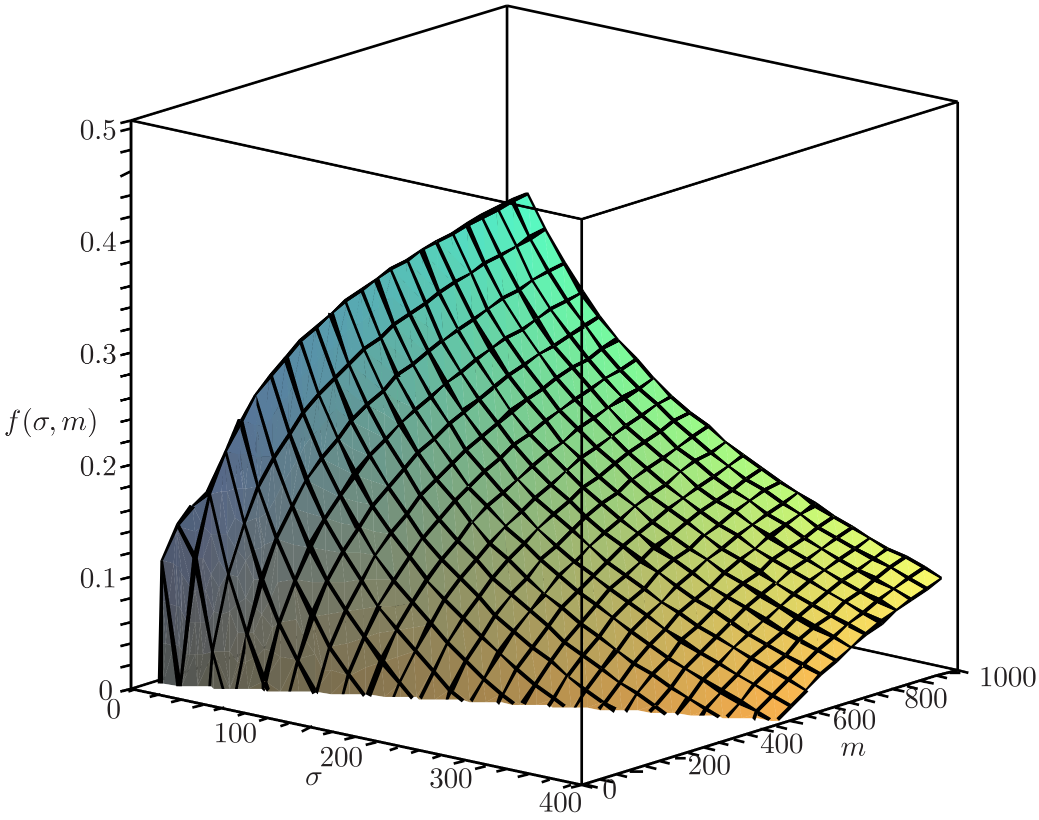


Abbildung 5.4: Der Nutzbereich aus Bedingung 5.3.12. Gezeigt ist hier die Funktion $f(\sigma, m)$, sofern diese positiv ist, da andernfalls der Nutzbereich für α nicht existiert. Die Grenzkurve wird durch Bedingung 5.3.15 charakterisiert. Die für das jeweilige σ maximale Stelle ist bestimmt durch Gleichung 5.3.17.

Extremwerte (in diesem Fall das Maximum, wie leicht überprüfbar ist) von f finden sich an den Nullstellen von f'. Wird also die Gleichung

$$\frac{1}{2}\sigma^{-\frac{3}{2}} - \frac{1}{\ln m}\sigma^{-1} = 0$$

nach m aufgelöst, so ergibt sich mit

$$m = e^{2\sqrt{\sigma}} \tag{5.3.17}$$

für jede Alphabetgröße σ der maximale Nutzbereich. Zur Verdeutlichung der für einen maximalen Nutzbereich nötigen Patterngröße m gegenüber der Alphabetgröße σ ist diese Gleichung 5.3.17 in Abbildung 5.5 dargestellt.

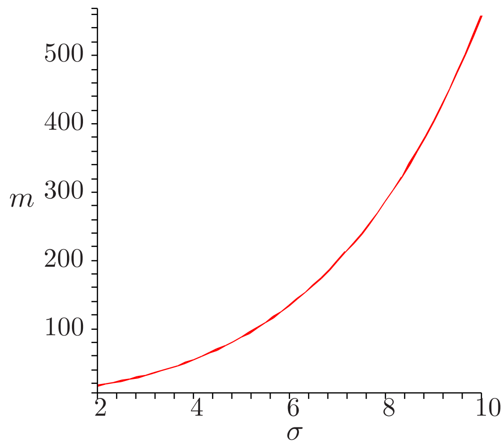


Abbildung 5.5: Abhängigkeit der Patterngröße m von σ für einen maximalen Nutzbereich. Um einen maximalen Nutzbereich zu erhalten muss aufgrund der exponentiellen Abhängigkeit das gesuchte Pattern selbst für recht kleine Alphabete verhältnismäßig lang sein (vgl. Gleichung 5.3.17).

Es sei an dieser Stelle explizit betont, dass der maximale Nutzbereich nicht einen Bereich beschreibt, für das die Patchwork-Verifikation im Sinne der Laufzeit den größten Nutzen hat, sondern nur für eine feste Alphabetgröße das maximale Intervall für den Fehlerlevel angibt, in dem dieser Fehlerlevel theoretisch (unter den oben getätigten Annahmen) für einen insgesamt nur linearen Einfluss auf den Filter sorgt.

5.3.3 Vergleich mit Hierarchischer Verifikation

Bei Verwendung der Hierarchischen Verifikation (siehe Kapitel 4.2.1) ist eine Dominanz der Suchphase für

$$\alpha < \frac{1}{log_\sigma m} \tag{5.3.18}$$

zu erwarten. Dieser Grenzwert entspricht genau der unteren Grenze des durch die Gleichung 5.3.12 gegebenen Intervalls (für $c = 1$). Bei der Hierarchischen Verifikation also verliert die Suchphase genau dann ihre Dominanz, wenn die Patchwork-Verifikation prinzipiell wieder eine lineare Laufzeit ermöglicht. Abbildung 5.6 zeigt genau dieses an einem Beispiel auf. Die durch die Bedingungen 5.3.12 und 5.3.13 gesetzten Grenzen linearer Verifikationszeit können leicht im Verlauf der Kurven erkannt werden, wobei auch offensichtlich und erwartungsgemäß die absolute Laufzeit im zweiten Intervall (Bedingung 5.3.12) höher ist. Zudem zeigt dieses Beispiel auch das recht gute Verhalten der Patchwork-Verifikation bei längeren Pattern und kleineren Alphabeten auf.

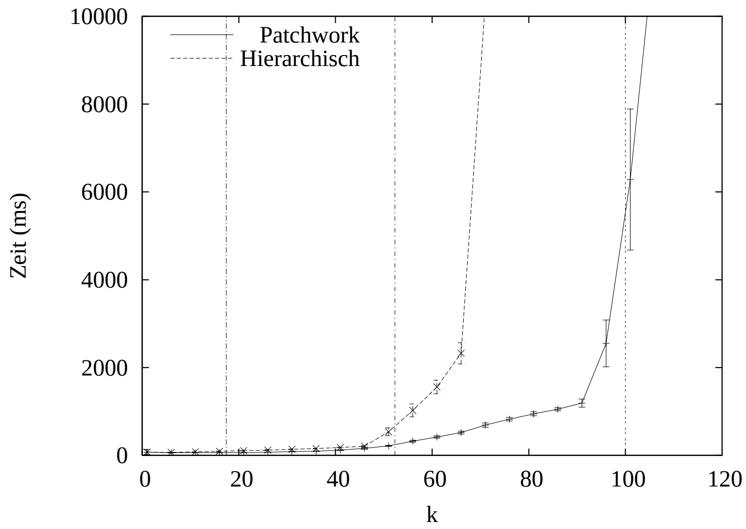
Am Beispiel eines größeren Alphabets zeigt Abbildung 5.7 das gesamte Laufzeitverhalten der Patchwork-Verifikation und der Hierarchischen Verifikation in Abhängigkeit des Fehlerlevels (wie auch bisher gegeben durch die Fehlerzahl k bei konstanter Patterngröße). Für sehr hohe Fehlerlevel (jenseits der oberen Grenze aus Bedingung 5.3.12) kann bei der Patchwork-Verifikation im Prinzip eine immer gleich hohe Laufzeit festgestellt werden. Diese liegt darin begründet, dass (fast) an jeder Stelle ein mögliches Approximate Matching identifiziert wird und so immer der ganze Text verifiziert wird. Ähnliches ist bei der Hierarchischen Verifikation zu sehen. Dort werden etwa mit Erreichen des Endes des Linearitätsintervalls (also Bedingung 5.3.18 bzw. die untere Grenze von Bedingung 5.3.12) die Ähnlichkeiten so stark, dass die Zahl der zu verifizierenden Hierarchischen Stufen zunimmt, bis schließlich jede Stufe immer verifiziert werden muss. Jedoch sind die Verfahren für die eben beschriebenen sehr hohen Fehlerlevel allerdings auch nicht gedacht.

Abbildung 5.8 zeigt wie Abbildung 5.7 die Suche im Text basierend auf einem Alphabet der Größe 28, der allerdings diesmal weder zufällig ist noch die Eigenschaften eines zufälligen Textes aufweist, sondern in Englisch abgefasst ist. Offensichtlich findet bei dem nicht mehr zufälligen Alphabet in der Praxis eine Verschiebung der durch die obigen Bedingungen errechneten Grenzen statt. Recht grob kann diese Verschiebung über die Entropie erfasst werden.

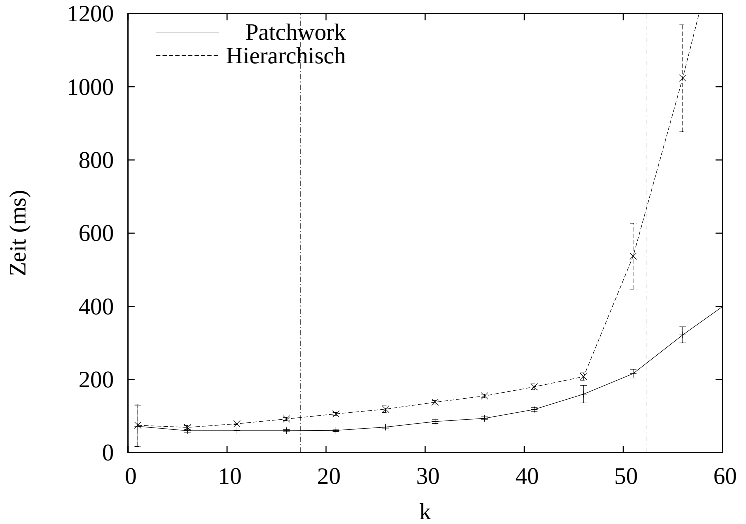
Die von Shannon [110] in die Informationstheorie eingeführte Entropie einer Information $H(I)$ ist definiert als:

$$H(I) = -\sum_i p_i \log_2 p_i \, .$$

Dabei gibt p_i die Wahrscheinlichkeit an, mit der das an der Stelle i der Information I vorgefundene Symbol allgemein in der Information I auftritt.

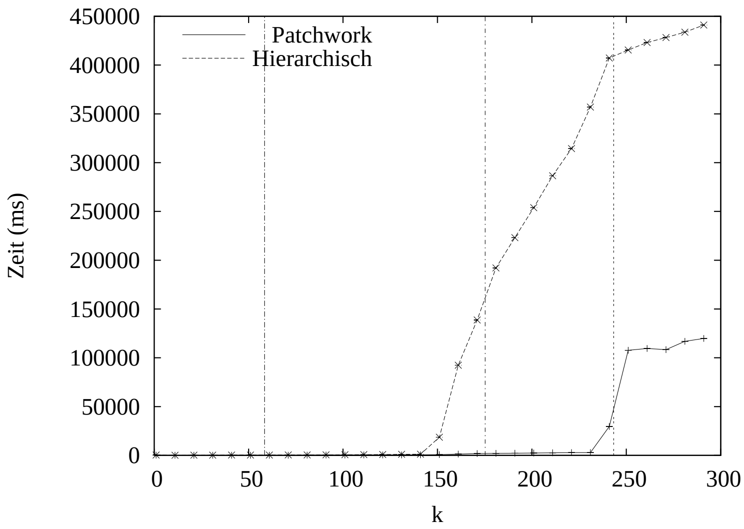


(a)

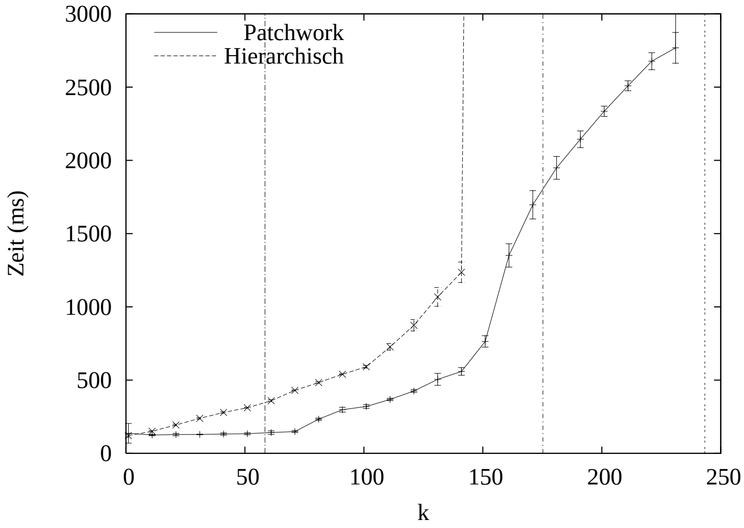


(b)

Abbildung 5.6: Patchwork-Verifikation und Hierarchische Verifikation auf einem Zufallstext der Länge 10000 aus einem vier-buchstabigen Alphabet bei der Suche nach einem Pattern der Länge $m = 200$. (a) Die linke senkrechte Linie markiert den Grenzwert für α bei einfacher Verifikation (Gleichung 5.3.13). Die mittlere Senkrechte zeigt den durch die Bedingung 5.3.18 gegebenen Grenzwert an, der auch der unteren Intervallgrenze in Bedingung 5.3.12 entspricht. Die obere Grenze dieses Intervalls wird durch die rechte Senkrechte aufgezeigt. (b) Darstellung des ersten Teils des in (a) gezeigten Graphen zur Verdeutlichung von Details.

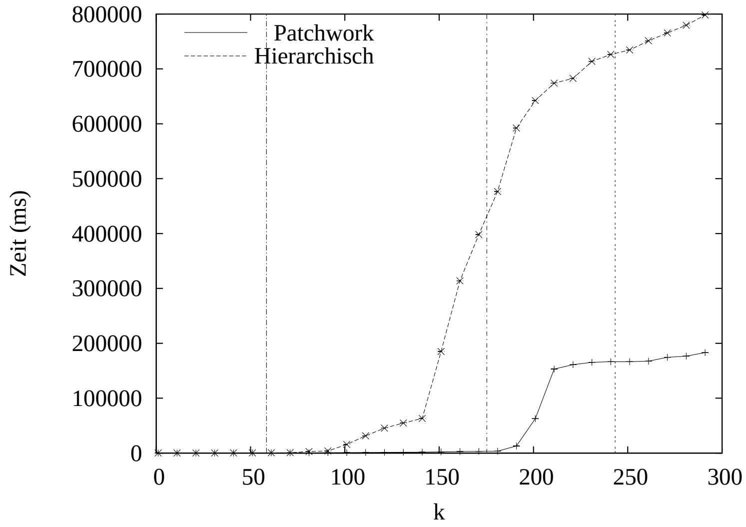


(a)

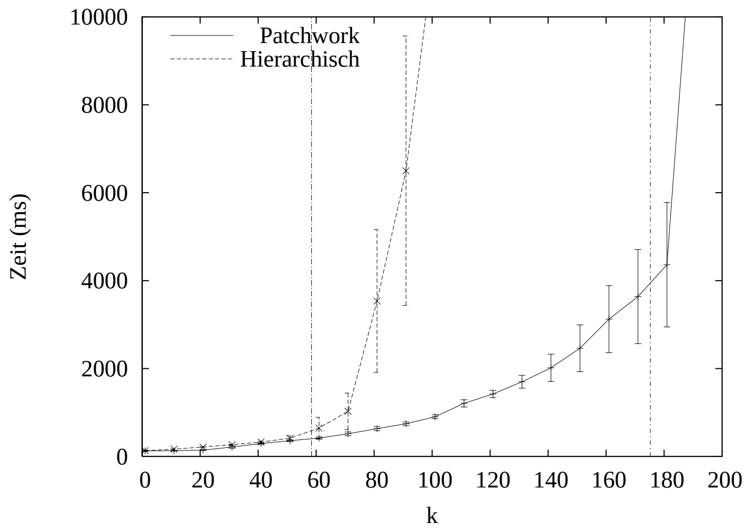


(b)

Abbildung 5.7: Suche in einem zufälligen Text mit $\sigma = 28$, $n = 100000$ und $m = 300$. In (a) ist die Laufzeit der Suche eines Patterns unter Berücksichtigung des gesamten Spektrums möglicher Fehlerlevel (hier im Abstand von jeweils 1/30) dargestellt. (b) zeigt anhand mehrerer solcher Suchen bei Einschränkung der maximalen Berechnungszeit den Bereich bis zur oberen Grenze des Nutzbereichs der Patchwork-Verifikation an. Sowohl in (a) als auch in (b) sind die wichtigen Bedingungen 5.3.13 (linke senkrechte Line) und 5.3.12 (der Nutzbereich, gegeben durch die beiden rechten Senkrechten) markiert.



(a)



(b)

Abbildung 5.8: Suche in einem englischen Text mit $\sigma = 28$, $n = 100000$, $m = 300$. (a) und (b) entsprechen der Abbildung 5.7, wobei hier allerdings ein englischsprachiger Text zugrunde liegt. Wieder sind die wichtigen Bedingungen genauso markiert, aber es ist deutlich, dass diese theoretischen Werte nicht mehr so mit den gemessenen Werten in Einklang stehen.

Über die Entropie kann die Anzahl an Bits berechnet werden, die zur Darstellung eines Zeichens des Textes notwendig sind. Dazu wird für jeden Buchstaben i des Alphabets die Wahrscheinlichkeit p_i berechnet, mit der dieser an einer Textstelle verwendet wird (die Wahrscheinlichkeit ist gleich der Häufigkeit des Buchstabens im Text geteilt durch die Textlänge). Danach wird nach der gegeben Entropieformel für alle σ Symbole σ_i des Alphabets summiert, also:

$$H = -\sum_{i=1}^{\sigma} p_i \log_2 p_i \quad \text{mit } p_i = \frac{1}{n} \sum_{j=1}^{n} \begin{cases} 1 & \text{falls } t_j = \sigma_i \\ 0 & \text{falls } t_j \neq \sigma_i \end{cases} .$$

Liegt im Text eine vollständige Gleichverteilung vor, also $p_i = 1/\sigma$ für alle i, so entspricht, wie leicht zu sehen ist, die Entropie H dem Logarithmus (zur Basis 2) der Alphabetgröße σ. Andersherum kann auf diese Weise ein Text, für den die Entropie x festgestellt wurde, in erster Näherung mit einem gleichverteilten Zufallstext auf Basis eines Alphabets der Größe 2^x verglichen werden.

Der in Abbildung 5.8 verwendete Text hat einen Entropiewert von $4,02$ und kann demnach näherungsweise verglichen werden mit einem reinen Zufallstext basierend auf einem Alphabet der Größe $2^{4,02} \approx 16$. Abbildung 5.9 zeigt dieselbe Kurve wie Abbildung 5.8(a), diesmal wurden jedoch die Grenzwerte der Bedingungen unter Betrachtung eines Alphabetes der Größe 16 eingetragen. Die Übereinstimmung der theoretischen Grenzen mit den praktischen Werten ist nun deutlich größer als zuvor.

Der hier auch verwendete DNS-Datensatz hat einen Entropiewert von 2.04. Dies ist ein starkes Indiz dafür, dass dieser Datensatz (auch wenn, wie in Kapitel 5.2.2 erläutert, das reale Alphabet dahinter aus mehr als vier Buchstaben besteht) mit einem Zufallstext basierend auf einem Alphabet der Größe $\sigma = 4$ verglichen werden kann. Abbildung 5.10 zeigt vergleichsweise den Verlauf der Suchzeiten in einem Zufallstext und in dem DNS-Datensatz. Der prinzipielle Verlauf beider Kurven ist nahezu identisch und bestätigt damit zugleich die Nutzbarkeit der Patchwork-Verifikation bei der Suche in realen DNS-Daten, die aufgrund verschiedener Unsicherheiten dann doch ein größeres Alphabet zur Grundlage haben, als nur das bekannte aus den vier Basen `A`,`C`,`G` und `T` bestehende Alphabet.

5.4 Resultat

In diesem Kapitel wurde ein Verfahren zur Verifikation gezeigt, welches den Filteransatz der Aufteilung in die exakte Suche auch für höhere Fehlerlevel α anwendbar macht, dabei gleichzeitig völlig unabhängig von der konkreten Implementierung der Suchphase ist. Das Verfahren liefert bei jedem Aufruf immer alle Approximate Matchings im betrachteten Bereich, kann aber leicht dahingehend vereinfacht werden, nur Approximate Matchings zu liefern, die nicht auch im Bereich der letzten Verifikation liegen.

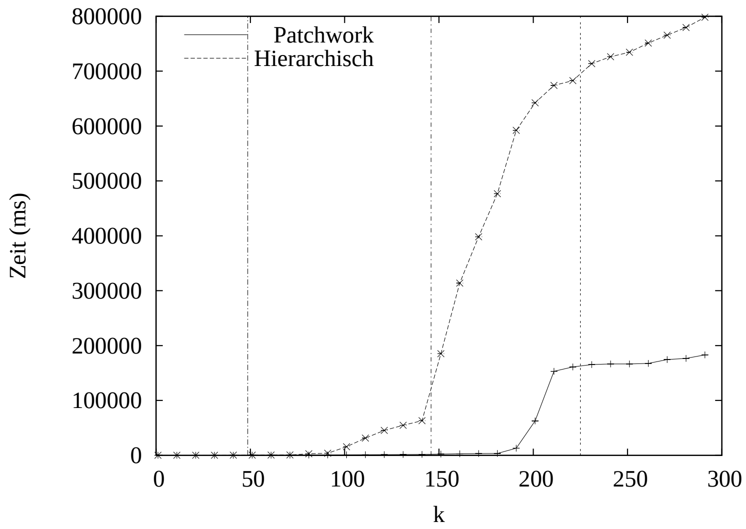
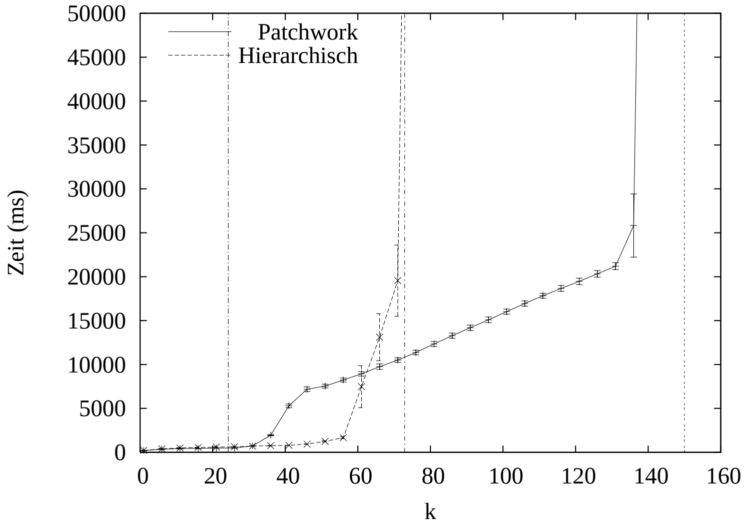


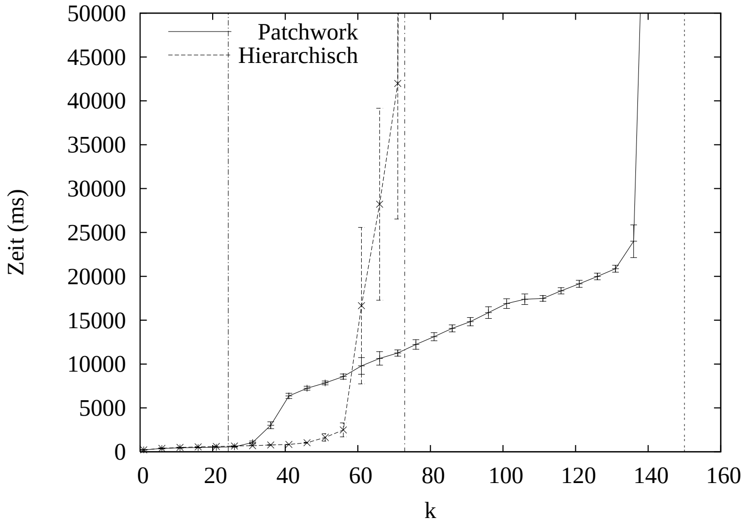
Abbildung 5.9: Suche in einem englischen Text mit $\sigma = 28$, $n = 100000$, $m = 300$. Die senkrechten Linien markieren die Bedingungen 5.3.13 (linke Line) und 5.3.12 (der Nutzbereich gegeben durch die beiden rechten Linien). Als Grundlage für die Berechnung der Grenzwerte dieser Bedingungen wurde die sich aus der Entropie ergebende Alphabetgröße von 16 verwendet.

Im Rahmen der Analyse konnten die Grenzen der Anwendbarkeit dieses Verfahrens berechnet und demonstriert werden. Es hat sich dabei gezeigt, dass das Verfahren gerade bei längeren Pattern und kleineren Alphabeten (wie z. B. DNS) bei gleichzeitig relativ hohen Fehlerleveln im Vergleich gut abschneidet. Der vollständige Nutzbereich der Patchwork-Verifikation wird durch die Gleichungnen 5.3.12 und 5.3.13 beschrieben. Doch insbesondere zeigt sich dabei, dass die Patchwork-Verifikation von genau dem Fehlerlevel an wieder mit einer insgesamt linearen Laufzeit aufwarten kann, ab dem die Hierarchische Verifikation diese Eigenschaft verliert.

Das Verfahren der Patchwork-Verifikation wurde speziell für die Zerlegung des Pattern in $k + 1$ Teilpattern konzipiert. Im Allgemeinen spricht jedoch nichts dagegen, das Verfahren auch bei einer $k + s$ Zerlegung zu verwenden, allerdings sind dann die analytischen Ergebnisse, die auf der $k + 1$ Zerlegung basieren, so nicht mehr gültig.



(a)



(b)

Abbildung 5.10: Vergleich der Suche in einem Zufallstext mit $\sigma = 4$ (a) mit der Suche in einem DNS Datensatz (b). In beiden Fällen stimmen die Längen der Suchpattern ($m = 300$) und Textes (Länge des vollständigen gegebenen Genoms) überein. In beiden Grafiken sind wieder die Bedingungen 5.3.13 (linke Line) und 5.3.12 (der Nutzbereich gegeben durch die beiden rechten Linien) eingetragen, wobei als Berechnungsgrundlage in beiden Fällen $\sigma = 4$ verwendet wurde.

6 Grammatiken

Ordnung ist die Lust der Vernunft, aber Unordnung ist die Wonne der Phantasie.

(Paul Claudel, frz. Diplomat, 1868-1955)

Die vorangehenden Kapitel zeigten verschiedene Algorithmen zur Lösung des ASM-Problems auf und ebenso auch verschiedene Methoden zur Verbesserung des Filteransatzes für dieses Problem. Ein zentraler Punkt dieser Arbeit ist die Idee, eine Grammatik bei der Lösung des ASM-Problems zu nutzen.

In diesem Kapitel soll zuerst in Kapitel 6.1 die Verbindung zwischen dem Problem des Approximate String Matching, insbesondere bei Betrachtung des Filteransatzes (vgl. Kapitel 4.1), und der möglichen Nutzung von Grammatiken verdeutlicht werden. Anschließend soll in Kapitel 6.2 ein kurzer Abschnitt über Grammatiken im Allgemeinen die grundlegenden Begriffe erläutern. Aus der dann in Kapitel 6.3 folgenden Übersicht über verschiedene Verfahren zur Erzeugung von Grammatiken ergibt sich zwangsläufig in dieser Arbeit die Konzentration auf den Sequitur-Algorithmus, da dieser nicht nur sehr effizient arbeitet, sondern zugleich auch die für das ASM-Problem gegebenen Eingabedaten direkt verarbeitet.

6.1 Grammatiken und das ASM-Problem

Um den möglichen Einsatz von Grammatiken beim Approximate String Matching absehen zu können, ist es sinnvoll, Klarheit darüber zu gewinnen, was von einer Grammatik erwartet wird, bzw. was sie zu „leisten“ vermag.

Es wird hier für das Approximate String Matching davon ausgegangen, dass neben dem Text (und dem Suchpattern) keine weiteren Informationen vorliegen. Die Herleitung einer Grammatik zu diesem Text kann also nur auf dem Text selbst basieren. Dies ist nicht generell bei jeder Herleitung von Grammatiken der Fall. So werden bei manchen Problemen in der Linguistik zusätzliche Informationen für einen Grammatikaufbau verwendet wie beispielsweise die Zuordnung von Wortklassen zu Wörtern. Im vorliegenden Fall erzeugt eine Grammatik des Textes eine rein text-inhärente Struktur. Aus dieser

Struktur gewonnene Erkenntnisse beinhalten (betrachte dazu in Kapitel 6.3 die verschiedenen Methoden, Grammatiken zu erzeugen) in Form von Regeln Informationen über im Sinne der Grammatik gleichwertige Textbereiche. Durch die Verschachtelung von Regeln ist diese Information in hierarchischer Form sehr fein gegliedert.

Für das Approximate String Matching kann daraus potentiell ein Nutzen gezogen werden, wenn die Gleichwertigkeit von Textbereichen im Sinne der Grammatik auch einer Gleichwertigkeit von Textbereichen im Sinne des ASM entspricht. Dann ist nämlich jede Regel prinzipiell nur einmal zu betrachten, egal wie oft diese zur Erzeugung des Textes Verwendung findet.

In der Praxis entspricht die Gleichwertigkeit von Textbereichen im Sinne des ASM der Identität von Textbereichen; denn schließlich muss es möglich sein, jedes Auftreten des Patterns (also an jeder möglichen Stelle) im Text (mit einer beliebigen Anzahl an Fehlern) genau zu identifizieren. Die von der Grammatik identifizierte Struktur des Textes entspricht dann einer Wiederholungsstruktur: jede Regel zeigt einen Textbereich auf, der identisch an verschiedenen Stellen im Text vorkommt. Derartige Wiederholungen entsprechen im Prinzip kodierungstechnisch redundanten Informationen und die Identifikation dieser Redundanzen kann als eine Vorstufe für Kompressionsverfahren betrachtet werden. In der Tat wird zum Beispiel das Sequitur-Verfahren (siehe Kapitel 6.3.4) zur Erzeugung einer Grammatik auch für ein Kompressionsverfahren genutzt [96].

Für das ASM ergibt sich unter Betrachtung einer geeigneten Grammatik also die Möglichkeit, nur den redundanzreduzierten Text zu berücksichtigen, bzw. konkret jede Regel nur einmal zu bearbeiten. Der Filteransatz (siehe Kapitel 4.1) erscheint dazu fast ideal geeignet, da er darauf basiert, schnell ganze Bereiche des Textes zu untersuchen und auf eine weitere nötige Betrachtung hin zu klassifizieren. Ein Filter, der also eine ganze Regel als nicht relevant verwirft, erspart durch die Grammatikkonstruktion somit auch die Betrachtung jeder weiteren Stelle, an der die Regel Verwendung findet. Ist es darüber hinaus notwendig, eine Regel eingehender zu untersuchen, so ist diese Untersuchung ebenfalls nur einmal für jede Regel notwendig (und nicht für jedes Auftreten der Regel).

Im Hinblick auf diese Nutzung von Grammatiken sollen im Folgenden Grammatiken und deren Erzeugung näher betrachtet werden.

6.2 Formales zu Grammatiken

Formal gesehen ist eine *Grammatik* G ein Tupel bestehend aus vier Komponenten. Für eine Grammatik $G = (N, \Sigma, P, S)$ bezeichnet N die Menge der *nicht-terminalen Symbole*, oder auch einfach *Nicht-Terminale* genannt. Nicht-Terminale sind spezielle Symbole, die während der Produktion von Sätzen durch die Grammatik Verwendung finden, jedoch niemals in einem von der Grammatik vollständig erzeugten Wort vorkommen (der Begriff „nicht-terminal“ deutet an, dass das Symbol noch weiterer Auflösung bedarf).

Die Menge Σ ist die Menge der *terminalen Symbole*, oder auch *Terminale*. Sie ist disjunkt zur Menge der Nicht-Terminalen, also $N \cap \Sigma = \emptyset$. Die Menge der Terminalen ist das Alphabet für alle durch die Grammatik erzeugten Sätze.

P bezeichnet die Menge der *Regeln*, auch *Produktionen* genannt. Dabei gilt im Allgemeinen:

$$P \subseteq (N \cup \Sigma)^* \cdot N \cdot (N \cup \Sigma)^* \times (N \cup \Sigma)^*.$$

Dies bedeutet letztlich, dass bei Anwendung einer Regel $R \in P$ ein Wort w aus Terminalen und Nicht-Terminalen (wobei mindestens ein Nicht-Terminal enthalten sein muss) durch ein anderes solches Wort $\bar{w}$, das allerdings auch leer sein kann und nicht zwingend ein Nicht-Terminal enthalten muss, ersetzt wird. Notiert wird dies üblicherweise einfach als $w \rightarrow \bar{w}$.

Durch verschiedene Einschränkungen der Regeln entsteht die so genannte Chomsky-Hierarchie [24]. Sofern nicht anders angemerkt, beschränkt sich diese Arbeit auf *kontextfreie* Grammatiken, bei denen die Menge der Regeln im Vergleich zum allgemeinen Fall so eingeschränkt ist, dass

$$P \subseteq N \times (N \cup \Sigma)^*$$

ist.

S bezeichnet das Startsymbol und dient als Ausgangspunkt für die Anwendung der Regeln. Damit die Regeln dann überhaupt etwas bewirken können, muss notwendigerweise S zu der Menge der Nicht-Terminale gehören, also $S \in N$.

6.3 Erzeugung von Grammatiken

Die Erzeugung von Grammatiken ist ein in der Linguistik beheimatetes Problem. Verschiedene zentrale Fragen, wie z. B. die Frage danach, wie Kinder die Muttersprache lernen, führten zu der Aufgabe der Erzeugung von Grammatiken. Häufig anhand spezieller künstlicher Sprachen wurden einige Algorithmen zur Erzeugung von Grammatiken auf der Basis von gegebenen Sätzen entwickelt [6]. Die für den allgemeinen Fall wichtigsten Verfahren werden in den jetzt folgenden Kapiteln 6.3.1 bis 6.3.4 erläutert.

Bei der Betrachtung dieser Verfahren bleibt jedoch festzuhalten, dass zum Zwecke der Nutzung einer generierten Grammatik beim Approximate String Matching keine verallgemeinernde Grammatik erwünscht ist, sondern die Grammatik exakt den Eingabetext reproduzieren muss (vgl. Kapitel 6.1). Diese Bedingung erfüllen nicht alle der hier genannten Verfahren, doch sollen auch sie der Vollständigkeit halber genannt werden. Weiterhin steht beim Approximate String Matching nur ein Text in Form eines langen „Satzes“ und nicht mehrerer Sätze zur Verfügung. Bei Beachtung des problemlosen Umgangs mit dieser Beschränkung sind insbesondere die Algorithmen MK10 von Wolff (in Kapitel 6.3.1) und Sequitur von Nevill-Manning und Witten (in Kapitel 6.3.4) zu nennen. Da der Sequitur-Algorithmus die größte Flexibilität bei zugleich größter Effizienz zeigt, bildet dieser die ideale Basis für den Einsatz beim Approximate String Matching.

6.3.1 Wort-Segmentierung nach Wolff

Ein Gebiet der Linguistik versucht Wortgrenzen vorherzusagen. Bei dieser so genannten *Linguistischen Segmentierung* dient als Eingabe ein Datenstrom einer Sprache und die Ausgabe liefert eine Segmentierung dieses Datenstroms in Wörter.

Wolff zeigt einen Algorithmus MK10, der durch Analyse von Wiederholungen Wortgrenzen entdecken soll und dabei eine Grammatik der Eingabe aufbaut [134, 135, 136]. Dazu durchläuft er die Eingabe von links nach rechts und betrachtet dabei die *Digramme* des Textes. Ein *Digramm* ist ein q-Gramm mit $q = 2$, also einfach eine Einheit zweier aufeinanderfolgender Textsymbole. Immer, wenn während des Textdurchlaufs ein Digramm häufiger als zehn Mal (ein von Wolff willkürlich gewählter Wert) gesehen wird, so wird dieses Digramm durch ein nicht-terminales Symbol ersetzt, also durch ein Symbol, welches nicht im Alphabet des Textes enthalten ist und eine Regel darstellt. Die Bearbeitung des Textes wird dann entweder fortgesetzt, oder am Textanfang wieder neu begonnen, wobei in jedem Fall aber die bis dahin gezählten Häufigkeiten der Digramme für die neu beginnende Zählung auf Null gesetzt werden. Der Algorithmus terminiert, wenn in einem vollständigen Durchlauf kein Digramm ersetzt wurde.

In einer späteren Version wurde der Algorithmus leicht in der Form modifiziert, dass der Text immer vollständig durchlaufen wird, bevor dann das am häufigsten auftretende Digramm ersetzt wird. Abbildung 6.1 zeigt exemplarisch die Funktionsweise des Algorithmus.

`abcabacab`	$\rightarrow$	$X\texttt{c}X\texttt{ac}X$	$\rightarrow$	$XY\texttt{a}Y$
`ab`: 3	$X = \texttt{ab}$	$X\texttt{c}$: 1		XY: 1
`bc`: 1		$\texttt{c}X$: 2	$Y = \texttt{c}X$	$Y\texttt{a}$: 1
`ca`: 2		$X\texttt{a}$: 1		$\texttt{a}Y$: 1
`ba`: 1		`ac` : 1		
`ac`: 1				

Abbildung 6.1: Das Prinzip des Algorithmus von Wolff, angewendet am Beispiel des Textes $T = \texttt{abcabacab}$. Die verschiedenen Textdurchläufe sind nebeneinander dargestellt. Das im ersten Durchlauf am Häufigsten auftretende Digramm `ab` (kommt dreimal vor), wird durch das Symbol X ersetzt. Nach dem zweiten Durchlauf wird $\texttt{c}X$ durch Y ersetzt, wohingegen nach dem dritten Durchlauf kein Digramm mehrfach auftaucht und somit der Algorithmus terminiert.

Dadurch, dass für jede Ersetzung der Text einmal durchlaufen werden muss, ergibt sich im schlechtesten Fall (worst case) eine Laufzeit von $O(n^2)$, wobei n die Größe des Eingabetextes darstellt.

6.3.2 Vollständige Menge aller Grammatiken nach VanLehn und Ball

VanLehn und Ball [128] erzeugen eine Grammatik, um für unbekannte Sätze eine Aussage treffen zu können, ob diese zu der durch die Grammatik definierten Sprache gehören oder nicht. Zur Erzeugung der Grammatik werden als Eingabe positive und negative Beispielsätze verwendet, also Sätze, die gesichert zur Sprache gehören und auch solche, die definitiv nicht zur Sprache gehören. Jeder zur Sprache gehörige Satz soll durch die Grammatik beschrieben werden, während die negativen Beispiele keinesfalls durch die Grammatik beschrieben werden dürfen.

Der Algorithmus von VanLehn und Ball verwendet die Idee, alle *möglichen* Grammatiken, also die Grammatiken, die konsistent mit den verwendeten Beispielsätzen sind, zugleich zu generieren und verfügbar zu halten. Dieser Ansatz hat den Vorteil, dass prinzipiell ein inkrementelles Erzeugen („Lernen“) der Grammatik durch Hinzufügen weiterer Beispielsätze möglich ist, da immer alle notwendigen Informationen vollständig vorhanden sind.

Die direkte Umsetzung dieser Idee wird allerdings durch die prinzipiell unendliche Größe des Raumes aller möglichen Grammatiken verhindert. VanLehn und Ball erreichen durch sinnvolle Einschränkung der Grammatikform eine endliche Größe des Raumes der möglichen Grammatiken. Zugelassen werden nur kontextfreie Grammatiken, die zusätzlich den folgenden Bedingungen genügen:

1. Die rechte Seite einer Regel darf nicht leer sein.

2. Befindet sich auf der rechten Seite der Regel nur ein einzelnes Symbol, so handelt es sich bei diesem Symbol in jedem Fall um ein Terminal.

3. Jedes nicht-terminale Symbol findet zur Erzeugung eines positiven Beispielsatzes Verwendung.

Eine Grammatik, die diesen Bedingungen genügt, wird *simpel* genannt. Hopcroft und Ullman [53] haben gezeigt, dass diese Bedingungen für kontextfreie Grammatiken keine wirkliche Einschränkung darstellen, da jede kontextfreie Grammatik in eine simple Grammatik überführt werden kann.

VanLehn und Ball fordern als weiteres Kriterium die *Reduziertheit* einer Grammatik, was bedeutet, dass das Entfernen einer beliebigen Regel die Grammatik inkonsistent zu den Beispielsätzen macht.

Um eine einfache Erzeugung aller möglichen Grammatiken zu ermöglichen, lockern VanLehn und Ball das Kriterium der Reduziertheit ein wenig (d. h. in einigen Fällen können nicht-reduzierte Grammatiken auftreten). So erhalten sie einen Algorithmus, der inkrementell für jeden neuen Beispielsatz aus der Menge der bisher möglichen Grammatiken die neue Menge aller möglichen Grammatiken erzeugt.

Es ist anzumerken, dass der Algorithmus trotz der gemachten Einschränkungen nicht effizient, d. h. im schlechtesten Fall nicht polynomiell, ist. Eine genaue Laufzeitanalyse wird von VanLehn und Ball jedoch nicht gegeben.

6.3.3 Regelkonstruktion und -verschmelzung auf Satzmengen

Von Langley [68] stammt ein Algorithmus, der im Prinzip dasselbe Ziel verfolgt wie der Algorithmus von VanLehn und Ball (siehe Kapitel 6.3.2). Aus verschiedenen Beispielsätzen (sowohl positiver, als auch negativer) soll eine Grammatik gebildet werden, die eine Sprache beschreibt und dann dazu dient, für andere Sätze eine Aussage zu machen, ob diese zu der Sprache gehören oder nicht. Der Algorithmus von Langley betrachtet als kleinste Einheit Worte (um auch Wortklassen identifizieren zu können), könnte jedoch problemlos auf die Anwendung mit einzelnen Buchstaben übertragen werden.

Ein wesentliches Grundprinzip bei der Konstruktion der Grammatik ist das Bilden von Regeln auf der Basis von wiederholt auftretenden Satzteilen im Datensatz.

Der Algorithmus erwartet als Eingabe eine Menge an positiven und negativen Beispielsätzen. Initial wird aus den Beispielsätzen eine hierarchisch flache Grammatik aufgestellt, die mit den gegebenen positiven Beispielsätzen konsistent ist und zugleich keines der negativen Beispiele zulässt. Dazu wird für jedes Wort innerhalb eines jeden positiven Beispielsatzes eine eigene Regel aufgestellt. Dies ist beispielhaft in den oberen beiden Abschnitten in Abbildung 6.2 dargestellt.

Aus dieser initialen Grammatik wird weitergehend in einem „Lernprozess“ eine nach einem bestimmten Evaluationsmaß minimale Grammatik erzeugt. Das von Langley verwendete Maß ist die *Einfachheit* einer Grammatik, welches die Gesamtzahl der auf der rechten Seite der Regeln notierten Symbole zählt (beachte, dass Langley wortbasiert arbeitet und so nicht jeder Buchstabe eines Wortes zählt, sondern nur das Wort selbst). Die im zweiten Abschnitt in Abbildung 6.2 dargestellte Grammatik hat so einen Einfachheitswert von 30.

Der „Lernprozess“ selbst wechselt zwischen Phasen des Verschmelzens von Regeln und des Erzeugens von Regeln. Während das Erzeugen von Regeln die durch die Grammatik abgedeckte Menge an möglichen Sätzen nicht verändert, verallgemeinert das Verschmelzen die Grammatik, reduziert aber keinesfalls die Abdeckung durch die Grammatik. Das bedeutet, dass während des „Lernprozesses“ die von der initialen Grammatik erzeugten Sätze auch immer weiterhin erzeugt werden können.

Die Phase des Verschmelzens von Regeln (mit der nach dem Erzeugen der initialen Grammatik fortgefahren wird) betrachtet alle Möglichkeiten, Regeln paarweise zu vereinen. Von diesen Möglichkeiten wird mit der Grammatik fortgefahren, die die größte Einfachheit aufweist. Sind mehrere der erzeugten Grammatiken von der gleichen Einfachheit, so wird zufällig eine davon ausgewählt. Dieses Verschmelzen wird solange wiederholt, bis unter dem Einfachheitsmaß einmal keine Verbesserung mehr erzielt werden konnte. Mit der dann aktuellen Grammatik wird in der Phase der Regelerzeugung fortgefahren.

```
die Katze sah die Maus
die Katze hörte eine Maus
eine Katze sah
die Maus hörte
eine Katze hörte die Maus
eine Maus sah
```

S	$\rightarrow$	$R_1R_2R_3R_1R_4$	R_1	$\rightarrow$	`die`
S	$\rightarrow$	$R_1R_2R_5R_6R_4$	R_2	$\rightarrow$	`Katze`
S	$\rightarrow$	$R_6R_2R_3$	R_3	$\rightarrow$	`sah`
S	$\rightarrow$	$R_1R_4R_5$	R_4	$\rightarrow$	`Maus`
S	$\rightarrow$	$R_6R_2R_5R_1R_4$	R_5	$\rightarrow$	`hörte`
S	$\rightarrow$	$R_6R_4R_3$	R_6	$\rightarrow$	`eine`

S	$\rightarrow$	$R_1R_2R_3R_1R_2$	R_2	$\rightarrow$	`Katze`
S	$\rightarrow$	$R_1R_2R_3$	R_2	$\rightarrow$	`Maus`
R_1	$\rightarrow$	`die`	R_3	$\rightarrow$	`sah`
R_1	$\rightarrow$	`eine`	R_3	$\rightarrow$	`hörte`

S	$\rightarrow$	$R_7R_3R_7$	R_2	$\rightarrow$	`Katze`
S	$\rightarrow$	R_7R_3	R_2	$\rightarrow$	`Maus`
R_1	$\rightarrow$	`die`	R_3	$\rightarrow$	`sah`
R_1	$\rightarrow$	`eine`	R_3	$\rightarrow$	`hörte`
R_7	$\rightarrow$	R_1R_2			

Abbildung 6.2: Das Prinzip der Grammatikbildung nach Langley. Positive Beispielsätze, ganz oben angegeben, führen zu der initialen Grammatik, die im zweiten Abschnitt zu sehen ist. Der Phase des Verschmelzens von Regeln führt zu der im dritten Abschnitt gezeigten Grammatik, welche wiederum durch die Phase der Erzeugung von Regeln in die Grammatik des letzten Abschnitts transferiert wird.

Die Grammatik im dritten Abschnitt von Abbildung 6.2 wurde so gebildet durch Verschmelzen der Regeln R_1 mit R_6 (damit sinkt der Einfachheitswert auf 25, da ein vollständiger Satz so redundant wird), R_3 mit R_5 (bei resultierender Einfachheit von 17) und abschließend R_2 mit R_4 (bei einer Einfachheit von 14 für die Grammatik).

Die Phase der Regelerzeugung betrachtet alle Möglichkeiten, neue Terme zu generieren, die aus Paaren von nicht-terminalen Symbolen R_iR_j, die in der Grammatik irgendwo aufeinander folgend auftreten. Für jede Alternative wird eine neue Regel $R_k \rightarrow R_iR_j$ angelegt und in der Grammatik wird jedes Vorkommen von R_iR_j durch R_k ersetzt. Die nach dem Evaluationsmaß beste dieser Alternativen wird für die weitere Bearbeitung ausgewählt. Auf diese Weise werden solange alle möglichen Alternativen betrachtet, bis die Regelgenerierung zu keiner Verbesserung mehr führt. In diesem Fall wechselt der Algorithmus wieder zur Phase des Verschmelzens. Wird in keiner der beiden Phasen eine Verbesserung erzielt, so terminiert der Algorithmus.

Im untersten Abschnitt in Abbildung 6.2 ist die Grammatik (mit der Einfachheit 13) dargestellt, die sich durch Erzeugung der Regel $R_7 \rightarrow R_1R_2$ ergibt.

Sind unter den in der Eingabemenge gegebenen Sätzen auch negative Beispiele, so müssen die durch Verschmelzen entstandenen Grammatiken jeweils noch mit diesen abgeglichen werden.

Ein Analyse der Laufzeit dieses Algorithmus wir von Langley nicht gegeben.

Unabhängig von Langley entwickelten Stolcke und Omohundro [113] einen im Grundprinzip sehr ähnlichen Algorithmus. Ihr Algorithmus verwendet ein probabilistisches Evaluationsmaß, das neben der Einfachheit auch die Fähigkeit der Grammatik berücksichtigt, die gegebenen Sätze zu erkennen (in dem Sinne, dass keine zu starke Verallgemeinerung stattfindet, denn schließlich werden hier negative Beispiele nicht verwendet). Zudem werden bei Stolcke und Omohundro die Beispielsätze nacheinander und nicht vollständig in der initialen Grammatik verarbeitet.

6.3.4 Sequitur

Der *Sequitur*-Algorithmus wurde von Nevill-Manning und Witten [95, 97] entworfen, um eine hierarchische Repräsentation einer gegebenen Sequenz zu erhalten, die es gegebenenfalls ermöglicht, Erkenntnisse über eine der Sequenz zugrunde liegende Struktur zu gewinnen.

Sequitur verarbeitet als Eingabe nicht mehrere Sätze, sondern nur einen kontinuierlichen Datenstrom, den Eingabetext. Durch das Ersetzen von wiederholten Textteilen durch Regeln, die genau diese Teilstrings produzieren wird in Form einer kontextfreien Grammatik diese hierarchische Struktur erzeugt.

Der Sequitur-Algorithmus basiert auf einem Prinzip der „Kürze“, welches durch zwei grundlegende Eigenschaften für die erzeugten Grammatiken vollständig definiert wird:

- (E1) *Einmaligkeit von Digrammen*
  Jedes Digramm (das sind zwei direkt benachbarte Symbole) ist innerhalb der Grammatik einzigartig, also nicht mehr als einmal verwendet.
- (E2) *Nützlichkeit von Regeln*
  Jede Regel der Grammatik ist nützlich, findet also an mindestens zwei Stellen Verwendung.

Der Algorithmus selbst besteht darin, dafür Sorge zu tragen, dass diese beiden Eigenschaften für die erzeugte Grammatik immer erfüllt sind.

Der Algorithmus beginnt mit einer leeren Regel S, die am Ende der Verarbeitung die ganze Eingabe beschreibt. Initial stellt also $S \rightarrow \epsilon$ die ganze Grammatik dar. Dann verarbeitet der Algorithmus den gegebenen Text zeichenweise und jedes betrachtete Zeichen wird an die Hauptregel S angehängt. Bei Verarbeitung des Zeichens t_j entsteht also aus $S \rightarrow s_1 \cdots s_i$ die Regel $S \rightarrow s_1 \cdots s_i t_j$.

Die letzten beiden Symbole $s_i t_j$ der Regel S bilden ein neues Digramm. Ist dieses Digramm bereits an einer anderen Stelle in der Grammatik vorhanden, so ist die Eigenschaft (E1) verletzt. Um diese Eigenschaft zu korrigieren, müssen zwei Fälle unterschieden werden. Handelt es sich bei dem anderen Auftreten des Digramms um die rechte Seite einer bereits existierenden Regel R, so kann diese Regel wieder verwendet werden und das neu aufgetretene Digramm $s_i t_j$ in der Regel S wird durch einen Verweis auf die Regel (also das nicht-terminale Symbol R) ersetzt. Die Regel S wird somit zu $S \rightarrow s_1 \cdots s_{i-1} R$. Im anderen Fall ist das Digramm noch nicht als Regel bekannt und es wird eine neue Regel R in die Grammatik eingeführt, die genau dieses Digramm erzeugt, also $R \rightarrow s_i t_j$. An den beiden Stellen, an denen das Digramm bisher auftrat, wird anstelle des Digramms das nicht-terminale Symbol R als Verweis auf die neue Regel gesetzt. Das Beispiel in Abbildung 6.3 zeigt diese beiden Fälle bei der Verarbeitung des sechsten und des neunten Zeichens.

Nach dieser Korrektur der Eigenschaft (E1) kann es trotzdem sein, dass (E1) aufgrund des mehrfachen Vorhandenseins des Digramms $s_{i-1} R$ nicht erfüllt ist. Sequitur wendet daher den Korrekturmechanismus für die Eigenschaft (E1) solange iterativ an, bis dadurch keine Änderung an der Grammatik mehr vorgenommen wurde und somit (E1) erfüllt ist.

Jedes Mal, wenn ein Digramm durch eine Regel ersetzt wurde, kann es passieren, dass bei der Korrektur betroffene Regeln die Eigenschaft (E2) nicht mehr erfüllen (wie z. B. beim Verarbeiten des zehnten Zeichens in Abbildung 6.3), also nur noch einmal verwendet werden. Um die Eigenschaft (E2) wieder zu erfüllen, wird deswegen geprüft, ob eines der beiden Symbole des verarbeiteten Digramms auf eine Regel verweist, die nun nur noch einmal Verwendung findet. Ist dies der Fall, so wird diese Regel *expandiert*, d. h. die rechte Seite der Regel an der Stelle der Verwendung direkt eingesetzt und die Regel selbst wird aus der Grammatik entfernt. Im Beispiel in Abbildung 6.3 ist diese Situation bei der Verarbeitung des zehnten Zeichens zu beobachten.

Bei entsprechender Implementierung mit doppelt-verketteten Listen und einer (hinreichend großen) Hash-Tabelle erreicht der Algorithmus eine Laufzeit von $O(n)$.

Zeichen-nummer	aktueller Text	abgeleitete Grammatik			Anmerkung
1	a	S	$\rightarrow$	a	
2	ab	S	$\rightarrow$	ab	
3	abc	S	$\rightarrow$	abc	
4	abcd	S	$\rightarrow$	abcd	
5	abcdb	S	$\rightarrow$	abcdb	
6	abcdbc	S	$\rightarrow$	abcdbc	bc ist doppelt enthalten
		S	$\rightarrow$	aR_1dR_1	(E1) erzwungen
		R_1	$\rightarrow$	bc	
7	abcdbca	S	$\rightarrow$	aR_1dR_1a	
		R_1	$\rightarrow$	bc	
8	abcdbcab	S	$\rightarrow$	aR_1dR_1ab	
		R_1	$\rightarrow$	bc	
9	abcdbcabc	S	$\rightarrow$	aR_1dR_1abc	bc ist doppelt enthalten
		R_1	$\rightarrow$	bc	
		S	$\rightarrow$	aR_1dR_1aR_1	(E1) erzwungen;
		R_1	$\rightarrow$	bc	aR_1 ist doppelt enthalten
		S	$\rightarrow$	R_2dR_1R_2	(E1) erzwungen
		R_1	$\rightarrow$	bc	
		R_2	$\rightarrow$	aR_1	
10	abcdbcabcd	S	$\rightarrow$	R_2dR_1R_2d	R_2d ist doppelt enthalten
		R_1	$\rightarrow$	bc	
		R_2	$\rightarrow$	aR_1	
		S	$\rightarrow$	$R_3R_1R_3$	(E1) erzwungen; R_2 wird
		R_1	$\rightarrow$	bc	nur einmal verwendet
		R_2	$\rightarrow$	aR_1	
		R_3	$\rightarrow$	R_2d	
		S	$\rightarrow$	$R_3R_1R_3$	(E2) erzwungen
		R_1	$\rightarrow$	bc	
		R_3	$\rightarrow$	aR_1d	

Abbildung 6.3: Die Arbeitsweise von Sequitur. Die Zeichen des Textes abcdbcabcd werden nacheinander an die Hauptregel der Grammatik angehängt. Wird dadurch eine der Eigenschaften (E1) oder (E2) verletzt, so wird diese umgehend durch Anwenden des entsprechenden im Text beschriebenen Korrekturmechanismus wieder hergestellt.

7 Filteransatz mit Grammatiknutzung - Prinzip und Durchführung

Ein Kompromiss, das ist die Kunst, einen Kuchen so zu teilen, dass jeder meint, er habe das größte Stück bekommen.

(Ludwig Erhard, dt. Politiker, 1897-1977)

In Kapitel 2 wurde das Problem des Approximate String Matching vorgestellt und die Kapitel 3 und 4 präsentierten Algorithmen zu dessen Lösung. Einen besonderen Schwerpunkt bilden dabei die in Kapitel 4.1 aufgezeigten, auf einem Filterprinzip basierenden Algorithmen. Wesentlich für das durchschnittliche Laufzeitverhalten eines solchen Filteralgorithmus ist der Anteil der zweiten Phase, die darin besteht, die in der ersten Phase identifizierten Stellen möglicher Approximate Matchings zu überprüfen. Diese Überprüfung, oder auch Verifikation, für die üblicherweise einer der grundsätzlichen Lösungsalgorithmen (siehe Kapitel 3.1) Verwendung findet, sollte die Gesamtlaufzeit nach Möglichkeit nicht dominieren.

Mit verschiedenen Methoden wurde bei Filteralgorithmen versucht, das Verhältnis der beiden Phasen zu Gunsten der Suchphase zu verschieben (siehe Kapitel 4.2). Die Idee der Verwendung einer Grammatik zu genau diesem Zweck wurde bereits in Kapitel 6 angesprochen. Kurz gesagt soll eine mittels des Sequitur-Algorithmus aus dem Text abgeleitete Grammatik dazu verwendet werden, bei der Suche nach Approximate Matchings jeden unterschiedlichen Textabschnitt nach Möglichkeit nur einmal zu betrachten. Dieses Kapitel befasst sich ausführlich mit dieser Idee und der konkreten Umsetzung (siehe auch [100]), während das nächste Kapitel (Kapitel 8) sich mit der Analyse beschäftigt.

Zuerst werden in Kapitel 7.1 die Möglichkeiten dieses Ansatzes genauer diskutiert, bevor in Kapitel 7.2 das konkrete Prinzip und die tatsächliche Ausführung des Algorithmus erläutert werden. Nach einem Abschnitt zur Korrektheit des Algorithmus (Kapitel 7.3), folgt eine Erläuterung von verschiedenen Algorithmusvariationen (Kapitel 7.4).

7.1 Kriterien für den Ansatz

Wie in Kapitel 6.1 bereits dargelegt, scheint das Filterprinzip der ideale Ansatz zur Verfolgung der grundsätzlichen Idee, eine Grammatik des Textes zu nutzen, um von Wiederholungen des Textes zu profitieren und den Anteil der Verifikationsphase an der gesamten Suche so zu reduzieren. Jedoch ist dafür nicht jede Klasse von Filteransätzen (vgl. Kapitel 4.1.1) gleichermaßen gut geeignet.

Bei einer Zerlegung des Problems in kleinere Teilinstanzen, werden Probleminstanzen betrachtet, die klein genug sind, um sie mit einem speziellen, auf diesen Fall sehr gut angepassten Algorithmus zu lösen. Da für diese Arbeit kein solcher angepasster Algorithmus zugrunde liegt und auch der Fokus nicht auf diesen speziellen Suchalgorithmen liegt, bildet die Zerlegung des Problems in Teilprobleme der exakten Suche den weiteren Ausgangspunkt. Die exakte Suche kann mit jedem beliebigen exakten Suchalgorithmus behandelt werden und bietet so den Rückgriff auf bewährte Algorithmen.

Weiterhin führt der Ansatz des Betrachtens einer Grammatik des Textes zwangsläufig zu der Sichtweise, die von dem Vorhandensein der Fehler im Pattern ausgeht. Bei dieser Sichtweise können Stücke des Patterns im Text gesucht werden (vgl. Kapitel 4.1.1), was letztlich genau die Möglichkeit der Nutzung der Grammatik bei der Suche eröffnet.

In der Klasse der Filteralgorithmen mit dem Prinzip der Zerlegung des Problems in das Problem der exakten Suche bei gleichzeitiger Annahme der Existenz aller Fehler im Pattern (siehe Kapitel 4.1.2) weist der Ansatz von Wu und Manber (Kapitel 4.1.2.3) die größte Flexibilität auf, da er prinzipiell jede beliebige disjunkte Aufteilung des Patterns in Subpattern behandeln kann. Aus diesem Grund baut die hier verfolgte Idee auf dem Ansatz von Wu und Manber auf.

Vor der Diskussion von Details der Algorithmuskonstruktion ist es sinnvoll, sich die bisher ergebenen Zielvorstellungen zu vergegenwärtigen.

- Der Algorithmus soll auf dem Filterprinzip basieren und dabei dem grundsätzlichen Ansatz von Wu und Manber folgen. In der Suchphase soll grundsätzlich jeder beliebige Algorithmus zur exakten Suche verwendet werden können, ebenso wie in der Verifikationsphase keine Abhängigkeit von dem verwendeten Algorithmus bestehen soll. Im Idealfall können also diese beiden grundsätzlichen Algorithmen beliebig ausgetauscht werden und bleiben zudem unabhängig einsetzbar. Das Zusammenspiel von Such- und Verifikationsphase muss dazu also auf einer höheren Ebene gesteuert werden, die auch die Grammatik in Betracht zieht.

- Wie bereits in Kapitel 6.1 angesprochen, sollen die Regeln der Grammatik so in den Such- und Verifikationsprozess eingebunden werden, dass nach Möglichkeit jede Regel sowohl im Rahmen der Suchphase als auch während der Verifikation nur maximal einmal betrachtet wird. Prinzipiell besteht so gegenüber der Betrachtung der vollständigen Textlänge in beiden Phasen die Möglichkeit, gewisse wiederholte Textbereiche von der nochmaligen näheren Betrachtung auszuschließen.

Ansatzmöglichkeiten

Insbesondere vor dem Hintergrund des Gedankens der einmaligen Betrachtung einer Regel ist es die nächstliegende Idee, zuerst die Regeln separat zu betrachten und dann aus den dabei gewonnenen Erkenntnissen ein Gesamtergebnis für den Eingabetext zu generieren. Im Folgenden werden kurz die Gedanken erläutert, die dazu geführt haben, innerhalb dieser Arbeit diesen Ansatz zugunsten eines anderen Ansatzes zu verwerfen. Die Regeln der Grammatik unterliegen einer hierarchischen Einbettung, d. h. innerhalb einer Regel können mehrere andere Regeln liegen und jede Regel kann zudem selbst in verschiedenen anderen Regeln enthalten sein. Eine Betrachtung dieser Hierarchie von unten nach oben entspricht der Vorgehensweise, immer nur Regeln zu untersuchen, die keine eingebetteten noch nicht untersuchten Regeln enthalten. Den Ausgangspunkt für dieses Vorgehen bilden die Regeln, die selbst keine eingebetteten Regeln enthalten, also auf unterster Ebene liegen.

Die exakte Suche nach Teilpattern in dieser Regelstruktur gestaltet sich aufgrund der hierarchischen Einbettung an den Übergangsbereichen als recht problematisch. Um zu vermeiden, eine Regel immer dort, wo sie eingebettet vorliegt, erneut durchsuchen zu müssen, wäre so etwas wie ein Status für jedes Teilpattern nötig, der die Anzahl der exakt übereinstimmenden End- bzw. Anfangszeichen am jeweiligen Regelanfang bzw. -ende speichert. Es ist weiterhin zu berücksichtigen, dass gerade kürzere Regeln vollständig innerhalb eines Teilpatterns enthalten sein können, und dass einzelne Textteile gegebenenfalls auch überhaupt nicht in einer Regel enthalten sind. Im Prinzip dieselbe Problematik tritt auch bei einer regelweise durchgeführten Betrachtung während der Verifikationsphase auf. Um die Berechnungen an diesen problematischen Übergängen richtig kontrollieren zu können, ist neben der „Regelsicht“ auch eine übergeordnete Sicht auf der Textebene notwendig.

Genau an dieser Stelle ergibt sich ein anderer Ansatz, der auf der übergeordneten Sicht auf der Textebene basiert. Es ist möglich, in den beiden Phasen jeweils immer die Textebene zu betrachten und dann auf eine Regel der Grammatik zurückzugreifen, wenn diese Regel auf der Textebene soweit behandelt wurde, dass eine gesicherte Auskunft für jedes weitere Auftreten der Regel daraus abgeleitet werden kann.

Konkret wird also die exakte Suche in der ersten Phase nicht auf den Regeln, sondern auf dem Text in Originalform durchgeführt. Immer wenn der Suchfortschritt im Text soweit gelungen ist, dass eine Regel vollständig durchsucht wurde, können die Suchergebnisse für jedes weitere Auftreten der Regel kopiert werden (aber auch hier sind die Randbereiche der Regeln zu beachten). Dann kann bei der fortgesetzten Suche im Text jedes weitere Auftreten dieser Regel übersprungen werden.

Prinzipiell ist die Verifikation in der zweiten Phase ebenso zu behandeln. Es werden nicht die einzelnen Regeln überprüft, sondern genau die zur Überprüfung identifizierten Textbereiche. Wurde dann eine Regel an der Stelle des ersten Auftretens vollständig bewertet (d. h. jeder Buchstabe der Regel wurde entweder überprüft oder als nicht zu überprüfen klassifiziert), so können die Ergebnisse in diesem Bereich für jedes andere

Auftreten der Regel kopiert werden (unter Berücksichtung der Randbereiche). Weiterhin bedürfen die Stellen, an denen die Regel dann wiederholt auftritt, keiner erneuten Verifikation.

Dieser Ansatz vereinfacht gegenüber dem zuerst genannten Ansatz die Behandlung der Regelgrenzen deutlich. Für jede betrachtete Regel ist es nur notwendig, für die Suche und für die Verifikation jeweils Gültigkeitsbereiche zu definieren, die dann von der folgenden Suche und Verifikation immer ausgeschlossen werden.

Im folgenden Kapitel werden dieser Ansatz und dessen Umsetzung detailliert beschrieben.

7.2 Algorithmus zum Filteransatz mit Grammatiknutzung

Dieses Kapitel stellt den in Kapitel 7.1 bereits angerissenen Ansatz der Verwendung einer Grammatik beim Approximate String Matching ausführlich in seiner Umsetzung dar.

Zuerst wird das allgemeine Prinzip des sich ergebenden Algorithmus dargestellt. Die dann folgenden fünf Unterkapitel beschäftigen sich mit der Umsetzung in ein Programm und zeigen neben dem genauen Vorgehen auch sich ergebende Probleme auf.

7.2.1 Das Algorithmusprinzip

Durch den Grundgedanken des Algorithmus, der Verwendung einer Grammatik des Textes bei der Lösung des Approximate String Matching Problems, sind die beiden wichtigen Algorithmusbestandteile bereits definiert. Erst muss eine Grammatik aus dem Text abgeleitet werden, bevor diese bei der Suche nach den Approximate Matchings genutzt werden kann.

Vorbereitung

Der Aufbau der Grammatik kann als eine erste Vorberechnungsstufe auf dem Text verstanden werden, was den Algorithmus letztendlich auch zu einem Offline-Algorithmus macht. Mittels des Sequitur-Algorithmus (siehe Kapitel 6.3.4) wird eine Grammatik aus dem Text T hergeleitet. Durch die Grammatik wird der Text zugleich in einer anderen, nicht linearen Form repräsentiert. Der Text ist so definiert durch eine Startregel, die auf eine Sequenz von Buchstaben und anderen Regeln verweist. Jede andere Regel repräsentiert (nach der Sequitur-Bedingung (E2)) einen Textbereich, der mindestens zweimal in T enthalten ist. Für jede Regel wird in der Vorberechnungsphase zusätzlich die Länge des durch sie repräsentierten Textbereichs gesichert. Ebenso wird für jede Regel eine Liste berechnet, die die jeweiligen Textpositionen angibt, an denen diese Regel Verwendung findet. Der Schritt der Vorberechnung betrifft also im Wesentlichen die Grammatik und damit den Text. Doch auch das Pattern muss initial bearbeitet werden. Wie bereits in den Kapiteln 6.1 und 7.1 diskutiert, wird hier auf der Filtermethode nach Wu und Manber aufgebaut. Da dieser Filter $k + 1$ Teilstücke des Patterns verwendet, muss also das Pattern entsprechend aufgeteilt werden. In dieser Arbeit wird eine gleichmäßige Aufteilung in Subpattern vorgenommen, wobei prinzipiell auch andere, nicht gleichmäßige Aufteilungen möglich sind (z. B. mit Berücksichtigung der Auftrittswahrscheinlichkeiten für die einzelnen Buchstaben). Weitere Vorberechnungen auf dem Pattern P finden nur insofern statt, als dies für den verwendeten Algorithmus zur exakten Suche notwendig ist (d. h. in der Regel werden Verschiebungstabellen vorberechnet).

Nach der Vorberechnung folgt als nächster großer Schritt das Suchen der Approximate Matchings selbst. Diese Suche basiert grundsätzlich auf den beiden Phasen der exakten Suche von Teilpattern und der Überprüfung bzw. Verifikation von Textbereichen, die während der exakten Suche als mögliche Kandidaten für Approximate Matchings erkannt wurden. Neben diesen beiden Phasen ist hier jedoch noch eine dritte Phase notwendig, die die Informationen auswertet, die in den ersten beiden Phasen gewonnen wurden.

Für die exakte Suche wird hier eine wie in [14] und [87] beschriebene Erweiterung des Algorithmus von Sunday [115, 116] genutzt, die die gleichzeitige Suche mehrerer Pattern erlaubt. In der Verifikationsphase wird hier der Algorithmus von Chang und Lampe [20] verwendet. Anstelle dieser hier genutzten Algorithmen könnten jedoch auch beliebige andere Algorithmen gebraucht werden, die dieselbe Funktion erfüllen (siehe Kapitel 7.1).

Ablauf

Insgesamt stellt die Suche nach Approximate Matchings unter Verwendung der erstellten Grammatik nichts anderes dar als eine Schleife, die die drei oben angesprochenen Phasen beständig wiederholt. Suchphase, Verifikationsphase und die Phase der Informationsauswertung werden also immer wieder nacheinander durchlaufen.

Der Algorithmus arbeitet auf einer Liste von Suchintervallen, die Bereiche des Textes darstellen, auf denen die exakte Suche nach den Subpattern durchgeführt werden muss. Initial hat diese Liste mit $[1, n]$ nur einen Eintrag, also nur einen Bereich, der den ganzen Text abdeckt.

Zur Bearbeitung wird immer das erste und damit aktuelle Intervall dieser Liste an Suchintervallen ausgewählt. Ist diese Liste leer, so ist keine Suche mehr durchzuführen, und die Ausführung der Wiederholung der drei Phasen kann gestoppt werden. Andernfalls wird der durch das Intervall gegebene Textbereich auf das Vorkommen eines der Subpattern hin untersucht. Diese Suche endet genau dann, wenn entweder eines der Subpattern gefunden werden konnte, oder aber wenn sich keines der Subpattern im gegebenen Intervall im Text befindet. Im letzteren Fall wird das aktuelle Intervall aus der Liste gelöscht, und solange noch ein Intervall in der Liste enthalten ist, wird die Bearbeitung mit dem dann ersten Intervall fortgesetzt.

Wurde ein Subpattern im gegebenen Suchintervall gefunden, so wird ein Verifikationsbereich bestimmt, der diese Fundstelle beinhaltet. Dabei wird die Breite dieses Verifikationsbereichs so gewählt, dass garantiert jedes mögliche Approximate Matching in diesem Bereich enthalten ist, das das gefundene Subpattern enthält. Mittels des Verifikationsalgorithmus werden dann alle Approximate Matchings (in Form von Endpositionen) in diesem Bereich bestimmt. Die Position jedes gefundenen Approximate Matching wird in einer Liste gespeichert, in der alle diese „Treffer“ gesammelt werden. Die vollständige Information der Verifikation, also das exakt gefundene Subpattern, der Verifikationsbereich und die Verifikationsergebnisse, werden zu einem Datum, ab jetzt *Verify* genannt, zusammengefasst. Zum Zwecke der Auswertung in der nächsten Phase wird das Verify in einer Liste abgespeichert.

Die Phase der Informationsauswertung beginnt damit, das erste (und somit auch das aktuelle) Suchintervall der Liste an Suchintervallen zu aktualisieren. Dazu wird der Intervallanfang auf die Position genau rechts der Fundstelle des Subpatterns gesetzt; denn genau ab dort wurde die Suche nicht mehr fortgeführt. Um nun einen direkten Nutzen aus den gewonnenen Informationen ziehen zu können, werden nacheinander die folgenden drei Berechnungen durchgeführt:

1. Es wird geprüft, ob eine Regel (an der Position des ersten Auftretens dieser Regel im Text) einen der Verifikationsbereiche vollständig abdeckt. Ist dies der Fall, so werden alle Positionen gefundener Approximate Matchings (im abgespeicherten Verify enthalten) für jedes weitere Auftreten der Regel vervielfältigt (siehe Abbildung 7.1) und in die Liste, die alle Approximate Matchings enthält, gespeichert.

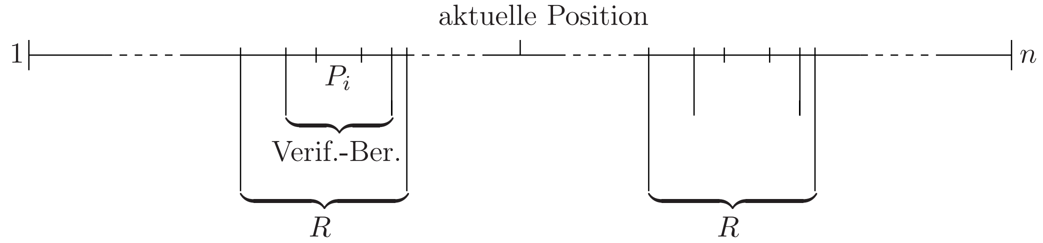


Abbildung 7.1: Vervielfältigung der Verifikationsergebnisse. Das zuvor exakte gefundene Subpattern P_i führte zu der Überprüfung eines Bereichs. Dieser Verifikationsbereich wird vollständig von der Regel R abgedeckt. Jedes innerhalb dieses Verifikationsbereichs vorhandene Approximate Matching ist bei jedem anderen Auftreten der Regel R ebenso vorhanden.

2. Wurde ein Verifikationsbereich nicht vollständig von einer Regel abgedeckt, so wird überprüft, ob eine Regel zumindest das Subpattern P_i abdeckt, das zur Verifikation geführt hat. Ist dies der Fall, so wird der Verifikationsbereich für jedes andere Auftreten der Regel entsprechend dupliziert (siehe Abbildung 7.2). Die so entstandenen neuen Verifikationsbereiche werden in einer Liste gespeichert, die Bereiche enthält, die noch überprüft werden müssen (wohlgemerkt, es ist bekannt, dass das Subpattern P_i an einer bestimmten Stelle in jedem dieser Verifikationsbereiche enthalten ist).

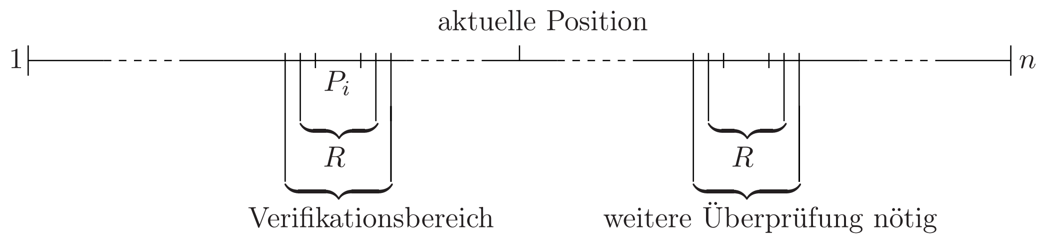


Abbildung 7.2: Vervielfältigung der Verifikationsbereiche. Die Regel R deckt nur das Subpattern P_i vollständig ab, nicht jedoch den vollständigen Verifikationsbereich. Für jedes weitere Auftreten der Regel R, wird ein Verifikationsbereich zur späteren Überprüfung vorgemerkt.

3. Jede Regel, deren erstes Auftreten soweit in dem bereits durchsuchten Textbereich liegt, dass sie nicht mehr in noch folgende Verifikationen hineinreichen kann, bedarf keiner weiteren Betrachtung und kann deswegen von den folgenden Betrachtungen ausgenommen werden. Dazu wird für jedes weitere Auftreten einer solchen Regel ein Bereich aus der Liste der Suchintervalle herausgenommen, der definitiv nicht mehr nach den Subpattern durchsucht werden muss (siehe Abbildung 7.3).

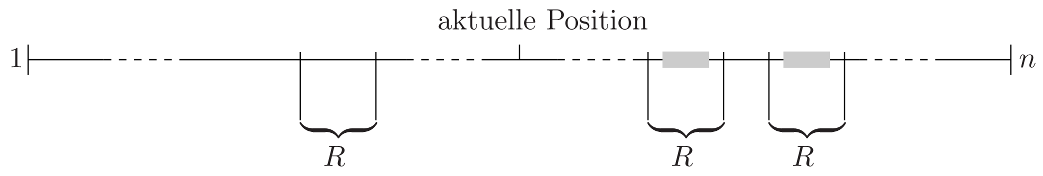


Abbildung 7.3: Vermeiden der Suche in bereits behandelten Regeln. Für jedes weitere Auftreten der Regel R wird ein Textbereich (grau markiert) von der weiteren Suche ausgeschlossen.

Mit der so modifizierten Liste an Suchintervallen wird, wieder bei der Suchphase beginnend, weiter fortgefahren.

Wurde die Schleife der beständigen Abfolge von Suche, Verifikation und Auswertung der Information verlassen, wird abschließend noch die Liste der Bereiche überprüft, die zuvor zur späteren Verifikation gesichert wurden. Die hier erzielten Ergebnisse vervollständigen die Liste der Positionen von Approximate Matchings im Text.

7.2.2 Vorberechnungen

Wie aus Kapitel 7.2.1 ersichtlich ist die Durchführung gewisser Vorberechnungen für den Algorithmus notwendig. Im Folgenden werden die auf dem Text und dem Pattern durchgeführten Vorberechnungen näher erläutert.

7.2.2.1 Vorberechnungen auf dem Text

Um überhaupt auf der Basis einer Grammatik arbeiten zu können, muss diese zuerst erstellt werden. Mittels des Sequitur-Algorithmus (siehe Kapitel 6.3.4) wird also eine Grammatik des Textes erstellt. Zudem werden dann in einem Durchlauf durch die Grammatik verschiedene Merkmale und Eigenschaften der Grammatik berechnet, die für den Algorithmus wichtig sind:

- Die Regellänge. Für jede Regel R der Grammatik wird die (expandierte) Länge $R.length$ bestimmt, d. h. die Anzahl der Buchstaben des Textes, die diese Regel abdeckt. Die Regellänge ist notwendig, um schnell von der Regel abgedeckte Bereiche identifizieren zu können.

- Die Regelpositionen. Für jede Regel R wird eine sortierte Liste $R.PositionList$ erzeugt, die alle Textpositionen enthält, an denen sie auftritt. Diese Positionen werden notwendig, um einfach Informationen, die im Bereich der Regel liegen, auf alle anderen Vorkommen der Regel zu vervielfältigen.

- Eine Sammlung aller Regeln. Ein Array *SR* („Sortierte Regeln") wird konstruiert, in welchem die Regeln in der Reihenfolge ihres erstmaligen Auftretens im Text abgelegt sind. Durch die Position im Array bekommen die Regeln auch implizit Nummern zugewiesen (die Nummerierung beginnt bei 0) und ein direkter Zugriff auf bestimmte Regeln gestaltet sich einfach.

- Die Regelnummer der hierarchischen Inklusion. Mit der Grammatik ist eine hierarchische Anordnung der Regeln entstanden. Für jede Regel R wird in $R.inclNumber$ die Nummer der Regel abgelegt, die sich an der Stelle des ersten Auftretens von R im Text genau eine Ebene über R befindet, also die Nummer der Regel, die R vollständig beinhaltet. Ist R beim ersten Auftreten in keiner Regel enthalten (also auf der obersten Ebene), so wird dies durch den Wert -1 angezeigt (was keine gültige Nummer einer Regel ist). Diese Regelnummer der Inklusion wird verwendet, um die Gesamtverarbeitung zu beschleunigen: sollen in einem Schritt beide Regeln R und $R.inclNummer$ betrachtet werden, so wird hier nur auf die umfassendere Regel $R.inclNummer$ zurückgegriffen.

- Die ersten Regelenden im Text. Ein Array *TR* („Text-Regeln") wird erzeugt, das zu jeder Textposition die Nummer der letzten Regel enthält, die bis zu dieser Textposition vollständig im Text enthalten ist. Dabei ist für jede Regel wieder nur das erste

Auftreten im Text von Interesse. Mittels *TR* kann einfach für einen Textbereich festgestellt werden, welche Regeln in diesem Bereich von Interesse sind.

- Die maximale Regelnummer im Text. In einem Array *TL* („Top Level") wird für jede Textposition die höchste Nummer aller Regeln notiert, die diese Stelle abdecken (was der Abdeckung durch die oberste hierarchische Ebene entspricht). Wird eine Stelle nicht durch eine Regel abgedeckt, so wird die Nummer der Stelle zuvor übertragen. *TL* dient der vereinfachten Erkennung, ob eine Regel an einer Textstelle überhaupt von Interesse sein kann.

7.2.2.2 Vorberechnungen auf dem Pattern

Die einzige auf dem Pattern P durchgeführte Vorberechnung ist die Aufteilung des Patterns in die $k+1$ disjunkten Teilpattern, die in der Suchphase notwendig sind. Hier wird eine gleichmäßige Aufteilung $P = P_1 \dots P_{k+1}$ vorgenommen, so dass mit $p_l = m$ div $(k+1)$ und $q = m$ mod $(k+1)$ gilt:

$$\begin{array}{lcll} |P_i| & = & p_l + 1 & \text{für} \quad i = 1, \dots, q \\ |P_i| & = & p_l & \text{für} \quad i = q+1, \dots, k+1 \end{array}$$

Dabei stellt div die ganzzahlige Division und mod die Modulo-Operation dar.

Wenn sich $m = |P|$ also nicht restlos ganzzahlig durch $k+1$ teilen lässt, sind bei dieser Aufteilung die ersten q Teilpattern um 1 länger.

Die Länge des längsten Teilpattern wird in P_{max} gespeichert, wobei hier P_{max} aufgrund der gleichmäßigen Aufteilung entweder gleich p_l (im Fall, dass m restlos durch $k+1$ teilbar ist) oder $p_l + 1$ entspricht.

Im Allgemeinen ist die vorgenommene Aufteilung jedoch egal, solange in P_{max} die Länge des längsten Teilpatterns vermerkt wird.

7.2.3 Initialisierung

Die Initialisierung des Algorithmus entspricht der Erzeugung eines wohldefinierten Ausgangszustands der weiter in den folgenden Algorithmusphasen benötigten Datenstrukturen. Folgende Auflistung zeigt die getätigten Initialisierungen:

- $SL = \{[1, n]\}$
  Die Liste der Suchbereiche *SL* wird mit dem Intervall initialisiert, das den Text vollständig abdeckt.

- $HitList = \emptyset$
  Die Liste $HitList$, in der alle gefundenen Approximate Matchings notiert werden, ist zu Beginn leer.

- $VL = \emptyset$
  Die Liste VL, in der im Laufe des Algorithmus die einzelnen Verifys gespeichert werden, ist zu Beginn leer. Konkret bestehen die später hier gespeicherten Verify-Datensätze aus jeweils folgenden Informationen:
  - Die Nummer des Teilpatterns, welches exakt im Text gefunden wurde und so die hier gespeicherte Verifikation angestoßen hat.
  - Die Position im Text, an der das Teilpattern exakt gefunden wurde.
  - Die Anfangs- und die Endposition des Verifikationsbereichs.
  - Die Anzahl an Positionen, an denen ein Approximate Matching in diesem Verifikationsbereich gefunden werden konnte.
  - Ein Zeiger auf die erste dieser Positionen in der $HitList$, wo sie gespeichert vorliegen.
- $CL = \emptyset$
  Die Liste CL („Check List") der am Ende noch separat zu prüfenden Textbereiche ist am Anfang leer.
- $R_l = -1$
  Die Nummer der zuletzt bearbeiteten Regel ist immer in R_l notiert. Da die Suche entlang des Textes erfolgt, können Regeln mit kleineren Nummern immer als schon bearbeitet betrachtet werden (beachte dazu, dass die Regelnummerierung auf Grundlage der Arrays SR erfolgt).

7.2.4 Suche und Verifikation

Nach der Vorverarbeitung und der Initialisierung kann der Algorithmus mit der Suche nach den Approximate Matchings fortfahren. Dies geschieht durch die Wiederholung der drei Bearbeitungsphasen der Suche, Verifikation und Informationsauswertung. Gesteuert wird die Anzahl dieser Wiederholungen durch die Elemente in der Liste mit den Suchintervallen SL.

Der jetzt zuerst dargestellte Teilalgorithmus *SearchAndVerify* zeigt diese Steuerung und den Ablauf dieser wiederholten Schleife. Der Aufruf der Routine zur exakten Suche in Zeile 3 stellt die Suchphase dar, während in Zeile 7 die Verifikationsphase zum Tragen kommt. Die Phase der Informationsauswertung wird im Teilalgorithmus *ProcessRules* durchgeführt, der in Zeile 10 aufgerufen wird.

Es sei noch bemerkt, dass in Zeile 9 $hPos$ einen Zeiger auf die erste der in der vorangehenden Zeile in die $HitList$ eingefügten Positionen darstellt.

Algorithmus *SearchAndVerify*

while $SL \neq \emptyset$ **do**
  $[t_b, t_e] = SL.first$
  Suche linear nach $P_1, \ldots, P_{k+1}$ in $[t_b, t_e]$
  if P_i gefunden an Stelle t_x **then**
    $pos_b = t_x - (m + k - 1)$
    $pos_e = t_x + (m + k - 1)$
    Verifikation von $[pos_b, pos_e]$
    Füge die h gefundenen Positionen in die $HitList$ ein
    Füge das Verify $(t_x, i, pos_b, pos_e, h, hPos)$ an VL an
    ProcessRules
  else
    Lösche $[t_b, t_e]$ aus SL

7.2.5 Phase der Informationsauswertung

In der Phase der Informationsauswertung werden die Regeln der Grammatik verwendet, um gefundene Approximate Matchings auf andere Textstellen zu vervielfältigen, und um die Suche in manchen Textbereichen vollends zu vermeiden.

Der Teilalgorithmus *ProcessRules* betrachtet die Regeln, die in dem zuletzt untersuchten Textintervall endeten (es ist hier immer nur das erste Auftreten einer Regel von Interesse) und entscheidet, wie damit weiter zu verfahren ist. Eine weitere, für die Regeln überprüfte Bedingung betrachtet die Nützlichkeit der Regel (siehe Zeile 4 im Teilalgorithmus *ProcessRules*). Damit eine Regel überhaupt zu einer Einsparung führen kann, muss sie hinreichend lang sein, so dass die später aufgrund dieser Regel von der weiteren Suche ausgeschlossenen Bereiche überhaupt existieren.

Algorithmus *ProcessRules*

1. Ersetze $[t_b, t_e]$ durch $[t_x + 1, t_e]$ in SL
2. **for** $i = R_l + 1$ **to** $TR[t_x]$ (∗ nur Regeln, die bis t_x zumindest einmal auftraten ∗)
3. $R = SR[i]$ (∗ Regel holen ∗)
4.   **if** $(R.length \geq 2 * P_{max} - 1)$ **and** $(R.inclNumber > TR[t_x])$ **then**
5. $CheckVL(R, i)$
6. $UpdateSL(R)$
7. $R_l = TR[t_x]$ (∗ keine Regel muss zweifach betrachtet werden ∗)

In Zeile 5 wird der Teilalgorithmus *CheckVL* aufgerufen. Dort wird die gegebene Regel daraufhin überprüft, ob sie eines der in VL gegebenen Verifikationsintervalle vollständig oder zumindest das darin enthaltene Teilpattern abdeckt. Liegt das erste Auftreten der betrachteten Regel vor einem Verifikationsintervall, so kann diese Regel aufgrund der Anordnung der Verifys auch alle weiteren Verifikationsintervalle nicht abdecken (Zeile 6 in Teilalgorithmus *CheckVL*). Deckt die Regel jedoch einen überprüften Bereich ab, werden entsprechend der Art der Abdeckung für jedes Vorkommen der Regel die gefunden Approximate Matchings vervielfältigt oder noch zu prüfende Intervalle generiert (vgl. Kapitel 7.2.1).

Algorithmus *CheckVL*
Eingabe: Regel R, Nummer i von R

for $(t_x, j, pos_b, pos_e, h, hPos) \in VL$
    if $TL[pos_e] < i$ **then** (∗ ist dieses Verify weiter von Bedeutung? ∗)
        Lösche dieses Verify aus *VL*
    else
        if $R.PositionList.first + R.length - 1 < t_x$ **then**
            return (∗ keines der Verifys wird von R abgedeckt ∗)
        else
            if R bedeckt $[pos_b, pos_e]$ vollständig **then**
                Vervielfältige die Approximate Matchings dieses Verifikationsintervalls auf jede Position in *R.PositionList*
            elseif R bedeckt das Subpattern P_j **then**
                Vervielfältige dieses Verifikationsintervall auf jede in *R.PositionList* enthaltene Position und füge diese Intervalle in *CL* ein

In Teilalgorithmus *ProcessRules* findet in Zeile 6 nach obigem Abgleich der Regel R mit der Liste *VL* der Verifikationsintervalle ein Aufruf des Teilalgoritmus *UpdateSL* statt. Dort wird anhand der gegeben Regel die Liste der noch zu betrachtenden Suchintervalle *SL* dadurch aktualisiert, dass die Bereiche herausgenommen werden, die durch Vervielfältigung dieser Regel nicht weiter berücksichtigt werden müssen. Der jeweils ausgeschlossene Bereich ist dabei so gewählt, dass jedes Teilpattern (das größte Teilpattern hat die Größe P_{max}) in jedem Fall auch an den Bereichsgrenzen noch gefunden werden kann.

Algorithmus *UpdateSL*
Eingabe: Regel R

for $t_r \in R.PositionList$
    Schließe $[t_r + P_{max} - 1, t_r + R.length - P_{max}]$ aus *SL* aus

Damit überhaupt durch die Betrachtung der Regel gewisse Suchbereiche ausgeschlossen werden können, sollte die Regel so lang sein, dass das Intervall $[t_r + P_{max} - 1, t_r + R.length - P_{max}]$ noch Symbole enthält. D. h. es muss gelten

$$(t_r + R.length - P_{max}) - (t_r + P_{max} - 1) \geq 0\,,$$

was äquivalent ist mit

$$R.length \geq 2P_{max} - 1\,.$$

Genau diese Bedingung wird zuvor in Teilalgorithmus *ProcessRules* in Zeile 4 abgefragt.

7.2.6 Finale Prüfung

Wurde die Schleife abgeschlossen, die die einzelnen Phasen beständig wiederholt (siehe Algorithmus *SearchAndVerify*), so bleibt noch die Prüfung der von Algorithmus *CheckVL* in der Liste *CL* gesammelten Intervalle. Mit dem Verifikationsalgorithmus werden diese Bereiche verifiziert und alle so gefundenen Positionen von Approximate Matchings werden in die *HitList* eingefügt.

7.3 Korrektheit

Im Prinzip ergibt sich die Korrektheit des Algorithmus aus der Konstruktion desselben, jedoch sollen hier noch einmal kurz die wichtigen Punkte dazu zusammengefasst werden. Die generelle Basis des Algorithmus wird durch den Filteransatz nach Wu und Manber gebildet (siehe Kapitel 4.1.2.3). Damit auch bei der Beeinflussung von Such- und Verifikationsphase durch die Regeln der Grammatik weiterhin alle Approximate Matchings im Text gefunden werden, müssen folgende Punkte sichergestellt sein:

- Jede Stelle im Text, an der eines der Teilpattern exakt vorliegt, muss gefunden werden.
- Eine hinreichende Nachbarschaft um jede solche gefundene Stelle muss auf das Vorhandensein von Approximate Matchings geprüft werden.

Exakt im Text vorkommende Teilpattern werden mit dem Algorithmus dann gefunden, wenn sie in einem der betrachteten Suchbereiche enthalten sind. Die so gefundenen Stellen werden auch einer Überprüfung unterzogen. Da der insgesamt durchsuchte Bereich nur durch die Betrachtung von Regeln verkleinert wird, ist also diese Verkleinerung von Interesse.

Für jede der betrachteten Regeln wird für jedes Auftreten der Regel ein gewisser Bereich aus der Liste der Suchintervalle entfernt. Dieser Bereich ist genauso gewählt (vgl. Kapitel 7.2.5, Teilalgorithmus *UpdateSL*), dass jedes der gesuchten Teilpattern, das nicht vollständig innerhalb der Regel liegt, nicht mit von der weiteren Suche ausgeschlossen wird. Damit bleibt in Bezug auf die exakte Suche nur noch zu gewährleisten, dass innerhalb der ausgeschlossenen Bereiche auch jedes Teilpattern gefunden und ein entsprechender Bereich der Verifikation unterzogen wird.

Eine Regel wird nur dann genauer betrachtet (und kann damit zu Ausschlüssen von Bereichen führen), wenn sie beim ersten Auftreten (und damit natürlich bei jedem Auftreten) bereits vollständig nach den Teilpattern durchsucht wurde. Die dort gefundenen Teilpattern führten jeweils zu einer Verifikation und die so gewonnenen Daten (inkl. des die Verifikation auslösenden Teilpatterns) wurden in Form eines Verify-Datums gesichert. In jedem weiteren Auftreten der Regel (und damit auch in jedem dann ausgeschlossenen Bereich) wurden also bereits alle exakt vorkommenden Teilpattern identifiziert. Um auch die Verifikation jeder dieser Stellen sicherzustellen, wird für jeden beim ersten Auftreten der Regel verifizierten Bereich überprüft, ob der ganze Bereich innerhalb der Regel liegt. Ist dies der Fall, so ist die Verifikation auch für jedes weitere Auftreten der Regel gültig. Andernfalls, wenn nur das gefundene Teilpattern, nicht aber der ganze Verifikationsbereich innerhalb der Regel liegt, so bleibt nichts anderes übrig, als bei jedem weiteren Auftreten der Regel einen entsprechenden Bereich erneut zu verifizieren. Dies erreicht der Algorithmus durch Sichern dieser Bereiche und eine entsprechende finale Prüfung (siehe Kapitel 7.2.6).

Der Algorithmus überprüft also alle relevanten Bereiche und liefert (durch Sicherung von positiven Verifikationsergebnissen) alle Approximate Matchings im Text.

7.4 Varianten

Es gibt verschiedene Möglichkeiten ein wenig steuernd auf den Algorithmus Einfluss zu nehmen. In diesem Abschnitt werden die wichtigsten dieser Möglichkeiten näher erläutert.

7.4.1 Minimale Länge der Regeln

In Zeile 2 des Teilalgorithmus *UpdateSL* wird das Intervall bestimmt, welches für jedes Auftreten einer betrachteten Regel aus der Liste der noch zu durchsuchenden Bereiche herausgenommen wird. Damit dieses Intervall überhaupt existiert, muss – wie zuvor in Kapitel 7.2.5 gezeigt – für die Regellänge *R.length* die Bedingung

$$R.length \geq 2P_{max} - 1$$

gelten. Diese Bedingung wird in Teilalgorithmus *ProcessRules* in Zeile 4 vor der näheren Betrachtung einer jeden Regel überprüft.

Je höher der Fehlerlevel α ist, desto kürzer werden die gesuchten Teilpattern, was wiederum zur Betrachtung sehr kurzer Regeln führt. Prinzipiell bedeutet dies, dass mehr Regeln betrachtet werden und dass so auch mehr Informationen vervielfältigt werden können. Das Betrachten einer Regel hat jedoch immer gewisse Kosten für den allgemeinen Verwaltungsaufwand zur Folge, und auch das Herausnehmen nur einer einzigen Stelle aus dem zu durchsuchenden Bereich ist nicht zwingend von Vorteil, wenn beispielsweise nur deswegen die Phase der exakten Suche dort unterbrochen wird.

Um situationsabhängig an dieser Stelle Einfluss nehmen zu können, wurde über den Parameter `MINIMAL_RULELENGTH` die Möglichkeit geschaffen, eine feste minimale Länge der betrachteten Regeln festzulegen. Dieser Parameter wird zusätzlich zu obiger Bedingung in Zeile 4 des Teilalgorithmus *ProcessRules* abgefragt.

7.4.2 Bereichsabgleich vor finaler Prüfung

Zum Abschluss des Algorithmus werden noch alle in der Liste *CL* vermerkten Textbereiche überprüft (siehe Kapitel 7.2.6). Diese Überprüfung stellt sicher, dass alle Stellen, an denen zuvor Teilpattern gefunden wurden, auch einer Prüfung unterzogen werden. Jedoch ist es möglich, dass zum Zeitpunkt der finalen Prüfung in der Liste einige Intervalle enthalten sind, die zuvor durch einen anderen „Auslöser“ dann doch bereits überprüft wurden.

Um hier unnötigen Verifikationsaufwand zu vermeiden, können die jeweiligen Verify-Datensätze gesichert werden, anstatt sie in Zeile 3 von Teilalgorithmus *CheckVL* zu verwerfen. Damit wird es vor der Prüfung der Liste *CL* möglich, diese durch einen einfachen Bereichsabgleich von Intervallen zu bereinigen, die bereits geprüft wurden.

Ein Parameter `REMOVE_VER_FROM_CHECK` wurde so eingeführt, um bei dem Algorithmus zwischen diesen beiden Möglichkeiten wählen zu können. Es zeigte sich jedoch, dass dieser Bereichsabgleich aufgrund des geringen Verwaltungsaufwands bei gleichzeitig großem Nutzen immer sinnvoll ist.

7.4.3 Gleichzeitige Betrachtung mehrerer Regeln - Minimumswahl

Der Teilalgorithms *CheckVL* wird nacheinander für jede der betrachteten Regeln ausgeführt. Dabei werden jedes Mal zum Teil die Listen *HitList* und *CL* durchlaufen. Durch gleichzeitige Betrachtung aller in Teilalgorithmus *ProcessRules* ausgewählten Regeln, besteht die Möglichkeit diese mehrfachen Durchläufe auf insgesamt maximal einen zu reduzieren.

Dazu wird pro Regel R ein Zeiger auf das erste Element der *R.Positionlist* in einem Feld gespeichert. Beim Durchlauf durch dieses Feld wird immer die minimale aller Positionen gefunden und entsprechend verarbeitet (d. h. gegebenenfalls wird ein Eintrag in die entsprechenden Listen vorgenommen). Auf diese Art und Weise werden zugleich die Positionslisten aller beteiligen Regeln durchlaufen und dabei aufwendige Suchen nach der richtigen Einfügestelle in den Listen *HitList* und *CL* gespart.

Ein Parameter `MINIMUM_OPT` ermöglicht es, dieser Art der Regelbearbeitung zu folgen.

7.4.4 Verifikation

In Kapitel 7.2.4 wurde mit dem Teilalgorithmus *SearchAndVerify* der Teil des Algorithmus vorgestellt, der die Abfolge von Such-, Verifikations- und Informationsauswertungsphase steuert. In den Zeilen 5 und 6 wird dort für ein an der Stelle t_x im Text gefundenes Subpattern die Anfangsposition t_b und die Endposition t_e des zu überprüfenden Bereichs berechnet.

Für die Bestimmung der Bereichsgrenzen wird dort im allgemeinen Fall davon ausgegangen, dass von diesem Bereich nur bekannt ist, dass das Subpattern P_i an der Stelle t_x exakt im Text auftritt. Das Subpattern könnte also durchaus das letzte im Pattern P sein und auch nur ein Zeichen lang sein, was für den zu überprüfenden Bereich bedeutet, dass sich das ganze Pattern (bis auf das eine Symbol) und alle k Fehler noch vor t_x befinden könnten. Der Bereichsanfang berechnet sich demnach als

$$pos_b = t_x - m - k + 1\,. \tag{7.4.1}$$

Für die Berechnung des Bereichsendes ist die Möglichkeit in Betracht zu ziehen, dass P_i das erste Teilpattern von P ist, im Approximate Matching also alle m Patternsymbole sowie die k Fehler noch folgen können:

$$pos_e = t_x + m + k - 1\,. \tag{7.4.2}$$

Im Allgemeinen ist ein jeweils möglichst kleiner Verifikationsbereich von Interesse. Unter Verwendung von leicht verfügbarem zusätzlichen Wissen ist es möglich, diesen durch die Gleichungen 7.4.1 und 7.4.2 gegebenen Bereich $[pos_b, pos_e]$ noch zu verkleinern.

Sei nun zusätzlich die Position p_x $(0 \leq p_x \leq m-1)$ des Subpatterns P_i in P, sowie die Länge $|P_i|$ von P_i gegeben. Unter der Annahme, dass das Teilpattern P_i innerhalb des Patterns P einzigartig ist (was sich bei der initialen Patternzerlegung (siehe Kapitel 7.2.2.2) leicht feststellen lässt), können – wie im Folgenden beschrieben – verbesserte Grenzen des Verifikationsbereichs berechnet werden.

Für den Bereichsanfang ist zu beachten, dass genau alle vor P_i in P liegenden Symbole (also genau p_x Stück) und die k Fehler am Anfang eines möglichen Approximate Matching liegen könnten. Andererseits könnten in einem Approximate Matching ab der Position t_x noch die k Fehler und $m - p_x$ Symbole von P folgen (also inklusive der Position t_x). Für die neuen Bereichsgrenzen ergibt sich somit:

$$pos_b = t_x - k - p_x \tag{7.4.3}$$
$$pos_e = t_x + k + m - p_x - 1 \tag{7.4.4}$$

Für den Fall des nicht in P einzigartig vorhandenen Subpatterns P_i, kann der Verifikationsbereich ebenso, wenn auch nur sehr wenig, verkleinert werden. Dazu ist zu bedenken, dass p_x aufgrund der Nicht-Einzigartigkeit von P_i keine wirkliche Aussagekraft hat, aber die Länge des Subpatterns P_i zumindest soweit berücksichtigt werden kann, dass keines der Symbole von P_i vor der Position t_x liegt. Eine gegenüber der Ausgangsvariante (Gleichung 7.4.2) verbesserte Bereichsendposition lässt sich leider damit nicht erreichen. Die neuen Bereichsgrenzen im Fall der Nicht-Einzigartigkeit von P_i sind also:

$$pos_b = t_x - k - m + |P_i| \tag{7.4.5}$$
$$pos_e = t_x + k + m - 1 \tag{7.4.6}$$

Ist P_i nicht einzigartig in P, aber gibt dabei p_x immer die Position des ersten der gleichen Subpattern im Pattern an, so kann statt Gleichung 7.4.5 auch

$$pos_b = t_x - k - p_x \tag{7.4.7}$$

Verwendung finden, da mit p_x immer eine Anzahl von Symbolen gegeben ist, die sich garantiert vor P_i in P befinden.

Für den Algorithmus hier werden ausschließlich die durch die Gleichungen 7.4.3 bis 7.4.6 beschriebenen Grenzen des Verifikationsbereichs verwendet.

Auch wenn in der Regel ein möglichst kleiner Verifikationsbereich wichtig ist, so sind auch Fälle möglich, wo die Anwendung der Grenzen aus den Gleichungen 7.4.1 und 7.4.2 Vorteile hat (siehe Kapitel 5).

8 Filteransatz mit Grammatiknutzung - Analyse

> Das Argument gleicht dem Schuss einer Armbrust – es ist gleichermaßen wirksam, ob ein Riese oder ein Zwerg geschossen hat.
>
> *(Francis Bacon, engl. Staatsmann und Philosoph, 1561-1626)*

Im vorangehenden Kapitel 7 wurde ein Algorithmus vorgestellt, der die Idee umsetzt, mittels einer Grammatik mehrfach auftretende Textabschnitte während der Suche nach Approximate Matchings nur einmal zu betrachten.

Dieses Kapitel geht zuerst (in Kapitel 8.1) auf praktische Aspekte im Zusammenhang mit dem Algorithmus ein, die es erlauben, die in Kapitel 8.2 folgende Analyse experimentell zu unterlegen. Die wichtigen Ergebnisse werden abschließend in Kapitel 8.3 zusammengefasst.

8.1 Praktische Aspekte

Für die Implementierung und Ausführung des bereits vorgestellten Algorithmus sind noch verschiedene Details von Bedeutung, die an dieser Stelle erläutert werden. Im ersten Unterabschnitt werden neben konkreten implementierungstechnischen Aspekten beim eigentlichen Algorithmus auch die zur Anwendung kommenden Vergleichsalgorithmen knapp diskutiert. Der zweite Unterabschnitt vermittelt die zum Verständnis der durchgeführten Experimente notwendigen Informationen.

8.1.1 Implementierungstechnische Details

Zur Vereinfachung der Beschreibungen insbesondere bei den Vergleichen wird der oben vorgestellte Algorithmus von nun an *GraI* genannt, was in Anlehnung an die Nutzungsweise der Grammatik „grammatik-basierter Index“ bedeuten soll.

Notwendig für den Algorithmus GraI sind die im Teilalgorithmus *SearchAndVerify* in Kapitel 7.2.4 aufgerufenen Algorithmen für die exakte Suche und die Verifikation.

Für die exakte Suche der Subpattern wird eine Variante des Algorithmus von Sunday [115, 116] verwendet, die wie die in [14] und [87] beschriebene Version, zur gleichzeitigen Suche mehrerer Pattern angepasst wurde.

Zur Verifikation eines Bereichs wird der Algorithmus von Chang und Lampe mit Cut-Off-Heuristik genutzt (siehe Kapitel 3.1.5), der allerdings durch die Möglichkeit eines frühzeitigen Abbruchs der Überprüfung erweitert wurde. Die Idee dabei ist die Erkennung von Situationen, in denen unmöglich ein weiteres Approximate Matching vorhanden sein kann, und die Terminierung des Verifikationsalgorithmus genau in diesem Moment. Konkret ist eine solche Situation erreicht, wenn der letzte Wert einer Spalte der DP-Matrix so groß ist, dass im weiteren Verlauf der Verifikation keine Spalte in der DP-Matrix mehr existieren kann, die mit einem für ein Approximate Matching zulässigen Wert endet. Unter Ausnutzung der Eigenschaft der DP-Matrix, dass sich (unter Verwendung der Levenshtein-Distanz) benachbarte Zellen in dieser Matrix um maximal Eins unterscheiden können, kann von jedem Endwert einer Spalte auf die maximal möglichen Endwerte der Folgespalten geschlossen werden. Hier wird die Verifikation genau dann abgebrochen, wenn in der aktuell betrachteten Spalte der Spaltenendwert größer ist als die Anzahl der noch zu betrachtenden Spalten zuzüglich der Fehlerzahl k, denn dann kann die letzte Spalte nicht mehr mit einem Wert kleiner oder gleich k enden (siehe Abbildung 8.1).

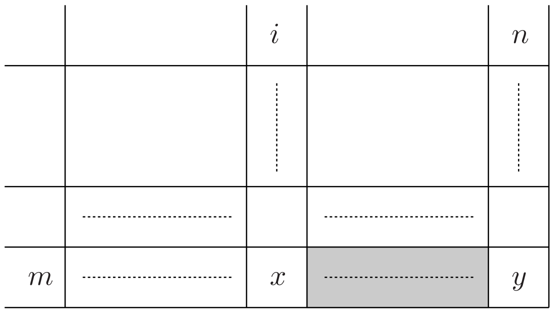


Abbildung 8.1: Frühzeitiger Abbruch bei der Verifikation. Ist x der Wert der DP-Matrix in der untersten Zeile in Spalte i, so kann sich dieser Wert bestenfalls von Spalte zu Spalte im grau markierten Bereich jeweils nur um 1 verringern. Der Wert von y kann also nicht kleiner sein als $x - (n - i)$. Gilt nun $x-(n-i) > k$, so endet in den Spalten i bis n kein Approximate Matching und die Verifikation kann abgebrochen werden.

Der Algorithmus GraI arbeitet mit verschiedenen Listen, auf denen überwiegend mit Einfügeoperationen, aber auch mit Lösch- und Intervallverschmelzungsoperationen gearbeitet wird. Jede verwendete Liste ist als eine sortierte Liste angelegt, bei der bei jeder Operation auf die Einhaltung der Sortierung geachtet wird. Es ist oft der Fall, dass nacheinander Elemente betrachtet werden, die zum Beispiel beim Einfügen dann in genau derselben Reihenfolge in der Liste auftreten (vgl. Teilalgorithmus *CheckVL* in Kapitel 7.2.5). In solchen Momenten sind normale Listenoperationen, die ausgehend vom Listenanfang immer die geeignete Stelle suchen, nicht sehr effektiv. Aus diesem Grund nutzen alle hier verwendeten Listen einen Listeniterator, der die Suche nach der richtigen Operationsstelle immer an der Stelle der zuletzt durchgeführten Operation ansetzt. Damit der einfache Listendurchlauf in beide Richtungen möglich ist, sind die Listen doppelt-verkettet organisiert.

Der Algorithmus GraI verwendet ebenso für die Ausgabe der gefundenen Approximate Matching eine sortierte Liste *HitList*. Um nur die Ergebnisse auszugeben, ist an dieser Stelle keine sortierte Liste notwendig, sondern im Gegenteil noch mit zusätzlichem Aufwand verbunden. GraI generiert die Ausgabe nicht linear, sondern aufgrund der Nutzung der Grammatik in einer recht „willkürlichen“ Reihenfolge, bei der durchaus einzelne Stellen doppelt auftreten können. Um eine vollständige Vergleichbarkeit mit anderen Algorithmen zu gewährleisten, wurde diese Ergebnisliste sortiert und mit Methoden zur Entfernung von doppelten Ergebnissen implementiert. Dieser für die Berechnung des eigentlichen Ergebnisses nicht notwendige Aufwand wird so doch als Bestandteil des Algorithmus gewertet und ist in jeder Laufzeitmessung mit enthalten.

Um den Algorithmus GraI bewerten und vergleichen zu können, ist das Gesamtszenario des Approximate String Matching Problems implementiert. Neben GraI werden als Vergleichsalgorithmen zwei andere Filteralgorithmen gestellt.

Der allgemeine Filteralgorithmus nach Wu und Manber (siehe Kapitel 4.1.2.3), der auch die Basis für GraI darstellt, wird hier mit *Pk1* bezeichnet, was eine Kurzbenennung für das Partitionieren des Patterns in $k{+}1$ Teile darstellen soll. Das Pattern wird wie bei GraI gleichmäßig aufgeteilt. Während der folgenden Phasen exakter Suche und Verifikation werden dieselben Algorithmen (und Implementierungen) verwendet, die auch bei GraI zum Einsatz kommen.

Als zweiter Vergleichsalgorithmus wird hier der Algorithmus *SLEQ* („static location of exact q-grams“) von Sutinen und Tarhio verwendet [119]. Dabei handelt es sich um eine von mehreren Algorithmusvarianten, die alle auf dem in Kapitel 4.1.3.2 erläuterten Prinzip basieren. Zwei wesentliche Kriterien sind bezeichnend für die Auswahl von SLEQ als Vergleichsalgorithmus:

- SLEQ arbeitet als indexbasierter Algorithmus ebenso wie GraI „offline“ (siehe Kapitel 2.6).

- GraI ist ein Algorithmus der Klasse der Filteralgorithmen, die bei einer Aufteilung des Problems in exakte Suche von der Annahme ausgehen, die Fehler seien im Pattern vorhanden. Der erste gewählte Vergleichsalgorithmus Pk1 ist auch aus dieser Klasse und zählt (nach [84, 85]) zu den schnellsten Algorithmen seiner Klasse (für Fehlerlevel, die die Dominanz der Suchphase erlauben). Um einen Vergleich mit einem Offline-Algorithmus der anderen Klasse (Aufteilung in exakte Suche bei Annahme des Fehlers im Text) zu haben, fiel die Entscheidung auf SLEQ als Vergleichsalgorithmus.

Der SLEQ-Algorithmus funktioniert wie in Kapitel 4.1.3.2 beschrieben, allerdings mit der kleinen Änderung, dass der Parameter s bei der $k+s$ Zerlegung ideal aus der Schrittweite und der Größe der dem Text entnommenen q-Samples berechnet wird.

SLEQ verwendet hier zur Verifikation dieselben Routinen wie GraI, muss jedoch für die exakte Suche eigene Routinen nutzen, die den Index (hier als Hashtabelle organisiert) entsprechend betrachten.

Bei SLEQ ist die Wahl der Größe q der q-Samples und der Schrittweite h ($h \geq q$), in der die q-Samples dem Text entnommen werden, notwendig. Das jeweils optimale s für die angenommene Patternaufteilung wird aus diesen Werten berechnet. Im Rahmen dieser Arbeit wurde SLEQ mit den Parametern $q = 3$ und $h = 3$ verwendet (genaueres dazu geht aus Kapitel 8.2.4 hervor), sofern nichts anderes vermerkt ist.

Insgesamt wurde versucht, alle verwendeten Algorithmen so vergleichbar wie möglich zu halten. Aus diesem Grund wurde überall, wo dieses möglich ist (d. h. insbesondere bei der Eingabe, der Ausgabe und der Verifikation), auf dieselben Routinen zurückgegriffen.

8.1.2 Erläuterungen zu den Experimenten

Sowohl der Algorithmus GraI, als auch die Vergleichsalgorithmen Pk1 und SLEQ sind hier vollständig in Java implementiert. Jeder dieser Algorithmen verwendet zur Verifikation genau dieselbe Routine (eine leichte Modifikation des Algorithmus von Chang und Lampe, siehe Kapitel 8.1.1), die zu einem gegebenen Textintervall und einer Anzahl maximal erlaubter Fehler alle Endpunkte von Approximate Matchings in diesem Intervall liefert. Während zudem auch das gesamte Rahmenprogramm bei allen drei Algorithmen identisch ist, benötigt zumindest SLEQ eine eigene Routine für die exakte Suche (siehe Kapitel 8.1.1). Insgesamt wurde versucht, die höchstmögliche Vergleichbarkeit zu erhalten.

Aufgrund des Entstehungszeitraumes dieser Arbeit und verschiedener Arbeitsorte kamen bei der Programmausführung verschiedene Rechner zum Einsatz. Alle in diesem Kapitel durchgeführten Berechnungen wurden entweder auf einem 2.4 GHz Intel Pentium 4 Linux PC, oder auf einem 1.5 GHz Pentium M Windows PC ausgeführt. Selbstverständlich sind alle direkten Vergleiche immer unter den gleichen Bedingungen durchgeführt worden.

Die absolute Systemlaufzeit ist bei den durchgeführten Vergleichen nicht von Bedeutung, was ansonsten bei dieser Anwendung eine optimierte Implementierung in C erfordert hätte. Wichtig beim Vergleich ist eine qualitative Betrachtung der Ergebnisse insbesondere im Hinblick auf die in Kapitel 8.2 gewonnenen Erkenntnisse. Dies bedeutet, dass gerade in Bezug auf das Laufzeitverhalten die direkten Vergleiche und die Änderungen in Abhängigkeit von der zulässigen Fehlerzahl zu betrachten sind.

In den Experimenten werden die verschiedenen Algorithmen dazu genutzt, in verschiedenen Texten nach Approximate Matchings von Pattern zu suchen. Die gesuchten Pattern werden durchweg dem jeweils gegebenen Text an zufälligen Positionen entnommen. Die Bestimmung der Zufallswerte wird dabei mit dem Algorithmus des *Mersenne Twisters* [76] vorgenommen. In den Experimenten wurden jeweils mindestens 20 Pattern zur Suche verwendet. In den Abbildungen finden sich dann auch jeweils entsprechende Balken, die die Standardabweichung aufzeigen. Ausgenommen davon sind einzelne Experimente, bei denen im Vorfeld schon eine sehr hohe Rechenzeit absehbar war.

Die Texte, die die Grundlage der Suchen bilden, entstammen – solange nicht anders vermerkt – einer der vier folgenden Kategorien:

- Zufallstext. Der Text einer beliebigen Länge n wird zufällig (unter Nutzung des Mersenne Twisters) aus einem Alphabet einer beliebigen Größe σ generiert.

- Englischer Text. Die King-James-Bibel bildet den Ausgangspunkt für Texte in englischer Sprache. Der Text ist vollständig in Großbuchstaben konvertiert und alle Trennzeichen werden, abgesehen von Zeilenumbrüchen, auf Leerzeichen abgebildet. Der so bearbeitete Bibeltext basiert auf einem Alphabet der Größe $\sigma = 28$. Um einen Text einer beliebigen Länge n zu erhalten, wird der konvertierte Bibeltext ab der entsprechenden Länge n abgeschnitten.

- DNS. Als Testdaten biologischen Ursprungs findet hier das vollständige Genom des Bakteriums Haemophilus Influenzae Rd Verwendung. Neben den Symbolen des vier-buchstabigen DNS-Alphabets (A,C,G,T) ist noch ein sehr geringer Prozentsatz (deutlich weniger als 0.01%) an anderen Symbolen vorhanden, die verschiedene Mehrdeutigkeiten erklären. So steht zum Beispiel ein P für Pyrin, was sowohl Cytosin (C) als auch Thymin (T) sein kann, und ein W für eine schwache („weak“) Bindung, was dann entweder Adenin (A) oder Thymin (T) bedeutet. Das dem Text zugrunde liegende Alphabet erreicht damit insgesamt eine Größe von $\sigma = 11$. Der vollständige Datensatz hat eine Länge von etwa 1.77 MB und wird, sofern notwendig, auf die gegebene Länge n beschränkt.

- Programmquellcode. Der Quellcode eines C-Programms wurde von Kommentaren bereinigt als Testdatum verwendet. Der sich ergebende Text hat eine Größe von $n = 264102$ und basiert letztendlich auf einem Alphabet der Größe $\sigma = 71$.

8.2 Analyse

Zum besseren Verständnis des praktischen Verhaltens des Algorithmus GraI ist eine analytische Betrachtung unerlässlich. Zu diesem Zwecke wird in Kapitel 8.2.1 berechnet, ab wann GraI gegenüber dem zugrunde liegenden Algorithmus Pk1 eine verbesserte Filtereffizienz aufweist. Dabei wird auch gezeigt, dass die durchschnittliche Länge der Regeln in der Grammatik im Vorfeld berechenbar ist.

Basierend auf diesen Betrachtungen der Filtereffizienz wird in Kapitel 8.2.2 die Laufzeit des Algorithmus insbesondere im Hinblick auf den Bereich einer dominanten Suchphase (was dem sinnvollen Nutzbereich entspricht) untersucht.

Anschließend werden in Kapitel 8.2.3 die Auswirkungen der Algorithmusvariante der Minimumswahl (siehe Kapitel 7.4.3) betrachtet, bevor in Kapitel 8.2.4 die zuvor für GraI gewonnenen Erkenntnisse explizit in Bezug auf andere Algorithmen und verschiedene Datensätze zusammengefasst werden. Kapitel 8.2.5 betrachtet dann GraI direkt unter der Perspektive der Anwendung.

Insgesamt ist es in diesem Kapitel an vielen Stellen aufgrund der Nutzung der Grammatik nur durch Abschätzungen möglich, konkrete Aussagen zu treffen.

8.2.1 Die Filtereffizienz

Im Vergleich zum Algorithmus Pk1 wurde GraI so konzipiert, dass der Aufruf der Verifikationsroutine an einigen Stellen eingespart werden kann. Durch diese Einsparungen wird direkt die *Filtereffizienz* („Filtration Efficiency") beeinflusst, welche ein Maß darstellt, den Wirkungsgrad verschiedener Filteralgorithmen zu vergleichen.

Grundsätzlich existieren zwei verschiedene Möglichkeiten eine *Filtereffizienz* zu berechnen:

- Filtereffizienz bezüglich der Textlänge (nach [119]):
  $$f_n = (n - n_p)/n$$
  n_p bezeichnet dabei die Anzahl der Symbole des Textes, die während der Verifikation betrachtet werden. f_n beschreibt somit den Gesamtanteil der überprüften Symbole am Text.

- Filtereffizienz bezüglich der Anzahl der Approximate Matchings (nach [117]):
  $$f = mat_r/mat_p$$
  mat_r bezeichnet dabei die Anzahl gefundener (realer) Approximate Matchings, während mat_p die Anzahl der vom Filter identifizierten potenziellen Approximate Matchings darstellt. So beschreibt f also das Verhältnis von wirklich vorhandenen Approximate Matchings zu den zuvor als mögliche Approximate Matchings identifizierten.

Im Verlauf der hier getätigten Experimente wurde nur f berechnet. Ein potenzielles Approximate Matching wird jedes Mal dann gezählt, wenn der Verifikationsalgorithmus aufgerufen wird. Nachdem der Text vollständig durchsucht wurde und somit alle Approximate Matchings gefunden wurden, kann f dann berechnet werden.

In der Regel ist f kleiner als 1, weil mehr mögliche Approximate Matchings identifiziert werden, als später real gefunden werden. Jedoch kann f auch größer als 1 werden, wenn die Verifikation, die dem Entdecken eines potenziellen Approximate Matching folgt, mehr als nur ein wirkliches Approximate Matching erkennt und weiterhin diese realen Approximate Matchings später (im weiteren Verlauf des Algorithmus) nicht auch anderen potenziellen Approximate Matchings zugeordnet werden.

Für den Vergleich des Algorithmus GraI mit dem grundsätzlichen Filteralgorithmus Pk1 kann eine abschätzende Bedingung bestimmt werden, die angibt, wann sich die Filtereffizienz von GraI gegenüber der von Pk1 verbessert. Dazu wird ausgenutzt, dass das Filterprinzip bei beiden Algorithmen identisch ist. Es ist somit nur notwendig, die Fälle zu betrachten, in denen ein potenzielles Approximate Matching zu weiteren potenziellen oder realen Approximate Matchings führt. Diese Fälle können nur dann eintreten, wenn die Bedingung

$$R.length \geq 2P_{max} - 1$$

gilt (siehe Kapitel 7.4.1). Natürlich ist nicht zwangsläufig jedes exakt im Text vorhandene Subpattern (also potenzielles Approximate Matching) Teil einer Regel, aber Bereiche, die nicht durch Regeln abgedeckt werden, werden in beiden Algorithmen gleich behandelt. Prinzipiell reicht es für eine bessere Filtereffizienz aus, wenn nur einmal ein potenzielles Approximate Matching zu mehreren Approximate Matchings führt. Um jedoch von einem wirklich wahrnehmbaren Effekt sprechen zu können, ist es notwendig, dass obige Bedingung für die meisten Regeln gilt. Dazu wird als Regellänge $R.length$ die durchschnittliche Länge aller Regeln l_{av} zugrunde gelegt und ferner wird angenommen, dass $P_{max} = \frac{m}{k+1}$ gilt. Somit ergibt sich die Bedingung

$$l_{av} \geq 2\frac{m}{k+1} - 1\,.$$

Durch Ersetzen von $k+1$ durch k im Nenner kann diese Ungleichung noch verschärft werden. Zudem wird damit erreicht, dass der Fehlerlevel $\alpha = k/m$ eingesetzt werden kann. Durch Umformung ergibt sich dann

$$\alpha \geq \frac{2}{l_{av}+1} \tag{8.2.1}$$

als Bedingung für eine bessere Filtereffizienz bei GraI im Vergleich zu Pk1.

Die hier einfach angenommene durchschnittliche Regellänge kann leicht für jeden konkreten Fall aus der Grammatik berechnet werden. Allerdings ist es auch möglich, diese durchschnittliche Regellänge l_{av} im Allgemeinen zu bestimmen. Dazu wird eine Aufteilung des Textes in Blöcke gleicher Länge angenommen. Die Wahrscheinlichkeit dafür, dass ein Block einem bestimmten String (einer Regel) entspricht, beträgt dann $p = 1/\sigma^l$, während die Wahrscheinlichkeit dafür, dass ein Block nicht diesem bestimmten String entspricht, $q = 1 - p = 1 - 1/\sigma^l$ ist.

Wird nun für eine feste Länge l der erste Block „festgehalten“, so ist die Wahrscheinlichkeit dafür, dass der durch den ersten Block dargestellte Substring zweimal im Text auftritt (und somit als Regel gezählt werden kann), die Summe aus den Wahrscheinlichkeiten für eine Übereinstimmung des zweiten mit dem ersten Block, für eine Übereinstimmung des dritten mit dem ersten, aber nicht mit dem zweiten Blocks und so fort. Formal ergibt sich also die Summe $p + qp + \ldots + q^{n/l-2}p = p\sum_{i=0}^{n/l-2} q^i$.

Bei der Betrachtung des zweiten Blocks ergibt sich im Prinzip dieselbe Summe, allerdings multipliziert mit q, weil dieser nicht mit dem ersten Block identisch ist, also $pq\sum_{i=0}^{n/l-2} q^i$. Allgemein ergibt sich bei der Betrachtung von Block j

$$pq^{j-1} \sum_{i=0}^{(\frac{n}{l}-2)-(j-1)} q^i$$

als Wahrscheinlichkeit dafür, dass dieser eine Regel ist.

Bezeichnet nun $w(l)$ für eine feste Länge l die Summe der Wahrscheinlichkeiten für jeden möglichen Block j, dann ist

$$\begin{aligned}
w(l) &= \sum_{j=1}^{\frac{n}{l}-1} pq^{j-1} \sum_{i=0}^{(\frac{n}{l}-2)-(j-1)} q^i \\
&= \sum_{j=0}^{\frac{n}{l}-2} pq^j \sum_{i=0}^{\frac{n}{l}-2-j} q^i \\
&= p\sum_{j=0}^{\frac{n}{l}-2} q^j \frac{1-q^{\frac{n}{l}-1-j}}{1-q} \\
&= \sum_{j=0}^{\frac{n}{l}-2} (q^j - q^{\frac{n}{l}-1}) \\
&= \left(\sum_{j=0}^{\frac{n}{l}-2} q^j\right) - \left(\frac{n}{l}-1\right) q^{\frac{n}{l}-1} && (8.2.2) \\
&= \frac{1-q^{\frac{n}{l}-1}}{1-q} - \left(\frac{n}{l}-1\right) q^{\frac{n}{l}-1} && (8.2.3)
\end{aligned}$$

Unglücklicherweise wird aufgrund der Beschränktheit der Fließkommadarstellung der Nenner in Gleichung 8.2.3 Null für große l, so dass hier zur Berechnung auf die Form aus Gleichung 8.2.2 zurückgegriffen werden muss.

	4	8	10	20	40	80
10000	4.99	3.51	3.17	2.60	2.06	2.00
	5.01	3.69	3.38	2.57	2.12	2.02
50000	6.03	4.17	3.87	3.02	2.55	2.04
	6.15	4.25	3.98	3.22	2.46	2.07
100000	6.48	4.50	4.08	3.16	2.81	2.14
	6.68	4.51	4.15	3.52	2.70	2.15
500000	7.55	5.19	4.79	3.82	3.03	2.78
	7.72	5.27	4.69	3.96	3.36	2.59
1000000	8.02	5.53	5.03	3.97	3.14	2.93
	8.17	5.63	4.98	4.03	3.69	2.83

Tabelle 8.1: Vergleich von theoretisch und praktisch ermittelten Werten für die durchschnittliche Regellänge l_{av}. Die Spalten betrachten unterschiedliche Alphabetgrößen σ. Die Zeilen unterteilen in erster Linie nach verschiedenen Textlängen n, jedoch sind für jede Textlänge und jede Alphabetgröße immer zwei Werte gegeben. Der obere der beiden Werte ist der theoretisch für l_{av} ermittelte Wert, während der untere (grau unterlegte) Wert immer den im Experiment gemessenen (als Mittelwert von 10 Versuchen) Wert darstellt.

Durch Bilden der gewichteten und anschließend normierten Summe über jede mögliche Regellänge kann mittels $w(l)$ wie folgt die durchschnittliche Regellänge l_{av} ermittelt werden:

$$l_{av} = \frac{\sum_{l=2}^{\frac{n}{2}} l w(l)}{\sum_{l=2}^{\frac{n}{2}} w(l)} \tag{8.2.4}$$

Tabelle 8.1 stellt verschiedene durch Gleichung 8.2.4 gewonnene Werte solchen Werten gegenüber, die praktisch an Zufallstexten gemessen wurden.

Auch wenn die theoretisch berechneten Werte für die durchschnittliche Regellänge l_{av} recht gut mit den praktischen Werten übereinstimmen, so sind doch in Einzelfällen auch größere Abweichungen nicht ausgeschlossen. Deswegen wurde bei den zur Filtereffizienz durchgeführten Experimenten für die Berechnung der in Bedingung 8.2.1 gegebenen Schranke der real in der Grammatik ermittelte Durchschnittswert für die Regellänge verwendet.

Abbildung 8.2 zeigt anhand zweier Beispiele die Filtereffizienz bei der Suche auf einem Zufallstext. In Abbildung 8.2(a) ist die gemessene durchschnittliche Regellänge $l_{av} = 6.63$. Eingesetzt in Gleichung 8.2.1 ergibt sich ein Fehlerlevel von $\alpha = 0.262$ (was bei $m = 10$ dann $k \geq 2.62$ bedeutet), für den die Filtereffizienz bei GraI besser (d. h. größer) ist als bei Pk1. Dieser berechnete Wert steht in guter Übereinstimmung mit dem im Experiment erzielten Ergebnis. Abbildung 8.2(b) zeigt einen Fall, bei dem der berechnete Wert (wie aus der Analyse zu erwarten ist) nur die ungefähre Position für die in GraI verbesserte Filtereffizienz angibt. Die durchschnittliche Regellänge ist dort $l_{av} = 3.67$, was nach Gleichung 8.2.1 zu $\alpha \geq 0.428$ führt. Mit der betrachteten Patternlänge $m = 10$ ist der Effekt der besseren Filtereffizienz ab $k = 4.28$ zu erwarten. Jedoch ist es nicht erstaunlich, dass der Verbesserungseffekt schon für kleinere k auftritt; denn prinzipiell kann jede einzelne Regel, die die Bedingung erfüllt, zu einer Verbesserung der Filtereffizienz führen, während aber erst der wirklich deutliche Effekt zu erwarten ist, wenn dies im Durchschnitt der Fall ist.

In beiden in der Abbildung 8.2 gezeigten Fällen ist die Filtereffizienz von GraI und Pk1 besser als die von SLEQ. Im Allgemeinen ist dies nicht so, sondern vielmehr kann SLEQ bei etwas längeren Suchpattern und vor allem bei geringen Fehlerleveln eine im Vergleich recht hohe Filtereffizienz aufweisen, zumal gerade GraI nach Gleichung 8.2.1 wirkliche Vorteile erst bei mittleren Fehlerleveln hat. Abbildung 8.3 zeigt genau dieses anhand von Suchen in einem Zufallstext und in einem Text in englischer Sprache auf. Deutlich ist, dass die nach Gleichung 8.2.1 berechnete Grenze dafür, dass GraI gegenüber Pk1 eine verbesserte Filtereffizienz besitzt, immer im Bereich der mittleren Fehlerlevel liegt, in denen auch SLEQ keine gute Filtereffizienz mehr aufweisen kann.

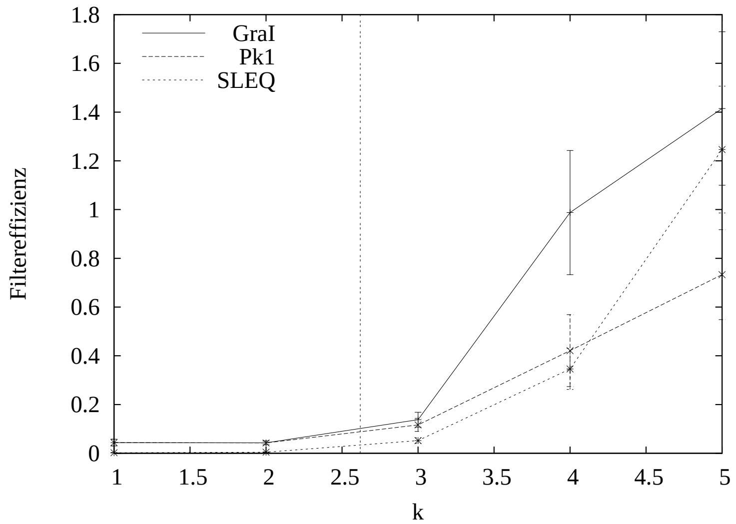
Abbildung 8.3 zeigt aber auch einen anderen, im Hinblick auf die Anwendung von GraI wichtigeren Effekt. Für einen englischen Text entstehen im Vergleich zu einem zufälligen Text mit etwa gleich großem Alphabet (wie dies hier der Fall ist) aufgrund der stärkeren „Strukturierung“ längere Regeln. So liegt auch, wie in Abbildung 8.3(b) zu sehen, aufgrund der längeren durchschnittlichen Regellänge die Grenze im Vergleich zum Zufallstext deutlich weiter links, also bei kleineren Fehlerleveln. Abbildung 8.4 zeigt einen Detailausschnitt aus Abbildung 8.3(b), der die Veränderung der Filtereffizienz von GraI gegenüber Pk1 verdeutlicht.

8.2.2 Laufzeitbetrachtung

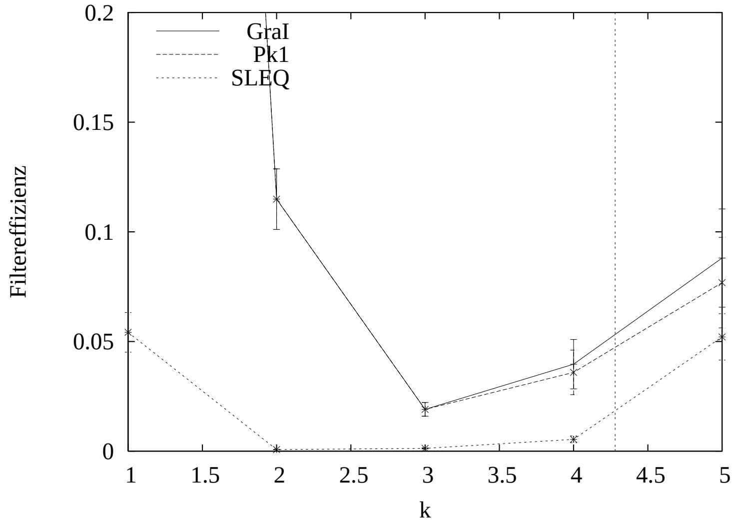
Viel wichtiger als die Filtereffizienz (vgl. Kapitel 8.2.1) ist aus praktischer Sicht die Laufzeit eines Algorithmus, bzw. gerade bei Filteralgorithmen der maximale Fehlerlevel α, für den der Gesamtalgorithmus noch die Laufzeit der Suchphase hat (siehe Kapitel 4.1).

Im Folgenden werden ausgehend von der Betrachtung der Auswirkungen der Filtereffizienz auf die Laufzeit zwei wichtige auf Algorithmusvarianten basierende Anpassungen des Algorithmus vorgenommen (Kapitel 8.2.2.1), die dann eine korrekte Bestimmung des maximalen Fehlerlevels erst ermöglichen (Kapitel 8.2.2.2).

Den Abschluss der Laufzeitbetrachtungen bildet ein Abschnitt über den in der Vorberechnungsphase anfallenden Rechenaufwand (Kapitel 8.2.2.3).

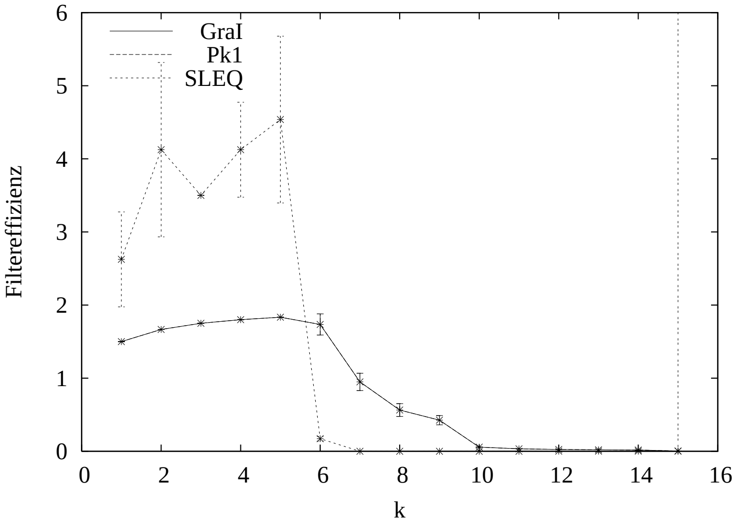


(a) $\sigma = 4, n = 100000, m = 10, l_{av} = 6.63$

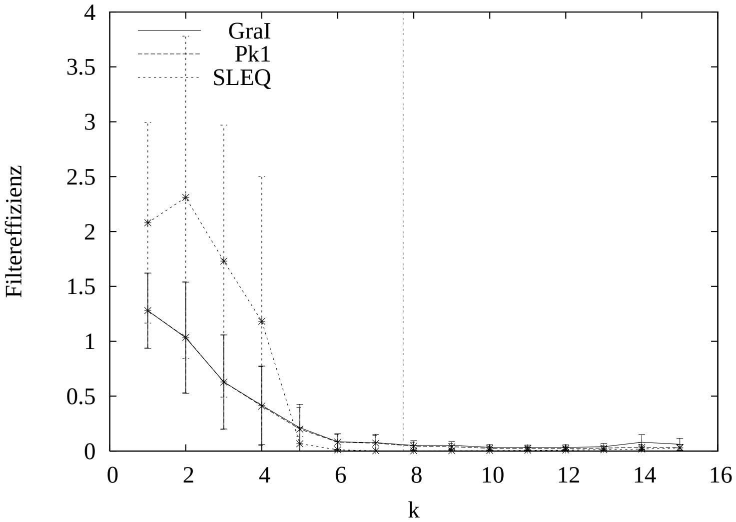


(b) $\sigma = 30, n = 20000, m = 10, l_{av} = 3.67$

Abbildung 8.2: Die Filtereffizienz bei kurzen Pattern. Mit Gleichung 8.2.1 kann grob die Fehlerzahl k berechnet werden, ab der die Filtereffizienz von GraI sich im Vergleich zu der von Pk1 verbessert. Der ermittelte Wert ist durch eine senkrechte Linie gekennzeichnet. Damit der Effekt besser sichtbar wird, wurde in (b) die Darstellung auf eine maximale Filtereffizienz von 0.2 beschränkt.



(a) $\sigma = 30, n = 100000, m = 30, l_{av} = 3.00$, Zufallstext



(b) $\sigma = 28, n = 20000, m = 30, l_{av} = 7.72$, Englischer Text

Abbildung 8.3: Die Filtereffizienz bei längeren Pattern. GraI erzielt bei geringeren Fehlerleveln und nicht zu kurzen Pattern keine bessere Filtereffizienz als Pk1. Vielmehr können in diesem Fall beide Algorithmen weder auf Zufallstext noch auf englischem Text mit SLEQ konkurrieren. Die durch Gleichung 8.2.1 gegebene Bedingung ist jeweils durch eine senkrechte Linie markiert.

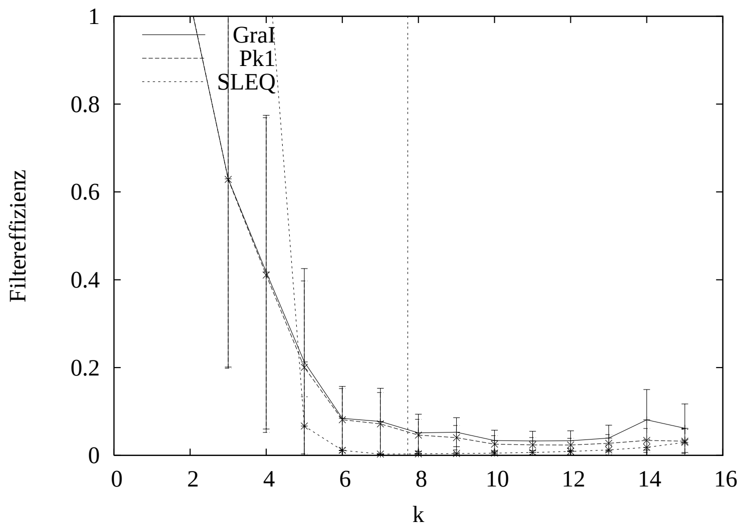


Abbildung 8.4: Detailausschnitt aus Abbildung 8.3(b). Bedingung 8.2.1 (gegeben durch die senkrechte Linie) markiert auch hier (bei englischem Text) etwa die Stelle, an der die Filtereffizienz von GraI gegenüber Pk1 ansteigt.

8.2.2.1 Auswirkungen der Filtereffizienz

Wie in Kapitel 8.2.1 festgestellt wurde, wird eine im Vergleich zu Pk1 bessere Filtereffizienz bei mittleren Fehlerleveln erreicht, die bis in den Bereich hereinragen, ab dem die Fehler ein Übergewicht bekommen (vgl. Kapitel 2.5).

Der Algorithmus GraI wurde basierend auf Pk1 konzipiert. Ein zentraler Punkt dabei ist die Vervielfältigung von gefundenen Approximate Matchings über Regeln der Grammatik. Im Falle dieser Vervielfältigung wird also eine bessere Filtereffizienz erzielt und zugleich kann an Verifikationsaufwand gespart werden. Ein anderer zentraler Punkt der Konzeption ist die Vervielfältigung von möglichen Approximate Matchings. Dabei verändert sich nicht die Filtereffizienz und es wird auch kein Verifikationsaufwand, sondern nur wenig der im Aufwand eher unbedeutenden exakten Suche eingespart.

Für die nun folgenden Laufzeitbetrachtungen ist es also interessant, den Algorithmus GraI ab dort zu betrachten, wo er gegenüber Pk1 eine bessere Filtereffizienz aufweisen kann.

In Abbildung 8.5 wird der Beispielfall aus Abbildung 8.4 aufgegriffen. Diesmal sind anstelle der Filtereffizienzen die Ausführungszeiten (ohne Vorberechnungen) der verschiedenen Algorithmen aufgetragen.

Es ist in Abbildung 8.5 schön zu sehen, dass hier GraI für mittlere Fehlerlevel zugleich mit der besseren Filtereffizienz auch eine bessere Laufzeit aufweist. Dies ist im Allgemeinen jedoch nicht zwingend der Fall. Vielmehr kann prinzipiell eine bessere Filtereffizienz

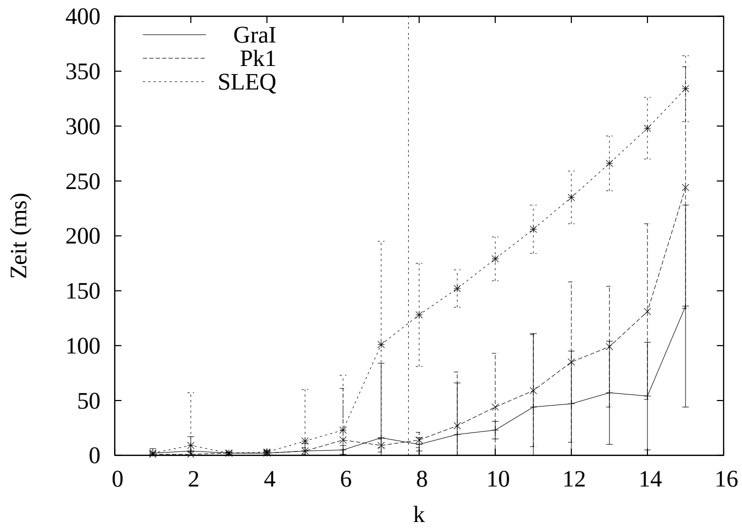


Abbildung 8.5: Suchzeiten des Beispiels aus Abbildung 8.4. Die senkrechte Linie kennzeichnet die Bedingung 8.2.1 für eine verbesserte Filtereffizienz von GraI gegenüber Pk1.

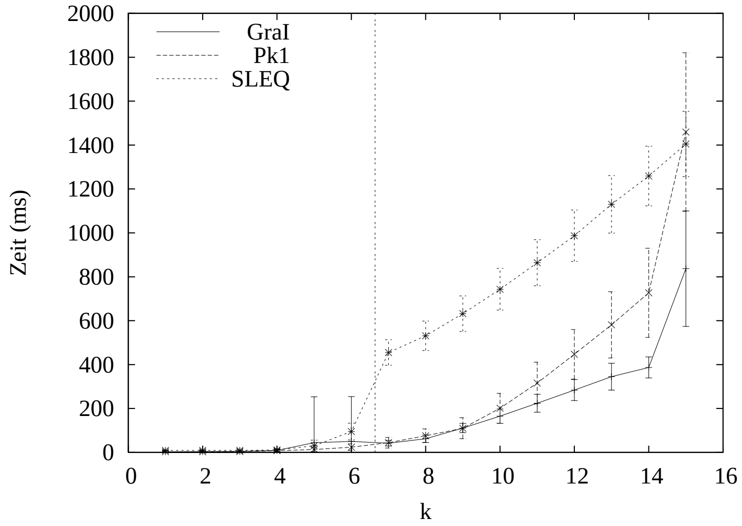
nur eine verbesserte Laufzeit bewirken, wenn der Aufwand zur Erzeugung dieser besseren Filtereffizienz nicht in demselben Maße ansteigt wie der so gewonnene Vorteil.

Bei dem bisher gezeigten Algorithmus GraI ist dies jedoch nicht ganz der Fall, sondern wie gleich deutlich werden wird, besteht eine Abhängigkeit von der Textlänge n. Allerdings wird im Folgenden auch gezeigt werden, dass mittels zweier sehr kleiner Modifikationen des Algorithmus GraI eine bessere Filtereffizienz auch unabhängig von der Textlänge erreicht werden kann.

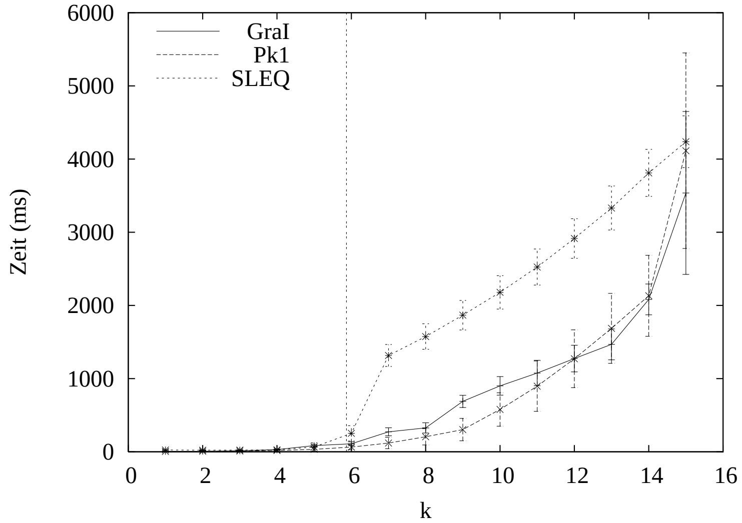
Bei dem zuvor in Abbildung 8.5 gezeigten Beispiel wurde die Textlänge n so gering gewählt, dass die Einflüsse, die den Aufwand für die höhere Filtereffizienz bei längeren Textlängen in die Höhe treiben, nicht ins Gewicht fallen. Abbildung 8.6 führt dieses Beispiel für längere Texte fort. Es ist deutlich zu erkennen, dass sich die Laufzeit von GraI im Vergleich zu Pk1 trotz der verbesserten Filtereffizienz (angezeigt durch die senkrechten Linien) bei größer werdendem n verschlechtert.

Eine genaue Betrachtung der Grundform von GraI (wie in Kapitel 7.2 beschrieben) vermag diesen Effekt zu erklären. Genauer gesagt kommt dieser ungewünschte Effekt durch die Überlagerung von zwei Einflüssen zustande:

1. Viele (insbesondere kurze) Regeln werden im Teilalgorithmus *CheckVL* in Kapitel 7.2.5 betrachtet, aber da sie weder einen Verifikationsbereich noch eines der exakt gefundenen Teilpattern abdecken, passiert dort nichts weiter und insbesondere wird gegenüber Pk1 auch kein Aufwand eingespart. Diese Regeln werden jedoch



(a) $\sigma = 28, n = 100000, m = 30, l_{av} = 8.03$, Englischer Text



(b) $\sigma = 28, n = 300000, m = 30, l_{av} = 9.22$, Englischer Text

Abbildung 8.6: Einfluss der Textlänge bei der Grundform von GraI. Die Kosten zum Erreichen einer besseren Filtereffizienz sind beim unmodifizierten Algorithmus GraI für wachsende n steigend, so dass die Auswirkungen der Filtereffizienz auf die Laufzeit im direkten Vergleich zu Pk1 nicht mehr so stark bemerkbar sind bzw. einen negativen Effekt haben. Die durch Gleichung 8.2.1 gegebene Bedingung für eine bei GraI gegenüber Pk1 verbesserte Filtereffizienz ist jeweils durch eine senkrechte Linie markiert.

auch alle im Teilalgorithmus *UpdateSL* behandelt und entsprechende Bereiche werden von der weiteren Betrachtung durch den Suchalgorithmus ausgenommen. Genau an dieser Stelle entsteht ein nicht offensichtlicher Mehraufwand. Die prinzipielle Arbeitsweise des Sunday-Algorithmus, der hier bei der exakten Suche Verwendung findet (vgl. Kapitel 8.1.1), erlaubt es gerade an Stellen, an denen kein Pattern gefunden wird, relativ schnell durch den Text zu springen. Das bedeutet, dass gerade bei kürzeren Regeln hier nicht Zeit bei der Suche eingespart wird, sondern vielmehr durch die Zerstückelung des Suchbereichs der Suchalgorithmus zum Anhalten und späteren Neu-Ansetzen und somit zu einem Mehraufwand gezwungen wird.

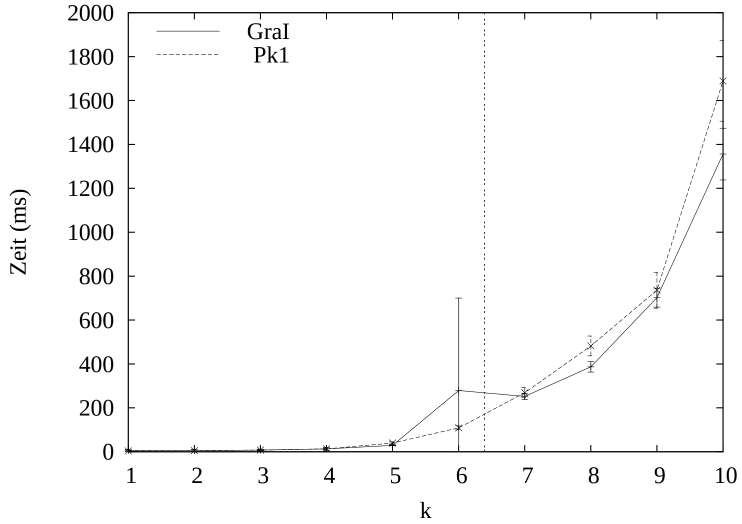
2. Regeln, die im Teilalgorithmus *CheckVL* betrachtet werden und auch zumindest ein exakt gefundenes Teilpattern abdecken, führen zu Einträgen in die Liste *CL* der zu prüfenden Bereiche. Insbesondere kürzere Regeln sind im Schnitt häufiger im Text anzutreffen und führen so zu mehr Einträgen. Je mehr Einträge in *CL* vorhanden sind, desto aufwendiger gestaltet sich das Einfügen neuer Einträge (Verschmelzungen von Einträgen seien dabei vernachlässigt - sie reduzieren zwar die Anzahl der Einträge, doch das Verschmelzen selbst muss auch durchgeführt werden). Werden also zu viele kurze Regeln betrachtet, kann der Aufwand für die Bearbeitung einer einzelnen Regel leicht die bei der Suche eingesparte Zeit übertreffen (beachte, dass die Suche in diesem Fall nicht über den Bereich hinweg „springt“, sondern ein Teilpattern identifizieren wird).
   Dieser Einfluss der Einträge in der Liste *CL* tritt für höhere Fehlerlevel auf, als dies beim erstgenannten Einfluss der Fall ist, da es hier um Regeln geht, die Teilpattern (die bei höheren Fehlerleveln kleiner sind) vollständig abdecken.

Diese beiden Einflüsse können getrennt gesteuert bzw. eliminiert werden.

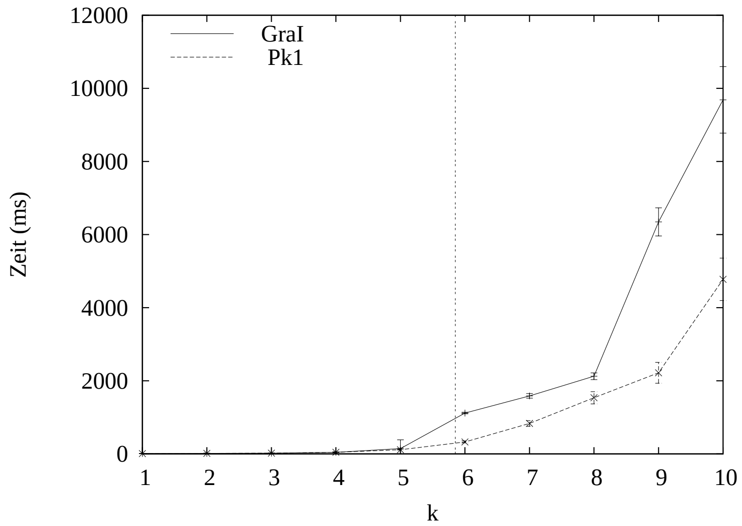
Im ersten Fall ist es essenziell, dafür zu sorgen, dass Regeln, die kein exakt gefundenes Pattern abdecken, nicht den noch zu durchsuchenden Bereich verkleinern, also nicht im Teilalgorithmus *UpdateSL* verarbeitet werden. Dies ist sehr einfach durch eine entsprechende direkte Abfrage zu erreichen. Dafür wurde hier ein Parameter `RULE_REMOVAL` eingeführt, mit dem der Aufruf des Teilalgorithmus *UpdateSL* für die entsprechenden Regeln an- und abgeschaltet werden kann. Abbildung 8.7 verdeutlicht anhand einer Suche in einem Zufallstext noch einmal den oben beschriebenen Effekt, während in Abbildung 8.8 dann konkret auch die Wirkung des Parameteres `RULE_REMOVAL` demonstriert wird.

Es sei noch darauf hingewiesen, dass bei dieser Vorgehensweise immer ein Anteil an Regeln nicht verarbeitet wird, der in direkter Relation zur Textlänge steht. Je länger der Text ist, desto mehr Regeln existieren zwar, aber ebenso decken auch mehr Regeln nicht exakt gefundene Teilpattern ab.

Im zweiten Fall ist dafür Sorge zu tragen, dass nicht zu viele kurze Regeln betrachtet werden (siehe oben). In Kapitel 7.4.1 wurde bereits ein Parameter `MINIMAL_RULELENGTH` eingeführt, der die Möglichkeit schafft, die Länge der betrachteten Regeln zu bestimmen. Die Angabe einer konstanten Länge kann hier im Einzelfall in Bezug auf die Textstruktur

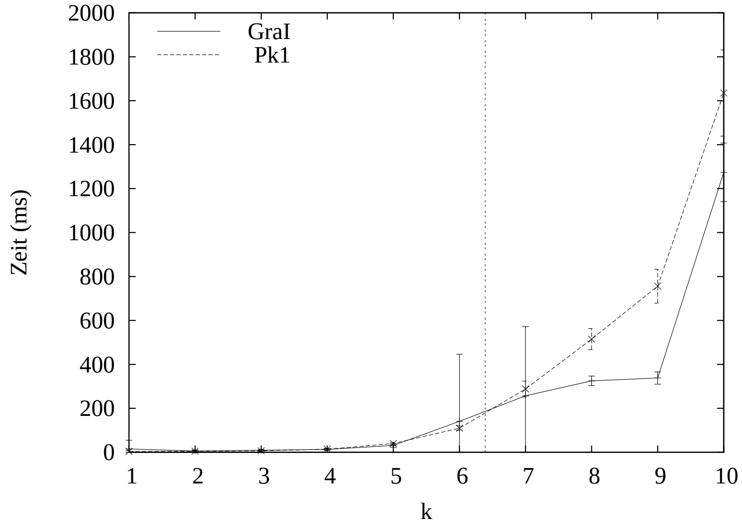


(a) $\sigma = 6, n = 100000, m = 20, l_{av} = 5.26$, Zufallstext

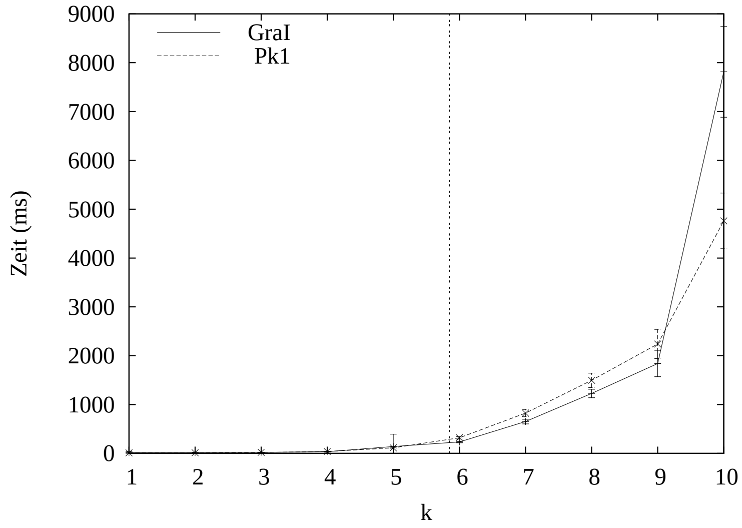


(b) $\sigma = 6, n = 300000, m = 20, l_{av} = 5.84$, Zufallstext

Abbildung 8.7: Einfluss der Textlänge bei der Grundform von GraI. Wie auch schon in Abbildung 8.6 sind hier die bei der Grundform von GraI steigenden Kosten für das Erreichen einer höheren Filtereffizienz bei längeren Texten zu sehen. Der hier zugrunde gelegte Zufallstext führt nicht zu einer so starken Regelbildung wie ein englischsprachiger Text, dennoch ist der Effekt deutlich wahrzunehmen. (Die durch Gleichung 8.2.1 gegebene Bedingung ist wieder jeweils durch eine senkrechte Linie markiert.)



(a) $\sigma = 6, n = 100000, m = 20, l_{av} = 5.26$, Zufallstext



(b) $\sigma = 6, n = 300000, m = 20, l_{av} = 5.84$, Zufallstext

Abbildung 8.8: Die Wirkung von `RULE_REMOVAL`. Das in Abbildung 8.7 gezeigte Beispiel ist hier neu berechnet unter der Verwendung des `RULE_REMOVAL`-Parameters. Neben der deutlichen Wirkungsweise des Parameters ist anhand der klaren Veränderung der Laufzeit bei höheren Fehlerleveln auch zu erkennen, dass der Einfluss des zerstückelten Suchbereichs bereits in Abbildung 8.7(a) vorhanden ist, dort allerdings nicht so offensichtlich zu Tage tritt, dass eine bessere Filtereffizienz keine bessere Suchzeit bedeutet.

ideal sein, jedoch ist im Allgemeinen eine Unabhängigkeit dieses Wertes von der Textlänge notwendig, da schließlich mit der Textlänge die Anzahl der Regeln und auch die Anzahl längerer Regeln wächst. Sinnvollerweise wird hier als textlängenabhängige Einstellung l_{av} (bzw. als ganzzahliger Wert dann $\lceil l_{av} \rceil$) verwendet. Damit ist dafür gesorgt, dass mit wachsender Textlänge und damit mehr zu betrachtenden Regeln auch der im Durchschnitt pro Regel gegenüber Pk1 erzielte Gewinn ansteigt. Um die größtmögliche Flexibilität zu erhalten, wurde der Parameter `MINIMAL_RULELENGTH` auf die Möglichkeit des automatischen Wählens dieses Wertes l_{av} erweitert.

Abbildung 8.9 greift das Beispiel aus Abbildung 8.7 wieder auf und demonstriert daran den Nutzen der (parametrisierten) Modifikation von GraI, die nur noch Regeln betrachtet, deren Länge mindestens der Durchschnittslänge aller Regeln entspricht.

Um den Gesamteffekt der oben genannten Einflüsse zu eliminieren, reicht es aus, die beiden entsprechenden Modifikationen von GraI zu kombinieren. In Abbildung 8.10 ist das zuvor gezeigte Beispiel aus Abbildung 8.7 wieder aufgegriffen und mit der Kombination dieser beiden Modifikationen neu berechnet.

Anhand eines Textes in englischer Sprache demonstrieren die Experimente in Abbildung 8.11 die für GraI mit den Modifikationen gewonnene lineare Laufzeitabhängigkeit von der Textlänge n. Diese lineare Abhängigkeit bedeutet zugleich auch, dass die Bedingung 8.2.1 für eine gegenüber Pk1 verbesserte Filtereffizienz zugleich die Bedingung für eine verbesserte Laufzeit darstellt.

In der Abbildung 8.11 ist auch die Abhängigkeit der Laufzeit von dem Fehlerlevel (gegeben durch k bei festem m) zu erkennen. Dies soll im Folgenden näher betrachtet werden.

8.2.2.2 Dominanz der Suchphase

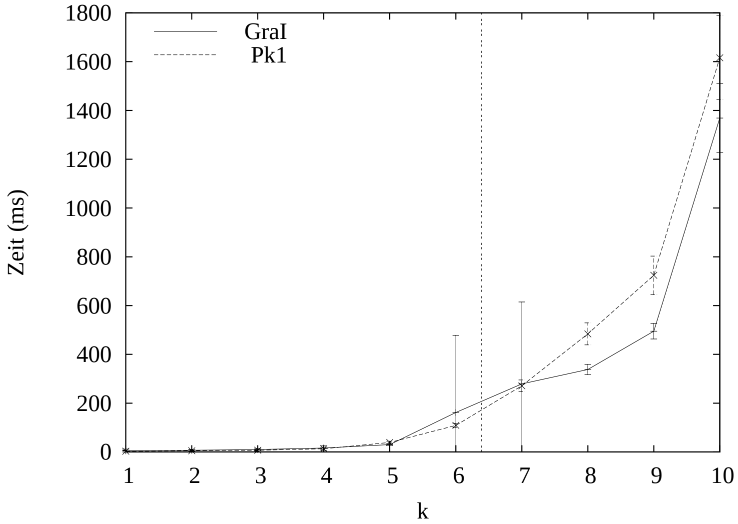
Bei sehr hohen Fehlerleveln muss zwangsläufig irgendwann der gesamte Text überprüft werden, was für den Filteralgorithmus bedeutet, dass seine Laufzeit durch die Laufzeit des Verifikationsalgorithmus bestimmt wird. Aus diesem Grund wird für Filteralgorithmen üblicherweise ein Fehlerlevel α angegeben, bis zu welchem die Suchphase des Filteralgorithmus dominiert, also im Schnitt insgesamt für die Verifikationsphase nur eine in der Textlänge lineare Laufzeit benötigt wird (siehe auch Kapitel 4.1).

Für den Algorithmus Pk1 liegt diese Grenze bei

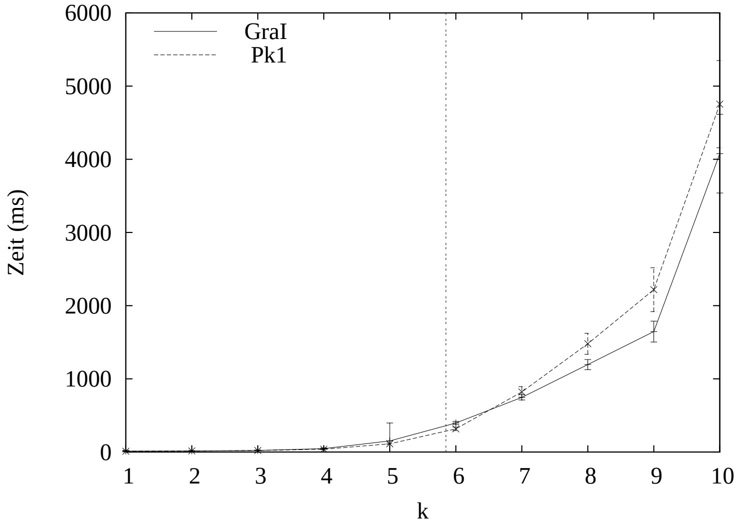
$$\alpha \leq \frac{1}{3log_{\sigma}m} \tag{8.2.5}$$

(siehe Kapitel 4.1.2.3).

Zuvor wurde gezeigt, dass der Algorithmus GraI sich gegenüber Pk1 für die durch Gleichung 8.2.1 gegebene Bedingung verbessert. Diese Verbesserung bringt natürlich auch eine Verschiebung der Grenze für eine in n lineare Laufzeit der Verifikationsphase mit sich. Im Folgenden soll diese Verschiebung qualitativ erfasst werden.

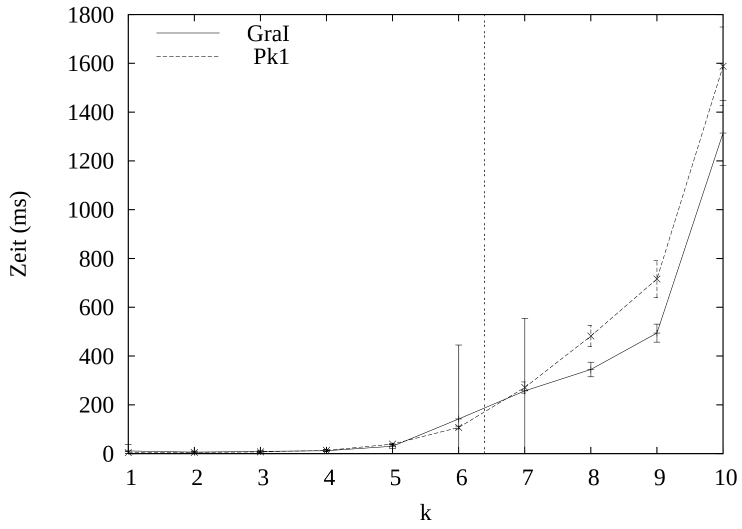


(a) $\sigma = 6, n = 100000, m = 20, l_{av} = 5.26$, Zufallstext

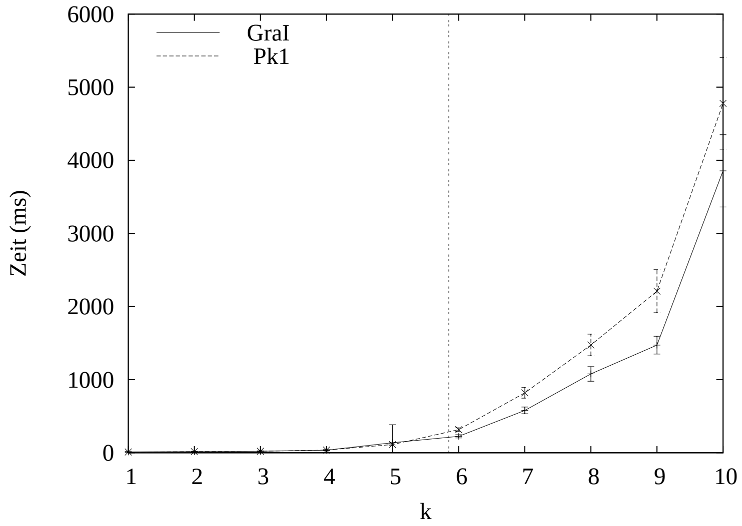


(b) $\sigma = 6, n = 300000, m = 20, l_{av} = 5.84$, Zufallstext

Abbildung 8.9: Die Wirkung von `MINIMAL_RULELENGTH` $= l_{av}$. Das in Abbildung 8.7 gezeigte Beispiel ist mit dem durch `MINIMAL_RULELENGTH` modifizierten Algorithmus hier neu berechnet. Die Wirkung dieser Modifikation greift für höhere Fehlerlevel als dies bei der `RULE_REMOVAL`-Modifikation der Fall ist.



(a) $\sigma = 6, n = 100000, m = 20, l_{av} = 5.26$, Zufallstext



(b) $\sigma = 6, n = 300000, m = 20, l_{av} = 5.84$, Zufallstext

Abbildung 8.10: Die Wirkung der Kombination der beiden Modifikationen `RULE_REMOVAL` und `MINIMAL_RULELENGTH` $= l_{av}$. Je nach Fehlerlevel ist der Einfluss der beiden Modifikationen verschieden stark.

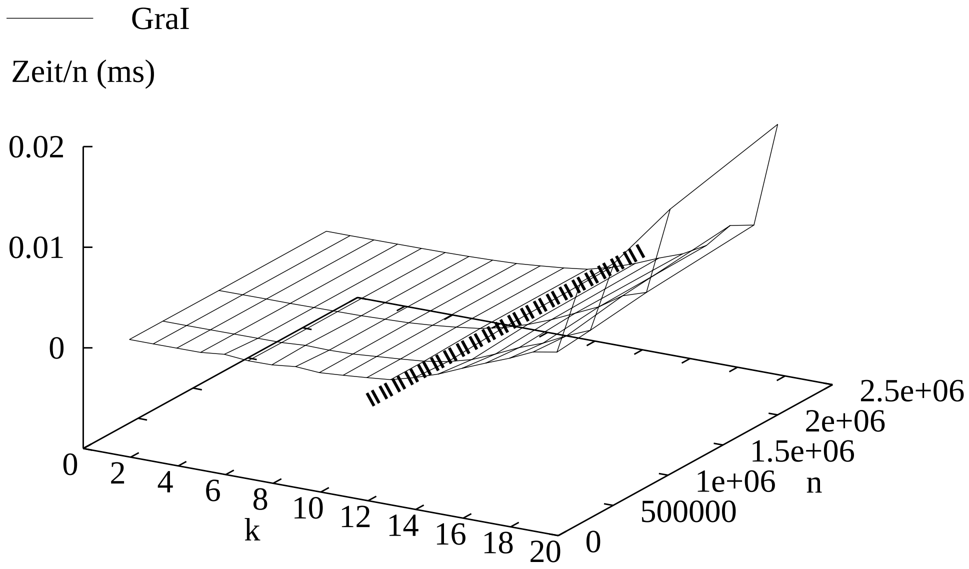


Abbildung 8.11: Die Suche von Pattern der Länge 40 in einem englischen Text mit der durch `RULE_REMOVAL` und `MINIMAL_RULELENGTH` $= l_{av}$ modifizierten Version von GraI. Aufgetragen ist die durchschnittliche Suchzeit geteilt durch die Länge des betrachteten Textes. Das lineare Verhalten der Laufzeit bezüglich n wird dadurch deutlich. Der in der Abbildung eingetragene Balken markiert Bedingung 8.2.5.

Der Ansatz zur Berechnung der Grenze für einen linearen Einfluss der Verifikationsphase bei Pk1 ergibt sich aus der Überlegung, dass eine Verifikation bei jedem exakt gefundenen Teilpattern auftritt und die Gesamtkosten der Verifikation $O(n)$ sein sollen. Mit einem Verifikationsaufwand von jeweils $(m+2k)^2$ und einer Wahrscheinlichkeit für ein exakt gefundenes Teilpattern an einer Textstelle nach Anhang A ergibt sich

$$n\frac{k+1}{\sigma^{\frac{m}{k+1}}}(m+2k)^2 \leq n\,.$$

Es bestehen zwei Möglichkeiten, die Verschiebung dieser Grenze bei GraI zu sehen. Zum einen ist es möglich, den durch die Nutzung der Grammatik entstehenden Vorteil als Verringerung der einzelnen Verifikationskosten zu betrachten. Zum anderen kann dieser Vorteil auch als Verkürzung des zu betrachtenden Textes gesehen werden. Hier wird die zweite Sichtweise verfolgt, die obigen Ansatz dann mit einer durch die Grammatik modifizierten Textlänge n' zu

$$n'\frac{k+1}{\sigma^{\frac{m}{k+1}}}(m+2k)^2 \leq n \tag{8.2.6}$$

verändert.

Zur Bestimmung dieser modifizierten Textlänge n' ist es notwendig, den Einfluss der Grammatik abzuschätzen. Die Frage ist also, welcher Anteil des Textes im Schnitt betrachtet wird, oder andersherum gesehen, welcher Anteil durch die Regeln so abgedeckt ist, dass er nicht mehrfach betrachtet wird.

Da im Algorithmus nur Regeln betrachtet werden, die mindestens die Länge l_{av} haben, brauchen kürzere Regeln bei der Bestimmung des durch Regeln abgedeckten Textanteils nicht berücksichtigt zu werden. Zur Vereinfachung werden hier ebenso längere Regeln nicht berücksichtigt und es wird angenommen, dass der Text genau in Blöcke der Länge l_{av} unterteilt werden kann. Die Wahrscheinlichkeit, dass einer der n/l_{av} Blöcke dann genau einem bestimmten String (also einer Regel der Länge l_{av}) entspricht, ist $p = 1/\sigma^{l_{av}}$, während $q = 1-p$ dann die Wahrscheinlichkeit dafür ist, dass ein Block nicht diesem String entspricht. Tritt ein Block mindestens zweimal auf, so bildet er potenziell eine Regel und braucht daher prinzipiell auch nur beim ersten Auftreten verifiziert zu werden. Es ist jetzt von Interesse, welcher Anteil des Textes dann so nicht mehr betrachtet werden muss.

Es sei nun der erste der Blöcke betrachtet. Dann ist

$$\binom{\frac{n}{l_{av}}-1}{i} p^i q^{(\frac{n}{l_{av}}-1)-i}$$

die Wahrscheinlichkeit dafür, dass i der folgenden $n/l_{av}-1$ Blöcke mit dem ersten Block identisch sind. Unter Berücksichtigung aller möglichen Anzahlen von folgenden identischen Blöcken, kann für den ersten Block berechnet werden, wie oft dieser im Mittel noch auftritt:

$$\sum_{i=1}^{\frac{n}{l_{av}}-1} i\binom{\frac{n}{l_{av}}-1}{i} p^i q^{(\frac{n}{l_{av}}-1)-i}\,.$$

Da die hier auftretende Verteilung genau die Binomialverteilung darstellt, entspricht dieser Mittelwert dem Erwartungswert der Binomialverteilung:

$$\left(\frac{n}{l_{av}} - 1\right) p\,.$$

Bei der Betrachtung des zweiten und aller folgenden Blöcke muss zudem noch berücksichtigt werden, dass zum einen die Anzahl der noch verbleibenden möglichen Blöcke sinkt, und dass zum anderen der jeweils betrachtete Block nicht identisch ist mit den vorherigen Blöcken. So ergibt sich

$$\left(\frac{n}{l_{av}} - j\right) pq^{j-1}$$

als Mittelwert für die Anzahl der zum jeweils betrachteten Block j identischen Blöcke.

Aus der Summe der Anzahlen der jeweils identischen Blöcke ergibt sich durch Multiplikation mit der Blocklänge l_{av} der Anteil des Textes $\bar{n}$, der aufgrund der Abdeckung von Regeln nicht betrachtet werden muss:

$$\begin{aligned}
\bar{n} &= l_{av} \sum_{j=1}^{\frac{n}{l_{av}}-1} \left(\frac{n}{l_{av}} - j\right) pq^{j-1} \\
&= l_{av} p \left(\frac{n}{l_{av}} \sum_{k=0}^{\frac{n}{l_{av}}-2} q^k - \frac{1}{q} \sum_{k=1}^{\frac{n}{l_{av}}-1} kq^k \right) && (8.2.7) \\
&= l_{av} p \left(\frac{n}{l_{av}} \frac{1 - q^{\frac{n}{l_{av}}-1}}{1-q} - \frac{1}{q} \frac{q - \frac{n}{l_{av}} q^{\frac{n}{l_{av}}} + \left(\frac{n}{l_{av}} - 1\right) q^{\frac{n}{l_{av}}+1}}{(1-q)^2} \right) \\
&= n\left(1 - q^{\frac{n}{l_{av}}-1}\right) - l_{av} \frac{1 - \frac{n}{l_{av}} q^{\frac{n}{l_{av}}-1} + \left(\frac{n}{l_{av}} - 1\right) q^{\frac{n}{l_{av}}}}{p} && (8.2.8)
\end{aligned}$$

Eine radikale Vereinfachung (und Verkleinerung des Wertes) lässt sich durch Annahme eines unendlich langen Textes bei gleichzeitiger Fixierung von l_{av} erreichen:

$$\bar{n} = n - \frac{1}{p} l_{av} = n - l_{av} \sigma^{l_{av}}\,. \tag{8.2.9}$$

Mit $\bar{n}$, dem Anteil des Textes, der aufgrund der Grammatiknutzung nicht betrachtet wird, ergibt sich eine modifizierte Textlänge von $n' = n - \bar{n}$. Mit dieser modifizierten Textlänge kann die Gleichung 8.2.6 so umgeformt werden, dass sich eine Grenze für den Fehlerlevel ergibt, bis zu welchem die Suchphase dominiert.

Aus

$$(n-\bar{n})\frac{k+1}{\sigma^{\frac{m}{k+1}}}(m+2k)^2 \leq n$$

ergibt sich für $k \leq m-1$ und $k \leq m/2$ (jeweils passend ersetzt)

$$(1-\frac{\bar{n}}{n})\frac{m^3}{\sigma^{\frac{m}{k+1}}} \leq 1\,.$$

Mit $\frac{m}{k+1} \approx \frac{1}{\alpha}$ ergibt sich

$$(1-\frac{\bar{n}}{n})m^3 \leq \sigma^{\frac{1}{\alpha}}\,,$$

was aufgelöst nach α

$$\begin{aligned} \alpha &\leq \frac{1}{\log_\sigma\left((1-\frac{\bar{n}}{n})m^3\right)} \\ &= \frac{1}{\log_\sigma(1-\frac{\bar{n}}{n}) + 3\log_\sigma m} \end{aligned} \tag{8.2.10}$$

bedeutet.

Bei der Berechnung dieses Wertes ist es jedoch problematisch, $\bar{n}$ aus Gleichung 8.2.8 zu verwenden, da die beschränkte Fließkommadarstellung im Computer schnell dazu führt, dass die Potenzen von q Null werden, ohne wirklich Null zu sein. Zur praktischen Berechnung muss also auf die Form aus Gleichung 8.2.7 zurückgegriffen werden. Es besteht auch die Möglichkeit, auf die ein wenig ungenauere Form aus Gleichung 8.2.9 auszuweichen, wobei sich dann Gleichung 8.2.10 weiter vereinfachen lässt zu:

$$\alpha \leq \frac{1}{\log_\sigma \frac{l_{av}\sigma^{l_{av}}}{n} + 3\log_\sigma m}\,. \tag{8.2.11}$$

Es ist zu beachten, dass an dieser Stelle nicht einfach die praktisch ermittelte durchschnittliche Regellänge l_{av} verwendet werden kann, sondern vielmehr ist es hier wichtig, den theoretischen Wert zu nutzen. Bei stark strukturierten Texten entstehen längere Regeln, die zu einer deutlich längeren durchschnittlichen Regellänge und auch zu einem größeren Anteil von durch Regeln abgedecktem Text führen. Weicht die theoretische durchschnittliche Regellänge stark von der praktisch ermittelten ab, so sind die Wahrscheinlichkeiten für die einzelnen Blöcke nicht mehr gleich und obige Berechnung damit so nicht gültig.

Zu einem weiteren direkten Vergleich von Algorithmus GraI mit Algorithmus Pk1 sei nun der Bereich näher betrachtet, in dem die Suchphase dominant ist, also Bedingung 8.2.5 gilt. Es gilt zudem, dass sich GraI unter Bedingung 8.2.1 in jedem Fall besser verhält als Pk1. Werden diese beiden Bedingungen kombiniert, so ergibt sich ein Intervall für den Fehlerlevel α, in dem sowohl die Suchphase dominiert, als auch GraI ein besseres Laufzeitverhalten hat:

$$\frac{2}{l_{av}+1} \leq \alpha \leq \frac{1}{3\log_\sigma m}\,. \tag{8.2.12}$$

Daraus ergibt sich als Bedingung für die Existenz dieses Intervalls die Ungleichung

$$l_{av} \geq 6\log_\sigma m - 1\,. \tag{8.2.13}$$

Dadurch dass die durchschnittliche Regellänge bei wachsender Textlänge auch größer wird, ist diese Bedingung prinzipiell eigentlich immer zu erreichen, jedoch sind dafür unter Umständen (gerade bei rein zufälligen Textdaten) sehr lange Texte notwendig. Dies wird zum Beispiel bei der Betrachtung eines Zufallstextes auf einem Alphabet der Größe $\sigma = 80$ deutlich. Selbst bei einer Textlänge von 10000000 Symbolen, wird theoretisch $l_{av} = 3.12$ erreicht, während dann bei der Patterngröße $m = 20$ so gerade mit $6\log_\sigma m - 1 = 3.10$ der gegebene Wert unterschritten wird. Anders ist dies bei stärker strukturierten Texten, die in der Praxis eine größere durchschnittliche Regellänge l_{av} besitzen. So ist beispielsweise bei einem englischen Text mit einem Alphabet der Größe $\sigma = 28$ und Pattern der Länge $m = 40$: $6\log_\sigma m - 1 = 5.64$. Dieser Wert wird vom gemessenen Wert $l_{av} = 8.63$ klar überschritten. Abbildung 8.12 zeigt anhand zweier Beispiele die Situationen, in denen dieses Intervall für α einmal existiert und einmal nicht existiert.

8.2.2.3 Die Vorberechnungsphase

In Kapitel 7.2.2 ist die Vorberechnungsphase von Algorithmus GraI beschrieben.

Die Vorberechnungsphase für den Text (Kapitel 7.2.2.1) gliedert sich in zwei Teile: die Erzeugung der Grammatik und das Berechnen der benötigten zusätzlichen Informationen. Die Grammatik des Textes wird vollständig in $O(n)$ berechnet (siehe Kapitel 6.3.4). Die zusätzlichen Informationen werden während eines Durchlaufs durch die Grammatik gewonnen, was also zu einem Aufwand von $O(n)$ führt.

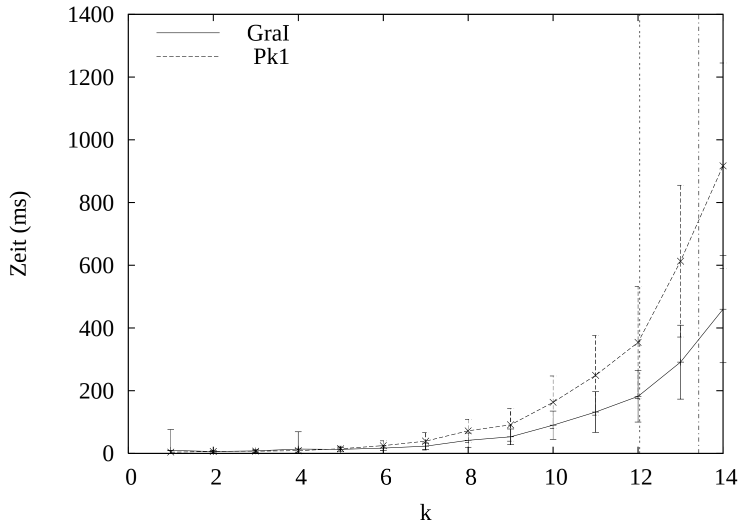
Die Vorberechnungsphase für das Pattern (Kapitel 7.2.2.2) benötigt nur $O(m)$ Zeit für die Zerlegung des Patterns.

8.2.3 Der Effekt der Minimumswahl

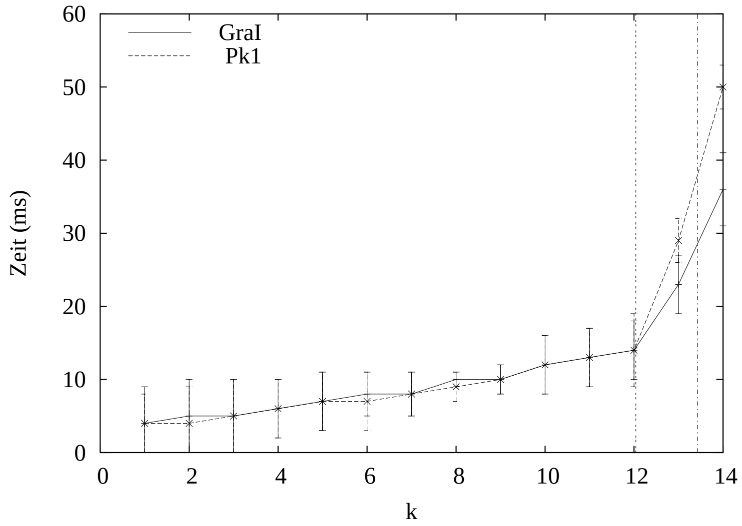
Der Parameter `MINIMUM_OPT` (siehe Kapitel 7.4.3) zielt darauf ab, die Laufzeit des Algorithmus im Falle des gleichzeitigen Durchlaufs durch die Positionslisten mehrerer Regeln zu verbessern.

Eine Betrachtung der Wirkung dieser Einstellung im Vergleich zum Algorithmus ohne diese Einstellung, lässt drei verschiedene Bereiche erwarten:

1. Je kleiner der Fehlerlevel, desto länger sind die exakt gesuchten Teilpattern. Dies wiederum bedeutet ein selteneres Auftreten der Teilpattern und damit eine geringere Chance, von Regeln abgedeckt zu werden. Der Parameter `MINIMUM_OPT` kann



(a) $\sigma = 28, n = 200000, m = 40, l_{av} = 8.63$, englischer Text



(b) $\sigma = 28, n = 200000, m = 40, l_{av} = 3.45$, Zufallstext

Abbildung 8.12: Bereich dominierender Suchphase. Die linke Senkrechte markiert die Bedingung für eine insgesamt (in n) lineare Verifikationszeit (Bedingung 8.2.5) beim Algorithmus Pk1, während die rechte Senkrechte die entsprechend für GraI modifizierte Bedingung (Gleichung 8.2.10) aufzeigt. In (a) ist Bedingung 8.2.13 erfüllt, in (b) hingegen nicht.

aber einen positiven Effekt nur haben, wenn mehrere Regeln gleichzeitig betrachtet werden, die auch exakt gefundene Teilpattern (oder gar ganze Verifikationsbereiche) abdecken. So ist durch den Mehraufwand für `MINIMUM_OPT` ohne gleichzeitigen positiven Nutzen eine höhere Laufzeit im betrachteten Fall zu erwarten.

2. Werden durch den wachsenden Fehlerlevel die exakt gesuchten Teilpattern kleiner, so ist es häufiger zu erwarten, dass Regeln diese Teilpattern vollständig abdecken und so die entsprechenden Positionslisten der Regeln durchlaufen werden. Es ist also damit zu rechnen, dass der generelle Mehraufwand zum Teil ausgeglichen wird und dass sogar ein positiver Effekt verzeichnet werden kann.

3. Wird ein Fehler von $k = m/2 - 1$ erreicht, so sind die exakt gesuchten Pattern nur noch von der Länge 2, was zugleich der Länge der kleinsten Regeln entspricht. Wird der Text als unendlich lang angenommen, so wird schließlich jedes mögliche Pattern der Länge 2 von einer Regel abgedeckt sein. Bei den betrachteten endlichen Texten ist dieser Fall je eher zu erwarten, je kleiner σ und je größer n ist. Exakt gefundene Subpattern sind also so gut wie immer von Regeln abgedeckt. Beim gleichzeitigen Durchlauf von Positionslisten mehrerer Regeln kann so ein deutlicher positiver Effekt des Parameters `MINIMUM_OPT` erwartet werden.

Es ist möglich, eine sehr grobe Einschätzung dafür zu geben, wann der zweite dieser Bereiche in etwa erreicht sein kann. Dazu sei der Abstand $rdist$ zwischen zwei Regeln betrachtet. Ist dieser jeweils größer als der Abstand zwischen zwei exakt gefundenen Teilpattern, so ist insgesamt zu erwarten, dass Teilpattern von Regeln abgedeckt werden. Mit einem durchschnittlichen Abstand zwischen zweien dieser exakten Matchings (nach Anhang A) von $dist = \frac{\sigma^{m/(k+1)}}{k+1}$ ergibt sich also als grobe Bedingung für den zweiten Bereich

$$rdist > dist \Leftrightarrow rdist > \frac{\sigma^{\frac{m}{k+1}}}{k+1} \,. \tag{8.2.14}$$

Zur Bestimmung des Abstands $rdist$ zweier Regeln sei vereinfachend angenommen, der Text sei unendlich lang und bei den gerade betrachteten Regeln handele es sich um zwei bestimmte Strings der Länge l_{av} (siehe Gleichung 8.2.4). Zudem sei der erste dieser Strings „festgehalten" und so ist es noch nötig, den Abstand rd zum zweiten String zu bestimmen, um den Gesamtabstand

$$rdist = l_{av} + rd \tag{8.2.15}$$

zwischen zwei Regeln zu erhalten (siehe Abbildung 8.13).

Zur Bestimmung von rd sei nun weiter $p = 1/\sigma^{l_{av}}$ die Wahrscheinlichkeit dafür, dass an einer Textstelle der String der Länge l_{av} existiere und $q = 1-1/\sigma$ die Wahrscheinlichkeit dafür, dass an einer Stelle dieser String nicht beginnt (nur bewertet durch das erste Symbol des Strings). Damit ist pq^i für $i \geq 0$ die Wahrscheinlichkeit dafür, dass $rd = i$ ist (q als Faktor für jede Stelle, an der der String nicht ist und p als Faktor für das Vorhandensein).

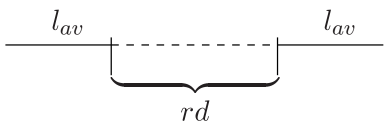


Abbildung 8.13: Ausgangsszenario zur groben Bestimmung eines Abstands zwischen Regeln.

Durch Summieren aller dieser Wahrscheinlichkeiten, jeweils gewichtet mit dem Abstand, kann so ein Durchschnittswert für rd ermittelt werden, wenn diese Summe noch zur Normierung durch die Summe der Wahrscheinlichkeiten geteilt wird [57]:

$$rd = \frac{p + \sum_{i=1}^{\infty} ipq^i}{\sum_{i=0}^{\infty} pq^i} = \frac{1 + \sum_{i=1}^{\infty} iq^i}{\sum_{i=0}^{\infty} q^i}$$

Die Geometrischen Reihen lassen weitere Vereinfachung zu:

$$rd = \frac{1 + \frac{q}{(1-q)^2}}{\frac{1}{1-q}} = \frac{(1-q)^2 + q}{1-q} \,.$$

Mit $q = 1 - \frac{1}{\sigma}$ ergibt sich

$$rd = \frac{1}{\sigma} + (\sigma - 1)\,.$$

Mit Gleichung 8.2.15 ergibt sich als sehr grob abgeschätzter Gesamtabstand $rdist$ zwischen zwei Regeln

$$rdist = \frac{1}{\sigma} + \sigma - 1 + l_{av}\,. \tag{8.2.16}$$

Eingesetzt in Bedingung 8.2.14 führt dies zur der Bedingung

$$\frac{1}{\sigma} + \sigma - 1 + l_{av} > \frac{\sigma^{\frac{m}{k+1}}}{k+1}\,, \tag{8.2.17}$$

aus der sich jeweils ein entsprechender Wert für k bestimmen lässt.

Abbildungen 8.14 und 8.15 zeigen beispielhaft Verläufe, an denen der zweite der oben genannten Bereiche klar zu erkennen ist. Auch wird dabei nochmals deutlich, dass die in Gleichung 8.2.16 berechnete Abschätzung keine klare Abgrenzung darstellt, sondern vielmehr allenfalls als grobe Richtlinie verstanden werden muss. Die Suchzeiten bei sehr wenigen Fehlern sind in diesen Abbildungen so gering, dass der Mehraufwand im ersten der Bereiche nicht zu erkennen ist. Auch der dritte Bereich hebt sich hier nicht deutlich hervor. Anders ist dies bei den in den Abbildungen 8.16 und 8.17 gegebenen Beispielen, wo jeweils der genannte dritte Bereich eindeutig erkennbar ist. Jedoch ist auch hier der erste Bereich nicht wahrzunehmen. Dazu ist ein Beispiel wie in Abbildung 8.18 notwendig, das eine höhere Zeitauflösung aufzeigt. Schön ist dort der Effekt zu erkennen, dass der zweite Bereich sehr weit nach rechts verschoben ist und sich so direkt dem Einfluss des dritten Bereichs unterordnen muss.

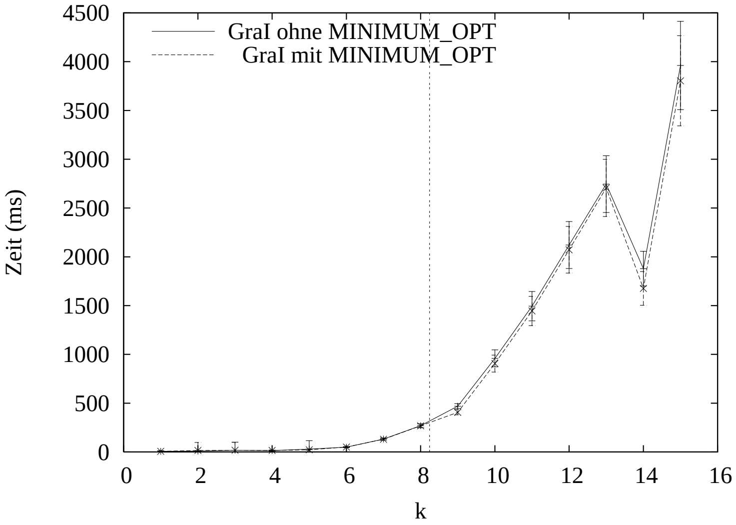


Abbildung 8.14: Die Auswirkungen des Parameters `MINIMUM_OPT`. Bei einer auf einem Zufallstext durchgeführten Suche ($\sigma = 4$, $n = 100000$, $m = 30$) ergibt sich bei einer errechneten durchschnittlichen Regellänge von $l_{av} = 6.48$ für die Fehlerzahl k nach Gleichung 8.2.17 die Bedingung $k \geq 8.24$ (mit der senkrechten Linie markiert).

Bei den bisherigen Betrachtungen zur Minimumswahl wurde der Algorithmus in der in Kapitel 7.2 vorgestellten Grundform betrachtet, für welche dieser Parameter `MINIMUM_OPT` auch eingeführt wurde.

In Kapitel 8.2.2 erwiesen sich im allgemeinen Fall jedoch zwei kleine Modifikationen des Algorithmus als notwendig, um für die Dominanz der Suchphase einen textlängenunabhängigen Grenzwert (für den Fehlerlevel) angeben zu können. Eine der Modifikationen setzte die Mindestlänge der überhaupt betrachteten Regeln auf die Durchschnittslänge aller Regeln. Diese Modifikation erweist sich als äußerst ungünstig für den Parameter `MINIMUM_OPT`. Die durch den Parameter erzielte Verbesserung basiert auf der gleichzeitigen Behandlung mehrerer Regeln im Teilalgorithmus *CheckVL* in Kapitel 7.2.5. Das Auftreten mehrerer Regeln in dem entsprechenden Abstand ist insbesondere bei kurzen Regeln zu erwarten, die durch die Modifikation allesamt nicht mehr betrachtet werden. Zudem ist die minimal mögliche Regellänge 2, die maximal mögliche Länge aber unbe-

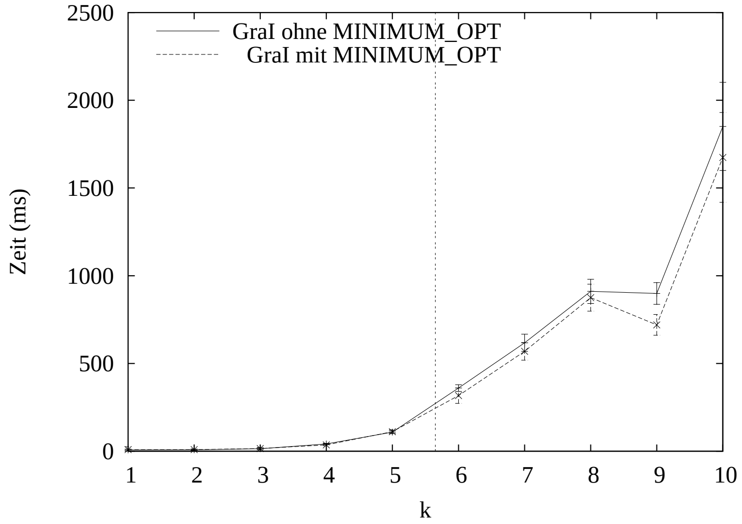


Abbildung 8.15: Die Auswirkungen des Parameters `MINIMUM_OPT`. Bei einem im Vergleich zu Abbildung 8.14 kürzeren Pattern ($\sigma = 4$, $n = 100000$, $m = 20$, $l_{av} = 6.48$) verändert sich auch der durch Bedingung 8.2.17 gegebene Grenzwert für k zu $k \geq 5.65$.

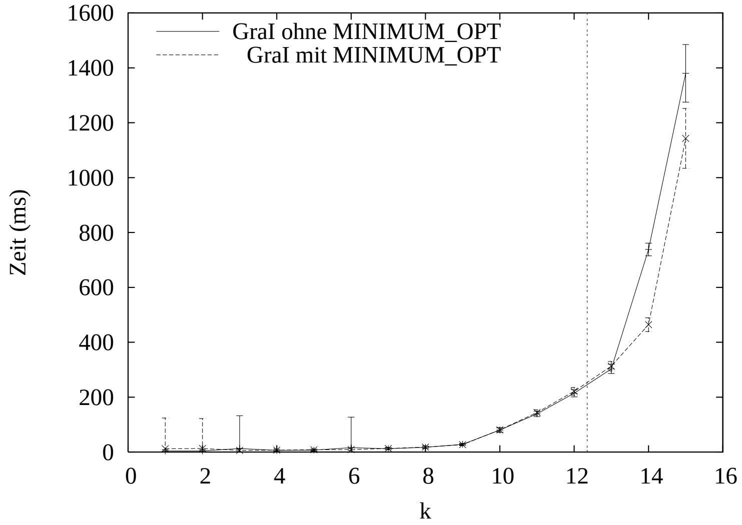


Abbildung 8.16: Die Auswirkungen des Parameters `MINIMUM_OPT`. Die genannten ersten und zweiten Bereiche sind ist bei einer Suche im Zufallstext mit $\sigma = 10$, $n = 100000$, $m = 30$ und $l_{av} = 4.09$ nicht deutlich auszumachen. Der durch die Bedingung 8.2.17 gegebene Wert für k ($k \geq 12.35$) ist mit einer Senkrechten markiert.

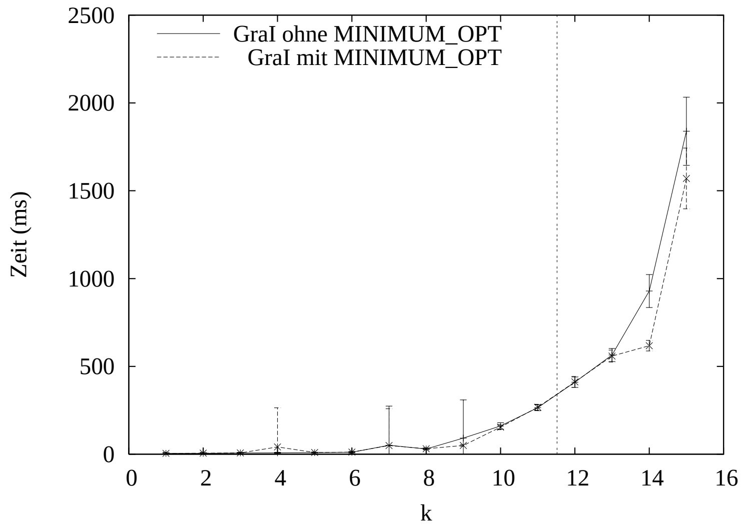


Abbildung 8.17: Die Auswirkungen des Parameters `MINIMUM_OPT`. Im Gegensatz zum Beispiel in Abbildung 8.16 liegt hier ($\sigma = 8$, $n = 100000$, $m = 30$, $l_{av} = 4.5$) der durch Bedingung 8.2.17 gegebene Grenzwert ($k \geq 11.52$) deutlich weiter entfernt vom Anfang des dritten Bereichs.

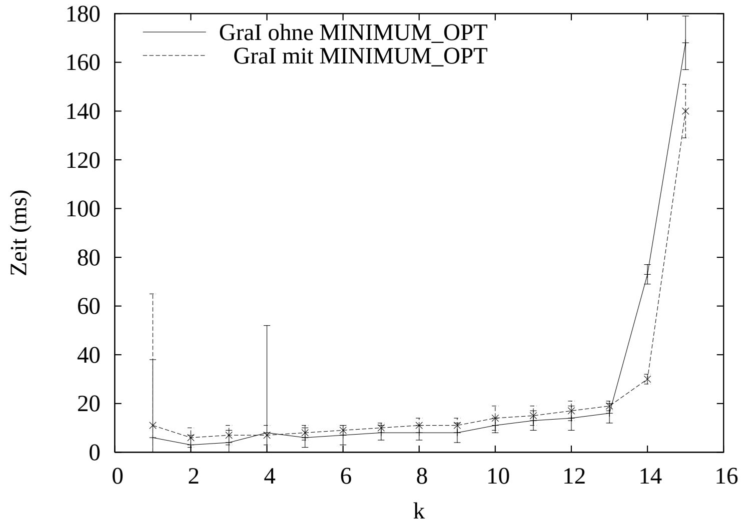


Abbildung 8.18: Die Auswirkungen des Parameters `MINIMUM_OPT`. Der durch Bedingung 8.2.17 gegebene Grenzwert für den zweiten Bereich liegt hier ($\sigma = 80$, $n = 300000$, $m = 30$, $l_{av} = 2.58$) mit $k \geq 17.02$ klar oberhalb des Beginns des dritten Bereichs ($k \geq m/2 - 1$). Der erste Bereich ist deutlich zu erkennen und reicht bis zum Beginn des dritten Bereichs.

schränkt. Dies bedeutet, dass von weit über der Hälfte der Regeln zu erwarten ist, dass sie kürzer sind als die durchschnittliche Regellänge.

In der Praxis bleibt beim modifizierten Algorithmus GraI von dem Einfluss des Parameters `MINIMUM_OPT` nur noch der oben genannte erste Bereich erhalten, welcher sich allerdings negativ auswirkt. Abbildung 8.19, in der das Beispiel aus Abbildung 8.15 wieder aufgegriffen wird, verdeutlicht dies anschaulich. Die dargestellten Kurven zeigen Suchen unter Anwendung des modifizierten Algorithmus GraI. Ein positiver Effekt ist nicht mehr wahrzunehmen, jedoch ist (wenn auch nur schwach) erkennbar, dass die Kurve mit der Verwendung des Parameters `MINIMUM_OPT` immer minimal oberhalb der anderen Kurve verläuft.

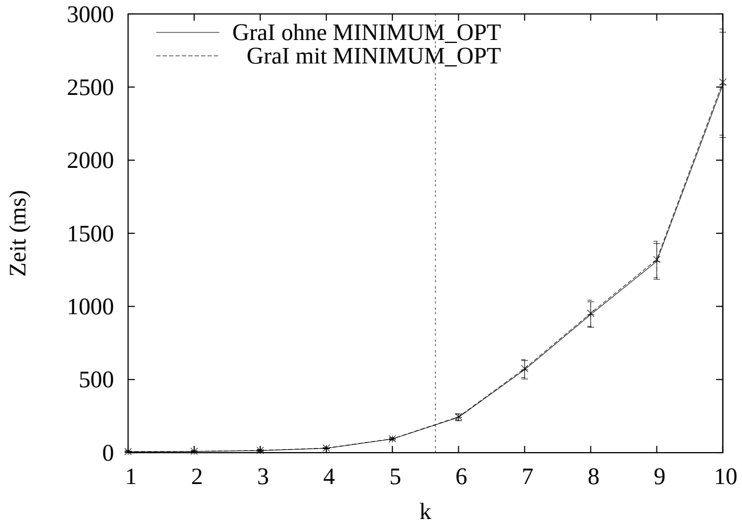


Abbildung 8.19: Die Auswirkungen des Parameters `MINIMUM_OPT` beim modifizierten Algorithmus am Beispiel eines Zufallstextes mit $\sigma = 4$, $n = 100000$, $m = 20$ und $l_{av} = 6.48$ (vgl. Abbildung 8.14). Der durch Bedingung 8.2.17 gegebene Grenzwert für k ist durch eine senkrechte Linie markiert, jedoch ist kein positiver Einfluss mehr wahrzunehmen, sondern nur noch der Mehraufwand.

8.2.4 GraI im Vergleich

Bei einem Vergleich verschiedener Online-Algorithmen zum Approximate String Matching zeigte Navarro in [84, 85] auf, dass das generelle Grundprinzip von Algorithmus Pk1 bei $m > \sigma$ für Fehlerlevel im Dominanzbereich der Suchphase (also für $\alpha < \frac{1}{3 \log_\sigma m}$) den schnellsten Algorithmus ermöglicht (dort kam eine bitparallele Implementierung zum Einsatz).

Der Algorithmus GraI nutzt die Möglichkeiten von Vorberechnungen, um unter den oben genannten Bedingungen (siehe Kapitel 8.2.2.1 und 8.2.2.2) eine Verbesserung von Pk1 zu erreichen.

An dieser Stelle soll der Algorithmus GraI praktisch in Relation zu einem anderen Offline-Algorithmus gesetzt werden. Ausgewählt wurde der SLEQ-Algorithmus von Sutinen und Tarhio [119], der auch auf dem Prinzip der exakten Suche basiert und von einer $k+s$ Zerlegung der zu findenden Approximate Matchings ausgeht (siehe dazu auch Kapitel 8.1.1).

Der SLEQ-Algorithmus verwendet bei der Suche einen q-Gramm-Index des Textes, wobei die q-Gramme dem Text jeweils im Abstand von $h \geq q$ entnommen werden. Prinzipiell dominiert bei SLEQ die Suchphase für $\alpha < 1/\log_\sigma m$. Jedoch wird der praktisch nutzbare Bereich, in dem der Fehlerlevel liegen kann, klar eingeschränkt durch die Parameter des Algorithmus [119]. Aus der Annahme der bei einer $k+s$ Zerlegung idealen Schrittweite von $h = \lfloor (m-k-q+1)/(k+s) \rfloor$ [119], kann bei gegebenem h der Parameter s als $s = (m-k-q+1)/h - k$ berechnet werden. Für die Aufteilung in exakte Suche, muss $s \geq 1$ gelten, woraus sich als Bedingung für die gewählte Schrittweite h

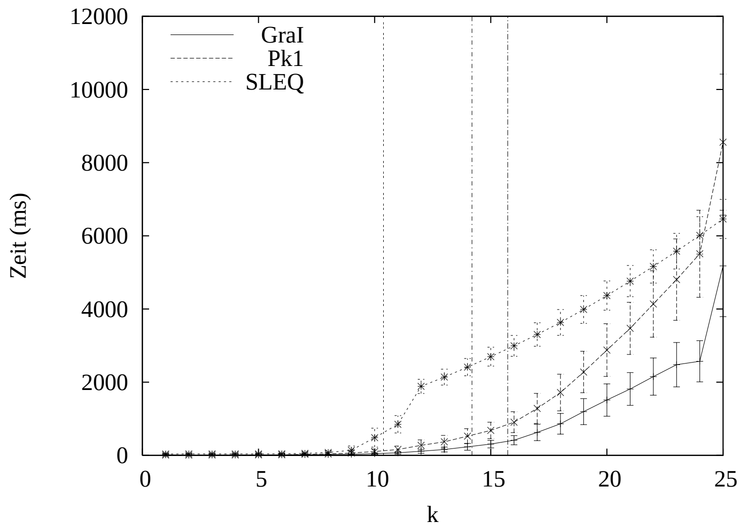
$$q \leq h \leq \frac{m-q-k+1}{k+1} \tag{8.2.18}$$

ergibt. Für ein festes q ergibt sich daraus eine Bedingung für die Fehlerzahl k:

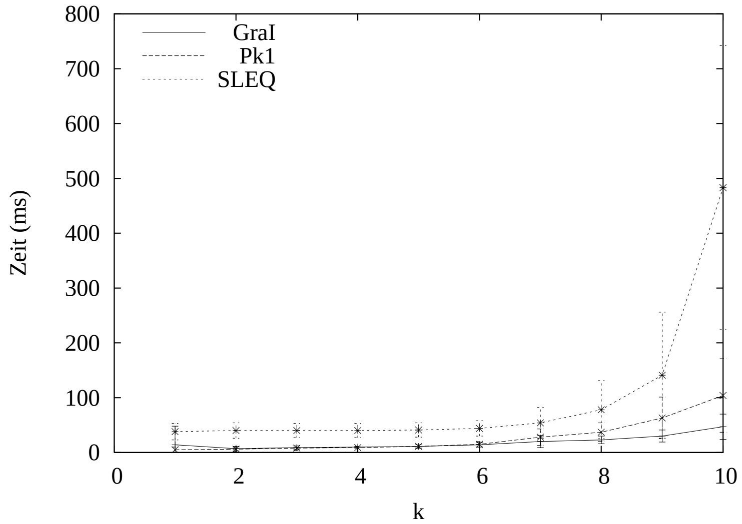
$$k \leq \frac{m-2q-1}{q+1}\,. \tag{8.2.19}$$

Abbildung 8.20 zeigt anhand eines Beispiels der Suche in englischem Text das Verhalten der Algorithmen GraI und Pk1 im Vergleich zu SLEQ. Die dort verwendete Patterngröße von $m = 50$ führt bei gewähltem $q = 3$ zu einer nach Gleichung 8.2.19 bestimmten Fehlerzahl von $k \leq 10.75$. Aus dieser Fehlerzahl ergibt sich mit Gleichung 8.2.18 die ideale Schrittweite $h = 3$ für die Entnahme der q-Gramme aus dem Text. In Abbildung 8.20(a) ist bis zu einem Fehlerlevel von $\alpha = 1/2$ der Gesamtverlauf der Laufzeiten eingetragen. Auch wenn die prinzipielle Grenze für die Dominanz der Suchphase für SLEQ hier bei $\alpha \leq 1/\log_\sigma m$, also $k \leq 42.59$ liegt, so ist doch der interessante Bereich der, in dem bei GraI die Suchphase dominiert. Zur besseren Unterscheidbarkeit der verschiedenen Kurven wurde dieser Bereich in Abbildung 8.20(b) nur eingeschränkt bis $k = 10$ dargestellt. Diese Grenze stellt auch zugleich die für SLEQ nach Gleichung 8.2.19 maximale sinnvolle Fehlerzahl dar.

Auffällig ist in Abbildung 8.20(b) der bei kleinen Fehlerzahlen (bei denen wenig Verifikationen stattfinden) relativ konstant höhere Aufwand bei SLEQ. Der Grund dafür liegt in der grundsätzlich zweistufigen Suche des Algorithmus. Auch wenn die Suche der q-Gramme im Index schnell ist, so muss doch für die jeweilige Überprüfung, ob s q-Gramme in einem bestimmten Bereich liegen, immer noch eine eingeschränkte Suche der jeweiligen q-Gramme vorgenommen werden.



(a)



(b)

Abbildung 8.20: Filteralgorithmen im Vergleich. In (a) sind die Suchzeiten von Pattern der Länge $m = 50$ in einem englischen Text ($\sigma = 28, n = 200000, l_{av} = 8.63$) bis zu einem Fehlerlevel von $\alpha = 1/2$ eingetragen. Die linke Senkrechte markiert die Fehlerzahl, ab der GraI im Schnitt besser ist als Pk1 (Bedingung 8.2.1). Für Werte von k links von der mittleren bzw. der rechten senkrechten Linie dominiert die Suchphase bei Pk1 (Gleichung 8.2.5) bzw. bei GraI (Gleichung 8.2.10). In (b) ist ein Ausschnitt aus (a) für kleinere Fehlerzahlen dargestellt.

Bei Algorithmus GraI hingegen ist der Aufwand der Suchphase gegenüber Pk1 gerade bei kleinen Fehlerzahlen nicht wirklich verändert, was dazu führt, dass die schnelle Suche mit dem Sunday-Algorithmus hier GraI und Pk1 gegenüber SLEQ besser macht.

Für das Beispiel in Abbildung 8.20 war die Größe der q-Gramme bereits vorher gewählt. Die optimale Größe für q ist bei SLEQ jedoch $q = \log_\sigma m$, was in genanntem Beispiel dann $q = 1.17$ (praktisch also einem Index mit einzelnen Buchstaben) entspräche und nicht dem gewählten $q = 3$.

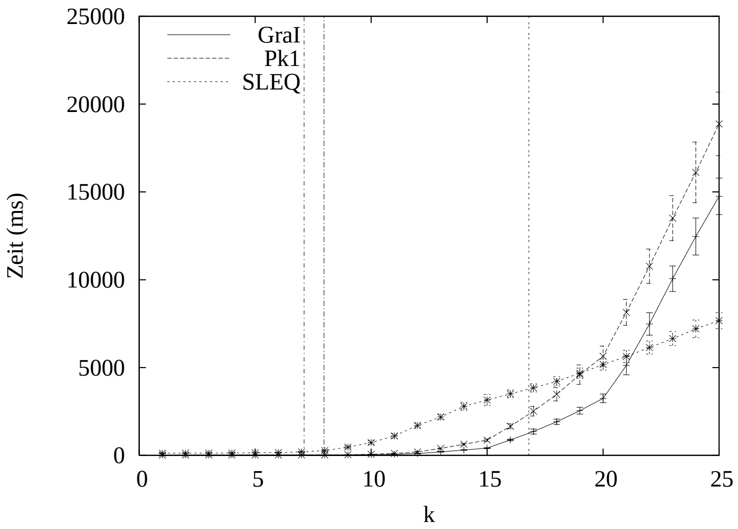
Abbildung 8.21 zeigt ein Beispiel der Suche in einem Zufallstext eines Alphabetes der Größe $\sigma = 4$. Die Größe der gesuchten Pattern ist dabei $m = 64$, was eine ideale q-Gramm Größe von $q = 3$ zur Folge hat. Die q-Gramme in diesem Beispiel wurden mit der Schrittweite $h = 3$ entnommen, die sich nach Gleichung 8.2.18 ergibt. Die prinzipielle Grenze für die Dominanz der Suchphase für SLEQ liegt hier bei $\alpha \leq 1/\log_4 64$, also $k \leq 21.33$, wobei durch die Parameter nach Gleichung 8.2.19 $k = 14.25$ als obere Grenze gilt.

Anders als bei dem zuvor gezeigten Beispiel ist in Abbildung 8.21 das durch Gleichung 8.2.12 gegebene Intervall nicht vorhanden (mit $l_{av} = 6.62$ ist hier Bedingung 8.2.13 nicht erfüllt), welches den Bereich beschreibt, in dem GraI besser ist als Pk1 und zugleich noch die Suchphase dominant ist. In Abbildung 8.21(a) ist deutlich zu erkennen, dass GraI – wie bereits in Kapitel 8.2.1 diskutiert – auch schon für kleinere k als dem theoretischen Wert besser ist als Pk1. Doch auch diese „Verschiebung“ reicht nicht bis in den Bereich, in dem für GraI noch die Suchphase dominiert (siehe Abbildung 8.21(b)).

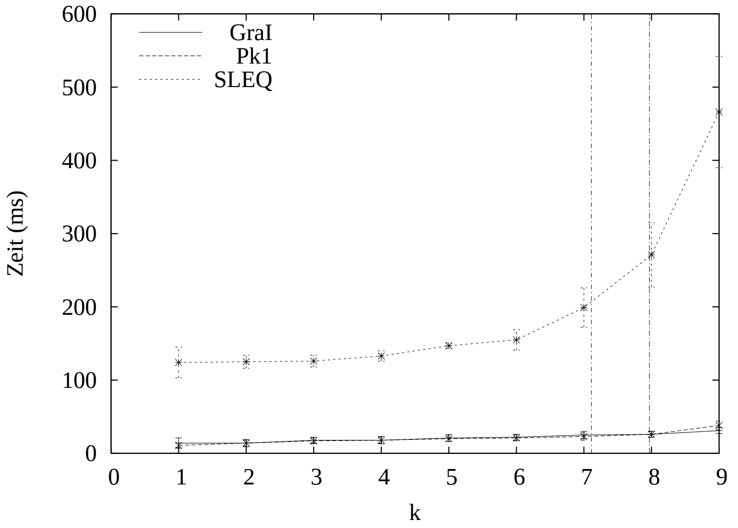
Auffällig bei diesem Beispiel sind wieder die grundsätzlich höheren Kosten der Suche bei SLEQ, die sich insbesondere bei geringen Fehlerzahlen bemerkbar machen.

Abbildung 8.22 greift die Parameter des Beispiels aus Abbildung 8.21 mit zufälligem Text wieder auf, verwendet jedoch diesmal einen DNS-Datensatz (beschränkt auf dieselbe Länge). Da das Alphabet des DNS-Datensatzes im wesentlichen auch nur aus vier Symbolen besteht, weist die Betrachtung der sich ergebenden durchschnittlichen Regellänge, die um 0.17 größer ist als beim Zufallstext, schon darauf hin, dass sich die Ergebnisse nicht allzu stark von denen des Zufallstextes unterscheiden können. Die veränderte durchschnittliche Regellänge verschiebt dann auch nur minimal (um 0.37) die Grenze nach links, ab der GraI theoretisch ein besseres Verhalten hat als Pk1.

Auffällig ist jedoch gerade bei SLEQ ein schlechteres Verhalten insbesondere bei geringen Fehlerzahlen. Tatsache ist, dass die absoluten Suchzeiten auf dem DNS-Datensatz bei jedem der Algorithmen etwas höher liegen, was an der ungleichmäßigen Verteilung der Symbole des Alphabets liegt (im vorliegenden DNS-Datensatz ist der Anteil an `A` und `T` höher als der an `G` und `C`): Die Auswahl der zu suchenden Substrings berücksichtigt diese Verteilung nicht, was dann zur Folge hat, dass mit einer größeren Wahrscheinlichkeit häufiger vorkommende Substrings gesucht werden. Bei SLEQ ist der sich dadurch ergebende Effekt noch deutlicher vorhanden als etwa bei Pk1 oder GraI, was daran liegt, dass er durch die anders geartete Suchphase stärker wirkt.

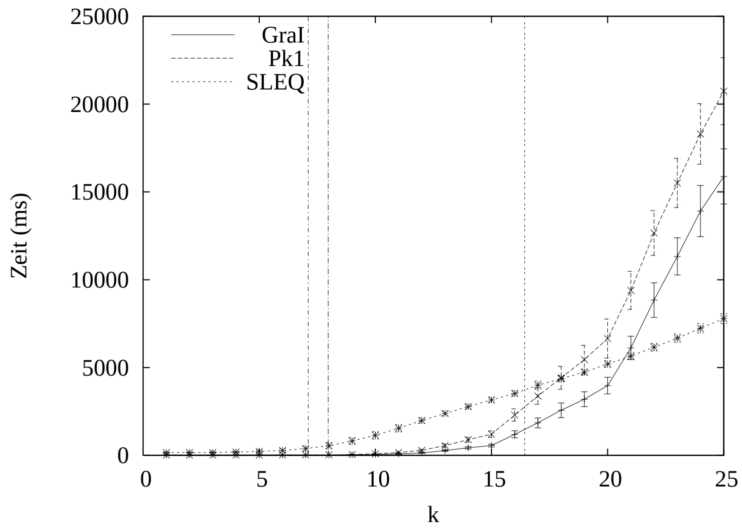


(a)

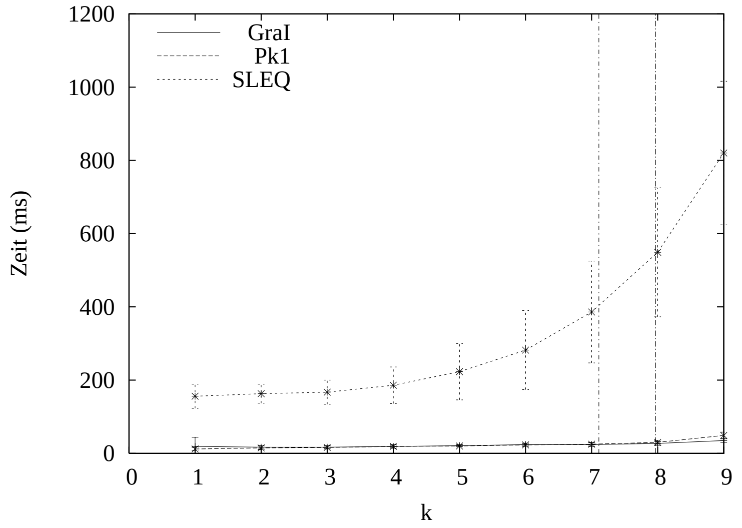


(b)

Abbildung 8.21: Filteralgorithmen im Vergleich. In (a) sind die Suchzeiten von Pattern der Länge $m = 64$ in einem Zufallstext ($\sigma = 4, n = 100000, l_{av} = 6.62$) bis zu einer Fehlerzahl eingetragen, die höher ist als alle interessanten Grenzen. Für Werte von k links von der linken bzw. der mittleren senkrechten Linie dominiert die Suchphase bei Pk1 (Gleichung 8.2.5) bzw. bei GraI (Gleichung 8.2.10). Die rechte Senkrechte markiert die Fehlerzahl, ab der GraI im Schnitt besser ist als Pk1 (Bedingung 8.2.1). In (b) ist ein Ausschnitt aus (a) für kleinere Fehlerzahlen dargestellt.



(a)



(b)

Abbildung 8.22: Filteralgorithmen im Vergleich. In (a) sind die Suchzeiten von Pattern der Länge $m = 64$ in einem DNS-Datensatz ($n = 100000, l_{av} = 6.79$) eingetragen. Die linke Senkrechte gibt den durch Gleichung 8.2.5 gegebenen k Wert an, bis zu dem die Suchphase bei Pk1 dominiert. Der entsprechende Wert für GraI, gegeben durch Gleichung 8.2.10, wird durch die mittlere Senkrechte angezeigt. Die rechte Senkrechte markiert die Fehlerzahl, ab der GraI im Schnitt besser ist als Pk1 (Bedingung 8.2.1). In (b) ist ein Ausschnitt aus (a) für kleinere Fehlerzahlen dargestellt.

Eine noch ungleichmäßigere Verteilung der verschiedenen Symbole des Alphabets ist bei Programmquelltext gegeben. Zudem ist dabei aber auch eine deutlich höhere Strukturierung vorhanden, da der Programmquelltext selbstverständlich der Grammatik der Programmiersprache folgt[1].

Die Suche in einem solchen Programmquelltext führt zu dem in Abbildung 8.23 aufgezeigten Laufzeitverhalten. Deutlich ist wieder der vergleichsweise hohe Suchaufwand von SLEQ bei geringen Fehlerleveln. Zudem sind in der Grafik aber auch die Auswirkungen der starken Strukturierung der Daten deutlich zu sehen: Das durch Gleichung 8.2.12 gegebene Intervall, das den Bereich beschreibt, in dem GraI besser ist als Pk1 und in dem noch bei beiden Algorithmen die Suchphase dominant ist, erstreckt sich über eine relativ große Spanne möglicher Werte für k. In Abbildung 8.23(a) ist dieses Intervall durch die linke und mittlere Senkrechte gegeben. Ebenso relativ groß ist der Bereich, in dem nach Gleichung 8.2.10 für GraI noch die Suchphase dominiert, aber für Pk1 schon die Verifikationskosten Überhand gewinnen. Die Grenzen dieses Bereichs sind in Abbildung 8.23(a) durch die mittlere und die rechte Senkrechte gekennzeichnet.

Diese Beispiele zeigen im Zusammenhang mit den vorher theoretisch erzielten Ergebnissen deutlich, dass der Algorithmus GraI den zugrunde liegenden Algorithmus Pk1 in Bezug auf die Suchzeiten verbessert. Der Grad der Verbesserung bzw. die Frage, ob die Verbesserung überhaupt den Bereich der dominanten Suchphase betrifft, hängt stark von den Daten ab, auf denen gesucht wird.

Insgesamt zählt der Algorithmus Pk1 schon zu den schnellsten Algorithmen für das Problem des Approximate String Matching. Der Algorithmus GraI verändert Pk1 zwar, hält dabei aber negative Einflüsse außen vor und so kommt dieses Merkmal eindeutig auch GraI zugute.

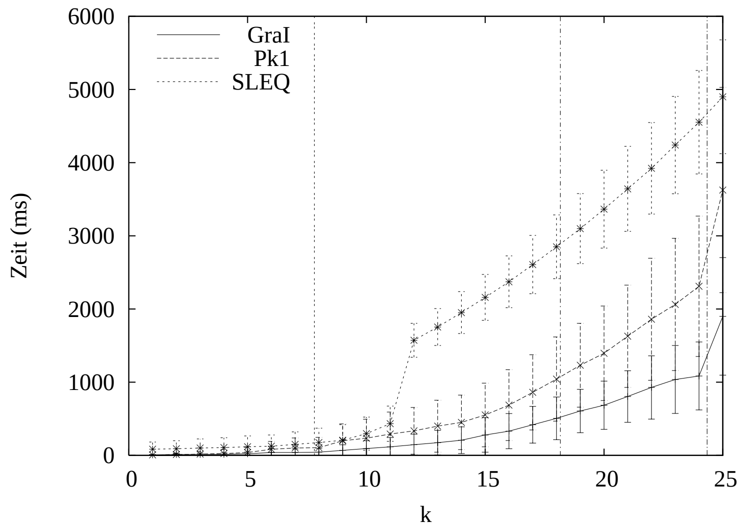
8.2.5 Zur Anwendung von GraI

In Kapitel 8.2.4 wurde bereits aufgezeigt, dass eine potenzielle Anwendung des Algorithmus GraI von den Daten abhängt, auf denen gesucht wird. Die Anwendbarkeit von GraI ist dann gegeben ist, wenn

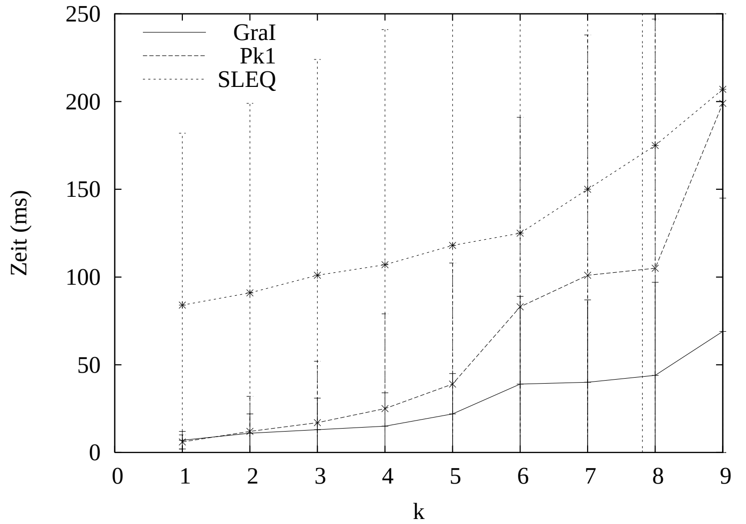
- GraI den zugrunde liegenden Algorithmus Pk1 in Bezug auf die Laufzeit der Suchanfrage verbessert, und wenn zugleich
- die Suchphase dominant, der Aufwand der Verifikationsphase also vernachlässigbar ist.

Diese beiden Bedingungen wurden in den Kapiteln 8.2.1 und 8.2.2.2 berechnet und lassen sich in Abhängigkeit vom Fehlerlevel α formulieren.

[1] Für einen Einblick in solche Grammatiken sei z. B. auf die C++ Grammatik in [114] verwiesen.



(a)



(b)

Abbildung 8.23: Filteralgorithmen im Vergleich. In (a) sind die Suchzeiten von Pattern der Länge $m = 50$ in einem Programmquellcode ($n = 264102, l_{av} = 11.80$) eingetragen. Die Fehlerzahl, von der ab GraI im Schnitt besser wird als Pk1 (gegeben durch Bedingung 8.2.1) ist mit linken senkrechten Linie markiert. Die mittlere und die rechte Senkrechte geben jeweils die Werte an, bis zu denen bei Pk1 (Gleichung 8.2.5) und GraI (Gleichung 8.2.10) die Suchphase dominiert. In (b) ist ein Ausschnitt aus (a) für kleinere Fehlerzahlen dargestellt.

Für die erste Bedingung ergibt sich (siehe auch Gleichung 8.2.1):

$$\alpha \geq \frac{2}{l_{av} + 1} . \tag{8.2.20}$$

Da sich die erste Bedingung im direkten Vergleich auf Pk1 bezieht, kann für die zweite Bedingung auch die für Pk1 formulierte Form (vgl. Gleichung 8.2.5) verwendet werden, welche auch (als abgeschwächte Form der verfeinerten Bedingung 8.2.11) zugleich für GraI gültig ist:

$$\alpha \leq \frac{1}{3log_{\sigma}m} . \tag{8.2.21}$$

Damit diese beiden Bedingungen gleichzeitig erfüllt sind, muss für α also

$$\frac{2}{l_{av} + 1} \leq \alpha \leq \frac{1}{3 \log_{\sigma} m} \tag{8.2.22}$$

gelten (vgl. auch Gleichung 8.2.12). Dieses Intervall charakterisiert den Bereich der potenziellen Nutzung des Algorithmus GraI. Existiert dieses Intervall für α, so ist die Anwendung von GraI sinnvoll. Dieses Intervall sei nun als *Nutzbereich* bezeichnet.

Auf den ersten Blick erscheint die Frage interessant, bei welcher Art von Daten eine prinzipielle Nutzbarkeit von GraI vorliegt. Dies ist gleichbedeutend mit der Existenz des Nutzbereichs bei einer minimalen betrachteten Patterngröße von $m = 2$, da dann der Nutzbereich auch für jede andere Patterngröße existiert. Für einen Datensatz mit der Alphabetgröße σ kann dann berechnet werden, für welche Textlänge n die durchschnittliche Regellänge l_{av} erreicht ist, die die Existenz des Nutzbereichs garantiert.

Anhand zweier solcher Berechnungen lässt sich schnell erkennen, dass der Wert einer solchen Aussage jedoch nur recht beschränkt ist: Bei Betrachtung des Genoms des X-Chromosoms von Drosophila Melanogaster sind beispielsweise etwa 11.4 Millionen Symbole notwendig, um die durchschnittliche Regellänge $l_{av} = 11.97$ zu erreichen, die für die Existenz des Nutzbereichs notwendig ist. Bei einem englischen Text (King-James-Bibel mit einem genutzten Alphabet der Größe $\sigma = 62$) reichen hingegen schon weniger als 100 Symbole, um die für die Existenz des Nutzbereichs notwendige durchschnittliche Regellänge von $l_{av} = 3,36$ zu erreichen. So lässt sich zwar erkennen, dass eine Suche auf Genomdaten mit GraI allenfalls auf sehr langen Datensätzen sinnvoll sein kann. Jedoch gerade bei Datensätzen mit einer stärker ausgeprägten potenziellen Anwendungsmöglichkeit von GraI ist eine qualitativ stärkere Aussage von Interesse, zumal auch die fehlerbehaftete Suche nach Strings der Länge 2 im Allgemeinen nicht typisch ist.

Im Folgenden werden verschiedene Datensätze betrachtet und es wird jeweils der Nutzbereich für variierende n und m auf zwei verschiedene Arten aufgezeigt: Zum einen werden die den Nutzbereich beschränkenden Fehlerlevel $\alpha_{min} = \frac{2}{l_{av}+1}$ und $\alpha_{max} = \frac{1}{3 \log_{\sigma} m}$ bestimmt (vgl. Gleichung 8.2.22). Zum anderen wird die für die Existenz des Nutzbereichs mindestens nötige durchschnittliche Regellänge $l_{av_{min}}$ der sich real ergebenden durchschnittlichen Regellänge l_{av} gegenübergestellt. Dabei ergibt sich $l_{av_{min}}$ durch gleichsetzen von α_{min} und α_{max}:

$$\begin{aligned} \frac{2}{l_{av_{min}}+1} &= \frac{1}{3\log_\sigma m} \\ \Longleftrightarrow \quad l_{av_{min}} &= 6\log_\sigma m - 1 \end{aligned} \tag{8.2.23}$$

In beiden Fällen ist an den Grafiken ablesbar, wann der Nutzbereich existiert. Doch während der aufgetragene Bereich des Fehlerlevels eine direkte praktische Bedeutung hat, ist die Angabe der durchschnittlichen Regellänge eher zur Abschätzung des weiteren Gesamtverhaltens eines solchen Datensatzes geeignet.

Zu beachten ist bei allen betrachteten Datensätzen, dass die hier genannte Alphabetgröße immer die prinzipielle Alphabetgröße des Datensatzes ist. Sie entspricht also der Anzahl verschiedener Symbole, die bei der vollen Textlänge von $n = 100000$ wirklich verwendet werden. Für die Berechnung der Werte $(\alpha_{min}, \alpha_{max}, l_{av_{min}})$ wird immer die Größe des wirklich verwendeten Alphabets genutzt. Diese kann dadurch bei kleineren Textlängen ($n < 100000$) von der prinzipiellen Alphabetgröße nach unten abweichen, dass einzelne, seltener auftretende Symbole in den dann abgeschnittenen Textbereichen nicht auftreten.

8.2.5.1 Genomdaten

Als auf dem DNS-Alphabet basierende biologische Testdaten findet das vollständige Genom von Haemophilus Influenzae Rd Verwendung, welches jeweils auf die entsprechende, notwendige Länge eingeschränkt ist. Nur ein einziges Symbol der auf 100000 Symbole beschränkten Datensequenz entstammt nicht dem vier-buchstabigen DNS-Alphabet, sondern ist eine unbekannte Base.

Die vollständige Grammatik dieses Datensatzes ist für den Abdruck zu groß. Doch zur Veranschaulichung zeigt Abbildung 8.24 die Grammatik, die sich bei Betrachtung der ersten 100 Symbole des Genoms ergibt. Aus der vollständigen Grammatik bei 10000 Symbolen ist Abbildung 8.25 gewonnen, die die Verknüpfungen von Regeln (ab der Länge 12 [2]) in der Grammatik zeigt. Jede der Regeln ist dort durch eine Ellipse repräsentiert, in der die Nummer und die Länge der Regel vermerkt sind. Diese Informationen sind hier jedoch nicht von Bedeutung. Die Grafik soll einzig und allein ein Gefühl für die Stärke der Strukturierung der sich ergebenden Grammatik vermitteln. Es ist deutlich zu sehen, dass nur sehr wenige Regeln die Länge 12 erreichen und diese dann auch nicht hierarchisch miteinander verknüpft sind.

In Anbetracht der Anwendung von GraI auf diesem Genom-Datensatz zeigt Abbildung 8.26 die Entwicklung des für die Existenz des Nutzbereichs maximal möglichen

[2] Die Länge 12 wurde hier gewählt, damit alle entsprechenden Abbildungen in den Kapiteln 8.2.5.1 - 8.2.5.6 optimal vergleichbar und zugleich noch darstellbar sind.

Eingabesequenz:

`TATGGCAATTAAAATTGGTATCAATGGTTTTGGTCGTATCGGCCGTATCG`
`TATTCCGTGCAGCACAACACCGTGATGACATTGAAGTTGTAGGTATTAAC`

Grammatik:

$$
\begin{array}{lcl}
S & \rightarrow & R_1R_2R_3\mathtt{T}R_4R_4R_5R_1R_3R_2R_6R_5\mathtt{T}R_7R_8R_7R_1\mathtt{T}R_9R_{10}R_{10} \\
 & & R_{11}R_{12}R_9\mathtt{G}R_{13}\mathtt{G}R_{12}R_{14}R_4\mathtt{G}R_{14}R_{15}R_2R_{15}R_6R_4\mathtt{C} \\
R_1 & \rightarrow & \mathtt{T}R_{13} \\
R_2 & \rightarrow & \mathtt{GG} \\
R_3 & \rightarrow & R_{11}R_{13} \\
R_4 & \rightarrow & \mathtt{AA} \\
R_5 & \rightarrow & R_6R_2 \\
R_6 & \rightarrow & \mathtt{TT} \\
R_7 & \rightarrow & R_{16}R_1R_{16} \\
R_8 & \rightarrow & \mathtt{GC} \\
R_9 & \rightarrow & \mathtt{C}R_{16}\mathtt{T} \\
R_{10} & \rightarrow & R_8\mathtt{A} \\
R_{11} & \rightarrow & \mathtt{CA} \\
R_{12} & \rightarrow & \mathtt{A}R_{11} \\
R_{13} & \rightarrow & \mathtt{AT} \\
R_{14} & \rightarrow & R_6\mathtt{G} \\
R_{15} & \rightarrow & \mathtt{TA} \\
R_{16} & \rightarrow & \mathtt{CG}
\end{array}
$$

Abbildung 8.24: Die Grammatik der ersten 100 Symbole des DNS-Datensatzes.

Abbildung 8.25: Regeln, die sich bei Verwendung des DNS-Datensatzes ergeben. Von den insgesamt 2892 (durchnummerierten) Regeln der Grammatik erreichen nur die gezeigten 9 eine Länge von mindestens 12.

Fehlerlevels α_{max} gegenüber dem minimal notwendigen Fehlerlevel α_{min}. In den Bereichen, in denen die durch α_{max} gegebene Fläche oberhalb der durch α_{min} gegebenen Fläche liegt, existiert nach den oben genannten Gleichungen der Nutzbereich.

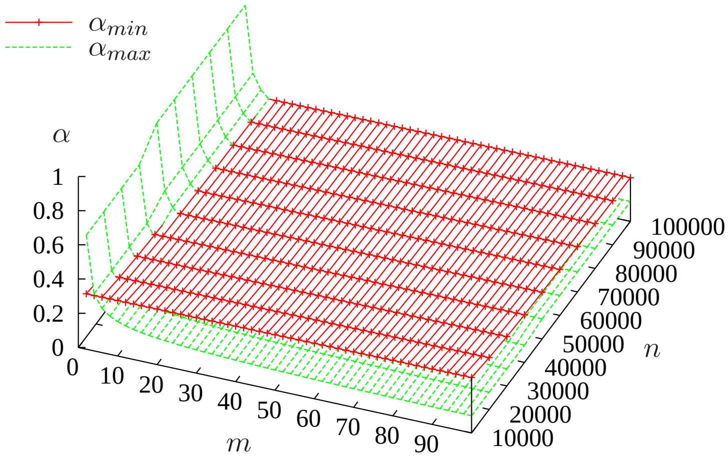


Abbildung 8.26: Nutzbereich bei einem DNS-basierten Datensatz. Der Nutzbereich existiert für die Parametersätze, für die $\alpha_{max} \geq \alpha_{min}$ ist.

Auch in Abbildung 8.27 wird deutlich, wann der Nutzbereich existiert. Dies ist der Fall, wenn dort die auf den Daten ermittelte durchschnittliche Regellänge l_{av} größer ist als die nach Gleichung 8.2.23 minimal notwendige Regellänge $l_{av_{min}}$.

Es ist deutlich zu erkennen, dass die Verwendung von GraI nur für sehr kurze Pattern sinnvoll ist, bzw. dass extrem große Datensätze vorliegen müssen, damit der Einsatz von GraI sich auch bei der Suche nach längeren Pattern lohnen kann.

8.2.5.2 Aminosäurensequenz

Als weitere Testdatenmenge biologischer Art wird die Aminosäurensequenz verwendet, die sich aus der Aneinanderreihung von 188 Genen[3] von Caenorhabditis Elegans ergibt. Das bei dem gesamten Datensatz genutzte Alphabet hat eine Größe von $\sigma = 20$.

Wie schon zuvor in Kapitel 8.2.5.1 wird auch für diesen Datensatz exemplarisch in Abbildung 8.28 die sich ergebende Grammatik für die ersten 100 Symbole dargestellt. In Abbildung 8.29 sind aus der Grammatik des Datensatzes mit 10000 Symbolen die Regeln

[3] In alphabetischer Reihenfolge dem Namen nach dem ExPASY Proteomics Server entnommen.

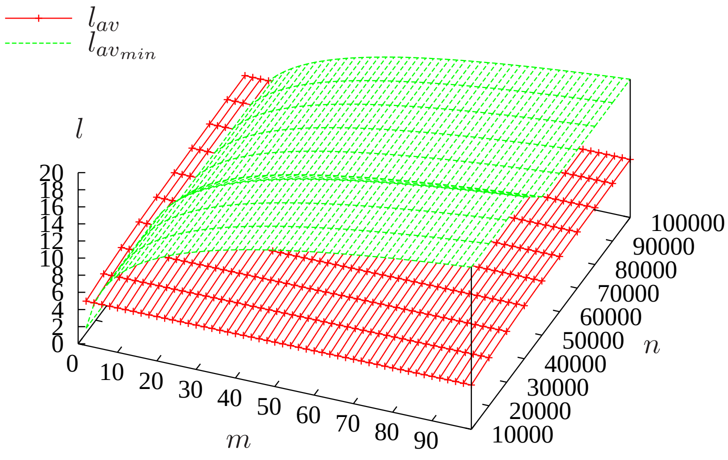


Abbildung 8.27: Nutzbereich bei einem DNS-basierten Datensatz. Der Nutzbereich existiert für die Parametersätze, für die $l_{av} \geq l_{av_{min}}$ ist.

mit einer Länge von mindestens 12 herausgesucht und entsprechend der hierarchischen Inklusion miteinander (durch Pfeile) verbunden. Auch hier sind die Regelnummern und -längen nicht von Interesse, sondern nur die Regelverknüpfungen. Es sind zwar nicht sehr viele Regeln der Länge 12 oder größer vorhanden, doch zeigt sich bei einigen von diesen die Einbettung in noch längere Regeln.

Abbildung 8.30 zeigt für die Verwendung dieses Datensatzes mit GraI, in welchem Bereich sich der Fehlerlevel bewegen darf, damit der Nutzbereich existiert.

Abbildung 8.31 demonstriert auf dem Datensatz die Entwicklung der durchschnittlichen Regellänge l_{av}, welche dort der minimalen durchschnittlichen Regellänge gegenübergestellt wird, die für die Existenz des Nutzbereichs notwendig ist.

Hinsichtlich der Patternlänge ist GraI auf diesem Datensatz besser verwendbar als für die DNS-Daten in Kapitel 8.2.5.1. Dennoch sollte m für $n \leq 100000$ doch nicht größer sein als etwa 18, wobei allerdings dann der Fehlerlevel immerhin mindestens 0.28 betragen sollte (oder je nach Textlänge, bzw. der damit verbundenen durchschnittlichen Regellänge, auch höher).

8.2.5.3 Zeitreihe

Als weiteres Testdatum fungiert hier eine aus Börsendaten gewonnene Zeitreihe. Dazu werden die minütlichen Open-Werte des DAX (ab dem 28.08.2003) in Form von Differenzwerten verwendet. Dabei ergibt sich ein verwendetes Alphabet der Größe $\sigma = 141$.

Eingabesequenz:

`MPITKDGPFCLKVMNKDKKSVAKFLRNEPKRSYKQTGLASLFGCCSDTED`
`TMEVLDDNGLIIATSFLHHDQFRGILITMKDPAGKVLIGIQASRDQKDVF`

Grammatik:

$$S \rightarrow \mathtt{MP}R_1R_2\mathtt{GPFCL}R_3\mathtt{MN}R_2\mathtt{KKSVAK}R_4\mathtt{RNEPKRSYKQT}R_5R_6\mathtt{LFGCCS}R_7\mathtt{E}R_7$$
$$\mathtt{MEVLDDN}R_5\mathtt{IIATS}R_4\mathtt{HH}R_8\mathtt{FR}R_9\mathtt{L}R_1\mathtt{M}R_2\mathtt{PAG}R_3\mathtt{LI}R_9\mathtt{Q}R_6\mathtt{R}R_8R_2\mathtt{VF}$$
$$R_1 \rightarrow \mathtt{IT}$$
$$R_2 \rightarrow \mathtt{KD}$$
$$R_3 \rightarrow \mathtt{KV}$$
$$R_4 \rightarrow \mathtt{FL}$$
$$R_5 \rightarrow \mathtt{GL}$$
$$R_6 \rightarrow \mathtt{AS}$$
$$R_7 \rightarrow \mathtt{DT}$$
$$R_8 \rightarrow \mathtt{DQ}$$
$$R_9 \rightarrow \mathtt{GI}$$

Abbildung 8.28: Die Grammatik der ersten 100 Symbole des Protein-Datensatzes.

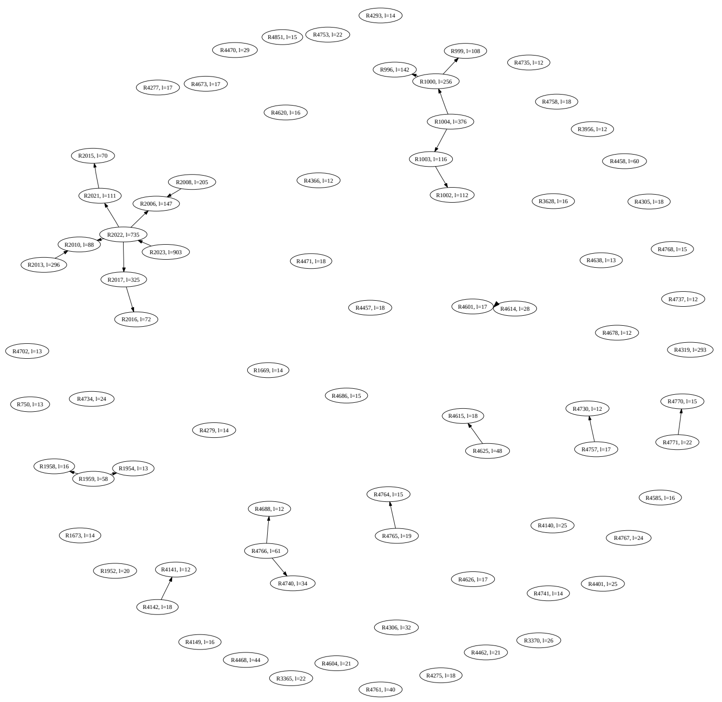


Abbildung 8.29: Regeln, die sich bei Verwendung des Protein-Datensatzes ergeben. Von den insgesamt 4857 (durchnummerierten) Regeln der Grammatik erreichen die gezeigten 78 eine Länge von mindestens 12.

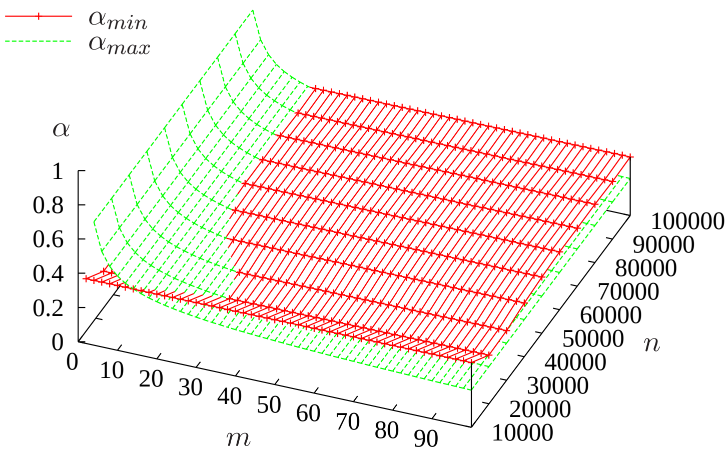


Abbildung 8.30: Nutzbereich bei einer Aminosäuresequenz. Der Nutzbereich existiert für die Parametersätze, für die $\alpha_{max} \geq \alpha_{min}$ ist.

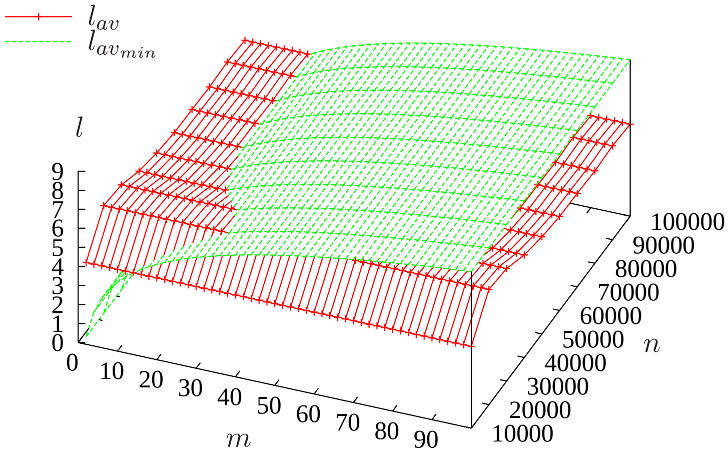


Abbildung 8.31: Nutzbereich bei einer Aminosäuresequenz. Der Nutzbereich existiert für die Parametersätze, für die $l_{av} \geq l_{av_{min}}$ ist.

Die sich aus diesen Daten (bei Betrachtung der ersten 100 Werte) ergebende Grammatik ist in Abbildung 8.32 dargestellt. Abbildung 8.33 zeigt alle Regeln mit der Mindestlänge 12 in der Grammatik des Datensatzes von 10000 Symbolen. Während die Regeln selbst oder auch deren Nummern nicht von Bedeutung sind, ist es jedoch wichtig, dass unter diesen Regeln wie beim DNS-Datensatz aus Kapitel 8.2.5.1 keine weitere gegenseitige Einbettung besteht. Gerade weil diese Regeln zum Teil sogar deutlich länger als 12 sind (bis hin zur Länge 281), ist diese Feststellung ein Indiz dafür, dass die Daten nicht übermäßig stark strukturiert sind.

Eingabesequenz:

```
-111 0.5 -132.5 3 2 -3.5 3 2.5 0.5 -3 2 0 -3 1 -0.5 -1 -1.5 3 -0.5
4.5 1.5 1 0 0 1.5 -1.5 -0.5 1 -1 -1 -2 1.5 2 -2 0 2 -2.5 0.5 1 -4.5
1.5 -0.5 -2 1 0 -0.5 -0.5 -0.5 0.5 -2.5 -0.5 1 0 -1 0.5 1.5 0 0.5
-0.5 0 1 -1 0.5 -0.5 1 0 -0.5 0 0 -0.5 1.5 0.5 0.5 1 0.5 0.5 -1 -1
0.5 0 0 -0.5 0 1.5 -1.5 0.5 0.5 0 1 0 0 -0.5 0 1 0 2 0.5 1 -0.5 1
```

Grammatik:

$S \rightarrow$ -111 0.5 -132.5 3 2 -3.5 3 2.5 0.5 -3 2 0 -3 1 -0.5 -1 -1.5 3 -0.5 4.5 1.5 $R_1 R_2 R_3 R_4$ -2 1.5 2 -2 0 2 -2.5 R_5 -4.5 1.5 -0.5 -2 R_1 -0.5 -0.5 -0.5 0.5 -2.5 $R_6 R_7$ 1.5 0 0.5 R_8 1 $R_7 R_6 R_8 R_9$ 1.5 0.5 $R_5 R_{10} R_4$ 0.5 0 $R_9 R_2 R_{10} R_{11} R_9 R_{11}$ 2 $R_5 R_3$

$R_1 \rightarrow$ 1 0

$R_2 \rightarrow$ 0 1.5 -1.5

$R_3 \rightarrow$ -0.5 1

$R_4 \rightarrow$ -1 -1

$R_5 \rightarrow$ 0.5 1

$R_6 \rightarrow$ R_3 0

$R_7 \rightarrow$ -1 0.5

$R_8 \rightarrow$ -0.5 0

$R_9 \rightarrow$ 0 -0.5

$R_{10} \rightarrow$ 0.5 0.5

$R_{11} \rightarrow$ 0 R_1

Abbildung 8.32: Die Grammatik der ersten 100 Symbole des Zeitreihen-Datensatzes.

Bei der Verwendung von GraI zur Suche auf diesen Daten sind in Abbildung 8.34 die minimalen und maximalen Fehlerlevel aufgetragen, für die der Nutzbereich jeweils noch existiert.

Abbildung 8.35 stellt die auf den Daten erreichte durchschnittliche Regellänge l_{av} der für die Existenz des Nutzbereichs minimal notwendigen durchschnittlichen Regellänge $l_{av_{min}}$ gegenüber.

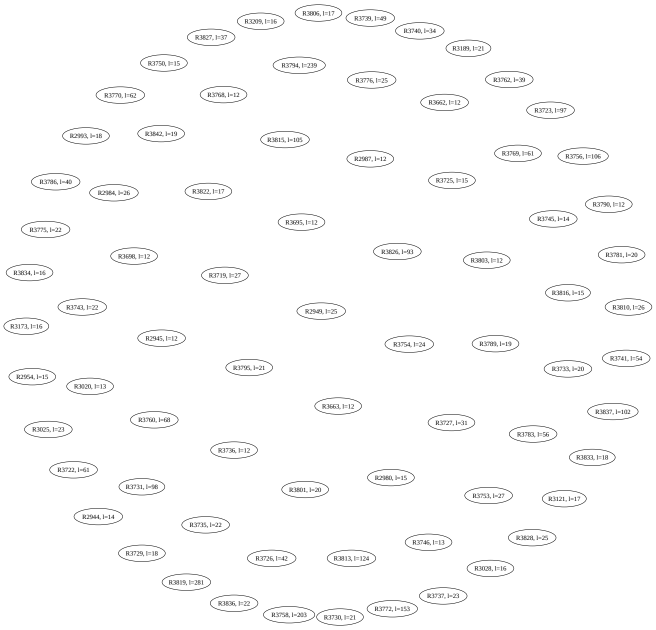


Abbildung 8.33: Regeln, die sich bei Verwendung des Zeitreihen-Datensatzes ergeben. Von den insgesamt 4329 (durchnummerierten) Regeln der Grammatik erreichen die gezeigten 75 eine Länge von mindestens 12.

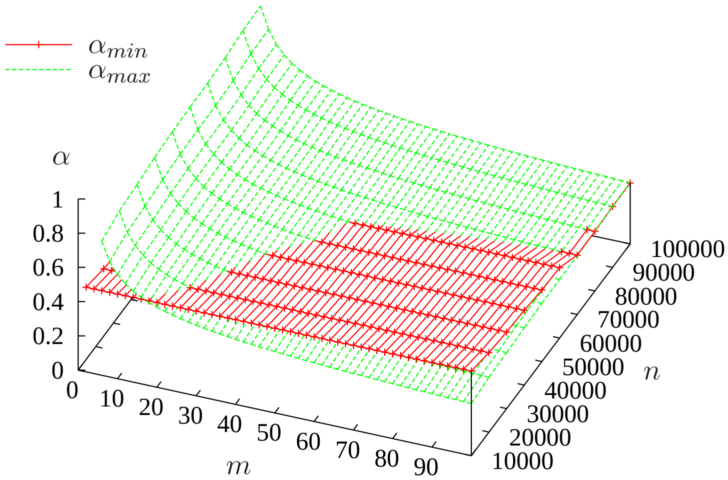


Abbildung 8.34: Nutzbereich bei einer Zeitreihe. Der Nutzbereich exisitert für die Parametersätze, für die $\alpha_{max} \geq \alpha_{min}$ ist.

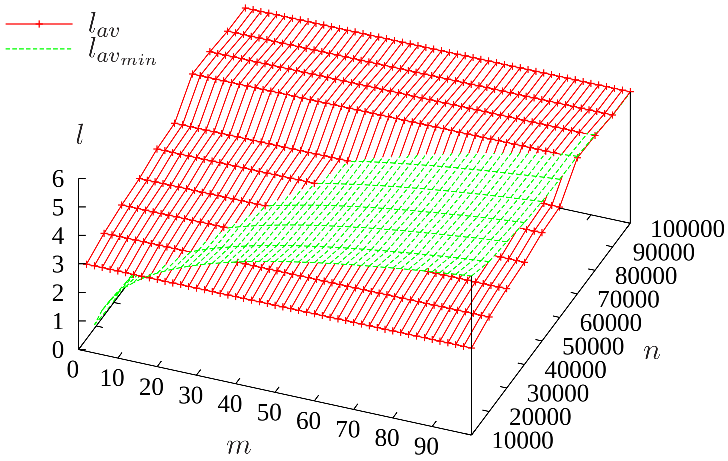


Abbildung 8.35: Nutzbereich bei einer Zeitreihe. Der Nutzbereich existiert für die Parametersätze, für die $l_{av} \geq l_{av_{min}}$ ist.

In den Abbildungen zeigt sich, dass GraI auf diesem Datensatz hinsichtlich der Patternlänge, insbesondere bei nicht zu kleiner Textlänge, gut verwendbar ist. Zu beachten ist jedoch dann, dass der bei der Suche verwendete Fehlerlevel mit mindestens 0.34 schon relativ hoch sein muss.

8.2.5.4 Englischsprachiger Text, reduziertes Alphabet

Der Text der King-James-Bibel wird auf zwei verschiedene Arten als Testdatum verwendet. In Kapitel 8.2.5.5 wird die Betrachtung der Fehlerlevel auf dem unverarbeiteten Text durchgeführt, während hier der Text in Großbuchstaben konvertiert ist und zudem alle Satzzeichen jetzt Leerzeichen sind. Das sich ergebende Alphabet hat (inklusive der Zeilenumbrüche) eine Größe von $\sigma = 28$.

Das Aussehen der sich ergebenden Grammatik ist in Abbildung 8.36 beispielhaft für die ersten 100 Symbole des Textes gezeigt. In Abbildung 8.37 ist die Strukturierung des Datensatzes angedeutet. Dazu sind alle Regeln der Grammatik des Textes mit 10000 Symbolen als Ellipse dargestellt, die mindestens die Länge 12 haben. Eine Regel ist in der Abbildung immer dann mit einer anderen verbunden, wenn diese in der anderen enthalten ist. Hierbei wurde mehr Wert darauf gelegt, dass die Struktur deutlich wird, als dass die in den Ellipsen jeweils vermerkten Regelnummern und -längen zu erkennen sind.

Eingabesequenz:

`IN_THE_BEGINNING_GOD_CREATED_THE_HEAVEN_AND_THE_EARTH_`$\backslash n$
`AND_THE_EARTH_WAS_WITHOUT_FORM__AND_VOID__AND`

Grammatik:

$$
\begin{array}{lcl}
S & \rightarrow & R_1\texttt{\_}R_2\texttt{BEG}R_1\texttt{N}R_1\texttt{G\_GO}R_3\texttt{CR}R_4\texttt{TE}R_5\texttt{H}R_4\texttt{VEN\_}R_6\backslash n \\
 & & R_6\texttt{WAS\_WI}R_7\texttt{OUT\_FORM\_}R_8R_3\texttt{VOI}R_3R_8\texttt{D} \\
R_1 & \rightarrow & \texttt{IN} \\
R_2 & \rightarrow & R_7\texttt{E\_} \\
R_3 & \rightarrow & \texttt{D\_} \\
R_4 & \rightarrow & \texttt{EA} \\
R_5 & \rightarrow & R_3R_2 \\
R_6 & \rightarrow & R_9R_5R_4\texttt{R}R_7\texttt{\_} \\
R_7 & \rightarrow & \texttt{TH} \\
R_8 & \rightarrow & \texttt{\_}R_9 \\
R_9 & \rightarrow & \texttt{AN}
\end{array}
$$

Abbildung 8.36: Die Grammatik der ersten 100 Symbole des bearbeiteten Textes.

Anhand von Abbildung 8.38 ist zu erkennen, unter welchen Bedingungen der Nutzbereich existiert, wenn mittels GraI in diesem bearbeiteten Text gesucht wird.

Abbildung 8.39 zeigt wiederum die ermittelte durchschnittliche Regellänge l_{av}, die hier fast immer größer ist als die für die Existenz des Nutzbereichs notwendige durchschnittliche Regellänge $l_{av_{min}}$.

Die Abbildungen zeigen offensichtlich, dass GraI hinsichtlich der Patternlänge sehr gut einsetzbar ist. Innerhalb des betrachteten Bereichs existiert der Nutzbereich im Prinzip bei fast jeder Patternlänge. Zudem ist in diesem Fall der minimal notwendige Fehlerlevel mit mindestens 0.22 auch recht moderat.

8.2.5.5 Englischsprachiger Text

Wie schon in Kapitel 8.2.5.4 wird auch hier der Text der King-James-Bibel als Testdatum verwendet. Diesmal jedoch ist der Text nicht manipuliert und verwendet so ein Alphabet der Größe $\sigma = 60$.

Als Beispiel für die sich ergebende Grammatik ist in Abbildung 8.40 die Grammatik aufgezeigt, die sich aus den ersten 100 Symbolen des Textes berechnet. Abbildung 8.41 zeigt wieder anhand der Regeln mit der Mindestlänge 12 den Grad der Strukturierung der Eingabe von 10000 Textsymbolen auf. Im Vergleich zum bearbeiteten Text (Kapitel 8.2.5.4) ist abgesehen von der leicht höheren Anzahl langer Regeln kein sehr großer Unterschied zu erkennen. Das deutet darauf hin, dass in Bezug auf den Nutzbereich für GraI bei diesem Text auch keine große Veränderung zu erwarten ist.

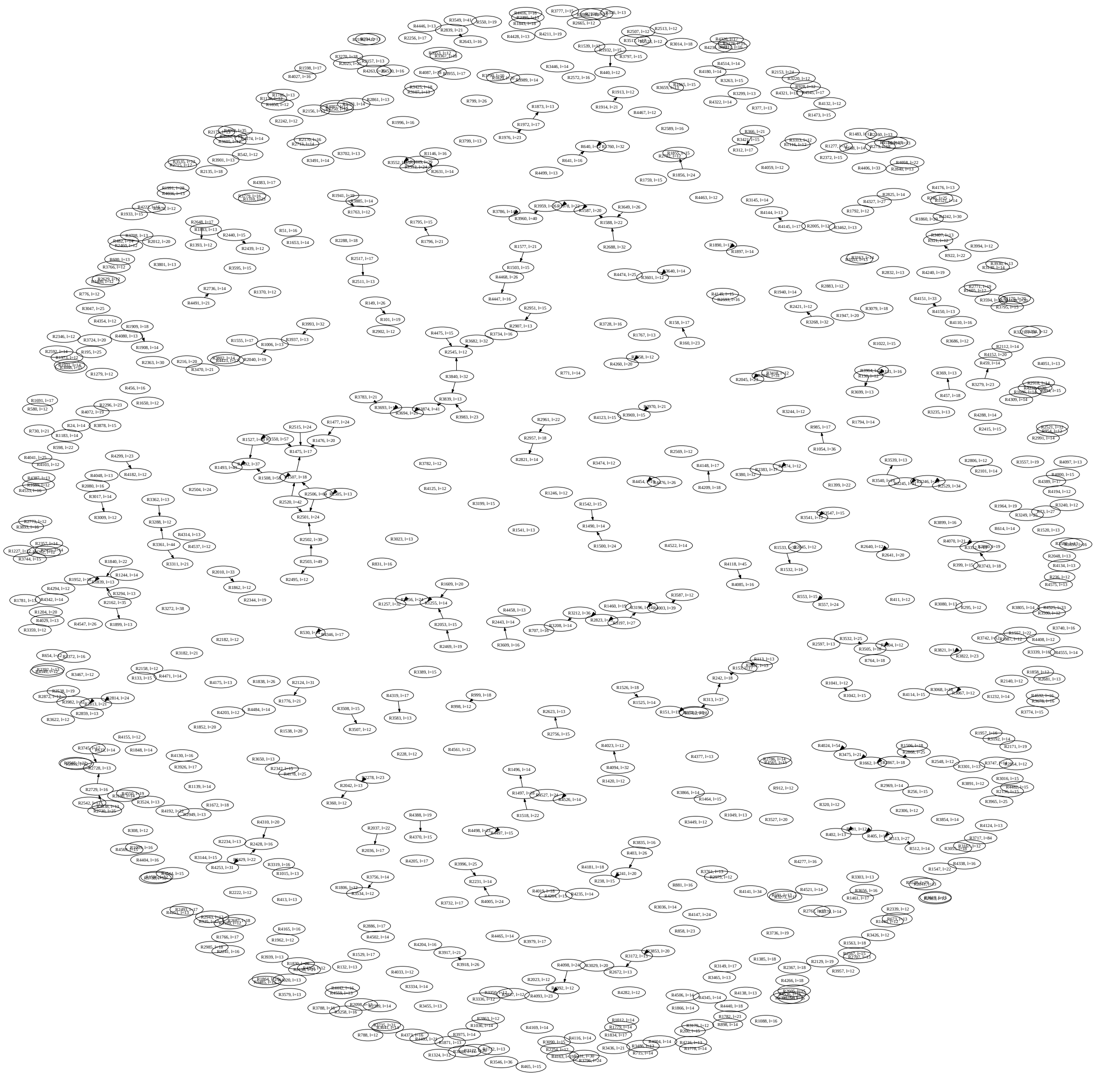


Abbildung 8.37: Regeln, die sich bei Verwendung des englischen Textes (mit reduziertem Alphabet) ergeben. Von den insgesamt 4581 (durchnummerierten) Regeln der Grammatik erreichen die gezeigten 783 eine Länge von mindestens 12.

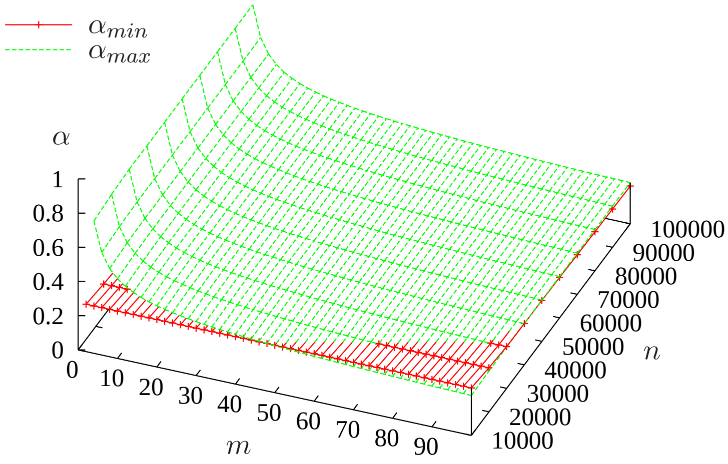


Abbildung 8.38: Nutzbereich bei einem englischen Text (mit reduziertem Alphabet). Der Nutzbereich existiert für die Parametersätze, für die $\alpha_{max} \geq \alpha_{min}$ ist.

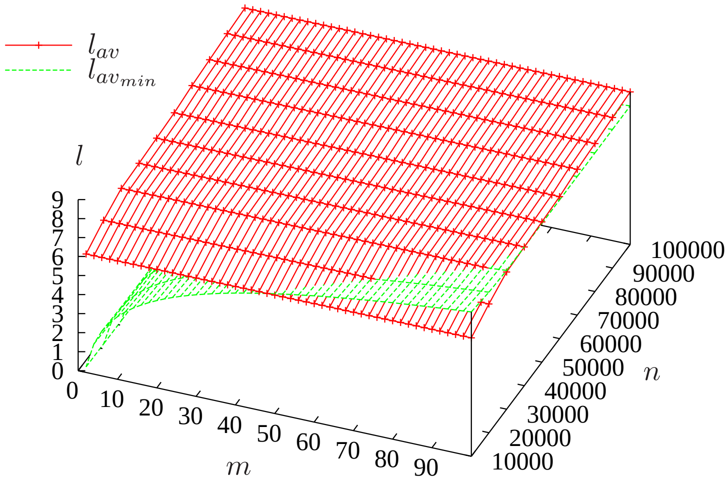


Abbildung 8.39: Nutzbereich bei einem englischen Text (mit reduziertem Alphabet). Der Nutzbereich existiert für die Parametersätze, für die $l_{av} \geq l_{av_{min}}$ ist.

Eingabesequenz:

`In the beginning God created the heaven and the earth.`$\backslash r \backslash n$
`And the earth was without form, and void; an`

Grammatik:

$$
\begin{array}{lll}
S & \rightarrow & \mathtt{I}R_1R_2\mathtt{beg}R_3\mathtt{n}R_3\mathtt{g\_Go}R_4\mathtt{cr}R_5\mathtt{te}R_6\mathtt{h}R_5\mathtt{ve}R_1\mathtt{a}R_7\mathtt{.}\backslash r\backslash n \\
 & & \mathtt{A}R_7R_8\mathtt{as}R_8\mathtt{i}R_9\mathtt{out\_form,}R_{10}R_4\mathtt{void;}R_{10} \\
R_1 & \rightarrow & \mathtt{n\_} \\
R_2 & \rightarrow & R_9\mathtt{e\_} \\
R_3 & \rightarrow & \mathtt{in} \\
R_4 & \rightarrow & \mathtt{d\_} \\
R_5 & \rightarrow & \mathtt{ea} \\
R_6 & \rightarrow & R_4R_2 \\
R_7 & \rightarrow & \mathtt{n}R_6R_5\mathtt{r}R_9 \\
R_8 & \rightarrow & \mathtt{\_w} \\
R_9 & \rightarrow & \mathtt{th} \\
R_{10} & \rightarrow & \mathtt{\_an}
\end{array}
$$

Abbildung 8.40: Die Grammatik der ersten 100 Symbole des Textes der King-James-Bibel. Leerzeichen des Eingabetextes sind in der Grammatik als Unterstriche (_) dargestellt.

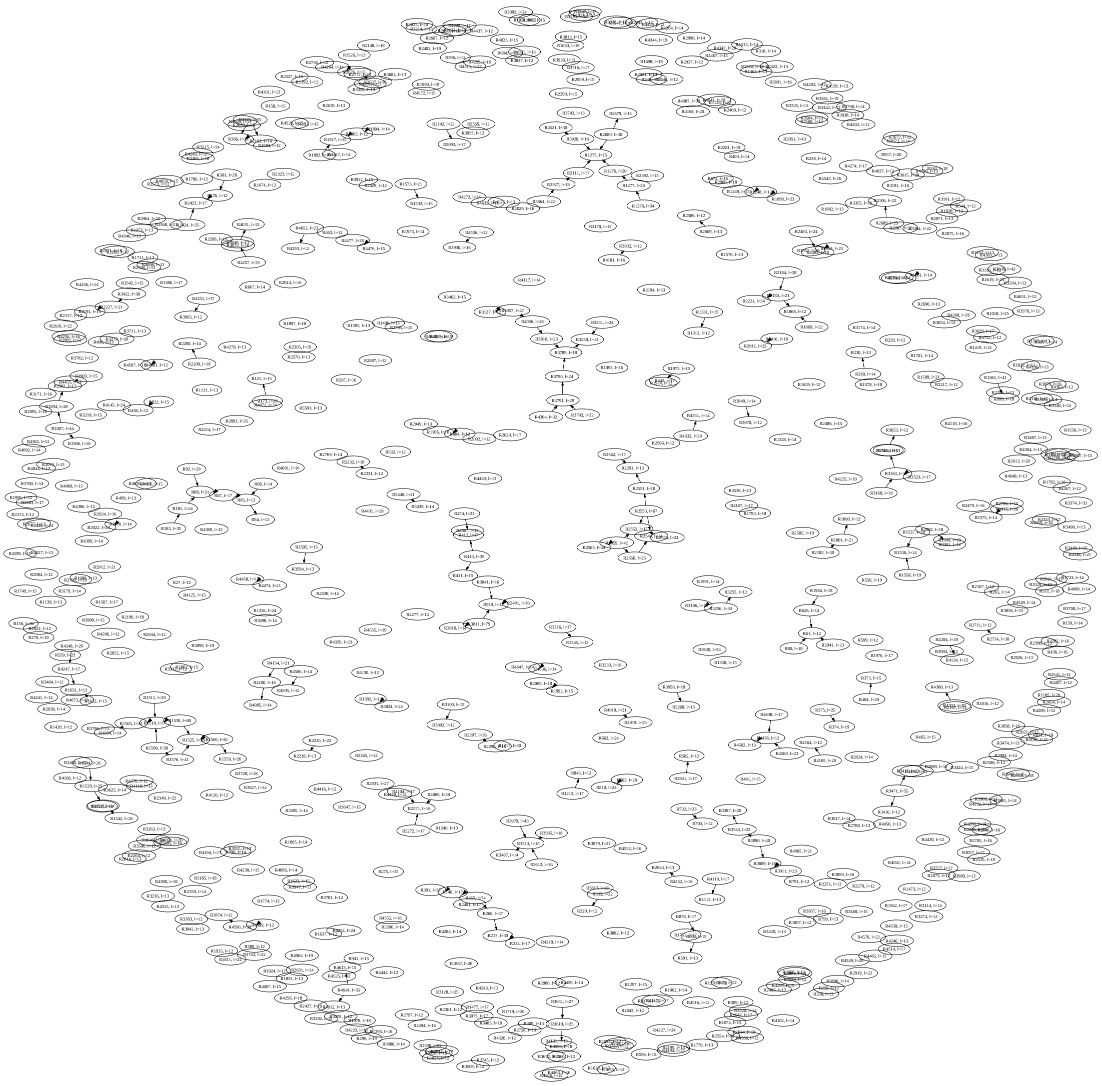


Abbildung 8.41: Regeln, die sich bei Verwendung des englischen Textes ergeben. Von den insgesamt 4653 (durchnummerierten) Regeln der Grammatik erreichen die gezeigten 795 eine Länge von mindestens 12.

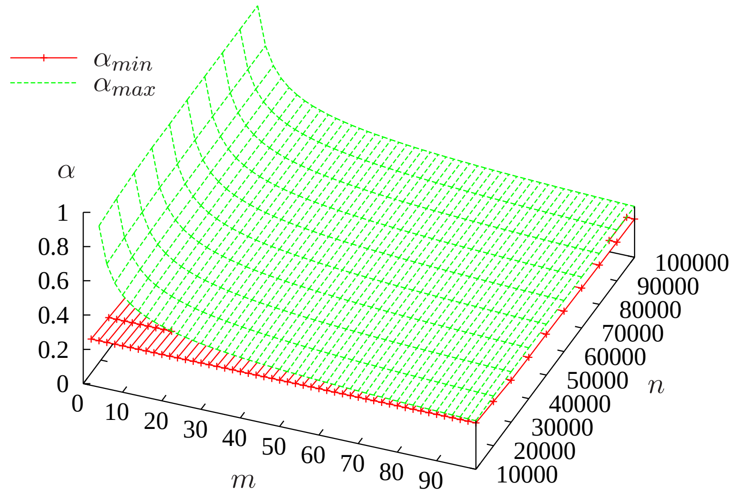


Abbildung 8.42: Nutzbereich bei einem englischen Text. Der Nutzbereich existiert für die Parametersätze, für die $\alpha_{max} \geq \alpha_{min}$ ist.

Die für die Existenz des Nutzbereichs minimalen und maximalen Fehlerlevel sind in Abbildung 8.42 eingetragen.

Die in Abbildung 8.43 eingetragene, erreichte durchschnittliche Regellänge l_{av} ist in diesem Fall immer größer als die für die Existenz des Nutzbereichs notwendige Minimallänge.

Auf diesem Datensatz ist GraI also gut nutzbar, da zumindest unter den betrachteten Bedingungen der Nutzbereich immer existiert. Vor allem bei kleineren Pattern ergibt sich eine recht große Spannweite für den möglichen Fehlerlevel, der jedoch auch hier mindestens 0.22 betragen muss.

8.2.5.6 Programmquellcode

Der von Kommentaren befreite Quellcode eines C-Programms dient als weiteres Testdatum. Der Quellcode verwendet insgesamt ein Alphabet aus 97 verschiedenen Symbolen.

In Abbildung 8.44 ist die Grammatik dargestellt, die sich aus den ersten 101 Symbolen des Quelltextes ergibt. Abbildung 8.45 zeigt alle Regeln der Grammatik des auf 10000 Zeichen beschränkten Progammquelltextes, die mindestens die Länge 12 haben. Um die Struktur der Grammatik dadurch zu verdeutlichen, ist eine Regel immer dann durch einen Pfeil mit einer anderen Regel verbunden, wenn sie in der anderen enthalten ist. Auch hier musste zugunsten der Darstellbarkeit der Strukturierung darauf verzichtet

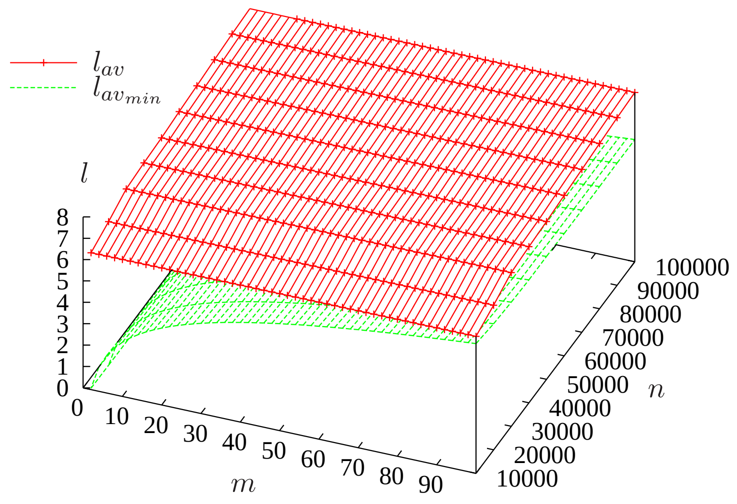


Abbildung 8.43: Nutzbereich bei einem englischen Text. Der Nutzbereich existiert für die Parametersätze, für die $l_{av} \geq l_{av_{min}}$ ist.

werden, die innerhalb der Ellipsen verzeichneten Regelnummern und -längen erkennbar zu halten.

Die Anwendung von GraI auf dem Programmquelltext wird in Abbildung 8.46 genauer betrachtet. Die dort vorgenommene Betrachtung des Nutzbereichs in Abhängigkeit des Fehlerlevels zeigt, dass in dem fraglichen Rahmen der Nutzbereich immer existiert.

Auch in Abbildung 8.47 ist erkennbar, dass der Nutzbereich unter allen betrachteten Parameterbereichen existiert. Zudem hebt sich die ermittelte durchschnittliche Regellänge l_{av} so stark von der minimal notwendigen durchschnittlichen Regellänge ab, dass die Existenz des Nutzbereichs noch weit über den hier dargestellten Bereich hinaus zu erwarten ist.

Die Anwendung von GraI auf diesen Quellcode-Daten ist auf breiter Ebene möglich. Es gibt keine besonderen Beschränkungen hinsichtlich der Patternlänge und auch der erlaubte Fehlerlevel darf beginnend ab 0.17 in einem recht großen Rahmen variieren.

8.3 Resultat

In diesem und dem vorangehenden Kapitel wurde die Entwicklung eines Algorithmus GraI aufgezeigt, der das Problem des Approximate String Matching löst. Dieser Algorithmus basiert auf dem Filteralgorithmus Pk1, der eine Aufteilung des gesuchten Patterns

Eingabesequenz:

`#define main_version ''$Revision: 1.115 $''`$\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n$
$\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n$
$\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n\backslash r\backslash n$

Grammatik:

$S \rightarrow$ `#def`R_1`e_ma`R_1`_ver`R_2`_''$Revi`R_2`:_1.115_$''`$R_3R_3R_3R_4R_5$
$R_1 \rightarrow$ `in`
$R_2 \rightarrow$ `sion`
$R_3 \rightarrow R_4R_4$
$R_4 \rightarrow R_5R_5$
$R_5 \rightarrow R_6R_6$
$R_6 \rightarrow \backslash r\backslash n$

Abbildung 8.44: Die Grammatik der ersten 101 Symbole des Programmquellcodes. Leerzeichen des Eingabetextes sind in der Grammatik als Unterstriche (_) dargestellt.

in $k+1$ Teile vornimmt, und diese Teilpattern dann exakt im Text sucht. Immer da, wo eines der Teilpattern im Text gefunden wird, wird ein Bereich des Textes mit einem Verifikationsalgorithmus auf das Vorhandensein eines Approximate Matching überprüft. Der Algorithmus GraI kombiniert den Algorithmus Pk1 mit der Nutzung einer Grammatik (Sequitur, vgl. Kapitel 6.3.4), die den Text beschreibt.

Die Grammatik wird in einem Vorberechnungsschritt aus dem Text erzeugt, weshalb GraI einen Offline-Algorithmus darstellt (siehe Kapitel 2.6). In dem Vorberechnungsschritt werden zudem noch wichtige, in der Grammatik enthaltene Informationen geeignet zusammengefasst.

Der Vorberechnungsschritt kann insgesamt in $O(n)$ Zeit ausgeführt werden. Der in diesem Schritt belegte Speicherplatz ist von derselben Ordnung $O(n)$.

Bei der eigentlichen Suche nach den Approximate Matchings folgt GraI den Grundgedanken von Pk1. Darüber hinaus werden die Regeln der Grammatik dazu genutzt, die durch sie abgedeckten Textbereiche, die ja mehrfach auftreten, nur einmal zu prüfen.

Der Algorithmus Pk1 zählt in der Praxis für Fehlerlevel $\alpha < 1/(3\log_\sigma m)$ zu den schnellsten Algorithmen zur Lösung des Problems des Approximate String Matching. Dieser durch den Fehlerlevel definierte Nutzbereich (oder Bereich der potenziellen Algorithmusnutzung) ist der Bereich, in dem die Suchphase des Filteralgorithmus dominant ist und der in der Verifikationsphase entstehende Aufwand insgesamt vernachlässigbar gering ist. Für Fehlerlevel in diesem Bereich erreicht der Algorithmus Pk1 also die Laufzeit des in der Suchphase (exakte Suche!) verwendeten Algorithmus.

Der Algorithmus GraI erweitert den Algorithmus Pk1, wobei jedoch bei beliebigem Fehlerlevel die allgemeine Näherung $O(kn\log_\sigma(m)/\sigma)$ für die durchschnittliche Laufzeit

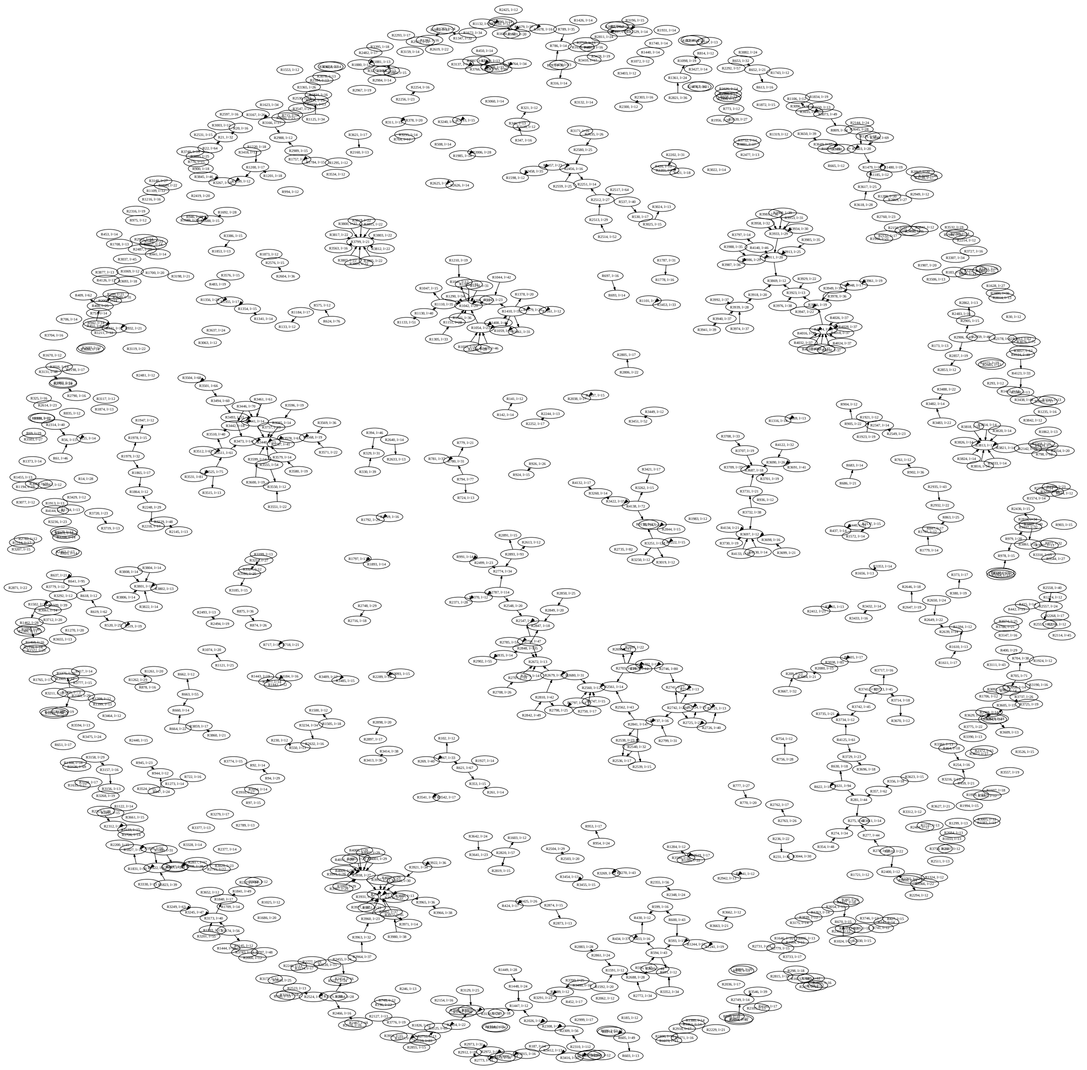


Abbildung 8.45: Regeln, die sich bei Verwendung des Programmquelltextes ergeben. Von den insgesamt 4146 (durchnummerierten) Regeln der Grammatik erreichen die gezeigten 1119 eine Länge von mindestens 12.

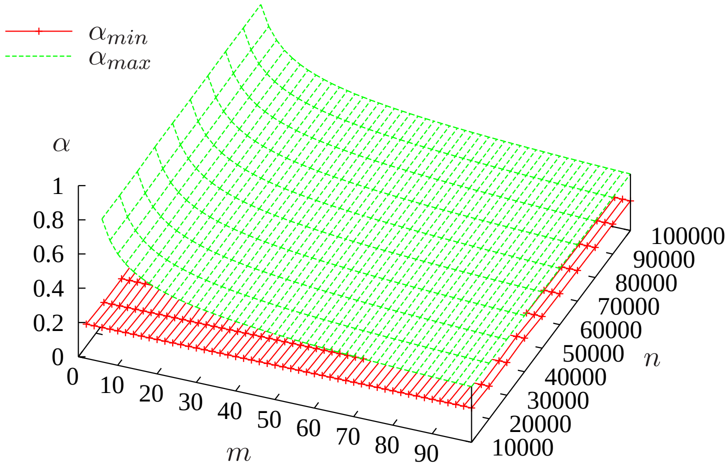


Abbildung 8.46: Nutzbereich bei einem Programm-Quellcode. Der Nutzbereich existiert für die Parametersätze, für die $\alpha_{max} \geq \alpha_{min}$ ist.

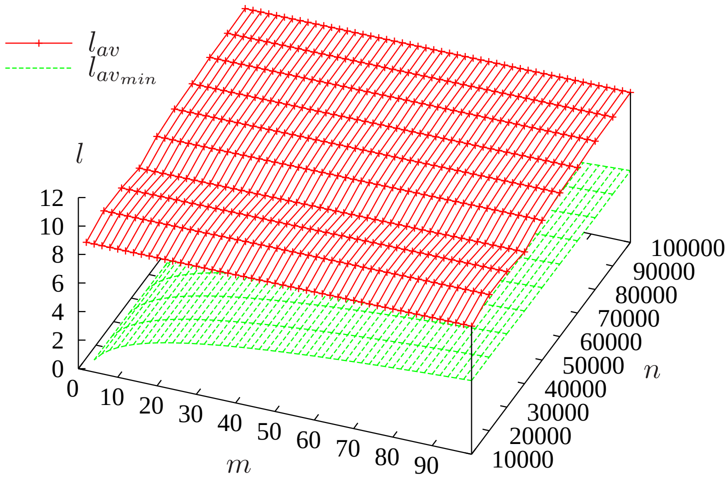


Abbildung 8.47: Nutzbereich bei einem Programm-Quellcode. Der Nutzbereich existiert für die Parametersätze, für die $l_{av} \geq l_{av_{min}}$ ist.

[85] weiterhin gültig bleibt (bei Verwendung des Sunday-Algorithmus für die exakte Suche). Der Bedarf an Speicherplatz beträgt für GraI $O(n)$, da die Datenstrukturen der Vorberechnungsphase weiter erhalten bleiben und zudem die im Verlauf der weiteren Suche verwendeten Listen jeweils höchstens die Länge n haben können.

Die tatsächlich durch die Nutzung der Grammatik erreichten Verbesserungen in der Laufzeit hängen von der Struktur des Textes ab. Konkret zeigt GraI für Fehlerlevel $\alpha > 2/(l_{av}+1)$ eine bessere Laufzeit als Pk1. Dabei wird durch die durchschnittliche Länge l_{av} der von der Grammatik über dem Text erzeugten Regeln genau die Abhängigkeit von der Textstruktur geschaffen. So führen hochgradig strukturierte Texte, wie beispielsweise der Quellcode von Programmen, zu großen Werten von l_{av}, während unstrukturierte Texte, wie beispielsweise Zufallsdaten, sehr kleine durchschnittliche Regellängen zur Folge haben.

Mit der verbesserten Laufzeit erreicht GraI gegenüber Pk1 auch einen erweiterten Nutzbereich, also einen größeren Bereich, in dem die Suchphase Dominanz aufweist. Dieser Bereich wird bestimmt durch den Anteil des Textes $\bar{n}$, der durch Regeln (zumindest der Länge l_{av}) abgedeckt wird und läßt sich damit durch $\alpha < \frac{1}{\log_\sigma(1-\bar{n}/n)+3\log_\sigma m}$ abschätzen. Stark vereinfachend kann $\bar{n}$ durch $n - l_{av}\sigma^{l_{av}}$ ersetzt werden, wobei dann wiederum die direkte Abhängigkeit von der Struktur der Textdaten deutlich wird.

Je höher der Fehlerlevel ist, desto deutlicher wirkt sich bei GraI die Laufzeitverbesserung gegenüber Pk1 aus, jedoch beschränkt die Grenze des Nutzbereichs den Fehlerlevel. Es läßt sich eine einfache Bedingung für die durchschnittliche Regellänge l_{av} ableiten, die angibt, wann auf jeden Fall (mit verschärfter Grenze des Nutzbereichs) GraI im Nutzbereich ein besseres Laufzeitverhalten aufweisen kann als Pk1: $l_{av} \geq 6\log_\sigma m - 1$.

In der Praxis zeigt sich dann, dass übereinstimmend mit der Theorie GraI den Vorteil der hohen Geschwindigkeit (im Vergleich zu anderen Algorithmen) von Pk1 wahren und unter den entsprechenden Bedingungen verbessern kann.

Die Sequitur-Grammatik, auf der GraI aufbaut, wird inkrementell konstruiert, d. h. wenn der Text um einen Buchstaben verlängert wird, kann auch die Grammatik einfach um diesen Buchstaben erweitert werden. Die Berechnung der für GraI nötigen Daten aus der Grammatik (in der Vorberechnungsphase) könnte dahingehend noch in den Grammatikaufbau integriert werden, wobei dann allerdings sorgfältig auf die einfache Erweiterbarkeit der verwendeten Arrays geachtet werden muss. Die so erhaltene Veränderung würde GraI zwar weiterhin einen Offline-Algorithmus sein lassen, im speziellen Fall aber, dass der Eingabetext nur am Ende ergänzt wird, würde nach dem erstmaligen Grammatikaufbau jede weitere Textergänzung „online“ behandelt werden können.

9 Zusammenfassung und Ausblick

> Trübes Wasser wird klar, wenn man es ruhig stehen lässt – genauso kann man mit Ruhe, Geduld und Zeit die Wahrheit nach und nach klar ans Licht treten lassen.
>
> *(Laotse, chinesischer Philosoph)*

In dieser Arbeit wurden Algorithmen für das Approximate String Matching diskutiert. Bereits existierende Algorithmen wurden besprochen, wobei auf die Algorithmen, die das Filterprinzip nutzen, besonderes Augenmerk gerichtet wurde. Für diese Algorithmen wurde ein Klassifikationsschema vorgestellt. Eine der Klassen dieses Schemas gewinnt dadurch besondere Bedeutung, dass die in der Praxis schnellsten Algorithmen zu dieser Klasse gehören. Dieses Fundament nutzen die beiden im weiteren Verlauf der Arbeit vorgestellten Algorithmen *Patchwork-Verifikation* und *GraI*, um eine beschleunigte Lösung des Problems zu erreichen. In beiden Fällen basiert die erreichte Beschleunigung auf der Idee, Redundanzen geschickt auszunutzen. Es zeigte sich, dass dieses Vorgehen in den Fällen, in denen die betrachteten Redundanzen existieren, hervorragend funktioniert. Aber auch dann, wenn keine Redundanzen existieren, sind die entwickelten Algorithmen nicht von Nachteil, da sie auf das Verhalten eines bestehenden schnellen Algorithmus zurückfallen.

Die Patchwork-Verifikation betrachtet Redundanzen auf algorithmischer Ebene. Konkret werden Doppelbetrachtungen bei der Verifikation vermieden, die bei Filteralgorithmen insbesondere bei höheren Fehlerleveln häufiger auftreten können. Das Prinzip der Patchwork-Verifikation ist ein allgemeiner Ansatz für die Verifikation, der nicht in die Suchphase des Filteralgorithmus integriert ist, wie das bei alternativen Ansätzen oft der Fall ist. Die Patchwork-Verifikation liefert wie die vollständige Verifikation immer alle Positionen der Approximate Matchings im betrachteten Bereich und bleibt dadurch – anders als inkrementell arbeitende Verfahren – flexibel einsetzbar.

Im Rahmen einer ausführlichen Analyse der Patchwork-Verifikation wurde die Bedingung berechnet, ab der die Patchwork-Verifikation gegenüber der einfachen (vollständigen) Verifikation des Verifikationsbereichs ein abweichendes und zugleich besseres Verhalten aufweist. Weiterhin wurde der für Filteralgorithmen extrem wichtige Nutzbereich bestimmt, in dem die Verifikation insgesamt nur einen in der Textlänge linearen Anteil am Gesamtaufwand hat. Ebenso konnte die minimale Patterngröße bestimmt werden, von der an der Einsatz der Patchwork-Verifikation erst sinnvoll ist. Dem folgend wurde auch die Patterngröße bestimmt, die im Sinne der (in Bezug auf den Fehlerlevel) flexibelsten Einsetzbarkeit der Patchwork-Verifikation ideal ist.

Ein Vergleich mit dem Verfahren der Hierarchischen Verifikation, welches einen der Patchwork-Verifikation ähnlichen Zweck verfolgt, zeigt die klaren Stärken der Patchwork-Verifikation bei mittleren bis höheren Fehlerleveln. Dies wurde wie alle anderen Ergebnisse auch anhand der Implementierung an praktischen Beispielen bestätigt.

Die Idee hinter GraI ist die Betrachtung von Redundanzen im Eingabetext. Konkret wird eine aus dem Text abgeleitete Grammatik verwendet, um mehrfach auftretende Textteile, die durch Regeln der Grammatik identifiziert werden, nur einfach abzuarbeiten. Vor dem Hintergrund der beabsichtigten Nutzung wurden verschiedene Grammatiken geprüft, wobei sich die Sequitur-Grammatik als geeignet herausstellte. Als vernünftige Basis für die Nutzung der Grammatik beim Approximate String Matching stellte sich der Filteransatz heraus, der Stücke des gesuchten Patterns exakt im Text sucht. Dieses zugrunde liegende Filterprinzip gilt als sehr schnell in der Praxis und ist zudem der Grund dafür, dass GraI dann ebenso in dieselbe Klasse von Filteralgorithmen einzuordnen ist.

Um aus der Grundidee einen praktisch einsetzbaren Algorithmus GraI zu entwickeln, mussten aus der Grammatik noch Zusatzinformationen generiert werden. Hierzu wurden unter anderem ein Array, das den direkten Zugriff auf alle Regeln ermöglicht, sowie eine Liste, die für jede Regel ihre Positionen im Text verwaltet, angelegt. Mit Hilfe dieser Zusatzinformationen und einiger anderer Details konnten die durch die Grammatik gegebenen Redundanzinformationen schnell und gezielt genutzt werden.

Während der ausführlichen Analyse des Algorithmus erwies sich die durchschnittliche Länge der Regeln der Grammatik als grundlegender Wert für alle weiteren Betrachtungen. Es ist gelungen, diese durchschnittliche Regellänge direkt zu berechnen, ohne dass die Grammatik selbst berechnet werden muss. Damit konnte dann theoretisch vorhergesagt werden, wann verschiedene im Rahmen der Analyse hergeleitete Bedingungen erfüllt sind. Dies gilt zum Beispiel für die Bedingung, die angibt, wann GraI gegenüber dem zugrunde liegenden Filteralgorithmus eine bessere Filtereffizienz aufweist. Die theoretisch berechnete Grenze wurde in praktischen Beispielen bestätigt.

Während der weiteren Untersuchung von GraI zeigte sich bei länger werdenden Texten ein negativer Effekt, der als die Summe zweier Schwächen der Grundidee identifiziert werden konnte. Durch geschicktes Ausbalancieren des Grammatikeinflusses im Verlauf des Algorithmus ist es gelungen, beide Schwächen zu eliminieren.

Weiterhin konnte abgeschätzt werden, wie sich bei GraI der Nutzbereich gegenüber dem zugrunde liegenden Filteralgorithmus verbessert. Damit wurde es möglich, den Parameterbereich zu bestimmen, bei dem bei GraI nicht nur die Suchphase dominant ist, sondern die Suche mit GraI auch eine geringere Laufzeit hat als mit dem zugrunde liegenden Filteralgorithmus. Anhand praktischer Experimente konnte im Vergleich demonstriert werden, dass GraI unter den aufgezeigten Bedingungen klar schneller arbeitet als die anderen betrachteten Algorithmen.

Die im Rahmen der Analyse berechneten Bedingungen für den Nutzbereich von GraI sind abhängig von den Textdaten. Aus diesem Grund wurden verschiedene Datensätze auf die Anwendbarkeit von GraI untersucht und es wurde aufgezeigt, wo dort jeweils die Nutzungsmöglichkeiten liegen.

Im Zusammenhang mit der Realisierung der praktischen Experimente wurde auch der Algorithmus von Chang und Lampe als Verifikationsalgorithmus für GraI implementiert. Dabei konnte dieser Algorithmus noch durch die Möglichkeit eines frühzeitigen Abbruchs verbessert werden.

Die im Verlauf dieser Arbeit vorgestellten Algorithmen wurden ausführlich behandelt. Im Sinne des Approximate String Matching ist damit ein in sich geschlossenes Bild entstanden. Dennoch ergeben sich auch hier noch Erweiterungsmöglichkeiten und Perspektiven für zukünftige Arbeit:

Wie in der Analyse der Patchwork-Verifikation deutlich wurde, grenzt der erweiterte Nutzbereich genau an den Nutzbereich der Hierarchischen Verifikation. Es ist damit eine nahe liegende Möglichkeit, diese beiden Verfahren in der Praxis in einem zu vereinen, indem je nach Situation das entsprechende Verfahren Verwendung findet.

Das Spektrum der Möglichkeiten, auf GraI aufzubauen, ist nicht zuletzt auch aufgrund der Komplexität deutlich breiter als bei der Patchwork-Verifikation. So ist es möglich, den inkrementellen Aufbau der Grammatik besser zu nutzen und die Berechnung der Zusatzinformationen in die Grammatikkonstruktion zu integrieren. Der sich damit ergebende Algorithmus wäre zumindest in dem Spezialfall eines sich höchstens verlängernden Textes – was beispielsweise eine Datenbank und das Hinzufügen weiterer Daten sein könnte – ein Online-Algorithmus.

Die Unabhängigkeit von dem konkret verwendeten Verifikationsalgorithmus bietet bei GraI die Möglichkeit, einen je nach Situation idealen Algorithmus zu wählen. Unter den entsprechenden (im Rahmen der Analyse der Patchwork-Verifikation aufgezeigten) Bedingungen kann es auch Sinn haben, die Patchwork-Verifikation mit GraI zu kombinieren.

Die Einsatzmöglichkeiten einer Grammatik beim Approximate String Matching im Allgemeinen wurden hier ausführlich diskutiert. Dabei zeigte sich der hier umgesetzte Algorithmus GraI als die logische Konsequenz des Gedankens der Nutzung einer Grammatik beim „exakten“ Approximate String Matching. Eine weitere Möglichkeit jedoch besteht darin, einen „inexakten“ Ansatz zu verfolgen. So ist es gerade bei biologischen Anwendungen oft nicht so wichtig, jedes Approximate Matching selbst zu entdecken. Vielmehr ist dann die Identifizierung von Bereichen bzw. Datenbankeinträgen mit höherer Ähnlichkeit zum Suchwort gewünscht, die dann später mit anderen Verfahren separat untersucht werden. Aufgrund des hohen Datenaufkommens ist dann bei der Suche die Geschwindigkeit der wichtigste Faktor. Dazu könnte die Grammatik, die den Text beschreibt, durch das Zusammenführen ähnlicher Regeln in eine kompaktere Grammatik konvertiert werden. Diese neu gewonnene Grammatik beschriebe dann zwar nicht mehr ganz genau die Eingabe, könnte gegebenenfalls aber zu einer schnelleren Erkennung ähnlicher Bereiche genutzt werden.

Ganz allgemein in Bezug auf das Approximate String Matching scheint es sehr nützlich zu sein, eine Bibliothek der wichtigsten Filteralgorithmen aufzubauen. Für die dort zur Verwendung gespeicherten Algorithmen müssten zudem die jeweiligen Nutzungsparameter ersichtlich sein, da gerade die in der Praxis nützlichen Filteralgorithmen in der Laufzeit relativ sensibel auf die Umgebungsparameter reagieren.

Abschließend sei noch einmal die ganz zu Beginn bereits genannte Datenhierarchie betrachtet, welche den Gesamtkontext umreißt, in dem diese Arbeit über das Approximate String Matching motiviert ist:

A string of alpha/numeric characters needs to be parsed to become data;
Data needs to be interpreted to become information;
Information needs to be applied to become knowledge;
Knowledge needs to be integrated into experience to be meaningful;
And experience needs to be tested against the context of truth to become wisdom.

J. Thom Mickelson, 2001[1]

Die Schwierigkeit, große Mengen an Daten zu interpretieren oder Informationen anzuwenden, ist selbstverständlich situationsabhängig und bleibt auch weiter bestehen. Jedoch konnten mit dieser Arbeit zusätzliche Möglichkeiten zur Unterstützung bei der Durchführung des jeweiligen Schrittes zur Gewinnung von Informationen oder Wissen in dieser Hierarchie präsentiert werden. Das für die Daten- und Informationsverarbeitung wichtige Fundament hat damit an Breite gewonnen.

[1] http://exalter.net/resource/axiom.html

A Durchschnittlicher Abstand zweier exakter Matchings

Betrachtet sei eine Menge von r Pattern der gleichen Länge l und ein Text der Länge n. Weiterhin sei eine Gleichverteilung für die Buchstaben des zugrunde liegenden Alphabets Σ mit $|\Sigma| = \sigma$ angenommen, d. h. jeder Buchstabe ist an jeder Stelle eines Textes mit gleichen Wahrscheinlichkeit $1/\sigma$ vorhanden. Gesucht ist der durchschnittliche Abstand der Positionen im Text, an denen ein beliebiges Pattern der Menge exakt gefunden werden kann.

Die Wahrscheinlichkeit dafür, dass an einer beliebigen Textstelle beginnend ein bestimmtes Pattern genau vorkommt, beträgt offensichtlich $1/\sigma^l$, da jeder der l Buchstaben des Patterns aufeinander folgend übereinstimmen mit dem Text übereinstimmen muss. Die Wahrscheinlichkeit für das Vorkommen von einem der r Pattern beträgt dann

$$p = \frac{r}{\sigma^l}\,.$$

Die Wahrscheinlichkeit dafür, dass im Abstand 1 im Text eines dieser r Pattern vorkommt, ist also gleich p. Für den Abstand 2 bedeutet dies, dass im Abstand 1 keines der Pattern vorkommt, dafür aber danach, was also eine Wahrscheinlichkeit von $(1-p)p$ bedeutet. Im Allgemeinen ist die Wahrscheinlichkeit $pdist(i)$ für ein Vorkommen eines der r Pattern in genau dem Abstand i gleich

$$pdist(i) = (1-p)^{i-1}p\,.$$

Der durchschnittliche Abstand $dist(n)$ zwischen zwei Textpositionen, an denen jeweils eines der r Pattern exakt vorkommt, berechnet sich für einen Text der Länge n dann wie folgt:

$$\begin{aligned} dist(n) &= \sum_{i=1}^{n} i\, pdist(i) \\ &= p\sum_{i=1}^{n} i(1-p)^{i-1} \end{aligned}$$

Nun ist die Textlänge bei langen Texten nicht mehr für diesen Abstand ausschlaggebend und zur Vereinfachung sei nun ein endloser Text angenommen. Es ergibt sich dann für den durchschnittlichen Abstand *dist*:

$$\begin{aligned} dist &= \lim_{n\to\infty} p \sum_{i=1}^{n} i(1-p)^{i-1} \\ &= \lim_{n\to\infty} \frac{p}{1-p} \underbrace{\sum_{i=1}^{n} i(1-p)^{i}}_{\substack{\text{„pseudo-geometrische" Reihe} \\ \text{mit Grenzwert } \frac{1-p}{p^2} \text{ für } (1-p)<1}} \\ &= \frac{p}{1-p}\frac{1-p}{p^2} \\ &= \frac{1}{p} \end{aligned}$$

Somit ist der durchschnittliche Abstand von Textpositionen, an denen ein beliebiges Pattern einer Menge von r Pattern der Länge l exakt gefunden werden kann

$$dist = \frac{\sigma^l}{r}\,.$$

Über den Autor

Im Jahr 1993 begann er ein Studium der Informatik an der Universität Dortmund, welches er im Jahr 1999 mit dem Erreichen des Diploms (Dipl.-Inform.) beendete. Das Schwerpunktgebiet des Studiums wurde durch die Bereiche der digitalen Bildverarbeitung und des geometrischen Modellierens gebildet. Die Diplomarbeit befasste sich mit molekularen Kyptographiesystemen und damit mit einem Thema, das in den Bereich des Biocomputings eingeordnet wird. Im Anschluss an das Studium war er von 2000 bis 2004 wissenschaftlicher Mitarbeiter in der Arbeitsgruppe von Prof. Dr. Wolfgang Banzhaf am Lehrstuhl für Systemanalyse im Fachbereich Informatik an der Universität Dortmund. Dort beschäftigte er sich zunächst weiter mit den Themen des Biocomputings, aber auch mit einem anderen alternativen Computermodell, dem Quantenrechnen. Ausgehend vom Bereich des Biocomputings konzentrierte er sich schließlich auf die Bioinformatik und deren Grundlagen, woraus auch die vorliegende Arbeit entstanden ist. Im Frühjahr 2004 verbrachte er drei Monate als „visiting research student“ am Fachbereich Informatik der Memorial University of Newfoundland in St.John's, Kanada.

Veröffentlichungen

A. Leier und C. Richter und W. Banzhaf und H. Rauhe, *Cryptography with DNA binary strands*, BioSystems, 57, pp. 13–22, 2000.

C. Richter and A. Leier and W. Banzhaf and H. Rauhe, *Private and Public Key DNA Steganography*, Preliminary Proceedings of the Sixth DIMACS Workshop on DNA-Computing, Leiden, pp. 273–274, 2000.

C. J. Richter und W. Banzhaf, *Patchwork verification in a filtering approach to approximate pattern matching*, eingereicht beim Journal of Discrete Algorithms, 2004.

C. J. Richter und W. Banzhaf, *A filtering algorithm for approximate pattern matching with reduced verification*, eingereicht beim Journal of Discrete Algorithms, 2004.

Literaturverzeichnis

[1] A. V. Aho und M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search*, Communications of the ACM, 16 (6), pp. 333–340, Juni 1975.

[2] L. Allison, D. Powell, und T. I. Dix, *Compression and Approximate Matching*, The Computer Journal, 42 (1), pp. 1–10, 1999.

[3] L. Allison, T. Edgoose, und T. I. Dix, *Compression of Strings with Approximate Repeats*, in Proceedings of the Sixth International Conference on Intelligent Systems for Molecular Biology (ISMB'98), J. Glasgow, T. Littlejohn, F. Major, R. Lathrop, D. Sankoff, und C. Sensen (Hrsg.), pp. 8–16, AAAI Press, California, USA, Juni 1998.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, und D. J. Lipman, *Basic Local Alignment Search Tool*, Journal of molecular biology, 215, pp. 403–410, 1990.

[5] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, und D. J. Lipman, *Gapped BLAST and PSI-BLAST: a new generation of protein database search programs*, Nucleic acids research, 25, pp. 3389–3402, 1997.

[6] D. Angluin und C. H. Smith, *Inductive Inference: Theory and Methods*, ACM Computing Surveys, 15 (3), pp. 237–269, 1983.

[7] A. Apostolico, M. Crochemore, Z. Galil, und U. Manber (Hrsg.), *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM'92)*, LNCS, Vol. 644, Tucson, Arizona, USA, Springer, April/Mai 1992.

[8] A. Apostolico und C. Guerra, *The longest common subsequence problem revisited*, Algorithmica, 2, pp. 315–336, 1987.

[9] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, und I. A. Faradžev, *On economical construction of the transitive closure of an oriented graph*, Soviet mathematics - doklady, 11 (5), pp. 1209–1210, 1970. Russisches Original in: Doklady Akademii Nauk SSSR, 194(3):487–488, 1970.

[10] M. Atallah, Y. Genin, und W. Szpankowski, *Pattern matching image compression: Algorithmic and empirical results*, IEEE Trans. Pattern Analysis and Machine Intelligence, 21 (7), pp. 618–627, 1999.

[11] R. Baeza-Yates und B. Ribeiro-Neto (Hrsg.), *Modern Information Retrieval*, Addison Wesley Longman Publishing Co. Inc., 1999.

[12] R. A. Baeza-Yates und G. H. Gonnet, *A new apporach to text searching*, Communications of the ACM, 35 (10), pp. 74–82, 1992. Vorfassung in ACM SIGR'89.

[13] R. A. Baeza-Yates und G. H. Gonnet, *Fast String Matching with Mismatches*, Information and Computation, 108 (2), pp. 187–199, 1994.

[14] R. A. Baeza-Yates und G. Navarro, *Faster approximate string matching*, Algorithmica, 23 (2), pp. 127–158, 1999. Vorfassungen in Proceedings of CPM'96 (LNCS, Vol. 1075, 1996) und in Proceedings of WSP'96, Carleton Uni. Press, 1996.

[15] R. A. Baeza-Yates und C. H. Perleberg, *Fast and Practical Approximate String Matching*, in Apostolico et al. [7], pp. 185–192.

[16] R. A. Baeza-Yates und C. H. Perleberg, *Fast and practical approximate string matching*, Information Processing Letters, 59 (1), pp. 21–27, 1996. Vorfassung in CPM'92, LNCS 644, 1992.

[17] R. E. Bellman, *Dynamic Programming*, Princeton University Press, 1957.

[18] R. S. Boyer und J. S. Moore, *A Fast String Searching Algorithm*, Communications of the ACM, 20, pp. 762–772, 1977.

[19] S. Burkhardt, *Filter Algorithms for Approximate String Matching*, PhD thesis, Department of Computer Science, Saarland University, Germany, 2002.

[20] W. I. Chang und J. Lampe, *Theoretical and Empirical Comparisons of Approximate String Matching Algorithms*, in Apostolico et al. [7], pp. 175–184.

[21] W. I. Chang und E. L. Lawler, *Sublinear Approximate String Matching and Biological Applications*, Algorithmica, 12, pp. 327–344, 1994. Vorfassung in FOCS'90.

[22] W. I. Chang und T. G. Marr, *Approximate String Matching and Local Similarity*, in Crochemore und Gusfield [29], pp. 259–273.

[23] C. Charras und T. Lecroq, *Handbook of Exact String Matching Algorithms*, King's College London Publications, 2004.

[24] N. A. Chomsky, *Three Models for the Description of Language*, IRE Transactions on Information Theory, 2, pp. 113–124, 1956.

[25] A. L. Cobbs, *Fast Approximate Matching using Suffix Trees*, in Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM'95), Z. Galil und E. Ukkonen (Hrsg.), LNCS, Vol. 937, pp. 41–54, Springer, Juli 1995.

[26] W. W. Cohen, P. Ravikumar, und S. E. Fienberg, *A Comparison of String Distance Metrics for Name-Matching Tasks*, in IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03), S. Kambhampati und C. A. Knoblock (Hrsg.), pp. 73–78, August 2003.

[27] R. Cole und R. Hariharan, *Approximate string matching: a simpler faster algorithm*, in *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms (SODA'98)*, pp. 463–472, ACM Press, Januar 1998.

[28] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, und W. Rytter, *Speeding Up Two String-Matching Algorithms*, Algorithmica, 12, pp. 247–267, 1994.

[29] M. Crochemore und D. Gusfield (Hrsg.), *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM'94)*, LNCS, Vol. 807, Asilomar, California, USA, Springer, Juni 1994.

[30] F. J. Damerau, *A technique for computer detection and correction of spelling errors*, Communications of the ACM, 7 (3), pp. 171–176, März 1964.

[31] G. Das, R. Fleischer, L. Gąsieniec, D. Gunopulos, und J. Kärkkäinen, *Episode Matching*, in Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM'97), A. Apostolico und J. Hein (Hrsg.), LNCS, Vol. 1264, pp. 12–27, Springer, Juni 1997.

[32] M. O. Dayhoff, R. V. Eck, und C. M. Park, *A Model of Evolutionary Change in Proteins*, pp. 89–99, National biomedical research foundation, Washington, DC., USA, 1972.

[33] M. O. Dayhoff, R. M. Schwartz, und B. C. Orcutt, *A Model of Evolutionary Change in Proteins*, pp. 345–352, National biomedical research foundation, Washington, DC., USA, 1978. Supplement 3.

[34] S. Dreyfus, *Richard Bellman on the Birth of Dynamic Programming*, Operations Research, 50 (1), pp. 48–51, 2002.

[35] R. Durbin, S. Eddy, A. Krogh, und G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*, Cambridge University Press, 1998.

[36] D. G. Elliman und I. T. Lancaster, *A review of segmentaion and contextual analysis techniques for text recognition*, Pattern Recognition, 23 (3/4), pp. 337–346, 1990.

[37] P. Gage, *A new algorithm for data compression*, The C Users Journal, 12 (2), pp. 23–38, 1994.

[38] Z. Galil und R. Giancarlo, *Data Structures and Algorithms for Approximate String Matching*, Journal of Compexity, 4, pp. 33–72, 1988.

[39] Z. Galil und K. Park, *An Improved Algorithm for Approximate String Matching*, SIAM Journal on Computing, 19 (6), pp. 989–999, 1990. Vorfassung in ICALP'89 (LNCS, Vol. 372, 1989).

[40] F. Galisson, *The Fasta and Blast programs*, 2000. Tutorial - Eighth International Conference on Intelligent Systems for Molecular Biology (ISMB 2000).

[41] F. Galisson, *Pairwise sequence comparison*, 2000. Tutorial - Eighth International Conference on Intelligent Systems for Molecular Biology (ISMB 2000).

[42] R. Giancarlo und D. Sankoff (Hrsg.), *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM'00)*, LNCS, Vol. 1848, Montreal, Canada, Springer, Juni 2000.

[43] A. J. Gibbs und G. A. McIntyre, *The Diagram, a method for Comparing Sequences*, European Journal of Biochemistry, 16, pp. 1–11, 1970.

[44] R. Giegerich, F. Hischke, S. Kurtz, und E. Ohlebusch, *Static and Dynamic Filtering Methods for Approximate String Matching*, Tech. Rep. 96-01, Technische Fakultät, Abteilung Informationstechnik, Universität Bielefeld, 1996.

[45] R. Giegerich, F. Hischke, S. Kurtz, und E. Ohlebusch, *A General Technique to Improve Filter Algorithms for Approximate String Matching*, in Proceedings of the Fourth South American Workshop on String Processing (WSP'97), R. Baeza-Yates (Hrsg.), pp. 38–52, Carleton University Press, November 1997.

[46] O. Gotoh, *An Improved Algorithm for Matching Biological Sequences*, Journal of Molecular Biology, 162, pp. 705–708, 1982.

[47] R. Grossi und F. Luccio, *Simple and efficient string matching with k mismatches*, Information Processing Letters, 33 (3), pp. 113–120, 1989.

[48] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[49] P. A. V. Hall und G. R. Dowling, *Approximate String Matching*, ACM Computing Surveys, 12 (4), pp. 381–402, 1980.

[50] D. Harel und R. E. Tarjan, *Fast Algorithms for Finding Nearest Common Ancestors*, SIAM Journal on Computing, 13 (2), pp. 338–355, 1984.

[51] W. Henning, *Genetik*, Springer-Verlag, Berlin, 1995.

[52] D. S. Hirschberg, *Algorithms for the Longest Common Subsequence Problem*, Journal of the ACM, 24 (4), pp. 664–675, 1977.

[53] J. E. Hopcroft und J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Massachusetts, 1979.

[54] R. N. Horspool, *Practical Fast Searching in Strings*, Software—Practice and Experience, 10, pp. 501–506, 1980.

[55] M. A. Jaro, *Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida*, Journal of the American Statistical Association, 84, pp. 414–420, 1989.

[56] M. A. Jaro, *Probabilistic Linkage of Large Public Health Data Files (disc:P687-689)*, Statistics in Medicine, 14, pp. 491–498, 1995.

[57] E. T. Jaynes, *Probability Theory : The Logic of Science*, Cambridge University Press, 2003.

[58] P. Jokinen, J. Tarhio, und E. Ukkonen, *A Comparison of Approximate String Matching Algorithms*, Software—Practice and Experience, 26 (12), pp. 1439–1458, Dezember 1996.

[59] P. Jokinen und E. Ukkonen, *Two algorithms for approximate string matching in static texts*, in Proceedings of 16th International Symposium on Mathematical Foundations of Computer Science (MFCS'91), A. Tarlecki (Hrsg.), LNCS, Vol. 520, pp. 240–248, Springer-Verlag, Berlin, September 1991.

[60] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, und S. Arikawa, *A Unifying Framework for Compressed Pattern Matching*, in *Proceedings of the 6th International Symposium on String Processing and Information Retrieval (SPIRE'99)*, pp. 89–96, IEEE Computer Society Press, September 1999.

[61] D. E. Knuth, J. H. Morris, jr., und V. R. Pratt, *Fast Pattern Matching in Strings*, SIAM Journal on Computing, 6 (2), pp. 323–350, Juni 1977.

[62] J. Kärkkäinen, G. Navarro, und E. Ukkonen, *Approximate String Matching over Ziv-Lempel Compressed Text*, in Giancarlo und Sankoff [42], pp. 195–209.

[63] K. Kukich, *Techniques for Automatically Correcting Words in Text*, ACM Computing Surveys, 24 (4), pp. 377–439, 1992.

[64] S. Kurtz, *Approximate String Searching under Weighted Edit Distance*, in Ziviani et al. [141], pp. 156–170.

[65] G. M. Landau, E. W. Myers, und J. P. Schmidt, *Incremental String Comparison*, SIAM Journal on Computing, 27 (3), pp. 557–582, 1998.

[66] G. M. Landau und U. Vishkin, *Fast String Matching with k Differences*, Journal of Computer and System Sciences, 37, pp. 63–78, 1988. Vorfassung in FOCS'85.

[67] G. M. Landau und U. Vishkin, *Fast Parallel and Serial Approximate String Matching*, Journal of Algorithms, 10, pp. 157–169, 1989. Vorfassung in ACM STOC'85.

[68] P. Langley, *Simplicity and Representation Change in Grammar Induction*, Tech. rep., Robotics Laboratory, Computer Science Department, Stanford University, Stanford, CA, USA, 1995.

[69] N. J. Larsson und A. Moffat, *Offline Dictionary-Based Compression*, in Proceedings of the IEEE Data Compression Conference (DCC '99), J. A. Storer und M. Cohn (Hrsg.), pp. 296–305, IEEE Computer Society Press, Los Alamitos, CA, USA, März 1999.

[70] V. I. Levenshtein, *Binary Codes Capable of Correcting Deletions, Insertions and Reversals*, Soviet Physics - Doklady, 10 (8), pp. 707–710, 1966. Russisches Original in *Doklady Akademii Nauk SSSR*, 163, 4, 845–848, 1965.

[71] H. Lodish, A. Berk, S. L. Zipursky, P. Matsudaira, D. Baltimore, und J. E. Darnell, *Molecular Cell Biology*, W.H.Freeman & Co, Yew York, USA, fourth edition ed., 2000.

[72] T. Luczak und W. Szpankowski, *A lossy data compression based on string matching: Preliminary Analysis and Suboptimal algorithms*, in Crochemore und Gusfield [29], pp. 102–112.

[73] T. Luczak und W. Szpankowski, *A Suboptimal Lossy Data Compression Based on Approximate Pattern Matching*, IEEE Transactions on Information Theory, 43, pp. 1439–1451, 1997.

[74] W. J. Masek und M. S. Paterson, *A Faster Algorithm for Computing String Edit Distances*, Journal of Computer and System Sciences, 20, pp. 18–31, 1980.

[75] W. J. Masek und M. S. Paterson, *How to Compute String-Edit Distances Quickly*, in Sankoff und Kruskal [104], 1983.

[76] M. Matsumoto und T. Nishimura, *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulation, 8 (1), pp. 3–30, 1998.

[77] T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, und S. Arikawa, *Bit-Parallel Approach to Approximate String Matching in Compressed Texts*, in *Proceedings of the 7th*

International Symposium on String Processing and Information Retrieval (SPIRE'00), pp. 221–228, IEEE CS Press, September 2000. Vorfassung in Tech. Rep. DOI-TR-174, Dept. of Informatics, Kyushu University, 2000.

[78] G. Mittenhuber, *Phylogenomics — ein neues Arbeitsfeld in der Postgenomära*, BIOspektrum, 8 (5), pp. 616–618, 2002.

[79] V. Mäkinen, G. Navarro, und E. Ukkonen, *Approximate Matching of Run-Length Compressed Strings*, Algorithmica, 35, pp. 347–369, 2003.

[80] K. P. Murphy, *Biological Sequence Comparison: An Overview of Techniques*, Mai 1994. `http://citeseer.nj.nec.com/murphy94biological.html`.

[81] E. W. Myers, *An O(ND) difference algorithm and its variations*, Algorithmica, 1, pp. 251–266, 1986.

[82] E. W. Myers, *An Overview of Sequence Comparison Algorithms in Molecular Biology*, Tech. Rep. TR91-29, Dept. of Computer Science, U. of Arizona, Tucson, AZ, 1991.

[83] E. W. Myers, *A Sublinear Algorithm for Approximate Keyword Searching*, Algorithmica, 12, pp. 345–374, 1994. Vorversion in Tech. Rep. TR-90-25, Dept. of Computer Science, Univ. of Arizona, 1990.

[84] G. Navarro, *Approximate Text Searching*, PhD thesis, Department of Computer Science, University of Chile, Santiago, Chile, Dezember 1998.

[85] G. Navarro, *A Guided Tour to Approximate String Matching*, ACM Computing Surveys, 33 (1), pp. 31–88, 2001.

[86] G. Navarro und R. Baeza-Yates, *A Practical q-Gram Index for Text Retrieval Allowing Errors*, CLEI Electronic Journal, 1 (2), 1998. Vorversion in Proceedings of the XXII Latin American Conference on Informatics (CLEI'97).

[87] G. Navarro und R. Baeza-Yates, *Very Fast and Simple Approximate String Matching*, Information Processing Letters (IPL), (72), pp. 65–70, 1999.

[88] G. Navarro und R. Baeza-Yates, *A Hybrid Indexing Method for Approximate String Matching*, Journal of Discrete Algorithms, 1 (1), pp. 205–239, 2000. Vorversion in CPM'99.

[89] G. Navarro und R. Baeza-Yates, *Improving an Algorithm for Approximate Pattern Matching*, Algorithmica, 30, pp. 473–502, 2001. Vorversion in Tech. Rep. TR/DCC-98-5, Dept. of Computer Science, University of Chile.

[90] G. Navarro, R. Baeza-Yates, E. Sutinen, und J. Tarhio, *Indexing Methods for Approximate String Matching*, IEEE Data Engineering Bulletin, 24 (4), pp. 19–27, 2001. Special issue on Managing Text Natively and in DBMSs. Invited paper.

[91] G. Navarro, T. Kida, M. Takeda, A. Shinohara, und S. Arikawa, *Faster Approximate String Matching over Compressed Text*, in *Proceedings of the 11th IEEE Data Compression Conference (DCC '01)*, pp. 459–468, IEEE Computer Society Press, Los Alamitos, CA, USA, März 2001.

[92] G. Navarro und M. Raffinot, *Fast and Flexible String Matching by Combining Bit-Parallelism and Suffix Automata*, ACM Journal of Experimental Algorithmics (JEA), 5 (4), 2000. Vorversion in Proceeedings of CPM'98. LNCS, Springer Verlag, New York.

[93] G. Navarro, E. Sutinen, J. Tanninen, und J. Tarhio, *Indexing Text with Approximate q-grams*, in Giancarlo und Sankoff [42], pp. 350–363.

[94] S. B. Needleman und C. D. Wunsch, *A general Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins*, Journal of Molecular Biology, 48, pp. 443–453, 1970.

[95] C. G. Nevill-Manning, *Inferring Sequential Structure*, PhD thesis, Department of Computer Science, University of Waikato, New Zealand, Mai 1996.

[96] C. G. Nevill-Manning und I. H. Witten, *Compression and explanation using hierarchical grammars*, Computer Journal, 40 (2/3), pp. 103–116, 1997.

[97] C. G. Nevill-Manning und I. H. Witten, *Identifying Hierarchial Structures in Sequences: A Linear-time Algorithm*, Journal of Artificial Intelligence Research, 7, pp. 67–82, 1997.

[98] W. R. Pearson, *Protein sequence comparison and Protein evolution*, 2000. Tutorial - Eighth International Conference on Intelligent Systems for Molecular Biology (ISMB 2000).

[99] W. R. Pearson und D. J. Lipman, *Improved tools for biological sequence comparison*, Proceedings of the National Academy of Sciences of the United States of America (PNAS), 85, pp. 2444–2448, 1988.

[100] C. J. Richter und W. Banzhaf, *A filtering algorithm for approximate pattern matching with reduced verification*, 2004. Eingereicht beim Journal of Discrete Algorithms.

[101] C. J. Richter und W. Banzhaf, *Patchwork verification in a filtering approach to approximate pattern matching*, 2004. Eingereicht beim Journal of Discrete Algorithms.

[102] R. L. Rivest, *Practical-match retrieval algorithms*, SIAM Journal on Computing, 5 (1), pp. 19–50, 1976.

[103] S. C. Sahinalp und U. Vishkin, *Approximate Pattern Matching Using Locally Consistent Parsing*, 1997. Manuscript, University of Maryland Institute for Advanced Computer Studies (UMIACS), 1997. Extended abstract erschienen in Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science (FOCS'96), Oct. 1996, Burlington, Vermont, USA, pp.320–328.

[104] D. Sankoff und J. B. Kruskal (Hrsg.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, Massachusetts, 1983.

[105] D. Sankoff und S. Mainville, *Common Subsequences and Monotone Subsequences*, in Sankoff und Kruskal [104], 1983.

[106] B. Schieber und U. Vishkin, *On Finding Lowest Common Ancestors: Simplification and Parallelization*, SIAM Journal on Computing, 17 (6), pp. 1253–1262, Dezember 1988.

[107] K. U. Schulz, *Nachkorrektur von Ergebnissen einer optischen Charaktererkennung*, 2003. Skript für ein Hauptseminar an der Ludwig-Maximilians-Universität, München.

[108] P. H. Sellers, *On the Theory and Computation of Evolutionary Distances*, SIAM Journal on Applied Mathematics, 26 (4), pp. 787–793, Juni 1974.

[109] P. H. Sellers, *The Theory and Computation of Evolutionary Distances: Pattern Recognition*, Journal of Algorithms, 1, pp. 359–373, 1980.

[110] C. E. Shannon, *A Mathematical Theory of Communication*, Bell System Technical Journal, 27, pp. 379–423, 623–656, 1948.

[111] F. Shi, *Fast approximate string matching with q-blocks sequences*, in Ziviani et al. [141], pp. 257–271.

[112] T. F. Smith und M. S. Waterman, *Identification of Common Molecular Subsequences*, Journal of Molecular Biology, 147, pp. 195–197, 1981.

[113] A. Stolcke und S. Omohundro, *Inducing Probabilistic Grammars by Bayesian Model Merging*, in Grammatical Inference and Applications (ICGI-94), R. C. Carrasco und J. Oncina (Hrsg.), LNCS, Vol. 862, pp. 106–118, Springer-Verlag, Berlin, 1994.

[114] B. Stroustrup, *The C++ Programming Language/Special Edition*, Addison-Wesley, Reading, Massachusetts, 2000.

[115] D. M. Sunday, *A very fast substring search algorithm*, Communications of the ACM, 33 (8), pp. 132–142, August 1990.

[116] D. M. Sunday, R. Baeza-Yates, F. T. Krogh, B. Ziegler, und P. R. Sibbald, *Technical Correspondence - Notes on a very fast substring search algorithm*, Communications of the ACM, 35 (4), pp. 132–137, April 1992.

[117] E. Sutinen, *Approximate Pattern Matching with the q-gram Family*, PhD thesis, Department of Computer Science, University of Helsinki, Finland, 1998. Tech. Rep. TR A-1998-3.

[118] E. Sutinen und J. Tarhio, *On Using q-Gram Locations in Approximate String Matching*, in Proceedings of Third Annual European Symposium on Algorithms (ESA'95), P. Spirakis (Hrsg.), LNCS, Vol. 979, pp. 327–340, Springer-Verlag, Berlin, September 1995.

[119] E. Sutinen und J. Tarhio, *Filtration with q-Samples in Approximate String Matching*, in Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96), D. S. Hirschberg und G. Myers (Hrsg.), LNCS, Vol. 1075, pp. 50–63, Springer, Juni 1996.

[120] T. Takaoka, *Approximate Pattern Matching with Samples*, in Proceedings of ISAAC'94, D.-Z. Du und X.-S. Zhang (Hrsg.), LNCS, Vol. 834, pp. 234–242, Springer-Verlag, Berlin, 1994.

[121] J. Tarhio und E. Ukkonen, *Approximate Boyer-Moore String Matching*, SIAM Journal on Computing, 22 (2), pp. 243–260, 1993. Vorfassung in SWAT'90 (LNCS, vol. 447, 1990).

[122] E. Ukkonen, *Algorithms for Approximate String Matching*, Information and Control, 64, pp. 100–118, 1985. Vorfassung präsentiert bei der International Conference on "Foundations of Computation Theory", Sweden, Aug. 1983.

[123] E. Ukkonen, *Finding Approximate Patterns in Strings*, Journal of Algorithms, 6, pp. 132–137, 1985.

[124] E. Ukkonen, *Approximate string-matching with q-grams and maximal matches*, Theoretical Computer Science, 92, pp. 191–211, 1992.

[125] E. Ukkonen, *Approximate String-Matching over Suffix Trees*, in Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM'93), A. Apostolico, M. Crochemore, Z. Galil, und U. Manber (Hrsg.), LNCS, Vol. 684, pp. 228–242, Springer, Juni 1993.

[126] E. Ukkonen, *On-Line Construction of Suffix Trees*, Algorithmica, 14, pp. 249–260, 1995.

[127] E. Ukkonen und D. Wood, *Approximate String Matching with Suffix Automata*, Algorithmica, 10, pp. 353–364, 1993. Vorfassung in Rep. A-1990-4, Dept. of Computer Science, Univ. of Helsinki, Apr. 1990.

[128] K. Vanlehn und W. Ball, *A Version Space Approach to Learning Context-free Grammars*, Machine Learning, 2 (1), pp. 39–74, 1987.

[129] R. A. Wagner und M. J. Fischer, *The String-to-String Correction Problem*, Journal of the Association for Computing Machinery, 21 (1), pp. 168–173, Januar 1974.

[130] M. S. Waterman, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Chapman & Hall–CRC Press, 1995.

[131] M. S. Waterman, T. F. Smith, und W. A. Beyer, *Some Biological Sequence Metrics*, Advances in Mathematics, 20, pp. 367–387, 1976.

[132] T. A. Welch, *A Technique for High Performance Data Compression*, IEEE Computer Magazine, 17 (6), pp. 8–19, 1984.

[133] W. E. Winkler, *The State of Record Linkage and Current Research Problems*, Report RR99/04, U.S. Bureau of the Census, Statistical Research Division, 1999.

[134] J. G. Wolff, *An Algorithms for the Segmentation of an Artificial Language Analogue*, British Journal of Psychology, 66 (1), pp. 79–90, 1975.

[135] J. G. Wolff, *The Discovery of Segments in Natural Language*, British Journal of Psychology, 68, pp. 97–106, 1977.

[136] J. G. Wolff, *Language Acquisition and the Discovery of Phrase Structure*, Language and Speech, 23, part 3, pp. 255–269, 1980.

[137] S. Wu und U. Manber, *Fast text searching allowing errors*, Communications of the ACM, 35 (10), pp. 83–91, 1992.

[138] S. Wu, U. Manber, und E. W. Myers, *A subquadratic algorithm for approximate limited expression matching*, Algorithmica, 15 (1), pp. 50–67, 1996. Vorfassung als Tech. Rep. TR29-36, Computer Science Dept., Univ. of Arizona, 1992.

[139] J. Ziv und A. Lempel, *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, 23 (3), pp. 337–343, May 1977.

[140] J. Ziv und A. Lempel, *Compression of Individual Sequences via Variable-Rate Coding*, IEEE Transactions on Information Theory, 24 (5), pp. 530–536, Sept 1978.

[141] N. Ziviani, R. Baeza-Yates, und K. S. Guimarães (Hrsg.), *Proceedings of the Third South American Workshop on String Processing (WSP'96)*, International Informatics Series, Vol. 4, Recife, Brazil, Carleton University Press, August 1996.

[142] J. Zobel und P. Dart, *Finding approximate matches in large lexicons*, Software—Practice and Experience, 25 (3), pp. 331–345, Mar 1995.

[143] J. Zobel und P. Dart, *Phonetic String Matching: Lessons from Information Retrieval*, in Proceedings of the 19th annual international ACM SIGIR conference on Research and Development in Information Retrieval (SIGIR'96), H.-P. Frei, D. Harman, P. Schaübie, und R. Wilkinson (Hrsg.), pp. 166–172, ACM Press, August 1996.

Stichwortverzeichnis