Indexgestützte Textanalyse mit symmetrischen kompakten direkten azyklischen Wortgraphen

Tobias Englmeier

München 2020

Indexgestützte Textanalyse mit symmetrischen kompakten direkten azyklischen Wortgraphen

Tobias Englmeier

Inauguraldissertation
zur Erlangung des Doktorgrades der Philosophie
an der Ludwig-Maximilians-Universität

vorgelegt von
Tobias Englmeier
aus München
2020

Erstgutachter: Prof. Dr. Klaus U. Schulz
Zweitgutachter: PD Dr. Stefan Langer
Tag der mündlichen Prüfung: 10. Juli 2020

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Zusammenfassung

Ein *symmetrischer direkter azyklischer Wortgraph (SCDAWG)* ist eine komplexe String-Indexstruktur, die alle Infixe einer Textmenge und deren Inversen effizient abspeichert. In der vorliegenden Arbeit wird aufgezeigt, wie sich diese Indexstruktur für unterschiedliche Aufgaben aus dem Bereich der Textanalyse anwenden lässt.

Im ersten Teil der Arbeit werden die Relationen zwischen verschiedenen String-Indexstrukturen und ihre effiziente Berechnung behandelt. Das Ende dieses Abschnitts bildet die formale Beschreibung der SCDAWG-Struktur.

Der zweite Teil beinhaltet die ausführliche Darstellung einer eigenen Implementierung der SCDAWG-Struktur sowie Implementierungsdetails grundlegender Suchfunkionen, die die Struktur zu einer invertierten Datei erweitern.

Im dritten Teil steht das Auffinden längster gemeinsamer Teilwörter im Fokus, wobei dort verschiedene Anwendungsmöglichkeiten, die auf diesen basieren, diskutiert werden. Einen Kernpunkt stellt ein kombiniertes Verfahren zur globalen Alignierung zweier Strings dar.

Schließlich wird im letzten Teil eine Methode zur Identifikation charakteristischer minimaler Teilwörter erörtert und deren Einsatzmöglichkeit bei der überwachten Dokumentenklassifikation behandelt.

Einer Frage entspricht immer eine Methode des Findens. Oder man könnte sagen: Eine Frage bezeichnet eine Methode des Suchens.[1]

Vorwort

Der Drang, neues Wissen zu gewinnen, liegt tief verborgen in der Natur des Menschen und hat maßgeblich zur Entwicklung und zum Überleben unserer Spezies beigetragen. Während in grauer Vorzeit nur die verbale und gestikuläre Kommunikation zur Erweiterung des eigenen Wissens führen konnte, existieren seit etwa 5000 Jahren Schriftsysteme, die einen von anderen Lebewesen unabhängigen Wissenserwerb erlauben. Die anfänglichen Tontafeln aus Uruk sind zwar mittlerweile handlicheren und mit weniger Aufwand herzustellenden Papierstapeln, die wir als Bücher kennen, gewichen, am grundlegenden Konzept der persistenten Speicherung von Wissen hat sich jedoch nichts geändert. Neben dem Griff ins heimische Bücherregal steht uns seit etwa 30 Jahren mit dem World-Wide-Web eine zweite nahezu unerschöpfliche Wissensquelle zur freien Verfügung.

Die Suche nach Informationen spielt sich aber, egal welches Medium zugrunde liegt, stets gleich ab. Nachdem sich der Wunsch, den eigenen lokalen Wissensvorrat über ein bestimmtes Thema zu erweitern, konkretisiert hat, sollen möglichst schnell passende Textstellen gefunden werden, die das gesuchte Thema betreffen und so unseren Wissensdurst befriedigen. Je mehr Wissen jedoch in einem Medium gespeichert ist, desto schwieriger wird es, genau die Stellen zu finden, denen unser Interesse gilt. Damit die Zeit, die zum Finden der passenden Textstellen gebraucht wird, nicht uferlos wird, wurden bereits früh Referenzsysteme entwickelt, die einen schnellen Zugriff auf die gesuchten Stellen erlauben. Ein solches tabellarisches Referenzsystem wird auch als *Register* oder *Index* bezeichnet.

Die nächste Abbildung zeigt einen Auszug eines Index, bei dem zu allen Städtenamen Schottlands jeweils ein Verweis auf die Seitenzahlen des Buches (*Owen's New Book of Roads (1814)*) angegeben ist, sodass, wenn etwa Informationen zur Infrastruktur von Inverness gesucht werden, sofort ersichtlich ist, dass auf den Seiten 183, 185 und 190 nachzuschlagen ist. Die Zeit, die richtige Buchseite zu finden, wird damit durch die Verwendung des Index stark verkürzt und der Wissensdurst des

[1] Ludwig Wittgenstein in *Philosophische Untersuchungen, Frankfurt am Main 1981, S. 77.*

Lesers wird befriedigt, ohne dass dieser sein Vorhaben vorzeitig frustriert abbricht.

INDEX TO THE ROADS OF SCOTLAND. xxxi

	Pages.		Pages.
Dunblain	182, 188	Jedburgh	179
Dundee	185, 189	Inverary	181, 182
Dunse	188	Inverbervie	185, 189
Dysart	186	Invergordon	183
Easter-Wemys	186	Inver Inn	183
Ecclesecham	188	Inverkeithing	183
Edinburgh	175, 177, 178	Inverness	183, 185, 190
Elgin	185	Johnny Grot's	183
Ellan	186	Irvine	189
Elvanfort	187	Keith	184
Falkirk	181, 182	Kelso	178, 190
Falla	177	Kenmore	189
Fernyhall	178	Kennaway	186
Fettercairn	184, 185	Kilmarnock	188
Flyboat	179	Kilrenny	186
Fochabers	184, 185	Killin	189
Forfar	184, 189	Kilsyth	181
Forres	185	Kincardine-o-neil	184
Fort Augustus	182, 190	Kinghorn	185
Fort George	184	King's-house Inn	182
Fort William	182, 190	Kingswell	187, 188
Frazerburgh	186	Kinross	183
Galloway, New	180	Kirk Andrews	178
Galston	180, 187	Kirkcaldy	185, 186
Garvimore	182	Kirkintilloch	181
Gatehouse of Fleet	190	Kirkliston	181
General's Hut	190		

Auszug aus dem Index von *Owen's New Book of Roads (1814)*

Eng verwandt mit dem Register ist der Begriff der *Konkordanz*. Konkordanzen werden seit Langem zum Beispiel in der Bibelforschung dazu verwendet, Textpassagen zu vergleichen beziehungsweise Textpassagen ähnlichen Inhalts zu finden. Obwohl durch Indizes somit auch in analogen Werken die Zeitdauer des Suchprozesses nach den Vorkommen bestimmter Informationen stark verringert werden kann, stellt, je größer der Index wird, der manuelle Nachschlageprozess oftmals eine weitere Hürde dar.

Einem solchen Problem sah sich in den 1940er Jahren der Jesuitenmönch Roberto Busa bei der Verfassung seiner Doktorarbeit über das Werk Thomas von Aquins gegenüber. Um den dafür betriebenen Aufwand künftig zu verringern, begann Busa 1949 in Kooperation mit IBM einen elektronischen Index, den *Index Thomasticus* aufzubauen, in dem nunmehr Konkordanzsuchen auf dem indexierten Korpus Thomas von Aquins wesentlich bequemer und schneller erledigt werden konnten. Wenngleich die Erfolgsgeschichte von Roberto Busa oft auch als Initialzündung der *Digitalen Geisteswissenschaften* oder *Digital-Humanities* gilt [43], bedurfte es noch einiger Zeit, bis der ursprünglich auf Lochkarten gespeicherte Index Thomasticus Nachahmer fand, beziehungsweise elektronische Indexsuchen für ein breites Publikum verfügbar wurden. Als Grundkonzept elektronischer Indizes kann der von Ted Nelson [55] geprägte Begriff des *Hypertexts* gesehen werden, in dem nun *Links* das automatische Springen zu einer anderen Textstelle übernehmen. Dieses neue Konzept, das die lineare Lesereihenfolge eines Textes aufbricht, führte

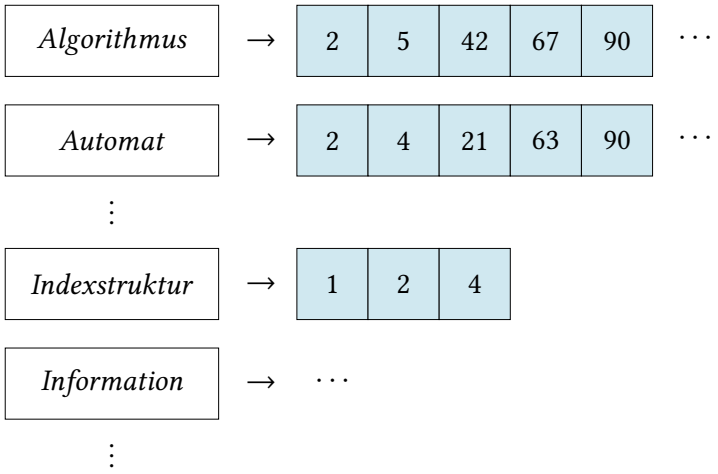
schließlich zur Entwicklung des *Hypertext Transfer Protocolls (HTTP)* durch den britischen Informatiker und Physiker Sir Tim Berners Lee, dessen Spezifikation den Grundbaustein des modernen World-Wide-Webs bildet. Als Inhaltsstruktur eines Hypertext-Dokuments dient die ebenfalls von Berners Lee entwickelte Auszeichnungssprache *Hypertext Markup Language (HTML)*.

Durch die rasante Entwicklung des WWWs begünstigt sind mittlerweile fast alle indexbasierten Suchsysteme über das Netz abrufbar. Somit war es nur eine Frage der Zeit, bis auch der Index Thomasticus anhand eines bequemen Web-Interfaces [6] einem jeden Interessenten frei zur Verfügung gestellt wurde.

Invertierte Dateien

Entwicklungen solcher Suchmaschinen werden dem Fachbereich des *Information Retrieval* zugeordnet. Das Ziel eines solchen IR-Systems ist es damit, Benutzern dabei zu helfen, die gesuchten Informationen effektiv aufzufinden [3].

Da der Index eines IR-Systems immer von den indexierten Wörtern auf die Stellen zurückweist, an welchen diese in einem Werk einer Textsammlung auftreten, wird im Kontext des IR oft vom *invertierten Index* oder von *invertierten Dateien* gesprochen. Eine solche invertierte Datei besteht aus einem *Lexikon* oder *dictionary*, das alle *keywords*, also die Wörter, nach denen gesucht werden kann, enthält. Von jedem keyword existiert ein Pointer, der auf eine Liste, die meist als *invertierte Liste* oder *postings list* bezeichnet wird, zeigt. Die Einträge der invertierten Listen werden *postings* genannt und geben an, in welchen Dokumenten ein keyword auftritt. [48] Die nächste Grafik zeigt ein Beispiel einer invertierten Datei.



Beispiel einer invertierten Datei.

Im Beispielindex würden für das keyword *Algorithmus* die Dokumente 2,5,42,67 und 90 gefunden, für das Suchwort *Indexstruktur* die Dokumente mit der Nummer 1,2 und 4. An der Länge der invertierten Listen lässt sich die *document frequency* ablesen, mit der ein keyword in der Dokumentenkollektion auftritt [48]. Die document frequency des Wortes *Indexstruktur* beträgt etwa im Beispiel 3. Neben einem solchen Index, der zu einer Suchanfrage lediglich die Nummern oder Ids der Dokumente, in denen ein keyword auftritt, liefert, werden häufig auch andere Informationen innerhalb der Postinglisten gespeichert. In [15] werden neben der hier gezeigten Variante noch der *Häufigkeitsindex*, der *Positionsindex* und *schemaunabhängige Indizes* unterschieden. Die hier beschriebene einfachste Variante wird dort *docid-Index* benannt, da das Ergebnis stets die Liste aller gefundenen Dokumentennummern ist. Ein Häufigkeitsindex speichert somit zusätzlich noch die absoluten Häufigkeiten jedes keywords innerhalb des jeweiligen Dokuments, in dem es auftritt, wobei die Häufigkeit als tf für *term frequency* bezeichnet wird und die Dokumenten-Id als d_{id}. Damit haben dort die Einträge in den invertierten Listen die Form von Paaren (d_{id}, tf).

Weiterhin muss der Positionsindex jede Vorkommensposition des Auftretens eines keywords in einem Dokument speichern. Dies bedeutet, dass sich in jeder Postingliste Tupel der Form $(d_{id}, tf, \langle p_0, ..., p_{tf} \rangle)$ befinden. Damit unterstützt ein Positionsindex alle Suchoperationen eines Häufigkeitsindex, wobei weitere Suchfunktionen, wie etwa die Suche nach Phrasen, ermöglicht werden. In einem schemaunabhängigen Index werden nur die Positionsinformationen innerhalb der invertierten Listen gespeichert. [15] Genauere Angaben zur Implementierung invertierter Dateien beziehungsweise weitere Informationen zu deren Anwendungen im Information Retrieval können in [3, 48, 15] gefunden werden.

Ausblick

Für die im späteren Verlauf der Arbeit eingeführte String-Indexstruktur des CDAWGs wird mit Definition 6.1.1 eine genaue Beschreibung gegeben, wie diese Indexstruktur als invertierte Datei eingesetzt werden kann.

Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in vier Teile, die jeweils ein oder mehrere Kapitel umfassen:

In **Teil I** werden zunächst grundlegende mathematische Begriffe und Formalismen eingeführt, auf denen die nachfolgenden Kapitel aufbauen. Im Anschluss an Grundlagen aus Automaten- und Graphentheorie folgt eine

kurze Einführung in das Thema des *String-Matchings*. Darin werden einige Grundprobleme dieses Gebiets zusammen mit klassischen Lösungsstrategien erläutert. Im dritten Kapitel von Teil I werden verschiedene String-Indexstrukturen und deren effiziente Berechnungsmöglichkeiten beschrieben. Schließlich wird dort die in dieser Arbeit zentrale SCDAWG-Struktur eingeführt.

Teil II widmet sich der Implementierung der SCDAWG-Struktur. Dabei wird neben einer on-line Variante auch ein off-line Verfahren gezeigt. Für beide Ansätze werden ausführliche Pseudocode-Beschreibungen gegeben. Zudem wird auf einige vorteilhafte Eigenschaften der SCDAWG-Indexstruktur verwiesen. Im zweiten Kapitel von Teil II werden generelle Suchalgorithmen auf einem SCDAWG-Index besprochen, die teilweise als Hilfsfunktionen der in den nächsten Kapiteln beschriebenen Verfahren dienen.

Teil III beschäftigt sich in zwei Kapiteln mit der Anwendbarkeit der SCDAWG-Struktur für die Aufgabenstellung, Gemeinsamkeiten oder Parallelen in Texten zu finden. Dabei werden zunächst verschiedene Definition gemeinsamer Teilwörter innerhalb einer SCDAWG-Struktur untersucht und anschließend ein Algorithmus besprochen, der das Auffinden der längsten gemeinsamen Teilwörter einer Textsammlung im Kontext ihrer Vorkommen zulässt. Im gleichen Kapitel folgen zwei Anwendungsbeispiele dieses Verfahrens, bevor in einem zweiten Kapitel eine Methode zur globalen Alignierung zweier Strings behandelt wird.

Im abschließenden **Teil IV** wird aufgezeigt, wie sich textuelle Charakteristiken einer Dokumentensammlung mit Hilfe der SCDAWG-Struktur identifizieren lassen. Als Anwendungsbeispiel dieser Methode wird ein simpler Klassifikator erstellt, der auf der Extraktion kürzester eindeutiger Teilwörter basiert. Dieser Klassifikator wird auf verschiedenen Dokumentensammlungen trainiert und ausgewertet.

Teil I.

String-Matching und String-Indexstrukturen

1. Terminologie

In diesem Kapitel werden grundlegende Formalismen bezüglich Zeichenketten, Graphen- und Automatentheorie erläutert, die das Fundament aller nachfolgend beschriebenen Methoden und Algorithmen bilden.

1.1. Grundlagen

Definition 1.1.1 (Alphabet)**.** Ein *Alphabet* ist eine endliche Menge Σ von Zeichen (Symbolen).

Beispiel 1.1.1. $\Sigma := \{a, b, c, \ldots, z\}$ ist ein Alphabet, das alle kleinen lateinischen Buchstaben enthält. Genauso ist auch $\Sigma := \{Der, Mann, die, Frau, Hund\}$ ein Alphabet, dessen Elemente aus einer arbiträren Menge von Token besteht.

Die Elemente von Σ werden als *Symbole* oder *Zeichen* bezeichnet, wobei $\sigma \in \Sigma$ stets ein Zeichen eines Alphabets repräsentiert.

Definition 1.1.2 (Wort)**.** Ein *Wort* w der Länge n ist eine *endliche Zeichenkette* mit $n \geq 0$ Zeichen aus Σ. Das i-te Zeichen eines Wortes w wird durch w_i mit $1 \leq i \leq n$ angegeben.

Beispiel 1.1.2. Sei $\Sigma := \{a, b, c\}$ ein Alphabet, dann ist $w = abcbc$ ein Wort über Σ der Länge 5. $w_1 = a$ und $w_5 = c$.

Das leere Wort der Länge $n = 0$ wird durch ε notiert. Im späteren Verlauf wird ein Wort w auch als *Eingabewort* oder *Eingabestring* bezeichnet.

Definition 1.1.3 (Kleenesche Hülle)**.** Die Menge aller Wörter über einem Alphabet Σ wird durch Σ^* angegeben, wobei gilt:

$\Sigma^n =$ Menge aller Wörter der Länge n über Σ,

$\Sigma^* = \bigcup_{i \in \mathbb{N}} \Sigma^i$,

$\Sigma^+ = \bigcup_{i > 0} \Sigma^i$.

Definition 1.1.4 (Sprache)**.** Eine *(formale) Sprache L* über einem Alphabet Σ ist eine Teilmenge L von Σ^*.

Definition 1.1.5 (Konkatenation)**.** Durch die *Konkatenation* $x \circ y$ zweier Wörter x der Länge m und y der Länge n ergibt sich ein Wort w der Länge $m + n$, wobei gilt:

$$w_i := \begin{cases} x_i \text{ für } 1 \leq i \leq m, \\ y_j \text{ für } i = m + j, i \leq m + n, 1 \leq j \leq n. \end{cases}$$

Anstelle von $x \circ y$ wird abgekürzt auch xy geschrieben.

Definition 1.1.6 (Umkehrung)**.** Sei w ein Wort der Länge m dann gilt für alle Zeichen des umgekehrten Wortes w^{rev} : $w_i^{rev} = w_{m+1-i}$.

Definition 1.1.7 (Präfix, Infix, Suffix)**.** Seien $w, x, y, z \in \Sigma^*$.

1. Wenn $w = xy$ dann heißt x *Präfix* von w. x heißt *echtes* Präfix wenn gilt: $w \neq y$.
2. Wenn $w = xyz$ dann heißt y *Infix*, *Faktor*, *Teilwort* oder *Substring* von w.
3. Wenn $w = yz$ dann heißt z *Suffix* von w. z heißt *echtes* Suffix wenn gilt: $w \neq y$.

Präfix(w), *Infix(w)* und *Suffix(w)* bezeichnen die entsprechenden Mengen aus Präfixen, Infixen und Suffixen eines Wortes w.

Beispiel 1.1.3. Sei $w = abcbc$, dann ist:

ab ein echtes Präfix,

cbc ein echtes Suffix,

cb und $abcbc$ ein Infix von w.

Definition 1.1.8 (Unmittelbarer linker und rechter Kontext)**.** Sei $w \in \Sigma^+$ und $x, y, u, v \in \Sigma^*$ sowie $\sigma \in \Sigma$. Wenn w die Form $w = ux\sigma yv$ hat, dann heißt σ *unmittelbarer rechter Kontext* von x und *unmittelbarer linker Kontext* von y.

Beispiel 1.1.4. Sei $w = aabcc$, dann ist:

a unmittelbarer linker Kontext des Infixes bc,

b unmittelbarer rechter Kontext des Präfixes aa,

und a, b, c sind unmittelbare linke und rechte Kontexte von ε.

1.2. Graphen

Definition 1.2.1 (Gerichteter gelabelter Graph)**.** Ein *gerichteter gelabelter Graph* G über einem Alphabet Σ ist ein geordnetes Paar (V, E) für das gilt:

V ist die Menge der *Knoten*,

E ist eine Menge von Tripeln $E \subseteq V \times \Sigma^* \times V$, die als *gelabelte Kanten* bezeichnet werden.

Die Beschriftung einer Kante $(v, \alpha, v') \in E$ ist somit eine Zeichenkette $\alpha \in \Sigma^*$, die als *Label* bezeichnet wird. Alternativ zur Tripelschreibweise werden gelabelte Kanten auch als $v \xrightarrow{\alpha} v'$ geschrieben.

Die Richtung, in der die Kanten in einem gerichteten Graphen durchlaufen werden können, werden durch Pfeile angezeigt:

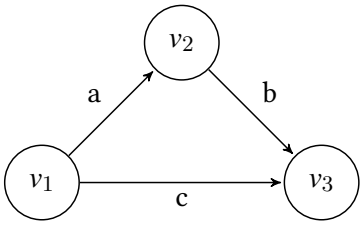


Abbildung 1.1.: Gerichteter gelabelter Graph.

Definition 1.2.2 (Mengen ausgehender und eingehender Kanten eines Knotens)**.** Seien $G = (V, E)$ ein gerichteter gelabelter Graph und $v, v' \in V$ zwei Knoten. Dann ist $E^{v\rightarrow} \subseteq E$ die Menge aller ausgehenden Kanten von v der Form (v, α, v') und $E^{\rightarrow v} \subseteq E$ die Menge aller eingehenden Kanten von v der Form (v', α, v), mit $\alpha \in \Sigma^*$.

Definition 1.2.3 (Direkter Vorgänger und Nachfolger eines Knotens)**.** Seien $v, v' \in V$, dann ist v *direkter Vorgänger* von v' und v' *direkter Nachfolger* von v, falls ein $\alpha \in \Sigma^*$ existiert, mit $(v, \alpha, v') \in E$.

Definition 1.2.4 (Eingangs- und Ausgangsgrad)**.** Sei $G = (V, E)$ ein gerichteter gelabelter Graph, so heißt:

$|E^{\rightarrow v}|$ der *Eingangsgrad* von v,

$|E^{v\rightarrow}|$ der *Ausgangsgrad* von v.

Definition 1.2.5 (Pfad)**.** Ein *Pfad* zwischen zwei Knoten v_1 und v_n bezeichnet eine Folge von Knoten $v_1, ..., v_j, v_{j+1}, ..., v_n$ mit $1 \leq j \leq n$, wobei zwei aufeinanderfolgende Knoten v_j und v_{j+1} innerhalb der Knotenfolge durch eine Kante (v_j, α, v_{j+1}), $\alpha \in \Sigma^*$, verbunden sind. Gilt $v_1 = v_n$, heißt der Pfad zwischen v_1 und v_n *Zyklus*, falls $n > 2$, leerer Pfad, falls $n = 1$.

Definition 1.2.6 (Zusammenhängender Graph)**.** Ein gerichteter Graph $G = (V, E)$ heißt *zusammenhängend*, wenn je zwei Knoten durch einen Pfad verbunden sind.

Definition 1.2.7 (Gerichteter azyklischer Graph (DAG))**.** Ein gerichteter Graph $G = (V, E)$ heißt *azyklischer gerichteter Graph* oder *DAG* wenn er keinen Pfad enthält, der einen Zyklus beschreibt.

Definition 1.2.8 (Kompakter gerichteter Graph (CDAG))**.** Ein gerichteter Graph $G = (V, E)$ ist *kompakt*, falls für keinen Knoten $v \in V$ gilt: $|E^{\rightarrow v}| = |E^{v \rightarrow}| = 1$.

Jeder gerichtete Graph $G = (V, E)$ kann in einen kompakten gerichteten Graphen überführt werden, indem alle Pfade zwischen zwei Knoten v und v', für die für alle Knoten v_j innerhalb des Pfades gilt, dass diese Ausgangsgrad und Eingangsgrad 1 besitzen, entfernt werden und durch eine einzige Kante zwischen v und v' mit dem Label, das aus der Konkatenation der Labels des Pfades besteht, ersetzt werden. Ist ein kompakter gerichteter Graph nach Definition 1.2.7 zyklenfrei, wird er als *Compact Directed Acyclic Graph* bzw. *CDAG* bezeichnet.

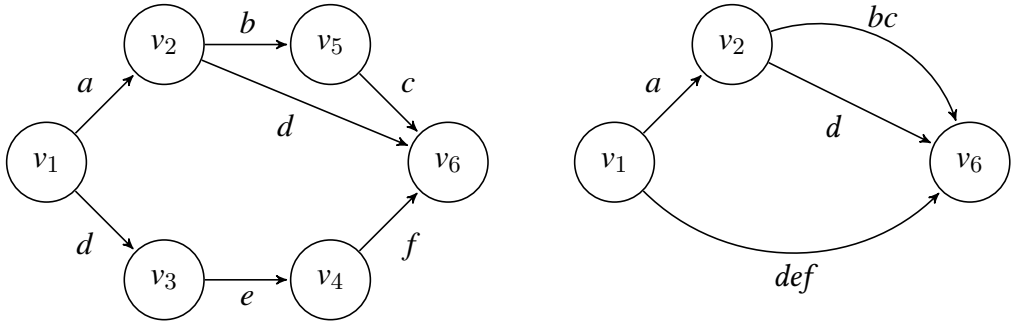


Abbildung 1.2.: DAG (links) und CDAG (rechts) mit gelabelten Kanten.

1.3. Bäume

Definition 1.3.1 (Gerichteter gelabelter Baum)**.** Ein *gerichteter gelabelter Baum T* ist ein gerichteter gelabelter Graph (V, E), für den zusätzlich gilt:

1. Es existiert genau ein Wurzelknoten $v_\varepsilon \in V$, sodass T von v_ε aus zusammenhängend ist, und damit jeder Knoten $v \in V$ von v_ε aus erreichbar ist.
2. Zwischen zwei Knoten v und $v' \in V$ existiert nur ein Pfad $(v, \ldots, v')$.
3. T ist zyklenfrei und für $T \neq \emptyset$ gilt: $|E| = |V| - 1$.

Jeder Knoten $v \in V$, mit Ausgangsgrad 0 wird als *Blatt* bezeichnet. Entspricht ein Baum T nach Definition 1.2.8 einem CDAG, so wird er als *kompakter gerichteter gelabelter Baum* bezeichnet.

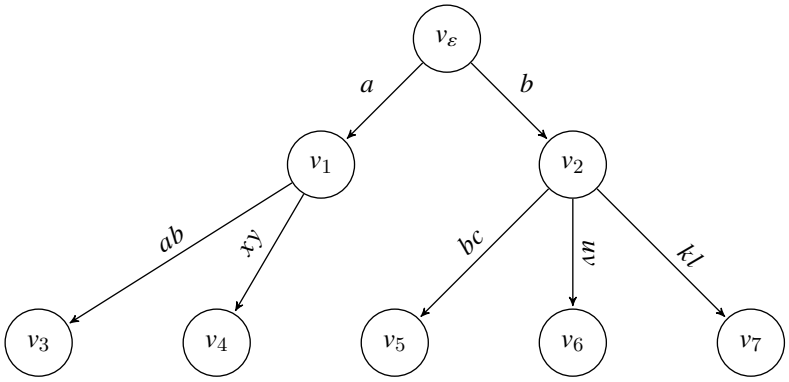


Abbildung 1.3.: Gerichteter gelabelter Baum mit dem Wurzelknoten v_ε.

1.4. Tiefensuche

Zur Erforschung eines gerichteten gelabelten Baumes $T = (V, E)$ kann eine *Tiefensuche* beziehungsweise *Depth-First Search (DFS)* von einem Knoten $v \in V$ aus gestartet werden. Es existieren verschiedene Suchstrategien, die die Ausgabe der Knoten steuern. Die folgend darstellte ist als *Präordnung* bekannt. Dabei werden beginnend mit dem Startknoten alle Teilbäume unterhalb des Startknotens von oben nach unten durchsucht, solange bis alle Knoten des Baumes besucht wurden. Die Entscheidung, in welcher Reihenfolge die Geschwisterknoten von T abgearbeitet werden, ergibt sich aus der lexikographischen Ordnung der Labels der von einem Knoten ausgehenden Kanten.

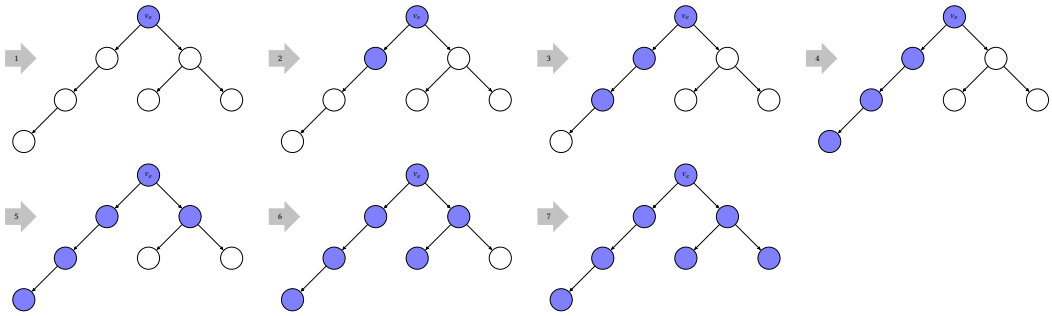


Abbildung 1.4.: Schematischer Ablauf einer Tiefensuche in Präordnung in einem gerichteten gelabelten Baum beginnend beim Wurzelknoten v_ε.

Werden die Knoten in *Postordnung* ausgegeben, erfolgt deren Rückgabe beginnend mit den Teilbäumen von unten nach oben, bis letztlich der Startknoten ausgegeben wird.

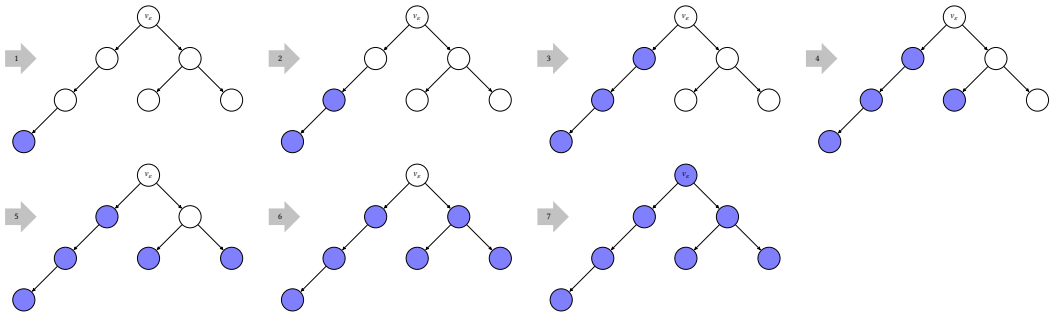


Abbildung 1.5.: Schematischer Ablauf einer Tiefensuche in Postordnung in einem gerichteten gelabelten Baum beginnend beim Wurzelknoten v_ε.

Die Implementierung erfolgt mittels Rekursion. Bei der Tiefensuche auf einem gerichteten gelabelten Graphen muss darauf geachtet werden, dass kein Knoten mehrfach besucht wird. Dies wäre etwa bei nicht zyklenfreien Graphen leicht der Fall. Bereits besuchte Knoten werden dazu markiert, beziehungsweise in einer Liste *visited* gespeichert. Mit dem booleschen Parameter *postorder* wird gesteuert, ob die Ausgabe der Knoten mithilfe der Funktion *callback* in Präordnung, also beim ersten Treffer, oder in Postordnung nach der Ausgabe aller Nachfolgeknoten erfolgt. Bei jedem Aufruf von $\mathit{callback}(v)$ ist der damit verbundene Knoten v nach der gewählten Ordnung vollständig abgearbeitet und kann nun weiterverarbeitet werden. Soll zum Beispiel die in Abbildung 1.5 gezeigte Einfärbung der Knoten vorgenommen werden, würde die Blaufärbung jedes einzelnen besuchten Knotens innerhalb von $\mathit{callback}(v)$ geschehen.

Algorithmus 1 Tiefensuche auf einem gerichteten Graphen

Vorbedingung: Ein gerichteter Graph $G = (V, E)$

```
1:  visited ← {}
2:  procedure DFS(v, postorder)
3:      visited.push(v)
4:      if ¬ postorder then
5:          callback(v)                          ▷ Ausgabe in Präordnung
6:      end if
7:      for each (v, α, v′) ∈ E^{v→} do
8:          if v′ ∉ visited then
9:              DFS(v′, postorder)
10:         end if
11:     end for
12:     if postorder then
13:         callback(v)                          ▷ Ausgabe in Postordnung
14:     end if
15: end procedure
```

Eine Tiefensuche von einem Knoten $v_s \in V$ aus wird in allen folgenden Pseudocode-Beschreibungen so angegeben, wobei die Nachfolger jedes Knotens v nach lexikographischer Ordnung seiner Übergangslabels durchlaufen werden:

```
for each v ∈ DFS(v_s, (true | false)) do
    ...
end for
```

Die Laufzeit einer Tiefensuche liegt im schlechtesten Fall bei $O(|V| + |E|)$, da durch die Verwendung der *visited*-Liste alle Pfade des Graphen maximal einmal durchlaufen werden. Dies setzt allerdings voraus, dass jede Kante in $O(1)$ durchwandert werden kann, was durch die Speicherung des Graphen als Adjazenzliste [64] erreicht werden kann.

1.5. Breitensuche

Neben der Tiefensuche wird zudem eine *Breitensuche* oder *Breadth-First Search (BFS)* zur Traversierung eines gerichteten gelabelten Baumes $T = (V, E)$ betrachtet. Beginnend von einem beliebigen Knoten $v_s \in V$ werden bei der *BFS* alle Knoten, die unterhalb von v_s liegen, ebenentreu durchsucht.

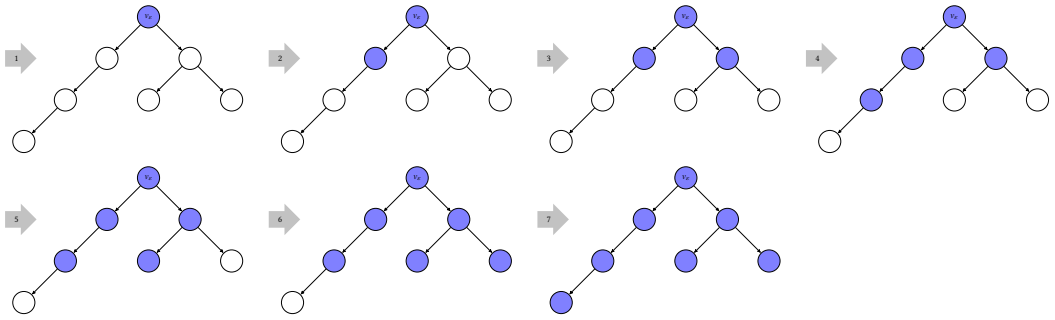


Abbildung 1.6.: Schematischer Ablauf einer Breitensuche in einem gerichteten gelabelten Baum beginnend beim Wurzelknoten v_ε.

Die Implementierung der Breitensuche wird mittels einer Warteschlange *(queue)* umgesetzt. Damit bei einer BFS auf einem gerichteten gelabelten Graphen keine Knoten doppelt besuchten werden, werden wie bei DFS bereits untersuchte Knoten markiert *(visited)*. Der erste markierte Knoten wird an den Anfang der Warteschlange geschrieben. Anschließend wird dieser gleich wieder aus der Schlange entfernt und für seine Nachfolger überprüft, ob sie schon besucht wurden. Die nicht besuchten Nachfolger werden der Warteschlange hinzugefügt und dieser Vorgang solange wiederholt, bis die Warteschlange leer ist.

Algorithmus 2 Breitensuche auf einem gerichteten Graphen

Vorbedingung: Ein gerichteter Graph $G = (V, E)$.

```
1:  procedure BFS(v_s)
2:      visited ← {}
3:      queue ← {}
4:      visited.push(v_s)
5:      queue.push(v_s)
6:      while queue ≠ ∅ do
7:          v ← queue.pop()
8:          callback(v)                          ▷ Ausgabe des Knotens v
9:          for each (v, α, v') ∈ E^{v→} do
10:             if v' ∉ visited then
11:                 visited.push(v')
12:                 queue.push(v')
13:             end if
14:         end for
15:     end while
16: end procedure
```

Die Laufzeit der Breadth-First-Suche liegt gleichermaßen bei $O(|V| + |E|)$, da erneut jede Kante und jeder Knoten von G einmal besucht werden. Analog zur DFS wird in allen folgenden Algorithmusbeschreibungen in Pseudocode, in denen eine Breadth-First-Suche von einem beliebigen Knoten v_s aus auf einem Graphen G gebraucht wird, diese so angegeben:

for each $v \in BFS(v_s)$ **do**
    ...
end for

Ausblick

Die in Kapitel 3.6 beschriebene SCDAWG-Struktur ist ein zentraler Eckpfeiler dieser Arbeit und die Tiefensuche (Algorithmus 1) bildet dabei oft die Grundlage der Anwendungen auf einer solchen Struktur. Da eine SCDAWG-Struktur aber zwei Arten von Übergängen besitzt, Linksübergänge, die mit E_L und Rechtsübergänge, die mit E_R notiert werden, wird die hier vorgestellte generische Implementierung der DFS, wenn diese auf eine duale SCDAWG-Struktur angewendet wird, zur Traversierung ihrer rechten Kanten E_R genutzt. Tiefensuchen, die Folgen von rechten oder linken Übergängen behandeln, werden explizit innerhalb der Codebeschreibungen angegeben. Die Abbildungen 5.3 und 5.4 verdeutlichen die Notwendigkeit, Folgen rechter oder linker Kanten zu betrachten.

1.6. Deterministische endliche Automaten

Definition 1.6.1 (Deterministischer endlicher Automat)**.** Ein *deterministischer endlicher Automat (DEA) A* wird durch ein Quintupel $A = (Q, \Sigma, \delta, s, F)$ definiert, sodass gilt:

Q ist eine endliche Menge von *Zuständen*,

Σ ist ein endliches Alphabet, genannt das *Eingabealphabet*,

$\delta : Q \times \Sigma \to Q$ ist die *Übergangsfunktion*,

$s \in Q$ ist der ausgezeichnete *Startzustand*,

$F \subseteq Q$ ist die Menge der *Finalzustände* des Automaten.

Wenn die Übergangsfunktion $\delta(q, \sigma)$ für alle $q \in Q$ und für alle $\sigma \in \Sigma$ definiert ist, wird A als *total* oder *vollständig* bezeichnet, ansonsten als *partiell*.

Definition 1.6.2 (Verallgemeinerte Übergangsfunktion)**.** Die *verallgmeinerte Übergangsfunktion* $\delta^* : Q \times \Sigma^* \to Q$ lässt sich induktiv wie folgt ausdrücken.

$$\delta^*(q, \varepsilon) := q,$$
$$\delta^*(q, w\sigma) := \delta(\delta^*(q, w), \sigma).$$

Definition 1.6.3 (Sprache von A)**.** Die Sprache eines DEA A, $L(A)$ ist definiert als:

$$\{w \in \Sigma^* \mid \exists f \in F : \delta^*(s, w) = f\}$$

Beispiel 1.6.1. Sei $A = (\{s, q_1, q_2\}, \{a, b\}, \delta, s, \{q_2\})$ mit $\delta = \{(s,a,s), (s,b,q_1), (q_1,b,q_2), (q_2,a,q_1)\}$, dann ergibt sich folgende graphische Darstellung zu A:

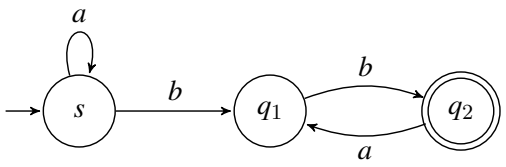


Abbildung 1.7.: DEA mit L(A) = a*bb (ab)*

Der in Abbildung 1.7 gezeigte Automat wird Definition 1.6.1 entsprechend als *partieller* DEA bezeichnet, da die Zustände q_1 und q_2 keine Übergänge für alle

$\sigma \in \Sigma$ besitzen. Die nächste Abbildung zeigt den gleichen DEA mit *vollständiger* Übergangsfunktion, was durch das Hinzufügen eines Fallenzustandes $\emptyset \notin F$ erreicht wird.

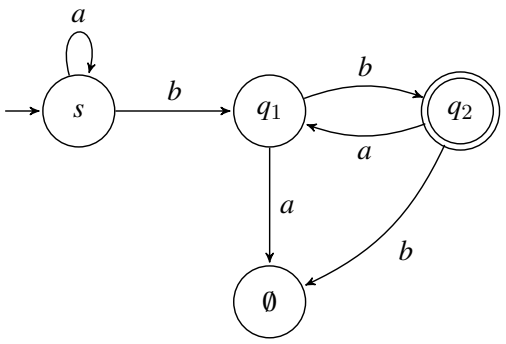


Abbildung 1.8.: DEA mit vollständiger Übergangsfunktion

1.6.1. Minimierung

Im Folgenden wird aufgezeigt, wie zu einem gegebenen vollständigen DEA $A = (Q, \Sigma, \delta, s, F)$ ein äquivalenter DEA A' mit minimaler Zustandsmenge erzeugt werden kann.

Definition 1.6.4 (k-Ununterscheidbarkeit)**.** Sei $A = (Q, \Sigma, \delta, s, F)$ ein DEA . Zwei Zustände $p, q \in Q$ werden als *k-ununterscheidbar* bezeichnet gdw. $\forall w \in \Sigma^*$ mit $|w| \leq k$ für $k \geq 0$ gilt:

$$\delta^*(p, w) \in F \iff (q, w) \in F$$

Definition 1.6.5 (Ununterscheidbarkeit)**.** Zwei Zustände $p, q \in Q$ eines DEA A heißen *ununterscheidbar* gdw. sie $\forall k \in \mathbb{N}$ k-ununterscheidbar sind.

Definition 1.6.6 (Erreichbarer Zustand)**.** Ein Zustand $q \in Q$ eines DEA A ist *erreichbar* wenn $\exists w \in \Sigma^* : q = \delta^*(s, w)$

Beispiel 1.6.2. Sei $A = (\{q_1, q_2, q_3, q_4, q_5\}, \{a, b\}, \delta, q_1, \{q_4\})$, mit $\delta = \{(q_1,a,q_2), (q_1,b,q_5), (q_2,a,q_3), (q_2,b,q_4), (q_3,a,q_3), (q_3,b,q_4), (q_4,a,q_5), (q_4,b,q_5), (q_5,a,q_5), (q_5,b,q_5)\}$.

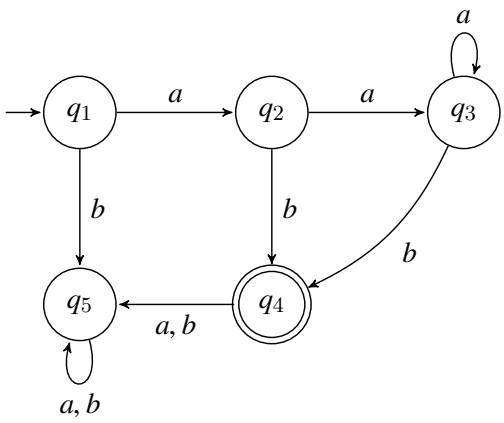


Abbildung 1.9.: DEA mit ununterscheidbaren Zuständen q_2 und q_3.

Definition 1.6.7 (k-Ununterscheidbarkeit als Äquivalenzrelation)**.** Die Äquivalenz bzw. k-Ununterscheidbarkeit zweier Zustände $p, q \in Q$ lässt sich anhand einer Äquivalenzrelation $\sim_i$ mit $i \geq 0$ definieren:

$$p \sim_0 q \iff p, q \in F \text{ oder } p, q \notin F,$$
$$p \sim_{i+1} q \iff p \sim_i q \text{ und } \delta(p, \sigma) \sim_i \delta(q, \sigma) \forall \sigma \in \Sigma.$$

Lemma 1.6.1. Sei $A = (Q, \Sigma, \delta, s, F)$ ein DEA. Für $i \geq 0$ sei die Äquivalenzrelation $\sim_i$ wie oben definiert.

1. Zwei Zustände p, q sind k-ununterscheidbar genau dann, wenn $p \sim_k q$ gilt.
2. Es gibt ein $n \leq |Q|$, sodass die Äquivalenzrelationen $\sim_{n+k}$ für alle $k \geq 0$ übereinstimmen.
3. Es sind p, q ununterscheidbar genau dann wenn $p \sim_n q$.

Aus Lemma 1.6.1 folgt, dass sich durch die k-Ununterscheidbarkeit von Zuständen Äquivalenzklassen ergeben, die k-ununterscheidbare Zustände zusammenfassen. Für jede Äquivalenzklasse $[p] = \{p \in Q \mid p \sim_n q\}$ wird ein Repräsentant $Rep([p])$ gewählt.

Definition 1.6.8 (Reduzierter DEA)**.** Einen DEA A nennt man *reduziert*, wenn sich alle Zustände von Q paarweise unterscheiden und jeder Zustand vom Startzustand s aus erreichbar ist.

Lemma 1.6.2. Zu jedem DEA A kann ein äquivalenter reduzierter Automat A' algorithmisch berechnet werden.

Es kann gezeigt [] werden, dass ein solcher reduzierter Automat $A' = (Q', \Sigma, \delta', s', F')$ wie folgt konstruiert werden kann:

$$Q' := \{[q] \mid q \in Q\},$$

$$\delta([q], \sigma) := [\delta(q, \sigma)],$$

$$s' := [s],$$

$$F' := \{[f] \mid f \in F\}.$$

Für den Automaten aus Abbildung 1.9 ergibt sich folgender reduzierter DEA A':

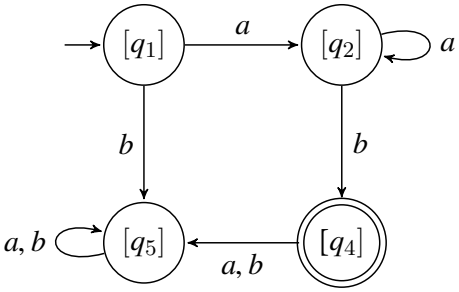


Abbildung 1.10.: Reduzierter äquivalenter DEA A' mit $L(A') = L(A)$.

1.6.2. Satz von Myhill-Nerode

Um zu zeigen, dass ein reduzierter endlicher Automat A der kleinste bzw. minimale DEA ist, der die Sprache $L(A)$ erkennt, kann das Theorem von Myhill-Nerode [56] verwendet werden:

Definition 1.6.9 (Nerode-Äquivalenzrelation)**.** Es sei L eine beliebige Sprache auf einem Alphabet Σ. Auf Σ^* wird die Nerode Äquivalenzrelation $\sim_L$ definiert:

$$v_1 \sim_L v_2 \iff (\forall s \in \Sigma^* : v_1 s \in L \iff v_2 s \in L).$$

In einer Nerode-Äquivalenzklasse befinden sich alle Wörter $v \in \Sigma^*$, die mit genau denselben Suffixen s aus Σ^* zu einem Wort der Sprache L ergänzt werden können.

Beispiel 1.6.3. Die Sprache des Beispiel-Automaten in Abbildung 1.10 lautet $L(A') = aa^*b$. Daraus ergeben sich folgende Äquivalenzklassen unter $\sim_L$:

$$[\varepsilon] = \{\varepsilon\}\ s = aa^*b$$

$$[a] = \{a, aa, aaa, aaa, \ldots\}\ s = a^*b$$

$$[ab] = \{ab, aab, aaab, aaab, \ldots\}\ s = \varepsilon$$

$$[b] = \{b, bb, ..., aba, aaba, abb, aabb, \ldots\}\ s = \emptyset$$

Es ergeben sich im Beispiel drei Nerode-Äquivalenzklassen, die mit gleichen Rechtskontexten ihre Elemente zu Wörtern von $L(A')$ ergänzen. Die vierte Äquivalenzklasse ($[b]$) beinhaltet alle Wörter, für die kein s angegeben werden kann, sodass deren Elemente in $L(A')$ liegen. Jede Nerode-Aquivalenzklasse entspricht damit auch der Sprache eines Zustands aus A'. Daraus kann geschlussfolgert werden, dass A' *minimal* ist, da er keinen weiteren Zustand enthält, dessen Sprache nochmals einer Nerode-Äquivalenzklasse entspricht. Somit gilt:

Lemma 1.6.3. Sei A ein DEA mit $A = (Q, \Sigma, \delta, s, F)$ mit vollständiger Übergangsfunktion. Wenn die Anzahl der Zustände $|Q|$ dem Index seiner zugehörigen Nerode-Äquivalenzrelation $|\Sigma^*/\sim_L|$ entspricht, ist A *minimal*.

Theorem 1.6.1 (Satz von Myhill-Nerode)**.** Es sei L $\subseteq \Sigma^*$ und $\sim_L$ wie oben. Dann ist L regulär gdw. gilt: $|\Sigma^*/\sim_L| < \infty$.

Die Sprachen von A bzw. seinem äquivalenten minimalen Automaten A' sind somit regulär, da die Anzahl der Nerode-Äquivalenzklassen endlich ist.

2. String-Matching

Einführend werden einige Grundprobleme aus dem Bereich des String-Matchings besprochen und wohlbekannte Lösungsansätze diskutiert, die ohne die Vorberechnung einer Indexstruktur für einen oder mehrere Eingabetexte ablaufen. Einerseits dienen diese Methoden so zum Vergleich mit den nachfolgend beschriebenen indexbasierten Textverarbeitungsverfahren, andererseits können sie teilweise auch mit diesen kombiniert werden.

2.1. Exaktes Matching

Gegeben seien ein Wort $t \in \Sigma^*$ (auch als *Text* bezeichnet) und ein Wort $p \in \Sigma^*$ (auch als *Pattern* bezeichnet) über einem endlichen Alphabet Σ. Beim *exakten Matching* sollen nun alle Vorkommen von p in t ermittelt werden. Wenn zum Beispiel die Vorkommen von $p = ba$ in $t = baaba$ gefunden werden sollen, würden als Ergebnis die Positionen 1 und 4 geliefert werden.

2.1.1. Naiver Ansatz

Als naives Vorgehen können der Reihe nach von links nach rechts die Zeichen von p in t verglichen werden. Würden die Zeichen an einer Position übereinstimmen, würde die nächste Position von p in t verglichen werden, bis entweder p erschöpft ist oder aber die Zeichen zweier Positionen von p und t nicht gleich sind. Ist dies der Fall, wird p eine Position nach rechts verschoben, und der Vergleich beginnt von Neuem. Wenn die letzte Position von t erreicht ist, kann gestoppt werden.

2 Vergleiche	3 Vergleiche	4 Vergleiche	6 Vergleiche
|*ba*	|*ba*	|*ba*	|*ba*
|*baaba*	*b*|*aaba*	*ba*|*aba*	*baa*|*ba*

Tabelle 2.1.: Naives exaktes Matching des Patterns *ba* im Text *baaba*.

Da so im schlechtesten Fall so viele Vergleiche zwischen p und t stattfinden, wie p lang ist, und das beginnend von jeder Position in t, liegt die Zeitkomplexität

dieses Ansatzes bei $O(mn)$, wenn m die Länge von p ist und n die Länge von t. Eine detaillierte Untersuchung des naiven Ansatzes kann in [31] gefunden werden.

2.1.2. DEA-basierter Ansatz

Eine andere Methode, die benutzt werden kann, basiert auf dem in 1.6 eingeführten Konzept des deterministischen endlichen Automaten. Für $p = aabbccd$ kann folgender DEA angenommen werden:

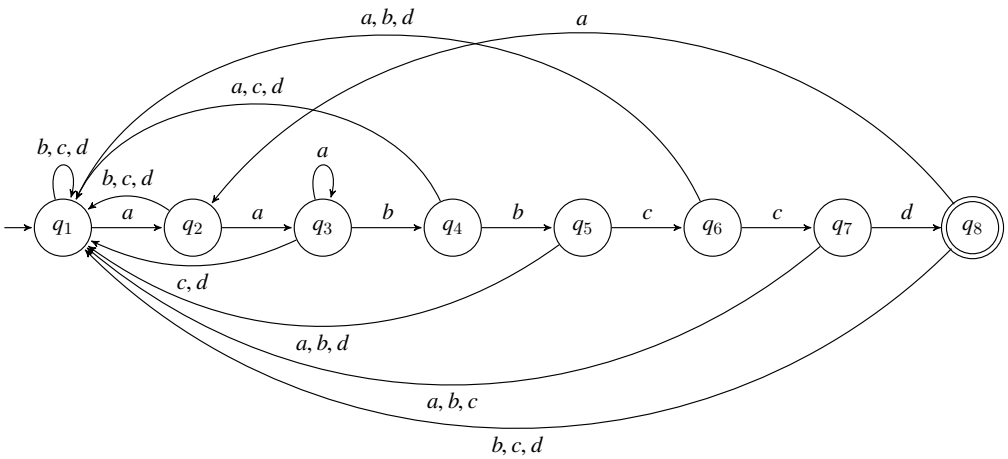


Abbildung 2.1.: DEA $A(p)$ des Patterns $aabbccd$.

Der minimale Pattern-Matching-Automat $A(p)$ erkennt bei beliebiger Eingabe eines Textes t über Σ das Pattern p, wann immer er für dieses in den Endzustand gelangt. Verglichen mit dem naiven Ansatz verwirft dieser Ansatz bereits gefundene passende Stellen nicht sofort wieder. Im Beispiel bleibt, sobald zweimal der Buchstabe a gelesen wurde, der Automat bei Eingabe weiterer a's in q_3. Nur wenn c oder d von q_3 aus gelesen werden, geht er in den Anfangszustand q_1 zurück. Beim Einlesen mehrerer a's werden die beiden letzten Vorkommen somit als Beginn des gesuchten Patterns p benutzt. Ein effizientes Verfahren zur Berechnung von $A(p)$ ist der Knuth-Morris-Pratt Algorithmus oder kurz KMP Algorithmus [44]. Die Implementierung wird anhand einer Präfixtabelle durchgeführt, zu deren Aufbau das Pattern komplett durchlaufen werden muss. Damit ergibt sich eine Laufzeit von $O(m)$ für diesen Schritt, wobei $m = |p|$. Sobald $A(p)$ aufgebaut ist, laufen Anfragen in $O(n)$, wobei $n = |t|$.

Theorem 2.1.1 (Knuth, Morris, Pratt [44])**.** Das Exakt-Matching-Problem kann in $O(m + n)$ Zeit und mit $O(n)$ Speicherverbrauch gelöst werden.

Eine detaillierte Beschreibung der Konstruktion von $A(p)$ mit Hilfe des KMP-Algorithmus kann in [19] nachgeschlagen werden. Als eine mögliche Alternative zum KMP-Algorithmus kann zusätzlich der Boyer-Moore- Algorithmus [10] betrachtet werden. Wenn gleichzeitig mehrere Pattern mit t verglichen werden sollen, kann der auf einem Trie (siehe 3.1) basierende Aho-Corasick Algorithmus [1] angewendet werden.

2.2. Longest Common Factor (LCF)

Eine etwas andere Aufgabe ergibt sich, wenn gemeinsame Teilwörter innerhalb von zwei Wörtern gefunden werden sollen.

Definition 2.2.1 (LCF)**.** Seien $u, v, w \in \Sigma^*$, dann heißt w *längstes gemeinsames Teilwort* oder *longest common factor* von u und v, wenn gilt: w ist Infix von u und v und hat die maximale Länge unter den gemeinsamen Infixen von u und v.

Beispiel 2.2.1. Seien $u, v \in \Sigma^*$ mit

$$u = ccabcdda,$$
$$v = abcdddabc$$

dann stellt $abcdd$ das längste gemeinsame Teilwort von u und v dar.

2.2.1. Naiver Ansatz

Um das längste gemeinsame Teilwort algorithmisch zu finden, können alle Elemente der Menge *Infix(u)* bestimmt werden und anschließend für jedes Element $inf_i \in Infix(u)$ $(1 \leq i \leq |Infix(u)|)$ geprüft werden, ob inf_i innerhalb von v auftritt, also Element von *Infix(v)* ist. Ist dies der Fall, wird inf_i gemerkt und mit dem nächsten Element $inf_{i+1} \in Infix(u)$ fortgefahren. Gilt zusätzlich $|inf_{i+1}| > |inf_i|$, ersetzt inf_{i+1} das zuvor vermerkte Infix inf_i als Kandidaten für das längste gemeinsame Teilwort in u und v. Dieses Verfahren wird solange wiederholt, bis alle Elemente aus *Infix(u)* überprüft wurden. Da zur Bestimmung von *Infix(u)* aber $O(m^2)$ Schritte nötig sind und jedes Element hieraus mit v verglichen werden muss, liegt die Gesamtlaufzeit bei $O(m^2 * n)$, falls zur Überprüfung der in 2.1.2 erwähnte KMP-Algorithmus mit linearer Laufzeit verwendet wird.

2.2.2. Dynamische Programmierung

Eine bessere Laufzeit kann durch den Einsatz von dynamischen Programmiertechniken erzielt werden. Dafür wird eine Matrix M der Größe $(m + 1) \times (n + 1)$ ge-

braucht, die die Länge der gemeinsamen Teilwörter zwischen u und v aus den vorherig gespeicherten Längen berechnet. Dazu wird jede Zelle der Matrix zunächst mit 0 besetzt und anschließend für jedes u_i, $1 \leq i \leq m$, alle $v_1, ..., v_n$ geprüft, ob $u_i = v_j$. Wenn dies der Fall ist, wird $M_{i,j}$ auf den Wert des diagonal über ihm liegenden Feldes $M_{i-1,j-1} + 1$ gesetzt. Die Matrix M sähe somit für die obigen Beispielwörter folgendermaßen aus:

		a	b	c	d	d	d	a	b	c
	0	0	0	0	0	0	0	0	0	0
c	0	0	0	1	0	0	0	0	0	1
c	0	0	0	1	0	0	0	0	0	1
a	0	1	0	0	0	0	0	1	0	0
b	0	0	2	0	0	0	0	0	2	0
c	0	0	0	3	0	0	0	0	0	3
d	0	0	0	0	4	1	1	0	0	0
d	0	0	0	0	1	5	2	0	0	0
a	0	1	0	0	0	0	0	3	0	0

Abbildung 2.2.: Matrix M der Wörter $u = ccabcdda$ und $v = abcdddabc$.

Das längste gemeinsame Teilwort aus *Infix(w)* ist im Beispiel $abcdd$. In M ist dessen Länge durch den Wert 5 im Feld $M_{7,5}$ zu erkennen. Gleichermaßen ist erkennbar, dass dieser Wert aus den diagonal über ihm liegenden Werten berechnet wurde. Die so entstandene Diagonale ist grün gekennzeichnet. Da bei diesem Verfahren auch Überlappungen innerhalb gemeinsamer Teilwörter erlaubt sind, existieren mehrere kürzere Teilwörter (in Abbildung 2.2 rot markiert). Zur Identifikation der längsten Teilwörter müssen nun während der Berechnung der Matrix diejenigen Zellen gemerkt werden, die das Ende eines längsten Teilwortes bilden. Anschließend können die längsten Teilwörter durch Rückverfolgung der Diagonalen gefunden werden. Dabei kann immer dann gestoppt werden, wenn eine Zelle den Wert 0 enthält. Der Zeit- und Speicherverbrauch der dynamischen Programmiermethode zur Berechnung aller LCF zweier Wörter u, v liegt bei $\mathcal{O}(mn)$.

2.3. Approximatives Matching

Im Unterschied zum exakten Matching ist beim *approximativen Matching* eine gewisse Fehlertoleranz erlaubt. Beispielsweise wenn Rechtschreibfehler korrigiert werden sollen oder bei OCR-Erkennung Zeichenfehler auftreten, existieren Fehler, die einen exakten Vergleich unmöglich machen.

Ein klassisches Verfahren zur Bestimmung der Distanz zwischen zwei Strings wurde 1965 von Vladimir I. Levenshtein eingeführt [46]. Die als *Levenshtein-Distanz* bekannte Funktion ist wie folgt definiert.

Definition 2.3.1. (Levenshtein-Distanz [46]) Die *Levenshtein-Distanz* oder der *Editierabstand* $d_L(u, v)$ zweier Wörter $u, v \in \Sigma^*$ ergibt sich durch die minimale Anzahl an Löschungen, Einfügungen und Ersetzungen von Symbolen, die nötig sind, um u nach v zu transformieren.

Beispiel 2.3.1. Wenn $u = blume$ und $v = bruder$, ergibt sich ein Editierabstand von 3. Dabei werden folgende drei Operationen gebraucht:

	Ersetzen von l durch r	Ersetzen von m durch d	Einfügen von r
blume	*blume*	*brume*	*bruder*
bruder	*bruder*	*bruder*	*bruder*

Tabelle 2.2.: Bestimmen der Levensthein-Distanz d_L zwischen *blume* und *bruder*.

Ein früher Ansatz zur Berechnung wurden von Wagner & Fischer [78] veröffentlicht. Der Wagner-Fischer-Algorithmus berechnet d_L durch dynamisches Programmieren unter Zuhilfenahme einer iterativ aufzubauenden Matrix der Größe $m \times n$ ($m = |u|, n = |v|$), die die Editierabstände aller Präfixe des ersten Wortes zu allen Präfixen des zweiten Wortes beinhaltet. Begonnen wird mit dem Feld in der linken oberen Ecke, das die Levenshtein-Distanz des leeren Wortes zum leeren Wort enthält und damit immer mit 0 besetzt wird. Die erste Zeile der Matrix enthält damit die natürlichen Zahlen von 0 bis n und damit die Levenshtein-Abstände des leeren Wortes zu allen Präfixen von v. Analog enthält die erste Spalte die natürlichen Zahlen von 0 bis m, was den Levenshtein-Abständen des leeren Wortes zu allen Präfixen von u entspricht. Die restlichen Felder der Matrix werden nun dynamisch aus den Werten von jeweils drei umliegenden Feldern wie folgt berechnet: Sind die Buchstaben u_i und v_j identisch, werden zu addierende Kosten von $c = 0$ gemerkt, andernfalls Kosten von $c = 1$. Der Wert des Feldes $M_{i,j}$ ergibt sich dann aus dem Minimum von $(M_{i-1,j}+1, M_{i,j-1}+1, M_{i-1,j-1}+c)$. Das $M_{i-1,j}$-te Feld entspricht jeweils der Löschung eines Zeichens, das $M_{i,j-1}$-te Feld einer Einfügung und das $M_{i-1,j-1}$-te einer Ersetzung. Das Ergebnis der Berechnung von d_L ist

		b	r	u	d	e	r
	0	1	2	3	4	5	6
b	1	0	1	2	3	4	5
l	2	1	1	2	3	4	5
u	3	2	2	1	2	3	4
m	4	3	3	2	2	3	4
e	5	4	4	3	3	2	3

Abbildung 2.3.: Matrix M zur Berechnung des Editierabstandes der Wörter $u = blume$ und $v = bruder$.

letztlich an Feld $M_{m,n}$ abzulesen. Abbildung 2.3 zeigt M für die beiden Wörter aus Beispiel 2.3.1. Bei diesem algorithmischen Vorgehen liegen sowohl die Speicher- als auch die Zeitkomplexität bei $O(mn)$.

Zu einem gegebenen Text $t \in \Sigma^*$ der Länge n und einem Pattern $p \in \Sigma^*$ der Länge m soll also unter Berücksichtigung einer Fehlertoleranz k die Menge der Positionen j gefunden werden, für die ein i existiert, sodass $d(p, t_{i\ldots j}) \leq k$ gilt [52]. Um die Suche aller Vorkommen $t_{i\ldots j}$, die zu einem Suchpattern p einen Levenshtein-Abstand kleiner gleich k aufweisen, durchzuführen, kann die Berechnung von M dahingehend abgeändert werden, dass jede Position t_j als eine mögliche Startposition eines Matches gilt. Dies wird durch die Initialisierung der ersten Zeile von M mit 0 erreicht, was bedeutet, dass das leere Pattern ohne Fehler an jeder Position des Textes gematcht werden kann. Dann wird die erste Spalte mit den natürlichen Zahlen von 0 bis m vorbesetzt. Nacheinander werden nun die Spalten zu jedem Buchstaben t_j des Textes aktualisiert, indem die i-te Position in der neu zu berechnenden Spalte $M_{0\ldots m,j}$, falls $p_i = t_j$ gilt, mit dem Wert des Feldes $M_{i-1,j-1}$ besetzt wird oder andernfalls mit dem Minimum von $(M_{i-1,j}, M_{i,j-1}, M_{i-1,j-1}) + 1$. Als Ergebnis werden für $1 \leq j \leq n$ alle Textpositionen, an denen $M_{m,1\ldots n} \leq k$ gilt, bereitgestellt [66].

Gilt das Interesse allerdings der genauen Abfolge von Einfügungs-, Löschungs- und Ersetzungsoperationen, auch als Editiertranskript bezeichnet, die notwendig sind, um ein Wort in ein anderes zu überführen, kann dieses Problem mithilfe eines Tracebacks gelöst werden. Dazu werden beim Aufbau der oben beschriebenen Matrix M zusätzliche Pointer eingefügt. Wird zum Beispiel der Wert des Feldes $M_{i,j}$

algorithmisch aus dem Wert des Feldes $M_{i,j-1} + 1$ berechnet, wird ein Pointer von $M_{i,j}$ nach $M_{i,j-1}$ gesetzt. Dasselbe gilt für die beiden anderen Berechnungsmöglichkeiten für das Feld $M_{i,j}$. Dabei ist es auch möglich, dass mehr als ein Pointer von $M_{i,j}$ ausgehend gesetzt wird, wodurch mehrere Editiertranskripte gefunden werden können, indem Pointerpfade vom Feld $M_{m,n}$ zum Feld $M_{0,0}$ verfolgt werden. Da im Feld $M_{m,n}$ die kleinstmögliche Anzahl an Editieroperationen gefunden wird, die nötig ist, um das eine zu betrachtende Wort in das andere zu überführen, wird auch bei Rückverfolgung der entsprechenden Pointerpfade jeweils ein optimales Editiertranskript gefunden [31].

2.4. Globales Alignment

Eine andere Möglichkeit, eine Beziehung zweier Wörter oder Texte zu beschreiben, ergibt sich, wenn anstelle der Minimierung ihrer Unterschiede ihre Gemeinsamkeiten maximiert werden [31]. Wird diese Sichtweise benutzt, spricht man von der *Alignierung* zweier oder mehrerer Zeichenketten. Aus mathematischer Sicht ist die Berechnung eines optimalen Alignments zu der des optimalen Editiertranskripts äquivalent [65].

Definition 2.4.1. (Paarweises globales Alignment [31]) Ein *paarweises globales Alignment* zweier Wörter $u, v \in \Sigma^*$ wird dadurch erzeugt, dass bei beiden Wörtern entweder innerhalb oder an den Rändern Leerzeichen beziehungsweise *Gapsymbole* eingefügt werden und gleichzeitig beide Wörter so übereinander gestellt werden, dass jeder Buchstabe und jedes Gapsymbol des einen Wortes über einem Buchstaben oder Gapsymbol des anderen positioniert ist. Zwei Gapsymbole dürfen nicht übereinander stehen.

Beispiel 2.4.1. Wenn $u = blume$ und $v = bruder$ ergibt sich folgendes globale Alignment, wobei als Gapsymbol „-“ gewählt wurde.

b l u m e -
b r u d e r

Somit kann ein globales Alignment, das sich auf zwei komplette Wörter bezieht und nicht nur auf Teile dieser (lokales Alignment), ganz einfach in ein Editiertranskript überführt werden und umgekehrt. Dabei entsprechen zwei gegenüberstehende Buchstaben, die nicht identisch sind, einer Ersetzung, ein Leerzeichen im ersten Wort entspricht einer Einfügung des entsprechenden Buchstabens aus dem zweiten Wort und ein Leerzeichen im zweiten Wort entspricht einer Löschung des korrespondierenden Buchstabens im ersten Wort. Trotz der mathematischen

Äquivalenz von Editiertranskript und Alignment besteht unter einer anderen Betrachtungsweise doch ein Unterschied zwischen beiden, denn das Editiertranskript fokussiert den Blick auf die Veränderungsereignisse, die die Überführung des einen Wortes in das andere ermöglichen, während das Alignment nur eine bestehende Relation zwischen zwei Wörtern ausdrückt [31]. Die Berechnung eines paarweisen globalen Alignments mittels dynamischer Programmierung ist damit nahezu identisch zur Berechnung des Editiertranskripts, wobei anstelle der minimalen Kosten für die notwendigen Editieroperationen die maximale Ähnlichkeit zwischen zwei Wörtern berechnet wird. Erstmals wurde dies von Needleman und Wunsch [54] beschrieben und auf die Bestimmung der Ähnlichkeiten von Gensequenzen angewendet. Hirschberg erreichte eine Verbesserung der Berechnung des Editierabstands, indem er die Bestimmung von M nach dem divide-and-conquer-Prinzip in Teilprobleme aufteilte und so nur $O(n)$ Speicher verbrauchte [34]. Ein von Masek und Paterson entwickelter Algorithmus bietet eine bessere Zeitkomplexität, wobei er die Berechnung des Editierabstands zweier Wörter bei diskreten Kosten in $O(n^2/\log n)$ erreicht [49].

Ausblick

In Kapitel 9.1 findet sich die Beschreibung eines indexbasierten globalen Alignierungsverfahrens für zwei Wörter u, v. Dessen Grundidee ist es, durch die Vorberechnung einer Indexstruktur exakte Übereinstimmungen zwischen u und v zu finden, die bereits „perfekt“ aligniert sind. Anschließend können diese Matches durch den im nächsten Kapitel beschriebenen LCS-Algorithmus zu einem globalen Alignment zusammengesetzt werden.

2.5. Longest Common Subsequence (LCS)

Eng verknüpft mit dem Editierabstand d_L zweier Wörter u, v ist die Identifikation von Teilfolgen innerhalb selbiger.

Definition 2.5.1. (Teilfolge) Eine *subsequence* oder *Teilfolge* innerhalb eines Wortes $w = w_1 ... w_n$ ist ein Wort $s = w_{i_1} ... w_{j_k}$ mit $k \leq n$ und $i_1 < ... < j_k$.

Beispiel 2.5.1. Sei $w = abcab$, dann sind zum Beispiel abb oder bab Teilfolgen von w.

Definition 2.5.2 (LCS)**.** Seien $u, v \in \Sigma^*$ mit $m = |u|, n = |v|$ und s_{lcs} eine Teilfolge des Wortes u, dann heißt s_{lcs} *längste gemeinsame Teilfolge* oder *longest common subsequence*, falls s_{lcs} eine Teilfolge von v ist und die maximale Länge unter allen gemeinsamen Teilfolgen von u, v hat.

Beispiel 2.5.2. Seien $u, v \in \Sigma^*$ mit

$$u = aefbghcdije,$$
$$v = kalmbcodpeq,$$

dann stellt *abcde* eine *längste gemeinsame Teilfolge oder longest common subsequence* innerhalb von u und v dar.

Zuerst wird wieder der naive Ansatz betrachtet. Insgesamt existieren 2^m Teilfolgen innerhalb von u. Damit läge die Zeitkomplexität dieses Ansatzes bei $O(2^m * n)$. Es lässt sich zeigen, dass das LCS-Problem einen Spezialfall des Editierabstands darstellt, wenn bei der Berechnung nur noch Einfügungen oder Löschungen erlaubt werden [31], [18]. Damit kann das LCS-Problem wiederum in $O(n^2)$ durch dynamische Programmierung anhand einer Matrix der Größe $m \times n$ gelöst werden. Ebenfalls eine Verbesserung dieser Laufzeit stellt damit der LCS-Algorithmus von Hunt & Szymanski dar [37]. Dieses Verfahren läuft in $O((r+n)\log n)$, wobei r die Gesamtanzahl aller geordneten Positionspaare bezeichnet, in den zwei Zeichen aus u_i und v_j übereinstimmen. Im schlechtesten Falle benötigt dieses Verfahren $O(n^2 \log n)$ Berechnungszeit, im besten Fall jedoch nur $O(n \log n)$. Gusfield [31] schlägt ein vergleichbares Verfahren vor, indem das LCS-Problem anhand eines einfacheren Kombinatorikproblems, das die längste aufsteigende Folge innerhalb einer Folge von natürlichen Zahlen findet, gelöst werden kann. Damit wird ebenfalls eine Zeitkomplexität von $O(r \log n)$ und ein Speicherverbrauch von $O(n)$ erzielt. Im folgenden Abschnitt wird eine kurze Zusammenfassung dieses Verfahrens gegeben.

2.5.1. Longest Increasing Subsequence (LIS)

Definition 2.5.3. (Aufsteigende Teilfolge [31]) Sei $\Pi = \pi_1, ..., \pi_n$ eine Liste von natürlichen Zahlen mit $|\Pi| = n$. Eine *increasing subsequence* oder *aufsteigende Teilfolge* über Π ist eine Teilfolge $s_> = \pi_{i_1}, ..., \pi_{r_k}$ mit $k \leq n$ und $i_1 < ... < r_k$ sowie $\pi_{i_1} < \pi_{j_2} < ... < \pi_{r_k}$. Wenn die Länge $|s_>|$ einer aufsteigenden Teilfolge $s_>$ im Vergleich zu den Längen aller möglichen aufsteigenden Teilfolgen von Π maximal ist, wird sie *längste aufsteigende Teilfolge* oder *longest increasing subsequence (LIS)* bezüglich der Menge Π genannt.

Beispiel 2.5.3. Sei $\Pi = 2, 4, 1, 2, 4, 5$, dann sind zum Beispiel $\{2, 4, 5\}$ oder $\{1, 2, 4\}$ aufsteigende Teilfolgen von Π.

Definition 2.5.4. (Absteigende Teilfolge [31]) Sei $\Pi = \pi_1, ..., \pi_n$ eine Liste von natürlichen Zahlen mit $|\Pi| = n$. Eine *decreasing subsequence* oder *absteigende Teil-*

folge über Π ist eine Teilfolge $s_{\leq} = \pi_{i_1}, ..., \pi_{r_k}$ mit $k \leq n$ und $i_1 < ... < r_k$ sowie $\pi_{i_1} \geq \pi_{j_2} \geq ... \geq \pi_{r_k}$. Im Vergleich zu aufsteigenden Teilfolgen müssen bei absteigenden Teilfolgen die Werte nicht strikt fallen.

Beispiel 2.5.4. Sei $\Pi = 5, 3, 6, 3, 2, 9, 1$, dann sind zum Beispiel $\{5, 3, 2\}$, $\{5, 3, 3, 2, 1\}$ oder $\{9, 1\}$ absteigende Teilfolgen von Π.

Definition 2.5.5. (Cover [31]) Ein *Cover* $C(\Pi)$ ist eine Menge von absteigenden Teilfolgen aus Π, sodass alle Elemente $\pi_i \in \Pi$ in $C(\Pi)$ enthalten sind. Die Länge eines Covers $\left|C(\Pi)\right|$ ergibt sich aus der Anzahl seiner enthaltenen absteigenden Teilfolgen von Π. Ein Cover ist ein *kleinstes* Cover von Π, wenn seine Länge im Vergleich mit allen anderen Covern von Π minimal ist.

Beispiel 2.5.5. Sei $\Pi = 1, 6, 9, 2, 7, 8, 3, 4, 9, 10, 5$ ergibt sich für $C(\Pi)$ folgendes Cover der Länge 6:

$$C(\Pi) = \left|\begin{array}{c|c|c|c|c|c} 1 & 6 & 9 & 8 & 9 & 10 \\ & 2 & 7 & 4 & 5 & \\ & & 3 & & & \end{array}\right|$$

Lemma 2.5.1. (Gusfield [31]) Wenn die Länge einer aufsteigenden Folge $s_>$ über Π der Länge von $C(\Pi)$ entspricht ($|s_>| = \left|C(\Pi)\right|$) ist $s_>$ eine längste aufsteigende Teilfolge von Π und $C(\Pi)$ das kleinste Cover von Π.

Um die längste aufsteigende Folge bezüglich einer Zahlenfolge Π zu finden, kann diese nach Lemma 2.5.1 in ein kleinstes Cover überführt werden, sodass dieses eine aufsteigende Teilfolge enthält, die genau eine Zahl aus jeder absteigenden Teilfolge aus $C(\Pi)$ beinhaltet.

2.5.2. Konstruktion des Covers

Zur algorithmischen Konstruktion eines Covers $C(\Pi)$ kann die Zahlenfolge Π von links nach rechts durchlaufen werden und jedes Element π_i ans Ende der ersten absteigenden Folge geschrieben werden, wenn π_i kleiner oder gleich dem letzten Element dieser Folge ist. Falls π_i nicht kleiner oder gleich dem letzten Element der ersten absteigenden Folge ist, wird es solange mit allen nachfolgenden absteigenden Folgen verglichen, bis der Vergleich positiv ausfällt. Wird gar keine absteigende Teilfolge gefunden, an die π_i angehängt werden kann, bildet π_i den Beginn einer neuen absteigenden Teilfolge in $C(\Pi)$. Im obigen Beispiel ist dies etwa für die ersten drei Zahlen der Fall, da $1 < 6 < 9$. Ein so erzeugtes Cover wird als *greedy Cover* bezeichnet. Wenn Π bereits eine aufsteigende Folge etwa $1, 2, 3, 4...$ darstellt,

müssten für jedes Element aus Π neue absteigende Folgen zu $C(\Pi)$ hinzugefügt werden. Dies stellt somit also den schlechtesten Fall dar und führt zu einer Zeitkomplextiät von $O(n^2)$. Wenn eine sortierte Liste der jeweils letzten Elemente aller absteigenden Teilfolgen unterstützt wird, kann der Vergleich des letzten Elements durch binäre Suche auf dieser Liste in $O(n \log n)$ erfolgen.

2.5.3. Finden der LIS

Um die längste aufsteigende Folge in einem Cover in $C(\Pi)$ zu finden, kann folgendes Lemma betrachtet werden:

Lemma 2.5.2. (Gusfield [31]) Es existiert eine aufsteigende Teilfolge $s_>$ aus Π, sodass diese genau eine Zahl aus jeder absteigenden Folge aus $C(\Pi)$ enthält. Damit ist $s_>$ die längste mögliche Folge und $C(\Pi)$ das kleinste Cover von Π.

Zum Auffinden der LIS in einem Cover $C(\Pi)$ kann das Cover von hinten durchsucht werden, wobei bei einer beliebigen Zahl x aus $s_{<_n}$ ($n = \left|C(\Pi)\right|$) gestartet werden kann. Diese Zahl stellt den Endpunkt der LIS dar. Anschießend werden alle absteigenden Folgen $s_{<_{n-1}}, ..., s_{<_1}$ von oben nach unten durchsucht und jeweils die erste kleinere Zahl y im Vergleich zu x gesucht. Ist diese gefunden, wird $x = y$ gesetzt, y gemerkt und die nächste absteigende Folge durchsucht. Da das greedy Cover so nur einmal durchsucht wird, liegt die Laufzeit bei $O(n)$. Für das obige Beispiel wird damit als LIS die Folge $1, 6, 7, 8, 9, 10$ gefunden:

$$C(\Pi) = \left|\begin{array}{c|c|c|c|c|c} 1 & 6 & 9 & 8 & 9 & 10 \\ & 2 & 7 & 4 & 5 & \\ & & 3 & & & \end{array}\right|$$

2.5.4. Übersetzung des LCS-Problems in das LIS-Problem

Um das LCS-Problem mithilfe des Lösungsansatzes für das LIS-Problem zu behandeln, wird zunächst folgende Defintion betrachtet.

Definition 2.5.6. (Vorkommenshäufigkeit) Seien $u, v \in \Sigma^*$. $r(i)$ bezeichnet die Vorkommenshäufigkeit des i-ten Zeichens von u in v. Sei zusätzlich $r = \sum_{i=1}^{m} r(i)$.

Beispiel 2.5.6. Sei $u = aa$ und $v = aabba$, dann ist $r(1) = 3, r(2) = 3$ und $r = 6$.

Soll nun für zwei Wörter $u, v \in \Sigma^*$ deren LCS ermittelt werden, wird zuerst eine Liste $\Pi(u, v)$ der Länge r erzeugt, in der alle Zeichen von u durch die absteigend sortierte Liste der Positionen ihrer jeweiligen Vorkommen in v kodiert werden. Damit ergibt sich eine Bijektion der Form $\sigma_i \in \Sigma \rightarrow s_{<_i} \in \Pi$.

Beispiel 2.5.7. Sei $u = babcde$ und $v = dcbabca$ dann ergeben sich für die Zeichen von u als absteigende Folgen ihrer Positionen in v:

$a = 7, 4$

$b = 5, 3$

$c = 6, 2$

$d = 1$

$\Pi(u, v)$ enthält damit: $5, 3, 7, 4, 5, 3, 6, 2, 1$. Da das Zeichen e in v keine Entsprechung hat, kann es auch nicht Teil einer längsten gemeinsamen Folge sein, weswegen e auch keine absteigende Folge in $\Pi(u, v)$ darstellt.

Theorem 2.5.1. (Gusfield [31]) Jede aufsteigende Teilfolge $s_>$ aus $\Pi(u, v)$ gibt eine gleich lange gemeinsame Teilfolge von u und v an und umgekehrt.

Theorem 2.5.2. (Gusfield [31]) Das LCS-Problem kann in $O(r \log n)$ gelöst werden.

Zum Schluss folgt noch ein kurzer Blick auf die beiden ursprünglichen Beispielwörter $u = aefbghcdije$, $v = kalmbcodpeq$ (siehe Beispiel 2.5.1). Deren längste gemeinsame Folge $abcde$ kann mit dem LIS-Verfahren wie eben beschrieben bestimmt werden. Für die Positionen der Zeichen von u in v ergeben sich die absteigenden Folgen:

$a = 2$

$b = 5$

$c = 6$

$d = 8$

$e = 10$

Da kein Zeichen aus u in v mehrmals vorkommt und insgesamt nur die Zeichen a, b, c, d, e in u und v gemeinsam auftreten, ergeben sich nur einelementige Listen, sodass $\Pi(u, v)$ die Zahlenfolge $2, 5, 6, 8, 10$ enthält. Hieraus ergibt sich als Cover

$$C(\Pi) = \left| \; 2 \; \right| \; 5 \; \left| \; 6 \; \right| \; 8 \; \left| \; 10 \; \right|$$

das das Ablesen der längsten aufsteigenden beziehungsweise längsten gemeinsamen Teilfolge zwischen u und v sehr einfach macht. An diesem Beispiel wird noch

einmal deutlich, dass die Komplexität dieses Verfahrens maßgeblich von der Konstante r abhängt. Im Beispiel gilt aufgrund der Tatsache, dass alle gemeinsamen Zeichen von u und v nur ein einziges Mal auftreten $r = n$. Wie gut sich dieses Verfahren auch auf großen Datenmengen schlägt, hängt damit stark vom verwendeten Alphabet ab. Wenn etwa ein kleineres Alphabet benutzt wird, dessen Zeichen sich innerhalb von u und v stark wiederholen, steigt die Größe von r an und damit die gesamte Laufzeit. Wird jedoch ein Alphabet benutzt, dessen die Zeichen keine starken Wiederholungen innerhalb von u und v erwarten lassen, kann die längste gemeinsame Teilfolge im besten Fall sogar in $O(n \log n)$ (wenn $r = n$ gilt) bestimmt werden. Weitergehende Untersuchungen zum LCS- bzw. LIS-Problem können bei [62, 35, 36] und [24] nachgeschlagen werden. Eine für kleine Alphabete effizientere Lösung des LIS-Problems wurde von Kutz et al. [14] beschrieben.

Wenn anstelle einer gemeinsamen Teilfolge zwischen zwei Wörtern die *längste wiederholte Teilfolge* beziehungsweise *longest repeated subsequence (LRS)* innerhalb eines Wortes aus Σ^* errechnet werden soll, können die zur Lösung des LCS-Problems benutzten Methoden verwendet werden, wobei w mit sich selbst verglichen wird. Die LRS innerhalb des Wortes $w = abcc$ ist dementsprechend abc, da diese an Positionen $1, 2, 3$ und $1, 2, 4$ auftritt.

3. String-Indexstrukturen und effiziente Berechnung

In diesem Kapitel wird eine theoretische Beschreibung der SCDAWG-Struktur entwickelt. Dazu werden zunächst formale Definitionen verschiedener String-Indexstrukturen erläutert. Zuerst wird dafür der Suffixtrie formal eingeführt und dieser dann zum Suffixbaum, DAWG und CDAWG weiterentwickelt, sodass sich am Ende die formale Beschreibung der SCDAWG-Struktur ergibt. Gleichzeitig werden on-line Algorithmen, die eine direkte Konstruktion der vier Vorgängerstrukturen des SCDAWGs ermöglichen, beleuchtet, woraufhin ein kurzer Einblick in die on-line Konstruktion der SCDAWG-Struktur mit Hilfe des Algorithmus von Inenaga et al. folgt, der in Kapitel 5 ausführlich behandelt wird.

Zwischen den nachfolgend beschriebenen String-Indexstrukturen, Suffixtrie, Suffixbaum, DAWG und CDAWG bestehen Relationen, die in der folgenden Übersicht [20, 41] visuell dargestellt sind.

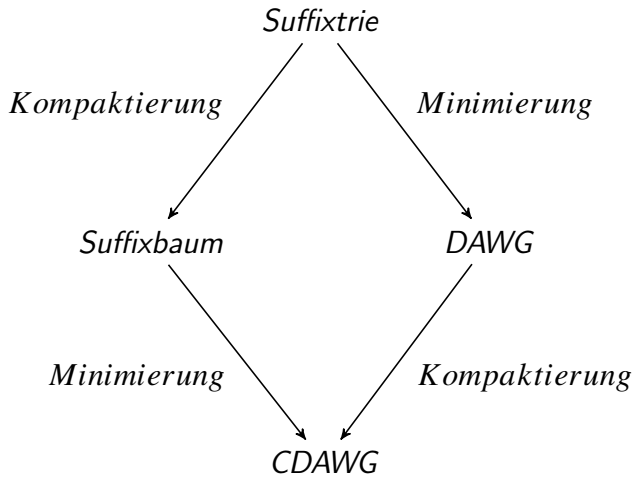


Abbildung 3.1.: Relationen der vier Indexstrukturen untereinander.

3.1. Suffixtrie

Als erste Indexstruktur, die die Infixe einer Menge von Wörtern W effizient speichert, kann zunächst ein *Trie* beziehungsweise ein *Präfixbaum*, der von de la Briandais [23] und Fredkin [27] entwickelt wurde, betrachtet werden. Ein Trie entspricht dabei einem Baum, in dem gemeinsam auftretende Präfixe eines jeden $w \in W$ zu gemeinsamen Knoten zusammengefasst werden.

Beispiel 3.1.1. Sei $W = \{Bau, Baum, Ball, Hund, Huhn\}$, so ergibt sich folgender *Trie(W)*:

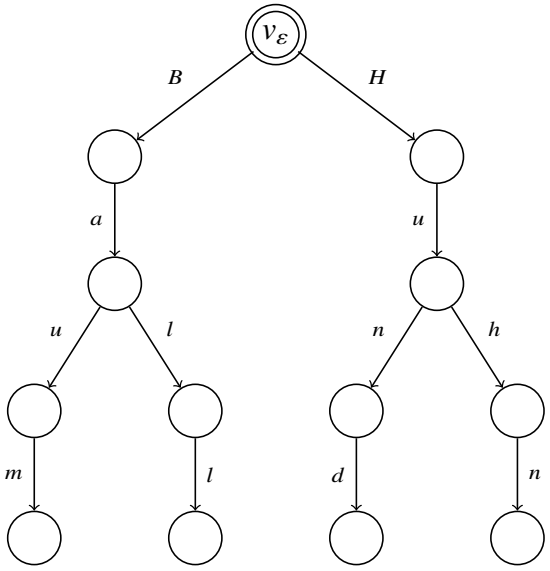


Abbildung 3.2.: Trie der Eingabewörter *Bau, Baum, Ball, Hund, Huhn.*

Die Präfixe *Ba, Bau* und *Hu* werden in *Trie(W)* durch gemeinsame Knoten zusammengefasst, was zur Vermeidung von Redundanzen führt. Damit stellt ein Trie zwar eine effiziente Methode dar, alle Präfixe einer gegebenen Menge von Wörtern zu speichern und sie auffindbar zu machen, Infixe und Suffixe wie zum Beispiel *und* oder *all* können jedoch nicht effizient gefunden werden, da es für diese Teilwörter keinen Pfad von der Wurzel aus gibt.

Damit Infixe wie *und* oder *all* ebenfalls in *Trie(W)* repräsentiert würden, müssten diese zu W hinzugefügt werden, woraufhin eigene Pfade für diese Wörter entstehen würden. Es lässt sich also folgern, dass neben den eigentlichen Eingabewörtern alle Suffixe zu W addiert werden müssten, damit alle Infixe in einem Trie zugänglich gemacht werden könnten. Betrachtet man die Menge aller Teilwörter

eines einzigen Beispielwortes *aabbccd*, die in folgender Tabelle zeilenweise nach ihrer Länge und spaltenweise nach ihrem Anfangsbuchstaben sortiert dargestellt sind,

	a	b	c	d	ε
1	*a*	*b*	*c*	*d*	
2	*aa*, *ab*	*bb*, *bc*	*cc*, *cd*		
3	*aab*, *abb*	*bbc*, *bcc*	*ccd*		
4	*aabb*, *abbc*	*bbcc*, *bccd*			
5	*aabbc*, *abbcc*	*bbccd*			
6	*aabbcc*, *abbccd*				
7	*aabbccd*				

Tabelle 3.1.: Alle Teilwörter mit Suffixen (rot) des Wortes *aabbccd*.

wird klar, dass ein Trie über allen Suffixen eines Wortes w verwendet werden kann, woraufhin zusätzlich alle Präfixe der Suffixe von w eigene Knoten bilden und damit für alle Infixe (etwa für *bbc*) von der Wurzel aus erreichbare Pfade entstehen.

Beispiel 3.1.2. Sei $w = aabbccd$, so ergibt sich folgender Suffixtrie:

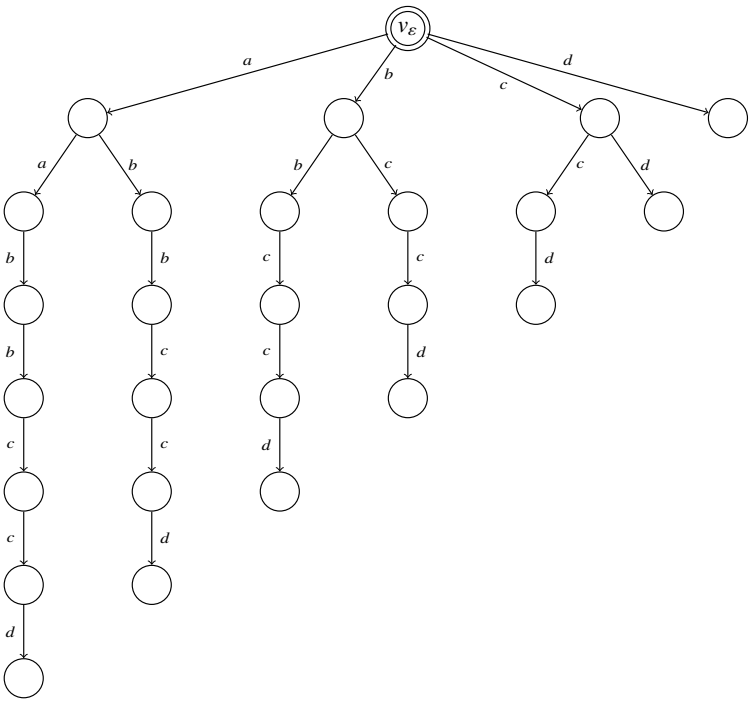


Abbildung 3.3.: Suffixtrie des Eingabewortes *aabbccd*.

Am Beispiel ist zu erkennen, dass alle Infixe von w von der Wurzel aus erreicht werden können. Hieraus lässt sich nun eine allgemeine Definition eines Suffixtries für ein gegebenes Wort $w \in \Sigma^*$ ableiten.

Definition 3.1.1 (Suffixtrie, vgl. [42])**.** Ein Suffixtrie *STrie(w)* über einer Zeichenkette w ist ein gerichteter Baum (V, E) (siehe Definition 1.3.1) für den gilt:

1. $V = \{x \mid x \in \mathit{Infix(w)}\}$.
2. $E = \{(x, \sigma, x\sigma) \mid \sigma \in \Sigma \text{ und } x, x\sigma \in \mathit{Infix(w)}\}$.

Aus Definition 3.1.1 lassen sich folgende Eigenschaften eines Suffixtries ableiten:

> Seien $v_k, v_l, v_m \in V$ und $\sigma, \gamma \in \Sigma$ sowie $E^{v_k \rightarrow} = \{e_1, \ldots, e_n\}$ mit $n \geq 1$ die Menge der ausgehenden Kanten von v_k, dann muss für $e_i = (v_k, \sigma, v_l)$ und $e_j = (v_k, \gamma, v_m)$ $\sigma \neq \gamma$ gelten.
>
> Für jedes Suffix w' aus w existiert ein Knoten $v \in V$, sodass es von der Wurzel v_ε ein Pfad $(v_\varepsilon, \ldots, v)$ gibt, wobei die Konkatenation der Labels dieses Pfads w' ergibt.
>
> Ein beliebiger Suffixtrie *STrie(w)* über einem Wort $w \in \Sigma^*$ hat immer genauso viele Knoten wie das Eingabewort w Infixe. Es gilt also: $|V| = |\mathit{Infix(w)}|$.

Für ein Wort w der Länge n, in dem alle Symbole unterschiedlich sind, gilt:

$$|\mathit{Infix(w)}| = \sum_{i=0}^{n}(n-i) = n \cdot n - \sum_{i-1}^{n} i =$$
$$n^2 - \frac{n(n-1)}{2} = n^2 - \frac{n^2-n}{2} = n^2 - \frac{n^2}{2} + \frac{n}{2} =$$
$$\frac{n^2}{2} + \frac{n}{2} = \frac{n^2+n}{2} = \frac{n(n+1)}{2}$$

Dabei stellt ein solches w den schlechtesten Fall für die Größe eines Suffixtries dar. Wenn beispielsweise $w = abcde$ mit $n = 5$, dann gilt für *STrie(w)*: $|V| = 15$. Die obere Schranke für die Knotenanzahl von *STrie(w)* liegt demnach $|V| = O(n^2)$. Für einen Suffixtrie über einem beliebigen $w \in \Sigma^*$ lässt sich als untere Grenze der durchschnittlichen Größe von $|V|$ ebenfalls ein quadratischer Wert abschätzen [76]. Zudem gilt für die Gesamtanzahl der Kanten von *STrie(w)* aus Definition 3.1.1: $|E| = |V| - 1$.

3.2. Suffixbaum

Wegen der quadratischen Anzahl an Knoten und Kanten und des damit einhergehenden hohen Speicherverbrauchs wurden komprimierte Varianten des Suffixtries entwickelt, die 1968 von Morrison [51] als Patricia-Tries eingeführt und wenig später durch Weiner [79] und McCreight [50] als Suffixbäume bekannt wurden. Zur Erstellung eines Suffixbaumes werden alle eindeutigen Pfade eines Suffixtries mit Ausgangsgrad 1 komprimiert, was einen Suffixbaum zu einem kompaktierten Suffixtrie macht. Hierfür kann folgende Äquivalenzrelation über den Infixen eines Wortes betrachtet werden:

Definition 3.2.1. (Start-Äquivalenz) Sei $w \in \Sigma^*$. Zwei Wörter $x, y \in \Sigma^*$ sind unter $x \sim_w^L y$ start-äquivalent in $w \iff startSet_w(x) = startSet_w(y)$, wobei $startSet_w(u)$ für $u \in \Sigma^*$ die Menge aller Startpositionen von u in w ergibt.

Die Äquivalenzklasse von $x \sim_w^L y$ wird durch $[x]_w^L$ notiert.

Wörter der Menge $\{u \mid u \notin Infix(w)\}$, für die gilt $startSet_w(u) = \emptyset$, werden durch eine weitere Äquivalenzklasse zusammengefasst.[1]

Lemma 3.2.1. Zudem gilt für $\sim_w^L$:

1. $\sim_w^L$ ist links-invariant bezüglich Σ^*.
2. Wenn $x \sim_w^L y$ gilt, ist entweder x ein Präfix von y oder y ein Präfix von x.
3. xy und x sind start-äquivalent gdw. auf jedes Vorkommen von x ein Vorkommen von y folgt.
4. Das längste Element $x \in \Sigma^*$ der Äquivalenzklasse $[x]_w^L$ bildet den *Repräsentanten* $Rep([x]_w^L)$. x ist damit ein Suffix von w oder es kommt in zwei verschiedenen unmittelbaren rechten Kontexten (Definition 1.1.8) in w vor.[2]

Beispiel 3.2.1. Betrachtet man alle Infixe des Beispielwortes *aabbccd*, ergibt sich für $\sim_w^L$ folgendes Bild:

[1] vgl. [8]
[2] vgl. [9]

	a	b	c	d	ε
1	a	b	c	d	
2	aa, ab	bb, bc	cc, cd		
3	aab, abb	bbc, bcc	ccd		
4	$aabb$, $abbc$	$bbcc$ $bccd$			
5	$aabbc$, $abbcc$	$bbccd$			
6	$aabbcc$, $abbccd$				
7	$aabbccd$				

Tabelle 3.2.: Unter $\sim_w^L$ äquivalente Infixe von $aabbccd$.

Anhand der Einfärbungen beziehungsweise der Startpositionen der in Tabelle 3.2 dargestellten Infixe lassen sich 11 Äquivalenzklassen bezüglich $\sim_w^L$ erkennen[3]:

$[a]_w^L = \{a\}$, $[b]_w^L = \{b\}$, $[c]_w^L = \{c\}$, $[d]_w^L = \{d\}$, $[\varepsilon]_w^L = \{\varepsilon\}$,
$[cd]_w^L = \{cd\}$,
$[ccd]_w^L = \{cc, ccd\}$,
$[bccd]_w^L = \{bc, bcc, bccd\}$,
$[bbccd]_w^L = \{bb, bbc, bbcc, bbccd\}$,
$[abbccd]_w^L = \{ab, abb, abbc, abbcc, abbccd\}$,
$[aabbccd]_w^L = \{aa, aab, aabb, aabbc, aabbccd\}$.

Die so resultierenden Äquivalenzklassen beinhalten alle start-äquivalenten Teilwörter von w, die entweder Suffixe von w sind oder in mindestens zwei verschiedenen unmittelbaren rechten Kontexten in w aufgetreten sind. Letztere repräsentieren im Beispiel die Infixe a, b und c mit Startpositionen: $startSet(a) = \{1, 2\}$, $startSet(b) = \{2, 3\}$ und $startSet(c) = \{5, 6\}$.

Beispiel 3.2.2. Sei $w = aabbccd$, so ergibt sich folgender Suffixbaum:

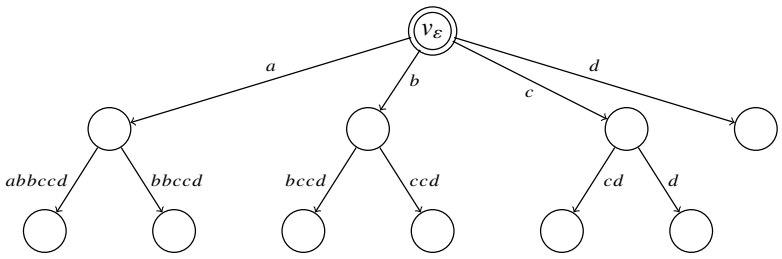


Abbildung 3.4.: Suffixbaum des Eingabewortes $w = aabbccd$.

[3]Die degenerierte Klasse, die alle Wörter $v \in \Sigma^* \setminus Infix(w)$ enthält, wird nicht weiter betrachtet.

Anhand der Äquivalenzklassen von $\sim_w^L$ wird ersichtlich, dass alle Pfade mit Ausgangsgrad 1, die im *STrie(w)* kein Suffix von w bilden, zu gemeinsamen Knoten zusammengefasst werden. Es lässt sich erneut eine formale Definition ableiten.

Definition 3.2.2 (Suffixbaum, vgl. [42])**.** Ein Suffixbaum *STree(w)* über einer Zeichenkette w ist ein Baum (V, E) (siehe Definition 1.3.1) für den gilt:

1. $V = \{Rep([x]_w^L) \mid x \in Infix(w)\}$.
2. $E = \left\{ (Rep([x]_w^L), \sigma\beta, Rep(Rep([x]_w^L)\sigma)) \;\middle|\; \begin{array}{l} \sigma \in \Sigma, \beta \in \Sigma^* \text{ und } x, x\sigma \in Infix(w), \\ Rep(Rep([x]_w^L)\sigma) = x\sigma\beta, \\ Rep([x]_w^L) \neq Rep(Rep([x]_w^L)\sigma) \end{array} \right\}$.

Aus Definition 3.2.2 lassen sich folgende Eigenschaften eines Suffixbaumes ableiten:

> Seien $v_k, v_l, v_m \in V$ und $\sigma, \gamma \in \Sigma$ und $\alpha, \beta \in \Sigma^*$ sowie $E^{v_k \rightarrow} = \{e_1, \ldots, e_n\}$ mit $n \geq 1$ die Menge der ausgehenden Kanten von v_k, dann muss für $e_i = (v_k, \sigma\alpha, v_l)$ und $e_j = (v_k, \gamma\beta, v_m)$ für $i \neq j$ stets $\sigma \neq \gamma$ gelten.
>
> Für jedes Suffix w' aus *Suffix(w)* existiert ein Knoten $v \in V$, sodass es von der Wurzel v_ε einen Pfad $(v_\varepsilon, \ldots, v)$ gibt, wobei die Konkatenation der Labels dieses Pfads w' ergibt.

Lemma 3.2.2. (McCreight [50]) Die maximale Knotenanzahl, die *STree(w)* für ein Wort der Länge n ($n > 1$) erreichen kann, liegt bei $|V| = 2n - 1$. Diese obere Schranke wird für *STree(w)* bei einem $w \in \Sigma^*$ der Form $w = \sigma^m\gamma$ ($\sigma, \gamma \in \Sigma, \sigma \neq \gamma$) erreicht, da nun in *STrie(w)* nur genau eine Kantenfolge mit Ausgangsgrad > 1 existiert, die unter $\sim_w^L$ zusammengefasst werden kann:

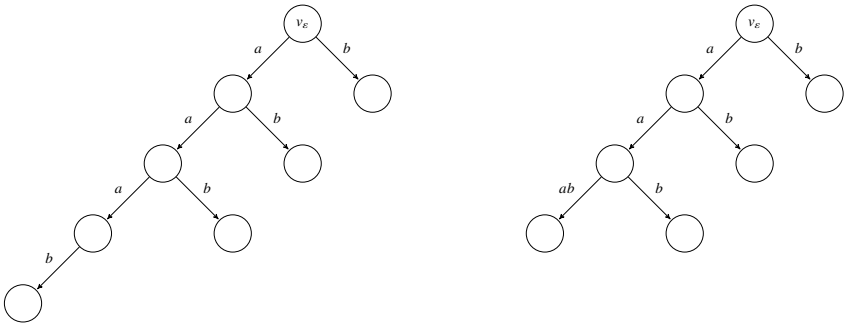


Abbildung 3.5.: Suffixtrie (links) und Suffixbaum (rechts) des Eingabewortes *aaab*.

Höchstens liegt die Anzahl der Knoten und Kanten eines Suffixbaumes *STree(w)* bei $O(n)$. Eine präzisere Abschätzung der unteren Schranke der durchschnittlichen Größe für beliebige $w \in \Sigma^*$ kann in [7] nachgeschlagen werden.

3.3. Directed Acyclic Word Graph (DAWG)

Eine zweite Möglichkeit, einen gegebenen Suffixtrie *STrie(w)* in eine speichereffizientere äquivalente Struktur zu überführen, wurde 1985 von Blumer et al. [9] eingeführt. Diesen Indexstrukturen, die als *Directed Acyclic Word Graphs* oder *DAWGs* bekannt sind, liegt die umgekehrte Betrachtungsweise der Infixe eines Wortes zugrunde, wie sie im zuvor eingeführten Suffixbaum verwendet wird.

Definition 3.3.1. (End-Äquivalenz) Sei $w \in \Sigma^*$. Zwei Wörter $x, y \in \Sigma^*$ sind unter $x \sim_w^R y$ end-äquivalent in $w \iff endSet_w(x) = endSet_w(y)$, wobei $endSet_w(u)$ für $u \in \Sigma^*$ die Menge aller Endpositionen von u in w ergibt.

Die Äquivalenzklasse von $x \sim_w^R y$ wird durch $[x]_w^R$ notiert.

Wörter der Menge $\{u \mid u \notin Infix(w)\}$, für die gilt $endSet_w(v) = \emptyset$, werden durch eine weitere Äquivalenzklasse zusammengefasst.[4]

Lemma 3.3.1. (Blumer et al. [9]) Zudem gilt für $\sim_w^R$:

1. $\sim_w^R$ ist rechts-invariant bezüglich Σ^*.
2. Wenn $x \sim_w^R y$ gilt, ist entweder x ein Suffix von y oder y ein Suffix von x.
3. xy und x sind end-äquivalent gdw. jedes Vorkommen von y von einem Vorkommen von x angeführt wird.
4. Das längste Element $x \in \Sigma^*$ der Äquivalenzklasse $[x]_w^R$ bildet den *Repräsentanten* $Rep([x]_w^R)$. x ist damit ein Präfix von w oder es kommt in zwei verschiedenen unmittelbaren linken Kontexten (Definition 1.1.8) in w vor.

Beispiel 3.3.1. Betrachtet man alle Infixe des Beispielwortes *aabbccd* ergibt sich für $\sim_w^R$ folgendes Bild:

	a	b	c	d	ε
1	*a*	*b*	*c*	*d*	
2	*aa*, *ab*	*bb*, *bc*	*cc*, *cd*		
3	*aab*, *abb*	*bbc*, *bcc*	*ccd*		
4	*aabb*, *abbc*	*bbcc bccd*			
5	*aabbc*, *abbcc*	*bbccd*			
6	*aabbcc*, *abbccd*				
7	*aabbccd*				

Tabelle 3.3.: Unter $\sim_w^R$ äquivalente Infixe von *aabbccd*.

[4] vgl. [8]

Anhand der Einfärbungen bzw. Endpositionen der in Tabelle 3.3 dargestellten Infixe lassen sich 10 Äquivalenzklassen bzgl. $\sim_w^R$ erkennen[5]:

$[a]_w^R = \{a\}$, $[b]_w^R = \{b\}$, $[c]_w^R = \{c\}$, $[\varepsilon]_w^R = \{\varepsilon\}$,
$[aa]_w^R = \{aa\}$,
$[aab]_w^R = \{aab, ab\}$,
$[aabb]_w^R = \{aabb, abb, bb\}$,
$[aabbc]_w^R = \{aabbc, abbc, bbc, bc\}$,
$[aabbcc]_w^R = \{aabbcc, abbcc, bbcc, bcc, cc\}$,
$[aabbccd]_w^R = \{aabbccd, abbccd, bbccd, bccd, ccd\}$.

Nach Lemma 3.3.1 beinhaltet jede Äquivalenzklasse solche Infixe, die entweder mindestens in zwei verschiedenen unmittelbaren linken Kontexten aufgetreten sind oder aber Präfixe von w sind.

Beispiel 3.3.2. Sei $w = aabbccd$, so ergibt sich folgender DAWG:

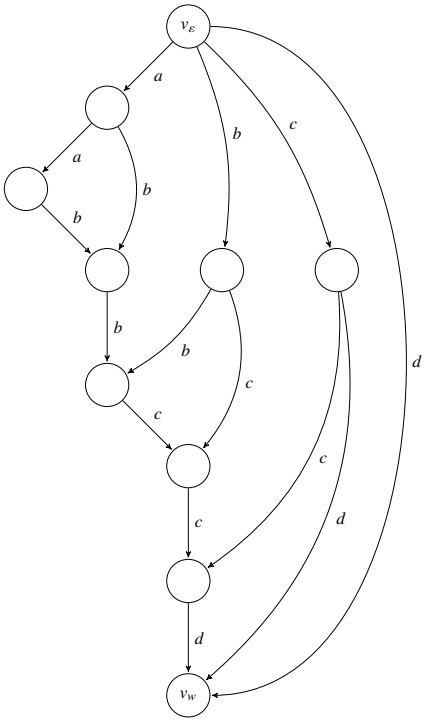


Abbildung 3.6.: DAWG des Eingabewortes $aabbccd$.

[5]Die degenerierte Klasse, die alle Wörter $v \in \Sigma^* \setminus \mathit{Infix}(w)$ enthält, wird nicht weiter betrachtet.

Wiederum lässt sich aus der zugrundeliegenden Äquivalenzrelation ($\sim_w^R$) eine formale Defintion eines DAWGs ableiten.

Definition 3.3.2 (DAWG, vgl. [42])**.** Ein DAWG *DAWG(w)* über einer Zeichenkette w ist ein gerichteter azyklischer Graph (V, E) (siehe Definition 1.2.7), für den gilt:

1. $V = \{Rep([x]_w^R) \mid x \in \mathit{Infix(w)}\}$.
2. $E = \{Rep([x]_w^R), \sigma, Rep([x\sigma]_w^R) \mid \sigma \in \Sigma \text{ und } x, x\sigma \in \mathit{Infix(w)}\}$.

Aus Definition 3.3.2 lassen sich folgende Eigenschaften eines DAWGs ableiten:

> Seien $v_k, v_l, v_m \in V$ und $\sigma, \gamma \in \Sigma$ und $\alpha, \beta \in \Sigma^*$ sowie $E^{v_k\rightarrow} = \{e_1, \ldots, e_n\}$ mit $n \geq 1$ die Menge der ausgehenden Kanten von v_k, dann muss für $e_i = (v_k, \sigma, v_l)$ und $e_j = (v_k, \gamma, v_m)$ $\sigma \neq \gamma$ gelten.
>
> Es existiert ein Knoten $v_w \in V$, der Sink genannt wird. Für jedes Suffix w' aus w existiert ein Pfad $(v_\varepsilon, \ldots, v_w)$, wobei die Konkatenation der Labels dieses Pfads w' ergibt.

Lemma 3.3.2. (Blumer et al. [9]) Die maximale Knotenanzahl, die *DAWG(w)* für ein Wort der Länge n ($n > 1$) erreichen kann, liegt bei $|V| = 2n - 1$. Diese obere Schranke wird für *DAWG(w)* bei einem $w \in \Sigma^*$ der Form $w = \sigma\gamma^n$ ($\sigma, \gamma \in \Sigma, \sigma \neq \gamma$) erreicht, da nun *STrie(w)* genau zwei Zustände enthält, die unter $\sim_w^R$ zusammengefasst werden können:

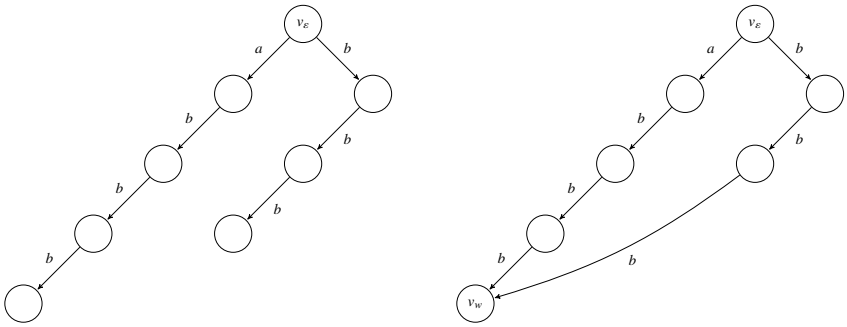


Abbildung 3.7.: Suffixtrie (links) und DAWG (rechts) des Eingabewortes *baaa*.

Theorem 3.3.1. (Blumer et al. [9]) Die maximale Kantenanzahl, die *DAWG(w)* erreichen kann, liegt für $w \in \Sigma^*$, $n \geq 1$ bei $|E| \leq 3n - 3$. Im Fall von $w = \sigma\gamma^m$ ($\sigma, \gamma \in \Sigma, \sigma \neq \gamma$) liegt sie bei $|E| = 2n - 1$.

Höchstens liegt die Anzahl der Knoten und Kanten eines DAWG *DAWG(w)* bei $O(n)$. Eine präzisere Abschätzung der unteren Schranke der durchschnittlichen Größe für beliebige $w \in \Sigma^*$ kann in [7] nachgeschlagen werden.

Konstruktion DAWG aus Suffixtrie

Um algorithmisch aus einem Suffixtrie einen DAWG herzustellen, kann die in Kapitel 1.6.1 vorgestellte Konstruktion benutzt werden. Aus der Perspektive deterministischer endlicher Automaten betrachtet, ergibt sich für den Suffixtrie aus Beispiel 3.1.2 folgender vollständiger DEA A_{STrie}, dessen Endzustände die Suffixe von *aabbccd* repräsentieren:

Abbildung 3.8.: Vollständiger deterministischer endlicher Automat A_{STrie} des Beispiel-Suffixtries des Eingabewortes *aabbccd*.

Werden die Zustände von A_{STrie} nun mittels der zuvor beschriebenen Konstruktion auf ihre Äquivalenz bzw. Ununterscheidbarkeit hin geprüft, ergibt sich für die Äquivalenzrelation $\sim_5$ eine maximale Verfeinerung:

$Q_1^{(5)} = \{q_\varepsilon\}, Q_2^{(5)} = \{q_2\}, Q_3^{(5)} = \{q_6\}, Q_4^{(5)} = \{\emptyset\}, Q_5^{(5)} = \{q_7, q_{11}\}, Q_6^{(5)} = \{q_8, q_{12}, q_{16}\}, Q_7^{(5)} = \{q_9, q_{13}, q_{17}, q_{20}\}, Q_8^{(5)} = \{q_3\}, Q_9^{(5)} = \{q_{10}, q_{14}, q_{18}, q_{21}, q_{23}\}, Q_{10}^{(5)} = \{q_4\}, Q_{11}^{(5)} = \{q_1, q_5, q_{15}, q_{19}, q_{22}, q_{24}, q_{25}\}.$

Es existieren fünf Klassen, die zwei oder mehrere ununterscheidbare Zustände enthalten. Aus dieser Einteilung lässt sich nun ein reduzierter DEA A_{STrie_R} ableiten, der alle Suffixe des Eingabestrings erkennt:

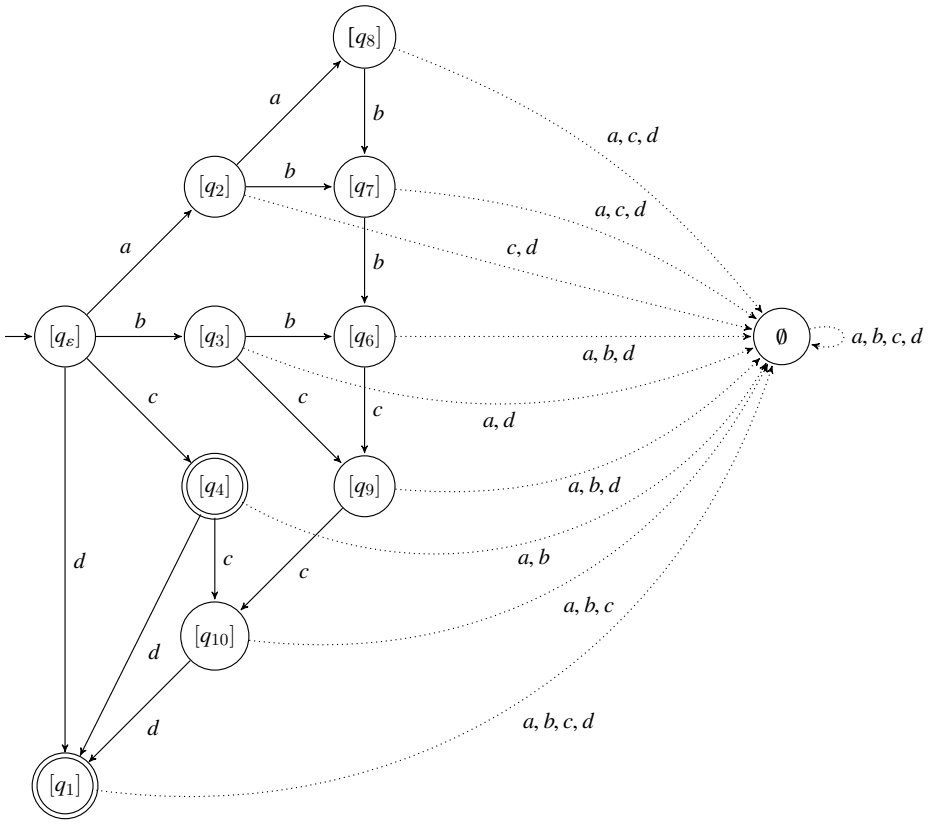


Abbildung 3.9.: Reduzierter Suffixautomat des Eingabewortes *aabbccd*.

Nach Lemma 1.6.3 ist A_{STrie_R} der kleinste DEA, der die Suffixe von *aabbccd* beziehungsweise die Sprache von A_{STrie_R} erkennt, wenn der Index seiner Nerode-Äquivalenzrelation $|\Sigma^*/\sim_L|$ der Anzahl seiner Zustände $|Q|$ entspricht.

Für A_{STrie_R} ergeben sich diese Äquivalenzklassen unter der Nerode-Relation $\sim_L$:

$[\varepsilon] = \{\varepsilon\}\ s = aabbccd$,
$[a] = \{a\}\ s = abbccd$, $[b] = \{b\}\ s = bccd, ccd$, $[c] = \{c\}, s = cd, d$,
$[aa] = \{aa\}\ s = bbccd$,
$[aab] = \{aab, ab\}\ s = bccd$,
$[aabb] = \{aabb, abb, bb\}\ s = ccd$,
$[aabbc] = \{aabbc, abbc, bbc, bc\}\ s = cd$,
$[aabbcc] = \{aabbcc, abbcc, bbcc, bcc\}\ s = d$,
$[aabbccd] = \{aabbccd, abbccd, bbccd, bccd, ccd, cd, d\}\ s = \varepsilon$,
$[ba] = \{ba, bba, abaa, dd, \ldots\}\ s = \emptyset$.

Da sich für $\sim_L$ ebenfalls elf Äquivalenzklassen ergeben und dies $|Q|$ von A_{STrie_R} entspricht, repräsentiert dieser einen minimalen DEA. Für das Beispiel ist zudem erkennbar, dass bis auf die Klasse $[ba]$, die alle Wörter enthält, die nicht in $L(A_{STrie_R})$ sind, $\sim_w^R = \sim_L$ gilt. Für eine formale Sprache $L = \mathit{Suffix}(w)$ lässt sich die Äquivalenz der Nerode-Relation und der Relation der End-Äquivalenz zeigen.

Beweis 3.3.1 (Äquivalenz von $\sim_{\mathit{Suffix}(w)}$ und $\sim_w^R$)**.**

Sei $L = \mathit{Suffix}(w)$ eine formale Sprache.
$\forall x, y \in \Sigma^*$ gilt :
$x \sim_{\mathit{Suffix}(w)} y \iff (\forall z \in \Sigma^* : xz \in \mathit{Suffix}(w) \iff yz \in \mathit{Suffix}(w)) \iff$
$\mathit{Endset}(x) = \mathit{Endset}(y) \iff x \sim_w^R$.■

Wenn bei A_{STrie_R} auf Vollständigkeit verzichtet wird, lässt sich demnach folgern:

Korollar 3.3.1. Die Menge der Knoten (V) eines DAWG *DAWG(w)* entspricht der Menge der Zustände (Q) eines partiellen DEA A_{STrie_R}.

Hieraus folgt nun, dass durch die Minimierung eines Suffixtries *STrie(w)* algorithmisch eine DAWG-Struktur erzeugt wird, wobei der resultierende DEA A_{STrie_R} einen minimalen Automaten darstellt, der genau die Menge *Suffix(w)* erkennt.

Würden alle Zustände eines partiellen DEA A_{STrie_R} zur Menge der Endzustände hinzugefügt, würde dieser *Infix(w)* erkennen. Blumer et. al [9] konnten jedoch zeigen, dass dieser Automat, der *Infix(w)* erkennt, nicht immer der minimale Automat ist, sondern dass dies nur gilt, wenn w die Form $\sigma_1 \ldots \sigma_n \gamma$ $(\sigma_i, \gamma, \in \Sigma)$ mit $\gamma \notin \{\sigma_1, \ldots, \sigma_n\}$ hat. Soll ein DAWG für ein Wort w minimal sein, kann dies auch durch das Anfügen eines Spezialzeichens $\gamma \notin \Sigma$ am Ende von w erreicht werden. Durch das Anfügen eines solchen Spezialzeichens wird sichergestellt, dass jede Kante $e = (v_k, \gamma, v_m)$ mit $v_k \gamma \in$ *Suffix(w)* zum Sinkknoten führt.

3.4. Compact Directed Acyclic Word Graph (CDAWG)

Aus einem minimierten (kompaktierten) Suffixtrie lässt sich durch erneute Kompaktierung (Minimierung) aus einem DAWG (Suffixbaum) ein *Compact Directed Acyclic Word Graph* bzw. *CDAWG* herstellen. Diese Strukturen wurden von Blumer et al. [8] als speichereffizientere Weiterentwicklung der DAWGs im Jahre 1987 eingeführt. Als Grundlage dient die Äquivalenzrelation $\sim_w$, die die Eigenschaften von $\sim_w^L$ und $\sim_w^R$ vereint.

Lemma 3.4.1. (Blumer et al. [8])

$$\sim_w \text{ ist die transitive Hülle von } \sim_w^L \cup \sim_w^R .$$

Für die Beispielklassen der beiden Relationen $\sim_w^L$ und $\sim_w^R$ des Wortes *aabbcc* entstehen durch Bildung der transitiven Hülle über der Vereinigung der beiden Relationen folgende neue Äquivalenzklassen für $\sim_w$:

$[a]_w = \{a\}$, $[b]_w = \{b\}$, $[c]_w = \{c\}$, $[\varepsilon]_w = \{\varepsilon\}$,
$[aabbccd]_w = \{aabbccd, aabbc, abbcc, abbccd, aabb, abbc, bbcc, bbccd, aab, abb, bbc, bcc, bccd, aa, ab, bb, bc, cc\}$.

Beispiel 3.4.1. Sei $w = aabbccd$, so ergibt sich folgender CDAWG:

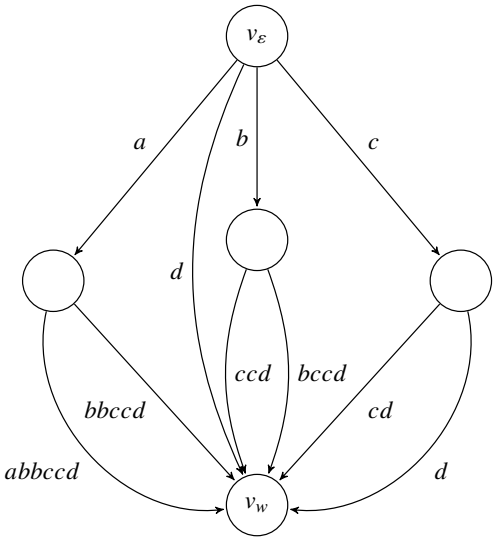


Abbildung 3.10.: CDAWG des Eingabewortes *aabbccd*.

Wiederum lässt sich aus der zugrundeliegenden Äquivalenzrelation ($\sim_w$) eine formale Definition eines CDAWGs ableiten.

Definition 3.4.1 (CDAWG, vgl. [42])**.** Ein CDAWG *CDAWG(w)* über einer Zeichenkette w ist ein gerichteter azyklischer Graph (V, E) (siehe Definition 1.2.7) für den gilt:

1. $V = \{Rep([x]_w) \mid x \in Infix(w)\}$.
2. $E = \left\{ (Rep([x]_w), \sigma\beta, Rep(Rep([x]_w)\sigma) \;\middle|\; \begin{array}{l} \sigma \in \Sigma, \alpha, \beta \in \Sigma^* \text{ und } x \in Infix(w), \\ Rep(Rep([x]_w)\sigma) = \alpha Rep([x]_w)\sigma\beta, \\ Rep([x]_w) \neq Rep(Rep([x]_w)\sigma) \end{array} \right\}$.

Aus Definition 3.4.1 lassen sich folgende Eigenschaften eines CDAWGs ableiten:

> Seien $v_k, v_l, v_m \in V$ und $\sigma, \gamma \in \Sigma$ und $\alpha, \beta \in \Sigma^*$ sowie $E^{v_k\rightarrow} = \{e_1, \ldots, e_n\}$ mit $n \geq 1$ die Menge der ausgehenden Kanten von v_k, dann muss für $e_i = (v_k, \sigma\alpha, v_l)$ und $e_j = (v_k, \gamma\beta, v_m)$ $\sigma \neq \gamma$ gelten.
>
> Es existiert ein Knoten $v_w \in V$, der Sink genannt wird. Für jedes Suffix w' aus w existiert ein Pfad $(v_\varepsilon, \ldots, v_w)$, wobei die Konkatenation der Labels dieses Pfads w' ergibt.

Lemma 3.4.2. (Blumer et al. [8], Crochemore und Vérin [20]) Die maximale Knotenanzahl, die *CDAWG(w)* für ein Wort der Länge n ($n > 1$) erreichen kann, liegt bei $|V| = n + 1$. Diese obere Schranke wird für *CDAWG(w)* bei einem $w \in \Sigma^*$ der Form $w = \sigma^n$ ($\sigma \in \Sigma$) erreicht. Die maximale Anzahl an Kanten liegt bei $|E| \leq 2|w| - 2$ und wird bei $w = \sigma^n\gamma$ ($\sigma, \gamma \in \Sigma, \sigma \neq \gamma$) erreicht.

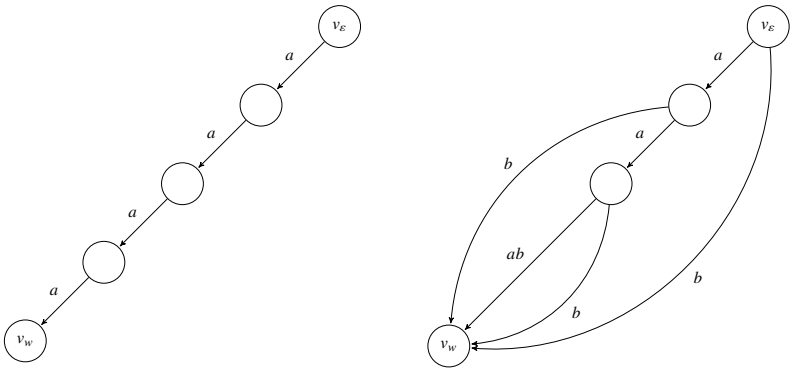


Abbildung 3.11.: CDAWGs der Wörter *aaaa* (links) und *aaab* (rechts).

Der Platzbedarf eines CDAWGs liegt wiederum höchstens bei $O(n)$, sowohl Knoten- als auch Kantenanzahl liegen aber unter der eines Suffixbaumes oder DAWGs. Eine präzisere Abschätzung der unteren Schranke der durchschnittlichen Knotenanzahl für beliebige $w \in \Sigma^*$ kann in [7] nachgeschlagen werden.

Ausblick

Bei der direkten on-line Konstruktion von Suffixbäumen und CDAWGs werden manche Suffixe nicht explizit durch einen Knoten, sondern nur implizit repräsentiert. Inenaga et. al [41] unterscheiden deshalb zwischen einer zur on-line Konstruktion geeigneten Defintion eines Suffixbaumes beziehungsweise CDAWGs und den bisherigen Definitionen (siehe Definitionen 3.2.2 und 3.4.1). Es lässt sich jedoch zeigen, dass sobald ein eindeutiges Symbol am Ende jedes Eingabewortes steht, das sonst an keiner anderen Stelle innerhalb eines Wortes auftritt, beide Definitionen äquivalent sind [41].

3.5. On-line Konstruktion der Indexstrukturen

Aufgrund des quadratischen Speicherbedarfs stellt es in der Praxis keine sinnvolle Methode dar, zuerst einen Suffixtrie zu konstruieren, um diesen dann anschließend zu einem Suffixbaum, DAWG (siehe Abschnitt 3.3) beziehungsweise CDAWG zu reduzieren. Somit wird nun die Frage behandelt, wie die vier Indexstrukturen direkt zu einem gegebenen Eingabewort w konstruiert werden können. Besondere Aufmerksamkeit liegt dabei auf der *on-line* Eigenschaft dieser Algorithmen. On-line bedeutet, dass die Eingabe nicht a priori bekannt sein muss. Die Zeichen von w werden von links nach rechts der Reihe nach verarbeitet, ohne dass w zuvor komplett gelesen werden muss. Da, je kleiner und effizienter eine Indexstruktur ist, die Anforderungen an den zugehörigen on-line-Algorithmus zunehmen, entspricht die Reihenfolge, in der die Indexstrukturen behandelt werden, der des vorherigen Kapitels. Obwohl diese Strategie damit nicht der historischen Entwicklung entspricht[6], stellt sie doch eine sinnvolle Vorgehensweise dar. Der Grund dafür liegt auch in der etwas eingängigeren Relation zwischen Suffixtrie und Suffixbaum, die bereits zuvor erläutert wurde. Diese Verfahrensweise wird auch in [41] verfolgt.

3.5.1. Algorithmus von Ukkonen - Suffixtrie

Nachfolgend wird nun die on-line Konstruktion der Suffixtries, die von Ukkonen in [75] gegeben wird, betrachtet. Dabei werden einige wichtige Konzepte eingeführt, die zum Verständnis der nachfolgenden Algorithmen unerlässlich sind.

[6] Der erste on-line Algorithmus zur Konstruktion eines DAWGs wurde bereits 1985 von Blumer et al. [9] vorgestellt und ein im obigen Sinne on-line arbeitender Algorithmus zur Konstruktion von Suffixbäumen wurde erst 1995 von Ukkonen [75] eingeführt.

Erweiterung des Suffixtries

Als erstes wird die Definition des Suffixtries *STrie(w)*, der in 3.1.1 als gerichteter Baum beschrieben wird, um zwei zusätzliche Eigenschaften erweitert. Ein erweiterter Suffixtrie *STrie'(w)* erhält zunächst eine zusätzliche Funktion, die wie folgt definiert wird.

Definition 3.5.1. (Suffixlink) Seien $v_i, v_j \in V$ zwei Knoten von *STrie'(w)* $= (V, E)$ und $\sigma \in \Sigma$. Ein *Suffixlink* oder *Suffixpointer* ist eine Abbildung $v_j \rightarrow v_i$, die durch die Funktion $sl(v)$ ausgedrückt wird. Es gilt $sl(v_j) = v_i$, wenn v_j die Form σv_i hat.

Da *STrie'(w)* Knoten für alle Elemente aus *Infix(w)* besitzt, und diese mit jeweils einem Zeichen verbunden sind, existiert von jedem Knoten v_j aus ein Suffixlink zu dem Knoten v_i der das nächste kürzere Suffix von v_j repräsentiert. Mit Hilfe der Suffixlinks können damit zu jedem Zeitpunkt durch Verfolgung von Suffixlink-Ketten zu jedem Knoten v_j dessen Suffixe beginnend beim längsten bis zum kürzesten abgefragt werden. Dieses Konzept spielt ein Schlüsselrolle bei der Konstruktion der vier behandelten Indexstrukturen in linearer Zeit. Erstmalig wurden dabei Suffixlinks von McCreight [50] bei dessen Suffixbaum-Konstruktion eingesetzt. Suffixlinks spielen aber auch ein wichtige Rolle innerhalb des im letzten Kapitel erwähnten Aho-Corasick Algorithmus [1], wo sie als *failure-transitions* bezeichnet werden. Ukkonen merkte hierbei an, dass ein um Suffixlinks erweiterter Suffixtrie dem im Aho-Corasick Algorithmus benutzten Trie entspräche [75]. Da das leere Wort ε nach Definition 1.1.7 ein Suffix jedes Wortes aus Σ^* ist, führen in *STrie'(w)* von allen Knoten v_i, die ein einelementiges Infix von w darstellen, Suffixlinks zu v_ε. Zusätzlich zu $sl(v)$ wird ein Hilfsknoten $v_\perp$ eingefügt, der noch oberhalb des Wurzelknotens v_ε liegt und somit die Konstruktion etwas vereinfacht, da durch ihn Fallunterscheidungen bezüglich des Wurzelknotens entfallen.

Definition 3.5.2. (Hilfsknoten $v_\perp$) Sei $v_\perp \notin V$ und $V' = V \cup \{v_\perp\}$ die Knotenmenge des erweiterten Suffixtries *STrie'(w)*. Dann ist $E' = E \cup \{(v_\perp, \sigma, v_\varepsilon) \mid \sigma \in \Sigma\}$ die Menge der Übergänge des erweiterten Suffixtries. Zudem gilt: $sl(v_\varepsilon) = v_\perp$ sowie $sl(v_\perp) = \emptyset$.

Ausgehend von $v_\perp$ existiert damit für jedes Zeichen aus Σ ein Übergang, der zum Wurzelknoten führt. $v_\perp$ repräsentiert damit alle Inversen σ^{rev} der Zeichen des endlichen Alphabets Σ. Um nicht jede dieser Kanten zeichnen zu müssen, wird in allen Darstellungen, in denen $v_\perp$ auftritt, ein einziger Übergang mit Σ als Übergangssymbol benutzt. Die folgende Abbildung zeigt den erweiterten Suffixtrie des Wortes $w = cbca$, dessen Knoten mit entsprechenden Suffixlinks (rot) verbunden sind.

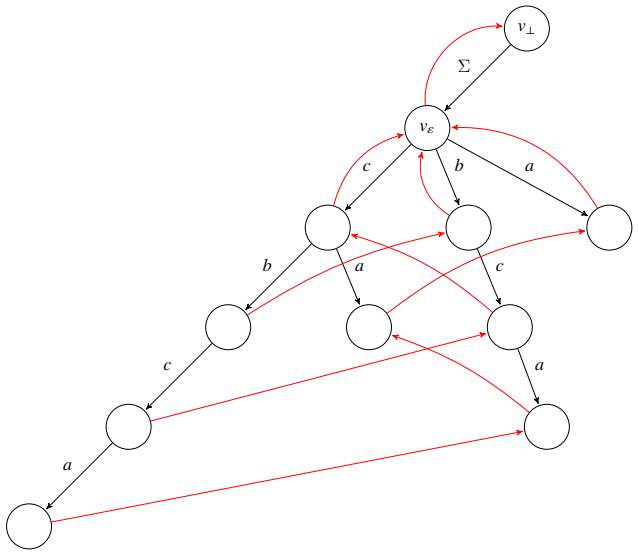


Abbildung 3.12.: Suffixtrie für $w = cbca$ mit Suffixlinks (rot).

In der Abbildung ist zu erkennen, dass die Suffixlinks von jedem Knoten v_j aus *STrie'(w)* eine direkte Verbindung zum nächstlängeren Suffix von v_j herstellen. Folgt man etwa dem Suffixlink von dem Knoten, der *bca* repräsentiert, so erreicht man direkt denjenigen Knoten, der das Suffix *ca* abbildet.

Konstruktion des Suffixtries

Im ersten Schritt der Konstruktion von *STrie'(w)* werden Wurzelknoten v_ε und Hilfsknoten $v_\perp$ hinzugefügt und diese, wie im vorherigen Abschnitt erläutert, miteinander verbunden. Nun werden die Zeichen von $w = w_1 \ldots w_n$ der Reihe nach von links nach rechts verarbeitet, bis w erschöpft ist. Beginnend mit dem Wurzelknoten wird geprüft, ob ein Übergang mit dem ersten Zeichen w_1 existiert. Ist dies nicht der Fall, wird ein neuer Knoten v_1 eingefügt und mit einer Kante $(v_\varepsilon, w_1, v_1)$ mit v_ε verbunden. Der Suffixlink der Wurzel (initial hinzugefügt) wird nun verfolgt und der Hilfsknoten besucht. Da dieser für alle Zeichen aus Σ Übergänge besitzt, ist die Prüfung, ob von dort aus eine Kante mit w_1 existiert, ebenfalls erfolgreich und es besteht kein Bedarf, nochmals einen Knoten einzufügen. Somit wird ein neuer Suffixlink hinzugefügt, sodass $sl(v_1) = v_\varepsilon$, und das nächste Zeichen w_2 eingelesen. Die Verarbeitung aller weiteren Zeichen $w_2 \ldots w_n$ läuft nun gleichermaßen ab. Zuerst wird immer dem alten Blatt v_{l-1} (Siehe Abbildung 3.13, Schritt 3) das neue Blatt v_l hinzugefügt und durch eine w_i-Kante mit v_{l-1} verbunden. Anschließend werden die Suffixpointer verfolgt und an jedem so besuchten Knoten v_k geprüft, ob dort ein w_i-Übergang existiert. Wenn dies nicht der Fall ist, werden ein neuer Knoten

v_{ki} und eine Kante (v_k, w_i, v_{ki}) erzeugt. v_{ki} stellt immer ein Suffix des neuen Blattes v_l dar. Deshalb wird nun ein neuer Suffixlink mit $sl(v_l) = v_{ki}$ eingefügt. Bestünde von v_k schon ein w_i-Übergang, müsste nur der Suffixlink von v_l nach v_{ki} eingefügt werden. Wenn von v_k schon ein w_i-Übergang existiert, kann die Verfolgung der Suffixlinks abgebrochen werden, andernfalls wird wiederum der Suffixlink von v_k aus verfolgt. Durch das Traversieren der Suffixlink-Ketten wird also sichergestellt, dass in jedem Durchgang beginnend vom letzten hinzugefügten Blatt alle Suffixe, die durch ein neues Zeichen w_i entstehen, eingefügt werden. Zur Verdeutlichung werden die einzelnen Konstruktionsschritte noch einmal für $w = cbca$ angeführt.

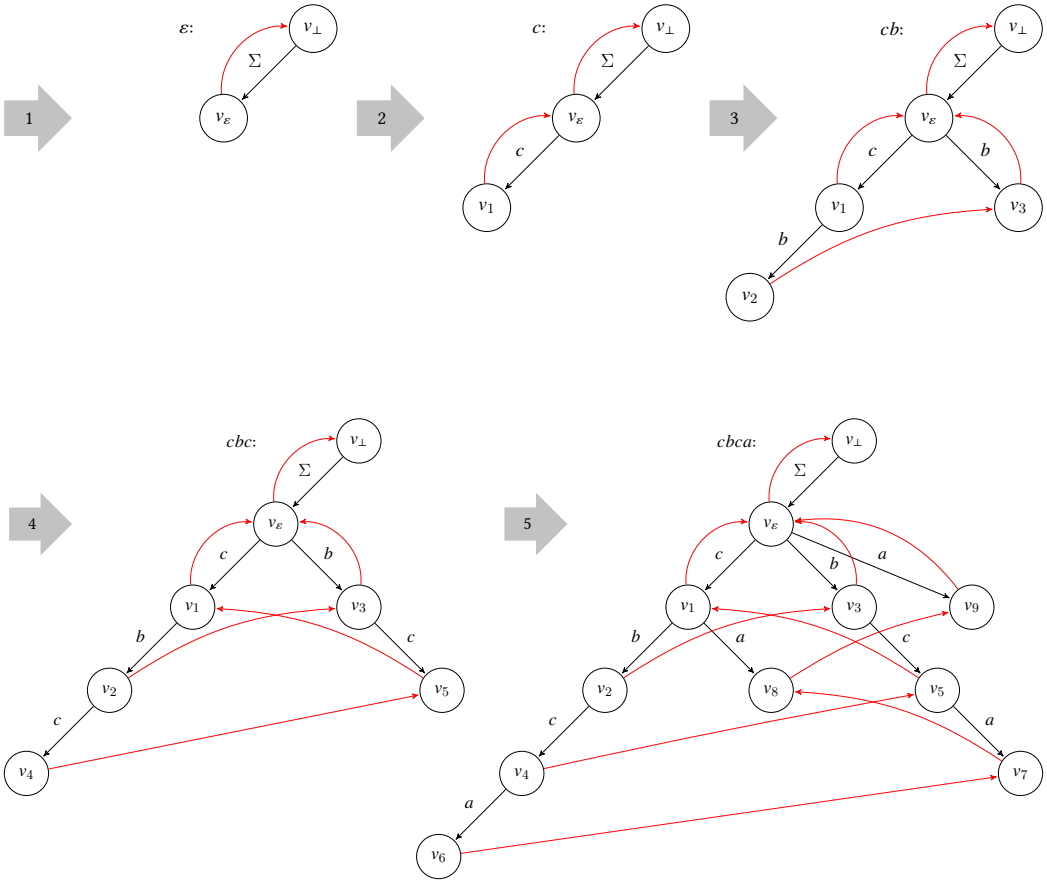


Abbildung 3.13.: Suffixtrie Konstruktionschritte für w=$cbca$.

Ukkonen's Algorithmus zum Aufbau eines Suffixtries läuft damit sozusagen „rundenbasiert“ ab, was bedeutet, dass wann immer ein neues Zeichen hinzugefügt wird, durch die Traversierung der Suffixlink-Ketten alle neuen Suffixe, die entstehen, dem Graphen angefügt werden. Damit stellt das Verfahren klar einen on-line

Algorithmus im obigen Sinne dar. Aufgrund der Anwendung der Suffixlinks entspricht der Zeitverbrauch dieses Algorithmus' der Größe des konstruierten Suffxtries, was zu folgendem Theorem führt.

Theorem 3.5.1. (Ukkonen [75]) Ein Suffixtrie *STrie'(w)* über einem endlichen Alphabet Σ kann in $O(n^2)$ Zeit- und Speicherplatzverbrauch konstruiert werden.

3.5.2. Algorithmus von Ukkonen - Suffixbaum

Das nachfolgend beschriebene Verfahren zur Konstruktion eines Suffixbaumes wurde von Ukkonen [75] 1995 erstmalig vorgestellt. Es basiert auf den Grundlagen der vorherigen Konstruktion des Suffixtries und stellt damit verglichen mit den älteren Suffixbaum-Konstruktionen von Weiner [79] (1973) und McCreight [50] (1976) ein echtes on-line Verfahren dar. Alle drei Algorithmen laufen aber in linearem Zeit- und Speicherverbrauch ab. Aufgrund seiner on-line Eigenschaft gilt Ukkonens Verfahren in der Fachliteratur oft als natürlichstes und damit auch am leichtesten verständliches Verfahren. Wie in Kapitel 3.2 ausgeführt ist ein Suffixbaum *STree(w)* die kompakte Darstellung seines Suffixtries, was bedeutet, dass alle eindeutigen Pfade aus *STrie(w)* zu komplexen Labels zusammengefasst werden.

Erweiterung des Suffixbaums

Die Knotenmenge V' eines erweiterten Suffixbaumes *STree'(w)* $= (V', E')$ besteht demnach nur noch aus denjenigen Knoten des zugehörigen erweiterten Suffixtries *STrie'(w)*,

(1) die verzweigende Knoten mit zwei oder mehr Übergängen sind oder

(2) Blattknoten darstellen, die keine Übergänge besitzen.

Zur Menge dieser sogenannten *expliziten Knoten* zählen somit auch die Wurzel v_ε und der Hilfsknoten $v_\perp$. Alle anderen Knoten in *STrie'(w)*, die genau einen Übergang besitzen, werden *implizite Knoten* des Suffixbaums *STree'(w)* genannt.

Um die Suffixbaum-Struktur on-line aufzubauen, muss es trotzdem möglich sein, auch auf die Zeichen innerhalb eines komplexen Labels zuzugreifen. Hierzu wird ein Referenzpaar benutzt, über das sowohl implizite als auch explizite Knoten innerhalb der Struktur angesprochen werden können.

Definition 3.5.3. (Referenzpaar (v_i, α) [75]) Als *Referenzpaar* oder *reference pair* eines impliziten und damit nicht verzweigenden Knotens v_{STrie} aus *STree'(w)* oder eines expliziten Knotens v_{STree} in *STree'(w)* wird ein Paar (v_i, α) aus einem (expliziten) Knoten v_i aus *STree'(w)* und einem echten Infix α von w, bezeichnet, wobei v_i

einen Vorgänger von v_{STrie} oder v_{STree} darstellt und α sich aus der Konkatenation der Labels auf dem Pfad von v_i nach v_{STrie} beziehungsweise v_{STree} ergibt. Ist dabei α minimal und somit v_i der nächstliegende Vorgänger von v_{STrie} oder $v_i = v_{STree}$, wird (v_i, α) als *kanonisches Referenzpaar* bezeichnet, wobei für einen Knoten v_{STree} damit (v_{STree}, ε) das kanonische Referenzpaar darstellt.

Als eine effiziente Darstellung einer Zeichenkette $\alpha \in \Sigma^*$ kann ein Pointerpaar (k,p) benutzt werden, wobei k auf den Anfang und p auf das Ende der Zeichenkette zeigt. Dieses Implementierungsdetail spielt eine wichtige Rolle für die Linearität des Algorithmus, da durch die Benutzung von Pointern ein konstanter Anteil an Speicherplatz ($O(1)$) zur Speicherung der Labels gebraucht wird. Die expliziten Knoten von *STree'(w)* entsprechen den im vorherigen Überblick der Struktur definierten Knoten bezüglich der Äquivalenzrelation $\sim_w^L$ (siehe Definition 3.2.2). Eine Kante $e \in E^{v\rightarrow}$ in *STree'(w)* zwischen zwei (expliziten) Knoten v_i und v_j wird nun mithilfe eines Pointerpaares (k, p) als $e = (v_i, (k, p), v_j)$ dargestellt.

Zusätzlich wird sich neben dem Hilfsknoten $v_\perp$ (siehe Definition 3.5.2) der Suffixfunktion $sl(v_i)$ (siehe Definition 3.5.1), die aus der Konstruktion des Suffixtries bekannt ist, bedient. Suffixlinks sind wie in *STrie'(w)* Abbildungen zweier expliziter Knoten $v_j \rightarrow v_i$, wobei $v_j = \sigma\alpha v_i$ ($\sigma \in \Sigma, \alpha \in \Sigma^*$) gelten muss.

Konstruktion des Suffixbaums

Beim on-line Aufbau von *STrie'(w)* wurden ebenfalls zwei Arten von Knoten erzeugt, Blattknoten und Knoten, die den Beginn eines neuen Astes markieren. Die Darstellung der Labels als Pointerpaar erlaubt es nun, alle neuen Blätter des Suffixbaumes, die beim Einlesen eines neuen Zeichens hinzukommen, effizienter einzufügen, als dies bei der Suffixtrie-Konstruktion möglich war. Dafür werden sozusagen alle Suffixe von w, die durch einen Blattknoten dargestellt werden, „automatisch“ aktualisiert, indem die Endpointer der Blattknoten immer auf die Position des zuletzt eingelesenen Zeichens von w gesetzt werden. Da diese zum Ende der Prozedur bei $|w|$ ankommen, ist es nicht wichtig, alle p-Pointer der Blätter explizit hochzuzählen, sondern sie werden abschließend nur einmal auf $|w|$ gesetzt. Damit ist es nicht mehr nötig, explizite Blattknoten zu unterstützen, sondern es werden sogenannte *offene Übergänge* benutzt, die einen Knoten v_i auf einen imaginären Blattknoten zeigen lassen. Somit ergibt sich folgende Definition:

Definition 3.5.4. (Offener Übergang) [75]) Jede Kante $e \in E^{v\rightarrow}$ in *STree'(w)* $= (V', E')$ ausgehend von einem Knoten v_i, die zu einem Blatt führt, heißt *offener Übergang* oder *open transition* und wird als $e = (v_i, (k, \infty), v_{\infty_j})$ repräsentiert.

Es ist für diese Kanten also weder notwendig, den rechten Pointer des Labels darzustellen noch den Zielknoten v_{∞_j}, ein imaginäres Blatt, auf den die Kante führt.

Das Symbol ∞ zeigt an, dass das Label der Kante weiter wachsen kann. Um nun die Blätter final zu aktualisieren, müssen am Ende des Aufbaus von *STree'(w)* $= (V, E)$ nur alle ∞ Symbole durch $|w|$ ersetzt werden. Die nachfolgende Abbildung zeigt die Verwendung offener Kanten anhand der ersten drei Konstruktionsschritte von *STree'(cbca)*[7] :

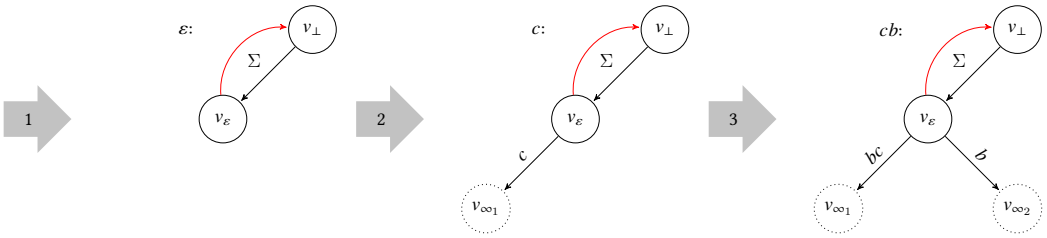


Abbildung 3.14.: Offene Kanten bei der Konstruktion von *STree'(cbca)*.

Implizite Suffixlinks

Die Einfügeposition eines Knotens, der kein Blatt in *STrie'(w)* ist und somit einen neuen Ast starten lässt, wird durch die Suffixlink-Kette entdeckt, wobei dieser neue Knoten immer Suffix des zuletzt hinzugefügten Blattes ist. Somit beschreiben die Suffixlinks in *STrie'(w)* ein Strickleitermuster (mit Ausnahme der Suffixlinks, die auf v_ε führen), welches das Springen von einem Ast des Baumes auf den nächsten ermöglicht. Durch die explizite Darstellung aller Suffixe beziehungsweise Suffixlinks werden so zu jedem neu hinzugefügten Blattknoten all jene Stellen identifiziert, die noch einen neuen Knoten und einen neuen Übergang benötigen. Da in *STree'(w)* nicht alle Suffixe explizit dargestellt werden, kann diese Relation auch zwischen zwei impliziten Knoten bestehen. Ist dies der Fall, muss ein korrespondierender impliziter Knoten gefunden werden, ohne dass dieser direkt durch einen expliziten Suffixlink erreicht werden kann.

Zur Lösung dieses Problems müsste es also zusätzlich Suffixlinks zwischen impliziten Knoten geben. Dies wird mit Hilfe des oben eingeführten Referenzpaares simuliert. Um die Stelle, an der die neuen Suffixe eingefügt werden, zu finden, wird ein kanonisches Referenzpaar benutzt, das als *aktiver Punkt* oder *active point* bezeichnet wird. Durch die Darstellung als kanonisches Referenzpaar kann der aktive Punkt entweder einen expliziten oder einen impliziten Knoten repräsentieren. Zu Beginn befindet sich der aktive Punkt auf dem Wurzelknoten und lautet damit $(v_\varepsilon, \varepsilon)$, wobei anstelle des leeren Wortes das Referenzpaar in Pointerschreibweise $(v_\varepsilon, (p+1, p))$ dargestellt wird. Solange sich der aktive Punkt auf einen expliziten

[7] Zur leichteren Lesbarkeit werden in den folgenden Konstruktionsabbildungen anstelle der Pointerschreibweise die zugehörigen Infixe als Labels angegeben.

Knoten bezieht, werden neue Suffixe durch das Anfügen eines neuen Blattes an diesen Knoten in *STree'(w)* eingefügt. Im Beispiel ist dies in den Schritten 2 und 3, die die Suffixe c, b und cb anfügen, zu erkennen. Dies ändert sich, wenn ein Zeichen eingelesen wird, das bereits bekannt ist. Im Beispiel ist das dritte Zeichen c, das bereits zuvor in Schritt 2 gelesen wurde, das erste wiederholte Zeichen von w. Demnach wird der aktive Punkt nun von v_ε um eine Position nach unten auf der Kante verschoben, die mit c beginnt. Dies geschieht durch Erhöhen des vorderen Pointers. Solange die im Folgenden eingelesenen Zeichen ebenfalls auf dieser Kante zu finden sind, würde der aktive Punkt jeweils um eine Position auf dieser Kante verschoben werden. Im Beispiel folgt nun aber ein a. Da a nicht auf der Kante zu finden ist, sondern dort als nächstes ein b steht, muss an dieser Stelle ein neuer expliziter Knoten für c entstehen, da dieses Zeichen damit in zwei unmittelbaren rechten Kontexten (b und a) aufgetreten ist und damit eine Äquivalenzklasse der Relation $\sim_w^L$ darstellt. Die bestehende Kante $(v_\varepsilon, cbc, v_{\infty_1})$ wird damit aufgetrennt und ein neuer Knoten v_1 mit einer Kante (v_ε, c, v_1) eingefügt. Die beiden Blätter werden wieder durch offene Kanten (v_1, bc, v_{∞_2}) und (v_1, a, v_{∞_3}) dargestellt. Der neue Knoten erhält nun einen Suffixlink $sl(v_1) = v_\varepsilon$, da der referenzierte Knoten des aktiven Punktes v_ε ist und damit ein Suffix von v_1 repräsentiert.

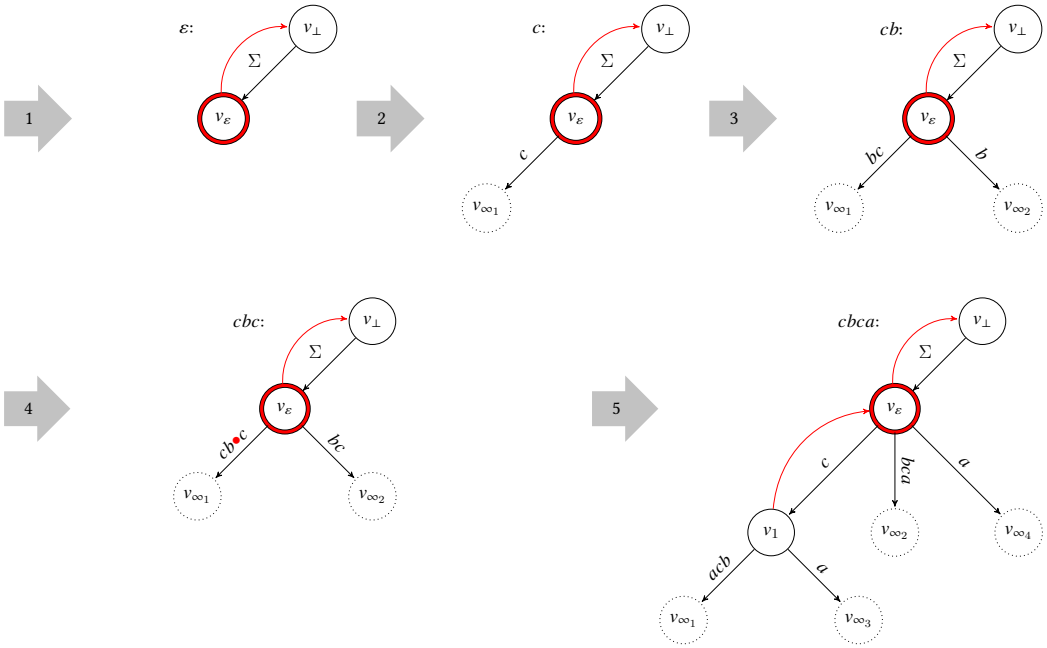


Abbildung 3.15.: Schrittweise Konstruktion von *STree'(w)* für $w = cbca$.

Um nun der Suffixlink-Kette aus *STrie'(w)* entsprechend alle anderen Stellen zu finden, an denen möglicherweise noch weitere Suffixe eingefügt werden müssten,

wird der Suffixlink des durch den aktiven Punkt referenzierten Knotens verfolgt und vom so erreichten Knoten das Label, das zum neuen Knoten führt (Siehe Abbildung 3.15, Schritt 5), gelesen. Da $v_\perp$ mit jedem Buchstaben einen Übergang besitzt, führt dies nun erneut zu v_ε. Von dort aus wird nun letztlich ein neues Blatt eingefügt mit $(v_\varepsilon, a, v_{\infty_4})$. Abbildung 3.15 zeigt noch einmal alle Schritte für *STree'(cbca)*, wobei, wenn der aktive Punkt auf einem expliziten Knoten liegt, dieser rot umrandet ist. Liegt der aktive Punkt auf einem impliziten Knoten, wird dies durch einen zusätzlichen roten Punkt (•) auf der jeweiligen Kante angezeigt. Das Prinzip der impliziten Suffixlinks soll nun noch einmal anhand eines erweiterten Beispiels wiederholt werden. Wenn zusätzlich noch die Zeichen bcd hinzugefügt werden, sodass $w = cbcabcd$, wird zunächst wieder der aktive Punkt von der Wurzel aus um zwei Zeichen nach unten verschoben, da beide Zeichen b und c auf der bc-Kante, die vom aktiven Punkt ausgeht, gelesen werden können.

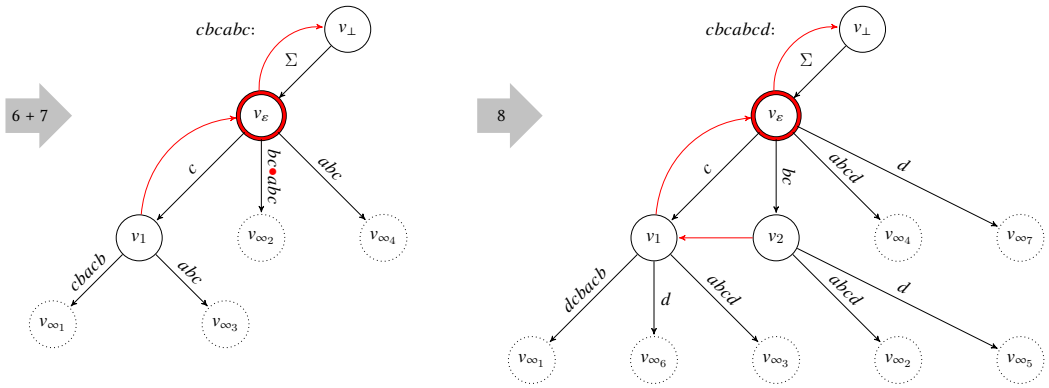


Abbildung 3.16.: Konstruktionsschritte *STree'(cbcabc)* ⟶ *STree'(cbcabcd)*.

Das nächste Zeichen d stellt nun wieder einen neuen unmittelbaren Rechtskontext für bc dar. Somit kann der aktive Punkt nicht weiter verschoben werden und die Kante muss erneut aufgetrennt werden. Durch die Trennung entsteht wieder ein neuer Knoten v_2 mit jeweils zwei Blättern. Der Knoten des aktiven Punktes ist immer noch v_ε. Die erneute Verfolgung des Suffixlinks führt wieder auf $v_\perp$. Von dort wird wieder das Wort des Labels, das zum neuen Knoten führt, gelesen und der aktive Punkt auf den expliziten oder impliziten Knoten, der erreicht wird, gesetzt. Diese Kante (oder der Teil der alten Kante, bis zu dem der aktive Punkt maximal verschoben werden konnte) stellt somit immer das längste wiederholte Suffix des bisher verarbeiteten Präfixes von w dar. Im Beispiel lautet dieses nun bc. Wenn es von $v_\perp$ aus gelesen wird, führt dies auf v_1. Da v_1 zwar explizit ist, aber noch keine Kante mit d hat, wird wieder ein neues Blatt erzeugt und ein Suffixlink $sl(v_2) = v1$ gelegt. Die Länge der gelesenen Kante bc beträgt zwei. Das bedeutet, dass auch

zwei Suffixe zu *STree'(w)* hinzugefügt werden müssen. Somit fehlt noch ein Suffix. Der aktive Punkt steht nun auf v_1, dessen Suffixlink auf v_ε führt. Dort existiert wiederum keine Kante mit d, weswegen wieder ein weiteres Blatt mit $(v_\varepsilon, d, v_\infty)$ in *STree'(w)* eingefügt wird. Das Auslesen der Kante oberhalb des aktiven Punktes wird als *Kanonisierungsschritt* oder *walkdown* bezeichnet. Wie am Beispiel zu sehen ist, wird mit dieser Technik die Suffixlinkkette aus *STrie'(w)* simuliert und mit absteigender Länge werden die Stellen gefunden, an welchen die neuen Suffixe hinzugefügt werden müssen. Die so erzeugten Zustände, die immer dann entstehen, wenn der aktive Punkt inmitten einer Kante stehen bleibt, entsprechen denen von $\sim_w^L$. Maximal müssen in der Traversierung $|w|$ Suffixe hinzugefügt werden, was zur Folge hat, dass Ukkonens Algorithmus *STree'(w)* in $O(n)$ Zeit aufbaut. Formale Beschreibungen des Algorithmus und Pseudocodedarstellungen können etwa in [75, 41, 31] oder [18] nachgeschlagen werden.

3.5.3. Algorithmus von Blumer et al. - DAWG

Beim on-line Aufbau einer DAWG-Struktur *DAWG(w)* müssen im Vergleich mit einem Suffixbaum keine komplexen Labels unterstützt werden. Dafür müssen aber zu jedem Zeitpunkt die Knoten des DAWGs die Äquivalenzklassen von $\sim_w^R$ abbilden, was wiederum bedeutet, dass alle Infixe mit derselben Endpositionsmenge stets in *DAWG(w)* zu Knoten zusammengefasst werden. Ein erster on-line Algorithmus wurde von Blumer et al. [9] im Jahre 1985 vorgestellt.

Erweiterung des DAWG

Um dort einen effizienten on-line Aufbau zu ermöglichen, werden dem DAWG inital wieder zusätzliche Informationen beigefügt. Neben der aus den vorherigen Algorithmen bekannten Suffixfunktion $sl(v)$ werden die Übergänge zwischen den Knoten, die die Äquivalenzklassen von $\sim_w^R$ darstellen, in primäre und sekundäre Kanten unterteilt.

Definition 3.5.5 (Primäre und sekundäre Kanten)**.** Seien $x, y \in \mathit{Infix}(w)$. Eine Kante (v_i, σ, v_j) mit dem Label $\sigma \in \Sigma$ zwischen dem Repräsentanten beziehungsweise Knoten $v_i = \mathit{Rep}([x]_w^R)$ und dem Knoten $v_j = \mathit{Rep}([y]_w^R)$ wird dabei als *primär* bezeichnet, wenn $\mathit{Rep}([y]_w^R) = \mathit{Rep}([x]_w^R)\sigma$ gilt, ansonsten heißt sie *sekundär*.

Konstruktion des DAWG

Der Ablauf des Algorithmus' wird im Folgenden wieder für das Wort $w = cbcabcd$ dargestellt. Initial wird ein Wurzelknoten v_ε erstellt, der das leere Wort repräsen-

tiert. Gleichzeitig stellt dieser Knoten auch den momentanen Sinkknoten dar. Mit dem Einlesen des ersten Buchstabens c wird einer neuer Knoten v_1 erstellt, der nun das Teilwort c repräsentiert und zum neuen Sink wird. Eine mit c gelabelte primäre Kante führt nun von v_ε nach v_1, da $c = \varepsilon c$ gilt. Zusätzlich wird nun ein Suffixlink $sl(v_1) = v_\varepsilon$ gesetzt, da gilt $v_1 = cv_\varepsilon$. Beim Einlesen des nächsten Buchstabens b kommen ein neuer Sinkknoten v_2, der das Teilwort cb repräsentiert, eine mit b gelabelte primäre Kante von v_1 nach v_2 und eine mit b gelabelte sekundäre Kante von v_ε nach v_2 hinzu. Letztere entsteht über das Ablaufen der Suffixkette von v_1 aus, wodurch der Knoten v_ε gefunden wird, der seinerseits noch keine ausgehende Kante mit dem Label b besitzt. Da die Bedingung für eine primäre Kante hier nicht gegeben ist, wird eine sekundäre Kante erzeugt. Abschließend wird der Suffixlink $sl(v_2) = v_\varepsilon$ gesetzt, da $v_2 = bv_\varepsilon$ gilt, da v_2 alle Suffixe der Äquivalenzklasse $[cb]^R_w$ und somit auch das Teilwort b beinhaltet. Wird nun erneut ein c gelesen, wird wiederum ein neuer Sink v_3 eingeführt, der mit einer primären Kante mit v_2 verbunden ist. Der Suffixlink von v_2 führt erneut auf v_ε. Nun kann aber keine neue Sekundärkante eingefügt werden, da es bereits eine primäre Kante (v_ε,c,v_1) gibt. v_1 stellt das längste wiederholte Suffix des bisher gelesenen Präfixes von w und somit auch von v_3 dar, weswegen ein Suffixlink $sl(v_3) = v_1$ entstehen muss.

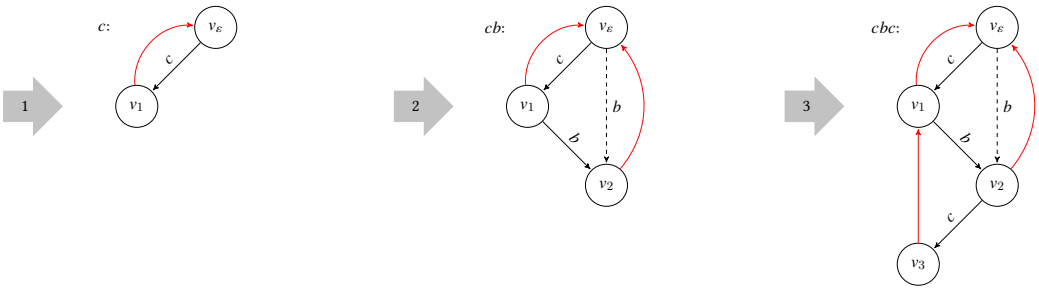


Abbildung 3.17.: Konstruktionsschritte für *DAWG'(cbc)*.

Aufsplitten von Äquivalenzklassen

Im vierten Schritt wird zum ersten Mal ein a eingelesen. Das hat zur Folge, dass wieder ein neuer Sink (v_4) entsteht und durch das Ablaufen der Suffixlink-Kette von v_3 aus zwei sekundäre Übergänge (v_1, a, v_4) und (v_ε, a, v_4) hinzukommen. Am Ende von Schritt 4 wird wiederum der Suffixlink von v_4 gesetzt. Dieser zeigt auf v_ε, da bei der Traversierung der Suffixkette kein Knoten gefunden werden kann, der einen Übergang mit a besitzt. Wird als nächstes allerdings ein b gelesen, tritt dieses Zeichen nun in einem neuen unmittelbaren linken Kontext auf. Das führt in Schritt 5 nun neben dem obligatorischen Anhängen des neuen Sinkknotens v_5 zur Entstehung eines zweiten Knotens $v_6 = Rep([b]^R_w)$, da b nun nicht mehr in

der gleichen Äquivalenzklasse wie cb bleiben kann. Zum Auffinden der korrekten Stelle, an der v_6 eingefügt wird, wird erneut die Suffixlink-Kette von v_4 aus verfolgt. Dies führt zurück auf den Wurzelknoten, von wo aus mit b ein sekundärer Übergang nach v_2 führt. Dieser Übergang wird nun in einen primären Übergang umgewandelt, der auf v_6 zeigt. v_6 erhält zudem alle ausgehenden primären und sekundären Übergänge von v_2 als sekundäre Übergänge und den Suffixlink von v_2. Im Beispiel erhält v_6 dadurch nur eine sekundäre Kante (v_6, c, v_3) und einen Suffixpointer $sl(v_6) = v_\varepsilon$. Der Suffixlink von v_2 muss zudem auf v_6 umgeleitet werden, da nun gilt $v_2 = cv_6$. v_6 stellt nun das längste wiederholte Suffix des bisher gelesenen Präfixes dar. Somit wird noch ein Suffixpointer $sl(v5) = v_6$ gesetzt, der den aktuellen Sink mit v_6 verbindet. Im vorletzten Schritt folgt wiederum der Buchstabe c. Dieser bewirkt erneut ein Aufspalten einer Äquivalenzklasse, da nun bc nicht mehr mit cbc in einer Klasse sein darf. Der DAWG-Struktur werden wiederum ein neuer Sink (v_7) und ein weiterer Knoten $v_8 = Rep([bc]_w^R)$ beigefügt. Die Einfügeposition von v_8 wird wiederum durch die Verfolgung des Suffixlinks von v_5 (dem zuletzt eingefügten Sink) und der Identifikation einer sekundären Kante ausgehend von dessen Zielknoten v_6 bestimmt.

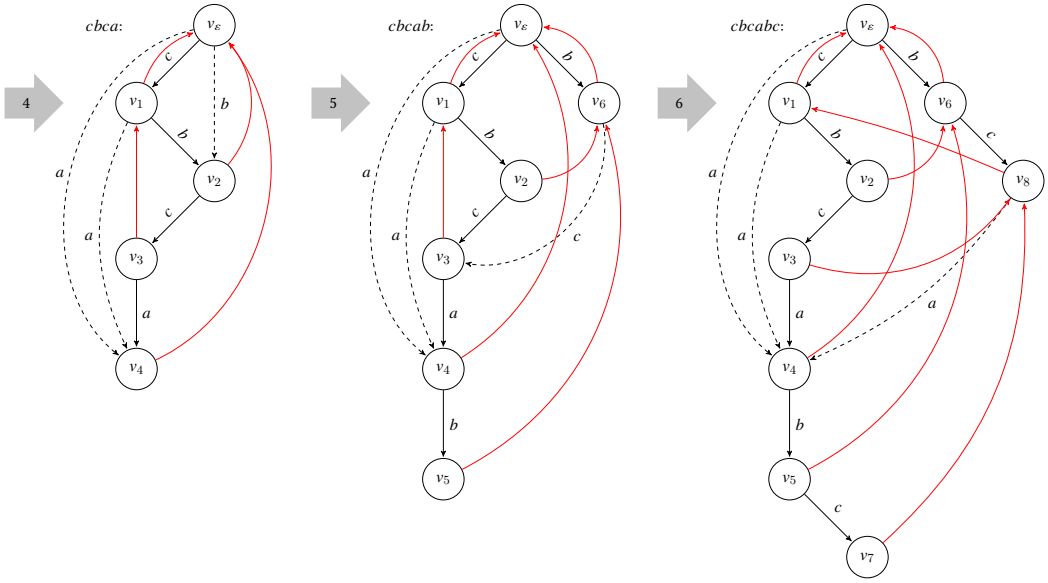


Abbildung 3.18.: Konstruktionsschritte *DAWG'(cbca)* ⟶ *DAWG'(cbcabc)*.

Analog zum vorherigen Schritt bekommt v_8 Kopien der Kanten von v_3 und dessen Suffixpointer, wobei der Suffixpointer von v_3 auf v_8 umgeleitet wird. Durch die Suffixkette wird als nächstes nun von v_6 aus die Wurzel getroffen. Dieser Knoten besitzt eine sekundäre Kante (v_ε, a, v_4), aber keine, die auf v_3 führt, den ursprüng-

lichen Knoten, der bc und cbc durch $[cbc]_w^R$ repräsentierte und deshalb gesplittet werden musste. Damit muss nichts weiter modifiziert werden und Schritt 6 ist vollendet. Befänden sich auf der Suffixkette sekundäre Kanten, die auf einen aufgetrennten Knoten zeigten, müssten diese immer auf den neuen Knoten, der das längste wiederholte Suffix darstellt, umgeleitet werden. Der finale Schritt bedingt kein weiteres Splitten einer Äquivalenzklasse. Durch das Lesen des letzten Zeichens d müssen jedoch drei sekundäre Kanten, deren Startknoten alle auf der Suffixlink-Kette von v_7 gefunden werden, generiert werden, wobei jede davon auf den finalen Sinkknoten v_9 zeigt. In der nachfolgenden Abbildung ist der so resultierende DAWG des Wortes $cbcabcd$ abgebildet.

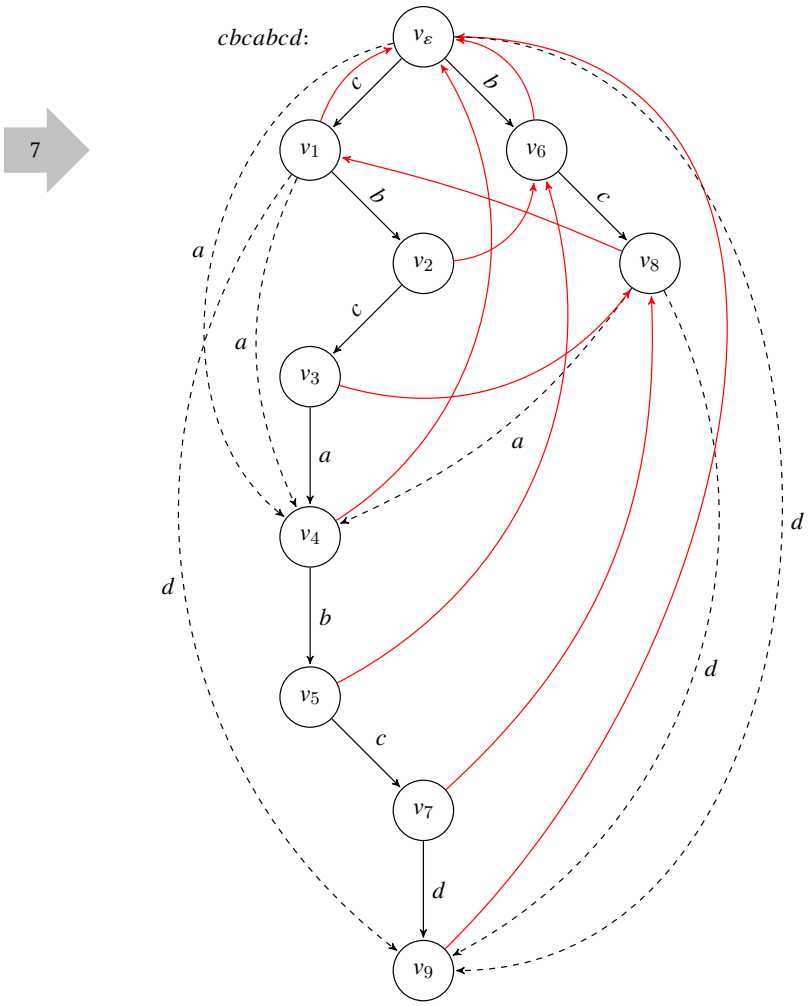


Abbildung 3.19.: *DAWG'(w)* für $w = cbcabcd$ mit Suffixlinks (rot) .

3.5.4. Algorithmus von Inenaga et al. - CDAWG

In linearer Zeit ablaufende Algorithmen zur Konstruktion eines Compact Directed Acyclic Wordgraphs (CDAWG) wurden bereits von Blumer et al. [8] und Crochemore und Vérin [20] beschrieben. Beide Algorithmen stellen aber keine online Verfahren dar. Im ersten Falle wird zuerst eine DAWG-Struktur erstellt, die anschließend durch Zusammenfassung aller nichtverzweigenden Pfade zu komplexen Labels zu einer kompaktierten Version des initial erstellten DAWGs transfomiert wird. Der Algorithmus von Crochemore und Vérin konstruiert direkt eine CDAWG-Struktur zu einem gegebenen Wort w und verbraucht damit weniger Speicher als die erste Variante. Das Verfahren basiert aber auf McCreights Algorithmus zur Konstruktion eines Suffixbaumes [50], weswegen es ebenfalls einen off-line Algorithmus darstellt, dem w vollständig bekannt sein muss, beziehungsweise kein weiteres Zeichen σ hinzugefügt werden kann, ohne dass $CDAWG(w\sigma)$ von neuem berechnet werden muss. Im Folgenden wird nun ein on-line Verfahren zur Konstruktion eines CDAWGs erläutert, das von Inenaga et al. [41] im Jahre 2001 vorgestellt wurde. Dieses Verfahren kann als eine Weiterentwicklung der Algorithmen von Blumer et al., Crochemore und Vérin gesehen werden. Es basiert auf Ukkonens Konstruktion der Suffixbäume (siehe Kapitel 3.5.2) und stellt damit ein on-line Verfahren dar. Alle drei CDAWG-Algorithmen basieren zudem maßgeblich auf der on-line Konstruktion des DAWGs (siehe Kapitel 3.5.3), die ebenfalls von Blumer et al. beschrieben wurde. Durch die Kombination beider Verfahren (online Aufbau von Suffixbaum und DAWG) unterstützen die CDAWG-Algorithmen die Äquivalenzrelation $\sim_w$, die sich aus den Relationen $\sim_w^L$ (Suffixbaum) und $\sim_w^R$ (DAWG) zusammensetzt (siehe Lemma 3.4.1).

Erweiterung des CDAWG

Da Ukkonens Suffixbaum-Konstruktion das Fundament des Algorithmus von Inenaga et al. bildet, erhält der CDAWG neben der Suffixfunktion $sl(v)$ und dem Hilfsknoten $v_\perp$ einen expliziten Sinkknoten v_w, der alle offenen Kanten, die in einem entsprechenden Suffixbaum entstehen, auf sich vereint. Damit werden alle Kanten der Form (v_i, α, v_w) wie beim Suffixbaum mit dem Einlesen eines neuen Zeichens automatisch um dieses erweitert. Gleichermaßen enthält ein CDAWG wie auch ein Suffixbaum komplexe Labels. Damit besteht auch bei diesem Verfahren die Notwendigkeit, Suffixlinks zwischen impliziten Knoten zu unterstützen, da nicht jedes Suffix explizit dargestellt wird. Aus diesem Grund wird ebenfalls wieder ein kanonisches Referenzpaar benutzt, das den aktiven Punkt darstellt und sich entweder auf einen impliziten oder expliziten Knoten bezieht.

Konstruktion des CDAWG

Erneut wird als Eingabewort zunächst $w = cbca$ betrachtet. In den ersten beiden Konstruktionsschritten von *CDAWG'(w)* werden $v_\perp$ und v_ε mit einer Kante, die alle Alphabetbuchstaben liest, und einem Suffixlink $sl(v_\varepsilon) = v_\perp$ verbunden und eine neue Kante mit (v_ε, c, v_w) sowie ein rückläufiger Suffixlink $sl(v_w) = v_\varepsilon$ angelegt. Der aktive Punkt befindet sich anfangs wiederum auf dem Wurzelknoten v_ε. Die beiden nächsten Zeichen b und c bewirken, dass eine weitere Kante von der Wurzel zum Sinkknoten mit dem Label b erzeugt wird und der aktive Punkt um eine Position auf der Kante, die mit c beginnend auf v_w zeigt, nach unten versetzt wird. Entsteht nun für c durch das nächste Zeichen a wieder ein neuer unmittelbarer rechter Kontext muss die c-Kante an der Stelle des aktiven Punktes aufgetrennt werden und ein neuer expliziter Knoten v_1 eingefügt werden. Anstelle eines neuen Blattes führen nun beide ausgehenden Kanten von v_1 auf v_w. Über den Suffixlink der Wurzel wird nun wieder die Stelle gefunden, an der das nächstkürzere Suffix a eingefügt wird. Diese befindet sich auf dem Wurzelknoten selbst, und somit wird dort eine neue Kante $(v_\varepsilon$,a, $v_w)$ hinzugefügt, da es noch keine a-Kante von der Wurzel aus gab. Schließlich erhält v_1 noch einen Suffixlink auf v_w. An dieser Stelle muss beachtet werden, dass in dem Fall, in dem der aktive Punkt weiter nach unten gewandert wäre, etwa bei dem Eingabewort $cbcba$, die b-Kante ebenfalls aufgetrennt werden müsste und diese nun auch auf v_1 zeigen müsste, da dann cb und b unter $\sim_w$ zu einer Äquivalenzklasse gehören würden.

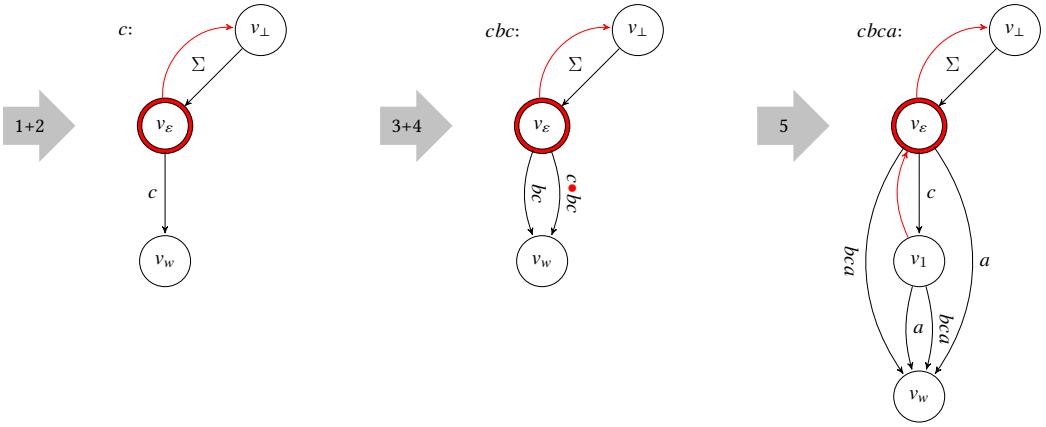


Abbildung 3.20.: Konstruktionsschritte für *CDAWG'(cbca)*.

Wird nun aber w um das Suffix $bcdcd$ erweitert, sodass $w = cbcabcdcd$, wird der aktive Punkt für die nächsten Zeichen b und c (Schritte 6 und 7) auf der b-Kante $(v_\varepsilon, bcabc, v_w)$ um zwei Positionen nach unten verschoben. Der erneute neue rechte Kontext d (Schritt 8) führt nun erst zu einer Trennung dieser Kante und einem

neuen Knoten v_2. c und bc befinden sich nun nicht innerhalb einer Äquivalenzklasse, da c alleine in bereits in einem neuen rechten Kontext (a) aufgetreten war. Somit existieren nun also zwei explizite Knoten v_1 und v_2, die c und bc repräsentieren. Jedoch ist c aber ein Suffix von bc, weshalb noch ein Suffixlink $sl(v_2) = v_1$ gebraucht wird. Da der Knoten des aktiven Punktes immer noch v_ε ist, wird nun dessen Suffixlink verfolgt und durch die Kante $(v_\perp, \Sigma, v_w)$ ein b gelesen, sodass dort nun die c-Kante betrachtet werden muss. Der anschließende walkdown führt nun auf v_1, der die Rolle des neuen aktiven Knotens übernimmt und eine Kante mit (v_1, d, v_w) erhält und den fehlenden Suffixlink $sl(v_2) = v_1$ hinzufügt. Abermals wird nun der Suffixlink des aktiven Knotens verfolgt und dieser auf v_w gesetzt. Von dort wird noch einmal eine d-Kante (v_ε, d, v_w) erzeugt. Die beiden nachfolgenden Zeichen (c und d, Schritte 9+10) bewirken wiederum eine Verschiebung des aktiven Punktes auf v_1, wobei dieser am Ende auf der d-Kante dieses Knotens steht und als längstes wiederholtes Suffix somit cd repräsentiert.

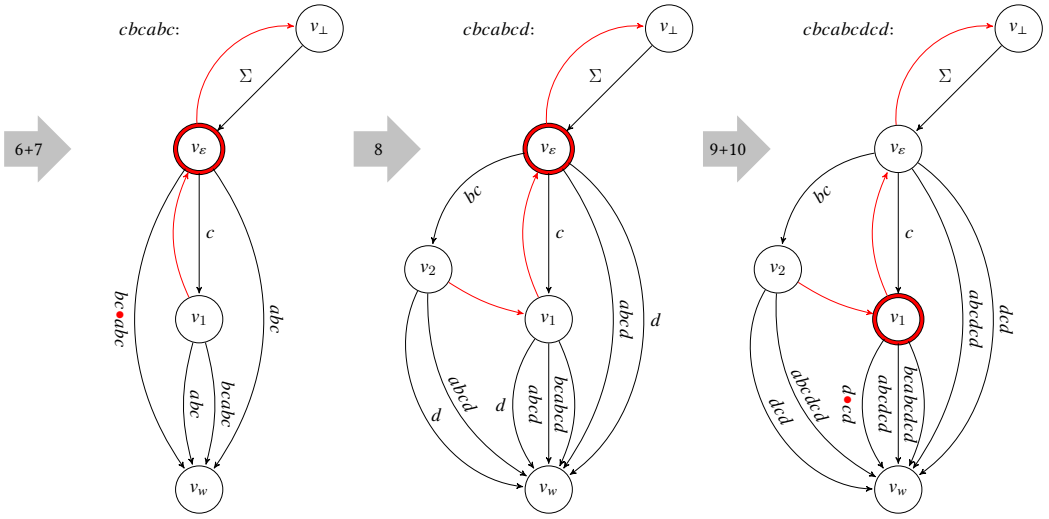


Abbildung 3.21.: Schritte für *CDAWG'(cbcabc)* → *CDAWG'(cbcabcdcd)*.

Umleitungen und neue linke Kontexte

Wird nun ein letztes Mal w um zwei Zeichen e und d erweitert, ergibt sich folgendes Bild. Zunächst bewirkt das Zeichen e (Schritt 11), dass die Kante (v_1, dcd, v_w) getrennt wird, da der aktive Punkt nicht weiter nach unten wandern kann. Der neue Knoten v_3 wird mit v_1 über d verbunden und erhält die obligatorischen beiden Kanten auf v_w. Mittels des Suffixlinks von v_1 wird nun als nächstes die d-Kante (v_ε, d, v_w) identifiziert. Diese existiert schon und führt auf v_w. Da nun aber mit v_3

ein d-Knoten entstanden ist, muss diese Kante zusätzlich auf v_3 umgeleitet werden, denn cd und d befinden sich innerhalb einer Äquivalenzklasse. Abschließend wird noch der Suffixlink von v_3 auf $sl(v_3) = v_\varepsilon$ gesetzt. Das letzte Zeichen ist nun erneut ein d (Schritt 12). Der unmittelbare linke Kontext des letztes d's ist das zuvor eingelesene e. Damit befinden sich nun cd und d nicht mehr in einer Äquivalenzklasse, was bedeutet, dass ein neuer Knoten v_4 für d entstehen muss. Dieser Split entspricht nun demjenigen aus der DAWG-Konstruktion. Der neue Knoten v_4 kopiert nun alle ausgehenden Kanten von v_3, da vom aktiven Punkt aus (v_ε) die Kante mit d auf v_3 zeigt. Neben den ausgehenden Kanten kopiert v_4 auch den Suffixlink von v_3 und der Suffixlink von v_3 wird auf v_4 umgeleitet, da d ein Suffix von cd ist. Nun muss noch die ausgehende d-Kante des aktiven Punktes auf v_4 umgeleitet werden. Am Ende des Schrittes wird der aktive Punkt auf v_4 gesetzt (da ein Knoten, der durch einen neuen linken Kontext entsteht, ein Suffix von w ist) und der Suffixlink von v_w auf v_4 umgeleitet.

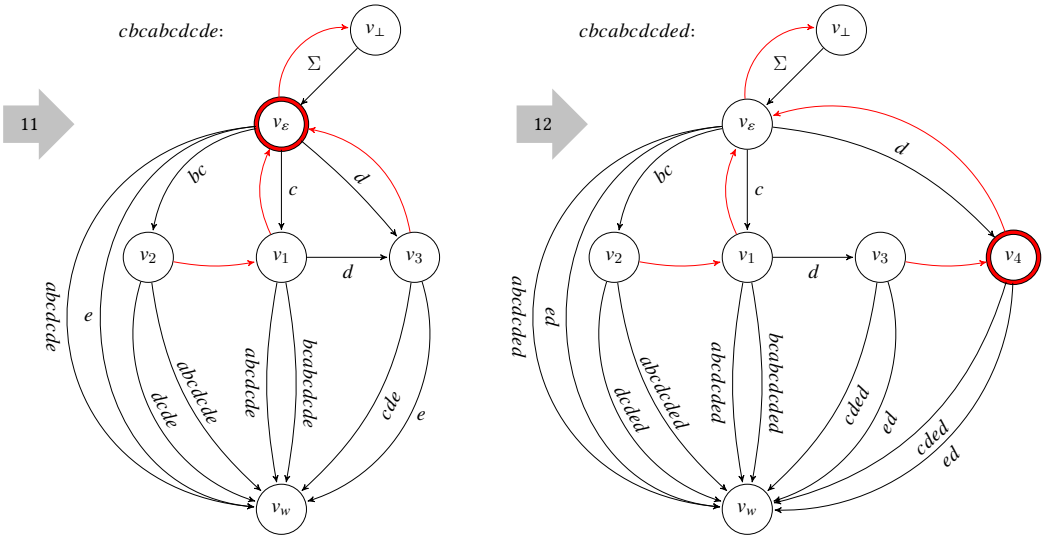


Abbildung 3.22.: Schritte für *CDAWG'(cbcabcdcd)* → *CDAWG'(cbcabcdcded)*.

3.6. Symmetric Compact Directed Acyclic Word Graph (SCDAWG)

Die bisher behandelten Indexstrukturen stellen anhand ihrer zugrundeliegenden Äquivalenzrelation die Menge *Infix(w)* eines Wortes w auf verschiedene Arten dar. Wenn zugleich aber auch die Infixe des umgekehrten Wortes w^{rev} unterstützt werden, wird von einer *symmetrischen* Indexstruktur gesprochen. Blumer et al. [8] zeigten, dass die Knotenmenge einer CDAWG-Struktur *CDAWG(w)*, die sich aus $\sim_w$ ableitet, der von *CDAWG(w^{rev})* entspricht und damit die Äquivalenzklasseneinteilung von $\sim_w$ von der Leserichtung von w unabhängig ist. Daraufhin konnte bewiesen werden, dass die symmetrische Variante eines CDAWG, die *Infix(w)* und *Infix(w^{rev})* unterstützt, in linearer Zeit aufgebaut werden kann. Betrachtet man erneut das obige Beispielwort $w = aabbccd$ sowie dessen Umkehrung $w^{rev} = dccbbaa$, so entstehen identische Äquivalenzklasseneinteilungen.

w: $[a]_w = \{a\}$, $[b]_w = \{b\}$, $[c]_w = \{c\}$, $[\varepsilon]_w = \{\varepsilon\}$,
$[aabbccd]_w = \{aabbccd, aabbc, abbcc, abbccd, aabb, abbc, bbcc, bbccd, aab, abb, bbc, bcc, bccd, aa, ab, bb, bc, cc\}$.

w^{rev}: $[a]_{w^{rev}} = \{a\}$, $[b]_{w^{rev}} = \{b\}$, $[c]_{w^{rev}} = \{c\}$, $[\varepsilon]_{w^{rev}} = \{\varepsilon\}$,
$[dccbbaa]_{w^{rev}} = \{dccbbaa, cbbaa, ccbba, dccbba, bbaa, cbba, ccbb, dccbb, baa, bba, cbb, ccb, dccb, aa, ba, bb, cb, cc\}$.

Beispiel 3.6.1. Für $w = aabbccd$, und $w^{rev} = dccbbaa$ entstehen folgende CDAWGs:

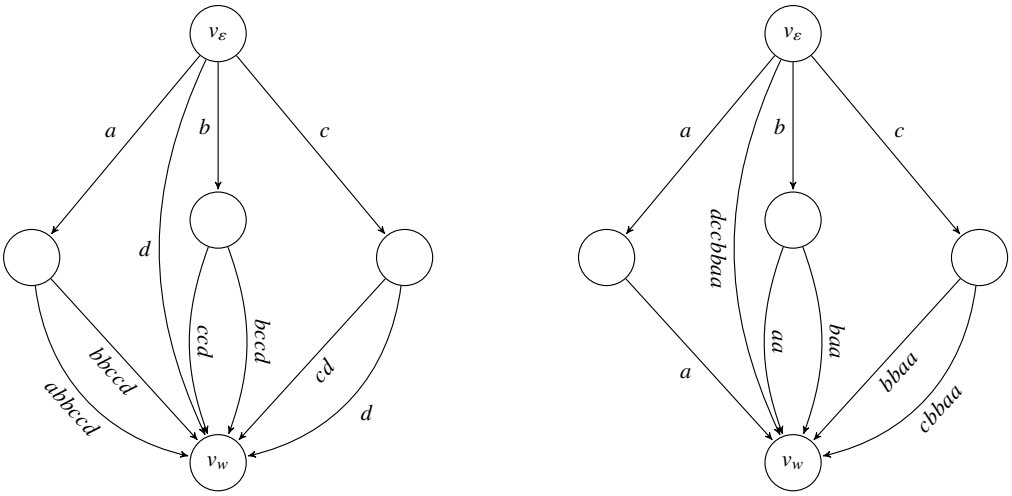


Abbildung 3.23.: $CDAWG(aabbccd)$ und $CDAWG(dccbbaa)$.

Es ist offensichtlich, dass $CDAWG(aabbccd)$ und $CDAWG(dccbbaa)$ zwar die gleiche Knoteneinteilung bilden, sie sich jedoch nicht exakt symmetrisch in Bezug auf ihre Kanten verhalten. Durch die Umkehrung von w endet w^{rev} nicht mehr mit einem eindeutigen Symbol, das w^{rev} beendet und zuvor noch an keiner Position aufgetreten war. In $CDAWG(aabbccd)$ dagegen endet jede Kante, die zum Sinkknoten v_w führt, mit dem eindeutigen Symbol d. In der on-line Konstruktion der Suffixbäume und (S)CDAWGs spielt ein solches Endsymbol eine wichtige Rolle, da es auch dazu führt, dass das $w_{|w|-1}$-te Zeichen von w immer von einem neuen unmittelbaren rechten Kontext gefolgt wird, was einen neuen expliziten Knoten erzeugt und damit verhindert, dass bestimmte Elemente aus *Infix(w)* nicht in der Struktur repräsentiert sind. Für die im vorherigen Kapitel vorgestellte on-line Konstruktion des CDAWGs würde dies für $w = dccbbaa$ zutreffen, sodass hier ein Endmarker $\gamma \notin \Sigma$ an w angehängt werden müsste, damit für das Suffix a ein expliziter Knoten entstünde.

Vereinigt man nun $CDAWG(aabbccd)$ und $CDAWG(dccbbaa)$, ergibt sich ein *symmetric compact directed acyclic word graph (SCDAWG)*, der sowohl die Infixe für w von links nach rechts als auch von rechts nach links gelesen abbildet und durch entsprechende Kanten verbindet. Die Kanten, die die Lesereihenfolge von w^{rev} darstellen, werden blau gezeichnet, die von w schwarz.

Beispiel 3.6.2. Sei $w = aabbccd$, so ergibt sich folgender SCDAWG:

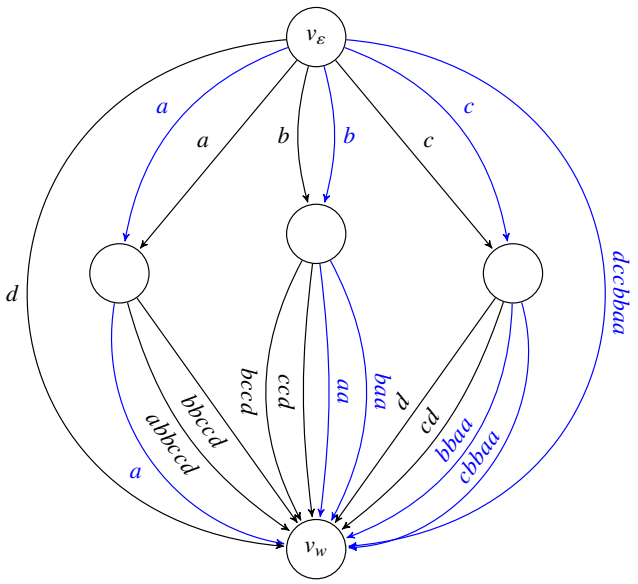


Abbildung 3.24.: SCDAWG des Eingabewortes $aabbccd$.

Die vormals gegebene Definition des CDAWGs (siehe Definition 3.4.1) wird, um eine SCDAWG-Struktur formal anzugeben, erweitert. Dabei liegt anstelle des Paares (V, E) eines gerichteten azyklischen Graphen ein Tripel (V, E_R, E_L) zugrunde, wobei E_R die Menge der rechten Übergänge und E_L die Menge der linken Übergänge beschreibt.

Definition 3.6.1 (SCDAWG, vgl. [41])**.** Ein SCDAWG *SCDAWG(w)* über einer Zeichenkette w ist ein gerichteter azyklischer Graph (V, E_R, E_L) für den gilt:

1. $V = \{Rep([x]_w) \mid x \in \mathit{Infix(w)}\}$.
2. $E_R = \left\{ (Rep([x]_w), \sigma\beta, Rep(Rep([x]_w)\sigma) \;\middle|\; \begin{array}{l} \sigma \in \Sigma, \alpha, \beta \in \Sigma^* \text{ und } x \in \mathit{Infix(w)}, \\ Rep(Rep([x]_w)\sigma) = \alpha Rep([x]_w)\sigma\beta, \\ Rep([x]_w) \neq Rep(Rep([x]_w)\sigma) \end{array} \right\}$.
3. $E_L = \left\{ (Rep([x]_w), \delta\gamma, Rep(\gamma Rep([x]_w)) \;\middle|\; \begin{array}{l} \gamma \in \Sigma, \delta, \omega \in \Sigma^* \text{ und } x \in \mathit{Infix(w)}, \\ Rep(\gamma Rep([x]_w)) = \delta\gamma Rep([x]_w)\omega, \\ Rep([x]_w) \neq Rep(\gamma Rep([x]_w)) \end{array} \right\}$.

Für einen SCDAWG gelten zusätzlich folgende Eigenschaften:

> Für alle $x \in \mathit{Infix(w)}$ gilt: Jeder Knoten $v \in V$ entspricht dem Repräsentanten $Rep([x]_w) = Rep([x]_{w^{rev}})$ einer Äquivalenzklasse aus $\sim_w$ und ihrer entsprechenden Äquivalenzklasse $\sim_{w^{rev}}$.
>
> Seien $v_k, v_l, v_m \in V$ und $\sigma, \gamma \in \Sigma$ und $\alpha, \beta \in \Sigma^*$ sowie $E_R^{v_k\rightarrow} = \{e_1, \ldots, e_n\}$ mit $n \geq 1$ die Menge der ausgehenden rechten Kanten von v_k, dann muss für $e_i = (v_k, \sigma\alpha, v_l)$ und $e_j = (v_k, \gamma\beta, v_m)$ $\sigma \neq \gamma$ gelten. Für $E_L^{v_k\rightarrow} = \{f_1, \ldots, f_n\}$, der Menge der ausgehenden linken Kanten von v_k, gilt für $f_i = (v_k, \sigma\alpha, v_l)$ und $f_j = (v_k, \gamma\beta, v_m)$ ebenfalls $\sigma \neq \gamma$.
>
> Es existiert ein Knoten $v_w \in V$, der Sink genannt wird. Für jedes Suffix w' aus w existiert ein Pfad rechter Kanten $(v_\varepsilon, \ldots, v_w)$, wobei die Konkatenation der Labels dieses Pfads w' ergibt. Ebenso existiert für jedes Suffix $w^{rev'}$ aus w^{rev} ein Pfad linker Kanten $(v_\varepsilon, \ldots, v_w)$, wobei die Konkatenation der Labels dieses Pfads $w^{rev'}$ ergibt.

3.6.1. Algorithmus von Inenaga et al. - SCDAWG

Die ursprüngliche SCDAWG-Konstruktion, die von Blumer et al. [8] gegeben wurde, basiert auf der Beobachtung, dass die linken Übergänge E_L eines *SCDAWG(w)* den umgekehrten Suffixlinks des zugehörigen *DAWG(w)* entsprechen. Wie bei der *CDAWG(w)*-Konstruktion von Blumer et al. muss allerdings zuerst *DAWG(w)* konstruiert werden und dieser anschließend kompaktiert werden, damit *SCDAWG(w)*

entsteht. Damit läuft dieses Verfahren zwar in linearer Zeit, es stellt aber wiederum einen off-line Algorithmus dar und verbraucht durch die initiale Konstruktion des DAWGs viel Speicherplatz. Inenaga et al. veröffentlichten in [42] einen Algorithmus, der gleichzeitig zur on-line Konstruktion eines CDAWGs [41] (siehe Abschnitt 3.5.4) alle Übergänge aus E_L mit aufbaut und damit einen SCDAWG on-line in linearer Zeit erstellt. Weitere Relationen zwischen den String-Indexstrukturen und deren Suffixlink-Strukturen können in [41] und [29] gefunden werden.

Konstruktion des SCDAWGs

Der simultane Aufbau der linken und rechten Übergänge einer SCDAWG-Struktur wird erneut exemplarisch anhand des vorherigen Konstruktionsbeispiels betrachtet. Zur on-line Konstruktion des SCDAWGs werden außer denjenigen, die zur on-line Konstruktion des CDAWGs benötigt wurden, und der in Definition 3.6.1 gegebenen Unterscheidung der Übergänge in die beiden Mengen E_R (rechte Übergänge) und E_L (linke Übergänge) keine weiteren Strukturen benötigt. Initial wird erneut $w = cbca$ als Eingabewort gelesen.

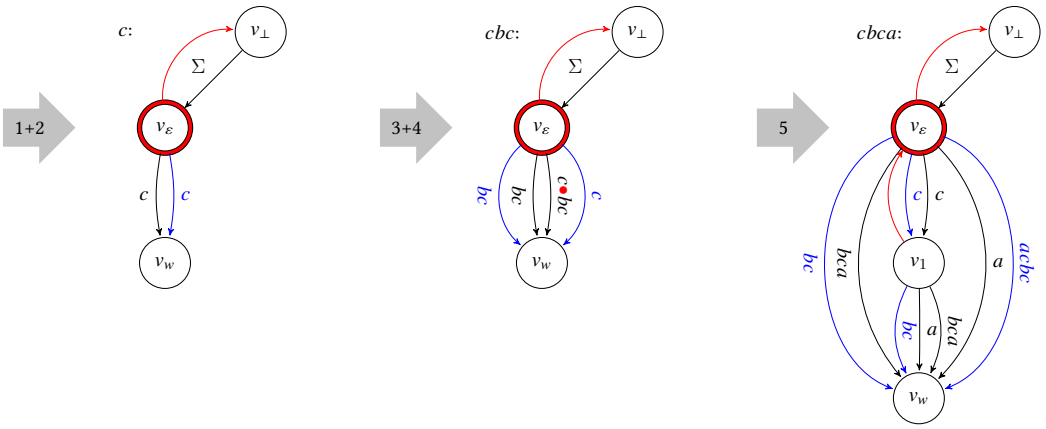


Abbildung 3.25.: Konstruktionsschritte für *SCDAWG'(cbca)*.

In den ersten Schritten (1-4) der Konstruktion entstehen für jedes neue Zeichen, das zuvor noch nicht in w aufgetreten war, linke Übergänge von v_ε zum Sinkknoten v_w. Die Endpointer der Labels dieser linken Kanten stehen jeweils auf 1 und enden bei der aktuell eingelesenen Position von w, wodurch sich in w^{rev} das rückwärts gelesene Infix aus w, das über diese Kante dargestellt wird, ablesen lässt. Wann immer also beim Einlesen von w_i eine neue rechte Kante vom aktiven Knoten (hier v_ε) zum Sinkknoten v_w angelegt wird, wird gleichzeitig die entsprechende linke Kante mit $(v_\varepsilon, (i, 1), v_w)$ für w^{rev} mit erzeugt. Ein automatisches Update des

Endpointers einer linken Kante ist damit nicht nötig, da w nicht nach links, sondern nur nach rechts erweitert werden kann.

Auftrennen rechter und linker Kanten

In den Schritten 3 und 4 wird der aktive Punkt verschoben, was auf die Erzeugung linker Kanten keinen direkten Einfluss hat. Nach Schritt 4 beschreibt *SCDAWG'(cbc)* keinen vollständigen Suffixautomaten, da ein eindeutiges Abschlusssymbol fehlt, durch dessen neuen unmittelbaren rechten Kontext implizite Knoten zu expliziten werden. Dieses wird wie zuvor mit dem Zeichen a in Schritt 5 gelesen und bewirkt damit eine Trennung der rechten Kante $(v_\varepsilon, cbca, v_w)$ und einen neuen Knoten v_1. Analog zur rechten aufgetrennten Kante muss nun der linke Übergang (v_ε, c, v_w) getrennt werden. Dabei wird die Kante auf den neuen Knoten $v1$ umgeleitet, wodurch die linke Kante (v_ε, c, v_1) entsteht. Der vordere Teil der aufgetrennten linken Kante entspricht dabei gleichzeitig dem umgekehrten Suffixlink des neuen Knotens. Zudem entsteht nun noch eine zweite linke Kante, die mit bc von v_1 nach v_w zeigt. Anschließend wandert der aktive Punkt auf die Wurzel, wodurch letztlich noch das Kantenpaar (v_ε, a, v_w) und $(v_\varepsilon, acbc, v_w)$ des aktuell gelesenen Buchstabens ($w_i = a$) erstellt wird.

Wird w nun wieder auf $cbcabcdcd$ gesetzt, entsteht ein zweiter Knoten v_2, der das Infix bc repräsentiert. Dies bewirkt, dass die linke Kante (v_ε, bc, v_w) aufgetrennt werden muss. Da v_1 ein Suffix von v_2 ist, wird die linke Kante (v_1, bc, v_w) nun ebenfalls aufgetrennt und damit auf v_2 umgeleitet. Wie im vorherigen Schritt 5 werden nun wieder zwei neue linke Kanten (v_2, c, v_w) und $(v_2, acbc, v_w)$ von v_2 aus angelegt, die die noch fehlenden Infixe c und $acbc$ aus w^{rev} darstellen. Am Ende von Schritt 8 werden noch die linke und rechte Kante für d vom Wurzelknoten aus zum Sinkknoten angelegt. In den Schritten 9 und 10 springt der aktive Punkt nach unten, sodass der aktive Knoten am Ende von Schritt 10 auf v_1 sitzt und der aktive Punkt auf der davon ausgehenden dcd-Kante. Beim Aufsplitten einer rechten Kante wird somit die entsprechende linke Kante ebenfalls mit aufgetrennt und auf den neu entstandenen Knoten umgelenkt beziehungsweise der übrige Teil durch eine neue linke Kante vom neuen Knoten auf den alten Zielknoten dargestellt. Der Startpointer der linken Kante, die von einem Splitknoten v_r ausgeht, liegt bei der Position des Zeichens links vom ersten Vorkommen von v_r, der Endpointer bei 1. Wie bei der Trennung einer rechten Kanten wird dann noch eine zweite linke Kante vom Splitknoten v_r zum Sinkknoten erzeugt, deren Endpointer bei 1 und deren Startpointer bei $i - |v_r| - 1$ steht, und damit das fehlende Infix für w^{rev} ergänzt. Der neue Splitknoten kann in bestimmten Fällen jedoch ein Präfix des Zielknotens der getrennten linken Kante sein. Wenn dies der Fall ist, kopiert der neue Knoten alle linken Kanten des Zielknotens und fügt gegebenenfalls noch eine neue linke

Kante, die bei $i - |v_r| - 1$ startet, hinzu, falls diese nach dem Kopieren noch nicht existiert.

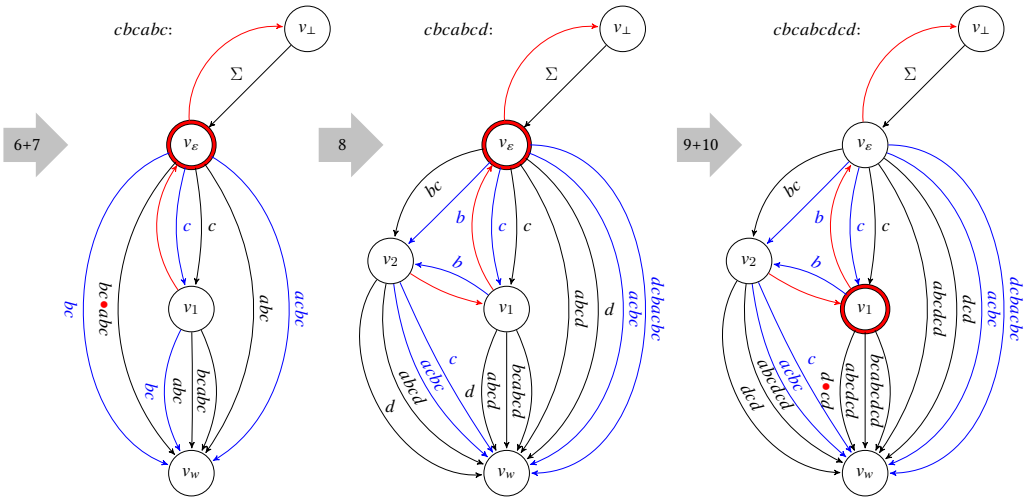


Abbildung 3.26.: Schritte für *SCDAWG'(cbcabc)* → *SCDAWG'(cbcabcdcd)*.

Für $w = cbcabcdcde$ (Schritt 11) wird mit v_3 ein neuer Knoten erstellt und dementsprechend die linke Kante $(v_\varepsilon, dcbacbc, v_w)$ aufgeteilt. Da der aktive Punkt nun aber nicht auf v_ε steht, erhält v_3 nur eine eingehende linke Kante (v_ε, dc, v_3), die wiederum seinem umgekehrten Suffixlink entspricht. Eine linke Kante von v_1 nach v_3 darf nicht entstehen, da d immer rechts von c steht. Stände der aktive Punkt wie in den ersten beiden Splits auf der Wurzel v_ε, würde noch eine zweite linke Kante gebraucht werden. Da die geteilte rechte Kante erneut kein Präfix darstellt, erhält v_3 zwei ausgehende linke Kanten. Wenn nun der aktive Punkt durch Verfolgen der Suffixlink-Kette nach v_ε wieder auf v_1 steht, erhält v_1 noch eine fehlende linke Kante $(v_1, dcbacbc, v_w)$, obwohl hier bereits eine rechte Kante, die mit d startet, nach v_3 existiert. Zum Schluss erfolgt diese Prüfung noch einmal von der Wurzel aus, dort werden wieder eine rechte und eine linke Kante mit e zum Sinkknoten angelegt.

Abspaltung eines Knotens

Im letzten Schritt entsteht durch den neuen Linkskontext von d ein separater Knoten für dieses Zeichen. Dafür wird die linke dc-Kante (v_ε, dc, v_3) aufgetrennt, sodass diese nun über v_4 auf v_3 führt. Abschließend muss von v_4 nur eine weitere linke Kante mit e zum Sinkknoten führen, da e den neuen linken Kontext darstellt, der zur Abspaltung des d-Knotens v_4 von v_3 (cd) geführt hat. Weitere Präfixe in

w, die mit d enden, enden gleichzeitig mit cd und sind deshalb von v_3 ausgehend zu finden.

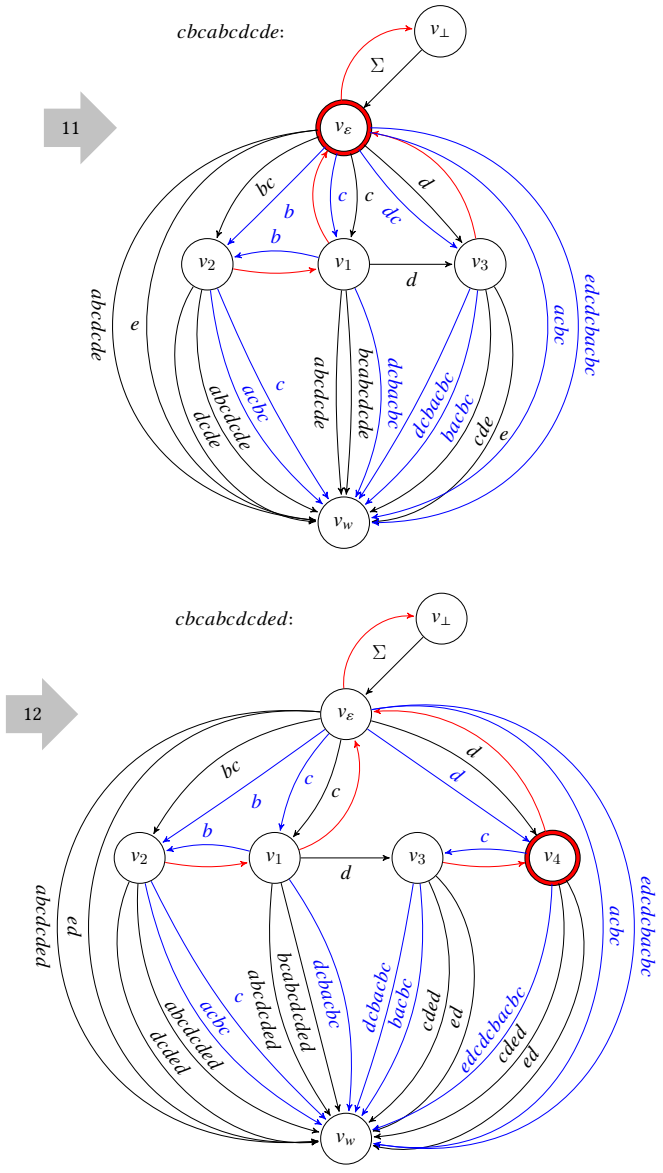


Abbildung 3.27.: Schritte für *SCDAWG'(cbcabcdcde)* → *SCDAWG'(cbcabcdcded)*.

3.6.2. Generalisierung

Die Konstruktion eines DAWGs beziehungsweise CDAWGs für eine Menge von Wörtern wurde erstmals von Blumer et al. beschrieben [8]. Suffixbäume, die eine Menge von Eingabewörtern abbilden, werden oft als generalisierte Suffixbäume bezeichnet [31]. Ursprünglich wurde das Konzept eines Suffixbaums für eine Menge von Wörtern W von Kosaraju [45] beschrieben [42]. Der Aufbau einer (S)CDAWG-Struktur für eine Menge von Eingabewörtern $W = \{w^1, w^2, \ldots, w^n\}$ gestaltet sich einfach, wenn die auf Ukkonens Suffixbaum-Konstruktion basierte on-line Konstruktion von Inenaga et al. [42] benutzt wird. Aufgrund der on-line Eigenschaft des Algorithmus kann ein Wort nach dem anderen verarbeitet werden, ohne dass die lineare Laufzeit beeinträchtigt wird. Wird ein neues Wort eingelesen, entstehen Knoten beziehungsweise linke und rechte Übergänge, die nach den neuen unmittelbaren rechten und linken Kontexten, die dabei auftreten, aktualisiert werden. Dabei entstehen in *(S)CDAWG'(W)* $|W|$ Sinknoten $v_{w^1}, v_{w^2}, \ldots, v_{w^n}$. Gleichzeitig muss darauf geachtet werden, dass kein Wort aus W Präfix eines anderen Wortes aus W ist. Inenaga et al. bezeichnen dies als Präfixeigenschaft und definieren diese wie folgt.

Definition 3.6.2 (Präfixeigenschaft [42])**.** Sei $W = \{w^1, w^2, \ldots, w^n\}$ mit $w^i \in \Sigma^*$ für $1 \leq i \leq n$ und $n \geq 1$. Dann hat W die *Präfixeigenschaft* gdw. $w^i \notin \mathit{Präfix}(w^j)$ für alle $1 \leq i \neq j \leq n$.

Damit für jedes beliebige W die Präfixeigenschaft gilt, kann jedem Element aus W ein Endmarker $\$ \notin \mathit{Infix(W)}$ angefügt werden.

Beispiel 3.6.3. Sei $W = \{abc, ab\}$, so gilt nach Definition 3.6.2 für W die Präfixeigenschaft nicht, da $ab \in \mathit{Präfix(abc)}$. Sei $W' = \{abc\$, ab\$\}$ so gilt diese für W', da $ab\$ \notin \mathit{Präfix(abc\$)}$.

Somit folgt schließlich das Theorem:

Theorem 3.6.1. (Inenaga et al. [42]) Sei Σ ein endliches Alphabet. Für eine beliebige Menge W aus Eingabwörtern über Σ kann *CDAWG'(W)* on-line in $O(|W|)$ Zeit- und Speicherbedarf aufgebaut werden.

Da die Präfixeigenschaft in *SCDAWG'(W)* nun auch für alle $w^{i^{rev}}$, $1 \leq i \neq j \leq n$, gelten soll, wird allen Wörtern aus W zusätzlich ein Startmarker $\# \notin \mathit{Infix}(W^{rev})$ vorangestellt. Durch das Hinzufügen von Start- und Endmarkern wird also sichergestellt, dass alle Präfixe von W und W^{rev} in *SCDAWG'(W)* repräsentiert sind.

Die nachfolgende Abbildung zeigt eine SCDAWG-Struktur der beiden Wörter $w^1 = \#ababc\$$ und $w^2 = \#abcab\$$, die auch in [8] als Beispielwörter dienen, ohne Suffixlinks und den Hilfsknoten $v_\perp$.

Beispiel 3.6.4. Sei $W = \{\#ababc\$, \#abcab\$\}$, so ergibt sich folgender SCDAWG:

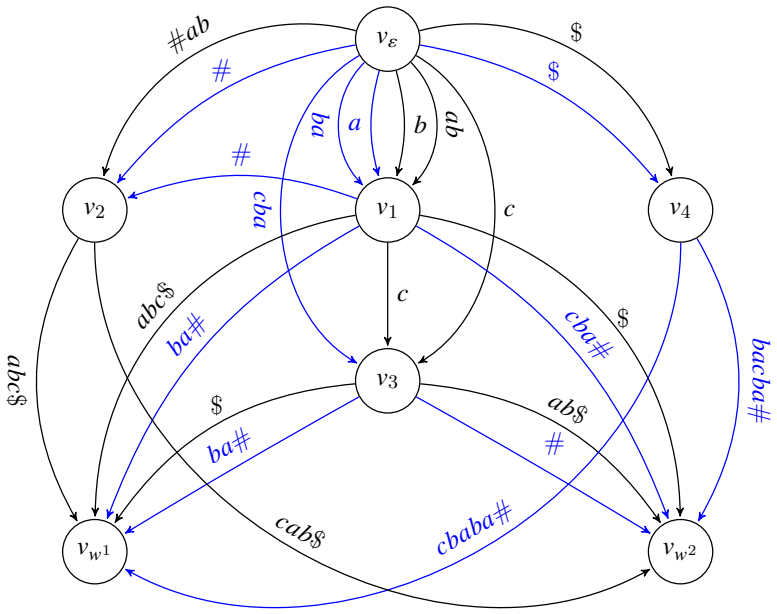


Abbildung 3.28.: *SCDAWG'({#ababc$, #abcab$}).*

Für $w^1 = \#abcbc\$$ entsteht neben Wurzel- und Sinkknoten nur ein weiterer Knoten v_1, der das Infix ab repräsentiert. Wird nun $w^2 = \#abcab\$$ eingelesen, wird zuerst der neue Sinkknoten v_{w^2} hinzugefügt. Anschließend entstehen nacheinander die Knoten v_2, v_3 und v_4 für $\#ab$, abc und $\$$. Die dabei entstehenden rechten und linken Kanten führen, wenn sie ausschließlich Infixe aus w^1 betreffen, auf v_{w^1}, wenn sie ausschließlich Infixe aus w^2 darstellen, auf v_{w^2}. Alle anderen linken und rechten Kanten betreffen Infixe aus beiden Wörtern und führen somit auf die inneren Knoten der Struktur. Der letzte Knoten v_4 wird wie im letzten Schritt des vorigen Beispiels durch einen neuen unmittelbaren Linkskontext (b) ausgelöst. Hierdurch wird die linke Kante $\$cbaba\#$ aufgetrennt und über den neuen Knoten v_4 umgeleitet. Von v_4 starten, da dieser Knoten den Endmarker repräsentiert, nur noch linke Übergänge auf beide Sinkknoten. Im Gegensatz dazu besitzt v_2 nur

rechte Übergänge auf beide Sinkknoten, da dieser Knoten das Präfix $\#ab$, das den Startmarker beinhaltet, darstellt.

4. Zusammenfassung Teil I.

Die in Kapitel 2 beschriebenen Methoden aus dem Bereich des exakten und inexakten String-Matchings geben einen ersten Einblick in die Probleme, die sich dabei gerade für große Datenbestände in Bezug auf Zeit- und Speicherverbrauch ergeben können. Zwar existieren effiziente und elegante Algorithmen wie der KMP-Algorithmus, die lineare Komplexität besitzen, doch auch diese Verfahren sind für größere Textsammlungen nicht optimal, da, wenn häufig nach verschiedenen Mustern gesucht werden soll, für jedes Pattern die gesamten Berechnungen neu durchgeführt werden müssen. Im nächsten Kapitel werden Indexstrukturen besprochen, mit deren Hilfe nach einmaliger Berechnung beliebige Exakt-Matching-Anfragen effizienter beantwortet werden können, ohne dass für jede Anfrage eine Neuberechnung nötig wird. Für komplexe Relationen zwischen zwei Texten, wie die Bestimmung des Editierabstands oder die globale Alignierung, existieren optimale Lösungsverfahren (siehe Abbildung 2.3), deren Laufzeiten aber oft unbefriedigend sind, wenn etwa viele Textpaare auf einmal verarbeitet werden müssen. Auch hier kann es hilfreich sein, zuerst eine String-Indexstruktur vorzuberechnen, die die Identifikation bestehender Gemeinsamkeiten innerhalb einer Textsammlung erleichtert. Weitere Beschreibungen grundlegender String-Verarbeitungsprobleme und deren Lösungsansätze sind in [31, 19, 18] oder [53] zu finden.

Die formale Definition von Suffixtrie, Suffixbaum, DAWG und CDAWG in Kapitel 3 sowie die Beschreibung der von Blumer et. al [9, 8], Ukkonen [75] und Inenaga et al. [42, 41] erdachten on-line Algorithmen zur Konstruktion dieser String-Indexstrukturen war nötig, um schließlich die direkte Konstruktion der symmetrischen CDAWG-Struktur einer Menge von Eingabewörtern W, die ebenfalls von Inenaga et al. [41] gegeben wurde, zu erläutern. Das bei der on-line Konstruktion aller Strukturen essentielle Konzept der Suffixlinks beinhaltet noch weitere interessante Relationen zwischen den Strukturen, die ebenfalls in [41] zusammengefasst sind. Dabei wird beispielsweise auf eine weitere Indexstruktur, die von Stoye [72] beschriebenen Affix-Bäume, die eine bidirektionale Suffixbaum-Struktur bilden, und deren lineare Konstruktion durch Maaß [47] verwiesen. In [39] wird zudem ein noch speichereffizienteres Verfahren zur Konstruktion bidirektionaler Suffixbäume eingeführt.

Teil II.

Eigene Implementierung und On-line Suche

5. Implementierung der SCDAWG-Struktur

Gegenstand dieses Kapitels ist die ausführliche Codebeschreibung einer eigenen Implementierung des on-line Algorithmus' von Inenaga et al. zum Aufbau einer SCDAWG-Struktur. Zum besseren Verständnis und zur Vergleichbarkeit wird anschließend eine ebenfalls eigens erdachte off-line Implementierung der SCDAWG-Struktur besprochen.

5.1. Pseudocode-Beschreibung

Da in [41] ebenfalls nur eine Pseudocode-Darstellung zur on-line Konstruktion symmetrischer Suffixbäume gegeben wird und somit nach derzeitigem Kenntnisstand keine Pseudocode-Beschreibung des on-line Algorithmus' von Inenaga et. al [41] zur Erzeugung einer generalisierten SCDAWG-Struktur existiert, wird nun eine selbst entwickelte Implementierung dieses Verfahrens vorgestellt. Die folgende Beschreibung des on-line Algorithmus von Inenaga et al. zur Konstruktion einer symmetrischen CDAWG-Struktur S_C für eine endliche Menge von Eingabewörtern W über einem endlichen Alphabet Σ im Pseudocode basiert auf den Pseudocode-Darstellungen von Inenaga et al. [42], [41] und Ukkonen [75]. Im Kern entspricht das hier beschriebene Programm der on-line Konstruktion der CDAWG-Struktur aus [42]. Damit kann bei der Implementierung des Aufbaus der SCDAWG-Struktur auch schrittweise vorgegangen werden, indem zuerst die on-line Variante zur Erzeugung der CDAWG-Struktur implementiert wird und diese in einem Zwischenschritt durch das in Kapitel 5.2 gezeigte off-line Verfahren zu einer SCDAWG-Struktur erweitert wird. Ein Vorteil dieses Vorgehens ist es, dass mit der off-line Variante eine Vergleichsimplementierung zur Verfügung steht, die zu Testzwecken eingesetzt werden kann, um die hier beschriebene vollständige on-line Variante zum Aufbau der SCDAWG-Struktur zu verifizieren. Zum Testen der dadurch erhaltenen Struktur S_C kann, wenn keine off-line Implementierung vorliegt, auch stets eine CDAWG-Struktur der umgekehrten Eingabewörter erstellt werden und die dabei entstehenden Übergänge können dann mit den linken Übergängen der im on-line SCDAWG-Verfahren erzeugten Übergänge verglichen werden. Beim Ver-

gleich sollte jedoch stets darauf geachtet werden, dass die verwendeten Eingabewörter die bereits erwähnten Start- und Endmarker besitzen. Die folgende Codebeschreibung zeigt eine Prozedur *build_scdawg*, die für eine Wortmenge W und ein endliches Alphabet Σ den Aufbau der SCDAWG-Struktur übernimmt.

Algorithmus 3 Konstruktion einer SCDAWG-Struktur $S_C = (V, E_R, E_L)$

Vorbedingung: Eine endliche Menge von Zeichenketten $W = \{w^1, \ldots, w^n\}$ mit $|W| > 0$ über einem endlichen Alphabet Σ.

```
1:  procedure build_scdawg(W, Σ)
2:      V ← V ∪ {v_ε, v_⊥}                          ▷ +v Wurzel- und Hilfsknoten
3:      |v_ε| ← 0 , |v_⊥| ← −1
4:      sl(v_ε) ← v_⊥                               ▷ SL→ von v_ε → v_⊥
5:      for i ← 1 to n do
6:          V ← V ∪ {v_{w^i}}                       ▷ +v Neuer Sinkknoten v_{w^i}
7:          (v_s, k) ← (v_ε, 1)
8:          for j ← 1 to |w^i| do
9:              E_R ← E_R ∪* {(v_⊥, (−j, −j), v_ε, i)}   ▷ R→ von v_⊥ → v_ε
10:             (v_s, k) ← update(v_s, (k, j), i)        ▷ Aufruf der Hauptfunktion
11:         end for
12:         set_suffixlink(v_{w^i}, v_s, k, i)   ▷ Suffixlink und linke Kante zwischen s und v_{w^i}
13:     end for
14: end procedure
```

Initial werden Wurzelknoten v_ε und $v_\perp$ der Knotenmenge V hinzugefügt (Zeile 2) und deren Länge mit 0 beziehungsweise -1 besetzt. Mit Hilfe der Suffixlink-Funktion *sl* wird ein Suffixpointer in umgekehrter Richtung erzeugt. Für jedes Wort aus W erhält S_C einen Sinkknoten v_{w^i} (Zeile 6) und der aktive Knoten (v_s) sowie der Startpointer des kanonisches Referenzpaars werden initialisiert. Anschließend wird für jedes Zeichen w^i_j aus w^i eine rechte Kante angelegt, die von $v_\perp$ nach v_ϵ führt, und die Hauptroutine $update(v_s, (k, j), i)$ aufgerufen. Das Vereinigungszeichen $\cup^*$ gibt an, dass eine bereits existente Kante, die den gleichen Startpointer besitzt, durch diese neue Kante ersetzt wird. Bei der Erstellung jeder neuen Kante wird zusätzlich der Index des momentan bearbeiteten Wortes w^i in dieser vermerkt. Der Wortindex i wird somit auch an die Hauptroutine *update* übergeben.[1]

[1] Einige Kommentare sind durch die Symbole $\overset{+}{v}$ für das Hinzufügen eines neuen Knotens sowie $\overset{R}{\rightarrow}$ für das Anlegen einer neuen rechten Kante beziehungsweise $\overset{L}{\leftarrow}$ für das Erzeugen einer neuen linken Kante und $\overset{SL}{\rightarrow}$ für das Anlegen eines Suffixlinks gekennzeichnet.

Algorithmus 4 Hauptfunktion der SCDAWG-Konstruktion

1: **function** $update(v_s, (k, p), i)$
2: $\quad \sigma \leftarrow w_p^i$, $v_{oldr} \leftarrow \emptyset, r_{end} \leftarrow \emptyset$
3: $\quad$ **while not** $check\_endpoint(v_s, (k, p-1), \sigma, i)$ **do**
4: $\quad\quad$ **if** $k \leq p-1$ **then** ▷ Impliziter Fall
5: $\quad\quad\quad$ **if** $v_{s'} = extension(v_s, (k, p-1), i)$ **then**
6: $\quad\quad\quad\quad redirect\_edge(v_s, (k, p-1), v_r, i)$
7: $\quad\quad\quad\quad (v_s, k) \leftarrow canonize(sl(v_s), (k, p-1), v_r, r_{end}, i)$
8: $\quad\quad\quad\quad$ **continue**
9: $\quad\quad\quad$ **else**
10: $\quad\quad\quad\quad v_{s'} \leftarrow extension(v_s, (k, p-1), i)$
11: $\quad\quad\quad\quad v_r, r_{end} \leftarrow split\_edge(v_s, (k, p-1), i)$
12: $\quad\quad\quad$ **end if**
13: $\quad\quad$ **else** ▷ Expliziter Fall
14: $\quad\quad\quad v_r \leftarrow v_s$
15: $\quad\quad$ **end if**
16: $\quad\quad E_R \leftarrow E_R \cup^* \{(v_r, (p, |w^i|), v_{w^i}, i)\}$ ▷ $\xrightarrow{R}$ von $v_r \rightarrow v_{w^i}$
17: $\quad\quad$ **if** $v_{oldr} \neq \emptyset$ **then**
18: $\quad\quad\quad set\_suffixlink(v_{oldr}, v_r, k + |v_r| - |v_s|, i)$
19: $\quad\quad$ **end if**
20: $\quad\quad l \leftarrow k - |v_s|$
21: $\quad\quad$ **if** $v_r \neq v_\varepsilon$ **then**
22: $\quad\quad\quad l \leftarrow l - 1$
23: $\quad\quad$ **end if**
24: $\quad\quad$ **if not** $(v_r, w_l^i \alpha, v_{r'}) \in E_L^{v\rightarrow}$ **then** ▷ $\alpha \in \Sigma^*$
25: $\quad\quad\quad E_L \leftarrow E_L \cup^* \{(v_r, (l, 1), v_{w^i}, i)\}$ ▷ $\xleftarrow{L}$ von $v_r \rightarrow v_{w^i}$
26: $\quad\quad$ **end if**
27: $\quad\quad v_{oldr} \leftarrow v_r$
28: $\quad\quad (v_s, k) \leftarrow canonize(sl(v_s), (k, p-1), v_r, r_{end}, i)$
29: $\quad$ **end while**
30: $\quad$ **if** $v_{oldr} \neq \emptyset$ **then**
31: $\quad\quad set\_suffixlink(v_{oldr}, v_s, k, i)$
32: $\quad$ **end if**
33: $\quad l \leftarrow k - |v_s| - 1$
34: $\quad$ **if not** $(v_r, w_l^i \alpha, v_{r'}) \in E_L^{v\rightarrow}$ **then** ▷ $\alpha \in \Sigma^*$
35: $\quad\quad E_L \leftarrow E_L \cup^* \{(v_r, (l, 1), v_{w^i}, i)\}$ ▷ $\xleftarrow{L}$ von $v_r \rightarrow v_{w^i}$
36: $\quad$ **end if**
37: $\quad$ **return** $separate\_node(v_s, (k, p), i)$
38: **end function**

Innerhalb der *while*-Schleife, die durch *check_endpoint*$((v_s, (k, p-1), \sigma, i)$ gesteuert wird, wird überprüft, ob sich der aktive Punkt auf einen expliziten oder impliziten Knoten bezieht. Anfangs stehen p und k beide auf 1, wodurch der Test in Zeile 4 nicht erfüllt ist, damit Zeile 14 ausgeführt und v_r auf v_s gesetzt wird. Da der aktive Punkt damit auf v_ε sitzt, entsteht nun eine rechte Kante zum Sinkknoten v_{w^i} (Algorithmus 4, Zeile 16). Der Endpointer dieser rechten Kante wird auf $|w^i|$ gesetzt[2]. In Zeile 17 wird überprüft, ob nun ein Aufruf der Prozedur *set_suffixlink* $(v_{oldr}, v_r, k + |v_r| - |v_s|, i)$, die nachfolgend aufgeführt ist, stattfindet.

Algorithmus 5 Unterfunktion zum Handling der Suffixpointer

```
1: procedure set_suffixlink(v_s, v_t, k, i)
2:     sl(v_s) ← v_t                          ▷ SL→ von v_s → v_t
3:     E_L ← E_L ∪* {(v_t, (k − |v_t|−1, k − |v_s|), v_s, i)}   ▷ L← von v_t → v_s
4: end procedure
```

Da aber in *update* die Variable v_{oldr} in Zeile 2 mit $\emptyset$ vorbesetzt wurde, wird v_{oldr} erst in Zeile 27 auf v_r beziehungsweise auf v_s gesetzt. Wäre v_{oldr} nicht leer, würde ein neuer Suffixlink von diesem Knoten aus angelegt werden und gleichzeitig eine umgekehrte linke Kante. In den Zeilen 20-23 wird nun der Startpointer l einer neuen linken Kante mit $k - |v_s|$ berechnet und diese in Zeile 25 angelegt. k bezeichnet den Startpointer einer rechten von v_s ausgehenden Kante. Wird von diesem die Länge des Startknotens abgezogen, bezieht sich l auf die Startposition von v_s im Zielknoten, von der aus nach links erweitert werden kann. Wenn v_s nicht auf der Wurzel steht, muss für l noch ein Zeichen mehr nach links gegangen werden. Die in Algorithmus 4 erzeugten linken Kanten führen immer vom aktuellen Knoten v_s zum Sinkknoten v_{w^i}. Sie werden allerdings nur erzeugt, wenn diese Kante nicht durch einen der anderen Schritte, etwa durch *set_suffixlink* oder *canonize* bereits entstanden ist.

In Zeile 28 von *update* wird *canonize* mit dem Zielknoten des Suffixlinks von v_s aufgerufen, der im Anfangsfall auf $v_\perp$ führt. Da der Startpointer nun größer als $p - 1 = 0$ ist, wird kein impliziter Fall erkannt und $v_\perp$ zurückgegeben. Dies bedeutet, dass keine weiteren Suffixe angefügt werden müssen und damit auch kein walkdown zur Identifikation weiterer Einfügestellen nötig ist. Der erneute Aufruf von *check_endpoint*$(v_s, (k, p-1), \sigma, i)$ liefert nun *true*, während er inital

[2] Damit stellt die hier vorgeschlagene Implementierung im strengeren Sinne keine echte on-line Implementierung dar, da die Länge von w^i bekannt ist. Zur Implementierung eines streng im on-line Sinne arbeitenden Algorithmus kann, wie von Inenaga et al. [42] vorgeschlagen, anstelle der Länge jedes Wortes eine Variable genutzt werden, deren Wert sich mit jedem eingelesenen Zeichen erhöht.

false zurückgab, da nun eine neue rechte Kante zum Sinkknoten existiert, deren Label mit dem Zeichen σ beginnt.

Algorithmus 6 Unterfunktion zur Kanonisierung des Referenzpaares

```
1:  function canonize(v, (k, p), v_r, r_end, i)
2:      if k ≤ p then                                          ▷ Impliziter Fall
3:          (v, (k', p'), v') ← (v, w_k^i α, v') ∈ E_R^{v→}     ▷ Finde w_k^i-Kante, α ∈ Σ*
4:          while p' − k' ≤ p − k do
5:              k ← k + p' − k' + 1
6:              v ← v'
7:              if k ≤ p then
8:                  (v, (k', p'), v') ← (v, w_k^i α, v') ∈ E_R^{v→}   ▷ Finde w_k^i-Kante, α ∈ Σ*
9:                  if v_r ≠ ∅ then
10:                     l ← k' − |v|−1
11:                     if v = v_ε then
12:                         l ← k' + 1
13:                     end if                                 ▷ γ ∈ Σ*
14:                     (v, (k'', p''), v'') ← (v, w_l^i γ, v'') ∈ E_L^{v→}   ▷ Finde w_l^i-Kante
15:                     if k'' − p'' + 1 + |v|< |v''| then
16:                         p''' ← r_end − |v_r|+1
17:                         E_L ← E_L ∪* {(v, (k'', p'''), v_r, i)}   ▷ Umleitung ←^L w_l^i-Kante
18:                     end if
19:                 end if
20:             end if
21:         end while
22:     end if
23:     return (v, k)
24: end function
```

In beiden Durchläufen wurde ein expliziter Fall erkannt, da k nie größer oder gleich p war. Existiert aber nun eine ausgehende σ-Kante, wird die Schleife beendet und in *update* noch einmal *set_suffixlink* aufgerufen, was keine Änderung des bisherigen Graphen bewirkt, da ein Suffixlink von v_ε nach $v_\perp$ bereits existiert. Zum Ende hin wird noch einmal $canonize(v, (k.p), \emptyset, \emptyset, i)$ innerhalb von *separate_node* $(v_s, (k, p), i)$ aufgerufen. Dort ist p nun gleich k und führt somit zum impliziten Fall, in dem zunächst in *canonize* in Zeile 3 von v beziehungsweise $v_\perp$ aus die ausgehende Kante gesucht wird, die mit dem w_k^i-ten Zeichen beginnt. Da von $v_\perp$ mit allen Zeichen Kanten der Form $(v_\perp, (-l, -l), v_\varepsilon)$ starten, wird nun k um eins erhöht. Damit ist k' kleiner als p und *separate_node* gibt (s', k') zurück. Die beiden

zusätzlichen Parameter v_r und r_{end} werden in *canonize* zur Umleitung linker Kanten gebraucht, wenn ein neuer Knoten v_r entstanden ist. Da dies am Anfang von *separate_node* nicht der Fall ist, werden diese Parameter dort leer gelassen. Wird nun im nächsten *update*-Schritt ein nächstes noch unbekanntes Zeichen gelesen, wird in der gleichen Abfolge eine neue Kante mit diesem Zeichen zum Sinkknoten erstellt.

Algorithmus 7 Unterfunktion zur Bestimmung des Endpunkts

```
1: function check_endpoint(v, (k, p), σ, i)
2:     if k ≤ p then                                          ▷ Impliziter Fall
3:         (v, (k', p'), v', s_nr) ← (v, w_k^i α, v', s_nr) ∈ E_R^{v→}   ▷ Finde w_k^i-Kante, α ∈ Σ*
4:         return σ = w_{k'+p−k+1}^{s_nr}
5:     else                                                   ▷ Expliziter Fall
6:         return (v, σα, v') ∈ E_R^{v→}                      ▷ Test auf Existenz einer σ-Kante
7:     end if
8: end function
```

Wenn wie in Abbildung 3.25 Schritt 4 (cbc) nun ein bereits bekanntes Zeichen folgt, wird durch *check_endpoint* erkannt, dass bereits eine Kante existiert, die mit σ startet und damit die $while$-Schleife übersprungen. Nach einem zweiten Aufruf von *set_suffixlink* wird im finalen Kanonisierungsschritt nun k nicht erhöht, da die Länge $(p' - k')$ der ausgehenden σ-Kante von v_s beziehungsweise v_ε nicht kleiner oder gleich der der aktiven Kante $(p - k)$ ist. Dies entspricht damit der Versetzung des aktiven Punkts innerhalb einer Kante. Für den nächsten Durchlauf ist nun entscheidend, ob der aktive Punkt noch weiter versetzt werden kann oder ein neuer Knoten entstehen muss, wenn dies nicht möglich ist, und damit das aktuell eingelesene Suffix nicht auf einer bisher existenten Kante gefunden werden kann.

5.1.1. Auftrennen einer Kante

Wird wie im Konstruktionsbeispiel einer SCDAWG-Struktur ein Zeichen (in Abbildung 3.25 Schritt 5 ($w = cbca$)) gelesen, das nun einen neuen unmittelbaren rechten Kontext bezüglich des aktiven Punktes bildet, wird dies durch *check_endpoint* innerhalb des impliziten Falls erkannt. Dafür wird zuerst die ausgehende Kante, die mit dem Zeichen w_k^i startet, gesucht und auf dieser Kante der nächste Buchstabe ($w_{k'+p-k+1}^{s_{nr}}$)[3], der nach dem Teilstück, das durch die aktive Kante dargestellt wird, folgt, mit dem aktuell eingelesenen Zeichen σ verglichen. Im Konstruktionsbeispiel in Schritt 5 ist $w_{k'+p-k+1}^{s_{nr}} = c$ und $\sigma = a$. Damit stellt σ einen neuen

[3] s_{nr} gibt stets den in einer Kante gespeicherten Wortindex an.

unmittelbaren rechten Kontext dar und die *while*-Schleife in *update* wird gestartet. Da k nun kleiner gleich $p-1$ ist, wird dort ein weiterer Test ausgeführt, der in der Subroutine $extension(v_s,(k,p-1),i)$ stattfindet.

Algorithmus 8 Unterfunktion zur Identifikation von v'

```
1: function extension(v, (k, p), i)
2:     if k ≤ p then                                   ▷ Impliziter Fall
3:         (v, (k', p'), v') ← (v, w_k^i α, v') ∈ E_R^{v→}   ▷ Finde w_k^i-Kante, α ∈ Σ*
4:         return v'
5:     else
6:         return v                                    ▷ Expliziter Fall
7:     end if
8: end function
```

Die Funktion *extension* liefert damit, wenn ein impliziter Fall besteht, den Zielknoten v' der w_k^i-Kante, die von v aus startet, zurück, ansonsten v. Im momentanen Szenario ($w^1 = cbca$) würde damit v' zurückgegeben, der Test in Algorithmus 4, Zeile 5 scheitert aber trotzdem, da dort v' noch nicht besetzt wurde. Damit wird *extension* erneut in Zeile 10 ausgeführt, v' auf v_{w^i} gesetzt und ein neuer Knoten durch $split\_edge(v_s(k,p-1),i)$ erzeugt. Würde v' dem Ergebnis von *extension* entsprechen, würde eine Umleitung existierender Kanten auf den neu erzeugten Splitknoten v_r nötig sein, da dieser nun das nächstkürzere Suffix bezüglich dieser Kanten darstellen würde. Zum Auftrennen der w_k^i-Kante wird diese in *split_edge* in Zeile 2 gesucht und ein neuer Knoten v_r erstellt. Die Länge von v_r setzt sich aus der Länge des Startknotens v der aufzutrennenden Kante und der Länge des auf der Kante nach unten gewanderten Teilstücks $p-k$ zusammen. Die bisherige w_k^i-Kante wird dann auf v_r umgeleitet und deren Label um $p-k$ verkürzt. Dementsprechend wird im nächsten Schritt für den zweiten Teil der aufzutrennenden Kante eine neue rechte Kante erzeugt (Algorithmus 9, Zeile 6), deren Startpointer bei $k'+p-k+1$ liegt, während ihr Endpointer auf p' zeigt. Zielknoten der zweiten Kante ist v', der im Beispielszenario nun den aktuellen Sinkknoten v_{w^i} darstellt.

Nach der Aufspaltung der rechten Kante werden die ausgehenden linken Kanten von v_r behandelt. Dafür wird zunächst überprüft, ob v auf die Wurzel zeigt. Alle eingehenden linken Kanten, die auf v_r zeigen, werden an anderer Stelle entweder durch *set_suffixlink* oder *canonize* erzeugt. Nun wird überprüft, ob der neue Knoten v_r ein Präfix von v' ist (Algorithmus 9, Zeile 7). Ist dies nicht gegeben, entsteht für v_r eine neue ausgehende linke Kante, die den neuen rechten Kontext bezüglich $w^{i^{rev}}$ darstellt. Der Startpointer dieser Kante liegt erneut beim Startpointer der rechten w_k^i-Kante abzüglich $(|v|+1)$ (Algorithmus 9, Zeile 8). Der Endpointer zeigt

auf die Startposition von v', die sich durch $p' - |v'|+1$ ablesen lässt.

Algorithmus 9 Unterfunktion zum Auftrennen einer Kante

```
1:  function split_edge(v, (k, p), i)
2:      (v, (k', p'), v') ← (v, w_k^i α, v') ∈ E_R^{v→}        ▷ Finde w_k^i-Kante, α ∈ Σ*
3:      V ← V ∪ {v_r}                                          ▷ v^+ Neuer Splitknoten v_r
4:      |v_r| ← |v| + p − k + 1
5:      E_R ← E_R ∪* {(v, (k', k' + p − k), v_r, i)}           ▷ Umleitung -R-> w_k^i-Kante
6:      E_R ← E_R ∪* {(v_r, (k' + p − k + 1, p'), v', i)}      ▷ -R-> von v_r → v'
7:      if |v| + p' − k' + 1 ≠ |v'| then
8:          l ← k' − |v| − 1
9:          E_L ← E_L ∪* {(v_r, (l, p' − |v'| + 1), v', i)}    ▷ <-L- von v_r → v'
10:     else                                                   ▷ Präfixfall
11:         for each (v', (k'', p''), v'', s_nr) ∈ E_L^{v→} do
12:             E_L ← E_L ∪* {(v_r, (k'', p''), v'', s_nr)}    ▷ <-L- von v_r → v''
13:         end for
14:     end if
15:     return v_r, k' + p − k
16: end function
```

Wenn v_r ein Präfix von v' ist, kann v_r alle linken Kanten von v' kopieren. Neben v_r gibt *split_edge* auch noch die Endposition dieses Knotens ($k' + p - k$) zurück. Beide Rückgabeparameter werden zur späteren Erzeugung der noch fehlenden linken Kanten benötigt. Zurück in *update* erhält v_r durch die beiden Aufrufe von *set_suffixlink* innerhalb von *update* einen Suffixlink zum aktiven Knoten. Indes wird gleichzeitig eine umgekehrte linke Kante erzeugt, deren Label dem des oberen Teils der aufgetrennten linken Kante entspricht. Zudem muss aber trotzdem noch überprüft werden, ob eine linke w_l^i-Kante existiert (Algorithmus 4, Zeile 24), da diese Kante auch durch den Kopiervorgang im Präfixfall nicht zwangsläufig entsteht.

5.1.2. Umleiten einer Kante

Auf das Aufsplitten einer Kante folgt nun wieder ein Kanonisierungschritt, der die möglichen Einfügepositionen aller nächstkürzeren Suffixe liefert. Führt so die im nächsten Durchlauf der *while*-Schleife identifizierte w_k^i-Kante auf denselben Knoten v' (Algorithmus 4, Zeile 5) wie die im vorigen Durchlauf aufgetrennte w_k^i-Kante, wird diese durch *redirect_edge*$(v_s, (k, p-1), v_r, i)$ auf v_r umgeleitet, da der

neue Knoten nun ein Präfix des Labels dieser Kante repräsentiert. In den Schritten 10 und 11 des Beispiels (siehe Abbildungen 3.26, 3.27) ist dies an der d-Kante $(v_\varepsilon, dcd, v_w)$, die erst auf v_w zeigt und auf v_3 umgeleitet wird, nachvollziehbar.

Algorithmus 10 Unterfunktion zur Umleitung einer Kante

1: **function** *redirect_edge*$(v, (k, p), v_r, i)$
2: $\quad (v, (k', p'), v', s_{nr}) \leftarrow (v, w_k^i \alpha, v', s_{nr}) \in E_R^{v\rightarrow}$ ▷ Finde w_k^i-Kante, $\alpha \in \Sigma^*$
3: $\quad E_R \leftarrow E_R \cup^* \{(v, (k', k' + p - k), v_r, s_{nr})\}$ ▷ Umleitung $\xrightarrow{R} w_k^i$-Kante
4: $\quad$ **return** k', s_{nr}
5: **end function**

Die Umleitung linker Kanten findet nicht in *redirect_edge* selbst, sondern innerhalb des Kanonisierungsschritts, der nach der Erzeugung neuer Knoten abläuft, in der Funktion *canonize* statt. Der Grund hierfür ist, dass *redirect_edge* immer nur am Ende eines walkdowns aufgerufen wird, wenn eine rechte Kante umgeleitet werden muss. Mögliche Umleitungen linker Kanten, die sich auf dem Weg nach unten befinden, würden so übersprungen werden. In *canonize* wird, wenn v_r nicht leer ist, zunächst wieder der Startpointer l anhand der dafür üblichen Berechnung mittels einer rechten Kante und deren Startpointer beziehungsweise Ausgangsknoten bestimmt. Wenn der Startknoten v die Wurzel darstellt, wird nur das Zeichen rechts von k' gewählt, da durch das Besuchen von $v_\perp$ bereits ein Zeichen abgeschnitten worden ist. In beiden Fällen wird nun mit l die bisherige von v ausgehende linke Kante gesucht (Algorithmus 6, Zeile 14). Wenn diese Kante zusammen mit der Länge des Startknotens v kürzer als ihr Zielknoten v'' ist (Algorithmus 6, Zeile 15), muss diese nun auf den neuen Knoten v_r umgeleitet werden. Die Startposition dieser Kante ergibt sich aus dem Startpointer der alten Kante k'', der Endpointer aus der Startposition des neuen Knotens v_r, die sich aus dessen Endposition r_{end} abzüglich seiner Länge berechnet.

5.1.3. Abspalten eines Knotens

Muss wie im letzten Schritt der Beispielkonstruktion (siehe Abbildung 3.27) durch einen neuen unmittelbaren linken Kontext ein neuer Knoten aus einem bestehenden Knoten herausgelöst werden, so wird dies in der Implementierung durch die letzte Funktion, die in *update* aufgerufen wird, *separate_node*$(v_s, (k, p), i)$ umgesetzt.

Algorithmus 11 Unterfunktion zum Abspalten eines Knotens

```
1:  function separate_node(v, (k, p), i)
2:      (v', k') ← canonize(v, (k, p), ∅, ∅, i)
3:      if k' ≤ p then
4:          return (v', k')                                  ▷ Impliziter Fall
5:      end if
6:      if |v'|= |v|+p − k + 1 then
7:          return (v', k')
8:      end if
9:      V ← V ∪ {v_r'}                                       ▷ v^+ Neuer Separate-Knoten v_r'
10:     |v_r'|← |v|+p − k + 1
11:     for each (v', (k'', p''), v'', s_nr) ∈ E_R^{v→} do
12:         E_R ← E_R ∪* {(v_r', (k'', p''), v'', s_nr)}     ▷ R→ von v_r' → v''
13:     end for
14:     l, s_nr ← redirect_edge(v, (k, p), v'_r, i)
15:     set_suffixlink(v_r', sl(v'), p + 1, i)
16:     set_suffixlink(v', v_r', l + 1, s_nr)
17:     E_L ← E_L ∪* {(v_r', (p − |v_r'|, 1), v_{w^i}, i)}   ▷ L← von v_r' →, v_{w^i}
18:     repeat
19:         redirect_edge(v, (k, p), v_r', i)
20:         (v, k) ← canonize(sl(v), (k, p − 1), v_r', p − |v_r'|, i)
21:     until (v', k') ≠ canonize(v, (k, p), ∅, ∅, i)
22: end function
```

Ein neuer unmittelbarer Linkskontext wird erkannt, wenn der Test in Zeile 6 fehlschlägt. Anschließend wird ein neuer Knoten $v_{r'}$ erzeugt. $v_{r'}$ ist wie im Beispielablauf erwähnt immer ein Suffix von v'. Deshalb kopiert $v_{r'}$ alle rechten ausgehenden Kanten von v' (Algorithmus 11, Schleife in Zeilen 11-13). Folgerichtig muss der Suffixlink von v' nun auf $v_{r'}$ zeigen und der Suffixlink von $v_{r'}$ auf das Ziel des alten Suffixlinks ($sl(v')$) von v'. Die beiden umgekehrten linken Kanten, die dabei erzeugt werden, starten nun für den Suffixlink von $v_{r'}$ bei $p+1$ und für den von v' bei $l+1$. l selbst gibt die Startposition der aktuellen aktiven Kante an. Im Beispiel liegt $l+1$ damit bei Position 8 und bezieht sich auf den Buchstaben c. Damit ist $l+1$ immer der Buchstabe, der das letzte Vorkommen des ursprünglichen linken Kontextes, in dem $v_{r'}$ auftritt, darstellt, was wiederum bedeutet, dass mit diesem Label eine linke Kante von v' wegführen muss. Die von $sl(v)$ nach $v_{r'}$ führende linke Kante wird durch $p+1$ angegeben und bezieht sich damit auf das zuletzt eingelesene Zeichen. Im Beispiel ist dies Position 11 und damit das Zei-

chen d. Der zweite (neue) unmittelbare linke Kontext, in dem $v_{r'}$ nun auftritt, wird durch eine neue linke Kante (Algorithmus 11, Zeile 17) angegeben, deren Startposition auf das Zeichen, das links neben dem durch $v_{r'}$ repräsentierten Infix steht, gesetzt wird. Wie in den vorherigen Runden, in denen Kanten aufgetrennt werden mussten, muss nun auch hier überprüft werden, ob noch andere linke oder rechte Kanten auf den neuen Knoten $v_{r'}$ umgeleitet werden müssen. Zur Identifikation der dafür nötigen Stellen wird analog dazu erneut die Funktion *canonize* genutzt.

In der hier vorgestellten Implementierung fehlt zum Schluss eine linke Kante, die jedes $w^{i^{rev}}$ darstellt und von v_s auf v_{w^i} führt. Diese Kante wird deshalb in *build_scdawg* (Algorithmus 3, Zeile 12) nachgeholt, sobald alle Zeichen eines Wortes w^i verarbeitet wurden. Da diese Kanten innerhalb von *set_suffixlink* entstehen, werden gleichzeitig Suffixlinks von v_{w^i} auf v_s erzeugt. Aufgrund der Verwendung gleicher Start- und Endmarker für jedes Wort aus W ist es auch in der nachfolgenden off-line Konstruktion wichtig, diese Suffixlinks zu erzeugen, da ansonsten die entsprechenden linken Kanten in umgekehrter Richtung fehlen. Nach Beendigung dieses letzten Schrittes werden nun alle Infixe von W und alle Infixe von W^{rev} in der SCDAWG-Struktur dargestellt.

5.2. Off-line Konstruktion

Wenn die zu indexierenden Eingabewörter bekannt sind, kann anstelle der on-line Konstruktion einer SCDAWG-Struktur auch zuerst eine CDAWG-Struktur erstellt werden. Anschließend können dann alle linken Übergänge erzeugt werden. Dieses Vorgehen beschreibt damit einen off-line Algorithmus, bei dem ein *CDAWG'(W)* zwar durch den on-line Algorithmus von Inenaga et al. erstellt wird, aber erst in einem zweiten Schritt die linken Übergänge erzeugt werden. Damit wird ähnlich verfahren wie bei der off-line Konstruktion eines CDAWGs, die von Blumer et al. [8] beschrieben wurde, bei der zuerst eine DAWG-Struktur aufgebaut wird, die anschließend zu einer CDAWG-Struktur kompaktiert wird. Bei der off-line Konstruktion einer SCDAWG-Struktur ergibt sich aber der Vorteil, dass die dafür notwendige CDAWG-Struktur nicht mehr Speicherplatz verbraucht, da die Knotenmengen von *CDAWG'(W)* und *SCDAWG'(W)* identisch sind. Somit kann als Ausgangspunkt der im vorherigen Kapitel respektive in [42] gegebene Pseudocode benutzt werden, wenn alle Stellen, an denen linke Übergänge erzeugt werden, entfernt werden. Um nachträglich alle linken Übergänge zu erzeugen, kann die in Algorithmus 1 beschriebene Tiefensuche auf *CDAWG'(W)* angewendet werden. Die hier gezeigte Prozedur *add_left_edges()* zeigt das nachträgliche Hinzufügen der linken Übergänge bei einer endlichen Menge von Wörtern W.

Algorithmus 12 Hinzufügen der linken Kanten zu *CDAWG'(W)*

Vorbedingung: Die CDAWG-Struktur $C = (V, E)$ einer endlichen Menge von Zeichenketten W mit $W = \{w^1, \ldots, w^i, \ldots, w^n\}$ und $w^i \in \Sigma^*$.

```
1:  procedure add_left_edges()
2:      for each v ∈ DFS(v_ε, false) do
3:          for each (v, (k, p), v') ∈ E_R^{v→} do
4:              if v = v_ε then
5:                  l ← k
6:                  E_L ← E_L ∪* {(v, (l, p − |v'|+1), v')}        ▷ ←^L von v → v'
7:              else
8:                  l ← k − |v|−1
9:                  if p − k + |v|+1 = |v'| then
10:                     continue
11:                 end if
12:                 E_L ← E_L ∪* {(v, (l, 1), v')}                 ▷ ←^L von v → v'
13:             end if
14:         end for
15:     end for
16:     for each v ∈ DFS(v_ε, false) do                           ▷ Umdrehen der Suffixlinks
17:         for each (v, (k, p), v') ∈ E_R^{v→} do
18:             v'' ← sl(v')
19:             E_L ← E_L ∪* {(v'', (p − |v''|, p − |v'| + 1), v')}   ▷ ←^L von v'' → v'
20:         end for
21:     end for
22:     for each v ∈ DFS(v_ε, true) do                             ▷ Kopieren der Präfix-Kanten
23:         for each (v, (k, p), v') ∈ E_R^{v→} do
24:             if p − k + |v|+1 = |v'| then
25:                 for each (v', (k', p'), v'') ∈ E_L^{v'→} do
26:                     if ¬(v, (k', p'), v'') ∈ E_L^{v→} then
27:                         E_L ← E_L ∪* {(v, (k', p'), v'')}      ▷ ←^L von v → v''
28:                     end if
29:                 end for
30:             end if
31:         end for
32:     end for
33: end procedure
```

Die Subroutine *add_left_edges()* benötigt drei nacheinander ablaufende Tiefensuchen auf der CDAWG-Struktur C, um dieser alle linken Kanten hinzuzufügen. In jedem Durchlauf werden alle Knoten und Kanten genau einmal durchlaufen, was durch die in Algorithmus 1 gezeigte Implementierung gewährleistet wird. Dabei ist jedoch die Reihenfolge, in der die Tiefensuchen ablaufen, wichtig, denn in der ersten DFS können falsche linke Übergänge entstehen, die durch die zweite DFS korrigiert werden. Die erste Tiefensuche (Algorithmus 12, Zeilen 2-15) fügt, wenn v auf der Wurzel steht, eine linke Kante hinzu, deren Startpointer mit k besetzt wird und deren Endpointer auf die Position des ersten Zeichens von v' zeigt. Diese ergibt sich aus der Endposition p von v' abzüglich dessen Länge plus 1. In Zeile 9 wird nun überprüft, ob v ein Präfix des Zielknotens v' darstellt. Ist dies der Fall, wird die innere Schleife weitergeschaltet und das nächste Kind von v' betrachtet. Wenn v kein Präfix darstellt, wird eine Kante erzeugt, die zu einem der vorhandenen Sinkknoten, der durch v' angesprochen wird, führt, und deren Endpointer damit bei 1 liegt. Der Startpointer dieser Kante liegt bei der Position des Zeichens, das sich links von v befindet, was dementsprechend durch $k - |v| - 1$ angegeben wird.

Umkehrung der Suffixlinks

In der zweiten DFS, die ebenfalls in Präordnung läuft, wird analog zur Funktion *set_suffixlink* (siehe Algorithmus 5) der on-line Konstruktion jeweils zu jedem Suffixlink eine linke Kante in umgekehrter Richtung angelegt. Es ist, wie bereits erwähnt, wichtig, dass dieser Schritt nach der ersten DFS abläuft, da so eventuell falsche linke Kanten berichtigt werden.

Kopieren der Präfix-Kanten

Die letzte DFS muss in Postordnung ablaufen, da dort jedes längere Präfix die linken Kanten seines kürzeren Vorgängers kopiert. Im Beispiel (siehe Abbildung 3.27) ist v_1 (c) ein Präfix seines Zielknotens v_3 (cd). Wenn v_1 nun eine der von v_3 mit σ startenden Kanten nicht schon besitzt, wird diese kopiert. Im Beispiel ist dies anhand der Kante $(v_3, dcbacbc, v_w)$ beziehungsweise $(v_1, dcbacbc, v_w)$ nachvollziehbar. Wichtig ist jedoch, dass die von v_3 mit b startende Kante nicht mitkopiert wird, da v_1 diese bereits im vorherigen Schritt beim Umdrehen der Suffixlinks erhalten hat.

Die off-line Variante der SCDAWG-Struktur bietet eine gute Übersicht über die verschiedenen Fälle, die beim Hinzufügen linker Kanten auftreten können. Somit stellt sie bei einer Implementierung einen sinnvollen Einstiegspunkt dar.

5.3. Vorteile der SCDAWG - Struktur

5.3.1. Speicherverbrauch

Wenngleich sich alle behandelten Indexstrukturen bis auf die Suffixtries linear in Bezug auf ihre Größe und den damit verbrauchten Speicher verhalten, benötigen alle Strukturen deutlich mehr Speicher als ihre urprünglichen Eingabewörter (siehe Lemmas 3.2.2, 3.3.2, 3.4.2). Der Grund dafür liegt in den Äquivalenzrelationen, die je nach Kontexten eine Vielzahl neuer Knoten und Kanten erzeugen. Vergleicht man die String-Indexstrukturen in Bezug auf Knoten- und Kantenanzahl, zeigt sich für verschieden große Eingabestrings folgendes Bild.

	Suffixtrie	Suffixbaum	DAWG	CDAWG	SCDAWG
$\#abcbc\$\#abcab\$$: 13 Bytes, 13 Zeichen					
Knoten	90	20	20	5	5
Kanten	89	19	19	12	21
Text-Seite: 7 Kilobyte, 6945 Zeichen					
Knoten	21,418,626	7,355	11,995	1,163	1,163
Kanten	21,418,625	7,354	14,915	4,083	8,094
Ausschnitt aus Text-Korpus: 106 Kilobyte, ca. 106000 Zeichen.					
Knoten	-	161,001	165,962	25,273	25,273
Kanten	-	161,000	227,515	86,826	170,358
Text-Korpus: 1.2 Megabyte, ca. 1.250.000 Zeichen					
Knoten	-	1,921,704	1,922,811	366,070	366,070
Kanten	-	1,921,703	2,730,597	1,173,856	2,355,669

Tabelle 5.1.: Größenvergleich der Indexstrukturen für verschiedene Eingaben.

Aufgrund der Äquivalenzrelation $\sim_w$, die sowohl linke als auch rechte Kontexte berücksichtigt, schneidet die CDAWG-Struktur in Bezug auf den Speicherverbrauch besser ab als Suffixtrie, Suffixbaum und DAWG. Für die beiden letzten Beispiele wurde ein kleines Textkorpus und ein Ausschnitt aus diesem indexiert. Wegen des quadratischen Speicherverbrauchs konnte dafür jedoch kein Suffixtrie mehr berechnet werden. Die nachfolgende Abbildung stellt die sich ergebenden Knoten- und Kantenanzahlen graphisch in Millionen dar. Die Werte der ersten Beispiele für die Suffixtries sind in beiden Darstellungen aufgrund ihres quadratischen Wachstums nahezu auf der Y-Achse zu finden. Da sich Suffixbaum und DAWG in Bezug auf ihre Knotenzahl ähnlich verhalten, überlagern sich deren Werte in der ersten Grafik fast vollständig.

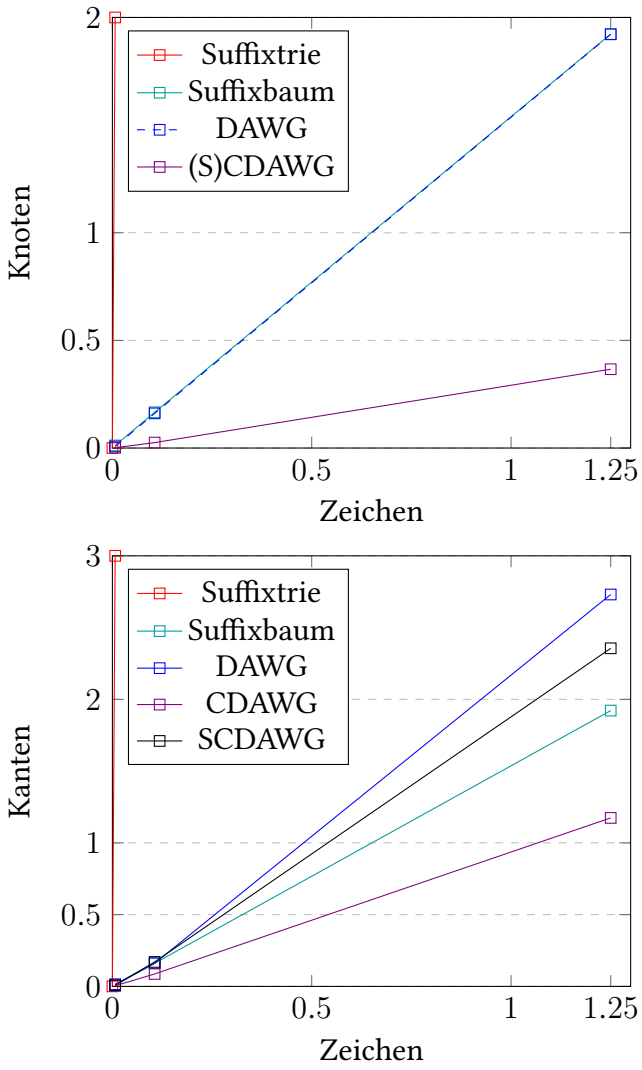


Abbildung 5.1.: Knoten- und Kantenanzahlen der Indexstrukturen für größer werdende Eingabestrings dargestellt in Millionen.

Wegen ihrer zusätzlichen linken Übergänge liegt mit Blick auf die Kantenanzahl die symmetrische CDAWG-Struktur sogar oberhalb des Suffixbaums und etwa bei der doppelten Anzahl des einfachen CDAWGs. Da die Knotenzahl eines SCDAWGs aber genau der eines CDAWGs entspricht, spielt dieser Umstand nur eine untergeordnete Rolle, zumal sich in der simultanen Unterstützung der Infixe aller umgekehrten Wörter, die *SCDAWG(W)* liest, wertvolle Informationen verbergen, die dabei helfen, bestimmte Teilmengen aus Infixen ausfindig zu machen.

5.3.2. Zeitverbrauch

Der in Kapitel 3.6.1 beschriebene on-line Algorithmus von Inenaga et. al [41] weist nicht nur lineare Speicherkomplexität auf, sondern ist auch im Zeitverbrauch linear. Zur Einordnung werden an dieser Stelle einige Zeitmessungen für die Konstruktion einer SCDAWG-Struktur für verschieden große Eingabewörter angegeben. Bei den hier verwendeten Eingabetexten handelt es sich um Auszüge verschiedener Korpora, denen auch verschiedene Alphabete zugrundeliegen.

	7 KB	150 KB	500 KB	1,5 MB	5,5 MB
Laufzeit in Sek.	0,007	0,7	1,03	8,3	33,5
Zeichen	6944	143.664	235.718	1.542.283	5.600.651

Tabelle 5.2.: Zeitverbrauch des on-line Algorithmus zur Konstruktion einer SCDAWG-Struktur bei verschieden großen Eingabewörtern.

Stellt man die bei der Indexierung der Eingabewörter verbrauchte Zeit in Sekunden den Zeichenanzahlen selbiger gegenüber, ergibt sich die folgende Grafik, die die zeitliche Linearität des Algorithmus verdeutlicht:

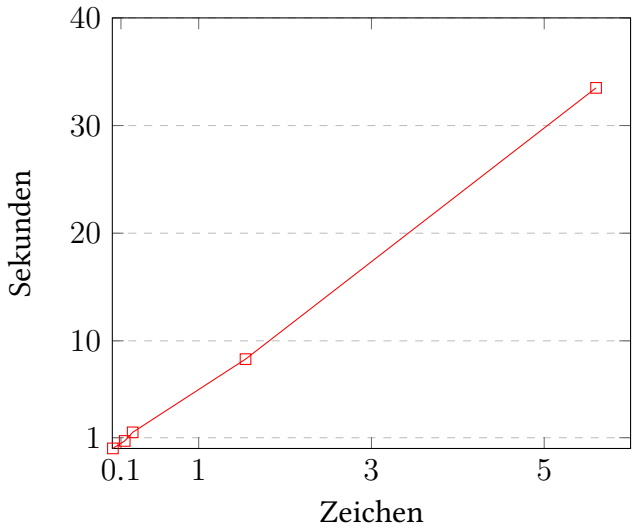


Abbildung 5.2.: Zeitverbrauch bei der on-line Konstruktion einer SCDAWG-Struktur größer werdender Eingabewörter dargestellt in Sekunden im Verhältnis zur Anzahl der Zeichen der Eingabewörter dargestellt in Millionen.

5.3.3. Beschaffenheit der Knoten

Bei einer Indexierung von Martin Luther Kings berühmter „I Have a Dream“-Rede vom 28.8.1963, die aus ca. 1.600 Wörtern mit circa 9.000 Zeichen besteht, entstehen in einer DAWG-Struktur beziehungsweise in einem Suffixbaum etwa 13.000 Knoten während die (S)CDAWG-Struktur nur etwa 2.000 Knoten benötigt. Die geringere Knotenzahl einer (S)CDAWG-Struktur ist, wie zuvor erwähnt, in der in Lemma 3.4.1 beschriebenen Relation $\sim_w$ begründet. Ordnet man die Knoten des Suffixbaums und des DAWGs absteigend der Länge nach, so werden für die Textpassage „ *From every mountainside, let freedom ring*“ folgende Knoten generiert:

Suffixbaum ($\sim_w^L$)	DAWG ($\sim_w^R$)
„ *From every mountainside, let freedom ring*“	„ *From every mountainside, let freedom ring*“
„*From every mountainside, let freedom ring*“	„ *From every mountainside, let freedom ring*“
„*rom every mountainside, let freedom ring*“	„ *From every mountainside, let freedom rin*“
„*om every mountainside, let freedom ring*“	„ *From every mountainside, let freedom ri*“
„*m every mountainside, let freedom ring*“	„ *From every mountainside, let freedom r*“
„ *every mountainside, let freedom ring*“	„ *From every mountainside, let freedom* “
„*every mountainside, let freedom ring*“	„ *From every mountainside, let freedom*“
„*very mountainside, let freedom ring*“	„ *From every mountainside, let freedo*“
„*ery mountainside, let freedom ring*“	„ *From every mountainside, let freed*“
„*ry mountainside, let freedom ring*“	„ *From every mountainside, let free*“
„*y mountainside, let freedom ring*“	„ *From every mountainside, let fre*“
⋮	⋮

Tabelle 5.3.: Beispiele nach links oder rechts abgeschlossener Knoten.

Unter $\sim_w^L$ entstehen somit Knoten, die auf der rechten Seite abgeschlossen sind (vgl. Tabelle 3.2) und auf der linken jeweils kürzer werden, für $\sim_w^R$ ergibt sich das umgekehrte Bild (vgl. Tabelle 3.3). In einer (S)CDAWG-Struktur hingegen entstehen nur Knoten, die sowohl nach links als auch nach rechts abgeschlossen sind, weshalb dort nur ein Knoten „ *From every mountainside, let freedom ring*“ existiert. Obwohl ein Knoten für die Textpassage also in allen drei Indexstrukturen gebildet wird, ist es in einem (S)CDAWG am einfachsten, solche Knoten zu identifizieren, die natürlichere Textabschnitte, die innerhalb des Ursprungstexts wiederholt auftreten, repräsentieren. Ordnet man die Knoten des (S)CDAWGs der Länge nach absteigend, erhält man als die längsten 20 Knoten:

.I have a dream today!I have a dream that one day
. We can never be satisfied as long as our
From every mountainside, let freedom ring
We can never be satisfied as long as
. With this faith, we will be able to
. One hundred years later, the Negro
With this faith, we will be able to
ne hundred years later, the Negro
We cannot be satisfied as long as
have come to realize that their
. One hundred years later, the
I have a dream that one day eve
day when all of God's children
be satisfied as long as the
.I have a dream that one day
sweltering with the heat of
ne hundred years later, the
e refuse to believe that the

Tabelle 5.4.: Die 20 längsten Knoten im (S)CDAWG des Textes „I Have a Dream“ von Martin Luther King.

5.3.4. Verbindungen zwischen Knoten

In Tabelle 5.4 fällt auf, dass Knoten existieren, die ein Infix eines anderen Knotens sind, etwa „*. We can never be satisfied as long as our*“ und „ *We can never be satisfied as long as*“. Dies trifft natürlich auch auf viele kürzere Knoten zu. In der nachfolgenden Abbildung 5.3 ist eine SCDAWG-Struktur der beiden Wörter $w^1 = \#aabcd\$$ und $w^2 = \#abbce\$$ dargestellt. In diesem Beispiel führt, bedingt durch die Kombination der gleichzeitigen Darstellung der Infixe von w und w^{rev} einer SCDAWG-Struktur, von jedem kürzeren Knoten v mindestens eine linke oder eine rechte Kante zu einem direkten Kind v' von v, wenn v ein Infix von v' ist. So existiert zum Beispiel ein Knoten v_4, der das Infix b repräsentiert. Zu Knoten v_3, der das Infix ab abbildet, existiert von v_4 aus nur eine linke Kante mit dem Übergangslabel a (Bereich rot umrandet), da b nur rechts von a auftritt und es damit keine rechte Kante von v_4 nach v_3 geben kann. Die Tatsache, dass in w^1 und w^2 b ein Infix von ab ist, kann also nur an der symmetrischen CDAWG-Struktur abgelesen werden. Selbiges gilt auch für Knoten v_6, der das Abschlusssymbol $\$$ darstellt und von dem damit ausschließlich linke Kanten starten können.

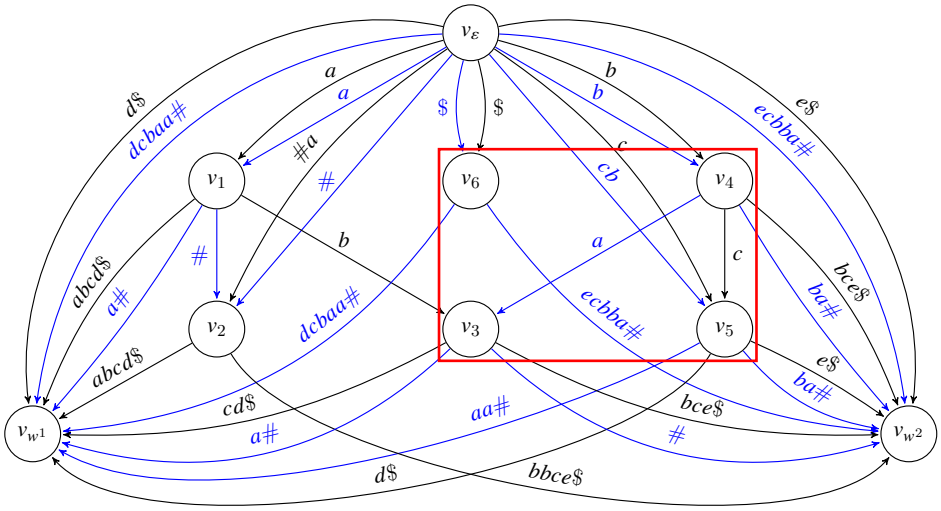


Abbildung 5.3.: *SCDAWG*({#*aabcd*$, #*abbce*$}).

Verallgemeinert lässt sich sagen, dass, wenn v ein Präfix irgendeines längeren Nachfolgeknotens v' ist, ein Pfad aus rechten Kanten von v nach v' existiert, oder aber, wenn v ein Suffix eines längeren Nachfolgeknotens v' ist, eine Folge linker Kanten von v nach v' existiert. Wie das nächste Beispiel zeigt, müssen, um alle Vorkommen eines Knotens v innerhalb seiner Nachfolger zu erkennen, entweder alle maximalen Pfade von rechten Übergängen bis zu Knoten, die nicht mehr nach rechts erweitert werden können, gefolgt von maximalen Pfaden linker Kanten traversiert werden oder umgekehrt maximale Folgen linker Kanten gefolgt von maximalen Folgen rechter Kanten.

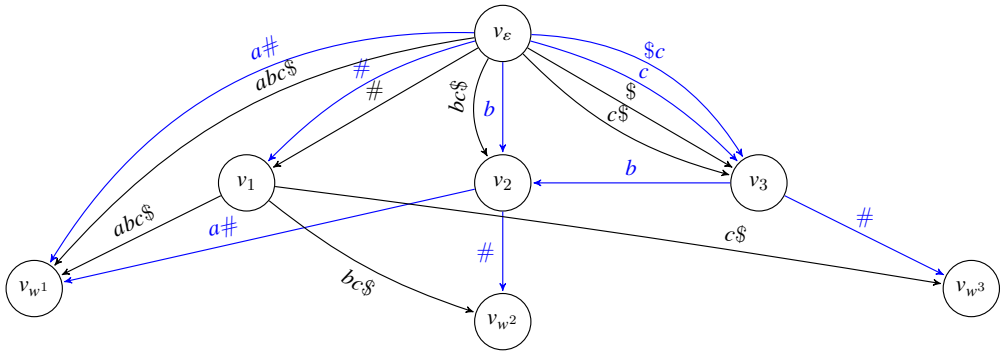


Abbildung 5.4.: *SCDAWG*({#*abc*$, #*bc*$, #*c*$}).

Betrachtet man die SCDAWG-Struktur für $W = \{\#abc\$, \#bc\$, \#c\$\}$, die in Abbildung 5.4 dargestellt ist, gibt es zum Beispiel für das Abschlusssymbol \$ einen maximalen rechten Pfad, der auf Knoten v_3 führt. Von dort existieren nun zwei maximale Folgen linker Kanten, die einerseits auf v_{w^3} und andererseits über v_2 auf v_{w^2} führen. Dieses Vorgehen, alle maximalen rechten Folgen zu verfolgen und anschließend die linken Folgen, führt, wie auch für jedes andere Infix aus W erkennbar ist, dazu, dass alle längeren Knoten, die dieses Infix beinhalten, der Reihe nach besucht werden. Dasselbe gilt für die umgekehrte Methodik, wenn von einem Knoten aus zuerst alle linken Pfade maximal verfolgt werden und dann alle rechten. Diese Eigenschaft ergibt sich direkt aus der Äquivalenzrelation $\sim_w$.

6. On-line Suchfunktionen auf einer SCDAWG-Struktur

Im folgenden Kapitel werden eigene Implementierungen grundlegender Suchverfahren aus dem Bereich des Text-Minings beschrieben, deren Ausgangspunkt stets eine SCDAWG-Struktur ist. Die dargestellten on-line Verfahren bilden zum Teil die Grundlage der in Teilen III und IV beschriebenen Anwendungen.

6.1. Der CDAWG als invertierte Datei

Um eine CDAWG-Struktur als invertierte Datei, beziehungsweise als Suchindex über einer Menge von Wörtern W benutzen zu können, wird zunächst der Begriff der invertierten Datei in Anlehnung an Blumer et al. [8] präzisiert.

Definition 6.1.1 (Invertierte Datei [8])**.** Sei $W = \{w^1, w^2, \ldots, w^n\}$ mit $w^i \in \Sigma^*$ für $1 \leq i \leq n$ und $n \geq 1$ und $K \in \Sigma^*$ eine Menge von *Schlüsselwörtern* beziehungsweise *keywords*. Dann ist eine *invertierte Datei* oder ein *invertierter Index* für (Σ, K, W) eine abstrakte Datenstruktur, die die folgenden Funktionen implementiert:

1. *find*: $\Sigma^* \rightarrow K$, wobei für $s \in \Sigma^*$ $x = find(s)$ das längste Präfix x von s darstellt, sodass $x \in K$ ein Teilwort eines Textes aus W ist.

2. *freq*: $K \rightarrow \mathbb{N}$, wobei $freq(s)$ die Anzahl der Vorkommen von s als Teilwort in Wörtern aus W ist.

3. *locations:* $K \rightarrow 2^{\mathbb{N}\times\mathbb{N}}$, wobei $locations(s)$ eine Menge geordneter Paare der Textnummern und Startpositionen innerhalb von W ist, an denen s auftritt.

Bei Anwendung auf eine (S)CDAWG-Struktur über einer endlichen Menge von Zeichenketten W stellt nun *Infix(W)* die Menge der keywords dar. Da sich Definition 6.1.1 auf eine CDAWG-Struktur bezieht und auch keine expliziten Codebeschreibungen angegeben werden, werden im Folgenden eigene Implementierungsansätze der Funktionen $find(s)$, $freq(s)$ und $locations(s)$ vorgestellt, die auf einer SCDAWG-Struktur basieren.

6.2. Auffinden eines Teilwortes in einer Wortmenge

Anstelle von $find(s)$ wird eine Funktion *find_p* implementiert, der neben dem Anfragewort s als zweiter Parameter ein Knoten aus der Knotenmenge von S_C übergeben wird. Damit wird in S_C, wenn $find\_p(s, v)$ mit dem Wurzelknoten v_ε aufgerufen wird, ein Suchpfad traversiert, sodass entweder s vollständig oder mit x das längste echte Präfix von s als Infix eines Wortes $w_i \in W$ gefunden wird. Es spielt keine Rolle, ob ein solcher Suchpfad auf einem Knoten oder inmitten eines Labels endet. Die Zeitdauer liegt damit bei $O(x)$ [8]. Der anschließend dargestellte Pseudocode realisiert $find\_p(s, v)$ über einer SCDAWG-Struktur S_C anhand einer rekursiven Funktion.

Algorithmus 13 Auffinden von Teillabelfolgen von einem Knoten $v \in V$

Vorbedingung: Die SCDAWG-Struktur $S_C = (V, E_R, E_L)$ einer endlichen Menge von Zeichenketten W mit $W = \{w^1, \ldots, w^i, \ldots, w^n\}$ und $w^i \in \Sigma^*$.

```
1:  function find_p(s, v)
2:      x ← ε
3:      (v, (k, p), v') ← (v, s_1 α, v') ∈ E_R^{v→}        ▷ Finde s_1-Kante, α ∈ Σ*
4:      if (v, (k, p), v') = ∅ then
5:          return x
6:      end if
7:      i ← get_w((v, (k, p), v'))
8:      j ← k, h ← 1
9:      while j ≤ p and h ≤ |s| do
10:         if w_j^i = s_h then
11:             x ← x ∘ w_j^i
12:         else
13:             return x
14:         end if
15:         j ← j + 1, h ← h + 1
16:     end while
17:     return x ∘ find_p(s_h ... s_|s|, v')
18: end function
```

Ausgehend von einem beliebigen Knoten v wird überprüft, ob es eine ausgehende rechte Kante von diesem Knoten gibt, deren Label mit dem ersten Zeichen aus s startet (Zeile 2). Wird keine solche Kante gefunden, wird das bisher gefundene längste Präfix x von s zurückgegeben. Andernfalls wird für die gefundene Kante

$(v, (k, p), v')$ durch die Hilfsfunktion *get_w* der Index i des ersten Wortes $w_i \in W$ geholt, in dem das Kantenlabel auftritt. In der darauffolgenden *while*-Schleife werden die Zeichen der Anfrage s mit denen des Labels der von v ausgehenden Kante abgeglichen und im Falle eines Mismatches x zurückgegeben. Jedes übereinstimmende Zeichen wird zu x addiert (Zeile 11). Entsprachen alle Zeichen des Labels (k, p) einem Präfix von s, ruft sich die Funktion in Zeile 17 selbst mit dem verbleibenden Suffix der Anfrage $s_h \ldots s_{|s|}$ und dem Zielknoten der Kante v' auf.

6.3. Bestimmung von Häufigkeiten

Damit mit *find_p*(s, v_ε) auch die Häufigkeit, mit der s innerhalb von W auftritt, ermittelt werden kann, muss jedem Knoten von S_C ein Häufigkeitslabel hinzugefügt werden. Dazu kann folgende Prozedur *add_freq_labels* betrachtet werden. Zur Speicherung aller Knotenhäufigkeiten wird zusätzlich eine globale Variable *nodes_freqs* benutzt, die innerhalb der Funktion verwendet wird. Die Funktion *add_freq_labels* entspricht wiederum einer rekursiven Tiefensuche, bei der jeder Knoten genau einmal betrachtet wird. Inital wird *add_freq_labels* also mit dem Wurzelknoten v_ε und einer leeren Liste, die alle bereits besuchten Knoten speichert, aufgerufen. In *nodes_freqs* wird nun jedem Sinkknoten v_{w^i}, der besucht wird, der Wert 1 gesetzt und dieser Wert anschließend zurückgeliefert (Zeilen 4-6). Nachfolgend wird für jede rechte Kante des aktuell besuchten Knotens v, wenn es sich bei diesem nicht um einen Sinkknoten handelt, seine Häufigkeit aus der Häufigkeit seiner Nachfolger berechnet (Zeilen 10-12). Ist die Häufigkeit eines direkten Nachfolgers v' bereits durch einen anderen Knoten bestimmt worden, so wird diese nicht erneut berechnet, sondern aus der Ergebnisliste entnommen (Zeile 14). Besitzt ein besuchter Knoten keine rechten Übergänge, wird die Häufigkeit von v anhand seiner linken Nachfolger bestimmt. Als Beispiele solcher linker Pfade können erneut die Wege von v_3 nach v_{w^1} oder von v_3 nach v_{w^2}, die in Abbildung 5.4 illustriert sind, betrachtet werden. Schließlich muss in jedem Rekursionsschritt die Häufigkeit von v in *nodes_freqs* gespeichert (Zeile 27) und diese zurückgegeben werden (Zeile 28).

Wenn durch *add_freq_labels* zu jedem Knoten aus S_C seine Vorkommenshäufigkeit in W gespeichert wurde, kann eine Funktion *find_freq*(s, v) mittels eines Lookups innerhalb von *find_p*(s, v) realisiert werden, bei dem die Häufigkeit des letzten Knotens des durchwanderten Pfads ausgegeben wird. Die Zeitkomplexität von *find_freq*(s, v) liegt wieder bei $O(x)$ [8]. Der Zeitverbrauch von *add_freq_labels* liegt, da S_C einmal komplett durchwandert wird, bei $O(|V| + |E|)$.

Algorithmus 14 Hinzufügen von Frequenzlabels zu einer SCDAWG-Struktur

Vorbedingung: Die SCDAWG-Struktur $S_C = (V, E_R, E_L)$ einer endlichen Menge von Zeichenketten W mit $W = \{w^1, \ldots, w^i, \ldots, w^n\}$ und $w^i \in \Sigma^*$.

```
1:  nodes_freqs ← {}   ▷ Globale Variable, die zu jedem Knoten dessen Häufigkeit speichert.
2:  function add_freq_labels(v, visited)
3:      visited.push(v)
4:      if v ∈ V_W then                    ▷ Sei V_W die Menge der Sinkknoten in S_C
5:          nodes_freqs[v] ← 1
6:          return 1
7:      end if
8:      v_freq ← 0
9:      for each (v, (k, p), v') ∈ E_R^{v→} do
10:         if v' ∉ visited then
11:             v'_freq ← add_freq_labels(v', visited)
12:             v_freq ← v_freq + v'_freq
13:         else
14:             v_freq ← nodes_freqs[v']
15:         end if
16:     end for
17:     if E_R^{v→} = ∅ then
18:         for each (v, (k, p), v') ∈ E_L^{v→} do
19:             if v' ∉ visited then
20:                 v'_freq ← add_freq_labels(v', visited)
21:                 v_freq ← v_freq + v'_freq
22:             else
23:                 v_freq ← nodes_freqs[v']
24:             end if
25:         end for
26:     end if
27:     nodes_freqs[v] ← node_freq
28:     return node_freq
29: end function
```

Ausblick

Die Funktion *add_freq_labels* wird in einigen später beschriebenen Anwendungen als Vorbild einer generischen Funktion zur globlen Annotation der Knoten einer SCDAWG-Struktur gesehen. Varianten von *add_freq_labels* werden etwa in den Kapiteln 8.3.1 und und 11.4 dazu verwendet, alle Knoten einer SCDAWG-Struktur mit Metainformationen zu versehen. Da deren Implementierung jedoch nur eine geringfügige Modifikation von *add_freq_labels* bedeutet, werden diese nicht mehr explizit angegeben.

6.4. Finden aller Startpositionen

Das Finden der Startpositionen, mit welchen ein Suchwort s innerhalb einer Wortmenge W auftritt, wird in [8] durch $locations(s)$ realisiert. $locations(s)$ lässt sich wie folgt beschreiben:

Lemma 6.4.1. (Blumer et al. [8]) Sei $s \in Infix(W)$ und $Rep([s]_w) = \gamma s \beta$, mit $\gamma, \beta \in \Sigma^*$, L = $\cup_{a \in \Sigma}$ $locations(s\beta a, v)$, T = $\{\langle i, j\rangle : s\beta \in Suffix(w^i)$ und $j = \left|w^i\right| - |s\beta|\}$, dann ist $locations(s) = L \cup T$.

Dieses Lemma drückt aus, dass die Berechnung von $locations(s)$ für jedes $s \in Infix(W)$ rekursiv abläuft unter der Annahme, dass der Knoten $v = Rep([s]_w)$ bereits gefunden wurde und die Länge von β zuvor bestimmt wurde. Hierzu wird die Liste L rekursiv berechnet, indem alle Knoten betrachtet werden, die über ausgehende Kanten von $Rep([s]_w)$ erreicht werden. L wird dann mit der Liste T vereinigt, für die die Startpositionen j von s in Bezug auf alle durch die erreichten Sinkknoten v_{w^i} repräsentierten Wörter $w^i \in W$ durch Abziehen der Länge von $s\beta$ von $|w^i|$, berechnet werden. Da die verwendeten Listen disjunkt sind und in jedem rekursiven Aufruf nur nicht leere Listen entstehen, ist die benötigte Zeit linear mit $O(m)$, wobei m die Länge der final bestimmten Liste bezeichnet. Addiert man die für das Auffinden von $v = Rep([s]_w)$ und die zur Bestimmung von β benötigte Zeit der Größe $O(|s|)$, ergibt sich für $locations(s)$ ein linearer Zeitaufwand der Größe $O(|s|+m)$. [8]

In einer SCDAWG-Struktur, die stets das #-Symbol als Start- und das $-Symbol als Endmarker verwendet, muss der beschriebene Algorithmus erneut abgewandelt werden, da wie in Kapitel 5.3.4 gezeigt manche Pfade nur durch linke Kanten verbunden sind. Diese abgewandelte Variante von $locations(s)$ wird als *find_loc* (s, v) bezeichnet.

Um zuerst $Rep([s]_w)$ zu ermitteln, kann die in Algorithmus 15 beschriebene Funktion *find_rep*(s, v, pl) betrachtet werden, die eine Modifikation von Algorithmus 13 darstellt. Initial wird *find_rep* mit einem Anfragewort s, dessen Repräsentant gesucht wird, sowie dem Wurzelknoten v_ε und einer Variablen $pl = 0$, die

die Länge des durchlaufenen Pfades ermittelt, aufgerufen. Anstelle des in *find_p* (s) verwendeten Präfixes x wird hier nun die Anzahl (mc) der übereinstimmenden Zeichen von s mit dem Kantenlabel gemerkt. Ist nach dem Ende der Schleife die Differenz von s und mc gleich 0, so werden die bisher gelaufene Pfadlänge und der Zielknoten v' zurückgegeben. Ansonsten ruft sich *find_rep* selbst mit dem verbleibenden Suffix von s und der bisherigen Pfadlänge auf. Sind nun zu einem beliebigen Wort s dessen Repräsentant beziehungsweise Knoten gefunden und gleichzeitig die Länge des abgelaufenen Pfades vermerkt, wird nun die eigentliche Funktion zum Auffinden der Startpositionen von s aufgerufen.

Algorithmus 15 Finden von $Rep([s]_w)$

Vorbedingung: Die SCDAWG-Struktur $S_C = (V, E_R, E_L)$ einer endlichen Menge von Zeichenketten W mit $W = \{w^1, \ldots, w^i, \ldots, w^n\}$ und $w^i \in \Sigma^*$

```
1:  function find_rep(s, v, pl)          ▷ s = Suchwort, v = Startknoten, pl = Pfadlänge
2:      (v, (k, p), v') ← (v, s_1 α, v') ∈ E_R^{v→}          ▷ Finde s_1-Kante, α ∈ Σ*
3:      if (v, (k, p), v') = ∅ then
4:          return ∅
5:      end if
6:      i ← get_w((v, (k, p), v'))
7:      j ← k, h ← 1
8:      pl ← pl + (p − k)
9:      mc ← 0
10:     while j ≤ p and h ≤ |s| do
11:         if w_j^i = s_h then
12:             mc ← mc + 1
13:         else
14:             return ∅
15:         end if
16:         j ← j + 1, h ← h + 1
17:     end while
18:     if |s| − mc = 0 then
19:         return (v', pl)
20:     else
21:         return find_rep(s_h ... s_|s|, v', pl)
22:     end if
23: end function
```

Zur Berechnung jeder Startposition wird nun die Funktion *find_loc*(v, pl) mit dem Repräsentanten $Rep([s]_w)$ und der von v_ε bis zu $Rep([s]_w)$ gelaufenen Pfad-

länge aufgerufen. Zur Bestimmung der Startposition jedes Vorkommens von s in w^i wird wie in Lemma 6.4.1 beschrieben von der Länge von w^i die Summe der bis dahin gelaufenen Pfadlängen rechter Übergänge, durch die verschiedene Suffixe von w^i entstehen, abgezogen. Somit wird, wann immer ein Sinkknoten gefunden wird, diese Berechnung ausgeführt und der Ergebnisliste *result* ein entsprechendes Positionspaar angefügt, das aus der Startposition und dem Index des jeweiligen Sinkknotens besteht. Letzterer gibt an, auf welches Wort aus W sich die gefundene Startposition bezieht. Die Identifikation des Wortindex wird hier durch eine Funktion $index\_of(v, V_W)$ übernommen, die zu jedem Sinkknoten aus der Menge der Sinkknoten V_W dessen Index liefert.

Algorithmus 16 Finden der Startpositionen eines Wortes s

Vorbedingung: Die SCDAWG-Struktur $S_C = (V, E_R, E_L)$ einer endlichen Menge von Zeichenketten W mit $W = \{w^1, \ldots, w^i, \ldots, w^n\}$ und $w^i \in \Sigma^*$.

```
1:  function find_loc(v, pl)                ▷ v = Startknoten, pl = Pfadlänge
2:      result ← {}
3:      if v ∈ V_W then                     ▷ Sei V_W die Menge der Sinkknoten in S_C
4:          result.push(index_of(v, V_W), p − pl)
5:      end if
6:      for each (v, (k, p), v') ∈ E_R^{v→} do
7:          if v' ∈ V_W then
8:              result.push(index_of(v', V_W), p − pl − (p − k))
9:          else
10:             result ← result + find_loc(v', pl + (p − k))
11:         end if
12:     end for
13:     if E_R^{v→} = ∅ then
14:         for each (v, (k, p), v') ∈ E_L^{v→} do
15:             if v' ∈ V_W then
16:                 result.push(index_of(v', V_W), p + |v'|−pl)
17:             else
18:                 result ← result + find_loc(v', pl)
19:             end if
20:         end for
21:     end if
22:     return result
23: end function
```

Wird ein Knoten besucht, der keine rechten Übergänge mehr besitzt und selbst auch kein Sinkknoten ist, werden nur noch linke Übergänge verfolgt, bis ein Sink-

knoten getroffen wird. Beim Verfolgen eines linken Pfades wird die Pfadlänge jedoch nicht mehr durch die linken Kanten verändert, da Erweiterungen nach links bei den Startpositionsberechnungen von s keine Relevanz besitzen.

6.5. Beispiele für *find_p*,*find_freq* und *find_loc*

Werden die drei Index-Funktionen wie beschrieben implementiert, liefern sie für die gewählten beiden Beispielmengen folgende Ergebnisse:

Beispiel 6.5.1. Sei $W = \{\#a\$, \#ab1\$, \#abc\$\}$, so ergibt sich für

$$\textit{find\_p}(abcd) = abc,$$

$$\textit{find\_freq}(a) = 3,$$

$$\textit{find\_loc}(a) = \{\langle 1, 2\rangle, \langle 2, 2\rangle, \langle 3, 2\rangle\}.$$

Beispiel 6.5.2. Sei $W = \{\#cockatoo\$, \#crocodile\$, \}$, so ergibt sich für

$$\textit{find\_p}(crow) = cro,$$

$$\textit{find\_freq}(oc) = 2,$$

$$\textit{find\_loc}(co) = \{\langle 1, 2\rangle, \langle 2, 5\rangle\}.$$

Beispiel 6.5.3. Sei $W = \{\#abc\$, \#bc\$, \#c\$\}$, so ergibt sich für

$$\textit{find\_p}(bcx) = bc,$$

$$\textit{find\_freq}(c) = 3,$$

$$\textit{find\_loc}(c) = \{\langle 1, 4\rangle, \langle 2, 3\rangle \langle 3, 2\rangle\}.$$

Beispiel 6.5.3 liegt die in Abbildung 5.4 gezeigte SCDAWG-Struktur zugrunde, bei der für die Aufrufe von $\textit{find\_freq}(c)$, beziehungsweise $\textit{add\_freq\_labels}()$ und $\textit{find\_loc}(c)$ zuerst rechte Erweiterungen verfolgt werden und im Anschluss linke Übergänge traversiert werden.

7. Zusammenfassung Teil II.

Die in Kapitel 5.1 gezeigte on-line Implementierung speichert basierend auf der in [75] vorgeschlagenen Idee, Kantenlabels als Pointerpaare darzustellen und in jedem Knoten die Länge des durch ihn repräsentierten Infixes zu speichern. Damit entspricht sie auch den Pseudocode-Darstellungen aus [42, 41]. Da eine SCDAWG-Struktur grundsätzlich stets mehr Kanten als Knoten besitzt, kann als eine speichersparende Alternative auch pro Kante nur die Länge ihres Labels gespeichert werden und in den Knoten Start- und Endposition der durch sie jeweils repräsentierten Infixe [1]. Die Startposition eines Kantenlabels ließe sich damit durch den jeweiligen Übergangsbuchstaben sowie den Repräsentanten des Ausgangsknotens berechnen. Dies wäre aber nur zur Darstellung der Kantenlabels nötig und würde für die eigentliche Berechnung der Indexstruktur keine Rolle spielen.

Wie in Kapitel 5.2 gezeigt, kann, wenn keine direkte on-line Konstruktion benötigt wird, auch zunächst eine CDAWG-Struktur nach dem Algorithmus aus [42] erstellt werden und in einem nachgeschalteten Schritt durch Algorithmus 12 die CDAWG-Struktur in eine SCDAWG-Struktur umgewandelt werden. Dabei ist jedoch zu beachten, dass zusätzlich Suffixlinks von den Sinkknoten aus zu den jeweils aktiven Knoten angelegt werden. Diese werden dann in Algorithmus 12 dazu verwendet, entsprechende Linksübergänge anzulegen, die ohne diese Suffixlinks fehlen würden.

Nach der on-line oder off-line Konstruktion einer SCDAWG-Struktur werden die in Kapitel 6 gezeigten Suchfunktionen implementiert. Hierdurch entsteht die Möglichkeit, die SCDAWG-Struktur zunächst als invertierte Datei zu benutzen, die es so in linearer Zeit proportional zur Länge des Suchstrings ermöglicht, das längste Präfix (siehe Algorithmus 13), die Häufigkeiten jedes Teilwortes oder aber die Vorkommenspositionen dieser (siehe Algorithmus 16) zu bestimmen. Zur Bestimmung aller Häufigkeiten wird jeder Knoten der SCDAWG-Struktur mit einem Frequenzlabel versehen (siehe Algorithmus 14). Diese Funktion dient im späteren Verlauf als Vorbild einer Annotationsfunktion, die etwa die Vorkommen bestimmter Metainformationen in jedem Knoten speichert, oder auch zur Annotation anderer globaler Kenngrößen, wie der document-frequency, jedes Knotens genutzt werden kann.

[1] Der Hinweis auf diese speichereffizientere Variante wurde von Dr. Stefan Gerdjikov gegeben.

Teil III.

Textuelle Gemeinsamkeiten

8. Identifikation längster gemeinsamer Teilwörter

Eine SCDAWG-Struktur S_C über einer beliebigen Anzahl von Eingabestrings W erlaubt es, alle Präfixe, Infixe und Suffixe der indexierten Dokumente in linearer Zeit durch Traversierung zu erkennen (siehe Algorithmus 13). Ähnlich wie bei der Berechnung des längsten gemeinsamen Teilwortes zwischen zwei Wörtern (siehe Kapitel 2.2) stellen aber, ohne dass ein Suchpattern angegeben wird, gemeinsame Teilwörter mehrerer Texte oft interessante Regionen innerhalb der Elemente von W dar.

8.1. Quasimaximale Knoten

Wird eine SCDAWG-Struktur über einer Menge von Wörtern oder Texten W aufgebaut, so enthalten ihre Knoten Infixe, die sowohl in einem Wort mehrmals oder in mehreren Wörtern aus W auftreten. In vielen Anwendungen wie etwa der Erkennung von Text-Reuse ist es sinnvoll, möglichst lange Matches zu erkennen, anhand derer dann Aussagen über Ähnlichkeiten innerhalb einer Teilmenge von W getroffen werden können. Dabei könnten entweder genau solche Teilwörter gefunden werden, die in *allen* Elementen von W auftreten, oder aber auch in *beliebigen Teilmengen* von W vorkommen. In jedem Fall müsste ein solches Teilwort in der Region seines Auftretens beziehungsweise in seinem jeweiligen Kontext *maximal* unter dem Aspekt sein, sodass es selbst nicht wieder innerhalb eines längeren derartigen Teilwortes enthalten ist.

8.1.1. Maximale Teilwörter in allen Texten

Um sich der Thematik des Auffindens maximaler Teilwörter anzunähern, werden zuerst vereinfachte Fälle, die ebenfalls interessante Teilprobleme darstellen, betrachtet.

Zum Auffinden aller maximalen Teilwörter, die in allen Wörtern aus W vorkommen, werden in S_C diejenigen Knoten gesucht, die über mindestens einen direkten

rechten Übergang auf jeden Sinkknoten aus V_W verfügen und keine rechten Übergänge auf andere Knoten aufweisen oder jeweils eine linke Kante auf jeden Sinkknoten haben. Da sich ein solcher Knoten, der ein längstes gemeinsames Teilwort darstellt, sozusagen in der „tiefsten“ Ebene in S_C vor den Sinkknoten befindet, wird er künftig als *quasimaximal* bezeichnet und die Menge, die jeden quasimaximalen Knoten enthält, als Q_{max}. Formalisiert ließe sich diese erste Idee wie folgt ausdrücken:

Definition 8.1.1 (Quasimaximale Knoten - Vorkommen in allen Wörtern aus W)**.** Ein Knoten $v \in V$ ist in der Menge Q_{max} enthalten, wenn:

$\forall v' \in V_W$ mit $\alpha \in \Sigma^*$ gilt: $(v, \alpha, v') \in E_R^{v\rightarrow}$ und $\nexists\ (v, \alpha, v') \in E_R^{v\rightarrow}$ mit $v' \notin V_W$, oder falls $v = \beta\$, \beta \in \Sigma^*$: $\forall v' \in V_W$ gilt: $(v, \alpha, v') \in E_L^{v\rightarrow}$.

Beispiel 8.1.1. Sei $W = \{\#1abc2def3\$, \#4abc5def6\$, \#7abc8def9\$\}$.

Für die Wörter w_1,w_2 und w_3 ließen sich nach Definiton 8.1.1 die beiden Knoten *abc* und *def* sowie die Start- und Abschlussknoten # und \$ der Menge Q_{max} zuordnen. Die folgende Illustration zeigt einen Ausschnitt aus der zugehörigen SCDAWG-Struktur S_C.

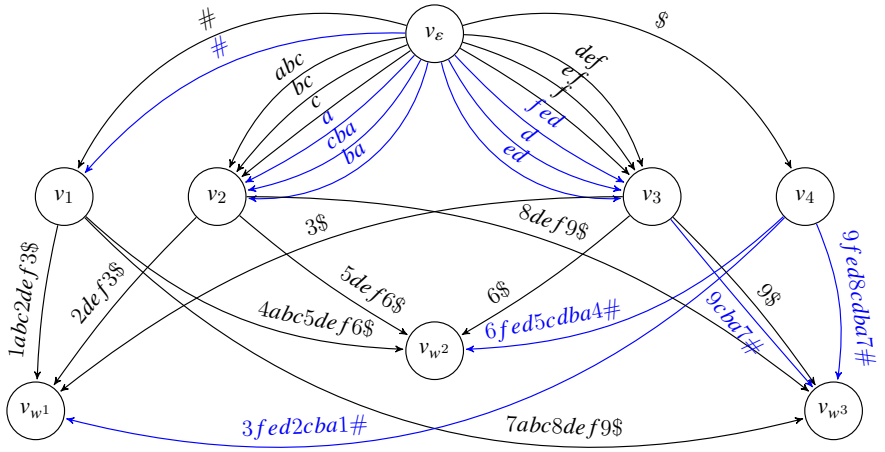


Abbildung 8.1.: Ausschnitt aus SCDAWG S_C mit quasimaximalen Knoten v_1, v_2, v_3 und v_4.

Die drei Knoten $v1, v2$ und $v3$, die die Infixe *abc*, *def* und # darstellen, besitzen jeweils 3 rechte Übergänge, von denen jeder auf einen der drei Sinkknoten von S_C führt. Der Abschlussknoten \$ hingegen hat drei linke Kanten (blau), von

denen ebenfalls jede zu einem Sinkknoten führt. Im Beispiel sind alle Knoten, die nicht in V_W sind beziehungsweise den Wurzelknoten v_ε darstellen, quasimaximal und somit in Q_{max}. Offensichtlich besitzt jedes gemeinsame Teilwort, das in der Menge der Eingabewörter gefunden werden kann, in diesem Beispiel keine gemeinsamen Zeichen mit anderen gefundenen Teilwörtern. Diese Annahme stellt zwar eine starke Vereinfachung dar, die unrealistisch für beliebige Eingabetexte ist, nichtsdestotrotz bietet sie einen guten Startpunkt, der nun erweitert werden kann, um zukünftig auch Teilwörter zu finden, die in mehrelementigen Teilmengen aus W vorkommen und/oder gemeinsame Alphabetzeichen aufweisen.

8.1.2. Maximale Teilwörter in mindestens zwei Texten

Um maximale gemeinsame Teilwörter zu identifizieren, die in mindestens zwei Texten von W vorkommen, können genau diejenigen Knoten als quasimaximal angesehen werden, deren direkte rechte und linke Nachfolger alle nur in genau einem Wort aus W vorkommen.

Definition 8.1.2 (Quasimaximale Knoten - Vorkommen in mindestens zwei Texten aus W)**.** Sei V_m die Menge der Knoten, die in mehr als einem Wort aus W auftreten[1]. Ein Knoten $v \in V$ ist in der Menge Q_{max} enthalten, wenn:

$$v \in V_m \text{ und } \forall v' \in V \text{ mit } (v, \alpha, v') \in E_R^{v\rightarrow} \text{ oder } (v, \alpha, v') \in E_L^{v\rightarrow} \text{ gilt: } v' \notin V_m.$$

8.1.3. Maximale Teilwörter im Kontext ihres Vorkommens

Mit Definition 8.1.2 können nun maximale Teilwörter in mehrelementigen Teilmengen von W identifiziert werden, jedoch ist der eingangs erwähnte Kontext des Auftretens eines maximalen gemeinsamen Teilwortes nicht berücksichtigt. Das nächste Beispiel verdeutlicht diese Problematik.

Beispiel 8.1.2. Sei $W = \{\#1abc2ab3\$, \#4abc5ab6\$, \#7abc8ef\$\}$.

Wenn die Eingabestrings aus Beispiel 8.1.1 nun leicht abgeändert werden, würde nach Definition 8.1.2 der Knoten, der das Infix ab beinhaltet, nicht zu Q_{max} gehören, obwohl die jeweils zweiten Vorkommen von ab in den ersten beiden Wörtern aus W in zwei unterschiedlichen unmittelbaren linken und rechten Kontexten ein längstes gemeinsames Teilwort darstellen. Ein erneuter Blick auf die SCDAWG-Struktur S_C bestätigt, dass $v_2 = ab$ nicht als quasimaximaler Knoten erkannt werden würde, da dieser Knoten ein Präfix des Knotens $v_3 = abc$ ist und somit eine

[1]Die Berechnung von V_m kann als Variante von Algorithmus 14 implementiert werden.

rechte Kante (v_2, c, v_3) existiert, wobei v_3 selbst nicht nur in einem Wort aus W auftritt.

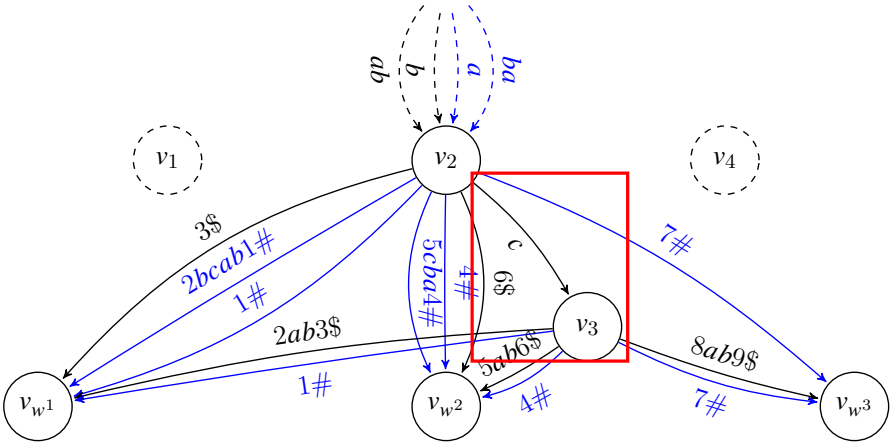


Abbildung 8.2.: Ausschnitt aus SCDAWG S_C, der eine rechte Kante mit dem Übergangslabel c zwischen den Knoten v_2 und v_3 zeigt.

Beispiel 8.1.2 zeigt damit, dass auch die Überlegung, die Definition 8.1.2 zugrundeliegt noch nicht weit genug geht, da durch diese das Problem von Vorkommen längster gemeinsamer Teilwörter, die gemeinsame Alphabetzeichen enthalten, nicht gelöst wird. Zu Beginn wurde bereits festgestellt, dass es aufgrund der großen Knotenmenge, die in S_C entsteht, falsch wäre, alle Knoten als gemeinsame Teilwörter zu erkennen, die ein Präfix oder Suffix eines längeren Knotens repräsentieren. Im Beispiel müssten jedoch v_2 und v_3 zur Menge Q_{max} gehören. Es müsste aber gleichzeitig festgehalten werden, dass in den Wörtern w^1 und w^2 nur das jeweils zweite Vorkommen von ab, das in beiden Wörtern an Position 7 beginnt und an Position 8 endet, als gemeinsames längstes Teilwort gewertet werden darf. Das erste Vorkommen, das die Positionen 3 und 4 besitzt, dürfte nicht aufgenommen werden, da es vollends in dem größeren Teilwort abc enthalten ist.

Bestimmung von Übergangspaaren

Wie lässt sich unterscheiden, dass das erste Vorkommen von ab nicht zu Q_{max} zählen darf, das zweite aber schon? Eine mögliche Lösung dafür liegt in den Abschlusseigenschaften, die sich aus der Bidirektionalität der SCDAWG-Struktur ergeben. Wenn immer ein eindeutiges Start und- Abschlusszeichen (hier stets # und $) den Eingabewörtern am Anfang und am Ende zugefügt wird, gilt, dass es von einem Knoten $v \in V$, der selbst nicht $ oder # enthält, sowohl eine rechte ausgehen-

de Kante $(v, \alpha, v_{w^i}) \in E_R^{v\rightarrow}$ mit $\alpha \in \Sigma^*$ als auch eine linke Kante $(v, \beta, v_{w^i}) \in E_L^{v\rightarrow}$ mit $\beta \in \Sigma^*$ gibt. Das Vorkommen eines Knotens, das an einer bestimmten Stelle in einem Eingabewort aus W ein längstes Infix darstellt, wird damit durch ein eindeutiges *Übergangspaar* aus einer rechten und einer *zugehörigen* linken Kante, die auf denselben Sinkknoten führen, eingegrenzt. *Zugehörig* bedeutet, dass die gesuchte linke Kante mit dem Zeichen auf v_{w^i} zeigt, das sich am linken Rand des durch die rechte Kante erweiterten Infixes von v befindet.

Dadurch werden diejenigen Vorkommen zu Q_{max} gezählt, die in ihrem jeweiligen Kontext maximal sind und somit dort nicht in einem längeren Teilwort vorkommen. Ein Knoten, der ein solches Übergangspaar besitzt, wird nun als *bedingt quasimaximal* bezeichnet, da er nur in bestimmten Kontexten ein längstes gemeinsames Infix darstellt, in anderen Kontexten jedoch nicht.

Bei der Suche nach den Übergangspaaren bedingt quasimaximaler Knoten kann von einem Knoten $v \in V$ aus folgendermaßen vorgegangen werden:

1. Suche nach rechten Erweiterungen von v auf einen Sinkknoten v_{w^i}.

2. Wird eine solche gefunden, ermittle das Zeichen σ_l, das den linken Rand des durch die rechte Kante erweiterten Infixes v darstellt.

3. Finde eine linke Kante, die mit σ_l auf v_{w^i} zeigt.

4. Wenn von v eine linke Kante gefunden wird, die mit σ_l auf v_{w^i} zeigt, liegt v mit seiner Endposition in Q_{max}.

Die Ermittlung des gesuchten Übergangszeichens σ_l kann mit Hilfe des Endpointers der rechten Kante und der Länge des Knotens v durchgeführt werden. Ein erneuter Blick auf Abbildung 8.2 zeigt, dass dort insgesamt fünf Übergangspaare existieren, die jeweils zwei Vorkommen von ab und drei Vorkommen von abc innerhalb von W identifizieren, ohne dass die Positionen mitgezählt würden, die das innerhalb von abc enthaltene ab betreffen. Die nächste Abbildung zeigt eine „vergrößerte“ Darstellung des Knotens $v2 = ab$ und des Sinkknotens $v_{w^2} = \#4abc5ab6\$$ mit ihren rechten und linken Übergängen.

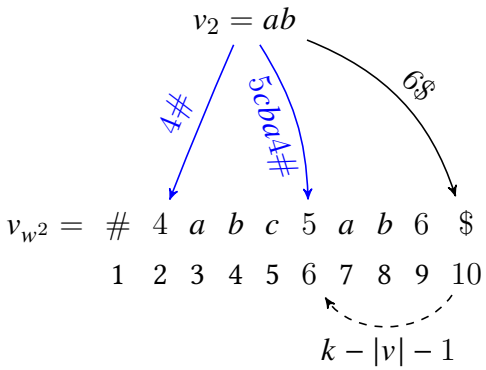


Abbildung 8.3.: Ausschnitt aus SCDAWG S_C, der zeigt, wie das zweite Vorkommen von ab im zweiten Beispielwort an Endposition 8 durch das Übergangspaar $((v_2, 6\$, v_{w^2}), (v_2, 5cba4\#, v_{w^2}))$ bedingt quasimaximal wird.

Anhand dieser Beobachtung lässt sich nun eine genauere Definition für einen bedingt quasimaximalen Knoten $v \in V$ angeben:

Definition 8.1.3 (Bedingt quasimaximale Knoten - Berücksichtigung des Kontexts)**.** Ein Knoten v ist in der Menge der gemeinsamen maximalen Teilwörter Q_{max} enthalten, wenn für v gilt:

1. $v \in V_m$,
2. $\exists (v, (k, p), v_{w^i}) \in E_R^{v\rightarrow}$,
3. $\sigma_l := w^i_{k-|v|-1}$,
4. $\exists (v, \sigma_l \alpha, v_{w^i}) \in E_L^{v\rightarrow}$.

Wie in Definition 8.1.2 ist v in der Menge V_m der Knoten, die in mehreren Wörtern aus W enthalten sind. Es muss nach (2.) eine rechte Kante, die von v auf einen beliebigen Sinkknoten v_{w^i} führt, geben. Nach (3.) ist σ_l nun das Zeichen aus Σ, das in v_{w^i} an der Position steht, die sich durch die Differenz des Startpointers der rechten Kante und der Länge von v abzüglich 1 ergibt. Existiert von v aus eine linke Kante mit dem Übergangszeichen σ_l und führt diese Kante wiederum auf v_{w^i}, ist die Definition erfüllt und v liegt in Q_{max}.

Zusätzlich zu jedem so gefundenen Knoten $v \in Q_{max}$ muss die Endposition gespeichert werden, sodass Q_{max} jeweils Tupel der Form (v, i, j) beinhaltet, die für jeden i-ten Eingabestring und einen Knoten die Endposition des durch Definition 8.1.3 beschriebenen gemeinsamen Teilwortes liefert.

8.1.4. Eindeutige Ketten

Wie das nächste Beispiel zeigen soll, reicht aber auch das so beschriebene Verfahren mit Definition 8.1.3 nicht aus, um alle gemeinsamen maximalen Teilwörter zwischen einer endlichen Menge von Eingabewörtern aufzuspüren.

Beispiel 8.1.3. Sei $W = \{\#1b2aaaaaa3\$, \#4bbbbbb5a6\$\}$.

In Beispiel 8.1.3 werden nur jeweils das b an Position 3 im ersten Wort und das Vorkommen von a an Position 9 im zweiten Wort durch die Bedingungen von Definition 8.1.3 als quasimaximale Positionen erkannt. Obwohl es sich bei a und b um gemeinsame Teilwörter beider Eingabewörter handelt, wird keines der Vorkommen, das sich innerhalb der Ketten von a's und b's befindet, gefunden.

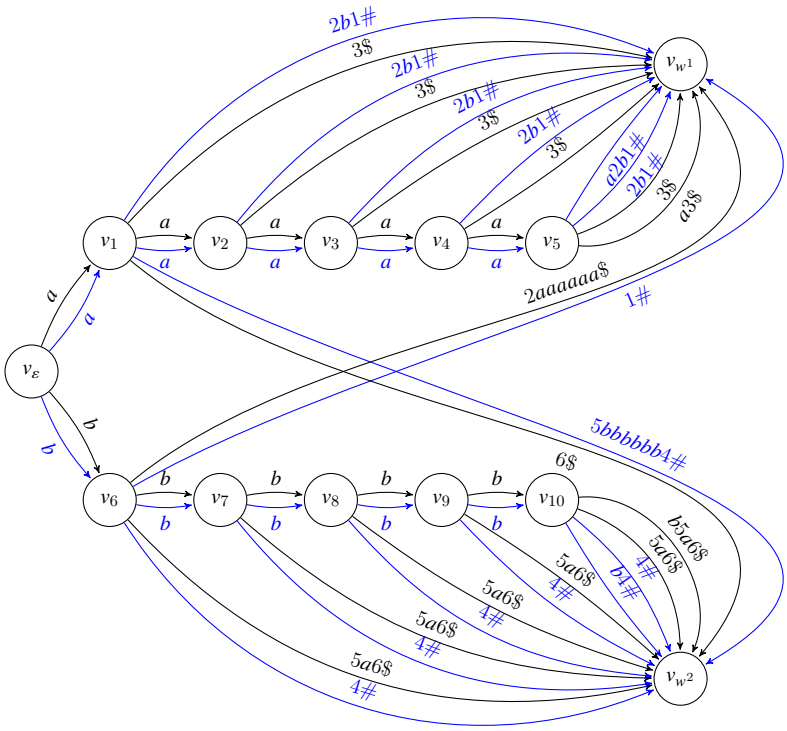


Abbildung 8.4.: Ausschnitt aus S_C für $W = \{\#1b2aaaaaa3\$, \#4bbbbbb5a6\$\}$.

Der Grund dafür ist, dass Bedingung (4.) aus Definition 8.1.3 verletzt wird. Jedes a oder b, das innerhalb einer eindeutig in einem der beiden Wörter auftretenden Kette von a's und b's vorkommt, besitzt nun das gleiche Übergangspaar,

das auf den linken und rechten Rand der Kette zeigt. Um trotzdem jedes a und jedes b innerhalb einer Kette als Vorkommen eines gemeinsamen Teilwortes aufzuspüren, kann von jedem Knoten v, der in mehreren Eingabewörtern vorkommt, eine Depth-First-Suche gestartet werden. Diese DFS läuft solange, bis ein Sinkknoten v_{w^i} erreicht wird, und sie verfolgt nur eindeutige Pfade, also Knoten die in nur einem der Eingabewörter auftreten. Für jeden Knoten, der so besucht wird, kann dann dessen zugehöriger Linkskontext, der ja durch Wiederholungen von sich selbst nach links verschoben worden ist, aus seiner Endposition abzüglich der Rekursionstiefe beziehungsweise der Länge des bisher durchlaufenen Pfads in jedem Schritt der DFS berechnet werden. Damit ist die Bedingung (4.) von Definition 8.1.3 wieder erfüllt und zu jedem rechten Übergang wird eine zugehörige linke Kante simuliert. Die Pfadlänge in die Berechnung mit aufzunehmen, ändert für jeden Knoten v, der in mehr als einem Eingabewort auftritt, nichts, da wenn von ihm ausgehend keine eindeutige Kette startet, diese 0 beträgt und somit keinerlei Auswirkungen hat.

8.1.5. Implementierung

Zur Bestimmung, ob ein Knoten in einem oder mehreren Eingabewörtern vorkommt, wird eine Variante von Algorithmus 14 angenommen, die als $get\_V_m()$ bezeichnet wird und die Menge der in mehreren Wörtern aus W auftretenden Knoten V_m liefert. Hieraus ergibt sich für das Verfahren zur Identifikation gemeinsamer längster Teilwörter innerhalb einer Menge von Eingabewörtern nun ein Grobentwurf.

1. Bestimme die Menge V_m aller Knoten, die in mehr als einem Eingabewort auftreten.
2. Für jeden Knoten $v \in V_m$ wende Definition 8.1.3 an.
3. Für jedes Kind $v' \notin V_m$ von v starte eine DFS und wende Definition 8.1.3 an, bis ein Sinkknoten erreicht wird.

Die Idee dieser Skizze ist es, alle Definition 8.1.3 entsprechenden Knoten als gemeinsame längste Teilwörter zu finden. Dazu werden zunächst alle Knoten markiert, die Vorkommen in mehreren Wörtern aus W haben. Jedes rechte Kind eines Knotens aus V_m muss anschließend auf das Vorkommen in einer eindeutigen Knotenkette untersucht werden, was mittels Rekursion erledigt werden kann, die sich solange aufrufen muss, bis das Ende jeder Kette, beziehungsweise ein Sinkknoten erreicht ist. Es folgt die formale Beschreibung des Algorithmus zum Auffinden der quasimaximalen Knoten in Pseudocode.

Algorithmus 17 Algorithmus zur Identifikation längster gemeinsamer Teilwörter

Vorbedingung: Die SCDAWG-Struktur $S_C = (V, E_R, E_L)$ einer endlichen Menge von Zeichenketten W mit $W = \{w^1, \ldots, w^i, \ldots, w^n\}$ und $w^i \in \Sigma^*$.

```
1:  function find_longest_common_substrings()
2:      result ← {}
3:      V_m ←get_V_m()                          ▷ Variante von Algorithmus 14
4:      for each v ∈ DFS(v_ε, false) do          ▷ DFS in Präordnung
5:          if v ∉ V_m then
6:              continue
7:          end if
8:          if # in v then                       ▷ Spezialfall für #-Knoten
9:              for each (v, (k, p), v') ∈ E_R^{v→} do
10:                 if v' ∈ V_W then
11:                     result.push((v,index_of(v, V_W), p))
12:                     continue
13:                 end if
14:             end for
15:         end if
16:         if $ in v then                       ▷ Spezialfall für $-Knoten
17:             for each (v, (k, p), v') ∈ E_L^{v→} do
18:                 if v' ∈ V_W then
19:                     result.push((v,index_of(v', V_W), p))
20:                     continue
21:                 end if
22:             end for
23:         end if
24:         find_longest_common_substrings_util(v, v, result, V_m, 0)
25:     end for
26:     return result
27: end function
```

Die Funktion ruft zuerst $get\_V_m()$ auf und startet dann eine DFS von v_ε aus. Für jeden Knoten wird überprüft, ob dieser in mehreren Eingabewörtern auftritt. Ist das nicht der Fall, wird die DFS weitergeschaltet. Für die Knoten, die den Startmarker # und Endmarker $ enthalten, werden der Ergebnisliste *result* die entsprechenden Tupel hinzugefügt, wobei deren Endpositionen anhand ihrer linken oder rechten Sinkübergänge bestimmt werden. Wenn ein Knoten in mehreren Strings auftritt und es sich nicht um Start- und oder Endknoten handelt, wird mit diesem die Hilfsfunktion *find_longest_common_substrings_util* aufgerufen.

Algorithmus 18 Hilfsfunktion zur Identifikation gemeinsamer Teilwörter

```
1:  procedure find_longest_common_substrings_util(v_s, v_t, Q_max, V_m, p_l)
2:      for each (v_t, (k, p), v') ∈ E_R^{v→} do
3:          if v' ∈ V_W then
4:              l ← k − |v|−1
5:              if (v_s, (l, p'), v'') ∈ E_L^{v_s→} then        ▷ Test, ob l-Kante existiert
6:                  if v'' ∉ V_m then
7:                      Q_max.push((v_s, index_of(v', V_W), k − p_l))
8:                  end if
9:              end if
10:         else if v' ∉ V_m then
11:             find_longest_common_substrings_util(v_s, v', Q_max, V_m, pl+p−k+1)
12:         end if
13:     end for
14: end procedure
```

Die Prozedur wird mit zwei Knoten v_s, v_t, die beim ersten Aufruf identisch sind, sowie der Tupelliste Q_{max}, V_m und einer Variablen für die Länge des gewanderten Pfades p_l aufgerufen. Zunächst wird anhand der rechten Kanten von v_t nach direkten Übergangspaaren gesucht, die Definition 8.1.3 entsprechen. Hierzu wird zuerst überprüft, ob der Zielknoten v' einer rechten Kante von v_t ein Sinkknoten ist. In Zeile 4 wird nun die in Abbildung 8.3 illustrierte Berechnung zur Bestimmung der Position l des linken Übergangszeichens durchgeführt. Existiert ein linker Übergang von v_s, der mit l startet und dessen Zielknoten v'' **nicht** in V_m enthalten ist (siehe noch einmal Abbildung 8.4), so erhält Q_{max} einen neuen Eintrag mit v_s. Die Endposition bestimmt sich durch k abzüglich der Pfadlänge, die 0 ist, insofern es sich bei v' direkt um einen Sinkknoten handelt. Wird von einem Knoten v_s ein Kind v' gefunden, das nicht in V_m enthalten ist, also nur in einem Eingabestring vorkommt, und kein Sinkknoten ist, ruft sich *find_longest_common_substrings_util* selbst solange mit v_s und v' auf, bis der Kindknoten v' wieder einen Sinkknoten darstellt.

Wird *get_V_m*() benutzt, um im Vorfeld die Knotenmenge V_m als Kandidaten für gemeinsame Regionen auszuwählen, steigt die Laufzeitkomplexität insgesamt um die einer weiteren Tiefensuche. Die Laufzeit von *find_longest_common_substrings* liegt damit bei der zweier Tiefensuchen zuzüglich der rekursiven Aufrufe, die in der Hilfsfunktion stattfinden. Jede DFS, die von einem Knoten $v \in V_m$ aus startet, muss aber nur einmal durchgeführt werden, da es für eine eindeutige Kette nur einen Knoten gibt, von dem aus sie gestartet werden kann. Somit kann die Anzahl der rekursiven Aufrufe nicht über der einer weiteren Tiefensuche liegen, was

bedeutet, dass die insgesamte Zeitkomplexität $> O(3|V|+|E|)$ ist, da die Struktur mindestens zweimal komplett durchsucht wird, sie aber nicht öfter als dreimal durchlaufen werden kann.

8.1.6. Beispiele für *find_longest_common_substrings*

Erneut werden die Ergebnislisten einiger Beispiele gezeigt, die bei der Anwendung von *find_longest_common_substrings* auf einem SCDAWG S_C über einer Menge von Wörtern W entstehen.

Beispiel 8.1.4. Für die drei Eingabewörter aus dem obigen Beispiel 8.1.2, $W = \{\#1abc2ab3\$, \#4abc5ab6\$, \#7abc8ab9\$\}$, werden folgende quasimaximale Knoten erkannt[2]

$Q_{max} = \{\langle \#, 1, 1\rangle, \langle \#, 2, 1\rangle, \langle \#, 3, 1\rangle, \langle abc, 1, 5\rangle, \langle abc, 2, 5\rangle, \langle abc, 3, 5\rangle, \langle ab, 1, 8\rangle, \langle ab, 2, 8\rangle, \langle ab, 3, 8\rangle, \langle \$, 1, 10\rangle, \langle \$, 2, 10\rangle, \langle \$, 3, 10\rangle\}$.

Als gemeinsame Teilwörter enthält Q_{max} für alle drei Eingabewörter *abc* an Endposition 5, *ab* an Endposition 8, sowie die Tupel, die die Start- und Endknoten für # und $ aufnehmen.

Beispiel 8.1.5. Analysiert man die beiden Eingabewörter aus Beispiel 8.1.3, $W = \{\#1b2aaaaaa3\$, \#4bbbbbb5a6\$\}$, werden folgende quasimaximale Knoten erkannt

$Q_{max} = \{\langle \#, 1, 1\rangle, \langle \#, 2, 1\rangle, \langle a, 1, 5\rangle, \langle a, 1, 6\rangle, \langle a, 2, 7\rangle, \langle a, 3, 8\rangle, \langle a, 1, 9\rangle, \langle a, 1, 10\rangle, \langle a, 2, 10\rangle, \langle b, 2, 3\rangle, \langle b, 2, 4\rangle, \langle b, 2, 5\rangle, \langle b, 2, 6\rangle, \langle b, 2, 7\rangle, \langle b, 2, 8\rangle, \langle b, 1, 3\rangle, \langle \$, 1, 12\rangle, \langle \$, 2, 12\rangle\}$.

Es werden somit alle Vorkommen von *a*'s und *b*'s gefunden, die sowohl in w^1 als auch in w^2 vorkommen, wobei die passende Endposition dieser Vorkommen innerhalb einer Knotenkette mithilfe der gewanderten Pfadlänge bestimmt wird.

Beispiel 8.1.6. Für die beiden Wörter aus Beipiel 2.2.1, die zur Illustration des längsten gemeinsamen Teilwortes zweier Wörter (LCF) benutzt wurden, ergibt sich nun folgendes Bild. Sei $W = \{\#ccabcdda\$, \#abcddddabc\$\}$, so ist die Menge der quasimaximalen Knoten:

$Q_{max} = \{\langle \#, 1, 1\rangle, \langle \#, 2, 1\rangle, \langle c, 2, 2\rangle, \langle c, 2, 3\rangle, \langle dd, 2, 7\rangle, \langle abcdd, 1, 8\rangle, \langle abcdd, 2, 6\rangle, \langle dda, 1, 9\rangle, \langle dda, 2, 9\rangle \ \langle abc, 2, 11\rangle \ \langle \$, 1, 10\rangle, \langle \$, 2, 12\rangle\}$.

[2] Anstelle jedes Knotennamens $v_1, v_2, ...v_n$ wird hier zum besseren Verständnis der Repräsentant jedes Knotens benutzt.

Es fällt auf, dass neben dem Knoten, der das längste gemeinsame Teilwort *abcdd* von w^1 und w^2 enthält, auch einige kürzere gemeinsame Teilwörter gefunden werden. Manche dieser Knoten werden durch *find_longest_common_substrings* nur innerhalb von einem der beiden Strings ausgegeben, da sie an anderen Positionen innerhalb eines anderen Wortes in einem längeren Knoten enthalten sind. Dies gilt zum Beispiel für das Infix *abc*, das insgesamt dreimal in beiden Wörtern auftritt, zweimal jedoch innerhalb des längeren Teilwortes *abcdd* vorkommt, sodass sich das einzig verbleibende „echte“ *abc* innerhalb von w^1 an Position 11 befindet. Zur Bestimmung des LCF zwischen w^1 und w^2 kann nun das Tupel ausgewählt werden, dessen Knoten den längsten Repräsentanten besitzt, beziehungsweise die Menge nach der Länge der Repräsentanten sortiert werden. Gleichzeitig lässt sich feststellen, dass es zwischen den gefundenen Teilwörtern auch *Überlappungen* geben kann. *dd*, das nur innerhalb von w^2 einen quasimaximalen Knoten darstellt, überlappt damit mit dem ersten *d* aus *dda*.

Ausblick

Das hier beschriebene Verfahren zur Identifikation gemeinsamer längster Teilwörter bildet die Basis des in Kapitel 9.1 diskutierten indexbasierten globalen Alignierungsverfahrens zweier Wörter w^1, w^2. Es ist jedoch offensichtlich, dass sich damit kein optimales Alignmentverfahren ergibt, da in manchen Fällen durch das Vorkommen eines gemeinsamen Teilwortes in einem anderen längeren Teilwort dieses „kürzere Teilwort“ nicht gefunden wird. Sollen zum Beispiel die beiden Strings

$$w^1 = \#abracadabrax\$$$
$$w^2 = \#abracadebray\$$$

aligniert werden, existiert mit $\#abracad$ (grün) ein gemeinsames Teilwort, das in w^1 und w^2 jeweils einmal auftritt. Innerhalb dieses Teilwortes befinden sich jedoch auch die Infixe *abra* und *bra*. *abra* hat in w^1 noch ein zweites Vorkommen (blau), weshalb es dort als ein gemeinsames Teilwort gefunden wird. In w^2 findet sich jedoch nur ein Vorkommen von *abra* (innerhalb von $\#abracad$), sodass dort *bra* als ein gemeinsames Teilwort gefunden wird. Um also *abra* (blau) und *bra* (rot) in Verbindung bringen zu können, wäre ein Refinement nötig, da verschiedene gemeinsame Teilwörter, auch wenn sie gemeinsame Zeichen enthalten oder sogar Suffixe voneinander sind, zunächst als disjunkt betrachtet werden.

8.2. Erkennung von lokalem Text-Reuse

Algorithmus 17 erkennt die Vorkommen aller längsten gemeinsamen Teilwörter, die innerhalb der Texte von W oder in Teilmengen von W auftreten. Diese Menge übereinstimmender Textfragmente kann benutzt werden, die *Wiederverwendung* oder den *Reuse* wiederkehrender Textabschnitte zwischen den Elementen von W ausfindig zu machen. Die Erkennung solcher wiederverwendeten Textabschnitte wird in vielen Anwendungen der Textanalyse gebraucht, wie etwa bei der Erkennung von Plagiaten oder aber der Identifikation von fast-Duplikaten innerhalb einer Textkollektion [67]. Letztere Aufgabe spielt etwa bei der Erstellung von Web-Suchmaschinen ein große Rolle, da es dort vermieden werden soll, identische Dokumente als Suchergebnis anzuzeigen. Die Wiederverwendung bestimmter Textfragmente stellt jedoch keineswegs ein neues Phänomen dar. Auch in historischen Dokumenten ist die Auffindung gleicher oder fast gleicher Textabschnitte von hohem Interesse. So kann die Verwendung von Zitaten, Sprichwörtern oder aufgrund von standardisiertem Textaufbau nahezu identischen Textbausteinen, die etwa in Rechtstexten oder Urkunden vorkommen, aufgezeigt werden. Ein weiteres Beispiel zur Erkennung von Text-Reuse in historischen Dokumenten ist das Finden von Nachdrucken und anderen textuellen Gemeinsamkeiten innerhalb von Zeitungen oder Zeitschriften. Diese Problematik wird zum Beispiel in [77, 68] behandelt, wobei hier eine OCR der historischen Zeitungskollektionen durchgeführt wurde, die zu weiteren Abweichungen zwischen identischen Passagen führte.

8.2.1. Maße zur Bestimmung von Textähnlichkeit

Die verschiedenen Arten, die zur Bestimmung der Ähnlichkeit von Texten existieren, lassen sich, wie von [4, 5] vorgeschlagen, in die drei Kategorien *inhaltlich*, *strukturell* und *stilistisch* unterteilen. Zur ersten Kategorie zählt eine Variante der in Kapitel 2.2 beschriebenen Berechnung des längsten gemeinsamen Teilwortes zweier Texte, die die Ähnlichkeit zweier Texte damit aufgrund des Textinhalts bestimmt. Dieses Vorgehen ist ebenfalls in [31] beschrieben und als *longest common substring measure* bekannt. Nachteilig ist hierbei, dass die Ähnlichkeit zweier Texte nur aufgrund exakter Übereinstimmung, die durch jede Abweichung beendet wird, bestimmt wird. Benutzt man anstelle des longest common substring measure die unter 2.5 beschriebene Methode zu Berechnung der Ähnlichkeit zweier Texte, besteht zwar diese Problematik nicht mehr, bei diesem Ansatz, der ja auch zur globalen Alignierung zweier Strings (siehe Kapitel 2.4) eingesetzt werden kann, wird aber eine lineare Ordnung für alle Übereinstimmungen vorausgesetzt. Für die Ähnlichkeitsbestimmung von Texten, die nur kleinere Sequenzen teilen, wäre ein solches Vorgehen somit ebenfalls nachteilig, da durch Änderungen in der Reihen-

folge der Übereinstimmungen diese nicht mehr als Teil der LCS erkannt werden würden. Aus diesem Grund werden für diese Aufgabe oft Verfahren eingesetzt, die wiederverwendete Textstellen *lokal* bestimmen und damit nicht auf eine bestimmte Reihenfolge festgelegt sind. Bekannte Verfahren wie [13] nutzen Chunks, also Substrings oder Subsequenzen als gemeinsame lokale textuelle Einheiten, die anschließend in eine numerische Repräsentation überführt werden, woraus sich eine Art *Fingerabrduck* eines jeden Textes generieren lässt [67]. Während diese oft bei der Plagiatserkennung benutzte Technik nicht im Fokus dieser Arbeit liegt, kann zur Bestimmung der Chunks, wie eingangs erwähnt, Algorithmus 17 benutzt werden. Wenn nur längste gemeinsame Teilwörter erkannt werden sollen, die in allen Wörtern aus W vorkommen, kann Q_{max} entweder im Nachhinein anhand dieses Kriteriums gefiltert werden, oder aber Algorithmus 17 wird dahingehend abgewandelt, dass er Definition 8.1.1 entspricht.

8.2.2. Visuelle Exploration von lokalem Text-Reuse

Die Menge textueller Gemeinsamkeiten einer Textkollektion kann auch zur Visualisierung der so entstandenen Relationen zwischen den Elementen eines Korpus verwendet werden. Hierfür werden nun Übersichtsgraphen zweier Korpora berechnet[3], die in Abbildung 8.5 abgebildet sind. Das erste Korpus enthält rund 20.000 deutsche Gedichte, die etwa eine Zeitspanne vom 17. bis zum 19. Jahrhundert abdecken[4]. Das zweite Korpus enthält die Liedtexte von circa 15.000 transkribierten Pop-Songs, die in den deutschen Single-Charts etwa ab Mitte der 1960er Jahre bis heute vertreten waren[5]. Die in den Übersichtsgraphen enthaltenen Knoten verbinden die gewonnenen quasimaximalen Knoten mit den im Korpus enthaltenen Dokumenten, die im Übersichtsgraphen ebenfalls als Knoten, die jeweils mit einer Textnummer versehen sind, dargestellt sind. So ergeben sich dichtere und weniger dichte Regionen, die von einer übergeordneten Perspektive einen Überblick über die Wiederverwendung lokaler Textfragmente im Korpus geben. Da ein solcher Übersichtsgraph sehr groß werden kann, können anschließend durch Heranzoomen bestimmte Teilbereiche genauer betrachtet werden und so interessante Regionen untersucht werden.

[3] Die hier dargestellten Ergebnisse wurden in Zusammenarbeit mit Dr. Stefan Gerdijkov gewonnen.

[4] Die verwendeten Daten entstammen dem Korpus des TextGrid Repositories [74].

[5] Dieses Korpus wurde von Stefanie Schneider zur Verfügung gestellt.

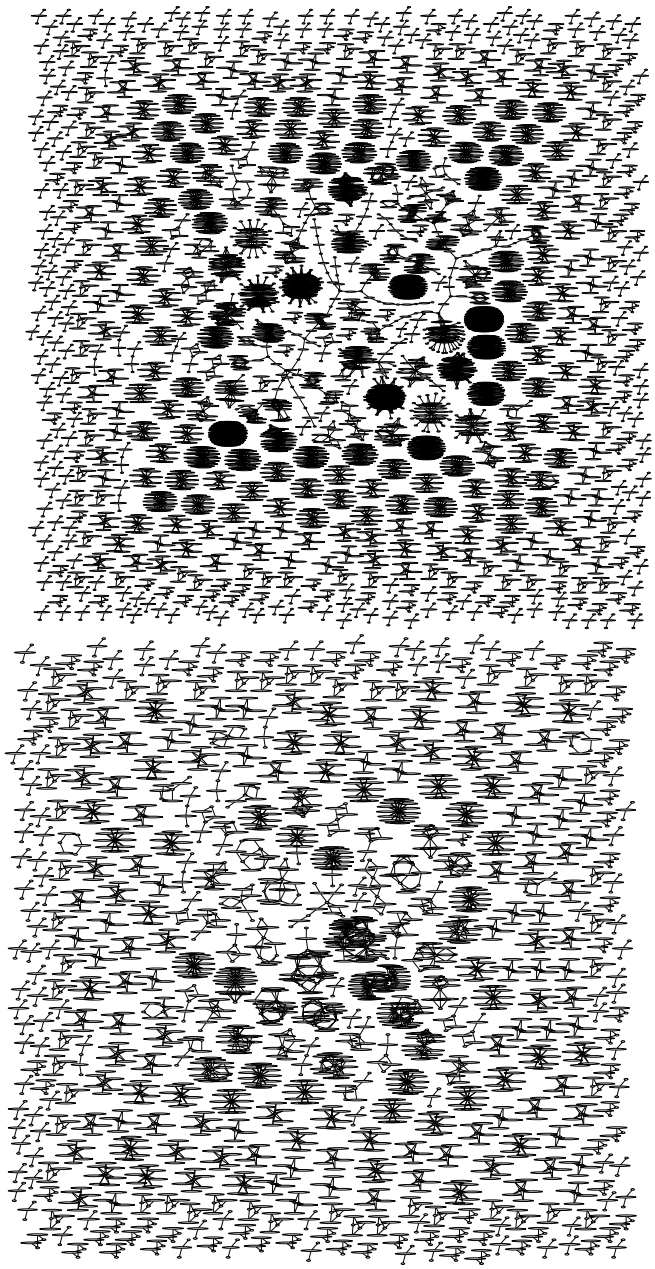


Abbildung 8.5.: Visualisierung von Text-Reuse innerhalb zweier Textkollektionen. Gedichte-Korpus oben, Liedtexte-Korpus unten.

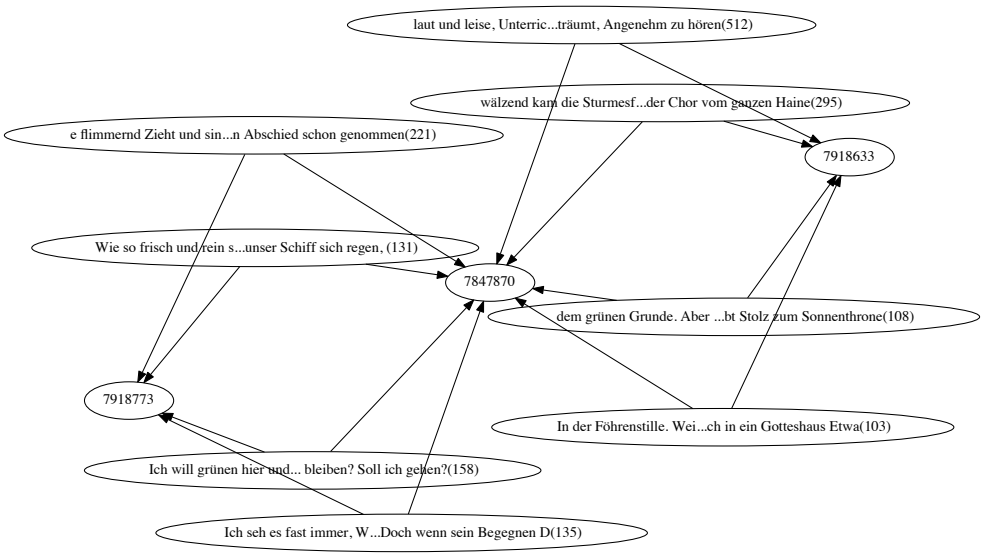


Abbildung 8.6.: Quasimaximale Knoten mit Pointern auf drei Gedichte - Vergrößerte Unterregion des oberen Graphen aus Abbildung 8.5.

In Abbildung 8.6 ist ein vergrößerter Ausschnitt des oberen Graphen aus Abbildung 8.5 zu sehen. Einige Beispiele für dort wiederverwendete Textpassagen sind:

„laut und leise, Unterric… träumt, Angenehme zu hören“

“wälzend kam die Sturmesf … der Chor vom ganzen Haine“

Innerhalb des Liedtexte-Korpus findet sich beispielsweise eine Unterregion, die einige Dokumente durch Phrasen verbindet, die das Wort „Hallelujah“ enthalten.

„She tied you to a kitche .. Hallelujah, Hallelujah“

„s seen the light It’s a – ah Hallelujah, Hallelujah“

Die dort verknüpften Dokumente stellen damit alle verschiedene Versionen von Leonard Cohens Ballade „Hallelujah“ dar, die vielfach (zum Beispiel von Jeff Buckley) gecovered wurde. Dichter verknüpfte Regionen des Liedtexte-Korpus verbinden damit fast immer verschiedene Versionen desselben Liedes und zeigen so, an welchen Stellen im Korpus Lieder von Künstlern neu interpretiert wurden. Die innerhalb der quasimaximalen Knoten angehängte Zahl gibt die Länge des jeweiligen Teilwortes an. Dadurch kann eine minimale Schranke festgelegt werden, anhand derer Dokumente verknüpft werden. In diesem Fall wurde eine Schranke von mindestens 50 Zeichen zugrundegelegt.

8.2.3. Hinzunahme von Metainformationen

Eine weitere Möglichkeit, Dokumente eines Korpus anhand von lokalem Text-Reuse zu vergleichen, ergibt sich, wenn zu jedem Dokument zusätzlich Metadaten berücksichtigt werden. Damit können etwa nun nicht mehr einzelne Dokumente miteinander verglichen werden, sondern Teilmengen von Dokumenten, die aufgrund von Metainformationen bestimmt werden. Beispielsweise können alle Dokumente eines Autors, eines Jahres oder eines Ortes so zusammengefasst und anhand ihrer gemeinsamen längsten Teilwörter mit anderen derartigen Gruppen verglichen werden.

Um eine SCDAWG-Struktur mit bestimmten Metainformationen anzureichern, wird diese (etwa der Autor eines Dokuments) zunächst in den jeweiligen Sinkknoten jedes Wortes aus W gespeichert. Die Annotation aller anderen Knoten erfolgt durch eine leicht abgewandelte Variante von Algorithmus 14. Ist jeder Knoten der SCDAWG-Struktur mit den zugehörigen Metainformationen angereichert, kann Algorithmus 17 so abgewandelt werden, dass zum Beispiel jeweils die längsten gemeinsamen Teilwörter zweier Autoren erkannt werden.

Abbildung 8.7 zeigt einen Übersichtsgraphen über gemeinsame Textwiederverwendung im Gedichte-Korpus, bei dem die Knoten nun jeden Autor repräsentieren und die Relationen zwischen den Knoten die Anzahl der gemeinsamen längsten Teilwörter bezüglich dieser darstellen. Die farbliche Einfärbung der Knoten repräsentiert dabei die zeitliche Dimension, wobei jeweils der Mittelpunkt der Lebensdaten jedes Autors gewählt wurde. Das Farbspekturm reicht dabei von rot über grün und blau bis lila von frühen Autoren des 17. Jahrhunderts zu späten des 19. Jahrhunderts.

Im unteren Teil des Graphen findet sich ein Pfad beginnend bei Anna Louisa Karsch (1756) (siehe Abbildung 8.7, ①) ⟶ Friedrich Gottlieb Klopstock (1763) ⟶ Ernst Schulze (1803) (siehe Abbildung 8.7, ②). Letzterer wird auch durch einen Pointer von Johann Wolfgang von Goethe (1794) erreicht. (siehe Abbildung 8.7, ③), wobei Goethe selbst wieder viele Nachfolger besitzt. In dieser Darstellung ist Ludwig Achim von Arnim (1806) (siehe Abbildung 8.7 , ④), eine noch zentralere Figur, die das Zentrum des größten Clusters bildet. Die temporale Farbinformation zeigt, dass sich in dieser Ansammlung hauptsächlich Autoren des 19. Jahrhunderts finden, während die des 17. und 18. Jahrhunderts den zweiten größeren Cluster im oberen rechten Teil bilden. Ein früher „einflussreicher“ Autor dieser Periode ist Simon Dach (1632) (siehe Abbildung 8.7, ⑤).

Die beiden großen Cluster weisen damit auf einen Wandel bezüglich der literarischen Konzepte hin, der sich zwischen Barock/Vorromantik und der Romantik vollzogen hat. Die drei Autoren (John Brinckman (1842), Fritz Reuter (1842) und Klaus Groth (1859) (siehe Abbildung 8.7, ⑥), die den Cluster unten rechts bilden,

sind nur untereinander verbunden, obwohl sie in die Zeit des Romantik-Clusters passen. Dies hat jedoch den Grund, dass alle drei Autoren ihre Werke auf Niederdeutsch verfasst haben und somit kaum textuelle Gemeinsamkeiten mit anderen zeitgenössischen Autoren besitzen.

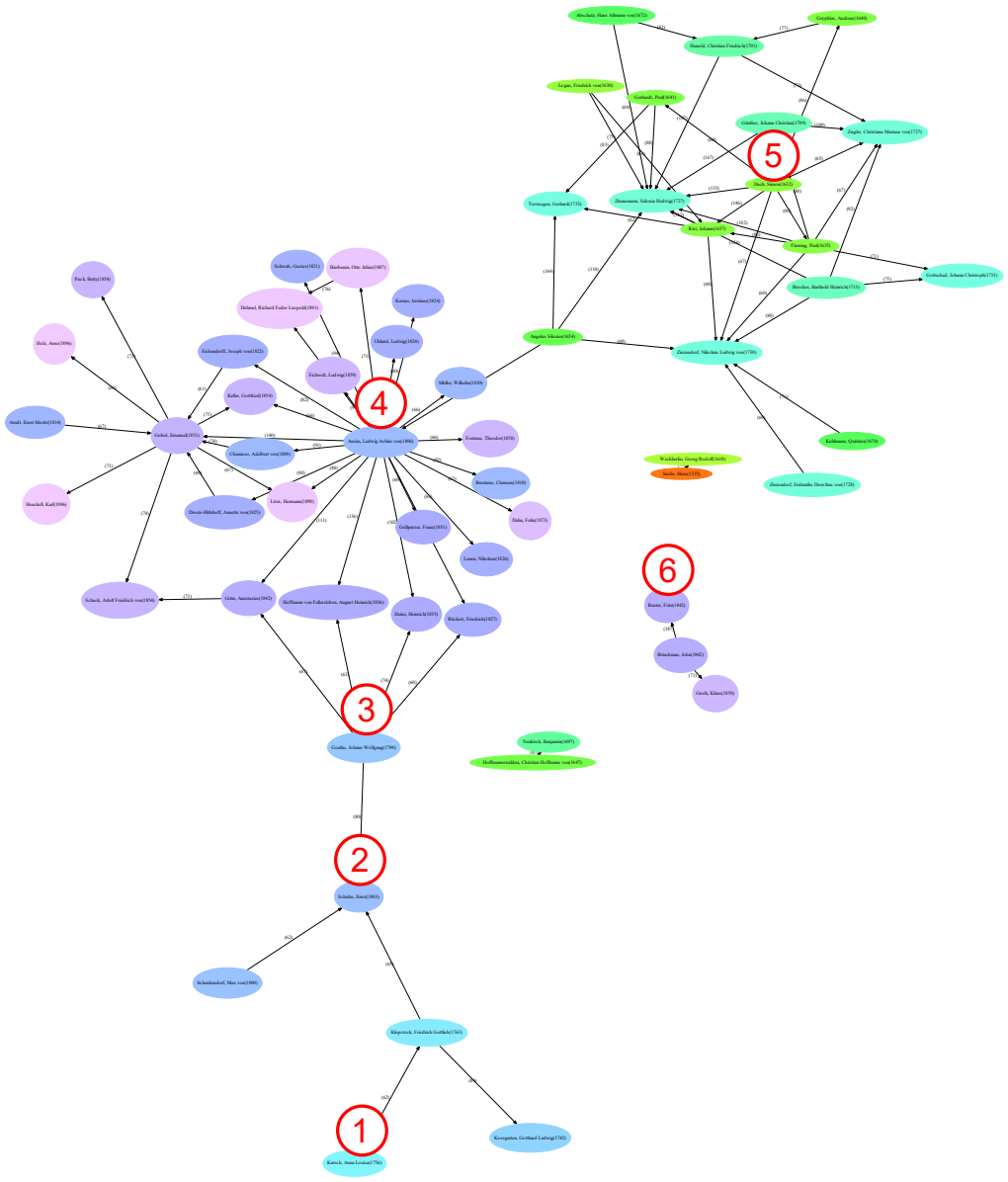


Abbildung 8.7.: Textuelle Gemeinsamkeiten von Autoren im Gedichte-Korpus. Die farbliche Einfärbung zeigt die temporale Dimension an.

8.3. Auffinden von Text in Textbildern

Die in Kapitel 8 beschriebene Prozedur zur Identifikation gemeinsamer längster Teilwörter soll nun zur Lösung eines bestimmten Problems zum Einsatz kommen. Die OCR historischer Drucke stellt eine große Herausforderung dar, da aufgrund des Alters der Dokumente, der verwendeten Fonts und Verschmutzungen spezielle Modelle trainiert werden müssen, um gute Erkennungsraten zu erzielen.

Zum Training solcher Modelle ist eine bestimmte Menge von *Ground Truth* nötig, anhand derer die Charakteristiken der zu erkennenden Dokumente erlernt werden. Bei einem solchen Training eines Modells zu einer OCR-Engine wird in der Regel zeilenweise vorgegangen, sodass für jede Bildzeile einer Seite eine entsprechende Ground-Truth-Zeile bereitsteht.

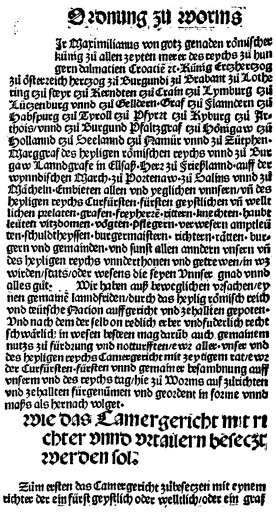
Die hier benutzten Ground-Truth-Daten[6], die im Zuge eines EU-Projekts [61, 38] zur Digitalisierung historischer Drucke entstanden sind, liegen zwar zeilenweise vor, sie enthalten anstelle von Zeilenkoordinaten aber nur seitenweise Koordinaten der transkribierten Dokumente im PAGE-XML Format [63] sowie Seitenbilder im tiff-Format. Damit die Daten als Trainingsmaterial eingesetzt werden können, müssen jedoch Zeilen-Koordinaten für entsprechende Zeilenbilder vorliegen. Um die Transkriptionen mit den entsprechenden Bildzeilen in Verbindung zu bringen, wird die in Kapitel 8 beschriebene Methode zur Identifikation gemeinsamer Teilwörter eingesetzt[7].

Zuerst wird mittels eines bereits vortrainierten Modells [71] eine OCR der vorhandenen Bildseiten durchgeführt. Die so erkannten Zeilen können nun anhand der längsten gemeinsamen Teilwörter, die sie mit den in den Ground-Truth-Seiten enthaltenen Zeilen teilen, einander zugeordnet werden. Über die so entstehende Zuordnung von Ground-Truth-Zeile und OCR-Zeile kann jede Ground-Truth- Zeile einem Zeilenbild beziehungsweise dessen Koordinaten zugeordnet werden, sodass diese dann zum Training verwendet werden können. Die Menge der im SCDAWG indexierten Wörter besteht damit aus zwei ungeordneten Mengen von Zeilen, in dem Zeilenpaare aus OCR und zugehöriger Ground-Truth gefunden werden sollen, deren Inhalt, etwa aufgrund von OCR-Fehlern, aber teils stark voneinander abweicht. Die folgenden Abbildungen zeigen anhand einer Beispielseite Ausschnitte aus Ground-Truth, Bildseiten sowie zugehöriger OCR-Ergebnisse.

[6] In diesem Experiment wurden 20 GT-Seiten des Werkes „Das Buch des heyligen Römischen Reichs unnderhalltunge“(1501) [21] verwendet.

[7] Das hier beschriebene Experiment wurde auf Initiative von Dr. Uwe Springmann durchgeführt.

```
        </Coords>
        <TextEquiv>
            <PlainText></PlainText>
            <Unicode>
            wie das Camergerit mit riter vnnd vrtailern ſbeeczt
            werden ſol.
            </Unicode>
        </TextEquiv>
    </TextRegion>
    <TextRegion id="r3" type="paragraph">
        <Coords>
            <Point x="785" y="2783"/>
            <Point x="2083" y="2783"/>
            <Point x="2083" y="2929"/>
            <Point x="785" y="2929"/>
        </Coords>
        <TextEquiv>
            <PlainText></PlainText>
            <Unicode>ů
            Zm eren das Camergerit ůſzbeeen mit eynem
            riter der ein für geyli oder wetli/oder ein graf
            </Unicode>
        </TextEquiv>
    </TextRegion>
```



Ordnung zu worms

Jr Maximilianus von gotz genaden römiſcher
künig zu allen zeyten merer des reychs zu hun
gern dalmatien Croatie ꝛc. Künig Erzhertzog
zu öſterreich hertzog zu Burgundi zu Brabant zu Rotha
ring czu ſteyre czu Kernden czu Crain czu Lymburg Gu
Luczenburg vnnd czu Gelldern Graf czu Flandern czu
Habſpurg czu Tyroll czu Pfyrt czu Kyburg czu Ar
thois vnnd czu Burgund Pfaltzgraf czu Hönigaw czu
Holland czu Seeland czu Namür vnnd zu Zütphen
Marggraf des heyligen römiſchen reychs vnnd zu Bur
gaw Landtgrafe in Ellſaß Herr zu Frießland auff der
wyndiſchen March zu Portenaw zu Salins vnnd zu
Mächeln Embieten allen vnd yeglichen vnſern vñ des
heyligen reychs Curfürſten fürſten geyſtlichen vñ welt
lichen prelaten grafen freyherren ritter knechten haupt
leüten vitzdomen vögten pflegern verweſern ampleü
ten ſchuldtheyſſen burgermaiſtern richtern rätten bur
gern vnd gemainden vnd ſunſt allen anderen vnſern vñ
des heyligen reychs vnderthonen vnd getrewen in wa
wirden ſtats oder weſens die ſeyen Unſer gnad vnnd
alles gůt. Wir haben auß bewegklichen vrſachen ey
nen gemainē landtfriden durch das heylig römiſch reich
vnd teütſche Nacion auffgericht vnd zehalten geboten
Vnd nach dem der ſelb on redlich erber vnd fuderlich recht
ſchwärlich in weſen beſtan mag darüb auch gemainem
nutz zu fürdrung vnd notturfften ewr alles vnſer vnd
des heyligen reychs Camergericht mit zeytigem rate ew
der Curfürſten fürſten vnnd gemainer beſtandung auff
vnſerm vnd des reychs tag hie zu Worms auf zurichten
vnd zehalten fürgenümen vnd geordent in forme vnnd
maß als hernach volget.

Wie das Camergericht mit rich
ter vnnd vrtailern beſetzt
werden ſol.

Zům erſten das Camergericht zubeſetzen mit eynem
richter der ein fürſt geyſtlich oder weltlich oder ein graf

Ordnung zu worms

Oronung n dMorms

Jr Maximilianus von gotz genaden römiſcher

Jr Waximiliaamms vwn gotz genaden rôniſther

künig zů allen zeyten merer des reychs zů hun

kunig ·ſ allen ze yttn merer des reyths zil hun

Abbildung 8.8.: Ausschnitt einer Ground-Truth-Seite im PAGE-XML Format mit ihrer zugehörigen Bildseite sowie Segmentierung der ersten drei Zeilen mit OCR-Ergebnissen aus dem IMPACT-Korpus.

Obwohl hier eine *Inkunabel*, also ein sehr frühes Druckwerk digitalisiert wurde, ist die Erkennungsrate der OCR so hoch, dass davon ausgegangen werden kann, dass innerhalb der GT-Zeilenpaare und der OCR-Zeilenpaare genug Übereinstimmungen existieren, die gemeinsame Teilwörter bilden. Alle GT-Zeilen sowie OCR-Zeilen werden pro Seite indexiert und dafür deren gemeinsame Teilwörter extra-

hiert. Die Knotenmenge wird nun zusätzlich darauf eingeschränkt, dass nur genau solche Teilwörter berücksichtigt werden, die exakt in zwei verschiedenen Wörtern (Zeilen) aus W vorkommen. Um dies zu erkennen, kann ähnlich wie bei der Bestimmung von Häufigkeiten (siehe Algorithmus 14) vorgegangen werden. Ordnet man diese Menge nun der Länge nach, können die längsten so erkannten Teilwörter pro Zeilenpaar gespeichert werden. Für die Beispielseite aus Abbildung 8.8 entsteht folgende Liste[8].

1. er weſens die ſeyen Vnnſer gnad vnnd$	19. en vnd yegli
2. en vnnd gemainer beſambnung a	20. ern vnſern v
3. arggraf des heyligen r	21. in weſen be
4. #gaw Lanndgrafe in E	22. enburg vnnd
5. #des heyligen rey□s	23.‚Wir haben
6. men vnd geordent in	24. ten gepoten
7. #vnſerm vnd des rey	25. #gern dalma
8. Burgund Pfaltzgra	26. Brabant z
9. erber vndfuderli	27. ů Lymburg
10. das Camergeri□t	28. ů Pfyrrt
11. #leüten vitzdomen	29. ter der e
12. n gotz genaden r	30. ür vnnd z
13. Salins vnnd zů$	31. burgermai
14. n merer des rey	32. freyherr
15. ung vnd nottur	33. gericht
16. #heyligen rey	34. ilern
17. das heylig r	34. orms$
18. ßs als herna	36. ſol.$

Tabelle 8.1.: Paarweise auftretende Teilwörter innerhalb der Beispielseite.

Für insgesamt 37 Zeilenpaare wurden 36 längste Teilwörter gefunden, die jeweils in genau zwei verschiedenen Zeilen auftreten. Werden anhand dieser längsten Übereinstimmungen die Zeilen einander zugeordnet, entsteht für die Beispielseite eine korrekte Zuordnung, bei der jeweils die richtige Ground-Truth-Zeile mit ihrer passenden OCR-Zeile verbunden wird. Für eines der 37 Zeilenpaare konnte allerdings kein eindeutiges gemeinsames Teilwort identifiziert werden, sodass dieses nicht in die Menge des Trainingsmaterials mit einfließen kann. In diesem Experiment wurde untersucht, wie gut sich die Zeilen von 20 Seiten Ground-Truth mit denen ihrer OCR verknüpfen lassen. Die nachstehende Tabelle zeigt in Spal-

[8]Bedingt durch fehlende Fonts erscheinen manche Teilwörter kürzer, als sie eigentlich sind.

ten 4 und 5, wie viele Zeilen der 20 Seiten einander zugeordnet werden konnten (*Zugeordnet*), und wie viele der Zuordnungen korrekt waren (*Richtig*).

Seite	*GT*	*OCR*	*Zugeordnet*	*Richtig*	$Zugeordnet_M$	$Richtig_M$
1	36	35	34	31	33	33
2	39	39	38	28	39	39
3	35	35	35	31	35	35
4	36	36	35	33	34	33
5	39	39	37	32	39	39
6	38	38	38	34	38	38
7	39	40	37	32	38	37
8	20	20	20	20	20	20
9	6	6	6	6	6	6
10	37	37	36	33	37	37
11	39	40	39	37	39	39
12	35	36	35	31	34	34
13	34	35	32	26	33	32
14	37	37	37	35	37	37
15	30	30	28	24	28	28
16	35	35	34	32	35	35
17	33	33	30	29	32	31
18	37	37	36	32	37	37
19	35	36	34	32	34	33
20	35	35	35	31	35	35
$\sum$	675	679	657	599	663	658

Tabelle 8.2.: Ergebnisse der Zuordnung GT- und OCR-Zeilen.

Von insgesamt 675 GT-Zeilen konnten demnach 599 richtigerweise einer passenden OCR-Zeile zugewiesen werden. Damit wurden etwa 88% richtig verknüpft und circa 12% falsch. Der Prozentanteil der Zeilenpaare, die zugeordnet wurden, beträgt 91%.

8.3.1. Hinzunahme von Metainformationen

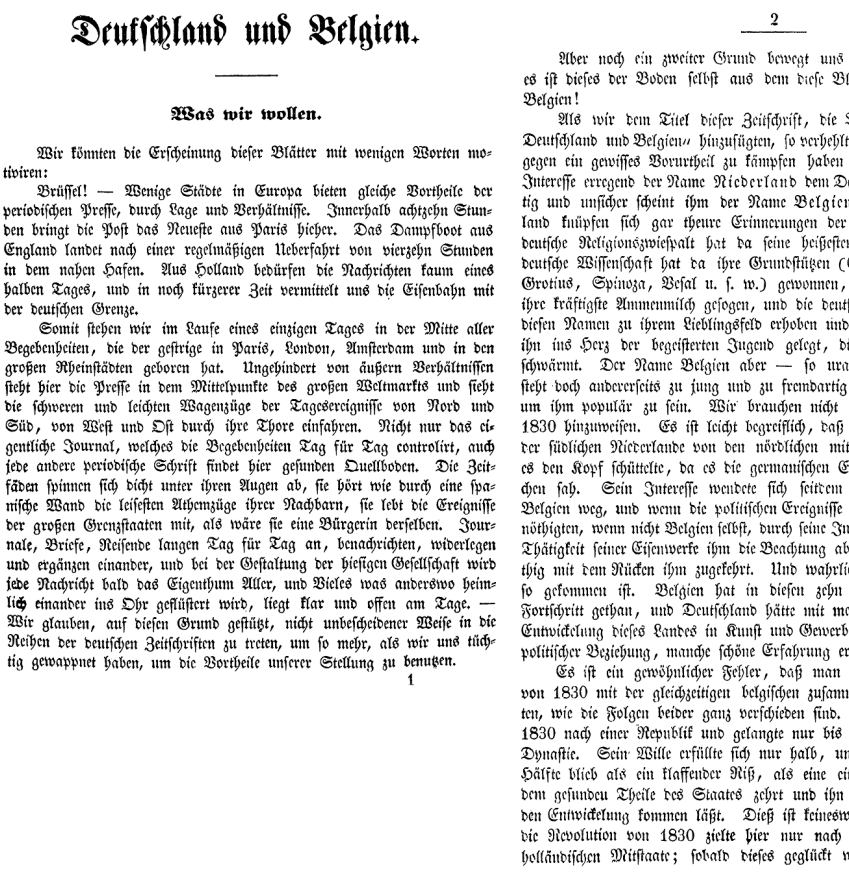
Bisher wurden basierend auf den längsten gemeinsamen Teilwörtern, die in zwei unterschiedlichen Wörtern auftreten, Paare gebildet. Damit entstehen jedoch auch innerhalb von Zeilen, die beide entweder zur Menge der Ground-Truth oder der

OCR gehören, Übereinstimmungen aufgrund ähnlichen Inhalts. Um diese Matches zu verhindern, kann als zusätzliche Metainformation die Herkunft jeder Zeile aufgenommen werden. Diese wird innerhalb der Sinkknoten gespeichert und kann dann wiederum durch ein ähnliches Vorgehen wie in Algorithmus 14 an kürzere Knoten weitergegeben werden, sodass nur solche Knoten als gemeinsame Teilwörter ausgewählt werden, die nicht nur genau zweimal auftreten, sondern jeweils einmal in einem Wort aus der Menge der Ground-Truth-Zeilen und einmal in einem Wort aus der Menge der OCR-Zeilen. Die beiden rechten Spalten ($Zugeordnet_M$,$Richtig_M$) in Tabelle 8.2 zeigen die Ergebnisse, wenn diese Metainformation berücksichtigt wird. Verglichen mit den vorigen Ergebnissen ist eine deutliche Verbesserung zu sehen, da von 663 gefundenen Teilwörtern nun 658 passende Zeilenpaare bilden. Prozentual ausgedrückt entspricht dies einem Verhältnis von 99,2%.

8.3.2. Zuordnung multipler OCR-Zeilen

In einem zweiten Versuch[9], bei dem es ebenfalls um die Zuordnung von OCR-Zeilen geht, wird das gleiche Vorgehen angewendet. Hierbei soll jedoch nun nicht die OCR ihrer Ground-Truth zugeordnet werden, sondern die OCR-Ergebnisse dreier verschiedener OCR-Engines, Ocropus [12, 11], Tesseract [69, 70] und Abbyy-FineReader einander. Zwar erzeugen alle drei OCR-Engines zeilengerechte Ausgaben, die Zeilensegmentierung kann manchmal aber voneinander abweichen. Um zu bestimmen, welche erkannten Zeilentripel dieselbe Textzeile betreffen, können erneut jeweils die längsten gemeinsamen Teilwörter dreier Zeilen gesucht werden, anhand derer eine Zuordnung entsteht. Hintergrund dieser Anwendung ist die Schaffung einer Vergleichsmöglichkeit verschiedener OCR-Ergebnisse, die bei einer automatischen Nachkorrektur der OCR hilfreich sein kann, zumal diese, wie bereits erwähnt, bei historischem Material viel leichter zu Fehlern neigt. Die hier verwendete Metainformation, die zu jeder Zeile gespeichert wird, ist somit die jeweilige OCR-Engine, die diese erzeugt hat. Damit wird wie im vorherigen Experiment ausgeschlossen, dass Zeilen einander zugeordnet werden, die zur selben Ursprungsmenge gehören. Als Testmaterial dienen 100 durch OCR erkannte Seiten der von Ignaz Kuranda gegründeten Wochenzeitschrift „Die Grenzboten“ des Jahrgangs 1841. Die nächste Abbildung zeigt die ersten beiden Seiten des verwendeten Materials.

[9] Dieses Experiment wurde zusammen mit Dr. Florian Fink durchgeführt. Die verwendeten OCR-Daten stammen ebenfalls von Dr. Uwe Springmann.



Deutschland und Belgien.

Was wir wollen.

Wir könnten die Erscheinung dieser Blätter mit wenigen Worten motiviren:

Brüssel! — Wenige Städte in Europa bieten gleiche Vortheile der periodischen Presse, durch Lage und Verhältnisse. Innerhalb achtzehn Stunden bringt die Post das Neueste aus Paris hieher. Das Dampfboot aus England landet nach einer regelmäßigen Ueberfahrt von vierzehn Stunden in dem nahen Hafen. Aus Holland bedürfen die Nachrichten kaum eines halben Tages, und in noch kürzerer Zeit vermittelt uns die Eisenbahn mit der deutschen Grenze.

Somit stehen wir im Laufe eines einzigen Tages in der Mitte aller Begebenheiten, die der gestrige in Paris, London, Amsterdam und in den großen Rheinstädten geboren hat. Ungehindert von äußern Verhältnissen steht hier die Presse in dem Mittelpunkte des großen Weltmarkts und sieht die schweren und leichten Wagenzüge der Tagesereignisse von Nord und Süd, von West und Ost durch ihre Thore einfahren. Nicht nur das eigentliche Journal, welches die Begebenheiten Tag für Tag controlirt, auch jede andere periodische Schrift findet hier gesunden Quellboden. Die Zeitfäden spinnen sich dicht unter ihren Augen ab, sie hört wie durch eine spanische Wand die leisesten Athemzüge ihrer Nachbarn, sie lebt die Ereignisse der großen Grenzstaaten mit, als wäre sie eine Bürgerin derselben. Journale, Briefe, Reisende langen Tag für Tag an, benachrichten, widerlegen und ergänzen einander, und bei der Gestaltung der hiesigen Gesellschaft wird jede Nachricht bald das Eigenthum Aller, und Vieles was anderswo heimlich einander ins Ohr gestüstert wird, liegt klar und offen am Tage. — Wir glauben, auf diesen Grund gestützt, nicht unbescheidener Weise in die Reihen der deutschen Zeitschriften zu treten, um so mehr, als wir uns tüchtig gewappnet haben, um die Vortheile unserer Stellung zu benutzen.

1

2

Aber noch ein zweiter Grund bewegt uns bei unserem Unternehmen, es ist dieses der Boden selbst aus dem diese Blätter hervorwachsen sollen: Belgien!

Als wir dem Titel dieser Zeitschrift, die Bezeichnung: „Blätter für Deutschland und Belgien“ hinzufügten, so verhehlten wir uns nicht, daß wir gegen ein gewisses Vorurtheil zu kämpfen haben werden. So poetisch und Interesse erregend der Name Niederland dem Deutschen klingt, so fremdartig und unsicher scheint ihm der Name Belgien. An das Wort Niederland knüpfen sich gar theure Erinnerungen der deutschen Geschichte. Der deutsche Religionszwiespalt hat da seine heißesten Kämpfer gefunden, die deutsche Wissenschaft hat da ihre Grundstützen (Erasmus, Justus Lipsius, Grotius, Spinoza, Vesal u. s. w.) gewonnen, die deutsche Kunst hat da ihre kräftigste Ammenmilch gesogen, und die deutsche Poesie hat daher auch diesen Namen zu ihrem Lieblingsfeld erhoben und Schiller und Göthe haben ihn ins Herz der begeisterten Jugend gelegt, die für Egmont und Posa schwärmt. Der Name Belgien aber — so uralt das Wort auch ist — steht doch andererseits zu jung und zu fremdartig dem Deutschen gegenüber, um ihm populär zu sein. Wir brauchen nicht erst auf die Ereignisse von 1830 hinzuweisen. Es ist leicht begreiflich, daß Deutschland die Trennung der südlichen Niederlande von den nördlichen mit Unmuth betrachtete, daß es den Kopf schüttelte, da es die germanischen Elemente den gallischen weichen sah. Sein Interesse wendete sich seitdem mit ziemlicher Kälte von Belgien weg, und wenn die politischen Ereignisse es nicht zur Aufmerksamkeit nöthigten, wenn nicht Belgien selbst, durch seine Industrie, durch die glänzende Thätigkeit seiner Eisenwerke ihm die Beachtung abzwang, da blieb es mißmuthig mit dem Rücken ihm zugekehrt. Und wahrlich, es ist nicht gut, daß es so gekommen ist. Belgien hat in diesen zehn Jahren einen riesenhaften Fortschritt gethan, und Deutschland hätte mit mehr Aufmerksamkeit auf die Entwickelung dieses Landes in Kunst und Gewerbe, in socialer und sogar in politischer Beziehung, manche schöne Erfahrung erwerben können.

Es ist ein gewöhnlicher Fehler, daß man die französische Revolution von 1830 mit der gleichzeitigen belgischen zusammenkettet, ohne zu betrachten, wie die Folgen beider ganz verschieden sind. Frankreich zielte im Jahr 1830 nach einer Republik und gelangte nur bis zu einer Veränderung der Dynastie. Sein Wille erfüllte sich nur halb, und die andere nicht erfüllte Hälfte blieb als ein klaffender Riß, als eine eiternde Wunde, welche an dem gesunden Theile des Staates zehrt und ihn nie zur Ruhe und gesunden Entwickelung kommen läßt. Dieß ist keineswegs mit Belgien der Fall; die Revolution von 1830 zielte hier nur nach einer Loslösung von dem holländischen Mitstaate; sobald dieses geglückt war, und die Aufregung,

Abbildung 8.9.: Erste und zweite Seite der von Ignaz Kuranda publizierten Wochenzeitschrift „Die Grenzboten“ (1841).

Verglichen mit der OCR auf den deutlich älteren Dokumenten des ersten Experiments liefern nun alle drei OCR-Engines sehr gute Erkennungsraten, sodass innerhalb jedes Zeilentripels aufgrund vieler gleich erkannter Zeichenfolgen durchschnittlich noch längere gemeinsame Teilwörter gefunden werden können. Alle drei OCR-Engines verwenden in diesem Fall für Frakturschrift optimierte Erkennungsmodelle. Die größten Unterschiede weist dabei Ocropus auf. Dort wird zum Beispiel das lange s, das in der Frakturschrift durch die Type ſ repräsentiert wird, durch „f“ anstelle eines normalen „s“ dargestellt. Die nächste Tabelle zeigt die OCR-Ergebnisse der drei Engines für einige Beispielzeilen der ersten beiden Seiten, wobei die Abkürzungen *O* für Orcopus, *T* für Tesseract und *A* für Abbyy-FineReader stehen. *B* steht für die jeweilige Bildzeile.

B	halben Tages, und in noch kürzerer Zeit vermittelt uns die Eiſenbahn mit
O	halben Tages und in noch kürzerer Zeit vermittelt uns die Eifenbahn mit
T	halben Tages und in noch kürzerer Zeit vermittelt uns die Eisenbahn mit
A	halben Tages und in noch kürzerer Zeit vermittelt uns die Eisenbahn mit
B	von 1830 mit der gleichzeitigen belgiſchen zuſammenkettet, ohne zu betrach=
O	von 18310 mit der gleichzeitigen belgiſchen zuſanmenkettet obne gu betrach
T	von 1830 mit der gleichzeitigen belgischen zusannnenkettet ohne zu betrach
A	voir 1830 mit der gleichzeitigen belgischen zusammenkettet ohne zu betrach
B	nöthigten, wenn nicht Belgien ſelbſt, durch ſeine Jnt uſtrie, durch die glänzende
O	ṅthigten wenn nicht Belgien ſelbſt durch ſeine Jnt uſtrie durch die gl̊ngende
T	nöthigten wenn nicht Belgien selbst durch seine Industrie durch die glänzende
A	nöthigten wenn nicht Belgien selbst durch seine Industrie durch die glänzende
B	deutſche Neligionszwieſpalt hat da ſeine heißeſten Kämpfer gefunden, die
O	deutſche eligivnowieſpalt hat da ſeine heißieſten Khmpfer gefiden die
T	deutsche Religionszwiespalt hat da seine heißesten Kämpfer gesunden die
A	deutsche Neligionszwiespalt hat da seine heißesten Kämpfer gefunden die

Tabelle 8.3.: Beispiele für OCR erkannte Zeilen der drei Engines auf den ersten beiden Seiten der Zeitschrift „Die Grenzboten“.

Es existieren zwar Unterschiede zwischen allen drei erkannten Zeilen, jedoch lassen sich auch große Gemeinsamkeiten ausmachen. Alles in allem besteht die Menge der Zeilen für O und A jeweils aus 3888 Zeilen, die für T aus 3916 Zeilen. Richtig zugewiesen werden konnten 3806 Zeilentripel, was einer korrekten Zuordnung von etwa 97% entspricht.

Neben dem praktischen Nutzen, der aus den guten Zuordnungsraten beider Experimente erwächst, zeigt diese Anwendung auch auf, wie flexibel eine SCDAWG-Struktur eingesetzt werden kann. Zum einen können beliebig viele Wörter indexiert und bei Bedarf auch weitere hinzugefügt werden. Zum anderen können, wenn die Struktur aufgebaut ist, in linearer Zeit weitere Kenngrößen für bestimmte Knotenteilmengen berechnet werden. Durch das Hinzufügen der Ursprungsmengen, in denen ein Knoten auftritt, als Metainformation konnten die gesuchten Knotenmengen weiter voneinander abgegrenzt werden. Die hierbei zugrundeliegende Annahme ist, dass für viele Zeilenpaare oder Tripel ein eindeutiges Teilwort gefunden werden kann, mit dessen Hilfe eine eindeutige Zuordnung möglich ist.

9. Globale Alignierung

In diesem Kapitel wird untersucht, wie sich durch eine lineare Ordnung der im vorherigen Kapitel extrahierten gemeinsamen Teilwörter ein globales Alignment zweier Dokumente erzeugen lässt. Dieses Verfahren wird im Vergleich zu einer optimalen Alignierungsmethode betrachtet.

9.1. Indexgestützte globale Alignierung

Die im vorherigen Abschnitt beschriebene Anwendung erzeugt eine Zuordnung zweier oder mehrerer Wörter (hier Zeilen) aus W anhand der längsten gemeinsamen Teilwörter, die innerhalb eines Paares oder Tupels auftreten. Damit wird die SCDAWG-Struktur dazu verwendet, alle längsten gemeinsamen Teilwörter zwischen diesen Einheiten auf einmal zu bestimmen. Durch die so entstehende gegenseitige Beeinflussung aller indexierten Wörter untereinander können zwar nicht immer für alle Paare oder Tupel die korrekten Partnerzeilen gefunden werden, wenn zusätzlich deren Ursprungsmenge berücksichtigt wird, ergeben sich für die beiden Experimente aber sehr hohe Zuordnungsraten. Der in Kapitel 8 vorgestellte Algorithmus 17 findet durch die gleichzeitige Betrachtung aller linken und rechten Übergänge (Übergangspaare), die auf einen Sinkknoten führen, aber nicht nur ein längstes gemeinsames Teilwort innerhalb einer bestimmten Teilmenge aus W, sondern alle längsten gemeinsamen Infixe, die an ihrer Vorkommensposition nicht mehr nach links oder rechts erweitert werden können.

9.1.1. Grundidee

Die so identifizierten bedingt quasimaximalen Knoten decken die Menge der längsten Teilwörter ab, die zwei Wörter w^1 und w^2 gemeinsam haben. Zur Erstellung eines globalen Alignments zwischen w^1 und w^2 müssten die Elemente von Q_{max} aber geordnet werden, sodass zwischen diesen diejenigen Infixe übrigbleiben, die nur in w^1 oder w^2 vorkommen. Der Grundgedanke ist es also, die Menge aller längsten gemeinsamen Teilwörter in eine lineare Folge zu bringen, sodass aus dieser ein „Alignierungsskelett“ entsteht, zwischen dessen Gliedern sich nicht gemeinsame Infixe befinden. Als Algorithmus zur Berechnung der linearen Folge

quasimaximaler Knoten kann der in Kapitel 2.5 vorgestellte LCS-Algorithmus verwendet werden.

Das LCS-Verfahren würde aber nicht mehr jedes Zeichen der Eingabestrings betrachten, sondern auf der Vorsegementierung, die sich durch die quasimaximalen Knoten ergibt, arbeiten. Zusätzlich müsste der LCS-Algorithmus aber modifiziert werden, da etwa wie beim Needleman-Wunsch Algorithmus gegebenenfalls mehrere optimale Alignierungen existieren, die gefunden werden sollen. Damit diese auch durch das LCS-Verfahren gefunden werden können, sollen alle möglichen Alignierungen in einem Alignment-Graphen gespeichert werden, in dem sie von einer Bewertungsfunktion gerankt werden, sodass dann eine Alignierung ausgewählt wird. Wie am Ende von Kapitel 8 bereits erwähnt, kann allerdings nicht davon ausgegangen werden, dass hierdurch immer ein optimales Alignment entsteht. Trotz dieses Nachteils bietet die Vorberechnung der quasimaximalen Knoten für die Laufzeit eines Alignierungsverfahrens große Vorteile, da alle in Q_{max} enthaltenen Infixe sich auf Stellen in den Eingabewörtern beziehen, die bereits „perfekt“ aligniert sind, wodurch die Laufzeit des LCS-Algorithmus sinkt, da dieser nun nur die längste aufsteigende Folge aus den gemeinsamen Teilwörtern zweier Wörter berechnen muss und nicht mehr eine LCS aus einzelnen Zeichen.

9.1.2. Implementierung

Aus diesen Überlegungen lässt sich wieder ein Grobentwurf zum Ablauf eines indexgestützten Alignierungsverfahrens ableiten.

1. Erstelle die SCDAWG-Struktur S_C für beide Eingabestrings w^1 und w^2.
2. Traversiere S_C und finde durch *find_longest_common_substrings* deren quasimaximale Knoten.
3. Berechne anhand der Endpositionen der gefunden quasimaximalen Knoten alle längsten aufsteigenden Folgen für die beiden Wörter und baue einen Alignierungsgraphen auf.
4. Bewerte alle Alignierungspfade und wähle einen aus.

Vorab wird eine SCDAWG-Struktur S_C für zwei Eingabestrings w^1, w^2 erzeugt. Detaillierte Angaben zum on-line-Aufbau von SCDAWGs finden sich im Unterkapitel 5.1.

Nach Definition 8.1.3 beziehungsweise dem Ablauf von Algorithmus 17 für zwei Eingabewörter w^1 und w^2 ergibt sich eine Menge Q_{max} von quasimaximalen Knoten, die gemeinsam auftretende maximale Teilwörter repräsentiert. Die erzeugten

Tripel (v, i, j), die aus dem Knoten v, dem Wortindex i und der Endposition j bestehen, werden nun so umgeformt, dass für alle gefundenen Knoten zwei Listen E_1 und E_2 entstehen, die alle Endpositionen der Knoten in w^1 und alle Endpositionen der Knoten in w^2 enthalten. Da jede Endposition nur einmal in den Listen E_1 und E_2 vorkommen kann, können diese zur Berechnung der LIS (siehe Kapitel 2.5.1) der beiden Strings verwendet werden. So findet sich zu jeder Endposition e_{1_j} in E_1 eines Knotens eine decreasing Subsequence aus Endpositionen aus E_2 $d_{1_j} = \{e_{2_k} \mid e_{2_k} \in E_2 \text{ mit } e_{2_k} > e_{2_{k+1}}, 1 \leq k \leq |E_2|\}$, an denen dieser Knoten in w^2 auftritt, mit $j, k \in \mathbb{N}$. Die formale Beschreibung des auf den SCDAWG gestützten Alignierungsalgorithmus gestaltet sich wie folgt:

Algorithmus 19 SCDAWG-gestützte globale Alignierung zweier Strings

Vorbedingung: Die SCDAWG-Struktur $S_C = (V, E_R, E_L)$ einer endlichen Menge von Zeichenketten W mit $W = \{w^1, w^2\}$ und $w^1, w^2 \in \Sigma^*$.

```
1:  function align()
2:      Q_max ← find_longest_common_substrings()   ▷ Aufruf von Algorithmus 17
3:      E_1 ← [|w^1|]
4:      E_2 ← [|w^2|]
5:      E_2map ← (v_i, {})
6:      for each (v, i, j) ∈ Q_max do              ▷ Umformen der Tripel (v, i, j)
7:          if i = 1 then
8:              E_1[j] ← v
9:          else
10:             E_2[j] ← v
11:         end if
12:     end for
13:     for i to |E_2| do
14:         E_2map[E_2[i]].push(i)
15:     end for
16:     A_g ← build_alignment_graph(E_1, E_2map)   ▷ Aufruf von Algorithmus 20
17:     result ← rank_alignments(g)                ▷ Aufruf von Algorithmus 23
18:     return result
19: end function
```

Nachdem in Schritt 1 die SCDAWG-Struktur zweier Eingabestrings w^1 und w^2 erstellt wurde, sucht die Funktion *align* alle quasimaximalen Knoten zweier Strings durch Algorithmus 17 (Schritt 2). Daraufhin werden alle Knoten nach ihrer Endposition in zwei Arrays E_1 und E_2 einsortiert. Eine aufsteigende Sortierung ergibt sich, da E_1 und E_2 mit den Längen der beiden Strings initialisiert werden.

Zudem wird eine Map $E_{2_{map}}$ mit dem jeweiligen quasimaximalen Knoten und allen Endpositionen im zweiten String befüllt, sodass dort, wenn als Schlüssel ein bestimmter Knoten gegeben wird, alle Endpositionen in w^2 zurückgegeben werden. Die nötigen Umformungen sind dem Ergebnis, das Algorithmus 17 liefert, geschuldet, sie sind jedoch linear in der Anzahl aller Endpositionen quasimaximaler Knoten. Wenn etwa nur das hier beschriebene globale Alignierungsverfahren implementiert werden soll, kann Algorithmus 17 auch so abgeändert werden, dass der Umformungschritt entfällt.

Nach den ersten beiden Schritten und den darauffolgenden Umformungen werden für die Schritte 3 und 4 zwei weitere Funktionen *build_alignment_graph* und *rank_alignments* gestartet. Erstere stellt die Hauptprozedur dar, die die Kombination der umsortierten quasimaximalen Knoten mit dem LCS-Algorithmus übernimmt. Die zweite Funktion bewertet alle entstehenden Alignierungen.

Aufbau des Alignierungsgraphen

Algorithmus 20 Algorithmus zum Aufbau des Alignierungsgraphen

```
1:  function build_alignment_graph(E_1, E_2map)
2:      A_g ← (V, E)
3:      V[−1] ← v_λ                                  ▷ v⁺ Starknoten
4:      gc_List ← {}
5:      gc ← build_greedy_cover(E_1, E_2map, gc_List)   ▷ Aufruf von Algorithmus 21
6:      for i ← 0 to gc[|gc|−1] do
7:          coverIndex ← gc[|gc|−1][i]
8:          V[coverIndex] ← v_i                      ▷ v⁺ Knoten der letzten Spalte
9:          E ← E ∪ {(v_λ, i + 1, v_i)}              ▷ → von v_λ → v_i
10:     end for
11:     determine_lis(gc, gc_List, |gc|−1)           ▷ Aufruf von Algorithmus 22
12:     return A_g
13: end function
```

In der Funktion zum Aufbau des Alignierungsgraphen wird zunächst ein neuer gerichteter Graph A_g instanziiert und diesem ein Startknoten v_λ angefügt. Zusätzlich wird eine Hilfsliste gc_{List} erzeugt und dann ein greedy Cover gc durch die in Algorithmus 21 beschriebene Funktion erzeugt. Die in gc gespeicherte Struktur eines greedy Covers entspricht der in Kapitel 2.5.1 eingeführten Datenstruktur, die dort zur Berechnung der longest increasing subsequence verwendet wurde. Als Einträge in diese zweidimensionale Liste werden, wie im Vorfeld erwähnt, nun

nicht mehr die Positionen der Vorkommen der einzelnen Zeichen von w^1 in w^2 benutzt, sondern die Endpositionen der quasimaximalen Knoten von w^1 in w^2. Die Hilfsliste gc_{List} wird zur Verbindung des greedy Covers mit den quasimaximalen Knoten benutzt. Diese enthält Tupel, die neben dem quasimaximalen Knoten dessen Endposition in beiden Strings und einen *ancestorIndex* besitzt. Letzterer wird dazu verwendet, die möglichen Vorgänger eines Eintrages einer Spalte des greedy Covers gc zu bestimmen. Hierdurch wird gewährleistet, dass mehrere mögliche Alignierungen, die aus der gleichen Anzahl quasimaximaler Knoten bestehen, erkannt werden können.

Nachdem die Funktion *build_greedy_cover* abgelaufen ist, erhält der initialisierte Alignierungsgraph A_g neue Knoten für die Einträge in der letzten Spalte des Covers. Jeder so erzeugte Knoten v_i wird mit dem Wurzelknoten v_λ verbunden. Als Übergangslabel können alle Kanten von A_g mit 1 beginnend aufsteigend durchnummeriert werden. Alternativ könnte auch die jeweilige Endposition in einem der beiden Strings oder aber die beiden zwischen zwei Knoten entstehenden Gaps angezeichnet werden. Die Kantenlabels von A_g besitzen aber keine funktionale Relevanz und können bei Bedarf auch leer gelassen werden. Durch das Hinzufügen aller Knoten aus der letzten Spalte von gc und deren Verbindung mit v_λ enthält A_g nun mehrere mögliche Startpunkte, von denen aus verschiedene Alignierungspfade durchlaufen werden können. Anschließend wir eine weitere Hilfsfunktion *determine_lis* aufgerufen, die den restlichen Alignierungsgraphen erzeugt.

Um einen Zugriff auf die Knoten von A_g über die Indizes von gc zu bekommen, lässt sich jeder hinzugefügte Knoten über einen *coverIndex*, der seine Position in gc angibt (Zeile 8), ansprechen. Damit kann später für alle Knoten über den Index in gc überprüft werden, ob dieser Knoten eine mögliche Verlängerung eines Alignierungspfades darstellt oder nicht.

Erstellung des greedy Covers

In Algorithmus 21 wird nun anhand der Funktion *build_greedy_cover* gezeigt, wie sich aus den umformatierten Eingabelisten quasimaximaler Knoten ein entsprechendes greedy Cover gc erzeugen lässt. Falls für ein mögliches indexbasiertes Alignment keine Bewertung gleichwertiger Alignierungen (gleiche Anzahl quasimaximaler Knoten) benötigt wird, kann nach Ablauf dieser Funktion auch die einfache Berechnung der LIS (siehe Kapitel 2.5.1) verwendet werden. Das Ergebnis der Funktion *build_greedy_cover* ist eine zweidimensionale Liste, die in Zeile 2 beziehungsweise in den Zeilen 4 - 6 mit der Länge des Endpositionen-Arrays E_1 initialisiert wird und für die Anzahl der Einträge in E_1 als Inhalt jeder Spalte ein leere Liste erhält. Außerdem wird noch eine boolesche Variable *firstEntry* mit *true* vorbesetzt.

Algorithmus 21 Algorithmus zum Aufbau des greedy Covers

```
1: function build_greedy_cover(E_1, E_2map, gc_List)
2:     gc ← [|E_1|]
3:     firstEntry ← true
4:     for i ← 0 to |gc| do
5:         gc[i] ← {}
6:     end for
7:     for i ← 0 to |gc| do
8:         if E_1[i] = ∅ then
9:             continue
10:        end if
11:        if E_2map[E_1[i]] = ∅ then        ▷ Wenn kein passender Knoten in E_2map
12:            continue
13:        end if
14:        for j ← E_2map[|E_1|−1] to 0 do
15:            decElem ← E_2map[E_1[i]][j]
16:            gc_List.push((E_1[i], i, decElem, −1))        ▷ Hinzufügen neues Tupel
17:            coverIndex ← |gc_List|−1
18:            if (j = |E_2map[E_1[i]]|−1 and firstEntry) then
19:                gc[0].push(coverIndex)
20:                firstEntry ← false
21:            else
22:                for k ← 0 to |gc| do
23:                    ancestorIndex ← gc[k − 1][|gc[k − 1]|−1]
24:                    lastPos ← gc_List[gc[k][|gc|−1]][2]    ▷ Letzter Covereintrag
25:                    if |gc[k]| = 0 then
26:                        gc[k].push(coverIndex)
27:                        gc_List[coverIndex][3] ← ancestorIndex
28:                        break
29:                    else if decElem ≤ lastPos then
30:                        gc[k].push(coverIndex)
31:                        if k = 0 then
32:                            gc[coverIndex][3] ← ancestorIndex − 1
33:                        else
34:                            gc[coverIndex][3] ← ancestorIndex
35:                        end if
36:                        break
37:                    end if
38:                end for
39:            end if
40:        end for
41:    end for
42:    return gc
43: end function
```

In Zeilen 8-13 werden die Endpositionen, an denen im zweiten String keine Endposition zu einem quasimaximalen Knoten aus dem ersten String existiert, übersprungen ($E_{2_{map}}[E_1[i]] = \emptyset$).

Betrachtet man die zwei Strings $\#4abc5ab6\$$ und $\#7abc8ef\$$ aus Beispiel 8.1.2, würde nun für den quasimaximalen Knoten, der ab repräsentiert, im zweiten String keine passende Endposition gefunden werden. Dies zeigt, dass bei diesem Verfahren quasimaximale Knoten als voneinander getrennte Einheiten behandelt werden.

Diese Schritte sind somit den Datenstrukturen und dem Ergebnis von Algorithmus 17 geschuldet und könnten durch eine Modifikation von Algorithmus 17 ausgelassen werden.

In Zeile 14 werden nun alle Endpositionen im zweiten String, die es zu einer Endposition eines quasimaximalen Knotens im ersten String gibt, rückläufig, also in absteigender Reihenfolge, mithilfe der Variablen j durchlaufen. Dies entspricht der Vorgabe aus dem LIS-Algorithmus (siehe Kapitel 2.5.1), dies in „decreasing order“ zu tun. Dementsprechend wird in Zeile 15 eine Variable *decElem* nacheinander mit jeder zugehörigen Endposition aus dem zweiten String besetzt.

Für jedes *decElem* wird in Zeile 16 ein Eintrag in gc_{List} vorgenommen, der den aktuellen Knoten ($E_1[i]$), die Endposition im ersten String (i) und die im zweiten String (*decElem*) sowie einen Eintrag für den *ancestorIndex* mit -1 erhält. Als nächstes wird eine Variable für den Eintrag in das Cover (*coverIndex*) selbst besetzt, die auf das eben eingefügte Tupel in gc_{List} zeigt. Wenn nun j auf die allerletzte Endposition in $E_{2_{map}}$ zeigt und gc noch keinen Eintrag erhalten hat (*firstEntry* = *false*), wird dem greedy Cover nun dieses Element, beziehungsweise der Verweis auf den zugehörigen Eintrag in gc_{List}, an die erste Stelle der ersten Spalte geschrieben (Zeile 19).

Wenn das Cover nicht leer ist, werden dessen Spalten durchsucht und überprüft, an welcher Stelle das neue Element eingefügt werden kann (Zeile 22-38). Hierbei wird die Varibale *ancestorIndex* besetzt, die die Indexposition des Vorgängereintrags der vorherigen Spalte beinhaltet. Dieser wird später dazu benutzt, mehrere mögliche Alignierungspfade innerhalb des Alignierungsgraphen zu finden. Zudem wird die Endposition im zweiten String des letzten Eintrags in gc_{List} in einer Variablen *lastPos* zwischengespeichert (Zeile 24).

Im Cover selbst befinden sich, wie bereits erwähnt, nur Referenzen (*CoverIndex*) auf die Einträge von gc_{List}, in der sich die eigentlichen Informationen zu jedem Covereintrag befinden. Um dem greedy Cover gc alle weiteren Einträge hinzuzufügen, wird in der Schleife unterschieden, ob es sich um den Anfang, also das erste Element einer neuen Spalte, handelt oder um ein Element, das kleiner oder gleich dem letzten Element einer bereits existierenden Spalte ist. Wenn *decElem* kleiner oder gleich einem der letzten Elemente einer Spalte ist (Zeile 29), dann wird ein neuer Eintrag innerhalb dieser Spalte vorgenommen. Ist dies nicht der Fall, handelt

es sich beim aktuellen Element um einen neuen Eintrag in eine noch leere Spalte des Covers (Zeile 25).

In beiden Fällen wird für das neu eingefügte Element das Attribut *ancestorIndex* (die vierte Position der Tupel, die hier stets mit Index 3 angesprochen wird) mit dem Index des vorherigen Eintrags besetzt. Jeder Eintrag erhält damit eine Referenz auf den Eintrag, aufgrund dessen ihm seine Position im Cover zugewiesen wurde.

Traversierung des greedy Covers

Durch die Besetzung des *ancestorIndex*, also den Verweis auf den vorangegangen nächstkleineren Covereintrag, können nun ähnlich wie beim Traceback der Rückwärtspointer, die zum Beispiel beim Needleman-Wunsch Algorithmus (siehe Kapitel 2.3) verwendet werden, mehrere gleichwertige längste aufsteigende Folgen beziehungsweise Alignierungen gefunden werden. Gleichwertig bedeutet, dass die längsten aufsteigenden Folgen die gleiche Komponentenanzahl, in diesem Falle quasimaximale Knoten, teilen. Der nachfolgend beschriebene Algorithmus 22 beziehungsweise die darin beschriebene Prozedur *determine_lis* findet rekursiv aufgrund dieser gespeicherten Referenzen oder Rückwärtspointer im greedy Cover gc alle längsten aufsteigenden Folgen zweier Eingabewörter und fügt diese in den Alignierungsgraphen A_g ein. Den Startpunkt von A_g bildet der in Algorithmus 20 initialisiert Graph, der Übergänge vom Startknoten v_λ auf alle Einträge der letzten Spalte von gc besitzt.

determine_lis unterscheidet sich von dem in [31] beschriebenen Verfahren dahingehend, dass in jeder Spalte in gc mit den letzten Knoten beginnend mehrere Pfade weiterverfolgt werden können. Hierfür wird für jeden Knoten aus einer Spalte *columnIndex*, der in gc eingefügt wurde, seine Endposition (Zeile 8) und der zugehörige Rückwärtspointer beziehungweise *ancestorIndex* bestimmt. Durch die Anweisungen in Zeilen 14 - 25 werden alle Spalteneinträge der vorherigen Spalte $columnIndex - 1$ berücksichtigt, die unterhalb des *ancestorIndex* des aktuellen Eintrags aus der Coverspalte *columnIndex* liegen. Wird dort für einen Eintrag der Spalte $columnIndex - 1$ eine Endpostion im zweiten String gefunden, die kleiner oder gleich der Endposition des aktuellen Eintrags aus $columnIndex - 1$ ist (Zeile 17), so handelt es sich um einen zulässigen Kandidaten, und es werden ein neuer Knoten (Zeile 21) und eine neue Kante (Zeile 23) in den Alignierungsgraphen eingefügt.

Solange der Spaltenindex *columnIndex* größer 0 ist, ruft sich die Funktion selbst mit der nächsten Coverspalte (Zeile 28) auf. Ist dies nicht der Fall, wird die Funktion beendet.

Algorithmus 22 Hilfsalgorithmus zur Erstellung des Alignierungsgraphen

Vorbedingung: Der vorinitialisierte Alignierungsgraph $A_g = (V, E)$.

```
1: function determine_lis(gc, gc_List, columnIndex)
2:     for i ← 0 to |gc| do
3:         coverIndex ← gc[columnIndex][i]
4:         v ← V[coverIndex]                          ▷ Finden des Knotens v
5:         if v = ∅ then
6:             continue
7:         end if
8:         endpos_s2 ← gc_List[coverIndex][2]
9:         ancestorIndex ← gc_List[coverIndex][3]
10:        if ancestorIndex = −1 then
11:            break
12:        end if
13:        for j ← index_of(gc[columnIndex − 1], ancestorIndex) to 0 do
14:            next_coverIndex ← gc[columnIndex − 1][j]
15:            next_endpos_s2 ← gc_List[next_coverIndex][2]
16:            if next_endpos_s2 ≤ endpos_s2 then
17:                if V[next_coverIndex] ≠ ∅ then
18:                    v_next ← V[next_coverIndex]
19:                else
20:                    V[next_coverIndex] ← v_next        ▷ v^+ Neuer Knoten
21:                end if
22:                E ← E ∪ {(v, j, v_next)}           ▷ ⟶ von v ⟶ v_next
23:            end if
24:        end for
25:    end for
26:    if columnIndex > 0 then
27:        determine_lis(gc, gc_List, columnIndex − 1)
28:    end if
29:    return
30: end function
```

Durch die Verknüpfung jedes Knotens aus A_g mit dem greedy Cover gc können, wenn A_g fertig aufgebaut ist, zu jedem Knoten die in gc_{List} gespeicherten Informationen abgerufen werden. Zur Berechnung möglicher Alignmentpfade kann A_g nun von v_λ aus mit einer Tiefensuche traversiert werden, wobei jeder so durchwanderte Pfad jeweils eine mögliche Alignierung von w^1 und w^2 darstellt.

Bewertung möglicher Alignments

Um aus allen möglichen Alignierungen eine „beste“ auszuwählen, wird eine Scoringfunktion gebraucht, die aufgrund der verschiedenen Zusammensetzungen der möglichen Alignierungspfade eine globale Alignierung auswählt. Im nun beschriebenen Algorithmus wird die Summe der Längen der in allen längsten aufsteigenden Folgen vorkommenden Teilwörter (quasimaximale Knoten) maximiert. Aus Gründen der Einfachheit wird angenommen, dass alle längsten gemeinsamen Folgen in eine Datenstruktur all_{lcs} übersetzt worden sind, die mittels einer Schleife durchlaufen werden kann. Eine solche Serialisierung kann durch eine einfache Traversierung des Alignmentgraphen erzeugt werden. Die von *rank_alignments* ausgewählte LCS wird aufgrund der Tatsache, dass der Alignierungsgraph rückwärts durchlaufen wird, bevor sie zurückgegeben wird, umgedreht.

Algorithmus 23 Scoringfunktion mehrerer längster aufsteigender Folgen

```
1:  function rank_alignments()
2:      max_sum ← 0
3:      result ← ∅
4:      for each lcs ∈ all_lcs do
5:          sum ← 0
6:          for i ← 0 to |lcs| do
7:              (v, endpos_s1, endpos_s2, ancestorIndex) ← lcs[i]
8:              sum ← sum + |v|
9:          end for
10:         if sum > max_sum then
11:             max_sum ← sum
12:             result ← lcs
13:         end if
14:     end for
15:     return reverse(result)
16: end function
```

9.1.3. Beispielablauf

Im Folgenden wird anhand der beiden Beispielstrings,

$$w^1 = \#111A222B333C444D\$,$$

$$w^2 = \#A111B222C333D444\$,$$

gezeigt, wie der auf einen SCDAWG gestützte globale Alignierungsalgorithmus arbeitet. Nach Algorithmus 19 werden zuerst alle quasimaximalen Knoten gesucht und deren Endpositionen durch die beiden Listen E_1, E_2 und $E_{2_{map}}$ dargestellt.

E_1	E_2	$E_{2_{map}}$
$E_1[1] = \#$	$E_2[1] = \#$	$E_{2_{map}}[\#] = [1]$
$E_1[4] = 111$	$E_2[2] = A$	$E_{2_{map}}[A] = [2]$
$E_1[5] = A$	$E_2[5] = 111$	$E_{2_{map}}[111] = [5]$
$E_1[8] = 222$	$E_2[6] = B$	$E_{2_{map}}[B] = [6]$
$E_1[9] = B$	$E_2[9] = 222$	$E_{2_{map}}[222] = [9]$
$E_1[12] = 333$	$E_2[10] = C$	$E_{2_{map}}[C] = [10]$
$E_1[13] = C$	$E_2[13] = 333$	$E_{2_{map}}[333] = [13]$
$E_1[16] = 444$	$E_2[14] = D$	$E_{2_{map}}[D] = [14]$
$E_1[17] = D$	$E_2[17] = 444$	$E_{2_{map}}[444] = [17]$
$E_1[18] = \$$	$E_2[18] = \$$	$E_{2_{map}}[\$] = [18]$

Tabelle 9.1.: Endpositionen-Listen der zwei Beispielstrings.

Da kein quasimaximaler Knoten in einem der beiden Strings mehrfach auftritt, enthält $E_{2_{map}}$ jeweils nur einelementige Listen. Die hier gezeigten Listen sowie die HashMap $E_{2_{map}}$ bilden die Basis für den Aufbau des greedy Covers. Wie bereits erwähnt, können auch andere Datenstrukturen gewählt werden, solange die Laufzeitkomplexität davon nicht beeinträchtigt wird. Nach dem Ablauf von Algorithmus 21 ergibt sich für das greedy Cover folgende Matrix:

1	5	9	13	17	18
	2	6	10	14	

Dies entspricht folgender Einordnung quasimaximaler Knoten:

#	111	222	333	444	$
	A	B	C	D	

Durch das Besetzen der Rückwärtspointer in der Hilfsstruktur gc_{List} ergibt sich nach dem Ablauf von Algorithmus 22 folgender, rückwärts zu lesender Alignierungsgraph A_g, der mehrere längste aufsteigende Folgen beinhaltet:

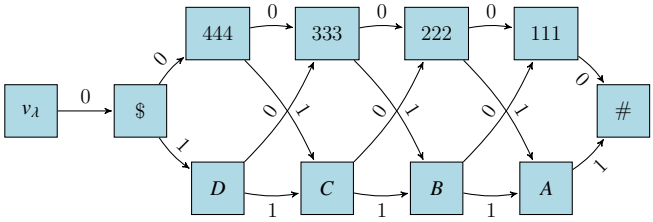


Abbildung 9.1.: Alignmentgraph zweier Beispielstrings.

Jeder Pfad, der vom letzten Knoten $ zum Startsymbol # führt, stellt eine potentielle Alignierung dar, da jeder quasimaximale Knoten, der ein zulässiger Vorgänger eines anderen ist, Teil einer längsten aufsteigenden Folge sein kann. So könnte zum Beispiel ein Pfad gewählt werden, in dem nur Knoten enthalten sind, die Ziffern beinhalten ($444333222111#). Es könnten ebenso Knoten benutzt werden, die nur Buchstaben beinhalten ($*DCBA*#). Aber auch jede andere Kombination aus sechs Knoten wie zum Beispiel ($*D*333*B*111#) ist zulässig. Insgesamt können 16 längste gemeinsame Folgen, die aus sechs Knoten bestehen, gefunden werden. Für die Beispielalignierung wird nach Anwendung der Scoringfunktion die gemeinsame Folge #111222333444$ als diejenige ausgewählt, deren summierte Knotenlänge maximal ist. Dies führt zur unten dargestellten Alignierung, wobei Gaps rot eingefärbt sind und als Gapsymbol für Einfügungen oder Löschungen die Tilde (~) gewählt wurde.

#	~	111	*A*	222	*B*	333	*C*	444	*D*	$
#	*A*	111	*B*	222	*C*	333	*D*	444	~	$

Abbildung 9.2.: Durch die Scoringfunktion ausgewähltes Alignment.

Ausblick

Die Traversierung des Alignierungsgraphen kann, wenn dieser zu viele gleichwertige Alignments enthält, zu einem Problem werden. Für die hier dargestellten Anwendungen, etwa im Bereich der Alignierung historischer OCR, ist dieser Fall jedoch unwahrscheinlich, da genug Übereinstimmungen zu erwarten sind. Wenn aber zum Beispiel Wörter aligniert werden, die etwa als einzige Gemeinsamkeit ein oft wiederholtes Leerzeichen teilen, sollte anstelle des Alignierungsgraphen die ursprüngliche LCS-Methode auf die quasimaximalen Knoten angewendet werden, bei der nur eine mögliche Alignierung gefunden wird.

9.1.4. Beispiele für *align*

Im Folgenden werden einige Beispielalignierungen kurzer Eingabewörter gezeigt, die durch *align* entstehen.

Beispiel 9.1.1. Für die ersten beiden Eingabewörter aus obigem Beispiel 8.1.2, $W = \{\#1abc2ab3\$, \#4abc5ab6\$\}$, entsteht folgendes Alignment.

#	1	*abc*	2	*ab*	3	$
#	4	*abc*	5	*ab*	6	$

Da in diesem Beispiel nur vier gemeinsame Teilwörter vorkommen (#, *abc*, *ab*, $), die jeweils genau einmal in w^1 und genau einmal in w^2 auftreten, existiert nur ein möglicher Pfad in A_g und somit auch nur ein mögliches Alignment.

Beispiel 9.1.2. In den beiden Eingabwörtern aus Beispiel 8.1.3, das zur Illustration der eindeutigen Ketten diente, $W = \{\#1b2aaaaaa3\$, \#4bbbbbb5a6\$\}$, stehen jeweils dem ersten *b* aus w^1 sechs *b*s aus w^2 gegenüber und dem ersten *a* aus w^2 sechs *a*s aus w^1.

#	1~~~~~	*b*	2	*a*	*aaaaa*3	$
#	4*bbbbb*	*b*	5	*a*	6~~~~~	$

Dies ergibt insgesamt 25 mögliche Alignierungen, die auch nach der Scoring-Funktion von Algorithmus 23 alle gleichwertig sind. Somit kann eine beliebige der 25 Möglichkeiten ausgewählt werden, da keine einer anderen gegenüber einen Vorteil verspricht.

Beispiel 9.1.3. Das nächste Beispiel zeigt ein Alignment zweier parallel durch zwei verschiedene OCR-Engines erkannte Zeilen innerhalb einer Seite aus „Die Grenzboten“.

#	*n*	*en in de*	~	*m Momente wo der Wagenzug anrasselte*	~	*die*	~	*gräßlich*	*e*
#	~	*en in de*		*m Momente wo der Wagenzug anrasselte*	*z*	*die*	*C*	*gräßlich*	*ks*
Ze	*r*	*schm*	*e*	*t*	*tere~~~*	*e*	*r*	$
Ze	*k*	*schm*	*k*	*t*	*kckking*	*e*	*x*	$

Obwohl beide Engines bei der Zeichenerkennung Fehler machen, existieren größere gemeinsame Regionen, die richtig erkannt wurden. Durch diese Regionen wird nun eine längste gemeinsame Folge gebildet, die dann zu einem globalen Alignment führt. In diesem Fall wurden beide Zeilen optimal aligniert, da in keiner Gap des einen Strings noch Zeichen enthalten sind, die mit Zeichen aus der zugehörigen Gap des anderen Strings übereinstimmen und auch keine alternative längste gemeinsame Folge existiert, die eine größere Abdeckung garantiert.

Beispiel 9.1.4. Das nächste Beispiel zeigt den Beginn zweier Versionen eines Gedichts. Die erste Version stammt von Friedrich Schiller, die zweite von Johann Wolfgang von Goethe[1].

Einer, *Friedrich Schiller*	***Sommer***, *Johann Wolfgang v. Goethe*
1) *Grausam handelt Amor mit mir! O! spielet, ihr Musen,* *Mit den Schmerzen, die er, spielend, im Busen erregt.*	1) *Grausam erweiset sich Amor an mir! O spielet, ihr Musen,* *Mit den Schmerzen, die er, spielend, im Busen erregt!*
2) *Manuskripte besitz ich wie kein Gelehrter noch König,* *Denn mein Liebchen, sie schreibt, was ich ihr dichtete, mir.*	2) *Manuskripte besitz ich wie kein Gelehrter noch König;* *Denn mein Liebchen, sie schreibt, was ich ihr dichtete, mir.*
3) *Wie im Winter die Saat nur langsam keimet, im Frühling* *Lebhaft treibet und schoßt, so war die Neigung zu dir.*	3) *Wie im Winter die Saat nur langsam keimet, im Sommer* *Lebhaft treibet und reift, so war die Neigung zu dir.*
4) *Immer war mir das Feld und der Wald und der Fels und die Gärten* *Nur ein Raum, und du machst sie, Geliebte, zum Ort.*	4) *Immer war mir das Feld und der Wald und der Fels und die Gärten* *Nur ein Raum, und du machst sie, Geliebte, zum Ort.*
5) *Raum und Zeit, ich empfind es, sind bloße Formen des Denkens,* *Da das Eckchen mit dir, Liebchen, unendlich mir scheint.*	5) *Raum und Zeit, ich empfind es, sind bloß Formen des Anschauns,* *Da das Eckchen mit dir, Liebchen, unendlich mir scheint.*
6) *Sorge! sie steiget mit dir zu Pferde, sie steiget zu Schiffe,* *Viel zudringlicher noch packet sich Amor mir auf.*	6) *Sorge! sie steiget mit dir zu Roß, sie steiget zu Schiffe;* *Viel zudringlicher noch packet sich Amor uns auf.*
7) *Schwer zu besiegen ist schon die Neigung, gesellet sich aber* *Gar die Gewohnheit zu ihr, unüberwindlich ist sie.*	7) *Neigung besiegen ist schwer; gesellet sich aber G Gewohnheit,* *Wurzelnd, allmählich zu ihr, unüberwindlich ist sie.*

[1] Die verwendeten Daten entstammen dem Korpus des TextGrid Repositories [74].

In diesem Beispiel wurden beide Gedichte nicht mehr zeilenweise eingelesen, sondern beide Werke wurden direkt indexiert und anschließend aligniert. Die so entstehenden Gaps (rot) zeigen nun anschaulich stilistische Unterschiede zwischen der Fassung von Schiller (links) und der von Goethe (rechts). Bei genauer Betrachtung fällt aber auch auf, dass die Alignierung hier nicht optimal funktioniert hat.

1. *Inkompatible Teilwörter:* Zum einen ist die erste Gap in Vers 1 zu groß, da dort das Wort *Amor* nicht erkannt worden ist. Die Ursache dieses Problems wurde bereits am Ende von Kapitel 8 erläutert. So geschieht es auch hier, dass in der linken Version *Amor* mit Endposition 22 zwar als gemeinsames Teilwort erkannt wird, dessen Pendant tritt jedoch in einem längeren Teilwort, *et sich Amor*, auf, was dazu führt, dass an dieser Stelle eine zu große Gap entsteht, die noch einer Nachbesserung bedarf.

2. *Überlappungen:* Ein zweites Problem ist in Vers 7 erkennbar. Dort wurde vor dem Wort *Gewohnheit* eine falsche Zeichfolge *G* eingefügt. Erneut liegt eine ähnliche Problematik wie in Vers 1 vor. In der linken Version lautet die Passage *...gesellet sich aber Gar die Gewohnheit...*, in der rechten *...gesellet sich aber Gewohnheit*. Dies führt dazu, dass die gemeinsamen Teilwörter *gesellet sich aber G* und *Gewohnheit* erkannt werden. In der linken Version bezieht sich das große *G* aus dem Teilwort *gesellet sich aber G* auf das Wort *Gar*, in der rechten auf *Gewohnheit*. Damit entsteht zwischen diesen beiden gemeinsamen Teilwörtern in der rechten Version eine Überlappung, die dazu führt, dass das fehlerhafte Zeichen eingefügt wird.

9.1.5. Behandlung von Überlappungen

Das erste in Beispiel 9.1.4 auftretende Problem kann nur schwer vermieden werden, da es technisch gesehen erst dann zu einem Problem wird, wenn die gemeinsamen Teilwörter in eine Ordnung gebracht werden, was dann dazu führt, dass wenn eine „zu lange“ Region gefunden wurde, diese verhindert, dass sie beziehungsweise ein Infix von ihr mit einer kürzeren Region in Verbindung gebracht wird.

Die Überlappungproblematik hingegen kann zu dem Zeitpunkt behandelt werden, wenn bereits eine längste gemeinsame Folge beziehungsweise ein Alignment gefunden wurde. Hierzu können die Elemente der gefundenen aufsteigenden Folge paarweise miteinander verglichen werden und durch die Bestimmung von Start- und Endpositionen Überlappungen zwischen den Teilwörtern erkannt werden. Wird eine Überlappung in einem der beiden Wörter gefunden, muss eines der beiden Teilwörter aus der LCS entfernt werden. Dies führt zwar wiederum dazu, dass übereinstimmende Zeichen nicht erkannt wurden, verhindert aber das fälschliche

Einfügen bestimmter Zeichenketten. Da die Implementierung dieses Vorgehens trivial ist, wird darauf verzichtet, den Pseudocode für eine mögliche Funktion *remove_overlaps* anzugeben. Die nachstehenden Illustrationen zeigen noch einmal die Überlappung aus Beispiel 9.1.4 sowie die gleiche Stelle, wenn zuvor die kürzere der überlappenden Regionen, (*Gewohnheit*), entfernt wurde.

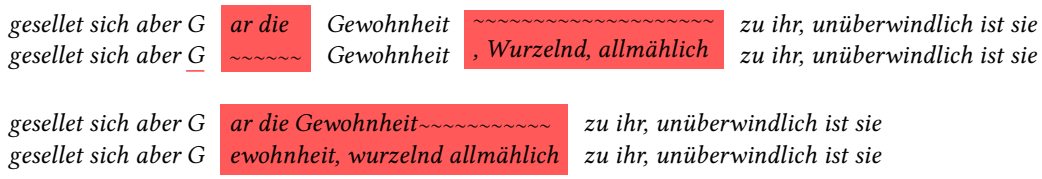
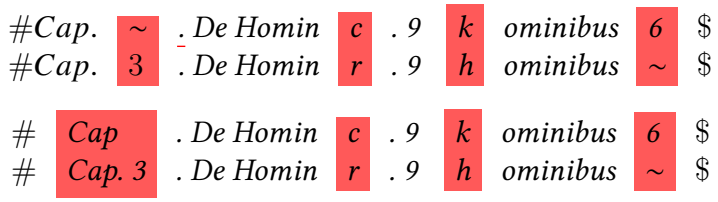


Abbildung 9.3.: Alignierte Zeilen mit überlappenden Teilwörtern und fehlerhafter Zeichenkette $G\sim$ … (oben) sowie ohne Überlappungen (unten).

Beispiel 9.1.5. Das nächste Beispiel zeigt noch einmal Überlappungen zwischen zwei verschiedenen OCR erkannten Zeilen, $w^1 = \#Cap.\ De\ Hominc.\ 9\ kominibus6\$$ und $w^2 = \#Cap.\ 3.\ De\ Hominr.\ 9\ hominibus\$$. Dort entsteht im ersten Wort durch das doppelte Vorkommen des Punktzeichens ein Überlapp zwischen den zwei Teilwörtern $\#Cap.$ und $.\ De\ Homin.$



Ähnlich wie im ersten Beispiel wird nun in w^1 ein falsches Zeichen (.) eingefügt. Entfernt man erneut die kürzere der beiden überlappenden Regionen, gehört das überlappende Zeichen zur ersten Gap, die dann allerdings wieder übereinstimmende Zeichen mit ihrer zugehörigen Gap enthält.

9.1.6. Nachkorrektur zu großer Gaps

Werden Überlappungen auf die hier beschriebene Art behoben, entstehen dadurch wie beim Problem der inkompatiblen Teilwörter weitere zu lange Gaps, die einer Nachkorrektur bedürfen. Die hier vorgestellte indexgestützte Alignment-Methode könnte somit als Basis-Alignment dienen, das anschließend weiter verfeinert werden kann. Hierzu müssten alle Gap-Paare überprüft werden und diese selbst wieder paarweise aligniert werden. Als Alignierungsmethode könnte zum Beispiel ein

Standardverfahren wie der Needleman-Wunsch-Algorithmus oder Hirschbergs Algorithmus dienen. Alternativ könnte aber auch wieder das SCDAWG-basierte Verfahren selbst eingesetzt werden. Damit würden iterativ alle entstehenden Gaps aligniert werden, was somit eine rekursive Prozedur beschreibt, die alle Lücken solange weiterzerlegt, bis keine Gemeinsamkeiten mehr existieren. Diese Idee ist in der nächsten Grafik bildlich dargestellt.

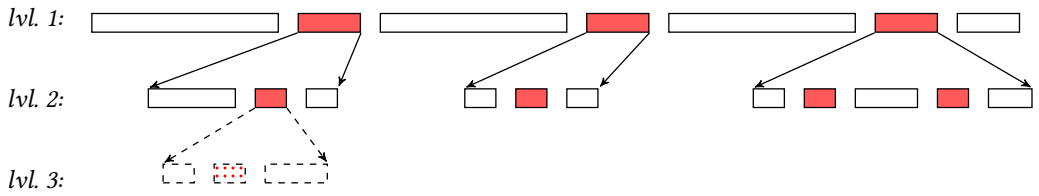


Abbildung 9.4.: Iterative Weiterzerlegung ungenau alignierter Gaps.

Dabei könnte, wenn die hier beschriebene indexbasierte Alignierungsmethode iterativ eingesetzt würde, folgendermaßen vorgegangen werden:

1. Erstelle mit Hilfe von *align* eine Basisalignierung zweier Wörter w^1, w^2.
2. Rufe *align* für alle Gaps auf und ersetze die jeweiligen Gaps durch die neuen Subalignments.
3. Wiederhole diese Prozedur solange, bis für keine Gap mehr eine weitere Subalignierung gefunden wird.

Wendet man diese Methodik auf das Gap-Paar in Abbildung 9.3 an, bei dem im Falle von Überlappungen das jeweils kürzere Teilwort entfernt wurde, entsteht für $W = \{\#ar\ die\ Gewohnheit\$, \#ewohnheit,\ Wurzelnd,\ allmählich\$\}$ für das greedy Cover diese Einordnung gemeinsamer Teilwörter:[2]

#	*a*	–	*i*	$
	r	*d*	–	
	–	*e*		
	–			
	ewohnheit			

Hieraus ergibt sich durch Algorithmus 20 nun wieder ein Alignierungsgraph, der mehrere gleich lange aufsteigende Folgen beinhaltet.

[2] Anstelle des Leerzeichens wird – verwendet.

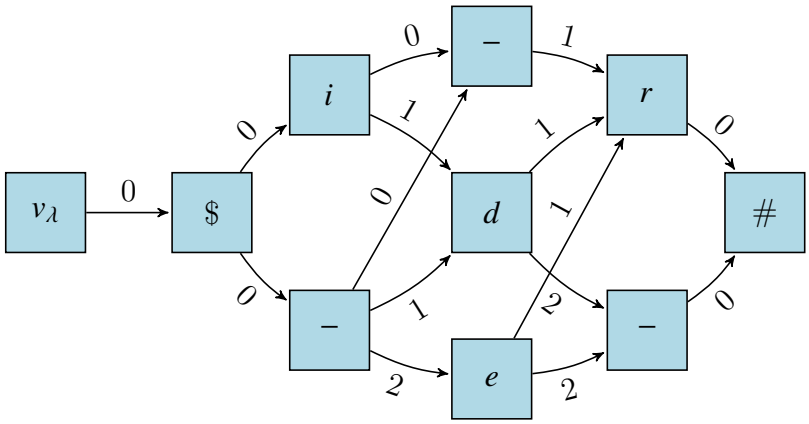


Abbildung 9.5.: Alignmentgraph des Gap-Paares.

Bereits auf den ersten Blick fällt auf, dass der Graph aus Abbildung 9.5 keinen Knoten für das längste gemeinsame Teilwort *ewohnheit* besitzt. Der Grund für diesen Umstand ist, dass innerhalb von w^1 dieses Teilwort ganz am Ende steht und in w^2 am Anfang. Zwischen diesen Vorkommen befinden sich nun aber weitere Zeichen, die längere gemeinsame Folgen innerhalb der beiden Wörter bilden, als dies mit dem Teilwort *ewohnheit* möglich ist, obwohl eine Folge mit diesem Teilwort insgesamt am meisten Zeichen beinhalten würde. Für den Alignierungsgraphen beziehungsweise den LCS-Algorithmus spielt die Länge eines darin enthaltenen Knotens aber kein Rolle, da er alle Knoten als gleichwertig behandelt.

Dieses Beispiel verdeutlicht, dass das indexbasierte Verfahren für die Alignierung kürzerer Wörter leicht auch fehlerhafte Alignierungen erzeugt. Wenn kurze Gaps zu erwarten sind, kann zur Nachkorrektur dieser aber auch ein rein zeichenbasiertes Verfahren benutzt werden.

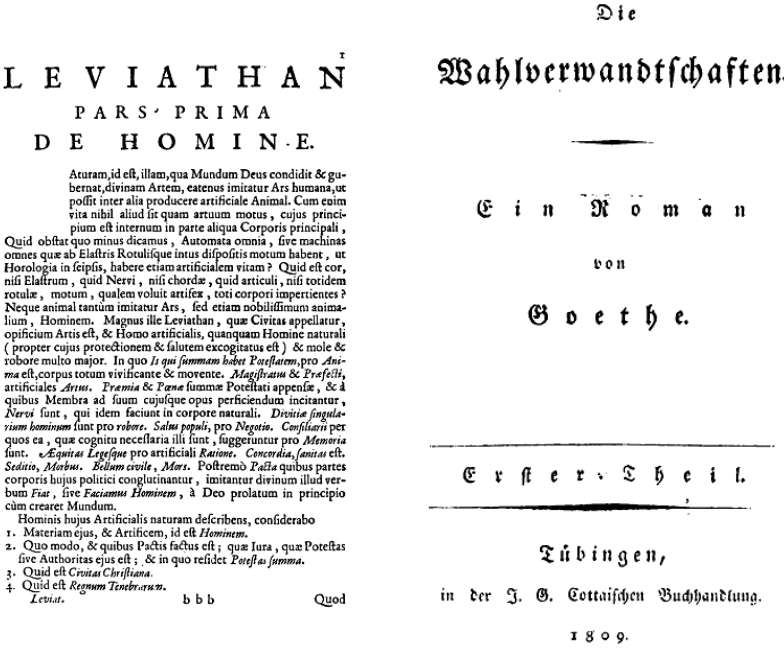
9.1.7. Komplexität und Fazit

Da dieses Verfahren eine Kombination aus SCDAWG und LCS-Algorithmus ist, bestimmt sich dessen Speicher- und Laufzeitbedarf aus diesen beiden zugrundeliegenden Komponenten, wenn bei der Verknüpfung der beiden Verfahren keine zusätzlichen Hürden entstehen. Zeit- und Speicherbedarf der (S)CDAWG-Struktur sind linear (siehe Kapitel 3.4) und die Laufzeit des LCS-Algorithmus (siehe Kapitel 2.5, Ende) beträgt im besten Fall $O(n \log n)$, wenn wenige Vorkommen gemeinsamer Zeichen des einen Strings im anderen existieren. Durch die Kombination der Indexstruktur mit diesem Algorithmus besteht das Alphabet des LCS-Algorithmus aus den als gemeinsam erkannten Teilwörtern, was bei vorhandenen Gemeinsam-

keiten oft dazu führt, dass die Laufzeit des LCS-Algorithmus stark sinkt. Im LCS-Algorithmus wurde die Anzahl der gesamten Vorkommen der Zeichen (hier gemeinsame Teilwörter) von w^1 in w^2 durch eine Konstante r ausgedrückt. Würde man beispielsweise durch zwei verschiedene OCR-Engines parallel erkannte Texte, die zusammen etwa 1.300 Zeichen lang sind, alleine mit Hilfe des LCS-Algorithmus alignieren, läge r bei etwa 130.000 Zeichen. Aufgrund der inhärenten Gemeinsamkeiten der Texte reduziert sich die Konstante r nun durch die Vorberechnung des SCDAWG-Index und die nachfolgende Identifkation der gemeinsamen Teilwörter auf etwa 30 Zeichen (Teilwörter) für die die längsten gemeinsamen Teilfolgen bestimmt werden müssen. Das Finden der gemeinsamen Teilwörter sowie die Bestimmung des finalen Alignments laufen ebenfalls linear ab, da sie Tiefensuchen auf gerichteten Graphen beschreiben. Dieser Effizienzgewinn birgt, wie die Beispiele 9.1.4 und 9.1.5 zeigen, den Nachteil, dass manche Gemeinsamkeiten nicht erkannt werden und an anderen Stellen Überlappungen auftreten. Während sich die Überlappungsproblematik nach der Erstellung des Alignierungsgraphen beheben lässt (siehe Abbildung 9.3 und Beispiel 9.1.5), können Ungenauigkeiten, die entweder durch inkompatible Teilwörter oder das Auflösen von Überlappungen entstehen, nur im Nachgang behoben werden. Hierbei ist es jedoch sicherer, ein konventionelles Verfahren zu verwenden, da, wie Abbildung 9.5 zeigt, vor allem bei kurzen Alignierungen eine suboptimale Zuordnung entstehen kann. Eine solche falsche Zuordnung kann allerdings auch bei längeren Eingaben durch die bereits erwähnten Problemfälle des indexbasierten Verfahrens entstehen. Zusätzlich kann die Unterstützung mehrerer LCS den Alignmentgraphen exponentiell wachsen lassen, wenn Texte aligniert werden, die immer wieder wenige unterschiedliche gemeinsamen Teilwörter besitzen. Dies wäre zum Beispiel bei der Alignierung eines deutschen Textes mit einem kyrillischen Text der Fall, da in diesen beiden Eingaben unter Umständen nur das Leerzeichen und einige Satzzeichen als gemeinsame Teilwörter auftreten. Dabei entstünde für den Alignmengraphen ein exponentielles Wachstum für alle Kombinationen von Leerzeichen, was natürlich vermieden werden sollte. Für diesen Fall kann jedoch auf die ursprüngliche LCS-Berechnung, wie sie in [31] beschrieben ist, zurückgegriffen werden. In [31] finden sich weiterführende Bemerkungen zur Komplexität des LCS-Algorithmus.

9.2. Alignierung von Dokumenten gleicher Sprache

Das SCDAWG-basierte Alignment-Verfahren liefert zwar nicht immer eine optimale Alignierung, für Alignment-Tasks, bei denen sehr viele Dokumente stapelweise verarbeitet werden sollen und in deren Eingabedaten hohe Übereinstimmungsraten zu erwarten sind, bietet die nahezu lineare Laufzeit des Verfahrens aber Vorteile. So wurde dieses Verfahren auch erfolgreich als Teil eines Systems, das im Zuge der OCR-D Initiative [57, 33] zur automatischen beziehungsweise halbautomatischen Postkorrektur historischer Dokumente [25] entwickelt wurde, erfolgreich eingesetzt.



LEVIATHAN

PARS PRIMA

DE HOMINE.

Aturam, id eſt, illam, qua Mundum Deus condidit & gubernat, divinam Artem, eatenus imitatur Ars humana, ut poſſit inter alia producere artificiale Animal. Cum enim vita nihil aliud ſit quam artuum motus, cujus principium eſt internum in parte aliqua Corporis principali, Quid obſtat quo minus dicamus, Automata omnia, ſive machinas omnes quæ ab Elaſtris Rotuliſque intus diſpoſitis motum habent, ut Horologia in ſeipſis, habere etiam artificialem vitam? Quid eſt cor, niſi Elaſtrum, quid Nervi, niſi chordæ, quid articuli, niſi totidem rotulæ, motum, qualem voluit artifex, toti corpori impertientes? Neque animal tantùm imitatur Ars, ſed etiam nobiliſſimum animalium, Hominem. Magnus ille Leviathan, quæ Civitas appellatur, opificium Artis eſt, & Homo artificialis, quanquam Homine naturali (propter cujus protectionem & ſalutem excogitatus eſt) & mole & robore multo major. In quo *Is qui ſummam habet Poteſtatem*, pro *Anima* eſt, corpus totum vivificante & movente. *Magiſtratus & Prefecti*, artificiales *Artus*. *Præmia & Pœnæ* ſummæ Poteſtati appenſæ, & à quibus Membra ad ſuum cujuſque opus perficiendum incitantur, *Nervi* ſunt, qui idem faciunt in corpore naturali. *Divitiæ ſingulariũ hominum* ſunt pro *robore*. *Salus populi*, pro *Negotio*. *Conſiliarii* per quos ea, quæ cognitu neceſſaria illi ſunt, ſuggeruntur pro *Memoria* ſunt. *Æquitas Legeſque* pro artificiali *Ratione*. *Concordia, ſanitas* eſt. *Seditio, Morbus*. *Bellum civile, Mors*. Poſtremò *Pacta* quibus partes corporis hujus politici conglutinantur, imitantur divinum illud verbum *Fiat*, ſive *Faciamus Hominem*, à Deo prolatum in principio cùm crearet Mundum.

Hominis hujus Artificialis naturam deſcribens, conſiderabo

1. Materiam ejus, & Artificem, id eſt *Hominem*.
2. Quo modo, & quibus Pactis factus eſt; quæ Iura, quæ Poteſtas ſive Authoritas ejus eſt; & in quo reſidet *Poteſtas ſumma*.
3. Quid eſt *Civitas Chriſtiana*.
4. Quid eſt *Regnum Tenebrarum*.

Leviat. bbb Quod

Die

Wahlverwandtſchaften.

Ein Roman

von

Goethe.

Erſter Theil.

Tübingen,

in der J. G. Cottaiſchen Buchhandlung.

1809.

Abbildung 9.6.: Scans der ersten Seiten aus Thomas Hobbes' „Leviathan“ (1651) und Goethes „Die Wahlverwandtschaften“ (1809).

Während in Kapitel 8.3 die Zeilen multipler OCRs historischer Dokumente einander zugeordnet wurden, sollen hier nun exakte paarweise Alignierungen bestimmter gleichsprachiger Einheiten (Zeilen, Paragraphen, Seiten, ganze Dokumente) erstellt werden. Da es für das Alignmentverfahren grundsätzlich keine Rolle spielt, ob dabei Zeilen, Seiten oder ganze Dokumente aligniert werden, wird hier zunächst seitenweise vorgegangen. Zuerst sollen nun die OCR-Ergebnisse dreier historischer Dokumente, die durch Tesseract und Abbyy-FineReader erkannt wur-

den, paarweise aligniert werden. Das erste Dokument stellt die Wochenzeitschrift „Die Grenzboten“ (1841) mit 520 Kilobyte dar. Zudem werden die ersten 96 Seiten aus Thomas Hobbes’ „Leviathan“ (1651) mit 440 Kilobyte sowie Johann Wolfgang von Goethes „Die Wahlverwandtschaften“ (1809) mit 306 Seiten und 1,19 Megabyte jeweils in zwei OCR-Ergebnissen aligniert. Insgesamt werden somit 503 Seiten paarweise aligniert[3]. Die nächste Abbildung zeigt zwei alignierte OCR-Seiten aus Hobbes’ „Leviathan“.

Cap. i. De Homtttc. 3C A P V T I.De Senfu, Ogitationes hominum primo fi gillatim , deinde alias ab aliis dc- pendentes, ut in fe rie confi derabo. I pfarum unaquaeque , alicu-jus qualic atis vel accidentis in Corpore externo, quod appellari l oletObjeft um, eft Apparitio fi ve Reprefentatt o. Quo d Objedum ag endo incorporis bumani Organa, nempe Oculos, Aures, & c. pro aiverfi ta-te Ad ionis diverfas producit Apparitiones! Origo omnium nora inatur Senfus . Nulla enim eft Animi conceptioquae non fuerat ante genita in aliquo Senfu um , vel tota fi mul, vel perpartes. Ab his autem primis conceptibus omnes poft ea derivantur.Cognitio caufae Sentiendi naturalis prae fenti inft ituto non eft ab-s olute necefl ariaj etiam alio loco deilia Sati s ampliter trad avimus.Veruntamen , ut praefe ntis Methodi part es impleantur , de eadem reiterum fe d breviter difl eremus.Caufa Senfionis eft Externum Corpus fi ve Objett um quod premie unius cujufque Organum proprium, vel immediate, ut in fe nfu Tattut& Gufiut , vel mediate, ut in PtfU

#~ Cap. x. De Homine. 3C A P V T I.De Senfit. Ogitationes hominum primo fi gillatim , deinde alias ab aliis de-_ pendentes, 11: in {E rie confi derabo. I pfarum unaquaeque , alicu-jus qualit atis vel accidentis in Corpore externo, quod appellari f oletObjefl um, efi Apparitio fi ve Reprefenmti o. 030 d Objeéfum ao endo incorporis bumani Organa, nempe Oculos, Aures, 85 c. pro d’iverfi tate Aa ionis diverfas producit Apparitionesi Origo omnium nom inatur Sen/m . Nulla enim eff Animi conceptioqua: non fuerat ante genita in aliquo SenfiJ um , vel tota fi mul, vel perpartes. Ab his autem primis conceptibus omnes pofi ea derivantur.Cognitio caufa: Sentiendi naturalis frz fenti inPc ituto non efl: ab-f olute necefl aria; etiam alio loco dei la em s ampliter traa avimus.Veruntamen , ut praelé ntis Methodi pan es impleantur , de eadem reiterum {E d breviter diff eremus.Caufa smfianu efl: Externum Corpus fi ve Objefl um quod premix: unius cujufque Organum proprium, vel immediate, at in {E nfu Tac’bu8.: Gnflm , vel mediate, ut in 75]»)

Abbildung 9.7.: Auszug zweier alignierter Seiten aus „Leviathan“.

Es ist wiederum erkennbar, dass so keine optimale Alignierung gefunden werden konnte, da sich innerhalb einiger Gaps noch gemeinsame Zeichen befinden[4]. Zur Auswertung wird die indexbasierte Alignierungsmethode mit Hirschergs Al-

[3]Die verwendeten OCR-Daten wurden von Dr. Uwe Springmann bereitgestellt, sie sind unter [26] frei verfügbar.

[4]Manche Gaps sind hier auf unterschiedlich codierte Zeichen zurückzuführen.

gorithmus[5], der auf dynamischer Programmierung basiert, verglichen beziehungsweise mit ihm kombiniert. $align_{SC}$ steht für das SCDAWG-gestützte Verfahren aus Algorithmus 20. $align_H$ bezeichnet Hirschbergs Methode und $align_{SC+H}$ steht für die Kombination aus beiden Varianten, in der erst durch Algorithmus 20 ein Basis-Alignment erstellt wird, dessen Gaps anschließend durch Hirschbergs Algorithmus aligniert werden (siehe Abbildung 9.4). Als viertes Verfahren wird das indexbasierte Verfahren selbst eingesetzt, um iterativ die Gaps seiner Basisalignierung zu korrigieren, dieses Verfahren wird als $align_{SC+IT}$ bezeichnet.

Zunächst folgt ein Blick auf Zeit- und Speicherverbrauch, wenn mit den vier Methoden die drei Bücher seitenweise aligniert werden. Die Laufzeit wird in Sekunden angegeben, der Speicherverbrauch in Megabyte. Es werden Durchschnittswerte aus 50 wiederholten Versuchen angegeben.

	$align_H$	$align_{SC}$	$align_{SC+H}$	$align_{SC+IT}$
„Die Grenzboten“ (1841), 101 Seiten				
Laufzeit in Sek.	4,1	0,76	0,82	0,82
Speicher in MB	125	77	136	154
„Leviathan“ (1651), 96 Seiten				
Laufzeit in Sek.	3,1	0,76	0,79	0,96
Speicher in MB	116	101	133	240
„Die Wahlverwandtschaften“ (1809), 306 Seiten				
Laufzeit in Sek.	1,46	0,72	0,75	0,78
Speicher in MB	243	122	158	321
Gesamt, 503 Seiten				
Laufzeit in Sek.	8,6	2,3	2,3	2,6
Speicher in MB	484	300	427	715

Tabelle 9.2.: Zeit- und Speicherverbrauch bei seitenweiser Alignierung der drei Testdokumente durch vier Alignierungsverfahren.

Die hier dargestellte Tabelle gibt, da eine Java-Implementierung zugrunde liegt, keine exakten Speicherwerte wieder, diese zeigen aber, dass alle vier Methoden in etwa gleich viel Speicher benötigen, um die Alignierungen durchzuführen. Die quadratische Zeitkomplexität von $align_H$ ist verglichen mit den indexbasierten Methoden klar zu erkennen. Bei vergleichsweise kurzen Eingaben, wie den hier

[5] Als Referenz kann diese frei verfügbare Implementierung in der Programmiersprache Java betrachtet werden [28].

verwendeten Seiten, bleibt sie mit insgesamt etwa neun Sekunden aber akzeptabel. Obwohl das dritte Dokument die meisten Seiten beinhaltet, wird es von $align_H$ im Vergleich zu den beiden anderen Dokumenten am schnellsten aligniert. Dies liegt an der höheren Segmentierung, die durch die größere Seitenanzahl entsteht. Bei allen SCDAWG-basierten Verfahren spielt dies keine Rolle. Verglichen mit der Zeit, die $align_{SC}$ benötigt, fällt die Nachkorrektur von $align_{SC+H}$ und $align_{SC+IT}$ kaum ins Gewicht. Die hier entstehenden Gaps sind so kurz, dass auch die Laufzeit des Hirschberg-Verfahrens dort keine Verlangsamung bringt. Bei $align_{SC+IT}$ werden nach einer einmaligen Korrektur nur selten weitere Zerlegungen nötig, meist werden also, wenn Gaps aligniert werden, diese so zugeordnet, dass keine weiteren Übereinstimmungen innerhalb eines Gap-Paares entstehen.

9.2.1. Alignment-Qualität

Zur Auswertung der Qualität der indexbasierten Alignment-Verfahren werden diese nun mit $align_H$ verglichen, da letzteres Verfahren ein optimales Alignment zweier Wörter w^1 und w^2 findet und dementsprechend als Goldstandard angesehen werden kann. Hierzu wird die folgende Berechnung zugrundegelegt. Die Güte einer indexbasierten Alignierung wird als *alignment-ratio* oder kurz *ar* bezeichnet. Diese ergibt sich aus dem Verhältnis der durchschnittlichen *match-ratios* von $mr^{\varnothing}_{SC}$ zu $mr^{\varnothing}_{H}$. $mr^{\varnothing}_{H}$ gibt das durchschnittliche Verhältnis der längsten gemeinsamen Teilfolge lcs_H, die $align_H$ erkennt, zur Gesamtlänge der beiden Wörter an. Auf die gleiche Art gibt $mr^{\varnothing}_{SC}$ das Verhältnis der LCS, die durch ein SCDAWG-basiertes Verfahren gefunden wird, zur Gesamtlänge der Eingabewörter an.

$$mr^{\varnothing}_{H} = \frac{\frac{|lcs_H|}{|w^1|} + \frac{|lcs_H|}{|w^2|}}{2} \qquad mr^{\varnothing}_{SC} = \frac{\frac{|lcs_{SC}|}{|w^1|} + \frac{|lcs_{SC}|}{|w^2|}}{2}$$

$$ar = \frac{mr^{\varnothing}_{SC}}{mr^{\varnothing}_{H}} \qquad ar^{\varnothing} = \frac{1}{n}\sum_{i=1}^{n} ar_i = \frac{ar_1 + ar_2 + \cdots + ar_n}{n}$$

Die nächste Tabelle zeigt das durchschnittliche Verhältnis von mr_{SC} zu mr_H, das für alle drei indexbasierten Alignierungsverfahren für alle Seiten der Dokumente bestimmt wurde. Somit bezeichnet $ar^{\varnothing}$ das arithmetische Mittel der *ar* der Seiten eines Dokuments. Gleichfalls werden erneut die summierten Ergebnisse dargestellt.

	$align_{SC}$	$align_{SC+H}$	$align_{SC+IT}$
„Die Grenzboten“ (1841), 101 Seiten			
$ar^{\varnothing}$	98,3%	99,6%	99,5%
opt	3/101	94/101	78/101
„Leviathan“ (1651), 96 Seiten			
$ar^{\varnothing}$	93,1%	99,4%	99,4%
opt	2/96	50/96	12/96
„Die Wahlverwandtschaften“ (1809), 306 Seiten			
$ar^{\varnothing}$	96,3%	99,3%	99,1%
opt	4/306	239/306	183/306
Gesamt, 503 Seiten			
$ar^{\varnothing}$	95,9%	99,4%	99,3%
opt	9/503	383/503	273/503

Tabelle 9.3.: Durchschnittliche *ar*-Werte und Anzahl optimaler Alignments bei paarweiser Alignierung der Seiten der drei Testdokumente.

Die Güte der Alignierungen von $align_{SC}$ liegt insgesamt bei etwa 95% richtig alignierter Seiten. In allen drei Experimenten zeigt sich, dass durch die nachgeschaltete Weiterzerlegung aller Gaps für alle drei Dokumente durchschnittliche Alignmentwerte von über 99% erreicht werden. Dabei schneiden $align_{SC+H}$ und $align_{SC+IT}$ etwa gleich gut ab. Dass keines der letztgenannten Verfahren dazu führt, dass immer optimale Alignierungen gefunden werden, ist auf das Problem der inkompatiblen Teilwörtern zurückzuführen, das bewirkt, dass letztlich nicht alle „zusammenpassenden“ Gaps Paare bilden, und so eine korrekte Subalignierung gefunden werden kann. Die jeweiligen Einträge in *opt* zeigen, wie viele der gefundenen Alignments optimal aligniert wurden. Auffällig ist dabei, dass $align_{SC}$ kaum auf Anhieb eine optimale Alignierung findet, während durch die verbesserten Verfahren diese Zahl ansteigt. Am besten schneidet dabei $align_{SC+H}$ ab, dessen optimale Subalignierungen insgesamt zu einem besseren Ergebnis gegenüber der indexabasierten Subalignierung führen.

9.2.2. Alignierung gesamter Dokumente

Nicht immer stehen Daten in der hier vorliegenden Segementierung (Zeilen) zur Verfügung, sondern es liegt eine gröbere Einteilung vor, die es nötig macht, größere Datenbestände auf einmal zu alignieren. Deshalb wird an dieser Stelle das obige

Experiment noch einmal wiederholt, anstelle einer seitenweisen Alignierung werden nun aber die gesamten Dokumente paarweise aligniert. In der entsprechenden Auswertungstabelle tritt die quadratische Zeitkomplexität von $align_H$ noch deutlicher zu Tage. Während bei zeilenweiser Alignierung auch $align_H$ nur wenige Sekunden benötigt hat, braucht es bei einer dokumentenweisen Alignierung nun schon mehrere Minuten. Insgesamt benötigt $align_H$ für die Alignierung aller Dokumente 1071 Sekunden beziehungsweise 17,85 Minuten. Die Laufzeiten der indexbasierten Verfahren hingegen sind zwar ebenfalls etwas angestiegen, sie benötigen aber wiederum nur wenige Sekunden, um eine globale Alignierung eines der drei Dokumentenpaare zu erstellen. Der Speicherverbrauch von $align_H$ ist mit dem seiner seitenweisen Alignierung zu vergleichen. Bei den SCDAWG-gestützten Methoden ist hier ein allgemeiner Anstieg des Speicherverbrauchs zu erkennen, da wenn größere Eingabestrings auf einmal aligniert werden, mehr Speicher für den Aufbau der Indexstruktur gebraucht wird.

	$align_H$	$align_{SC}$	$align_{SC+H}$	$align_{SC+IT}$
„Die Grenzboten“ (1841), 101 Seiten				
Laufzeit in Sek.	447	2,4	2,5	2,6
Speicher in MB	243	340	363	395
„Leviathan“ (1651), 96 Seiten				
Laufzeit in Sek.	261	4,8	5,2	5,5
Speicher in MB	182	1154	1197	1292
„Die Wahlverwandtschaften“ (1809), 306 Seiten				
Laufzeit in Sek.	363	3,5	3,8	3,9
Speicher in MB	97	797	836	913
Gesamt, 503 Seiten				
Laufzeit in Sek.	1071	10,7	11,5	12
Speicher in MB	522	2,291	2,396	2,600

Tabelle 9.4.: Zeit- und Speicherverbrauch bei dokumentenweiser Alignierung der drei Testdokumente durch vier Alignierungsverfahren.

Vergleicht man $align_{SC}$, $align_{SC+H}$ und $align_{SC+IT}$ untereinander, fällt erneut auf, dass kaum ein Unterschied in Bezug auf die Zeitdauer besteht, wenn eine nachgeschaltete Korrektur der Gaps hinzukommt. Die unterschiedlichen Speicherwerte bezüglich der drei Dokumente ist auf die interne Struktur dieser zurückzuführen. Im zweiten Dokument entsteht beispielsweise eine größere SCDAWG-Struktur, obwohl dieses Buch weniger Seiten beinhaltet als die anderen beiden Dokumente.

	$align_{SC}$	$align_{SC+H}$	$align_{SC+IT}$
„Die Grenzboten“ (1841), 101 Seiten			
ar	97,9%	99,9%	99,9%
„Leviathan“ (1651), 96 Seiten			
ar	84,5%	99,6%	99,2%
„Die Wahlverwandtschaften“ (1809), 306 Seiten			
ar	91,5%	99,8%	99,5%
Gesamt, 503 Seiten			
ar	91,3%	99,5%	99,4%

Tabelle 9.5.: *ar*-Werte der SCDAWG-basierten Alignierungsverfahren bei paarweiser Alignierung der drei Testdokumente.

Ein erneutes Bestimmen der *alignment_ratio* zeigt, dass auch bei paarweiser Alignierung der gesamten Dokumente eine gute Alignierungsqualität erreicht wird, die stets durch die nachgeschaltete Gap-Korrektur verbessert wird. Die größte Diskrepanz weist dabei $align_{SC}$ bei Thomas Hobbes' „Leviathan“auf. Hier steigt der *ar*-Wert von 84,5% auf über 99%. Dies gibt auch einen Hinweis darauf, dass die beiden OCRs dieses Dokuments insgesamt größere Abweichungen enthalten und damit eine größere Anzahl an Gaps nachkorrigiert werden muss. Wie Tabelle 9.4 jedoch zeigt, ist der dafür gebrauchte Zeitbedarf nicht signifikant. Eine optimale Alignierung wurde für keines der drei Dokumente erreicht.

Alignierung deutscher Bibelübersetzungen

Das folgende Teilexperiment zeigt die Anwendung des indexgestützten Alignment-Verfahrens im Kontext eines Dissertationsprojekts aus dem Fachbereich der germanistischen Linguistik. Ziel dieses Projektes ist es, den Einfluss von Martin Luthers Bibelübersetzung (1522) anhand des zweiten Kapitels des Lukasevangeliums auf die Entwicklung der deutschen Sprache zu untersuchen[6]. Hierzu wird diese Referenzübersetzung mit sieben jüngeren und älteren Übersetzungen verglichen. Ein dabei verwendeter Ansatz ist die paarweise Alignierung der Referenzübersetzung mit den anderen Übersetzungen, sodass anhand dieser Alignments Rückschlüsse auf Ähnlichkeiten in Textstruktur und Satzkonstruktion gezogen werden können.

[6]Die hier benutzten Daten stammen aus der aktuell entstehenden Dissertation „Sprechen wir heute noch wie Martin Luther? - Stellenwert der Lexik Luthers im deutschen Wortschatz der Gegenwart - aufgezeigt an Beispielen aus dem 2. Kapitel des Lukasevangeliums“ von Ursula Meier-Credner.

Entstehen innerhalb eines Alignments zweier Übersetzungen eher wenige lange Gaps, so kann geschlossen werden, dass dieses Paar viele ähnlich strukturierte Sätze enthält. Dies ergibt sich zum Beispiel bei einem Vergleich mit einer modernen Übersetzung der Luther'schen Bibel mit der Referenzübersetzung von 1522, obwohl beide Texte eine stark unterschiedliche Orthographie aufweisen. Die gleiche Vermutung liegt zugrunde, wenn ein Alignment der Referenzübersetzung mit einer anderen historischen, aber jüngeren Übersetzung ebenfalls vergleichsweise kurze Gaps aufweist. Auch dies kann wieder als ein Hinweis interpretiert werden, dass Zusammenhänge bestehen, die auf eine Beeinflussung durch die Referenzbibel zurückzuführen sind. Der Vergleich mit älteren Übersetzungen dient als Gegenbeispiel, da diese Texte nicht von der Referenzübersetzung beeinflusst wurden und somit vergleichsweise längere Gaps entstehen müssten.

An dieser Stelle folgt die Auswertung dreier paarweiser Alignierungen verschiedener Bibelübersetzungen. Dabei werden verglichen

1. Luther (1522) $\longleftrightarrow$ Mentelin (1466),
2. Luther (1522) $\longleftrightarrow$ Eck (1528),
3. Luther (1522) $\longleftrightarrow$ Luther (2017).

Das erste Paar stellt den Vergleich der historischen Lutherbibel mit einer älteren historischen Bibelübersetzung dar, das zweite den der Referenzübersetzung mit einer jüngeren historischen Übersetzung und das dritte den Vergleich der historischen Lutherbibel mit der modernen Luther-Übersetzung.

Da die hier zur Verfügung gestellten Daten nur Auszüge aus des Übersetzungen des zweiten Lukasevangeliums darstellen, spielen Zeit- und Speicherverbrauch keine entscheidende Rolle. Somit folgt erneut der Blick auf die Qualität der erzeugten Alignierungen, die wie in Experiment 3 anhand der optimalen Alignierung von $align_H$ gemessen wird.

	$align_{SC}$	$align_{SC+H}$	$align_{SC+IT}$
Luther (1522) ⟷ Mentelin (1466)			
ar	38,4%	92,8%	76,5%
Luther (1522) ⟷ Eck (1528)			
ar	67,9%	99,8%	96,1%
Luther (1522) ⟷ Luther (2017)			
ar	50,3%	94,6%	90,0%
Gesamt			
ar	52,2%	95,7%	87,5%

Tabelle 9.6.: *ar*-Werte der SCDAWG-basierten Alignierungsverfahren bei paarweiser Alignierung der drei verglichenen Bibelübersetzungen.

Im Vergleich mit den alignierten OCR-Daten sinkt hier vor allem die Güte der Basisalignierung von $align_{SC}$ ab. Über die drei Tests hinweg werden so nur 52,2% richtig aligniert. Jedoch steigt die Rate richtig alignierter Teilwörter wieder an, wenn $align_{SC+H}$ oder $align_{SC+IT}$ verwendet werden. Erneut schneidet $align_{SC+H}$ am besten ab, was insgesamt zu 95,7% richtigen Alignments führt. Die nachfolgenden Darstellung zeigen verkleinerte Visualisierungen der drei Alignierungen, wenn $align_{SC+H}$ verwendet wird. Die enstandenen Gaps sind wie zuvor rot markiert, Übereinstimungnen blau. Auf den ersten Blick ist erkennbar, dass die ältere Mentelin-Bibel weniger Gemeinsamkeiten mit der Luther-Bibel von 1521 aufweist als die beiden anderen Übersetzungen.

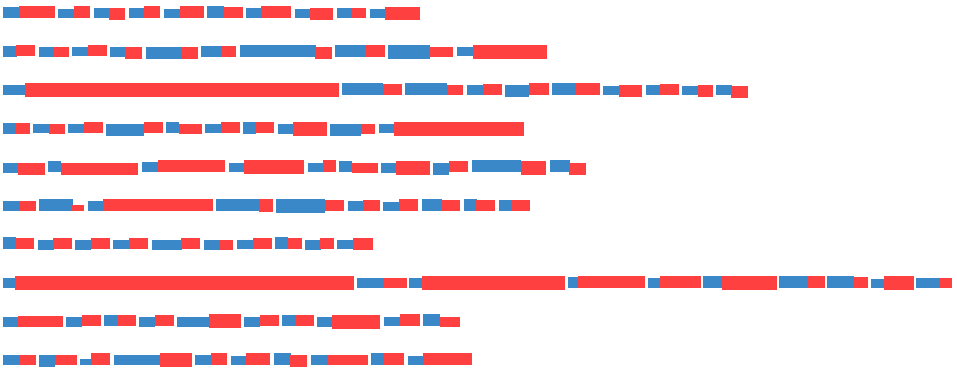


Abbildung 9.8.: Verkleinerte Darstellung der paarweisen Alignierung von Luther (1522) ⟷ Mentelin (1466).

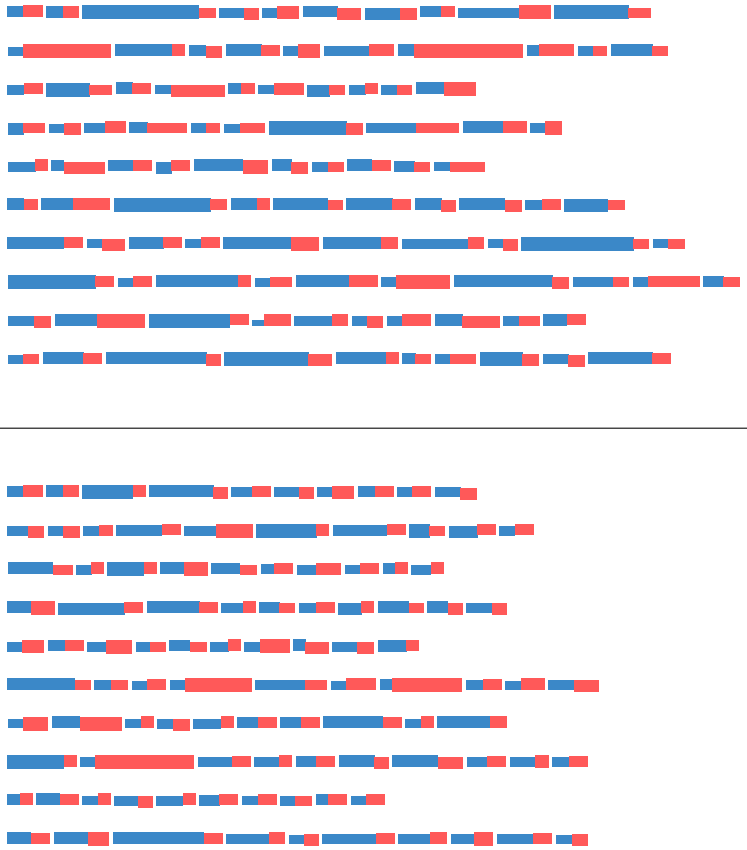


Abbildung 9.9.: Verkleinerte Darstellung der Alignierung von Luther (1522) ⟷ Eck (1528) oben und Luther (1522) ⟷ Luther (2017) unten.

Bestimmt man das Verhältnis zwischen Matches (blau) und Gaps (rot) ergibt sich

für Luther (1522) ⟷ Mentelin (1466) ∼ 1 : 1,

für Luther (1522) ⟷ Eck (1528) ∼ 2 : 1,

und für Luther (1522) ⟷ Luther (2017) ∼ 3 : 1.

Zu den beiden 1528 und 2017 erstellten Werken existiert somit mit zwei- oder dreimal so vielen Übereinstimmungen wie Abweichungen ein höherer Grad an Kohärenz, während bezüglich der älteren Fassung Matches und Gaps etwa im selben Verhältnis auftreten.

9.2.3. Alignierung stark differierender Dokumente

Während in den bisher verglichenen Datensätzen Dokumente aligniert wurden, deren Länge und Textstruktur sehr ähnlich waren, sollen abschließend die SCDAWG-basierten Alignierungsverfahren auf einen Datensatz angewendet werden, in dem größere Abweichungen als bisher zu erwarten sind. Hierfür werden Transliterationen von Fragmenten von Keilschriftzylindern aus der Zeit des babylonischen Herrschers Nabonid (Regierungszeit 556-539 v. Chr) aligniert.

Die nächste Abbildung zeigt die Rekonstruktion eines solchen Keilschriftzylinders, der im British Museum in London ausgestellt ist.

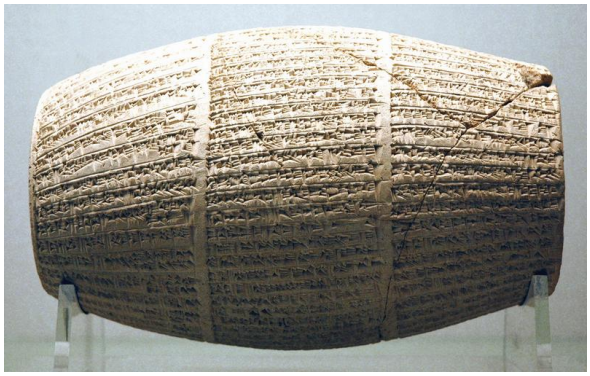


Abbildung 9.10.: Dreispaltiger Keilschriftzylinder aus der Kollektion des British Museum [16].

Die hier betrachteten 51 Zylinder enthalten alle Inschriften desselben Textes. Durch die Anfertigung solcher Kopien der gleichen Inschrift wurde sichergestellt, dass sowohl die Götter als auch mögliche Nachfahren darüber informiert wurden, unter welcher Herrschaft ein Tempel erbaut worden war. Damit war es also das Ziel, diese Keilschriftzylinder in möglichst großer Zahl herzustellen, um diese dann in den Fundamenten und Wänden eines neuen Tempels einzumauern.

Da die Tonzylinder jeweils nur in Fragmenten erhalten sind, führt deren unterschiedlicher Erhaltungszustand zu stark differierenden Textlängen. Diese fragmentierten Texte werden mit einem manuell zusammengesetzten Referenztext aligniert, der die vollständige Inschrift enthält[7]. Hieraus lassen sich Rückschlüsse gewinnen, in welchen Bereichen die jeweiligen Fragmente innerhalb des zusammengesetzten Haupttextes übereinstimmen und in welchen Bereichen nicht. Der Einfachheit halber werden die drei Spalten, in die sich die Texte der Zylinders gliedern, zu einem Text zusammengefasst. Damit existiert für jeden Zylinder ein Text,

[7]Die verwendeten Daten wurden von Dr. Frauke Weiershäuser und Dr. Jamie Novotny zur Verfügung gestellt.

der direkt mit dem Haupttext verglichen wird. Für eine genauere Untersuchung könnte jedoch auch die ursprüngliche Spalteneinteilung beibehalten werden und so könnten die Fragmente spaltenweise miteinander aligniert werden. Die nachstehende Abbildung zeigt Matches (weiß) und Missmatches (rot) eines der Fragmente innerhalb des Referenztextes. Dabei fallen drei Bereiche auf, die größere Übereinstimmungen besitzen und somit die Sektionen markieren, in welchen das hier betrachtete Zylinderfragment erhalten geblieben ist.

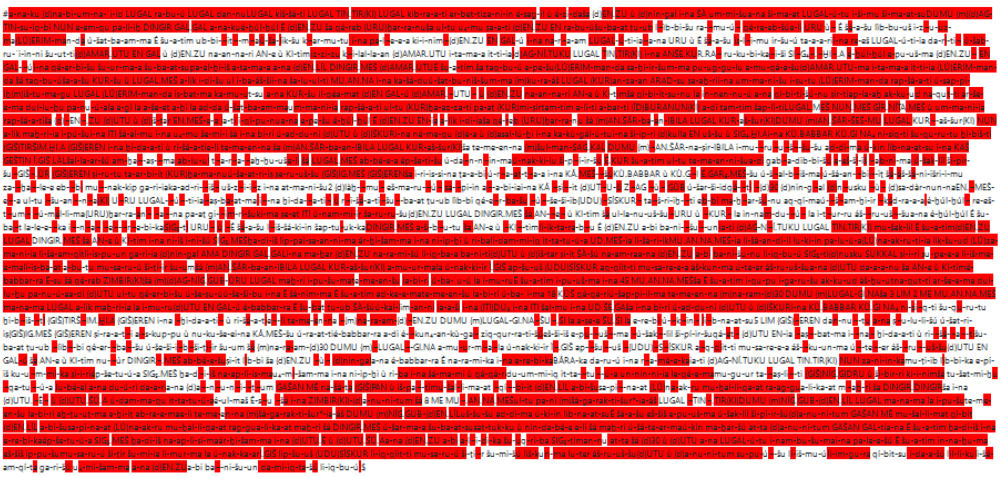


Abbildung 9.11.: Verkleinerte Darstellung einer Alignierung des Komposittexts mit einem der Keilschriftzylinder.

Das in diesem Beispiel benutzte Alignmentverfahren ist $align_{SC+H}$, also die Kombination aus SCDAWG-basierter Alignierung und Hirschbergs optimalem Verfahren. In der folgenden Tabelle wird erneut die durchschnittliche Alignierungsqualität der 51 paarweisen Alignierungen, die hier durchgeführt wurden, angegeben.

	$align_{SC}$	$align_{SC+H}$	$align_{SC+IT}$
Gesamt, 51 Dokumentenpaare			
$ar^{\varnothing}$	55,9%	88,4%	83,1%

Tabelle 9.7.: Durchschnittliche ar-Werte bei paarweiser Alignierung der 51 Tonzylinder.

Auch innerhalb dieses Datensatzes bestätigt sich das Bild der vorherigen Versuche. Die Korrektheit der paarweisen Alignierungen liegt jedoch etwas niedriger.

Dies war jedoch erwartbar, da aufgrund der teilweise erheblichen Textvarianz die Wahrscheinlichkeit, eine „falsche“ längste aufsteigende Folge als Basisalignment zu erhalten, steigt. Trotzdem werden auch in diesem Datensatz in vielen Fällen nahezu optimale Alignierungen gefunden, wobei erneut das kombinierte Verfahren $align_{SC+H}$ am besten abschneidet.

9.2.4. Fazit

Wenngleich keines der indexgestützten Alignmentverfahren optimal gearbeitet hat, so zeigt sich in allen Versuchen, dass für Anwendungen, die keine zu 100% exakte Alignierung benötigen, diese Verfahren interessante Alternativen darstellen, da bessere Laufzeiten zu erwarten sind als etwa bei Hirschbergs Verfahren. Damit wäre auch ein Einsatz bei der Erkennung von Plagiaten denkbar, bei der ein Text mit einer Vielzahl von Dokumenten verglichen wird.

9.3. Alignierung von Dokumenten verschiedener Sprachen

Neben der Erkennung von Gemeinsamkeiten innerhalb von Dokumenten, die in der gleichen Sprache verfasst wurden, ist auch die Erkennung von Gemeinsamkeiten zwischen in unterschiedlichen Sprachen geschriebenen Werken von großem Interesse. Während bei der maschinellen Übersetzung oft andere Alignment-Techniken, die probabilistisch arbeiten [58], verwendet werden, kann auch ein auf exakten Übereinstimmungen basiertes Verfahren bei der Abbildung von Texten verschiedener Sprachen aufeinander hilfreich sein. Matches könnten so zum Beispiel Eigennamen oder andere sprachübergreifende Wörter darstellen, die mit Hilfe von globaler Aliginierung erkannt werden würden. Solche Alignments könnten auch dann von Belang sein, wenn für eine der zu vergleichenden Sprachen bisher nur wenige Analysewerkzeuge wie part-of-speech-Tagger oder annotiertes Material zur Verfügung steht. Ein Ansatz, gemeinsame sprachliche Phänomene zwischen solchen *low-resource*-Sprachen zu finden, ist in [2] beschrieben. Die dort verwendeten Daten stammen aus dem Parallel-Bible-Corpus (PBC), das Übersetzungen des Neues Testaments in über 1000 Sprachen enthält[8]. Testweise werden hier vier Sprachpaarungen aus dem PBC-Korpus satzweise mithilfe der Alignmentverfahren aligniert und verglichen. Die Alignmentpaare setzen sich zusammen aus

[8] Das hier beschriebene Experiment wurde auf Initiative von Prof. Dr. Hinrich Schütze durchgeführt.

1. Deutsch (DEU) ⟷ Englisch (ENG),
2. Deutsch (DEU) ⟷ Französisch (FRA),
3. Deutsch (DEU) ⟷ Seychellenkreol (CRS),
4. Französisch (FRA) ⟷ Seychellenkreol (CRS).

Da bei satzweiser Alignierung die Zeitkomponente keinen großen Einfluss hat, finden sich in Tabelle 9.8 die erzielten Werte bezüglich der Alignierungsqualität für $align_{SC}$, $align_{SC+H}$ und $align_{SC+IT}$.

	$align_{SC}$	$align_{SC+H}$	$align_{SC+IT}$
DEU ⟷ ENG, 30951 Zeilen			
$ar^{\varnothing}$	52,7%	78,9%	76,9%
opt	38/30.951	772/30.951	613/30.951
DEU ⟷ FRA, 30951 Zeilen			
$ar^{\varnothing}$	51,5%	77%	75,1%
opt	107/30.951	592/30.951	446/30.951
DEU ⟷ CRS, 30951 Zeilen			
$ar^{\varnothing}$	51,4%	76,9%	75,1%
opt	85/30.951	730/30.951	618/30.951
FRA ⟷ CRS, 30951 Zeilen			
$ar^{\varnothing}$	58,9%	83,7%	82%
opt	167/30.951	1540/30.951	1219/30.951
Gesamt, 123804 Zeilen			
$ar^{\varnothing}$	53,6%	79,1%	77,2%
opt	397/123.804	3634/123.804	2896/123.804

Tabelle 9.8.: Durchschnittliche ar-Werte und Anzahl optimaler Alignments bei paarweiser Alignierung der Zeilen der vier Sprachpaare.

Die Ergebnisse zeigen im Vergleich zu den vorherigen Experimenten deutlich gesunkene Werte. Die beste Alignmentqualität bietet erneut $align_{SC+H}$. Insgesamt wird aber auch von diesem Verfahren nur in etwa 2,9% der Fälle eine optimale Alignierung gefunden, die der von $align_H$ entspricht. Von den verglichenen Sprachpaaren entstehen für FRA ⟷ CRS die besten Werte mit durchschnittlich 83,7% für $align_{SC+H}$. Dies ist mit der Tatsache erklärbar, dass Seychellenkreol sich aus

dem Französischen entwickelte, weshalb dort die größten Übereinstimmungen aller Vergleichspaare gefunden werden. Es zeigt sich also, dass wie in Kapitel 9.1 bereits beschrieben (siehe Abbildung 9.5), bei kurzen Eingabewörtern, die nur wenige Gemeinsamkeiten teilen, die SCDAWG-basierte Alignierung schlechtere Ergebnisse liefert. Liegen die Daten bereits in Zeilenform vor, so entfällt auch der Geschwindigkeitsvorteil der SCDAWG-basierten Alignmentverfahren, da dann die quadratische Zeitdauer von $align_H$ keine große Auswirkung mehr hat.

9.3.1. Bestimmung lokaler Intervalle

Es soll nun mit einer einfachen Heuristik auf den durch $align_{SC+H}$ entstandenen Alignments versucht werden, Subregionen zu bestimmen, die gut zusammenpassen und damit wie eingangs erwähnt Eigennamen oder andere Gemeinsamkeiten der verschiedenen Sprachen enthalten. Die Subregionen dürfen jedoch selbst auch Unterschiede beziehungsweise Gaps beinhalten. Damit stellen sie lokale Intervalle dar, die folgendermaßen berechnet werden.

1. Scanne von jedem Match einer LCS aus die gesamte LCS und summiere das Verhältnis von Matches und Gaps.
2. Unterschreitet das Matchverhältnis einen bestimmten Schwellenwert, beende das aktuell bearbeitete Intervall und untersuche den nächsten Startpunkt (Match), der rechts des zuletzt bearbeiteten Intervalls liegt.

Dieser Entwurf beschreibt einen in quadratischer Zeit ablaufenden Algorithmus, der Intervalle innerhalb einer Longest Common Subsequence erkennt. Die nächste Codebeschreibung zeigt diese Idee anhand der Funktion *find_intervalls*. Diese gibt eine Liste *result* zurück, die jeweils Listen derjenigen Tupel aus einer gegebenen längsten gemeinsamen Folge lcs enthalten, die unter Berücksichtigung einer Schranke *bound* lokale Intervalle bilden. Von jedem Match aus beginnend (Zeile 9) wird ein neues Intervall solange verlängert, bis dessen Verhältnis von Matches zu Gaps diese Schranke unterschreitet (Zeile 21). Zur inkrementellen Verlängerung der Intervalle wird eine zweite Schleife benutzt, die jeweils die Längen der gefundenen Übereinstimmungen (*matches*) und Abweichungen (*gaps*) aufsammelt und so in Zeile 20 das entsprechende Verhältnis (*match_ratio*) bestimmt. Die innere Schleife betrachtet dabei immer den nächsten Match (Zeile 14), insofern dieser existiert (Zeile 13). Die jeweiligen Gaps zwischen zwei Matches für w^1 und w^2 ergeben sich aus der Endposition des aktuellen Matches und der Startposition des nächsten Matches (Zeilen 16, 17). Hieraus ergibt sich zusammen mit der Länge des nächsten Matches (Zeile 18) das Matchverhältnis *match_ratio*. Liegt dieses unterhalb der festgelegten Schranke und besteht das gefundene Intervall mindestens aus

zwei Matches, so wird das aktuelle Intervall beendet und der Ergebnisliste hinzugefügt. Schließlich wird der Zähler i der äußeren Schleife auf den letzten Wert des Zählers der inneren Schleife gesetzt, wodurch sichergestellt wird, dass das nächste mögliche Intervall rechts vom Ende des gerade gefundenen Intervalls beginnt.

Algorithmus 24 Heuristik zur Bestimmung lokaler Intervalle

```
1:  function find_intervalls(lcs, bound)
2:      result ← {}
3:      for i ← 0 to |lcs| do
4:          matches ← 0
5:          gaps ← 0
6:          match_ratio ← 1
7:          intervall ← {}
8:          for j ← i to |lcs| do
9:              (v, endpos_s1, endpos_s2, ancestorIndex) ← lcs[j]
10:             matches ← matches + (|v|*2)
11:             intervall.push(lcs[j])
12:             next_match ← 0
13:             if j < |lcs|-1 then
14:                 (v', endpos_s1', endpos_s2', ancestorIndex') ← lcs[j + 1]
15:                 gap1 ← w^1_{endpos_s1}, ..., w^1_{endpos_s1'-|v'|}
16:                 gap2 ← w^2_{endpos_s2}, ..., w^2_{endpos_s2'-|v'|}
17:                 gaps ← gaps + |gap1|+|gap2|
18:                 next_match ← |v'|*2
19:             end if
20:             match_ratio ← ((matches + next_match)/gaps)/2
21:             if match_ratio < bound then
22:                 if |intervall| ≥ 2 then
23:                     result.push(intervall)
24:                     i ← j
25:                 end if
26:                 break
27:             end if
28:         end for
29:     end for
30:     return result
31: end function
```

Die nächste Abbildung zeigt einen Auszug der innerhalb der durch $align_{SC+H}$

alignierten Zeilen des Vergleichs FRA ⟷ CRS gefundenen Intervalle bei einer Schranke von 0.9. Die alignierten Zeilen wurden dabei anhand ihrer längsten vorkommenden Intervalle absteigend geordnet. Beginn und Ende eines jeweiligen Intervalls sind durch blau gefärbte geschweifte Klammern markiert. Aus Platzgründen sind die Endmarker der Intervalle manchmal abgeschnitten.

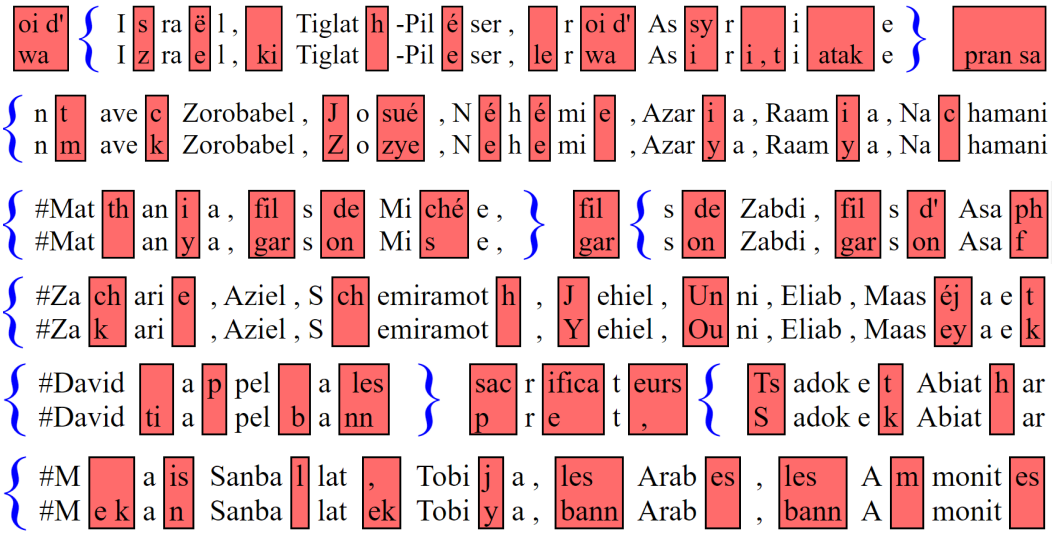


Abbildung 9.12.: Auszüge mit Intervallmarkierungen der zeilenweisen Alignierung von FRA ⟷ CRS.

Anhand der in Abbildung 9.12 dargestellten Intervalle ist leicht erkennbar, dass viele der dort markierten Intervalle Regionen umschließen, die sehr viele Eigennamen enthalten. Während dabei einige Namen wie „David“, „Eliab“ oder „Zorobabel“ in beiden Sprachen gleich geschrieben werden, sind anhand der Gaps (rot) in anderen Paaren die unterschiedlichen Schreibweisen der beiden Sprachen ablesbar. Beispiele solcher unterschiedlicher Schreibvarianten sind „Josué“ ⟷ „Zozye“, „Zacharie“ ⟷ „Zakari“, „Tsadok“ ⟷ „Sadok“ oder „Tobija“ ⟷ „Tobiya“. Als Gapsymbol bei Einfügungen beziehungsweise Löschungen wurde hier das leere Wort benutzt.

	DEU ⟷ ENG	DEU ⟷ FRA	DEU ⟷ CRS	FRA ⟷ CRS
Intervalle	56.614	40.170	33.094	69.299

Tabelle 9.9.: Anzahl der gefundenen Intervalle aller Vergleichspaare.

Tabelle 9.9 zeigt die Anzahl der unterschiedlichen Intervalle, die gefunden werden. Die französische Bibelübersetzung teilt hierbei am meisten Intervalle mit der in Seychellenkreol, während die deutsche Übersetzung mit dieser am wenigsten Intervalle bildet. Die nächste Grafik zeigt die absoluten Häufigkeiten (y-Achse) und die Längen (x-Achse) der Intervalle. Für alle vier Vergleichspaare werden dabei häufig auftretende kurze und selten vorkommende lange Intervalle identifiziert.

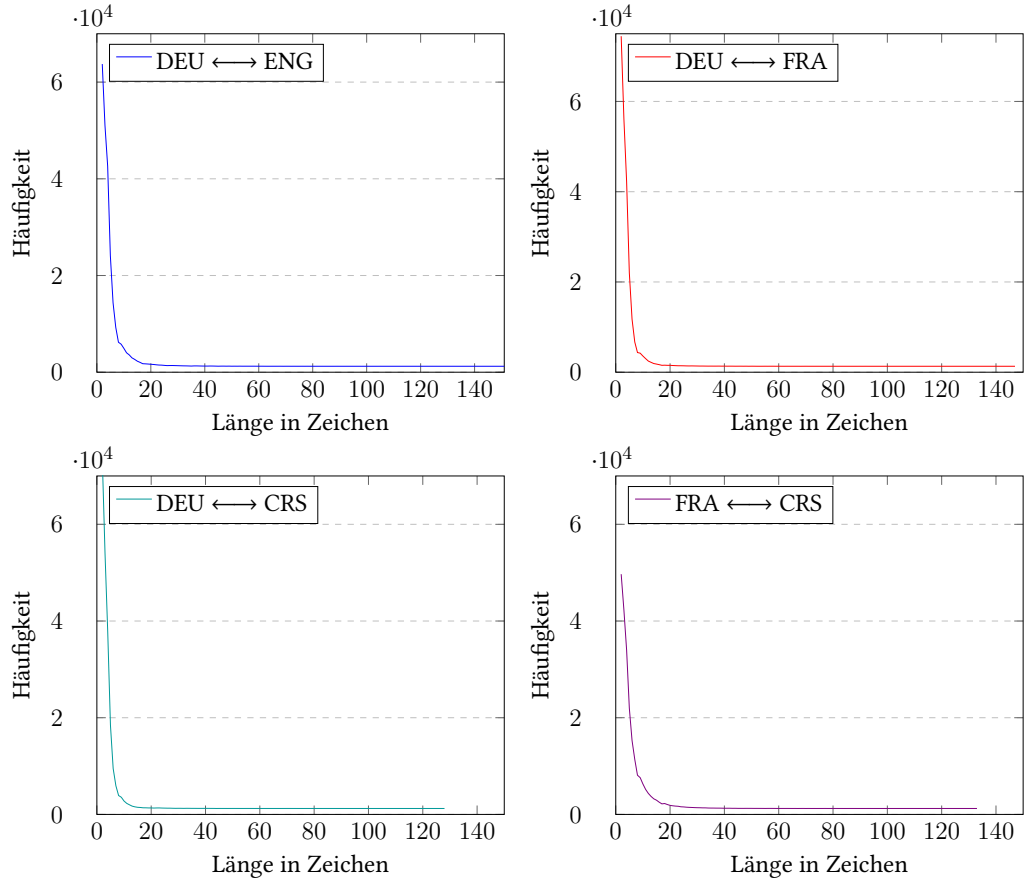


Abbildung 9.13.: Häufigkeiten und Längen der gefundenen Intervalle aller Vergleichspaare.

Unter der Annahme, dass die längsten Intervalle diejenigen sind, die viele Eigennamen enthalten, liegt somit der Schluss nahe, dass in allen Sprachpaaren solche Regionen in den zugrundeliegenden Alignierungen gefunden wurden. Im Vergleich FRA ⟷ CRS fällt auf, dass dort insgesamt mehr lange Intervalle gefunden werden. Diese Tatsache gibt erneut einen Hinweis auf den gemeinsamen Ursprung beider Sprachen.

Die so bestimmten Intervalle werden nun mittels einer einfachen Tokenisierung anhand der Wortgrenzen zerlegt und so Paare von Eigennamen extrahiert. Hieraus lassen sich Varianten der Eigennamen in allen vier Sprachen angeben. Die nächste Tabelle zeigt Auszüge aus der so entstehenden Zuordnung. Es wurden dabei nur Beispiele ausgewählt, die auch in allen Sprachen auftreten. Durch dieses Kriterium werden weitere fehlerhafte Zuordnungen, die etwa am Satzanfang auftreten, vermindert.

DEU	ENG	FRA	CRS
Nebukadnezar	Nebuchadnezzar	Nebucadnetsar	Naboukodonozor
Asarja	Azariah	Azaria	Azarya
Mose	Moses	Moïse	Moiz
Benajas	Benaiah	Benaja	Benaya
Galiläa	Galile	Galilee	Galilée
Hasiel	Haziel	Haziel	Hazyel
Benjamin	Benjamin	Benjamin	Benzamen
Israel	Israel	Israël	Izrael
Jerimoth	Jeremoth	Jeremoth	Yeremot
Lazarus	Lazarus	Lazare	Lazar
Assyrien	Assyria	Assyrieet	Asiri
Babel	Babylon	Babylone	Babilonn
Gedalja	Gedalih	Guedalia	Gedalya
Maria	Mary	Marie	Mari
Bethlehem	Bethlehem	Bethléhem	Betlehenm
Simson	Samson	Samson	Sanmson
Petrus	Peter	Peu	Pyer
Thamar	Tadmor	Thadmor	Tamar
Tobia	Tobijah	Tobija	Tobiya
Libanon	Lebanon	Libann	Libann
Herodes	Herod	Hérode	Herod
Magdalena	Magdalene	Magdala	Madlenn
Kadmoniter	Kadmonites	Kadmoniens	Kadmonit
Hazobeba	Hazzobebah	Hatsobéba	Hatsobeba

Tabelle 9.10.: Beispiele extrahierter Eigennamen in den vier Vergleichssprachen.

Durch diese Methode konnten insgesamt circa 1000 Namensvarianten gewonnen werden, die in allen vier Sprachen eine Entsprechung finden. Da eine sehr

simple Heuristik zugrundeliegt, die Wortpaarungen aufgrund eines beginnenden Großbuchstabens auswählt, enthalten manche Zuordnungen wie etwa „Zachariasm“, „Zechariah’s“, „Zacharie“, „Zakari“ oder „Rahels“, „Rachel’s“, „Rachel“, „Rasel“ auch gebeugte Lexeme. Solche könnten nachträglich durch Lemmatisierung entfernt werden. Zudem müssten zur weiteren Bereinigung der Liste, Wörter, die keine Eigennamen darstellen und am Satzanfang stehen, ausgefiltert werden wie etwa „Meister“, „Master“, „Maître“, „Met“ oder „Mein“, „My“, „Mo“, „Mon“.

10. Zusammenfassung Teil III.

Kapitel 8 führt den Begriff des *quasimaximalen Knotens* ein und zeigt, wie dieser in verschiedenen Definitionen (siehe Definitionen 8.1.1,8.1.2) gebraucht wird. Nachfolgend wird unter Berücksichtigung des Kontexts eine genauere Definiton der Vorkommen quasimaximaler Knoten gegeben, die längste Teilwörter innerhalb des Kontexts ihres Vorkommens findet (siehe Definition 8.1.3). Anhand dieser Definition wird ein Verfahren implementiert, das die Menge der quasimaximalen Knoten im Kontext ihres Vorkommens bestimmt (siehe Kapitel 8.1.5).

In den Kapiteln 8.2 und 8.3 werden zwei Anwendungen, die auf der Bestimmung längster gemeinsamer Teilwörter basieren, gezeigt. Das umfangreiche Thema der Erkennung von Text-Reuse wird dabei nur angeschnitten, trotzdem lässt auch hier die Möglichkeit, in linearer Zeit und mit linearem Speicherverbrauch alle längsten Teilwörter zu bestimmen, interessante Anwendungsfälle zu. Beide dargestellten Anwendungen profitieren von nachgeschalteter Hinzunahme von Metainformationen, die durch ein mit Algorithmus 14 vergleichbares Verfahren an die Knoten der SCDAWG-Struktur angehängt werden.

Während in den Kapiteln 8.2 und 8.3 eine Ordnung der längsten gemeinsamen Teilwörter keine Rolle spielt, wird im Kapitel 9.1 gezeigt, wie sich durch eine Kombination des in Kapitel 2.5 beschriebenen Verfahrens zur Bestimmung der längsten gemeinsamen Teilfolge zweier Strings mit der Bestimmung der längsten gemeinsamen Teilwörter durch Algorithmus 17 ein globales Alignment zweier Texte erzeugen lässt. Die Experimente dazu zeigen, dass die Laufzeit dieses Verfahrens klassischen Ansätzen überlegen ist, jedoch oft durch die dort aufgezeigten Probleme kein optimales Alignment liefert. Letzteres kann auch dann nicht gefunden werden, wenn zu große Gaps nachträglich korrigiert werden. Hierdurch wird jedoch oft eine nahezu optimale Alignierung gefunden, die in vielen Anwendungsbereichen ausreicht, wenn keine exakte zeichengenaue Alignierung gefordert ist.

Teil IV.

Textuelle Charakteristiken

11. Identifikation textueller Charakteristiken

Während in Kapiteln 8 und 9 die Identifizierung längster gemeinsamer Regionen im Vordergrund stand, kann eine SCDAWG-Struktur S_C über einer Wortmenge W auch dazu benutzt werden, Infixe zu finden, die bezüglich eines jeden Elements von W charakteristisch *sind. Als charakteristisch wird die Menge der kürzesten Infixe angesehen, die nur in einem Element von W auftreten. Diese Menge ist damit disjunkt zu allen anderen derartigen Infixmengen.*

11.1. Quasiminimale Knoten

In Anlehnung an die in Kapitel 8.1 eingeführte Definition der quasimaximalen Knoten wird die Knotenmenge eindeutiger kürzester Infixe in S_C als die der *quasiminimalen Knoten* bezeichnet. Die Menge Q_{min} enthält damit diejenigen Knoten v aus V, die nur in einem Wort aus W auftreten, wobei jeder Vorgänger von v in mehreren Wörtern auftritt. Damit bildet die Menge der quasiminimalen Knoten, wie in Abbildung 11.1 dargestellt, ein disjunktes Mengensystem. Jeder quasiminimale Knoten ist somit das Startglied der in Kapitel 8.1.4 beschriebenen eindeutigen Knotenketten. Im Graphen von Abbildung 8.4 wären damit v_1 (a) und v_6 (b) die einzigen Elemente von Q_{min}. Formal lässt sich dies folgendermaßen ausdrücken:

Definition 11.1.1 (Quasiminimale Knoten)**.** Ein Knoten v ist in der Menge der *quasiminimalen Knoten* Q_{min} enthalten, wenn für v gilt:

1. $v \notin V_m$,
2. $\forall v_k \in V$ mit $(v_k, (k, p), v) \in E_R^{\rightarrow v}$ gilt: $v_k \in V_m$.

Wenn also ein Knoten v selbst nicht innerhalb der Menge V_m der Knoten, die in mehr als einem Wort aus W auftreten, enthalten ist, jeder direkte Vorgängerknoten v_k aber in V_m inkludiert ist, dann ist v in der Menge quasiminimaler Knoten Q_{min} enthalten. Die linken Übergänge müssen hierbei nicht betrachtet werden, da, wenn

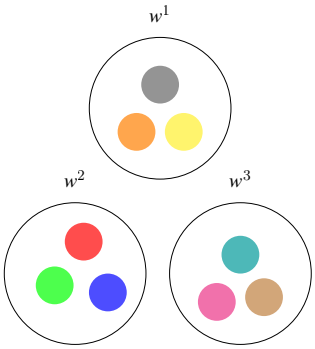


Abbildung 11.1.: Schematische Darstellung der Menge Q_{min} dreier Wörter w^1, w^2, w^3 als disjunktes Mengensystem.

als Start- und Endmarker jedes Wortes jeweils dasselbe Symbol benutzt wird, es keine Kette linker Übergänge geben kann, die zu einem Knoten führt, der nur in einem Wort enthalten und gleichzeitig minimal ist.

11.2. Implementierung

Aus Definition 11.1.1 lässt wiederum die Skizze einer Funktion angeben, die zu einer gegebenen SCDAWG-Struktur S_C über einer Wortmenge W die Menge aller quasiminimalen Knoten Q_{min} bestimmt. Die linken Kanten spielen bei diesem Entwurf keine Rolle, da es hier nur wichtig ist, jeden Knoten einmal zu besuchen. Somit kann wieder die DFS von Algorithmus 1 benutzt werden.

1. Bestimme die Menge V_m aller Knoten, die in mehr als einem Eingabewort auftreten.

2. Markiere alle Knoten, die Kinder eines Knotens, der nicht in V_m ist, sind, und speichere diese in der Menge V_u.

3. Liefere alle Knoten, die weder in V_W, V_m oder V_u sind.

Es folgt die Beschreibung des Algorithmus in Pseudocode, der die quasiminimalen Knoten in einer SCDAWG-Struktur erkennt.

Algorithmus 25 Funktion zur Identifikation quasiminimaler Knoten

Vorbedingung: Die SCDAWG-Struktur $S_C = (V, E_R, E_L)$ einer endlichen Menge von Zeichenketten W mit $W = \{w^1, \ldots, w^i, \ldots, w^n\}$ und $w^i \in \Sigma^*$.

```
function find_shortest_distinct_substrings()
    result ← {}
    V_m ← get_V_m()                          ▷ Variante von Algorithmus 14
    V_u ← get_V_u(V_m)                       ▷ Aufruf von Algorithmus 26
    for each v ∈ DFS(v_ε, false) do          ▷ DFS in Präordnung
        if v ∈ V_W then
            continue
        else if v ∈ V_m then
            continue
        else if v ∈ V_u then
            continue
        end if
        result.push(v)
    end for
    return result
end function
```

In der Funktion *find_shortest_distinct_substrings* werden zuerst die Mengen V_m und V_u bestimmt. V_m markiert alle Knoten mittels der Funktion *get_V_m* nach ihrem Vorkommen in einem oder mehreren Eingabewörtern. Anschließend wird die Funktion *get_V_u()* aufgerufen, die all jene Knoten in der Struktur speichert, die direkte Kinder eines Knotens sind, der nur in einem String auftitt. Diese Funktion ist nachfolgend beschrieben. Für das Auffinden der quasiminimalen Knoten werden die Ergebnisse dieser Subroutinen genutzt. Hierfür wird eine weitere DFS, die vom Wurzelknoten v_ε gestartet wird, ausgeführt. Für jeden Knoten wird nun überprüft, ob er in einer der beiden Listen vertreten ist, beziehungsweise, wenn sichergestellt wurde, dass der Knoten nur in einem Eingabewort auftritt, wird nachfolgend geprüft, ob er in der Ergebnismenge V_u der zweiten Funktion enthalten ist. Nur wenn Letzteres nicht zutrifft, ist ein quasiminimaler Knoten gefunden, denn dann steht fest, dass dieser Knoten kein Kind eines anderen Knotens, der nur in einem einzigen Wort aus W auftritt, ist.

Markieren direkter Nachfolger eindeutiger Vorkommen

Ziel der Hilfsfunktion *get_V_u* ist es also, die Knoten zu markieren, die direkte Kinder eines Knotens sind, der nur in einem Eingabewort von W vorkommt. Wie auch

bei den beiden anderen Schritten wird dies durch eine DFS, die vom Wurzelelement v_ε aus gestartet wird, erreicht.

Algorithmus 26 Hilfsfunktion zur Identifikation quasiminimaler Knoten

```
function get_V_u(V_m)
    result ← {}
    for each v ∈ DFS(v_ε, false) do                ▷ DFS in Präordnung
        if v ∈ V_W then
            continue
        else if v ∈ V_m then
            continue
        end if
        for each (v, (k, p), v') ∈ E_R^{v→} do
            if v' ∉ V_m then
                result.push(v')
            end if
        end for
        for each (v, (k, p), v') ∈ E_L^{v→} do
            if v' ∉ V_m then
                result.push(v')
            end if
        end for
    end for
    return result
end function
```

Insgesamt ist die Laufzeitkomplexität von Algorithmus 25 demnach linear, da drei Depth-First-Suchen hintereinander ausgeführt werden. In einem letzten Schritt kann nun noch die Vorkommenshäufigkeit der quasiminimalen Knoten bestimmt werden, wenn mit Hilfe von Algorithmus 14 die Häufigkeiten aller Knoten errechnet wurden.

11.3. Beispiele für *find_shortest_distinct_substrings*

Es werden einige Ablaufbeispiele des Algorithmus zur Bestimmung quasiminimaler Knoten betrachtet. Es wird dabei angenommen, dass, wie eben erwähnt, die insgesamten Knotenhäufigkeiten durch *add_freq_labels* bestimmt wurden und so-

mit zu jedem quasiminimalen Knoten dessen Vorkommenshäufigkeit angegeben werden kann.

Beispiel 11.3.1. Sei $W = \{\#abcabc\$, \#xyxyxz\$\}$, so ergibt sich für

$$\mathit{find\_shortest\_distinct\_substrings}() = \{(abc, 2), (x, 3)\}.$$

Als distinkte Regionen, die entweder in w^1 oder w^2 auftreten, werden die Infixe abc für w^1 und x für w^2 gefunden, die zweimal beziehungsweise dreimal auftreten. In w^1 stellt abc eine minimale Region dar, die nur dort auftritt. In w^2 wird aufgrund des Infixes xz nicht xy als minimale distinkte Region identifiziert, sondern x. Die folgende Abbildung zeigt die zugehörige SCDAWG-Struktur mit markierten quasiminimalen Knoten.

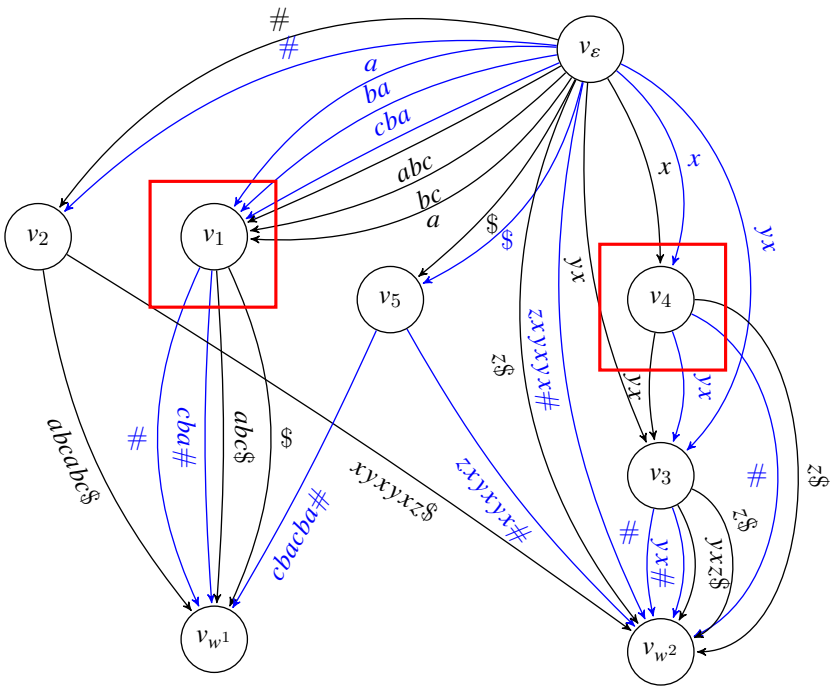


Abbildung 11.2.: *SCDAWG*$(W = \{\#abcabc\$, \#xyxyxz\$\})$.

In Abbildung 11.2 ist zu erkennen, dass die beiden Knoten $v1$ (abc) und $v4$ (x) jeweils nur in einem der beiden Eingabewörter auftreten und sie gleichzeitig die Eigenschaft von minimalen Infixen innerhalb der SCDAWG-Struktur haben.

Beispiel 11.3.2. Sei $W = \{\#abcabc\$, \#abab\$\}$, so ergibt sich für

$$\mathit{find\_shortest\_distinct\_substrings}() = \{(abc, 2)\}.$$

Werden die Strings aus Beispiel 11.3.1 dahingehend abgeändert, dass auch im zweiten Wort das Infix ab auftritt, enthält Q_{min} nur noch das Infix abc. Im zweiten String ist damit kein minimales Infix mehr inkludiert, das nicht auch im ersten auftritt.

Beispiel 11.3.3. Sei $W = \{\#abcabc\$, \#xyxyxz\$, \#x\$\}$, so ergibt sich für

$$\mathit{find\_shortest\_distinct\_substrings}() = \{(abc, 2), (xy, 2)\}.$$

Wenn den Eingabewörtern aus Beispiel 11.3.1 ein dritter String x hinzugefügt wird, ist der Knoten, der x repräsentiert, nicht mehr quasiminimal, da er nun in mehr als einem Eingabewort auftritt, weshalb nun aber dessen Nachfolger xyx wiederum quasiminimal wird.

11.4. Metainformationen

Die Identifikation distinkter Regionen basiert, wie in Definition 11.1.1 beschrieben, auf der Bestimmung der Menge V_m, die angibt, ob ein Infix entweder in genau einem Wort aus Σ^* auftritt oder in mehreren. Dies birgt aber Schwierigkeiten, wenn etwa in einem Korpus, das viele tausend Dokumente enthält, minimale Teilwörter bezüglich bestimmter Textklassen gefunden werden sollen. Um dies mit Algorithmus 25 zu erreichen, müssten im Vorfeld alle Texte, die zu einer Klasse gehören, zu einem einzigen Wort aus Σ^* zusammengefasst werden. Eine solche Zusammenfassung hat jedoch den Nachteil, dass die Information, welche Teilwörter in den ursprünglichen Dokumenten auftreten, verloren ginge.

Zur Lösung dieser Problematik können, wie bereits in Experiment 8.3 in Abschnitt 8.3.1 angewendet, verschiedene Klasseneinteilungen als Metainformationen innerhalb der Knoten der SCDAWG-Struktur gespeichert werden. Die hierfür benötigte Prozedur entspricht dem Hinzufügen der Frequenzlabels aus Algorithmus 14. Für die Berechnung quasiminimaler Knoten bedeutet dies, dass die Menge V_m damit durch eine Menge V_x ersetzt wird, die angibt, ob ein Infix bezüglich eines Meta-Attributs in mehreren oder in nur einem Dokument auftritt. Ist V_x bestimmt, verläuft die anschließende Berechnung von Q_{min}, wie in Algorithmus 25 und Algorithmus 26 gezeigt, ab. Wird diese Modifikation vorgenommen, können verschiedene Mengen quasiminimaler Knotern für verschiedene Meta-Attribute, wie etwa Autorennamen, Datumsangaben oder räumliche Verortungen berechnet werden, ohne dass die eigentliche Dokumenteneinteilung des verwendeten Korpus verloren geht.

11.5. Textklassifikation durch Mehrheitsentscheidung

Mit dem durch Algorithmus 25 beschriebenen Verfahren soll nun versucht werden, in verschiedenen Korpora, Dokumente anhand der erkannten charakteristischen Teilwörter zu klassifizieren. Dabei wird ein simples Verfahren angwendet, das aufgrund der häufigsten Vorkommen charakteristischer Teilwörter eine Entscheidung trifft, welcher Klasse ein zuvor unbekanntes Dokument zugeordnet wird. Aufgrund der verschiedenen großen Anzahlen extrahierten charakteristischer Teilwörter pro Klasse werden für alle Klassen stets nur so viele charakteristische Teilwörter betrachtet, wie für die kleinste Klasse gefunden wurden.

Zusätzlich wird zur Gewichtung der so entstandenen distinkten Regionen die *document frequency (df)* jedes quasiminimalen Knotens ermittelt. Die Berechnung dieses Maßes wird erneut in Anlehnung an Algorithmus 14 durchgeführt.

11.5.1. Gedichte nach Jahrhunderten

Das zuerst betrachtete Korpus enthält Gedichte in deutscher Sprache, die etwa in einer Zeitspanne von etwa 1650 - 1900 verfasst wurden[1]. Die hier verwendete Dokumentenbasis umfasst circa 20.000 Gedichte, die aufgrund der Lebensdaten ihrer Autoren auf drei Klassen, die das 17., 18. und 19. Jahrhundert repräsentieren, aufgeteilt werden. Dieser Einteilung liegt somit die Annahme zugrunde, dass manche Teilwörter nur innerhalb eines bestimmten Jahrhunderts benutzt werden, während andere diachron innerhalb des gesamten Zeitraumes auftreten. Damit wird die in Kapitel 11.4 dargestellte Methode der Einteilung der Dokumente anhand eines Meta-Attributs verwendet. Die Zuweisung eines Dokuments in eine der drei Klassen wird mit Hilfe des Durchschnitts aus den Geburts- und Sterbedaten des Autors eines jeden Gedichts realisiert. Die nachstehende Tabelle zeigt jeweils die am meisten auftretenden 20 quasiminimalen Knoten der drei Klassen, wenn diese absteigend nach ihrem *df*-Wert geordnet werden.

Für das 17. Jahrhundert fallen viele Substrings ins Auge, die historische Schreibweisen des Präfixes „Un“ darstellen, wie etwa „, Vn“,„n Vnd“ oder „Vng“. Diese Schreibweise mit „V“ wird in den Gedichten des 18. und 19. Jahrhunderts nicht mehr benutzt. Ein anderes hier nur in den lyrischen Werken des 17. Jahrhundert benutztes Suffix ist zum Beispiel „rumb “ wie in „Darumb“oder „Warumb“. Gleichermaßen wird auch das Infix „rewd“ beziehungsweise „Frew“, das der historischen Schreibweise für „(f)reud“ oder „Freu(d)“ entspricht, auch nur in den Texten, die hier dem 17. Jahrhundert zugewiesen wurden, gefunden.

[1]Die verwendeten Daten entstammen dem Korpus des TextGrid Repositories [74].

Die dem 18. Jahrhundert zugeschriebenen Gedichte enthalten beispielsweise das Suffix „ßenschaft“, das zu den beiden Wörtern „Wißenschaft“ und „Verlaßenschaft“ gehört. Viele andere dort auftretende Teilworter, wie etwa „s isch“ geben einen Hinweis, dass in dieser Klasse viele Texte enthalten sind, die in Dialekt verfasst wurden. Andere Beispiele dafür sind das Präfix „ chu“ sowie das Suffix „hunnt“, die eine Verbindung zu dem schweizerdeutschen Wort „chunnt“ erkennen lassen.

Für die Gedichte des 19. Jahrhunderts lassen sich ebenfalls dialektspezifische Teilwörter wie „ een“, das zum Beispiel im Berlinerischen anstelle von „ein“ verwendet wird, als solche Teilstrings erkennen, die in den meisten verschiedenen Dokumenten auftreten.

	17. Jh.	*df*	18. Jh.	*df*	19. Jh.	*df*
1	„, Vn“	230	„s isch“	62	„ een“	103
2	„rumb “	183	„ lueg“	52	„ un d“	103
3	„ jh“	161	„ chu“	51	„, un “	101
4	„n Vnd“	160	„hunnt“	43	„ //“	97
5	„en Vn“	155	„uegt“	43	„ vun“	97
6	„ / daß “	152	„chäze“	41	„. /“	96
7	„ Vnd w“	147	„geße“	39	“ggt “	94
8	„t Vn	145“	„kei “	39	„, / “	89
9	„ Vnd d“	144	„ jez “	37	„ę“	87
10	„Vnd s“	136	„warth“	36	„ un w“	86
11	„rewd“	123	„ chö“	36	„vun “	83
12	„arumb“	118	„ het e“	35	„k un “	83
13	„ dise“	118	„en. Allein, “	33	„œw “	82
14	„n vn“	115	„ isch n“	33	„keen “	80
15	„Frew“	111	„üze“	32	„l un “	77
16	„wde“	104	„ßenschaft“	32	„iggt “	76
17	„Vng“	103	„use“	32	„ œ“	74
18	„tausent“	103	„ uffe“	32	„t’ge S“	72
19	„r vn“	101	„isch’s “	31	„ un S“	70
20	„schaw“	101	„isch nit“	31	„ dor “	69

Tabelle 11.1.: Top 20 quasiminimale Knoten der drei Klassen 17. Jh., 18. Jh. und 19. Jh. nach document frequency absteigend sortiert.

Ebenfalls sind in den Infixen des 19. Jh. die Ligatur „œ“ sowie das Zeichen „ę “ zu finden. Außerdem finden sich in der Liste des 19. Jh. auch einige Transkriptionsmerkmale wie „ //“oder „. /“, die in den anderen Texten nicht auftreten.

Insgesamt wurden circa 1.127.000 distinkte Teilwörter identifiziert, die sich folgendermaßen auf die drei Klassen verteilen:

17. Jh. ~ 130.000

18. Jh. ~ 270.000

19. Jh. ~ 710.000

Die großen Unterschiede in den Anzahlen gefundener distinkter Teilwörter ergibt sich aus der Anzahl der Dokumente, die jeder Klasse zugewiesen wurde. Um einen Anhaltspunkt zu erhalten, ob diese Regionen für die jeweilige Zeitspanne charakteristische Merkmale darstellen, wird nun anhand eines Testsets überprüft, ob sich damit bisher nicht betrachtete Gedichte ihrer zugewiesenen Klasse zuordnen lassen. Das Testset enthält circa 1600 Dokumente wobei 530 dem 17. Jh., 245 dem 18. Jh. und 820 dem 19. Jh. zugeteilt wurden.

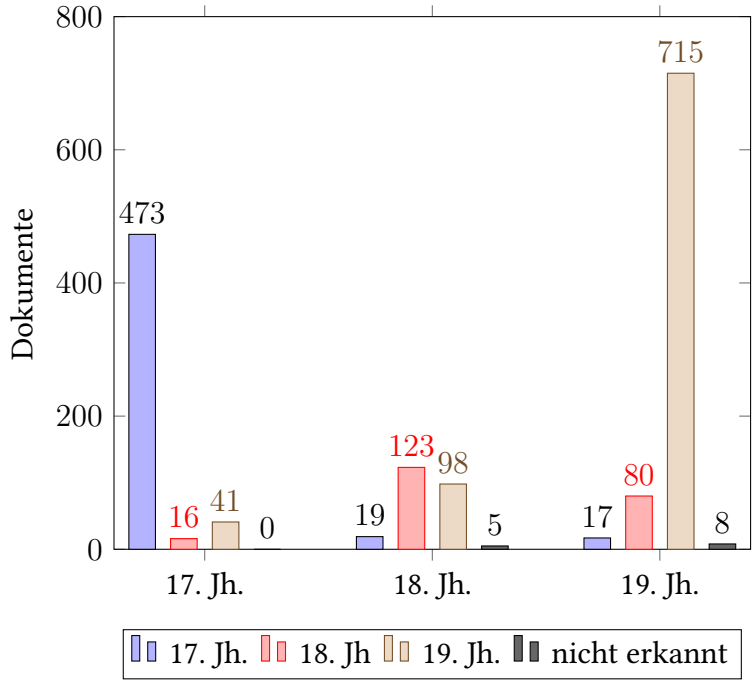


Abbildung 11.3.: Einordnung des Testsets anhand quasiminimaler Knoten.

Abbildung 11.3 zeigt die entstandenen Zuordnungen des Testsets, wenn aus den drei absteigend nach der document frequency sortierten Listen quasiminimaler Knoten jeweils 100.000 Infixe verwendet wurden. Die Entscheidung, welcher Klasse ein Testdokument angehört, wurde dabei anhand des maximalen Wertes der

Vorkommen quasiminimaler Knoten getroffen. Um das Vorkommen eines quasiminimalen Knotens in einem Testset-Dokument zu bestimmen, wird für jedes Test-Dokument eine SCDAWG-Struktur aufgebaut, die anschließend mit *find_p* durchsucht wird. Auf den ersten Blick ist ersichtlich, dass die Texte des Testsets, die dem 17. oder 19. Jahrhundert zugeteilt wurden, auch überwiegend in diese Klasse eingeordnet wurden. Im 18. Jahrhundert wurde jedoch nur etwa die Hälfte aller Testset-Dokumente auch ihrer ursprünglichen Klasse zugeordnet. Allerdings werden die anderen Dokumente aus dieser Testset-Klasse fast ausschließlich dem 19. Jahrhundert zugeschrieben. Da die hier vorgenommene Einteilung nach Jahrhunderten sicherlich nicht die Dynamik der Sprachevolution, die dem Korpus innewohnt, gerecht wird, ist aber auch dieses Ergebnis nicht verwunderlich. Die in der zweiten Klasse befindlichen Texte, die dem 19. Jahrhundert zugeschrieben wurden, wurden unter Umständen spät im 18. Jahrhundert verfasst, sodass die hier vorgenommene Einordnung dies widerspiegelt. Somit lässt sich auch bei dieser willkürlichen Einteilung der Gedichte nach Jahrhunderten feststellen, dass durch die Menge der quasiminimalen Knoten durchaus atomare Textbaussteine gefunden werden, die als charakteristisch angesehen werden können. Dies zeigt sich auch an der Tatsache, dass von den 1600 Dokumenten des Testsets nur 13 überhaupt keiner Klasse zugewiesen wurden, was bedeutet, dass für diese Dokumente kein einziges Vorkommen eines quasiminimalen Knotens aus einer der drei Listen gefunden wurde. Die durchschnittliche Accuracy liegt bei ca. 76% (17. Jh. ca. 89%, 18. Jh. ca. 51%, 19.Jh ca. 88%).

11.5.2. Dialektgebiete anhand althochdeutscher Urkunden

In einem zweiten Experiment sollen ähnlich wie im vorherigen Versuch ebenfalls wieder minimale sprachlich charakteristische Einheiten anhand einer bestimmten Klasseneinteilung erkannt werden. Die hier verglichenen Dokumente werden nun aber nicht mehr mittels ihres Entstehungszeitraumes voneinander abgegrenzt, sondern sie werden anhand ihrer geographischen Lage verortet. Dementsprechend werden die Dokumente in vier Klassen NO = Nordosten, NW = Nordwesten, SO = Südosten und SW = Südwesten eingeteilt, die dazu dienen sollen, dialektspezifische Teilwörter zu finden. Die Datengrundlage dieses Experimentes bildet eine Teilmenge des *Corpus der altdeutschen Originalurkunden bis zum Jahr 1300* [32, 22], das maßgeblich auf Friedrich Wilhelm zurückgeht. Die nun verwendete Teilmenge[2], die zur Extraktion distinkter Regionen genutzt wird, enthält 3.185 Dokumente, die sich wie folgt auf die vier Klassen verteilen:

[2]Der hier verwendete Datensatz wurde von Dr. Helmut Schmid zur Verfügung gestellt.

NO → 25,

NW → 86,

SO → 1155,

SW → 1919.

Das Beispielkorpus enthält damit eine deutliche Überzahl der Dokumente, die aus dem süddeutschen Raum stammen, während für den norddeutschen Raum vergleichsweise wenige Dokumente zur Verfügung stehen. Die folgende Tabelle zeigt erneut die nach *df* am höchsten bewerteten quasiminimalen Knoten.

	NO	*df*	NW	*df*	SO	*df*	SW	*df*
1	„n nach gotis geburt“	4	„n inde “	44	„t meine“	162	„ dis gesc “	290
2	„elen lu“	3	„ inde s“	43	„hous“	123	„is gescha“	283
3	„ von que“	3	„r inde “	42	„vercha“	99	„as dis“	277
4	„sleue “	3	„ inde v“	42	„ pei “	93	„lidig“	263
5	„ zweyhundir“	3	„ inde d“	40	„ perht“	92	„idig “	239
6	„ allerle“	3	„de dat “	39	„t datz“	90	„ inen“	214
7	„ kemerere “	3	„ dat si “	39	“dmar“	89	„tette “	202
8	„ wie albrecht von got“	3	„ inde a“	36	„ftich s“	87	„iemer “	198
9	„ myla “	3	„ne inde “	35	„tich sint d“	86	„ do man zalte von gott“	190
10	„me rechti“	3	„ dat wir “	34	„ch sint daz “	84	„dis wa“	172
11	„eyhundirt “	3	„it inde “	36	„ iar an sand“	80	„hent ald“	171
12	„ vnseme m“	3	„ inde b“	33	„ datz s “	78	„ hug “	169
13	„ r vnseme “	3	„ inde w“	31	„adma“	77	„ostenz“	168
14	„erenbeke “	3	„olne d“	31	„ chinde “	75	„ttes z“	166
15	„ misne “	3	„ inde i“	30	„fridre“	74	„ allen di“	153
16	„nstete “	3	„re inde “	29	„ meines “	73	„fribur“	148
17	„ nachkvmeling“	3	„nde inde “	28	„ prv“	72	„ne alle ge“	146
18	„neman von “	3	„ jnde da“	28	„ovsent iar “	72	„s war s“	145
19	„ vnde tun “	3	„ inde o “	28	„r datz“	71	„asil “	136
20	„ jan von “	3	„den dat “	27	„v chi “	71	„basi “	135

Tabelle 11.2.: Top 20 quasiminimale Knoten der vier Klassen NO, NW, SO, und SW nach document frequency absteigend sortiert.

Vergleicht man die Listen der vier Klassen, so fällt auf, dass die Klasse NO tendenziell längere Knoten enthält als die anderen drei, was darauf zurückzuführen ist, dass diese Klasse nur 25 Dokumente enthält und somit weniger kurze distinkte Regionen entstehen. Die insgesamte Verteilung der Dokumente auf die vier Klassen ist gleichermaßen an den Häufigkeiten der bestimmten *df*-Werte abzulesen. In

der Klasse, die Urkunden aus dem Nordwesten Deutschlands enthält, fallen vor allem Knoten, die die Infixe „inde“ und „dat“ enthalten, auf. Während Ersteres heutzutage nicht mehr gebräuchlich scheint, zeigt Letzteres auch aus heutiger Sicht klar in den nordwestlichen Sprachraum. Obwohl die Knoten, die diese beiden Infixe beinhalten, auch überlappen, wird ein kürzerer Knoten, der etwa nur „inde“ enthält, nicht als distinkte Region gefunden, da dieses Infix auch innerhalb anderer Klassen, etwa im Wort „Kinder“, auftritt. In der Klasse südöstlicher Texte ist beispielsweise das Infix „hous“ zu finden, das im schwäbischen Dialekt dem hochdeutschen Wort „haus“ entspricht und somit etwa in „goteshous“, „gasthous“ oder „housvrowen“ vorkommt. Die Klasse südwestlicher Texte beinhaltet beispielsweise die Infixe „ostenz“, „basi “, „asil “ und „fribur“, die auf die Orte Kostenz, Basel und Freiburg hinweisen.

Für das genutzte Testkorpus wurden absolut etwa 200.000 kürzeste distinkte Teilwörter extrahiert, die sich wie folgt auf die vier Klassen aufteilen:

NO ~ 630

NW ~ 6.300

SO ~ 58.000

SW ~ 135.000

Erneut wird nun ein Testset betrachtet und anhand der maximalen Vorkommen quasiminimaler Knoten eine Entscheidung getroffen, in welche Klasse ein unbekanntes Dokument eingeordnet wird. Die Zusammensetzung des Testsets entspricht etwa der der Ursprungsdaten:

NO → 3

NW → 10

SO → 144

SW → 240

Da für die Klasse NO nur etwa 600 charakteristische Infixe gefunden werden konnten, werden, verglichen mit dem vorherigen Experiment, bei dem die besten 100.000 Infixe genutzt wurden, zur Zuordnung nun nur die jeweils ersten 500 distinkten Regionen betrachtet, wenn diese nach ihrem df-Wert absteigend sortiert werden. Das Diagramm in Abbildung 11.4 zeigt die so resultierende Einordnung.

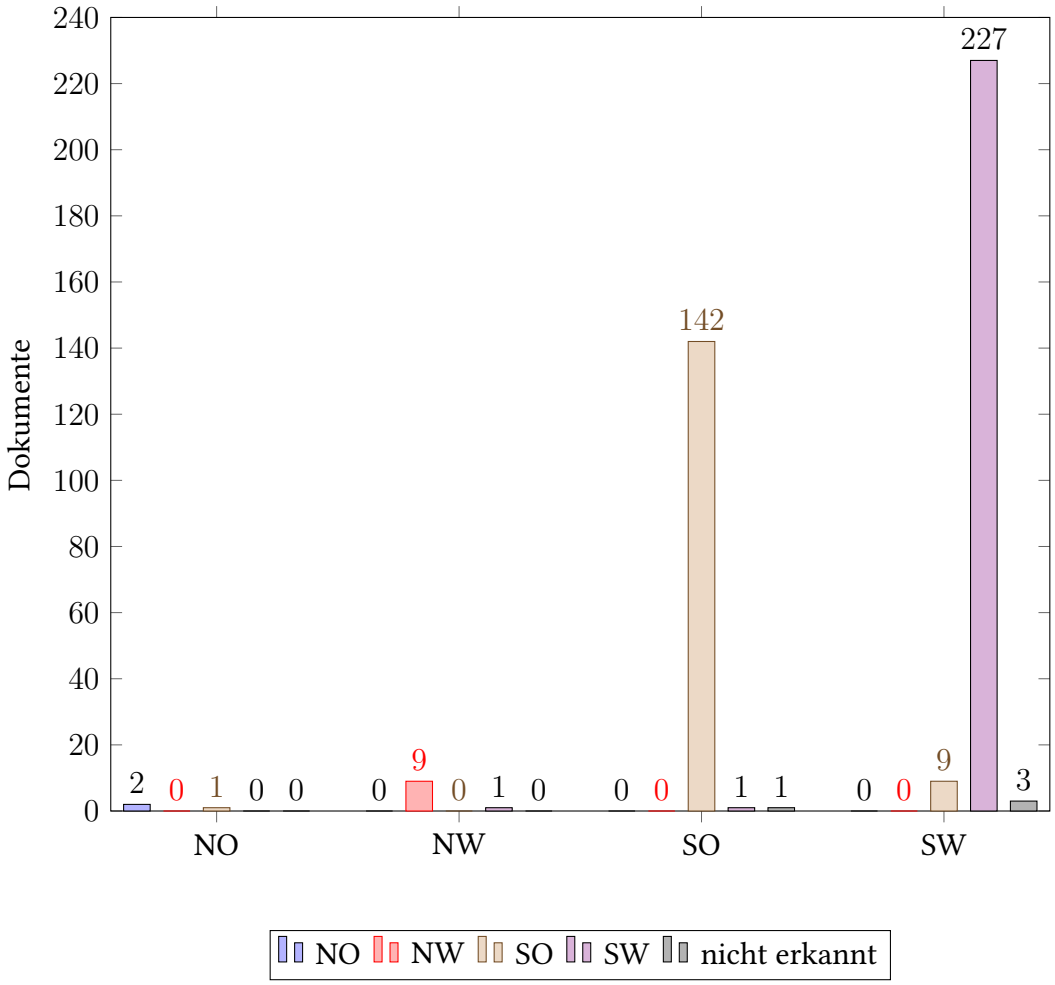


Abbildung 11.4.: Einordnung des Testsets anhand quasiminimaler Knoten im Korpus althochdeutscher Urkunden.

Obwohl mit den jeweils 500 höchstgerankten Infixen nun deutlich weniger charakteristische Features zur Verfügung stehen als beim vorherigen Experiment, zeigt sich doch eine hohe Rate richtiger Zuweisungen der Testset-Dokumente. Ebenfalls positiv hervorzuheben ist, dass von allen Test-Dokumenten nur vier überhaupt keiner Klasse zugewiesen werden konnten. Die durchschnittliche Accuracy beträgt damit etwa 88%, wobei diese aufgrund der wenigen Test-Dokumente der NO-Klasse nach unten gedrückt wird, da dort mit 2/3 korrekten die Accuracy nur bei ca. 66% beträgt, während sie für bei NW ca. 90%, für SO ca. 99% und SW ca. 96% liegt. Somit lässt sich festhalten, dass auch in diesem Experiment durch die Bestimmung quasiminimaler Knoten charakteristische minimale Infixe identifiziert

werden konnten, die sich gut dazu eignen, die betrachteten Testdokumente anhand ihres Dialektes zu erkennen.

11.5.3. Briefe nach Autoren

Das nächste Korpus enthält etwa 40.000 Briefe von sieben bekannten Persönlichkeiten, die in verschiedenen Sprachen verfasst wurden[3]. Erneut wird ein Testset bereitgestellt, das in diesem Fall aus circa 5000 Dateien besteht, wobei die Aufteilung des Testsets der des Ursprungskorpus entspricht. Die folgende Aufstellung gibt einen Überblick über das Korpus:

Autor	*Trainingsdokumente*	*Testdokumente*	*Sprache*
Wilhelm Busch	5018	627	de
Henrik Ibsen	7172	897	da
James Joyce	5459	682	en, it, fr
Franz Kafka	2238	280	de
Friedrich Schiller	2139	266	de
Johann Wolfgang v. Goethe	1840	228	de
Virginia Woolf	15211	1901	en
Gesamt	39077	4881	

Tabelle 11.3.: Übersicht über Briefe-Korpus bekannter Autoren.

Für alle Autoren erkennt man in den nach document-frequency gerankten Knoten oft solche, die Infixe von Grußformeln innerhalb der Briefe darstellen. Für Franz Kafka finden sich oft die Teilwörter „*Dein Franz*“, „*iebster Max*“ oder „*Franz K*“, bei Wilhelm Busch zum Beispiel „*. Busch*“, bei Goethe „*rüßen Sie Ihre l*“, „*. Goethe.*“ oder bei James Joyce „*Dear M*“, „*ly yours* “ oder „*s James Joyce*“. Ebenfalls treten dort einige Knoten auf, die sich auf bekannte Werke dieses Autors beziehen, wie etwa „*f Ulysses* “, „*Exiles*“. Henrik Ibsen ist der einzige Autor, dessen Briefe in norwegischer Sprache verfasst wurden. Damit finden sich für diesen Autor viele sehr kurze quasiminimale Knoten, die sehr häufig auftreten, wie etwa „*æ*“, „*på*“ oder „*øre*“. Alles in allem konnten etwa 384.000 charakteristische Teilwörter extrahiert werden. Diese werden nun erneut dazu benutzt, anhand eines simplen Mehrheitsentscheids die Dokumente des Testsets zu klassifizieren. Aufgrund der unterschiedlichen Anzahlen quasiminimaler Knoten, die für jede Klasse gefunden wurden, werden dazu jeweils die 10.000 am höchsten gerankten minimalen Teilwörter benutzt. Dies entspricht damit erneut dem Wert der kleinsten Liste, die in

[3]Der hier verwendete Datensatz wurde von Dr. Stefan Langer bereitgestellt.

diesem Korpus Johann Wolfgang von Goethe stellt. Die nächste Tabelle zeigt die Ergebnisse der Klassifikation.

Autor	*Total*	*Richtig*	*Nicht zugeordnet*	*Accuracy*
Wilhelm Busch	627	624	0	99,5%
Henrik Ibsen	897	878	1	97,9%
James Joyce	682	562	2	82,6%
Franz Kafka	280	256	1	91,7%
Friedrich Schiller	266	178	0	66,9%
Johann Wolfgang v. Goethe	228	139	1	61,2%
Virginia Woolf	1901	1896	3	99,8%
Gesamt	4881	4533	11	93,1%

Tabelle 11.4.: Ergebnisse der Klassifikation des Briefe-Korpus bekannter Autoren.

Wie Tabelle 11.4 zeigt, konnten auch in diesem Korpus insgesamt gute Accuracy-Werte erreicht werden. Besonders gut schneiden bei diesem Experiment Henrik Ibsen, Wilhelm Busch und Virginia Woolf ab. Während Henrik Ibsen jedoch der einzige Autor ist, der in diesem Korpus auf Norwegisch geschrieben hat, existieren für die Sprachen der anderen beiden Autoren auch andere Vertreter. Die schlechtesten Werte weisen Schiller und Goethe auf. Da beide Autoren zur selben Zeit lebten und oft auch untereinander korrespondierten, ist dieses Ergebnis durchaus nachvollziehbar. In Etwa 40% der Fälle werden die Briefe von Goethe und Schiller aber nicht nur dem jeweils anderen Autor zugeordnet, sondern Wilhelm Busch. Auch dies deutet daraufhin, dass durch die größere Synergie zwischen Goethe und Schiller weniger für diese beiden Autoren charakteristische Teilwörter gefunden werden. Diese Beobachtung wird auch von der Tatsache gestützt, dass für Goethe und Schiller insgesamt am wenigsten quasiminimale Knoten identifiziert werden konnten. Hieran lassen sich die Limitationen des vorgestellten Verfahrens gut erkennen. Befinden sich innerhalb der zu klassifizierenden Dokumentenklassen solche, die sich selbst sehr ähnlich sind, so sinkt die Größe der jeweiligen Komplementmengen minimaler Substrings dieser beiden Klassen und es existieren weniger deskriptive Teilwörter.

Zur Verbesserung dieser Ergebnisse könnten andere Methoden der Gewichtung und der Klassifikation eingesetzt werden als die hier benutzte document frequency, anhand derer per Mehrheits-Voting eine Entscheidung gefällt wurde. Dadurch könnten diejenigen Knoten, die keine guten Deskriptoren einer Klasse darstellen, abgeschwächt werden und somit solche, die auch bei größerer Synergie zweier oder mehrerer Klassen noch charakteristisch für eine Klasse sind, stärker an Einfluss gewinnen.

11.5.4. Chinesische Literatur nach Dynastie

In einem letzten Experiment zur Klassifikation anhand einer Mehrheitsentscheidung durch kürzeste distinkte Teilwörter wird ein Korpus betrachtet, das Dokumente chinesischer Literatur enthält und dem Pre-modern Chinese language corpus [80] entstammt[4]. Jedes Dokument ist einer von fünf Dynastien (Song, Yuan, Ming, Qing und Republic of China) zugeordnet. Verglichen mit den zuvor betrachteten Korpora in diesem Kapitel besteht dieses Korpus, das ein Teilkorpus des Pre-modern Chinese language corpus darstellt, insgesamt aus weniger Einzeldokumenten. Die insgesamte Größe (circa 76 Millionen Zeichen) übertrifft aber beispielsweise das in Kapitel 11.5.3 verwendete Briefe-Korpus (circa 12 Millionen Zeichen). Die folgende Tabelle gibt einen Überblick über das Korpus.

Dynastie	*Trainingsdokumente u. Zeichen*	*Testdokumente u. Zeichen*
Song	$8 \sim 12 * 10^6 \sim 35$ MB	$3 \sim 1 * 10^6 \sim 3$ MB
Yuan	$3 \sim 5 * 10^6 \sim 17$ MB	$1 \sim 0.4 * 10^6 \sim 1$ MB
Ming	$15 \sim 15 * 10^6 \sim 45$ MB	$6 \sim 8 * 10^6 \sim 22$ MB
Qing	$8 \sim 13 * 10^6 \sim 40$ MB	$3 \sim 6 * 10^6 \sim 17$ MB
Rep. of China	$17 \sim 13 * 10^6 \sim 39$ MB	$5 \sim 0.9 * 10^6 \sim 4$ MB
Gesamt	$51 \sim 58 * 10^6 \sim 176$ MB	$18 \sim 16 * 10^6 \sim 47$ MB

Tabelle 11.5.: Aufstellung über das Korpus chinesischer Literatur.

Verglichen mit indogermanischen Sprachen geben sinitische Sprachen oft keine expliziten Wortgrenzen an. Der folgende Auszug eines chinesischen Dokumentes der Ming-Dynastie verdeutlicht dies:

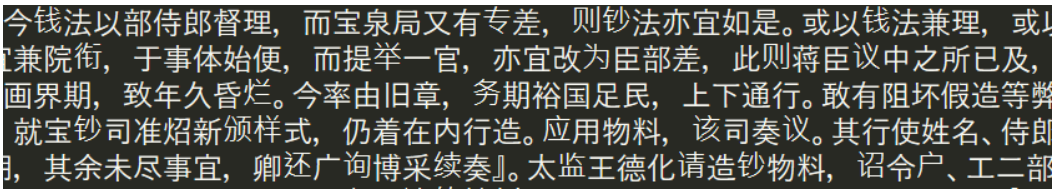


Abbildung 11.5.: Auszug eines chinesisches Textes der Ming-Dynastie.

Wie in [59] angemerkt, entsteht hieraus für Klassifikationsverfahren, die zunächst eine Tokenisierung der Dokumente in Wort-Tokens vornehmen, für sinitische Sprachen eine weitere Schwierigkeit, die bereits im Vorfeld zur Generierung

[4]Der hier verwendete Datensatz wurde von Jing Hu zur Verfügung gestellt.

von fehlerhaften Wort-Tokens führen kann. Im hier betrachteten rein zeichenbasierten Ansatz entsteht die Segmentierung des Korpus in Feature-Einheiten aufgrund der Kontexte, in welchen ein Teilwort auftritt, beziehungsweise seines Vorkommens als minimaler Substring einer bestimmten Klasse (hier Dynastien). Damit spielt es keine Rolle, ob explizite Wortgrenzen kodiert werden oder nicht.

Ebenfalls steigt im Vergleich zu einem Korpus indogermanischer Sprachen die Größe des verwendeten Alphabets stark an. Das Briefe-Korpus enthält etwa 200 unterschiedliche Zeichen für Σ, dem hier bearbeiteten Korpus chinesischer Literatur liegt ein Alphabet von etwa 14.000 unterschiedlichen Schriftzeichen vor. Da somit in jedem Schriftzeichen mehr Informationen kodiert sind als in einem europäischen Buchstaben, ist zu erwarten, dass die durchschnittliche Länge distinkter kürzester Teilwörter in diesem Korpus sinkt. Dem Korpus wurden damit etwa 30 Millionen quasiminimale Knoten entnommen, deren durchschnittliche Länge bei nur 3 Zeichen liegt. Im Briefe-Korpus aus Kapitel 11.5.3 lag die durchschnittliche Länge eines quasiminimalen Knotens bei etwa 7 Zeichen. Wenngleich also hier nun eine sehr große Anzahl von Substring-Features gefunden wurde, wird im folgenden Klassifikationsexperiment jeweils mit den 50.000 nach document frequency am höchsten gerankten distinkten Teilwörtern gearbeitet. Hieraus ergibt sich folgendes Ergebnis:

Dynastie	*Total*	*Richtig*	*Nicht zugeordnet*	*Accuracy*
Song	3	3	0	100%
Yuan	1	1	0	100%
Ming	6	6	0	100%
Qing	3	3	0	100%
Rep. of China	5	5	0	100%
Gesamt	18	18	0	100%

Tabelle 11.6.: Ergebnisse der Klassifikation des Korpus chinesischer Literatur nach Dynastie.

Tabelle 11.6 zeigt, dass alle Dokumente des Testsets korrekt ihrer zugehörigen Klasse zugeordnet werden konnten. Die im vorherigen Experiment gemachte Beobachtung konnte damit bestätigt werden. Wenn aufgrund eines größeren Alphabets weniger gemeinsam benutzte Zeichen existieren, lässt sich auch mit einem solch vergleichsweise einfachen Verfahren eine große Anzahl charakteristischer Teilwörter finden und damit sehr gute Klassifikationsergebnisse erzielen.

12. Zusammenfassung Teil IV.

Die in Kapitel 11 beschriebene Prozedur ermöglicht, über einer SCDAWG-Struktur kürzeste Teilwörter zu finden, die ausschließlich in einem der indexierten Wörter aus W auftreten. Damit diese auch für beliebige Teilmengen aus W gefunden werden können, wird in Kapitel 11.4 besprochen, wie sich Algorithmus 25 verändern kann, sodass die kürzesten distinkten Teilwörter für ebensolche Einteilungen anhand eines Metalabels identifiziert werden können. Im nachfolgenden Kapitel 11.5 wird gezeigt, wie dieses Vorgehen auf verschiedene Korpora angewendet werden kann, und dabei eine simple Strategie zur Dokumentenklassifikation verfolgt, die die extrahierten quasiminimalen Knoten benutzt, um noch unbekannten Dokumenten eine Klasse zuzuweisen. Je nach Zusammensetzung der verschiedenen Korpora konnten unterschiedliche Anzahlen minimaler Substrings gefunden werden, wobei für die Klassifikation stets gleich große Featurelisten quasiminimaler Knoten benutzt wurden, deren Länge derjenigen der kürzesten Liste entspricht. Zum Ranking der quasiminimalen Knoten wurde für jedes derartige Teilwort dessen document frequency bestimmt. Zur Berechnung der document frequencies aller im SCDAWG repräsentierten Teilwörter wurde auf eine Variante von Algorithmus 14 zurückgegriffen. Zur Bestimmung der Klasse eines Testdokumentes wurde ein einfacher Mehrheitsentscheid verwendet, der auf den zuvor nach df gerankten quasiminimalen Knoten basiert. Für die vier betrachteten Korpora konnten somit gute Klassifikationsraten erreicht werden, wobei, wenn innerhalb der Trainingsdaten Synergien zwischen einzelnen Vertretern unterschiedlicher Klassen bestanden, die Qualität dieses Verfahrens in manchen Fällen stark abnahm. Dies ist etwa in Abschnitt 11.5.1 anhand der Klasse des 18. Jahrhunderts zu sehen oder in Abschnitt 11.5.3 bei der vergleichsweise schlechten Erkennung von Goethe und Schiller. In den Korpora aus Abschnitt 11.5.2 und 11.5.4 ergaben sich sehr gute Accuracy-Werte, sodass hier davon ausgegangen werden kann, dass die ursprüngliche Klasseneinteilung eine sehr gute Abgrenzung der jeweiligen Subkorpora zulässt. Besonders im Beispiel des Korpus chinesischer Literatur, in dem zusätzlich ein stark vergrößertes Alphabet zugrundeliegt, konnten gute Ergebnisse erzielt werden[1]. Das hier beschriebene SCDAWG-basierte Verfahren zeigt nur einen möglichen Ansatz, eine String-Indexstruktur zur Bestimmung charakteristischer Teilwörter innerhalb einer Menge von Wörtern zu nutzen. So wird etwa in [59] ein auf Suffixbäumen basierendes Verfahren beschrieben, das maximale

[1] Eine genauere Untersuchung SCDAWG-gestützter Dokumentenklassifikation ist Gegenstand der momentan entstehenden Masterarbeit *Indexgestützte Dokumentenklassifikation mit minimalen Substrings* von Jing Hu.

Substrings zur Dokumentenklassifikation ermittelt. Inenaga beschreibt in seiner Doktorarbeit [40] weitere Verfahren, die dazu dienen, Teilwörter zu finden, mit deren Hilfe sich zwei Mengen von Wörtern gegeneinander abgrenzen lassen. Bei diesen Verfahren werden verschiedene String-Indexstrukturen wie Suffixbäume, DASGs (*Directed Acyclic Subsequence Graphs* oder EDASGs (*Episode Directed Acyclic Word Graphs*) unter Zuhilfenahme unterschiedlicher Scoring-Funktionen eingesetzt.

Fazit

Wie bereits von Blumer et al. in [8] erläutert, bietet die (S)CDAWG-Struktur viele Vorteile gegenüber konventionellen keyword-basierten Indizes, wenn es darum geht, beliebige Teilwörter, die nicht im Vorfeld bekannt sind, zu finden. In **Teil I** wurden zunächst die formalen Grundlagen und der Zusammenhang zwischen den String-Indexstrukturen Suffixtrie, Suffixbaum, DAWG und CDAWG beleuchtet und bekannte Algorithmen zu deren effizienter Konstruktion betrachtet. Anschließend folgte die formale Definition des SCDAWGs in Kapitel 3.6. In Abschnitt 3.6.2 folgte die Verallgemeinerung der SCDAWG-Struktur für eine Menge von Wörtern W. Hierbei wurden, damit sowohl für W als auch für W^{rev} die in Definition 3.6.2 beschriebene Präfixeigenschaft gilt, allen Elementen jeweils gleiche Start- und Endmarker hinzugefügt. Als Symbole für diese wurden stets $\#$ und $\$$ benutzt. Ein großer Vorteil dieses Vorgehens ist es, dass die Präfixeigenschaft gewährleistet ist, jedoch dem ursprünglichen Alphabet Σ nur zwei weitere Zeichen hinzugefügt werden müssen, wodurch sich für die Laufzeit keine Nachteile ergeben. Da alle Wörter nun die gleichen Start- und Endmarker besitzen, wird es für SCDAWG-Strukturen, die Wörter enthalten, die mit demselben Suffix enden, nötig, in einer Tiefensuche linke Erweiterungen zu verfolgen, wenn ein Knoten erreicht ist, der den Endmarker $\$$ enthält. Der Grund dafür ist, dass nun die Start- und Endmarker gemeinsam auftreten und somit innerhalb der Knoten der Struktur enthalten sind. Wird ein Knoten, der den Endmarker $\$$ enthält, besucht, so hat dieser nur noch linke Erweiterungen, da $\$$ stets am Ende der Elemente aus W auftritt.

Diese Eigenschaft der hier implementierten SCDAWG-Struktur ist in **Teil II** in Abschnitt 5.3.3 beziehungsweise Abbildung 5.4 dargestellt. Zur eigentlichen Implementierung der generalisierten SCDAWG-Struktur wurden zu Beginn von Teil II zwei Ansätze gezeigt, die diese Struktur entweder on-line oder aufbauend

auf der CDAWG-Konstruktion von Inenaga et al. [42] off-line erzeugen. Anschließend wurden einige Zeit- und Speichermessungen angeführt, die die Linearität der Implementierung darlegen und deren Speicherverbrauch im Vergleich zu den anderen behandelten Indexstrukturen zeigen. Zum Abschluss von Teil II wurden in Kapitel 6 Funktionen beschrieben, die die in Definition 6.1.1 gegebene Formalisierung einer invertierten Datei implementieren. Die dort vorgestellte Funktion aus Algorithmus 14 zeigt die globale Annotation der SCDAWG-Struktur mit Frequenzlabels. Algorithmus 14 dient somit auch als Vorlage weiterer globaler Annotationsfunktionen, die zum Beispiel dem Hinzufügen von Metainformationen dienen.

Ein Ziel dieser Arbeit war es, neben den Vorteilen, die lokale Suchanfragen auf einer SCDAWG-Struktur bieten, aufzuzeigen, wie eine SCDAWG-Struktur über einer Wortmenge W für globale Anfragen, die die komplette Struktur betreffen, eingesetzt werden kann, was in den letzten beiden Teilen dieser Arbeit untersucht wurde. In **Teil III** wurden globale Abfragen behandelt, die längste gemeinsame Teilwörter einer Dokumentensammlung erkennen lassen. Als Begriff eines Knotens, der ein längstes gemeinsames Teilwort repräsentiert, wurde der des *quasimaximalen Knotens* gewählt. Im Anschluss an einige erste Überlegungen in Bezug auf quasimaximale Knoten und deren mögliche formale Beschreibungen in Definition 8.1.1 und 8.1.2) wurde ein Verfahren vorgestellt, das sich sowohl linke als auch rechte Übergänge der Struktur zunutze macht und so anhand von Übergangspaaren, die jeweils aus einer linken und einer rechten Kante, die auf denselben Sinkknoten führen, bestehen, die längsten gemeinsamen Teilwörter einer Wortmenge W im jeweiligen Kontext ihres Auftreten innerhalb eines Wortes aus W identifiziert. Eine formale Definition dieser Methode ist in Definition 8.1.3 zu finden sowie eine beispielhafte Illustration in den Abbildungen 8.2 und 8.3. Bevor in Kapitel 8.1.5 die Beschreibung der Implementierung dieses Verfahrens folgte, wurde in Kapitel 8.1.4 eine Erweiterung diskutiert, die nötig wurde, da die gesuchten Übergangspaare aus linkem und rechtem Übergang in manchen Fällen nicht direkt auf die Sinkknoten führen, sondern sich eindeutige Knotenketten, die aus einer Folge von Knoten genau eines Wortes bestehen, zwischen dem Ausgangsknoten und dem jeweiligen Sinkknoten befinden. Diese Eigenschaft wurde weiterhin durch das Beispiel in Abbildung 8.4 verdeutlicht. Die darauffolgenden Kapitel und Unterkapitel beschäftigen sich mit der Anwendung von Algorithmus 17 im Kontext verschiedener Teilgebiete der digitalen Geisteswissenschaften. Für das sehr umfangreiche Gebiet der Identifikation von Text-Reuse wurde in Kapitel 8.2 ein inhaltsbasiertes Vorgehen gezeigt, dessen Grundlage die Menge der längsten gemeinsamen Teilwörter, die durch die zuvor beschriebene Methode gefunden wurden, bilden. Die so entstandenen Verknüpfungen jener mit allen Dokumenten, in denen sie auftreten, wurde für zwei Beispielkorpora in Abbildungen 8.5 und 8.6

visuell dargestellt. Zur weiteren Differenzierung der Knoten dieses Graphen wurden zusätzlich Metainformationen miteinbezogen, die durch Einfärbungen etwa in der Visualisierung aus Abbildung 8.7 sichtbar gemacht wurden. Die zweite gezeigte Anwendung beleuchtet ein praktisches Problem, das bei der Digitalisierung historischer Dokumente auftritt, deren Vorgehensweise aber auch allgemein zur Zuordnung von Texteinheiten innerhalb ungeordneter Mengen verschiedener Kategorien verstanden werden kann. Im in Kapitel 8.3 beschriebenen Fall werden Zeilenpaare oder Zeilentripel anhand des jeweils längsten gemeinsamen Teilwortes, das genau einmal innerhalb der jeweiligen Kategorien (hier OCR u. Ground-Truth oder OCR 1, OCR 2, OCR 3) auftritt, einander zugeordnet. Verglichen mit klassischen Ansätzen, wie der Bestimmung der Levenshtein-Distanz [46], die in Kapitel 2.3 kurz vorgestellt wurde, bietet die Vorberechnung der SCDAWG-Struktur hier den Vorteil, dass für beliebige Gruppierungen das jeweils längste gemeinsame Teilwort direkt ablesbar ist. Damit liegt der hier verfolgten Strategie keine globale Ähnlichkeitsbestimmung zugrunde, deren Berechnung durch die Levenshtein-Abstand für größere Abweichungsschranken oft aufwändig und teuer ist, sondern es werden nach dem Ausschlussprinzip anhand der längsten lokalen Übereinstimmung Zuweisungen vorgenommen. So können auch Einheiten korrekt zugeordnet werden, wenn diese nur noch sehr kurze Gemeinsamkeiten teilen, da andere mögliche Kandidaten bereits zuvor aussortiert, beziehungsweise anderen „besser“ passenden Partnern zugeteilt werden. Die für beide Anwendungsbeispiele erzielten Ergebnisse sind in Tabellen 8.2 und Abschnitt 8.3.2 zu finden.

Das letzte Kapitel in Teil III zeigt eine etwas komplexere Anwendung, die aber ebenfalls auf der Grundmenge quasimaximaler Knoten, die durch Algorithmus 17 bestimmt wird, fußt. Ziel war es, durch die Vorbestimmung dieser Menge eine Laufzeitverbesserung des in Kapitel 2.5 gezeigten Algorithmus zur Berechnung der längsten gemeinsamen Teilfolge zweier Wörter w^1 und w^2 zu erzielen. Die durch den SCDAWG-Index erreichte Komprimierung des Alphabets des LCS-Verfahrens ließ zwar, wie auch die Experimente in Kapitel 9.2 zeigen, bessere Laufzeiten zu, optimale globale Sequenzalignierungen konnten jedoch nicht in allen Fällen erreicht werden. Während Überlappungen, wie in Abschnitt 9.1.5 besprochen, erkannt und nachträglich korrigiert werden, können aufgrund der Tatsache, dass keine Relationen zwischen den Vorkommen quasimaximaler Knoten untereinander berücksichtigt werden, in manchen Fällen eigentlich vorhandene lokale Gemeinsamkeiten nicht immer erkannt werden, wie etwa Beispiel 9.1.5 zeigt. Die Nachkorrektur solcher falsch erkannter Gaps, die entweder iterativ durch die in Abschnitt 9.1.6 gezeigte Methode oder durch den Einsatz eines Standardverfahrens geschehen kann, verbessert die Alignmentqualität oft zwar erheblich, aufgrund des Problems *inkompatibler Teilwörter*, das am Ende von Abschnitt 9.1.4 beschrieben wird, können auch durch eine nachgeschaltete Korrektur nicht richtig

alignierte Gaps nicht zuverlässig korrigiert werden. Die Lösung dieser Problematik stellte sich als schwierig heraus, da im Vorfeld der eigentlichen Alignierung nur schwer eine Aussage getroffen werden kann, wann das Vorkommen eines kürzeren quasimaximalen Knotens anstelle eines längeren Vorkommens ausgewählt werden müsste. Dafür könnte als mögliche Verbesserung untersucht werden, ob durch die Berücksichtigung der Relationen quasimaximaler Knoten untereinander auch bestimmte Vorkommen innerhalb eines eigentlich längeren quasimaximalen Knotens mit berücksichtigt werden können und so ein kürzeres Vorkommen eines Infixes innerhalb eines längeren gleichzeitig mit seinem längeren Vorkommen innerhalb des LCS-Algorithmus berücksichtigt würde. Trotz dieser Problematik konnten aber auch für stark unterschiedliche Dokumente, wie etwa in Kapitel 9.2.3 untersucht, durch die nachgeschaltete Gap-Korrektur mit Hilfe von Hirschbergs Algorithmus [34] gute, wenn auch nicht optimale Ergebnisse erzielt werden. Die Verbesserung der Laufzeit durch die Komprimierung mittels des SCDAWG-Index zeigt sich vor allem, wenn große Texteinheiten auf einen Schlag aligniert werden sollen, wie Tabelle 9.4 in Abschnitt 9.2.2 belegt. Letztlich stellt Kapitel 9.3 noch eine Anwendung des indexgestützten Alignment-Verfahrens dar, bei der Texte verschiedener Sprachen miteinander aligniert werden. Wenngleich auch im dort benutzten Beispielkorpus verschiedener Übersetzungen des Neuen Testaments die Güte der erreichten Alignierungen nicht ganz optimal war, so lag sie mit etwa 80% doch so hoch, dass mit einer simplen Heuristik versucht werden konnte, Intervalle, die Regionen mit größeren Übereinstimmungen enthalten, zu bestimmen. Die gefundenen Intervalle wurden anschließend genutzt, um in den verschiedenen Sprachen Varianten der im Neuen Testament vorkommenden Eigennamen zu extrahieren. Insgesamt bietet das hier beschriebene Alignierungsverfahren eine Methode, deren Laufzeit für Anwendungen, bei denen sehr viele Texte nacheinander oder größere Textfragmente auf einmal aligniert werden, Vorteile bietet. Obwohl sich die hier beschriebene Methode auf eine paarweise Alignierung beschränkt, wäre eine Erweiterung des Verfahrens für multiple Alignierungen denkbar. Zum einen ist Algorithmus 17 nicht auf zwei Texte beschränkt, sondern liefert für beliebige Wortmengen deren längste gemeinsame Teilwörter im Kontext ihres Auftretens. Zum anderen ließe sich, wie von Gusfield [31] angemerkt, das LCS-Verfahren aus Kapitel 2.5 ebenfalls auf eine Anwendung für mehr als zwei Strings modifizieren. Damit stellt ein SCDAWG-gestütztes Multi-Alignment-Verfahren eine rein technische Hürde dar, die Modifikationen der in Kapitel 9.1.2 gezeigten Algorithmen bedeuten würde, sodass diese mehr als zwei Eingabewörter verarbeiten können.

Teil IV beschäftigt sich mit den für einzelne Texte oder durch Kategorisierung gebildete Textgruppen charakteristischen Teilwörtern. Diese werden als die minimalen Teilwörter gesehen, die nur in einem Text oder einer Gruppe von Texten auftreten. In Anlehnung an den Begriff des quasimaximalen Knotens wurde somit für

diese Art von Infixen die Notation des *quasiminimalen Knotens* eingeführt und diese durch Definition 11.1.1 formalisiert. Als Anwendungsgebiet des in 11.2 gezeigten Algorithmus 25 wurden in Kapitel 11.5 für eine Reihe von Textkollektionen die Mengen quasiminimaler Knoten bestimmt und durch ein einfaches Mehrheitsvoting Klassifikationen unbekannter Dokumente derselben Domäne vorgenommen. Für alle Testkorpora konnten mit dieser Methode gute bis sehr gute Ergebnisse erzielt werden. Sehr vorteilhaft erweist sich diese Art, charakteristische Einheiten zu bestimmen, auch für Sprachen, deren Wortgrenzen nicht durch ein spezielles Zeichen kodiert werden, da a priori keine Tokenisierung nötig ist, um alle quasiminimalen Knoten zu erkennen. Ein Klassifikationsbeispiel auf einem Korpus einer solchen Sprache ist in Kapitel 11.5.4 zu finden. Auch dort wurden sehr gute Klassifikationsergebnisse erzielt, was darauf hinweist, dass diese Methode für sinitische Sprachen gut geeignet scheint. Zur genaueren Auswertung dieser Klassifikation sollten noch weitere Vergleiche mit anderen machine-learning basierten Verfahren wie etwa Support-Vector-Maschinen [17] oder Multi-Layer-Perceptrons [60, 73] angestellt werden. Zur Untersuchung der Robustheit dieses Verfahrens kann der hier gewählte Ansatz, in dem genau so viele kürzeste distinkte Teilwörter verwendet werden, wie für die kleinste Klasse extrahiert werden konnten, abgeändert werden. So könnten zum Beispiel alle extrahierten Features genutzt werden oder aber eine kleinere Anzahl, wobei gleichzeitig auch andere Gewichtungsstrategien als die hier verwendete document-frequency evaluiert werden könnten.

Das Hauptthema dieser Arbeit war es, aufzuzeigen, wie flexibel ein SCDAWG, der über einer Menge von Texten aufgebaut wurde, für verschiedene Anwendungen aus den Bereichen der Computerlinguistik und der digitalen Geisteswissenschaften eingesetzt werden kann. Neben der Möglichkeit, Suchanfragen in linearer Zeit des Suchwortes zu beantworten, lässt ein SCDAWG auch Anfragen zu, die zu einem gefundenen Infix dessen beliebig lange linke und rechte Kontexte bezüglich der Äquivalenzrelation $\sim_w$ ermitteln können, indem entweder linke oder rechte Kanten von diesem aus verfolgt werden. Während diese Eigenschaft auch für viele Arten lokaler Suchanfragen einen großen Vorteil gegenüber konventionellen Indexierungssystemen bietet, lag der Fokus dieser Arbeit jedoch auf globalen Anfragen, die bestimmte Infixmengen berechnen. Auch dafür erwies es sich als Vorteil, alle Teilwörter einer Wortmenge W und deren Umkehrungen in einer Indexstruktur abzuspeichern. Durch die Vorberechnung der Indexstruktur und die nachgeschaltete Extraktion bestimmter Infix-Teilmengen können eine Vielzahl von Aufgaben auf einem Textkorpus bewältigt werden, ohne dass Einschränkungen bezüglich der Laufzeit entstehen. Gleichzeitig stellt es, wie von Crochemore [20] angemerkt, eine der erstaunlichsten Tatsachen auf dem Gebiet des Pattern-Matchings dar, dass die DAWG-Struktur und deren Weiterentwicklungen mit linearem Speicherverbrauch berechnet werden können.

A. Inhalt der beigefügten CD

Die beigefügte CD beinhaltet Implementierungen der ab Kapitel 3 beschriebenen Indexstrukturen sowie Beispielprogramme der in Kapiteln 6, 8, 9 und 11 gezeigten Anwendungen.

A.1. Aufbau der Indexstrukturen

Zunächst wird ein kommandozeilenbasiertes Java-Programm[1] betrachtet, das den Aufbau der in Kapitel 3 beschriebenen Indexstrukturen ermöglicht. Dabei werden die ebenfalls in Kapitel 3 gezeigten Algorithmen von Ukkonen, Blumer et al. und Inenaga et al. benutzt. Zur Implementierung der SCDAWG-Struktur wird die in Kapitel 5.2 gezeigte off-line Variante verwendet. Die Eingabe besteht dabei aus einer Textdatei[2], bei der jedes Wort aus der Wortmenge W genau eine Zeile einnimmt und somit mit einem Zeilenumbruch von nächsten Wort abgetrennt ist. Als Ausgabe wird jeweils eine Datei im *.dot* Format ausgegeben, die eine Visualisierung der aufgebauten Indexstruktur erlaubt. Im Folgenden wird ein Beispielaufruf des Programms gezeigt:

```
java -jar build_indexstructures.jar -s stree -i test.txt -o example
```

Dabei können folgende Parameter angegeben werden:

-i: Der Pfad der Eingabedatei - *erforderlich.*

-o: Der Name der Ausgabedatei (ohne Dateiendung) - *erforderlich.*

-s: Die aufzubauende Indexstruktur (`strie` | `stree` | `dawg` | `cdawg` | `scdawg`) - *erforderlich.*

-d: Einschalten von Debug-Meldungen - *optional.*

[1] Zum fehlerfreien Ablauf aller hier gezeigen Java-Programme sollte mindestens Java SE 8 benutzt werden.

[2] Als Zeichenkodierung aller Eingabedateien sollte UTF-8 gewählt werden und zur korrekten Ausgabe zusätzlich beim Aufruf die Option `-Dfile.encoding=utf8` gesetzt werden. Also `java -Dfile.encoding=utf8 -jar <Programmname>.jar <Programmparameter>`.

Wird das Programm *build_indexstructures.jar* etwa mit einer Testdatei, die die beiden Wörter *abcbc* und *abcab* enthält, aufgerufen, entsteht eine Ausgabedatei `example_stree.dot`, die mit der Graphenvisualisierungs-Software Graphviz [30] in eine graphische Darstellung überführt werden kann. Als Start- und Endmarker dienen stets die Zeichen # und $. Diese müssen nicht extra hinzugefügt werden und sollten an keiner anderen Stelle in der Eingabedatei auftreten.

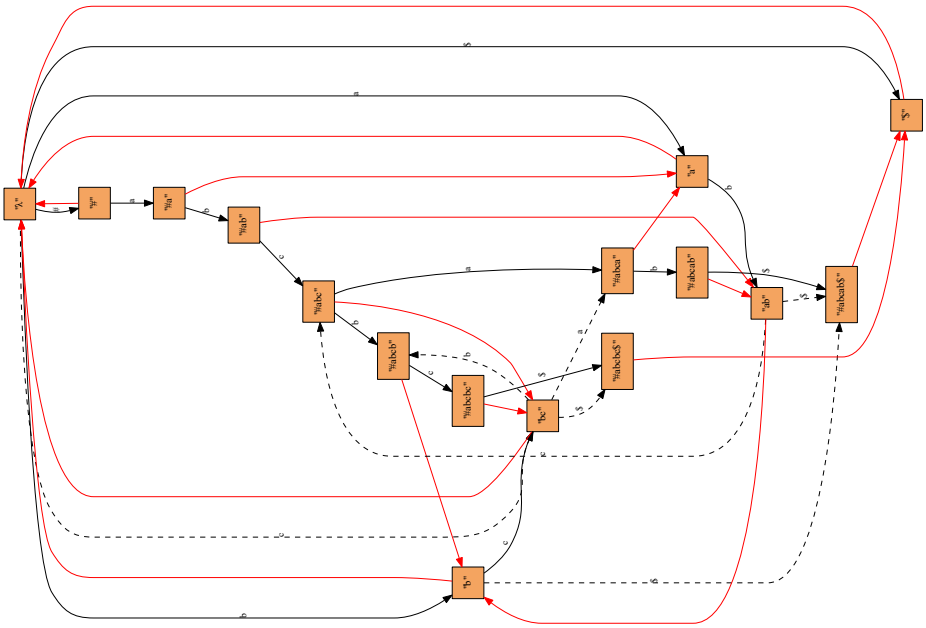


Abbildung A.1.: Mit *build_indexstructures.jar* erzeugte Visualisierung eines Beispiel-DAWGs der Wörter *abcbc* und *abcab*.

Gemäß der in dieser Arbeit verwendeten Konventionen entsprechen schwarze Kanten normalen Rechtsübergängen, blaue Linksübergängen und rot gefärbte Kanten Suffixlinks. Die im DAWG gestrichelten schwarzen Kanten stellen sekundäre Kanten dar (siehe Definition 3.5.5).

On-line Implementierung

Zur in Kapitel 5.1 gezeigten on-line Implementierung der SCDAWG-Struktur wird ein Python-Programm gegeben[3], das ebenfalls zu einer UTF-8 kodierten Eingabedatei eine Ausgabe im .dot-Format erzeugt. Dort werden ebenfalls mit *-i* und *-o* Eingabe- und Ausgabedateien angegeben. Zusätzlich ist auch noch eine in Python geschriebene off-line Variante enthalten, die gleichmaßen aufgerufen wird. Damit werden die on-line und off-line SCDAWG-Implementierungen in Python beispielsweise folgendermaßen aufgerufen:

```
python online_scdawg.py -i test.txt -o example
python offline_scdawg.py -i test.txt -o example
```

A.2. On-line Suchfunktionen

Die in Kapitel 6 gezeigten Index-Suchfunktionen *find_p*, *find_freq* und *find_loc* werden erneut anhand eines Java-Programms *search_index.jar* beispielhaft umgesetzt. Für die Wörter aus Beispiel 6.5.1 mit $W = \{\#cockatoo\$, \#crocodile\$, \}$ ließen sich durch folgenden Aufruf die dort dargestellten Ergebnisse generieren:

```
java -jar search_index.jar -q co -i test.txt -fp -fl -ff
```

Dabei können folgende Parameter angegeben werden:

-i: Der Pfad der Eingabedatei - *erforderlich.*

-q: Der Querystring - *erforderlich.*

-ff: Suche nach längsten Prefix - *optional.*

-fl: Ausgabe aller Häufigkeiten - *optional.*

-fp: Ausgabe der Vorkommenspositionen - *optional.*

-d: Einschalten von Debug-Meldungen - *optional.*

A.3. Finden der längsten gemeinsamen Teilwörter

Der in Kapitel 8 gezeigte Algorithmus 17 zur Identifikation der längsten gemeinsamen Teilwörter im Kontext ihres Vorkommens einer Wortmenge W wird ebenfalls

[3] Zum fehlerfreien Ablauf sollte Python 3 benutzt werden.

durch ein Java-Programm realisiert. Dabei wird erneut eine Eingabedatei eingelesen, die alle zu indexierenden Texte enthält, und anschließend die Liste aller Vorkommen bedingt quasimaximaler Knoten ausgegeben. Dieses lässt sich beispielsweise so aufrufen:

```
java -jar find_longest_common_substrings.jar -i test.txt
```

Dabei können folgende Parameter angegeben werden:

-i: Der Pfad der Eingabedatei - *erforderlich.*

-d: Einschalten von Debug-Meldungen - *optional.*

Die Ausgabe entspricht den in Beispielen 8.1.6 gezeigten Listendarstellungen quasimaximaler Knoten.

A.4. Paarweise globale Alignierung

Zur paarweisen globalen Alignierung wird ein Java-Programm *align_sc.jar* angegeben, das mit einer Textdatei, die genau zwei zu alignierende Strings enthält, aufgerufen wird. Dabei können die in Kapitel 9 beschriebenen Verfahren $align_{SC}$, $align_{SC+H}$, $align_{SC+IT}$ benutzt werden. Ein Beispielaufruf wird folgendermaßen angegeben:

```
java -jar align_SC.jar -i test_paar.txt -a1 -o example -f json -e
```

Dabei können folgende Parameter angegeben werden:

-a1: Alignment mit $align_{SC}$ - *optional.*

-a2: Alignment mit $align_{SC+IT}$ - *optional.*

-a3: Alignment mit $align_{SC+H}$ - *optional.*

-d: Einschalten von Debug-Meldungen - *optional.*

-e: Ausgabe der Alignmentqualität - *optional.*

-f: Angabe des Ausgabeformats (json|html) - *erforderlich.*

-i: Der Pfad der Eingabedatei - *erforderlich.*

-o: Der Name der Ausgabedatei (ohne Dateiendung) - *erforderlich.*

Mit den Optionen *a1*, *a2* und *a3* wird die jeweilige Alignmentmethode ausgewählt. Der Parameter *-f* gibt an, in welchem Format die Ausgabe erfolgen soll. Damit ein korrekter Ablauf gewährleistet wird, sollte mindestens eines der drei Alignierungsverfahren angegeben werden. Neben einer Ausgabe im JSON-Format existiert die Möglichkeit, eine HTML-Ausgabe zu generieren. Hierfür werden entsprechende CSS-Stylesheets mitgeliefert, die eine Ausgabe, wie sie anhand einer Beispielalignierung in der nächsten Abbildung gezeigt ist, erzeugen.

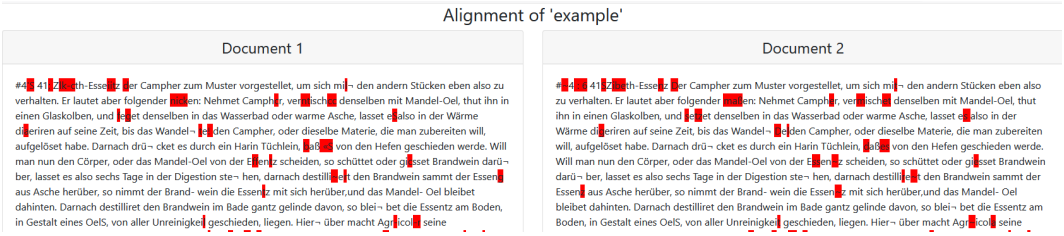


Abbildung A.2.: Mit *align_sc.jar* erzeugte Alignierung einer durch zwei OCR-Engines erkannten Beispielseite.

Nachstehend ist ein Auszug der alignierten Beispielseite im JSON-Format zu sehen:

```
{
        "match": " Wunder, wie",
        "endpos_s1": "3240",
        "endpos_s2": "3241",
        "gap_s1": "kSdievcrlv",
        "gap_s2": " es die verlo"
},{
        "match": "hrn",
        "endpos_s1": "3253",
        "endpos_s2": "3257",
        "gap_s1": "n ",
        "gap_s2": "en"
},{
        "match": "l Kräff",
        "endpos_s1": "3262",
        "endpos_s2": "3266",
        "gap_s1": "l",
        "gap_s2": "t"
},{...
```

Weiterhin kann mit der Option *-e* das in Kapitel 9.2.1 beschriebene Maß zum Vergleich der SCDAWG-gestützten Alignierungsverfahren mit einer optimalen Alignmentmethode ausgegeben werden. Dabei sollte aber darauf geachtet werden, dass die Zeitkomplexität der opitmalen Vergleichsverfahren quadratisch ist.

A.5. Finden textueller Charakterisitiken

Das Auffinden charakteristischer Teilwörter, die in Kapitel 11 behandelt wurden, wird durch ein Java-Programm *find_shortest_distinct_substrings.jar* realisiert, das eine Implementierung von Algorithmus 25 darstellt.

```
java -jar find_shortest_distinct_substrings.jar -i test.txt -m metalist.txt
```

Dabei können folgende Parameter angegeben werden:

-i: Der Pfad der Eingabedatei - *erforderlich.*

-m: Pfad zur Liste der Metadaten-Kategorien - *erforderlich.*

-d: Einschalten von Debug-Meldungen - *optional.*

Neben einer Eingabedatei, die wiederum jeden zu indexierenden Text in einer Zeile enthält, muss eine parallele Liste angegeben werden, die für jeden eingelesenen Text dessen Metadatenkategorie angibt. Der Pfad dieser Textdatei wird mit der Option *-m* angeben. Die folgende Tabelle zeigt zwei parallele Beispiellisten, bei welchen die ersten drei Texte der Metadatenkategorie *A* und die beiden letzten der Kategorie *B* zugeordnet sind.

Inputext	*Metakategorie*
abcbc	A
abcab	A
ababc	A
cocoa	B
cacoao	B

Tabelle A.1.: Beispieleingabe für *find_shortest_distinct_substrings.jar.*

Hieraus ergibt sich nun für Kategorie *A* als quasiminimaler Knoten „*b*“ und für die Kategorie *B* die Knoten „*o*“und „*#c*“, wobei das Symbol # den künstlich hinzugefügten Startmarker darstellt. Zusätzlich wird zu jedem gefundenen kürzesten Teilwort dessen document-frequency ausgegeben.

Literatur

[1] Alfred V Aho and Margaret J Corasick. "Efficient string matching: an aid to bibliographic search". In: *Communications of the ACM* 18.6 (1975), pp. 333–340.

[2] Ehsaneddin Asgari and Hinrich Schütze. "Past, present, future: A computational investigation of the typology of tense in 1000 languages". In: *arXiv preprint arXiv:1704.08914* (2017).

[3] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*. Vol. 463. ACM press New York, 1999.

[4] Daniel Bär, Torsten Zesch, and Iryna Gurevych. "A reflective view on text similarity". In: *Proceedings of the International Conference Recent Advances in Natural Language Processing 2011*. 2011, pp. 515–520.

[5] Daniel Bär, Torsten Zesch, and Iryna Gurevych. "Text reuse detection using a composition of text similarity measures". In: *Proceedings of COLING 2012*. 2012, pp. 167–184.

[6] Eduardo Bernot and Enrique Alarcón. *Index Thomasticus*. 2005. URL: `http://www.corpusthomisticum.org/it/index.age` (visited on 08/12/2019).

[7] Anselm Blumer, Andrzej Ehrenfeucht, and David Haussler. "Average sizes of suffix trees and dawgs". In: *Discrete Applied Mathematics* 24.1-3 (1989), pp. 37–45.

[8] Anselm Blumer et al. "Complete inverted files for efficient text retrieval and analysis". In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 578–595.

[9] Anselm Blumer et al. "The smallest automation recognizing the subwords of a text". In: *Theoretical computer science* 40 (1985), pp. 31–55.

[10] Robert S Boyer and J Strother Moore. "A fast string searching algorithm". In: *Communications of the ACM* 20.10 (1977), pp. 762–772.

[11] Thomas Breuel. "Recent progress on the OCRopus OCR system". In: *Proceedings of the International Workshop on Multilingual OCR*. ACM. 2009, p. 2.

[12] Thomas M Breuel. "The OCRopus open source OCR system". In: *Document Recognition and Retrieval XV*. Vol. 6815. International Society for Optics and Photonics. 2008, 68150F.

[13] Sergey Brin, James Davis, and Hector Garcia-Molina. “Copy detection mechanisms for digital documents”. In: *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. 1995, pp. 398–409.

[14] Gerth Stølting Brodal et al. “Faster algorithms for computing longest common increasing subsequences”. In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2006, pp. 330–341.

[15] Stefan Büttcher, Charles LA Clarke, and Gordon V Cormack. *Information retrieval: Implementing and evaluating search engines*. Mit Press, 2016.

[16] Wikimedia Commons. File: `Nabonidus_cylinder_sippar_bm1.jpg`. 2007. URL: `https://commons.wikimedia.org/wiki/File:Nabonidus%5C_cylinder%5C_sippar%5C_bm1.jpg` (visited on 02/25/2020).

[17] Corinna Cortes and Vladimir Vapnik. “Support vector machine”. In: *Machine learning* 20.3 (1995), pp. 273–297.

[18] Maxime Crochemore and Wojciech Rytter. *Jewels of stringology: text algorithms*. World Scientific, 2003.

[19] Maxime Crochemore and Wojciech Rytter. *Text algorithms*. Maxime Crochemore, 1994.

[20] Maxime Crochemore and Renaud Vérin. “On compact directed acyclic word graphs”. In: *Structures in Logic and Computer Science*. Springer, 1997, pp. 192–211.

[21] *Das Buch des heyligen Römischen Reichs unnderhalltunge*. URL: `http://mdz-nbn-resolving.de/urn:nbn:de:bvb:12-bsb10941763-1` (visited on 01/13/2020).

[22] Helmut De Boor. *Corpus der altdeutschen Originalurkunden bis zum Jahr 1300*. Vol. 1. M. Schauenburg, k.-g., 1929.

[23] Rene De La Briandais. “File searching using variable length keys”. In: *Papers presented at the the March 3-5, 1959, western joint computer conference*. ACM. 1959, pp. 295–298.

[24] Cees Elzinga, Sven Rahmann, and Hui Wang. “Algorithms for subsequence combinatorics”. In: *Theoretical Computer Science* 409.3 (2008), pp. 394–404.

[25] Tobias Englmeier, Florian Fink, and Klaus U Schulz. “AI-PoCoTo: Combining Automated and Interactive OCR Postcorrection”. In: *Proceedings of the 3rd International Conference on Digital Access to Textual Cultural Heritage*. ACM. 2019, pp. 19–24.

[26] Florian Fink and Uwe Springmann. *CIS-OCR-Testset*. URL: https://github.com/cisocrgroup/Resources/tree/master/ocrtestset (visited on 01/29/2020).

[27] Edward Fredkin. "Trie memory". In: *Communications of the ACM* 3.9 (1960), pp. 490–499.

[28] Valentinos Georgiades and Minh Nguyen. *Hirschberg.java*. URL: https://github.com/sodhibhupinder/learn/blob/master/src/learn/Hirschberg.java (visited on 01/04/2020).

[29] Robert Giegerich and Stefan Kurtz. "From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction". In: *Algorithmica* 19.3 (1997), pp. 331–353.

[30] *Graphviz, - Graph Visualisation Software*. URL: https://www.graphviz.org/ (visited on 06/04/2020).

[31] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

[32] Diether Haacke. "Das Corpus der altdeutschen Originalurkunden". In: *Beiträge zur Geschichte der deutschen Sprache und Literatur (PBB)* 1955.77 (1955), pp. 375–392.

[33] Elisa Herrmann. "Von der Vision zur Umsetzung: Der aktuelle Entwicklungsstand von OCR-D". In: ().

[34] Daniel S. Hirschberg. "A linear space algorithm for computing maximal common subsequences". In: *Communications of the ACM* 18.6 (1975), pp. 341–343.

[35] Daniel S Hirschberg. "Bounds on the number of string subsequences". In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 1999, pp. 115–122.

[36] Daniel S Hirschberg and Mireille Regnier. "Tight bounds on the number of string subsequences". In: *Journal of Discrete Algorithms* 1.1 (2000), pp. 123–132.

[37] James W Hunt and Thomas G Szymanski. "A fast algorithm for computing longest common subsequences". In: *Communications of the ACM* 20.5 (1977), pp. 350–353.

[38] *IMPACT Centre of Competence*. URL: https://www.digitisation.eu/ (visited on 12/23/2019).

[39] Shunsuke Inenaga. "Bidirectional construction of suffix trees". In: *Nord. J. Comput.* 10.1 (2003), p. 52.

[40] Shunsuke Inenaga. "String Processing Algorithms". PhD thesis. Kyushu University, 2003.

[41] Shunsuke Inenaga et al. "On-line construction of compact directed acyclic word graphs". In: *Discrete Applied Mathematics* 146.2 (2005), pp. 156–179.

[42] Shunsuke Inenaga et al. "On-line construction of symmetric compact directed acyclic word graphs". In: *Proceedings Eighth Symposium on String Processing and Information Retrieval*. IEEE. 2001, pp. 96–110.

[43] Fotis Jannidis, Hubertus Kohle, and Malte Rehbein. "Digital Humanities". In: *Eine Einführung, Stuttgart: Metzler* (2017).

[44] Donald E Knuth, James H Morris Jr, and Vaughan R Pratt. "Fast pattern matching in strings". In: *SIAM journal on computing* 6.2 (1977), pp. 323–350.

[45] S Rao Kosaraju. "Efficient tree pattern matching". In: *30th Annual Symposium on Foundations of Computer Science*. IEEE. 1989, pp. 178–183.

[46] Vladimir I Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady*. Vol. 10. 8. 1966, pp. 707–710.

[47] Moritz G Maaß. "Linear bidirectional on-line construction of affix trees". In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2000, pp. 320–334.

[48] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge university press, 2008.

[49] William J Masek and Michael S Paterson. "A faster algorithm computing string edit distances". In: *Journal of Computer and System sciences* 20.1 (1980), pp. 18–31.

[50] Edward M McCreight. "A space-economical suffix tree construction algorithm". In: *Journal of the ACM (JACM)* 23.2 (1976), pp. 262–272.

[51] Donald R Morrison. "PATRICIA—practical algorithm to retrieve information coded in alphanumeric". In: *Journal of the ACM (JACM)* 15.4 (1968), pp. 514–534.

[52] Gonzalo Navarro. "A guided tour to approximate string matching". In: *ACM computing surveys (CSUR)* 33.1 (2001), pp. 31–88.

[53] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[54] Saul B Needleman and Christian D Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of molecular biology* 48.3 (1970), pp. 443–453.

[55] Theodor H Nelson. "Complex information processing: a file structure for the complex, the changing and the indeterminate". In: *Proceedings of the 1965 20th national conference*. ACM. 1965, pp. 84–100.

[56] Anil Nerode. "Linear automaton transformations". In: *Proceedings of the American Mathematical Society* 9.4 (1958), pp. 541–544.

[57] Clemens Neudecker et al. "OCR-D: An end-to-end open source OCR framework for historical printed documents". In: *Proceedings of the 3rd International Conference on Digital Access to Textual Cultural Heritage*. ACM. 2019, pp. 53–58.

[58] Franz Josef Och and Hermann Ney. "A systematic comparison of various statistical alignment models". In: *Computational linguistics* 29.1 (2003), pp. 19–51.

[59] Daisuke Okanohara and Jun'ichi Tsujii. "Text categorization with all substring features". In: *Proceedings of the 2009 SIAM International Conference on Data Mining*. SIAM. 2009, pp. 838–846.

[60] Sankar K Pal and Sushmita Mitra. "Multilayer perceptron, fuzzy sets, classifiaction". In: (1992).

[61] Christos Papadopoulos et al. "The IMPACT dataset of historical document images". In: *Proceedings of the 2Nd international workshop on historical document imaging and processing*. ACM. 2013, pp. 123–130.

[62] Mike Paterson and Vlado Dančík. "Longest common subsequences". In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 1994, pp. 127–142.

[63] Stefan Pletschacher and Apostolos Antonacopoulos. "The page (page analysis and ground-truth elements) format framework". In: *2010 20th International Conference on Pattern Recognition*. IEEE. 2010, pp. 257–260.

[64] Gunter Saake and Kai-Uwe Sattler. *Algorithmen und Datenstrukturen: Eine Einführung mit Java*. dpunkt. verlag, 2014.

[65] Peter H Sellers. "On the theory and computation of evolutionary distances". In: *SIAM Journal on Applied Mathematics* 26.4 (1974), pp. 787–793.

[66] Peter H Sellers. "The theory and computation of evolutionary distances: pattern recognition". In: *Journal of algorithms* 1.4 (1980), pp. 359–373.

[67] Jangwon Seo and W Bruce Croft. "Local text reuse detection". In: *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*. 2008, pp. 571–578.

[68] David A Smith et al. "Detecting and modeling local text reuse". In: *IEEE/ACM Joint Conference on Digital Libraries*. IEEE. 2014, pp. 183–192.

[69] Ray Smith. "An overview of the Tesseract OCR engine". In: *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*. Vol. 2. IEEE. 2007, pp. 629–633.

[70] Ray Smith. "Tesseract ocr engine". In: *Lecture. Google Code. Google Inc* (2007).

[71] Uwe Springmann et al. "Ground Truth for training OCR engines on historical documents in German Fraktur and Early Modern Latin". In: *arXiv preprint arXiv:1809.05501* (2018).

[72] Jens Stoye. "Affix trees". In: (2000).

[73] Jiexiong Tang, Chenwei Deng, and Guang-Bin Huang. "Extreme learning machine for multilayer perceptron". In: *IEEE transactions on neural networks and learning systems* 27.4 (2015), pp. 809–821.

[74] *TextGridRep*. URL: `https://textgridrep.org/` (visited on 12/30/2019).

[75] Esko Ukkonen. "On-line construction of suffix trees". In: *Algorithmica* 14.3 (1995), pp. 249–260.

[76] Bálint Vásárhelyi. "An estimation of the size of non-compact suffix trees". In: *arXiv preprint arXiv:1604.01168* (2016).

[77] Aleksi Vesanto et al. "Applying BLAST to Text Reuse Detection in Finnish Newspapers and Journals, 1771-1910". In: *Proceedings of the NoDaLiDa 2017 Workshop on Processing Historical Language*. 2017, pp. 54–58.

[78] Robert A Wagner and Michael J Fischer. "The string-to-string correction problem". In: *Journal of the ACM (JACM)* 21.1 (1974), pp. 168–173.

[79] Peter Weiner. "Linear pattern matching algorithms". In: *14th Annual Symposium on Switching and Automata Theory (swat 1973)*. IEEE. 1973, pp. 1–11.

[80] Jiang Yanting. *Pre-modern Chinese corpus*. 2020. URL: `https://github.com/JiangYanting/Pre-modern_Chinese_corpus_dataset` (visited on 03/21/2020).

Danksagung

Meinem Doktorvater Prof. Dr. Klaus U. Schulz möchte ich zuallererst für seine überaus kompetente, dabei immer freundliche und gleichzeitig ruhige Art der Betreuung danken. Prof. Schulz hatte immer ein offenes Ohr für meine Anliegen und Ideen und hat mich so auch in schwierigen Phasen vor möglichen Irrwegen bewahrt und stets akribisch auf Probleme und mögliche Fehler aufmerksam gemacht.

Ebenfalls möchte ich Dr. Stefan Gerdjikov meinen tiefsten Dank aussprechen. Stefan Gerdjikov ist nicht nur einer der freundlichsten und geduldigsten Menschen, die mir je begegnet sind, sondern auch eine Koryphäe im Bereich der String-Indexstrukturen, ohne dessen Einsichten und Unterstützung diese Arbeit nicht zum Erfolg geführt hätte.

Weiterhin danke ich Prof. Dr. Hinrich Schütze, Dr. Uwe Springmann, Dr. Jamie Novotny, Dr. Frauke Weiserhäuser, Dr. Helmut Schmid, Ursula Meier-Credner und Stefanie Schneider für die Bereitstellung einiger Datensätze, die in den Experimenten dieser Arbeit Verwendung fanden. Auch Dr. Florian Fink danke ich für die vielen sachkundigen Gespräche und die daraus entstandene Unterstützung.

Außerdem möchte ich mich herzlich bei Christiane Bayer bedanken, die mir zum Ende der Arbeit viel geholfen und diese Korrektur gelesen hat.

Letztlich danke ich meiner Familie. Meinen Brüdern David und Elias und meiner Schwester Miriam danke ich für ihre Freundschaft und die generelle Unterstützung dieser Arbeit. Meinem Vater danke ich für die ebenfalls wohlwollende Unterstützung und auch dafür, mich früh für Programmierung und Informatik begeistert zu haben. Der größte Dank gebührt aber meiner Mutter für ihre Klugheit und Geduld und die bedingungslose Unterstützung, der ich mir von der ersten Minute meines Lebens bis heute sicher sein konnte.