Textalgorithmen

Vorlesung im Wintersemester 2005/06
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik

Ralf Stiebe
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
D-39106 Magdeburg
Email: stiebe@iws.cs.uni-magdeburg.de

Einleitung

Die Verarbeitung langer Zeichenketten (Strings, Texte) ist eine der grundlegenden Aufgaben in der Informatik. Es ist deshalb wenig verwunderlich, dass bereits ab 1970 zahlreiche Probleme aus diesem Gebiet formuliert und effizient gelöst wurden. Neue Herausforderungen ergaben sich in jüngerer Zeit durch das Aufkommen des *Internet* und der *Bioinformatik*.

Diese Vorlesung soll eine Einführung in einige wichtige Probleme und Algorithmen für Zeichenketten geben. Sie gliedert sich in folgende Abschnitte:

1. Exakte Wortsuche: Gegeben sind Wörter P (*Pattern*) und T (*Text*). Gesucht sind alle Vorkommen von P in T. Dieses Problem kommt sicherlich in jeder größeren Anwendung vor. Es existieren zahlreiche und zum Teil sehr unterschiedliche Algorithmen, die dieses Problem effizient lösen.

2. Exakte Suche nach mehreren Wörtern: Gegeben sind eine endliche Menge von Wörtern $\mathcal{P}$ und ein Text T. Gesucht sind alle Vorkommen von Wörtern aus $\mathcal{P}$ in T. Verwandte Aufgaben sind die exakte Suche in *Wörterbüchern*, bei der für ein Wort T entschieden werden soll, ob es in der Menge $\mathcal{P}$ (dem Wörterbuch) vorkommt, sowie die exakte Suche nach zweidimensionalen *Bildern*.

3. Ähnlichkeit von Zeichenketten und inexakte Suche: Das einfachste Maß für die Ähnlichkeit zweier Wörter S_1 und S_2 ist der *Levenshtein-Abstand*: die minimale Anzahl von Operationen der Form Ersetzen/Einfügen/Streichen eines Zeichens, um S_1 in S_2 umzuwandeln. Dieser Abstandsbegriff ist die Grundlage zahlreicher anderer Ähnlichkeitsmaße. Außerdem sucht man häufig nach *lokaler Ähnlickeit*, d.h. nach Regionen mit hoher Ähnlichkeit in ansonsten sehr verschiedenen Zeichenketten. Die Bestimmung des Levenshtein-Abstandes wie auch der lokalen Ähnlichkeiten ist duch *dynamische Programmierung* in quadratischer Zeit möglich.
   Bei der *inexakten Suche* sucht man alle Teilwörter in einem Text T, die zu einem gegebenen Suchwort P einen Levenshtein-Abstand von höchstens k besitzen. Neben der dynamischen Programmierung existieren für dieses Problem auch einige andere Lösungsansätze.
   Die Suche nach Ähnlichkeiten in mehr als zwei Zeichenketten (*multiple Alignments*) ist ein sehr kompliziertes Problem und von großer Bedeutung in der Bioinformatik. Im Rahmen dieser Vorlesung gehen wir auf diese Aufgabenstellung aber nur sehr kurz ein.

4. Indexstukturen für Texte: Ein Text T wird in einer Vorbereitungsphase (*Präprozessing*) so aufbereitet, dass man die Vorkommen beliebiger Suchwörter finden kann, ohne den Text zu durchsuchen. Im Idealfall ist der Suchaufwand unabhängig von der Textlänge. Es wurden verschiedene Datenstrukturen für die Indizierung entwickelt. In

dieser Vorlesung werden vor allem *Suffixbäume* und *Suffixarrays* behandelt. Indexstrukturen haben eine große Bedeutung für alle Anwendungen, bei denen häufig in (relativ) konstanten Daten gesucht werden soll, z.B. bei Suchmaschinen im Internet sowie bei Gen-Datenbanken. Weitere Anwendungen sind die Suche nach Regularitäten in Texten (z.B. lange Wiederholungen, sehr häufige Teilwörter), die für die Datenkompression sowie in der Bioinformatik von Interesse sind.

Neben ihrer großen Bedeutung in der Anwendung stellen die Textalgorithmen auch ein sehr interessantes theoretisches Studienobjekt dar. So kann man die Anwendung grundlegender Datenstrukturen und algorithmischer Techniken sowie verschiedener Beweisprinzipien gerade bei diesen doch recht elementaren Fragestellungen sehr gut studieren. Da diese Vorlesung im Rahmen der theoretischen Informatik gehalten wird, stehen diese Aspekte vielleicht etwas stärker im Vordergrund als in anderen Abhandlungen.

Literatur

In den letzten Jahren sind einige Bücher (in englischer Sprache) über Textalgorithmen erschienen, u.a. von Apostolico und Galil [1], Crochemore und Rytter [3, 4], Gusfield [6], Navarro und Raffinot [12] Smyth [14] und Stephen [13]. Auf französisch gibt es ein Buch von Crochemore, Hancart und Lecroq [5]. In den genannten Büchern findet man zahlreiche Verweise auf die Originalartikel. Die exakte Wortsuche wird außerdem in zahlreichen allgemeinen Büchern über Algorithmen betrachtet. Besonders ausführlich geschieht dies in deutscher Sprache im Buch von Heun [7].

Dieses Skript orientiert sich in großen Teilen am Buch von Gusfield [6], das ein sehr gut geschriebenes Lehrbuch ist und außerdem einen umfassenden Einblick in die Anwendung von Textalgorithmen in der Bioinformatik vermittelt. Für den praktisch interessierten Leser ist vor allem das Buch von Navarro und Raffinot [12] zu empfehlen, das vor allem die in den letzten Jahren entwickelten und in der Praxis sehr schnellen Algorithmen beschreibt und vergleicht. Zahlreiche Quellen gibt es auch im Internet. Hier seien nur drei genannt:

- `http://www.dei.unipd.it/~stelo/`
  eine umfangreiche Link-Sammlung von Stefano Lonardi,
- `http://www-igm.univ-mlv.fr/~lecroq/lec_en.html`
  die Homepage von Thierry Lecroq mit einer umfangreichen Präsentation (einschließlich Animationen) von Algorithmen zur exakten Wortsuche und zum Sequenzvergleich sowie einer umfassenden Bibliographie zu Textalgorithmen,
- `http://dcc.uchile.cl/~gnavarro/eindex.html`
  die Homepage von Gonzalo Navarro mit zahlreichen Originalarbeiten und Übersichtsartikeln aus den letzten Jahren sowie einigen nützlichen Programmen.

Kapitel 0

Grundlegende Begriffe und Notationen

0.1 Mathematische Notationen

Die Menge der natürlichen Zahlen (einschließlich 0) wird mit $\mathbb{N}$ bezeichnet. Die Potenzmenge einer Menge M wird mit $\mathcal{P}(M)$, die Mächtigkeit einer Menge M wird mit $|M|$, die leere Menge wird mit $\emptyset$ bezeichnet. Für eine reelle Zahl x ist $\lfloor x \rfloor$ die größte ganze Zahl, die nicht größer als x ist, sowie $\lceil x \rceil$ die kleinste ganze Zahl, die nicht kleiner als x ist. Auf den natürlichen Zahlen definieren wir die ganzzahlige Division *div* sowie die Modulo-Operation *mod* vermöge $n \,div\, m := \lfloor n/m \rfloor$ sowie $n \,mod\, m = n - (m \cdot (n \,div\, m))$.

0.2 Algorithmen und ihre Komplexität

Es werden die üblicherweise in einer Grundvorlesung *Algorithmen und Datenstrukturen* vermittelten Kenntnisse vorausgesetzt. Die Notation von Algorithmen in Pseudocode folgt den Konventionen aus dem Standard-Lehrbuch *Introduction to Algorithms* von Cormen, Leiserson und Rivest [2]. Insbesondere wird der Rumpf einer Schleife durch die Tiefe der Einrückung erkennbar.

Laufzeiten von Algorithmen werden in der Regel asymptotisch mittels der bekannten O-Notation ausgedrückt, die hier der Vollständigkeit halber definiert werden soll.

Definition 0.1 *Für eine Funktion $g : \mathbb{N} \to \mathbb{N}$ sind $O(g)$, $\Omega(g)$ bzw. $\Theta(g)$ die Funktionenklassen*

$$\begin{aligned} O(g) &= \{f : \mathbb{N} \to \mathbb{N} \mid \exists c \exists n_0 \forall n (n \geq n_0 \to f(n) \leq c \cdot g(n))\}, \\ \Omega(g) &= \{f : \mathbb{N} \to \mathbb{N} \mid g \in O(f)\}, \\ \Theta(g) &= \Omega(g) \cap O(g). \end{aligned}$$

Das heißt, $O(g)$ ist die Menge aller Funktionen, die bis auf einen konstanten Faktor höchstens so schnell wie g wächst; $\Omega(g)$ ist die Menge aller Funktionen, die bis auf einen konstanten Faktor mindestens so schnell wie g wächst; $\Theta(g)$ ist die Menge aller Funktionen, die bis auf einen konstanten Faktor genau so schnell wie g wächst. Für weitere Betrachtungen zu Komplexitäten von Algorithmen siehe ebenfalls [2].

Weiterhin verwenden wir das sogenannte Einheitskostenmodell (*unit cost model*), d.h. wir gehen davon aus, dass für Zahlen in der Größenordnung der betrachteten Wortlängen die arithmetischen Grundoperationen in konstanter Zeit ausgeführt werden können. Diese Annahmen sind durchaus realistisch, da heutige Rechner 32 Bit in einem Schritt verarbeiten können und die Länge der Texte in fast allen Anwendungen durch 2^{32} beschränkt ist.

0.3 Zeichenketten

Für ein Wort S wird die Länge von S durch $|S|$ bezeichnet. Das Wort der Länge 0 heißt das *leere Wort* und wird mit ε bezeichnet.

Es seien u und v Wörter. Gilt $u = u_1vu_2$, so nennt man v ein *Teilwort* oder ein *Infix* oder einen *Faktor* von u. Gilt $u = vu_2$, so nennt man v ein *Präfix* von u. Gilt $u = u_1v$, so nennt man v ein *Suffix* von u. Ist v ein Teilwort (bzw. Präfix bzw. Suffix) von u mit $v \neq u$, so nennt man v ein *echtes Teilwort* (bzw. *echtes Präfix* bzw. *echtes Suffix*) von u.

Für $1 \leq i \leq |u|$ ist $u[i]$ das Zeichen an der Stelle i von u. Das Teilwort von der Stelle i bis (einschließlich) zur Stelle j von u ist $u[i \dots j]$. Gilt $i > j$ oder $i > |u|$, so ist $u[i \dots j]$ *per definitionem* das leere Wort.

Die Menge aller Wörter über einem Alphabet Σ bezeichnen wir mit Σ^*, die Menge aller nichtleeren Wörter mit Σ^+, die Menge aller Wörter der Länge n mit Σ^n. Mit u^r bezeichnen wir das Wort u von rechts nach links gelesen.

Eine grundlegende Operation ist der Vergleich zweier Buchstaben. Ergibt sich bei einem solchen Vergleich eine Übereinstimmung, so sprechen wir von einem *Match*, anderenfalls von einem *Mismatch*.

Es sei $(\Sigma, <)$ ein total geordnetes Alphabet. Die *lexikografische Ordnung* $<_{\text{lex}}$ auf Σ^* erhält man wie folgt:

1. $\varepsilon <_{\text{lex}} \alpha$ für alle $\alpha \in \Sigma^+$.
2. Aus $a < b$ folgt $a\alpha <_{\text{lex}} b\beta$ für alle $\alpha, \beta \in \Sigma^*$.
3. Aus $\alpha <_{\text{lex}} \beta$ folgt $a\alpha <_{\text{lex}} a\beta$ für alle $a \in \Sigma$.

0.4 Endliche Automaten

Schließlich benötigen wir noch den Begriff des endlichen Automaten. Ausführlich werden endliche Automaten in einführenden Büchern zur Theoretischen Informatik (z.B. [8]) behandelt. Ein *nichtdeterministischer endlicher Automat (NEA)* ist ein Quintupel $A = (\Sigma, Z, \delta, I, F)$, wobei Σ ein Alphabet, Z eine endliche Zustandsmenge, $\delta \subseteq Z \times \Sigma \times Z$ eine Überführungsrelation oder Menge von Transitionen, $I \subseteq Z$ eine Menge von Startzuständen, $F \subseteq Z$ eine Menge von akzeptierenden Zuständen sind.

Die Relation δ wird wie folgt zur Relation $\delta^* \subseteq Z \times \Sigma^* \times Z$ erweitert:

1. $\delta^0 := \{(z, \varepsilon, z) : z \in Z\}$,
2. $\delta^{n+1} := \{(y, wa, z) : w \in \Sigma^n \wedge a \in \Sigma \wedge \exists z'((y, w, z') \in \delta^n \wedge (z', a, z) \in \delta\}$, für $n \geq 0$,
3. $\delta^* := \bigcup_{n=0}^{\infty} \delta^n$.

Die vom NEA $A = (\Sigma, Z, \delta, I, F)$ akzeptierte Sprache ist

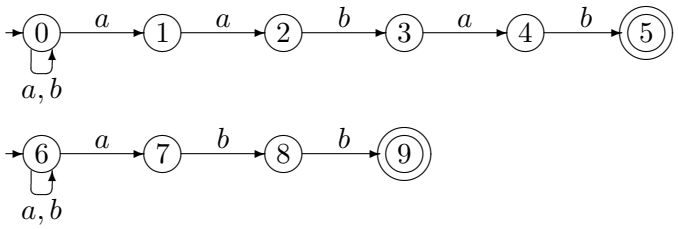
$$L(A) = \{w \in \Sigma^* : \exists z_1 \exists z_2 (z_1 \in I \wedge z_2 \in F \wedge (z_1, w, z_2) \in \delta^*)\}.$$

Ein NEA wird häufig durch seinen Graphen dargestellt. Dabei werden ein Zustand durch einen Knoten, eine Transition durch eine gerichtete und beschriftete Kante, ein Startzustand durch einen Pfeil von außen und ein Endzustand durch einen Doppelkreis gekennzeichnet. Für δ^* sowie $L(A)$ gibt es die folgende anschauliche Interpretation: (y, w, z) ist genau dann in δ^*, wenn es im Graphen des Automaten einen Weg vom Knoten y zum Knoten z mit der Beschriftung w gibt; w ist genau dann in $L(A)$ wenn es im Graphen einen mit w beschrifteten Weg von einem Startzustand zu einem akzeptierenden Zustand gibt.

Beispiel 0.1 Es sei $A = (\{a, b\}, \{0, 1, \ldots, 9\}, \delta, \{0, 6\}, \{5, 9\})$ der NEA mit

$$\begin{aligned}\delta = \{&(0, a, 0), (0, b, 0), (0, a, 1), (1, a, 2), (2, b, 3), (3, a, 4), (4, a, 5),\\ &(6, a, 6), (6, b, 6), (6, a, 7), (7, b, 8), (8, b, 9)\}.\end{aligned}$$

Der Graph von A hat folgendes Aussehen.



Die von A akzeptierte Sprache ist $\{a, b\}^*\{aabab, abb\}$, die Menge aller Wörter, die auf *aabab* oder *abb* enden. □

Oft sieht man die Überführungsrelation eines NEA als eine Funktion $\delta : Z \times \Sigma \to \mathcal{P}(Z)$ an, die zur Funktion $\delta^* : Z \times \Sigma^* \to \mathcal{P}(Z)$ erweitert wird. Dabei sind $\delta(Y, a) = \{z \in Z : (y, a, z) \in \delta \text{ für ein } y \in Y\}$ und $\delta(Y, w) = \{z \in Z : (y, w, z) \in \delta^* \text{ für ein } y \in Y\}$. Entsprechend kann man die akzeptierte Sprache als $L(A) = \{w \in \Sigma^* : \delta^*(I, w) \cap F \neq \emptyset\}$ schreiben.

Spezialfälle des NEA sind der *deterministische endliche Automat (DEA)* sowie der *partielle deterministische endliche Automat (partieller DEA)*. In beiden Fällen besteht die Startmenge I aus einem einzelnen Zustand z_0. Beim DEA ist δ eine Funktion von $\Sigma \times Z$ in Σ, d.h. für jeden Zustand y und jedes Symbol $a \in \Sigma$ existiert *genau ein* Zustand z mit $(y, a, z) \in \delta$. Beim partiellen DEA ist δ eine partielle Funktion von $\Sigma \times Z$ in Σ, d.h. für jeden Zustand y und jedes Symbol $a \in \Sigma$ existiert *höchstens ein* Zustand z mit $(y, a, z) \in \delta$. Für einen DEA besteht die Menge $\delta^*(z_0, w)$ für jedes Wort w aus genau einem Zustand; für einen partiellen DEA ist die Menge $\delta^*(z_0, w)$ für jedes Wort w einelementig oder leer.

Eine Erweiterung des NEA ist der *nichtdeterministische endliche Automat mit ε-Transitionen (ε-NEA)*, bei dem $\delta \subseteq Z \times (\Sigma \cup \{\varepsilon\}) \times Z$ gilt, also zusätzlich sogenannte ε-Transitionen der Form (y, ε, z) enthalten kann. Auf eine formale Definition von δ^* sowie $L(A)$ verzichten wir hier. Die graphische Interpretation ist analog zum NEA: Ein Wort w wird genau dann akzeptiert wenn es einen mit w beschrifteten Weg von der Menge I zur Menge F gibt.

Es ist bekannt, dass es zu jedem ε-NEA einen äquivalenten DEA gibt, der die gleiche Sprache akzeptiert. Bei der Konstruktion des äquivalenten DEA aus einem ε-NEA kann sich allerdings die Anzahl der Zustände exponentiell erhöhen.

Kapitel 1

Exakte Suche nach einem Wort

Wir betrachten die folgende grundlegende **Aufgabenstellung**: Gegeben sind ein Suchwort (*pattern*) P und ein Text T über einem Alphabet Σ mit $|P| = m$, $|T| = n$, $|\Sigma| = \sigma$. Gesucht sind alle Vorkommen von P in T.

Für die Lösung dieses grundlegenden Problems wurden zahlreiche Algorithmen entwickelt. Die meisten von ihnen nutzen die Idee der *Suchfenster*, d.h. man betrachtet jeweils einen Textausschnitt (Fenster) der Länge m und stellt durch Vergleiche von Zeichen fest, ob in diesem Fenster der Text mit dem Suchwort übereinstimmt. Anschließend wird das Suchfenster nach rechts verschoben. Die Länge der Verschiebung ist abhängig von den stattgefundenen Vergleichen und wird so gewählt, dass kein Vorkommen des Suchwortes übergangen wird. Die Algorithmen unterscheiden sich im wesentlichen dadurch, ob sie im Suchfenster mit den Vergleichen von links oder von rechts beginnen. Von links beginnende Algorithmen sind der naive (*brute force*) Algorithmus (der nach dem Abschluss einer Suchphase einfach um 1 verschiebt), die Suche mit *deterministischen endlichen Automaten* (eine Variante davon ist der bekannte *Knuth-Morris-Pratt-Algorithmus*) und der neuere *Shift-And-Algorithmus*, der einen *nichtdeterministischen endlichen Automaten* effizient mit Hilfe von Bit-Arithmetik implementiert. Zu den von rechts beginnenden Algorithmen zählen die Algorithmen von *Boyer-Moore* und *Horspool* sowie die neueren *Faktor-Algorithmen*. Wir untersuchen die genannten Algorithmen bezüglich ihrer Laufzeiten im schlechtesten sowie im mittleren Fall. Ein optimales Ergebnis für die Laufzeit im schlechtesten Fall liefert der Knuth-Morris-Pratt-Algorithmus, während die von rechts beginnenden Algorithmen im mittleren Fall sehr schnell sind.

Danach betrachten wir zwei weitere Algorithmen mit alternativen Lösungsansätzen: den Algorithmus von *Vishkin*, der sehr gut parallelisiert werden kann, sowie den *Karp-Rabin-Algorithmus*, der Hashing-Methoden verwendet. Das Kapitel wird abgeschlossen mit Betrachtungen zu unteren Schranken für Algorithmen zur exakten Suche nach einem Wort. Insbesondere wird der Beweis von Yao der unteren Schranke von $\Omega(\frac{n \log m}{m})$ für die mittlere Laufzeit skizziert.

1.1 Naiver Algorithmus

Man testet für jede Position von T, ob an ihr ein Vorkommen von P beginnt. Dazu werden von links nach rechts die Zeichen von T mit den entsprechenden Zeichen von P verglichen. Tritt ein Mismatch auf, so liegt an der aktuellen Position kein Vorkommen von P vor und der Test wird abgebrochen. Stellt man dagegen für alle m Zeichen Übereinstimmung fest,

so wurde ein Vorkommen gefunden. In beiden Fällen verschiebt man P um eine Stelle und beginnt den Test an der nächsten Position.

Algorithmus 1.1 Naiver Algorithmus zur Wortsuche

Eingabe: Wörter P, T mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

```
(1)   S ← ∅;
(2)   for k ← 1 to n − m + 1
(3)     i ← 1; j ← k;
(4)     while i ≤ m and P[i] = T[j]
(5)       i ← i + 1; j ← j + 1;
(6)     if i = m + 1 then S ← S ∪ {k};
(7)   return S;
```

Die Korrektheit des naiven Algorithmus ist evident. Die Zahl der erforderlichen Vergleiche kann durch $m \cdot (n - m + 1) = O(m \cdot n)$ abgeschätzt werden. Diese Schranke wird für $P = a^m$, $T = a^n$ auch erreicht.

Im durchschnittlichen Fall ist das Verhalten des naiven Algorithmus sehr viel besser. Wir bestimmen dazu $Comp(m)$, die mittlere Zahl der Vergleiche bis zum ersten Mismatch für zwei Wörter der Länge m. Für $0 \leq i \leq m$ sei p_i die Wahrscheinlichkeit, dass genau die ersten i Zeichen übereinstimmen. Dann gilt

$$Comp(m) = \sum_{i=0}^{m-1} p_i \cdot (i+1) + p_m \cdot m.$$

Für jeden Vergleich ist die Wahrscheinlichkeit eines Matches $\frac{1}{\sigma}$ und die eines Mismatches $1 - \frac{1}{\sigma}$. Damit ergibt sich

$$p_i = \frac{1}{\sigma^i} \cdot \left(1 - \frac{1}{\sigma}\right) \text{ für } 0 \leq i \leq m-1,\ p_m = \frac{1}{\sigma^m}$$

und folglich

$$\begin{aligned}
Comp(m) &= \sum_{i=0}^{m-1} \frac{1}{\sigma^i} \cdot \left(1 - \frac{1}{\sigma}\right) \cdot (i+1) + \frac{1}{\sigma^m} \cdot m \\
&= \sum_{i=0}^{m-1} \frac{1}{\sigma^i} \cdot (i+1) - \sum_{i=0}^{m-1} \frac{1}{\sigma^{i+1}} \cdot (i+1) + \frac{1}{\sigma^m} \cdot m \\
&= \sum_{i=0}^{m-1} \frac{1}{\sigma^i} \cdot (i+1) - \sum_{i=1}^{m} \frac{1}{\sigma^i} \cdot i + \frac{1}{\sigma^m} \cdot m \\
&= \sum_{i=0}^{m-1} \frac{1}{\sigma^i} - \frac{1}{\sigma^m} \cdot m + \frac{1}{\sigma^m} \cdot m \\
&= \frac{1 - \sigma^m}{1 - \frac{1}{\sigma}} \\
&< \frac{\sigma}{\sigma - 1}.
\end{aligned}$$

Die durchschnittliche Zahl der Vergleiche bei der Suche in einem Text der Länge n beträgt $(n-m+1)Comp(m)$, also rund $(n-m+1)\cdot\frac{\sigma}{\sigma-1} \in O(n)$.

Damit ist der naive Algorithmus für Texte in natürlichen Sprachen akzeptabel und wird wegen der einfachen Implementierbarkeit auch oft genutzt. Im Falle längerer Suchwörter sind jedoch der Horspool-Algorithmus bzw. die Faktor-Algorithmen, siehe Abschnitte 1.5 und 1.6, vorzuziehen.

1.2 Ränder und Perioden

Beim naiven Algorithmus wurde die Struktur des Suchwortes nicht betrachtet. Unbefriedigende Laufzeiten kommen dann zustande, wenn das Suchwort periodisch ist. Um bessere Ergebnisse als beim naiven Algorithmus zu erhalten, ist eine Vorverarbeitung (Präprozessing) des Suchwortes nötig. Bevor wir uns den einzelnen Algorithmen zuwenden, wollen wir einige Betrachtungen zu Periodizitäten in Wörtern vornehmen, die bei der Wortsuche genutzt werden können.

Definition 1.1 *Es sei P ein Wort der Länge m. Eine Zahl p mit $1 \le p \le m$ heißt* Periode *von P, wenn $P[i] = P[i+p]$ für alle $1 \le i \le m-p$ gilt. Die Länge der kürzesten Periode von P wird mit $Per(P)$ bezeichnet.*

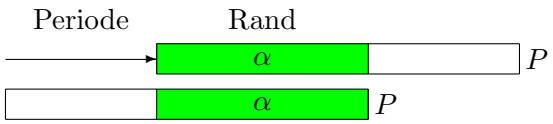
Definition 1.2 *Es sei P ein Wort. Ist α echtes Präfix und echtes Suffix von P, so nennt man α einen* Rand *von P. Die Länge des längsten Randes von P wird mit $Border(P)$ bezeichnet.*

Beispiel 1.1 Das Wort *abcabba* hat die Perioden $6, 7$ und die Ränder a, ε.
Das Wort *abcabcab* hat die Perioden $3, 6, 8$ und die Ränder $abcab, ab, \varepsilon$. □

Lemma 1.1 *Es sei P ein Wort der Länge m. Eine Zahl p, $1 \le p \le m$, ist genau dann eine Periode von P, wenn P einen Rand der Länge $(m-p)$ besitzt.*

Beweis. Die Behauptung des Lemmas folgt unmittelbar aus den Definitionen. □

Die Aussage von Lemma 1.1 wird durch folgende Skizze verdeutlicht.



Bei der Wortsuche bestehen folgende Zusammenhänge zu Perioden und Rändern, die bei der Suche mit endlichen Automaten (Abschnitt 1.3) sowie im Algorithmus von Vishkin (Abschnitt 1.7) ausgenutzt werden.

Lemma 1.2 *Es sei T ein Text, der an der Stelle k ein Vorkommen des Wortes β enthält.*

1. *Das nächste Vorkommen von β in T ist frühestens an der Stelle $k + Per(\beta)$.*
2. *An der Stelle $k + Per(\beta)$ befindet sich ein Vorkommen des längsten Randes von β.*

Beweis. 1. Angenommen, das nächste Vorkommen von β in T befinde sich an der Stelle $k+p$. Gilt $p \geq |\beta|$, so folgt $p \geq Per(\beta)$. Wir können uns also auf den Fall $p < |\beta|$ beschränken. Dann gilt $\beta[i] = T[k+p+i-1] = \beta[i+p]$ für $1 \leq i \leq |\beta| - p$. (Die erste Gleichheit gilt, da β an der Stelle $k+p$ vorkommt; die zweite, da β an der Stelle k vorkommt.) Damit ist p eine Periode von β, und es gilt $p \geq Per(\beta)$.

2. Da β an der Stelle k vorkommt und $Per(\beta)$ eine Periode von β ist, gilt $T[k+Per(\beta)+i-1] = \beta[i+Per(\beta)] = \beta[i]$ für $1 \leq i \leq |\beta| - Per(\beta) = Border(\beta)$. □

Mit Blick auf den naiven Algorithmus können wir folgende Verbesserung ableiten: Besteht im aktuellen Suchfenster eine Übereinstimmung mit dem Präfix β von P, so darf man das Fenster um den Betrag $Per(\beta)$ verschieben und die ersten $Border(\beta)$ Vergleiche auslassen. (Dies ist die Verschiebungsregel von Morris-Pratt, siehe den nächsten Abschnitt.) Von Interesse sind damit also auch die längsten Ränder bzw. die kürzesten Perioden aller Präfixe des Wortes.

Definition 1.3 *Für $1 \leq i \leq |P|$ seien $Border_i(P)$ bzw. $Per_i(P)$ die Länge des längsten Randes bzw. der kürzesten Periode von $P[1 \ldots i]$.*

Beispiel 1.2 Für $P = abcabba$ erhalten wir folgende Werte für $Border_i$ und Per_i.

i	1	2	3	4	5	6	7
$Border_i(P)$	0	0	0	1	2	0	1
$Per_i(P)$	1	2	3	3	3	6	6

□

Ein Algorithmus zur effizienten Ermittlung der Werte $Border_i(P)$ bzw. $Per_i(P)$ ergibt sich aus der folgenden Rekursionsbeziehung.

Lemma 1.3 *Es sei P ein Wort der Länge $m \geq 1$.*

1. *$Border_1(P) = 0$.*
2. *Es seien $1 \leq i < m$, $Border_i(P) = r$, $P[1 \ldots r] = \beta$ und $P[i+1] = a$. Dann gilt*
$$Border_{i+1}(P) = \begin{cases} r+1 & \text{falls } a = P[r+1], \\ Border(\beta a) & \text{sonst.} \end{cases}$$

Beweis. Die erste Aussage folgt unmittelbar aus der Definition des Randes. Für den Beweis der zweiten Aussage stellen wir zunächst fest, dass $Border_{i+1}(P) \leq Border_i(P) + 1$ gilt. Gilt nämlich $Border_{i+1}(P) > 0$, so hat der längste Rand von $P[1 \ldots i+1]$ die Form γa und γ ist dann offenbar ein Rand von $P[1 \ldots i]$. Nun unterscheiden wir die beiden möglichen Fälle.

1. Fall: $a = P[i+1] = P[r+1]$. Dann ist βa offenbar ein Rand von $P[1 \ldots i+1]$ und nach den obigen Bemerkungen auch der längste Rand von $P[1 \ldots i+1]$, d.h. $Border_{i+1}(P) = r+1$.

2. Fall: $a = P[i+1] \neq P[r+1]$. Nun ist βa kein Rand von $P[1 \ldots i+1]$, und es gilt $Border_{i+1}(P) \leq r$. Damit ist der längste Rand von $P[1 \ldots i+1]$ ein echtes Suffix von βa und ein Präfix von β, d.h. ein echtes Präfix von βa und somit ein Rand von βa. Umgekehrt ist auch der längste Rand von βa ein Rand von $P[1 \ldots i+1]$, d.h. $Border_{i+1}(P) = Border(\beta a)$. □

Algorithmus 1.2 Bestimmung der längsten Ränder

Eingabe: Wort P mit $|P| = m$
Ausgabe: Längen $Border_i(P)$ der längsten Ränder der Präfixe von P

```
(1)   Border_1 ← 0;
(2)   for i ← 1 to m − 1
(3)      r ← Border_i;
(4)      while r > 0 and P[r + 1] ≠ P[i + 1]
(5)         r ← Border_r;
(6)      if P[r + 1] = P[i + 1] then Border_{i+1} ← r + 1;
(7)                             else Border_{i+1} ← 0;
(8)   return (Border_1, ..., Border_m);
```

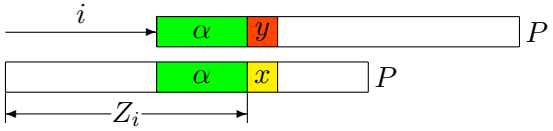
Satz 1.4 *Algorithmus 1.2 bestimmt die Werte $Border_i(P)$, $1 \leq i \leq m$, mit einem Aufwand von $O(m)$.*

Beweis. Die Korrektheit folgt aus der in Lemma 1.3 angegebenen Rekursion. Zu beachten ist dabei, dass für den längsten Rand β von $P[1 \ldots i]$ mit $|\beta| = r$ und für ein Symbol a gilt: $Border_r(\beta a) = Border_r(\beta) = Border_r(P)$.

In Bezug auf die Laufzeit müssen wir abschätzen, wie oft die **while**-Schleife durchlaufen wird. Die Variable r wird mit $Border_1$ also mit 0 initialisiert. Mit jedem Durchlauf durch die **while**-Schleife wird der Wert von r um mindestens 1 verringert. Der Wert von r kann im i-ten Durchlauf der **for**-Schleife höchstens einmal um 1 erhöht werden (falls nämlich am Ende des $(i-1)$-ten Schleifendurchlaufes die Zuweisung "$Border_i \leftarrow r + 1$ " vorgenommen wurde). Da der Wert von r niemals unter 0 sinkt, kann die **while**-Schleife folglich höchstens $(m-1)$-mal durchlaufen werden. □

In einem engen Zusammenhang zu den Perioden von P stehen auch die sogenannten Z-Werte.

Definition 1.4 *Es sei P ein Wort der Länge m. Für $1 \leq i \leq m$ sei $Z_i(P)$ die Länge r des längsten Präfixes von P, so dass i eine Periode von $P[1 \ldots r]$, aber nicht Periode von $P[1 \ldots r+1]$ ist.*



Satz 1.5 *Es sei P ein Wort der Länge m. Eine Zahl i, $1 \leq i \leq m$, ist genau dann eine Periode von P, wenn $Z_i(P) = m$ gilt. Ist i keine Periode, so gilt $P[Z_i(P)+1] \neq P[Z_i(P)+1-i]$; das heißt, die Positionen $Z_i(P)+1$ und $Z_i(P)+1-i$ sind* Zeugen *dafür, dass i keine Periode ist.*

Der Beweis des Satzes ergibt sich direkt aus der Definition. Anwendung finden die Z-Werte im Algorithmus von Vishkin (Abschnitt 1.7) sowie bei verschiedenen Beweisen.

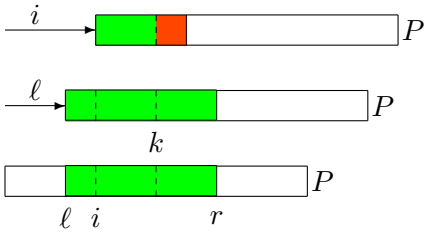
Beispiel 1.3 Für $P = abcabba$ ergibt sich

i	1	2	3	4	5	6	7
Z_i	1	2	5	4	5	7	7

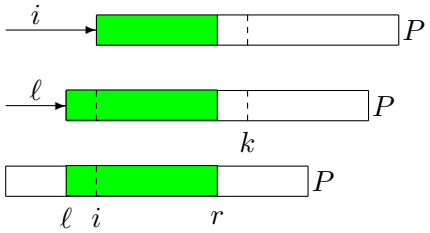
Damit sind 6 und 7 die einzigen Perioden von P. Für $i = 3$ gilt $P[Z_i(P) + 1] = Z[6] = b$ und $P[Z_i(P) + 1 - i] = Z[3] = c$, und damit ist 3 keine Periode von P. □

Schließlich soll noch gezeigt werden, dass man die Z-Werte in Linearzeit bezüglich der Wortlänge berechnen kann, was durch den Z-Algorithmus (Algorithmus 1.3) geschieht. Der Z-Algorithmus berechnet für ein Wort P der Länge m induktiv die Werte $Z_1(P), Z_2(P), \ldots, Z_m(P)$. Den Wert von $Z_1(P)$ erhält man durch explizites Vergleichen der Zeichen $P[j]$ und $P[j-1]$ ab $j = 2$. Im weiteren Verlauf speichert man den bisherigen höchsten Z-Wert in der Variablen r und den dazugehörigen Index in der Variablen ℓ. Da ℓ eine Periode von $P[1 \ldots r]$ ist, gilt insbesondere $P[i+1 \ldots r] = P[i-\ell+1 \ldots r-\ell]$. Für die Bestimmung von $Z_i(P)$ können wir nun den Wert $Z_{i-\ell}(P) =: k$ nutzen, wobei 2 Fälle zu unterscheiden sind.

1. Fall: $Z_{i-\ell} = k < r - \ell$



2. Fall: $Z_{i-\ell} = k \geq r - \ell$



Im 1. Fall folgt nach Definition von $Z_{i-\ell}$:
$P[i+1 \ldots k+\ell] = P[i-\ell+1 \ldots k] = P[1 \ldots k-i+\ell]$ und $P[k+\ell+1] = P[k+1] \neq P[k+1-i+\ell]$, d.h. $Z_i(P) = k + \ell = Z_{i-\ell}(P) + \ell$.

Im 2. Fall folgt nach Definition von $Z_{i-\ell}$:
$P[i+1 \ldots r] = P[i-\ell+1 \ldots r-\ell] = P[1 \ldots r-i]$, d.h. $Z_i(P) \geq r$. Den Wert $Z_i(P)$ erhält man nun durch explizite Vergleiche von $P[j]$ und $P[j-i]$ für $j > r$.

Algorithmus 1.3 Z-Algorithmus

Eingabe: Wort P, $|P| = m$
Ausgabe: $Z_i(P)$, $1 \leq i \leq m$

```
(1)   ℓ ← 1; r ← 1;
(2)   for i ← 1 to m − 1
(3)     if i < r and Z_{i−ℓ} < r − ℓ then Z_i ← Z_{i−ℓ} + ℓ;
(4)     else
(5)       if r < i then r ← i;
(6)       while r < m and P[r + 1] = P[r + 1 − i]
(7)         r ← r + 1;
(8)       Z_i ← r; ℓ ← i;
(9)   Z_m ← m;
(10)  return (Z_1, ..., Z_m);
```

Satz 1.6 *Der Z-Algorithmus berechnet für ein Wort P der Länge m die Werte $Z_i(P)$, $1 \leq i \leq m$, mit einem Aufwand von $O(m)$.*

Beweis. Die Korrektheit folgt aus den obigen Betrachtungen. Für den Beweis der Linearität der Laufzeit müssen wir die Anzahl der expliziten Vergleiche in der Bedingung der **while**-Schleife zählen. Mit einem positiven Vergleich wird der Wert von r um 1 erhöht. Da der Wert von r niemals verringert wird und nicht m überschreitet, gibt es insgesamt höchstens m positive Vergleiche. Nach einem negativen Vergleich wird die **while**-Schleife verlassen; es gibt also für jeden Wert von i höchstens einen negativen Vergleich, insgesamt höchstens $m-1$. □

1.3 Suche mit deterministischen endlichen Automaten

Um die Vorkommen von P zu finden, konstruiert man im Präprozessing den minimalen deterministischen endlichen Automaten (DEA) A_P, der die Sprache Σ^*P akzeptiert. Wie im Satz 1.7 gezeigt wird, spielt bei der Konstruktion des DEA die Tabelle der Ränder $Border_i(P)$ eine entscheidende Rolle. Die Suche erfolgt, indem man dem DEA A_P den Text T als Eingabe gibt. Erreicht A_P einen Endzustand, so wurde ein Vorkommen von P gefunden. Die Suchphase erfolgt in linearer Zeit, wobei alle Textzeichen in ihrer natürlichen Reihenfolge genau einmal betrachtet werden. Es handelt sich damit um einen sogenannten *Realzeit*-Algorithmus.

Ein wesentlicher Nachteil der expliziten Konstruktion des DEA sind die Laufzeit des Präprozessings sowie der Speicherplatz für den DEA jeweils in der Größenordnung $\Theta(|\Sigma|\cdot|P|)$. Dieser Nachteil wird in den Algorithmen von **Morris-Pratt** bzw. **Knuth-Morris-Pratt** behoben, indem im Präprozessing nicht die Überführungstabelle des DEA, sondern nur die Ränder der Präfixe gespeichert werden. Im Algorithmus von **Simon** werden schließlich nur die Transitionen ermittelt und gespeichert, die nicht zum Startzustand von A_P führen. Alle drei erwähnten Varianten benötigen einen zusätzlichen Speicherplatz von $O(|P|)$, unabhängig von der Alphabetgröße. Die Suchphase besitzt jeweils eine lineare Laufzeit, ist aber nicht mehr ein Realzeit-Algorithmus.

Aus theoretischer Sicht sind die in diesem Abschnitt betrachteten Algorithmen vor allem interessant, weil sie das Problem der exakten Suche in linearer Zeit für den schlechtesten Fall lösen. Die *praktische* Bedeutung der DEA-basierten Algorithmen ist eher gering, da sie im Mittel nicht schneller als der naive Algorithmus und wesentlich langsamer als der Horspool-Algorithmus sind.

Satz 1.7 *Es sei $P \in \Sigma^*$ mit $|P| = m$. Die Sprache Σ^*P wird akzeptiert durch den DEA*

$$A_P = (\Sigma, \{0,1,\ldots,m\}, \delta, 0, \{m\}) \text{ mit}$$

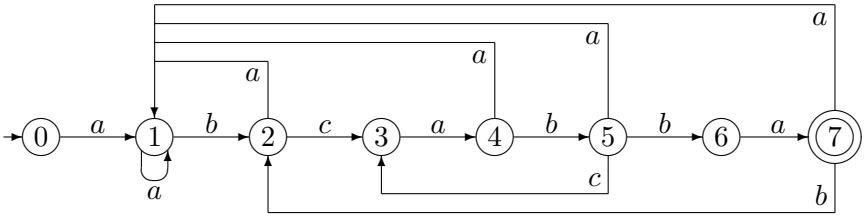
$$\delta(i,x) = \begin{cases} i+1 & \text{falls } 0 \leq i < m, x = P[i+1], \\ Border(P[1\ldots i]x) & \text{sonst.} \end{cases}$$

Beweis. Wir zeigen durch vollständige Induktion über $|T|$: Für jedes Wort T ist $\delta^*(0,T)$ die Länge des längsten Suffixes von T, das Präfix von P ist.

Die Induktionsbehauptung ist offenbar korrekt für $|T| = 0$. Gelte nun die Behauptung für einen Text T mit $|T| = n$ und sei $\delta^*(0,T) = i$. Dann ist $P[1\ldots i]$ das längste Suffix von T, das ein Präfix von P ist. Für $x \in \Sigma$ gilt $\delta^*(0,Tx) = \delta(i,x)$.
Ist $x = P[i+1]$, so gilt $\delta(i,x) = i+1$ und $P[1\ldots i+1]$ ist das längste Suffix von Tx, das

Präfix von P ist; d.h., die Induktionsbehauptung ist erfüllt.
Ist $x \neq P[i+1]$, so gilt $\delta(i,x) = \mathit{Border}(P[1\ldots i]x)$. Sei β das längste Suffix von Tx, das ein Präfix von P ist. Wir zeigen zunächst, dass β höchstens die Länge $\delta(i,x)$ haben kann. Im Falle von $\beta = \varepsilon$ ist diese Behauptung erfüllt. Anderenfalls gilt $\beta = \gamma x$, und γ ist ein Suffix von T, das ein Präfix von P ist. Nach Induktionsvoraussetzung gilt $|\gamma| \leq i$, aus $P[i+1] \neq x$ folgt $|\gamma| < i$. Damit ist γ ein Rand von $P[1\ldots i]$ und β ein Rand von $P[1\ldots i]x$, d.h. $|\beta| \leq \delta(i,x)$. Andererseits ist das Präfix $P[1\ldots\delta(i,x)]$ ein Suffix von $P[1\ldots i]x$ und nach Induktionsvoraussetzung auch ein Suffix von Tx. Damit folgt $|\beta| \geq \delta(i,x)$, d.h. $|\beta| = \delta(i,x)$. □

Beispiel 1.4 Für $P = abcabba$ ergibt sich der folgende DEA (Kanten zum Zustand 0 wurden weggelassen):



□

Satz 1.8 *Es sei $|\Sigma| = \sigma$ und $P \in \Sigma^*$ ein Wort der Länge m. Der Automat A_P kann mit einem Aufwand von $O(m \cdot \sigma)$ konstruiert werden.*

Beweis. Für die in der Überführungsfunktion benötigten Werte $\mathit{Border}(P[1\ldots i]x)$ gilt folgende Rekursion:

$$\mathit{Border}(P[1\ldots i]x) = \begin{cases} 0 & \text{falls } i = 0, \\ \delta(\mathit{Border}_i(P), x) & \text{falls } 0 < i \leq m. \end{cases}$$

Die Werte Border_i sind nach Satz 1.4 mit einem Aufwand von $O(m)$ berechenbar. Damit kann man für jeden Zustand i die Werte der Überführungsfunktion mit einem Aufwand von $O(\sigma)$ bestimmen. □

Algorithmus 1.4 DEA-Algorithmus zur Wortsuche

Eingabe: Wörter P, T über Σ mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

```
(1)  Konstruiere den DEA A_P = (Σ, {0,1,...,m}, δ, 0, {m});
(2)  S ← ∅; i ← 0;
(3)  for j ← 1 to n
(4)      i ← δ(i, T[j]);
(5)      if i = m then S ← S ∪ {j − m + 1};
(6)  return S;
```

Satz 1.9 *Algorithmus 1.4 findet alle Vorkommen von P in T mit einer Laufzeit von $O(n)$ (ohne Konstruktion von A_P).*

Beweis. Die Korrektheit folgt aus Satz 1.7. Die Laufzeitabschätzung ist trivial. □

Die Idee des *Suchfensters* wurde bisher nicht erwähnt; sie steckt aber gleichwohl implizit hinter dem DEA-Algorithmus. Eine Zustandsänderung von i nach $i+1$ bedeutet einen weiteren Vergleich im aktuellen Suchfenster. Ein Wechsel vom Zustand i zum Zustand $j \leq i$ bedeutet eine Verschiebung des Suchfensters um $i - j + 1$, wobei die Übereinstimmung der ersten j Zeichen garantiert ist.

Der Algorithmus von Morris-Pratt

Die Idee des Algorithmus von Morris und Pratt (**MP-Algorithmus**) ist, anstatt der Überführungsfunktion δ des DEA A_P die Werte $Border_i(P)$ zu speichern und in der Suchphase den Nachfolgezustand mittels der rekursiven Definition von δ aus Satz 1.7 zu berechnen.

Beispiel 1.5 Das Wort $P = abcabba$ hat folgende Werte für $Border_i$.

i	1	2	3	4	5	6	7
$Border_i$	0	0	0	1	2	0	1

Damit ergibt sich gemäß der rekursiven Definition von δ:

$$\begin{aligned}\delta(5,a) &= \delta(Border_5, a) = \delta(2,a) \text{ (wegen } P[6] \neq a) \\ &= \delta(Border_2, a) = \delta(0,a) \text{ (wegen } P[3] \neq a) \\ &= 1 \text{ (wegen } P[1] = a).\end{aligned}$$

□

Algorithmus 1.5 Morris-Pratt-Algorithmus

Eingabe: Wörter P, T über Σ mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

```
(1)  Bestimme die Werte Border_i(P);
(2)  S ← ∅; i ← 0;
(3)  for j ← 1 to n
(4)      while i ≠ 0 and P[i+1] ≠ T[j]
(5)          i ← Border_i(P);
(6)      if P[i+1] = T[j] then i ← i+1;
(7)      if i = m then S ← S ∪ {j−m+1};
(8)  return S;
```

Satz 1.10 *Der Morris-Pratt-Algorithmus findet alle Vorkommen von P in T mit einer Laufzeit von $O(n)$.*

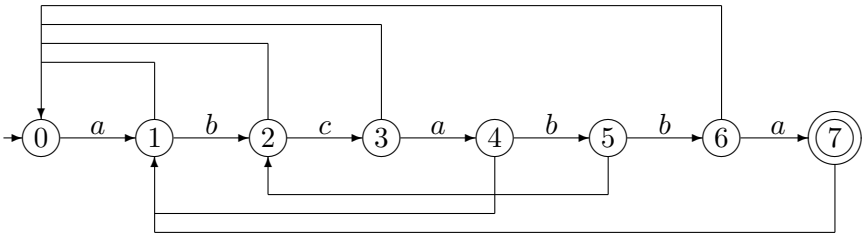
Beweis. Zum Beweis der Korrektheit stellen wir nur fest, dass in den Zeilen 4 bis 6 der Variablen i der Wert $\delta(i, T[j])$ zugewiesen wird, wobei δ die Überführungsfunktion des DEA A_P ist. Für die Abschätzung der Laufzeit zeigen wir ähnlich wie im Beweis von Satz 1.4, dass die Anzahl der Durchläufe der **while**-Schleife durch n beschränkt ist. □

Graphische Interpretation

Der Morris-Pratt-Algorithmus kann wie folgt graphisch interpretiert werden. Für ein Wort P der Länge m konstruieren wir einen Graphen mit den Knoten (Zuständen) $0, 1, \ldots, m$. Für $1 \leq i \leq m$ gibt es die beschriftete *Vorwärtskante* $(i-1, P[i], i)$ und die unbeschriftete *Rückwärtskante* von i nach $Border_i(P)$ (auch *failure link* genannt).

Ein Zustandsübergang geschieht wie folgt. Ist die Vorwärtskante aus dem aktuellen Zustand mit dem aktuellen Textzeichen beschriftet, so folgt man dieser Kante und ist fertig. Anderenfalls folgt man der Rückwärtskante und wiederholt die Prozedur, bis man der Vorwärtskante folgen kann oder der Knoten 0 erreicht ist. Erreicht man den Knoten m, so wurde ein Vorkommen gefunden.

Beispiel 1.6 Für $P = abcabba$ ergibt sich der folgende Graph:



□

Knuth-Morris-Pratt-Algorithmus

Der Morris-Pratt-Algorithmus wird durch folgende Beobachtung zum Knuth-Morris-Pratt-Algorithmus (KMP-Algorithmus) verfeinert: Gilt $P[Border_i(P)+1] = P[i+1] = a$, so kann $\delta(i, x)$ für $x \neq a$ *nicht* $Border_i(P)+1$ sein, da x auch nicht mit dem Zeichen an der Position $Border_i(P)+1$ von P übereinstimmt. Der MP-Algorithmus führt also unter Umständen Vergleiche aus, die nicht nötig sind. Beispielsweise gilt für das Wort $P = abcabba$: $P[2] = P[5] = b$, und damit kann $\delta(4, x)$ für $x \neq b$ nicht $Border_4(P)+1 = 2$ sein.

Eine genauere Betrachtung liefert folgendes Resultat: Bei der Definition der Überführungsfunktion δ des DEA A_P darf man $Border_i(P)$ durch den wie folgt definierten Wert $SBorder_i(P)$ ersetzen.

Definition 1.5 *Für $1 \leq i \leq |P|$ sei $SBorder_i(P)$ die Länge r des längsten Randes von $P[1 \ldots i]$ mit $P[r+1] \neq P[i+1]$ oder $r = 0$.*

Satz 1.11 *Für ein Wort P der Länge m, $1 \leq i \leq m$ und $r = Border_i(P)$ gilt:*

$$SBorder_i(P) = \begin{cases} r & \text{falls } r = 0 \text{ oder } P[i+1] \neq P[r+1], \\ SBorder_r(P) & \text{sonst.} \end{cases}$$

Auf den Beweis wird hier verzichtet. Die Werte $SBorder_i$ sind damit ebenfalls in linearer Zeit berechenbar. Den KMP-Algorithmus erhält man einfach, indem man im MP-Algorithmus $Border_i$ durch $SBorder_i$ ersetzt.

Beispiel 1.7 Das Wort $P = abcabba$ hat folgende Werte für $Border_i(P)$ sowie $SBorder_i(P)$.

i	1	2	3	4	5	6	7
$Border_i$	0	0	0	1	2	0	1
$SBorder_i$	0	0	0	0	2	0	1

Damit ergibt sich gemäß der rekursiven Definition von δ mittels *Border* bzw. *SBorder*:

$$\begin{aligned}
\delta(4,c) &= \delta(Border_4, c) = \delta(1,c) \text{ (wegen } P[5] \neq c) \\
&= \delta(Border_1, c) = \delta(0,c) \text{ (wegen } P[2] \neq c) \\
&= 0 \text{ (wegen } P[1] \neq c) \\
&\quad \text{bzw.} \\
\delta(4,c) &= \delta(SBorder_4, c) = \delta(0,c) \text{ (wegen } P[5] \neq c) \\
&= 0 \text{ (wegen } P[1] \neq c).
\end{aligned}$$

□

Allgemein kann man zeigen, dass der KMP-Algorithmus für jede Eingabe höchstens so viele Vergleiche wie der MP-Algorithmus benötigt, im schlechtesten Falle ($P = ab$, $T = a^n$) allerdings ebenfalls $2n - 2$.

Simon-Algorithmus

Eine genauere Betrachtung des Automaten A_P zeigt, dass höchstens $2m$ Kanten im Graphen des Automaten nicht zum Zustand 0 führen. Nur diese Kanten muss man explizit speichern. Auf diese Weise ist es möglich, den Automaten A_P mit einem Platzbedarf und in einer Zeit von $O(m)$ zu konstruieren und zu speichern.

Im folgenden bezeichnen wir eine Kante der Form $(k, x, k+1)$ als *Vorwärtskante*, eine Kante der Form $(k, x, j+1)$ mit $0 \leq j < k$ als *Rückwärtskante* und eine Kante der Form $(k, x, 0)$ als *triviale Kante*.

Lemma 1.12 *Es sei $P \in \Sigma^*$ ein Wort der Länge m. Der Graph des Automaten A_P enthält höchstens m Rückwärtskanten.*

Beweis. Ist $(k, x, j+1)$ eine Rückwärtskante, so ist $P[1 \ldots j]$ ein Rand von $P[1 \ldots k]$, d.h. $(k - j)$ ist eine Periode von $P[1 \ldots k]$. Weiterhin gilt $P[j+1] = x$ sowie $P[k+1] \neq x$ oder $k = m$. Das heißt, für $i = k - j$ erhalten wir $Z_i(P) = k$ und $j = k - i$. Damit hat jede Rückwärtskante die Form $(Z_i(P), P[Z_i(P) - i + 1], Z_i(P) - i + 1)$, $1 \leq i \leq m$, und es kann höchstens m Rückwärtskanten geben. □

Bei der Konstruktion von A_P kann man nun die trivialen Kanten weglassen und die Rückwärtskanten für jeden Zustand in einer Liste speichern. Dies ist in der Abbildung zu Beispiel 1.4 bereits geschehen. Die Zeit für die Konstruktion von A_P ohne triviale Kanten beträgt $O(n)$. Für die Bestimmung des Nachfolgezustands überprüft man zuerst, ob die Vorwärtskante mit dem aktuellen Textsymbol beschriftet ist. Danach durchsucht man die Liste der Rückwärtskanten. Sollte keine dieser Kanten mit dem aktuellen Textsymbol beschriftet sein, ist der Nachfolgezustand 0.

Algorithmus 1.6 Simon-Algorithmus

Eingabe: Wörter P, T über Σ mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

```
(1)   Konstruiere den DEA A_P = (Σ, {0,1,...,m}, δ, 0, {m}) ohne triviale Kanten;
(2)   S ← ∅; i ← 0;
(3)   for j ← 1 to n
(4)      if P[i+1] = T[j] then i ← i+1;
(5)      else
(6)         z ← 0;
(7)         foreach Rückwärtskante (i, x, i')
(8)            if x = T[j] then z ← i'; break;
(9)         i ← z;
(10)     if i = m then S ← S ∪ {j−m+1};
(11)  return S;
```

Die Anzahl der Schritte zur Ermittlung des Nachfolgezustandes beträgt somit maximal $1+r$, wobei r die Anzahl der Rückwärtskanten aus dem aktuellen Zustand ist. Wir wollen jetzt zeigen, dass trotz dieser Verzögerung die Gesamtlaufzeit der Suchphase linear (unabhängig von σ) ist.

Lemma 1.13 *Es sei $P \in \Sigma^*$ ein Wort der Länge m. Gehen von einem Knoten k im Graphen von A_P r Rückwärtskanten aus, so gilt für jede Rückwärtskante $(k, x, j+1)$: $k - j \geq r$.*

Beweis. Wie bereits erwähnt, besitzt $P[1 \ldots k]$ einen Rand der Länge j, d.h. $k - j$ ist eine Periode von $P[1 \ldots k]$. Dann muss aber jedes Teilwort von $P[1 \ldots k]$ der Länge $k - j$ alle in $P[1 \ldots k]$ vorkommenden Symbole enthalten. Da $P[1 \ldots k]$ mindestens r verschiedene Symbole enthält, folgt $k - j \geq r$. □

Satz 1.14 *Es seien $P, T \in \Sigma^*$ mit $|P| = m$, $|T| = n$. Der Simon-Algorithmus findet die Vorkommen von P in T in $O(n)$ Schritten.*

Beweis. Wir bestimmen die gesamte Anzahl der Durchläufe durch die **foreach**-Schleife. Die Variable i wird mit 0 initialisiert und höchstens n-mal um 1 erhöht. Gehen von einem Knoten i r Rückwärtskanten aus, so wird die **foreach**-Schleife höchstens r-mal durchlaufen und i um mindestens $(r-1)$ verringert. Da der Wert von i niemals kleiner als 0 wird, ist die Anzahl der Durchläufe durch die **foreach**-Schleife durch n beschränkt. □

Vergleich der DEA-Algorithmen

Alle 4 Varianten der Suche mit deterministischen endlichen Automaten haben eine Laufzeit von $O(n)$ für die Suchphase. Die Suche mit dem explizit angegebenen DEA ist der einzige Realzeit-Algorithmus, hat aber auch einen zusätzlichen Platzbedarf von $O(\sigma m)$. Die anderen Algorithmen benötigen nur einen zusätzlichen Platz von $O(m)$ und unterscheiden sich vor allem in der maximalen Anzahl der Schritte zur Berechnung des nächsten Zustandes (Verzögerung, *delay*). In der folgenden Tabelle werden die Algorithmen bezüglich ihrer Verzögerung für ein Suchwort der Länge m verglichen.

Algorithmus	max. Verzögerung
DEA-Algorithmus	1 (Realzeit)
MP-Algorithmus	m
KMP-Algorithmus	$\log_\Phi(m)$ mit $\Phi = \frac{1+\sqrt{5}}{2}$
Simon-Algorithmus	$1 + \log_2 \sigma$ (Rückwärtskanten als geordnete Listen)

1.4 Der Shift-And-Algorithmus

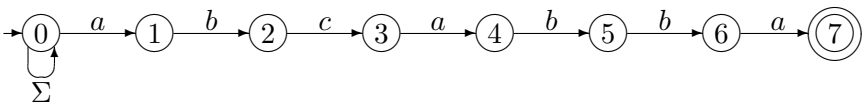
Der Shift-And-Algorithmus ist ein relativ neuer Algorithmus, der die durch Bit-Arithmetik mögliche Parallelisierung ausnutzt. Er ist effizient für die Suche nach P in T, wenn die Länge von P nicht größer ist als die Zahl der Bits in einem Computerwort (zur Zeit also 32 bzw. 64).

Es besteht ein enger Zusammenhang zum nichtdeterministischen Automaten (NEA), der die Sprache $\Sigma^* P$ akzeptiert. Dieser NEA hat für $|P| = m$ die Form

$$NEA_P = (\Sigma, \{0, 1, \ldots, m\}, \delta, 0, \{m\}) \text{ mit}$$

$$\delta = \{(i, P[i+1], i+1) : 0 \leq i < m\} \cup \{(0, x, 0) : x \in \Sigma\}.$$

Beispiel 1.8 Für $P = abcabba$ ergibt sich NEA_P wie folgt:



□

Die Grundidee des Shift-And-Algorithmus ist, für jedes Präfix $T[1 \ldots j]$ die Menge der erreichbaren Zustände zu berechnen. Diese Menge wird durch einen Bitvektor dargestellt. Durch Ausnutzen der Bit-Parallelität kann ein Schritt des NEA für hinreichend kurze Suchwörter in konstanter Zeit erfolgen.

Bitvektoren

Ein Bitvektor der Länge m ist ein Wort der Länge m über $\{0, 1\}$. Wir benutzen bei der Notation von Bitvektoren die Konvention, dass bei bekannter Länge m führende Nullen nicht aufgeschrieben werden. Insbesondere verwenden wir die Schreibweisen 0 statt 0^m und 1 statt $0^{m-1}1$. Außerdem werden die Bits in der Regel *von rechts nach links* nummeriert. Zur Manipulation von Bitvektoren benutzen wir die Operationen & (bitweises AND), | (bitweises OR), ˆ (bitweises XOR), ∼(bitweise Negation), << (Verschiebung (*Shift*) der Bits nach links), >> (Verschiebung der Bits nach rechts). Formal: Sind $A = a_m \cdots a_2 a_1$ und $B = b_m \cdots b_2 b_1$

Bitvektoren der Länge m und ist $k < m$ eine natürliche Zahl, so definieren wir:

$$\begin{aligned} A \,\&\, B &= (a_m \,\&\, b_m) \cdots (a_2 \,\&\, b_2)(a_1 \,\&\, b_1), \\ A \,|\, B &= (a_m \,|\, b_m) \cdots (a_2 \,|\, b_2)(a_1 \,|\, b_1), \\ A \hat{} B &= (a_m \hat{} b_m) \cdots (a_2 \hat{} b_2)(a_1 \hat{} b_1), \\ \sim A &= \sim a_m \cdots \sim a_2 \sim a_1, \\ A << k &= a_{m-k} \cdots a_2 a_1 0^k, \\ A >> k &= 0^k a_m \cdots a_{k+2} a_{k+1}. \end{aligned}$$

Für $k \geq m$ ergibt sich $A << k = A >> k = 0^m$. Die Bit-Operationen $\&$, $|$, $\hat{}$ bzw. $\sim$ sind auf $\{0,1\}^2$ bzw. auf $\{0,1\}$ wie folgt definiert:

a	b	$a \,\&\, b$	$a \,\|\, b$	$a \hat{} b$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

a	$\sim a$
0	1
1	0

Bitvektoren werden häufig genutzt, um Teilmengen einer endlichen Menge $M = \{1, 2, \ldots, m\}$ darzustellen. So entspricht einer Teilmenge $M' \subseteq M$ der Bitvektor $b_m \cdots b_2 b_1$ mit $b_i = 1$ genau dann, wenn $i \in M'$. Insbesondere sind die mengentheoretischen Operationen einfach zu realisieren:

- Einermenge $\{i\}$ erzeugen: $1 << (i-1)$.
- Komplement der Menge A: $\sim A$.
- Vereinigung der Mengen A und B: $A \,|\, B$.
- Durchschnitt der Mengen A und B: $A \,\&\, B$.
- Test, ob $i \in A$: $A \,\&\, (1 << (i-1)) \neq 0$.

In der Programmiersprache **Java** kann man einen Bitvektor der Länge 32 durch eine Variable des primitiven Datentyps **int** realisieren. Die genannten Operationen sind in dieser Sprache für den Typ **int** definiert, beanspruchen einen konstanten Aufwand und werden sehr schnell ausgeführt. Für die Realisierung längerer Bitvektoren benötigt man mehrere **int**-Variablen und entsprechend mehr Schritte für die Ausführung der Bitvektor-Operationen, die durch notwendige Überträge zusätzlich kompliziert werden.

Der Algorithmus

Im Shift-And-Algorithmus wird die Menge der erreichbaren Zustände des nichtdeterministischen endlichen Automaten $NEA_P = (\Sigma, \{0, 1, \ldots, m\}, \delta, 0, \{m\})$ durch einen Bitvektor $Z = z_m \cdots z_2 z_1$ der Länge m kodiert. Da der Zustand 0 immer erreichbar ist, wird er nicht kodiert. Dabei entspricht dem Zustand $i \in \{1, 2, \ldots, m\}$ das Bit z_i. Nach dem Einlesen des Textes $T[1 \ldots j]$ hat das Bit z_i den Wert 1 genau dann, wenn der Zustand i mit dem Wort

$T[1 \ldots j]$ erreichbar ist. Ein Vorkommen von P endet an der Stelle j genau dann, wenn das Bit z_m den Wert 1 besitzt.

Außer dem Bitvektor Z gibt es für jeden Buchstaben $x \in \Sigma$ einen Bitvektor $B[x]$ der Länge m. In $B[x]$ ist das i-te Bit von rechts genau dann 1, wenn $P[i] = x$ gilt. Die Vektoren $B[x]$ werden im Präprozessing zunächst alle mit 0 initialisiert. Gilt $P[i] = x$, so wird das i-te Bit von $B[x]$ mittels der Zuweisung

$$B[x] \leftarrow B[x] \,|\, (1 \ll (i-1))$$

auf 1 gesetzt.

Bei der Initialisierung für die Suche erhält Z den Wert 0. Für die Berechnung des aktualisierten Wertes von Z stellen wir fest, dass der Zustand i genau dann mit dem Zeichen x erreichbar ist, wenn aktuell der Zustand $(i-1)$ erreichbar ist und $P[i] = x$ gilt. Die Aktualisierung für Z und das Textzeichen x erfolgt in 3 Schritten:

1. $Z \leftarrow Z \ll 1$; Setze den Wert des Bits z_i auf den bisherigen Wert von z_{i-1}, $i \geq 2$.
2. $Z \leftarrow Z \,|\, 1$; Setze den Wert des Bits z_1 auf 1.
3. $Z \leftarrow Z \,\&\, B[x]$; Belasse den Wert 1 für ein Bit z_i genau dann, wenn $P[i] = x$ gilt.

Die Ausnutzung des Bit-Parallelismus erfolgt in den Schritten 1 und 3 durch Anwendung des *Shift*- bzw. des *And*-Operators; dies erklärt den Namen des Algorithmus.

Algorithmus 1.7 Shift-And-Algorithmus

Eingabe: Wörter P, T mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

```
(1)  foreach x ∈ Σ
(2)      B[x] ← 0;
(3)  for i ← 1 to m
(4)      B[P[i]] ← B[P[i]] | (1 << (i − 1));
(5)  S ← ∅; Z ← 0;
(6)  for j ← 1 to n
(7)      Z ← ((Z << 1) | 1) & B[T[j]];
(8)      if (Z & (1 << (m − 1))) ≠ 0 then S ← S ∪ {j − m + 1};
(9)  return S;
```

Beispiel 1.9 Für $\Sigma = \{a, b, c\}$, $P = abcabba$ und $T = abaabcabbab$ ergibt sich folgender Ablauf des Algorithmus (das rechteste Bit von Z ist unten). Das Ende eines Vorkommens erkennt man am 7. Bit von rechts.

$B[a]$	$B[b]$	$B[c]$
0	0	0
1	0	0
0	1	0
0	1	0
1	0	0
0	0	1
0	1	0
1	0	0

	a	b	a	a	b	c	a	b	b	a	b
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0
0	0	1	0	0	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0	1	0

□

Satz 1.15 *Der Shift-And-Algorithmus findet alle Vorkommen von P in T.*

Beweis. Zum Beweis der Korrektheit zeigt man per Induktion, dass in Z das i-te Bit von hinten tatsächlich genau dann 1 ist, wenn der Zustand i in NEA_P erreichbar ist. □

Die Laufzeit beträgt $O(m + \sigma \cdot \lceil \frac{m}{w} \rceil)$ für das Präprozessing und $O(n \cdot \lceil \frac{m}{w} \rceil)$ für die Suche, wobei w die Länge eines Computer-Wortes ist. Für hinreichend kurze Suchwörter P liefert der Shift-And-Algorithmus also einen Realzeit-Algorithmus zur exakten Wortsuche. Da die Operationen auf Bit-Ebene erfolgen, sind Implementierungen auch sehr schnell. Die Hauptbedeutung des Shift-And-Algorithmus ist jedoch, dass die Idee des Bit-Parallelismus ohne weiteres auf komplexere Suchprobleme verallgemeinert werden kann.

Shift-Or-Algorithmus

Der 2. Schritt in der Aktualisierung des Shift-And-Algorithmus (rechtestes Bit auf 1 setzen) ist erforderlich, da durch die *Shift*-Operation von rechts eine 0 nachgeführt wird. Dieser Schritt kann eingespart werden, wenn man mit den Komplementwerten von Z bzw. $B[x]$ rechnet. Man setzt also im Präprozessing $B'[x] \leftarrow\sim B[x]$ und in der Initialisierung $Z' \leftarrow\sim 0$. Als Aktualisierungsschritte bleiben dann:

1. $Z' \leftarrow Z' \texttt{<<} 1$;
2. $Z' \leftarrow Z' \mid B'[x]$;

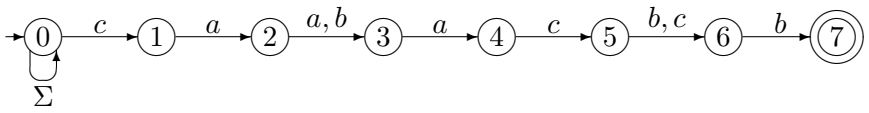
Die Enden der Vorkommen findet man durch Test des m-ten Bits von Z' auf 0.

Erweiterung: Buchstabenklassen.

Am Ende dieses Abschnittes soll der Shift-And-Algorithmus auf einige komplexere Suchprobleme erweitert werden. Zunächst geht es um die Suche nach Mustern mit *Buchstabenklassen*. Das Suchmuster P hat die Form $P = S_1 S_2 \cdots S_m$, wobei die S_i Teilmengen von Σ sind. Anstelle einzelner Buchstaben stehen im Muster also Mengen von Buchstaben. Eine Textstelle gegenüber einer Menge S muß mit einem Buchstaben aus S übereinstimmen.

Für die Suche nach Mustern mit Buchstaben-Klassen ist der Shift-And-Algorithmus sehr einfach zu verallgemeinern. Man braucht nämlich nur die Bytes $B[x]$ für die Zeichen aus Σ anzupassen. Im Byte $B[x]$ wird das i-te Bit von rechts genau dann auf 1 gesetzt, wenn $x \in S_i$ gilt. Der Suchalgorithmus wird angewendet wie bei der Suche nach einem Wort.

Beispiel 1.10 Mit dem Muster $P = ca\{a, b\}ac\{b, c\}b$ stimmen die Wörter *caaacbc*, *caaaccb*, *cabacbb* und *cabaccb* überein. Es ergibt sich der folgende NEA:



Für die einzelnen Buchstaben ergeben sich die folgenden Bitvektoren:

$$B[a] = 0001110,\ B[b] = 1100100,\ B[c] = 0110001.$$

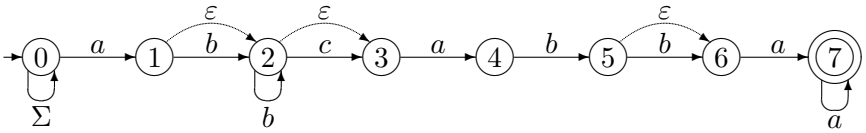
□

Erweiterung: Optionale und wiederholbare Symbole.

Unser Suchmuster P kann jetzt Zeichen aus Σ sowie die Zeichenfolgen $x?$, $x+$ und $x*$ mit $x \in \Sigma, ?, +, * \notin \Sigma$ enthalten. Die Interpretation der speziellen Folgen bedeutet: An der Stelle von $x?$ kann x oder ε stehen; an der Stelle von $x+$ kann ein beliebiges Wort aus $\{x\}^+$ stehen und an der Stelle von $x*$ kann ein beliebiges Wort aus $\{x\}^*$ stehen. Wir nennen $x?$ ein *optionales*, $x+$ ein *wiederholbares* und $x*$ ein *optionales und wiederholbares* Symbol. Wir betrachten hier nur das Problem, das Ende eines Vorkommens von P zu finden. Deshalb können wir o.B.d.A. annehmen, daß das erste Symbol nicht optional ist.

Der NEA für ein solches Suchmuster hat die gleiche Struktur wie der NEA für das Muster ohne Sonderzeichen. Außerdem gibt es für $x?$ und $x*$ eine ε-Kante parallel zur x-Kante sowie für $x+$ und $x*$ eine x-Schleife am Zielknoten.

Beispiel 1.11 Für $P =$ `ab*c?abb?a+` ergibt sich der folgende NEA:



Es gibt u.a. folgende Treffer: *aaba*, *abbbbcabbaaa*, *acabbaa*. □

Die einfache Struktur des NEA ermöglicht die Anwendung von Bit-Arithmetik. Wie beim Shift-And-Algorithmus wird die Menge der erreichbaren Zustände in einem Byte Z gespeichert. Im Präprozessing werden die Bytes $B[x]$ wie beim Shift-And-Algorithmus ermittelt. Außerdem benötigen wir weitere Bytes

- O für die Speicherung optionaler Symbole:
  Das i-te Bit von O ist genau dann 1, wenn das i-te Symbol optional ist.
- R für die Speicherung wiederholbarer Symbole:
  Das i-te Bit von R ist genau dann 1, wenn das i-te Symbol wiederholbar ist.
- I: für die Anfänge von ε-Pfaden:
  Das i-te Bit von I ist genau dann 1, wenn im Zustand i ein maximaler ε-Pfad beginnt.
- F: für die Enden von ε-Pfaden:
  Das i-te Bit von F ist genau dann 1, wenn im Zustand i ein maximaler ε-Pfad endet.

Die Aktualisierung geschieht für ein Textzeichen x in 3 Schritten:

1. $Z \leftarrow (((Z \ll 1)|1) \,\&\, B[x]) \,|\, (Z \,\&\, B[x] \,\&\, R)$
   Damit werden alle Nachfolgezustände ermittelt, die über eine x-Kante erreichbar sind
2. $E \leftarrow Z \,|\, F$
   Dies markiert alle Zustände, die schon erreicht oder das Ende eines ε-Pfades sind.
3. $Z \leftarrow Z \,|\, (O \,\&\, ((\sim(E - I))\hat{}E))$
   Das i-te Bit in $(E-I)$ hat genau dann den gleichen Wert wie das i-te Bit in E, wenn der Zustand i entweder zu keinem ε-Pfad gehört oder zu einem ε-Pfad gehört und irgendein weiter links gelegener Zustand des selben ε-Pfades schon erreicht ist.
   Damit ergibt die Operation $(O \,\&\, ((\sim(E - I))\hat{}E))$ im i-ten Bit genau dann den Wert 1,

wenn der Zustand i zu einem ε-Pfad gehört und irgendein weiter links gelegener Zustand des selben ε-Pfades schon erreicht ist; das heißt, in Z haben am Ende alle Bits den Wert 1, wenn die zugehörigen Zustände erreichbar sind.

Beispiel 1.12 Für unser Beispielwort $P = \texttt{ab*c?abb?a+}$ erhalten wir die Bitvektoren
$B[a] = 01001001$, $B[b] = 00110010$, $B[c] = 00000100$,
$O = 00100110$, $R = 01000010$, $I = 00010001$, $F = 00100100$. □

1.5 Die Algorithmen von Boyer-Moore und Horspool

In den bisher betrachteten Algorithmen wurde innerhalb des Suchfensters von links nach rechts vorgegangen. Ein Effekt war, dass jedes Textzeichen mindestens einmal betrachtet wurde, so dass auch im besten Fall mindestens eine lineare Anzahl von Vergleichen notwendig war. In den Algorithmen dieses und des nächsten Abschnitts wird im Suchfenster *von rechts nach links* verglichen und anschließend das Suchfenster nach verschiedenen Regeln verschoben. Dadurch ist es nicht mehr nötig, jedes Textzeichen zu betrachten. Trifft man z.B. im ersten Vergleich von $P = abcabba$ auf das Zeichen d, so kann im gesamten Suchfenster kein Vorkommen beginnen, da P das Textzeichen d nicht enthält. Man darf also um den Betrag 7 verschieben. Ist das letzte Textzeichen des Fensters ein c, so ist immerhin noch eine Verschiebung um den Betrag 4 möglich. Diese *Bad Character Regel* ist die Grundlage der beiden in diesem Abschnitt betrachteten Algorithmen.

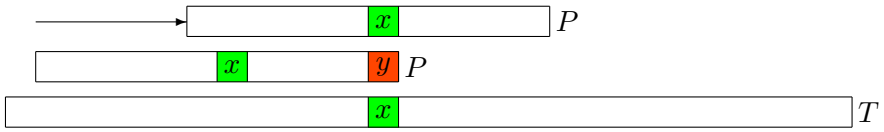
Der klassische Boyer-Moore-Algorithmus benutzt eine weitere Heuristik, die sogenannte *Good Suffix Regel*. Mit dieser Regel lassen sich weitere Verschiebungen erzielen, wenn mehrere Vergleiche positiv ausgehen, also ein Suffix des Suchwortes mit dem Text im Fenster übereinstimmt. Diese Regel ist für die mittlere Laufzeit allerdings relativ bedeutungslos, da es nur selten zu einer Übereinstimmung mit einem Suffix kommt. Außerdem erfordert sie ein relativ aufwendiges Präprozessing. Der einfachere Algorithmus von Horspool verwendet nur die Bad Character Regel und ist in der Praxis sehr effizient.

Dieser Abschnitt ist wie folgt aufgebaut: Nach der ausführlichen Präsentation der Algorithmen analysieren wir die mittlere Laufzeit des Horspool-Algorithmus. Es wird insbesondere gezeigt, dass die mittlere Laufzeit in der Größenordnung $O(\frac{n}{\sigma})$ liegt. Schließlich wird die Bad Character Regel erweitert, indem die Verschiebung nicht nur für einzelne Textzeichen, sondern für alle Wörter einer vorgegebenen Länge q ermittelt wird. Bei optimaler Wahl von q beträgt die mittlere Laufzeit des Horspool-Algorithmus $O(\frac{n \log m}{m})$ für ein Suchwort der Länge m und einen Text der Länge n.

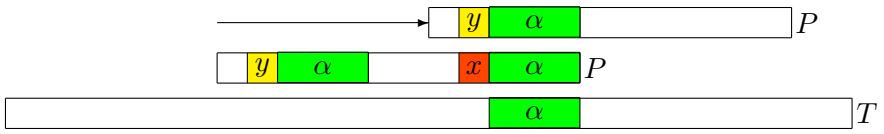
Der Boyer-Moore-Algorithmus

Zunächst nennen und formalisieren wir die beiden bereits genannten Verschiebungsregeln.

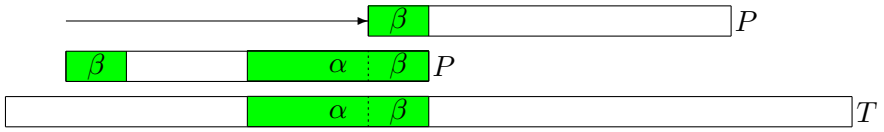
Bad Character Regel. Stimmt das letzte Zeichen von P nicht mit dem letzten Textzeichen x im Suchfenster überein, so darf man P so weit verschieben, dass das letzte Vorkommen von x in P auf diese Textposition trifft.



(Starke) Good Suffix Regel. Stimmt ein Suffix α von P mit dem aktuellen Text überein, und stimmt das Suffix $x\alpha$ nicht mit dem Text überein, so darf man P so weit verschieben, dass das letze Vorkommen von α in P, das kein Vorkommen von $x\alpha$ ist, gegenüber dem entsprechenden Text steht.



Sollte ein solches Vorkommen von α nicht existieren, so darf man P so weit verschieben, dass das längste Suffix von α, das ein Präfix von P ist, gegenüber dem Ende des aktuellen Textfensters steht.



Formal führen wir zur Beschreibung der beiden Heuristiken die folgenden Größen ein.

Definition 1.6 *Für $P \in \Sigma^*$ mit $|P| = m$ und $x \in \Sigma$ ist*

$$R_x(P) := \max\left(\{1 \le i \le m : P[i] = x\} \cup \{0\}\right),$$

d.h., $R_x(P)$ ist das letzte Vorkommen von x in P. Außerdem sei $Shift_x(P) = m - R_x(P)$.

Definition 1.7 *Sei $P \in \Sigma^*$ mit $|P| = m$. Für $1 \le i \le m-1$ sei*

$$\begin{aligned} L_i(P) &:= \max(J_1(P,i) \cup J_2(P,i) \cup \{0\}) \text{ mit} \\ J_1(P,i) &= \{j : m-i < j < m \wedge P[i+1 \ldots m] = P[j-(m-i)+1 \ldots j] \wedge P[j-(m-i)] \neq P[i]\}, \\ J_2(P,i) &= \{j : 1 \le j \le m-i \wedge P[1 \ldots j] = P[m-j+1 \ldots m]\}. \end{aligned}$$

Außerdem sei $L_0(P) := Border(P)$.

Offensichtlich ist $J_1(P,i)$ die Menge aller Positionen in P, an denen ein Vorkommen von $P[i+1 \ldots m]$, aber kein Vorkommen von $P[i \ldots m]$ endet; $J_2(P,i)$ ist die Menge aller Positionen in P, an denen ein Suffix von $P[i+1 \ldots m]$ endet, das ein Präfix von P ist. Aus den beiden Heuristiken ergibt sich:

Boyer-Moore-Verschiebungsregel

Es sei i die erste Stelle von rechts in P, bei der ein Mismatch mit dem Text im aktuellen Suchfenster auftritt. Gilt $i = m$, so verschiebe um $m - R_x(P) = Shift_x(P)$. Anderenfalls verschiebe um $m - L_i(P)$.

Beispiel 1.13 Für $P = abcabba$ mit $\Sigma = \{a, b, c, d\}$ erhalten wir:

x	a	b	c	d
R_x	7	6	3	0
$Shift_x$	0	1	4	7

i	0	1	2	3	4	5	6
L_i	1	1	1	1	1	1	4

Dies führt zu folgenden Verschiebungen.

Suchwort: Textausschnitt:	`abcabba` `......d...`	`abcabba` `......c...`	`abcabba` `.....aa..`	`abcabba` `....cba...`
Verschiebung:	7	4	3	6

□

Algorithmus 1.8 Boyer-Moore-Algorithmus

Eingabe: Wörter P, T mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

(1) *Präprozessing:* Bestimme $R_x(P)$, $x \in \Sigma$ und $L_i(P)$, $0 \leq i < m$.
(2) $S \leftarrow \emptyset; k \leftarrow m;$
(3) **while** $k \leq n$
(4) **if** $P[m] \neq T[k]$ **then** $k \leftarrow k + m - R_{T[k]}(P);$
(5) **else**
(6) $i \leftarrow m - 1;\ j \leftarrow k - 1;$
(7) **while** $i > 0$ **and** $P[i] = T[j]$
(8) $i \leftarrow i - 1;\ j \leftarrow j - 1;$
(9) **if** $i = 0$ **then** $S \leftarrow S \cup \{k - m + 1\};$
(10) $k \leftarrow k + m - L_i(P);$
(11) **return** $S;$

Satz 1.16 *Der Boyer-Moore-Algorithmus findet alle Vorkommen von P in T.*

Auf einen formalen Beweis verzichten wir hier. Informell wurde die Korrektheit der Verschiebungsregeln bereits diskutiert. Zur Laufzeit ist zu sagen, dass sie im schlechtesten Fall $O(mn)$ beträgt. Einen Algorithmus mit linearer Laufzeit im schlechtesten Fall erhält man, indem man folgende zusätzliche Regel (**Regel von Galil**) beachtet:

Regel von Galil
Gibt es in der aktuellen Phase eine Übereinstimmung mit dem Textstück $T[j+1 \ldots k]$ und beträgt die Verschiebung mindestens j,
so vergleiche in der nächsten Phase von rechts höchstens bis zur Stelle $k + 1$.

Der Beweis für dieses Laufzeit-Resultat ist ziemlich schwierig. Man kann ihn im Buch von Gusfield [6] finden.

Boyer-Moore-Präprozessing in Linearzeit

Die für die Verschiebung nach der Bad Character Regel benötigten Werte $R_x(P)$ können sehr einfach mit einem Aufwand von $O(m + \sigma)$ bestimmt werden:

Algorithmus 1.9 BM-Präprozessing: R_x-Werte

Eingabe: Wort $P \in \Sigma^*$, $|P| = m$
Ausgabe: $R_x(P)$ für $x \in \Sigma$
(1) **foreach** $x \in \Sigma$
(2) $\quad R_x \leftarrow 0$;
(3) **for** $i \leftarrow 1$ **to** m
(4) $\quad x \leftarrow P[i]$; $R_x \leftarrow i$;
(5) **return** $(R_x : x \in \Sigma)$;

Zur Bestimmung der für die Good Suffix Regel benötigten L_i-Werte kann man die Z-Werte des Wortes P^r heranziehen. Mit den Notationen aus Definition 1.7 gilt nämlich für $0 \leq i < m$:

$$J_1(P,i) = \{j : m-i < j < m \wedge m > Z_{m-j}(P^r) = (m-j)+(m-i)\},$$
$$J_2(P,i) = \{j : 1 \leq j \leq m-i \wedge Z_{m-j}(P^r) = m\}.$$

Für die Werte $L_i^2(P) = \max(J_2(P,i) \cup \{0\})$, $0 \leq i < m$, ergibt sich mit der Notation $L_m^2(P) = 0$ die folgende Rekursion:

$$L_i^2(P) = \begin{cases} (m-i) & \text{falls } Z_{m-i}(P^r) = m, \\ L_{i+1}^2(P) & \text{sonst .} \end{cases}$$

Um die L_i-Werte zu bestimmen, geht man von den L_i^2-Werten aus und setzt für wachsendes j den L_i-Wert für $(m-i) = Z_{m-j}(P^r) - (m-j)$, d.h. für $i = 2m - j - Z_{m-j}(P^r)$ auf j. Es ergibt sich damit der folgende Algorithmus:

Algorithmus 1.10 BM-Präprozessing: L_i-Werte

Eingabe: Wort $P \in \Sigma^*$, $|P| = m$
Ausgabe: $L_i(P)$ für $0 \leq i \leq m-1$
(1) **for** $j \leftarrow 1$ **to** $m-1$
(2) $\quad Z'_j \leftarrow Z_{m-j}(P^r)$;
(3) $L_m \leftarrow 0$;
(4) **for** $i \leftarrow m-1$ **downto** 1
(5) $\quad$ **if** $Z'_i = m$ **then** $L_i \leftarrow i$;
(6) $\quad\quad$ **else** $L_i \leftarrow L_{i+1}$;
(7) $L_0 \leftarrow L_1$;
(8) **for** $j \leftarrow 1$ **to** $m-1$
(9) $\quad i \leftarrow 2m - j - Z'_j$;
(10) $\quad$ **if** $Z'_j < m$ **then** $L_i \leftarrow j$;
(11) **return** $(L_0, L_1, \ldots, L_{m-1})$;

Satz 1.17 *Algorithmus 1.10 berechnet für ein Wort P der Länge m die Werte L_i ($0 \leq i \leq m-1$) in einer Zeit von $O(m)$.*

Beweis. Die Korrektheit folgt aus den oben angegebenen Beziehungen zu den Z-Werten von P^r. Die Z-Werte von P^r sind in linearer Zeit berechenbar. Der Rest des Algorithmus läuft offensichtlich in linearer Zeit. □

Horspool-Algorithmus

Wie schon erwähnt, benutzt der Horspool-Algorithmus nur die Bad Character Regel für die Verschiebung. Ergibt sich im ersten Vergleich eine Übereinstimmung, so wird am Ende der Phase um den Betrag 1 verschoben. Der Pseudocode des Horspool-Algorithmus ergibt sich also ganz einfach (Algorithmus 1.11).

Algorithmus 1.11 Horspool-Algorithmus

Eingabe: Wörter P, T mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

```
(1)   Präprozessing: Bestimme R_x(P), x ∈ Σ.
(2)   S ← ∅; k ← m;
(3)   while k ≤ n
(4)      if P[m] ≠ T[k] then k ← k + m − R_{T[k]}(P);
(5)      else
(6)         i ← m − 1; j ← k − 1;
(7)         while i > 0 and P[i] = T[j]
(8)            i ← i − 1; j ← j − 1;
(9)         if i = 0 then S ← S ∪ {k − m + 1};
(10)        k ← k + 1;
(11)  return S;
```

Im Vergleich zum Boyer-Moore-Algorithmus muss man feststellen, dass im Falle eines positiven Vergleichs im ersten Schritt einer Suchphase die Verschiebung nach der Bad Character Regel in der Regel kürzer ist als die nach der Good Suffix Regel. Da für größere Alphabete der erste Vergleich allerdings meistens negativ ausfällt, ist dies kein großer Nachteil. Der große Vorteil des Horspool-Algorithmus ist seine Einfachheit, besonders die Unkompliziertheit des Präprozessings. Zwei Bemerkungen noch zum Schluss:

1. Man kann im Falle eines positiven ersten Vergleichs nach dem Ende der Phase um den Betrag $m - R_{T[k]}(P[1 \ldots m-1])$ verschieben. Dann wird bis zum vorletzten Vorkommen von $T[k] = P[m]$ in P verschoben.

2. Nach einem positiven ersten Vergleich ist die Reihenfolge der weiteren Zeichvergleiche beliebig. Man kann insbesondere die weiteren Vergleiche *von links nach rechts* vornehmen. Dies ermöglicht eine Kombination mit den Ideen des KMP-Algorithmus und damit die Schaffung eines relativ einfachen und im Mittel schnellen Algorithmus mit linearer Laufzeit im schlechtesten Fall.

Durchschnittliche Laufzeit des Horspool-Algorithmus

Wir beenden den Abschnitt mit einigen Überlegungen zur *durchschnittlichen* Laufzeit des Horspool-Algorithmus.

Wir möchten die mittlere Laufzeit $T(m, n)$ für Suchwörter der Länge m und Texte der Länge n ermitteln. Dazu bestimmen wir die mittlere Anzahl der Vergleiche $Comp(m)$ sowie die mittlere Verschiebung $Shift(m)$ für Suchwörter und Textabschnitte der Länge m. Für die mittlere Laufzeit gilt $T(m, n) \approx \frac{n \cdot Comp(m)}{Shift(m)}$.

Analog zu den Betrachtungen für den naiven Algorithmus (Abschnitt 1.1) erhalten wir $\mathit{Comp}(m) \leq \frac{\sigma}{\sigma-1}$.

Für die mittlere Verschiebung betrachten wir den Fall, dass im ersten Vergleich ein Mismatch stattfand (bzw. dass die Bemerkung 1 zum Horspool-Algorithmus beachtet wurde). Die mittlere Verschiebung ergibt sich als

$$\mathit{Shift}(m) = \sum_{i=0}^{m-1} p_i \cdot (i+1),$$

wobei p_i die Wahrscheinlichkeit dafür ist, dass für ein $x \in \Sigma$ und ein zufälliges Wort $P \in \Sigma^k$, $\mathit{Shift}_x(P) = i+1$ ist. Offensichtlich gilt

$$p_i = \left(1 - \frac{1}{\sigma}\right)^i \cdot \frac{1}{\sigma} \text{ für } 0 \leq i \leq m-2, \;\; p_{m-1} = \left(1 - \frac{1}{\sigma}\right)^{m-1}, \text{ d.h.}$$

$$\begin{aligned} \mathit{Shift}(m) &= \sum_{i=0}^{m-2} \left(1 - \frac{1}{\sigma}\right)^i \frac{1}{\sigma} \cdot (i+1) + \left(1 - \frac{1}{\sigma}\right)^{m-1} \cdot m \\ &= \sum_{i=0}^{m-2} p^i (1-p) \cdot (i+1) + p^{m-1} \cdot m \text{ mit } p = 1 - \frac{1}{\sigma}. \end{aligned}$$

Analog zu den Betrachtungen für den naiven Algorithmus ergibt sich

$$\mathit{Shift}(m) = \frac{1-p^m}{1-p} = \sigma \left(1 - \left(1 - \frac{1}{\sigma}\right)^m\right).$$

Der Wert von $\mathit{Shift}(m)$ ist abhängig vom Verhältnis zwischen Alphabetgröße σ und Suchwortlänge m. Für $m \gg \sigma$ beträgt er ungefähr σ, während sich für $\sigma \gg m$ in etwa der Wert m ergibt. Für $\sigma = 100$ erhalten wir beispielsweise:

m	2	10	50	100	200	1000
$\mathit{Shift}(m)$	1.99	9.6	39.5	63.4	86.6	99.996

Erweiterte Bad Character Regel

Wie soeben gesehen, ist die mittlere Verschiebung durch die Alphabetgröße σ beschränkt. Für kleine Alphabete (z.B. DNA) ist dies entsprechend gering. Eine Vergrößerung der mittleren Verschiebung kann man erreichen, indem man die Bad Character Regel auf Wörter einer festen Länge q erweitert. Zunächst ermittelt man für jedes Wort α der Länge q das Ende $R_\alpha(P)$ seines letzten Auftretens in P. Falls kein Vorkommen von α in P existiert, setzt man $R_\alpha(P) = q-1$. In einer Suchphase betrachtet man immer die letzten q Symbole des Textfensters. Die erlaubte Verschiebung nach der Suchphase beträgt $\max\{1, m - R_\alpha(P)\}$.

Definition 1.8 *Für $q \geq 1$, $\alpha \in \Sigma^q$, $P \in \Sigma^*$, $|P| = m \geq q$ sei $R_\alpha(P)$ die rechteste Stelle in P, an der ein Vorkommen von α endet bzw. $(q-1)$, falls α nicht in P auftritt.*
Wir definieren $\mathit{Shift}_\alpha(P) := m - R_\alpha(P)$.

Es ergibt sich die *verallgemeinerte Bad Character Regel:* Ergeben die q letzten Zeichen im aktuellen Textfenster das Wort α, so verschiebe um den Betrag $\max\{1, \mathit{Shift}_\alpha(P)\}$.

Man stellt fest, dass die mittlere Zahl der Vergleiche pro Phase $q + O(1)$ und die mittlere Verschiebung $\Theta(\min\{\sigma^q, m\})$ beträgt. Das Präprozessing (Berechnung der R_α) erfordert einen Aufwand von $O(m + \sigma^q)$.

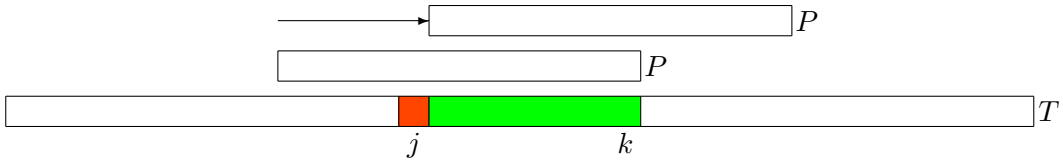
Man kann q natürlich abhängig von m wählen. Die optimale Wahl ist $q = \lceil \log_\sigma m \rceil$. Die mittlere Zahl der Vergleiche pro Phase beträgt dann $\mathit{Comp}(m) = \Theta(\log_\sigma m)$, die mittlere Verschiebung ist $\Theta(m)$. Der Aufwand für das Präprozessing ist $O(\sigma \cdot m)$. Die mittlere Laufzeit ergibt sich als $\Theta(\frac{n \cdot \log_\sigma m}{m})$.

In der Praxis ist die erweiterte Bad Character Regel nur für kleine Alphabete empfehlenswert, da für große Alphabete ein hoher zusätzlicher Speicheraufwand und ein langes Präprozessing nötig sind. So ist z.B. für $\sigma = 100$ und $q = 2$ bereits eine Tabelle mit 10000 Werten zu speichern. Man kann jedoch den benötigten Speicherplatz verringern, indem man mit *Hash-Funktionen* arbeitet. Durch eine Hash-Funktion wird ein Wort $\alpha \in \Sigma^q$ auf eine natürliche Zahl $0 \leq H(\alpha) \leq p - 1$ abgebildet, wobei $p \approx m$ gilt. Eine gute Wahl von H ist $\alpha \, mod \, p$, wobei α als eine Zahl aus dem Intervall $[0, \sigma^q - 1]$ angesehen wird und p eine Primzahl ist. Näheres zu Hash-Funktionen und ihrer Berechnung findet man im Abschnitt 1.8 über den Karp-Rabin-Algorithmus. Im Präprozessing berechnet man dann eine Tabelle R' der Größe p mit $R'_i = \max\{R_\alpha : H(\alpha) = i\}$. Der Rechenaufwand für die Bestimmung von R' beträgt $O(m+p)$. Während der Suche bestimmt man für das Suffix α des Suchfensters den Wert $H(\alpha)$ und verschiebt entsprechend dem Wert von $R'_{H(\alpha)}$. Von Bedeutung ist die erweiterte Bad Character Regel unter Verwendung von Hash-Funktionen jedoch vor allem für die effiziente Suche nach mehreren Wörtern (Wu-Manber-Algorithmus).

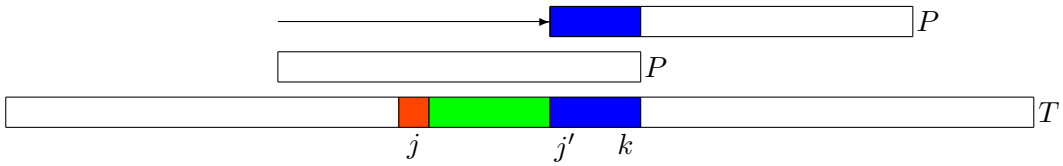
1.6 Algorithmen mit Suffixautomaten

Der Boyer-Moore-Algorithmus vergleicht in einer Phase so lange von rechts nach links, wie der Text mit einem *Suffix* des Suchwortes übereinstimmt. Wie bereits gesehen, ist die mittlere Zahl der Vergleiche $\frac{\sigma}{\sigma-1}$, und die mittlere Verschiebung beträgt σ. Um größere Verschiebungen zu erreichen, kann man die erweiterte Bad Character Regel benutzen.

Eine andere Idee besteht darin, die Vergleiche in einem Textfenster so lange fortzusetzen, wie das Suffix des Textes im Fenster mit einem *Faktor* des Suchwortes übereinstimmt. Es sei k das rechte Ende des aktuellen Suchfensters. Ist $T[j+1 \ldots k]$ ein Faktor von P, $T[j \ldots k]$ aber nicht, so kann ein Vorkommen von P frühestens an der Stelle $j + 1$ beginnen, d.h. man kann das Suchfenster um $k - j$ Stellen verschieben. Sollte an der Stelle k ein Vorkommen von P enden, so verschiebt man um eine Stelle.



Die Verschiebungsregel kann noch verbessert werden. Während der Vergleiche speichert man laufend die kleinste Zahl j', für die $T[j' \ldots k]$ ein *Präfix* von P ist. Nach der Suchphase darf man das linke Ende des Fensters bis zur Stelle j' verschieben.



Beispiel 1.14 Für das Wort $P = abcabba$ und das Textfenster $abcacab$ ergibt sich der längste Faktor cab, das längste Präfix ist ab. Nach der ersten Regel darf man um den Betrag 4 verschieben, nach der zweiten Regel um 5. □

Die Berücksichtigung der Präfixe führt in der Regel zu einer größeren Verschiebung, erfordert aber zusätzlichen Aufwand. Wir geben im folgenden immer die einfachere Variante ohne Berücksichtigung der Präfixe an. In Algorithmus 1.12 ist das allgemeine Schema für die Rückwärts-Suche nach Faktoren dargestellt.

Algorithmus 1.12 Faktor-Algorithmus(Prinzip)

Eingabe: Wörter P, T über Σ mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

```
(1)   k ← m;
(2)   while k ≤ n
(3)      j ← k;
(4)      while T[j ... k] ist Faktor von P
(5)         j ← j − 1;
(6)      if j = k − m then S ← S ∪ {k − m + 1}; k ← k + 1;
(7)                   else k ← j + m;
(8)   return S;
```

Die wesentliche Aufgabe besteht nun darin festzustellen, ob $T[j \dots k]$ ein Faktor von P ist. Dies wird durch endliche Automaten geleistet. Es gibt drei Varianten:

1. Man konstruiert den minimalen partiellen DEA, der die Suffixe von P^r akzeptiert und gibt diesem DEA das Wort im Textfenster von rechts nach links als Eingabe. Der Text von rechts ist genau so lange ein Faktor des Suchwortes, wie ein Nachfolgezustand erreichbar ist.
   Die Konstruktion des DEA ist in linearer Zeit möglich, aber sehr kompliziert. Deshalb hat diese Variante in der Praxis keine Bedeutung.

2. Die Idee ist wie bei der ersten Variante, aber statt des DEA verwendet man den NEA. Dieser hat eine sehr einfache Struktur und wird effizient durch Bitarithmetik implementiert.
   Diese Variante ist sehr schnell für Suchwörter, deren Länge relativ kurz (2 Computer-Wörter) ist.

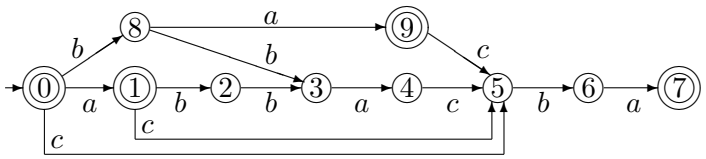
3. Es wird ein partieller DEA konstruiert, der die Suffixe von P^r und einige weitere Wörter akzeptiert (*Orakel*-DEA). Wiederum darf man die Suchphase abbrechen, sobald kein Nachfolgezustand existiert.
   Man benötigt einige Vergleiche mehr als in der ersten Variante. Dafür ist aber der Orakel-DEA sehr viel einfacher zu konstruieren. Diese Variante ist die beste für lange Suchwörter.

Deterministische Suffixautomaten

Zur Bestimmung des längsten Faktors von P an einer Textstelle verwendet man den *Suffixautmaten* von P^r.

Definition 1.9 *Es sei w ein Wort. Der* Suffixautomat *(auch* DAWG *für Directed Acyclic Word Graph) ist der minimale partielle deterministische endliche Automat, dessen akzeptierte Sprache die Menge der Suffixe von w ist.*

Beispiel 1.15 Für $P = abcabba$ ergibt sich der folgende DAWG von $P^r = abbacba$:



□

Ein Textstück $S = T[j \dots k]$ ist genau dann ein Faktor des Suchwortes, wenn für S^r ein Zustand des DAWG definiert ist; es ist zusätzlich genau dann ein Präfix, wenn dieser Zustand ein Endzustand ist.

Beispiel 1.16 Für das Wort $P = abcabba$ und das Textfenster $abcacab$ ergibt sich die Zustandsfolge $0 \xrightarrow{b} 8 \xrightarrow{a} 9 \xrightarrow{c} 5 \xrightarrow{a}$ undefiniert. Der längste Faktor ist folglich cab, das längste Präfix ist ab. □

Algorithmus 1.13 Backward DAWG Matching (BDM-Algorithmus)

Eingabe: Wörter P, T über Σ mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

```
(1)  Konstruiere den DAWG für P^r  A = (Σ, Z, δ, z_0, F);
(2)  S ← ∅; z ← z_0; k ← m;
(3)  while k ≤ n
(4)      j ← k;
(5)      while δ(z, T[j]) existiert
(6)          z ← δ(z, T[j]); j ← j − 1;
(7)      if j = k − m then S ← S ∪ {k − m + 1}; k ← k + 1;
(8)                   else k ← j + m;
(9)  return S;
```

Die Laufzeit des BDM-Algorithmus beträgt im schlechtesten Fall $\Theta(m \cdot n)$. Im Durchschnittsfall ist die Laufzeit $\Theta\left(\frac{n \cdot \ell_m}{m - \ell_m}\right)$, wobei ℓ_m die mittlere Länge des längsten Teilwortes von P ist, das Suffix eines Wortes der Länge m ist. Man kann zeigen, dass $\ell_m \approx \log_\sigma m$ gilt, d.h. die mittlere Laufzeit ist in der Größenordnung $\Theta\left(\frac{n \cdot \log_\sigma m}{m}\right)$.Durch Kombination mit Ideen des KMP-Algorithmus kann man einen Algorithmus gewinnen, der im schlechtesten Fall in linearer Zeit läuft und im Durchschnittsfall nur um einen konstanten Faktor langsamer wird.

Was das Präprozessing betrifft, so kann man zeigen, dass der DAWG eines Wortes der Länge m höchstens $2m$ Knoten und $4m$ Kanten besitzt und in linearer Zeit konstruiert werden kann. Dennoch ist die Konstruktion relativ kompliziert und verlangsamt die praktische Laufzeit.

Nichtdeterministische Suffixautomaten

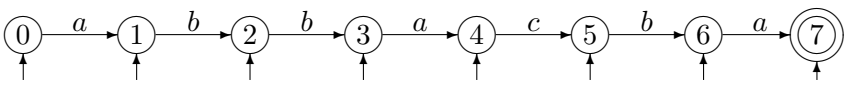
Der NEA, der die Suffixe von P^r akzeptiert, hat eine sehr einfache Struktur und lässt sich mit Hilfe von Bitarithmetik für hinreichend kurze Suchwörter effizient implementieren.

Definition 1.10 *Für ein Wort $S \in \Sigma^*$ der Länge m ist der Suffix-NEA der Automat*

$$A_S = (\Sigma, \{0, 1, \ldots, m\}, \delta, \{0, 1, \ldots, m\}, \{m\})$$

mit $\delta = \{(i-1, P[i], i)\}$ für $1 \leq i \leq m$.

Beispiel 1.17 Für $P = abcabba$ ergibt sich der folgende Suffix-NEA von $P^r = abbacba$:



□

Die Simulation des Suffix-NEA erfolgt ähnlich wie beim Shift-And-Algorithmus. Im Präprozessing wird jedem Buchstaben $x \in \Sigma$ ein Bitvektor $B[x]$ der Länge $m+1$ zugewiesen. In $B[x]$ ist das i-te Bit von hinten genau dann 1, wenn $P[i] = x$ gilt. Die Menge der erreichbaren Zustände sind in einem Bitvektor der $Z = z_m \cdots z_1 z_0$ kodiert, wobei das Bit z_i genau dann den Wert 1 besitzt, wenn im Suffix-NEA von P^r der Zustand $m - i$ erreichbar ist. Bei der Initialisierung von Z erhalten alle $(m+1)$ Bits den Wert 1. Die Aktualisierungsregel für Z lautet

$$Z \leftarrow (Z \texttt{>>} 1) \,\&\, B[T[j]].$$

Das betrachtete Textstück ist ein Faktor des Suchwortes, solange $Z \neq 0$ gilt. Ein Präfix liegt genau dann vor, wenn das rechteste Bit von Z gleich 1 ist.

Beispiel 1.18 Für das Wort $P = abcabba$ erhalten wir im Präprozessing die Bitvektoren $B_a = (\texttt{01001001}), B_b = (\texttt{00110010}), B_c = (\texttt{00000100})$.
Für das Textfenster *abcacab* ergibt sich die Bitvektoren-Folge
$(\texttt{11111111}) \xrightarrow{b} (\texttt{00110010}) \xrightarrow{a} (\texttt{00001001}) \xrightarrow{c} (\texttt{00000100}) \xrightarrow{a} (\texttt{00000000})$.
Der längste Faktor ist folglich *cab*, das längste Präfix ist *ab*. □

Algorithmus 1.14 Backward NDAWG Matching (BNDM-Algorithmus)

Eingabe: Wörter P, T mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

```
(1)   foreach x ∈ Σ
(2)     B[x] ← 0;
(3)   for i ← 1 to m
(4)     B[P[i]] ← B[P[i]] | (1 << (i − 1));
(5)   S ← ∅; k ← m;
(6)   while k ≤ n
(7)     Z ← 1^{m+1}; j ← k;
(8)     while Z ≠ 0
(9)       Z ← (Z >> 1) & B[T[j]];
(10)      if Z ≠ 0 then j ← j − 1;
(11)    if j = k − m then S ← S ∪ {k − m + 1}; k ← k + 1;
(12)                 else k ← j + m;
(13)  return S;
```

Suffix-Orakelautomaten

Für die Korrektheit der Verschiebungsregel des Faktor-Algorithmus ist entscheidend, dass $T[j \dots k]$ *kein Faktor* von P ist. Hingegen ist es unerheblich, ob $T[j+1 \dots k]$ ein Faktor ist. Man darf also ohne weiteres den Suffix-DEA durch einen DEA ersetzen, der außer den Suffixen von P^r eventuell weitere Wörter akzeptiert. Dies leistet der sogenannte *Orakel*-DEA.

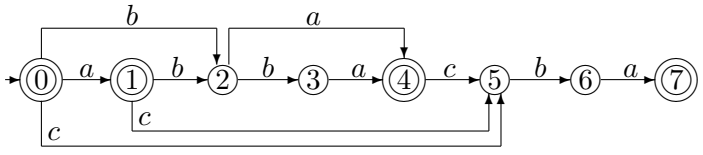
Der Orakel-DEA für ein Wort S mit $|S| = m$ wird ohne Berücksichtigung der Endzustände wie folgt induktiv aufgebaut:

1. Der Orakel-DEA $\varepsilon = S[1 \dots 0]$ besteht aus dem Startzustand 0 und besitzt keine Transitionen.

2. Sei der Orakel-DEA für $S[1 \dots i]$ mit der Transitionenmenge δ konstruiert. Man erhält den Orakel-DEA für $S[1 \dots i+1]$, indem man den Zustand $i+1$ hinzufügt und für jedes Suffix β von $S[1 \dots i]$ eine Transition $(\delta^*(0, \beta), S[i+1], (i+1))$ einfügt, sofern noch keine Transition $(\delta^*(0, \beta), S[i+1], j)$ mit $j \leq i$ existiert.

Die Endzustände des Orakel-DEA erhält man als $\{\delta^*(0, \beta) : \beta \text{ ist Suffix von } S\}$.

Beispiel 1.19 Für $P = abcabba$ ergibt sich der folgende Orakel-DEA von $P^r = abbacba$:



□

Offensichtlich akzeptiert der Orakel-DEA für S alle Suffixe von S. In der Regel werden noch weitere Wörter akzeptiert, in unserem Beispiel *abba* und *bba*. Der Orakel-DEA besitzt $m+1$ Zustände. Die Anzahl der Transitionen scheint auf den ersten Blick im schlechtesten

Fall quadratisch zu sein. Tatsächlich ist sie jedoch durch $2m-1$ beschränkt. Algorithmus 1.15 liefert eine Konstruktion in linearer Zeit. Wesentlicher Bestandteil der Konstruktion ist das Array *Supply*. Der Wert $Supply_i$ gibt das Ende des letzten Vorkommens von β_i in $S[1 \ldots i-1]$ an, wobei β_i das längste Suffix von $S[1 \ldots i]$ ist, das ein Faktor von $S[1 \ldots i-1]$ ist.

Algorithmus 1.15 Konstruktion des Orakel-DEA

Eingabe: Wort $S \in \Sigma$ mit $|S| = m$
Ausgabe: Orakel-DEA für S

```
(1)   Z ← {0}; δ ← ∅; E ← ∅; Supply_0 ← −1;
(2)   for i ← 1 to m
(3)       Z ← Z ∪ {i};
(4)       δ ← δ ∪ {(i − 1, S[i], i)};
(5)       Supply_i ← 0;
(6)       k ← Supply_{i−1};
(7)       while k ≥ 0
(8)           if Transition (k, S[i], j) existiert then Supply_i ← j; break;
(9)                                              else δ ← δ ∪ {(k, S[i], i)}; k ← Supply_k;
(10)  e ← m;
(11)  while e ≥ 0
(12)      E ← E ∪ {e}; e ← Supply_e;
(13)  return A = (Z, Σ, 0, δ, E);
```

Satz 1.18 *Algorithmus 1.15 konstruiert den Orakel-DEA von S mit einem Aufwand von $O(m)$.*

1.7 Duell-Algorithmus von Vishkin

Der in diesem Abschnitt behandelte Algorithmus hat eine völlig andere Grundidee als die bisher untersuchten. Es wird im wesentlichen ausgenutzt, dass der Abstand zweier Vorkommen des Suchwortes P entweder mindestens $m = |P|$ beträgt oder eine Periode von P ist. Haben zwei Textpositionen $k_1 < k_2$ einen Abstand $d = k_2 - k_1 < m$, der keine Periode ist, so kann es – unabhängig vom Text – kein Vorkommen von P an beiden Stellen geben. Für den gegebenen Text T kann man dann, unter Verwendung der in Abschnitt 1.2 eingeführten Z-Werte, mit einem einzigen Zeichenvergleich eine der beiden Positionen als Beginn eines Vorkommens von P ausschließen. Diese Ausschlussprozedur wird *Duell* genannt.

In der ersten Phase des Duell-Algorithmus von Vishkin wird eine Menge C von Kandidaten für ein Vorkommen von P ermittelt. Die Menge C enthält alle tatsächlichen Vorkommen von P im Text T, und Elemente in C sind paarweise *verträglich*, haben also als Abstand entweder eine Periode von P oder mindestens den Abstand m. Über die Aufnahme in die Kandidatenmenge C wird durch Duelle entschieden. In der zweiten Phase wird für jeden Kandidaten durch Zeichenvergleiche verifiziert, ob es sich tatsächlich um ein Vorkommen von P handelt. Da man die Periodeneigenschaft der Abstände zwischen den Kandidaten ausnutzen kann, braucht man für jede Textposition nur einen Vergleich vorzunehmen. Beide Phasen des Algorithmus besitzen eine lineare Laufzeit.

Der Duell-Algorithmus lässt sich sehr gut parallelisieren und auf zweidimensionale Bilder

verallgemeinern. Vor allem soll er hier jedoch besprochen werden, da er einer völlig anderen Grundidee als die anderen Algorithmen folgt.

1. Phase des Algorithmus

Definition 1.11 *Es sei P ein Wort mit $|P| = m$. Zwei natürliche Zahlen k_1 und k_2 mit $k_1 < k_2$ heißen* verträglich *bezüglich P, wenn es einen Text T gibt, der sowohl an der Stelle k_1 als auch an der Stelle k_2 ein Vorkommen von P enthält. Anderenfalls heißen k_1 und k_2* unverträglich.

Satz 1.19 *Es sei P ein Wort mit $|P| = m$. Zwei Zahlen k_1 und k_2 mit $k_1 < k_2$ sind genau dann verträglich bezüglich P, wenn ihre Differenz $d = k_2 - k_1$ eine Periode von P oder größer als m ist.*

Beweis. Sind k_1 und k_2 verträglich bezüglich P, so existiert ein Text T mit

$$T[k_1 \dots k_1 + m - 1] = T[k_2 \dots k_2 + m - 1] = P.$$

Gilt $d = k_2 - k_1 \leq m$, so folgt

$$P[1 + d \dots m] = T[k_1 + d \dots k_1 + m - 1] = T[k_2 \dots k_2 + m - 1 - d] = P[1 \dots m - d],$$

und d ist damit eine Periode von P.

Ist umgekehrt $d = k_2 - k_1$ eine Periode von P oder größer als m, so kann man einen Text T konstruieren, der P an den Stellen k_1 und k_2 enthält (Übungsaufgabe). □

Wichtig für die Laufzeit der 1. Phase ist die folgende "Transitivität" der Verträglichkeit.

Lemma 1.20 *Sind jeweils $k_1 < k_2$ und $k_2 < k_3$ verträglich bezüglich P, so sind auch k_1 und k_3 verträglich bezüglich P.*

Beweis. Zu zeigen ist, dass die Summe zweier Perioden eine Periode oder größer als $|P|$ ist. (Übungsaufgabe) □

Folgerung 1.21 *Ist $C = (k_1 < k_2 < \dots < k_r)$ eine Liste von paarweise bezüglich P verträglichen Positionen und ist die Position k, $k < k_1$, bezüglich P mit k_1 verträglich, so ist k mit allen Positionen aus C bezüglich P verträglich.*

Für den Ausschluss einer von zwei unverträglichen Positionen ist folgendes Lemma nützlich, aus dem sich auch die Duell-Prozedur ergibt.

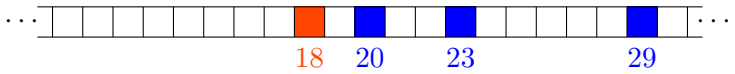
Lemma 1.22 *Sind $k_1 < k_2$ unverträglich bezüglich P und ist $d = k_2 - k_1$, so gilt*
$T[k_1 + Z_d(P)] \neq P[1 + Z_d(P)]$ (d.h. k_1 ist kein Vorkommen von P) oder
$T[k_1 + Z_d(P)] \neq P[1 + Z_d(P) - d]$ (d.h. k_2 ist kein Vorkommen von P).

Beweis. Da k_1 und k_2 unverträglich sind, ist d keine Periode von P, und es folgt $Z_d(P) < m$ sowie $P[1 + Z_d(P)] \neq P[1 + Z_d(P) - d]$ und damit die Behauptung. □

Prozedur: DUELL(k_1, k_2)
Eingabe: Wörter P, T, unverträgliche Positionen $k_1 < k_2$
Ausgabe: nicht ausgeschlossene Position von $\{k_1, k_2\}$
(1) $d \leftarrow k_2 - k_1$;
(2) **if** $T[k_1 + Z_d(P)] = P[1 + Z_d(P)]$ **then return** k_1;
(3) **else return** k_2;

Zur Ermittlung der Kandidatenmenge C benutzt man einen Stack, in den man von rechts beginnend, paarweise verträgliche Kandidaten aufnimmt. Wegen Folgerung 1.21 genügt es sicherzustellen, dass eine neu aufgenommene Position mit der Spitze des Stacks verträglich ist. Sollte die aktuell betrachtete Position k unverträglich mit der Stack-Spitze sein, so wird ein Duell durchgeführt. Gewinnt die Stack-Spitze, so wird k ausgeschlossen. Anderenfalls wird die Stack-Spitze entfernt und die Position k mit der neuen Stack-Spitze auf Verträglichkeit getestet.

Beispiel 1.20 Es sei $P = abcabcabcab$. Die kürzeste Periode von P ist 3. Die obersten Stackpositionen seien $20, 23, 29$; die aktuelle Position 18.



Die Positionen 18 und 20 sind unverträglich; es gilt $Z_2(P) = 2$. Ist $T[20] = c = P[3]$, so gewinnt die Position 18 das Duell, Position 20 scheidet aus; anderenfalls scheidet Position 18 aus. Sollte die Position 18 das Duell gewinnen, so sind 18 und 23 ebenfalls unverträglich und müssen sich ebenfalls duellieren. Gewinnt Position 18 erneut, so wird sie die neue Stackspitze, da 18 und 29 verträglich sind. □

Algorithmus 1.16 Algorithmus von Vishkin: 1. Phase
Eingabe: Wörter P,T, $|P| = m$, $|T| = n$
Ausgabe: Menge (Stack) C von Kandidaten für Vorkommen von P in T
(1) $C \leftarrow \{n + 1\}$;
(2) **for** $k \leftarrow n - m + 1$ **downto** 1
(3) **while** k und TOP(C) unverträglich **and** DUELL$(k,$ TOP$(C)) = k$
(4) POP(C);
(5) **if** k und TOP(C) verträglich **then** PUSH(C, k);
(6) **return** C;

Satz 1.23 *Nach dem Ablauf von Algorithmus 1.16 enthält die Menge C alle Vorkommen von P in T, wobei die Positionen in C paarweise verträglich bezüglich P sind. Die Laufzeit von Algorithmus 1.16 ist $\Theta(n)$.*

Beweis. Die Korrektheit folgt aus den Betrachtungen zu verträglichen Positionen (Satz 1.19 und Folgerung 1.21). Für die Laufzeit stellen wir fest, daß jede Position höchstens einmal in einem Duell unterliegt. Die Zahl der Duelle ist damit durch n beschränkt. □

2. Phase des Algorithmus

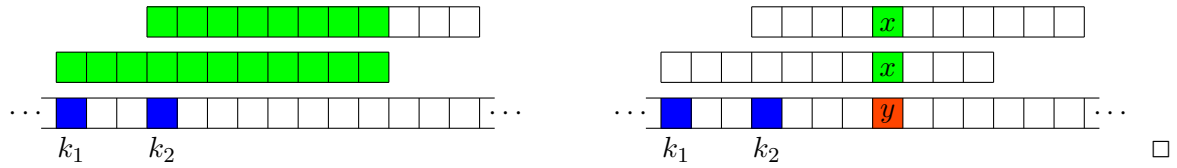
In der zweiten Phase wird für alle Positionen aus C getestet, ob tatsächlich ein Vorkommen vorliegt. Für die Linearität der Laufzeit sind folgende Aussagen entscheidend, die sofort aus der Definition der Periode folgen.

Lemma 1.24 *Es seien P und T Wörter mit $|P| = m$, $|T| = n$.*

1. *Ist $k_2 - k_1$ eine Periode von P und kommt P an der Stelle k_1 in T vor, so ist $T[k_2 \dots k_1 + m - 1]$ ein Präfix von P.*
2. *Ist $k_2 - k_1$ eine Periode von P und gilt $T[k_1 + i - 1] \neq P[i]$ für ein $i \in \{k_2 - k_1 + 1, \dots, m\}$, so ist $T[k_2 \dots k_1 + m - 1]$ kein Präfix von P.*

Damit kann die Vergleichsphase wie folgt vereinfacht werden: Beginnt an der Stelle k ein Vorkommen von P, das an der Textstelle j endet, so braucht man für den nächsten Kandidaten die Vergleiche erst ab Stelle $j + 1$ vorzunehmen. Gibt es beim Vergleich an der Textstelle j dagegen ein Mismatch, so scheiden auch alle weiteren Kandidaten bis j aus.

Beispiel 1.21 Es sei $P = abcabcabcab$; k_1 und $k_2 = k_1 + 3$ seien mögliche Kandidaten. Stimmt $T[k_1 \cdots k_1 + 10]$ mit P überein, so gibt es auch eine Übereinstimmung von $T[k_2 \cdots k_2 + 7]$ mit $P[1 \cdots 8]$. Gibt es dagegen ein Mismatch zwischen $P[i]$ und $T[k_1 + i - 1]$, so existiert auch ein Mismatch zwischen $P[i - 3]$ und $T[k_2 + i - 4] = T[k_1 + i - 1]$.



□

Algorithmus 1.17 Algorithmus von Vishkin: 2. Phase

Eingabe: Wörter P,T, $|P| = m$, $|T| = n$, Stack C (aus der 1. Phase)
Ausgabe: Menge S der Vorkommen von P in T

```
(1)   k ← Pop(C); i ← 1; j ← k;
(2)   while k ≤ n
(3)      while i ≤ m and P(i) = T(j)
(4)         i ← i + 1; j ← j + 1;
(5)      if i > m then
(6)         S ← S ∪ {k}; k ← Pop(C);
(7)         if k ≤ j then i ← j − k + 1;
(8)                  else j ← k; i ← 1;
(9)      else
(10)        while k ≤ j
(11)           k ← Pop(C);
(12)        j ← k; i ← 1;
(13)  return S;
```

Satz 1.25 *Algorithmus 1.17 bestimmt die Vorkommen von P in T mit einem Aufwand von $O(n)$.*

Beweis. Die Korrektheit folgt aus den Eigenschaften von C und Lemma 1.24. Die lineare Laufzeit ergibt sich, da mit jedem positiven Vergleich der Wert von j erhöht und mit jedem negativen Vergleich der Stack verringert wird. □

Parallelisierung

Die Bedeutung des Algorithmus von Vishkin liegt darin, daß er sich parallelisieren läßt. In den folgenden Betrachtungen verwenden wir das Modell der *CREW PRAM* (*concurrent read, exclusive write*). Zunächst soll dieses Modell eines Parallelrechners kurz informell erklärt werden; eine formale Definition findet man z.B. im Buch *An Introduction to Parallel Algorithms* von Joseph JáJá. Eine CREW PRAM verfügt über eine unbegrenzte Anzahl von Prozessoren, die auf einen Speicher zugreifen können. Dabei kann jeder Prozessor von jeder Stelle des Speichers lesen (concurrent read), aber jeder Prozessor hat seinen eigenen Speicherbereich, in dem er alleine schreiben darf (exclusive write). Die Prozessoren sind getaktet, führen also die Schritte synchronisiert aus.

Wichtigste Größen für die Performanz-Analyse von parallelen Algorithmen sind die *Zeit* und die *Arbeit*. Die Zeit ist die Anzahl der parallelen Schritte (also der Takte). Für die Bestimmung der Arbeit wird für jeden Prozessor die Anzahl der aktiven Takte gezählt und über alle Prozessoren summiert. Aus jedem parallelen Algorithmus kann man einen sequenziellen Algorithmus machen, dessen Laufzeit so groß wie die Arbeit des parallelen Algorithmus ist. Bei der Konstruktion paralleler Algorithmen strebt man an, die Zeit gegenüber der Laufzeit der sequenziellen Algorithmen zu verringern, ohne mehr Arbeit zu verrichten. So gelingt es bei der Parallelisierung des Algorithmus von Vishkin, eine Zeit der Größenordnung $O(\log m)$ und eine Arbeit von $O(n)$ zu erreichen.

Der parallele Duell-Algorithmus läuft erneut in 2 Phasen ab. Zuerst ermittelt man parallel eine Menge von Kandidaten. Anschließend werden die Kandidaten parallel verifiziert. Für die erste Phase unterteilt man den Text bis zur Stelle $n - m + 1$ in $\lceil (n - m + 1)/p \rceil$ Intervalle der Länge $p = Per(P)$ und ermittelt für jedes Intervall einen Kandidaten. Zunächst sind alle Positionen im Intervall Kandidaten für ein Vorkommen. In jedem Parallelschritt wird mittels Duellen die Anzahl der Kandidaten halbiert, bis schließlich nach $\lceil \log_2 p \rceil$ Schritten nur noch ein Kandidat übrig ist. In Algorithmus 1.18 ist die Kandidatensuche in einem Intervall dargestellt. Diese Suche wird in allen Intervallen parallel durchgeführt.

Algorithmus 1.18 Paralleler Duell-Algorithmus: 1. Phase

Eingabe: Wörter P,T, $Per(P) = p$, Intervall $[k, k + p - 1]$
Ausgabe: Kandidat c_k für Vorkommen von P im Intervall $[k, k + p - 1]$

```
(1)  pardo for r ← 0 to p − 1
(2)      c_{k+r} ← k + r;
(3)  for i ← 1 to ⌈log_2 p⌉
(4)      pardo for j ← 0 to ⌊p/2^i⌋
(5)          c_{k+j·2^i} ←DUELL(c_{k+j·2^i}, c_{k+j·2^i+2^{i−1}})
(6)  return c_k;
```

Man beachte, dass der Algorithmus auf einer CREW-PRAM implementiert werden kann. Der Prozessor mit dem Index $k + r$ darf dabei ausschließlich den Wert von c_{k+r} schreiben,

aber alle anderen Werte c_j lesen. Offensichtlich führt der Algorithmus in der ersten Phase $O(\log p)$ parallele Schritte aus. Für die Abschätzung der nötigen Arbeit müssen wir vor allem die insgesamt ausgeführte Anzahl von Duellen abschätzen. Man erhält pro Intervall $(p-1)$ Duelle, insgesamt also eine Arbeit von $O(p)$ pro Intervall und $O(n)$ insgesamt.

Die Verifikationsphase verläuft ganz ähnlich. Für einen Kandidaten c setzt man für jede Position $0 \leq r \leq m-1$ eine Boolesche Variable $B_{c,r}$ auf **true** genau, dann wenn das $(r+1)$-te Zeichen in P mit dem $(c+r)$-ten Zeichen in T übereinstimmt. Dann berechnet man in $\log_2 m$ parallelen Schritten den Wert von $\bigwedge_{0 \leq r < m} B_{c,r}$. Die Verifikation erfolgt für alle Kandidaten parallel.

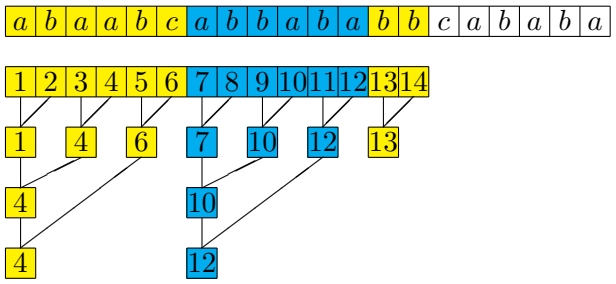
Algorithmus 1.19 Paralleler Duell-Algorithmus: 2. Phase

Eingabe: Wörter P,T, $|P| = m$, Position c
Ausgabe: **true** , falls $T[c \ldots c+m-1] = P$; **false** sonst.

(1) **pardo for** $r \leftarrow 0$ **to** $m-1$
(2) $\quad B_{c,r} \leftarrow (T[c+r] = P[1+r])$;
(3) **for** $i \leftarrow 1$ **to** $\lceil \log_2 m \rceil$
(4) $\quad$ **pardo for** $j \leftarrow 0$ **to** $\lfloor \frac{m}{2^i} \rfloor$
(5) $\quad\quad B_{c,j \cdot 2^i} \leftarrow B_{c,j \cdot 2^i} \wedge B_{c,j \cdot 2^i + 2^{i-1}}$
(6) **return** $B_{c,0}$;

Die Arbeit in der 2. Phase beträgt $O(m)$ für ein Intervall, insgesamt also $O(n \cdot \frac{m}{p})$. Ist das Suchwort nichtperiodisch, d.h. gilt $p > m/2$, so ist die Gesamtarbeit in der Größenordnung $O(n)$. Für periodische Suchwörter muss man die Verifikationsphase etwas sorgfältiger durchführen, um einen Arbeitsaufwand von $O(n)$ zu erhalten.

Beispiel 1.22 Es seien $P = abcabba$, $T = abaabcabbababbcababa$. Die Zeugentafel von P ist $Z(P) = (1, 2, 5, 4, 5, 7, 7)$; die kürzeste Periode von P ist 6. Für die Duell-Phase ergibt sich folgender Ablauf.



Aufteilung in Intervalle
Initialisierung: $c_k \leftarrow k$
Duelle: 1. Runde
Duelle: 2. Runde
Duelle: 3. Runde

Als Kandidaten für ein Vorkommen bleiben $4, 12, 13$ übrig. In der Verifikation ergibt sich 4 als das einzige tatsächliche Vorkommen. □

Weitere Informationen zur parallelen exakten Wortsuche findet man in Kapitel 2 des Buches von Apostolico und Galil sowie in Kapitel 7 des bereits erwähnten Buches von JáJá.

1.8 Karp-Rabin-Algorithmus

Die Idee des Karp-Rabin-Algorithmus ist, ein Wort $u \in \Sigma^m$ auf seinen "Fingerabdruck" (Hash-Funktion) $Hash(u) \in \mathbb{N}$ abzubilden. Im Laufe des Algorithmus werden die Werte $Hash(T_k) = Hash(T[k \dots k-m+1])$ für $1 \leq k \leq n-m+1$ gebildet. Stimmt $Hash(T_k)$ nicht mit $Hash(P)$ überein, so kommt an der Stelle k garantiert *nicht* das Wort P vor. Anderenfalls kann P an der Stelle k vorkommen, es muss aber nicht. Der Karp-Rabin-Algorithmus liefert also eine Menge von Kandidaten für das Vorkommen von P in T.

Damit der Algorithmus effizient abläuft, sollten folgende Bedingungen erfüllt sein:

- Die Funktion *Hash* berechnet für ähnliche Wörter unterschiedliche Werte.
- $Hash(T_{k+1})$ ist aus $Hash(T_k)$ in konstanter Zeit berechenbar.

Wir nehmen o.B.d.A. an, dass $\Sigma = \{0, 1, \dots, \sigma - 1\}$ ist. Dann können wir ein Wort $w = x_1 x_2 \dots x_{m-1} x_m$ mit der Zahl

$$H(w) = \sum_{i=1}^{m} x_i \cdot \sigma^{m-i}$$

identifizieren. Als Hash-Funktion eignet sich $Hash_q(w) = H(w) \bmod q$, wobei q eine Primzahl ist.

Für $a, b \in \Sigma$ und $\alpha \in \Sigma^{m-1}$ gilt $H(\alpha b) = H(a\alpha) \cdot \sigma - a \cdot \sigma^m + b$ und folglich

$$Hash_q(\alpha b) = (Hash_q(a\alpha) \cdot \sigma - a \cdot (\sigma^m \bmod q) + b) \bmod q.$$

Damit berechnet sich $Hash_q(T_{k+1})$ als

$$Hash_q(T_{k+1}) = (Hash(T_k) \cdot \sigma - T[k] \cdot s + T[k+m]) \bmod q,$$

wobei $s = \sigma^m \bmod q$ ist. Wegen $\sigma^k \bmod q = (\sigma \cdot (\sigma^{k-1} \bmod q)) \bmod q$ für alle $k \in \mathbb{N}$ kann man s mit einem Aufwand von $\Theta(m)$ berechnen. Damit kann man also $Hash_q(T_{k+1})$ mit konstanten Aufwand aus $Hash_q(T_k)$ berechnen. Analog können $Hash_q(P)$ und $Hash_q(T_1)$ mit einem Aufwand von $\Theta(m)$ berechnet werden.

Algorithmus 1.20 Karp-Rabin-Algorithmus

Eingabe: Wörter P, T über $\Sigma = \{0, 1, \dots, \sigma - 1\}$ mit $|P| = m, |T| = n$
Ausgabe: Menge C möglicher Vorkommen von P in T

```
(1)  C ← ∅;
(2)  Wähle eine Primzahl q;
(3)  s ← σ^m mod q; h ← Hash_q(P); H ← Hash_q(T[1...m]);
(4)  if H = h then C ← C ∪ {1};
(5)  for k ← 1 to n − m
(6)      H ← (H · σ − T[k] · s + T[k + m]) mod q;
(7)      if H = h then C ← C ∪ {k + 1};
(8)  return C;
```

Beispiel 1.23 Es seien $\Sigma = \{0,1,2,3\}$, $P = 30303$, $T = 10130303123231011203$. Für $q = 11$ bzw. $q = 17$ erhalten wir folgenden Ablauf. (Die Werte T_k sind an die Stelle $k+m-1 = k+4$ geschrieben.)

q	$Hash_q(P)$	$\sigma^m \bmod q$
11	5	1
17	3	4

q	1	0	1	3	0	3	0	3	1	2	3	2	3	1	0	1	1	2	0	3
11					9	5	9	5	7	8	10	9	3	1	2	6	1	3	0	3
17					12	13	1	3	1	6	15	11	1	1	13	7	4	6	3	15

Es bleibt nach diesen beiden Läufen nur die Stelle 4 als mögliches Vorkommen. □

Es gibt jetzt unterschiedliche Möglichkeiten, mit der Kandidatenmenge C umzugehen.

1. Man kann für jeden der Kandidaten zu testen, ob ein Vorkommen von P vorliegt. Dieses ist die einfachste Variante, die man in der Regel bei einer Implementierung wählen wird. Im schlechtesten Fall beträgt die Laufzeit $O(mn)$, im Mittel $O(n)$.

2. Man kann die Menge C als Ergebnis ausgeben. Damit haben wir einen probabilistischen Algorithmus, der ein fehlerhaftes Ergebnis liefern kann (nämlich Vorkommen zu deklarieren, wo keine sind). Im Abschnitt 4.4 des Buchs von Gusfield wird die Fehlerwahrscheinlichkeit genauer untersucht. Wählt man q als zufällige Primzahl aus dem Intervall $[1, mn^2]$, so beträgt die Fehlerwahrscheinlichkeit (für jede Eingabe) $O(1/n)$. Wiederholt man den Algorithmus k-mal und gibt die Menge aller Kandidaten aus, die jedes Mal gefunden wurden, so reduziert sich die Fehlerwahrscheinlichkeit auf $O(1/n^k)$.

3. Man kann mit einem Aufwand von $O(n)$ testen, ob die Kandidatenliste C einen Fehler enthält. Wird ein Fehler gefunden, so wiederholt man den Algorithmus (mit einer neuen Primzahl q). Für den Test teilt man die geordnete Kandidatenliste C in geordnete Teillisten $C_1, C_2, \ldots, C_r$ auf, wobei ein Element zu seinem Nachfolger in einer Liste C_i höchstens den Abstand $m/2$ hat, während der Abstand vom letzten Element von C_i zum ersten Element von C_{i+1} größer als $m/2$ ist. Betrachten wir nun eine Teilliste $C_i = \{k_1, k_2, \ldots, k_t\}$. Wir prüfen zunächst, ob an den Stellen k_1 und k_2 ein Vorkommen von P vorliegt. Sollte dies nicht der Fall sein, so enthält C einen Fehler. Anderenfalls ist $d = k_2 - k_1$ die kleinste Periode von P (Beweis Übungsaufgabe). Wenn alle Kandidaten aus C_i korrekt sind, so muss der Abstand zweier benachbarter Kandidaten jeweils d sein. Wir prüfen deshalb als nächstes, ob $k_j - k_{j-1} = d$ für $3 \leq j \leq t$ gilt. Sollte dies nicht der Fall sein, so enthält C einen Fehler. Anderenfalls brauchen wir für die möglichen Vorkommen $k_3, \ldots, k_t$ nur die letzten d Positionen zu testen.

Variante 1 liefert einen deterministischen Algorithmus mit mittlerer Laufzeit $O(n)$; Variante 2 ist ein probabilistischer Algorithmus mit der Laufzeit $O(n)$ und einer geringen Fehlerwahrscheinlichkeit (*Monte-Carlo*-Algorithmus); Variante 3 ist ein probabilistischer Algorithmus mit korrektem Ergebnis und einer erwarteten Laufzeit von $O(n)$ für jede Eingabe (*Las-Vegas*-Algorithmus).

Interessant am Karp-Rabin-Algorithmus ist, dass bei der Suche keine kombinatorischen Eigenschaften des Suchwortes P benutzt werden. Dies macht den Algorithmus geeignet für die Suche nach komplizierteren Strukturen, z.B. zweidimensionalen Bildern.

1.9 Eine untere Schranke für die mittlere Laufzeit

Zum Abschluss dieses Kapitels soll bewiesen werden, dass eine mittlere Laufzeit von $O(\frac{n \log m}{m})$ für einen Text der Länge n und ein Suchwort der Länge m – wie sie z.B. der Horspool-Algorithmus mit erweiterter Bad Character Regel oder die Faktor-Algorithmen erreichen – optimal ist, wenn $n \geq 2m - 1$ gilt. Wir folgen dabei im wesentlichen den Ideen von Yao aus dem Jahr 1979 [15].

Eine entscheidende Rolle spielen in Yaos Beweis sogenannte Zertifikate, das sind Wörter über dem Alphabet $\Sigma \cup \{\bullet\}$. Das Symbol $\bullet$ kann als Platzhalter betrachtet werden, der durch jedes beliebige Zeichen aus Σ ersetzt werden kann. Ein Wort über diesem erweiterten Alphabet mit l Zeichen aus Σ kann als ein Text angesehen werden, von dem bisher l Positionen durch Zeichenvergleiche bekannt sind. Ein solches Wort wird *Zertifikat* für ein Wort α genannt, wenn sich aus diesen l bekannten Zeichen alle Vorkommen von α im Text ergeben.

Definition 1.12 *Es sei $\bullet \notin \Sigma$ ein Symbol. $S_n(l)$ ist die Menge aller Wörter in $(\Sigma \cup \{\bullet\})^n$ mit genau l Symbolen aus Σ. Für $z \in S_n(l)$ ist $I(z)$ die Menge aller Wörter $w \in \Sigma^n$ mit $w[i] = z[i]$ für alle $1 \leq i \leq n$ mit $z[i] \in \Sigma$.*

Definition 1.13 *Ein Wort $z \in S_n(l)$ heißt* Zertifikat *für $\alpha \in \Sigma^*$, falls α für alle Wörter aus $I(z)$ an den gleichen Stellen vorkommt.*

Lemma 1.26 *Es sei $\alpha \in \Sigma^*$. Gibt es einen Text $T \in \Sigma^n$, für den man alle Vorkommen von α mit höchstens l Vergleichen finden kann, so gibt es für α ein Zertifikat aus $S_n(l)$.*

Beweis. Folgt sofort aus der Definition des Zertifikates. □

Beispiel 1.24 Sei $\Sigma = \{a, b\}$. Das Wort *abba* besitzt das Zertifikat $z = \bullet\bullet a \bullet a \bullet\bullet$. Für jeden Text $T \in I(z)$ kann durch zwei Vergleiche an den Stellen Stellen 3 und 5 festgestellt werden, dass *abba* nicht in T vorkommt. Der Vergleich an der Stelle 3 schließt die Startpositionen 1 und 2 aus, während der Vergleich an der Stelle 5 die Startpositionen 3 und 4 ausschließt.
Dagegen besitzt *abba kein* Zertifikat aus $S_7(1)$, d.h. für *jeden* Text der Länge 7 sind *mindestens* 2 Vergleiche notwendig, um alle Vorkommen von *abba* zu bestimmen. □

Lemma 1.27 *Es seien $n \geq 2m - 1$ und $\alpha \in \Sigma^m$. Besitzt α ein Zertifikat aus $S_n(l)$, so gibt es für α ein Zertifikat aus $S_{2m-1}(\lfloor l/q \rfloor)$ mit $q = \lfloor \frac{n}{2m-1} \rfloor$).*

Beweis. Es sei $z \in S_n(l)$ ein Zertifikat für α. Die Teilwörter

$$z_i = z[(i-1)(2m-1) + 1 \ldots i(2m-1)], 1 \leq i \leq q = \lfloor \frac{n}{2m-1} \rfloor,$$

sind jeweils auch Zertifikate für α. Diese q Wörter sind paarweise disjunkt und besitzen insgesamt höchstens l Zeichen aus Σ. Unter den q Wörtern gibt es folglich eins mit höchstens $\lfloor l/q \rfloor$ Zeichen aus Σ. □

Es genügt damit, folgenden Sachverhalt zu beweisen. Es existieren Konstanten $c_1 > 0, c_2 > 0$, $m_0 \in \mathbb{N}$ derart, dass es für alle $m \geq m_0$ mindestens $c_1 \sigma^m$ Wörter gibt, die kein Zertifikat aus $S_{2m-1}(l_m)$ mit $l_m = c_2 \log_\sigma m$ besitzen und somit auch kein Zertifikat aus $S_n(l_m \cdot \lfloor \frac{n}{2m-1} \rfloor)$ haben und folglich für jeden Text der Länge n mehr als $l_m \cdot \lfloor \frac{n}{2m-1} \rfloor$ Vergleiche benötigen. Der Beweis wird erbracht, indem man für ein Wort $z \in S_{2m-1}(l)$ die Anzahl A der Wörter

der Länge m abschätzt, die das Zertifikat z besitzen (das folgende *Counting Lemma*), und anschließend die Anzahl der Wörter, die überhaupt ein Zertifikat aus $S_{2m-1}(l)$ haben, durch $A \cdot |S_{2m-1}(l)|$ nach oben abschätzt.

Lemma 1.28 (Counting Lemma) *Ein Wort $z \in S_{2m-1}(l)$ mit $l < m$ ist für höchstens*

$$(1 - 1/\sigma^l)^{m/l^2} \cdot \sigma^m$$

Wörter aus Σ^m ein Zertifikat.

Beweis. Es sei J die Menge der Positionen, an denen z ein Zeichen aus Σ hat. Wegen $l < m$ kann z nur ein Zertifikat für Wörter sein, die in keinem Wort aus $I(z)$ vorkommen.

Für $0 \leq k < m$ definieren wir $B_k = \{i : 1 \leq i \leq m \wedge k + i \in J\}$. B_k ist die Menge der Positionen, die relevant für das Nicht-Vorkommen eines Wortes aus Σ^m an der Stelle $k+1$ in den Wörtern aus $I(z)$ sind.

Es sei $\alpha \in \Sigma^m$ ein Wort, für das z ein Zertifikat ist. Da α in keinem Wort aus $I(z)$ vorkommt, gibt es für alle $0 \leq k < m$ ein $i \in B_k$ mit $\alpha[i] \neq z[k+i]$.

Wir konstruieren jetzt eine Menge $K \subseteq \{0, 1, \ldots, m-1\}$ mit $|K| \geq \lceil m/l^2 \rceil$, so dass die Mengen B_k mit $k \in K$ paarweise disjunkt sind. Die Konstruktion von K erfolgt induktiv.

1. $k_1 = 0$.
2. Sind $k_1, k_2, \ldots, k_s$ bestimmt, so sei

$$D_s = \{k : B_k \cap (B_{k_1} \cup B_{k_2} \cup \cdots \cup B_{k_s}) = \emptyset\}.$$

   Gilt $D_s = \emptyset$, so ist $K = \{k_1, \ldots, k_s\}$.
   Anderenfalls ist $k_{s+1} = \min(D_s)$.

Zur Abschätzung von $|K|$ stellen wir folgende Überlegung an. Nach der Definition von K existiert für alle $0 \leq k < m$ ein $j_k \in (\bigcup_{i \in K} B_i) \cap B_k$, wobei $k + j_k \in J$ gilt. Die m Paare $(j_k, k + j_k)$, $0 \leq k < m$, sind offenbar paarweise verschieden. Für die Wahl von j_k gibt es höchstens $|\bigcup_{i \in K} B_i| \leq |K| \cdot l$ verschiedene Werte, für die Wahl von $k + j_k$ höchstens $|J| = l$ verschiedene Werte. Es lassen sich also höchstens $|K| \cdot l^2$ verschiedene Paare $(j_k, k+j_k)$ bilden, d.h. $m \leq |K| \cdot l^2$ bzw. $|K| \geq \lceil m/l^2 \rceil$.

Soll z nun ein Zertifikat für $\alpha \in \Sigma^m$ sein, muss insbesondere für jedes $k \in K$ ein $i \in B_k$ derart existieren, dass $\alpha[i] \neq z[k+i]$ gilt. Für jedes $k \in K$ gibt es $\sigma^{|B_k|} - 1$ Möglichkeiten, die Zeichen $\{\alpha[i] : i \in B_k\}$ zu wählen. An den Stellen i, die in keiner Menge B_k ($k \in K$) liegen, gibt es jeweils σ Möglichkeiten für die Wahl von $\alpha[i]$. Insgesamt kann man die Zahl der Wörter, für die z ein Zertifikat darstellt, abschätzen durch

$$\prod_{k \in K} (1 - 1/\sigma^{|B_k|})\sigma^m \leq (1 - 1/\sigma^l)^{|K|}\sigma^m \leq (1 - 1/\sigma^l)^{\lceil m/l^2 \rceil}\sigma^m.$$

□

Lemma 1.29 *Es sei $m > 2^{44}$ und $l = \frac{1}{2}\log_\sigma m$. Die Anzahl $Z_{2m-1}(l)$ der Wörter aus Σ^m, die ein Zertifikat aus $S_{2m-1}(l)$ besitzen, ist kleiner als $e^{-1}\sigma^m$.*

Beweis. $Z_{2m-1}(l)$ kann wie folgt abgeschätzt werden.

$$\begin{aligned} Z_{2m-1}(l) &\leq |S_{2m-1}(l)|(1-1/\sigma^l)^{m/l^2}\sigma^m = \binom{2m-1}{l}\sigma^l(1-1/\sigma^l)^{m/l^2}\sigma^m \\ &\leq ((2m-1)\sigma)^l(1-1/\sigma^l)^{m/l^2}\sigma^m \\ &\leq ((2m-1)\sigma)^l \exp\left(-\frac{m}{l^2\sigma^l}\right)\cdot\sigma^m \qquad \text{(wegen } \ln(1-1/\sigma^l)\leq -1/\sigma^l) \\ &= \exp\left(\ln((2m-1)\sigma)\cdot l - \frac{m}{l^2\sigma^l}\right)\cdot\sigma^m. \end{aligned}$$

Nach Einsetzen von $l = \frac{1}{2}\log_\sigma m$ ergibt sich die Abschätzung

$$\begin{aligned} Z_{2m-1}(l) &\leq \exp\left(\ln((2m-1)\sigma)\cdot\frac{1}{2}\log_\sigma m - \frac{4m}{(\log_\sigma m)^2 m^{1/2}}\right)\sigma^m \\ &= \exp\left(\ln((2m-1)\sigma)\cdot\frac{1}{2}\log_\sigma m - \frac{4m^{1/2}}{(\log_\sigma m)^2}\right)\sigma^m. \end{aligned}$$

Für $m > 2^{44}$ gilt $\frac{4m^{1/2}}{(\log_\sigma m)^2} > \ln((2m-1)\sigma)\cdot\frac{1}{2}\log_\sigma m + 1$ und damit $Z_{2m-1}(l) < e^{-1}\sigma^m$. □

Satz 1.30 *Für die Suche nach einem Wort der Länge $m > 2^{44}$ in einem Text der Länge $n \geq 2m-1$ benötigt man im Mittel mindestens $\frac{1-1/e}{2}\frac{n\log_\sigma m}{m}$ Vergleiche.*

Zu bemerken ist, dass die Abschätzungen sehr großzügig ausgefallen sind. Die Aussage von Satz 1.30 dürfte für sehr viel kleinere Werte von m gelten.

Kapitel 2

Exakte Suche nach Mengen von Wörtern

Gegeben sind ein Text T und eine endliche Menge von Wörtern $\mathcal{P} = \{P_1, \ldots, P_r\}$. Gesucht sind alle Vorkommen von Wörtern aus $\mathcal{P}$ in T. Ein Vorkommen von P_i an der Stelle k in T wird mit (k, i) angegeben. Anwendungen für dieses Problem existieren in vielfältiger Form, von der Suche nach einigen wenigen Wörtern in einem Text bis zur Suche nach Tausenden von möglichen genetischen Fingerabdrücken aus einer Datenbank in einer DNA-Sequenz.

Die gebräuchlichste Idee für die simultane Suche nach den Wörtern aus $\mathcal{P}$ ist die Konstruktion eines gerichteten Baumes (*Suchwort-Baum, Trie*), dessen Pfade mit den Suchwörtern beschriftet sind. Wichtigste Vertreter der Algorithmen mit Tries sind die Suche mit Hilfe von deterministischen endlichen Automaten (*Aho-Corasick-Algorithmus*) sowie der *Commentz-Walter-Algorithmus* als Verallgemeinerung des Boyer-Moore-Algorithmus. Wenn die Gesamtlänge der Suchwörter hinreichend klein ist, können die Algorithmen mit Bit-Arithmetik, z.B. der Shift-Or-Algorithmus, verallgemeinert werden. In der Praxis hat sich der *Wu-Manber-Algorithmus* bewährt, der den Horspool-Algorithmus verallgemeinert.

Am Ende des Kapitels werden wir uns mit zwei Problemen beschäftigen, die eng mit der Suche nach mehreren Wörtern zusammenhängen. Die *Suche in Wörterbüchern* ist in gewisser Weise das inverse Problem. Hier geht es darum festzustellen, ob ein Text T in einer Menge von Wörtern $\mathcal{P}$ vorhanden ist. Anwendungen finden sich beispielsweise in der Rechtschreibprüfung und in der Datenkompression (LZW-Algorithmus).

2.1 Suchwort-Bäume

Für die weiteren Betrachtungen seien

$$|T| = n,\ |\mathcal{P}| = r,\ \sum_{i=1}^{r} |P_i| = M,\ \max\{|P_i| : 1 \le i \le r\} = m,\ \min\{|P_i| : 1 \le i \le r\} = \mu.$$

Definition 2.1 *Es sei $\mathcal{P} = \{P_1, \ldots, P_r\}$ eine Menge von Wörtern. Der* Suchwort-Baum (Trie) *für $\mathcal{P}$ ist der gerichtete Baum Trie($\mathcal{P}$), der folgende Bedingungen erfüllt:*

1. *Jede Kante ist mit genau einem Buchstaben beschriftet.*
2. *Keine zwei Kanten mit dem gleichen Ausgangsknoten haben die gleiche Beschriftung.*

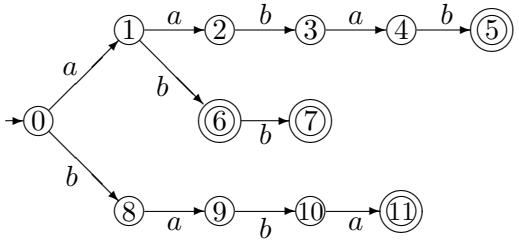
3. *Jeder Knoten v hat eine Information $info(v) \in \{0, 1, \ldots, r\}$. Ist die Beschriftung $\mathcal{L}(v)$ des Weges von der Wurzel nach v ein Wort P_i mit $1 \leq i \leq r$, so gilt $info(v) = i$; anderenfalls ist $info(v) = 0$.*

4. *Ist v ein Blatt, so gilt $\mathcal{L}(v) \in \mathcal{P}$.*

5. *Für jedes Wort $P \in \mathcal{P}$ gibt es einen Knoten v mit $\mathcal{L}(v) = P$.*

Bemerkungen. 1. Die Bezeichnung *Trie* kommt von *re**trie**val tree* und hat deshalb die Aussprache `[tri:]`.
2. $Trie(\mathcal{P})$ ist offensichtlich der minimale partielle DEA, der die Sprache $\mathcal{P}$ akzeptiert und dessen Graph ein Baum ist. Der Startzustand ist die Wurzel, Endzustände sind alle Knoten v mit $info(v) > 0$.

Beispiel 2.1 Der Suchwort-Baum für $\mathcal{P} = \{aabab, ab, abb, baba\}$ sieht wie folgt aus. Die Wurzel ist durch den Pfeil von außen gekennzeichnet, die Knoten sind in der Reihenfolge ihrer Einfügung nummeriert, und Knoten mit einer Information ungleich 0 sind durch Doppelkreise gekennzeichnet. Es gilt $info(5) = 1$, $info(6) = 2$, $info(7) = 3$, $info(11) = 4$.



□

Die Anzahl der Kanten im Trie von $\mathcal{P}$ ist beschränkt durch die Summe M der Längen der Wörter aus $\mathcal{P}$. Bei der Diskussion von Laufzeiten für Algorithmen mit Tries betrachten wir die Alphabetgröße σ zunächst als eine Konstante. Mit dem Einfluss der Alphabetgröße auf die Laufzeit und den daraus resultierenden Strategien für die Implementierung beschäftigen wir uns am Ende von Abschnitt 2.2 (Seite 54).

Die Konstruktion eines Tries ist sehr einfach. Man startet mit einem Baum, der nur aus der Wurzel besteht. Für jedes Suchwort P_k folgt man dem längsten existierenden Pfad, der mit einem Präfix von P_k beschriftet ist, und fügt am Ende dieses Pfades einen Pfad an, der mit dem verbleibenden Suffix von P_k beschriftet ist. Der letzte Knoten des Pfades von P_j wird mit der Information j versehen.

Algorithmus 2.1 Konstruktion des Suchwort-Baumes

Eingabe: Menge von Wörtern $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$
Ausgabe: Suchwort-Baum *Trie*($\mathcal{P}$)

```
(1)   V ← {root}; E ← ∅;
(2)   for k ← 1 to r
(3)      t ← 1; v ← root;
(4)      while t ≤ |P_k| and (E enthält die Kante (v, P_k[t], u))
(5)         v ← u; t ← t + 1;
(6)      for i ← t to |P_k|
(7)         u ← neuer Knoten; info(u) ← 0;
(8)         V ← V ∪ {u}; E ← E ∪ {(v, P[k], u)};
(9)         v ← u;
(10)     info(v) ← k;
(11)  return (V, E);
```

Naive Suche mit Tries

Für jede Textstelle k sucht man von der Wurzel des Suchwort-Baumes nach dem längsten Pfad, dessen Beschriftung ein Präfix von $T[k \ldots n]$ ist. Für jeden erreichten Knoten v mit $info(v) \neq 0$ wird das Vorkommen des entsprechenden Suchwortes an der Stelle k mitgeteilt. Ist der Weg nicht mehr fortsetzbar, bricht man die Suche ab und erhöht k um 1.

Algorithmus 2.2 Naive Suche mit Tries

Eingabe: Menge von Wörtern $\mathcal{P}$, Text T mit $|T| = n$
Ausgabe: Menge S der Vorkommen von Wörtern aus $\mathcal{P}$ in T

```
(1)   Konstruiere Trie(P) mit der Wurzel root.
(2)   S ← ∅;
(3)   for k ← 1 to n
(4)      j ← k; v ← root;
(5)      while (es gibt in Trie(P) eine Kante (v, T[j], u))
(6)         v ← u; j ← j + 1;
(7)         if info(v) > 0 then S ← S ∪ {(k, info(v))};
(8)   return S;
```

Die Laufzeit im schlechtesten Fall beträgt $\Theta(m)$ mit $m = \max\{|P| : P \in \mathcal{P}\}$ für die Suche an einer Textstelle und insgesamt $\Theta(n \cdot m)$. Auch im Durchschnitt ist die Laufzeit nicht mehr linear. Wir nehmen für unsere Betrachtungen der Einfachheit halber an, dass alle Wörter aus $\mathcal{P}$ die gleiche Länge m besitzen und dass $r \ll \sigma^m$ gilt, wobei σ die Mächtigkeit des Alphabetes ist. Die Wahrscheinlichkeit eines Mismatches innerhalb der ersten i Vergleiche in *einem* Suchwort ist $1 - \frac{1}{\sigma^i}$. Für r unabhängig gewählte Suchwörter ist somit die Wahrscheinlichkeit eines Mismatches innerhalb der ersten i Zeichen in jedem Wort $(1 - \frac{1}{\sigma^i})^r$. Wählt man $i = \log_\sigma r$, so erhält man eine Wahrscheinlichkeit von $1 - (1 - \frac{1}{r})^r \geq 1 - 1/e$ dafür, daß mindestens ein Wort eine Übereinstimmung der Länge $\log_\sigma r$ mit dem Text aufweist. Damit beträgt die mittlere Zahl der Vergleiche je Versuch $\Omega(\log_\sigma r)$ und die mittlere Gesamtlaufzeit $\Omega(n \cdot \log_\sigma r)$. Man

kann auch zeigen, dass die mittlere Zahl der Vergleiche je Versuch in der Größenordnung von $\Theta(\log_\sigma r)$ liegt und damit die Gesamtlaufzeit $\Theta(n \cdot \log_\sigma r)$ beträgt.

2.2 DEA-Suche und Aho-Corasick-Algorithmus

Suche mit einem DEA

Für ein einzelnes Wort $P \in \Sigma^*$ wurde in Abschnitt 1.3 der minimale DEA A_P konstruiert, der die Sprache $\Sigma^* P$ akzeptiert. Dabei entsprachen die Zustände von A_P den Präfixen von P, und der Zustand nach Einlesen eines Textes T entsprach dem längsten Suffix von T, das ein Präfix von P ist. Vollkommen analog kann man für eine endliche Menge $\mathcal{P} \subset \Sigma^*$ aus $\mathit{Trie}(\mathcal{P})$ einen DEA $A_\mathcal{P}$ konstruieren, der die Sprache $\Sigma^*\mathcal{P}$ akzeptiert (allerdings in der Regel nicht minimal ist). Es ist dabei zu beachten, dass $A_\mathcal{P}$ außer den Endzuständen von $\mathit{Trie}(\mathcal{P})$ noch weitere besitzen kann: Ein Knoten v ist genau dann ein Endzustand, wenn es ein *Suffix* von $\mathcal{L}(v)$ gibt, das in $\mathcal{P}$ enthalten ist.

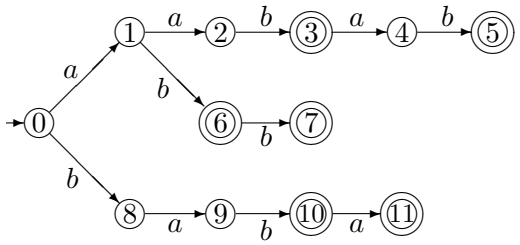
Satz 2.1 *Es sei $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ eine endliche Menge von Wörtern aus Σ^*. Die Sprache $\Sigma^*\mathcal{P}$ wird akzeptiert durch den DEA $A_\mathcal{P} = (\Sigma, V, \delta, root, F)$, wobei*

- *V die Knotenmenge von $\mathit{Trie}(\mathcal{P})$ ist,*
- *$root$ die Wurzel von $\mathit{Trie}(\mathcal{P})$ ist,*
- *F die Menge aller $v \in V$ ist, für die ein Suffix von $\mathcal{L}(v)$ in $\mathcal{P}$ ist,*
- *und die Überführungsfunktion δ für $v \in V, x \in \Sigma$ wie folgt definiert ist:*

  $\delta(v, x) = u$: *$\mathcal{L}(u)$ ist das längste Suffix von $\mathcal{L}(v)$, das Präfix eines Wortes aus $\mathcal{P}$ ist.*

Insbesondere ist $\delta(v, x) = u$, falls die Kante (v, x, u) in $\mathit{Trie}(P)$ enthalten ist.

Beispiel 2.2 Für $\mathcal{P} = \{aabab, ab, abb, baba\}$ erhalten wir den folgenden DEA. Die Knoten 3 und 10 werden zusätzlich zu Endzuständen, da ihre Pfade das Suffix $ab \in \mathcal{P}$ enthalten. Für die bessere Übersichtlichkeit sind nur die Kanten aus $\mathit{Trie}(\mathcal{P})$ angegeben, während die gesamte Überführungsfunktion in einer Tabelle dargestellt wird.



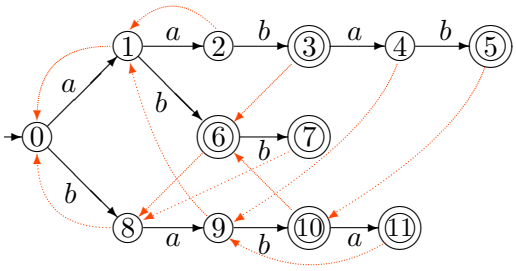
	0	1	2	3	4	5	6	7	8	9	10	11
a	1	2	2	4	2	11	9	9	9	2	11	2
b	8	6	3	7	5	7	7	8	8	10	7	10

□

Für die effiziente Konstruktion des DEA benötigt man eine geeignete Verallgemeinerung der *Border*-Tabelle auf Suchwort-Bäume.

Definition 2.2 *Für einen von der Wurzel verschiedenen Knoten v aus $Trie(\mathcal{P})$ sei $Fail_v$ der Knoten, für den $\mathcal{L}(Fail_v)$ das längste echte Suffix von $\mathcal{L}(v)$ ist, das Präfix eines Wortes aus $\mathcal{P}$ ist. Man bezeichnet $Fail_v$ als* Fehler-Link (failure link) *von v.*

Beispiel 2.3 Für $\mathcal{P} = \{aabab, ab, abb, baba\}$ erhalten wir folgende Fehler-Links:



□

Ähnlich wie die *Border*-Tabelle lassen sich die Fehler-Links effizient bestimmen, indem man Fehler-Links der Vorgänger verwendet. Wichtig ist dabei, dass man die Knoten des Suchwort-Baumes mit wachsender Tiefe, d.h. in der Reihenfolge der Breitensuche (*BFS-Ordnung*), einfügt.

Algorithmus 2.3 Bestimmung der Fehler-Links in einem Trie

Eingabe: $Trie(\mathcal{P})$ für Menge von Wörtern $\mathcal{P}$
Ausgabe: Tabelle *Fail* der Fehler-Links für $Trie(\mathcal{P})$

```
(1)   foreach Knoten v in Trie(P) der Tiefe 1
(2)       Fail_v ← root;
(3)   foreach Knoten v der Tiefe > 1 in BFS-Ordnung
(4)       x ← Beschriftung der Kante (parent(v), v);
(5)       t ← Fail_parent(v);
(6)       while (es gibt keine Kante (t, x, u) and t ≠ root)
(7)           t ← Fail_t;
(8)       if es gibt Kante (t, x, u) then Fail_v ← u;
(9)                                  else Fail_v ← root;
(10)  return Fail;
```

Satz 2.2 *Algorithmus 2.3 bestimmt die Fehler-Links mit einem Aufwand von $O(M)$.*

Die Endzustände und Überführungsfunktion δ des DEA $A_{\mathcal{P}}$ können nun sehr einfach aus den Fehler-Links bestimmt werden. Ein Knoten v ist genau dann ein Endzustand, wenn $info(v) > 0$ gilt oder $Fail(v)$ ein Endzustand ist. Die Überführungsfunktion ergibt sich induk-

tiv wie folgt:

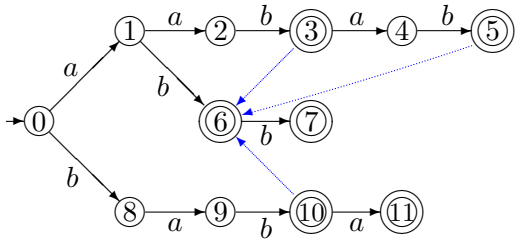
$$\delta(root, x) = \begin{cases} u & \text{falls } \mathit{Trie}(\mathcal{P}) \text{ die Kante } (root, x, u) \text{ enthält,} \\ root & \text{sonst.} \end{cases}$$

$$\delta(v, x) = \begin{cases} u & \text{falls } \mathit{Trie}(\mathcal{P}) \text{ die Kante } (v, x, u) \text{ enthält,} \\ \delta(\mathit{Fail}(v), x) & \text{sonst.} \end{cases}$$

Die DEA-Suche erfolgt, indem man dem DEA $A_{\mathcal{P}}$ den Text T als Eingabe gibt. Wird ein Endzustand erreicht, so ist das Ende eines Wortes aus $\mathcal{P}$ gefunden. Um darüber hinaus festzustellen, *welche* Wörter dem Endzustand entsprechen, werden sogenannte *Ausgabe-Links* eingeführt. Die Menge der Suffixe von $\mathcal{L}(v)$, die Wörter aus $\mathcal{P}$ sind bekommt man erneut rekursiv heraus. Sie besteht aus $P_{\mathit{info}(v)}$, falls $\mathit{info}(v) > 0$ gilt, sowie allen Suffixen von $\mathcal{L}(\mathit{Fail}(v))$, die Wörter aus $\mathcal{P}$ sind.

Definition 2.3 *Für einen von der Wurzel verschiedenen Knoten v aus $\mathit{Trie}(\mathcal{P})$ sei out_v der Knoten, für den $\mathcal{L}(\mathit{out}_v)$ das längste echte Suffix von v ist, das ein Wort aus $\mathcal{P} \cup \{\varepsilon\}$ ist. Man bezeichnet out_v als* Ausgabe-Link (output link) *von v.*

Beispiel 2.4 Für $\mathcal{P} = \{aabab, ab, abb, baba\}$ erhalten wir folgende Ausgabe-Links (Links zur Wurzel sind weggelassen):



Erreicht der DEA einen der Zustände 3 oder 10, so ist ein Vorkommen von ab gefunden; bei Erreichen von Zustand 5 sind ein Vorkommen von $aabab$ sowie ab gefunden. □

Die Bestimmung der Ausgabe-Links erfolgt wiederum induktiv mit Hilfe der Fehler-Links.

$$\mathit{out}(root) = root,$$

$$\mathit{out}(v) = \begin{cases} \mathit{Fail}_v & \text{falls } \mathit{info}(\mathit{Fail}(v)) > 0, \\ \mathit{out}_{\mathit{Fail}(v)} & \text{sonst.} \end{cases}$$

Bei der Suche mittels des DEA findet man alle Vorkommen, die an der Textstelle j enden, indem man vom erreichten Zustand den Ausgabe-Links folgt.

Algorithmus 2.4 DEA-Suche nach endlich vielen Wörtern

Eingabe: endliche Menge von Wörtern $\mathcal{P}$, Text T
Ausgabe: Menge S der Vorkommen von Wörtern aus $\mathcal{P}$ in T

```
(1)   Präprozessing: Konstruiere den DEA A_P und Ausgabe-Links;
(2)   S ← ∅; v ← root;
(3)   for j ← 1 to |T|
(4)       v ← δ(v, T[j]); i ← info(v);
(5)       if i > 0 then
(6)           S ← S ∪ {(j − |P_i| + 1, i)};
(7)       t ← out_v; i ← info(t);
(8)       while t ≠ root
(9)           S ← S ∪ {(j − |P_i| + 1, i)};
(10)          t ← out_t; i ← info(t);
(11)  return S;
```

Aho-Corasick-Algorithmus

Ein Nachteil des DEA ist, wie schon bei der Suche nach einem Wort, dass für jeden Zustand σ Nachfolgezustände definiert werden müssen. Dies ist besonders bei großen Alphabeten problematisch. Eine Lösung besteht wiederum darin, lediglich die Fehler-Links zu bestimmen und diese bei der Ermittlung des Nachfolgezustandes zu benutzen. Dies führt zum Aho-Corasick-Algorithmus.

Algorithmus 2.5 Aho-Corasick-Algorithmus

Eingabe: Menge von Wörtern $\mathcal{P}$, Text T
Ausgabe: Menge S der Vorkommen von Wörtern aus $\mathcal{P}$ in T

```
(1)   Präprozessing: Bestimme Trie(P) mit Fehler- und Ausgabe-Links;
(2)   S ← ∅; v ← root;j ← 1;
(3)   for j ← 1 to |T|
(4)       while es gibt keine Kante (v, T[j], u) and v ≠ root
(5)           v ← Fail_v;
(6)       if es gibt Kante (v, T[j], u) then v ← u;
(7)       i ← info(v);
(8)       if i > 0 then
(9)           S ← S ∪ {(j − |P_i| + 1, i)};
(10)      t ← out_v; i ← info(t);
(11)      while t ≠ root
(12)          S ← S ∪ {(j − |P_i| + 1, i)};
(13)          t ← out_t; i ← info(t);
(14)  return S;
```

Satz 2.3 *Algorithmus 2.5 findet die Vorkommen aller Wörter aus $\mathcal{P}$ in T mit einem Aufwand von $O(n + A)$, wobei A die gesamte Anzahl der Vorkommen von Wörtern aus $\mathcal{P}$ in T ist.*

Einfluss der Alphabetgröße

Bisher haben wir die Größe des Alphabetes als eine Konstante angesehen. Die Alphabetgröße wirkt sich dann auf die Laufzeit aus, wenn in einem Knoten nach einer ausgehenden Kante gesucht wird, die mit einem vorgegebenen Symbol $x \in \Sigma$ markiert ist (z.B. Zeile 4 im Aho-Corasick-Algorithmus). Es bieten sich folgende Möglichkeiten an, die von einem Knoten ausgehenden Kanten zu speichern:

1. als Adjazenzliste,
2. als geordnete Liste (Array),
3. als Array über der Indexmenge Σ.

Sei $d(v)$ der Ausgangsgrad des Knoten v. Der Aufwand für die Suche nach einer Nachfolgerkante von v beträgt $O(d(v))$ bei Variante 1, $O(\log(d(v)))$ bei Variante 2, $O(1)$ bei Variante 3. Allerdings wird der Zeitgewinn bei Variante 3 durch einen höheren Bedarf an Speicherplatz erkauft, Variante 2 erfordert ein zusätzliches Sortieren der Kanten. Außerdem ist für diese Variante die Aktualisierung komplizierter, wenn der Suchwortbaum *dynamisch* sein soll, d.h. wenn Wörter neu eingefügt bzw. gelöscht werden sollen. Es bietet sich an, im Falle großer Alphabete für jeden Knoten je nach seinem Ausgangsgrad eine der Varianten zu wählen. Für einen kleinen bis mittleren Grad sind die Varianten 1 oder 2 am besten geeignet, für einen großen Grad Variante 3. Üblicherweise werden hohe Ausgangsgrade nahe der Wurzel auftreten.

2.3 Weitere Algorithmen

Der Aho-Corasick-Algorithmus verallgemeinerte den Morris-Pratt-Algorithmus und erreichte wie dieser eine lineare Laufzeit im schlechtesten wie auch im mittleren Fall. Auch die anderen Ideen für die Suche nach einem einzelnen Wort kann man verallgemeinern, so den Bit-Parallelismus und die Suche von rechts nach links im aktuellen Suchfenster. Allerdings ist die Anwendung des Bit-Parallelismus, wie z.B. im Shift-And-Algorithmus, nur bei einer geringen Gesamtlänge der Suchwörter sinnvoll. Die Beschleunigung durch die Suche von rechts nach links wiederum ist bei weitem nicht so spektakulär wie bei der Suche nach einem einzelnen Wort, da die mittlere Anzahl der erfolgreichen Vergleiche im Suchfenster höher ist und die maximal mögliche Verschiebung durch die Länge des kürzesten Wortes beschränkt ist. Auf eine detaillierte Beschreibung der Algorithmen verzichten wir hier und verweisen auf die Bücher von Gusfield sowie Navarro/Raffinot.

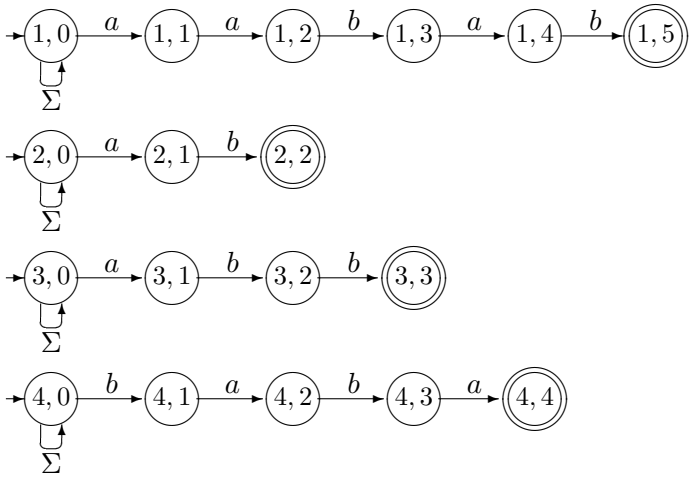
Shift-And-Algorithmus für Mengen

Es sei $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ eine Menge von Suchwörtern über Σ mit $|P_i| = m_i$. Analog zur Suche nach einem einzelnen Wort erhält man den NEA $NEA_{\mathcal{P}} = (Z, \Sigma, \delta, I, F)$ mit
$Z = \{(i,j) : 1 \leq i \leq r, 0 \leq j \leq m_i\}$, $I = \{(i,0) : 1 \leq i \leq r\}$, $F = \{(i,m_i) : 1 \leq i \leq r\}$ und der Menge der Transitionen

$$\delta = \{((i,0), x, (i,0)) : 1 \leq i \leq r, x \in \Sigma\} \cup \{((i,j-1), P_i[j], (i,j)) : 1 \leq i \leq r, 1 \leq j \leq m_i\},$$

der die Menge $\Sigma^*\mathcal{P}$ akzeptiert.

Beispiel 2.5 Für $\mathcal{P} = \{aabab, ab, abb, baba\}$ und $\Sigma = \{a, b, c\}$ ergibt sich der folgende NEA:



□

Man kodiert jetzt die Zustände (i, j) mit $j > 0$ durch einen Bitvektor der Länge $M = \sum_{i=1}^{r} m_i$, wobei dem Zustand (i, j) das Bit mit der Nummer $N_{i,j} := \sum_{k=1}^{i-1} m_k + j$ entspricht. Die Zustände $(i, 0)$ sind immer erreichbar und brauchen deshalb nicht kodiert zu werden. Für die Buchstaben des Alphabetes werden Bitmasken der Länge M erstellt. Dabei wird im Bitvektor B_x das zu (i, j) gehörige Bit mit der Nummer $N_{i,j}$ genau dann auf 1 gesetzt, wenn $P_i[j] = x$ gilt. Außerdem benötigen wir noch Bitvektoren I und F, um die Nachfolger der Startzustände bzw. die Endzustände darzustellen. Dabei sind im Bitvektor I genau die Bits an den Positionen $\sum_{k=1}^{i-1} m_k + 1$ mit 1 belegt, während es in F die Bits an den Positionen $\sum_{k=1}^{i} m_k$ sind (jeweils mit $1 \leq i \leq r$).

Beispiel 2.6 Für $\mathcal{P} = \{aabab, ab, abb, baba\}$ und $\Sigma = \{a, b, c\}$ erhalten wir folgende Bitvektoren der Länge $5 + 2 + 3 + 4 = 14$:

$B_a =$ `10100010101011`, $B_b =$ `01011101010100`, $B_c =$ `00000000000000`,
$I =$ `00001010010001`, $F =$ `10000101001000`. □

Der Bitvektor Z für die Menge der erreichbaren Zustände wird mit 0^M initialisiert. Ein Überführungsschritt für das Textzeichen x läuft in völliger Analogie zur Suche nach einem einzelnen Wort wie folgt ab:

$Z \leftarrow Z \ll 1;\ \ Z \leftarrow Z \,|\, I;\ \ Z \leftarrow Z \,\&\, B_x;$

Das Ende eines Suchwortes wurde genau dann gefunden, wenn $Z \,\&\, F \neq 0$ gilt. In diesem Falle ist zu prüfen, um welches Wort es sich handelt. Der Aufwand für eine Aktualisierung von Z beträgt $O(\lceil M/w \rceil)$, wobei w die Länge eines Computerwortes ist. Die Gesamtzeit liegt in der Größenordnung $O(M)$ für das Präprozessing und $O(\lceil M/w \rceil \cdot n)$ für die Suche in einem Text der Länge n (ohne die Bestimmung der jeweiligen Treffer). Damit ist der Shift-And-Algorithmus für Mengen nur effizient, wenn die Summe der Wortlängen klein ist. Dies ist beispielsweise der Fall, wenn man die Suche nach mehreren Wörtern als Bestandteil der *inexakten Suche nach einem Wort* einsetzt (siehe Kapitel 3).

Der Commentz-Walter-Algorithmus

Der Commentz-Walter-Algorithmus ist die Verallgemeinerung des Boyer-Moore-Algorithmus auf endlich viele Wörter. Sind die Wörter $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ gesucht, so konstruiert man

den Suchwort-Baum für $\mathcal{P}^r = \{P_1^r, P_2^r, \ldots, P_r^r\}$. In einer Phase vergleicht man den Text von rechts nach links mit den Zeichen des Baumes von der Wurzel aus und verschiebt entsprechend der Bad Character Regel oder der Good Suffix Regel, die jeweils auf den Fall mehrerer Wörter verallgemeinert werden.

Man kann nachweisen, dass die mittlere Laufzeit sublinear ist, wenn die Anzahl der Suchwörter nicht zu groß ist und dass das Präprozessing mit einem Aufwand von $\Theta(M)$ erfolgen kann. In der Praxis spielt der Commentz-Walter-Algorithmus jedoch kaum eine Rolle, da das Präprozessing sehr kompliziert ist, und einfachere Algorithmen mit besserer praktischer Performanz existieren. Die Bedeutung des Commentz-Walter-Algorithmus besteht darin, dass es der historisch erste Algorithmus für die Suche nach mehreren Wörtern mit sublinearer Laufzeit ist.

Horspool-Algorithmus für Mengen

Der Horspool-Algorithmus lässt sich sehr einfach auf endliche Mengen von Wörtern erweitern, indem als Verschiebung das Minimum der Verschiebungen gewählt wird, die sich nach der Bad Character Regel für die einzelnen Wörter ergeben.

Definition 2.4 *Es sei $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ eine Menge von Wörtern über Σ. Für $x \in \Sigma$ definieren wir $Shift_x(\mathcal{P}) = \min\{Shift_x(P_i) : 1 \le i \le r\}$.*

Es ist klar, dass die Verschiebung nicht größer als das Minimum μ der Längen der Wörter aus $\mathcal{P}$ sein kann. Analog zur Suche in einem Wort kann die Tabelle der Werte $Shift_x(\mathcal{P}), x \in \Sigma$, mit einem Aufwand von $O(\sigma + r\mu)$ konstruiert werden.

Die mittlere Anzahl von Vergleichen pro Phase liegt in der Größenordnung $\log_\sigma r$. Die Wahrscheinlichkeit, dass die Verschiebung für *ein* Suchwort mindestens $i+1$ für $0 \le i < \mu$ beträgt, ist $\left(1 - \frac{1}{\sigma}\right)^i$. Die Wahrscheinlichkeit, dass die Verschiebung für r unabhängige Suchwörter mindestens $i+1$ beträgt, ist dann $\left(1 - \frac{1}{\sigma}\right)^{ir}$. Die mittlere Verschiebung erhalten wir damit als

$$\sum_{i=0}^{\mu-1} \left(1 - \frac{1}{\sigma}\right)^{ri} = \frac{1 - \left(1 - \frac{1}{\sigma}\right)^{r\mu}}{1 - \left(1 - \frac{1}{\sigma}\right)^{r}}.$$

Damit ist der Horspool-Algorithmus nur für den Fall großer Alphabete und einer kleinen Anzahl von Suchwörtern interessant. Für $\sigma = 100$ und $\mu = 20$ erhält man z.B. folgende mittlere Anzahl von Vergleichen bzw. mittlere Verschiebungen:

Anzahl r der Suchwörter	1	2	5	10	20	50	100	200
mittlere Anzahl der Vergleiche	1.01	1.02	1.05	1.1	1.2	1.4	1.7	1.9
mittlere Verschiebung	18.2	16.6	12.9	9.1	5.4	2.5	1.6	1.2

Wu-Manber-Algorithmus

Wie bereits bei der Besprechung des Horspool-Algorithmus für ein Wort gesagt wurde, kann man die Bad Character Regel verallgemeinern, indem man die letzten q Symbole im Suchfenster zur Bestimmung der Verschiebung heranzieht. Analog zum Fall der Suche nach einem einzelnen Wort ergibt sich als optimale Wahl für die Blocklänge $q = \lceil \log_\sigma(\mu \cdot r) \rceil$. Die Anzahl der Blöcke beträgt σ^q und liegt damit zwischen $\mu \cdot r$ und $\mu \cdot r \cdot \sigma$. Diese Anzahl ist für den Fall großer Alphabete zu hoch, weshalb man wieder Hash-Werte verwendet. Eine weitere Idee des

Wu-Manber-Algorithmus ist es, nicht den Trie für die umgekehrten Suchwörter zu konstruieren, sondern eine zweite Hash-Funktion zu benutzen, die für jedes Wort aus $\mathcal{P}$ das Suffix der Länge q auf einen Hash-Wert abbildet.

Damit verwendet der Algorithmus zwei Hash-Funktionen $Hash_1 : \Sigma^q \to \{1, 2, \ldots, p_1\}$ und $Hash_2 : \Sigma^q \to \{1, 2, \ldots, p_2\}$, wobei idealerweise $p_1 \approx \mu \cdot r$ und $p_2 \approx r$ gelten. Sollte der Speicherplatz beschränkt sein, so muss man die Werte von p_1 und p_2 kleiner wählen, was natürlich die Effizienz beeinflusst. Außerdem gibt es zwei Arrays
$SHIFT$ der Länge p_1 und PAT der Länge p_2.

Der Wert $SHIFT_i$, $1 \leq i \leq p_1$, ist die minimale Verschiebung nach der verallgemeinerten Bad Character Regel über alle Wörter aus $\mathcal{P}$ und alle Wörter $\beta \in \Sigma^q$ mit $Hash_1(\beta) = i$.
PAT_j, $1 \leq j \leq p_2$, ist die Liste aller Indizes der Wörter aus $\mathcal{P}$, deren Suffix der Länge q den $Hash_2$-Wert j besitzt.

Eine Suchphase des Wu-Manber-Algorithmus läuft folgendermaßen ab. Gesucht sind die Vorkommen von Wörtern aus $\mathcal{P}$, die an der Stelle k enden, und es sei $T[k-q+1 \ldots k] = \beta$. Man berechnet zunächst $i = Hash_1(\beta)$. Gilt $SHIFT_i > 0$, so endet an der Stelle k kein Vorkommen eines Wortes aus $\mathcal{P}$, und man darf um den Betrag $SHIFT_i$ verschieben. Anderenfalls berechnet man $j = Hash_2(\beta)$ und überprüft für jedes Wort aus der Liste PAT_j, ob es tatsächlich an der Stelle k endet.

Beispiel 2.7 Es seien $\Sigma = \{0, 1, 2, 3\}$ und $P_1 = 1200321$, $P_2 = 03123232$, $P_3 = 101232310$. Die Blocklänge sei $q = 2$, die Hash-Funktionen seien definiert durch $Hash_1(ab) = ((4a + b) \, mod \, 11) + 1$ bzw. $Hash_2(ab) = ((4a + b) \, mod \, 3) + 1$ für $a, b \in \Sigma$.

Es ergibt sich die folgende $SHIFT$-Tabelle. Dabei enthalten die obere Zeile die möglichen Blöcke der Länge q, die mittlere Zeile die $Hash_1$-Werte für die Blöcke und die untere Zeile den $SHIFT$-Wert für den jeweiligen $Hash_1$-Wert.

Block	00,23	01,30	02,31	03,32	10,33	11	12	13	20	21	22
$Hash_1$	1	2	3	4	5	6	7	8	9	10	11
$SHIFT$	1	6	1	0	0	6	4	6	4	0	6

Es gilt $Hash_2(P_1) = 1$, $Hash_2(P_2) = 3$, $Hash_2(P_3) = 2$. Damit ergibt sich für die PAT-Tabelle: $PAT_1 = \{1\}$, $PAT_2 = \{3\}$, $PAT_3 = \{2\}$.

Die nächste Tabelle gibt die Arbeitsweise des Algorithmus für einige mögliche Situationen wieder. Der $Hash_2$-Wert wird nur berechnet, wenn der $SHIFT$-Wert 0 ist.

Text-Block	$Hash_1$	$SHIFT$	$Hash_2$	Aktion
00	1	1		Verschiebe um 1.
03	4	0	1	Teste, ob P_1 vorliegt. Dann verschiebe um 1.
12	7	4		Verschiebe um 4.
32	4	0	3	Teste, ob P_2 vorliegt. Dann verschiebe um 1.

□

Die praktische Laufzeit des Wu-Manber-Algorithmus hängt sehr stark von den verwendeten Hash-Funktionen ab. Da diese sehr häufig berechnet werden müssen, sollten sie sehr schnell zu bestimmen sein. (Die im Beispiel benutzten Funktionen sind in der Praxis nicht so günstig, da die Berechnung der Modulo-Funktion recht lange dauert.) Außerdem sollten die Werte möglichst gleichmäßig verteilt sein. Es kann weiterhin vorkommen, dass viele Suchwörter die

gleiche Endung besitzen und folglich den gleichen $Hash_2$-Wert liefern. In solchen Fällen ist es empfehlenswert, für die Berechnung von $Hash_2$ weitere Buchstaben einzubeziehen, z.B. den μ-ten Buchstaben von hinten.

Faktor-Algorithmen

Die Idee der Rückwärtssuche nach einem Faktor kann auf den Fall mehrerer Wörter verallgemeinert werden. Die Konstruktion des DEA, der die Suffixe aller Wörter akzeptiert (BDM), wird allerdings noch komplizierter als im Fall eines einzelnen Suchwortes. Die Verwendung des NEA unter Ausnutzung des Bit-Parallelismus (BNDM) ist nur effizient, wenn die Anzahl der Suchwörter klein ist. Relativ einfach ist der Orakel-Automat zu verallgemeinern. Auch hier wird ein DEA konstruiert, der die Suffixe aller Wörter und weitere Wörter akzeptiert. Dieser Algorithmus ist ähnlich effizient wie der Wu-Manber-Algorithmus. Für Details siehe das Buch von Navarro und Raffinot [12].

2.4 Suche in Wörterbüchern

Wir betrachten jetzt die folgende Problemstellung. Gegeben sind nach wie vor eine Menge von Wörtern $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ (ein Wörterbuch) und ein Wort T. Die Frage lautet, ob T in $\mathcal{P}$ enthalten ist.

Das Problem ist natürlich einfach zu lösen, indem man T mit allen Wörtern aus $\mathcal{P}$ vergleicht. Häufig ist jedoch die Menge $\mathcal{P}$ fest vorgegeben, und die Frage wird in der Zukunft für mehrere Wörter T gestellt, z.B. bei der Suche in einem Lexikon oder der Rechtschreibprüfung. In der Regel ist die Anzahl der Wörter aus $\mathcal{P}$ sehr viel größer als die Länge von T. Die Aufgabenstellung ist somit wie folgt zu präzisieren: Nach einem Präprozessing von $\mathcal{P}$ will man in der Lage sein, für ein Wort T möglichst in der Zeit $O(|T|)$ zu entscheiden, ob T in $\mathcal{P}$ enthalten ist.

Diese Aufgabe kann sehr elegant mit Hilfe von Suchwort-Bäumen gelöst werden. Hat man nämlich den Suchwort-Baum von $\mathcal{P}$ konstruiert, so muss man nur ermitteln, ob es einen mit T beschrifteten Weg von der Wurzel aus gibt und ob dieser Weg in einem Knoten endet, der einem Wort aus $\mathcal{P}$ entspricht. Der Aufwand ist $O(|T| \cdot \log_2 \sigma)$, wenn die ausgehenden Kanten für jeden Knoten lexikografisch geordnet sind.

Eine Alternative zur Verwendung von Suchwort-Bäumen stellt ein geordnetes Array dar. Dabei werden die Wörter aus $\mathcal{P}$ lexikografisch geordnet. (Dies ist ja auch die Situation, die man aus gedruckten Wörterbüchern bzw. Lexika kennt.) Für ein Wort T wird dann mittels binärer Suche festgestellt, ob es in $\mathcal{P}$ enthalten ist. Der Aufwand für die binäre Suche beträgt im schlechtesten Fall $O(\log_2 r \cdot |T|)$ und im mittleren Fall $O(\log_2 r)$.

Die Vorteile des geordneten Arrays sind die einfache Struktur und der sehr viel geringere Platzbedarf. Vorteile des Suchwort-Baumes sind die einfache Modifizierbarkeit (Hinzufügen bzw. Entfernen von Suchwörtern) sowie die einfache Verwendbarkeit für die *inexakte* Suche. Um Platz zu sparen, verwendet man allerdings häufig bei großen Wörterbüchern statt des Suchwort-Baumes den minimalen DEA, der die Menge $\mathcal{P}$ akzeptiert. Dieser DEA ist ein azyklischer gerichteter Graph und hat in der Regel sehr viel weniger Zustände (Knoten) als der Suchwort-Baum.

Der LZW-Algorithmus

Eine große Bedeutung haben Wörterbücher in der Datenkompression. Die Grundidee ist, dass man eine bereits einmal aufgetretene Zeichenkette bei späteren Vorkommen durch eine (kürzere) Referenz ersetzt. Die erste Arbeit zur Kompression mit Hilfe von Wörterbüchern stammt von LEMPEL und ZIV aus dem Jahr 1977. Hier stellen wir den LZW-Algorithmus, eine Variante von WELCH aus dem Jahre 1983, vor. Die Ausgabe ist eine Folge von Zahlen, den Indizes der im Wörterbuch gefundenen Wörter. Der Index eines Wortes entspricht dem Zeitpunkt seiner Einfügung. Am Anfang werden die einzelnen Buchstaben des Alphabets in ihrer lexikografischen Reihenfolge in das Wörterbuch aufgenommen. Ist der zu komprimierende Text T der Länge n bereits bis zur Position i verarbeitet, so sucht man im Wörterbuch das längste Wort $T[i \dots j]$, das im Wörterbuch enthalten ist. An die Ausgabe wird der Index des Wortes $T[i \dots j]$ im Wörterbuch angehängt. In das Wörterbuch wird das Wort $T[i \dots j+1]$ eingefügt.

Algorithmus 2.6 LZW-Kompression

Eingabe: Text $T \in \Sigma^*$, $|T| = n$, $\Sigma = \{a_1, a_2, \dots, a_\sigma\}$
Ausgabe: Komprimierter Text (Indexfolge) Z

(1) $D \leftarrow \{1 \rightarrow a_1, 2 \rightarrow a_2, \dots, \sigma \rightarrow a_\sigma\}$; $d \leftarrow \sigma + 1$;
(2) $Z \leftarrow \varepsilon$; $j \leftarrow 1$;
(3) **while** $j \leq n$
(4) $\quad i \leftarrow j$;
(5) $\quad$ **while** D enthält $T[i \dots j]$
(6) $\quad\quad j \leftarrow j + 1$;
(7) $\quad Z \leftarrow Z \cdot$ Index von $T[i \dots j-1]$;
(8) $\quad D \leftarrow D \cup \{d \rightarrow T[i \dots j]\}$; $d \leftarrow d + 1$;
(9) **return** Z;

Beispiel 2.8 Es sei $\Sigma = \{a, b, c\}$ und $T = bacbacbacba$. Dann ergibt sich folgender Ablauf des Algorithmus. Der Index eines Wortes im Wörterbuch ergibt sich aus seiner Position. Die Position im Text wird durch den senkrechten Strich angezeigt.

Wörterbuch	Text	Ausgabe	Neuer Eintrag
(a, b, c)	$\|bacbacbacba$	2	ba
(a, b, c, ba)	$b\|acbacbacba$	1	ac
(a, b, c, ba, ac)	$ba\|cbacbacba$	3	cb
(a, b, c, ba, ac, cb)	$bac\|bacbacba$	4	bac
$(a, b, c, ba, ac, cb, bac)$	$bacba\|cbacba$	6	cba
$(a, b, c, ba, ac, cb, bac, cba)$	$bacbacb\|acba$	5	acb
$(a, b, c, ba, ac, cb, bac, cba, acb)$	$bacbacbac\|ba$	4	

Die Ausgabe ist somit $(2, 1, 3, 4, 6, 5, 4)$. □

Für die Dekompression konstruiert man das Wörterbuch aus der Indexfolge (dem LZW-komprimierten Text). An den dekomprimierten Text wird das durch den Index bestimmte Wort aus dem Wörterbuch angehängt. Der nächste Eintrag im Wörterbuch ergibt sich aus dem aktuellen Wort und dem ersten Buchstaben des durch den folgenden Index bezeichneten Wortes.

Algorithmus 2.7 LZW-Dekompression

Eingabe: LZW-komprimierter Text Z, $|Z| = k$, Alphabet $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$
Ausgabe: dekomprimierter Text $T \in \Sigma^*$

(1) $D \leftarrow \{1 \rightarrow a_1, 2 \rightarrow a_2, \ldots, \sigma \rightarrow a_\sigma\}$; $d \leftarrow \sigma + 1$;
(2) $T \leftarrow \varepsilon$;
(3) **for** $i \leftarrow 1$ **to** $k - 1$
(4) $\quad w \leftarrow$ Wort mit Index $Z[i]$;
(5) $\quad T \leftarrow T \cdot w$;
(6) $\quad a \leftarrow$ erstes Symbol von Wort mit Index $Z[i + 1]$;
(7) $\quad D \leftarrow D \cup \{d \rightarrow wa\}$; $d \leftarrow d + 1$;
(8) $T \leftarrow T \cdot$ Wort mit Index $Z[k]$;
(9) **return** T;

Beispiel 2.9 Es seien $\Sigma = \{a, b, c\}$ und $Z = (2, 1, 3, 4, 6, 5, 4)$. Dann ergibt sich folgender Ablauf der Dekompression. Der Index eines Wortes im Wörterbuch ergibt sich aus seiner Position. Die Position im komprimierten Text wird durch den senkrechten Strich angezeigt.

Wörterbuch	komprimierter Text	Ausgabe	Neuer Eintrag
(a, b, c)	$(\|2, 1, 3, 4, 6, 5, 4)$	b	ba
(a, b, c, ba)	$(2\|1, 3, 4, 6, 5, 4)$	a	ac
(a, b, c, ba, ac)	$(2, 1\|3, 4, 6, 5, 4)$	c	cb
(a, b, c, ba, ac, cb)	$(2, 1, 3\|4, 6, 5, 4)$	ba	bac
$(a, b, c, ba, ac, cb, bac)$	$(2, 1, 3, 4\|6, 5, 4)$	cb	cba
$(a, b, c, ba, ac, cb, bac, cba)$	$(2, 1, 3, 4, 6\|5, 4)$	ac	acb
$(a, b, c, ba, ac, cb, bac, cba, acb)$	$(2, 1, 3, 4, 6, 5\|4)$	ba	

Die Ausgabe ist somit *bacbacbacba*. □

In tatsächlichen Implementierungen läuft einiges anders. So hat man z.B. kein unbeschränkt großes Wörterbuch, sondern eine feste Beschränkung auf üblicherweise $2^{12} = 4096$ Einträge. Davon sind die ersten $2^8 = 256$ für die Buchstaben des Alphabetes – des erweiterten ASCII-Codes – reserviert. Ist das Wörterbuch voll, so werden alle Einträge bis auf die ersten 256 und die am häufigsten gefundenen gelöscht. Dazu zählt man für jedes Wort im Wörterbuch die Anzahl seiner entdeckten Vorkommen. Hat außerdem die längste Übereinstimmung des Textes mit dem Wörterbuch die Länge 1, so wird statt des Indexes (12 Bit) der Buchstabe (8 Bit) ausgegeben. Ob die nächste Ausgabe ein Index oder ein Buchstabe ist wird, durch ein *Flag*-Bit signalisiert. Das heißt, man benötigt 9 bzw. 13 Bit für die Ausgabe eines Indexes.

Die LZW-Kompression erzielt recht gute Ergebnisse bei der Kompression von Texten und Bildern. Sie wird z.B. für das Bildformat GIF angewendet.

Kapitel 3

Ähnlichkeit und inexakte Suche

Neben der exakten Suche hat in den letzten Jahren die inexakte Suche bzw. die Suche nach Ähnlichkeiten an Bedeutung gewonnen. Im folgenden seien einige Aufgabenstellungen sowie zugehörige Anwendungen genannt.

1. *globale Ähnlichkeit:* Gesucht ist der "Abstand" zweier Zeichenketten, wobei eventuell die Unterschiede zu zeigen sind.
   Eine klassische Anwendung ist das Unix-Werkzeug `diff`, welches Unterschiede in Dateien anzeigt.

2. *inexakte Suche:* Gegeben sind ein Text T und ein Suchwort P. Gesucht sind die Teilwörter von T, die P hinreichend ähnlich sind.
   Dies ist eine Erweiterung des bislang betrachteten Suchproblems. Anwendungen finden sich sowohl in der Verarbeitung von Texten, die Rechtschreibfehler enthalten können, wie auch bei der Suche in genetischen Datenbanken.

3. *lokale Ähnlichkeit:* In zwei insgesamt sehr unterschiedlichen Zeichenketten sind Regionen mit hoher Ähnlichkeit gesucht.
   Anwendungen findet man vor allem in molekularbiologischen Datenbanken. Funktionale Ähnlichkeit unterschiedlicher Moleküle wird auf solche lokalen Sequenzähnlichkeiten zurückgeführt. Häufig wird das Problem der lokalen Ähnlichkeit auch für mehr als zwei Zeichenketten untersucht.

Bekanntestes Beispiel einer Abstandsfunktion für Zeichenketten beliebiger Länge ist der *Levenshtein-Abstand*. Er misst die minimale Anzahl der elementaren Operationen

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

die man benötigt, um eine Zeichenkette in eine andere zu überführen.

Um Ähnlichkeiten bzw. Unterschiede zweier Zeichenketten darzustellen, verwendet man so genannte *Alignments* (deutsch: Ausrichtungen). Im folgenden seien Σ ein Alphabet und $\_ \notin \Sigma$ ein Lückenzeichen.

Definition 3.1 *Es seien $S_1, S_2 \in \Sigma^*$. Ein* Alignment *von (S_1, S_2) ist ein Paar (S_1', S_2') mit $S_1', S_2' \in (\Sigma \cup \{\_\})^*$, wobei*

- S_1' *bzw.* S_2' *aus* S_1 *bzw.* S_2 *durch Einfügen von Lückenzeichen* _ *entsteht,*
- $|S_1'| = |S_2'|$ *gilt.*

Beispiel 3.1 Es seien $S_1 =$ `tempel` und $S_2 =$ `treppe`. Es gibt u.a. folgende Alignments:

```
t r e p p e _        t r e _ p p e _
t _ e m p e l        t _ e m p _ e l
```

□

Man kann ein Alignment als Darstellung einer Umwandlung mittels der oben genannten elementaren Operationen interpretieren. Stehen sich zwei unterschiedliche Zeichen aus Σ gegenüber, so wurde eine Ersetzung durchgeführt; ein Zeichen aus Σ über einem _ entspricht einer Löschung, ein _ über einem Zeichen aus Σ entspricht einer Einfügung. Im Beispiel entspricht das erste Alignment einer Umwandlung in 3 Schritten, das zweite Alignment einer Umwandlung in 4 Schritten. Das erste Alignment ist bezüglich der Anzahl der notwendigen Operationen besser als das zweite. (Tatsächlich ist es das optimale und hat als Bewertung den Levenshtein-Abstand 3.) Allgemeiner definieren wir Bewertungen von Alignments wie folgt.

Definition 3.2 *Es sei* $d : (\Sigma \cup \{\_\})^2 \to \mathbb{Z}$ *eine Bewertungsfunktion. Die* Bewertung $d(S_1', S_2')$ *eines Alignments* (S_1', S_2') *mit* $|S_1'| = |S_2'| = t$ *ist*

$$d(S_1', S_2') := \sum_{i=1}^{t} d(S_1'[i], S_2'[i]).$$

Im allgemeinen wird d eine symmetrische Funktion sein und $d(\_, \_) = 0$ gelten. Die Interpretation von d hängt davon ab, ob man Unterschiede oder Ähnlichkeiten sucht. Um den Levenshtein-Abstand zu bestimmen, benutzt man z.B. die Bewertungsfunktion d mit $d(x, x) = 0$ und $d(x, y) = 1$ für $x \neq y$ und sucht ein Alignment mit minimaler Bewertung.

Die Frage nach der globalen Ähnlichkeit kann man also wie folgt formulieren: Finde für zwei gegebene Wörter ein optimales Alignment. Außerdem betrachten wir folgende Varianten.

- Die Bewertungsfunktion wird so gewählt, dass die Bewertung eines Paares mit der Ähnlichkeit wächst. Gesucht ist dann das Alignment mit *maximaler* Bewertung. Üblich sind solche Ähnlichkeitsbewertungen bei Proteinen, wo die Ähnlichkeiten von Aminosäuren in deiner Tabelle (z.B. PAM, BLOSUM) angegeben werden.
- Es ist nur zu entscheiden, ob die Bewertung des optimalen Alignments unter einer gegebenen Schranke k liegt.
- Das Einfügen bzw. Löschen von zusammenhängenden Blöcken wird als *eine* Operation angesehen und gesondert bewertet (Alignment mit Lücken (*gaps*)).

Die Probleme der inexakten Suche sowie der lokalen Ähnlichkeit können ähnlich formuliert und mit ähnlichen Methoden wie das globale Alignment-Problem gelöst werden.

Dieses Kapitel ist wie folgt aufgebaut. Abschnitt 3.1 beschreibt den Standardalgorithmus zur Lösung des Alignment-Problems mittels dynamischer Programmierung. In den Abschnitten 3.2 bis 3.5 werden Varianten und Verbesserungen dieses Grundalgorithmus untersucht. Speziell für die inexakte Suche wurden weitere Algorithmen entworfen, die in der Praxis erheblich effizienter als das Standardverfahren sind. Diese Algorithmen werden im Abschnitt 3.6 gesondert besprochen. Schließlich geben wir in Abschnitt 3.7 eine Einführung in multiple Alignments (von mehr als zwei Zeichenketten).

3.1 Lösung durch dynamische Programmierung

Zunächst wollen wir eine effiziente Lösung des Grundproblems vorstellen. Dieses kann mit dynamischer Programmierung in quadratischer Zeit gelöst werden. Das Wesen der dynamischen Programmierung ist, dass ein Problem gelöst wird, indem Lösungen von Teilproblemen benutzt werden. Bestimmte triviale Teilprobleme werden direkt gelöst und dienen als Rekursionsbasis. Anschließend werden immer größere Teilprobleme unter Verwendung der bereits erhaltenen Teillösungen gelöst. In unserem konkreten Problem bestimmen wir die Bewertungen der minimalen Alignments der Präfixe von S_1 mit den Präfixen von S_2 durch dynamische Programmierung.

Im folgenden gelte $|S_1| = m, |S_2| = n$. Für $0 \leq i \leq m, 0 \leq j \leq n$ sei $D_{i,j}(S_1, S_2)$ die Bewertung des minimalen Alignments der Präfixe $S_1[1 \ldots i]$ bzw. $S_2[1 \ldots j]$. Wie üblich, schreiben wir einfach $D_{i,j}$, wenn S_1 und S_2 aus dem Kontext klar sind. Um den Wert $D(S_1, S_2)$ des optimalen Alignments von (S_1, S_2) zu finden, berechnen wir alle Werte $D_{i,j}$ und erhalten $D(S_1, S_2) = D_{m,n}$.

Satz 3.1 *Es sei $d : (\Sigma \cup \{\_\})^2 \to \mathbb{Z}$ eine Bewertungsfunktion mit $d(\_, \_) = 0$. Dann gilt:*

1. $D_{0,0} = 0$,
2. $D_{i,0} = D_{i-1,0} + d(S_1[i], \_)$ *für* $0 < i \leq m$,
3. $D_{0,j} = D_{0,j-1} + d(\_, S_2[j])$ *für* $0 < j \leq n$,
4. $D_{i,j} = \min\{D_{i-1,j-1} + d(S_1[i], S_2[j]), D_{i-1,j} + d(S_1[i], \_), D_{i,j-1} + d(\_, S_2[j])\}$ *für* $0 < i \leq m, 0 < j \leq n$.

Bemerkung: Die Voraussetzung bedeutet $d(\_, \_) = 0$, dass man nur Alignments betrachten muss, die keine Lückenzeichen an den gleichen Stellen in beiden Wörtern enthalten. Dass Alignments mit der Gegenüberstellung $(\_, \_)$ nicht einfach verboten werden liegt daran, dass diese Gegenüberstellungen in *multiplen Alignments* (Abschnitt 3.7) vorkommen und dort mit 0 bewertet werden.

Beweis. Die Behauptungen (1),(2) und (3) sind offenbar korrekt, da es zwischen einem Wort α und dem leeren Wort genau ein Alignment ohne Alignment-Paare $(\_, \_)$ gibt. Dieses besitzt den im Satz angegebenen Wert.

Um (4) zu zeigen, betrachten wir ein Alignment (α, β) von $(S_1[1 \ldots i], S_2[1 \ldots j])$. Es seien $x = S_1[i], y = S_2[j]$. Dann hat (α, β) eine der folgenden drei Formen:

(a) $\alpha = \alpha_1 x$, $\beta = \beta_1 y$
Dann ist (α_1, β_1) ein Alignment von $(S_1[1 \ldots i-1], S_2[1 \ldots j-1])$.

(b) $\alpha = \alpha_2 x$, $\beta = \beta_2 \_$
Dann ist (α_2, β_2) ein Alignment von $(S_1[1 \ldots i-1], S_2[1 \ldots j])$.

(c) $\alpha = \alpha_3 \_$, $\beta = \beta_3 y$
Dann ist (α_3, β_3) ein Alignment von $(S_1[1 \ldots i], S_2[1 \ldots j-1])$.

Ist (α, β) zudem ein optimales Alignment von $(S_1[1 \ldots i], S_2[1 \ldots j])$, so gilt in den einzelnen Fällen:

(a) $D_{i,j} = d(\alpha, \beta) = d(\alpha_1, \beta_1) + d(x, y) \geq D_{i-1,j-1} + d(x, y)$,

(b) $D_{i,j} = d(\alpha, \beta) = d(\alpha_2, \beta_2) + d(x, \_) \geq D_{i-1,j} + d(x, \_)$,

(c) $D_{i,j} = d(\alpha, \beta) = d(\alpha_3, \beta_3) + d(\_, y) \geq D_{i,j-1} + d(\_, y)$.

In jedem Fall haben wir $D_{i,j} \geq \min\{D_{i-1,j-1} + d(x,y), D_{i-1,j} + d(x, \_), D_{i,j-1} + d(\_, y)\}$.

Seien umgekehrt (α_1, β_1), (α_2, β_2) bzw. (α_3, β_3) die optimalen Alignments von $(S_1[1 \ldots i-1], S_2[1 \ldots j-1])$, $(S_1[1 \ldots i-1], S_2[1 \ldots j])$, $(S_1[1 \ldots i], S_2[1 \ldots j-1])$. Dann gilt

$$\begin{aligned} D_{i,j} &\leq d(\alpha_1 x, \beta_1 y) = d(\alpha_1, \beta_1) + d(x,y) = D_{i-1,j-1} + d(x,y) \\ D_{i,j} &\leq d(\alpha_2 x, \beta_2 \_) = d(\alpha_2, \beta_2) + d(x, \_) = D_{i-1,j} + d(x, \_) \\ D_{i,j} &\leq d(\alpha_3 \_, \beta_3 y) = d(\alpha_3, \beta_3) + d(\_, y) = D_{i,j-1} + d(\_, y) \end{aligned}$$

und damit $D_{i,j} \leq \min\{D_{i-1,j-1} + d(x,y), D_{i-1,j} + d(x, \_), D_{i,j-1} + d(\_, y)\}$. □

Aus dem Beweis von Satz 3.1 ergibt sich unmittelbar ein Algorithmus zur Berechnung der Bewertung des optimalen Alignments in quadratischer Zeit.

Algorithmus 3.1 Ermittlung des minimalen Alignment-Wertes

Eingabe: Wörter S_1, S_2, $|S_1| = m, |S_2| = n$
Ausgabe: Bewertung des minimalen Alignments von S_1 und S_2

```
(1)   D_{0,0} ← 0;
(2)   for i ← 1 to m
(3)       D_{i,0} ← D_{i-1,0} + d(S_1[i], _);
(4)   for j ← 1 to n
(5)       D_{0,j} ← D_{0,j-1} + d(_, S_2[j]);
(6)   for i ← 1 to m
(7)       for j ← 1 to n
(8)           D_{i,j} ← min{D_{i-1,j-1} + d(S_1[i], S_2[j]), D_{i-1,j} + d(S_1[i], _), D_{i,j-1} +
              d(_, S_2[j])};
(9)   return D_{m,n};
```

Der Algorithmus kann am besten mittels einer Tabelle veranschaulicht werden. Dabei wird der Wert $D_{i,j}$ in der Zeile i und der Spalte j eingetragen. Die Werte werden zeilenweise und in den Zeilen von links nach rechts bestimmt.

Beispiel 3.2 Für $S_1 =$ `tempel` und $S_2 =$ `treppe` ergibt sich bei Verwendung des Levenshtein-Abstandes die folgende Alignment-Tabelle

		t	r	e	p	p	e
	0	1	2	3	4	5	6
t	1	0	1	2	3	4	5
e	2	1	1	1	2	3	4
m	3	2	2	2	2	3	4
p	4	3	3	3	2	2	3
e	5	4	4	3	3	3	2
l	6	5	5	4	4	4	3

und damit die optimale Bewertung 3. □

Aus dem Beweis lässt sich außerdem ein rekursiver Algorithmus zur Bestimmung des optimalen Alignments ableiten. Dazu muss man nur ermitteln, welches der 3 möglichen optimalen Alignments von $(S_1[1 \ldots m-1], S_2[1 \ldots n-1])$, $(S_1[1 \ldots m-1], S_2[1 \ldots n])$, $(S_1[1 \ldots m], S_2[1 \ldots n-1])$, zu einem optimalen Alignment von (S_1, S_2) erweitert werden kann. Dies lässt sich aus den Werten $D_{m-1,n-1}$, $D_{m-1,n}$, $D_{m,n-1}$ sowie den Buchstaben $S_1[m]$, $S_2[n]$ bestimmen. Damit ergeben sich das letzte Alignment-Paar sowie die Präfixe, für die das "Vorgänger-Alignment" zu bestimmen ist. Sollten mehrere Vorgänger in Frage kommen, so darf man beliebig wählen.

Algorithmus 3.2 Ermittlung eines optimalen Alignments

Eingabe: Wörter S_1, S_2, $|S_1| = m, |S_2| = n$, Alignment-Tabelle $(D_{i,j})$
Ausgabe: ein optimales Alignment A von S_1 und S_2

(1) $A \leftarrow \varepsilon$; $(i,j) \leftarrow (m,n)$;
(2) **while** $(i,j) \neq (0,0)$
(3) $V \leftarrow \emptyset$;
(4) **if** $D_{i,j} = D_{i-1,j-1} + d(S_1[i], S_2[j])$ **then** $V \leftarrow V \cup \{\nwarrow\}$;
(5) **if** $D_{i,j} = D_{i,j-1} + d(\_, S_2[j])$ **then** $V \leftarrow V \cup \{\leftarrow\}$;
(6) **if** $D_{i,j} = D_{i-1,j} + d(S_1[i], \_)$ **then** $V \leftarrow V \cup \{\uparrow\}$;
(7) $p \leftarrow$ ein Element aus V;
(8) **if** $p = \nwarrow$ **then** $A \leftarrow (S_1[i], S_2[j]) \cdot A$; $i \leftarrow i-1$; $j \leftarrow j-1$;
(9) **if** $p = \leftarrow$ **then** $A \leftarrow (\_, S_2[j]) \cdot A$; $j \leftarrow j-1$;
(10) **if** $p = \uparrow$ **then** $A \leftarrow (S_1[i], \_) \cdot A$; $i \leftarrow i-1$;
(11) **return** A;

Auch Algorithmus 3.2 lässt sich sehr gut anhand der Tabelle der $D_{i,j}$-Werte veranschaulichen. Man startet in der Tabelle rechts unten, also im Feld (m,n). Das Vorgängerfeld sowie das letzte Alignment-Paar werden durch die Richtung des Pfeils p aus dem Algorithmus bestimmt.

Richtung	Vorgänger	letztes Alignment-Paar
$\nwarrow$	$(m-1, n-1)$	$(S_1[m], S_2[n])$
$\leftarrow$	$(m, n-1)$	$(\_, S_2[n])$
$\uparrow$	$(m-1, n)$	$(S_1[m], \_)$

Dann wird der Algorithmus vom Vorgängerfeld aus fortgesetzt, bis das Feld $(0,0)$ erreicht ist. Das Alignment entspricht somit einem Pfad von (m,n) nach $(0,0)$.

Beispiel 3.3 Für $S_1 =$ `tempel` und $S_2 =$ `treppe` erhält man bei Verwendung des Levenshtein-Abstandes folgenden Alignment-Pfad

		t	r	e	p	p	e
	0	1	2	3	4	5	6
t	1	0	1	2	3	4	5
e	2	1	1	1	2	3	4
m	3	2	2	2	2	3	4
p	4	3	3	3	2	2	3
e	5	4	4	3	3	3	2
l	6	5	5	4	4	4	3

und das (in diesem Fall einzige) optimale Alignment $\begin{matrix} \texttt{t} & \texttt{r} & \texttt{e} & \texttt{p} & \texttt{p} & \texttt{e} & \texttt{\_} \\ \texttt{t} & \texttt{\_} & \texttt{e} & \texttt{m} & \texttt{p} & \texttt{e} & \texttt{l} \end{matrix}$ □

Zusammenfassend können wir feststellen:

Satz 3.2 *Es seien S_1 und S_2 Wörter mit $|S_1| = m$, $|S_2| = n$. Die Algorithmen 3.1 und 3.2 bestimmen ein optimales Alignment von (S_1, S_2) mit einem Aufwand von $O(mn)$.*

Ähnlichkeitsbewertungen

Will man Ähnlichkeit messen d.h. die Alignment-Bewertung maximieren, so braucht man nur in der Rekursionsformel das Minimum durch ein Maximum zu ersetzen. Ein wichtiger Spezialfall ist das Problem der *längsten gemeinsamen Teilfolge (longest common subsequence).*

Definition 3.3 *Ein Wort α der Länge t heißt* Teilfolge *des Wortes S, wenn es eine Indexfolge $1 \leq j_1 < j_2 < \ldots < j_t \leq |S|$ mit $\alpha[i] = S[j_i]$ für alle $1 \leq i \leq t$ gibt.*

Die längste gemeinsame Teilfolge von S_1 und S_2 ist damit ein Wort maximaler Länge, das mit Unterbrechungen sowohl in S_1 und S_2 vorkommt. Die Bestimmung der längsten gemeinsamen Teilfolge ist offenbar das Alignmentproblem für die Bewertungsfunktion d mit

$$d(x,y) = \begin{cases} 1, & \text{falls } x = y \text{ und } x, y \in \Sigma \\ 0, & \text{sonst} \end{cases}$$

und dem Ziel der Maximierung. Die Übereinstimmungen in einem optimalen Alignment liefern eine maximale Teilfolge.

Beispiel 3.4 Für S_1 = `tempel` und S_2 = `treppe` ergibt sich für die längste gemeinsame Teilfolge die folgende Alignment-Tabelle einschließlich den möglichen Alignment-Pfaden:

		t	r	e	p	p	e
	0	0	0	0	0	0	0
t	0	1	1	1	1	1	1
e	0	1	1	2	2	2	2
m	0	1	1	2	2	2	2
p	0	1	1	2	3	3	3
e	0	1	1	2	3	3	4
l	0	1	1	2	3	3	4

Ein optimales Alignment ist z.B.

```
t r e _ p p e _
t _ e m p _ e l
```

mit der längsten Teilfolge `tepe`. □

Inexakte Suche

Das Problem der inexakten Suche nach einem Muster S_1 in einem Text S_2 lässt sich wie folgt formalisieren: Gesucht ist das optimale Alignment von S_1 mit einem Teilwort von S_2.

Dieses Problem kann gelöst werden, indem man durch dynamische Programmierung für $0 \leq i \leq m$, $0 \leq j \leq n$ jeweils den Wert $D'_{i,j}(S_1, S_2)$ des optimalen Alignments von $S_1[1 \ldots i]$ mit einem Suffix von $S_2[1 \ldots j]$ bestimmt.

Lemma 3.3 *Es seien S_1 und S_2 Wörter mit $|S_1| = m$, $|S_2| = n$. Weiter sei d eine Bewertungsfunktion für Alignments, und gesucht sei das minimale Alignment. Dann gilt:*
$D'_{0,0} = 0$, $D'_{0,j} = 0$, $D'_{i,0} = D'_{i-1,0} + d(S_1[i], \_)$,
$D'_{i,j} = \min\{D'_{i-1,j-1} + d(S_1[i], S_2[j]), D'_{i-1,j} + d(S_1[i], \_), D'_{i,j-1} + d(\_, S_2[j])\}$
für $1 \leq j \leq n$, $1 \leq i \leq m$.

Beweis. $D'_{0,j} = 0$ gilt, da $(\varepsilon, \varepsilon)$ das optimale Alignment von ε mit einem Suffix von $S_2[1 \ldots j]$ ist. Die Rekursionsbeziehung wird wie in Satz 3.1 bewiesen. □

Wie beim globalen Alignment lässt sich der Algorithmus am besten in einer Tabelle darstellen. In der untersten Zeile lassen sich die Bewertungen der besten Alignments von S_1 mit Teilwörtern von S_2, die an den entsprechenden Positionen enden, ablesen. Die Alignments können wiederum bestimmt werden, indem man einen Pfad von der untersten in die oberste Zeile konstruiert.

Beispiel 3.5 Gesucht sind die besten inexakten Vorkommen von S_1 = `fische` in S_2 = `fritzefischtefrische` bezüglich des Levenshtein-Abstandes.

		f	r	i	t	z	e	f	i	s	c	h	t	e	f	r	i	s	c	h	e
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f	1	0	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1
i	2	1	1	1	2	2	2	1	0	1	2	2	2	2	1	1	1	2	2	2	2
s	3	2	2	2	2	3	3	2	1	0	1	2	3	3	2	2	2	1	2	3	3
c	4	3	3	3	3	3	4	3	2	1	0	1	2	3	3	3	3	2	1	2	3
h	5	4	4	4	4	4	4	4	3	2	1	0	1	2	3	4	4	3	2	1	2
e	6	5	5	5	5	5	4	5	4	3	2	1	1	1	2	3	4	4	3	2	1

Die besten Treffer mit jeweils der Bewertung 1 enden an den Textpositionen 11, 12, 13 und 20 und liefern die Alignments

```
fische   fische   fisch_e          f_ische
fisch_ , fischt , fischte  und    frische .
```

□

Satz 3.4 *Die optimalen inexakten Vorkommen von S_1 in S_2 (einschließlich der optimalen Alignments) können mit einem Aufwand von $O(mn)$ gefunden werden.*

Lokale Alignments

Die dritte Grundaufgabe besteht darin, Ähnlichkeiten in zwei Zeichenketten S_1 und S_2 zu finden, die im Ganzen große Unterschiede aufweisen. Gesucht sind also Teilwörter von S_1 und S_2 mit hoher Ähnlichkeit. Formal lassen sich diese Bereiche hoher Ähnlichkeit durch lokale Alignments beschreiben.

Definition 3.4 *Ein* lokales Alignment *von* (S_1, S_2) *ist ein Alignment von* (α, β)*, wobei* α *bzw.* β *Infixe von* S_1 *bzw.* S_2 *sind.*

Zur Bewertung lokaler Alignments verwendet man Ähnlichkeitsfunktionen mit dem Optimierungsziel der Maximierung, da man bei Abstandsfunktionen stets das triviale lokale Alignment $(\varepsilon, \varepsilon)$ als bestes lokales Alignment erhalten würde. Es seien also im folgenden $S_1, S_2 \in \Sigma^*$ mit $|S_1| = m$, $|S_2| = n$ und $d : (\Sigma \cup \{\_\})^2 \to \mathbb{Z}$ eine Ähnlichkeitsfunktion. Für $0 \leq i \leq m$, $0 \leq j \leq n$ sei $D''_{i,j}$ die maximale Bewertung eines Alignments eines Suffixes von $S_1[1 \ldots i]$ mit einem Suffix von $S_2[1 \ldots j]$.

Lemma 3.5 *Es seien* S_1 *und* S_2 *Wörter mit* $|S_1| = m$*,* $|S_2| = n$*. Weiter sei* d *eine Bewertungsfunktion für Alignments mit* $d(x, \_) \leq 0$ *und* $d(\_, x) \leq 0$ *für* $x \in \Sigma$*, und gesucht sei das maximale Alignment. Dann gilt:* $D''_{0,0} = D''_{i,0} = D''_{0,j} = 0$*,*
$D''_{i,j} = \max\{D''_{i-1,j-1} + d(S_1[i], S_2[j]), D''_{i-1,j} + d(S_1[i], \_), D''_{i,j-1} + d(\_, S_2[j]), 0\}$*,*
für $1 \leq i \leq m$*,* $1 \leq j \leq n$*.*

Beweis. Die Beziehungen $D''_{0,0} = D''_{i,0} = D''_{0,j} = 0$ folgen, da das optimale Alignment von ε mit einem Teilwort von S_1 bzw. S_2 jeweils $(\varepsilon, \varepsilon)$ ist. Die Rekursionsformel für $D''_{i,j}$ wird ähnlich gezeigt wie in Satz 3.1. Da neben der Erweiterung der drei Vorgänger-Alignments auch das Alignment $(\varepsilon, \varepsilon)$ mit der Bewertung 0 in Frage kommt, ist $D''_{i,j}$ mindestens 0. □

Erneut lässt sich der Algorithmus am besten an einer Tabelle verdeutlichen. Das global beste lokale Alignment endet an einer Position mit maximalem D''-Wert. *Lokal optimale* Alignments, also solche, die weder durch Verkürzen noch Verlängern verbessert werden können, enden an den Positionen (i, j) mit

$$D''_{i,j} \geq \max\{D''_{i-1,j-1}, D''_{i,j-1}, D''_{i-1,j}, D''_{i+1,j+1}, D''_{i,j+1}, D''_{i+1,j}\}.$$

Der Alignment-Pfad führt von der Endposition zu einem Feld mit dem Wert 0. Im allgemeinen ist man jedoch nicht an allen lokal optimalen Alignments interessiert, sondern an lokalen Alignments mit einer gewissen Mindestbewertung.

Beispiel 3.6 Für das Ähnlichkeitsmaß: $d(x, x) = 2$ für $x \in \{a, b, c, d\}$, $d(x, y) = -1$, sonst und $S_1 = caabcacb$, $S_2 = dddadbdddddadabdd$ ergibt sich die folgende Tabelle, wobei die Pfade aller lokal optimalen Alignments mit einer Bewertung von mindestens 3 markiert sind.

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

Die optimalen lokalen Alignments sind $\begin{matrix}\texttt{aab}\\ \texttt{adb}\end{matrix}$, $\begin{matrix}\texttt{acb}\\ \texttt{adb}\end{matrix}$, $\begin{matrix}\texttt{a\_ab}\\ \texttt{adab}\end{matrix}$ und $\begin{matrix}\texttt{acb}\\ \texttt{a\_b}\end{matrix}$. □

3.2 Alignments mit beschränkter Fehlerzahl

Häufig ist man nur an globalen Alignments bzw. an inexakten Treffern interessiert, deren Levenshtein-Bewertung durch eine Zahl k beschränkt ist. Dann ist es nicht nötig, die gesamte Alignment-Tabelle auszurechnen, in der viele Zahlen größer als k sind. Man kann die Grundalgorithmen so modifizieren, dass für globale Alignments die Laufzeit $O(kn)$ beträgt und für die inexakte Suche die mittlere Laufzeit deutlich verringert wird.

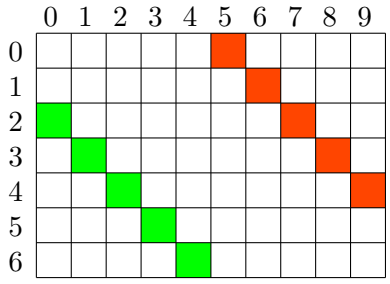
Globales Alignment

Gegeben sind eine natürliche Zahl k sowie zwei Zeichenketten S_1 und S_2 mit $|S_1| = m$, $|S_2| = n$. Es gelte $n - m = \delta \geq 0$. Es ist zu entscheiden, ob der Levenshtein-Abstand von S_1 und S_2 durch k beschränkt ist. Im positiven Fall ist ein optimales Alignment zu konstruieren.

Definition 3.5 *Für eine Matrix mit den Zeilen* $0, 1, \ldots, m$ *und den Spalten* $0, 1, \ldots, n$ *ist die* d*-te Diagonale* $Diag_d$ $(-m \leq d \leq n)$ *definiert als*

$$Diag_d = \{(i, j) : 0 \leq i \leq m \wedge 0 \leq j \leq n \wedge j - i = d\}.$$

Beispiel 3.7 In der folgenden Matrix sind die Diagonalen $Diag_{-2}$ und $Diag_5$ dargestellt.



□

Lemma 3.6 *Besitzen* S_1 *und* S_2 *einen Levenshtein-Abstand von höchstens* k*, so befindet sich der Pfad für das optimale Alignment innerhalb des Bereiches der Diagonalen* $Diag_d$ *mit* $-\lfloor (k - \delta)/2 \rfloor \leq d \leq \lfloor (k + \delta)/2 \rfloor$.

Beweis. Befindet sich ein Knoten (i, j) des Alignment-Graphen in der Diagonalen $d = (j - i)$, so existieren von diesem Knoten Kanten zu den Diagonalen $d-1$, d, $d+1$. Die Kanten zu den Diagonalen $d-1$ bzw. $d+1$ haben ein Gewicht von 1. Der Startpunkt des Alignment-Pfades ist $(0, 0)$, also auf der Diagonalen 0; der Endpunkt des Alignment-Pfades ist (m, n), also auf der Diagonalen δ.

Seien nun $-l \leq 0$ bzw. $r \geq \delta$ die Diagonale mit dem niedrigsten bzw. höchsten Index im Alignment-Pfad. Der Weg von $(0, 0)$ zur Diagonalen $-l$ kostet mindestens l; der von der Diagonalen l nach (m, n) kostet mindestens $l + \delta$. Die Gesamtkosten sind also mindestens $2l + \delta$. Wenn die Gesamtkosten höchstens k betragen, so folgt $l \leq -\lfloor (k - \delta)/2 \rfloor$. Analog zeigt man $r \leq \lfloor (k + \delta)/2 \rfloor$. □

Satz 3.7 *Es kann mit einem Aufwand von $O(km)$ ermittelt werden, ob der Levenshtein-Abstand von S_1 und S_2 durch k beschränkt ist und im positiven Fall ein optimales Alignment bestimmt werden.*

Beweis. Man kann den Grundalgorithmus abwandeln, wobei nur nach Alignments gesucht wird, deren Pfade im Bereich der Diagonalen $Diag_d$ mit $-\lfloor (k - \delta)/2 \rfloor \leq d \leq \lfloor (k + \delta)/2 \rfloor$ verlaufen (die konkrete Abwandlung bleibt als Übungsaufgabe). In jeder Zeile sind höchstens $(k + 1)$ Werte zu bestimmen, was einen Gesamtaufwand von $O(km)$ ergibt. □

Beispiel 3.8 Für $S_1 =$ `tempel` und $S_2 =$ `treppe` gilt $\delta = 0$. Damit braucht man für $k = 1$ nur die Diagonale 0 und für $k = 3$ nur die Diagonalen von -1 bis 1 zu betrachten. Es ergeben sich folgende Tabellen.

		t	r	e	p	p	e
	0						
t		0					
e			1				
m				2			
p					2		
e						3	
l							4

		t	r	e	p	p	e
	0	1					
t	1	0	1				
e		1	1	1			
m			2	2	2		
p				3	2	2	
e					3	3	2
l						4	3

Man beachte, dass in der ersten Tabelle im Tabellenfeld rechts unten eine 4 steht. Dies ist der Wert des besten Alignments, das nur die Diagonale 0 benutzt. Man kann aus der ersten Tabelle also nur folgern, dass der Levenshtein-Abstand größer als 1 ist. Aus der zweiten Tabelle folgt, dass der Levenshtein-Abstand 3 beträgt, und der optimale Alignment-Pfad wird wie gehabt konstruiert. □

Man kann den eben genannten Algorithmus auch benutzen, wenn keine Schranke gegeben ist, um einen $O(k^*m)$-Algorithmus zu erhalten, wobei k^* der (im Voraus unbekannte) Levenshtein-Abstand ist. Das heißt, für kleine Abstände erreicht man eine bessere Laufzeit als mit dem ursprünglichen Algorithmus, während bei hohen Abständen die Laufzeit asymptotisch gleich bleibt.

Satz 3.8 *Das optimale Alignment von (S_1, S_2) kann mit einem Aufwand von $O(k^*m)$ mit $k^* = D(S_1, S_2)$ bestimmt werden.*

Beweis. Man setzt zunächst $k \leftarrow 1$. Solange es nicht gelingt, ein optimales Alignment mit einer Bewertung von höchstens k zu bestimmen, verdoppelt man k. Die Anzahl der zu bestimmenden Werte ist beschränkt durch

$$\sum_{i=0}^{\lceil \log k^* \rceil} 2^i + 1(m+1) = O(k^* m).$$

□

Inexakte Suche

Für die inexakte Suche nach dem Wort S_1 im Text S_2 lässt sich der Basis-Algorithmus ebenfalls verbessern. Man berechnet die Einträge der Tabelle der $D'_{i,j}$-Werte nur an den Positionen, an denen aufgrund der Werte der Vorgänger ein Eintrag von höchstens k vorliegen könnte. Von Nutzen ist dabei folgender Satz über die $D'_{i,j}$-Werte unter Berücksichtigung des Levenshtein-Abstandes.

Lemma 3.9 *Es seien S_1 und S_2 Wörter mit $|S_1| = m$, $|S_2| = n$. Mit dem Levenshtein-Abstand als Abstandsmaß gilt*

1. $D'_{i,j} \geq D'_{i,j-1} - 1$ *für* $0 \leq i \leq m$, $1 \leq j \leq n$,
2. $D'_{i,j} \geq D'_{i-1,j} - 1$ *für* $1 \leq i \leq m$, $0 \leq j \leq n$,
3. $D'_{i,j} \geq D'_{i-1,j-1}$ *für* $1 \leq i \leq m$, $1 \leq j \leq n$.

Beweis. Wir beweisen alle drei Aussagen simultan durch Induktion über i und j.
Induktionsanfang: Wegen $D'_{0,j} = 0$ für $0 \leq j \leq n$ gilt Aussage 1 für $i = 0$. Wegen $D'_{i,0} = i$ für $0 \leq i \leq n$ gilt Aussage 2 für $j = 0$.
Induktionsvoraussetzung: Es seien für i, j mit $1 \leq i \leq m$ und $1 \leq j \leq n$ bereits $D'_{i,j-1} \geq D'_{i-1,j-1} - 1$ und $D'_{i,j-1} \geq D'_{i-1,j-1} - 1$ gezeigt.
Induktionsbehauptung: Dann gelten $D'_{i,j} \geq D'_{i,j-1} - 1$, $D'_{i,j} \geq D'_{i-1,j} - 1$, $D'_{i,j} \geq D'_{i-1,j-1}$.
Induktionsbeweis: Nach Definition gilt $D'_{i,j} \geq \min\{D'_{i-1,j-1}, D'_{i,j-1} + 1, D'_{i-1,j} + 1\}$.
Aus der Induktionsvoraussetzung folgen $D'_{i,j-1} + 1 \geq D'_{i-1,j-1}$ und $D'_{i-1,j} + 1 \geq D'_{i-1,j-1}$, woraus sich $D'_{i,j} \geq D'_{i-1,j-1}$ ergibt.
Nach Definition gilt weiterhin $D'_{i,j-1} \leq D'_{i-1,j-1} + 1$ und $D'_{i-1,j} \leq D'_{i-1,j-1} + 1$. Mit $D'_{i,j} \geq D'_{i-1,j-1}$ folgt daraus $D'_{i,j} \geq D'_{i,j-1} - 1$ und $D'_{i,j} \geq D'_{i-1,j} - 1$. □

Die eingeschränkte Alignment-Tabelle berechnet man am besten spaltenweise. Man stellt für die Spalte j eine Liste L_j auf, die alle Zahlen i mit $D'_{i,j} \leq k$ enthält. Wegen $D'_{i+1,j+1} \geq D'_{i,j}$ können in der Spalte $j+1$ nur die Werte an den Stellen 0 bzw. $i+1$ mit $i \in L_j$ nicht größer als k sein und müssen berechnet werden. Im schlechtesten Fall müssen nach wie vor alle Einträge ermittelt werden, d.h. die Laufzeit beträgt nach wie vor $\Theta(mn)$. Die mittlere Laufzeit ist jedoch in der Größenordnung $\Theta(kn)$. Erheblich vereinfacht wird der Algorithmus, wenn man für jede Spalte j den Wert $i_{\max}(j) = \max\{i : D'_{i,j} \leq k\}$ ermittelt und in der Spalte j' die D'-Werte bis zur Zeile $i_{\max}(j)$ bestimmt.

Beispiel 3.9 Für die Suche nach $S_1 =$ `fische` in $S_2 =$ `fritzefischtefrische` mit einem Levenshtein-Abstand von höchstens 1 werden folgende Werte der Alignment-Tabelle bestimmt.

		f	r	i	t	z	e	f	i	s	c	h	t	e	f	r	i	s	c	h	e
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f	1	0	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1
i		1	1	1	2	2	2	1	0	1	2	2	2	2	1	1	1	2	2	2	2
s			2	2	2				1	0	1					2	2	1			
c										1	0	1							1		
h											1	0	1							1	
e												1	1	1							1

□

Weitere Algorithmen für die inexakte Suche bei vorgegebener Fehlerschranke werden in Abschnitt 3.6 besprochen.

3.3 Alignments mit Lücken (Gaps)

Bisher ergab sich die Bewertung eines Alignments, indem die Bewertungen der gegenüberliegenden Zeichen addiert wurden. Diese Definition ermöglichte einfache Rekursionsbeziehungen und damit die effiziente Lösung durch dynamische Programmierung. Realistisch sind solche Bewertungen allerdings nicht unbedingt. Insbesondere sollte das Einfügen bzw. Löschen eines Wortes der Länge m als *eine* Operation angesehen werden und geringer bestraft werden als das Einfügen bzw. Löschen von m einzelnen Zeichen. Das Einfügen bzw. Löschen eines Wortes entspricht einer Folge von Lückenzeichen auf einer Seite des Alignments.

Definition 3.6 *Es sei (S_1', S_2') ein Alignment. Eine* Lücke (gap) *ist eine maximale Folge von Lückenzeichen in S_1' oder S_2'.*

Jedes Alignment (S_1', S_2') ist eindeutig zerlegbar als

$$(S_1', S_2') = (\alpha_1, \beta_1) \cdots (\alpha_k, \beta_k),$$

wobei, für $1 \leq i \leq k$, α_i und β_i entweder kein Lückenzeichen enthalten oder eins von beiden eine Lücke ist.

Die Abstandsfunktion d wird nur noch auf $\Sigma \times \Sigma$ definiert; außerdem gibt es eine *Gap*-Funktion $g : \mathbb{N}_+ \to \mathbb{R}_+$, die eine Lücke entsprechend ihrer Länge bewertet. Die Bewertung eines Alignments ergibt sich folgendermaßen:

$$\begin{aligned} d(\alpha, \beta) &:= \sum_{i=1}^{|\alpha|} d(\alpha[i], \beta[i]), \text{ falls } \alpha, \beta \in \Sigma^* \\ d(\alpha, \beta) &:= g(|\alpha|), \text{ falls } \alpha \text{ oder } \beta \text{ Lücke} \\ d(S_1', S_2') &:= \sum_{j=1}^{k} d(\alpha_j, \beta_j), \text{ bei obiger Zerlegung} \end{aligned}$$

Beispiel 3.10 Für die Gap-Funktion $g(n) = \sqrt{n}$ und die Abstandsfunktion wie beim Levenshtein-Abstand ergeben sich folgende Bewertungen für zwei Alignments von *abaabb* und *aaba*:

$$\begin{pmatrix} a & b & a & a & b & b \\ a & \_ & a & \_ & b & a \end{pmatrix} \text{ Zerlegung: } \begin{pmatrix} a \\ a \end{pmatrix} \begin{pmatrix} b \\ \_ \end{pmatrix} \begin{pmatrix} a \\ a \end{pmatrix} \begin{pmatrix} a \\ \_ \end{pmatrix} \begin{pmatrix} b & b \\ b & a \end{pmatrix} \text{ Bewertung: } 3$$

$$\begin{pmatrix} a & b & a & a & b & b \\ a & \_ & \_ & a & b & a \end{pmatrix} \text{ Zerlegung: } \begin{pmatrix} a \\ a \end{pmatrix} \begin{pmatrix} b & a \\ \_ & \_ \end{pmatrix} \begin{pmatrix} a & b & b \\ a & b & a \end{pmatrix} \text{ Bewertung: } 1+\sqrt{2}$$

Damit hat das zweite Alignment eine bessere Bewertung als das erste, während die Levenshtein-Bewertung für beide Alignments gleich ist. □

Für die Bestimmung eines optimalen Alignments kann man wiederum dynamische Programmierung anwenden. Für beliebige Gap-Funktionen ist eine kubische Laufzeit nötig, während die Laufzeit im Falle affiner Gap-Funktionen quadratisch bleibt. Wir untersuchen die Fälle im einzelnen.

Allgemeiner Fall

Die Länge eines Gaps am Ende eines Alignments (α, β) von $(S_1[1 \ldots i], S_2[1 \ldots j])$ ist begrenzt durch i, falls die Lücke am Ende von β auftritt, und durch j, falls sich die Lücke am Ende von α befindet.

Für $0 \leq i \leq m$, $0 \leq j \leq n$ und $-j \leq h \leq i$ sei $D_{i,j,h}$ der Wert eines optimalen Alignments von $(S_1[1 \ldots i], S_2[1 \ldots j])$, das mit einem Gap der Länge $|h|$ endet, wobei sich das Gap in S_2 bzw. in S_1 befindet, falls $h > 0$ bzw. $h < 0$ gilt. Weiterhin seien:

$$\begin{aligned} D_{i,j,+} &:= \min\{D_{i,j,h} : 1 \leq h \leq i\}, \\ D_{i,j,-} &:= \min\{D_{i,j,h} : -j \leq h \leq -1\}. \end{aligned}$$

Für den Wert des optimalen Alignments von $(S_1[1 \ldots i], S_2[1 \ldots j])$ gilt offenbar: $D_{i,j} = \min\{D_{i,j,+}, D_{i,j,-}, D_{i,j,0}\}$.

Satz 3.10 *Im Falle einer beliebigen Gap-Funktion kann das optimale Alignment mit einem Aufwand von $O(mn(m+n))$ gefunden werden.*

Beweis. Wir geben hier nur die Rekursionsformeln an und lassen den Korrektheitsbeweis als Übungsaufgabe. Für $0 \leq i \leq m$ und $0 \leq j \leq n$ gilt:

$$D_{0,j,h} = \begin{cases} g(j), & \text{falls } h = -j \\ \infty, & \text{sonst} \end{cases} \qquad D_{i,0,h} = \begin{cases} g(j), & \text{falls } h = i \\ \infty, & \text{sonst} \end{cases}$$

Für $1 \leq i \leq m$, $1 \leq j \leq n$ ergibt sich:

$$\begin{aligned} D_{i,j,0} &= D_{i-1,j-1} + d(S_1[i], S_2[j]) \\ D_{i,j,1} &= \min\{D_{i-1,j,-}, D_{i-1,j,0}\} + g(1) \\ D_{i,j,-1} &= \min\{D_{i-1,j,+}, D_{i-1,j,0}\} + g(1) \\ D_{i,j,h} &= D_{i-1,j,h-1} + g(h) - g(h-1), 2 \leq h \leq i \\ D_{i,j,h} &= D_{i,j-1,h+1} + g(|h|) - g(|h|-1), -j \leq h \leq -2 \end{aligned}$$

Die Konstruktion eines optimalen Alignments erfolgt wieder von der Position (m, n) aus. Man ermittelt einen Index h für den $D_{m,n} = D_{m,n,h}$ gilt. Ist $h = 0$, so endet das Alignment mit dem Paar $(S_1[m], S_2[n])$. Im Falle von $h > 0$ endet das Alignment mit einem Gap der Länge h in S_2. Für $h < 0$ endet das Alignment mit einem Gap der Länge $|h|$ in S_1. Damit ist in jedem Fall das Endstück des Alignments bestimmt, und man kann rekursiv das Alignment fortsetzen. □

Affine Gap-Funktionen

Wir betrachten jetzt eine Gap-Funktion der Form $g(n) = a(n-1) + b$ mit $a, b \leq 0$. Derartige Bewertungen von Lücken sind plausibel: es gibt eine Strafe b für das Eröffnen einer Lücke und eine Strafe a für das Vergrößern einer Lücke um 1. Für affine Gap-Funktionen lässt sich das optimale Alignment mit quadratischem Aufwand finden, da sich die Berechnung der Werte $V_{i,j,+}, V_{i,j,-}, V_{i,j,0}$ vereinfacht.

Satz 3.11 *Im Falle einer affinen Gap-Funktion kann das optimale Alignment mit einem Aufwand von $O(mn)$ gefunden werden.*

Beweis. Wegen $g(h+1) - g(h) = a$ für $h \geq 1$ ergeben sich folgende Beziehungen für $1 \leq i \leq m, 1 \leq j \leq n$:

$$\begin{aligned}
D_{0,0,0} &= 0, D_{0,0,-} = D_{0,0,+} = \infty, \\
D_{0,j,-} &= a(j-1) + b, D_{0,j,0} = D_{0,j,+} = \infty, \\
D_{i,0,+} &= a(i-1) + b, D_{i,0,0} = D_{i,0,-} = \infty, \\
D_{i,j,0} &= D_{i-1,j-1} + d(S_1[i], S_2[j]), \\
D_{i,j,-} &= \min\{D_{i,j-1,0} + b, D_{i,j-1,+} + b, D_{i,j-1,-} + a\}, \\
D_{i,j,+} &= \min\{D_{i-1,j,0} + b, D_{i-1,j,-} + b, D_{i-1,j,+} + a\}.
\end{aligned}$$

Das letzte Alignment-Paar findet man anhand der Werte $D_{m,n,0}$, $D_{m,n,+}$ und $D_{m,n,-}$. Der Alignment-Pfad wird wie gehabt rekursiv konstruiert. □

Beispiel 3.11 Gegeben seien $S_1 = abaaba$, $S_2 = abaaaaaabb$, die Abstandsfunktion d mit $d(x,x) = 0$, $d(x,y) = 2$ für $x \neq y$ sowie die Gap-Funktion g mit $g(n) = n + 3$ für $n > 0$. Dann ergibt sich folgende Alignment-Tabelle, wobei im Feld (i,j) der Eintrag $\begin{smallmatrix} D_{i,j,+} \\ D_{i,j,0} \\ D_{i,j,-} \end{smallmatrix}$ steht. Die markierten Tabellenfelder geben einen optimalen Alignment-Pfad wieder.

		a	b	a	a	a	a	a	a	b	b
	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
	∞	4	5	6	7	8	9	10	11	12	13
	4	8	9	10	11	12	13	14	15	16	17
a	∞	0	6	5	6	7	8	9	10	13	14
	∞	8	4	5	6	7	8	9	10	11	12
	5	4	8	9	10	11	12	13	14	15	16
b	∞	6	0	6	7	8	9	10	11	10	11
	∞	9	8	4	5	6	7	11	12	13	14
	6	5	4	8	9	10	11	14	15	14	15
a	∞	5	6	0	4	5	6	7	10	13	12
	∞	10	9	8	4	5	6	7	8	9	10
	7	6	5	4	8	9	10	11	12	13	14
a	∞	6	7	4	0	4	5	6	7	10	11
	∞	11	10	9	8	4	5	6	7	8	9
	8	7	6	5	4	8	9	10	11	12	13
b	∞	9	6	7	6	2	6	7	8	7	8
	∞	12	11	10	9	8	6	7	8	9	10
	9	8	7	6	5	6	10	11	12	11	12
a	∞	8	9	6	5	4	2	6	7	10	9
	∞	13	12	11	10	9	8	6	7	8	9

□

Eine Verallgemeinerung der affinen Gap-Funktionen stellen die monoton wachsenden konkaven Funktionen dar, d.h. solche Funktionen g, für die die Differenz $g(n+1)-g(n)$ monoton nicht wachsend ist (z.B. $g(n)=\sqrt{n}$ oder $g(n)=\log n$). Die Kosten für eine Erweiterung der Lücke werden dann mit wachsender Lückenlänge geringer. Ohne Beweis erwähnen wir, dass für solche Gap-Funktionen das optimale Alignment mit einem Aufwand von $O(mn\log\max\{m,n\})$ bestimmt werden kann. Der sehr interessante Algorithmus ist im Abschnitt 12.6 des Buches von Gusfield zu finden.

3.4 Lösung mit linearem Platzbedarf

Der bisher betrachtete Alignment-Algorithmus benötigte eine Laufzeit von $\Theta(mn)$ und einen Speicherplatz von $\Theta(mn)$. Bei üblichen Sequenzlängen von bis zu 10^4 ist bei der heutigen Technik die Laufzeit kein Problem, der Platzbedarf aber schon. Wir werden jetzt einen Algorithmus entwickeln, der mit linearem Platz auskommt und etwa die doppelte Laufzeit benötigt.

Ein genauerer Blick auf Algorithmus 3.1 zeigt, dass man die *Bewertung* des optimalen Alignments mit linearem Platz bestimmen kann, da man für die Bestimmung einer Zeile i der Alignment-Tabelle nur die Zeile $i-1$ benötigt. Es ist also für die Berechnung von $D(S_1,S_2)$ gar nicht nötig, die gesamte Alignment-Tabelle zu erzeugen. Man benötigt lediglich zwei Vektoren der Länge n, um die aktuelle und die vorherige Zeile zu speichern.

Algorithmus 3.2 zur Berechnung eines optimalen Alignments benötigt allerdings die gesamte Tabelle und damit quadratischen Platz. Um auch das optimale Alignment mit linearem Platzbedarf zu ermitteln, verfolgen wir eine *Teile-und-herrsche*-Strategie. Gilt $m=1$, so bestimmt man das optimale Alignment wie in Algorithmus 3.2. Anderenfalls ermittelt man ein Feld in der Zeile $\lfloor m/2\rfloor$, durch das der Alignment-Pfad verläuft. Ist dieses Feld in der Spalte p, so zerfällt das optimale Alignment von S_1 mit S_2 in ein optimales Alignment von $S_1[1\ldots\lfloor m/2\rfloor]$ mit $S_2[1\ldots p]$ sowie in ein optimales Alignment von $S_1[\lfloor m/2\rfloor+1\ldots m]$ mit $S_2[p+1\ldots n]$. Diese beiden Alignments werden rekursiv ermittelt.

Das Feld $(\lfloor m/2\rfloor,p)$ im optimalen Alignment-Pfad wird wie folgt bestimmt. Bis zur Zeile $\lfloor m/2\rfloor$ berechnet man wie gehabt die Werte $D_{i,j}$. Ab der Zeile $\lfloor m/2\rfloor$ berechnet man außerdem eine Zahl $P_{i,j}$ zwischen 0 und n, die das letzte Feld in der Zeile $\lfloor m/2\rfloor$ im optimalen Pfad nach (i,j) angibt. Die Initialisierungsregel für die Zeile $\lfloor m/2\rfloor$ lautet:

$$P_{\lfloor m/2\rfloor,j}=j \text{ für } 0\le j\le n.$$

Für $i>\lfloor m/2\rfloor$ erhält man die Rekursionsbeziehung:

$$P_{i,j}=\begin{cases}P_{i-1,j-1}, & \text{falls} D_{i,j}=D_{i-1,j-1}+d(S_1[i],S_2[j])\\ P_{i-1,j}, & \text{falls} D_{i,j}=D_{i-1,j}+d(S_1[i],\_)\\ P_{i,j-1}, & \text{falls} D_{i,j}=D_{i,j-1}+d(\_,S_2[j])\end{cases}$$

Der Wert $P_{m,n}$ gibt dann ein Feld in der Zeile $\lfloor m/2\rfloor$ an, durch das der Alignment-Pfad verläuft. Wie bei den D-Werten muss man auch von den P-Werten nur die beiden aktuell letzten Zeilen speichern.

Algorithmus 3.3 Alignment-Rekursiv (Hirschberg)

Eingabe: Wörter S_1, S_2, $|S_1| = m$, $|S_2| = n$
Ausgabe: Optimales Alignment von S_1 und S_2
Aufruf: $\mathrm{AR}(S_1, S_2)$
(1) **if** $m = 1$ **then return** Alignment nach Algorithmus 3.2;
(2) Bestimme Feld $(\lfloor m/2 \rfloor, p)$ im optimalen Alignment-Pfad;
(3) **return** $\mathrm{AR}(S_1[1 \ldots \lfloor m/2 \rfloor], S_2[1 \ldots p])$ $\mathrm{AR}(S_1[\lfloor m/2 \rfloor + 1 \ldots m], S_2[p+1 \ldots n])$;

Satz 3.12 *Algorithmus 3.3 bestimmt ein optimales Alignment mit einem Platzbedarf von $O(m + n)$ in einer Zeit von $O(mn)$.*

Beweis. Der Korrektheitsbeweis für die Rekursionsformel für die Werte $P_{i,j}$ erfolgt ähnlich wie der Beweis von Satz 3.1 durch vollständige Induktion und Betrachtung der möglichen Alignments. Der Platzbedarf für die Speicherung der P- und der D-Werte beträgt $O(n)$, da (wie bereits erwähnt) nur die aktuelle und die unmittelbar vorhergehende Zeile der Tabelle gespeichert werden müssen. Für die Ausgabe des Alignments ist ein Platz von $O(m + n)$ erforderlich.

Für den Beweis der Laufzeitabschätzung definieren wir die Berechnung eines Alignment-Feldes sowie die Bestimmung des Vorgänger-Feldes im Alignment-Pfad als jeweils einen Schritt. Wir behaupten und beweisen durch Induktion über m, dass die Anzahl der Schritte des Algorithmus durch

$$2(m + 1)(n + 1) + 5(m - 1) + (m + 2n)\lceil \log_2 m \rceil$$

beschränkt ist. Da man o.B.d.A. $m \leq n$ annehmen darf, ist dies ein Algorithmus mit einer Laufzeit von $O(mn)$.

Der Induktionsanfang wird bei diesem Beweis für $m = 1$ und $m = 2$ gelegt. Im Induktionsschritt folgert man aus der Gültigkeit der Abschätzung für m und $m+1$ auf die Gültigkeit für $2m$ und $2m + 1$.

IA: Für $m = 1$ beträgt die Anzahl der Schritte $(m+1)(n+1)$ für die Bestimmung der D-Werte und der P-Werte sowie höchstens $(m + 1)(n + 1)$ für die Bestimmung des Alignment-Pfades, insgesamt also höchstens $2(m + 1)(n + 1) = 4(n + 1)$.
Für $m = 2$ beträgt die Anzahl der Schritte $(m + 1)(n + 1) = 3(n + 1)$ für die Bestimmung der D-Werte und der P-Werte sowie $4(p + 1)$ und $4(n - p + 1)$ für die rekursive Bestimmung der Teil-Alignments. Der Gesamtaufwand beträgt also $7(n + 1) + 4 = 2(m + 1)(n + 1) + n + 5 < 2(m + 1)(n + 1) + 5(m - 1) + (m + 2n) \log_2 m$.

IS: Sei die Behauptung für beliebige n sowie für m und $m + 1$ gezeigt.
Für $2m$ beträgt die Anzahl der Schritte $(2m + 1)(n + 1)$ für die Bestimmung der D-Werte und der P-Werte sowie die Anzahl der Schritte für den rekursiven Aufruf mit den Parametern (m, p) und $(m, n - p)$. Nach Induktionsvoraussetzung kann diese Schrittzahl insgesamt wie folgt abgeschätzt werden:

$$\begin{array}{lll}
2(m+1)(p+1) & + 5(m-1) & + (m+2p)\lceil \log_2 m \rceil \\
+ 2(m+1)(n-p+1) & + 5(m-1) & + (m+2(n-p))\lceil \log_2 m \rceil \\
\hline
= (2m+1)(n+1) + m + n + 3 & + 5(2m-1) - 5 & + (2m+2n)\lceil \log_2 m \rceil \\
< (2m+1)(n+1) & + 5(2m-1) & + (2m+2n)\lceil \log_2 (2m) \rceil
\end{array}$$

Für $2m+1$ beträgt die Anzahl der Schritte $(2m+2)(n+1)$ für die Bestimmung der D-Werte und der P-Werte sowie die Anzahl der Schritte für den rekursiven Aufruf mit den Parametern (m,p) und $(m,n-p)$. Nach Induktionsvoraussetzung kann diese Schrittzahl insgesamt wie folgt abgeschätzt werden:

$$\begin{array}{lll}
2(m+1)(p+1) & +\ 5(m-1) & +\ (m+2p)\lceil\log_2 m\rceil \\
+\ 2(m+2)(n-p+1) & +\ 5m & +\ (m+1+2(n-p))\lceil\log_2(m+1)\rceil \\
\hline
<\ (2m+4)(n+2) & +\ 5(2m)-5 & +\ (2m+1+2n)\lceil\log_2(m+1)\rceil \\
=\ (2m+2)(n+1)+2m+1+2n+5 & +\ 5(2m)-5 & +\ (2m+1+2n)\lceil\log_2(m+1)\rceil \\
=\ (2m+2)(n+1) & +\ 5(2m) & +\ (2m+1+2n)\lceil\log_2(2m+1)\rceil
\end{array}$$

Beim Übergang von der vorletzten zur letzten Zeile nutzten wir aus, dass für jede natürliche Zahl $m \geq 1$ $\lceil\log_2(2m+1)\rceil = \lceil\log_2(2m+2)\rceil = \lceil\log_2(m+1)\rceil + 1$ gilt. Damit ist die Induktionsbehauptung für $2m$ und $2m+1$ bewiesen und der Beweis komplett. □

Beispiel 3.12 Für S_1 = `tempel` und S_2 = `treppe` erhalten wir folgenden Ablauf (in den Zeilen 4 – 6 steht jeweils als erstes der D-Wert und als zweites der P-Wert).

		t	r	e	p	p	e
	0	1	2	3	4	5	6
t	1	0	1	2	3	4	5
e	2	1	1	1	2	3	4
m	3	2	2	2	2	3	4
p	4, 0	3, 1	3, 1	3, 2	2, 3	2, 4	3, 4
e	5, 0	4, 1	4, 1	3, 1	3, 3	3, 3	2, 4
l	6, 0	5, 1	5, 1	4, 1	4, 1	4, 3	3, 4

Als Feld des Alignment-Pfades erhalten wir $(3,4)$ (in der Tabelle markiert); das optimale Alignment ergibt sich rekursiv als $\begin{pmatrix} \alpha_1 & \alpha_2 \\ \beta_1 & \beta_2 \end{pmatrix}$, wobei (α_1,β_1) ein optimales Alignment von (`tem`,`trep`) und (α_2,β_2) ein optimales Alignment von (`pel`,`pe`) sind. □

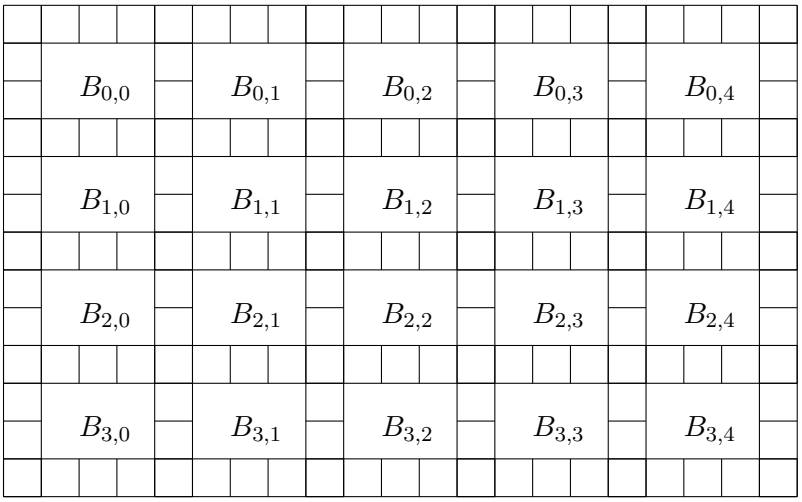
Das hier besprochene Verfahren zur Linearisierung des Platzbedarfes kann auf viele andere Algorithmen der dynamischen Programmierung übertragen werden. Es sei empfohlen, zur Übung das Rucksackproblem mit linearem Platzbedarf zu lösen.

3.5 Lösung in subquadratischer Zeit

In diesem Abschnitt soll ein Algorithmus zur Ermittlung des Levenshtein-Abstandes vorgestellt werden, der eine Laufzeit von $O(mn/\log(mn))$ besitzt. Ein ähnlicher Algorithmus wurde für die Multiplikation Boolescher Matrizen durch Arlazarov, Dinic, Kronrod und Faradzev angegeben, weshalb man dieses und verwandte Verfahren oft als *Four Russians Algorithmen* in der Literatur findet. Der hier präsentierte Algorithmus hat zwei Grundideen.

1. Statt der Werte $D_{i,j}$ bestimmt man Differenzen (*Offset*-Werte) benachbarter $D_{i,j}$-Werte. Der Vorteil der Verwendung von Offset-Werten besteht darin, dass sie nur Werte aus dem Bereich $\{-1,0,1\}$ annehmen können.

2. Man berechnet die Offset-Werte nur an bestimmten Stellen der Tabelle, genauer in den Zeilen ip und den Spalten jq, wobei p und q festgelegte natürliche Zahlen sind. Anschaulich überdecken wir die Alignment-Tabelle mit Blöcken der Größe $(p+1) \times (q+1)$, deren Ränder sich überlappen. Der Block $\mathbf{B}_{i,j}$ umfasst den Bereich, der durch die Zeilen von ip bis $(i+1)p$ sowie die Spalten von jq bis $(j+1)q$ eingegrenzt wird. Das folgende Bild zeigt eine Überdeckung für $m = 12, n = 20, p = 3, q = 4$.



Wie wir zeigen werden, sind durch den oberen und den linken Rand eines Blockes $B_{i,j}$ sowie durch die zugehörigen Teilwörter $S_1[ip+1 \ldots (i+1)p]$ und $S_2[jq+1 \ldots (j+1)q]$ der untere und der rechte Rand von $B_{i,j}$ eindeutig bestimmt. Es gibt damit eine *Blockfunktion*, die für den linken und oberen Rand sowie die zugehörigen Teilwörter eines Blockes den rechten und unteren Rand des Blockes festlegt. Da die Blockfunktion nur endlich viele Argumente hat, kann man sie in einem Präprozessing berechnen und in einer Tabelle speichern. In der Berechnungsphase bestimmt man den rechten und den unteren Rand eines Blockes durch Nachschlagen in der Tabelle.

Im Weiteren seien S_1 und S_2 Wörter über Σ mit $|\Sigma| = \sigma$, $|S_1| = m$, $|S_2| = n$. Der Einfachheit halber nehmen wir an, dass $m = pm'$ und $n = qn'$ für natürliche Zahlen p, q gilt.

Definition 3.7 *Es seien S_1 und S_2 Wörter mit $|S_1| = m$, $|S_2| = n$. Für $1 \leq i \leq m$ und $1 \leq j \leq n$ seien*

- $H_{i,j}(S_1, S_2) = D_{i,j}(S_1, S_2) - D_{i,j-1}(S_1, S_2)$ *(horizontaler Offset) und*
- $V_{i,j}(S_1, S_2) = D_{i,j}(S_1, S_2) - D_{i-1,j}(S_1, S_2)$ *(vertikaler Offset).*

Der Wert $D_{m,n}$ kann leicht aus den Offset-Werten berechnet werden, z.B. als: $D_{m,n} = \sum_{i=1}^{m} V_{i,0} + \sum_{j=1}^{n} H_{m,j}$. Um den Levenshtein-Abstand zu ermitteln, brauchen wir also tatsächlich nur die vertikalen bzw. horizontalen Offset-Werte auf den vertikalen bzw. horizontalen Rändern der Blöcke zu berechnen. Ähnlich wie die $D_{i,j}$-Werte lassen sich die Offset-Werte induktiv berechnen.

Satz 3.13 *Es seien S_1 und S_2 Wörter mit $|S_1| = m$, $|S_2| = n$. Es gilt*

- $H_{0,j} = 1$ *für* $1 \leq j \leq n$.
  $H_{i,j} = \min\{d(S_1[i], S_2[j]) - V_{i,j-1}, H_{i-1,j} - V_{i,j-1} + 1, 1\}$ *für* $1 \leq i \leq m$, $1 \leq j \leq n$.
- $V_{0,j} = 1$ *für* $1 \leq j \leq n$.
  $V_{i,j} = \min\{d(S_1[i], S_2[j]) - H_{i-1,j}, V_{i,j-1} - H_{i-1,j} + 1, 1\}$ *für* $1 \leq i \leq m$, $1 \leq j \leq n$.

Beweis. Wir führen den Beweis für die $H_{i,j}$-Werte; die Behauptung für die $V_{i,j}$-Werte kann ähnlich gezeigt werden.
Zunächst einmal gilt für beliebiges $1 \leq j \leq n$: $H_{0,j} = D_{0,j} - D_{0,j-1} = j - (j-1) = 1$.
Für $1 \leq i \leq m$, $1 \leq j \leq n$ gilt die Rekursionsbeziehung

$$\begin{aligned} D_{i,j} &= \min\{D_{i-1,j-1} + d(S_1[i], S_2[j]), D_{i-1,j} + 1, D_{i,j-1} + 1\} \\ &= \min\{D_{i,j-1} - V_{i,j-1} + d(S_1[i], S_2[j]), D_{i,j-1} - V_{i,j-1} + H_{i-1,j} + 1, D_{i,j-1} + 1\} \\ &= D_{i,j-1} + \min\{d(S_1[i], S_2[j]) - V_{i,j-1}, H_{i-1,j} - V_{i,j-1} + 1, 1\} \end{aligned}$$

und damit $H_{i,j} = D_{i,j} - D_{i,j-1} = \min\{d(S_1[i], S_2[j]) - V_{i,j-1}, H_{i-1,j} - V_{i,j-1} + 1, 1\}$. □

Als unmittelbare Folgerungen aus der Rekursionsbeziehung ergeben sich wichtige Eigenschaften der Offset-Vektoren, die im Algorithmus genutzt werden.

Folgerung 3.14 *Es seien S_1 und S_2 Wörter mit $|S_1| = m$, $|S_2| = n$.*

1. $H_{i,j} \in \{-1, 0, 1\}$ *für* $0 \leq i \leq m, 1 \leq j \leq n$, $V_{i,j} \in \{-1, 0, 1\}$ *für* $1 \leq i \leq m, 0 \leq j \leq n$.
2. *Für* $0 \leq i \leq m - p$ *und* $0 \leq j \leq n - q$ *sind die Vektoren* $(V_{i+1,j+q}, V_{i+2,j+q}, \ldots, V_{i+p,j+q})$ *und* $(H_{i+p,j+1}, H_{i+p,j+2}, \ldots, H_{i+p,j+q})$ *eindeutig durch* $(V_{i+1,j}, V_{i+2,j}, \ldots, V_{i+p,j})$, $(H_{i,j+1}, H_{i,j+2}, \ldots, H_{i,j+q})$, $S_1[i+1 \ldots i+p]$ *und* $S_2[j+1 \ldots j+q]$ *bestimmt.*

Damit gibt es eine *Blockfunktion*

$$f : \{-1, 0, 1\}^p \times \{-1, 0, 1\}^q \times \Sigma^p \times \Sigma^q \to \{-1, 0, 1\}^p \times \{-1, 0, 1\}^q,$$

wobei $f(\vec{V}_1, \vec{H}_1, \alpha, \beta) = (\vec{V}_2, \vec{H}_2)$ bedeutet, dass der Block mit dem linken Rand $\vec{V}_1$, dem oberen Rand $\vec{H}_1$, und den Teilwörtern α von S_1 und β von S_2 den rechten Rand $\vec{V}_2$ und den unteren Rand $\vec{H}_2$ besitzt. Für eine Eingabe $(\vec{V}_1, \vec{H}_1, \alpha, \beta)$ kann der Wert der Blockfunktion durch dynamische Programmierung mit einem Aufwand von $O(pq)$ bestimmt werden.

Beispiel 3.13 Es seien $p = 3$, $q = 4$, $\vec{V} = (1, 0, -1)$, $\vec{H} = (0, 1, 1, -1)$, $\alpha = abc$, $\beta = acba$. Dann ergibt sich folgende Alignment-Tabelle für die Offset-Werte (in den Feldern stehen jeweils als erstes die V-Werte und als zweites die H-Werte):

		a		c		b		a	
			0		1		1		-1
a	1	0	-1	0	1	0	1	1	0
b	0	1	0	0	0	-1	0	0	1
c	-1	0	1	0	0	1	1	0	0

Die V-Werte in der letzten Spalte sind $(1, 0, 0)$, die H-Werte in der letzten Zeile sind $(1, 0, 1, 0)$. Folglich hat die Blockfunktion für das Argument $((1, 0, -1), (0, 1, 1, -1), abc, acba)$ den Wert $((1, 0, 0), (1, 0, 1, 0))$. □

Der gesamte Algorithmus ergibt sich wie folgt:

1. Bestimme die Blockfunktion für alle möglichen Eingaben.
2. Berechne durch dynamische Programmierung auf den horizontalen Blockrändern die H-Werte und auf den vertikalen Blockrändern die V-Werte.
   (a) Setze $V_{i,0} = 1$, $H_{0,j} = 1$ für $1 \leq i \leq m$, $1 \leq j \leq n$.
   (b) Ermittle für jeden Block den rechten und unteren Rand durch Nachschlagen der Blockfunktion.

Die Laufzeit für Phase 1 beträgt $O((3|\Sigma|)^{p+q}pq)$, da es $(3|\Sigma|)^{p+q}$ Argumente für die Blockfunktion gibt und die Berechnung des Funktionswertes für ein Argument durch dynamische Programmierung einen Aufwand von $O(pq)$ kostet.

In der 2. Phase haben wir für $O(\frac{mn}{pq})$ Blöcke die Blockfunktion aufzusuchen. Dies kostet einen Aufwand von $O(p+q)$ pro Block, insgesamt also $O(\frac{mn(p+q)}{pq})$.

Nehmen wir der Einfachheit halber $m = n$ an und setzen wir $p = q = (\log_{3|\Sigma|} n)/2$, so erhalten wir einen Aufwand von $O(n \log^2 n)$ für die Berechnung der Blockfunktion und von $O(n^2/\log n)$ für die Bestimmung des Levenshtein-Abstandes. Zusammenfassend können wir feststellen:

Satz 3.15 *Für S_1, S_2 mit $|S_1| = |S_2| = n$ kann der Levenshtein-Abstand mit einem Aufwand von $O(n^2/\log n)$ bestimmt werden.*

Praktische Anwendung

Das bisher gezeigte Resultat ist eher von theoretischem Wert. Man rechnet leicht nach, dass selbst für kleine Werte von p, q und σ die Tabelle für die Blockfunktion sehr groß wird. Für praktische Anwendungen wählt man $q = 1$. Damit muss man alle $V_{i,j}$-Werte berechnen, kann aber durch Nutzung von Bitarithmetik eine Beschleunigung erreichen. Für die Berechnung der Blockfunktion benötigt man nicht ein konkretes Teilwort α der Länge p von S_1 und ein konkretes Symbol x von S_2, sondern nur den Bitvektor $B_x(\alpha)$ der Vorkommen von x in α. Man kann nämlich nachweisen, dass die Werte $(V_{i+1,j+1}, \ldots, V_{i+p,j+1})$ und $H_{i+p,j+1}$ nur von $(V_{i+1,j}, \ldots, V_{i+p,j})$ und $H_{i,j+1}$ und $B_x(\alpha)$ mit $\alpha = S_1[i+1 \ldots i+p]$, $x = S_2[j+1]$ abhängen (Übungsaufgabe). Da es 2^p verschiedene Bitvektoren der Länge p gibt, braucht man im Präprozessing nur $2^p \cdot 3^{p+1}$ Werte der Blockfunktion zu berechnen. Für $p = 6$ wären das z.B. 139968 Werte. Außerdem ist die Blockfunktion unabhängig vom Alphabet und muss damit nur ein einziges Mal berechnet werden. In der Berechnungsphase speichert man den Vektor $(V_{i+1,j+1}, \ldots, V_{i+p,j+1})$ in *einer* Integer-Variable und berechnet ihn in *einem* einzigen Schritt. Für die Berechnung von p aufeinander folgenden Zeilen benötigt man damit $p \cdot \sigma$ Schritte für die Bestimmung aller Bitvektoren sowie $n = |S_2|$ Schritte für die Bestimmung der Offset-Vektoren. Die erreichte Beschleunigung liegt also in der Größenordnung p.

Beispiel 3.14 Einen Vektor $V = (x_1, x_2, \ldots, x_p)$ über $\{1, 0, -1\}$ kann man auf natürliche Weise durch die ganze Zahl $Z(V) = \sum_{i=1}^{p} x_i \cdot 3^{p-i}$ darstellen. Ein Bitvektor $(b_1, b_2, \ldots, b_p)$ entspricht der natürlichen Zahl $\sum_{i=1}^{p} b_i \cdot 2^{p-i}$.
Für $p = 3$, $\vec{V} = (1, 0, -1)$, $\vec{H} = 0$ und den Bitvektor $B = 100$ erhält man als Ergebnis der Blockfunktion $((0, 1, 0), 1)$. In der Tabelle der Blockfunktion f findet man deshalb den Eintrag

$f : (8, 0, 4) \to (3, 1)$.
Für die Berechnung der Zeilen von $i+1$ bis $i+p$ stellt man für das zugehörige Teilwort $\alpha = S_1[i+1 \ldots i+p]$ und alle Buchstaben x die Bitvektoren $B_x = B_x(\alpha)$ auf. Zum Beispiel gilt für $\alpha = abc$ und $x = a$ $B_x = 100$ bzw. als ganze Zahl $B_x = 4$. Außerdem verwendet man die bereits zuvor ermittelten Werte $H_{i,j}, 1 \leq j \leq n$. Anstelle der V-Vektoren verwenden wir Zahlen $Z_{i+p,j} = Z((V_{i+1,j}, \ldots, V_{i+p,j}))$. Es gilt als Initialisierung $Z_{i+p,0} = \sum_{i=1}^{p} \cdot 3^{p-i} = (3^p - 1)/2$. Gilt $S_2[j] = x$, so lautet die Aktualisierungsregel $(Z_{i+p,j}, H_{i+p,j}) = f(Z_{i+p,j-1}, H_{i,j}, B_x)$. Für $\alpha = abc$, $x = a$, $Z_{i+p,j-1} = 8$, $H_{i,j} = 0$ ergibt sich damit $Z_{i+p,j} = 3$, $H_{i+p,j} = 1$. □

3.6 Weitere Algorithmen für die inexakte Suche

Wir wenden uns noch einmal einem sehr wichtigen Spezialfall der inexakten Suche zu. Gesucht sind die inexakten Vorkommen von P mit $|P| = m$ in T mit $|T| = n$, deren Levenshtein-Bewertung durch k beschränkt ist. Diese spezielle Aufgabenstellung hat eine große praktische Bedeutung. Neben dem Standard-Algorithmus sind weitere Lösungsansätze möglich, die häufig eine bessere Laufzeit aufweisen. Wir betrachten hier zwei Verfahren: das erste erweitert den Shift-And-Algorithmus, während das zweite mögliche Vorkommen mittels exakter Suche nach mehreren Wörtern herausfiltert.

Erweiterter Shift-And-Algorithmus

Man kann den NEA zur Erkennung von P aus dem Shift-And-Algorithmus (siehe Abschnitt 1.4) leicht in einen NEA zur Erkennung von P mit höchstens k Fehlern erweitern. Der NEA zur Erkennung mit k Fehlern besteht aus $(k+1)$ Kopien des NEA zur Erkennung ohne Fehler. Den Einfügungen, Ersetzungen und Löschungen entsprechen Transitionen aus der $(j-1)$-ten in die j-te Kopie.

Formal besitzt der NEA zur Erkennung von $P = x_1 x_2 \cdots x_m$ mit k Fehlern die Zustandsmenge $\{0, 1, \ldots, m\} \times \{0, 1, \ldots, k\}$, die Transitionenmenge

$$
\begin{aligned}
\delta &= \delta_0 \cup \delta_1 \cup \delta_2 \cup \delta_3 \cup \delta_4 \text{ mit} \\
\delta_0 &= \{((0,j), x, (0,j)) : 0 \leq j \leq k, x \in \Sigma\}, \\
\delta_1 &= \{((i-1,j), x_i, (i,j)) : 1 \leq i \leq m, 0 \leq j \leq k\}, \\
\delta_2 &= \{((i,j-1), x, (i,j)) : 0 \leq i \leq m, 1 \leq j \leq k, x \in \Sigma\}, \\
\delta_3 &= \{((i,j-1), \varepsilon, (i,j)) : 0 \leq i \leq m, 1 \leq j \leq k\}, \\
\delta_4 &= \{((i-1,j-1), x, (i,j)) : 1 \leq i \leq m, 1 \leq j \leq k, x \in \Sigma\},
\end{aligned}
$$

die Startzustände $\{(0,j) : 0 \leq j \leq k\}$ und die Menge der Endzustände $\{(m,j) : 0 \leq j \leq k\}$. Der Zustand (i,j) ist genau dann erreichbar, wenn ein Suffix des gelesenen Textes ein Vorkommen von $P[1 \ldots i]$ mit *höchstens* k Fehlern ist Die Transitionen in den einzelnen Zeilen von δ erklären sich wie folgt.

1. Die Startzustände sind immer erreichbar, da an jeder Stelle das leere Wort exakt vorkommt; daher die Transitionen aus δ_0.

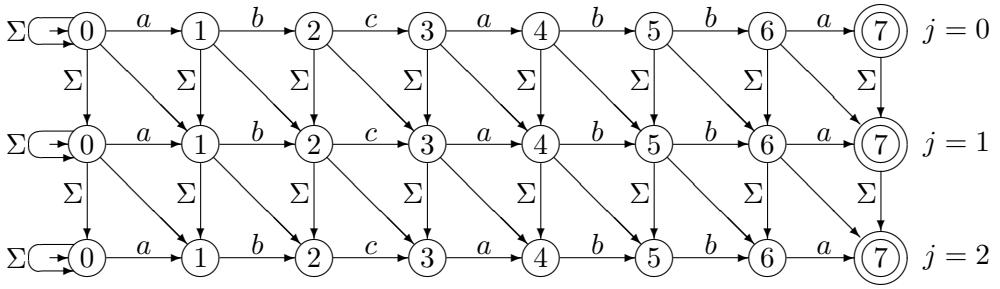
2. Wurde $P[1 \ldots i-1]$ mit höchstens j Fehlern gefunden und stimmt $P[i]$ mit dem nächsten Textsymbol überein, so hat man nach dem nächsten Symbol ein Vorkommen von $P[1 \ldots i]$ mit höchstens j Fehlern; daher die Transitionen aus δ_1.

3. Wurde $P[1 \ldots i]$ mit höchstens $j-1$ Fehlern gefunden, so hat man nach dem nächsten Textsymbol ein Vorkommen von $P[1 \ldots i]$ mit höchstens j Fehlern (in T ist gegenüber P ein Zeichen "eingefügt"); daher die Transitionen aus δ_2.

4. Wurde $P[1 \ldots i-1]$ mit höchstens $j-1$ Fehlern gefunden, so hat man an der aktuellen Textstelle auch ein Vorkommen von $P[1 \ldots i]$ mit höchstens j Fehlern (ein Zeichen von P ist in T "gelöscht"); daher die Transitionen aus δ_3.

5. Wurde $P[1 \ldots i-1]$ mit höchstens $j-1$ Fehlern gefunden, so hat man nach dem nächsten Textsymbol ein Vorkommen von $P[1 \ldots i]$ mit höchstens j Fehlern (unabhängig von der Übereinstimmung zwischen $P[i]$ und dem Text); daher die Transitionen aus δ_4.

Beispiel 3.15 Für $P = abcabba$ und $k = 2$ ergibt sich der folgende NEA (Kanten ohne Beschriftung entsprechen Transitionen mit Symbolen aus $\Sigma \cup \{\varepsilon\}$):



□

Für die Implementierung benötigen wir die Bitvektoren $B[x]$ wie gehabt für die Vorkommen der Symbole in P, Z_j, $0 \le j \le k$, für die alten Zustände des NEA und Z'_j, $0 \le j \le k$, für die neuen Zustände des NEA. Ein Endzustand ist erreicht, wenn das m-te Bit in einem der Vektoren Z_j gleich 1 ist.

In der *Initialisierung* werden die Zustände (i, j) mit $0 \le i \le j \le k$ aktiviert. Dies geschieht mittels der Zuweisungen

$$Z_0 \leftarrow 0;\ Z_j \leftarrow Z_{j-1} \,|\, (1 \mathbin{<<} (j-1)),\ \text{für } 1 \le j \le k.$$

Die *Aktualisierung* für ein Textsymbol x geschieht wie folgt:
1. $Z'_0 \leftarrow ((Z_0 \mathbin{<<} 1 \,|\, 1) \,\&\, B[x]$;
2. $Z'_j \leftarrow (((Z_j \mathbin{<<} 1 \,|\, 1) \,\&\, B[x]) \,|\, Z_{j-1} \,|\, (Z_{j-1} \mathbin{<<} 1) \,|\, (Z'_{j-1} \mathbin{<<} 1)$, für $1 \le j \le k$;
3. $Z_j \leftarrow Z'_j$, für $0 \le j \le k$.

Im ersten Schritt wird der Bitvektor für die Zustände $(i, 0)$ wie beim Shift-And-Algorithmus aktualisiert. Im zweiten Schritt aktualisiert man die Bitvektoren für die Zustände (i, j) mit $j > 0$. Die Operanden für die OR-Operation ergeben sich aus den zu simulierenden Transitionenmengen $\delta_0, \delta_1, \ldots, \delta_4$. Man beachte, dass man für die Berechnung des (neuen) Wertes Z'_j sowohl den (alten) Wert Z_{j-1} als auch den (neuen) Wert Z'_{j-1} benötigt. Damit kann dieser zweite Schritt nicht parallelisiert werden. Im dritten Schritt werden schließlich die alten Werte

der Zustandsvektoren Z_j durch die neuen Werte Z'_j ersetzt. Eine Aktualisierung kostet damit $O(k)$ Bitvektor-Operationen, für hinreichend kurze Suchwörter also $O(k)$ Schritte.

Eine verbesserte Variante des Algorithmus nutzt aus, dass die aktuell erreichbaren Zustände in der Diagonalen d, d.h. die Zustände $\{(i,j) : i - j = d\}$, nur von den zuvor erreichbaren Zuständen in den Diagonalen $d-1$ und d und dem Textzeichen abhängig sind. Dies ermöglicht es, mit einem einzigen Bitvektor D auszukommen, wobei die Reihenfolge der Zustände nach Diagonalen geordnet ist. Ist $(j+1)|P|$ kleiner als ein Computerwort, so ist der Aufwand konstant. Details findet man im Buch von Navarro und Raffinot.

Ein Filteralgorithmus

Das Ziel des hier beschriebenen Algorithmus ist, mögliche Kandidaten für ein inexaktes Vorkommen herauszufiltern. Dabei wird der folgende Zusammenhang zwischen der inexakten Suche und der exakten Suche nach mehreren Wörtern ausgenutzt.

Satz 3.16 *Es sei $P = P_1 P_2 \cdots P_{k+1}$ eine Zerlegung von P. Gilt $D(P,S) \leq k$, so enthält S ein exaktes Vorkommen eines der Wörter $\{P_1, P_2, \ldots, P_{k+1}\}$.*

Beweis. Gilt $D(P,S) \leq k$, so existiert ein Alignment von P und S mit einer Bewertung von höchstens k. Dieses Alignment wird in $k+1$ Teil-Alignments von $P_1, P_2, \ldots, P_{k+1}$ mit Teilwörtern von S zerlegt. Die Bewertung des gesamten Alignments ergibt sich als Summe der Bewertungen der Teil-Alignments. Da jedes Teil-Alignment mindestens die Bewertung 0 besitzt und die Summe der Bewertungen kleiner als $k+1$ ist, muss mindestens eins der Teil-Alignments mit 0 bewertet sein. □

Der Filter-Algorithmus funktioniert also wie folgt. Man zerlegt das Suchwort P in $k+1$ Teilwörter (in der Regel etwa gleicher Länge). Dann sucht man nach exakten Vorkommen dieser Teilwörter und versucht schließlich, diese exakten Vorkommen zu inexakten Vorkommen von P zu erweitern.

Beispiel 3.16 Es sei $P =$ `fische`, und die Fehlerschranke sei 1. Wir zerlegen $P = P_1 \cdot P_2$ mit $P_1 = fis$, $P_2 = che$. Im Text $T =$ `fritzefischtefrische` finden wir ein exaktes Vorkommen von `fis` an der Stelle 7 und eines von `che` an der Stelle 18. Dann sucht man nach einem Vorkommen von *che* mit Fehler höchstens 1, das an der Stelle 10 beginnt, sowie nach einem Vorkommen von *fis* mit Fehler höchstens 1, das an der Stelle 17 endet.

Effizient sind Filtermethoden, wenn P relativ lang und k klein ist. Im mittleren Fall wird die Suche erheblich beschleunigt, da exakte Vorkommen im Mittel relativ selten gefunden werden. Da die Gesamtlänge der Suchwörter bei der Suche nach mehreren Wörtern gleich $|P|$ und damit recht kurz ist, bieten sich als Algorithmen solche mit Bit-Arithmetik, z.B. der Shift-And-Algorithmus für Mengen sowie der NDAWG-Algorithmus für Mengen an.

3.7 Multiple Alignments

Wir wollen zuletzt das Problem des Alignments von mehr als zwei Sequenzen betrachten. Dieses ist seit einigen Jahren ein heißes Forschungsthema, da es einerseits entscheidende Bedeutung in der Molekularbiologie besitzt (insbesondere bei der Strukturvorhersage von Eiweißmolekülen) und andererseits nicht so einfach wie das Problem des paarweisen Alignments gelöst werden kann.

Motivation. Proteine aus einer gemeinsamen Familie besitzen ähnliche biochemische Eigenschaften und eine ähnliche dreidimensionale Struktur; sie können jedoch als Aminosäure-Sequenzen sehr unterschiedlich aussehen. Ein typisches Beispiel ist das Hämoglobin, das in Insekten wie in Säugetieren ähnliche Struktur und Eigenschaften aufweist, wobei entwicklungsbiologisch weit entfernte Arten sehr verschiedene Aminosäure-Sequenzen besitzen. Man nimmt an, dass die ähnlichen biochemischen Eigenschaften durch lokale Sequenz-Ähnlichkeiten bedingt sind. *Paarweise* lokale Alignments sind hier von geringem Nutzen, da bei zu großer globaler Ähnlichkeit nicht die wesentlichen Ähnlichkeiten entdeckt werden und bei zu großer Unterschiedlichkeit sich lokale Sequenzähnlichkeiten zufällig ergeben. Darum versucht man, die lokalen Ähnlichkeiten durch ein gemeinsames Alignment mehrerer Sequenzen der Familie zu finden. Hat man für eine Familie ein (hoffentlich korrektes) multiples Alignment ermittelt, so kann man für eine neu erhaltene Sequenz testen, ob diese zur Familie gehört, indem man sie mit dem Alignment vergleicht.

Definition 3.8 *Es seien $S_1, S_2, \ldots, S_k$ Wörter über Σ. Ein* multiples Alignment *von $(S_1, S_2, \ldots, S_k)$ ist ein k-Tupel $(S'_1, S'_2, \ldots, S'_k)$ von Wörtern gleicher Länge über $\Sigma \cup \{\_\}$, wobei S'_i für $1 \leq i \leq k$ aus S_i durch Einfügen von Leerzeichen $\_$ entsteht.*

Bewertung multipler Alignments. Als praktikabel haben sich sogenannte *Sum-of-Pairs*-Bewertungen erwiesen, bei denen die Bewertung eines k-Tupels als Summe der Bewertungen der $\frac{k(k-1)}{2}$ in diesem k-Tupel enthaltenen Paare entsteht.

Definition 3.9 *Es seien $S_1, S_2, \ldots, S_k$ Wörter über Σ und $d : (\Sigma \cup \{\_\})^2 \to \mathbb{Z}$ eine Bewertungsfunktion. Die* Sum-of-Pairs-Bewertung *(SP-Bewertung) bezüglich d ergibt sich für ein Alignment $(S'_1, S'_2, \ldots, S'_k)$ mit $|S'_i| = n$ für $1 \leq i \leq k$ als*

$$d(S'_1, S'_2, \ldots, S'_k) = \sum_{t=1}^{n} d(S'_1[t], S'_2[t], \ldots, S'_k[t]) \text{ mit}$$
$$d(a_1, a_2, \ldots, a_k) = \sum_{1 \leq i < j \leq k} d(a_i, a_j) \text{ für } a_1, a_2, \ldots, a_k \in \Sigma \cup \{\_\}.$$

Profile. Ein Vorteil der SP-Bewertung ist, dass sie symmetrisch ist, d.h. dass der Wert $d(a_1, a_2, \ldots, a_k)$ lediglich vom Häufigkeitsvektor der Symbole aus $\Sigma \cup \{\_\}$ in $(a_1, a_2, \ldots, a_k)$ abhängig ist. Ersetzt man die k-Tupel eines Alignments durch die zugehörigen Häufigkeitsvektoren, so entsteht ein *Profil*. Die SP-Bewertung kann auf natürliche Weise auf Häufigkeitsvektoren und Profile ausgedehnt werden. Dabei besitzt ein Profil die gleiche SP-Bewertung wie das definierende multiple Alignment. Häufig ist es bequem, ein Alignment durch sein Profil zu ersetzen bzw. sind die bestehenden Ähnlichkeiten besser aus dem Profil zu erkennen.

Definition 3.10 *Es seien $S_1, S_2, \ldots, S_k$ Wörter über $\Sigma = \{x_1, x_2, \ldots, x_\sigma\}$.*
Der Häufigkeitsvektor $h(a_1, a_2, \ldots, a_k)$ eines k-Tupels $(a_1, a_2, \ldots, a_k) \in (\Sigma \cup \{\_\})^k$ ist der Vektor $(h_0, h_1, \ldots, h_\sigma)$, wobei h_0 die Anzahl der Vorkommen von $\_$ sowie h_i für $1 \leq i \leq \sigma$ die Anzahl der Vorkommen von x_i in $(a_1, a_2, \ldots, a_k)$ sind.
Das Profil eines Alignments $(S'_1, S'_2, \ldots, S'_k)$ mit $|S'_i| = n$ für $1 \leq i \leq k$ ergibt sich als

$$h(S'_1[1], S'_2[1], \ldots S'_k[1]) h(S'_1[2], S'_2[2], \ldots S'_k[2]) \cdots h(S'_1[n], S'_2[n], \ldots S'_k[n]).$$

Definition 3.11 *Es seien $\Sigma = \{x_1, x_2, \ldots, x_\sigma\}$ ein Alphabet und $d : (\Sigma \cup \{\_\})^2 \to \mathbb{Z}$ eine Bewertungsfunktion.*
Die Bewertung eines Häufigkeitsvektors $(h_0, h_1, \ldots, h_\sigma)$ bezüglich d ergibt sich als

$$d(h_0, h_1, \ldots, h_\sigma) = \sum_{1 \le i < j \le k} h_i h_j d(x_i, x_j) + \frac{1}{2} \sum_{i=0}^{\sigma} h_i (h_i - 1) d(x_i, x_i).$$

Die Bewertung eines Profils $H_1 H_2 \cdots H_n$ aus den Häufigkeitsvektoren $H_1, H_2, \ldots, H_n$ ergibt sich als $d(H_1 H_2 \cdots H_n) = \sum_{j=1}^{n} d(H_j)$.

Beispiel 3.17 Für $\Sigma = \{a, b\}$ hat

das Alignment	das Profil
_ a a b _ a	3 0 0 0 3 0
b a a b _ a	0 4 4 2 0 3
_ a a a _ a	1 0 0 2 1 1
_ a a a b b	

Die Bewertung bezüglich des Levenshtein-Abstandes beträgt jeweils $3+0+0+4+3+3 = 13$.
□

Exakte Lösung und NP-Vollständigkeit. Sind die Zeichenketten $S_1, S_2, \ldots, S_k$ der Längen $n_1, n_2, \ldots, n_k$ gegeben, so können wir das globale Alignment-Problem analog zum paarweisen Problem lösen, indem wir durch dynamische Programmierung die Bewertungen $D_{t_1,t_2,\ldots,t_k}$ der optimalen multiplen Alignments von $S_1[1 \ldots t_1]$, $S_2[1 \ldots t_2]$, …, $S_k[1 \ldots t_k]$ bestimmen.

Graphentheoretisch ist dies die Suche nach dem kürzesten Weg von $(0, 0, \ldots, 0)$ nach $(n_1, n_2, \ldots, n_k)$ im Graphen mit Kantengewichten (V, E, d), wobei

$$V = \{(t_1, t_2, \ldots, t_k) : 0 \le t_i \le n_i, 1 \le i \le k\},$$
$$E = \{((t_1, t_2, \ldots, t_k), (t'_1, t'_2, \ldots, t'_k)) : (t_1, t_2, \ldots, t_k) \ne (t'_1, t'_2, \ldots, t'_k), t_i \le t'_i \le t_i + 1\}$$

und

$$d((t_1, t_2, \ldots, t_k), (t'_1, t'_2, \ldots, t'_k)) = \sum_{i<j} d_{i,j} \text{ mit}$$

$$d_{i,j} = \begin{cases} 0 & \text{, falls } t'_i = t_i \wedge t'_j = t_j \\ d(S_i[t'_i], S_j[t'_j]) & \text{, falls } t'_i = t_i + 1 \wedge t'_j = t_j + 1 \\ d(S_i[t'_i], \_) & \text{, falls } t'_i = t_i + 1 \wedge t'_j = t_j \\ d(\_, S_j[t'_j]) & \text{, falls } t'_i = t_i \wedge t'_j = t_j + 1 \end{cases}$$

Praktikabel ist diese Methode schon für kleine Werte von k nicht mehr, da der Aufwand $O(n^k)$ beträgt. Man kann sogar zeigen, dass das multiple Alignment-Problem **NP**-vollständig ist. Damit ist es sehr unwahrscheinlich, dass ein effizienter Algorithmus zur exakten Lösung dieses Problems existiert. Es wurden zahlreiche Heuristiken zur näherungsweisen Lösung des Alignment-Problems entwickelt. Wir beschäftigen uns im folgenden mit zwei Ansätzen. Zunächst geben wir eine Näherungsalgorithmus mit einer garantierten Fehlerschranke an. Danach stellen wir ein *Branch-and-Bound*-Verfahren vor, bei dem man die exakte Lösung bestimmt, jedoch dank der Verwendung von Abschätzungen nicht alle Knoten des Graphen betrachten muss.

Näherungslösungen. Eine häufig benutzte Heuristik zur Erzeugung einer Näherungslösung ist das Alignment von Alignments. Zunächst bildet man paarweise Alignments für jeweils ähnliche Wörter; später fügt man je 2 Alignments zu einem größeren zusammen, wobei die Teil-Alignments nicht mehr geändert werden. Damit wird das Alignment von Alignments zu einem paarweisen Alignment über einem neuen Alphabet. Im Falle von SP-Bewertungen kann man die Alignments durch ihre Profile ersetzen.

Die Qualität der Näherungslösung hängt wesentlich von der Reihenfolge der Alignment-Operationen ab. Häufig wird die Reihenfolge durch einen Baum festgelegt. Dieser Baum widerspiegelt oft entwicklungsbiologische Verwandschaften zwischen den verschiedenen Sequenzen (phylogenetischer Baum). Wir werden nun für einen speziellen Baum zeigen, dass ein Alignment an diesem Baum (für die Levenshtein-SP-Bewertung) eine garantierte Güte erreicht.

Definition 3.12 *Es seien $S_1, S_2, \ldots, S_k$ Wörter über Σ und $D : \Sigma^* \times \Sigma^* \to \mathbb{N}$ der Levenshtein-Abstand. Es sei $i^* \in \{1, 2, \ldots, k\}$ so gewählt, dass*

$$\sum_{j \neq i^*} D(S_{i^*}, S_j) = \min_{1 \leq i \leq k} \left\{ \sum_{j \neq i} D(S_i, S_j) \right\}$$

gilt.

Der zentrale Stern (central star) *für $S_1, S_2, \ldots, S_k$ ist der Graph $G = (V, E)$ mit $V = \{1, 2, \ldots, k\}$ und $E = \{(i^*, j) : j \neq i^*\}$.*

Das Alignment am zentralen Stern geschieht wie folgt. Man bildet zunächst die optimalen paarweisen Alignments von (S_{i^*}, S_j). Diese Alignments haben die Form $(S_{i^*}^{(j)}, S_j')$. Es sei $S_{i^*}^*$ das kürzeste Wort, das aus jedem Wort $S_{i^*}^{(j)}$ durch Einfügen von _ erhalten werden kann. Dann kann man jedes Alignment $(S_{i^*}^{(j)}, S_j')$ durch Einfügen von Paaren $(\_, \_)$ zu einem Alignment der Form $(S_{i^*}^*, S_j^*)$ umwandeln. Wegen $d(\_, \_) = 0$ hat das Alignment $(S_{i^*}^*, S_j^*)$ die gleiche Bewertung wie $(S_{i^*}^{(j)}, S_j')$. Unser gesamtes multiples Alignment ist $(S_1^*, S_2^*, \ldots, S_k^*)$.

Beispiel 3.18 Wir betrachten $S_1 = aaba, S_2 = baaba, S_3 = aaaa, S_4 = aaabb$. Dann gilt $i^* = 1$, und die optimalen paarweisen Alignments mit S_1 sind

$$\begin{array}{ccc} \_\ a\ a\ b\ a & a\ a\ b\ a & a\ a\ b\ \_\ a \\ b\ a\ a\ b\ a & a\ a\ a\ a & a\ a\ b\ b\ a \end{array}.$$

Wir erhalten dann $S_1^* = \_aab\_a$; das Alignment am zentralen Stern ergibt:

$$\begin{array}{c} \_\ a\ a\ b\ \_\ a \\ b\ a\ a\ b\ \_\ a \\ \_\ a\ a\ a\ \_\ a \\ \_\ a\ a\ b\ b\ a \end{array}$$

□

Satz 3.17 *Es sei d die Bewertung des Alignments von $S_1, S_2, \ldots, S_k$ am zentralen Stern. Dann gilt $d \leq 2D(S_1, S_2, \ldots, S_k)$.*

Beweis. O.B.d.A. gelte $i^* = 1$. Das Alignment am zentralen Stern sei $(S_1^*, S_2^*, \ldots, S_k^*)$, das optimale Alignment sei $(S_1', S_2', \ldots, S_k')$.

Es gilt $d = d(S_1^*, S_2^*, \ldots, S_k^*) = \sum_{i<j} d(S_i^*, S_j^*)$. Für $1 < i < j \leq k$ gilt die Dreiecksungleichung $d(S_i^*, S_j^*) \leq d(S_i^*, S_1^*) + d(S_1^*, S_j^*) = D(S_i, S_1) + D(S_1, S_j)$. Damit erhalten wir die Abschätzung

$$d \leq \sum_{1<j\leq k} D(S_1, S_j) + \sum_{1<i<j\leq k} [D(S_1, S_i) + D(S_1, S_j)] = (k-1) \sum_{1<j\leq k} D(S_1, S_j).$$

Andererseits gilt

$$2D(S_1, S_2, \ldots, S_k) = \sum_{i\neq j} d(S_i', S_j') \geq \sum_{i\neq j} D(S_i, S_j) \geq k \sum_{1<j\leq k} D(S_1, S_j)$$

und damit $d \leq \frac{2(k-1)}{k} D(S_1, S_2, \ldots, S_k)$. □

Branch-and-Bound-Verfahren. Die Bestimmung des optimalen Alignment-Pfades erfolgt im Prinzip nach dem DIJSKTRA-Algorithmus zur Suche nach dem kürzesten Weg von Knoten $\vec{0} = (0, \ldots, 0)$ nach Knoten $\vec{N} = (n_1, n_2, \ldots, n_k)$. Dabei wird allerdings ein Knoten, über den der minimale Pfad garantiert nicht führen kann, von der weiteren Betrachtung ausgeschlossen. Dazu benutzen wir folgende untere Abschätzung $A(v)$ für die Länge des kürzesten Weges von $v = (v_1, v_2, \ldots, v_k)$ nach $\vec{N}$:

$$\begin{aligned} A(v) &= \sum_{1\leq i<j\leq k} A_{i,j}(v_i, v_j) \text{ mit} \\ A_{i,j}(v_i, v_j) &:= D(S_i[v_i + 1 \ldots n_i], S_j[v_j + 1 \ldots n_j]) \\ &= D(S_i^r[1 \ldots n_i - v_i], S_j^r[1 \ldots n_j - v_j]). \end{aligned}$$

Für ein Paar (i, j) können alle Werte $A_{i,j}(v_i, v_j)$, $0 \leq v_i \leq n_i, 0 \leq v_j \leq n_j$, durch dynamische Programmierung mit einem Aufwand von $O(n_i n_j)$ berechnet werden. Dies geschieht in einem Präprozessing mit einem Gesamtaufwand von $O(k^2 n^2)$, wobei $n = \max\{n_1, n_2, \ldots, n_k\}$. Für einen Knoten v kann dann $A(v)$ durch Aufsuchen der Werte $A_{i,j}(v_i, v_j)$ mit einem Aufwand von $O(k^2)$ bestimmt werden.

Ist $D(v)$ die Länge des kürzesten Weges von $\vec{0}$ nach v, so kann der kürzeste Weg von $\vec{0}$ nach $\vec{N}$ nicht über v führen, wenn $D(v) + A(v) > M$ gilt, wobei M eine obere Schranke für die Länge des kürzesten Wegen von $\vec{0}$ nach $\vec{N}$ ist. Eine solche Schranke wird üblicherweise durch eine Näherungslösung ermittelt (z.B. Alignment am zentralen Stern) und im Laufe des Algorithmus durch wiederholte Berechnung von Näherungslösungen verschärft.

Im Algorithmus speichern wir eine Menge S von Knoten, für die der optimale Weg von $\vec{0}$ bereits bekannt ist, sowie eine Menge Q von Knoten, die Nachfolger von Knoten aus S sind. Zu jedem Knoten $v \in S \cup Q$ speichert man den Wert $d(v)$ des bisher besten Weges von $\vec{0}$ nach v. Initialisiert wird S mit $\emptyset$, Q mit $\{\vec{0}\}$ und $d(\vec{0})$ mit 0. In einem Schritt entfernt man aus Q den Knoten v mit minimalem Wert $d(v)$ und fügt v in S ein. Mit jedem Nachfolger w von v verfährt man folgendermaßen.

- Gilt $w \in Q$, so setzt man $d(w) \leftarrow \min\{d(w), d(v) + d(v, w)\}$.
- Gilt $w \notin Q \cup S$ und gilt $d(v) + d(v, w) + A(w) \leq M$, so setzt man $d(w) \leftarrow d(v) + d(v, w)$ und fügt w zu Q hinzu.

- Sollte außerdem $w = \vec{N}$ gelten, so aktualisiert man in beiden Fällen $M \leftarrow d(w)$.
- In den anderen Fällen ($w \in S$ oder $w \notin Q \cup S \wedge d(v) + d(v,w) + A(w) > M$) ändert man nichts.

Der Algorithmus terminiert, wenn Q leer ist.

Branch-and-Bound ist ein Standard-Verfahren zur Lösung schwieriger Optimierungsprobleme. Oft wird die optimale Lösung nicht in angemessener Zeit gefunden. Durch die Abschätzungen nach unten ($A(v)$-Werte) sowie nach oben (Näherungslösungen) kann man aber oft recht gut den Wert des optimalen Alignments abschätzen.

Weitere Methoden. Von den zahlreichen weiteren Verfahren zur Ermittlung von Näherungslösungen skizzieren wir kurz zwei sehr wichtige:

- Motif-Suche: Es wird eine charakteristische Teil-Sequenz *(motif, anchor, block)* gesucht, die in allen Wörtern inexakt vorkommt. Für die Fundstellen bildet man das Alignment. Dadurch wird jedes Wort in zwei Teile zerlegt. In diesen Teilen sucht man erneut nach Motiven.
- Hidden Markov Modelle: Es wird zunächst ein Profil als Startalignment bestimmt. Mit diesem Profil bildet man Alignments mit den einzelnen Sequenzen. Dabei entsteht ein neues Profil. Dieser Prozess wird fortgesetzt, bis sich das Profil nicht mehr ändert.

Kapitel 4

Indexstrukturen für Texte

4.1 Einführung

Bei allen bisher betrachteten Suchalgorithmen wurde das Suchwort einem Präprozessing unterzogen, der Text aber nicht. In der Suchphase war es deshalb nötig, den gesamten Text zu betrachten. Im besten Fall ergab sich eine Laufzeit von $O(n/m)$ bei Textlänge n und Suchwortlänge m. Im Falle großer Text-Datenbanken (Internet, genetische Datenbanken) mit vielen Anfragen in einem *unveränderlichen* Text ist eine solche Laufzeit zu hoch.

Wir stellen uns deshalb das folgende Problem. Gegeben ist ein sehr langer und unveränderlicher Text T der Länge n. Ziel ist es, den Text so aufzuarbeiten, dass die Zeit für die Suche nach einem Wort P (oder nach einem verallgemeinerten Suchmuster) unabhängig oder höchstens logarithmisch abhängig von n ist. In gewisser Weise ist diese Aufarbeitung vergleichbar mit einem *Index* in einem Buch, der für wichtige Stichwörter Verweise auf die zugehörigen Seiten des Buches gibt. Ein Index für die Suche kostet zusätzlichen Platz. Dieser zusätzliche Bedarf sollte höchstens linear in der Textlänge n sein. Auch sollte der Zeitaufwand für die Konstruktion in der Größenordnung $O(n)$ bzw. $O(n \log n)$ liegen.

In natürlichsprachlichen Texten sucht man häufig nicht nach beliebigen Zeichenketten, sondern nur nach *Folgen von Wörtern* (die durch spezielle Zeichen voneinander getrennt sind). Für solche Texte ist es möglich, einen so genannten *partiellen Index* anzulegen, der wesentlich weniger Platz als ein vollständiger Index benötigt.

Dieses Kapitel ist wie folgt gegliedert. In Abschnitt 4.2 werden einige Datenstrukturen für die Indizierung vorgestellt. Neben den bekanntesten (und für die Theorie sehr angenehmen) Strukturen *Suffixbaum* und *Suffix-Array* geht es dabei auch um Strukturen für die partielle Indizierung. Es wird kurz erläutert, wie das Problem der exakten Suche effizient mit Hilfe der einzelnen Strukturen gelöst werden kann.

In den Abschnitten 4.3 und 4.4 werden einige effiziente Algorithmen zur Konstruktion von Suffixbäumen bzw. von Suffix-Arrays besprochen.

Schließlich beschäftigt sich Abschnitt 4.5 mit weiteren Anwendungen von Indexstrukturen, wie z.B. der Suche nach Wiederholungen oder Anwendungen in Kompressionsalgorithmen.

4.2 Datenstrukturen für die Indizierung

Im folgenden sei S ein Wort der Länge n über dem Alphabet Σ der Mächtigkeit σ. Außerdem sei $\# \notin \Sigma$ ein Sondersymbol (Textende). Mit S_i, $1 \leq i \leq n+1$, bezeichnen wir das Suffix

$S[i \ldots n]$. Wir gehen davon aus, dass auf $S \cup \{\#\}$ eine totale Ordnung definiert ist, wobei $\#$ das kleinste Element bezüglich dieser Ordnung ist.

Suffixbäume

Die Grundidee ist, den Suchwort-Baum für die Menge der Suffixe des Textes $S\#$ zu konstruieren. Durch das Sondersymbol ist gewährleistet, dass kein Suffix von $S\#$ das Präfix eines anderen Suffixes von $S\#$ ist, d.h. dass es für jedes Suffix α von S ein Blatt mit der Pfad-Beschriftung $\alpha\#$ gibt. Das Blatt für das i-te Suffix ist mit i beschriftet.

Definition 4.1 *Es sei $S \in \Sigma^*$ ein Wort mit $|S| = n$ und $\# \notin \Sigma$ ein Sonderzeichen. Der* Suffix-Suchwortbaum (suffix trie) *von S ist $Trie(S_1\#, S_2\#, \ldots, S_n\#, \#)$.*

Der Suchwort-Baum wird jedoch nicht selbst benutzt, da er eine quadratische Anzahl von Knoten besitzt. Verantwortlich dafür sind innere Knoten mit dem Ausgangsgrad 1. Deshalb wird der Suchwort-Baum modifiziert — es entsteht der Suffixbaum mit einer linearen Anzahl von Knoten.

Definition 4.2 *Es sei $S \in \Sigma^*$ ein Wort und $\# \notin \Sigma$ ein Sonderzeichen. Der* Suffixbaum (suffix tree) *$\mathcal{T}(S)$ von S entsteht, indem man im Suffix-Trie von S jeden maximalen Weg, dessen innere Knoten einen Ausgangsgrad von 1 besitzen, durch eine Kante mit der Beschriftung dieses Weges ersetzt.*

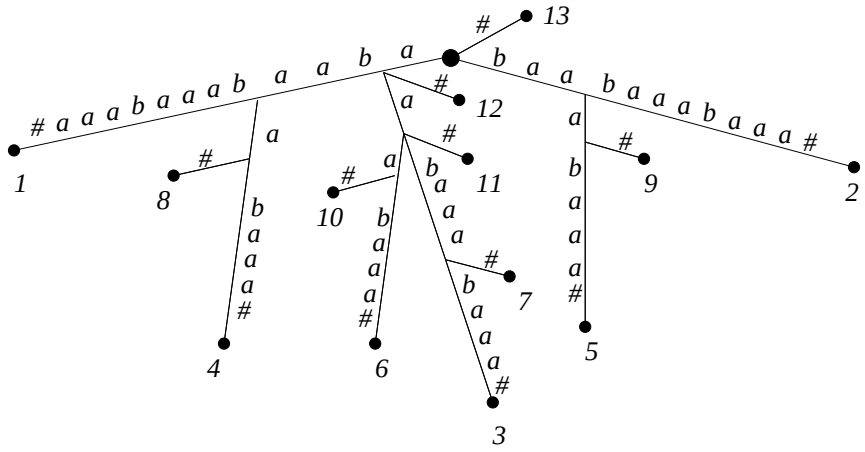
Lemma 4.1 *Die Zahl der Knoten von $\mathcal{T}(S)$ ist höchstens $2n$ mit $n = |S|$.*

Beweis. Wir konstruieren $\mathcal{T}(S)$, indem wir die Suffixe $S_1\#, S_2\#, \ldots, S_n\#, \#$ nacheinander einfügen. Das Einfügen des i-ten Suffixes erfolgt wie bei der Konstruktion des Suchwort-Baumes, indem man einen Pfad mit der Beschriftung $S[i \ldots n]$ sucht. Es gibt für jedes i eine Zahl t mit $i \leq t \leq n+1$, so dass ein Pfad mit der Beschriftung $S[i \ldots t-1]$, jedoch kein Pfad mit der Beschriftung $S[i \ldots t]$ existiert. Am Ende des Pfades mit der Beschriftung $S[i \ldots t-1]$ wird eine neue Kante mit der Beschriftung $S[t \ldots n]\#$ eingefügt. Eventuell wird dabei eine bestehende Kante in zwei Kanten geteilt. Damit entstehen für jedes Suffix ein neues Blatt und höchstens ein neuer innerer Knoten. □

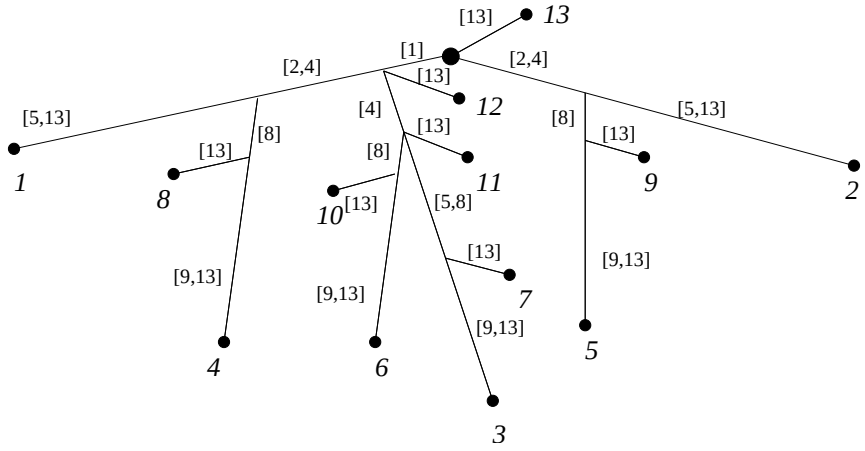
Um nicht nur die Knotenzahl, sondern auch die Gesamtgröße des Suffixbaumes linear zu beschränken, müssen wir die Kantenbeschriftungen komprimieren. Da die Beschriftung jeder Kante ein Infix von S ist, ersetzen wir eine Beschriftung α durch ein Paar $[i, j]$ mit $S[i \ldots j] = \alpha$. Damit hat jede Kante eine Beschriftung konstanter Länge. In unseren Beispielen werden wir für die besserer Übersichtlichkeit jedoch in der Regel weiterhin die explizite Beschriftung verwenden.

In der Praxis ist die komprimierte Beschriftung durchaus problematisch. Zum einen ist es zur Bestimmung einer Kantenbeschriftung notwendig, auf den Text zurückzugreifen; dies kostet viel Zeit, wenn der Text nicht in den Hauptspeicher passt und für das Aufsuchen der entsprechenden Textstellen auf externen Speicher zugegriffen werden muss. Zum anderen kommen lange Kantenbeschriftungen hauptsächlich nur an Kanten zu den Blättern vor. Damit empfiehlt sich für eine Implementierung von Suffixbäumen eine gemischte Strategie: Kanten zu internen Knoten beschriftet man explizit, während Kanten zu den Blättern durch Indizes bezeichnet werden.

Beispiel 4.1 Der Suffixbaum für $S = abaabaaabaaa$ sieht wie folgt aus:



Mit komprimierter Beschriftung ergibt sich folgendes Aussehen (statt $[i, i]$ schreiben wir $[i]$):



□

Die exakte Suche nach einem Wort P der Länge m in S ist ganz einfach. P kommt nämlich genau dann in S vor, wenn $\mathcal{T}(S)$ einen Pfad mit der Beschriftung P enthält. Die Positionen der Vorkommen von P ermittelt man, indem man alle Blätter unterhalb des Pfades mit der Beschriftung P bestimmt (z.B. durch Tiefensuche). Der Aufwand (ohne Berücksichtigung der Alphabetgröße σ) beträgt $O(m)$ für die Suche nach dem Pfad P und $O(A)$ für die Bestimmung der Blätter, wobei A die Anzahl der Blätter und damit der Vorkommen ist.

Beispiel 4.2 Mit Hilfe des Suffixbaumes von $S = abaabaaabaaa$ aus Beispiel 4.1 stellen wir unter anderem fest:

- Es existiert kein Pfad bb, d.h. das Wort bb kommt nicht in S vor.
- Es existiert ein Pfad aba, d.h. das Wort aba kommt in S vor. Als Vorkommen ergeben sich anhand der Blätter $\{1, 4, 8\}$.

□

Suffix-Arrays

Eine Alternative zum Suffixbaum stellt das *Suffix-Array* dar. Dieses gibt einfach die lexikographische Reihenfolge der Suffixe des zu indizierenden Textes S an. Das Suffix-Array ist einfach zu implementieren, unabhängig von der Alphabetgröße und benötigt üblicherweise ca. ein Viertel des Platzes eines Suffixbaumes.

Definition 4.3 *Es sei $S \in \Sigma^*$ mit $|S| = n$. Das* Suffix-Array *von S ist das $(n+1)$-dimensionale Feld A_S mit $A_S[i] = j$ genau dann, wenn S_j das lexikographisch i-te Suffix von S ist.*

Insbesondere gilt immer $A_S[i] = |S| + 1$, da das leere Wort das lexikographisch kleinste Suffix ist.

Beispiel 4.3 Für $S = abaabaaabaaa$ gilt $A_S = (13, 12, 11, 10, 6, 7, 3, 8, 4, 1, 9, 5, 2)$.
Für $S = mississippi$ gilt $A_S = (12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3)$. □

Die exakte Suche nach einem Wort P der Länge m in S erfolgt mit Hilfe des Suffix-Arrays von S durch *binäre Suche*. P kommt an der Position k genau dann vor, wenn die ersten m Zeichen von S_k mit P übereinstimmen. Die Vorkommen von P in S bilden damit ein zusammenhängendes Intervall des Suffix-Arrays A_S. Bei der binären Suche vergleicht man im ersten Schritt das Suchwort P mit dem Suffix S_k, das durch die mittlere Position in A_S gegeben ist. Ist P lexikographisch größer als die ersten m Zeichen von S_k oder ist S_k ein Präfix von P, so setzt man die Suche im rechten Teil des Arrays fort. Ist P lexikographisch kleiner als die ersten m Zeichen von S_k, so sucht man im linken Teil des Arrays weiter. Stimmt P mit den ersten m Zeichen von S_k überein, so gibt es ein Vorkommen von P, und man hat einen Punkt im Intervall zu P gefunden. Die Endpunkte des Intervalls kann man wiederum durch binäre Suche bestimmen. Der Aufwand beträgt im schlechtesten Fall $O(m \log n)$. Es gibt Algorithmen, die das Problem der exakten Suche mit einem geringeren Aufwand lösen ($O(m + \log n)$ bzw. $O(m)$). Diese benötigen allerdings zusätzliche Arrays der Größenordnung $\Theta(n)$.

Beispiel 4.4 Es sei $S = abaabaaabaaa$ mit dem Suffix-Array $A_S = (13, 12, 11, 10, 6, 7, 3, 8, 4, 1, 9, 5, 2)$. Die binäre Suche nach bb bzw. aba in S läuft wie folgt ab. Dabei ist i der im Suffix-Array betrachtete Index.

Suche nach bb:

1. $i = 7$; $A_S[i] = 3$; $S[3 \ldots 4] = aa < bb \rightarrow$ Suche rechts.
2. $i = 10$; $A_S[i] = 1$; $S[1 \ldots 2] = ab < bb \rightarrow$ Suche rechts.
3. $i = 12$; $A_S[i] = 5$; $S[5 \ldots 6] = ba < bb \rightarrow$ Suche rechts.
4. $i = 13$; $A_S[i] = 2$; $S[2 \ldots 3] = ba < bb \rightarrow bb$ kommt nicht vor.

Suche nach aba:

1. $i = 7$; $A_S[i] = 3$; $S[3 \ldots 5] = aab < aba \rightarrow$ Suche rechts.
2. $i = 10$; $A_S[i] = 1$; $S[1 \ldots 3] = aba = aba \rightarrow aba$ kommt vor.
   Es folgt die Suche nach der linken Intervall-Grenze für aba:

3. $i = 8$; $A_S[i] = 8$; $S[8 \dots 10] = aba = aba \rightarrow$ linke Grenze 8.
   Es folgt die Suche nach der rechten Intervall-Grenze für aba:

4. $i = 12$; $A_S[i] = 5$; $S[5 \dots 7] = baa > aba \rightarrow$ Suche links.

5. $i = 11$; $A_S[i] = 9$; $S[9 \dots 11] = baa > aba \rightarrow$ linke Grenze 10.

Das zu aba gehörige Intervall des Suffix-Arrays ist damit $[8, 10]$, entsprechend den Vorkommen $\{8, 4, 1\}$. □

Man beachte, dass die exakte Suche im Suffixbaum einen Knoten liefert, während sich im Suffix-Array ein Intervall ergibt. Allgemein kann man viele Algorithmen von Suffixbäumen auf Suffix-Arrays übertragen, indem man Knoten im Baum durch die entsprechenden Intervalle im Array ersetzt. Im Zeitaufwand ergibt sich ein zusätzlicher Faktor von $O(\log n)$.

Array der Listen der q-gramme

Eine weitere Möglichkeit der vollständigen Indizierung stellt das Array der Listen der q-gramme dar. Dazu ermittelt man einfach für jedes Teilwort α der Länge q die Vorkommen von α im Text.

Definition 4.4 *Es seien $S \in \Sigma^*$ mit $|\Sigma| = \sigma$, $|S| = n$ und $q \in \mathbb{N}$. Das* Array der Listen der q-gramme *von S ist das σ^q-dimensionale Feld L_S, wobei $L_S[\alpha]$ für jedes $\alpha \in \Sigma^q$ die Liste der Vorkommen von α in S in ihrer natürlichen Reihenfolge ist.*

Beispiel 4.5 Für $\Sigma = \{a, b\}$, $S = abaabaaabaaa$ und $q = 2$ ergibt sich folgendes Array der Listen der q-gramme:
$L_S(aa) = (3, 6, 7, 10, 11), L_S(ab) = (1, 4, 8), L_S(ba) = (2, 5, 9), L_S(bb) = \emptyset$. □

Die Konstruktion des Arrays ist sehr einfach in Linearzeit in einem Gang über den Text möglich. Für die exakte Suche nach einem Wort P beschränken wir uns der Einfachheit halber auf den Fall dass die Länge von P ein Vielfaches von q ist, also $|P| = m = qr$ gilt. Man zerlegt P in r Teilwörter jeweils der Länge q, d.h. $P = P_1 P_2 \cdots P_r$ und betrachtet die Listen $L_S(P_1), L_S(P_2), \dots L_S(P_r)$. Dann kommt P an der Stelle k genau dann vor, wenn $k + (i-1)q \in L_S(P_i)$ für alle $1 \leq i \leq r$ gilt. Um die Vorkommen von P zu finden, muss man jedes Element der Listen höchstens einmal betrachten. Der Zeitaufwand beträgt damit $O(\sum_{i=1}^{r} |L_S(P_i)|)$.

Beispiel 4.6 Für $\Sigma = \{a, b\}$, $S = abaabaaabaaa$ und $q = 2$ ergibt sich folgende Suche nach dem Wort $P = aaba = aa \cdot ba$:
$L_S(aa) = (3, 6, 7, 10, 11), L_S(ba) = (2, 5, 9)$. Für jede Position $k \in L_S(aa)$ muss geprüft werden, ob $k + 2 \in L_S(ba)$ gilt.
Dabei ergibt sich: $5 \in L_S(ba), 8 \notin L_S(ba), 9 \in L_S(ba), 12 \notin L_S(ba), 13 \notin L_S(ba)$. Die Vorkommen sind damit $3, 7$. □

Die Suche erscheint im Vergleich zur Suche in Suffixbäumen oder Suffix-Arrays ziemlich umständlich und langsam. Aber das Array der Listen der q-gramme besitzt zwei wesentliche Vorteile gegenüber Suffixbäumen und Suffix-Arrays. Zum einen benötigt es einen geringeren Speicherplatz als ein Suffix-Array. Zwar ist die Anzahl der zu speichernden Zahlen in etwa gleich, aber die Folgen in den einzelnen Arrays sind monoton wachsend, so dass man statt der

tatsächlichen Werte die Differenzen zweier Nachbarn speichern kann. Dies hat zur Folge, dass wesentlich kleinere Zahlen zu speichern sind, die zu ihrer Darstellung weniger Bits benötigen und auch besser komprimiert werden können. Zum anderen kann man die Suche nach den Vorkommen von P erledigen, indem man nur auf die für P relevanten Listen zugreift. Eine explizite Betrachtung des Textes in der Suchphase ist im Gegensatz zu Suffixbäumen und Suffix-Arrays nicht nötig. Dies ist insbesondere von Vorteil, wenn der Text so groß ist, dass er nicht in den Hauptspeicher passt und die Zahl der Zugriffe auf den Sekundärspeicher (z.B. Festplatte) möglichst klein gehalten werden soll.

Partielle Indizes

In den bisher betrachteten Datenstrukturen fand sich für jede Position des Textes ein Verweis (vollständiger Index). Damit beanspruchen die Indizes einen größeren Platz als der Text selbst. Dies ist häufig nicht akzeptabel, und oft ist ein voller Index auch nicht nötig. Zum Beispiel wird man sich bei Texten in natürlichen Sprachen nur für solche Positionen interessieren, an denen ein Wort beginnt. Die Größe eines zugehörigen partiellen Indexes ist damit linear in der Anzahl der Wörter. Die Platzersparnis ist gewaltig. Üblicherweise benötigt man nur ein Sechstel bis ein Fünftel des Platzes wie für einen vollständigen Index. Datenstrukturen sind der *partielle Suffixbaum* sowie das *partielle Suffix-Array*.

Beispiel 4.7 Wir betrachten den folgenden Satz. Die Anfänge von Wörtern (Index-Stellen) sind markiert.

```
brautkleid bleibt brautkleid und blaukraut bleibt blaukraut
1          12     18          29  33         43     49
```

Die Wörter in lexikographischer Reihenfolge sind: `blaukraut, bleibt, brautkleid, und`
Es ergibt sich folgendes partielles Suffix-Array: $(49, 33, 43, 12, 1, 18, 29)$. □

Invertierte Indizes

Eine weitere Möglichkeit der Indizierung strukturierter Texte (z.B. in natürlicher Sprache) besteht darin, einfach für jedes Wort eine Liste seiner Vorkommen aufzustellen.

Beispiel 4.8 Für den Satz in Beispiel 4.7 erhalten wir folgende Vorkommen der einzelnen Wörter.

```
blaukraut  33, 49
bleibt     12, 43
brautkleid 1, 18
und        29
```

□

Der volle invertierte Index enthält dabei alle Vorkommen jedes Wortes. Häufig besteht eine Textdatenbank aus vielen einzelnen Dateien (Beispiel: Internet-Suchmaschinen). Dann enthält der Index häufig lediglich für jedes Wort den Verweis auf die Dateien, in denen es vorkommt. Bei einer Suchanfrage kann man damit sehr schnell ermitteln, welche Dateien relevant sind. Für eine exakte Suche müssen die betreffenden Dateien geladen und mit den herkömmlichen Methoden durchsucht werden.

4.3 Konstruktion von Suffixbäumen

In diesem Abschnitt werden wir im wesentlichen den Algorithmus von McCreight aus dem Jahr 1976 besprechen, der den Suffixbaum zu einem Text in linearer Zeit konstruiert. Die Idee dieses Algorithmus ist, dass man die einzelnen Suffixe des Textes S, beginnend mit dem ersten und längsten in einen ursprünglich leeren Baum einfügt. Um die lineare Laufzeit zu erreichen, muss die besondere Struktur des Suffixbaumes ausgenutzt werden. Insbesondere macht man Gebrauch von der Tatsache, dass für jedes Symbol x und jedes Wort α mit dem Teilwort $x\alpha$ auch das Teilwort α in S vorkommt.

Wir führen zunächst einige notwendige technische Definitionen ein, formalisieren dann den naiven Algorithmus, entwickeln dann den Algorithmus von McCreight und geben einen Beweis für dessen lineare Laufzeit. Am Ende dieses Abschnittes besprechen wir kurz den historisch ersten Linearzeit-Algorithmus von Weiner (1973) sowie den Online-Algorithmus von Ukkonen (1992).

Technische Definitionen

Wir beginnen mit einigen technischen Definitionen für Suffixbäume bzw. allgemeiner für Bäume, deren Kanten mit Wörtern beschriftet sind.

- Eine *Position* im Suffixbaum ist entweder ein Knoten oder ein Paar (e, i), wobei $e = (v, \alpha, w)$ eine Kante ist und i eine Zahl mit $1 \leq i < |\alpha|$ ist. Im zweiten Fall sagen wir, dass sich die Position *im Inneren der Kante* e befindet.

- Der graphentheoretische Begriff *Pfad* wird erweitert, indem man als Anfangs- bzw. Endpunkt auch Positionen zulässt, während die inneren Punkte nur Knoten sind. Die *Beschriftung* eines Pfades erhält man durch Konkatenation der Beschriftungen der Kanten, wobei Kanten von der Anfangs- bzw. Endposition nur teilweise eingehen.

  Formal definieren wir:

  1. Es sei $e = (v, \alpha, w)$ eine Kante mit $|\alpha| = m$. Dann sind
     $(e, i) \to (e, j)$ mit $1 \leq i < j < m$ bzw. $(e, i) \to w$ mit $1 \leq i < m$ bzw.
     $v \to (e, i)$ mit $1 \leq i < m$ bzw. $v \to w$
     Pfade der Länge 1 mit den Beschriftungen
     $\alpha[i+1 \ldots j]$ bzw. $\alpha[i+1 \ldots m]$ bzw. $\alpha[1 \ldots i]$ bzw. α.
  2. Sind $p_0 \to v_1 \to v_2 \to \ldots \to v_k$ und $v_k \to p_{k+1}$ Pfade der Länge k bzw. 1 mit den Beschriftungen α bzw. β (wobei $v_1, \ldots, v_k$ Knoten sind), so ist $p_0 \to v_1 \to v_2 \to \ldots \to v_k \to p_{k+1}$ ein Pfad der Länge $k+1$ mit der Beschriftung $\alpha\beta$.

- Offensichtlich ist ein Pfad eindeutig durch seine Startposition p und seine Beschriftung β bestimmt. *Notation*: Pfad β von Position p.

- Eine *Position* wird durch die Beschriftung β des Pfades von der Wurzel identifiziert. *Notation*: Position β.

- Ist p eine Position mit der Beschriftung $\beta \in \Sigma^*$ und existiert eine Position q mit der Beschriftung βx mit $x \in \Sigma$, so sagen wir dass von p eine *Fortsetzung* mit x existiert; Notation $q = \mathit{Next}(p, x)$.

- Für eine Position p bezeichnen wir die Länge des Pfades von der Wurzel nach p als *Tiefe* von p, Notation $depth(p)$, und die Länge der Beschriftung $\mathcal{L}(p)$ als *String-Tiefe* von p.

Naiver Algorithmus

Analog zur Konstruktion des Suchwort-Baumes ergibt sich folgender naiver Algorithmus zur Konstruktion von Suffixbäumen, der bereits im Beweis von Lemma 4.1 beschrieben wurde. Dabei gibt p die aktuelle Position im Suffixbaum an.

Algorithmus 4.1 Konstruktion des Suffixbaumes (Naiver Algorithmus)

Eingabe: Wort S mit $|S| = n$
Ausgabe: Suffixbaum von S
(1) $\mathcal{T} \leftarrow (\{root\}, \emptyset)$;
(2) **for** $i \leftarrow 1$ **to** $n+1$
(3) $t \leftarrow i$; $p \leftarrow root$;
(4) **while** (von p gibt es eine Fortsetzung mit $S[t]$)
(5) $p \leftarrow Next(p, x)$; $t \leftarrow t+1$;
(6) Füge bei p eine neue Kante mit der Beschriftung $S[t \ldots n+1]$ zu einem neuen Blatt mit der Beschriftung i ein;
(7) **return** $\mathcal{T}$

Das *Einfügen* der neuen Kante in Algorithmus 4.1 geschieht folgendermaßen.

Ist p ein Knoten v, so werden ein neues Blatt b und die Kante $(v, S[t \ldots n+1], b)$ eingefügt. Die komprimierte Kantenbeschriftung lautet $[t, n+1]$.

Liegt dagegen p im Inneren einer Kante, d.h. $p = (e, j)$ mit $e = (v, \alpha, w)$, so wird ein neuer Knoten u zwischen v und w eingefügt und e durch $(v, \alpha[1 \ldots j], u)$ und $(u, \alpha[j+1 \ldots |\alpha|], v)$ ersetzt. Außerdem werden das neue Blatt b und die Kante $(u, S[t \ldots n+1], b)$ eingefügt. Benutzt man die komprimierte Beschriftung, so hat die ersetzte Kante e die Gestalt $(v, [l, r], w)$, und die neuen Kanten sind $(v, [l, l+j-1], u)$, $(u, [l+j, r], w)$ sowie $(u, [t, n+1], b)$.

Die Laufzeit des naiven Algorithmus ist $O(n^2)$, da das Einfügen des Suffixes $S_i\#$ im schlechtesten Fall $O(n-i)$ Schritte kostet. Die mittlere Laufzeit ist in der Größenordnung $\Theta(n \log n)$.

Algorithmus von McCreight

Wie beim naiven Algorithmus werden der Reihe nach die Suffixe $S_1\#, S_2\#, \ldots, S_n\#, \#$ eingefügt. Die Laufzeit wird verkürzt, indem einige Vergleiche eingespart werden. Im folgenden sei $\mathcal{T}_i(S)$ der Baum nach dem Einfügen von $S_1\#, S_2\#, \ldots, S_i\#$. $\mathcal{T}_0(S)$ besteht nur aus der Wurzel. Mit $t_i(S)$ bezeichnen wir die Stelle in S, für die beim Einfügen von $S_i\#$ keine Fortsetzung im Baum $\mathcal{T}_{i-1}(S)$ gefunden wird. Entscheidend für den Algorithmus von McCreight ist das folgende Lemma.

Lemma 4.2 *Es sei S ein Wort der Länge n. Gibt es für $1 \le i \le n+1$ in $\mathcal{T}_{i-1}(S)$ einen Pfad $x\alpha$, so gibt es in $\mathcal{T}_i(S)$ einen Pfad α.*

Beweis. Ein Pfad $x\alpha$ ist genau dann in $\mathcal{T}_{i-1}(S)$, wenn $S[j \ldots j+|\alpha|] = x\alpha$ für ein $j \le i-1$ gilt. Dann gilt aber $S[j+1 \ldots j+|\alpha|] = \alpha$, und damit ist der Pfad α in $\mathcal{T}_i(S)$. □

Aus dem Lemma ergeben sich einige wichtige Folgerungen.

Folgerung 4.3 *Es sei S ein Wort der Länge n.*

1. *Gibt es in $\mathcal{T}_{i-1}(S)$ einen Knoten $x\alpha$, so gibt es in $\mathcal{T}_i(S)$ einen Knoten α.*
2. *Es gilt $t_{i-1}(S) \leq t_i(S)$ für alle $1 \leq i \leq n+1$.*

Der Algorithmus von McCreight benutzt zwei Techniken, die im folgenden vorgestellt werden.

Skip/Count-Trick

Wenn im Voraus bekannt ist, dass der Pfad β im Baum enthalten ist, so braucht man bei der Suche nach der Endposition von β nicht für jede Position, sondern nur für die *Knoten* auf dem Pfad nach der Fortsetzung zu suchen. Befindet man sich an einem Knoten v und in β an der Stelle t, so bestimmt man die eindeutige Kante (v, α, w) mit $\alpha[1] = \beta[t]$, lässt die restlichen Vergleiche entlang der Kante aus (*skip*) und setzt im Knoten w und an der Stelle $\beta[t + |\alpha|]$ (*count*) fort. Genauer formulieren wir die Skip/Count-Suche wie folgt.

Skip/Count-Suche
Eingabe: Suffixbaum $\mathcal{T}$ (im Aufbau), Knoten v, $\beta \in \Sigma^*$, $|\beta| = m$,
Pfad von v mit Beschriftung β existiert
Ausgabe: Endposition des Pfades β ab v
SKIP-COUNT-SEARCH$(\mathcal{T}, v, \beta)$

```
(1)   t ← 1; u ← v;
(2)   while t ≤ m
(3)     Bestimme Kante e = (u, α, w) mit α[1] = β[t];
(4)     if t + |α| = m + 1 then return w;
(5)     if t + |α| > m + 1 then return (e, m − t + 1);
(6)     if t + |α| ≤ m then u ← w; t ← t + |α|;
```

Ist also vor der Suche bekannt, dass der Pfad β existiert, so kann durch die Skip/Count-Suche das Ende des Pfades β in einer Zeit gefunden werden, die linear in $depth(\beta)$ ist (statt linear in $|\beta|$). Zum Beispiel lässt sich die Endposition des Pfades *baabaaaba* im Suffixbaum aus Beispiel 4.1 in 2 Schritten finden.

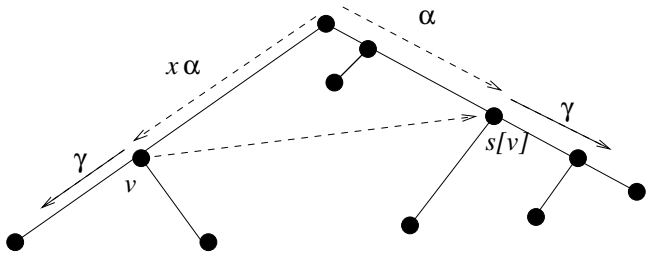
Für die Konstruktion des Suffixbaumes ist von Bedeutung, dass vor dem Einfügen von $S_i\#$ das Wort $S[i \ldots t_{i-1} - 1]$ in $\mathcal{T}_{i-1}$ enthalten ist, also die Skip/Count-Suche anwendbar ist.

Suffix-Links

Definition 4.5 *Es sei v ein Knoten in $\mathcal{T}(S)$ bzw. in $\mathcal{T}_i(S)$ mit der Pfad-Beschriftung $x\alpha$, $x \in \Sigma, \alpha \in \Sigma^*$. Gibt es einen Knoten w mit der Pfad-Beschriftung α, so nennen wir w das* Suffix-Link *von v, Bezeichnung $s[v]$.*

Suffix-Links können genutzt werden, um die Suche abzukürzen, wenn man von einer Position $x\alpha\gamma$ mit $x \in \Sigma$ nach der Position $\alpha\gamma$ sucht. Gibt es nämlich einen Knoten v mit der Pfadbeschriftung $x\alpha$, der ein Suffix-Link (mit Beschriftung α)besitzt, so kann man einfach

von $s[v]$ aus den Pfad mit der Beschriftung γ suchen. Diese Situation tritt beim Aufbau des Suffixbaumes auf.



Lemma 4.4 *Entsteht beim Einfügen von $S_{i-1}\#$ ein neuer innerer Knoten u, so besitzt u nach dem Einfügen von $S_i\#$ ein Suffix-Link.*

Beweis. Der Knoten u hat eine Pfadbeschriftung $x\beta$. Nach Folgerung 4.3 existiert nach dem Einfügen von $S_i\#$ ein Knoten mit der Pfadbeschriftung β, d.h. das Suffix-Link von u. Man beachte, dass dieser Knoten beim Einfügen von $S_i\#$ erreicht wird bzw. entsteht. Damit kostet das Setzen des Suffix-Links von u nur konstanten Aufwand. □

Wir können nun die $(i+1)$-te Phase des Algorithmus von McCreight formulieren:

1. Gilt $t_i = i$, so bestimme t_{i+1} durch zeichenweisen Vergleich ab der Wurzel und der Position $i+1$ in S.

2. Anderenfalls sei $S[i \ldots t_i - 1] = x_i\beta_i$; u_i sei der Knoten mit der Pfad-Beschriftung $x_i\beta_i$; v_i sei der letzte Knoten auf dem Pfad nach u_i, der schon in $\mathcal{T}_{i-1}$ vorhanden war; die Pfad-Beschriftung von v_i sei $x_i\alpha_i$; die Beschriftung der Kante von v_i nach u_i sei γ_i.

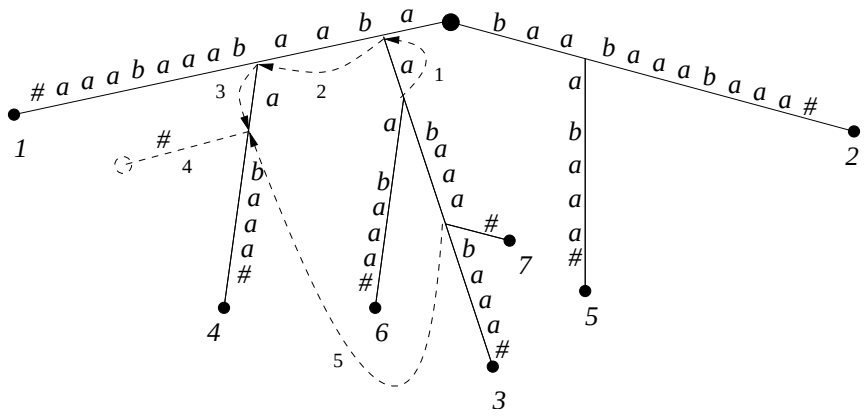
   Insbesondere besitzt v_i ein Suffix-Link in $\mathcal{T}_i(S)$; und es gilt $u_i \neq v_i$ genau dann, wenn u_i neu entstanden ist. Wir können nun das Ende des Pfades $S[i+1 \ldots t_i - 1] = \beta_i$ (gleichzeitig das Suffix-Link von u_i) wie folgt finden.

   (a) Ist v_i die Wurzel, so gilt $\beta_i = \gamma_i = S[i]\gamma_i'$. Suche von der Wurzel nach γ_i' mittels Skip/Count-Suche.

   (b) Ist v_i nicht die Wurzel, so suche von $s[v_i]$ aus den Pfad γ_i mittels Skip/Count-Suche.

   Wir bestimmen dann t_{i+1} durch zeichenweisen Vergleich ab der Position β_i im Baum und der Stelle t_i in S.

Beispiel 4.9 Wir betrachten die 8. Phase der Konstruktion des Suffixbaumes von *abaabaaabaaa*. In der 7. Phase wurde die neue Kante an dem neuen Knoten *aabaaa* eingefügt. Der letzte Knoten in $\mathcal{T}_6$ auf dem Pfad *aabaaa* war *aa*. Man folgt zunächst dem Suffix-Link zum Knoten *a* (1) und sucht von dort mittels Skip/Count-Trick den Pfad *baaa* (2–3). Da der Pfad im Inneren einer Kante endet, wird hier eine neue Kante mit der Beschriftung # eingefügt (4) und das Suffix-Link des in Phase 7 eingefügten Knoten gesetzt (5).



□

Algorithmus 4.2 Konstruktion des Suffixbaumes (McCreight)

Eingabe: Wort S mit $|S| = n$
Ausgabe: Suffixbaum von S

(1) $\mathcal{T} \leftarrow (\{root\}, \emptyset)$; $v \leftarrow root$; $u \leftarrow root$;
(2) **for** $i \leftarrow 1$ **to** $n+1$
(3) **if** $u = v$ **then** $\gamma \leftarrow \varepsilon$;
(4) **else** $\gamma \leftarrow$ Beschriftung der Kante von v nach u;
(5) **if** $u = root$ **then** $t \leftarrow i$;
(6) **else if** $v = root$ **then** $\gamma \leftarrow \gamma$ ohne erstes Zeichen;
(7) **else** $v \leftarrow s[v]$;
(8) $p \leftarrow$ Skip-Count-Search$(\mathcal{T}, v, \gamma)$; $q \leftarrow p$;
(9) **while** (von q gibt es eine Fortsetzung mit $S[t]$)
(10) $q \leftarrow Next(q, S[t])$; $t \leftarrow t+1$;
(11) $v \leftarrow$ letzter Knoten auf dem Pfad nach q;
(12) Füge bei q eine neue Kante mit der Beschriftung $S[t \ldots n+1]$ zu einem neuen Blatt mit der Beschriftung i ein;
(13) **if** $u \neq root$ **then** $s[u] \leftarrow p$;
(14) $u \leftarrow q$;

Zum Beweis der Linearität der Laufzeit des Algorithmus von McCreight zeigen wir zunächst folgendes Lemma. Dabei ist v_i wie oben definiert, und $depth_i(v)$ ist die Tiefe eines Knoten v in $\mathcal{T}_i(S)$.

Lemma 4.5 *Ist v_i nicht die Wurzel, so gilt $depth_i(s[v_i]) \geq depth_i(v_i) - 1$.*

Beweis. Alle von der Wurzel verschiedenen Knoten auf dem Weg nach v_i waren schon in $\mathcal{T}_{i-1}(S)$ vorhanden und besitzen damit (paarweise verschiedene) Suffix-Links, die sich auf dem Weg nach $s[v_i]$ befinden. Da höchstens einer der Knoten die Wurzel als Suffix-Link besitzt, folgt das Lemma. □

Satz 4.6 *Der Algorithmus von McCreight ermittelt den Suffix-Baum von S mit einer Laufzeit von $O(n)$.*

Beweis. Die Korrektheit folgt aus der Korrektheit der Verwendung von Suffix-Links und des Skip/Count-Tricks. Für die Laufzeitabschätzung betrachten wir getrennt den Aufwand während der Skip/Count-Suche bzw. während der Suche durch zeichenweise Vergleiche.

Die Skip/Count-Suche beim Einfügen von $S_i\#$ (i-te Phase) beginnt in einer Tiefe von mindestens $depth_{i-1}(v_{i-1}) - 1$ und endet in einer Tiefe von höchstens $depth_{i-1}(v_i)$. Wegen $depth_{i-1}(v_i) = depth_i(v_i)$ ergeben sich für die i-te Phase somit höchstens $depth_i(v_i) - depth_{i-1}(v_{i-1}) + 1$ Skip/Count-Schritte. Insgesamt ist die Zahl der Skip/Count-Schritte beschränkt durch

$$\sum_{i=2}^{n+1}[depth_i(v_i) - depth_{i-1}(v_{i-1}) + 1] = depth_{n+1}(v_{n+1}) - depth_1(v_1) + n = n.$$

Die zeichenweisen Vergleiche beginnen in der Phase $(i+1)$ an der Position t_i und enden an der Position t_{i+1}. Die Zahl der Schritte bei den zeichenweisen Vergleichen beträgt damit in der $(i+1)$-ten Phase $t_{i+1} - t_i + 1$. Insgesamt ergibt sich für die Zahl der expliziten Vergleiche eine Abschätzung von

$$\sum_{i=0}^{n}(t_{i+1} - t_i + 1) = 2n + 2.$$

□

Algorithmus von Weiner

Der Algorithmus von Weiner fügt die Suffixe, beginnend mit dem letzten und kürzesten, in den Baum ein. Man hat nach Ablauf jeder Phase einen Suffixbaum für ein immer größer werdendes Suffix des Textes konstruiert. Man erreicht ebenfalls eine lineare Laufzeit, benötigt aber mehr Zeit und Platz als im Algorithmus von McCreight. Die Bedeutung des Algorithmus von Weiner ist damit heute vor allem historisch.

Online-Algorithmus von Ukkonen

Der Algorithmus von Ukkonen konstruiert sogenannte *implizite* Suffixbäume, die kein ausgezeichnetes Endsymbol # verwenden und in denen es nicht notwendig zu jedem Suffix ein Blatt gibt. Genauer gesagt, gibt es für jeden impliziten Suffixbaum einen Index k derart, dass für alle $1 \leq j \leq k$ das j-te Suffix kein Präfix eines anderen Suffixes ist und damit ein Blatt j vorhanden ist, während für alle $j > k$ das j-te Suffix das Präfix eines anderen Suffixes ist und damit kein Blatt j existiert. (Dies gilt analog zu Lemma 4.2.) Der Suffixbaum von S ist offenbar der implizite Suffixbaum von $S\#$.

Die Grundidee des Algorithmus von Ukkonen ist, mit wachsendem i die impliziten Suffixbäume von $S[1 \ldots i]$ zu konstruieren. Man hat also nach jeder Phase einen impliziten Suffixbaum vorzuliegen. Die dabei vorzunehmenden Schritte sind im wesentlichen die gleichen wie beim Algorithmus von McCreight; sie werden nur etwas anders interpretiert.

Der wesentliche Vorteil des Algorithmus von Ukkonen ist, dass er *online* abläuft, d.h. nach der Betrachtung des i-ten Textzeichens liegt der implizite Suffixbaum bis zum i-ten Zeichen vor.

4.4 Konstruktion von Suffix-Arrays

Das Suffix-Array kann sehr einfach aus dem Suffixbaum konstruiert werden.

Satz 4.7 *A_S kann aus $\mathcal{T}(S)$ mit einem Aufwand von $O(|S|)$ ermittelt werden.*

Beweis. In jedem inneren Knoten werden die ausgehenden Kanten entsprechend ihrer Beschriftung und damit nach dem ersten Buchstaben ihrer Beschriftung geordnet. Dann ergibt sich A_S aus der Beschriftung der Blätter von $\mathcal{T}(S)$ in DFS-Ordnung. Der Aufwand für die Sortierung ist $O(|S| \cdot \log |\Sigma|)$, der Aufwand für die DFS $O(|S|)$. □

Die Konstruktion des Suffix-Arrays aus dem Suffix-Baum kann sinnvoll sein, wenn genügend Platz zur Verfügung steht, um den Suffix-Baum einmal zu ermitteln, aber nicht genug, um ihn dauerhaft zu speichern. Im folgenden geben wir platzsparende Konstruktionen an. Die erste stammt von Manber und Myers aus dem Jahr 1992, die zweite von Kärkkäinen und Sanders aus dem Jahr 2003. Beide Konstruktionen benutzen die *natürliche Ordnung* von Tupeln und die Sortierung mittels *Radix Sort*.

Definition 4.6 *$(M, <)$ sei eine total geordnete Menge. Die* natürliche Ordnung *$<_k$ auf M^k ist definiert als*

$$(a_1, a_2, \ldots, a_k) <_k (b_1, b_2, \ldots, b_k) : \Longleftrightarrow \exists i(\forall j(j < i \rightarrow a_j = b_j) \wedge a_i < b_i)$$

Eine Menge $A \subseteq M^k$ kann mit dem folgenden Algorithmus RADIX SORT sortiert werden.

Algorithmus 4.3 Radix Sort

Eingabe: Menge $A \subseteq M^k$, Ordnung $(M, <)$
Ausgabe: A sortiert nach $<_k$
(1) **for** $i \leftarrow k$ **downto** 1
(2) $A \leftarrow A$ *stabil* sortiert nach i-ter Komponente;

Gilt $|M| = r$ für einen festen Wert r, so kann für die stabile Sortierung das Verfahren BUCKET SORT verwendet werden. (Bei der Sortierung nach BUCKET SORT verwendet man für jeden der r möglichen Werte eine anfangs leere Liste, an die hier die Elemente aus A entsprechend dem Wert ihrer i-ten Komponente angehängt werden. Danach werden die Listen entsprechend der Ordnung in M aneinandergereiht. Dieses Verfahren ist offensichtlich stabil.)

Damit ergibt sich für die Sortierung von A mit $|A| = n$ ein Gesamtaufwand von $O(k(n+r))$. Ist k eine Konstante und gilt $r \leq n$ (was bei den folgenden Algorithmen der Fall sein wird), so beträgt der Aufwand $O(n)$.

Weiterhin benötigen wir für die Konstruktion des Suffix-Arrays das *inverse Suffix-Array*, das für eine Zahl j die Stellung des Suffixes S_j im Suffix-Array angibt.

Definition 4.7 *Es sei $S \in \Sigma^*$ mit $|S| = n$. Das* inverse Suffix-Array *von S ist das $(n+1)$-dimensionale Feld $\overline{A_S}$ mit $\overline{A_S}[j] = i$ genau dann, wenn $A_S[i] = j$.*

Konstruktion durch Verfeinerung

Es sei $S \in \Sigma^*$ mit $|S| = n$. Die Idee ist, die Zahlen $i \in \{1, 2, \ldots, n+1\}$ zunächst nach den Infixen der Länge 1, d.h. den Symbolen $S[i]$, dann nach den Infixen der Länge 2, dann nach den Infixen der Länge 4 bzw. allgemeiner in der i-ten Phase nach den Infixen der Länge 2^i zu ordnen. Das Suffix-Array ist erreicht, wenn keine zwei Infixe der Länge 2^i gleich sind. Dies ist nach spätestens $\log_2 n$ Phasen der Fall. Formal definieren wir eine Äquivalenz- und eine Ordnungsrelation, die *verfeinert* werden.

Definition 4.8 *Es sei $S \in \Sigma^*$ mit $|S| = n$. Wir definieren für $k \geq 1$ die folgenden (von S abhängigen) binären Relationen auf $\{1, \ldots, n+1\}$.*

$$\begin{aligned} i \equiv_k j &:\iff S[i \ldots i+k-1] = S[j \ldots j+k-1] \\ i \prec_k j &:\iff S[i \ldots i+k-1] <_{lex} S[j \ldots j+k-1] \end{aligned}$$

Man beachte: $S[i \ldots m]$ ist für $m > n$ als $S[i \ldots n]$ definiert.

Offensichtlich sind die Relationen $\equiv_k$ Äquivalenzrelationen, während die Relationen $\prec_k$ transitiv und antireflexiv sind. Aus $i \equiv_k i'$, $j \equiv_k j'$ und $i \prec_k j$ folgt $i' \prec_k j'$. Damit ist $\prec_k$ eine Ordnung auf den Äquivalenzklassen von $\equiv_k$ mit $I \prec_k J$ genau dann, wenn $i \prec_k j$ für $i \in I$ und $j \in J$. Außerdem gilt: Aus $i \equiv_{k+1} j$ folgt $i \equiv_k j$, und aus $i \prec_k j$ folgt $i \prec_{k+1} j$, d.h. die $(k+1)$-ten Relationen verfeinern jeweils die k-ten Relationen.

Lemma 4.8 *Für $S \in \Sigma^*$ mit $|S| = n$ gilt:*

1. $i \equiv_{k_1+k_2} j \iff (i \equiv_{k_1} j) \wedge (i + k_1 \equiv_{k_2} j + k_1)$,
2. $i \equiv_n j \iff i = j$,
3. $i \prec_{k_1+k_2} j \iff i \prec_{k_1} j \vee ((i \equiv_{k_1} j) \wedge (i + k_1 \prec_{k_2} j + k_1))$,
4. $i \prec_n j \iff S[i \ldots n] <_{lex} S[j \ldots n]$.

Beweis. Wir zeigen nur (3) und (4). Die Behauptungen (1) und (2) kann man analog beweisen.

(3) Es gilt

$$\begin{aligned} i \prec_{k_1+k_2} j &\iff S[i \ldots i+k_1+k_2-1] <_{lex} S[j \ldots j+k_1+k_2-1] \\ &\iff S[i \ldots i+k_1-1] <_{lex} S[j \ldots j+k_1-1] \vee \\ &\quad \Big(S[i \ldots i+k_1-1] = S[j \ldots j+k_1-1] \wedge \\ &\quad\quad S[i+k_1 \ldots i+k_1+k_2-1] <_{lex} S[j+k_1 \ldots j+k_1+k_2-1]\Big) \\ &\iff (i \prec_{k_1} j) \vee ((i \equiv_{k_1} j) \wedge (i + k_1 \prec_{k_2} j + k_1)). \end{aligned}$$

(4) Gilt nach Definition. □

Im Algorithmus werden die Äquivalenzklassen $\equiv_{2^k}$ und die Ordnungen $\prec_{2^k}$ für alle $k \geq 0$ bestimmt, bis jede Äquivalenzklasse einelementig ist. Nach spätestens $\lceil \log_2 n \rceil$ Phasen ist die Ordnung $\prec$ ermittelt. Der Algorithmus für die Konstruktion durch Verfeinerung benutzt die Arrays

- A: enthält nach Phase k die Menge $\{1, 2, \ldots, n+1\}$ geordnet nach $\prec_{2^k}$; am Ende A_S,
- $\overline{A}$: gibt nach Phase k die Äquivalenzklassen in $\equiv_{2^k}$ an; am Ende $\overline{A_S}$,
- B: zur Zwischenspeicherung.

Konstruktion des Suffix-Arrays durch Verfeinerung: Phase 0

(1) $A \leftarrow$ RADIX SORT($\{1, 2, \ldots, n+1\}$) nach Schlüssel $S[j]$;
(2) $\overline{A}[A[1]] \leftarrow 1$;
(3) **for** $i \leftarrow 2$ **to** $n+1$
(4) **if** $S[A[i]] = S[A[i-1]]$ **then** $\overline{A}[A[i]] \leftarrow \overline{A}[A[i-1]]$;
(5) **else** $\overline{A}[A[i]] \leftarrow \overline{A}[A[i-1]] + 1$;

Konstruktion des Suffix-Arrays durch Verfeinerung: Phase $k \geq 1$

(1) $A \leftarrow$ RADIX SORT($\{1, 2, \ldots, n+1\}$) nach Schlüssel $(\overline{A}[j], \overline{A}[j + 2^{k-1}])$;
(2) $B[A[1]] \leftarrow 1$;
(3) **for** $i \leftarrow 2$ **to** $n+1$
(4) **if** $(\overline{A}[A[i]] = \overline{A}[A[i-1]]$ **and** $\overline{A}[A[i] + 2^{k-1}] = \overline{A}[A[i-1] + 2^{k-1}])$
(5) **then** $B[A[i]] \leftarrow B[A[i-1]]$;
(6) **else** $B[A[i]] \leftarrow B[A[i-1]] + 1$;
(7) $\overline{A} \leftarrow B$;

Satz 4.9 *Die oben genannten Prozeduren konstruieren das Suffixarray von S in höchstens $\lceil \log_2 n \rceil + 1$ Phasen mit einem Gesamtaufwand von $O(n \log n)$*

Beweis. Wir müssen zeigen, dass im Array A nach Phase k die Suffixe von S nach $\prec_{2^k}$ geordnet sind und dass $\overline{A}[j]$ den Platz der Äquivalenzklasse von j bezüglich der Ordnung $\prec_{2^k}$ wiedergibt. Dieser Beweis erfolgt per Induktion über k.

In Phase 0 werden die Suffixe bezüglich ihres ersten Buchstaben geordnet, damit enthält A die Reihenfolge der Suffixe bezüglich $\prec_{2^0}$. Der Wert von $\overline{A}[j]$ ist gleich dem Wert für den Vorgänger j' von j in A, wenn $S[j] = S[j']$ und damit $j \equiv_{2^0} j'$ gilt, und anderenfalls gleich $\overline{A}[j'] + 1$. Damit gibt $\overline{A}[j]$ tatsächlich den Platz der Äquivalenzklasse von j bezüglich der Ordnung $\prec_{2^0}$ wieder.

Sei gezeigt, dass nach Phase k das Array A bezüglich $\prec_{2^k}$ geordnet ist und $\overline{A}[j]$ den Platz der Äquivalenzklasse von j bezüglich der Ordnung $\prec_{2^k}$ wiedergibt. Dann steht nach der Sortierung j in A vor j' genau dann, wenn einer der folgenden drei Fälle eintritt:

1. $\overline{A}[j] < \overline{A}[j']$,
2. $\overline{A}[j] = \overline{A}[j']$ und $\overline{A}[j + 2^k] < \overline{A}[j' + 2^k]$,
3. $\overline{A}[j] = \overline{A}[j']$ und $\overline{A}[j + 2^k] = \overline{A}[j' + 2^k]$ und j stand vorher in A vor j'.

Nach Induktionsvoraussetzung und gemäß Lemma 4.8 ist A damit nach der Sortierung bezüglich $\prec_{2^{k+1}}$ geordnet. Analog zu Phase 0 kann man zeigen, dass das Array B und nach der Phase auch $\overline{A}$ die Positionen der Indizes bezüglich der Ordnung $\prec_{2^{k+1}}$ enthält. □

Beispiel 4.10 Für $S =$ `mississippi` ergibt sich folgender Verlauf des Algorithmus.
Die Tabellen geben die Arrays A und $\overline{A}$ sowie in der Zeile für A die Grenzen zwischen den Äquivalenzklassen von $\equiv_{2^k}$ an.

Phase 0:

	1	2	3	4	5	6	7	8	9	10	11	12
A	12	2	5	8	11	1	9	10	3	4	6	7

	1	2	3	4	5	6	7	8	9	10	11	12
$\overline{A}$	3	2	5	5	2	5	5	2	4	4	2	1

Phase 1:

	1	2	3	4	5	6	7	8	9	10	11	12
A	12	11	8	2	5	1	10	9	4	7	3	6

	1	2	3	4	5	6	7	8	9	10	11	12
$\overline{A}$	5	4	9	8	4	9	8	3	7	6	2	1

Phase 2:

	1	2	3	4	5	6	7	8	9	10	11	12
A	12	11	8	2	5	1	10	9	7	4	6	3

	1	2	3	4	5	6	7	8	9	10	11	12
$\overline{A}$	5	4	11	9	4	10	8	3	7	6	2	1

Phase 3:

	1	2	3	4	5	6	7	8	9	10	11	12
A	12	11	8	5	2	1	10	9	7	4	6	3

	1	2	3	4	5	6	7	8	9	10	11	12
$\overline{A}$	6	5	12	10	4	11	9	3	8	7	2	1

□

Skew Algorithmus von Kärkkäinen und Sanders

Dieser neue Algorithmus (2003) konstruiert das Suffix-Array in linearer Zeit mit effizientem Platzaufwand. Er besteht im wesentlichen aus 3 Schritten.

1. Konstruiere das partielle Suffix-Array (sowie das inverse partielle Suffix-Array) für die Index-Menge $J = \{1 \leq j \leq n : j \, mod \, 3 \neq 1\}$. Dabei sortiert man die Teilwörter der Länge 3, die an den Positionen j mit $j \, mod \, 3 \neq 1$ beginnen, erhält ein neues Wort S' der Länge $2n/3$ über dem Alphabet $\{1, \ldots, \lceil 2n/3 \rceil\}$ und ruft rekursiv den Skew-Algorithmus für das Wort S' auf. Der Zeitaufwand ist $O(n)$ sowie der Aufwand für die rekursive Lösung des Problems der Größe $\lceil 2n/3 \rceil$.

2. Konstruiere (mit Hilfe des partiellen Suffix-Arrays für die Index-Menge J) das partielle Suffix-Array für die Index-Menge $\overline{J} = \{1 \leq j \leq n : j \, mod \, 3 = 1\}$. Der Zeitaufwand ist $O(n)$.

3. Konstruiere aus den partiellen Suffix-Arrays das gesamte Suffix-Array. Dieser Schritt ist ähnlich dem Mischen bei MERGE SORT und benötigt eine Zeit von $O(n)$.

Seinen Namen (*skew=asymmetrisch, schräg*) erhielt der Algorithmus, weil die "Teile und herrsche"-Strategie das Gesamtproblem in zwei Teilprobleme unterschiedlicher Größe zerlegt. Die Zeit $T(n)$ für die Lösung des Problems der Größe n erfüllt die Gleichung $T(n) = O(n) + T(2n/3)$. Damit ergibt sich eine gesamte Laufzeit von $O(n)$. Wir betrachten nun die einzelnen Schritte im Detail.

Skew Algorithmus – Schritt 1

Der Einfachheit halber nehmen wir an, dass $|S| = n = 3n'$ gilt. Außerdem sei das Alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$, und wir setzen $S[n+1] = S[n+2] = 0$.

Wir sortieren zunächst die Indizes $j \in J$ nach dem Schlüssel $S[j \ldots j+2]$ mittels RADIX SORT. (In der Terminologie des vorigen Abschnitts konstruieren wir die Äquivalenzklassen von $\equiv_3$ geordnet nach $\prec_3$.) Entsprechend der lexikographischen Ordnung von $S[j \ldots j+2]$ erhält jedes j mit $j \bmod 3 \neq 1$ einen *Namen* $N[j] \in \{1, \ldots, 2n'\}$ (die Nummer seiner Äquivalenzklasse). Es gilt also $N[j_1] < N[j_2]$ genau dann, wenn $S[j_1 \ldots j_1 + 2] <_{lex} S[j_2 \ldots j_2 + 2]$. Wir erhalten das Wort S' der Länge $2n'$ als

$$S' = N[2]N[5] \ldots N[3n'-1]N[3]N[6] \ldots N[3n'].$$

Einer Position j in S mit $j \bmod 3 \neq 1$ entspricht eineindeutig eine Position j' in S': Für $j = 3i - 1$ ist $j' = i$; für $j = 3i$ ist $j' = n' + i$.

Lemma 4.10 *Die Stellung von j im partiellen Suffix-Array von S für die Index-Menge J ist gleich der Stellung von j' im Suffix-Array von S'.*

Beweis. Es seien j_1 und j_2 Zahlen mit $j_1, j_2 \not\equiv 1 \bmod 3$. Wir müssen beweisen, dass $S[j_1 \cdots n] <_{lex} S[j_2 \cdots n]$ genau dann gilt, wenn $S'[j_1' \cdots 2n'] <_{lex} S'[j_2' \cdots 2n']$ gilt. Dies kann durch Fallunterscheidung gezeigt werden. Von Bedeutung ist, dass die Zeichen $S'[n'] = N[3n'-1]$ sowie $S'[2n'] = N[3n']$ – entsprechend den Teilwörtern $(S[n-1]S[n]0)$ sowie $(S[n]00)$ – jeweils genau einmal in S' vorkommen. □

Das partielle Suffix-Array von S für die Index-Menge J kann also ermittelt werden, indem man das Suffix-Array von S' rekursiv durch Anwendung des Skew Algorithmus bestimmt. Man beachte dabei, dass S' ein Wort über $\{1, \ldots, 2n'\}$ ist, d.h. RADIX SORT kann beim Ordnen der Tripel mit einem Aufwand von $O(n)$ ausgeführt werden.

Skew Algorithmus – Schritt 2

Nun ist die Ordnung der Suffixe S_j mit $j \bmod 3 = 1$ zu ermitteln. Offensichtlich ergibt sich diese aus der natürlichen Ordnung bezüglich $(S[j], S_{j+1})$.

Die Ordnung der Suffixe S_{j+1} mit $j \bmod 3 = 1$ ist aus Schritt 1 bekannt und durch das partielle Suffix-Array für die Menge J gegeben. Die Sortierung der S_j mit $j \bmod 3 = 1$ nach dem Schlüssel S_{j+1} ist damit trivial. Anschließend genügt eine Sortierung mit BUCKET SORT nach dem Schlüssel $S[j]$, um das partielle Suffix-Array von S bezüglich der Menge $\overline{J}$ zu erhalten.

Skew Algorithmus – Schritt 3

Nun sind noch die partiellen Suffix-Arrays zu mischen. Ähnlich wie beim Mischen im Sortieralgorithmus MERGE SORT beginnen wir in den partiellen Suffix-Arrays mit dem ersten Element und vergleichen die Suffixe bezüglich der lexikographischen Ordnung. Das kleinere Suffix wird in das totale Suffix-Array eingefügt; im zugehörigen partiellen Array wird zur nächsten Position gegangen.

Während dieses Mischens erfolgen Vergleiche von Suffixen der Form S_{3i+1} mit Suffixen der Form S_{3j+2} oder S_{3j}. Wir zeigen, dass jeder dieser Vergleiche mit konstantem Aufwand ausgeführt werden kann.

Zum Vergleich eines Suffixes S_{3i+1} mit einem Suffix S_{3j+2} vergleicht man die Paare $(S[3i+1], S_{3i+2})$ und $(S[3j+2], S_{3j+3})$. Dabei ergibt sich die Reihenfolge von S_{3i+2} und S_{3j+3} aus dem inversen partiellen Suffix-Array für die Index-Menge J.
Zum Vergleich eines Suffixes S_{3i+1} mit einem Suffix S_{3j} vergleicht man die Tripel $(S[3i+1], S[3i+2], S_{3i+3})$ und $(S[3j], S[3j+1], S_{3j+2})$. Dabei ergibt sich die Reihenfolge von S_{3i+3} und S_{3j+2} aus dem inversen partiellen Suffix-Array für die Index-Menge J.
In jedem Fall beträgt die Zeit für einen Vergleich $O(1)$, insgesamt $O(n)$.

Beispiel 4.11 Es sei $S =$ `mississsippi`.
Schritt 1. Wir erhalten die folgenden Tripel mit ihren Namen.

2	5	8	11	3	6	9	12	Position in S
`iss`	`iss`	`sip`	`pi0`	`ssi`	`sss`	`ipp`	`i00`	Teilwörter der Länge 3
3	3	5	4	6	7	2	1	Namen

Das neue Wort ist $S' =$ `33546721`.
Rekursiv erhalten wir als Suffix-Array von S': $(8, 7, 1, 2, 4, 3, 5, 6)$
und als partielles Suffix-Array von S für die Index-Menge J: $(12, 9, 2, 5, 11, 8, 3, 6)$.

Schritt 2. Als partielles Suffix-Array von S für die Index-Menge $\overline{J}$ ergibt sich: $(1, 10, 4, 7)$.

Schritt 3. Wir erhalten folgende partielle Suffix-Arrays und zu vergleichende Paare bzw. Tripel (die Paare bzw. Tripel werden unmittelbar beim Vergleich ermittelt und nicht in einem Array gespeichert):

1	10	4	7
`mi7`	`pp1`	`si8`	`ss2`
`m3`	`p5`	`s4`	`s6`

12	9	2	5	11	8	3	6
`i00`	`ip5`					`ss4`	`ss6`
		`i7`	`i8`	`p1`	`s2`		

Nach dem Mischen ergibt sich das Suffix-Array von S: $(12, 9, 2, 5, 1, 11, 8, 10, 4, 7, 3, 6)$. □

Eine Implementierung des Skew Algorithmus ist in der Originalarbeit von Kärkkäinen und Sanders [9] zu finden.

4.5 Anwendungen von Suffixbäumen und Suffix-Arrays

Die wichtigste Anwendung und Motivation von Text-Indizes ist die schnelle (exakte oder inexakte) Suche in einem festen Text. Außerdem kann man Regelmäßigkeiten, wie lange Wiederholungen, häufige Teilwörter oder Palindrome mit Hilfe von Suffixbäumen entdecken. Schließlich findet man Anwendungen bei effizienten Kompressionsverfahren (Lempel-Ziv, Burrows-Wheeler). Wir wollen im folgenden einige der genannten Anwendungen demonstrieren.

Suchprobleme

Die exakte Suche wurde bereits in Abschnitt 4.2 betrachtet. Der Vollständigkeit halber seien hier noch einmal die Laufzeiten genannt.

Satz 4.11 *Es seien Σ ein Alphabet mit $|\Sigma| = \sigma$, $T \in \Sigma^*$ ein Text mit $|T| = n$ und $P \in \Sigma^*$ ein Suchwort mit $|P| = m$. Die Zeit für die Suche nach allen Vorkommen von P in T beträgt*

- *$O(m \cdot \log \sigma + A)$ unter Verwendung des Suffixbaumes,*
- *$O(m \cdot \log n + A)$ unter Verwendung des Suffix-Arrays,*

wobei A die Anzahl der Vorkommen von P in T ist.

Für die inexakte Suche mittels Suffixbäumen gibt es zwei Strategien, die auf den Methoden aus Kapitel 3 beruhen. Man kann den Alignment-Algorithmus am Baum ausführen oder aber nach exakten Treffern für Teilwörter suchen und an den betreffenden Textstellen das Alignment vornehmen (Filterung). Bei beiden Strategien hat man das Problem, dass große Teile des Suffixbaumes durchsucht werden müssen. Die inexakte Index-basierte Suche ist damit auch heute noch ein wichtiges Forschungsgebiet.

Wiederholungen

Für die Analyse und auch für die Kompression von Texten sind Wiederholungen von großer Bedeutung. Mit Hilfe von Suffixbäumen kann man Wiederholungen leicht entdecken.

Definition 4.9 *Es sei S ein Wort mit $|S| = n$. Eine* Wiederholung *in S ist ein Tripel (ℓ, i, j) mit $1 \leq \ell \leq n$, $1 \leq i < j \leq n-\ell+1$ und $S[i \ldots i+\ell-1] = S[j \ldots j+\ell-1]$. Eine Wiederholung (ℓ, i, j) heißt* rechts-maximal*, wenn $(\ell+1, i, j)$ keine Wiederholung ist und* maximal*, wenn weder $(\ell+1, i, j)$ noch $(\ell+1, i-1, j-1)$ Wiederholungen sind.*

Häufig interessiert uns auch das Wort, das wiederholt vorkommt. Ist also (ℓ, i, j) eine Wiederholung bzw. eine rechts-maximale Wiederholung bzw. eine maximale Wierdholungin S, so sagen wir, dass das Wort $\alpha = S[i \ldots i+\ell-1] = S[j \ldots j+\ell-1]$ in S als Wiederholung bzw. als rechts-maximale Wiederholung bzw. als maximale Wiederholung in S vorkommt.

Lemma 4.12 *Ein Wort α kommt genau dann in $S \in \Sigma^*$*

1. *als Wiederholung vor, wenn der Pfad α im Suffixbaum von S existiert und nicht in einem Blatt bzw. in der Kante zu einem Blatt endet.*
2. *als rechts-maximale Wiederholung vor, wenn der Pfad α im Suffixbaum von S existiert und in einem inneren Knoten endet.*
3. *als rechts-maximale Wiederholung vor, wenn der Pfad α im Suffixbaum von S existiert und in einem inneren Knoten endet, in dessen Unterbaum zwei Blätter i, j mit $S[i-1] \neq S[j-1]$ existieren, wobei $S[0] := \#$ mit $\# \notin \Sigma$ gesetzt sei.*

Bemerkung: Einen Knoten mit der Eigenschaft 3 nennt man *links-divers*.
Beweis. Ein Wort α der Länge ℓ kommt genau dann als Wiederholung in S vor, wenn es Zahlen $i < j$ gibt, so dass $\alpha = S[i \ldots i+\ell-1] = S[j \ldots j+\ell-1]$ gilt. Wir können uns

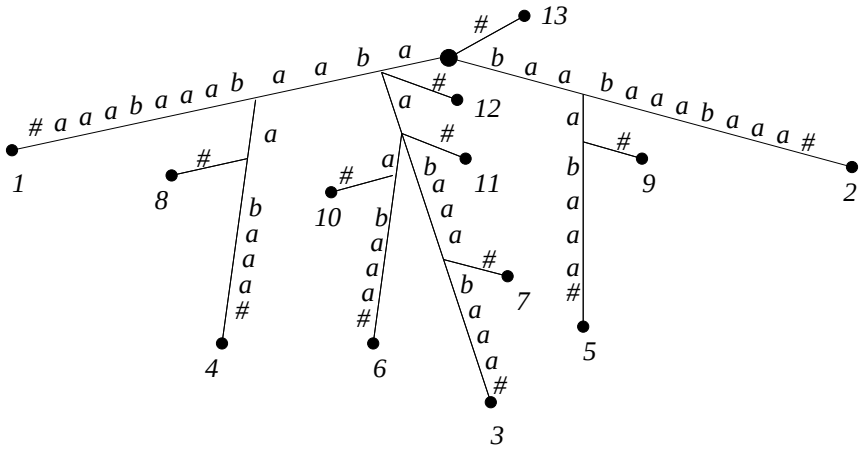
also darauf beschränken zu untersuchen, wann ein Tripel (ℓ, i, j) eine Wiederholung bzw. eine rechts-maximale Wiederholung bzw. eine maximale Wiederholung ist.

zu 1: (ℓ, i, j) ist genau dann eine Wiederholung in S, wenn im Suffixbaum von S die Beschriftungen der Pfade zu den Blättern i und j auf den ersten ℓ Zeichen übereinstimmen. Da sich die Pfade später trennen, darf der gemeinsame Pfad $\alpha = S[i \dots i + \ell - 1] = S[j \dots j + \ell - 1]$ nicht in einem Blatt oder in der Kante zu einem Blatt enden.

zu 2: (ℓ, i, j) ist genau dann eine rechts-maximale Wiederholung, wenn (ℓ, i, j) eine Wiederholung ist und $S[i + \ell] \neq S[j + \ell]$ gilt. Das bedeutet, dass der in Punkt 1 erwähnte gemeinsame Pfad zu den Blättern i und j genau die Länge ℓ besitzt, also nach dem gemeinsamen Präfix α endet. Damit muss der Pfad α in einem inneren Knoten enden.

zu 3: (ℓ, i, j) ist genau dann eine maximale Wiederholung, wenn (ℓ, i, j) eine rechts-maximale Wiederholung ist und $S[i - 1] \neq S[j - 1]$ gilt. Damit ist es notwendig, dass der gemeinsame Pfad $\alpha = S[i \dots i + \ell - 1] = S[j \dots j + \ell - 1]$ in einem links-diversen Knoten endet. Endet andererseits ein Pfad α in einem links-diversen Knoten v, so existieren im Unterbaum von v zwei Blätter i und j, deren letzter gemeinsamer Vorgänger v ist und die $S[i - 1] \neq S[j - 1]$ erfüllen. □

Beispiel 4.12 Wir betrachten das Wort $S = abaabaaabaaa$ und seinen Suffixbaum.



Wir können z.B. folgende Fakten feststellen:

- *aaab* tritt nicht als Wiederholung in S auf, da der zugehörige Pfad in einer Kante zu einem Blatt endet.
- *aba* tritt als Wiederholung in S auf, aber nicht rechts-maximal, da der zugehörige Pfad in einer Kante zu einem inneren Knoten endet. Die Wiederholungen sind $(3, 1, 4)$, $(3, 1, 8)$ und $(3, 4, 8)$.
- *baa* tritt als rechts-maximale Wiederholung in S auf, da der zugehörige Pfad in einem inneren Knoten endet. Die zugehörigen rechts-maximalen Wiederholungen sind $(3, 2, 5)$ und $(3, 2, 9)$. Die Wiederholung $(3, 5, 9)$ ist nicht rechts-maximal, da die Blätter 5 und 9 noch einen späteren gemeinsamen Vorgänger als den Knoten *baa* besitzen. Der Knoten *baa* ist weiterhin nicht links-divers, da $S[1] = S[4] = S[8] = a$ gilt. Eine maximale Wiederholung von *baa* existiert also nicht.

- *aa* tritt als maximale Wiederholung in S auf, da der Pfad *aa* in einem links-diversen Knoten endet. Der Unterbaum enthält nämlich die Blätter 6 und 11, und es gilt $S[5] \neq S[10]$. Wir haben allgemeiner $S[6] = S[10] = a$ und $S[2] = S[5] = S[9] = b$ und damit als maximale Vorkommen von *aa*: $(2, 3, 11)$, $(2, 6, 7)$, $(2, 6, 11)$, $(2, 10, 7)$, $(2, 10, 11)$.

□

Es bleibt noch zu klären, wie man die links-diversen Knoten des Suffixbaumes sowie die Menge aller maximalen Wiederholungen effizient bestimmt. Dazu führt man für jeden Knoten v eine Menge $\mathit{Left}(v) \subseteq \Sigma \cup \{\#\}$, die für jedes Blatt i im Unterbaum von v das Symbol $S[i-1]$ enthält, sowie für jedes $x \in \mathit{Left}(v)$ eine Liste $\mathit{Leaves}_x(v) \subseteq \{1, 2, \ldots, n+1\}$ ein, die alle Blätter i aus dem Unterbaum von v mit $S[i-1] = x$ enthält.

Der Knoten v ist genau dann links-divers, wenn $|\mathit{Left}(v)| > 1$ gilt. Hat die Beschriftung des Knoten v die Länge ℓ, so ist (ℓ, i, j) genau dann eine maximale Wiederholung, wenn es Kinder $w_1 \neq w_2$ von v und Symbole $x \neq y$ derart gibt, dass $i \in \mathit{Leaves}_x(w_1)$ und $j \in \mathit{Leaves}_x(w_2)$ gilt. Offensichtlich gelten folgende Rekursionsbeziehungen:

- $\mathit{Left}(v) = \{S[i-1]\}$, falls v das Blatt i ist.
  $\mathit{Leaves}_x(v) = \{i\}$, falls v das Blatt i und $x = S[i-1]$ ist.
  $\mathit{Leaves}_x(v) = \emptyset$, falls v das Blatt i und $x \neq S[i-1]$ ist.

- $\mathit{Left}(v) = \bigcup_{w \text{ Kind von } v} \mathit{Left}(w)$ für jeden inneren Knoten v.
  $\mathit{Leaves}_x(v) = \bigcup_{w \text{ Kind von } v} \mathit{Leaves}_x(w)$ für jeden inneren Knoten v.

Damit kann ein Linearzeitalgorithmus zur Bestimmung aller maximalen Wiederholungen abgeleitet werden.

Satz 4.13 *Die Menge aller maximalen Wiederholungen in einem Wort S der Länge n kann mit einem Aufwand von $O(n + A)$ bestimmt werden, wobei A die Anzahl der maximalen Wiederholungen in S ist.*

Beweis. Wir traversieren den Suffixbaum von S in einer *Bottom-Up*-Strategie, d.h. wir beginnen mit den Blättern und setzen jeweils mit Knoten fort, deren Kinder bereits abgearbeitet wurden. Mit Hilfe der oben angegebenen Rekursionen bestimmen wir für jeden Knoten v die Menge $\mathit{Left}(v)$ und die Listen $\mathit{Leaves}_x(v)$ aus den Daten der Kinder von v und schließlich die zu v gehörenden maximalen Wiederholungen. Zu beachten ist dabei, dass die Liste $\mathit{Leaves}_x(v)$ durch *Verkettung* der entsprechenden Listen der Kinder entsteht (nicht etwa durch Kopieren). □

Gemeinsame Teilwörter

Ein verwandtes Problem zur Suche nach Wiederholungen ist die Bestimmung gemeinsamer Teilwörter in zwei oder mehreren Texten. Für die Suche in mehreren Texten benutzt man gemeinsame Suffixbäume. In Analogie zum Suffixbaum kann man für ein k-Tupel von Wörtern $(S^1, \ldots, S^k)$ den *gemeinsamen Suffixbaum* $\mathcal{T}(S^1, \ldots, S^k)$ definieren. Die Beschriftungen der Wege zu den Blättern sind dabei die Suffixe von $S^1\#, \ldots, S^k\#$. Das Blatt für das Suffix $S^h[i \ldots |S^h|]$ erhält die Beschriftung (h, i). Tritt ein Suffix in mehreren Wörtern auf, so erhält das zugehörige Blatt also mehrere Beschriftungen. Der gemeinsame Suffixbaum kann mit einem verallgemeinerten McCreight-Algorithmus in linearer Zeit konstruiert werden.

Lemma 4.14 *Ein Wort α kommt genau dann in jedem der Wörter $S^1, S^2, \ldots, S^k$ vor, wenn der Pfad α im gemeinsamen Suffixbaum $\mathcal{T}(S^1, \ldots, S^k)$ existiert und für alle $1 \leq h \leq k$ zu einem Blatt mit einer Beschriftung (h, i) fortgesetzt werden kann.*

Bemerkung. Man beachte, dass die Blätter nicht notwendig paarweise verschieden sein müssen.

Gemeinsame Teilwörter lassen sich nun analog zu Wiederholungen ermitteln, indem man in einer *Bottom-Up*-Strategie für jeden Knoten v bestimmt, zu welchen der k Wörter der Knoten v ein Blatt im Unterbaum besitzt. Dazu führt man für jeden Knoten einen Bitvektor der Länge k ein, wobei das h-te Bit genau dann 1 sein soll, wenn ein Blatt zum Wort h im Unterbaum vorhanden ist. Die Beschriftung des Pfades zum Knoten ist genau dann Teilwort in allen Wörtern, wenn der Bitvektor 1^k ist. Die Bestimmung der Bitvektoren ist mit einem Aufwand von $O(n \cdot \lceil k/w \rceil)$ möglich, wobei n die Summe der Längen von $S^1, S^2, \ldots, S^k$ und w die Länge eines Computerwortes ist. (Zur Bestimmung des Bitvektors eines inneren Knotens muss man nämlich nur das ODER der Bitvektoren seiner Kinder bilden.) Insbesondere ergibt sich das *längste gemeinsame Teilwort* durch den Knoten mit Bitvektor 1^k und maxmialer String-Tiefe. Wir können damit schließen:

Satz 4.15 *Das längste gemeinsame Teilwort von $S^1, S^2, \ldots, S^k$ kann mit einem Aufwand von $O(n \cdot \lceil k/w \rceil)$ bestimmt werden.*

Lempel-Ziv-Kompression

In Abschnitt 2.4 wurde bereits der LZW-Algorithmus besprochen, in dem die Kompression mit Hilfe von Wörterbüchern realisiert wurde und unter Verwendung von Tries implementiert wurde. Hier soll nun der erste Algorithmus von Lempel und Ziv aus dem Jahr 1997 (allgemein als LZ77 bekannt) vorgestellt und mittels Suffixbäumen implementiert werden. Die Grundidee ist wie folgt:

Der Text $T[1 \ldots i]$ sei bereits komprimiert. Das längste Präfix von $T[i+1 \ldots n]$, das in $T[1 \ldots i]$ beginnt, habe die Länge ℓ_i und komme zum ersten Mal an der Stelle p_i vor. Man beachte, dass das frühere Vorkommen des Präfixes nicht bis zur Stelle i enden muss.
Gilt $\ell_i > 0$, so kann man den Text $T[i+1 \ldots i+\ell_i]$ durch (p_i, ℓ_i) kodieren. Gilt $\ell_i = 0$, so speichert man $(0, T[i+1])$.

Beispiel 4.13 Es ergeben sich z.B. folgende LZ77-Komprimierten:

- `abaababaabaab` $\rightarrow$ `(0,a)(0,b)(1,1)(1,3)(2,5)(1,2)`.
- `abababababaa` $\rightarrow$ `(0,a)(0,b)(1,11)`.

Das zweite Beispiel zeigt, dass für Texte mit hoher Periodizität die LZ77-Kompression besser ist als die LZW-Kompression. □

Umgekehrt lässt sich die Dekompression sehr einfach mit linearem Aufwand ausführen:

Sei der Text $T[1 \ldots i]$ bereits dekomprimiert. Ist das nächste Zeichen der Komprimierten (p, ℓ) mit $p > 0$, so hängt man den Text $T[p \ldots p+\ell-1]$ an. Ist das nächste Zeichen der Komprimierten $(0, x)$ mit $x \in \Sigma$, so hängt man den Text x an.

Mit Hilfe von Suffixbäumen kann man den Ziv-Lempel-Algorithmus als Linearzeitalgorithmus implementieren.

Satz 4.16 *Der Ziv-Lempel-Algorithmus (LZ77) kann so implementiert werden, dass die Laufzeit $O(n)$ für einen Text der Länge n beträgt.*

Beweis. Seien bei der Konstruktion des Suffixbaumes bereits die Suffixe $S_1, S_2, \ldots S_i$ eingefügt. Dann ergibt sich (p_i, ℓ_i) beim Einfügen von $S[i+1 \ldots n]$ durch die Position, an der das neue Blatt eingefügt wird. Der Aufwand dafür beträgt $O(\ell_i)$. Die Anzahl der Schritte für die Konstruktion des Suffixbaumes beträgt $O(n)$. Den Aufwand für die Bestimmung der Komprimierten erhält man durch Aufsummierung der Längen ℓ_i über alle Positionen i, für die (p_i, ℓ_i) zu bestimmen ist. Seien $i_0 = 0, i_1, \ldots, i_k$ die Positionen, an denen (p_i, ℓ_i) zu bestimmen sind. Es gilt $i_{j+1} = i_j + \max\{\ell_{i_j}, 1\} \geq i_j + \ell_{i_j}$ und damit

$$\sum_{j=0}^{k} \ell_{i_j} \leq \sum_{j=0}^{k-1} (i_{j+1} - i_j) + \ell_{j_k} = i_{j_k} + \ell_{j_k} = n.$$

Damit beträgt der Gesamtaufwand $O(n)$. □

Burrows-Wheeler-Transformation

Ein noch effizienteres Kompressionsverfahren als die Algorithmen der Lempel-Ziv-Familie wurde 1994 von Burrows und Wheeler vorgeschlagen. Dabei wird im ersten und wichtigsten Schritt ein Wort S in ein Wort S' transformiert, dessen Zeichen eine Permutation von S darstellen. Diese Transformation ist umkehrbar eindeutig, was natürlich für Kompressionsalgorithmen unabdingbar ist. Die entscheidende Eigenschaft der Transformierten S' ist, dass sie lange Blöcke mit nur einem oder zwei Symbolen enthält. Dies macht sie allgemein für die Kompression sehr gut geeignet. Für die Burrows-Wheeler-Transformation benötigt man die Suffixe von S in ihrer lexikographischen Reihenfolge, was die Verwendung von Suffix-Arrays nahelegt.

Definition 4.10 *Es sei $S \in \Sigma^*$ ein Wort mit $|S| = n$ und $\# \notin \Sigma$ ein Sonderzeichen. Die* Burrows-Wheeler-Transformierte *von S ist das Wort S' der Länge $(n+1)$ mit $S'[i] = S[j-1]$, wobei $S[j \ldots n]$, $1 \leq j \leq n+1$, das lexikographisch i-te Suffix von S ist und $S[0] := \#$ sei.*

Mit Hilfe des Suffix-Arrays A_S lässt sich die Burrows-Wheeler-Transformierte definieren als $S'[i] = S[A_S[i] - 1]$.

Beispiel 4.14 Für $S = abcabca$ erhalten wir:

$$A_S = (8, 7, 4, 1, 5, 2, 6, 3),$$
$$S' = \texttt{a c c \# a a b b}$$

□

Eine alternative Interpretation ist die folgende: Schreibe die zyklischen Permutationen von $S\#$ in lexikografischer Reihenfolge untereinander. Die letzte Spalte ist die Burrows-Wheeler-Transformierte. In unserem Beispiel $S = \texttt{abcabca}$ erhalten wir:

```
#abcabc  a
a#abcab  c
abca#ab  c
abcabca  #
bca#abc  a
bcabca#  a
ca#abca  b
cabca#a  b
```

Satz 4.17 *Die Burrows-Wheeler-Transformierte von S mit $|S| = n$ kann in $O(n)$ Schritten ermittelt werden.*

Beweis. Das Suffix-Array von S kann mit einem Aufwand von $O(n)$ erzeugt werden. Die Burrows-Wheeler-Transformierte kann aus dem Suffix-Array sehr einfach mit linearem Aufwand ermittelt werden. □

Satz 4.18 *Ein Wort S der Länge n kann aus seiner Burrows-Wheeler-Transformierten in $O(n)$ Schritten ermittelt werden.*

Beweis. Von der Matrix der zyklischen Permutationen sind die erste Spalte $\overline{S}$ (Symbole der Transformierten S' in lexikografischer Ordnung) und die letzte Spalte (die Burrows-Wheeler-Transformierte S') bekannt. Man bestimmt $S[i]$ induktiv wie folgt (die Korrektheit wird ebenfalls durch Induktion über i bewiesen).

$i = 1$*:* Die Position von # in S' sei j. Dann ist $S[1] = \overline{S}[j]$.
$i \leftarrow i + 1$*:* Im Schritt i werde $S[i]$ an der Stelle von $\overline{S}$ gefunden, an der zum k-ten Mal der Buchstabe $S[i]$ in $\overline{S}$ vorkommt.
Dann findet man $S[i+1]$ an der Stelle von $\overline{S}$, an der zum k-ten Mal der Buchstabe $S[i]$ in S' vorkommt.

Um die Rücktransformation in Linearzeit durchzuführen definieren wir

- ein Array P über der Indexmenge $\Sigma \cup \{\#\}$ mit $P[a] = \sum_{b<a} |S'|_b$,
- für $a \in \Sigma \cup \{\#\}$ das Array V_a der Größe $|S'|_a$ mit
  $V_a[k] = j :\Longleftrightarrow\ S'[j] = a \wedge |S'[1 \ldots j]|_a = k$.

$P[a]$ gibt an, wie viele Zeichen in $\overline{S}$ vor a stehen. $V_a[k]$ gibt die Position des k-ten Vorkommens von a in S' an. Die Arrays P und V_a können in linearer Zeit bestimmt werden. Ist in Schritt i die Position j von S' erreicht und gilt $S[i] = \overline{S}[j] = a$, so wird in Schritt $i+1$ die Position $V_a[j - P[a]]$ von S' erreicht. Der Aufwand pro Schritt ist damit konstant. □

Beispiel 4.15 Wir führen für *acc#aabb* die Rücktransformation aus. Das # in S' steht unter dem dritten a (1), das dritte a in S' steht unter dem zweiten b (2), das zweite b in S' steht unter dem zweiten c (3), usw.

$\overline{S}$:	#	a	a	a	b	b	c	c
S':	a	c	c	#	a	a	b	b
	8	7	4	1	5	2	6	3

Die Rücktransformation ergibt damit *abcabca#*. □

Der Wert der Burrows-Wheeler-Transformation besteht darin, dass in der Transformierten oft das gleiche Symbol hintereinander auftritt. Genauer: Tritt $x\beta$ k_1-mal und β k_2-mal in S auf, so enthält die Transformierte einen Block der Länge k_2, in dem k_1-mal das Symbol x vorkommt.

Beispiel 4.16 Enthält ein Text das Wort *Algorithmus* 20 Mal und das Wort *Logarithmus* 5 Mal und keine weiteren Vorkommen von *ithmus*, so enthält die Transformierte z.B.

- einen Block der Länge 25 mit r,
- einen Block der Länge 25 mit 20 o und 5 a,
- jeweils einen Block der Länge 20 mit g bzw. l bzw. A,
- jeweils einen Block der Länge 20 mit g bzw. o bzw. L.

□

Das Auftreten von langen Blöcken desselben Symbols kann in anderen Kompressionsverfahren ausgenutzt werden. Ein bekannte Implementierung ist das Programm `bzip2`. Dort wird die Burrows-Wheeler-Transformierte zunächst einer *Move-to-front*-Transformation unterzogen. Der Effekt ist, dass sich die Häufigkeit stark zugunsten der lexikographisch kleinsten Symbole verlagert; bei ASCII-Text dominieren nach der Move-to-Front-Transformation die Zeichen `\0, \1`. Diese Transformierte kann nun sehr gut mit Hilfe des Huffman-Algorithmus komprimiert werden.

Literaturverzeichnis

[1] Alberto Apostolico and Zvi Galil, *Pattern Matching Algorithms.* Oxford University Press, New York, 1997.

[2] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[3] Maxime Crochemore and Wojciech Rytter, *Text Algorithms.* Oxford University Press, New York, 1994.

[4] Maxime Crochemore and Wojciech Rytter, *Juwels of Stringology.* World Scientific, Singapore, 2002.

[5] Maxime Crochemore, Christophe Hancart, et Thierry Lecroq, *Algorithmique du texte.* Vuibert, Paris, 2001.

[6] Dan Gusfield, *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, 1997.

[7] Volker Heun, *Grundlegende Algorithmen.* Vieweg Verlag, 2000.

[8] John Hopcroft, Rajeev Motwani und Jeffrey Ullman, *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Pearson Studium, München, 2002.

[9] Juha Kärkkäinen und Peter Sanders, *Simple linear work suffix array construction.* In: Proceedings of ICALP, 2003.

[10] Hans Werner Lang, *Algorithmen in Java.* Oldenbourg Verlag, 2003.

[11] Jesper Larsson, *Structures of String Matching and Data Compression*, Dissertation, Lund University, Sweden, 1999.
`http://www.cs.lth.se/~jesper/research.html`

[12] Gonzalo Navarro and Mathieu Raffinot, *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.
`http://www.dcc.uchile.cl/~gnavarro/FPMbook/`

[13] G.A. Stephen, *String Searching Algorithms.* World Scientific, Singapore, 1994.

[14] W. F. Smyth, *Computing Patterns in Strings*, Pearson Addison Wesley (UK), 2003.

[15] A.C. Yao, The complexity of pattern matching for a random string, *SIAM Journal on Computing*, 8(3):368-387, 1979.

Inhaltsverzeichnis