Algorithmen auf Sequenzen

Dipl.-Inf. Dominik Kopczynski
JProf. Dr. Tobias Marschall
Dr. Marcel Martin
Prof. Dr. Sven Rahmann

Lehrstuhl XI, Fakultät für Informatik, TU Dortmund

INF-BSc-315, Wintersemester 2014/2015
INF-BSc-315, Wintersemester 2012/2013
INF-BSc-315, Sommersemester 2011
INF-BSc-315, Sommersemester 2010
Spezialvorlesung DPO 2001, Wintersemester 2009/10
Spezialvorlesung DPO 2001, Sommersemester 2008

Entwurf vom 24. April 2020

Inhaltsverzeichnis

Vorbemerkungen

Dieses Skript enthält Material der Vorlesung „Algorithmen auf Sequenzen“, die ich an der TU Dortmund seit 2008 gehalten habe. Es gibt dieses Modul einerseits als 3V+1Ü (6 LP) als Spezialvorlesung in den Schwerpunktgebieten 4, 6 und 7 laut Diplomprüfungsordnung (DPO 2001), andererseits als Bachelor-Wahlmodul (INF-BSc-315) mit reduziertem Umfang von 2V+1Ü. Die behandelten Themen variieren ein wenig von Semester zu Semester. Grundsätzlich sind Kapitel und Abschnitte mit Stern (*) im Titel eher der Spezialvorlesung als dem Bachelor-Wahlmodul zuzuordnen.

Das Skript befindet sich noch in der Entwurfsphase; es ist somit wahrscheinlich, dass leider noch einige Fehler darin enthalten sind, vor allem in den neueren Abschnitten. Ich bedanke mich herzlich bei Katharina Diekmann und Jakob Bossek, die bereits zahlreiche Fehler gefunden und verbessert haben. Für die verbleibenden bin selbstverständlich ich allein verantwortlich.

Dortmund, Oktober 2014 Sven Rahmann

KAPITEL 1

Motivation und Einführung

In der Sequenzanalyse beschäftigen wir uns mit der Analyse von sequenziellen Daten, also Folgen von Symbolen. Sequenzen sind „eindimensional“ und daher einfach darzustellen und zu analysieren. Schwieriger sind zum Beispiel Probleme auf Graphen. Viele Informationen lassen sich in Form von Sequenzen darstellen (serialisieren). Man kann sogar behaupten, dass sich *jede* Art von Information, die zwischen Menschen ausgetauscht werden kann, serialisieren lässt. Auch die Darstellung von beliebigen Informationen im Speicher eines Computers erfolgt letztendlich als Bit-Sequenz.

1.1 Beispiele und Fragestellungen der Sequenzanalyse

Einige natürliche Beispiele für Sequenzen sind

- Biosequenzen (DNA, RNA, Proteine). Aber: Genome sind komplexer als nur eine DNA-Sequenz; d.h. die Darstellung eines Genoms als Zeichenkette stellt eine vereinfachende Modellannahme dar.
- Texte (Literatur, wissenschaftliche Texte). Die Kunst hinter guter Literatur und hinter guten wissenschaftlichen Arbeiten besteht darin, schwierige, komplex zusammenhängende Sachverhalte in eine logische Abfolge von einzelnen Sätzen zu bringen.
- Quelltexte von Programmen
- Dateien, Datenströme. Komplexe Datenstrukturen werden serialisiert, um sie persistent zu machen.
- Zeitreihen, Spektren (Audiosignale, Massenspektren, ...).

Die Sequenzanalyse umfasst unter anderem folgende Probleme:

- *Mustersuche*: Wir suchen in einer vorgegebenen Sequenz ein bestimmtes Muster, z.B. einen regulären Ausdruck. Ein Beispiel ist die „Suchen"-Funktion in Textverarbeitungsprogrammen. Die Mustersuche kann *exakt* oder *approximativ* erfolgen. Bei der approximativen Suche sollen nicht nur exakt passende, sondern auch ähnliche Muster gefunden werden (z.B. Meier statt Mayer).
- *Sequenzvergleich*: Ermitteln und Quantifizieren von Gemeinsamkeiten und Unterschieden verschiedener gegebener Sequenzen. Dies ist eine wichtige Anwendung im Kontext biologischer Sequenzen, aber auch im Bereich der Versions- und Revisionskontrolle (CVS, Subversion, git, Mercurial, etc.).
- *Kompression*: Wie kann eine gegebene Symbolfolge möglichst platzsparend gespeichert werden? Je mehr Struktur bzw. Wiederholungen in einer Sequenz vorkommen, desto besser kann man sie komprimieren. Dies liefert implizit ein Maß für die Komplexität einer Sequenz.
- *Muster- und Signalentdeckung*: Im Gegensatz zur Mustersuche, wo nach einem bekannten Muster gesucht wird, geht es hier darum, „Auffälligkeiten" in Sequenzen zu entdecken, zum Beispiel häufig wiederholte Teilstrings (nützlich für Genomanalyse, Kompression) Ein Beispiel: Wenn man einen englischen Text vor sich hat, der durch eine einfache monalphabetische Substitution verschlüsselt wurde, kann man sich relativ sicher sein, dass der häufigste Buchstabe im Klartext einem „e" entspricht.

1.2 Grundlegende Definitionen

Wir wollen nötige Grundbegriffe nun formal einführen.

1.1 Definition (Alphabet)**.** Ein *Alphabet* ist eine (endliche oder unendliche) Menge.

Wir befassen uns in der Regel mit *endlichen* Alphabeten, die wir normalerweise mit Σ (manchmal mit A) bezeichnen.

1.2 Definition (Indexmenge)**.** Eine *Indexmenge* ist eine endliche oder abzählbar unendliche linear geordnete Menge.

Wir erinnern an den Begriff *lineare Ordnung* (auch: totale Ordnung) in der Definition der Indexmenge: Eine Relation $\leq$ heißt Halbordnung, wenn sie reflexiv ($a \leq a$), transitiv ($a \leq b$ und $b \leq c \implies a \leq c$) und antisymmetrisch ($a \leq b$ und $b \leq a \implies a = b$) ist. Eine Halbordnung ist eine totale Ordnung oder lineare Ordnung, wenn zudem je zwei Elemente vergleichbar sind, also $a \leq b$ oder $b \leq a$ für alle a, b gilt.

Wir bezeichnen Indexmengen mit $\mathcal{I}$. Typische Beispiele für Indexmengen sind $\mathbb{N}$, $\mathbb{Z}$ und $\{1, \ldots, N\}$ mit der üblichen Ordnung $\leq$.

1.3 Definition (Sequenz)**.** Eine Sequenz ist eine Funktion $s : \mathcal{I} \to \Sigma$, oder äquivalent, ein Tupel $s \in \Sigma^{\mathcal{I}}$.

Normalerweise befassen wir uns mit endlichen Sequenzen; dann ist $\mathcal{I} = \{0, \ldots, n-1\}$ für ein $n \in \mathbb{N}$. (Wir beginnen meist bei 0 und nicht bei 1 mit der Indizierung.) Für $\mathcal{I} = \{1, \ldots, n\}$ oder $\mathcal{I} = \{0, \ldots, n-1\}$ schreibt man vereinfachend auch Σ^n statt $\Sigma^{\mathcal{I}}$.

Sequenztyp	**Alphabet Σ**
DNA-Sequenz	{ A,C,G,T }
Protein-Sequenz	20 Standard-Aminosäuren
C-Programme	ASCII-Zeichen (7-bit)
Java-Programme	Unicode-Zeichen
Audiosignal (16-bit samples)	$\{0, \ldots, 2^{16} - 1\}$
Massenspektrum	Intervall $[0, 1]$ (unendlich) oder Double

Tabelle 1.1: Beispiele für Sequenzen über verschiedenen Alphabeten

1.4 Definition (Wörter, Mere, Gramme)**.** Die Elemente von Σ^n nennt man *Wörter*, *Tupel*, *Strings*, *Sequenzen* der Länge n sowie n*-Mere* oder n*-Gramme* (englisch: n-mers, n-grams) über Σ.

Für das i-te Element einer Sequenz schreiben wir s_i (Indizierung wie bei Folgen in der Mathematik) oder $s[i]$ (programmier-typische Indizierung), selten auch $s(i)$ (Funktionsschreibweise der Mathematik).

1.5 Beispiel (Sequenz)**.** $s =$ `AGGTC` ist eine Sequenz mit $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ (DNA-Alphabet), $\mathcal{I} = \{0, 1, 2, 3, 4\}$ in der üblichen Ordnung. Beispielsweise bildet s die 3 auf `T` ab, $s[3] =$ `T`. ♡

Tabelle 1.1 zeigt einige Beispiele für Sequenzen über verschiedenen Alphabeten.

1.6 Beispiel (Darstellung einer Sequenz in Java und Python)**.** In der Programmiersprache Java können Sequenzen auf unterschiedliche Arten repräsentiert werden, zum Beispiel als `String` (wenn $\Sigma \subset$ Unicode) oder `A[]` oder `ArrayList<A>` oder `Map<I,A>`.

In Python gibt es Strings, die durch Anführungzeichen (einfache oder doppelte) begrenzt und standardmäßig als Unicode-codiert interpretiert werden und den `bytes`-Typ, der „rohe“ Bytes repräsentiert. Ferner gibt es Listen (`list`), die durch `[]` begrenzt werden und veränderbar sind, und Tupel (`tuple`), die durch `()` begrenzt werden und nicht veränderbar sind. Es gibt auch „Wörterbücher“ (dictionaries, `dict`), die durch `{}` begrenzt werden; hier muss die Indexmenge ein unveränderbarer Typ sein (wie Strings oder Tupel). Python-Beispiele sind:

```
s = "ABCDE"
s = ['A','B','C','D','E']
s = ('A','B','C','D','E')
d = dict(enumerate(s))   # liefert {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E'}
# s[2] und d[2] liefern jeweils 'C'                                    ♡
```

In der Statistik spricht man häufig von *Zeitreihen* statt von Sequenzen. Hier hat die Indexmenge die Funktion eines Zeitparameters, und das Alphabet ist meist eine Teilmenge der reellen Zahlen. Zeitreihen sind also spezielle Sequenzen. In den Anwendungen der Informatik ist das Alphabet häufiger kategoriell (ungeordnet).

Wir kommen nun zu weiteren Definitionen im Zusammenhang mit Sequenzen.

1.7 Definition (Σ^+, Σ^*, leerer String ε)**.** Wir definieren $\Sigma^+ := \bigcup_{n \geq 1} \Sigma^n$ und $\Sigma^* := \bigcup_{n \geq 0} \Sigma^n$, wobei $\Sigma^0 = \{\varepsilon\}$ und ε *der leere String* ist. Der leere String ε ist der einzige String der Länge 0. Damit ist Σ^* die Menge aller *endlichen* Strings über Σ.

1.8 Definition (Teilstring, Teilsequenz, Präfix, Suffix)**.** Sei $s \in \Sigma^*$ ein String. Wir bezeichnen mit $s[i]$ den Buchstaben, der in s an der Stelle i steht. Dabei muss $i \in \mathcal{I}$ sein. Wir schreiben $s[i \dots j]$ für den *Teilstring* von i bis j (einschließlich). Falls $i > j$, ist per Definition $s[i \dots j] = \varepsilon$. Eine *Teilsequenz* von s definieren wir als $(s_i)_{i \in I}$ mit $I \subset \mathcal{I}$. Eine Teilsequenz ist im Gegensatz zum Teilstring also nicht notwendigerweise zusammenhängend. Die Begriffe Teilstring und Teilsequenz sind daher auseinanderzuhalten.

Weiter definieren wir $s[\dots i] := s[0 \dots i]$ und $s[i \dots] := s[i \dots |s| - 1]$ und bezeichnen solche Teilstrings als *Präfix* beziehungsweise *Suffix* von s. Wenn t ein Präfix (Suffix) von s ist und $t \neq \varepsilon$ und $t \neq s$, dann bezeichnen wir t als *echtes Präfix (Suffix)* von s.

Ferner definieren wir die Menge aller Präfixe / Suffixe von s durch

$$\text{Prefixes}(s) := \{\, s[\dots i] \mid -1 \leq i < |s| \,\}$$

und

$$\text{Suffixes}(s) := \{\, s[i \dots] \mid 0 \leq i \leq |s| \,\}.$$

Für eine Menge $S \subset \Sigma^*$ von Wörtern definieren wir

$$\text{Prefixes}(S) := \bigcup_{s \in S} \text{Prefixes}(s)$$

bzw.

$$\text{Suffixes}(S) := \bigcup_{s \in S} \text{Suffixes}(s).$$

1.3 Nützliche Literatur

Folgende Bücher (und andere) können beim Erarbeiten des in diesem Skript enthaltenen Stoff nützlich sein:

- **?**, *Flexible Pattern Matching in Strings*
- **?**, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*
- **?**, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*
- **?**, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*
- **?**, *Introduction to Computational Genomics – A Case Studies Approach*

Die genauen Quellenangaben befinden sich im Literaturverzeichnis.

KAPITEL 2

Bitsequenzen

In diesem Kapitel beschäftigen wir uns mit dem einfachsten denkbaren Sequenztyp, nämlich Sequenzen von Bits. Ein Bit (kurz für *binary digit*, Binärzahl) kann genau zwei Zustände annehmen, die wir mit Null und Eins bezeichnen. Die in diesem Kapitel betrachteten Sequenzen sind also aus der Menge $\{0, 1\}^*$.

2.1 Repräsentation und Manipulation von Bitsequenzen

Bitsequenzen sind im Computer die natürliche Form der Darstellung jeder Art von Information. Daher sollte man beim Programmieren darauf achten, Bitsequenzen auch möglichst hardwarenah zu verwenden und tatsächlich nur ein Bit pro Bit zu speichern. Natürlich kann man beispielsweise in Python die eingebauten Listen `[0,1,1,0,1]` verwenden und nur mit Nullen und Einsen befüllen. In diesem Fall wird jedoch an jeder Stelle der Liste ein Verweis auf ein Objekt (auf das Null-Objekt oder das Eins-Objekt) gespeichert. Diese Verweise sind Zeiger und benötigen jeweils, je nach Rechnerarchitektur, 32 oder 64 Bits. Das Listenobjekt und das Null- und Eins-Objekt benötigen ihrerseits auch noch ein wenig (konstant viel) Speicher, sagen wir c Bits. Damit wurde man $c + 64n$ Bits Speicher für 64 Bits benötigen, was keine gute Idee ist.

Nicht alle Sprachen unterstützen Bitsequenzen (oder Bit-Arrays) direkt. In Python kann man die `bitarray`-Bibliothek[1] verwenden, die in C geschrieben ist. Die meisten Sprachen erlauben allerdings hardwarenahen Zugriff auf zusammenhängende Speicherbereiche (Arrays) von Maschinenwörtern. Wir erinnern dazu an einige Begriffe: Ein Bit kann zwei Zustände annehmen; ein Byte besteht aus acht Bits und kann $2^8 = 256$ Zustände annehmen. Ein

[1] https://pypi.python.org/pypi/bitarray/

(Maschinen-)Wort besteht aus $W \in \{16, 32, 64, 128\}$ Bits; die genaue Zahl W (die „Wortbreite“) ist abhängig von der Maschinenarchitektur, aber fast immer eine Zweierpotenz; wir setzen $W = 64$ voraus. Will man n Bits speichern, benötigt man dafür $\lceil n/W \rceil$ Wörter.

Sei $s = (s[i])_{i=0}^{n-1}$ eine Bitsequenz der Länge n. Sei dazu $B = (B[j])_{j=0}^{\lceil n/W \rceil - 1}$ die Sequenz der Maschinenwörter. Wir betrachten nun Operationen auf s und wie diese mit B implementiert werden.

Bit-Operationen auf Bitsequenzen. Zunächst betrachten wir einfache bitweise Operationen auf Bitsequenzen. Seien s, t Bitsequenzen derselben Länge n.

Die *bitweise Negation* $\sim s$ einer Sequenz s ist definiert als

$$(\sim s)[i] := \sim s[i],$$

wobei $\sim 0 := 1$ und $\sim 1 := 0$.

Das *bitweise Und* $s \,\&\, t$ ist definiert als

$$(s \,\&\, t)[i] := s[i] \,\&\, t[i],$$

wobei $0 \,\&\, 0 := 0$, $0 \,\&\, 1 := 0$, $1 \,\&\, 0 := 0$ und $1 \,\&\, 1 := 1$.

Das *bitweise Oder* $s \mid t$ ist definiert als

$$(s \mid t)[i] := s[i] \mid t[i],$$

wobei $0 \mid 0 := 0$, $0 \mid 1 := 1$, $1 \mid 0 := 1$ und $1 \mid 1 := 1$.

Das *bitweise Exklusive Oder* $s \oplus t$ ist definiert als

$$(s \oplus t)[i] := s[i] \oplus t[i],$$

wobei $0 \oplus 0 := 0$, $0 \oplus 1 := 1$, $1 \oplus 0 := 1$ und $1 \oplus 1 := 0$. Man kann noch weitere Operationen (nand, equiv, etc.) definieren.

Die Implementierung mittels der Wortsequenz B ist hier ganz einfach: Man wendet die gewüunschte Operation einfach nacheinander oder parallel auff jedes Wort in B an. Für fast alle diese Bit-Operationen stehen Maschineninstruktionen zur Verfügung, so dass eine entsprechende Operation grundsätzlich direkt in den entsprechenden Prozessorbefehl übersetzt werden kann.

Bit-Operationen auf einzelnen Wörtern. Wir nummerieren die Bits in einem Wort traditionell von rechts nach links! Das hängt mit der Wertigkeit der Bits in der Binärdarstellung von Zahlen zusammen: In einer Binärzahl wie $(10011)_2$ hat das rechteste Bit (Nummer 0) die Wertigkeit $2^0 = 1$, das Bit links daneben (Nummer 1) die Wertigkeit $2^1 = 2$, allgemein das k-te Bit die Wertigkeit 2^k, so dass sich hier (von rechts nach links) der Wert $1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 = 19$ ergibt.

Wir betrachten auf den W Bits eines Wortes $s = (s[k])_{k=0}^{W-1}$ nun die Operationen *Linksverschiebung* $\ll$ und *Rechtsverschiebung* $\gg$ um jeweils $b \geq 0$ Bits. Es ist

$$(s \ll b)[i] = s[i - b],$$

falls $0 \leq i - b < W$, ansonsten wird der Wert als Null definiert. Analog ist

$$(s \gg b)[i] = s[i + b],$$

falls $0 \leq i + b < W$, sonst Null.

Die Linksverschiebung um b Bits entspricht (sofern kein Überlauf auftritt) einer Multiplikation mit 2^b. Die Rechtsverschiebung um b Bits entspricht einer ganzzahligen Division (ohne Rest) durch 2^b.

Die Operationen Links- und Rechtsverschiebung sind auf längeren Bitsequenzen verwirrend, weil wird die Wörter in der Regel aufsteigend von Links nach Rechts, die Bits innerhalb eines Wortes aber von Rechts nach Links nummerieren. Daher wenden wir diese Operationen nur innerhalb eines Wortes an.

Zugriff auf Bit i. Wir wollen nun in s den Zustand von $s[i] \in \{ 0, 1 \}$ bestimmen. Bit i mit $0 \leq i < n$ steht im Wort mit dem Index $j := \lfloor i/W \rfloor$ und ist darin das Bit Nummer $k := i - jW = i\%W$, wobei % die Modulo-Operation bezeichnet.

Da W eine Zweierpotenz ist, $W = 2^w$ (für $W = 64$ ist $w = 6$) lässt sich die Berechnung $i \mapsto (j, k) = (\lfloor i/W \rfloor, i\%W)$ effizient mit Bit-Operationen gestellten: Die ganzzahlige Division durch W entspricht einer Rechtsverschiebung ($\gg$) um w Bits, und der Rest enstspricht gerade den niederwertigsten w Bits von i, so dass man diesen durch Verunden mit einem Wort M aus w Einsen an den niederwertigsten Bits erhält. Dieses entspricht wiederum dem Wert $M = (\underbrace{0 \ldots 0}_{W-w}\underbrace{1 \ldots 1}_{w})_2 = 2^w - 1 = (1 \ll w) - 1$. Man berechnet also die Abbildung

$$i \mapsto (j, k) = (i \gg w,\, i \,\&\, M)\,.$$

Umgekehrt berechnet man

$$(j, k) \mapsto i = j \ll w \mid k = j \ll w + k\,.$$

Da nach der Linksverschiebung ($\ll$) die rechten w Bits auf Null gesetzt sind, spielt es keine Rolle, ob man k addiert oder mit k verodert, da in k nach Voraussetzung nur die rechten w Bits gesetzt sein können.

Um $s[i]$ zu bestimmen, müssen wir also das k-te Bit aus $B[j]$ auslesen. Dies geschieht durch den Ausdruck $B[j]$ & 2^k; dieser hat einen Wert in $\{ 0, 2^k \}$. Um die Werte 0 oder 1 zu erhalten, kann man das Ergebnis entweder um k Bits nach rechts verschieben oder einfach nur auf „ungleich Null“ testen. Insgesamt ist also

$$s[i] = \Big(\big(B[j] \,\&\, (1 \ll k)\big) \neq 0\Big) = \Big(\big(B[i \gg w] \,\&\, (1 \ll (i \,\&\, M))\big) \neq 0\Big)\,.$$

Setzen und Löschen von Bit i. Da man im RAM meist nur auf einzelne Wörter, aber nicht auf einzelne Bits zugreifen kann, muss man, um ein Bit zu setzen oder zu löschen, zunächst das ganze Wort auslesen, neu berechnen und zurückschreiben. Zum Index i berechnen wir Wortnummer j und Bitnummer k wie gehabt. Um das Bit zu setzen, unabhängig davon, ob es vorher gesetzt oder gelöscht war, verodern wir $B[j]$ mit einer Bitmaske, in der nur Bit k

gesetzt ist; diese hat den Wert $(1 \ll k) = 2^k$. Um das Bit zu löschen, verunden wir es mit der negierten Maske $\sim(1 \ll k)$. Zusammengefasst:

$$\begin{aligned} \text{Bit } i \text{ setzen:} &\quad B[j] \leftarrow B[j] \mid (1 \ll k) \\ \text{Bit } i \text{ löschen:} &\quad B[j] \leftarrow B[j] \,\&\, \sim(1 \ll k) \end{aligned}$$

Da man aus i die Indexzahlen j und k in konstanter Zeit (sogar mit wenigen Maschinenbefehlen, also sehr schnell) berechnen kann, kostet das Auslesen, Setzen und Löschen einzelner Bits auch nur konstante Zeit und geht in der Praxis schnell.

Dünnbesetzte Bitsequenzen. Mit der bisher betrachteten Methode beötigt man zum Speichern eines Feldes von n Bits $n + o(n)$ Bits. Der $o(n)$ Term enthält alle benötigten Verwaltungsinformationen wie beispielsweise $\log n$ Bits zum Speichern der Länge n. (Alle Logarithmen in diesem Kapitel sind Logarithmen zur Basis 2). In der Praxis fällt dieser Term kaum ins Gewicht.

Wenn man vorher weiß, dass in der Anwendung nur wenige Eins-Bits (oder wenige Null-Bits) auftreten, ist es ggf. sparsamer, nicht die einzelnen Bits, sondern nur die Indizes der Eins-Bits (Null-Bits) zu speichern. Angenommen, es gibt nur m Eins-Bits; dann benötigt man dafür $m \log n$ Bits. Mit Verwaltungsinformationen kommt man auf $(m + O(1)) \log n$ Bits. Ist $m < n/\log n$ (ein durchaus häufiger Fall), kann sich diese Speicherreduktion lohnen. Der Nachteil ist, dass man nicht mehr alle der Operationen Auslesen, Setzen und Lösen in konstanter Zeit durchführen kann. Es gibt aber noch bessere Codierungsmethoden in solchen Fällen; wir gehen hier nicht näher darauf ein.

2.2 Felder von Zahlen in Theorie und Praxis

Um eine Zahl aus dem Zahlenbereich $\{0, \ldots, 2^b - 1\}$ (oder bei Zweierkomplementarstellung aus dem Bereich $\{-2^{b-1}, \ldots, -1, 0, 1, \ldots, 2^{b-1} - 1\}$) darzustellen, benötigt man b Bits. Anders gesagt: Ist eine obere Schranke $z \geq 1$ für darzustellende Zahlen bekannt (und dies ist der Kernpunkt!), dann kann jede Zahl im Bereich $\{0, \ldots, z\}$ (einschließlich z) mit $b(z) := 1 + \lfloor \log z \rfloor$ Bits repräsentiert werden.

Um nun n Zahlen in diesem Bereich zu speichern, sind also $n \cdot b(z)$ Bits notwendig. Man erkennt leicht die Vor- und Nachteile dieses Verfahrens: Der Zugriff auf die i-te Zahl ist in konstanter Zeit möglich, man muss ja nur die $b(z)$ Bits ab dem Index $i \cdot b(z)$ auswählen; die Startpositionen der dargestellten Zahlen in der Bitsequenz sind äquidistant. Sind aber viele der dargestellten Zahlen von deutlich kleinerer Größenordnung als z, dann ist diese Art der Darstellung sehr verschwenderisch.

In der Praxis ergibt sich ein weiteres Problem: Auf modernen Rechnern ist nämlich die Registerbreite mit $W = 32$ oder $W = 64$ im wesentlichen vorgegeben (wenn man die effizienten CPU-Operationen einsetzen will); eine Registerbreite müsste man softwareseitig wie oben beschrieben selbst implementieren. Natürlich beeinflusst das in der Theorie immer nur die „konstanten Faktoren“ in der Laufzeit oder im Speicherbedarf; in der Praxis sind diese Effekte auf modernen Rechnerarchitekturen jedoch erheblich.

Während sich Theorie-Ergebnisse daher relativ elegant mit der $\mathcal{O}$-Notation unter Vernachlässigung konstanter Faktoren darstellen lassen, verwendet man in der Praxis allerlei Tricks, um (auch bei bereits asymptotisch optimalen Verfahren) Platz oder Zeit zu sparen.

Wir geben ein einfaches Beispiel aus der Praxis: Wir betrachten ein Array A von n nichtnegativen Zahlen, von denen viele zwischen 0 und 255 liegen, einige aber auch sehr groß werden können (aber kleiner als 2^{64} sind). Statt nun $64n$ Bits zu verwenden, benutzen wir ein Byte-Feld mit $8n$ Bits und speichern den Wert 255 bei Index i, sofern $A[i] \geq 255$. Es sei $m := |\{ i \mid A[i] \geq 255 \}|$ die Anzahl solcher Ausnahmen. Nach Voraussetzung ist m klein. Wir speichern nun alle Ausnahmen in zwei Arrays I und X der Länge m, so dass I die Ausnahme-Index-Werte i in aufsteigender Reihenfolge und X die entsprechenden $A[i]$-Werte enthält. Dafür werden also $2 \cdot 64 \cdot m$ Bits benötigt. Der passende Index j mit $X[j] = A[i]$ muss in I mit Hilfe binärer Suche gefunden werden, das dauert $\mathcal{O}(\log m)$ Zeit.

Insgesamt muss man beim Speicherbedarf $64n$ mit $8n + 128m$ Bits vergleichen. Bei der Zugriffszeit hat man bei der ersten Variante immer $\mathcal{O}(1)$ gegenüber der anderen Variante mit $\mathcal{O}(1)$ im Fall einer nicht-Ausnahme und $\mathcal{O}(\log m)$ im Fall einer Ausnahme. Das ist (praktisch gesehen) so gut wie konstant für kleine Werte von m. Wird insbesondere das Array A linear in einem Indexbereich $i_1 \ldots i_2$ durchlaufen, muss man nur den ersten Ausnahmeindex j in I suchen; die folgenden Ausnahme-Werte folgen ja konsekutiv in X.

Es solte klar sein, dass man dieses Beispiel verallgemeinern kann und ein solches Array objektorientiert implementieren kann, so dass der Benutzer nicht merkt (und nicht wissen muss), dass mit Ausnahmetabellen gearbeitet wird.

2.3 Population Count

Sei x ein einzelnes Maschinenwort aus W Bits. Wir betrachten das einfache aber interessante Problem, die Anzahl der 1-Bits in x zu zählen (*population count* oder *popcount*, Einwohnerzahl; auch: Hamming-Gewicht).

Zuvor rufen wir uns noch kurz eine der Grundannahmen des RAM-Modells ins Gedächtnis: Wenn wir Probleme auf Sequenzen der Länge n betrachten, nehmen wir normalerweise an, dass wir Operationen wie Addition, Multiplikation, etc. auf $\Theta(\log n)$ Bits in konstanter Zeit durchführen können. Das bedeutet zum Beispiel, dass wir eine Rechnung wie $n + n$ in konstanter Zeit durchfuhren können (statt in $\mathcal{O}(\log n)$ Zeit), obwohl wir ja $\Theta(\log n)$ Bits betrachten müssen. Das ist insofern realistisch, als Instruktionen auf einer W-Bit-Architektur auf W-Bit-Wörtern elementar als Schaltkreise realisiert sind und man niemals Sequenzen betrachten wird, die länger als $n = 2^W$ sind. Wir setzen also immer $W = \Theta(\log n)$ voraus, wenn wir mit Sequenzen der Länge n arbeiten.

Manchen Rechnerarchitekturen wie Cray oder Intel SSE 4.2 (seit 2008) bieten für Maschinenwörter einen eigenen popcount-Befehl. Wir diskutieren hier, wie man popcount mit anderen elementaren Operationen implementiert, wenn es popcount selbst nicht als elementare Operation gibt.

Gegeben sei ein Wort $x = (x_{W-1}, \ldots, x_0)$ (die Indizierung erfolgt rückwärts, da wir ein einzelnes Maschinenwort der Länge W betrachten). Wir werden x in $\log W$ Schritten (ist

```
Gegeben:     x = ( 1 1 0 1 1 0 0 0 1 1 1 1 1 1 1 1 )
Addition 0:  x = ( 1|1|0|1|1|0|0|0|1|1|1|1|1|1|1|1 ) (Einergruppen)
        x & M1 = ( 0|1|0|1|0|0|0|0|0|1|0|1|0|1|0|1 ) Auswahl gerader Bits
     x>>1 & M1 = ( 0|1|0|0|0|1|0|0|0|1|0|1|0|1|0|1 ) Auswahl ungerader Bits
Addition 1:  x = ( 1 0|0 1|0 1|0 0|1 0|1 0|1 0|1 0 ) Zweiergruppen
        x & M2 = ( 0 0|0 1|0 0|0 0|0 0|1 0|0 0|1 0 ) Auswahl gerader Zweiergruppen
     x>>2 & M2 = ( 0 0|1 0|0 0|0 1|0 0|1 0|0 0|1 0 ) Auswahl ungerader Zweierguppen
Addition 2:  x = ( 0 0 1 1|0 0 0 1|0 1 0 0|0 1 0 0 ) Vierergruppen
        x & M3 = ( 0 0 0 0|0 0 0 1|0 0 0 0|0 1 0 0 ) Auswahl gerader Vierergruppen
     x>>4 & M3 = ( 0 0 0 0|0 0 1 1|0 0 0 0|0 1 0 0 ) Auswahl ungerader Vierergruppen
Addition 3:  x = ( 0 0 0 0 0 1 0 0|0 0 0 0 1 0 0 0 ) Achtergruppen
        x & M4 = ( 0 0 0 0 0 0 0 0|0 0 0 0 1 0 0 0 ) Auswahl gerader Achtergruppen
     x>>8 & M4 = ( 0 0 0 0 0 0 0 0|0 0 0 0 0 1 0 0 ) Auswahl ungerader Achtergruppen
Addition 4:  x = ( 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 ) eine Sechzehnergruppe: Wert 12
```

Tabelle 2.1: Beispiel zur Berechnung der Funktion *popcount* mit Hilfe elementare bitweiser und arithmetischer Operationen. Bitgruppen sind zur Illustration in jedem Schritt durch vertikale Striche getrennt. Nach Addition j enthält jede Gruppe in x die Anzahl der 1-Bits der entsprechenden Gruppe im ursprünglichen Wort.

$W = \mathcal{O}(\log n)$, dann sind das $\mathcal{O}(\log \log n)$ Schritte) so modifizieren, dass zum Schluss x den population count seines ursprünglichen Wertes enthält.

Jedes der W Bits in x für sich zählt bereits korrekt die Anzahl seiner 1-Bits (0 oder 1). Es folgen nun $w = \log W$ Summationsschritte. Nach Schritt j, $0 \leq j \leq w$, denken wir uns x in $W/2^j$ Gruppen von jeweils 2^j Bits unterteilt. Das Bitmuster jeder Gruppe in x repräsentiert die Zahl der 1-Bits dieser Gruppe im ursprünglichen Wort. Der Ausgangszustand entspricht also dem Zustand nach Schritt 0.

In Schritt 1 definieren wir eine Bitmaske, die die Bits mit geradem Index auswählt, also $M_1 = (0101 \ldots 01)_2$. Wir berechnen nun aus dem Ausgangswert x den Ausdruck $(x \,\&\, M_1) + ((x \gg 1) \,\&\, M_1)$ (das Plus ist ein normales arithmetisches Plus) und weisen diesen wieder x zu.

Allgemein definieren wir für Schritt j die Bitmaske M_j, die aus der Wiederholung von 2^{j-1} Nullen gefolgt von 2^{j-1} Einsen besteht und setzen $x \leftarrow (x \,\&\, M_j) + ((x \gg 2^{j-1}) \,\&\, M_j)$.

Von Schritt zu Schritt wird die Gruppengröße verdoppelt und die Zahlen aus je zwei kleineren Gruppen addiert. Ein Beispiel für $W = 16$ soll dies verdeutlichen; siehe Tabelle 2.1. Man kann dies in C-Code mit einigen Tricks noch effizienter codieren[2]; hier wird $W = 64$ angenommen.

```
1  const uint64 M1   = 0x5555555555555555; // 0101....
2  const uint64 M2   = 0x3333333333333333; // 00110011...
3  const uint64 M3   = 0x0f0f0f0f0f0f0f0f; // 0000111100001111...
4  const uint64 H256 = 0x0101010101010101; // 256^0 + 256^1 + 256^2 + ...
5  int popcount(uint64 x) {
6      x -= (x >> 1) & M1;              // Zweiergruppen
7      x = (x & m2) + ((x >> 2) & M2); // Vierergruppen
8      x = (x + (x >> 4)) & M3;         // Achtergruppen
9      return (x * H256) >> 56;
10     // die letzten 8 Bits von x + (x<<8) + (x<<16) + (x<<24) + ...
```

[2] Quelle: http://en.wikipedia.org/wiki/Hamming_weight

```
11 }
```

Wenn man vorher weiß, dass die Anzahl der 1-Bits in einem Wort klein ist (etwa $m = \mathcal{O}(\log W)$ 1-Bits), ist folgendes Verfahren in der Praxis interessant: Mit der Instruktion `x = x & (x-1)` lässt sich das rechteste 1-Bit von x löschen (Beweis: Übung). Dies wiederholt man so lange wie $x \neq 0$ gilt, und zählt dabei die Iterationen.

```
1 int popcount(uint64 x) {
2     int count;
3     for (count=0; x; count++)  x &= x-1;
4     return count;
5 }
```

Zum Schluss stellen wir noch eine Methode mit Hilfe von vorberechneten Tabellen vor, die auch Skeptiker davon überzeugt, dass man den population count von $\mathcal{O}(\log n)$ Bits in konstanter Zeit berechnen kann. Hierbei sei n hinreichend groß.

Es sei $K > 1$ eine Konstante und $B := (\log n)/K$. (Wir verzichten der Lesbarkeit halber auf Rundungsoperationen zu ganzen Zahlen in der Darstellung; B wird aber als ganzzahlig angekommen.) Wir berechnen für alle 2^B Zahlen im Bereich $\{0, \ldots, 2^B - 1\}$ die population counts vor und speichern sie in einer Tabelle. Eine Zahl hat B Bits, ihr population count daher $\log B$ Bits. Die Tabelle benötigt also $2^B \cdot \log B$ Bits, das sind $\mathcal{O}(n^{1/K} \cdot \log\log n) = o(n)$. Für jede Gruppe von B Bits können wir also den population count einfach in konstanter Zeit in der Tabelle ablesen. Um auf den population count von $\log n$ Bits zu kommen, müssen wir konstant viele nachgeschlagene Werte (nämlich K) addieren; auch dies kostet nur konstante Zeit. Je größer K gewählt wird, um so weniger zusätzlicher Speicher wird benötigt, aber der konstante Zeitfaktor wächst.

Wir haben nun verschiedene Verfahren kennengelernt, um den population count eines Maschinenwortes der Länge W effizient zu berechnen. In unserem Maschinenmodell nehmen wir an, dass $W = O(\log n)$ gilt und dass die Berechnung in konstanter Zeit möglich ist.

2.4 Zählanfragen an Bitsequenzen

Sei s eine Bitsequenz der Länge n. Gesucht ist die Anzahl der Einsen in $s[\ldots i]$, die wir mit $\mathit{rank}_s(i)$ bezeichnen. Der Name *rank* ist ein wenig unglücklich gewählt, hat sich aber eingebürgert. Es handelt sich dabei um nichts anderes als den popcount von $s[\ldots i]$.

Natürlich lässt sich diese Funktion leicht mit Hilfe einer Schleife über $i + 1$ Bits berechnen; das kostet $\mathcal{O}(i)$ Zeit.

Tatsächlich können wir ja aber den popcount von $W = \Theta(\log n)$ Bits in konstanter Zeit berechnen, so dass wir nur $\mathcal{O}(i/\log n)$ Zahlen summieren müssen. (Das Wort, das das i-te Bit enthält, muss ggf. gesondert behandelt werden, indem man höherwertige Bits maskiert, bevor man die Maschinenwort-popcount-Operation aufruft.)

Es gilt also: Man kann für ein festes $i < |s| = n$ die Zahl $\mathit{rank}_s(i)$ in $\mathcal{O}(n/\log n)$ Zeit berechnen.

Wir wenden uns nun der Frage zu, wie wir Zählanfragen $rank_s(i)$ für Präfixe beliebiger Länge i auf der Bitsequenz s in *konstanter* Zeit beantworten können, wenn ein wenig zusätzlichen Speicher für vorverarbeitete Informationen bereitstellen.

Hätten wir $n \log n$ zusätzliche Bits zur Verfügung, wäre das Problem trivial: Wir speichern einfach für jeden Index i die Zahl $rank_s(i)$ in einem Array ab.

Wir wollen aber versuchen, mit $o(n)$ Bits auszukommen, so dass der Mehrbedarf an Speicher pro Bit asymptotisch gegen Null geht und nicht wächst. Offenbar kann man also nicht jeden Wert vorberechnen und abspeichern. Auch wenn man nur jeden k-ten Wert speichert (k konstant), benötigte man noch $(n \log n)/k$ Bits.

Der erste Teil der Lösung liegt darin, k eben nicht konstant, sondern als $k = \lceil (\log n)^2 \rceil$ zu wählen. Für die gespeicherten Werte werden dann nur $(n \log n)/(\log n)^2 = n/\log n \in o(n)$ Bits benötigt. Zu einer Position i bestimmt man nun $\max \{ m \mid mk \leq i \}$ und schlägt in der vorberechneten Tabelle an Index m die Zahl $rank_s(mk)$ nach. Nun muss man noch die 1-Bits im Bereich $mk + 1$ bis i zählen; das sind $\mathcal{O}((\log n)^2)$ viele Bits. Das können wir noch nicht in konstanter Zeit; wir sind also noch nicht fertig.

Wir nennen jeden Abschnitt der Länge k einen Superblock. Jeden Superblock unterteilen wir in $\sqrt{k}$ Blöcke der Länge $\sqrt{k}$. Für den j-ten Block innerhalb eines Superblocks, $0 \leq j < \sqrt{k}$, ist gespeichert, wie viele 1-Bits innerhalb des Superblocks bis zum Ende des j-ten Blocks enthalten sind. Diese Zahl beträgt maximal k und kann daher in $\log k$ Bits gespeichert werden. Insgesamt gibt es $(n/k) \cdot \sqrt{k} = n/\sqrt{k}$ Blöcke. Benötigt werden für diese Zahlen also $(n/\log n) \cdot \log \log n \in o(n)$ Bits, da $\sqrt{k} = \log n$.

Mit diesem zweistufigen Schema sind wir schon fertig! Wir benötigen insgesamt nur $o(n)$ Bits für die Superblock-Tabelle und die Block-Tabelle zusammen. Um $rank_s(i)$ zu berechnen, finden wir zu i zunächst den Index des entsprechenden Superblocks und bestimmen mit Hilfe der Superblock-Tabelle die Anzahl der 1-Bits bis dorthin in konstanter Zeit. Innerhalb des Superblocks bestimmen wir den Index des korrekten Blocks und addieren mit Hilfe der Block-Tabelle die Anzahl der 1-Bits vor dem Block in konstanter Zeit. Die verbleibenden höchstens $\log n$ Bits passen in ein Maschinenwort und wir bestimmen ihren population count ebenfalls in konstanter Zeit.

Wir fassen zusammen.

2.1 Satz. *In einer Bitsequenz der Länge n lässt sich $rank_s(i)$ für jedes i mit $0 \leq i < n$ in konstanter Zeit berechnen, wenn man $o(n)$ zusätzlichen Bits für die Superblock- und Block-Tabellen aufwendet.*

In der Praxis ergibt sich für realistische Werte von n immer noch ein erheblicher Mehrbedarf. Als Übung schlagen wir vor, das hier vorgeschlagene Verfahren einmal selbst zu implementieren (und zu debuggen!). Wie hoch ist (in Prozent) der zusätzliche Speicherbedarf für verschiedene Werte von n?

Da ein linearer Scan durch einen kurzen Teil einer Bitsequenz relativ schnell ist, wird man in der Praxis eine passende Stichproben-Rate k gemäß vorhandenem Speicher auswählen, auf die zweite Stufe des Verfahrens verzichten und die bis zu k Bits linear mehrere durch population-count-Operationen von aufeinander folgenden Maschinenwörtern berechnen. Das ist einfacher zu implementieren und (außer für extrem große n) außerdem schneller. Auch hier schlagen wir vor, zu experimentieren. Nähere Hinweise gibt die Arbeit von **?**.

KAPITEL 3

Pattern-Matching-Algorithmen für einfache Strings

In diesem Abschnitt betrachten wir das einfachste Pattern-Matching-Problem, das vorstellbar ist, und verschiedene Algorithmen zu seiner Lösung.

3.1 Das Pattern-Matching-Problem

3.1 Problem (einfaches Pattern-Matching)**.**

Gegeben: Alphabet Σ, Text $T \in \Sigma^n$, Pattern/Muster $P \in \Sigma^m$. Das Muster ist also ein einfacher String (später: komplexere Muster). Üblicherweise ist $m \ll n$.

Gesucht (3 Varianten):

1. Entscheidung: Ist P ein Teilstring von T?
2. Anzahl: Wie oft kommt P als Teilstring von T vor?
3. Aufzählung: An welchen Positionen (Start- oder Endposition) kommt P in T vor?

•

Algorithmen, die eine dieser Fragen beantworten, lassen sich oft (aber nicht immer) auf einfache Weise so modifizieren, dass sie auch die anderen beiden Fragen beantworten. Wir werden hier vor allem die vollständige Aufzählung der Positionen betrachten.

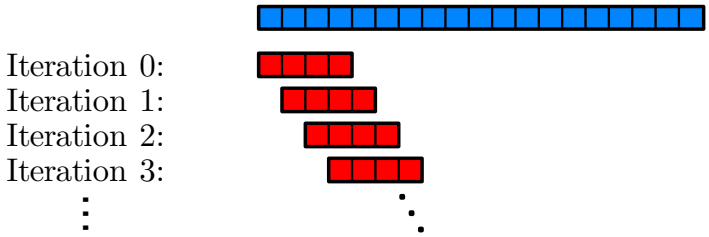


Abbildung 3.1: Naiver Algorithmus zum Pattern-Matching. Rot: Pattern. Blau: Text.

3.2 Ein naiver Algorithmus

Zunächst behandeln wir einen sehr einfachen (naiven) Algorithmus. Das Pattern wird in jeder Iteration mit einem Teilstring des Textes verglichen und nach jedem Vergleich um eine Position nach rechts verschoben. Der Vergleich in einer Iteration endet, sobald feststeht, dass das Pattern hier nicht passt (beim ersten nicht übereinstimmenden Zeichen, engl. mismatch).

Wir geben Algorithmen als Python-Code an, wobei wir Version 3 der Sprache verwenden. Der Code sollte sich nahezu wie Pseudocode lesen lassen, hat aber den Vorteil ausführbar zu sein. Alle Funktionen in diesem Abschnitt liefern zu Pattern P und Text T (die als String oder Liste vorliegen sollten) nacheinander alle Paare von Start- und Endpositionen, an denen P in T vorkommt. Dabei ist nach Python-Konvention die Startposition Teil des passenden Bereichs, die Endposition aber nicht mehr.

```
1 def naive(P, T):
2     m, n = len(P), len(T)
3     for i in range(n - m + 1):
4         if T[i:i+m] == P:
5             yield (i, i + m)
```

3.2 Bemerkung (Generatorfunktionen und `yield` in Python)**.** Der Befehl `yield` kann in Python verwendet werden, um aus einer Funktion eine Folge von Werten zurückzugeben. Ein Aufruf von `yield` liefert einen Wert zurück, ohne die Funktion zu beenden. Eine Funktion, die von `yield` Gebrauch macht, nennt man *Generatorfunktion*. Man kann einen Generator in einer *for*-Schleife verwenden und so über alle zurückgelieferten Werte iterieren:

```
1 P, T = "abba", "bababbabbabbab"
2 for (i, j) in naive(P, T):
3     print(i, j, T[i:j])
```

Das obige Code-Fragment gibt zum Beispiel alle Start- und Endpositionen aus, die vom naiven Algorithmus zurückgeliefert werden, und dazu die entsprechende Textstelle, die natürlich `abba` lauten sollte.

Der naive Algorithmus benötigt $\mathcal{O}(mn)$ Zeit (worst-case), da in jeder der $n-m+1$ Iterationen (Fensterpositionen) jeweils bis zu m Zeichen verglichen werden. Für lange Muster und Texte ist diese Laufzeit nicht akzeptabel.

Im Durchschnitt (average-case) ist dieser Algorithmus auf zufälligen Texten gar nicht so schlecht, weil in Zeile 4 im Schnitt sehr schnell ein nicht passendes Zeichen (Mismatch) gefunden wird. Die folgende Analyse macht eine präzise Aussage. Wir gehen davon aus, dass sowohl Text als auch Muster zufällig in folgendem Sinn gewählt sind: An jeder Stelle wird jeder Buchstabe (unabhängig von den anderen Stellen) fair ausgewürfelt; die Wahrscheinlichkeit beträgt also für jeden Buchstaben $1/|\Sigma|$. Wenn zwei zufällige Zeichen verglichen werden, dann beträgt die Wahrscheinlichkeit, dass sie übereinstimmen, $p := |\Sigma|/|\Sigma|^2 = 1/|\Sigma|$.

Vergleichen wir in Zeile 4 ein zufälliges Muster P mit einem Textfenster W der Länge m, dann ist die Wahrscheinlichkeit, dass wir beim j-ten Zeichenvergleich ($j = 1, \ldots, m$) die erste Nichtübereinstimmung feststellen, genau $p^{j-1}(1-p)$. Die Wahrscheinlichkeit, dass alle m Zeichen übereinstimmen, beträgt p^m; in diesem Fall wurden m Vergleiche benötigt. Die erwartete Anzahl an Vergleichen für ein Muster der Länge m ist also

$$E_m := mp^m + \sum_{j=1}^{m} j\, p^{j-1}\,(1-p).$$

Dies ließe sich exakt ausrechnen; wir möchten jedoch eine Schranke für beliebige Musterlänge m erhalten und lassen dazu $m \to \infty$ gehen. Da $E_m < E_{m+1}$ für alle m, ist

$$E_m < E_\infty := \sum_{j=1}^{\infty} j\, p^{j-1}\,(1-p) = (1-p)\sum_{j=0}^{\infty} j\, p^{j-1}.$$

Betrachten wir die Abbildung $p \mapsto \sum_{j=0}^{\infty} j\, p^{j-1}$, stellen wir fest, dass sie die Ableitung von $p \mapsto \sum_{j=0}^{\infty} p^j = 1/(1-p)$ ist, also mit $1/(1-p)^2$ übereinstimmt. Damit ist $E_\infty = (1-p)/(1-p)^2 = 1/(1-p)$.

Aus der Definition $p = 1/|\Sigma|$ folgt nun insgesamt

$$E_m < \frac{|\Sigma|}{|\Sigma| - 1}.$$

Sogar für ein nur 2-buchstabiges Alphabet folgt $E_m < 2$ für alle Musterlängen m. Für $|\Sigma| \to \infty$ (sehr große Alphabete) gilt sogar $E_m \to 1$. Das ist intuitiv verständlich: Bei einem sehr großen Alphabet ist die Wahrscheinlichkeit, dass schon der erste Zeichenvergleich scheitert, sehr groß, und man benötigt fast niemals mehr als diesen einen Vergleich.

3.3 Satz. *Sei $|\Sigma| \geq 2$. Seien ein Muster der Länge m und ein Text der Länge n zufällig gleichverteilt gewählt. Dann beträgt die Worst-case-Laufzeit des naiven Algorithmus $\mathcal{O}(mn)$, aber die erwartete Laufzeit lediglich $\mathcal{O}(nE_m) = \mathcal{O}(n)$, da $E_m < 2$ für alle m.*

Als Übung: Analysiere die erwartete Laufzeit, wenn die Buchstaben des Alphabets mit unterschiedlichen Wahrscheinlichkeiten vorkommen. Sei $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$; die Wahrscheinlichkeit für den Buchstaben σ_i sei $p_i \geq 0$ an jeder Stelle, unabhängig von den anderen Stellen. Natürlich ist $\sum_{i=1}^{k} p_i = 1$.

3.3 NFA-basiertes Pattern Matching

Offensichtlich hat ein (theoretisch) bestmöglicher Algorithmus eine Worst-case-Laufzeit von $\Omega(n+m)$, denn jeder Algorithmus muss mindestens den Text (in $\Omega(n)$) und das Pattern (in $\Omega(m)$) einmal lesen (das Muster könnte ja an jeder Stelle des Textes vorkommen).

Wir wollen einen in diesem Sinne optimalen Algorithmus mit einer Laufzeit von $\Theta(n+m)$ herleiten.

Zunächst wiederholen wir nichtdeterministische endliche Automaten (engl. *non-deterministic finite automaton, NFA*). NFAs können sich in mehreren Zuständen gleichzeitig befinden.

3.4 Definition (NFA)**.** Ein NFA ist ein Tupel $(Q, Q_0, F, \Sigma, \Delta)$, wobei

- Q eine endliche Menge von Zuständen,
- $Q_0 \subset Q$ eine Menge von Startzuständen,
- $F \subset Q$ eine Menge von akzeptierenden Zuständen,
- Σ das Eingabealphabet und
- $\Delta\colon Q \times \Sigma \to 2^Q$ eine nichtdeterministische Übergangsfunktion ist.

Hierbei ist 2^Q eine andere Schreibweise für $\mathcal{Q}$, also die Potenzmenge von Q.

Wir verbinden mit dieser Definition folgende Semantik: Es gibt stets eine Menge aktiver Zustände $A \subset Q$. Am Anfang ist $A = Q_0$. Nach dem Lesen eines Textzeichens $c \in \Sigma$ sind die Zustände aktiv, die von A durch Lesen von c gemäß der Übergangsfunktion Δ erreicht werden können. Der bisher eingelesene String wird akzeptiert, wann immer $A \cap F \neq \emptyset$.

Die Übergangsfunktion $\Delta\colon Q \times \Sigma \to 2^Q$ gibt zu jedem (q, c) eine Menge an Folgezuständen an. Dies kann auch die leere Menge sein. Es ist oft hilfreich, die Übergangsfunktion so zu erweitern, dass wir Mengen von Zuständen übergeben können; d.h. wir erweitern den Definitionsbereich der ersten Komponente von Q auf 2^Q durch

$$\Delta(A, c) := \bigcup_{q \in A} \Delta(q, c)\,.$$

Darüber hinaus ist es nützlich, wenn wir in der zweiten Komponente nicht nur einzelne Zeichen, sondern ganze Strings übergeben können. Wir erweitern den Definitionsbereich also in der zweiten Komponente auf Σ^* durch

$$\Delta(A, \varepsilon) := A$$

und induktiv

$$\Delta(A, xc) := \Delta(\Delta(A, x), c)$$

für $x \in \Sigma^*$ und $c \in \Sigma$. Wir haben nun also eine Funktion $\Delta\colon 2^Q \times \Sigma^* \to 2^Q$ definiert.

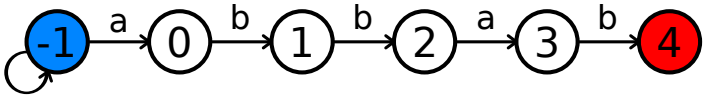


Abbildung 3.2: NFA zum für das Muster `abbab`. Der Startzustand (-1) ist blau hinterlegt; der akzeptierende Zustand ist rot dargestellt.

Epsilon-Transitionen. Eine Erweiterung des NFA-Mechanismus, die nützlich ist, NFAs aber nicht mächtiger macht (sie erkennen nach wie vor genau die regulären Sprachen), besteht darin, sogenannte Epsilon-Transitionen zuzulassen. Das sind Zustandsübergänge ohne das Lesen eines Zeichens. Hierzu definieren wir für jeden Zustand q seinen *ε-Abschluss* E_q; das ist die Menge der Zustände, die von q aus „sofort“ erreicht wird und setzen $\Delta(q, \varepsilon) := E_q$. Für nichtleere Strings wird Δ wie oben induktiv definiert.

Ein NFA für das Pattern-Matching-Problem. Das Pattern-Matching-Problem für das Muster P ist gelöst, wenn wir einen Automaten angeben, der alle Strings der Form $\Sigma^* P$ akzeptiert, also immer genau dann in einem akzeptierenden Zustand ist, wenn zuletzt P gelesen wurde. Ein solcher NFA ist sehr einfach zu konstruieren und besteht aus einer Kette von Zuständen, entlang deren Kanten P buchstabiert ist, sowie einer Schleife im Startzustand, die beim Lesen eines beliebigen Zeichens benutzt wird, so dass der Startzustand nie verlassen wird.

Nummeriert man die Zustände mit -1 (Start), $0, \ldots, |P|-1$, dann ist der NFA (zu gegebenem Eingabealphabet Σ) formal wie folgt definiert (ein Beispiel findet sich in Abbildung 3.2):

- $Q = \{ -1, 0, \ldots, m-1 \}$ mit $m = |P|$
- $Q_0 = \{ -1 \}$
- $F = \{ m-1 \}$
- $\Delta(-1, P[0]) = \{ -1, 0 \}$ und $\Delta(-1, c) = \{ -1 \}$ für alle $c \neq P[0]$;
  für $0 \leq q \leq m-2$ ist $\Delta(q, P[q+1]) = \{ q+1 \}$ und $\Delta(q, c) = \{ \}$ für alle $c \neq P[q+1]$;
  sowie $\Delta(m-1, c) = \{ \}$ für alle $c \in \Sigma$.

Es gilt folgende Invariante.

3.5 Lemma (Invariante der NFA-Zustandsmenge)**.** Sei $A \subset Q$ die aktive Zustandsmenge des NFA. Es ist $q \in A$ genau dann, wenn die letzten $q+1$ gelesenen Zeichen dem Präfix $P[\ldots q]$ entsprechen. Insbesondere ist der Zustand -1 stets aktiv und der Zustand $|P|-1$ genau dann aktiv, wenn die letzten $|P|$ Zeichen mit dem Pattern identisch sind.

Beweis. Die Invariante folgt direkt aus der Konstruktion des Automaten. □

Aus dem Lemma ergibt sich direkt folgender Satz:

3.6 Satz. *Der in diesem Abschnitt konstruierte NFA akzeptiert genau die Sprache $\Sigma^* P$.*

Man kann beim Lesen eines Texts die aktive Zustandsmenge A eines NFA verfolgen und erhält so einen Algorithmus, der aber auch die Laufzeit $\mathcal{O}(mn)$ hat, denn die Menge A hat die Größe $\mathcal{O}(m)$.

3.4 DFA-basiertes Pattern-Matching und der Knuth-Morris-Pratt-Algorithmus

Die explizite Formulierung des Pattern-Matching-Problems als NFA hat einen Nachteil: Mehrere Zustände können gleichzeitig aktiv sein, sodass die Aktualisierung der Zustandsmenge in jedem Schritt $\mathcal{O}(m)$ Zeit kostet. Die Idee dieses Abschnitts ist es, statt des NFA einen DFA zu benutzen, der nur einen aktiven Zustand hat. Wir werden sehen, dass es damit möglich ist, jedes Textzeichen nur einmal zu lesen und dabei nur (amortisiert) konstante Zeit pro Zeichen zu verwenden.

3.4.1 DFA-Konstruktion

Eine einfache Lösung ist folgende: Wir wandeln den NFA in einen äquivalenten deterministischen endlichen Automaten (engl. deterministic finite automaton, DFA) um.

3.7 Definition (DFA)**.** Ein DFA ist ein Tupel $(Q, q_0, \Sigma, F, \delta)$ mit

- endliche Zustandsmenge Q
- Startzustand $q_0 \in Q$
- endliches Alphabet Σ (Elemente: „Buchstaben“)
- akzeptierende Zustände $F \subset Q$
- Übergangsfunktion $\delta : Q \times \Sigma \to Q$

Mit dieser Definition verbinden wir folgende Semantik: Der Automat startet im Zustand q_0 und liest nacheinander Zeichen aus Σ. Dabei ordnet die Übergangsfunktion δ dem Paar (q, c) einen neuen Zustand zu; q ist der alte Zustand und c das gelesene Zeichen. Ist der neue Zustand in F, gibt der Automat das Signal „akzeptiert“.

Hier suchen wir einen Automaten, der immer dann akzeptiert, wenn die zuletzt gelesenen $|P|$ Zeichen mit P übereinstimmen, und daher wie der NFA genau die Strings der Form $\Sigma^* P$ akzeptiert. Wenn man mitzählt, wie viele Textzeichen bereits gelesen wurden, kann man die Textpositionen ausgeben, an denen der Automat akzeptiert; dies entspricht den Endpositionen des Patterns im Text.

Das Transformieren eines NFA in einen DFA kann ganz allgemein mit der *Teilmengenkonstruktion*, machmal auch *Potenzmengenkonstruktion* genannt, geschehen. Dabei kann es theoretisch passieren, dass der äquivalente DFA zu einem NFA mit k Zuständen bis zu 2^k Zustände hat (Zustände des DFA entsprechen Teilmengen der Zustandsmenge des NFA). Wir werden aber gleich sehen, dass sich beim Pattern-Matching-Problem die Zahl der Zustände zwischen NFA und DFA *nicht* unterscheidet. In jedem Fall kann ein DFA jedes gelesene Zeichen in konstanter Zeit verarbeiten, sofern die Übergangsfunktion δ, die jeder Kombination aus aktuellem Zustand und gelesenem Zeichen einen eindeutigen Nachfolgezustand zuordnet, vorberechnet ist und als Tabelle vorliegt. Wir werden jedoch auch sehen, dass man ohne wesentlichen Zeitverlust nicht die ganze δ-Funktion vorberechnen muss (immerhin $|\Sigma| \cdot |Q|$ Werte), sondern sie bereits mit $m = |P|$ Werten (wobei $|Q| = |P| + 1$) kompakt darstellen kann.

Warum nun hat der DFA genau so viele Zustände wie der NFA und nicht mehr? Das folgt aus folgender für diesen Abschnitt zentraler Beobachtung.

3.8 Lemma. Sei A die aktive Zustandsmenge des Pattern-Matching-NFA. Sei $a^* := \max A$. Dann ist A durch a^* eindeutig bestimmt. Der äquivalente DFA hat genauso viele Zustände wie der NFA.

Beweis. Der Wert von a^* bestimmt die letzten $a^* + 1$ gelesenen Zeichen des Textes; diese sind gleich dem Präfix $P[\dots a^*]$. Ein Zustand $q < a^*$ ist genau dann aktiv, wenn die letzten $q+1$ gelesenen Zeichen ebenso gleich dem Präfix $P[\dots q]$ sind, also wenn das Suffix der Länge $q+1$ des Präfix $P[\dots a^*]$ (das ist $P[a^* - q \dots a^*]$) gleich dem Präfix $P[\dots q]$ ist.

Da es also zu jedem a^* nur eine mögliche Zustandsmenge A mit $a^* = \max A$ gibt, hat der DFA auf jeden Fall nicht mehr Zustände als der NFA. Da aber auch jeder NFA-Zustand vom Startzustand aus erreichbar ist, hat der DFA auch nicht weniger Zustände als der NFA. □

3.9 Beispiel (NFA-Zustandsmengen)**.** Für `abbab` gibt es im NFA die folgenden möglichen aktiven Zustandsmengen, und keine weiteren:

$a^* = -1$: $\{-1\}$
$a^* = 0$: $\{-1, 0\}$
$a^* = 1$: $\{-1, 1\}$
$a^* = 2$: $\{-1, 2\}$
$a^* = 3$: $\{-1, 0, 3\}$
$a^* = 4$: $\{-1, 1, 4\}$

♡

Statt die DFA-Zustände durch die Zustandsmengen des NFA zu benennen, benennen wir sie nur anhand des enthaltenen maximalen Elements a^*. Es ist klar, dass -1 der Startzustand und $m-1$ der einzige akzeptierende Zustand ist. Aufgrund der Eindeutigkeit der zugehörigen Menge A können wir zu jedem Zustand und Zeichen den Folgezustand berechnen, also eine Tabelle δ erstellen, die die DFA-Übergangsfunktion repräsentiert.

Formal ergibt sich der DFA wie folgt:

- $Q = \{-1, 0, \dots, m-1\}$ ($m+1$ Zustände)
- $q_0 = -1$
- Σ ist das Alphabet des Textes und Patterns
- $F = \{m-1\}$
- Übergangsfunktion $\delta : Q \times \Sigma \to Q$ wie folgt: Zu $q \in Q$ und $c \in \Sigma$ berechne die zugehörige eindeutige NFA-Zustandsmenge $A(q)$ mit $q = \max A(q)$. Wende hierauf die NFA-Übergangsfunktion für c an und extrahiere das maximale Element als neuen Zustand, berechne also $\max \Delta(A(q), c)$.

Zur Illustration berechnen wir in Beispiel 3.9 den Nachfolgezustand zu $q = 3$ nach Lesen von `a`. Der entsprechende NFA-Zustand ist $\{-1, 0, 3\}$; durch Lesen von `a` gelangt man von -1 nach $\{-1, 0\}$, von 0 nach $\{\ \}$, und von 3 nach $\{\ \}$. Die Vereinigung dieser Mengen ist $\{-1, 0\}$ und entspricht dem DFA-Zustand 0. So verfährt man mit allen Zuständen und Zeichen. Ein Beispiel ist in Abbildung 3.3 zu sehen. Die Berechnung funktioniert in jedem Fall in $\mathcal{O}(m^2|\Sigma|)$ Zeit, aber es gibt eine bessere Lösung, zu der wir in Abschnitt 3.4.2 kommen. ⇑ 14.04.11

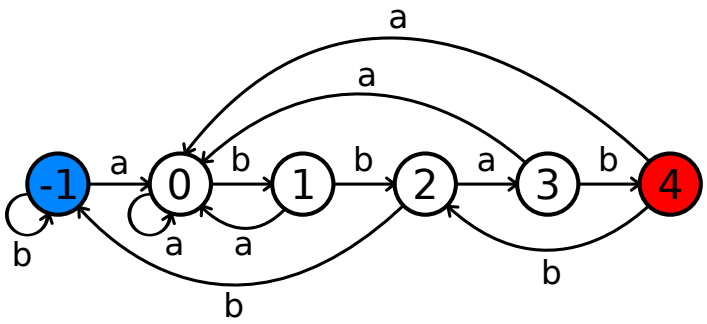


Abbildung 3.3: Deterministischer endlicher Automat (DFA) für die Suche nach dem Pattern `abbab`. Dabei ist der Startzustand in blau und der einzige akzeptierende Zustand in rot eingezeichnet.

Der folgende Code realisiert das DFA-basierte Pattern-Matching, sofern die Funktion `delta` die korrekte Übergangsfunktion δ implementiert. (Man beachte, dass es in Python unproblematisch ist, Funktionen an andere Funktionen zu übergeben.)

```
1  def DFA_with_delta(m, delta, T):
2      q = -1
3      for i in range(len(T)):
4          q = delta(q, T[i])
5          if q == m - 1:
6              yield (i-m+1, i+1)
7
8  def DFA(P, T):
9      delta = DFA_delta_table(P)
10     return DFA_with_delta(len(P), delta, T)
```

Hier gehen wir davon aus, dass es eine Funktion `DFA_delta_table` gibt, die `delta` korrekt aus dem Pattern vorberechnet. Wie diese effizient aussieht, sehen wir gleich.

3.4.2 Der Knuth-Morris-Pratt-Algorithmus

⇓ 21.04.11

Wir kommen jetzt zu einer „platzsparenden“ Repräsentation der Übergangsfunktion δ, die darüber hinaus noch in Linearzeit, also $\mathcal{O}(m)$, zu berechnen ist (**?**).

Die lps-Funktion. Die Grundidee ist einfach: Wenn im DFA-Zustand $q < m-1$ das „richtige“ Zeichen $P[q+1]$ gelesen wird, gelangt man zum Zustand $q+1$, kommt also „weiter“ im Pattern. Dies entspricht dem Fall, dass der maximale Zustand in der NFA-Zustandsmenge $A(q)$ erhöht wird und sich die Menge dementsprechend ändert. Wenn aber das falsche Zeichen gelesen wird, müssen die anderen Zustände in $A(q)$ daraufhin untersucht werden, ob diese durch das gelesene Zeichen verlängert werden können. Benötigt wird also eine Möglichkeit, von q auf alle Werte in $A(q)$ zu schließen.

Wir erinnern an Lemma 3.5: Es ist $a \in A(q)$ genau dann, wenn die letzten $a+1$ gelesenen Zeichen, die ja gleich $P[q-a \ldots q]$ sind, da wir uns im Zustand q befinden, dem Präfix $P[\ldots a]$ entsprechen.

Im Wesentlichen stehen wir also vor der Frage: Welche Präfixe von P sind gleich einem echten Suffix von $P[\dots q]$? Um *alle* diese Präfixe zu bekommen, genügt es aber, das *längste* zu speichern. Kürzere kann man dann durch iteriertes Verkürzen erhalten (**TODO: mehr Detail**). Daher definieren wir zu jeder Endposition q in P eine entsprechende Größe.

3.10 Definition (lps-Funktion)**.** Zu $P \in \Sigma^m$ definieren wir $lps : \{0, \dots, m-1\} \to \mathbb{N}$ folgendermaßen:

$$lps(q) := \max\{\, |s| < q+1 \,:\, s \text{ ist Präfix von } P \text{ und Suffix von } P[\dots q] \,\}.$$

Mit anderen Worten ist $lps(q)$ die Länge des längsten Präfix von P, das ein echtes Suffix von $P[0 \dots q]$ (oder leer) ist. Man beachte, dass $lps(-1)$ nicht definiert ist und auch nicht benötigt wird. Die lps-Funktion ist die zentrale Definition des KMP-Algorithmus.

3.11 Beispiel (lps-Funktion)**.**

`q`	0	1	2	3	4	5	6
`P[q]`	a	b	a	b	a	c	a
`lps[q]`	0	0	1	2	3	0	1

In der obersten Zeile steht der Index der Position, darunter das Pattern P und darunter der Wert von lps an dieser Stelle. ♡

Welcher Bezug besteht nun genau zwischen der NFA-Zustandsmenge $A(q)$ und $lps(q)$?

3.12 Lemma. Es ist $A(q) = \{\, q, lps(q)-1, lps(lps(q)-1)-1, \dots, -1 \,\}$; d.h. die aktiven NFA-Zustände sind q und alle Zustände, die sich durch iteriertes Anwenden von lps und Subtraktion von 1 ergeben, bis schließlich der Startzustand -1 erreicht ist.

Beweis. Zustand q ist nach Definition von $A(q)$ der größte Zustand in $A(q)$. Aus Lemma 3.5 folgt, dass $a \in A(q)$ genau dann gilt, wenn $P[q-a \dots q] = P[\dots a]$, also das Präfix der Länge $a+1$ von P gleich dem Suffix der Länge $a+1$ von $P[\dots q]$ ist. Das größte solche $a < q$ ist also die Länge des längsten solchen Präfix, $lps(q)$, minus 1. Der resultierende Zustand a ist entweder $a = -1$; dann gab es kein passendes Präfix und folglich keinen weiteren aktiven NFA-Zustand. Oder es ist $a \geq 0$; dann gibt es erstens keinen weiteren aktiven NFA-Zustand zwischen a und q (sonst hätten wir ein längeres Präfix gefunden); zweitens können wir das Lemma jetzt auf a anwenden und so insgesamt induktiv beweisen. □

Simulation der DFA-Übergangsfunktion mit lps. Mit Hilfe der lps-Funktion bekommt man also die gesamte Menge $A(q)$ für jedes q. Somit muss man die DFA-Übergangsfunktion δ nicht vorberechnen, sondern kann in jedem Schritt den benötigten Wert „on-the-fly“ mit Hilfe der lps-Funktion bestimmen. Solange das gelesene Zeichen c nicht das nächste des Patterns ist (insbesondere gibt es kein nächstes wenn wir am Ende des Patterns stehen, $q = m-1$) und wir nicht im Startzustand $q = -1$ angekommen sind, reduzieren wir q auf das nächstkürzere passende Präfix. Zuletzt prüfen wir, ob das Zeichen jetzt zum Pattern passt (das muss nicht der Fall sein, wenn wir in $q = -1$ gelandet sind) und erhöhen den Zustand gegebenenfalls. Die Funktion `delta` kann man, wenn die lps-Funktion bereits berechnet wurde, wie folgt implementieren.

```
1 def DFA_delta_lps(q, c, P, lps):
2     """for pattern P, return the
3     next state from q after reading c, computed with lps"""
4     m = len(P)
5     while q == m-1 or (P[q+1] != c and q > -1):
6         q = lps[q] - 1
7     if P[q+1] == c:  q += 1
8     return q
```

3.13 Bemerkung (Partielle Funktionsauswertung)**.** Um aus `DFA_delta_lps` eine Funktion `delta` zu erhalten, der man kein Pattern `P` und kein `lps`-Array mehr übergeben muss, kann man *partielle Funktionsauswertung* benutzen. Python bietet dazu im Modul `functools` die Funktion `partial` an. Wir nehmen an, es gibt eine weitere Funktion `compute_lps`, die zu einem Pattern P die zugehörige lps-Funktion berechnet. Dann erhalten wir die Übergangsfunktion wie folgt:

```
1 import functools
2 delta = functools.partial(DFA_delta_lps, P=P, lps=compute_lps(P))
```

Die so erhaltene `delta`-Funktion kann man an obige `DFA`-Funktion übergeben.

Insgesamt sieht der KMP-Algorithmus damit so aus:

```
1 def KMP(P, T):
2     lps = KMP_compute_lps(P)
3     delta = functools.partial(DFA_delta_lps, P=P, lps=lps)
4     return DFA_with_delta(len(P),delta,T)
```

In der Originalarbeit von **?** ist der Algorithmus so angegeben, dass der Code für `DFA_delta_lps` und `DFA_with_delta` miteinander verschränkt ist. Unsere Darstellung macht aber klar, dass die lps-Funktion nur eine kompakte Darstellung der Übergangsfunktion des DFA ist.

Laufzeitanalyse.

3.14 Lemma. Die Laufzeit des Knuth-Morris-Pratt-Algorithmus auf einem Text der Länge n ist $\mathcal{O}(n)$, wenn die lps-Funktion des Patterns bereits vorliegt.

Beweis. Es ist klar, dass ein Aufruf von `DFA_delta_lps` $\mathcal{O}(m)$ Zeit kosten kann und insgesamt diese Funktion von `DFA` $\mathcal{O}(n)$-mal aufgerufen wird. Dies würde eine Laufzeit von $\mathcal{O}(mn)$ ergeben, also nicht besser als der naive Algorithmus. Diese Analyse ist aber zu ungenau. Obwohl einzelne Aufrufe von `delta_lps` maximal m Iterationen der while-Schleife durchführen können, ist die *Gesamtzahl* der while-Durchläufe beschränkt. Wir analysieren daher *amortisiert*. Dazu bemerken wir, dass bei jedem Durchlauf von Zeile 6 in `DFA_delta_lps` der Wert von q echt kleiner wird (um mindestens 1). Da q aber nicht unter -1 fallen kann und auch insgesamt höchstens n-mal erhöht wird (Zeile 7), kann Zeile 6 insgesamt auch höchstens n-mal aufgerufen werden. Die Bedingung der umhüllenden while-Schleife kann höchstens doppelt so oft getestet werden. Insgesamt ist die Anzahl der while-Tests also durch $2n$ beschränkt; dies ist in $\mathcal{O}(n)$. □

Wir zeigen später noch, dass sich die lps-Funktion in $\mathcal{O}(m)$ Zeit berechnen lässt, so dass wir insgesamt den folgenden Satz bewiesen haben.

3.15 Satz. *Der Knuth-Morris-Pratt-Algorithmus findet alle Vorkommen eines Musters $P \in \Sigma^m$ in einem Text $T \in \Sigma^n$ in $\mathcal{O}(m+n)$ Zeit.*

Das ist ein befriedigendes Ergebnis; es gibt allerdings einen kleinen Nachteil: Obwohl insgesamt nur $\mathcal{O}(n)$ Zeit zum Durchlaufen des Textes benötigt wird, können einzelne Iterationen bis zu m Schritte benötigen. Liegt der Text nur als Datenstrom vor, so dass jedes Zeichen unter Realzeitbedingungen in einer bestimmten Zeit bearbeitet werden muss, ist der KMP-Algorithmus also nicht geeignet.

Tabellieren der Übergangsfunktion. Für Realzeitanwendungen ist es besser, die Übergangsfunktion δ vorzuberechnen und als Tabelle abzuspeichern. Dann kann jedes Zeichen in konstanter Zeit verarbeitet werden. Wir zeigen dies hier mit Hilfe eines Dictionaries, das allerdings für Realzeitanwendungen wiederum weniger geeignet ist.

Uns kommt es darauf an, dass man mit Hilfe der lps-Funktion die gesamte δ-Tabelle in optimaler Zeit $\mathcal{O}(m \cdot |\Sigma|)$ erstellen kann, wenn man bei der Berechnung von $\delta(q, \cdot)$ ausnutzt, dass $\delta(q', \cdot)$ für alle $q' < q$ bereits berechnet ist.

Der Folgezustand von $q = -1$ ist 0, wenn das richtige Zeichen $P[0]$ gelesen wird, sonst -1. Der Folgezustand von $0 < q < m-1$ ist $q+1$, wenn das richtige Zeichen $P[0]$ gelesen wird, und ansonsten der entsprechende Folgezustand des Zustands $lps[q]-1$, der schon berechnet worden ist. Der Folgezustand von $q = m-1$ ist immer der entsprechende Folgezustand von $lps[m-1]-1$.

Das folgende Codefragment realisiert diese Regeln. Die return-Zeile verpackt die delta-Tabelle in eine Funktion, da `DFA_with_delta` die Übergabe einer Funktion erwartet.

```
1  def DFA_delta_table(P):
2      alphabet, m = set(P), len(P)
3      delta = dict()
4      lps = KMP_compute_lps(P)
5      for c in alphabet:      delta[(-1,c)] = 0 if c == P[0] else -1
6      for q in range(m):
7          for c in alphabet: delta[(q, c)]       = delta[(lps[q]-1, c)]
8          if q < m-1:        delta[(q, P[q+1])] = q + 1
9      # wrap delta into a function that returns -1 if (q,c) not in dict:
10     return lambda *args: delta.get(args, -1)
```

Die Tabelle δ benötigt Platz $\mathcal{O}(|\Sigma| \cdot m)$, und die Funktion `DFA_delta_table` berechnet diese in optimaler Zeit $\mathcal{O}(|\Sigma| \cdot m)$. Dies ist eine Verschlechterung gegenüber KMP: lps benötigt nur $O(m)$ Platz und Zeit zur Berechnung. Die Verbesserung liegt darin, dass jeder Schritt in konstanter Zeit ausgeführt werden kann und der Aufwand pro Schritt nicht wie bei KMP schwankt.

Zusammenfassung und Berechnung der lps-Funktion. Wir zeigen hier noch einmal den KMP-Algorithmus in der ursprünglichen Form (?), aus der der Bezug zum DFA mit seiner `delta`-Funktion nicht so klar hervorgeht.

```
def KMP_classic(P, T):
    q, m, n, lps  =  -1, len(P), len(T), KMP_compute_lps(P)
    for i in range(n):
        while q == m-1 or (P[q+1] != T[i] and q > -1):
            q = lps[q] - 1
        if P[q+1] == T[i]:  q += 1
        # Invariante (I) trifft an dieser Stelle zu.
        if q == m-1: yield (i+1-m, i+1)
```

Wie man sieht, ist dies nur eine Refaktorisierung von `KMP` wie oben angegeben. Wir vergegenwärtigen uns aber hieran noch einmal, welche Invariante (I) für den Zustand q am Ende von Schleifendurchlauf i gilt:

$$\text{(I):}\quad q = \max\{\, k : T[i-k \ldots i] = P[0 \ldots k] \,\}\,.$$

Also ist $q+1$ die Länge des längsten Suffixes des bisher gelesenen Textes, das ein Präfix von P ist. Solange ein Match nicht mehr verlängert werden kann, entweder weil $q = m-1$ oder weil der nächste zu lesende Buchstabe des Textes nicht zu P passt ($T[i] \neq P[q+1]$), verringert der Algorithmus q soviel wie nötig, aber so wenig wie möglich. Dies geschieht in der inneren while-Schleife mit Hilfe der *lps*-Tabelle; und zwar solange, bis ein Suffix des Textes gefunden ist, welches mit einem Präfix des Patterns kompatibel ist, oder bis $q = -1$ wird.

Noch offen ist die Berechnung der lps-Funktion, die als Vorverarbeitung durchgeführt werden muss. Interessanterweise können wir die lps-Tabelle mit einer Variante des KMP-Algorithmus berechnen. Für die Berechnung eines Wertes in der Tabelle wird immer nur der schon berechnete Teil der Tabelle benötigt:

```
def KMP_compute_lps(P):
    m, q = len(P), -1
    lps = [0] * m  # mit Nullen initialisieren, lps[0] = 0 ist korrekt
    for i in range(1, m):
        while q > -1 and P[q+1] != P[i]:
            q = lps[q] - 1
        if P[q+1] == P[i]:  q += 1
        # Invariante (J) trifft an dieser Stelle zu.
        lps[i] = q+1
    return lps
```

Die Invariante (J) lautet:

$$\text{(J):}\quad q = \max\{\, k < i : P[i-k \ldots i] = P[0 \ldots k] \,\}\,,$$

also ist $q+1$ die Länge des längsten Präfix von P, das auch ein (echtes oder leeres) Suffix von $P[i \ldots]$ ist. Der Beweis erfolgt mit den gleichen Argumenten wie für den KMP-Algorithmus mit Induktion. Also ist die Laufzeit $\mathcal{O}(m)$.

3.5 Shift-And-Algorithmus: Bitparallele Simulation von NFAs

Gegenüber dem DFA hat ein NFA bei der Mustersuche den Vorteil, dass er wesentlich einfacher aufgebaut ist (vergleiche Abbildung 3.2 mit Abbildung 3.3). Um die Mustersuche mit einem NFA zu implementieren, müssen wir die aktive Menge A verwalten. Diese Menge hat die Größe $\mathcal{O}(m)$. Der resultierende Algorithmus hat folglich eine Laufzeit von $\mathcal{O}(mn)$. Daher haben wir einigen Aufwand betrieben, um den Automaten zu determinisieren. In der Praxis kann sich das Simulieren eines NFAs jedoch auszahlen, wenn wir die Menge der aktiven Zustände als Bitvektor codieren. Dann kann man ausnutzen, dass sich Operationen auf vielen Bits (32, 64, oder auch 256, 1024 auf spezieller Hardware wie FPGAs) parallel durchführen lassen. Asymptotisch (und theoretisch) ändert sich an den Laufzeiten der Algorithmen nichts: $\Theta(mn/64)$ ist immer noch $\Theta(mn)$, aber in der Praxis macht eine um den Faktor 64 kleinere Konstante einen großen Unterschied.

Wir betrachten zunächst den Fall eines einfachen Patterns P und gehen später darauf ein, wie wir das auf eine endliche Menge von Patterns verallgemeinern können. Wir müssen also einen linearen NFA (vgl. Abbildung 3.2 auf Seite 17) simulieren. Die aktiven Zustände verschieben sich um eine Position nach rechts, und zwar genau dann, wenn das eingelesene Zeichen zur Kantenbeschriftung passt. Ansonsten „verschwindet“ der aktive Zustand.

Dieses Verhalten können wir mit den Bitoperationen *shift* und *and* simulieren. Der resultierende Algorithmus heißt deshalb auch *Shift-and-Algorithmus*. Wir nehmen an, dass das Pattern höchstens so viele Zeichen wie ein Register Bits hat (dies trifft in der Praxis oft zu). Für lange Patterns ist diese Methode nicht empfehlenswert, weil das Pattern dann auf mehrere Register aufgeteilt werden muss und man sich manuell um Carry-Bits kümmern muss.

Der Startzustand ist immer aktiv; wir müssen ihn also nicht explizit simulieren. Daher vereinbaren wir folgende Zustandsbenennung: Zustand $0 \leq q < |P|$ ist aktiv, wenn $P[0 \dots q]$ gelesen wurde. Der Startzustand bekommt keine Nummer bzw. die -1. Dies entspricht der bekannten Benennung aus Abschnitt 3.3.

Der Integer A mit $|P|$ Bits repräsentiert die Zustandsmenge. Bit q von A repräsentiert die Aktivität von Zustand q. Da der Startzustand in A nicht enthalten ist und alle anderen Zustände zu Beginn nicht aktiv sind, initialisieren wir A mit dem Wert $0 = (0, \dots, 0)_2$. Nun läuft das Muster im Automaten „von links nach rechts“, während man die Bits in einem Integer gewöhnlich „von rechts nach links“ hochzählt, Bit 0 also ganz rechts hinschreibt. Daher wird im Folgenden die Verschiebung der aktiven Zustände nach rechts durch eine Bit-Verschiebung nach links ausgedrückt; davon sollte man sich nicht verwirren lassen!

Zum Lesen eines Zeichens c führen wir einen Bit-Shift nach links durch, wobei wir ein neues 1-Bit von rechts aus dem (nicht repräsentierten) Startzustand explizit hinzufügen müssen; dies geschieht durch Veroderung mit $1 = (0, \dots, 0, 1)_2$. Dann streichen wir alle geshifteten Bits, die nicht dem Lesen von c entsprechen durch Ver*und*en mit einer Maske $mask^c$ für $c \in \Sigma$; dabei ist $mask^c$ folgendermaßen definiert: $mask^c_i = 1$ wenn $P[i] = c$ und $mask^c_i = 0$ sonst. Um zu entscheiden, ob das Pattern an der aktuellen Position matcht, prüfen wir, ob Zustand $m - 1$ aktiv ist.

Eine Python-Implementierung kann wie in Abbildung 3.4 aussehen. Der Funktionsaufruf `ShiftAnd_single_masks(P)` berechnet ein Tripel, bestehend aus (1) einer Funktion, die

```
def ShiftAnd_single_masks(P):
    """for a single pattern P, returns (mask,ones,accepts), where
    mask is a function such that mask(c) returns the bit-mask for c,
    ones is the bit-mask of states after start states, and
    accepts is the bit-mask for accept states."""
    mask, bit = dict(), 1
    for c in P:
        if c not in mask: mask[c] = 0
        mask[c] |= bit
        bit *= 2
    return (dict2function(mask, 0), 1, bit // 2)

def ShiftAnd_with_masks(T, masks, ones, accept):
    """yields each (i,b) s.t. i is the last text position of a match,
    b is the bit pattern of active accepting states at position i."""
    A = 0  # bit-mask of active states
    for (i,c) in enumerate(T):
        A = ((A << 1) | ones) & masks(c)
        found = A & accept
        if found !=0:  yield (i, found)

def ShiftAnd(P, T):
    m = len(P)
    (mask, ones, accept) = ShiftAnd_single_masks(P)
    return ((i-m+1, i+1)
            for (i,_) in ShiftAnd_with_masks(T, mask, ones, accept)
    )
```

Abbildung 3.4: Python-Implementierung des Shift-And-Algorithmus.

Masken zurückliefert, (2) einer Bitmaske mit Einsen aus Startzuständen zur Veroderung, (3) einer Bitmaske mit Einsen zum Test auf akzeptierende Zustände. Die Masken werden als `dict` verwaltet, das in eine Funktion verpackt wird, die bei einem Zeichen, das nicht im Pattern vorkommt, korrekterweise die Bitmaske 0 zurück liefert.

3.16 Bemerkung. Zur Implementierung: Man kann in Python mit wenig Aufwand ein Wörterbuch (Hashtabelle; `dict`) in eine Funktion „verpacken“, so dass die Funktion den Wert aus dem Wörterbuch liefert, wenn der Schüssel darin vorkommt, und andernfalls einen Default-Wert. Mit dem Aufruf `dict2function(d,default)` wird das Wörterbuch `d` verpackt. Diese Funktion ist einfach zu implementieren; man lese dazu auch die Dokumentation der `get`-Methode eines Wörterbuchs.

```
def dict2function(d, default=None):
    return lambda x: d.get(x, default)
```

Weiter ist `ShiftAnd_with_masks(T, masks, ones, accept)` eine Generatorfunktion, die für jede Textposition i, an der ein Treffer endet, das Paar aus i und der Bitmaske aktiven akzeptierenden Zustände liefert. Man beachte, dass aber nicht die Startposition des Matches zurückgeliefert wird. Das hat vor allem Bequemlichkeitsgründe: Das Pattern und seine Länge werden der Funktion gar nicht explizit übergeben. Die aufrufende Funktion `ShiftAnd(P, T)`

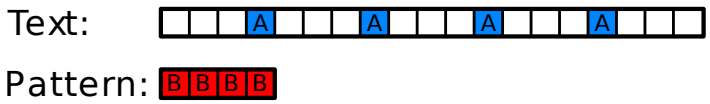


Abbildung 3.5: Einfaches Beispiel, das belegt, dass es im besten Fall genügt, jedes m-te Zeichen anzuschauen um festzustellen, dass ein Pattern der Länge m nicht vorkommt.

kennt aber die Patternlänge und liefert die korrekte Startposition und Endposition aller Matches wie KMP.

3.17 Bemerkung. Zu `enumerate(T)`: Dies ist ein verbreitetes Idiom, wenn man gleichzeitig beim Iterieren den Index und das Element benötigt. In anderen Programmiersprachen iteriert man mit Hilfe des Indexes: `for i in range(len(T))`, und setzt zuerst in jedem Schleifendurchlauf das Element `c=T[i]`. Mit der Konstruktion `for (i,c) in enumerate(T)` wird das auf elegante Weise gelöst.

3.18 Bemerkung. Eine Python-Implementierung ist eigentlich nicht sehr sinnvoll, da in Python ganze Zahlen als Objekte verwaltet werden und die Bit-Operationen nicht direkt auf der Hardwareebene angewendet werden. Shift-And und Shift-Or (s.u.) sollte man eigentlich in C programmieren, damit man die Vorteile voll ausnutzen kann.

Laufzeit. Die Laufzeit ist $\mathcal{O}(mn/w)$, dabei ist w die Registerlänge (word size). Wenn P nicht in ein Register passt, muss man das Übertragsbit beim Shift beachten. Wenn wir annehmen, dass $m \leq w$ bzw. $m/w \in O(1)$ gilt, erhalten wir eine Laufzeit von $\mathcal{O}(n)$. Wenn diese Annahme erfüllt ist, erhalten wir einen Algorithmus, der in der Praxis sehr schnell ist. Allgemein gilt, dass bitparallele Algorithmen solange effizient sind, wie die aktiven Zustände in ein Registerwort passen. Vor allem können sie also bei kurzen Mustern eingesetzt werden. Ihr Vorteil liegt in ihrer großen Flexibilität, die wir später noch schätzen lernen werden: Grundsätzlich ist es immer einfacher, einen nichtdeterministischen endlichen Automaten aufzustellen und bitparallel zu simulieren als einen äquivalenten deterministischen Automaten zu konstruieren.

Shift-Or. Im Falle eines einzelnen Strings kann man sich die Veroderung mit 1 (`starts`) in jedem Schritt sparen, wenn man die Bitlogik umkehrt (0 statt 1). Beim shift-left kommt sowieso eine Null von rechts. Entsprechend muss man auch die Logik der Masken und des Tests auf Akzeptanz umkehren. Dies liefert den *Shift-Or-Algorithmus*. Bei mehreren Strings ist dies nicht sinnvoll, da es mehrere „Start“-Zustände gibt.

3.6 Die Algorithmen von Horspool und Sunday

Im KMP-Algorithmus und im Shift-And-Algorithmus wird in jeder Iteration genau ein Zeichen des Textes verarbeitet. Wir wollen nun die Frage stellen, ob dies wirklich nötig ist. Wie viele Zeichen eines Textes der Länge n müssen mindestens angeschaut werden, um kein Vorkommen des gesuchten Patterns der Länge m zu übersehen? Wenn wir weniger als n/m

Zeichen betrachten, gibt es im Text irgendwo einen Block der Länge m, in dem wir kein Zeichen angeschaut haben. Damit können wir nicht festgestellt haben ob sich dort ein Vorkommen befindet. Es ist jedoch unter Umständen tatsächlich möglich, mit $\mathcal{O}(n/m)$ Schritten auszukommen. Wenn z.B. das Muster ausschließlich aus dem Buchstaben `B` besteht und der Text an jeder m-ten Stelle den Buchstaben `A` enthält (wie in Abbildung 3.5 gezeigt), dann kann man durch Anschauen jedes m-ten Zeichens feststellen, dass nirgendwo ein Match vorhanden sein kann.

Idee. Zu jedem Zeitpunkt gibt es ein aktuelles Suchfenster der Länge m. Dies entspricht dem Teilstring des Textes, der gerade mit dem Muster verglichen wird. Wir betrachten zuerst das *letzte* (rechteste) Zeichen des Suchfensters im Text. Algorithmen, die so vorgehen, sind zum Beispiel:

- Boyer-Moore-Algorithmus (**?**, klassisch, aber meist langsamer als die folgenden Algorithmen, deshalb überspringen wir ihn hier, siehe **?**, Abschnitt 2.2),
- Horspool-Algorithmus (**?**, sehr einfache Variante des Boyer-Moore Algorithmus),
- Sunday-Algorithmus (**?**).

Typischerweise haben diese Algorithmen eine best-case-Laufzeit von $\mathcal{O}(n/m)$ und eine worst-case-Laufzeit von $\mathcal{O}(m \cdot n)$. Durch Kombination mit Ideen von $\mathcal{O}(n+m)$ Algorithmen, z.B. dem KMP-Algorithmus, lässt sich eine worst-case-Laufzeit von $\mathcal{O}(n+m)$ erreichen. Das ist jedoch vor allem theoretisch interessant. Der Boyer-Moore-Algorithmus erreicht so in der Tat eine Laufzeit von $\mathcal{O}(n+m)$, allerdings mit relativ kompliziertem Code und daher großen Proportionalitätskonstanten in der $\mathcal{O}$-Notation.

Insbesondere bei großen Alphabeten lohnt sich der Einsatz des hier vorgestellten Horspool-Algorithmus, da bei großen Alphabeten die Chance groß ist, einen Mismatch zu finden, der uns erlaubt, viele Zeichen zu überspringen.

Ablauf des Horspool-Algorithmus. Wir betrachten das *letzte* (rechteste) Zeichen des Suchfensters im Text, sagen wir $a \in \Sigma$.

TEST-PHASE: Wir prüfen zuerst, ob a mit dem letzten Zeichen von P übereinstimmt. Wenn nicht, geht es weiter mit der SHIFT-PHASE. Wenn ja, prüfen wir das ganze Fenster auf Übereinstimmung mit P, bis wir entweder ein nicht passendes Zeichen finden oder eine exakte Übereinstimmung verifiziert haben. Dieser Test kann von rechts nach links oder links nach rechts erfolgen; häufig kann auf Maschinenebene eine `memcmp`-Instruktion genutzt werden.

SHIFT-PHASE: Unabhängig vom Ausgang der TEST-PHASE verschieben wir das Fenster. Sei $\ell[a]$ die Position des rechtesten a in P ohne das letzte Zeichen, sofern eine solche Position existiert. Andernfalls sei $\ell[a] := -1$. Also $\ell[a] := \max\{0 \leq j < m-1 : P[j] = a\}$, wobei hier das Maximum über die leere Menge gleich -1 gesetzt wird. Dann verschieben wir das Fenster um $\mathit{shift}[a] := m - 1 - \ell[a]$ Positionen (siehe Abbildung 3.6).

Damit können wir keinen Match verpassen, denn kürzere Shifts führen nach Konstruktion immer dazu, dass das bereits gelesene a in P nicht passt.

Die Werte $\mathit{shift}[a]$ werden für jedes Zeichen a, das in P vorkommt, vorberechnet; für alle anderen Zeichen ist $\mathit{shift}[a] = m$.

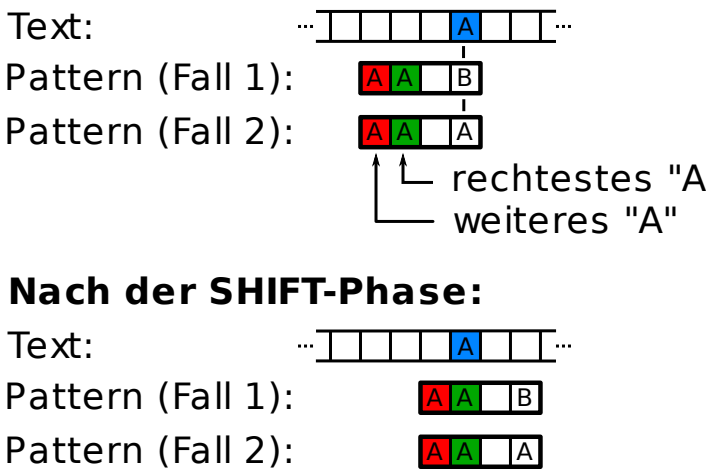


Abbildung 3.6: Illustration des Horspool-Algorithmus. Zunächst wird das Zeichen ganz rechts im aktuellen Fenster mit dem rechtesten Zeichen des Patterns verglichen. Das Fenster wird in der SHIFT-Phase soweit verschoben, dass das rechteste Vorkommen dieses Buchstabens im Pattern (außer dem letzten Zeichen) auf dieser Stelle zu liegen kommt. Dabei ist es unerheblich, ob wir vorher einen Match (Fall 1) oder einen Mismatch (Fall 2) beobachtet haben.

In Abbildung 3.7 ist der Horspool-Algorithmus implementiert. Die Implementierung ist so gestaltet, dass die SHIFT-Phase nicht verlassen wird, bis das letzte Zeichen passt (oder der Text zu Ende ist). In der Test-Phase werden nur noch die ersten $m-1$ Zeichen des Fensters verglichen; dieser Vergleich wird nicht im Detail spezifiziert (Zeile 17) und kann die Zeichen in beliebiger Reihenfolge testen. Der hier gezeigte `==`-Test auf Teilstrings ist in Python ineffizient und nur konzeptionell zu verstehen.

⇑ 21.04.11

⇓ 28.04.11

Laufzeit-Analyse. Die Best-case-Laufzeit ist $\Theta(m+n/m)$: Im besten Fall vergleicht man immer ein Zeichen, das im Pattern nicht vorkommt und kann um m Positionen verschieben. Die Vorberechnung der Shift-Funktion kostet offensichtlich $\mathcal{O}(m)$ Zeit. Die Worst-case-Laufzeit ist $O(m+mn)$: Die while-Schleife wird $\mathcal{O}(n)$-mal durchlaufen; jeder Test in Zeile 17 dauert im schlimmsten Fall $\mathcal{O}(m)$.

Interessant ist die Average-case-Laufzeit. Eine exakte probabilistische Analyse ist nicht ganz einfach. Eine einfache Abschätzung ist folgende: Wir untersuchen in jedem Fenster den Erwartungswert der Anzahl der Zeichenvergleiche und den Erwartungswert der Shiftlänge.

Die erwartete Anzahl der Zeichenvergleiche in einem zufälligen Textfenster hatten wir (bei der Analyse des naiven Algorithmus) als < 2 erkannt, solange das Alphabet aus mindestens zwei Buchstaben besteht. Da aber durch die letzte Verschiebung mindestens ein Buchstabe passt (ob der beim Fenstervergleich erreicht wird, ist unbekannt), ist das Fenster nicht mehr ganz zufällig. Trotzdem können wir die erwartete Anzahl der Vergleiche durch $2+1=3$ abschätzen, und der Erwartungswert ist konstant und hängt nicht von m ab.

Die genauen Wahrscheinlichkeiten für die Shiftlänge hängen von P ab. Wir können den Erwartungswert aber abschätzen. Sei Σ_P die Menge der in P vorkommenden Zeichen. Es ist $|\Sigma_P| \leq \min\{m, |\Sigma|\}$. Wir dürfen annehmen, dass jeweils ein Zeichen in Σ_P zur Shiftlänge

```
def Horspool_shift(P):
    """return Horspool shift function pattern P"""
    shift, m = dict(), len(P)  # start with empty dict
    for j in range(m-1): shift[P[j]] = m-1-j
    return dict2function(shift, m)

def Horspool(P, T):
    m, n  = len(P), len(T)
    shift = Horspool_shift(P)
    last, Plast = m-1, P[m-1]
    while True:
        # Shift till last character matches or text ends
        while last < n and T[last] != Plast:
            last += shift(T[last])
        if last >=n: break  # end of T
        # Test remaining characters; then shift onwards
        if T[last-(m-1):last] == P[0:m-1]:
            yield (last-m+1, last+1)  # match found
        last += shift(Plast)
```

Abbildung 3.7: Python-Implementierung des Horspool-Algorithmus.

$1, 2, 3, |\Sigma_P|$ gehört: Wenn Zeichen wiederholt werden, werden andere Shiftlängen größer, nicht kleiner. Die Zeichen in $\Sigma \setminus \Sigma_P$ haben die Shiftlänge m.

Das letzte Zeichen des Fensters ist ein zufälliges aus Σ. Die erwartete Shiftlänge ist darum mindestens

$$\frac{1}{|\Sigma|} \cdot \left[\sum_{i=1}^{|\Sigma_P|} i + (|\Sigma| - |\Sigma_P|) \cdot m \right] = \frac{|\Sigma_P|(|\Sigma_P|+1)}{2|\Sigma|} + m(1 - |\Sigma_P|/|\Sigma|). \tag{3.1}$$

Wir betrachten mehrere Fälle (ohne konkrete Annahmen über Σ_P und Σ kann man nichts weiter aussagen). Es ist stets $\Sigma_P \subseteq \Sigma$.

Großes Alphabet $|\Sigma| \in \Theta(m)$**:**

- Es sei $|\Sigma_P| \in \mathcal{O}(1)$, das Pattern-Alphabet also klein gegenüber dem Alphabet. Dann liefert der zweite Summand in 3.1, dass die erwartete Shiftlänge $\Theta(m)$ ist.

- Es sei $|\Sigma_P| \in \Theta(m)$. Dann liefert der erste Summand, dass die erwartete Shiftlänge ebenfalls $\Theta(m)$ ist.

Kleines Alphabet $|\Sigma| \in \mathcal{O}(1)$**:** Im Fall $|\Sigma_P| < |\Sigma|$ liefert uns der zweite Summand, dass die erwartete Shiftlänge $\Theta(m)$ ist. Im Fall $|\Sigma_P| = |\Sigma|$ ist die erwartete Shiftlänge lediglich $(|\Sigma| + 1)/2$, also $\mathcal{O}(1)$.

Wir halten fest: In der Praxis (und das sieht man auch der Analyse an) ist der Horspool-Algorithmus gut, wenn das Alphabet Σ groß ist und die im Muster verwendete Buchstabenmenge Σ_P demgegenüber klein.

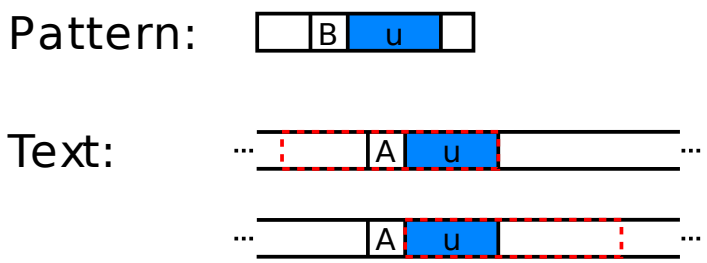


Abbildung 3.8: Illustration des Teilstring-basierten Pattern-Matchings. Das Pattern enthält (irgendwo) den Teilstring u, nicht aber $\mathtt{A}u$. Links von u wurde im Text der Buchstabe $\mathtt{A}$ gelesen. Da $\mathtt{A}u$ aber kein Teilstring des Patterns ist, können wir das Fenster (rot gestrichelt) soweit verschieben, dass der Beginn des Fensters auf das gefundene Vorkommen von u fällt. (Wenn das Fenster weniger weit verschoben würde, enthielte es $\mathtt{A}u$, was aber kein Teilstring des Patterns ist.)

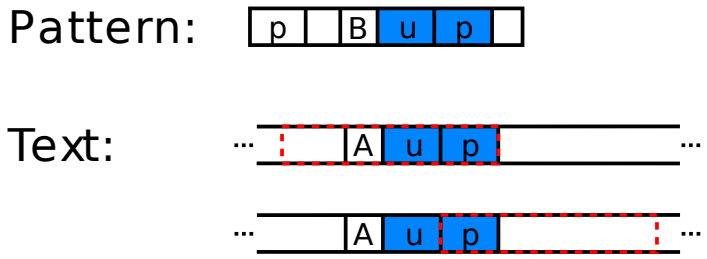


Abbildung 3.9: Die Verschiebung kann unter Umständen noch größer als in Abbildung 3.8 ausfallen, wenn das längste Suffix p der bisher gelesenen Zeichen bekannt ist, das ein Präfix des Patterns ist.

Sunday-Algorithmus. Variante von **?**: Berechne Shifts nicht anhand des letzten Zeichens des Suchfensters, sondern anhand des Zeichens dahinter. Dadurch sind längere Shifts möglich, aber es muss auch ein Zeichen mehr verglichen werden. In der Regel führt diese Variante zu einem langsameren Algorithmus.

3.7 Backward Nondeterministic DAWG Matching

3.7.1 Teilstring-basierter Ansatz

Die Möglichkeit, weite Teile des Textes zu überspringen, ist sehr wünschenswert, und es stellt sich die Frage, wie man möglichst weit springen kann. Dabei wird das aktuell betrachtete Fenster (wie auch beim Horspool-Algorithmus) von rechts nach links gelesen.

Die Idee besteht nun darin, nicht nur solange von rechts nach links zu lesen, bis es ein Mismatch mit dem Pattern gibt, sondern solange, bis der gelesene Teil *kein Teilstring* des Patterns ist. Daraus ergibt sich dann sofort, wie weit wir das Fenster verschieben können, ohne ein Vorkommen zu verpassen (siehe Abbildung 3.8). Daher spricht man hier von einem Teilstring-basierten Ansatz.

Wir können das Fenster noch weiter verschieben, wenn wir nachhalten, was das längste Suffix des aktuellen Fensters ist, das auch ein Präfix des Patterns ist (siehe Abbildung 3.9).

Wir benötigen dazu eine Datenstruktur, die uns erlaubt

1. einem gelesenen Fenster von rechts nach links Zeichen anzufügen,
2. festzustellen, ob der bisher gelesene Teil ein Teilstring des Patterns ist,
3. festzustellen, ob der bisher gelesene Teil sogar ein Präfix des Patterns ist.

3.7.2 Der Suffixautomat

Diese genannten Anforderungen werden erfüllt durch den Suffixautomaten des *reversen Patterns*. Der (deterministische) *Suffixautomat* für den String x ist ein DFA mit folgenden Eigenschaften:

- Es existiert vom Startzustand aus ein Pfad mit Label y genau dann, wenn y ein Teilstring von x ist.
- Der Pfad mit Label y endet genau dann in einem akzeptierenden Zustand, wenn y ein Suffix von x ist.
- Es muss nicht zu jedem Zustand und jedem Buchstaben eine ausgehende Kante geben; fehlende Kanten führen in einen implizit vorhandenen besonderen Zustand „FAIL“.

Wird der Suffixautomat für das reverse Pattern P^{rev} zu Pattern P konstruiert, so erlaubt die zweite Eigenschaft das Erkennen von Suffixen von P^{rev}, also Präfixen von P. Eigentlich könnte der Suffixautomat auch *Teilstringautomat* heißen.

Wie kann man einen solchen Automaten kostruieren? Wie immer ist es zunächst einfacher, einen äquivalenten nichtdeterministischen Automaten anzugeben. Der Automat besteht lediglich aus einer Kette von $|P| + 1$ Zuständen plus einem Startzustand. Vom Startzustand gibt es zu jedem Zustand ε-Übergänge. Ein Beispiel ist in Abbildung 3.10 gezeigt. Durch die Epsilon-Transitionen vom Startzustand in jeden Zustand kann man an jeder Stelle des Wortes beginnen, Teilstrings zu lesen. Man gelangt auch sofort in den akzeptierenden Zustand, denn man hat ja das leere Suffix des Wortes erkannt. Man beachte, dass im Startzustand *keine* Σ-Schleife vorliegt, denn der Automat wird jeweils nur auf ein Fenster angewendet (von rechts nach links). Sobald die aktive Zustandsmenge leer wird, wird das Fenster nicht weiter bearbeitet; dies entspricht dem Zustand FAIL des entsprechenden deterministischen Automaten.

Es gibt nun zwei Möglichkeiten:

1. Bitparallele Simulation des nichtdeterministischen Suffixautomaten. Man erhält den BNDM-Algorithmus (Backward Non-deterministic DAWG Matching).
2. Konstruktion des deterministischen Suffixautomaten. Dieser ist ein DAWG (directed acyclic word graph). Man erhält den BDM-Algorithmus (Backward DAWG Matching). Der NFA kann mit der Teilmengenkonstruktion in einen äquivalenten DFA überführt (und ggf. noch minimiert) werden; hierfür ist aber Linearzeit nicht garantiert. Die Konstruktion des deterministischen Suffixautomaten ist in Linearzeit möglich, aber

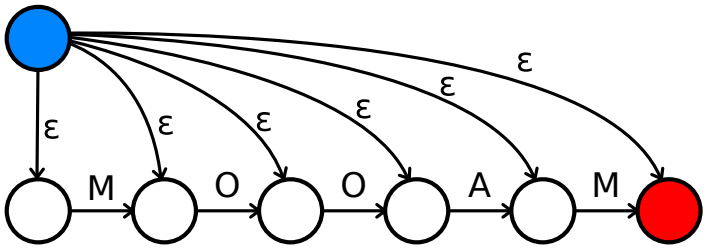


Abbildung 3.10: Nichtdeterministischer Suffixautomat für das Wort `MOOAM` (blau: Startzustand, rot: akzeptierender Zustand)

```
def BNDM(P, T):
    mask, _, accept = ShiftAnd_single_masks(P[::-1]) # reverse pattern
    return BNDM_with_masks(T, mask, accept, len(P))

def BNDM_with_masks(T, mask, accept, m):
    n, window = len(T), m  # current window is T[window-m:window]
    while window <= n:
        A = (1 << m) - 1  # bit mask of m Ones: all states active
        j, lastsuffix = 1, 0
        while A != 0:
            A &= mask(T[window-j]) # process j-th character from right
            if A & accept != 0:    # accept state reached
                if j == m:         # full pattern found?
                    yield (window - m, window)
                    break
                else:              # only found proper prefix
                    lastsuffix = j
            j += 1
            A = A << 1
        window += m - lastsuffix   # shift the window
```

Abbildung 3.11: Python-Code zum BNDM-Algorithmus

kompliziert. Wir gehen hier nicht darauf ein und greifen ggf. auf die beschriebene ineffizientere Teilmengen-Konstruktion zurück.

3.7.3 Backward Nondeterministic DAWG Matching (BNDM)

Zu Beginn sind durch die Epsilon-Transitionen alle Zustände aktiv (der Startzustand und der ganz linke Zustand werden nicht verwaltet; vgl. Abbildung 3.10) Es werden solange Zeichen im aktuellen Fenster (von rechts nach links) gelesen, wie noch Zustände aktiv sind. Die aktive Zustandsmenge A wird wie beim Shift-And-Algorithmus durch Links-Schieben der Bits und Ver*und*en mit Masken aktualisiert. Nach jeder Aktualisierung wird getestet, ob der akzeptierende Zustand aktiv ist. Wenn nach j gelesenen Zeichen der akzeptierende Zustand aktiv ist, wissen wir, dass wir ein passendes Suffix (des reversen Patterns) der Länge j gelesen haben, also das Präfix der Länge j von P. Entsprechend können wir das

Fenster so verschieben, dass der Beginn des nächsten Suchfensters auf dieses Präfix fällt, nämlich um $m - j$ Zeichen ($j = 0$ entspricht dem leeren Präfix).

Der Code ist relativ einfach; siehe Abbildung 3.11. Die Masken werden wie beim Shift-And-Algorithmus berechnet, nur dass das *reverse* Pattern zu Grunde gelegt wird. Der Wert von j, bei dem zuletzt ein Präfix (Suffix des reversen Patterns) erkannt wurde, wird in der Variablen `lastsuffix` festgehalten. Die aktive Zustandsmenge sind zu Beginn alle betrachteten m Zustände, also $A = (1, \ldots, 1)_2 = 2^m - 1 = (1 \ll m) - 1$. (Auf gar keinen Fall wird zu Berechnung von Zweierpotenzen eine Potenzfunktion aufgerufen!) Wurde nach dem Lesen des j-ten Zeichens von rechts ein Präfix erkannt, gibt es zwei Möglichkeiten: Wenn noch nicht das ganze Pattern erkannt wurde ($j \neq m$), dann wird der entsprechende Wert von j als `lastsuffix` gespeichert. Wurde bereits das ganze Pattern erkannt ($j = m$), dann wird dieses gemeldet, aber `lastsuffix` nicht neu gesetzt (sonst käme es zu keiner Verschiebung; dass das ganze Muster ein (triviales) Präfix ist, haben wir schon gewusst; es interessiert aber das nächstkürzere Präfix).

3.7.4 Backward DAWG Matching (BDM)

In der Praxis ist die explizite Konstruktion des deterministischen Suffixautomaten so aufwändig, dass man (bei kurzen Mustern) lieber BNDM verwendet oder (bei langen Mustern) auf eine andere Alternative zurückgreift (das Suffixorakel, s.u.). Wir beschreiben hier die deterministische Variante nur der Vollständigkeit halber.

In der Vorverarbeitungsphase erstellen wir zu Pattern P den (deterministischen) Suffixautomaten von P^{rev}, beispielsweise aus dem entsprechenden NFA mit der Teilmengenkonstruktion. Während der Suche gibt es stets ein aktuelles Suchfenster der Länge $|P| = m$. Wir lesen das Fenster von rechts nach links mit dem deterministischen Suffixautomaten, solange nicht der Zustand FAIL eintritt, der der leeren aktiven Zustandsmenge des NFA entspricht.

Die Übergangsfunktion δ des DFA kann man wieder mit Hilfe einer Hashtabelle codieren und diese in einer Funktion verpacken, die FAIL zurückliefert, wenn zu einer gewünschten Kombination aus aktuellem Zustand und Textzeichen kein expliziter Folgezustand bekannt ist.

Ob wir nun mit FAIL abbrechen oder ein Match finden, wir verschieben in jedem Fall das Fenster genau wie bei BNDM erläutert, so dass ein Präfix des Fensters mit einem Präfix von P übereinstimmt (vgl. Abbildung 3.9).

Dieser Algorithmus braucht im schlimmsten Fall (worst-case) $\mathcal{O}(mn)$ Zeit. Durch Kombination mit KMP lässt sich wieder $\mathcal{O}(n)$ erreichen (ohne Beweis). Im besten Fall braucht der Algorithmus, wie auch der Horspool-Algorithmus, $\mathcal{O}(n/m)$ Zeit. Eine average-case-Analyse führt auf $O(n \log_{|\Sigma|} m/n)$ (ohne Beweis).

3.8 Erweiterte Patternklassen

Bitparallele Algorithmen wie Shift-And, Shift-Or und BNDM lassen sich relativ leicht von einfachen Patterns auf erweiterte Patterns verallgemeinern, indem man passende nichtdeterministische Automaten betrachtet. In diesem Abschnitt stellen wir einige der Möglichkeiten

vor: Verallgemeinerte Strings, Gaps beschränkter Länge `-x(,)-`, optionale Zeichen `?*` und Wiederholungen `+*`). Für kurze Patterns sind bitparallele Algorithmen also nicht nur sehr schnell, sondern auch sehr flexibel.

3.8.1 Verallgemeinerte Strings

Verallgemeinerte Strings sind Strings, die *Teilmengen* des Alphabets als Zeichen besitzen. (Normale Strings bestehen aus einfachen Zeichen, also gewissermaßen ein-elementigen Teilmengen des Alphabets). Dies erlaubt uns, beispielsweise die Stringmenge { `ACG,AGG` } kompakt als `A[CG]G` zu schreiben, wobei hier `[CG]` die Menge aus `C` und `G` bedeutet. Bitparallele Algorithmen können solche Zeichenklassen fast ohne Änderung bereits verarbeiten; der einzige Unterschied liegt in der Vorverarbeitung der Masken: Bisher hatte an jeder Position immer genau eine der Masken ein 1-Bit; jetzt können mehrere Masken an einer bestimmten Position ein 1-Bit besitzen. Ein wichtiger Spezialfall ist, dass an einer Position *jedes* Zeichen erlaubt ist; dies wird im DNA-Alphabet als `N` und ansonsten auch mit `x`, `X` oder Σ ausgedrückt. Betrachten wir $P =$ `aabaXb` mit $\Sigma = \{$ `a`, `b` $\}$, so erhalten wir folgende Masken.

	`bXabaa`
$mask^a$	011011_2
$mask^b$	110100_2

3.8.2 Gaps beschränkter Länge

Unter einem *Gap beschränkter Länge* verstehen wir in diesem Abschnitt eine nichtleere Folge beliebiger Zeichen, deren Länge durch u und v beschränkt ist; wir verwenden die Notation `x(`u`,`v`)`. Beispielsweise sei das Pattern $P =$ `b-a-x(1,3)-b` gegeben. Ein entsprechender NFA ist in Abbildung 3.12 zu sehen. Die beliebigen Zeichen werden durch Σ-Übergänge dargestellt, die flexible Länge durch ε-Übergänge, die alle vom *ersten* Zustand eines `x(,)`-Blocks ausgehen. Es gibt also vom initialen Zustand aus genau $v - u$ mit ε beschriftete Kanten zu den $v - u$ Folgezuständen. Im Beispiel werden zwei ε-Übergänge benötigt, da $3 - 1 = 2$.

Einschränkungen:

1. Wir erlauben `x(,)` im Muster nicht ganz vorne und nicht ganz hinten.
2. Wir erlauben nicht zweimal hintereinander `x(,)`, denn

   $$\ldots\texttt{-x(}u,v\texttt{)-x(}u',v'\texttt{)-}\ldots \quad \text{ist äquivalent zu} \quad \ldots\texttt{-x(}u+u',v+v'\texttt{)-}\ldots.$$

3. Wir erlauben keine Gaps mit Länge 0, d.h. `x(`u,v`)` ist nur für $u > 0$ erlaubt. Diese Einschränkung umgehen wir im nächsten Unterabschnitt mit einer anderen Konstruktion.

Implementierung:

1. normaler Shift-And-Schritt für Textzeichen c:

   $$A \leftarrow \big((A \ll 1) \,|\, 1\big) \;\&\; mask^c$$

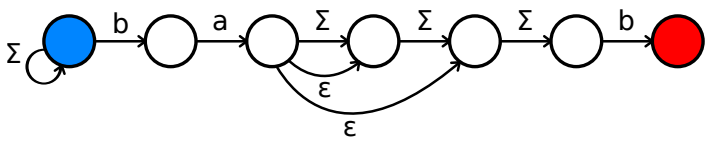


Abbildung 3.12: NFA zur Suche nach dem Pattern `b-a-x(1,3)-b`.

2. Simulation der ε-Übergänge

Zur Simulation der ε-Übergänge verwenden wir Subtraktion, die auf Bit-Ebene den gewünschten Effekt des Auffüllens hat. Wir verwenden zwei Bitmasken: In der Maske I setzen wir alle Bits, die zu den Anfängen der `x(,)`-Blöcke gehören, von denen also die ε-Kanten ausgehen. In der Maske F setzen wir alle Bits, die zu Zuständen *nach* den gehören, die von der letzten ε-Kante eines `x(,)`-Blocks erreicht werden.

Durch den Ausdruck $A\&I$ werden zunächst alle aktiven Initialzustände mit ausgehenden ε-Kanten ausgewählt. Durch Subtraktion von F werden alle Bits zwischen einem aktiven I-Zustand (inklusive) und dem zugehörigen F-Zustand (exklusive) gesetzt.

Das folgende Beispiel verdeutlicht die Wirkungsweise:

F	`0100001000`
$A\&I$	`0000000001`
Diff.	`0100000111`

Da zum F-Bit links kein gesetztes I-Bit gehört, bleibt es bestehen und kann danach explizit gelöscht werden. Zum F-Bit rechts gibt es ein gesetztes I-Bit, und durch die Subtraktion wird das F-Bit gelöscht, während es sich gleichzeitig nach rechts bis zum I-Bit ausbreitet.

Ist der zu einem F-Zustand gehörende I-Zustand nicht aktiv, verbleibt ein 1-Bit im F-Zustand; dieses wird explizit durch Verundung mit dem Komplement $\sim F$ gelöscht. Aufgrund der Einschränkung, dass ein Gap nicht die Länge 0 haben darf, ist kein F-Zustand gleichzeitig I-Zustand des nächsten Gaps.

Es ergibt sich insgesamt die Update-Formel:

$$A \leftarrow A \mid \big[\big(F - (A \;\&\; I)\big) \;\&\; \sim F\big]$$

3.19 Bemerkung. Die Subtraktionstechnik kann man immer dann einsetzen, wenn in einem Bitvektor ein ganzer Bereich mit Einsen gefüllt werden soll.

3.8.3 Optionale und wiederholte Zeichen im Pattern*

Wir betrachten Patterns der Art `ab?c*e+`, dabei wirken die Zeichen `?`, `*` und `+` immer auf das vorhergegangene Zeichen. Es steht

- `b?` für ε oder `b`
- `c*` für ε, `c`, `cc`, `ccc`, ...

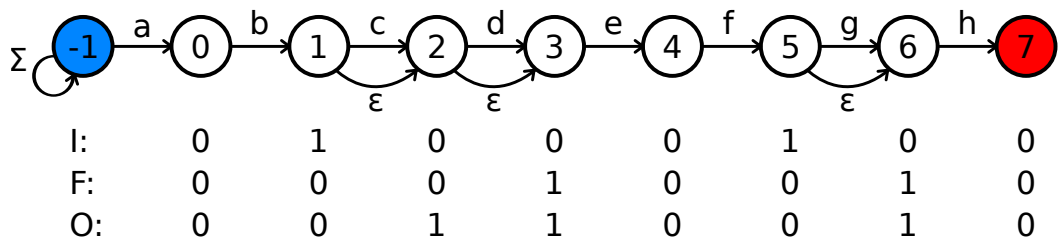


Abbildung 3.13: NFA für das Pattern `abc?d?efg?h` mit den dazugehörigen Bitmasken. Achtung: Das niederwertigste Bit steht hier links (unter dem Zustand 0).

- `e+` für `ee*`

Wir fangen mit Strings an, die nur `?` (und nicht `*` oder `+`) enthalten. Wieder suchen wir nach einer Möglichkeit, den NFA bitparallel zu simulieren.

Fall 1: Zeichen mit `?` haben mindestens ein normales Zeichen dazwischen. Dieser Fall ist relativ einfach, denn aufgrund der Annahme gibt es keine Ketten von ε-Kanten. Wir müssen also nur aktive Zustände entlang einzelner ε-Kanten propagieren. Dazu sei I eine Maske der Bits, die zu Zuständen vor Zeichen mit Fragezeichen gehören („Initialzustände“). Aktive Initialzustände werden geshiftet und mit den aktiven Zuständen verodert:

$$A \leftarrow A \,|\, (A \,\&\, I) \ll 1,$$

Fall 2: Fragezeichen dürfen beliebig verteilt sein. Hierbei können (lange) Ketten von ε-Kanten entstehen, wie z.B. in Abbildung 3.13. Wir nennen eine maximal lange Folge von aufeinander folgenden Zeichen mit `?` einen *Block*. Wir benötigen mehrere Masken, die die „Blöcke“ der `?` beschreiben.

- I: je Block der Start der ersten ε-Kante,
- F: je Block das Ende der letzten ε-Kante,
- O: alle Endzustände von ε-Kanten.

Abbildung 3.13 zeigt den NFA zur Suche nach dem Pattern `abc?d?efg?h` und die dazugehörigen Bitmasken I, F und O. Als Binärzahl geschrieben ergeben sich also die Bitmasken

$$\begin{aligned} I &= \texttt{00100010}, \\ F &= \texttt{01001000}, \\ O &= \texttt{01001100}. \end{aligned}$$

Es sei $A_F := A \,|\, F$ die Menge der Zustände, die entweder aktiv sind oder hinter einem `?`-Block liegen. Wir wollen zeigen, dass die folgenden Operationen das korrekte Update durchführen.

$$A \leftarrow A \,\Big|\, \Big[O \,\&\, \big((\sim (A_F - I)) \oplus A_F\big)\Big] \tag{3.2}$$

dabei steht das Symbol $\oplus$ für exklusives Oder (xor).

Als Beispiel nehmen wir an, dass nach der Shift-And-Phase die Zustände -1 und 1 aktiv sind; das entspricht $A = 00000010$. Wir werten nun (**??**) und (3.2) Schritt für Schritt aus:

$$\begin{aligned} A_F = A \,|\, F &= \texttt{01001010} \\ I &= \texttt{00100010} \\ (A_F - I) &= \texttt{00101000} \\ \sim (A_F - I) &= \texttt{11010111} \\ A_F &= \texttt{01001010} \\ \big(\sim (A_F - I)\big) \oplus A_F &= \texttt{10011101} \\ O &= \texttt{01001100} \\ O \,\&\, \big(\sim (A_F - I)\big) \oplus A_F &= \texttt{00001100} \end{aligned}$$

Das Ergebnis sagt uns, dass die Zustände 2 und 3 zusätzlich aktiviert werden müssen. Das stimmt mit Abbildung 3.13 überein, denn die Zustände 2 und 3 sind vom Zustand 1 aus über ε-Kanten erreichbar.

Nun erweitern wir den Algorithmus so, dass auch Wiederholungen (`*` und `+`) erlaubt sind. Dabei steht `+` für ein oder mehr Vorkommen und `*` für null oder mehr Vorkommen des jeweils vorangehenden Buchstabens. Beispiel: `abc+def*g*h`

Verwendete Bitmasken: $A, mask[c], rep[c], I, F, O$:

- $A_i = 1 \iff$ Zustand i aktiv
- $mask[c]_i = 1 \iff P[i] = c$
- $rep[c]_i = 1 \iff P[i]$ wiederholbar (`+` oder `*`)
- I, F, O : für optionale Blöcke (`?` und `*`), wie oben.

Schritte des Algorithmus:

$$\begin{aligned} A &\leftarrow \Big[\big((A \ll 1) \,|\, 1\big) \,\&\, mask[c]\Big] \,\Big|\, \Big[A \,\&\, rep[c]\Big] \\ A_F &\leftarrow A \,|\, F \\ A &\leftarrow A \,\Big|\, \Big[O \,\&\big((\sim (A_F - I)) \oplus A_F\big)\Big] \end{aligned}$$

⇑ 28.04.11

3.9 Backward Oracle Matching (BOM)*

BNDM (bitparallele Simulation eines NFA) ist wie immer dann empfehlenswert, wenn das Pattern höchstens so lang ist wie die Registerbreite der benutzten Maschine. Für lange Patterns müsste man auf die deterministische Variante BDM zurückgreifen, aber die Konstruktion des DFA ist aufwändig, was sich in der Praxis unter Umständen nicht auszahlt. Was tun?

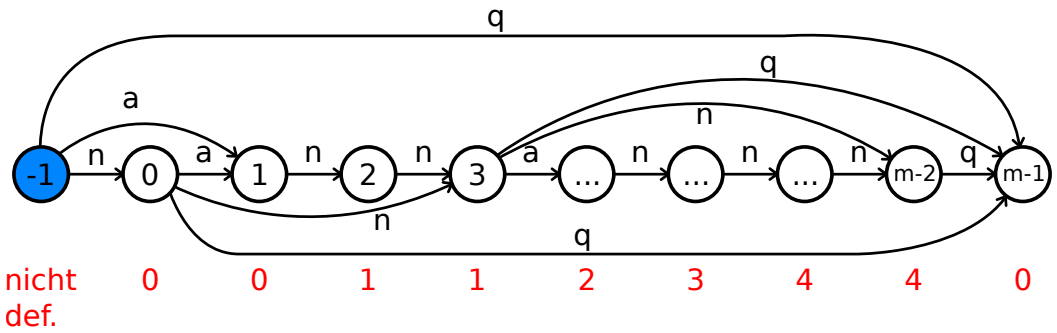


Abbildung 3.14: Suffix-Orakel für den String `nannannnq` (blau: Startzustand, rot: Suffix-Funktion *suff*)

Wir betrachten eine Vereinfachung. Eigentlich benötigen wir vor allem einen Automaten, der erkennt, wann ein String *kein* Teilstring von P ist. Das bedeutet, wir suchen einen Automaten, der Teilstrings von P und evtl. einige zusätzliche Strings akzeptiert. Dadurch kann man das Fenster weniger schnell (und evtl. weniger weit) verschieben. Wenn die Datenstruktur jedoch einfacher aufzubauen ist, kann sich das unter Umständen lohnen.

Gewünschte Eigenschaft:

- u ist Teilstring von $P \Rightarrow$ Es gibt einen Pfad vom Startzustand aus, der u buchstabiert.

Die Rückrichtung fordern wir nicht streng, sondern begnügen uns mit der Forderung, dass nicht zu viele zusätzliche Strings erkannt werden. Präziser formuliert: Es wird nur ein einziger String der Länge m erkannt (nämlich P), und es werden keine längeren Strings erkannt. Um diese Forderung zu erfüllen, werden mindestens $m+1$ Zustände benötigt. Zusätzlich verlangen wir, dass der Automat auch nicht mehr als $m+1$ Zustände und nicht mehr als $2m-1$ Übergänge hat. Eine solche entsprechende Datenstruktur nennen wir ein *Teilstring-Orakel* zu P.

Das algorithmische Prinzip beim String-Matching bleibt wie vorher: Man liest ein Textfenster rückwärts und wendet darauf das Orakel von $x := P^{\text{rev}}$ an. Sobald festgestellt wird, dass der gelesene String kein Teilstring von P^{rev} ist, kann man das Fenster hinter das letzte gelesene Zeichen verschieben (vgl. Abbildung 3.8).

Konstruktion des Teilstring-Orakels. Wir beschreiben jetzt die Konstruktion des Orakels; danach weisen wir die gewünschten Eigenschaften nach.

Wir beginnen mit dem Startzustand -1 und fügen dann in $m := |x|$ Iterationen jeweils einen Zustand und mindestens eine Kante hinzu. In Iteration $0 \leq i < m$ wird das Wort um den Buchstaben x_i verlängert, indem Zustand i und mindestens die Kante $i-1 \xrightarrow{x_i} i$ hinzugefügt werden. Kanten von $i-1$ nach i nennen wir *innere Kanten*.

Nun ist für $i \geq 1$ weiter zu beachten, dass alle schon existierenden Teilstrings, die mit x_{i-1} enden, um x_i verlängert werden können. Enden diese ohnehin in Zustand $i-1$, so geschieht dies automatisch durch die neue innere Kante. Enden diese aber in einem früheren Zustand $j < i-1$, von dem es noch keine ausgehende Kante mit Label x_i gibt, so muss die Kante $j \xrightarrow{x_i} i$ zusätzlich eingefügt werden. Solche Kanten nennen wir *äußere Kanten*. Woher wissen

wir, ob es Teilstrings (Suffixe von $x[0 \ldots i-1]$) gibt, die die Einfügung von äußeren Kanten nötig machen?

Wir definieren eine Hilfsgröße: Sei *suff*$[i]$ der Zustand, in dem das längste Suffix von $x[0 \ldots i]$ (also des Präfix der Länge i von x) endet, das *nicht* in i endet. Wir setzen *suff*$[-1] := -1$ (oder undefiniert). Offensichtlich ist dann *suff*$[0] = -1$ (entsprechend dem leeren Suffix) und stets *suff*$[i] < i$. Hat man von jedem Zustand Zugriff auf das längste Suffix, erhält man alle Suffixe durch Iteration, bis man eine existierende passende ausgehende Kante findet oder im Zustand -1 angekommen ist. Der Zielzustand einer solchen existierenden Kante (bzw. -1) ist dann offenbar das gesuchte neue *suff*$[i]$.

Es ergibt sich folgender Algorithmus zur Berechnung der Übergangsfunktion δ des Orakels. (**TODO: Code an Nummerierung anpassen.**)

```
1  def deltaTableBOM(x):
2      m = len(x)
3      delta = dict()
4      suff = [None] * (m+1)
5      for i in range(1, m+1):
6          a = x[i-1]
7          delta[(i-1, a)] = i
8          k = suff[i-1]
9          while k is not None and (k, a) not in delta:
10             delta[(k, a)] = i
11             k = suff[k]
12         suff[i] = delta[(k, a)] if k is not None else 0
13     return lambda *args: delta.get(args, None)
```

In Zeile 7 werden die inneren Kanten erzeugt, in Zeile 10 die äußeren Kanten. In Zeile 12 wird *suff*$[i]$ gesetzt, entweder auf das Ziel einer existierenden Kante mit richtigem Buchstaben (dann gibt es das entsprechende Suffix und alle kürzeren Suffixe schon), oder auf den Startzustand -1, wenn sich dort bereits befindet und es keine ausgehende Kante mit dem korrekten Buchstaben gab (k == None). Abbilung 3.14 zeigt ein Suffix-Orakel und die zugehörige Funktion *suff*.

Anwendung des Orakels. Die delta-Funktion wird zum reversen Muster konstruiert; darauf wird dann die oben erwähnte Grundidee angewendet, das Fenster hinter das zuletzt gelesene Zeichen zu verschieben.

```
1  def matchesBOM(P, T):
2      delta = deltaTableBOM(P[::-1])
3      return BOMwithDelta(T, delta, len(P))
```

```
1  def BOMwithDelta(T, delta, m):
2      n = len(T)
3      window = m  # current window is T[window - m : window]
4      while window <= n:
5          q, j = 0, 1  # start state of oracle, character to read
6          while j <= m and q is not None:
7              q = delta(q,T[window-j])
8              j += 1
9          if q is not None: yield (window-m, window)
```

```
10          window += m-j+2
```

3.10 Auswahl eines geeigneten Algorithmus in der Praxis

Einen Überblick, wann in Abhängigkeit von Alphabetgröße und Patternlänge welcher Algorithmus in der Praxis vorteilhaft ist, findet sich bei **?**, Figure 2.22.

Zusammenfassend lässt sich Folgendes sagen: Ab einer hinreichend großen Alphabetgröße ist immer der Horspool-Algorithmus der schnellste. Bei sehr kleinen Alphabeten und kurzen Patterns ist Shift-Or gut (ein optimierter Shift-And). BNDM ist gut, solange der Automat bitparallel in einem Register verarbeitet werden kann. Für lange Muster bei einem kleinen Alphabet hat sich BOM als am effizientesten erwiesen.

Gute Algorithmen werden bei zunehmender Patternlänge nicht langsamer ($\mathcal{O}(mn)$ ist naiv!), sondern schneller ($\mathcal{O}(n/m)$ ist anzustreben!).

KAPITEL 4

Volltext-Indizes

Wenn wir in einem feststehenden Text (der Länge n) oft nach verschiedenen Mustern (der Länge m) suchen wollen, dann kann es sinnvoll sein, den Text vorzuverarbeiten und eine *Index-Datenstruktur* aufzubauen. Das kann einen Vorteil in der Gesamtlaufzeit bringen:

Online Mustersuche	Index-basierte Mustersuche	
$\mathcal{O}(m)$	$\mathcal{O}(n)$	Vorverarbeitung
$\mathcal{O}(n)$	$\mathcal{O}(m)$	Suche (ein Muster)
$\mathcal{O}\big(k(m+n)\big)$	$\mathcal{O}(n+km)$	insgesamt (k Muster)

Um natürlichsprachliche Texte zu indizieren (z.B. für Suchmaschinen), bieten sich Wort-basierte Verfahren an. Beim Indizieren von biologischen Texten ohne Wortstruktur, beispielsweise DNA, benötigen wir hingegen Index-Datenstrukturen, die die Suche nach beliebigen Teilstrings (ohne Rücksicht auf Wortgrenzen) erlauben. Diese werden *Volltext-Indizes* genannt. In diesem Kapitel geht es um Datenstrukturen, die dies ermöglichen. Hierzu gibt es zwei populäre Indexstrukturen, nämlich den *Suffixbaum* und das *Suffixarray*. Zudem gibt es den *q-gram-Index*, den wir aber aus Zeitgründen nicht diskutieren. Achtung: Nicht jede der genannten Datenstrukturen erreicht die oben genannten Laufzeiten.

4.1 Suffixbäume

Der Grundgedanke hinter Suffixbäumen ist, dass man einem beliebigen Muster p schnell ansehen kann, ob es in einem zuvor indizierten Text t vorkommt. Dazu wird zu t eine Baumstruktur so aufgebaut, dass jedes Suffix von t einem Pfad von der Wurzel zu einem Blatt

entspricht. Da jeder Teilstring von t ein Präfix eines Suffixes von t ist, muss man zur Teilstringsuche nur den richtigen Pfad von der Wurzel des Baums verfolgen; dies sollte sich in $O(|t|)$ Zeit machen lassen. Ein Problem dabei könnte aber der Platzbedarf des Baumes werden: Erzeugen wir aus allen Suffixen einen Trie (so dass wir im Prinzip den Aho-Corasick-Algorithmus anwenden können), benötigen wir zu viel Platz, denn die Gesamtlänge aller Suffixe ist $n + (n-1) + (n-2) + \cdots + 1 = \Theta(n^2)$, wobei $|t| = n$.

Es wäre schön, wenn jedes Blatt des (zu definierenden) Suffixbaums genau einem Suffix von t entsprechen würde. Da es aber möglich ist, dass gewisse Suffixe noch an anderer Stelle als Teilstrings (also als Präfix eines anderen Suffix) vorkommen, ist dieser Wunsch nicht notwendigerweise erfüllbar. Dies kann nur garantiert werden, wenn das letzte Zeichen des Strings eindeutig ist, also sonst nicht vorkommt. Es hat sich daher etabliert, einen *Wächter* (sentinel) an den zu indizierenden String anzuhängen; dieser wird gewöhnlich mit \$ bezeichnet.

Wir listen nun einige wünschenswerte Eigenschaften des Suffixbaums zu $t\$$ auf:

- Es gibt eine Bijektion zwischen den Blättern des Suffixbaums und den Suffixen von $t\$$.
- Die Kanten des Baums sind mit nicht-leeren Teilstrings von $t\$$ annotiert.
- Ausgehende Kanten eines Knotens beginnen mit verschiedenen Buchstaben.
- Jeder innere Knoten hat ≥ 2 Kinder (es gibt also keine Knoten, wenn sich die Kante nicht verzweigt).
- Jeder Teilstring von $t\$$ kann auf einem Pfad von der Wurzel ausgehend „abgelesen“ werden.

Wir definieren nun einige hilfreiche Begriffe, um danach Suffixbäume formal zu definieren.

4.1 Definition. Ein *gewurzelter Baum* ist ein zusammenhängender azyklischer Graph mit einem speziellen Knoten r, der *Wurzel*, so dass alle Kanten von der Wurzel weg weisen. Die *Tiefe* $depth(v)$ eines Knotens v ist seine Distanz von der Wurzel; das ist die Anzahl der Kanten auf dem eindeutigen Pfad von der Wurzel zu v. Insbesondere ist $depth(r) := 0$.

Sei Σ ein Alphabet. Ein Σ-*Baum* oder *Trie* ist ein gewurzelter Baum, dessen Kanten jeweils mit einem einzelnen Buchstaben aus Σ annotiert sind, so dass kein Knoten zwei ausgehende Kanten mit dem gleichen Buchstaben hat. Ein Σ^+-*Baum* ist ein gewurzelter Baum, dessen Kanten jeweils mit einem nichtleeren String über Σ annotiert sind, so dass kein Knoten zwei ausgehende Kanten hat, die mit dem gleichen Buchstaben beginnen.

Ein Σ^+-Baum heißt *kompakt*, wenn kein Knoten (außer ggf. der Wurzel) genau ein Kind hat (d.h., wenn jeder innere Knoten mindestens zwei Kinder hat).

Sei v ein Knoten in einem Σ- oder Σ^+-Baum. Dann sei $string(v)$ die Konkatenation der Kantenbeschriftungen auf dem eindeutigen Pfad von der Wurzel zu v. Wir definieren die *Stringtiefe* eines Knoten v als $stringdepth(v) := |string(v)|$. Diese ist in einem Σ^+-Baum normalerweise verschieden von $depth(v)$.

Sei $x \in \Sigma^*$ ein String. Existiert ein Knoten v mit $string(v) = x$, dann schreiben wir $node(x)$ für v. Ansonsten ist $node(x)$ nicht definiert. Es ist $node(\varepsilon) = r$, die Wurzel.

Ein Σ- oder ein Σ^+-Baum T *buchstabiert* $x \in \Sigma^*$, wenn x entlang eines Pfades von der Wurzel abgelesen werden kann, d.h., wenn ein (möglicherweise leerer) String y und ein Knoten v

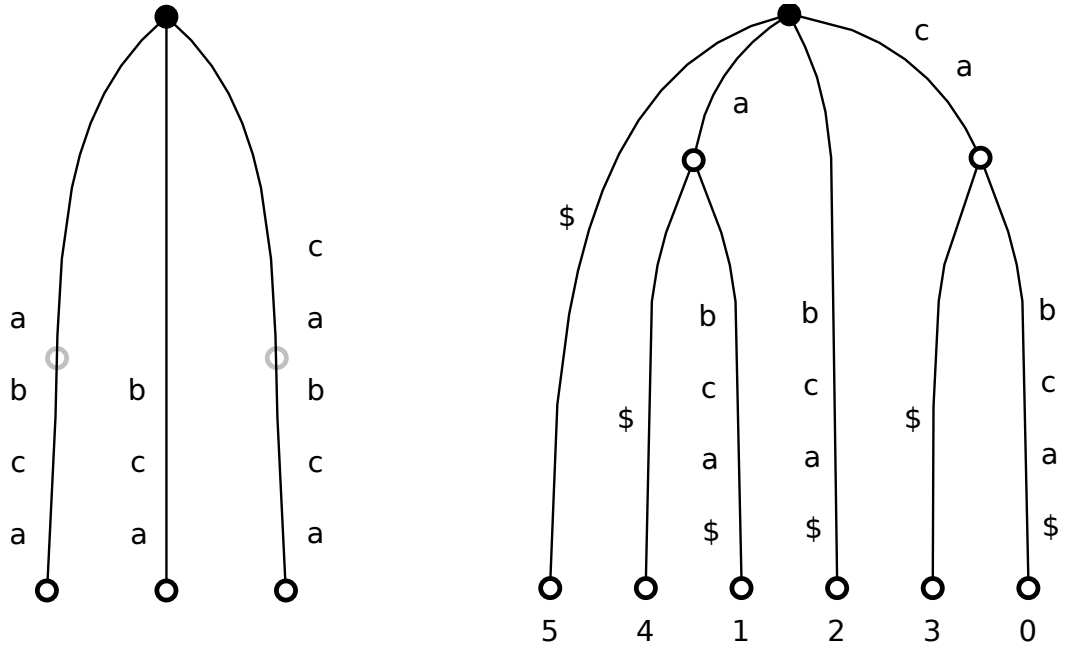


Abbildung 4.1: Suffixbäume für die Strings `cabca` und `cabca$`. Der Wächter sorgt dafür, dass für jedes Suffix genau ein Blatt existiert. Die Zahlenfolge unter dem rechten Bild gibt die Positionen im Text an und entspricht dem Suffixarray, da in jedem Knoten die ausgehenden Kanten sortiert sind.

existieren mit $string(v) = xy$ (dabei liegt v ggf. „unterhalb“ von x). Es sei $words(T)$ die Menge der Strings, die T buchstabiert.

4.2 Definition (Suffixbaum)**.** Der *Suffixbaum* von $s \in \Sigma^*$ ist der kompakte Σ^+-Baum mit $words(T) = \{ s' \mid s' \text{ ist ein Teilstring von } s \}$. Man beachte, dass wir im Normalfall zu s den Suffixbaum von $s\$$ betrachten, nicht den von s selbst, um eine Bijektion zwischen Suffixen und Blättern herzustellen.

Wir wollen erreichen, dass der Speicherplatzbedarf für einen Suffixbaum nur linear in der Stringlänge ist. Die Kompaktheit des Σ^+-Baums ist ein wichtiger Schritt, wie das folgende Lemma zeigt.

4.3 Lemma. Sei T der Suffixbaum zu $s\$$ mit $|s\$| = n$. Dann hat T genau n Blätter, es existieren $\leq n - 1$ innere Knoten und $\leq 2(n - 1)$ Kanten.

Beweis. Im Detail: Übung. Idee: Der Verzweigungsgrad ist ≥ 2 in jedem inneren Knoten. Zähle eingehende, ausgehende Kanten, innere Knoten und Blätter. □

An den Kanten des Suffixbaums zu $s\$$ stehen Teilstrings von $s\$$, deren Gesamtlänge noch quadratisch sein kann. Dies kann man verhindern, indem man die Teilstrings an den Kanten durch Indexpaare (i, j) repräsentiert: Das Paar (i, j) entspricht dem Label $s[i \dots j]$ und benötigt daher nur konstanten Platz. Insgesamt benötigt ein Suffixbaum also (nur) $\mathcal{O}(n)$ Platz, obwohl $\mathcal{O}(n^2)$ Teilstrings indiziert werden. Abbildung 4.1 illustriert einen Suffixbaum.

Man kann einen (einzigen) Suffixbaum zu mehreren verschiedenen Strings $s_1, \ldots, s_k$ erzeugen. Da man jedoch beim Aneinanderhängen Teilstrings erzeugen könnte, die in keinem der Eingabestrings vorkommen, muss man dabei die Strings voneinander durch verschiedene Trennzeichen trennen.

4.4 Definition. Seien $s_1, \ldots, s_k$ Strings über Σ. Seien $\$_1 < \$_2 < \cdots < \$_k$ Zeichen, die nicht in Σ vorkommen und lexikographisch kleiner sind als jedes Zeichen in Σ. Der (verallgemeinerte) Suffixbaum zu $s_1, \ldots, s_k$ ist der Suffixbaum zu $s_1\$_1 s_2\$_2 \ldots s_k\$_k$.

Für die Konstruktion eines Suffixbaums gibt es mehrere Möglichkeiten (wobei wir bei den Laufzeiten voraussetzen, dass das Alphabet konstante Größe hat).

- $\mathcal{O}(n^2)$ durch Einfügen aller Suffixe in einen Trie (unter Berücksichtigung der genannten Tricks, um den Speicherplatzbedarf linear zu halten)
- Algorithmus von **?**: $\mathcal{O}(n)$, online (String kann verlängert werden). Diesen Algorithmus stellen wir in Abschnitt 4.3 vor. Es ist bemerkenswert (und wichtig), dass die Konstruktion eines Suffixbaums in Linearzeit möglich ist.

Wir fassen die wichtigste Aussage in einem Satz zusammen.

4.5 Satz. *Zu einem String s kann der Suffixbaum von $s\$$ in $\mathcal{O}(n)$ Zeit konstruiert werden und benötigt $\mathcal{O}(n)$ Platz. Es besteht eine Bijektion zwischen den echten Suffixen von $s\$$ und den Blättern des Suffixbaums; jedes Blatt ist mit der eindeutigen Startposition des entsprechenden Suffixes von s annotiert.*

4.2 Suffixarrays

Ein Suffixbaum hat einen Platzbedarf von $\mathcal{O}(n)$, wie wir gezeigt haben. Allerdings ist die in $\mathcal{O}(n)$ versteckte Konstante verhältnismäßig groß. Eine Möglichkeit, die Konstante zu verkleinern, bieten *Suffixarrays*.

Ein Suffixarray eines Strings $s\$$ mit $|s\$| = n$ ist definiert als die Permutation `pos` von $\{0, \ldots, n-1\}$, die die Start**pos**itionen der Suffixe in lexikographischer Reihenfolge angibt. Zum Beispiel ist $(5, 4, 1, 2, 3, 0)$ das Suffixarray des Strings `cabca$`.

Das Suffixarray enthält also dieselben Informationen wie die unterste Ebene eines Suffixbaums (siehe Abbildung 4.1). Es kann daher in Linearzeit durch eine einfache Durchmusterung des Suffixbaums konstruiert werden, wenn man in jedem inneren Knoten sicherstellt, die ausgehenden Kanten in sortierter Reihenfolge (anhand des ersten ausgehenden Buchstaben) zu durchlaufen. Interessanter ist natürlich die Frage, ob das Suffixarray auch in Linearzeit konstruiert werden kann, ohne zuvor den Suffixbaum zu erstellen. Diese Frage wurde im Jahr 2003 positiv beantwortet.

Wir zeigen hier, dass man das Suffixarray sehr einfach mit der Sortierfunktion der Standardbibliothek konstruieren kann. Man muss lediglich beim Vergleich zweier Positionen p_1, p_2 nicht die Ordnung der entsprechenden natürlichen Zahlen bestimmen, sondern die Ordnung der Suffixe, die an diesen Positionen beginnen. Optimale Sortierverfahren haben eine Komplexität von $\mathcal{O}(n \log n)$ Vergleichen; der Vergleich zweier Suffixe kostet nicht konstante Zeit, sondern $\mathcal{O}(n)$ Zeit; damit ergibt sich eine Laufzeit von $\mathcal{O}(n^2 \log n)$, was wesentlich schlechter

ist als das optimale $\mathcal{O}(n)$, aber dafür fast keinen Implementierungsaufwand erfordert und für kleine Beispiele in jedem Fall praktikabel ist.

```
def suffix(T):
    """gibt eine Funktion zurueck,
    die bei Eingabe i das i-te Suffix von T zurueckgibt"""
    def suf(i):
        return T[i:]
    return suf

def suffixarray(T):
    """berechnet das Suffixarray von T mit der sort-Funktion,
indem explizit Textsuffixe verglichen werden.
Laufzeit: Je nach sort-Implementierung O(n^3) oder O(n^2 log n)"""
    pos = list(range(len(T)))  # 0 .. |T|-1
    pos.sort(key = suffix(T))  # Sortierschluessel: suffix-Funktion
    return pos
```

Ein Suffixarray repräsentiert zunächst nur die „Blattebene“ des Suffixbaums. Um die restliche Struktur des Baums zu repräsentieren, benötigen wir ein zweites Array, das sogenannte `lcp`-Array (für *longest common prefix*), das die gemeinsame Präfixlänge von zwei im Suffixarray benachbarten Suffixen angibt. Formal:

$$\texttt{lcp}[0] := \texttt{lcp}[n] := -1$$
$$\texttt{lcp}[r] := \max\{|x| : x \text{ ist Präfix von } S[\texttt{pos}[r-1]\ldots] \text{ und von } S[\texttt{pos}[r]\ldots]\}$$

Die Werte -1 am linken und rechten Rand haben eine Wächter-Funktion.

4.6 Beispiel (Suffixarray mit `pos` und `lcp`)**.** Wir zeigen das Suffixarray zu `cabca$`.

r	`pos`$[r]$	`lcp`$[r]$	Suffix
0	5	-1	`$`
1	4	0	`a$`
2	1	1	`abca$`
3	2	0	`bca$`
4	3	0	`ca$`
5	0	2	`cabca$`
6	–	-1	–

♡

Um Suffixbaum und Suffixarray in Beziehung zu setzen, sind folgende Aussagen von Bedeutung, die man sich an Beispiel 4.6 veranschaulichen kann.

4.7 Lemma. Jeder innere Knoten v des Suffixbaums entspricht einem Intervall $I_v := [L_v, R_v]$ im Suffixarray. Die Menge $\{\, \texttt{pos}[r] \mid L_v \le r \le R_v \,\}$ entspricht der Blattmenge unter v.

Sei $d_v := \mathit{stringdepth}(v)$. Dann gilt:

- $\texttt{lcp}[L_v] < d_v$
- $\texttt{lcp}[R_v + 1] < d_v$

- $\texttt{lcp}[r] \geq d_v$ für $L_v < r \leq R_v$
- $\min\{\,\texttt{lcp}[r] \mid L_v < r \leq R_v\,\} = d_v$

Es ist zweckmäßig, für ein Intervall mit den Eigenschaften aus Lemma 4.7 den Begriff des d-Intervalls einzuführen.

4.8 Definition. Sei $(\texttt{pos}, \texttt{lcp})$ ein Suffixarray. Ein Intervall $[L, R]$ heißt *d-Intervall*, wenn $\texttt{lcp}[L] < d$, $\texttt{lcp}[R+1] < d$, $\texttt{lcp}[r] \geq d$ für $L < r \leq R$, und $\min\{\,\texttt{lcp}[r] \mid L < r \leq R\,\} = d$.

4.9 Satz. *Es besteht eine Bijektion zwischen den Knoten eines Suffixbaums (Wurzel, innere Knoten, Blätter) der Stringtiefe d und den d-Intervallen des Suffixarrays.*

Die d-Intervalle für alle d bilden zusammen einen Baum, den sogenannten `lcp`*-Intervallbaum, wenn man festlegt, dass ein Intervall ein Kind eines anderen ist, wenn es ein Teilintervall davon ist. Dessen Baumtopologie ist genau die Suffixbaumtopologie.*

Wir werden sehen, dass sich manche Probleme eleganter mit dem Suffixarray (`pos` und `lcp`) lösen lassen, andere wiederum eleganter mit einem Suffixbaum. Man kann das Suffixarray noch um weitere Tabellen ergänzen, zum Beispiel um explizit die inneren Knoten und ihre Kinder zu repräsentieren.

4.3 Ukkonens Algorithmus: Suffixbaumkonstruktion in Linearzeit

Vorbemerkungen zur Alphabet-Abhängigkeit. Wir gehen in diesem Abschnitt davon aus, dass das Alphabet eine konstante Größe hat. Ist dies nicht der Fall, hängen die Laufzeiten der Suffixbaum-Algorithmen davon ab, mit welcher Datenstruktur die Menge der Kinder eines inneren Knotens verwaltet wird. Sei c_v die Anzahl der Kinder von Knoten v; stets ist $c_v \in \mathcal{O}(|\Sigma|)$.

- Mit einer verketteten Liste benötigt man $\mathcal{O}(c_v)$ Platz, aber auch $\mathcal{O}(c_v)$ Zeit, um ein bestimmtes Kind zu finden. Insgesamt benötigt man $\mathcal{O}(n)$ Platz.
- Mit einem balancierten Baum benötigt man $\mathcal{O}(c_v)$ Platz (mit einer größeren Konstanten als bei der Liste), aber nur $\mathcal{O}(\log c_v)$ Zeit, um ein bestimmtes Kind zu finden. Insgesamt benötigt man $\mathcal{O}(n)$ Platz.
- Mit einem direkt adressierbaren Array benötigt man $\mathcal{O}(|\Sigma|)$ Platz pro Knoten und daher insgesamt $\mathcal{O}(n|\Sigma|)$ Platz. Dafür kann man ein bestimmtes Kind in konstanter Zeit finden.
- Mit perfektem Hashing ist es theoretisch möglich, $\mathcal{O}(c_v)$ Platz und $\mathcal{O}(1)$ Zugriffszeit zu bekommen.

Ist $|\Sigma| \in \mathcal{O}(1)$, werden alle diese Fälle äquivalent.

Übersicht. Wir konstruieren zu $s \in \Sigma^*$ den Suffixbaum von $s\$$ mit $|s\$| = n$. Der Algorithmus geht in n Phasen $0, 1, \ldots, n-1$ vor. In Phase i wird sichergestellt, dass das Suffix, das an Position i beginnt, im Baum repräsentiert ist. Der Algorithmus arbeitet online, das bedeutet, nach jeder Phase i haben wir den Suffixbaum des jeweiligen Präfixes $s[\ldots i]$ konstruiert. (Da ein solches Präfix nicht notwendigerweise mit einem Wächter endet, ist dies allerdings kein Suffixbaum im definierten Sinn, da nicht jedes Suffix durch ein Blatt repräsentiert ist!) Um einen Linearzeit-Algorithmus zu erhalten, darf jede Phase amortisiert nur konstante Zeit benötigen.

Ein „naives“ Einfügen eines Suffixes würde aber $\mathcal{O}(n)$ Zeit kosten; damit käme man insgesamt auf quadratische Zeit. Es sind also einige Tricks nötig. Diese nennen wir hier und beschreiben sie unten im Detail.

1. Automatische Verlängerung von Blattkanten: In Phase i endet der String aus Sicht des Algorithmus an Position i; das aktuell eingefügte Suffix besteht also nur aus einem Zeichen. Später verlängern sich entlang der Blattkanten alle bereits eingefügten Suffixe automatisch bis zum jeweiligen Stringende. Das funktioniert folgendermaßen: Die Kantenbeschriftungen werden durch ein Paar von Indizes repräsentiert. Dabei steht ein besonderes Ende-Zeichen E für „bis zum Ende des bis jetzt verarbeiteten Strings“. In Phase i wird E stets als i interpretiert. Alle Kanten, die zu Blättern des Baums führen, verlängern sich somit in jeder Phase automatisch um ein Zeichen (ohne dass explizit an jeder Kante ein Index erhöht wird).

2. Wissen über das aktuelle Suffix: Der Algorithmus verwaltet eine *aktive Position* im Suffixbaum. Die aktive Position nach Phase i entspricht dem längsten Suffix von $s[\ldots i]$, das auch Teilstring von $s[\ldots i-1]$ ist.

3. Suffixlinks: Um im Baum schnell von $node(ax)$ mit $a \in \Sigma$, $x \in \Sigma^*$ zu $node(x)$ springen zu können, werden zwischen diesen Knoten Verbindungen eingefügt; diese heißen *Suffixlinks*. Ein Suffixlink entspricht also dem Abschneiden des ersten Zeichens. Damit die aktive Position von Phase zu Phase schnell aktualisiert werden kann, sind die Suffixlinks essentiell.

4. Überspringen von Kanten: Bewegt man sich entlang von Kanten im Baum, deren Beschriftung man kennt, so müssen diese nicht Zeichen für Zeichen gelesen werden, sondern können ggf. in einem einzigen Schritt übersprungen werden.

Details. Zu Beginn von jeder Phase i stehen wir an der aktiven Position, die dem Ende von Phase $i-1$ entspricht (Definition s.o.). Zu Beginn von Phase 0 ist dies die Wurzel, denn der Baum besteht nur aus der Wurzel. Eine aktive Position kann stets wie folgt durch ein Tripel beschrieben werden: (Knoten, Buchstabe, Tiefe). Der Knoten besagt, in oder unter welchem Knoten die aktive Position liegt. Liegt die aktive Position nicht direkt in einem Knoten, gibt der Buchstabe an, auf welcher vom gegebenen Knoten ausgehenden Kante die Position liegt, und die Tiefe gibt die Anzahl der Zeichen auf dieser Kante an, die man überspringen muss, um zur aktiven Position zu kommen. Ist die aktive Position direkt ein Knoten, ist die Tiefe 0 und der Buchstabe egal.

Ausgehend von der aktiven Position wird geprüft, ob das Zeichen $s[i]$ von dort aus bereits im Baum gelesen werden kann: Ist die aktive Position ein Knoten, wird also geprüft, ob

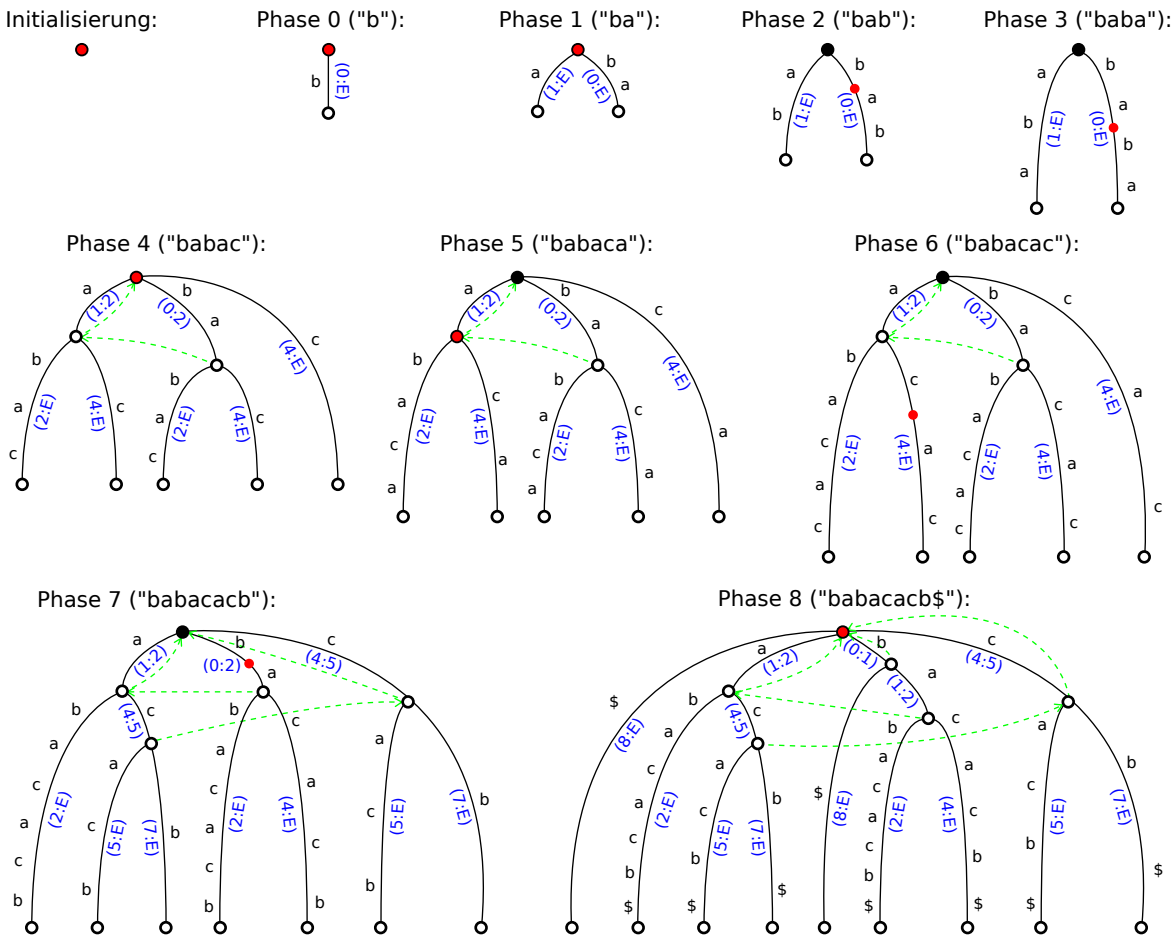


Abbildung 4.2: Schrittweise Konstruktion des Suffixbaumes von $s =$ `babacacb$` mittels Ukkonen's Algorithmus. Die jeweils aktive Position ist mit einem roten Kreis gekennzeichnet. Suffixlinks sind grün dargestellt. Die schwarzen Kantenbeschriftungen dienen lediglich der besseren Übersicht, gespeichert werden die blau eingezeichneten Indexpaare; dabei steht E für das Ende des Strings.

eine entsprechende ausgehende Kante exisitiert. Ist die aktive Position in einer Kante, wird geprüft, ob das nächste Zeichen der Kantenbeschriftung mit $s[i]$ übereinstimmt.

Ist dies der Fall, passiert in Phase i überhaupt nichts, außer dass die aktive Position um das gelesene Zeichen verschoben wird: Von einem Knoten gehen wir also in die entsprechende ausgehende Kante (und erreichen möglicherweise schon einen tieferen Knoten). In einer Kante gehen wir ein Zeichen weiter (und erreichen möglicherweise auch einen Knoten). Warum muss nicht mehr getan werden? Das Zeichen $s[i]$ kam bereits früher vor, sonst wäre es nicht entlang einer Kante lesbar gewesen; d.h., es existiert bereits als Suffix. Allerdings(!) kann es sich in den folgenden Phasen herausstellen, dass die (nun mindestens zwei) Suffixe, die mit $s[i]$ beginnen, unterschiedlich fortgesetzt werden. Spätestens wird dies der Fall sein, sobald der Wächter gelesen wird. Insbesondere haben wir im Moment noch kein Blatt i erzeugt. Das bedeutet, wir müssen uns in einer späteren Phase darum kümmern. Wenn wir also mehrere einfache Phasen dieser Art hintereinander haben, fügen wir eine Zeit lang gar keine neuen Blätter in den Baum ein und müssen dies zu einem späteren Zeitpunkt nachholen. Wir merken uns daher stets die Nummer ℓ des zuletzt eingefügten Blatts. Zu Beginn ist $\ell = -1$, da noch kein Blatt existiert.

Jetzt betrachten wir den Fall, dass sich das Zeichen $s[i]$ *nicht* von der aktiven Position aus lesen lässt. Die aktive Position zeigt uns an, inwieweit das Suffix, das an Position $\ell + 1$ beginnt (zu dem das Blatt also noch nicht existiert) bereits im Baum vorhanden ist. Da sich nun $s[i]$ nicht mehr an der aktiven Position im Baum lesen lässt, muss es an der aktiven Position (und möglicherweise an weiteren) eingefügt werden. Dabei wird nun das nächste Blatt erzeugt; wir erhöhen also ℓ um eins und erzeugen Blatt ℓ. Ist die aktive Position ein Knoten, bekommt dieser eine neue Blattkante zu Blatt ℓ mit Beschriftung (i, E). Liegt die aktive Position innerhalb einer Kante, müssen wir an dieser Stelle die Kante aufsplitten und einen neuen Knoten erzeugen. Dieser hat nun zwei Kinder: einerseits die Fortsetzung der „alten“ Kante, andererseits das neue Blatt ℓ mit Beschriftung (i, E). Es ist darauf zu achten, die Beschriftung der alten Kante korrekt aufzuteilen: Aus (a, b) wird oberhalb des neuen Knotens $(a, a + \text{Tiefe} - 1)$ und unterhalb $(a + \text{Tiefe}, b)$. „Tiefe“ war dabei die Anzahl der Zeichen auf der Kante oberhalb der aktiven Position.

Ein Beispiel soll dies verdeutlichen: Der String beginne mit `babac`. Phasen 0 und 1 erzeugen jeweils einfach eine von der Wurzel ausgehende Blattkante zu Blatt 0 und 1. Phasen 2 und 3 lesen einfach nur das folgende `b` und `a`; die aktive Position ist auf der Blattkante zu Blatt 0 in Tiefe 2 (`ba` wurde gelesen). In Phase 4 kann hier nun das `c` nicht mehr gelesen werden: An der aktiven Position (`ba`) entsteht ein neuer Knoten mit dem alten Kind 0 (Kante: `bac`) und dem neuen Kind 2 (Kante: `c`).

Das war aber noch nicht alles. Bisher ist folgendes passiert: Seit Phase ℓ konnten wir die Suffixe bereits im Baum finden und haben keine Blätter mehr eingefügt. Erst jetzt, in Phase i, mussten wir uns um das Blatt ℓ kümmern. Es ist jetzt ebenso möglich, dass wir auch für alle Blätter zwischen ℓ und i noch etwas tun müssen. Dazu müssen wir mit der aktiven Position vom Suffix $\ell < i$ zum Suffix $\ell + 1$ übergehen; dieses liegt normalerweise ganz woanders im Baum.

Im Beispiel `babac` gibt es in Phase 4 auch die Blätter 3 und 4 noch nicht. Das Suffix an Position 3 lautet momentan `ac`. Von der bisher aktiven Position `ba` (entsprach Suffix 2) müssen wir nun zur neuen aktiven Position `a` (für Suffix 3) übergehen (Abschneiden des ersten Zeichens) und dort erneut prüfen, ob sich von dort aus das `c` lesen lässt. Das ist nicht

der Fall, also muss hier ebenfalls ein Knoten eingefügt werden. Danach gehen wir, wiederum durch Abschneiden des ersten Zeichens, zu Suffix 4 über (die aktive Position entspricht dabei dem leeren String ε, ist also die Wurzel). Dort gibt es wiederum noch kein `c`, also fügen wir eine neue Blattkante an die Wurzel an. Die aktive Position verbleibt in der Wurzel. Eine Illustration ist in Abbildung 4.2, Phasen 0–4, zu sehen.

Suffixlinks. Die entscheidende Frage bei diesem Vorgehen ist: Wie kommt man schnell von der alten aktiven Position, die dem String ax entspricht (mit $a \in \Sigma$ und $x \in \Sigma^*$), zu der neuen aktiven Position, die dem String x entspricht, also durch Abschneiden des ersten Buchstaben erreicht wird? Natürlich kann man von der Wurzel aus einfach x ablesen, aber das kostet Zeit $\mathcal{O}(|x||)$ und wir haben nur konstante Zeit zur Verfügung.

Das entscheidende Hilfsmittel sind *Suffixlinks*. Gehört zu ax ein Knoten, dann gibt es mindestens zwei verschiedene Fortsetzungen von ax, etwa axb und axc. Das heißt aber auch, es gibt mindestens auch xb und xc als Suffixe, d.h., es gibt auch zu x einen Knoten im Baum. Wir ziehen nun eine Kante von $v := node(ax)$ zu $node(x)$; diese heißt Suffixlink von v. In einem Knoten, in dem ein solcher Suffixlink existiert, müssen wir also diesem nur folgen, um in konstanter Zeit zur nächsten aktiven Position zu kommen. Meistens sind wir jedoch an einer aktiven Position, in der wir gerade einen neuen Knoten erzeugt haben, in dem es noch gar keinen Suffixlink gibt! In dem Fall gehen wir zum nächsthöheren Knoten (das geht in konstanter Zeit; wir wissen ja, unter welchem Knoten die aktive Position lag); dieser hat einen Suffixlink, dem wir folgen können. Von dort aus müssen wir an die entsprechend tiefere Position „absteigen“. Es sei $ax = ayz$, dabei entspreche ay dem Knoten mit dem Suffixlink und z dem String an der Kante darunter. Wir haben also den Suffixlink $ay \to y$, und müssen von y aus wieder dem String z folgen. Auch dies könnte zu lange dauern, wenn wir dabei die Zeichen in z einzeln lesen. Wir wissen ja aber, dass der String z von y aus existiert; d.h., wir können direkt von y aus die richtige Kante anhand des ersten Zeichens wählen und in die entsprechende Tiefe gehen, ohne die Zeichen einzeln zu lesen. So erreichen wir die neue aktive Position in konstanter Zeit.

Ein Problem kann dabei auftauchen: Während auf dem Pfad $ax = ayz$ es entlang z keine Knoten gab, kann dies auf dem Pfad $x = yz$ durchaus der Fall sein. In dem Fall stellen wir fest, dass die von y ausgehende Kante nicht die ausreichende Länge $|z|$ hat und gelangen in einen Knoten, von dem aus wir wieder die richtige nächste Kante wählen und die verbleibende Länge absteigen, usw. Bei einer amortisierten Analyse zeigt sich, dass dies die Laufzeit insgesamt nicht verschlechtert, obwohl einzelne Phasen mehrere Schritte erfordern.

Wir dürfen nicht vergessen, sobald wir auch an der neuen Position entweder einen Knoten gefunden oder eingefügt haben, den Suffixlink vom vorher eingefügten Knoten zu ziehen, damit wir ihn in einer späteren Phase nutzen können.

Abschluss. Zum Abschluss des Algorithmus wird an allen Kanten der Platzhalter E durch $n - 1$ ersetzt und der Baum damit „eingefroren“. Weiterhin könnten die Suffixlinks nun gelöscht werden, um Platz zu sparen. Manche Algorithmen, die auf Suffixbäumen arbeiten, benötigen diese allerdings ebenfalls. Beispiele finden sich weiter unten. Die Knoten können ggf. (auf Kosten von zusätzlichem Speicherplatz) noch mit weiteren Informationen annotiert werden, etwa mit ihrer Stringtiefe oder der Anzahl der Blätter unterhalb.

4.10 Beispiel (Ukkonen-Algorithmus)**.** Abbildung 4.2 zeigt den Verlauf des Algorithmus für $s =$ `babacacb$`. ♡

Analyse. Die Laufzeit von Ukkonen's Algorithmus auf einem String der Länge n beträgt $\mathcal{O}(n)$, ist also linear. Um das einzusehen, muss man wieder amortisiert analysieren.

In manchen Phasen i (wenn $s[i]$ an der aktiven Position bereits gelesen werden kann) ist fast nichts zu tun, außer die aktive Position um das gelesene Zeichen zu verschieben. Allerdings müssen wir später für unser Nichtstun bezahlen und das Blatt i (und entsprechende Kanten) später erzeugen. Da wir insgesamt n Blätter und $\leq n-1$ Kanten erzeugen, dauert dies insgesamt $\mathcal{O}(n)$. Entscheidend ist nun die Gesamtzeit, die die Positionswechsel von $node(ax)$ nach $node(x)$ dauern. Sofern schon ein Suffixlink existiert, geht dies in konstanter Zeit pro Positionswechsel. Ist aber $node(ax)$ ein gerade neu eingefügter Knoten, existiert noch kein Suffixlink, und wir müssen zum nächsthöheren Knoten gehen (konstante Zeit), dem Suffixlink dort folgen (konstante Zeit) und wieder im Baum absteigen. Beim Absteigen müssen wir ggf. mehrere Knoten besuchen, so dass dies nicht immer in konstanter Zeit funktionieren kann! Amortisiert aber können wir nicht in jeder Phase beliebig tief absteigen und in jeder Phase maximal um Tiefe 1 aufsteigen; zu Beginn und am Ende sind wir an der Wurzel. Amortisiert dauern also alle Abstiege insgesamt $\mathcal{O}(n)$ Zeit.

4.4 Berechnung eines Suffix-Arrays in Linearzeit

Offensichtlich kann man mit Ukkonen's Suffixbaum-Konstruktionsalgorithmus mit einer Traversierung des Baums auch das Sufffixarray `pos` in Linearzeit berechnen.

Seit 2003 ist bekannt, dass das Suffixarray auch in Linearzeit berechnet werden kann, ohne eine Baumstruktur aufzubauen. Damit ist es möglich, ein Suffixarray mit deutlich weniger Speicheraufwand zu konstruieren als über den Umweg eines Suffixbaums. Hier stellen wir einen besonders eleganten Algorithmus aus dem Jahr 2009 vor: Induced Sorting von Ge Nong, Sen Zhang und Wai Hong Chan. Er ist nicht nur relativ einfach, sondern auch praxistauglich.

4.4.1 Grundstruktur des Algorithmus

Die Grundidee ist einfach: Kernpunkt ist ein Reduktionsschritt, bei dem die Länge des zu bearbeitenden Textes mindestens um die Hälfte reduziert wird. Es ergibt sich ein neuer Text, dessen Suffixarray rekursiv berechnet wird. Aus dem Ergebnis wird das Suffixarray des Originaltextes rekonstruiert.

Wir beginnen mit ein paar notwendigen Definitionen.

4.11 Definition (L-Position, S-Position)**.** Sei $s\$$ ein String mit Wächter der Länge n, so dass $s[n-1] = \$$. Sei $0 \leq p < n-1$ eine Textposition. Wir sagen, p ist eine *L-Position* (L für "larger"), wenn lexikographisch $s[p\ldots] > s[p+1\ldots]$, ansonsten eine *S-Position* (S für "smaller"). (Wegen des Wächters am Ende können zwei Suffixe nicht gleich sein.) Die Wächterposition $n-1$ wird als S-Position definiert.

```
            0.........1.........2.    |   LMS-String-Namen
Position p  01234567890123456789 01    |   $      $
Sequenz s   gccttaacattattacgccta$     |   aaca   A
Typ         LSSLLSSLSLLSLLSSLSSLLS     |   acgc   B
LMS?         *   *  *  *  *  *   *     |   atta   C
LMS-Strings  cctta  atta  acgc   $     |   atta   C
LMS-Strings      aaca  atta  ccta$     |   ccta$  D
Sequenz s'   E   A  C  C  B  D   $     |   cctta  E
```

Abbildung 4.3: Links: Beispiel einer DNA-Sequenz mit L-, S- und LMS-Positionen, sowie den LMS-Teilstrings. Rechts: Die LMS-Strings in sortierter Reihenfolge und mit Namen versehen. Die Namen erhalten die Ordnung der LMS-Teilstrings, und gleiche Teilstrings bekommen den gleichen Namen. Die Sequenz s' entsteht aus s, indem man die LMS-Teilstrings durch ihre jeweiligen Namen ersetzt.

Die Typinformation kann in Linearzeit durch einen Rückwärtsscan durch den Text berechnet und in einem Bitvektor `type` gespeichert werden (z.B. durch 1 für S, 0 für L):

- Setze $\texttt{type}[n-1] := \texttt{S}$.
- Für $p = n-2, \ldots, 0$:
  1. Falls $s[p] > s[p+1]$: setze $\texttt{type}[p] := \texttt{L}$
  2. Sonst, falls $s[p] < s[p+1]$: setze $\texttt{type}[p] := \texttt{S}$
  3. Sonst (es besteht Zeichengleichheit an Position i, also wird die Ordnung durch den Typ an der bereits berechneten Position $i+1$ festgelegt): setze $\texttt{type}[p] := \texttt{type}[p+1]$

4.12 Definition (LMS-Intervall, LMS-Teilstring)**.** Manche der S-Positionen sind dadurch ausgezeichnet, dass sich links von ihnen eine L-Postion befindet. Diese Positionen nennen wir *LMS-Positionen* ("leftmost S").

(Notiz: Die Position $n-1$ des Wächters ist stets eine LMS-Position. Die Feststellung, ob eine S-Position sogar eine LMS-Position ist, kann mit Hilfe des Bitvektors `type` in konstanter Zeit erfolgen.)

Ein Positionspaar $[i, j]$ mit $i < j$ heißt *LMS-Intervall* von s, wenn sowohl i als auch j eine LMS-Position ist und es keine LMS-Positionen zwischen i und j gibt.

Zu jedem LMS-Intervall $[i, j]$ gehört der entsprechende *LMS-Teilstring* $s[i \ldots j]$. Jede LMS-Position (außer der ersten und der Wächterposition) gehört zu zwei LMS-Intervallen und kommt damit in zwei LMS-Teilstrings vor, einmal als erstes Zeichen und einmal als letztes Zeichen.

Ein Beispiel ist in Abbildung 4.3 dargestellt. Der Algorithmus besteht nun aus zwei Ideen:

1. Die Suffixe, die an LMS-Positionen (wir nenen sie LMS-Suffixe) beginnen, werden sortiert. Dazu wird die lexikographische Reihenfolge der LMS-Teilstrings ermittelt. Wenn alle LMS-Teilstrings verschieden sind, ist damit die Reihenfolge der LMS-Suffixe schon festgelegt. Wenn es aber gleiche LMS-Teilstrings gibt (wie `atta` in Abbildung 4.3),

dann wird ein reduzierter Text gebildet: Jeder LMS-Teilstring wird durch ein einzelnes Symbol ersetzt. Die Symbole werden in lexikographischer Reihenfolge vergeben. Gleiche Teilstrings bekommen natürlich das gleiche Symbol zugewiesen. Auf dem reduzierten Text wird nun rekursiv das Suffixarray berechnet; aus diesem erhält man dann also die Reihenfolge der LMS-Teilstrings.

2. Aus der nun bekannten Reihenfolge der LMS-Suffixe wird die Ordnung der verbleibenden Suffixe berechnet (L-Positionen, sowie S-Positionen, die nicht LMS sind).

Wie die beiden Schritte effizient durchgeführt werden können, wird im Folgenden beschrieben. Wir beginnen mit ein paar einfachen Beobachtungen, die die Rekursion betreffen.

- Da per Definition der Wächter stets ein eigener LMS-Teilstring und lexikographisch kleiner als jeder andere Teilstring ist, gibt es ihn auch (wieder als LMS-String) in der reduzierten Sequenz. Man kann ihm also auf jeder Rekursionsebene den gleichen Namen `$` geben (vgl. Abbildung 4.3).
- Das Alphabet kann von einer Rekursionsebene zur nächsten wachsen: Im Beispiel der Abbildung 4.3 bestand das initiale Alphabet aus `a`, `c`, `g`, `t`, nach dem ersten Reduktionsschritt jedoch aus `A`–`E`. (In der Praxis werden als neue Namen fortlaufende ganze Zahlen vergeben; der Wächter wird dann durch die Null dargestellt. Die Verwendung von Großbuchstaben dient hier nur der besseren Erklärung.)
- Ist die Sequenzlänge vor einem Reduktionsschritt gleich n (inklusive des Wächters), beträgt sie danach höchstens $\lfloor n/2 \rfloor$ (inklusive des Wächters). Das liegt daran, dass zu einem LMS-String immer mindestens drei Positionen gehören (SLS) und sich zwei aufeinander folgende LMS-Strings in einer Position überlappen. Man kann also immer eiem LMS-String mindestens zwei "eigene" Positionen zuweisen. Eine Ausnahme bildet der Wächter; dafür kann aber die erste Position des Strings keine LMS-Position sein.

Wenn wir nun zeigen können, dass die beiden Schritte (1) Ermitteln der Sortierung der LMS-Teilstrings, (2) Einsortieren der nicht-LMS-Suffixe, wenn die Sortierung der LMS-Suffixe bekannt ist, jeweils in Linearzeit möglich sind (sagen wir in weniger als $c_1 n$ bzw. $c_2 n$ Operationen für jeweils Schritt (1) und (2)), dann ergibt sich als Gesamtlaufzeit $T(n)$ für eine Sequenz der Länge n mit $C := c_1 + c_2$:

$$\begin{aligned} T(1) &= \mathcal{O}(1); \\ T(n) &\leq c_1 n + T(n/2) + c_2 n \\ &= Cn + T(n/2) \\ &= Cn + Cn/2 + T(n/4) \\ &\leq Cn(1 + 1/2 + 1/4 + \dots) + T(1) \\ &= 2Cn + \mathcal{O}(1) \;=\; \mathcal{O}(n). \end{aligned}$$

Es bleibt also nur zu klären, wie man die beiden Schritte in Linearzeit ausführen kann!

4.4.2 Einsortieren der Nicht-LMS-Suffixe

Wir beginnen mit dem zweiten Schritt, da er einfacher ist. Die Ausgangslage ist, dass wir die LMS-Suffixe (d.h., ihre Positionen) entweder rekursiv oder durch direktes Ablesen sortiert und das reduzierte Suffixarray gebildet haben. Im Beispiel von Abbildung 4.3 sind

```
             0         1         2
Position p   01234567890123456789 01
Sequenz s    gccttaacattattacgccta$
Typ          LSSLLSSLSLLSLLSSLSSLLS
LMS?          *   *  *  *  *  *   *

p'           0  1  2  3  4  5  6
s'           E  A  C  C  B  D  $   reduzierter Text
p[p']        1  5  8 11 14 17 21   Originalpositionen

r'           0  1  2  3  4  5  6
pos'[r]      6  1  4  3  2  5  0   reduziertes Suffixarray
p[pos'[r]]  21  5 14 11  8 17  1   Suffixarray der LMS-Positionen

rank r    0| 1  2  3  4  5  6| 7  8  9 10 11 12|13 14|15 16 17 18 19 20 21|
bucket    $| a  a  a  a  a  a| c  c  c  c  c  c| g  g| t  t  t  t  t  t  t|
pos/(0)  21| .  .  5 14 11  8| .  .  .  . 17  1| .  .| .  .  .  .  .  .  .|

pos/(a)  21|20  .  5 14 11  8| .  .  .  . 17  1| .  .| .  .  .  .  .  .  .|
pos/(a)  21|20  .  5 14 11  8| .  .  .  . 17  1| .  .|19  .  .  .  .  .  .|
pos/(a)  21|20  .  5 14 11  8| .  .  .  . 17  1| .  .|19  4  .  .  .  .  .|
pos/(a)  21|20  .  5 14 11  8| .  .  .  . 17  1| .  .|19  4 13  .  .  .  .|
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1| .  .|19  4 13  .  .  .  .|
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1| .  .|19  4 13 10  .  .  .|
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1|16  .|19  4 13 10  .  .  .|
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1|16  0|19  4 13 10  .  .  .|
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1|16  0|19  4 13 10  3  .  .|
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1|16  0|19  4 13 10  3 12  .|
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1|16  0|19  4 13 10  3 12  9|
pos/(a)   .|20  .  .  .  .  .| 7  .  .  .  .  .|16  0|19  4 13 10  3 12  9|

pos/(b)   .|20  .  .  .  .  8| 7  .  .  .  .  .|16  0|19  4 13 10  3 12  9|
pos/(b)   .|20  .  .  . 11  8| 7  .  .  .  .  .|16  0|19  4 13 10  3 12  9|
pos/(b)   .|20  .  .  . 11  8| 7  .  .  .  .  2|16  0|19  4 13 10  3 12  9|
pos/(b)   .|20  .  .  . 11  8| 7  .  .  . 18  2|16  0|19  4 13 10  3 12  9|
pos/(b)   .|20  .  .  . 11  8| 7  .  . 15 18  2|16  0|19  4 13 10  3 12  9|
pos/(b)   .|20  .  .  . 11  8| 7  .  1 15 18  2|16  0|19  4 13 10  3 12  9|
pos/(b)   .|20  .  .  . 11  8| 7 17  1 15 18  2|16  0|19  4 13 10  3 12  9|
pos/(b)   .|20  .  . 14 11  8| 7 17  1 15 18  2|16  0|19  4 13 10  3 12  9|
pos/(b)   .|20  .  6 14 11  8| 7 17  1 15 18  2|16  0|19  4 13 10  3 12  9|
pos/(b)   .|20  5  6 14 11  8| 7 17  1 15 18  2|16  0|19  4 13 10  3 12  9|
```

Abbildung 4.4: Oben ist der Text mit markierten LMS-Positionen zu sehen (vgl. Abbildung 4.3, darunter der reduzierte Text und die Abfolge der LMS-Positionen. Wenn aus dem reduzierten Text das Suffixarray gebildet wird, entsteht daraus das Suffixarray der LMS-Positionen des Orignialtexts. Es folgt die Konstruktion des Suffixarrays `pos` in zwei Schritten (a): Sortieren der L-Positionen bei gegebenen sortierten LMS-Positionen und (b): Sortieren der S-Positionen bei gegebenen L-Positionen.

dies die Positionen $(1, 5, 8, 11, 14, 17, 21)$, die lexikographisch sortiert das Teil-Suffixarray $(21, 5, 14, 8, 11, 17, 1)$ bilden; dieses liegt uns vor.

Entscheidend ist nun, dass man die L-Positionen und die verbleibenden S-Positionen in der richtigen Reihenfolge in ihre "Buckets" einsortiert.

4.13 Definition (Bucket)**.** Ein maximales Intervall des Suffixarrays `pos`, in dem die referenzierten Suffixe mit dem selben Buchstaben beginnen, heißt Bucket (Eimer). Es gibt genau so viele Buckets wie Buchstaben im Alphabet, plus einen für den Wächter.

Die Größe der Buckets berechnet man einfach durch Zählen der Buchstaben im Originaltext. (Wir setzen nicht voraus, dass das Alphabet eine konstante Größe hat, sondern erlauben eine Größe von $\mathcal{O}(n)$; wir können dabei annehmen, dass das Alphabet höchstens aus den Zahlen von 0 bis $n-1$ besteht, wobei die 0 die Rolle des Wächters übernimmt. Dies erlaubt das Speichern der Bucket-Größen in einem Feld der Größe n.)

Wichtig ist nun folgende Beobachtung.

4.14 Lemma. In einem Bucket des Suffixarrays stehen die L-Positionen vor den S-Positionen.

Beweis. Seien $a < b < c$ Elemente des Alphabets. Betrachten wir den Bucket, in dem alle Positionen stehen, deren Suffixe mit b beginnen, eine L-Position q und eine S-Position p. Definitionsgemäß bedeutet L-Position, dass das Suffix an Position q größer ist als das an $q+1$, d.h., es ist von der Form b^+a (ein oder mehrere bs, gefolgt von einem kleineren Buchstaben). S-Position bedeutet hingegen, dass das Suffix an Position p von der Form b^+c ist (es folgt irgendwann ein größerer Buchstabe). Daher sind alle L-Positionen im b-Bucket lexikographisch kleiner als die S-Positionen im selben Bucket. □

Die verblüffende Erkenntnis besteht nun darin, dass wir (a) die bereits sortierten LMS-Positionen (eine Teilmenge der S-Positionen) nutzen können, um die L-Positionen korrekt zu sortieren und dann (b) die sortierten L-Positionen nutzen, um alle S-Positionen zu sortieren.

Schritt (0): Zur Initialisierung von `pos` schreiben wir eine Markierung, die "unbekannt" bedeutet, an jede Stelle. Wir markieren Beginn und Ende jedes Buckets durch geeignete Zeiger. Nun schreiben wir die Positionen des Teil-Suffixarrays der LMS-Positionen an das Ende ihrer jeweiligen Buckets (siehe Abbildung 4.4, `pos/(0)`).

Schritt (a): Wir durchlaufen das Array `pos` von links nach rechts mit der Indexvariablen r. Ist `pos`$[r]$ nicht definiert, überspringen wir Index r. Ansonsten betrachten wir die Vorgängerposition zu `pos`$[r]$, also `pos`$[r]-1$. Ist dies eine L-Position, schreiben wir sie an die vorderste freie Stelle in ihren Bucket. (Ist dies eine S-Position, tun wir nichts und überspringen Index r.) Der Ablauf ist in Abbildung 4.4 in den Zeilen `pos/(a)` dargestellt. Beachte, dass auch initial unbekannte Werte, die in den ersten Iterationen bekannt werden, in späteren Schritten dazu dienen, weitere Werte einzusortieren. Es ist zunächst nicht offensichtlich, dass jetzt alle L-Positionen im Array `pos` vorhanden sind und an korrekter Stelle stehen. Wir beweisen dies sogleich.

Wir könnten nun die existierenden S-Positionen in `pos` (das sind die LMS-Positionen) auf "unbekannt" setzen, was wir der Illustration halber in Abbildung 4.4 am Ende von Schritt (a)

tun, da wir sie nicht mehr benötigen und ihre Reihenfolge aus den L-Positionen neu berechnen. Dies ist aber nicht explizit notwendig, da wir die entsprechenden Inhalte in Schritt (b) einfach überschreiben.

Schritt (b): Wir durchlaufen das Array `pos` von *rechts nach links* mit der Indexvariablen r. Ist `pos`$[r]$ nicht definiert, überspringen wir Index r. (Dieser Fall kommt allerdings nicht vor!) Ansonsten betrachten wir die Vorgängerposition zu `pos`$[r]$, also `pos`$[r] - 1$. Ist dies eine S-Position, schreiben wir sie an die *hinterste* freie Stelle in ihren Bucket (die Buckets enthalten zu Beginn von Schritt (b) keine LMS-Positionen mehr!). Der Ablauf ist in Abbildung 4.4 in den Zeilen `pos/(b)` dargestellt.

Wir sehen, dass damit alle S-Positionen (bis auf den Wächter, dessen Position aber sowieso bekannt ist) einsortiert wurden. Damit ist das Suffixarray vollständig.

Es ist wiederum nicht offensichtlich, dass alle S-Positionen im Array `pos` vorhanden sind und an korrekter Stelle stehen. Wir kommen nun zu den Beweisen der Korrektheit von Schritt (a) und (b).

4.15 Lemma (Korrektheit von Schritt (a))**.** Wenn zu Beginn alle LMS-Positionen jedes Buckets in korrekter relativer Reihnefolge im Suffixarray stehen, dann sind nach Schritt (a) alle L-Positionen des Textes im Suffixarray vorhanden und stehen an ihrer korrekten Stelle.

Vor dem formalen Beweis wollen wir dies illustrieren. Ein Eintrag einer L-Position in `pos` kann nur durch eine LMS-Position oder durch eine andere L-Position verursacht werden. In Abbildung 4.5 ist für das Beispiel detailierter dargestellt, welche vorhandenen Einträge für jeweils neue Einträge verantwortlich sind.

Beweis. Wir halten eine offensichtliche Beobachtung fest: Steht Textposition p an Index r von `pos` und ist $p - 1$ eine L-Position, dann steht $p - 1$ an einem Index $r' > r$. (Das folgt sofort aus der Definition einer L-Position.)

Damit ist sichergestellt, dass jede L-Position $p - 1$ von einer geeigneten LMS- oder L-Position p aus gesetzt wird.

Wir zeigen nun: Nach jeder Iteration des Algorithmus von Schritt (a) stehen die eingetragenen Positionen in korrekter lexikographischer Reihenfolge.

Zu Beginn ist dies gewiss richtig, da nur die LMS-Positionen eingetragen sind; diese wurden als korrekt lexikographisch sortiert vorausgesetzt.

Der Beweis erfolgt nun mittels Induktion über die Iterationsschritte von Schritt (a) per Widerspruch. Angenommen, es gibt einen Schritt, bei dem zum erstem Mal Positionen in falscher Reihenfolge stehen. Es gibt dann Indizes $r_1 < r_2$ mit eingetragenen Positionen p_1, p_2, so dass aber das an p_1 beginnende Suffix lexikopgraphisch größer ist als das an p_2 beginnende.

Die Situation kann höchstens dann auftreten, wenn beide Positionen p_1 und p_2 L-Positionen im gleiche Bucket sind. (Denn: Positionen in verschiedenen Buckets sind automatisch korrekt sortiert, L-Positionen werden korrekt vor S-Positionen im selben Bucket eingetragen). Die Suffixe an den Positionen p_1, p_2 beginnen also mit dem gleichen Buchstaben, etwa bx_1, bx_2, mit lexikographisch $x_1 > x_2$ an Positionen $p_1 + 1$, $p_2 + 1$. Diese beiden Positionen müssen aber schon weiter links an den Indizes r'_1, r'_2 eingetragen gewesen sein, sonst hätte man ja die Einträge zu p_1 und p_2 nicht vornehmen können. Im Fall $r'_1 < r'_2$ wäre der Fehler also bereits

```
             0         1         2
Position p   01234567890123456789 01
Sequenz s    gccttaacattattacgccta $
Typ          LSSLLSSLSLLSLLSSLSSLLS
LMS?          *   *  *  *  *  *   *

rank r    0| 1  2  3  4  5  6| 7  8  9 10 11 12|13 14|15 16 17 18 19 20 21|
bucket    $| a  a  a  a  a  a| c  c  c  c  c  c| g  g| t  t  t  t  t  t  t|
pos/(0)  21| .  .  5 14 11  8| .  .  .  . 17  1| .  .| .  .  .  .  .  .  .|
         ^S|vL               |                  |     |                      |
pos/(a)  21|20  .  5 14 11  8| .  .  .  . 17  1| .  .| .  .  .  .  .  .  .|
           |^L                |                  |     |vL                    |
pos/(a)  21|20  .  5 14 11  8| .  .  .  . 17  1| .  .|19  .  .  .  .  .  .|
           |      ^S          |                  |     |   vL                 |
pos/(a)  21|20  .  5 14 11  8| .  .  .  . 17  1| .  .|19  4  .  .  .  .  .|
           |         ^S       |                  |     |      vL              |
pos/(a)  21|20  .  5 14 11  8| .  .  .  . 17  1| .  .|19  4 13  .  .  .  .|
           |            ^S    |                  |     |         vL           |
pos/(a)  21|20  .  5 14 11  8| .  .  .  . 17  1| .  .|19  4 13 10  .  .  .|
           |               ^S|vL                |     |                      |
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1| .  .|19  4 13 10  .  .  .|
           |                  |--                ^S   |vL   |                      |
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1|16  .|19  4 13 10  .  .  .|
           |                  |                  ^S|   vL|                      |
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1|16  0|19  4 13 10  .  .  .|
           |                  |                  |-- --|-- ^L       vL        |
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1|16  0|19  4 13 10  3  .  .|
           |                  |                  |     |      ^L       vL     |
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1|16  0|19  4 13 10  3 12  .|
           |                  |                  |     |         ^L       vL|
pos/(a)  21|20  .  5 14 11  8| 7  .  .  . 17  1|16  0|19  4 13 10  3 12  9|
           |                  |                  |     |           -- -- --|
```

Abbildung 4.5: Ausführung von Schritt (a) im Detail. Mit einem Zeiger (`^`) wird der Text gescannt. Dabei trifft man auf die vorsortierten LMS-Positionen (`^S`), aber auch auf in vorherigen Iterationen eingetragene L-Positionen (`^L`). Ist die Vorgängerposition eine L-Position, wird diese an die erste freie Position in ihren Bucket eingetragen (`vL`). Ist die Vorgängerposition keine L-Position, wird die aktuelle Position übersprungen (`--`). Beachtenswert: Einzutragende Positionen treten nur stets von der aktuellen Position auf; am Ende sind alle L-Positionen eingetragen.

für die Suffixe x_1 und x_2 an Positionen $p_1 + 1$ und $p_2 + 1$ aufgetreten, was im Widerspruch zur Voraussetzung steht, dass der Fehler für p_1 und p_2 zum ersten Mal auftritt. Im Fall $r'_2 < r'_1$ wäre aber das lexikographisch kleine Suffix bei p_2 durch den von links nach rechts scannenden Algorithmus auch links von p_1 in den b-Bucket einsortiert worden und der Fehler wäre nicht aufgetreten.

Die hypothetisch angenommene Situation, dass es einen Schritt gibt, bei dem zum ersten Mal Positionen in falscher Reihenfolge stehen, kann also nicht auftreten, und der Beweis ist erbracht, dass am Ende des Algorithmus alle L-Positionen vorhanden, insgesamt an den richtigen Indizes (zu Beginn jedes Buckets) und relativ zueinander in der richtigen Reihenfolge stehen. Daher steht jede L-Position an korrekter Position im Suffixarray. □

4.16 Lemma (Korrektheit von Schritt (b))**.** Wenn nach Schritt (a) alle L-Positionen im Suffixarray vorhanden und an ihrer korrekten Stelle stehen, dann sind nach Schritt (b) alle Positionen des Textes im Suffixarray vorhanden und an ihrer korrekten Stelle.

Beweis. Der Beweis läuft analog "rückwärts" zu Schritt (a) mit Hilfe folgender offensichtlicher Beobachtung, die aus der Definition einer S-Position folgt: Steht Textposition p an Index r von `pos` und ist $p-1$ eine S-Position, dann steht $p-1$ an einem Index $r' < r$. Damit ist sichergestellt, dass in Schritt (b) beim Scan von rechts nach links jede S-Position $p-1$ von einer geeigneten L-Position oder bereits gesetzten S-Position p aus gesetzt wird. □

4.4.3 Sortieren und Benennen der LMS-Teilstrings

Wir stellen die Frage, was passiert, wenn man den Algorithmus aus dem vorigen Abschnitt zu Beginn mit "falsch" sortierten LMS-Suffixen initialisiert (da man beispielsweise die korrekte Sortierung noch nicht kennt).

Konkret: Wir schreiben im Initialisierungsschritt die Positionen der LMS-Suffixe in einer beliebigen Reihenfolge (z.B. aufsteigend nach Position im Text) an das Ende ihrer entsprechenden Buckets und lassen dann die Schritte (a) und (b) unverändert ablaufen.

Natürlich können wir nicht erwarten, dass nun das korrekte Suffixarray erzeugt wird. Überraschend ist jedoch, dass nach den Schritten (a) und (b) alle LMS-Teilstrings korrekt sortiert sind. (Achtung: Nicht die LMS-Suffixe sind korrekt sortiert; wenn es zwei gleiche LMS-Teilstrings gibt, können diese eine beliebige (falsche) Reihenfolge haben!).

Wir erinnern an die Definitionen der LMS-Positionen (Definition 4.11 und der LMS-Teilstrings (Definition 4.12). Bei der folgenden Diskussion betrachten wir explizit nicht den Wächter. Ein LMS-Teilstring hat die folgende Struktur: Er endet mit einer S-Position (sogar eine LMS-Position). Davor steht eine beliebige Anzahl (aber mindestens eine) L-Positionen; davor steht wiederum eine beliebige Anzahl (aber mindestens eine) S-Positionen (als regulärer Ausdruck: S^+L^+S). Wir nennen ein Suffix eines LMS-Teilstrings, das (einige oder alle) L-Positionen und die finale S-Position enthält, ein *LS-Suffix* eines LMS-Teilstrings. Analog nennen wir ein Suffix eines LMS-Teilstrings, das (einige oder alle) der vorderen S-Positionen, alle L-Positionen und die finale S-Position enthält, ein *SLS-Suffix* eines LMS-Teilstrings.

Nach der Initialisierung sind zumindest die Zeichen an den LMS-Positionen (als Strings der Länge 1) korrekt relativ zueinander sortiert: Gleiche Strings stehen nebeneinander im gleichen Bucket. Eine Betrachtung des Beweises von Lemma **??** zeigt nun, dass nach Schritt (a)

alle LS-Suffixe der LMS-Strings korrekt relativ zueinander sortiert sind. Analog sind nach Schritt (b) die SLS-Suffixe der LMS-Teilstrings korrekt relativ zueinander sortiert; damit also auch die LMS-Teilstrings!

Man muss die Beweise nur so umschreiben, dass statt von Textsuffixen von LMS-Teilstrings des Texts die Rede ist und die Möglichkeit in Betracht ziehen, dass Gleichheit auftreten kann – die Vergleiche enden mit der finalen LMS-Position jedes LMS-Teilstrings.

Zur Übung wird empfohlen, den Prozess analog zu Abbildung 4.4 mit beliebigen initialen Permutationen der LMS-Positionen innerhalb jedes Buckets durchzuspielen und zu verifizieren, dass am Ende die relative Sortierung der LMS-Positionen stets korrekt ist.

Sind die LMS-Teilstrings sortiert, ist das Benennen (mit aufsteigenden ganzen Zahlen) einfach: Man durchläuft das Array und prüft bei jeder LMS-Position, ob der dort beginnende LMS-Teilstring der gleiche ist wie der vorherige LMS-Teilstring. Wenn ja, vergibt man die gleiche Zahl wie zuletzt, ansonsten die nächsthöhere.

4.5 Berechnung des `lcp`-Arrays in Linearzeit

Ebenso wie das Array `pos` kann man das Array `lcp` direkt in Linearzeit aus einem Suffixbaum erhalten, wenn man ihn erstellt hat. Dazu durchläuft man den Baum rekursiv mit einer Tiefensuche, wobei man die Kinder jedes inneren Knoten in lexikographischer Reihenfolge abarbeitet. Das Suffixarray `pos` erhält man als Abfolge der Blätter, wenn jedes Blatt mit der Startposition des entsprechenden Suffix annotiert ist. Der `lcp`-Wert zwischen zwei Blättern ist die Stringtiefe des höchsten Knoten, den man auf dem Pfad zwischen zwei benachbarten Blättern besucht.

Der Sinn des Suffixarrays ist jedoch (unter anderem), die speicheraufwändige Konstruktion des Suffixbaums zu vermeiden. Wir zeigen hier, wie man `lcp` in Linearzeit aus `pos` gewinnen kann.

Das `lcp`-Array kann prinzipiell einfach gemäß der Definition konstruiert werden: Um `lcp`$[r]$ zu berechnen, testet man, wie lang das längste gemeinsame Präfix der Suffixe ist, die an den Positionen `pos`$[r-1]$ und `pos`$[r]$ beginnen. So ein Test dauert $O(n)$ Zeit, wenn n die Stringlänge ist; insgesamt benötigt man also $O(n^2)$ Zeit.

Es geht jedoch in Linearzeit, wenn man `lcp` nicht in aufsteigender r-Reihenfolge berechnet, sondern (gewissermaßen) in aufsteigender Reihenfolge der Suffixstartpositionen p im Text.

```
1  def lcp_linear(pos, text):
2      """Berechnet zu Suffixarray pos das lcp-Array"""
3      n = len(pos)
4      lcp = [-1] * n
5      rank = [0] * n
6      for r in range(n):
7          rank[pos[r]] = r
8      lp = 0  # aktuelle Praefixlaenge
9      for p in range(n-1):
10         r = rank[p]
11         pp = pos[r-1]
12         while text[p+lp]==text[pp+lp]:
```

```
13              lp += 1
14          lcp[r] = lp
15          lp = max(lp-1, 0)
16      return lcp
```

Dazu definieren wir das zu `pos` inverse Array `rank`: Es ist $\texttt{rank}[p] = r$ genau dann, wenn $\texttt{pos}[r] = p$. $\texttt{rank}[p]$ ist der lexikographische Rang des Suffix an Position p. Das ist wohldefiniert, da `pos` eine Permutation (also eine bijektive Abbildung) und damit invertierbar ist. Es lässt sich offensichtlich aus `pos` in Linearzeit berechnen (Zeilen 6–7).

Um nun `lcp` zu berechnen, beginnen wir mit $p = 0$ und berechnen dazu $r = \texttt{rank}[p]$. Für dieses r berechnen wir wie oben beschrieben $\ell := \texttt{lcp}[r]$, indem wir die Suffixe an den Positionen p und $\texttt{pos}[r-1]$ vergleichen. Das kostet einmalig $O(n)$ Zeit. Nun gehen wir zu $p^+ := p + 1$ mit $r^+ := \texttt{rank}[p^+]$ über.

Was wissen wir über das längste gemeinsame Präfix der Suffixe an p^+ und $\texttt{pos}[r^+ - 1]$? Wenn $\texttt{pos}[r^+ - 1] = \texttt{pos}[r-1]+1$ ist, dann ist dessen Länge genau $\ell - 1$, denn wir haben gegenüber dem letzten Vergleich einfach das erste Zeichen abgeschnitten. Hat aber $\texttt{pos}[r^+ - 1]$ einen anderen Wert, dann kann das Präfix zumindest nicht kürzer sein. In jedem Fall gilt also

$$\ell^+ := \texttt{lcp}[r^+] \geq \max\{\ell - 1, 0\},$$

so dass die ersten $\ell - 1$ Zeichen nicht verglichen werden müssen.

Eine amortisierte Analyse zeigt, dass dies insgesamt einen $O(n)$-Algorithmus ergibt.

4.6 Anwendungen

4.6.1 Exaktes Pattern Matching

Wir betrachten das klassische Problem, ein Muster $P \in \Sigma^m$ in einem Text $T \in \Sigma^n$ zu finden. Wir gehen davon aus, dass wir zu T bereits einen Volltextindex konstruiert haben. Das Muster komme z mal in T vor.

Lösung mit einem Suffixbaum. Wir betrachten wieder die drei Fragestellungen des einfachen Pattern Matching.

Kommt P überhaupt vor? Wir untersuchen, ob es einen mit P beschrifteten Pfad ausgehend von der Wurzel des Suffixbaums gibt. Das Lesen eines Zeichens kostet konstante Zeit, so dass wir in $\mathcal{O}(m)$ Zeit die Existenz von P testen können. (Die Zeit, die wir in einem Knoten benötigen, um die korrekte ausgehende Kante zu finden, hängt von der Alphabetgröße und der Datenstruktur für die Kinder ab, aber das Alphabet ist konstant.) Kommt P nicht vor, kennen wir zumindest das längste Präfix von P, das vorkommt.

Wie oft kommt P vor? Wenn P vorkommt, durchmustern wir den Suffixbaum unterhalb der Position, die P entspricht, und zählen die Blätter. Dies benötigt insgesamt $\mathcal{O}(m+z)$ Zeit. Ist bereits jeder innere Knoten mit der Anzahl der Blätter unterhalb annotiert, können wir die Antwort direkt anhand der Annotation des Knoten, der P entspricht oder als nächster unterhalb von P liegt, ablesen und benötigen insgesamt nur $\mathcal{O}(m)$ Zeit. Dies kostet allerdings pro Knoten konstant mehr Speicherplatz, um die Annotationen zu verwalten.

Wo kommt P vor? Da wir hier in jedem Fall die Blätter (und zugehörigen Startpositionen der Suffixe) aufzählen müssen, benötigen wir in jedem Fall $\mathcal{O}(m + z)$ Zeit.

Lösung mit einem Suffixarray. Die entscheidende Beobachtung ist: Pattern P entspricht einem d-Intervall $[L, R]$ im Suffixarray mit $d \geq m$. Genauer: Sei $T \in \Sigma^n$ der Text und $S_T(r) := T[\texttt{pos}[r] \ldots]$ das lexikographisch r-t kleinste Suffix von T. Die Relationen $\leq_m, =_m, \geq_m$ zwischen zwei Strings bezeichnen lexikographisch kleiner gleich, gleich, größer gleich unter Berücksichtigung ausschließlich der ersten m Zeichen. Dann sind die Intervallgrenzen L und R gegeben als

$$L := \min\big[\,\{\, r \mid P \geq_m S_T(r) \,\} \cup \{\, n \,\}\,\big],$$
$$R := \max\big[\,\{\, r \mid P \leq_m S_T(r) \,\} \cup \{\, -1 \,\}\,\big].$$

P kommt genau dann in T vor, wenn $R \geq L$.

Wir betrachten nun wieder die drei Fragestellungen.

Kommt P überhaupt vor? Die Intervallgrenzen L und R finden wir mit je einer binären Suche. Eine binäre Suche dauert $\log_2 n$ Schritte. In jedem Schritt müssen bis zu m Zeichen verglichen werden: Die Laufzeit ist $\mathcal{O}(m \log n)$.

Wie oft kommt P vor? Die Beantwortung dieser Frage kostet nicht mehr Zeit als der Test, ob P überhaupt vorkommt, denn die Anzahl der Vorkommen ist $z = R - L + 1$.

Wo kommt P vor? Sobald das Intervall $[L, R]$ bekannt ist, lassen sich die Startpositionen von P aufzählen; dies kostet zusätzlich $\mathcal{O}(z)$ Zeit: $\texttt{pos}[L], \texttt{pos}[L+1] \ldots, \texttt{pos}[R]$, oder in Python als Generatorausdruck: `(pos[r] for r in range(L, R+1))`.

Man kann auch mit Suffixarrays das Entscheidungsproblem in $\mathcal{O}(m)$ Zeit lösen, wenn man ein zusätzliches Array anlegt, dass (auf geschickte Weise) die Eltern-Kind-Beziehungen aller Intervalle des `lcp`-Intervallbaums abspeichert; siehe **?**.

4.6.2 Längster wiederholter Teilstring eines Strings

Lösung mit einem Suffixbaum. Sei $s \in \Sigma^*$ gegeben. Der Suffixbaum zu $s\$$ buchstabiert nach Definition alle Teilstrings von $s\$$. Ein Teilstring t von s kommt genau dann mehrfach in s vor, wenn man sich nach dem Ablesen von t von der Wurzel aus in einem inneren Knoten befindet oder ein innerer Knoten darunter liegt. Wenn wir einen längsten wiederholten String suchen, genügt es also, einen inneren Knoten mit größter Stringtiefe zu finden. Dies kann man mit einer einfachen Traversierung des Baums leicht erreichen.

Lösung mit einem Suffixarray. Die Stringtiefe der inneren Knoten wird durch die `lcp`-Werte dargestellt. Wir müssen daher nur den maximalen `lcp`-Wert bestimmen; sei r^* so dass $\ell^* := \texttt{lcp}[r^*]$ maximal ist. Dann stimmen die Länge ℓ^*-Präfixe der Suffixe an Positionen $\texttt{pos}[r^* - 1]$ und $\texttt{pos}[r^*]$ überein und sind ein längster wiederholter Teilstring.

4.6.3 Kürzester eindeutiger Teilstring eines Strings

Man findet den kürzesten eindeutigen (nur einmal vorkommenden) nichtleeren Teilstring von s (ohne den Wächter \$; der ist immer eindeutig und hat Länge 1) wie folgt.

Lösung mit einem Suffixbaum. Eindeutig sind genau die vom Suffixbaum buchstabierten Strings, die auf einer Blattkante enden. Gesucht ist also ein innerer Knoten v minimaler Stringlänge, dessen ausgehende Kanten eine Blattkante e enthält, welche nicht nur mit einem \$ beschriftet ist. Der bis v buchstabierte Substring u' ist Präfix des gesuchten eindeutigen Teilstrings u. Alle Teilstrings, die sich nun durch Konkatenation von u' mit den Zeichen aus der Blattkante erstellen lassen, sind eindeutig. Konkateniert man u' mit dem ersten Zeichen der Blattkante, erhält man einen eindeutigen String minimaler Länge.

Lösung mit einem Suffixarray. Jeder Index r des Suffixarrays entspricht genau einem Blatt, nämlich $\texttt{pos}[r]$. Welche Stringtiefe hat der innere Knoten direkt über diesem Blatt? Wir betrachten die beiden `lcp`-Werte, die das Blatt $\texttt{pos}[r]$ betreffen; das sind $\texttt{lcp}[r]$ und $\texttt{lcp}[r+1]$. Das Blatt hängt an dem tieferen dieser beiden. Eindeutig ist also der zu diesem Blatt gehörende String, wenn er eine Länge von $M_r := 1 + \max\{\texttt{lcp}[r], \texttt{lcp}[r+1]\}$ hat. Allerdings müssen wir solche Strings ausschließen, die mit dem Wächter \$ enden, wenn also $\texttt{pos}[r] + M_r = n$ ist. Wir suchen also $r^* = \operatorname{argmin}\{\, M_r \mid \texttt{pos}[r] + M_r < n \,\}$; die ist leicht mit einem Durchlauf des `lcp`-Arrays zu finden. Dann ist das Präfix der Länge M_{r^*} des Suffixes an Position $\texttt{pos}[r^*]$ eindeutig und minimal lang.

4.6.4 Längster gemeinsamer Teilstring zweier Strings

Wir zeigen, dass sich die Länge $\operatorname{lcf}(s,t)$ des längsten gemeinsamen Teilstrings zweier Strings s, t in Zeit $\mathcal{O}(|s|+|t|)$ berechnen lässt. Dies ist bemerkenswert, da man im Jahr 1970 (vor der Erfindung von Suffixbäumen) noch davon ausging, dass hierfür keine Lösung in Linearzeit möglich sei. Es gibt verschiedene Möglichkeiten.

Lösung mit Suffixbaum. Die erste Lösung benutzt einen verallgemeinerten Suffixbaum von s und t, also den Baum zu $s\#t\$$. Dieser wird beispielsweise mit Ukkonen's Algorithmus in Zeit $\mathcal{O}(|s|+|t|)$ konstruiert. Man kann nun jedes Blatt entweder s oder t zuordnen, indem man seine Position mit der von # vergleicht. Ein Blatt wird mit 1 (binär 01) annotiert, wenn es zu s gehört und mit 2 (binär 10), wenn es zu t gehört. Mit Hilfe einer bottum-up-Traversierung kann man nun jeden inneren Knoten annotieren: Die Annotation eines inneren Knoten ist das logische Oder der Annotation seiner Kinder. Ein innerer Knoten ist also genau dann mit 1 annotiert, wenn darunter nur Blätter aus s sind, genau dann mit 2, wenn darunter nur Blätter aus t sind, und genau dann mit 3 (binär 11), wenn darunter Blätter aus beiden

Strings sind. Sind nun alle Knoten annotiert, iterieren wir erneut über alle inneren Knoten (dies kann auch parallel zur Annotation geschehen) und finden einen Knoten, der unter denen mit der Annotation 3 die höchste Stringtiefe aufweist. Dieser Knoten entspricht einem gemeinsamen Teilstring von s und t, und gemäß der Auswahl gibt es keinen längeren.

Lösung mit Suffixarray. Die zweite Lösung verwendet ein Suffixarray aus `pos` und `lcp` von $s\#t\$$. Dieses wird nur einmal von links nach rechts durchsucht; ein Index r läuft von 1 bis $n-1$. Zunächst prüfen wir, ob die Startpositionen $\texttt{pos}[r-1]$ und $\texttt{pos}[r]$ zu verschiedenen Strings (s und t) gehören. Ist dies der Fall, betrachten wir $\texttt{lcp}[r]$, das die Länge des gemeinsamen Präfixes dieser Suffixe beschreibt. Bei der Iteration merken wir uns den größten solchen `lcp`-Wert.

Lösung mit Suffixbaum und Matching Statistics. Eine dritte Lösung verwendet einen kleineren Index, nämlich den Suffixbaum nur eines der beiden Strings s (z.B. des kürzeren), in dem allerdings noch die Suffixlinks (etwa aus der Ukkonen-Konstruktion) enthalten sein müssen. Für jede Position p des anderen Strings t wird nun die Länge $\ell_p^{t|s}$ des längsten Strings berechnet, der in t bei Position p beginnt und (irgendwo) in s vorkommt (also die Länge des längsten gemeinsamen Teilstrings von s und t, der bei p in t beginnt). Die Elemente des Vektors $\ell^{t|s} = (\ell_p^{t|s})$ heißen auch *Matching Statistics* von t gegen s. Natürlich ist dann

$$\mathrm{lcf}(s,t) = \max_{0 \le p < |t|} \ell_p^{t|s}.$$

Wie berechnet man nun $\ell_p := \ell_p^{t|s}$? Für $p = 0$ ist das klar: Man sucht im Suffixbaum von s so lange nach dem Präfix von t, bis man kein passendes Zeichen mehr findet. Die so erreichte Stringtiefe ist ℓ_0. Wenn p erhöht wird, muss man den gefundenen String links um ein Zeichen verkürzen. Dazu dienen genau die Suffixlinks. Da man (wie bei Ukkonen's Algorithmus) innerhalb einer Kante sein kann, muss man eventuell zunächst zum nächsthöheren Knoten und von dort aus den Suffixlink benutzen, und ein Stück wieder einen anderen Ast hinunter. Von dort aus sucht man nach weiteren passenden Zeichen, bis man keine mehr findet. Es ist also stets $\ell_p \ge \ell_{p-1} - 1$. Man benötigt Zeit $\mathcal{O}(|s|)$ zur Konstruktion des Suffixbaums und amortisiert Zeit $\mathcal{O}(|t|)$ zum Berechnen von $\ell^{t|s}$. Dabei kann das Maximum online gefunden werden, so dass man den Vektor (ℓ_p) nicht speichern muss.

Matching Statistics sind darüber hinaus nützlich vielen Algorithmen, um die lokale Eindeutigkeit von Strings zu charakterisieren: Ist $\ell_p^{t|s}$ klein, dann kommen lange Teilstrings von t um Position p herum in s nicht vor.

4.6.5 Maximal Unique Matches (MUMs)

Eine Variante von gemeinsamen Teilstrings sind unique matches. Ein String u ist ein *unique match* von s und t, wenn u genau einmal in s und genau einmal in t vorkommt. Ein unique match u heißt maximal, wenn weder au noch ua für irgendein $a \in \Sigma$ ebenfalls ein unique match ist, wenn also u nicht nach links oder rechts verlängert werden kann, ohne die unique match Eigenschaft zu verlieren. (Achtung: maximal heißt nicht maximal lang; es kann maximale unique matches ganz verschiedener Länge in s und t geben!)

Lösung mit Suffixbaum. Wir betrachten die inneren Knoten v mit genau zwei Kindern, von denen eines ein Blatt aus s und das andere ein Blatt aus t ist. Damit sind die Eigenschaften unique match und Rechtsmaximalität sichergestellt. Wir müssen noch prüfen, dass links von den beiden Suffixen die Zeichen nicht übereinstimmen. An den Blättern lesen wir die Starpositionen p_s und p_t der Suffixe ab und vergleichen explizit $s[p_s - 1]$ und $t[p_t - 1]$. Sind diese verschieden, dann ist der Teilstring der Länge $stringdepth(v)$ an Positionen p_s und p_t ein MUM.

Lösung mit Suffixarray. Wie erkennen wir einen inneren Knoten v, der zwei Blattkinder und keine weiteren Kinder hat, im Suffixarray? Sei d die Stringtiefe von v. Dann muss es einen Index r geben mit $\texttt{lcp}[r] = d$; ferner müssen die benachbarten `lcp`-Werte beide kleiner als d sein. Wir suchen also eine Konstellation im `lcp`-Array, die für irgendein d so aussieht: $(\ldots, < d, d, < d, \ldots)$. So eine Stelle r nennen wir ein *isoliertes lokales Maximum* im `lcp`-Array. Alle solchen Stellen r untersuchen wir auf folgende Kriterien (wie auch bei der Suffixbaumlösung): (1) Von $\texttt{pos}[r-1]$ und $\texttt{pos}[r]$ liegt genau eine Position p_s in s und eine Position p_t in t; (2) $s[p_s - 1] \neq t[p_t - 1]$.

4.7 Die Burrows-Wheeler-Transformation (BWT)

4.7.1 Definition und Eigenschaften

Wir haben schon gesehen, dass es nützlich sein kann, zu einem String $s \in \Sigma^*$ und gegebenem Rang r das Zeichen $s[\texttt{pos}[r]-1]$ anzuschauen, nämlich zum Beispiel um die Linksmaximalität von maximal unique matches zu überprüfen.

4.17 Definition (Burrows-Wheeler-Transformation, BWT)**.** Sei $s\$$ ein String mit Wächter, $|s\$| = n$. Sei `pos` das Suffixarray von $s\$$. Die Abbildung $r \mapsto b_r := s[\texttt{pos}[r] - 1]$ nennt man *Burrows-Wheeler-Transformation*. Für das r mit $\texttt{pos}[r] = 0$ sei $b_r := s[n-1] = \texttt{\$}$. Ebenso wird das Ergebnis $bwt(s) := b = (b_0, \ldots, b_{n-1})$ die Burrows-Wheeler-Transformierte von s oder einfach die BWT von s genannt. Wir schreiben auch $\texttt{bwt}[r]$ für b_r.

Ein technisches Problem ist, dass es mehrere Varianten der Definition gibt. Diese unterscheiden sich in der Behandlung des String-Endes. Wir gehen immer davon aus, dass dem eigentlichen String ein Wächter \$ angehängt wird, der ansonsten im String nicht vorkommt und der kleiner als alle Buchstaben des Alphabets ist.

Um die BWT zu bilden, betrachtet man also die Suffixe von $s\$$ in lexikographischer Reihenfolge und zählt die Buchstaben *vor* der Startposition der Suffixe in dieser Reihenfolge auf. Da der Wächter eindeutig ist, ist klar ersichtlich, welches das letzte Zeichen im String ursprünglichen String ist. Da `pos` eine Permutation der Zahlen zwischen 0 und $n-1$ ist, ist die BWT eine Permutation der Buchstaben von s.

Eine alternative Definition besteht darin, in der BWT den Wächter wegzulassen und explizit anzugeben, zu welchem r-Wert das letzte (oder erste) Zeichen des ursprünglichen Strings gehört.

r	pos$[r]$	bwt$[r]$	Suffix
0	5	a	$
1	4	c	a$
2	1	c	abca$
3	2	a	bca$
4	3	b	ca$
5	0	$	cabca$

r	pos$[r]$	bwt$[r]$	Suffix
0	4	c	a
1	1	c	abca
2	2	a	bca
3	3	b	ca
4	0	a	cabca

Abbildung 4.6: BWTs von cabca$ (links) und von cabca (ohne Wächter, rechts). Da man ohne Wächter der BWT ccaba alleine nicht ansieht, welches das letzte Zeichen im Originaltext ist, muss man dessen r-Wert (hier: das a bei $r = 4$) gesondert angeben. Die BWTs lauten also accab$ bzw. (ccaba,4).

4.18 Beispiel (Suffixarray mit pos und bwt)**.** Abbildung 4.6 zeigt das Suffixarray pos und die BWT bwt zu cabca$ einerseits und zu cabca andererseits. Zur Veranschaulichung sind auch die Suffixe noch angegeben. Die BWTs sind also accab$ (es ist Zufall, dass der Wächter wieder am Ende steht) bzw., wenn kein Wächter vorhanden ist, das Paar (ccaba, 4). ♡

Da das Suffixarray pos in Linearzeit konstruiert werden kann und die BWT trivialerweise in Linearzeit aus pos erhalten wird, folgt:

4.19 Satz. *Die BWT zu einem String s kann in Linearzeit berechnet werden.*

Die BWT wurde erstmals von Burrows und Wheeler definiert und lange Zeit nicht beachtet. Heute kann man jedoch sagen, dass sie aufgrund ihrer vielen Anwendungen für Strings das Analogon zur Fourier-Transformation ist.

Invertierbarkeit in Linearzeit. Bemerkenswert ist, dass die BWT invertiert werden kann, d.h., dass s aus seiner BWT rekonstruiert werden kann. Wir wissen: Die BWT besteht aus den Zeichen *vor* den sortierten Suffixen. Nach bwt$[r]$ folgt also das r-t kleinste Zeichen des Strings. Da die BWT aus den gleichen Zeichen wie der String besteht, müssen wir nur die Buchstaben der BWT sortieren. Wir erhalten für das Beispiel folgendes Bild:

r	0	1	2	3	4	5
bwt$[r]$	a	c	c	a	b	$
sortiert	$	a	a	b	c	c

Wir wissen, dass $ das letzte Zeichen ist; darauf folgt ein c, wie wir Spalte 5 entnehmen. Also beginnt der ursprüngliche String mit einem c. Das nächste Zeichen sollten wir wieder aus der Tabelle ablesen können, indem wir nun die richtige Spalte mit c ablesen. Es gibt jedoch zwei cs (Spalten 1 und 2). Welches ist das richtige?

4.20 Lemma. Das k-te Auftreten eines Symbols $a \in \Sigma$ in der BWT entspricht dem k-ten Auftreten von a als erster Buchstabe bei den sortierten Suffixen.

Beweis. (Es bringt wenig, den Beweistext zu lesen. Es genügt, lange genug Abbildung 4.6 (links) zu betrachten!) „Hinter“ den einzelnen Zeichen der BWT denken wir uns die Suffixe in lexikographischer Ordnung, wie in der Abbildung gezeigt. Betrachte nur diejenigen r, in denen in der BWT ein bestimmter Buchstaben $a \in \Sigma$ steht. Hängen wir hinter das `bwt`-Zeichen das entsprechende Suffix, zeigt sich, dass die betrachteten Zeilen in lexikographischer Reihenfolge stehen (der erste Buchstabe a ist gleich, die Suffixe sind lexikographisch sortiert). Dieselbe Reihenfolge ergibt sich natürlich, wenn wir nur die Suffixe (jetzt alle) betrachten. Also entspricht das k-te a in der BWT dem k-ten a bei den sortierten Suffixen. □

Im Beispiel haben wir in Spalte 5 das zweite `c` in der sortierten Reihenfolge gesehen. Dieses entspricht also auch dem zweiten `c` in der BWT, also dem in Spalte 2. Wir lesen ab, dass ein `a` folgt, und zwar das zweite. Dieses finden wir in der BWT in Spalte 3; es folgt ein `b`. Dieses finden wir in Spalte 4; es folgt ein `c` (das erste). Dieses finden wir in Spalte 1; es folgt ein `a` (das erste). Dieses finden wir in Spalte 0; es folgt das `$`, und wir sind fertig. Der Text lautet also `cabca$`, was laut Beispiel 4.18 stimmt.

Gleichzeitig können wir bei diesem Durchlauf auch das Suffixarray `pos` aus der BWT konstruieren. Wir müssen nur in die jeweilige Spalte, in der wir gerade arbeiten, die aktuelle Position schreiben; diese laufen wir ja von 0 bis $n-1$ durch. Wir haben nacheinander für Positionen $0, \ldots, 5$ die Spalten $5, 2, 3, 4, 1, 0$ besucht (dies entspricht dem `rank`-Array). Schreiben wir in die entsprechende Spalte die entsprechende Position, erhalten wir folgendes Resultat:

r	0	1	2	3	4	5
`bwt`$[r]$	a	c	c	a	b	$
sortiert	$	a	a	b	c	c
`pos`	5	4	1	2	3	0

In der Tat ist dies das Suffixarray aus Beispiel 4.18.

Wir behaupten nun, dass sich das Durchlaufen der Tabellenspalten in Linearzeit realisieren lässt. Dazu müssen wir in jedem Schritt zwei Dinge jeweils in konstanter Zeit feststellen.

1. Wenn in Zeile „sortiert“ in Spalte r das Zeichen a steht, das wie vielte a in dieser Zeile ist das?
2. Die Antwort auf Frage 1 sei k. In Zeile `bwt`, in welcher Spalte finden wir das k-te a? Dies ist der neue Wert für r.

Die erste Frage ist einfach zu beantworten: Wir benötigen ein Array `less` der Größe $|\Sigma| + 1$ (plus eins wegen `$`), so dass `less`$[a]$ die Anzahl der Buchstaben in s ist, die kleiner als a sind. Da die Zeile sortiert ist, sind die Spalten 0 bis `less`$[a]-1$ mit kleineren Zeichen belegt, und es folgt $k = r - \texttt{less}[a] + 1$. Beispiel:

a	$	a	b	c
`less`$[a]$	0	1	3	4

In Spalte $r = 5$ steht in Zeile „sortiert“ ein `c`; es ist `less`$[\texttt{c}] = 4$, also sind Spalten 0–3 mit den kleineren Zeichen als `c` belegt; also ist in Spalte 5 das $k = 5 - 4 + 1 = 2$-te `c`.

Für die zweite Frage erstellen wir (konzeptionell) $|\Sigma|+1$ Listen $\texttt{bwtfind}_a$, für jeden Buchstaben $a \in \Sigma \cup \{\$\}$ eine, so dass $\texttt{bwtfind}_a[k]$ genau die Position des k-ten a in `bwt` angibt. Alle diese Listen können nacheinander in alphabetischer Reihenfolge in einem einzigen Array `bwtfind` der Größe n gespeichert werden, so dass dann $\texttt{bwtfind}_a[k] = \texttt{bwtfind}[\texttt{less}[a] + k - 1]$ ist. (Die -1 kommt daher, dass wir bei k mit 1 beginnen zu zählen.) Beispiel:

a	\$	a		b	c	
k	1	1	2	1	1	2
`bwtfind`	5	0	3	4	1	2

Bemerkenswerterweise lässt sich das Finden des nächsten r-Werts nun wie folgt zu einem Schritt zusammenfassen: Aus $\texttt{bwtfind}_a[k] = \texttt{bwtfind}[\texttt{less}[a]+k-1]$ und $k = r - \texttt{less}[a]+1$ folgt, dass wir vom aktuellen r zu $\texttt{bwtfind}[r]$ übergehen. Das Array `bwtfind` lässt sich wie beschrieben in Linearzeit berechnen. Konzeptionell verwendet man dabei *Bucket Sort*: Die Zeichen werden zunächst gezählt (entspricht der Tabelle `less`); dann werden die r-Werte stabil sortiert direkt an die richtigen Stellen in `bwtfind` geschrieben.

Das Gesamtverfahren kann wie folgt angegeben werden; hierbei wird nur das Array `bwtfind` benötigt und keine explizite Repräsentation der sortierten `bwt`.

```
1 def inverse_bwt(b):
2     bwtfind = bwt_bucket_sort(b)  # berechne bwtfind
3     r = bwtfind[0]  # r=0 ist $; bwtfind[0] also Startspalte
4     for i in range(n):
5         r = bwtfind[r]
6         yield b[r]
```

Wir fassen zusammen:

4.21 Satz. *Aus der BWT eines Strings kann man in Linearzeit den String selbst und gleichzeitig das Suffixarray* `pos` *rekonstruieren.*

4.7.2 Anwendung: Pattern Matching mit Backward Search

Wir haben gesehen, dass man mit dem Suffixarray `pos` das Intervall, das alle Vorkommen eines Musters P mit $|P| = m$ enthält, in Zeit $\mathcal{O}(m \log n)$ finden kann. Hier zeigen wir eine elegante Lösung, wie man mit Hilfe der BWT und einem weiteren Array das selbe Problem in Zeit $\mathcal{O}(m)$ löst, indem man die Buchstaben von P rückwärts abarbeitet. Dieses Vorgehen simuliert also *nicht* die Suche im Suffixbaum!

Wir benötigen ein neues Hilfsarray `Occ`, so dass für $a \in \Sigma$ und $r \in \{0, \ldots, n-1\}$ die Zahl $\texttt{Occ}[a, r]$ angibt, wie viele as in der BWT im Intervall $[0, r]$ stehen. (Für $r < 0$ sei sinnvollerweise stets $\texttt{Occ}[a, r] := 0$.) Wir erinnern daran, dass $\texttt{less}[a]$ für jedes $a \in \Sigma$ angibt, wie viele Zeichen insgesamt in der BWT kleiner als a sind.

Die Grundidee ist, nach dem Lesen jedes Zeichens von P die Intervallgrenzen $[L, R]$ anzupassen. Erinnerung: Wir suchen

$$L := \min\{\, r \in \{0, \ldots, n-1\} \mid P \geq_m S_T(r) \,\},$$
$$R := \max\{\, r \in \{0, \ldots, n-1\} \mid P \leq_m S_T(r) \,\},$$

wobei wir $L := n$ bzw. $R = -1$ setzen, wenn das Minimum bzw. Maximum über die leere Menge gebildet würde.

Ist $P = \varepsilon$, dann ist $L = 0$ und $R = n - 1$, da $P =_0 x$ für alle Suffixe x gilt. Interessant ist nun, wie sich L und R verändern, wenn man vor dem aktuellen P ein Zeichen anfügt.

4.22 Lemma (Backward Search)**.** Sei $P^+ := aP$; sei $[L, R]$ das bekannte Intervall zu P und $[L^+, R^+]$ das gesuchte Intervall zu P^+. Dann ist

$$L^+ = \texttt{less}[a] + \texttt{Occ}[a, L-1],$$
$$R^+ = \texttt{less}[a] + \texttt{Occ}[a, R] - 1.$$

Beweis. Da P^+ mit a beginnt, ist klar, dass wir ein Subintervall des Intervalls $[L_a, R_a]$ suchen, in dem die Suffixe mit a beginnen. Das gesamte Intervall ist durch $L_a = \texttt{less}[a]$ und $R_a = \texttt{less}[a] + \texttt{Occ}[a, n-1] - 1$ gegeben. Da für $P = \varepsilon$ jedenfalls $L = 0$, $R = n - 1$ und $P^+ = a$, $L^+ = L_a$, $R^+ = R_a$ ist, stimmt die zu beweisende Gleichung für $|P^+| = 1$.

Jetzt zum allgemeinen Fall: Die BWT-Zeichen im aktuellen Intervall $[L, R]$ zeigen uns, welche Buchstaben vor den aktuellen Suffixen (deren erste $|P|$ Zeichen alle mit P übereinstimmen) stehen. Von den BWT-Zeichen müssen wir die as „auswählen“, also feststellen, welchen as im Intervall $[L_a, R_a]$ im Suffixarray diese entsprechen. Das Problem haben wir aber schon gelöst: Lemma 4.20 sagt uns, dass das k-te a in der BWT und das k-te a als Anfangsbuchstabe eines Suffixes einander entsprechen. Wenn wir also wissen, wie viele as in der BWT bis L vorkommen (nämlich $\texttt{Occ}[a, L]$), dann wissen wir auch, dass wir ab L_a genauso viele as überspringen müssen, um L^+ zu erhalten. Entsprechendes gilt für die Berechnung der rechten Grenze R^+. □

Das Gesamtverfahren beruht nun einfach auf der iterativen Anwendung des Lemmas, indem man P von hinten nach vorne aufbaut. Zum Schluss ist $z = R - L + 1$ die Anzahl der Vorkommen von P im Text.

Einige wichtige Bemerkungen:

- Ein praktisches Problem ist, dass `Occ` relativ viel Speicherplatz benötigt, nämlich $||\Sigma|n|$ ganze Zahlen, was besonders bei großen Alphabeten ein Problem werden kann. Man kann sich aber helfen, indem man `Occ` nicht für jede Position, sondern nur für jede k-te Position abspeichert. Um den $\texttt{Occ}[a, r]$ zu bekommen, muss man dann bei Index $\lfloor r/k \rfloor$ nachschauen und die verbleibenden $r - \lfloor r/k \rfloor$ Zeichen direkt in der BWT anschauen und die as zählen.
- Besonders schön an diesem Verfahren ist, dass man einen stufenlosen Kompromiss zwischen Speicherplatz für `Occ` und Laufzeit für die Suche hinbekommt. Speichert man in `Occ` nur jede k-te Position, benötigt man $OO(|\Sigma|n/k)$ Platz (bei 32-bit Integers genau $4|\Sigma|n/k$ Bytes) und $\mathcal{O}(km)$ Suchzeit (im Erwartungswert muss mann $k/2$ Zeichen bei jedem Schritt in der BWT lesen). Dabei lässt sich aber nutzen, dass das Lesen mehrerer Zeichen hintereinander in der BWT kaum länger dauert als der Zugriff auf ein einzelnes Zeichen; der Grund ist die Cache-Architektur moderner CPUs. Bei einem 4 Gbp DNA-Text (passt gerade noch in mit 32-bit indizierte Arrays) und $k = 128$ benötigt man für `Occ` etwa 512 MB und muss durchscnittlich in jedem Schritt 64 Zeichen hintereinander in der BWT lesen.

- Man beachte, dass man für das Finden der Intervallgrenzen nur `less` und `Occ` und eventuell `bwt` benötigt, also (bei geeigneter Samplingrate k von `Occ`) relativ wenig Speicherplatz. Ist $k > 1$, muss `bwt` zum Zählen verfügbar sein. Man benötigt weder den Text selbst noch `pos`, solange man die Positionen nicht ausgeben will.
- Benutzt man nicht den Text selbst, sondern den reversen Text, kann man darauf wieder „vorwärts“ suchen.

4.7.3 Anwendung: Kompression mit bzip2

Idee. Einer der weiteren Vorteile der BWT von strukturierten Texten ist, dass sie sich gut als Vorverarbeitungsschritt zur verlustfreien Kompression eignet.

Die Idee dabei ist, dass Teilstrings, die im Text oft wiederholt werden, in der BWT lange Runs desselben Buchstaben ergeben.

Beispiel: In einem Roman wird man häufig das Wort „sagte“ finden. Es gibt also (unter anderem) ein Intervall im Suffixarray, in dem die Startpositionen der Suffixe stehen, die mit „agte“ beginnen. An den entsprechenden Stellen der BWT steht ein „s“. Nun kann natürlich auch „fragte“ häufiger vorkommen, was auch zu mehreren „agte…“-Suffixen führt, die sich mit den anderen durchmischen, ebenso andere Wörter wie „betagte“, etc.

Insgesamt wird in diesem Bereich der BWT vielleicht häufig ein „s“, seltener ein „r“, ganz selten ein „t“ zu sehen sein, so dass der entsprechende Ausschnitt so aussehen könnte: …sssssrsrrsssssrrsstsssssssssrssssrrrss…. Es liegt auf der Hand, dass sich solche Abschnitte relativ gut komprimieren lassen.

Natürlich hat nicht jeder Text diese Eigenschaft. Zufällige Strings sehen nach Anwendung der BWT immer noch zufällig aus. Die Intuition ist, dass sich Wiederholungen, die man in strukturierten Texten immer findet, in der BWT in lange Runs eines oder zumindest weniger verschiedener Buchstaben „übersetzen“.

Das bekannte Kompressionsprogramm `bzip2` basiert auf der BWT; wir schauen uns die einzelnen Schritte genauer an.

Blockweise Bearbeitung. Der Text ist immer eine Datei, die als eine Folge von Bytes (0..255) angesehen wird. Das Programm `bzip2` arbeitet blockweise, also nicht auf dem ganzen Text, sondern immer auf einem Block der Datei separat. Die einzeln komprimierten Blöcke werden hintereinandergehängt. Eine typische Blockgröße ist 500 KB, was aus Zeiten stammt, als PCs noch (sehr) kleine Hauptspeicher von wenigen MB hatten. Natürlich wäre eine bessre Kompression möglich, wenn man den gesamten Text auf einmal betrachten würde. Da man allerdings das Suffixarray berechnen muss (und in `bzip2` dafür kein Linearzeitalgorithmus verwendet wird), hat man sich aus Platz- und Zeitgründen entschieden, jeweils nur einen Block zu betrachten. Ein weiterer Vorteil der blockweisen Bearbeitung ist folgender: Sind (durch Materialfehler auf der Festplatte) irgendwann einige Bits in der komprimierten Datei falsch, können die nicht betroffenen Blöcke immer noch rekonstruiert werden.

Die Blockgröße kann in Grenzen eingestellt werden (Optionen `-1` für 100k bis `-9` für 900k). Größere Blöcke benötigen bei der Kompression und Dekompression mehr Hauptspeicher und

führen zu längeren Laufzeiten (da ein nichtlinearer Suffixarray-Algorithmus verwendet wird), erreichen aber unter Umständen eine wesentlich bessere Kompression bei großen Dateien.

Kompressionsschritte. Es werden für jeden Block folgende drei Schritte ausgeführt:

1. Berechne die BWT des Blocks. Es entsteht eine Folge von Bytes, die eine Permutation der ursprünglichen Folge ist.
2. Wende auf die BWT die Move-to-front-Transformation an.

   Hierbei entsteht eine neue Bytefolge, tendenziell aus vielen kleinen Zahlen, insbesondere Nullen bei vielen wiederholten Zeichen (runs).
3. Wende auf das Resultat die Huffman-Codierung an. Normalerweise benötigt jedes Byte 8 bits. Die Huffman-Codierung ersetzt in optimaler Weise häufige Bytes durch kürzere Bitfolgen und seltene Bytes durch längere Bitfolgen. Da nach der Move-to-front-Transformation viele Nullen (und andere kleine Bytes) vorkommen, ergibt sich hierdurch eine erhebliche Einsparung.

Alle diese Transformationen sind invertierbar. Beim Dekodieren werden die Transformationen in umgekehrter Reihenfolge ausgeführt.

Bemerkungen. Man sollte einmal die manual pages zu `bzip2` lesen.

Da jede Datei komprimierbar sein soll und eine Datei alle Byte-Werte enthalten kann, kann man kein besonderes Zeichen für das Stringende reservieren. Man muss also in der Implementierung bei der Berechnung des Suffixarrays das Stringende besonders behandeln.

Es kann sich bei natürlichsprachlichen Texten lohnen, nicht den Text selbst, sondern ihn rückwärts gelesen zu komprimieren. Der Grund ist, dass man in der BWT die Zeichen *vor* den Suffixen betrachtet, und zwar ein paar Buchstaben es relativ gut erlauben vorherzusagen, was davor steht (zum Beispiel häufig „s“ bei „agte“, aber noch besser erlauben vorherzusagen, was dahinter steht (zum Beispiel „e“ bei „sagt“), so dass die BWT des reversen Textes noch besser komprimierbar ist. Hierzu kann man selbst gut Experimente machen.

KAPITEL 5

Approximatives Pattern-Matching

Bisher haben wir stets nach exakten Übereinstimmungen zwischen Pattern und Text gesucht. In vielen Anwendungen ist es jedoch sinnvoll, auch auf approximative Übereinstimmungen von Textteilen mit dem gegebenen Muster hinzuweisen, etwa bei der Suche nach „Ressourcenbeschränkung“ auch die (teilweise falsch geschriebenen) Varianten „Resourcen-Beschränkung“ oder „Ressourcenbeschraenkung“ zu finden. Bisher können wir dieses Problem nur lösen, indem wir alle Alternativen aufzählen und diese sequenziell abarbeiten. Alternativ könnten wir Algorithmen für Patternmengen anwenden (siehe Kapitel 7).

In diesem Kapitel

- definieren wir Abstands- und Ähnlichkeitsmaße zwischen Strings,
- betrachten wir Algorithmen, die diese Maße zwischen zwei Strings berechnen,
- geben wir Algorithmen an, die alle Teilstrings in einem Text finden, die zu einem gegebenen Pattern höchstens einen vorgegebenen Abstand aufweisen.

5.1 Abstands- und Ähnlichkeitsmaße

Wir definieren zunächst einige Distanzmaße. Nicht alle davon sind Metriken. Wir erinnern zunächst an die Definition einer Metrik.

5.1 Definition (Metrik)**.** Sei X eine Menge. Eine Funktion $d : X \times X \to \mathbb{R}_{\geq 0}$ heißt *Metrik* genau dann, wenn

1. $d(x, y) = 0$ genau dann, wenn $x = y$ (Definitheit),
2. $d(x, y) = d(y, x)$ für alle x, y (Symmetrie),

3. $d(x, y) \leq d(x, z) + d(z, y)$ für alle x, y, z (Dreiecksungleichung).

Vergleicht man nur Strings gleicher Länge, bietet sich die Hamming-Distanz d_{H} als Abstandsmaß an. Formal muss man für jedes Alphabet Σ und jede Stringlänge n eine eigene Funktion $d_{\mathrm{H}}^{\Sigma^n}$ definieren; der Zusatz Σ^n wird jedoch in der Notation weggelassen.

5.2 Definition (Hamming-Distanz)**.** Für jedes Alphabet Σ und jedes $n \geq 0$ ist auf $X := \Sigma^n$ die *Hamming-Distanz* $d_{\mathrm{H}}(s, t)$ zwischen Strings s, t definiert als die Anzahl der Positionen, in denen sich s und t unterscheiden.

Man beachte, dass die Hamming-Distanz für $|s| \neq |t|$ zunächst nicht definiert ist. Aus Bequemlichkeitsgründen kann man sie als $+\infty$ definieren. Man sieht durch Nachweisen der Eigenschaften, dass d_{H} eine Metrik auf Σ^n ist.

Die q-gram-Distanz $d_q(s, t)$ zwischen zwei beliegen Strings $s, t \in \Sigma^*$ definiert sich über die in s und t enthaltenen q-grams. Es ist durch Beispiele leicht zu sehen, dass es sich nicht um eine Metrik handelt, insbesondere kann man zwei verschiedene Strings mit q-gram-Distanz 0 finden.

5.3 Definition (q-gram-Distanz)**.** Für einen String $s \in \Sigma^*$ und ein q-gram $x \in \Sigma^q$ sei $N_x(s)$ die Anzahl der Vorkommen von x in s. Dann ist die *q-gram-Distanz* zwischen s und t definiert als

$$d_q(s, t) := \sum_{x \in \Sigma^q} |N_x(s) - N_x(t)|.$$

Die Edit-Distanz (auch: Levenshtein-Distanz) ist das am häufigsten verwendete Abstandsmaß zwischen Strings.

5.4 Definition (Edit-Distanz, Levenshtein-Distanz)**.** Die *Edit-Distanz* zwischen zwei Strings s und t ist definiert als die Anzahl der Edit-Operationen, die man *mindestens* benötigt, um einen String in einen anderen zu überführen. *Edit-Operationen* sind jeweils Löschen, Einfügen und Verändern eines Zeichens.

In einem gewissen Sinn sind Distanz- und Ähnlichkeitsmaße symmetrisch und lassen sich (bei manchen Anwendungen) äquivalent durch einander ersetzen. Manche Eigenschaften lassen sich jedoch natürlicher durch Distanzen ausdrücken (so wie oben), andere durch Ähnlichkeiten (so wie die folgenden).

5.5 Definition (Längster gemeinsamer Teilstring)**.** Die Länge des *längsten gemeinsamen Teilstrings* $\mathrm{lcf}(s, t)$ („f“ für factor) von $s, t \in \Sigma^*$ ist die Länge eines längsten Strings, der sowohl Teilstring von s als auch von t ist. Ein *Teilstring* der Länge ℓ von $s = (s_0, \ldots, s_{|s|-1}) \in \Sigma^*$ ist ein String der Form $(s_i, s_{i+1}, \ldots, s_{i+\ell-1})$ für $0 \leq i \leq |s| - \ell$.

5.6 Definition (Längste gemeinsame Teilsequenz)**.** Die Länge der *längsten gemeinsamen Teilsequenz* $\mathrm{lcs}(s, t)$ von $s, t \in \Sigma^*$ ist die Länge eines längsten Strings, der sowohl Teilsequenz von s als auch von t ist. Eine *Teilsequenz* der Länge ℓ von $s = (s_0, \ldots, s_{|s|-1}) \in \Sigma^*$ ist ein String der Form $(s_{i_0}, \ldots, s_{i_{\ell-1}})$ mit $0 \leq i_0 < i_1 < \cdots < i_{\ell-1} < |s|$.

Abstandsmaße kann man beispielsweise als $d_{\mathrm{lcs}}(s, t) := \max\{\, |s|, |t| \,\} - \mathrm{lcs}(s, t)$ und $d_{\mathrm{lcf}}(s, t) := \max\{\, |s|, |t| \,\} - \mathrm{lcf}(s, t)$ erhalten. Sind diese Metriken?

Die obige Liste ist keinesfalls vollständig; es lassen sich weitaus mehr sinnvolle und unsinnige Abstands- und Ähnlichkeitsmaße auf Strings definieren.

5.2 Berechnung von Distanzen und Ähnlichkeiten

Hamming-Distanz. Sind zwei Strings gegeben, ist die Berechnung der Hamming-Distanz sehr einfach: Man iteriert parallel über beide Strings, vergleicht sie zeichenweise und summiert dabei die Anzahl der verschiedenen Positionen.

q-gram-Distanz. Die Berechnung der q-gram Distanz von $s, t \in \Sigma^*$ ist ähnlich einfach. Es sei $|s| = m$ und $|t| = n$. Es können höchstens $\min\{ m + n - 2q + 2, |\Sigma|^q \}$ verschiedene q-grams in s oder t vorkommen. Wir unterscheiden zwei Fälle:

- $|\Sigma|^q = O(m + n - 2q + 2)$: In diesem Fall übersetzen wir das Alphabet Σ (bei einem unendlichen Alphabet nur die in s, t verwendeten Zeichen) bijektiv in $\{ 0, \ldots, |\Sigma| - 1 \}$ und fassen ein q-gram als Zahl zur Basis $|\Sigma|$ mit q Stellen (also zwischen 0 und $|\Sigma|^q - 1$) auf. In einem Array der Größe $|\Sigma|^q$ zählen wir für jedes q-gram die Anzahl der Vorkommen in s und ziehen davon die Anzahl der Vorkommen in t ab und addieren zum Schluss die Differenzbeträge. Die Laufzeit ist $O(m + n)$; der Speicherbedarf ebenfalls.
- $m + n - 2q + 2 \ll |\Sigma|^q$: In diesem Fall macht es keinen Sinn, alle $|\Sigma|^q$ verschiedenen q-grams zu betrachten und zu zählen. Stattdessen wird man die Strings s, t durchlaufen und die dort vorhandenen q-grams hashen. Unter der (realistischen) Voraussetzung, dass der Zugriff auf ein bestimmtes q-gram amortisiert in konstanter Zeit möglich ist, beträgen Laufzeit und Speicherbedarf hier ebenfalls $O(m + n)$.

Edit-Distanz. Wir betrachten das Problem der Berechnung der Edit-Distanz $d(s, t)$ von $s, t \in \Sigma^*$. Es gibt viele Möglichkeiten, einen String s in einen String t durch Edit-Operationen zu verwandeln. Da die Edit-Operationen die Reihenfolge der Buchstaben nicht ändern, genügt es, den Prozess von links nach rechts zu betrachten. Dies lässt sich auf (mindestens) zwei Arten visualisieren: Als *Alignment* (siehe unten) oder als Pfad im *Edit-Graph* (siehe Abschnitt 5.3).

5.7 Definition (Alignment)**.** Ein Alignment A von $s, t \in \Sigma^*$ ist ein String über $(\Sigma \cup \{-\})^2 \setminus \{(-,-)\}$ mit $\pi_1(A) = s$ und $\pi_2(A) = t$, wobei π_1 ein String-Homomorphismus mit $\pi_1\big(A_i\big) = \pi_1\big((a,b)_i\big) := a$ für $a \in \Sigma$ und $\pi_1\big((-,b)_i\big) := \varepsilon$ ist. Analog ist π_2 der String-Homomorphismus mit $\pi_2\big(A_i\big) = \pi_2\big((a,b)_i\big) := b$ für $b \in \Sigma$ und $\pi_2\big((a,-)_i\big) := \varepsilon$.

5.8 Definition (Kosten eines Alignments)**.** Die Kosten eines Alignments berechnen sich als die Summe der Kosten der Spalten:

$$d(a,b) := \begin{cases} 0 & a = b, \\ 1 & a \neq b; \end{cases}$$

dabei ist ist $a = -$ oder $b = -$ erlaubt.

5.9 Beispiel (Alignments)**.** Einige mögliche Alignments von `ANANAS` und `BANANE` sind

```
ANANAS------
------BANANE
```
(Kosten 12),
```
ANANAS-
-BANANE
```
(Kosten 4),
```
-ANANAS
BANANE-
```
(Kosten 3).

♡

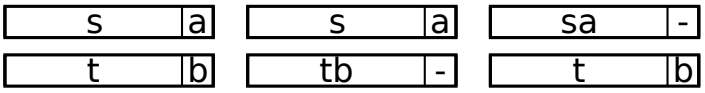


Abbildung 5.1: Ein Alignment der Strings sa und tb kann auf genau 3 verschieden Arten enden. Dabei sind $s, t \in \Sigma^*$ und $a, b \in \Sigma$.

Das Problem, die Edit-Distanz zu berechnen, ist äquivalent dazu, die minimalen Kosten eines Alignments zu finden: Jedes Alignment ist eine Vorschrift, s mit Hilfe von Edit-Operationen in t umzuschreiben (oder t in s). Umgekehrt entspricht jede Edit-Sequenz genau einem Alignment.

Ein Alignment muss auf genau eine von drei Arten enden; siehe Abbildung 5.1. Aus dieser Beobachtung ergibt sich das folgende Lemma zur Berechnung der Edit-Distanz.

5.10 Lemma (Rekurrenz zur Edit-Distanz)**.** Seien $s, t \in \Sigma^*$, sei ε der leere String, $a, b \in \Sigma$ einzelne Zeichen. Es bezeichne d die Edit-Distanz. Dann gilt:

$$\begin{aligned} d(s, \varepsilon) &= |s|, \\ d(\varepsilon, t) &= |t|, \\ d(a, b) &= \begin{cases} 1 & \text{falls } a \neq b, \\ 0 & \text{falls } a = b, \end{cases} \\ d(sa, tb) &= \min \left\{ \begin{array}{l} d(s,t) + d(a,b), \\ d(s, tb) + 1, \\ d(sa, t) + 1. \end{array} \right\} \end{aligned} \tag{5.1}$$

Beweis. Die elementaren Fälle sind klar; wir wollen Gleichung (5.1) beweisen. Die Richtung „$\leq$“ gilt, da alle drei Möglichkeiten zulässige Edit-Operationen für sa und tb darstellen. Eine Illustration findet sich in Abbildung 5.1. Um die Ungleichung „$\geq$“ zu zeigen, führen wir einen Widerspruchsbeweis mit Induktion. Annahme: $d(sa, tb) < \min\{\ldots\}$. Ein Alignment von sa, tb muss jedoch auf eine der o.g. Arten enden: Entweder a steht über b, oder a steht über –, oder – steht über b. Je nachdem, welcher Fall im optimalen Alignment von sa und tb eintritt, müsste bereits $d(s, t)$ oder $d(s, tb)$ oder $d(sa, t)$ kleiner als optimal gewesen sein (Widerspruch zur Induktionsannahme). □

Die Edit-Distanz kann mit einem *Dynamic-Programming-Algorithmus* berechnet werden. Dynamic Programming (DP) ist eine algorithmische Technik, deren Anwendung sich immer dann anbietet, wenn Probleme im Prinzip rekursiv gelöst werden können, dabei aber (bei naiver Implementierung) dieselben Instanzen des Problems wiederholt gelöst werden müssten.

Die Edit-Distanz wird mit dem Algorithmus von **?** wie folgt berechnet. Seien $m := |s|$ und $n := |t|$. Wir verwenden eine Tabelle $D[i, j]$ mit

$$D[i, j] := \text{Edit-Distanz der Präfixe } s[\ldots i-1] \text{ und } t[\ldots j-1].$$

(Hier rächt sich nun, dass wir die Indizierung von Sequenzen bei 0 beginnen; wir brauchen nun nämlich eine Zeile und Spalte, um die leeren Präfixe zu behandeln; daher die Verschiebung

um -1. Intuitiver ist vielleicht zu sagen, $D[i,j]$ ist die Edit-Distanz zwischen dem s-Präfix der Länge i und dem t-Präfix der Länge j.)

Wir initialisieren die Spalte 0 und die Zeile 0 durch $D[i,0] = i$ für $0 \leq i \leq m$ und $D[0,j] = j$ für $0 \leq j \leq n$ (siehe Lemma 5.10). Alle weiteren Werte können berechnet werden (Lemma 5.10), sobald die Nachbarzellen „darüber“, „links daneben“ und „links darüber“ bekannt sind:

$$D[i,j] = \min \left\{ \begin{array}{l} D[i-1,j-1] + d(s[i-1], t[j-1]), \\ D[i-1,j] + 1, \\ D[i,j-1] + 1 \end{array} \right\}. \tag{5.2}$$

Die Auswertung kann in verschiedenen Reihenfolgen erfolgen, z.B. von links nach rechts, oder von oben nach unten. In jedem Fall können wir nach Ausfüllen der Tabelle die Edit-Distanz von s und t im Feld $D[m,n]$ ablesen.

Die Berechnung einer Zelle der DP-Tabelle ist in konstanter Zeit möglich. Damit ist der Zeitbedarf $\mathcal{O}(nm)$. Sind wir nur an der Edit-Distanz, aber nicht am Alignment selbst interessiert, brauchen wir nur jeweils die zuletzt berechnete Spalte/Zeile zu speichern (je nachdem, ob wir zeilen- oder spaltenweise die Tabelle füllen). Damit kommen wir auf einen Platzbedarf von $\mathcal{O}(\min\{m,n\})$. Falls nach dem Erstellen der Tabelle das Alignment rekonstruiert werden soll, wird es aufwändiger (siehe die Kommentare zum Traceback in Abschnitt 6.1). Mit anderen Varianten von Alignments befassen wir uns in Kapitel 6 noch ausführlich. In der dortigen Sprechweise erstellen wir zur Berechnung der Edit-Distanz ein globales Alignment.

Longest Common Subsequence. Zur Berechnung von $\mathrm{lcs}(s,t)$ lässt sich eine analoge DP-Idee wie bei der Edit-Distanz verwenden. Nur ist hier zu beachten, dass nicht „Kosten“ minimiert, sondern „Punkte“ (Anzahl der Spalten mit identischen Zeichen) maximiert werden sollen. Außerdem ist eine Ersetzung von Zeichen verboten; man kann nur Zeichen einfügen oder löschen oder identische Zeichen untereinander schreiben. Mit

$$L[i,j] := \text{Länge der längsten gemeinsamen Teilsequenz von } s[\ldots i-1] \text{ und } t[\ldots j-1]$$

ergibt sich(**TODO: Initialisierung**)

$$L[i,j] = \max \left\{ \begin{array}{l} L[i-1,j-1] + [\![s[i-1] = t[j-1]]\!], \\ L[i-1,j], \\ L[i,j-1] \end{array} \right\}.$$

Wiederum benötigt man hierfür $O(\min\{m,n\})$ Platz und $O(mn)$ Zeit. Es gibt zahlreiche Ideen, die Berechnung im Falle sehr ähnlicher Sequenzen zu beschleunigen.

Longest Common Factor. Die Berechnung von $\mathrm{lcf}(s,t)$ haben wir bereits in Abschnitt 4.6.4 diskutiert: Wir haben gesehen, dass die Berechnung in Linearzeit $\mathcal{O}(m+n)$ möglich ist, wenn man ein Suffixarray oder einen Suffixbaum verwendet. Eine interessante Übung ist, hier einen (weniger effizienten) DP-Algorithmus mit $\mathcal{O}(mn)$ Laufzeit im Geiste der anderen Algorithmen dieses Abschnitts anzugeben.

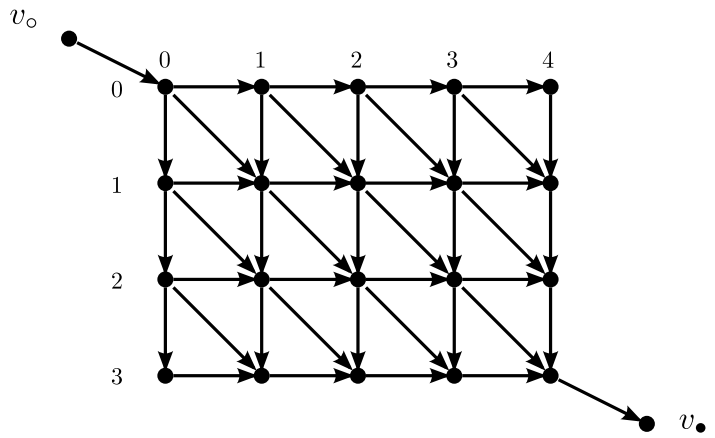


Abbildung 5.2: Beispiel für einen Edit- oder Alignment-Graphen für ein globales Alignment.

5.3 Der Edit-Graph

Wir drücken das Problem, ein Sequenzalignment zu berechenen, nun mit Hilfe eines Graphen aus. In Abbildung 5.2 wird ein Edit-Graph (globaler Alignment-Graph) dargestellt.

5.11 Definition (globaler Alignment-Graph, Edit-Graph)**.** Der *globale Alignment-Graph* oder auch *Edit-Graph* ist wie folgt definiert:

- Knotenmenge $V := \{(i,j) : 0 \leq i \leq m, 0 \leq j \leq n\} \cup \{v_\circ, v_\bullet\}$
- Kanten:

	Kante	Label	Kosten
horizontal	$(i,j) \to (i,j+1)$	$\binom{-}{t_j}$	1
vertikal	$(i,j) \to (i+1,j)$	$\binom{s_i}{-}$	1
diagonal	$(i,j) \to (i+1,j+1)$	$\binom{s_i}{t_j}$	$[\![s_i \neq t_j]\!]$
Initialisierung	$v_\circ \to (0,0)$	ε	0
Finalisierung	$(m,n) \to v_\bullet$	ε	0

Jeder Pfad zwischen $v_\circ$ und $v_\bullet$ entspricht (durch die Konkatenation der Kantenlabel) genau einem Aligment von s und t.

Mit der Definition des Edit-Graphen haben wir das Problem des Sequenzalignments als billigstes-Wege-Problem in einem Edit-Graphen formuliert: Finde den Weg mit den geringsten Kosten von $v_\circ$ nach $v_\bullet$.

Der Wert $D[i,j]$ lässt sich nun interpretieren als die minimalen Kosten eines Pfades vom Startknoten zum Knoten (i,j).

Die Rekurrenz (5.2) lässt sich jetzt so lesen: Jeder Weg, der in (i,j) endet, muss als Vorgängerknoten einen der drei Knoten $(i-1,j)$, $(i-1,j-1)$ oder $(i,j-1)$ besitzen. Die minimalen Kosten erhält man also als Minimum über die Kosten bis zu einem dieser drei Knoten plus die Kantenkosten nach (i,j); genau das sagt (5.2).

Tabelle 5.1: Anzahl der Alignments $N(m,n)$ für $0 \le m,n \le 4$.

$m \mid n$	0	1	2	3	4	...
0	1	1	1	1	1	...
1	1	3	5	7	9	...
2	1	5	13	25	41	...
3	1	7	25	63	129	...
4	1	9	41	129	321	...
⋮	⋮	⋮	⋮	⋮	⋮	⋱

5.4 Anzahl globaler Alignments

Wie viele globale Alignments zweier Sequenzen der Längen m und n gibt es? Diese Anzahl $N(m,n)$ ist gleich der Anzahl der Möglichkeiten, eine Sequenz der Länge m in eine andere Sequenz der Länge n mit Hilfe von Edit-Operationen umzuschreiben. Dies ist ebenfalls gleich der Anzahl der Pfade im Edit-Graph vom Start- zum Zielknoten. Offensichtlich hängt $N(m,n)$ nur von der Länge der Sequenzen (und nicht von den Sequenzen selbst) ab.

5.12 Lemma (Anzahl der Pfade im Edit-Graph)**.** Für die Anzahl $N(m,n)$ der Pfade im Edit-Graph vom Startknoten nach (m,n) gilt:

$$\begin{aligned} N(0,0) &= 1, \\ N(m,0) &= 1, \\ N(0,n) &= 1, \\ N(m,n) &= N(m-1,n-1) + N(m,n-1) + N(m-1,n). \end{aligned} \tag{5.3}$$

Beweis. Abzählen anhand von Abbildung 5.1 oder Abbildung 5.2. □

Anhand einiger Beispiele (Tabelle 5.1) sieht man, dass es schon für kleine m und n relativ viele Alignments gibt. In welcher Größenordung liegen die Diagonalwerte $N(n,n)$? Offensichtlich gilt $N(n,n) > 3N(n-1,n-1)$ und damit $N(n,n) > 3^n$. Man kann ausrechnen, dass sich asymptotisch $N(n,n) = \Theta\big(\sqrt{n}\cdot(1+\sqrt{2})^{2n+1}\big)$ ergibt; d.h., das Wachstum ist exponentiell mit Basis $(1+\sqrt{2})^2 \approx 5.8$.

5.5 Approximative Suche eines Musters in einem Text

Wir behandeln hier nur den Fall der Edit-Distanz, da er der wichtigste in Anwendungen ist. Wir betrachten zuerst eine Modifikation des DP-Algorithmus zum globalen Alignment und dann eine bit-parallele NFA-Implementierung, die sich flexibel auf erweiterte Muster (Zeichenklassen, Wildcards) verallgemeinern lässt.

		A	M	O	A	M	A	M	A	O	M
	0	**0**	0	0	0	0	**0**	0	0	0	0
M	1	1	**0**	1	1	0	1	**0**	1	1	0
A	2	1	**1**	1	1	1	0	1	**0**	1	1
O	3	2	2	**1**	2	2	1	1	1	**0**	1
A	4	3	3	2	**1**	2	2	2	1	**1**	1
M	5	4	3	3	2	**1**	2	2	2	2	**1**

Abbildung 5.3: DP-Tabelle für ein semi-globales Sequenzalignment des Patterns `MAOAM` mit dem Text `AMOAMAMAOM`. Die blaue Linie entspricht der $last_k$-Funktion für einen maximal erlaubten Fehler von $k = 1$, d.h. für alle Zellen $D[i, j]$ unter der Linie gilt: $i > last_k(j)$. In rot gekennzeichnet sind die durch Backtracing gewonnenen Alignments der beiden gefundenen Treffer.

5.5.1 DP-Algorithmus von Ukkonen

Wenn wir approximativ nach einem gegebenen Patten P mit $|P| = m$ in einem Text T mit $|T| = n$ suchen wollen, ist es nicht zielführend, die Edit-Distanz bzw. ein globales Alignment des Textes und des Musters zu berechnen, da wir zulassen, dass das Muster an jeder beliebigen Stelle im Text beginnen darf. Wir müssen die Definition der Tabelle D auf die veränderte Situation anpassen.

Wir definieren $D[i, j]$ als die minimale Edit-Distanz des Präfixes $P[\ldots i-1]$ mit einem Teilstring $T[j' \ldots j-1]$ mit $j' \leq j$. Daraus ergibt sich die Initialisierung der nullten Zeile als $D[0, j] = 0$ für $0 \leq j \leq n$. Andere Änderungen des DP-Algorithmus ergeben sich nicht. Ist die maximale Edit-Distanz k vorgegeben, suchen wir nach Einträgen $D[m, j] \leq k$; denn das bedeutet, dass das Pattern P an Position $j - 1$ des Textes mit höchstens k Fehlern (Edit-Operationen) endet (und bei irgendeinem $j' \leq j$ beginnt).

Man spricht hier auch von einem semiglobalen Alignment (das gesamte Pattern P wird als approximativer Teilstring des Textes gesucht). Sinnvollerweise erfolgt die Berechnung der Tabelle spaltenweise. Der Algorithmus benötigt (wie beim globalen Alignment) $\mathcal{O}(mn)$ Zeit und $\mathcal{O}(m)$ Platz, wenn nur die Endpositionen der Matches berechnet werden sollen.

Wir behandeln jetzt eine Verbesserung des modifizierten DP-Algorithmus von **?**. Wenn eine Fehlerschranke k vorgegeben ist, so müssen wir nicht die komplette DP-Tabelle berechnen, da wir nur an den Spalten interessiert sind, in denen ein Wert kleiner gleich k in der untersten Zeile steht. Formal definieren wir

$$last_k(j) := \max\{\, i \mid D[i, j] \leq k,\ D[i', j] > k \text{ für alle } i' > i \,\}.$$

Werte $D[i, j]$ mit $i > last_k(j)$ müssen nicht berechnet werden. Dieser Sachverhalt ist in Abbildung 5.3 illustriert. Die Frage ist, wie man die Funktion $last_k$ berechnen kann, ohne die komplette Spalte der Tabelle zu kennen. Für die erste Spalte ist die Berechnung von $last_k$ einfach, denn $D[i, 0] = i$ für alle i und damit $last_k(0) = k$. Für die weiteren Spalten nutzen

wir aus, dass sich benachbarte Zellen immer maximal um eins unterscheiden können (was man mit einem Widerspruchsbeweis zeigen kann). Das bedeutet, dass $last_k$ von einer Spalte zur nächsten um maximal eins ansteigen kann: $last_k(j+1) \leq last_k(j)+1$. Man berechnet die $(j+1)$-te Spalte also bis zur Zeile $last_k(j)+1$ und verringert dann möglicherweise $last_k(j+1)$ anhand der berechneten Werte.

Die Verwendung von $last_k$ bringt bei (relativ) kleinen k und (relativ) großen m in der Praxis tatsächlich einen Vorteil. Man kann beweisen, dass der verbesserte Algorithmus auf zufälligen Texten eine erwartete Laufzeit von $\mathcal{O}(kn)$ statt $\mathcal{O}(mn)$ hat.

Die folgende Funktion liefert nacheinander die Endpositionen j im Text, an denen ein Treffer mit höchstens k Fehlern endet und jeweils die zugehörige Fehlerzahl d. Bei Bedarf könnte man Code hinzufügen, um für jede dieser Positionen durch Traceback ein Alignment zu ermitteln (siehe Abschnitt 6.1).

```
def ukkonen(P,T,k, cost=unitcost):
    """liefert jede Position j zurueck,
an der ein <=k-Fehler-Match von P in T endet,
und zwar in der Form (j,d) mit Fehlerzahl d."""
    m, n = len(P), len(T)
    Do = [k+1 for i in range(m+1)]
    Dj = [i for i in range(m+1)]
    lastk = min(k,m)
    for j in range(1,n+1):
        Dj, Do = Do, Dj
        Dj[0] = 0
        lastk = min(lastk+1,m)
        for i in range(1,lastk+1):
            Dj[i] = min( Do[i]+1,
                         Dj[i-1]+1,
                         Do[i-1]+cost(P[i-1],T[j-1]) )
        while Dj[lastk]>k: lastk -= 1
        if lastk==m: yield (j-1,Dj[m])
```

5.5.2 Fehlertoleranter Shift-And-Algorithmus

Eine andere Idee zur approximativen Suche besteht in der Konstruktion eines NFAs. Dabei verwenden wir wir einen „linearen“ NFA wie beim Shift-And-Algorithmus für einfache Muster. Allerdings betreiben wir nun $k+1$ solche Automaten parallel, wenn k die Anzahl der maximal erlaubten Fehler ist. Formaler ausgedrückt verwenden wir den Zustandsraum $Q = \{0,\ldots,k\} \times \{-1,0,\ldots,|P|-1\}$ für die Suche nach dem Pattern P mit maximal k Fehlern. Ein solcher Automat ist für `abbab` in Abbildung 5.4 illustriert.

Die Idee dahinter ist, dass folgende Invariante gilt: Der Zustand (i,j), also der Zustand in Zeile i und Spalte j ist genau dann aktiv, wenn zuletzt $P[\ldots j]$ mit $\leq i$ Fehlern gelesen wurde. Aus dieser Invariante folgt beispielsweise: Ist in einer Spalte j Zustand (i,j) aktiv, dann auch alle darunter, also (i',j) mit $i' > i$.

Wir konstruieren Kanten und setzen die Startzustände so, dass die Invariante gilt (was wir dann natürlich per Induktion beweisen müssen).

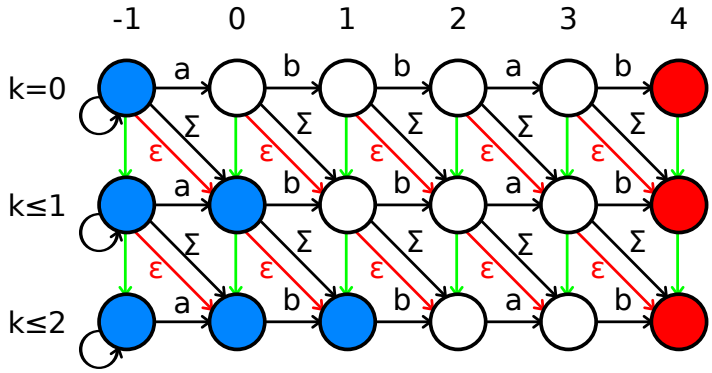


Abbildung 5.4: NFA zum Finden von approximativen Vorkommen des Patterns `abbab` aus dem Alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Der Automat findet alle Vorkommen, die maximal eine Edit-Distanz von $k = 2$ haben. Blaue Zustände: Startzustände; rote Zustände: akzeptierend. Grüne vertikale Kanten: Σ-Kante für Insertionen. Rote diagonale Kanten: ε-Kanten für Deletionen. Schwarze diagonale Kanten: Σ-Kanten für Substitutionen.

- Jede Zeile für sich entspricht dem Shift-And-Automaten.
- Zeile i und $i+1$ sind für $0 \leq i < k$ durch drei Arten von Kanten verbunden.
  1. Zustand (i, j) und $(i+1, j)$ sind für alle $-1 \leq j \leq m$ durch eine Σ-Kante verbunden, also eine Kante, an der jeder Buchstabe gelesen werden darf. Dies entspricht einer Insertion in das Muster: Wir lesen einen Textbuchstaben, ohne die Position j im Muster zu erhöhen.
  2. Zustand (i, j) und $(i+1, j+1)$ sind für alle $-1 \leq j < m$ durch eine Σ-Kante verbunden. Dies entspricht einer Substitution (oder auch einer Identität, aber wenn der korrekte Buchstabe kommt, gibt es ja zusätzlich die Kante, die in der selben Zeile verbleibt): Wir lesen einen Textbuchstaben und schreiten im Muster voran, erhöhen dabei aber die Fehlerzahl.
  3. Zustand (i, j) und $(i+1, j+1)$ sind weiterhin für alle $-1 \leq j < m$ durch eine ε-Kante verbunden, also eine Kante, entlang der man keinen Textbuchstaben liest. Dies entspricht einer Deletion in P: Wir springen in P eine Position weiter, ohne ein Textzeichen zu lesen.
- Startzustände sind alle (i, j) mit $0 \leq j \leq i \leq k$ (ein „Dreieck“ links unten im Automaten); diese Menge erfüllt die Invariante und hat alle erreichbaren ε-Kanten bereits zum Ziel verfolgt, ist also bezüglich ε-Kanten abgeschlossen.

Der induktive Beweis der Invariante folgt nun folgender Idee: Wie kann nach dem Lesen von t Textzeichen Zustand (i, j) aktiv sein? Das ist genau dann der Fall, wenn ein Schritt vorher $(i-1, j)$ aktiv war und wir über eine Insertions-Kante gegangen sind oder $(i-1, j-1)$ aktiv war und wir über eine Substitutions- oder Deletions-Kante gegangen sind oder $(i, j-1)$ aktiv war und wir über eine horizontale $P[j]$-Kante gegangen sind. Aus diesen vier (nicht exklusiven) Fallunterscheidungen ergibt sich mit Hilfe der Induktionsannahme die Behauptung.

Der konstruierte NFA wird mit einem bit-parallelen Shift-Ansatz simuliert. Für jede Zeile i definieren wir einen Bitvektor A_i. Die Bitvektoren aus dem jeweils vorhergegangenen Schritt bezeichnen wir mit $A_i^{(alt)}$. Es ergeben sich prinzipiell folgende Operationen beim Lesen eines Texteichens c; hierbei gehen wir davon aus, dass alle Zustände (auch die in Spalte -1) durch Bits repräsentiert werden.

- $A_0 \leftarrow (A_0^{(alt)} \ll 1|1)\ \&\ mask[c]$
- $A_i \leftarrow \big((A_i^{(alt)} \ll 1|1)\ \&\ mask[c]\big)\ |\ \underbrace{\big(A_{i-1}^{(alt)}\big)}_{\text{Einfügungen}}\ |\ \underbrace{\big(A_{i-1}^{(alt)} \ll 1\big)}_{\text{Ersetzungen}}\ |\ \underbrace{\big(A_{i-1} \ll 1\big)}_{\text{Löschungen}}$

  für $0 < i \leq k$.

Implementierung. Die Implementierung ist komplizierter, wenn man (wie üblich) die Spalte -1 weglassen möchte, um Bits zu sparen: Dabei „verliert“ man nämlich auch die davon ausgehenden Epsilon-Kanten. Die Lösung besteht darin, in Zeile $i > 0$ nach der Verundung mit der c-Maske sicherzustellen, dass die i Bits $0, \ldots, i-1$ gesetzt sind; dies geschieht mit einer Veroderung mit $2^i - 1$. Um nicht parallel alte und neue Bitmasken speichern zu müssen, kann man zunächst „von unten nach oben“ ($i = k, \ldots, 1, 0$) den Shift-And und die Veroderung mit den alten Werten durchführen und dann „von oben nach unten“ ($i = 1, \ldots, k$) die Updates mit den neuen Werten. Statt Schleifen zu verwenden würde man für jedes feste (kleine) k eine eigene Funktion implementieren, um die Geschwindigkeit zu maximieren.

Erweiterte Patternklassen. Statt ein einfaches Pattern $P \in \Sigma^m$ kann man auch erweiterte Patternklassen verwenden. Das Grundprinzip ist, dass sich die Erweiterungen aus Abschnitt 3.8 orthogonal mit der k-Fehler-Konstruktion dieses Abschnitts (k+1-faches Kopieren der Zeilen, Einfügen entsprechender Kanten) kombinieren lassen.

Für die Suche nach einem erweiterten Pattern der Länge m mit maximal k Fehlern ergibt sich so stets eine Laufzeit von $\mathcal{O}\big(n \cdot m/w \cdot (k+1)\big)$. Dabei ist n die Textlänge und w die Wortgröße der Rechnerarchitektur. In der Regel wenden wir bit-parallele Algorithmen nur an, wenn $m/w \in \mathcal{O}(1)$ ist, bzw. sogar nur, wenn das Pattern kürzer als die Registerlänge ist.

Effiziente diagonale Implementierung*. Bisher haben wir den k-Fehler-NFA zeilenweise implementiert. Dazu benötigten wir $k+1$ Register oder Speicherplätze. Ein Update erfolgte durch Iterieren über alle $k+1$ Zeilen, was $\mathcal{O}(k)$ Zeit kostet. Es wäre schön, wenn man die aktiven Zustände des gesamten Automaten in einem einzigen Schritt aktualisieren könnte.

Die Beobachtung, dass sich auf den Diagonalen ε-Kanten befinden, führt zu der Idee, den Automaten nicht zeilen-, sondern *diagonalenweise* darzustellen (vgl. Abbildung 5.5). Wir speichern die Bitvektoren für alle Diagonalen (getrennt durch jeweils eine Null) hintereinander in einem langen Bitvektor. Die 0-te Diagonale D_0 muss nicht gespeichert werden, da alle Zustände auf dieser Diagonalen ohnehin immer aktiv sind. Insgesamt ergibt sich also ein Speicherbedarf von $(m-k)(k+2)$ Bits.

Die Update-Schritte müssen nun angepasst werden. Kritisch ist hierbei die Behandlung der ε-Kanten: Taucht innerhalb einer Diagonale ein aktiver Zustand auf, muss dieser sofort bis ans Ende der Diagonale propagiert werden. Ist beispielsweise Zustand $(0, 1)$ aktiv, müssen

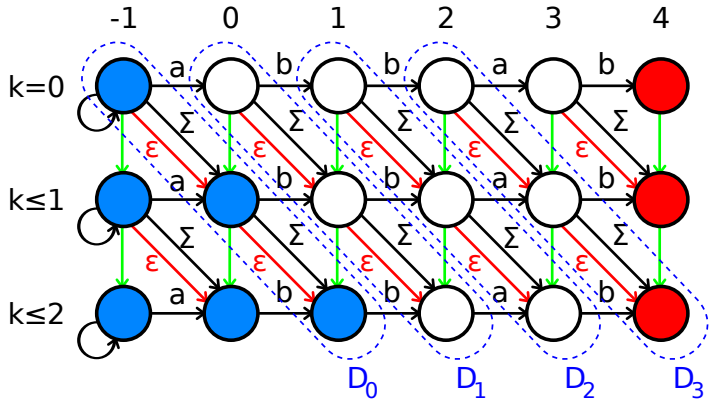


Abbildung 5.5: Verdeutlichung der Repräsentation eines k-Fehler-NFAs durch Diagonalen.

(ohne eine Schleife zu verwenden!) sofort auch $(1, 2)$ und $(2, 3)$ aktiviert werden. Dies ist möglich, da die entsprechenden Zustände in der Diagonal-Darstellung direkt benachbart sind (im Gegensatz zur zeilenweisen Darstellung). Eine entsprechende bit-parallele Technik wurde in Abschnitt 3.8.2 behandelt.

Der Vorteil besteht darin, dass die Schleife mit $k+1$ Schritten entfällt; dies ist insbesondere ein Vorteil, wenn alle Bits in ein Prozessorregister passen. Der Nachteil besteht darin, dass sich diese Technik nicht ohne weiteres auf alle erweiterten Patternklassen anwenden lässt.

5.5.3 Fehlertoleranter BNDM-Algorithmus*

Im vorangegangenen Abschnitt haben wir den Shift-And Algorithmus verallgemeinert. Das heißt, wir haben Automaten bit-parallel simuliert, die einen Text Zeichen für Zeichen von links nach rechts lesen. Eine Alternative ist wieder das Lesen eines Textfensters von rechts nach links, was im besten Fall zu längeren Shifts und damit weniger Vergleichen führt; genauer erhalten wir eine Best-case-Laufzeit von $\mathcal{O}(kn/m)$. Im schlimmsten Fall werden jedoch $\mathcal{O}(knm)$ Vergleiche benötigt.

Wir führen die Konstruktion des entsprechenden Automaten hier nicht im Detail durch, sondern listen lediglich seine Eigenschaften auf. Der Automat

- erkennt das reverse Pattern p^{rev} mit bis zu k Fehlern.
- hat aktive Zustände, solange ein Teilstring von p mit höchstens k Fehlern gelesen wurde,
- erreicht den akzeptierenden Zustand, wenn ein Präfix von p mit höchstens k Fehlern gelesen wurde.
- besteht aus $k+1$ Kopien des BNDM-Automaten, die analog zum fehlertoleranten Shift-And-Automaten verbunden sind.

Algorithmus:

1. Initial werden alle Zustände des Automaten aktiviert.

2. Lies $m - k$ Zeichen im aktuellen Fenster von rechts nach links (ein Match mit $\leq k$ Fehlern kann nun bemerkt werden).
3. Im Erfolgsfall: Vorwärtsverifikation des Patternvorkommens.
4. Verschiebung des Fensters wie beim BNDM-Algorithmus.

5.5.4 Fehlertoleranter Backward-Search-Algorithmus*

Wie beim fehlertoleranten Shift-And- oder BNDM-Algorithmus hängt die Laufzeit nach wie vor von der Textlänge n ab. Aber auch hierfür gibt es einen Ansatz die Laufzeit lediglich von der Patternlänge m abhängig zu machen. Hierbei wird die Idee des NFA aufgegriffen und mit dem Backward-Search-Algorithmus verbunden. Dabei wird nicht wie beim Shift-And der aktive Zustand in den Zuständen des Automaten gespeichert, sondern die aktuellen BS Intervalle.

Wir initialisieren eine leere Matrix M, die $k+1$ Zeilen und $m+1$ Spalten enthält. Hierbei kann eine Zelle aus M mehrere Intervalle enthalten, wodurch es sich empfiehlt diese Einträge in einem *Set* abzuspeichern. In $M[0][0]$ fügen wir das komplette Intervall $[0, n-1]$ ein. Nun führen wir für alle Intervalle aus jedem $M[i][j], \forall 0 \leq i \leq k, 0 \leq j < m$ eine update Operation des Intervalle mit dem j-ten Zeichen des reversen Patterns durch. Handelt es sich um kein leeres Intervall (also $L \leq R$), wird dieses Intervall in $M[i][j+1]$ hinzugefügt.
Zusätzlich führen wir in den Zellen aller $0 \leq i < k$ Zeilen update Operationen durch, die den Edit-Operationen entsprechen.

- Deletion: Bei einer Deletion wird das j-te Zeichen im reversen Pattern gelöscht, dem entsprechend wird das Intervall nicht verändert und kann somit unverändert eine Zeile tiefer in die Zelle $M[i+1][j+1]$ hinzugefügt werden.
- Insertion: Hierbei wird vor dem j-ten Zeichen im reversen Pattern jeweils ein Zeichen aus dem Alphabet $c \in \Sigma$ eingefüegt. Es werden also $|\Sigma|$ viele update Operationen durchgeführt und die neuen Intervalle in der nachfolgenden Zeile in Zelle $M[i+1][j]$ hinzugefügt, sofern die Grenzen der Intervalle $L \leq R$ entsprechen.
- Substitution: Ähnlich, wie bei der Insertion werden hierbei update Operationen für alle $c \in \Sigma \setminus P[j]$ für das j-te Zeichen im reversen Pattern durchgeführt und in der nachfolgenden Zeile in Zelle $M[i+1][j+1]$ hinzugefügt, sofern die Grenzen der Intervalle $L \leq R$ entsprechen.

Das Abarbeiten der Matrix kann entweder spalten- oder zeilenweise erfolgen. In der letzten Spalte der Matrix stehen alle Inervalle mit $\leq k$ Fehlern. Abbildung 5.6 veranschaulicht die Funktionsweise des Algorithmus.

Insgesamt sieht der Algorithmus so aus:

```
1 def suffixarray(T): sorted(range(len(T)), key=lambda i: T[i:])
2 def bwt(T, T_sa): "".join([T[i - 1] if i > 0
3     else T[len(T) - 1] for i in T_sa])
4
5 def update(L, R, c, less, occ):
6     L = less[c] + occ[c][L - 1 if L > 0 else 0]
7     R = less[c] + occ[c][R if R > 0 else 0] - 1
```

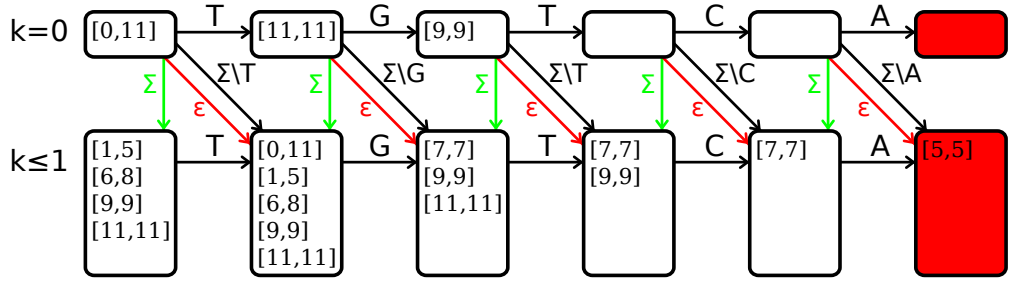


Abbildung 5.6: NFA mittels BS zum Finden von approximativen Vorkommen des Patterns `ACTGT` im Text `AAAACGTACCT$` aus dem Alphabet $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. Der Automat findet alle Vorkommen, die maximal eine Edit-Distanz von $k = 1$ haben. Rote Zustände: akzeptierend. Grüne vertikale Kanten: Σ-Kante für Insertionen. Rote diagonale Kanten: ε-Kanten für Deletionen. Schwarze diagonale Kanten: Σ-Kanten für Substitutionen.

```
    return L, R

def approx_BS(T, P, k, Alphabet)
    T_sa = suffixarray(T)
    T_bwt = bwt(T, T_sa)
    less = less_table(T_bwt)
    occ = occurance(T_bwt)
    M = [[set() for j in range(len(P) + 1)] for i in range(k + 1)]
    M[0][0].add((0, len(T) - 1))
    P = P[::-1] # Pattern wegen BS umdrehen

    for i in range(k + 1):
        for j in range(len(P)):
            for L_old, R_old in M[i][j]:
                # Match
                L, R = update(L_old, R_old, P[j], less, occ)
                if L <= R: M[i][j + 1].add((L, R))

                if i < k:
                    # Deletion
                    M[i + 1][j + 1].add((L_old, R_old))

                    # Insertion / Substitution
                    for c in Alphabet:
                        L, R = update(L_old, R_old, c, less, occ)
                        if L <= R:
                            # Insertion
                            M[i + 1][j].add((L, R))
                            # Substitution
                            if c != P[j]: M[i + 1][j + 1].add((L, R))
```

KAPITEL 6

Paarweises Sequenzalignment

6.1 Globales Alignment mit Scorematrizen und Gapkosten

⇓ 16.06.11

Wir erinnern an die Berechnung der Edit-Distanz mit dem DP-Algorithmus in Abschnitt 5.2 und an die Visualisierung als „billigster“ Pfad im Edit-Graph in Abschnitt 5.3.

In der biologischen Sequenzanalyse lösen wir häufig das selbe Problem in einer etwas anderen Darstellung. Die Unterschiede sind nahezu trivial und verändern den Algorithmus nicht grundsätzlich:

- Statt Kosten zu minimieren, maximieren wir einen Score.
- Der Score $score(A)$ eines Alignments A ist die *Summe* der Scores seiner Bestandteile (Spalten). Diese Additivität ist eine grundlegende Voraussetzung für die effiziente Berechenbarkeit optimaler Alignments. Spalten repräsentieren einen Übereistimmung (Match), Substitution (Mismatch) oder eine Lücke (Gap).
- Scores für Matches und Mismatches werden individuell abhängig von den Nukleotiden oder Aminosäuren festgelegt. Dies geschieht mit einer sogenannten *Scorematrix*, die zu jedem Symbolpaar eine Punktzahl festlegt. Grundsätzlich kann die Scorematrix auch unsymmetrisch sein, wenn die Sequenzen verschiedene Rollen spielen. Ferner kann sie auch positionsabhängig variieren.
- Ebenfalls werden Gapkosten g festgelegt; dies entspricht einem Score (Symbol gegen Gap) von $-g$. Grundsätzlich können Gapkosten auch symbolabhängig und positionsabhängig festgelegt werden; wir verzichten hier darauf, aber siehe auch Abschnitt 6.3.

- Es genügt häufig nicht, nur den optimalen (maximalen) Score über alle Alignments zu berechnen, sondern man möchte auch ein zugehöriges optimales Alignment sehen. Hierzu verwendet man wieder *Backtracing*, das wir im Folgenden erläutern: Statt nur in einer DP-Matrix $S[i,j]$ den optimalen Score der Sequenzpräfixe der Längen i und j zu speichern, merken wir uns zusätzlich in einer weiteren Matrix $T[i,j]$, welche der Vorgängerzellen zum Maximum geführt hat.

Alignments hatten wir bereits in Abschnitt 5.2, Definition 5.7 definiert. Übersetzt man den in Lemma 5.10 durch (5.1) und (5.2) angedeuteten DP-Algorithmus auf die neue Situation, erhält man folgenden Code. Hierbei werden immer nur 2 Zeilen der DP-Matrix `S` (für Score, statt vorher D für Distanzen) im Speicher gehalten, nämlich die alte Zeile `So` und die aktuell zu berechnende Zeile `Si`. Gleichzeitig wird eine $(m+1)\times(n+1)$-Traceback-Matrix `T` erstellt, in der jede Zelle die Koordinaten der maximierenden Vorgängerzelle enthält. Die Ausgabe besteht aus dem optimalen Scorewert und einem optimalen Alignment, das aus `T` mit Hilfe einer `traceback`-Funktion gewonnen wird.

```
def align_global(x, y, score=standardscore, gap=-1):
    m, n = len(x), len(y)
    So = [None]*(n+1)                # alte Zeile der DP-Matrix
    Si = [j*gap for j in range(n+1)] # aktuelle Zeile der DP-Matrix
    T  = [[(0,0)] + [(0,j) for j in range(n)]]  # 0. Zeile Traceback
    for i in range(1,m+1):
        So,Si = Si,So
        Si[0] = i*gap
        T.append([(i-1,0)]+[(None,None)]*n)   # Traceback, i. Zeile
        for j in range(1,n+1):
            Si[j], T[i][j] = max(
                    (So[j-1]+score(x[i-1],y[j-1]), (i-1,j-1)),
                    (Si[j-1]+gap, (i,j-1)),
                    (So[j]+gap,   (i-1,j))
                    )
    return (Si[n], traceback(T, m, n, x, y))
```

Bei den Funktionsparametern ist `score` eine Funktion, die zu zwei Symbolen einen Scorewert zurückgibt; per Default hier auf `standardscore` gesetzt, die wie folgt definiert ist:

```
standardscore = lambda a,b: 1 if a==b else -1
```

Die `traceback`-Funktion setzt aus der Traceback-Matrix `T`, beginnend bei `(i,j)=(m,n)`, ein Alignment aus den Strings `x` und `y` zusammen. Das Alignment wird hier zurückgegeben als eine Liste von Paaren (2-Tupeln); jedes Paar entspricht einer Spalte des Alignments.

```
def traceback(T, i,j, x,y, GAP="-"):
    a = list()
    while True:
        ii,jj = T[i][j]
        if (ii,jj) == (i,j): break
        xx, yy = x[ii:i], y[jj:j]
        if i-ii < j-jj:
            xx += GAP*(j-jj-(i-ii))
        else:
            yy += GAP*(i-ii-(j-jj))
        a += [(xx,yy)]
```

```
12            i,j = ii,jj
13        return list(reversed(a))
```

6.2 Varianten des paarweisen Alignments

6.2.1 Ein universeller Alignment-Algorithmus

Wir kommen zurück auf den in Abschnitt 5.3 definierten Edit-Graphen, den wir im aktuellen Kontext (Maximierung mit allgemeiner Scorefunktion und Gapkosten statt Minimierung mir Einheitskosten) als *Alignment-Graph* bezeichnen. Die Unterschiede zu Definition 5.11 bestehen ausschließlich in den Kantengewichten (allgemeine Scores statt Einheitskosten).

Wir erläutern, wie sich der oben beschriebene DP-Algorithmus (globales Alignment) mit Hilfe des Graphen ausdrücken lässt. Diese Formulierung ist so allgemein, dass wir sie später auch für andere Varianten verwenden können, wenn wir die Struktur des Graphen leicht ändern.

Wir definieren $S(v)$ als den maximalen Score aller Pfade von $v_\circ$ nach v. Offensichtlich ist $S(v_\circ) = 0$. Für alle $v \neq v_\circ$ in topologisch sortierter Reihenfolge berechnen wir

$$S(v) = \max_{w:\, w \to v \in E} \{ S(w) + \text{score}(w \to v) \},$$

$$T(v) = \operatorname*{argmax}_{w:\, w \to v \in E} \{ S(w) + \text{score}(w \to v) \}.$$

Durch die Berechnung in topologisch sortierter Reihenfolge ist sichergestellt, dass $S(w)$ bereits bekannt ist, wenn es bei der Berechnung von $S(v)$ ausgewertet werden soll. Offensichtlich ist dazu notwendig, dass der Alignmentgraph keine gerichteten Zyklen enthält, was nach Definition 5.11 sichergestellt ist.

Den optimalen Score erhalten wir als $S(v_\bullet)$. Den optimalen Pfad erhalten wir durch Traceback von $v_\bullet$ aus, indem wir jeweils zum durch T gegebenen Vorgänderknoten gehen: $v_\bullet \to T(v_\bullet) \to T(T(v_\bullet)) \to \cdots \to T^k(v_\bullet) \to \cdots \to v_\circ$

6.2.2 „Free End Gaps“-Alignment

Um die Frage zu beantworten, ob sich zwei Stücke DNA (mit Fehlern) überlappen, benötigen wir eine Variante des globalen Alignments. Wir wollen Gaps am Ende bzw. am Anfang nicht bestrafen (daher der Name „free end gaps“). Wir benutzen beispielhaft folgendes Punkteschema: Gap am Ende: 0; innerer Gap: -3; Mismatch: -2; Match: $+1$. Wichtig ist, dass ein Match einen positiven Score bekommt, da sonst immer das leere Alignment optimal wäre.

Das „Free End Gaps“-Alignmentproblem lässt sich mit dem universellen Algorithmus aus Abschnitt 6.2.1 lösen, wenn man den Alignment-Graphen geeignet modifiziert. Ein Alignment kann (kostenfrei) entweder an einer beliebigen Position in der einen Sequenz und am Beginn der anderen Sequenz beginnen, oder am Beginn der einen Sequenz und an einer beliebigen Position der anderen Sequenz. Das heißt, es gibt (kostenfreie) Initialisierungskanten von $v_\circ$ nach $(i, 0)$ und $(0, j)$ für alle existierenden i und j. Analoges gilt für das Ende eines solchen

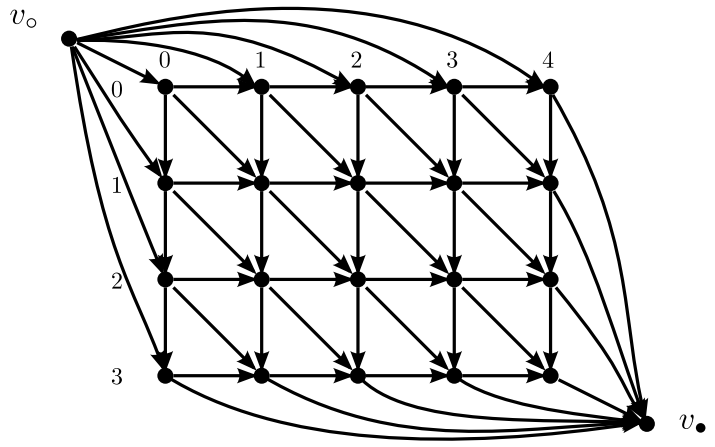


Abbildung 6.1: Alignment-Graph für ein „Free End Gaps“-Sequenzalignment.

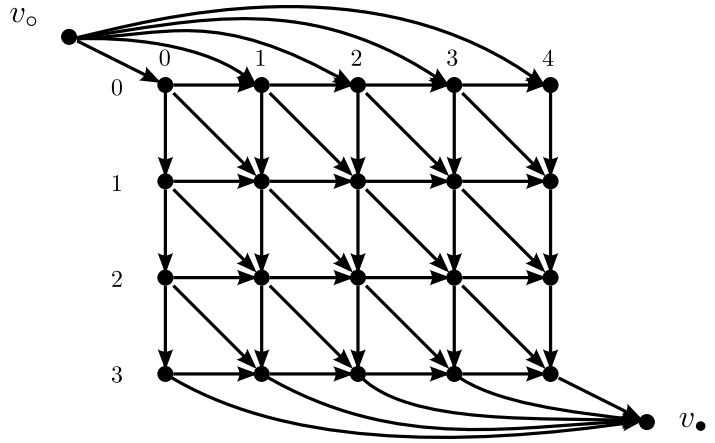


Abbildung 6.2: Alignment-Graphen für ein semi-globales Sequenzalignment, d.h. zur Suche nach dem besten approximativen Vorkommen eines Musters in einem Text.

Alignments: Es gibt kostenfreie Finalisierungskanten von (i, n) und (m, j) nach $v_\bullet$ für alle existierenden i und j. Mit kostenfrei ist gemeint, dass der zugehörige Score den Wert 0 hat. Der zugehörige Alignment-Graph findet sich in Abbildung 6.1.

6.2.3 Semiglobales Alignment (Mustersuche)

Eine weitere Alignment-Variante erhalten wir, wenn wir ein Muster in einem Text suchen und uns dabei für die beste Übereinstimmung interessieren. Dieses Problem wurde ausführlich in Abschnitt 5.5 behandelt; dabei wurden jedoch Einheitskosten vorausgesetzt. Die dort angewendeten Optimierungen (Ukkonen's Algorithmus, bit-parallele NFA-Simulation) lassen sich nicht mehr ohne Weiteres anwenden, wenn eine allgemeine Score-Funktion maximiert werden soll. Tatsächlich aber finden wir das beste Vorkommen (das mit dem höchsten Score) des Musters im Text, indem wir wieder den Alignment-Graphen modifizieren: Das Muster muss ganz gelesen werden, im Text können wir an jeder Stelle beginnen und aufhören. Wir

benötigen also Initialisierungskanten von $v_\circ$ nach $(0, j)$ für alle j und Finalisierungskanten von (m, j) nach $v_\bullet$ für alle j; siehe Abbildung 6.2. Es handelt sich um ein *semiglobales Alignment*, global aus Sicht des Muster, lokal aus Sicht des Textes.

6.2.4 Lokales Alignment

Beim lokalen Alignment von $s, t \in \Sigma^*$ suchen wir Teilstrings s' (von s) und t' (von t), die optimal global alignieren. Dieses Problem wurde von **?** formuliert und mittels DP gelöst. Ein naiver Ansatz wäre, für alle Paare von Teilstrings von s und t ein globales Alignment zu berechnen (Gesamtzeit: $\mathcal{O}(m^2n^2)$).

Um den universellen Algorithmus aus Abschnitt 6.2.1 nutzen zu können, stellen wir wieder einen Alignment-Graphen auf. Die nötige Änderung ist das Hinzufügen von vielen Initialisierungs- und Finalisierungskanten mit Score 0.

- Initialisierungskanten: $v_\circ \to (i, j)$ für alle $0 \leq i \leq m$ und $0 \leq j \leq n$
- Finalisierungskanten: $(i, j) \to v_\bullet$ für alle $0 \leq i \leq m$ und $0 \leq j \leq n$

Wir verzichten auf eine unübersichtliche Visualisierung, da es sehr viele Kanten gibt.

Explizite Formulierung Der Übersichtlichkeit halber schreiben wir den Algorithmus noch einmal explizit auf (auch wenn dies ein Spezialfall des universellen Algorithmus ist). Wir nehmen an, dass der Score für ein Gap negativ ist (Gaps werden also bestraft).

```
def align_local(x,y, score=standardscore, gap=-1):
    m, n = len(x), len(y)
    best, besti, bestj = 0, 0, 0
    So = [None]*(n+1)
    Si = [0 for j in range(n+1)]
    T  = [[(0,j) for j in range(n+1)]]
    for i in range(1,m+1):
        So,Si = Si,So
        Si[0] = 0
        T.append([(i,j) for j in range(n+1)])
        for j in range(1,n+1):
            Si[j], T[i][j] = max(
                    (So[j-1]+score(x[i-1],y[j-1]), (i-1,j-1)),
                    (Si[j-1]+gap, (i,j-1)),
                    (So[j]+gap,   (i-1,j)),
                    (0,           (i,j)  )
                    )
            if Si[j]>best:
                best,besti,bestj = Si[j],i,j
    return(best, traceback(T,besti,bestj,x,y))
```

Im Vergleich zum Code für das globale Alignment in Abschnitt 6.1 fällt auf, dass wir über 4 Möglichkeiten (statt dort 3) maximieren: Ein Scorewert von 0 ist immer dadurch möglich, dass man das Alignment (durch eine Initialisierungskante) bei (i, j) beginnen lässt. Da wir den Knoten $v_\circ$ nicht explizit modellieren, setzen wir den Traceback-Vorgänger `T[i][j]` in diesem Fall auf (i, j) selbst. Das Traceback beginnt beim lokalen Alignment nicht bei (m, n),

sondern bei dem Knoten mit dem besten Scorewert (gewissermaßen der Vorgänger von $v_\bullet$). Daher wird in jeder Zelle geprüft, ob sie den bisher besten Scorewert `best` übertrifft, und
⇑ 16.06.11 ggf. werden die Koordinaten dieser Zelle als `(besti,bestj)` gespeichert.

6.3 Allgemeine Gapkosten

⇓ 30.06.11

Bisher hat ein Gap der Länge ℓ einen Score von $g(\ell) = -\gamma \cdot \ell$, wobei $\gamma \geq 0$ die Gapkosten sind (negativer Score eines Gaps). Diese Annahme ist bei biologischen Sequenzen nicht sonderlich realistisch. Besser ist es, die Kosten für das Eröffnen eines Gaps höher anzusetzen als die für das Verlängern eines Gaps.

Im Fall ganz allgemeiner Gapkosten, d.h. wir machen keine Einschränkungen die Funktion g betreffend, lässt sich der Alignment-Graph so modifizieren, dass es Gap-Kanten nach (i, j) von allen (i', j) mit $i' < i$ und allen (i, j') mit $j' < j$ gibt. Es ergibt sich eine Laufzeit von $O\big(mn(n+m)\big)$, denn wir müssen in jedem der mn Knoten des Alignment-Graphen $\mathcal{O}(n+m)$ mögliche Vorgänger untersuchen.

Eine etwas effizientere Möglichkeit sind *konvexe Gapkosten*, dabei ist $g(\ell)$ konkav, z.B. $g(\ell) = -\log(1 + \gamma\ell)$. Konvexe Gapkosten führen (mit etwas Aufwand) zu einer Laufzeit von $O(mn \log(mn))$; den Algorithmus überspringen wir jedoch.

Eine relativ einfache, aber schon relativ realistische Modellierung erreichen wir durch *affine Gapkosten*

$$g(\ell) := -c - \gamma(\ell - 1) \text{ für } \ell \geq 1.$$

Dabei gibt die Konstante c die Gapöffnungskosten (gap open penalty) und γ die Gaperweiterungskosten (gap extension penalty) an.

6.3.1 Algorithmus zum globalen Alignment mit affinen Gapkosten

Wir entwickeln die Idee für einen effizienten Algorithmus für ein Alignment mit affinen Gapkosten mit einer Laufzeit von $O(mn)$. Wir wählen $c \geq \gamma \geq 0$, wobei c der *gap open penalty* und γ der *gap extend penalty* sind. Die Idee ist, jeden Knoten des Alignmentgraphen (und damit die DP-Tabelle) zu verdreifachen. Wir bezeichnen den optimalen Score eines Alignments der Präfixe $s[\ldots i-1]$ und $t[\ldots j-1]$ wieder mit $S(i, j)$, also

$$S(i,j) := \max\big\{ score(A) \,\big|\, A \text{ ist ein Alignment von } s[\ldots i-1] \text{ und } t[\ldots j-1] \big\}.$$

Zusätzlich definieren wir nun

$$V(i,j) := \max\left\{ score(A) \,\middle|\, \begin{array}{l} A \text{ ist ein Alignment von } s[\ldots i-1] \text{ und } t[\ldots j-1] \\ \text{das mit einem Gap (–) in } t \text{ endet} \end{array} \right\},$$

$$H(i,j) := \max\left\{ score(A) \,\middle|\, \begin{array}{l} A \text{ ist ein Alignment von } s[\ldots i-1] \text{ und } t[\ldots j-1] \\ \text{das mit einem Gap (–) in } s \text{ endet} \end{array} \right\}.$$

Dann ergeben sich folgende Rekurrenzen:

$$V(i,j) = \max\Big\{S(i-1,j) - c,\ V(i-1,j) - \gamma\Big\},$$
$$H(i,j) = \max\Big\{S(i,j-1) - c,\ H(i,j-1) - \gamma\Big\},$$
$$S(i,j) = \max\Big\{S(i-1,j-1) + \text{score}(s[i-1], t[j-1]),\ V(i,j),\ H(i,j)\Big\}.$$

Der Beweis dieser Rekurrenzen verläuft wie immer. Dabei beachtet man, dass die Vorgänger der V-Knoten stets oberhalb, die der H-Knoten stets links von (i,j) liegen. Es gibt jeweils die Möglichkeit, ein neues Gap zu beginnen (Score $-c$) oder ein schon bestehendes zu verlängern (Score $-\gamma$). Zu beachten ist, dass hierbei die klassische Traceback-Funktion nicht angewendet werden kann, da es sein, dass eine Gapweiterführung und ein Match den gleichen Score haben können.

6.4 Alignments mit Einschränkungen

Wir behandeln nun die Frage, wie Alignments gefunden werden können, wenn ein (oder mehrere) Punkte auf dem Alignmentpfad vorgegeben sind. Wir stellen also die Frage nach dem optimalen Pfad, der durch einen Punkt (i,j) verläuft. Die Lösung erhalten wir, indem wir zwei globale Alignments berechnen:

$$(0,0) \to (i,j) \text{ und } (i,j) \to (m,n)$$

Etwas schwieriger wird es, wenn wir dies für alle Paare (i,j) *gleichzeitig* berechnen wollen. Der naive Ansatz, für jeden Punkt (i,j) die beiden Alignments zu berechnen, bräuchte $\mathcal{O}(m^2n^2)$ Zeit. Der Schlüssel zu einer besseren Laufzeit liegt in der Beobachtung, dass der Score des optimalen Pfades von $(0,0)$ nach (i,j) für alle (i,j) bereits in der DP-Tabelle als $S(i,j)$ gespeichert ist, d.h. die Lösung „der ersten Hälfte“ des Problems können wir einfach ablesen.

Um auch den optimalen Score von (i,j) nach (m,n) einfach ablesen zu können, müssen wir nur den Algorithmus rückwärts ausführen (d.h., alle Kanten im Alignmentgraphen umkehren, und von (m,n) nach $(0,0)$ vorgehen). Auf diese Weise erhalten wir eine zweite Matrix $R(i,j)$ mit den optimalen Scores der Pfade $(m,n) \to (i,j)$. (Statt den Algorithmus neu zu implementieren, kann man auch den Standard-Algorithmus auf die reversen Strings anwenden, dann aber Vorsicht bei den Indizes!) Summieren wir die Matrizen zu $S + R$, erhalten wir als Wert bei (i,j) genau den optimalen Score aller Pfade, die durch (i,j) laufen. Insgesamt benötigt dies nur etwas mehr als doppelt soviel Zeit wie eine einfache Berechnung der DP-Tabelle.

(**TODO: suboptimale Alignments**)

6.5 Alignment mit linearem Platzbedarf

Ein Problem bei der Berechnung von Alignments ist der Platzbedarf von $\mathcal{O}(mn)$ für die Traceback-Matrix. Wir stellen die Methode von **?** vor, die mit Platz $\mathcal{O}(m+n)$ auskommt. Sie beruht auf einem Divide-and-Conquer-Ansatz.

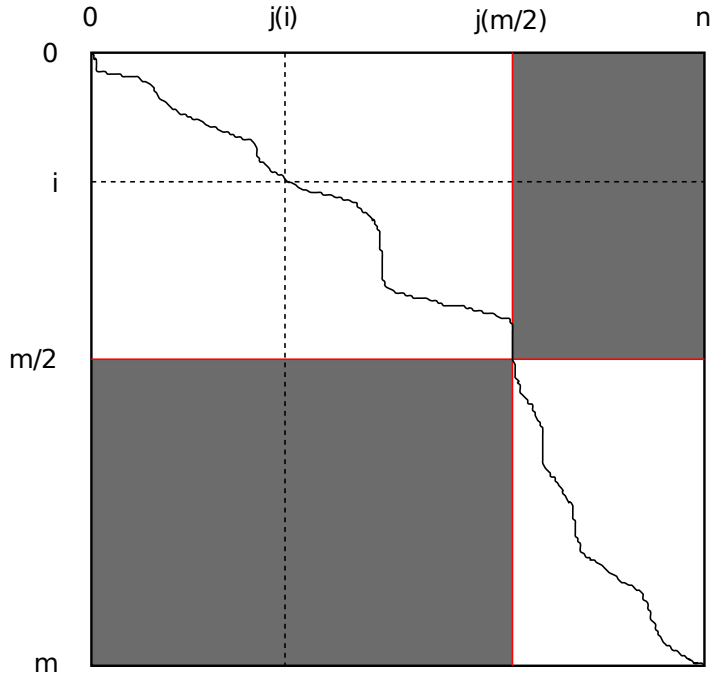


Abbildung 6.3: Zerlegung des Problems des globalen Sequenzalignments in zwei Teilprobleme. Die grauen Bereiche brauchen nicht weiter betrachtet zu werden.

6.5.1 Globales Alignment

Sei $j(i)$ der kleinste Spaltenindex j, so dass das optimale Alignment durch (i, j) geht. Finde $j(m/2)$, indem die obere Hälfte der Matrix S und die und die untere Hälfte der Matrix R berechnet wird. Addiere Vorwärts- und Rückwärts-Scores für Zeile $m/2$. Dabei ist

$$j(m/2) = \min \operatorname{argmax} \left\{ S\left(\frac{m}{2}, j\right) + R\left(\frac{m}{2}, j\right), j) : 0 \leq j \leq n \right\}$$

Dadurch haben wir das Problem in zwei Teilprobleme zerlegt, die wir rekursiv lösen können:

1. Alignment von $(0, 0) \rightarrow (m/2, j(m/2))$
2. Alignment von $(m/2, j(m/2)) \rightarrow (m, n)$

Platzbedarf: $\mathcal{O}(m) + \mathcal{O}(n) \in \mathcal{O}(m + n)$.

Zeitbedarf: $\mathcal{O}(mn + mn/2 + mn/4 + \cdots) \in \mathcal{O}(2mn) = \mathcal{O}(mn)$.

6.5.2 Lokales Alignment

Sobald bekannt ist, dass das optimale Alignment von $(i_{start}, j_{start}) \rightarrow (i_{end}, j_{end})$ läuft, müssen wir nur ein globales Alignment auf diesem Ausschnitt berechnen. Die Schwierigkeit besteht aber darin, diesen Bereich zu bestimmen.

Definiere $start(i,j)$ als die Koordinaten (i',j') des Startpunktes des optimalen Alignments, das in (i,j) endet. Die Idee besteht nun darin, für jede Zelle den Wert von $start(i,j)$ parallel zum Scorewert $S(i,j)$ zu verwalten.

$$S(i,j) = \max \begin{cases} 0 & \rightarrow start(i,j) = (i,j) \\ S(i-1,j-1) + score \begin{pmatrix} s_i \\ t_j \end{pmatrix} & \rightarrow start(i,j) = start(i-1,j-1) \\ S(i-1,j) - \gamma & \rightarrow start(i,j) = start(i-1,j) \\ S(i,j-1) - \gamma & \rightarrow start(i,j) = start(i,j-1) \end{cases}$$

Im Rahmen einer normalen Berechnung der S-Matrix berechnet man also auch die *start*-Matrix. Sobald man den Endpunkt (i^*,j^*) des optimalen lokalen Alignment gefunden hat, hat man nun auch $start(i^*,j^*)$ und damit den Anfangspunkt des Alignments. Auf das Rechteck zwischen $start(i^*,j^*)$ und (i^*,j^*) wendet man nun die Idee aus dem vorigen Unterabschnitt an (denn ein optimales lokales Alignment ist per Definition ein optimales globales Alignment auf optimal gewählten Teilstrings).

6.6 Statistik des lokalen Alignments

Wann ist der Score eines lokalen Alignments „hoch genug“, um beispielsweise auf eine biologisch bedeutsame Sequenzähnlichkeit zu schließen?

Auch zufällige Sequenzen haben Score größer gleich 0. Eine wichtige Frage ist in diesem Zusammenhang, wie die Verteilung der Scores von Alignments zufälliger Sequenzen aussieht. Annahmen:

- $score(Match) > 0$,
- $score(Gap) < 0$,
- Erwarteter Score für zwei zufällige Zeichen ist < 0.

Dann gilt für große t und $m,n \rightarrow \infty$:

$$\mathbb{P}(Score \geq t) \approx K \cdot m \cdot n \cdot e^{-\lambda t},$$

dabei sind $K > 0$ und $\lambda > 0$ Konstanten abhängig vom Score-System und dem Textmodell. Eine wesentliche Beobachtung ist, dass sich die Wahrscheinlichkeit einen Score $\geq t$ zu erreichen, ungefähr verdoppelt, wenn wir die Länge einer der Sequenzen verdoppeln. Eine andere Beobachtung ist, dass die Wahrscheinlichkeit exponentiell mit t fällt (für bereits große t).

6.7 Konzeptionelle Probleme des lokalen Alignments

Obwohl sich die Idee des lokalen Alignments nahezu univesell durchgesetzt hat und mit Erfolg angewendet wird, so gibt es dennoch mindestens zwei konzeptionelle Probleme, die bereits in der Definition des Alignmentscores liegen. Wir nennen erwähnen hier diese Probleme und geben einen Hinweis auf eine Umformulierung des Alignmentproblems.

Schatten-Effekt: Es kann dazu kommen, dass ein längeres Alignment mit relativ vielen Mismatches und InDels einen besseren Score hat als als ein kürzeres Alignments das aber fast ausschließlich aus Matches besteht. Möglicherweise ist das kürzere Alignment biologisch interessanter.

Mosaik-Effekt: Wenn sich einzelne Bereiche sehr gut alignieren lassen, dazwischen aber Bereiche liegen, die sich nicht oder schlecht alignieren lassen, kann es dazu kommen, dass ein langes Alignment, bestehend aus den guten und den schlechten Bereichen, einen besseren Score hat als die guten Bereiche jeweils alleine. Interessant wären aber eigentlich die guten kurzen Alignments separat.

Der Grund für Schatten- und Mosaik-Effekt liegt darin, dass die Zielfunktion additiv ist; der Score eines Alignments A ist die Summe der Scores der Spalten A_i:

$$Score(A) := \sum_{i=1,\ldots,|A|} Score(A_i).$$

Auf diese Weise können lange Alignments, auch wenn sie weniger gute Regionen enthalten, insgesamt einen besseren Score erhalten als kurze Alignments, auch wenn diese „perfekt“ sind.

Es liegt also nahe, nach einem Score zu suchen, der *längennormalisiert* ist. Andererseits kann man auch nicht einfach durch die Länge des Alignments teilen, denn dann wären sehr kurze Alignments (etwa aus nur einer Spalte) kaum zu schlagen. Eine pragmatische Lösung ist, eine Konstante $L > 0$ (z.B. $L = 200$) zu definieren und dazu den normalisierten Alignmentscore

$$NormScore_L(A) := \frac{1}{|A| + L} \cdot \sum_{i=1,\ldots,|A|} Score(A_i).$$

Gesucht ist also das Alignment A^*, das $NormScore_L(A)$ unter allen Alignments A maximiert. Die bekannten DP-Algorithmen können wir nicht anwenden, da diese auf der Additivität des Scores basieren.

Ein Lösungsvorschlag stammt von **?**. Sie führen einen neuen Parameter $\lambda \geq 0$ ein und maximieren

$$\begin{aligned} DScore_{\lambda,L}(A) &:= Score(A) - \lambda(|A| + L) \\ &= \sum_{i=1}^{|A|} \left(Score(A_i) - \frac{\lambda(|A| + L)}{|A|} \right) \end{aligned}$$

Dieser Score lässt sich also additiv schreiben, indem von der ursprünglichen additiven Scorefunktion für jede Spalte der Wert $(\lambda + \lambda L/|A|)$ abgezogen wird. Dieses Maximierungsproblem ist also äquivalent zu einem „normalen“ Alignmentproblem, wobei der Score abhängig von λ ist.

? konnten zeigen, dass es immer ein $\lambda \geq 0$ gibt, so dass die Lösung A^* des beschriebenen Problems die Lösung des längennormalisierten Alignmentproblems ist. Dieses (unbekannte) λ kann durch geeignete Kriterien mit einer binären Suche gefunden werden. In der Praxis werden dafür 3–5 Iterationen benötigt, so dass die Berechnung eines längennormalisierten Alignments etwa 5-mal langsamer ist als die eines „normalen“ Alignments. Die verwendete

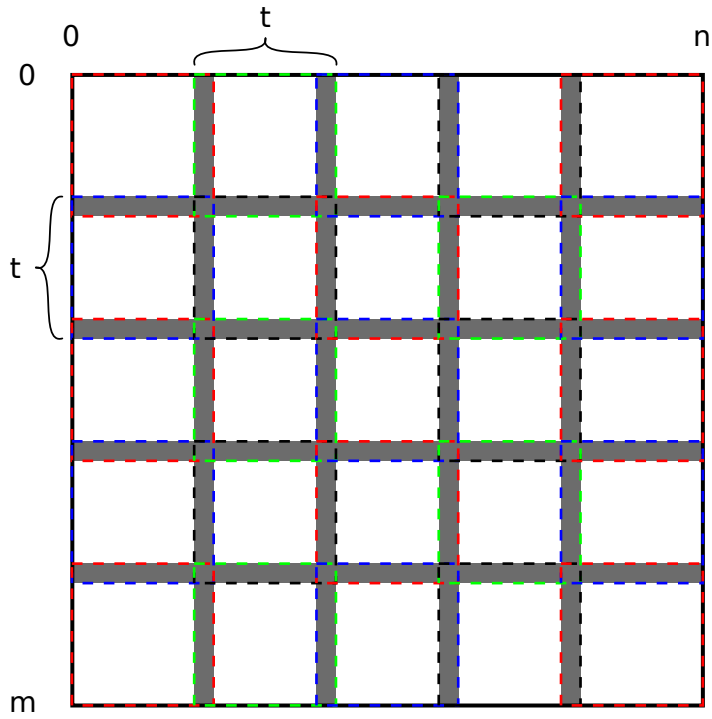


Abbildung 6.4: Four-Russians-Trick: Zerlegung der Alignmentmatrix in überlappende Blöcke der Größe $t \times t$. Der Überlappungsbereich ist jeweils grau hinterlegt. Zur besseren Unterscheidung sind die Blöcke verschieden eingefärbt.

Methode (Verfahren von Dinkelbach) ist ein aus der Optimierung bekanntes Standardverfahren, mit dem man ein fraktional-lineares Programm (die Zielfunktion ist der Quotient zweier linearer Funktionen) unter geeigneten Voraussetzungen als lineares Programm schreiben kann.

Trotz der genannten Vorteile (Vermeidung des Schatten-Effekts und des Mosaik-Effekts) hat sich das längennormalisierte Alignment bisher nicht durchgesetzt, vielleicht wegen der Beliebigkeit des Parameters L, mit dem man steuert, wie lang die optimalen Alignments in etwa sind. ⇑ 30.06.11

6.8 Four-Russians-Trick*

Wir erwähnen zum Abschluss noch eine Technik, mit der man die Berechnung der DP-Tabelle beschleunigen kann, indem nicht einzelne Zellen, sondern ganze Blöcke von Zellen (Quadrate) auf einmal berechnet werden.

Annahmen:

- Alphabetgröße σ ist klein: $\mathcal{O}(1)$,
- Wertebereich der Score-Funktion ist klein, z.B. $c = |\{-1, 0, +1\}| = 3$,
- Der Einfachheit halber: $m = \Theta(n)$.

Die Idee besteht darin, die DP-Tabelle in überlappende $t \times t$-Blöcke zu zerlegen und diese Blöcke vorzuberechnen (siehe Abbildung 6.4). Insgesamt existieren (nach Abzug des Startwerts oben links in jedem Block) nur $(c\sigma)^{2t}$ verschiedene Blöcke. Das resultiert in einem Algorithmus mit Laufzeit von $\mathcal{O}(n^2/t^2)$. Die Frage ist, wie t zu wählen ist, damit das Vorberechnen noch durchführbar ist. Oder asymptotisch ausgedrückt: die Größe der Tabelle soll nicht exponentiell in n wachsen, aber t soll trotzdem größer als $\mathcal{O}(1)$ sein, so dass die Laufzeit des Gesamtalgorithmus sich verbessert. Wir wählen $t = \log_{(c\sigma)^2} n$. Wir gehen vom RAM-Modell aus, d.h., die Blöcke können von 0 bis n durchnummeriert werden und ein Zugriff auf einen Block benötigt $\mathcal{O}(1)$ Zeit.

Es ergibt sich dann eine Laufzeit von $\mathcal{O}(n^2/(\log n)^2)$.

KAPITEL 7

Pattern-Matching-Algorithmen für Mengen von Patterns

Häufig ist es von Interesse, nicht einen bestimmten String, sondern einen beliebigen aus einer Menge von Strings in einem Text zu finden, etwa {Meier, Meyer, Maier, Mayer}.

Wir betrachten also eine Menge von Patterns $P = \{P^1, \ldots, P^K\}$ mit ggf. unterschiedlichen Längen; sei $m_k := |P^k|$ für $1 \leq k \leq K$ und ferner $m := \sum_k m_k$.

Natürlich kann man einen der bekannten Algorithmen einfach mehrmals aufrufen, für jedes Pattern einmal. Der folgende naive Algorithmus ruft KMP (oder einen anderen anzugebenden Algorithmus) für jedes Pattern in `P` auf. Die Ausgabe besteht aus allen Tripeln (i, j, k), so dass $T[i:j] = P^k$.

```
1 def naive(P,T, simplealg=simplepattern.KMP):
2     """yields all matches of strings from P in T as (i,j,k),
3     such that T[i:j]==P[k]"""
4     return ( match+(k,)                # append pattern number
5              for (k,p) in enumerate(P) # iterate over all patterns
6              for match in simplealg(p,T)
7     )
```

Es gibt jedoch effizientere Lösungen, bei denen der Text nicht K mal durchsucht werden muss.

Des Weiteren lässt sich oft noch eine spezielle Struktur des Patterns ausnutzen. Im obigen Meier-Beispiel lässt sich die Menge auch kompakter (in der Notation regulärer Ausdrücke) als `M[ae][iy]er` schreiben. Hier gibt es einzelne Positionen, an denen nicht nur ein bestimmter Buchstabe, sondern eine Menge von Buchstaben erlaubt ist.

Wir gehen in diesem Kapitel auf verschiedene spezielle Pattern-Klassen ein. Hier sind häufig bit-parallele Techniken die beste Wahl.

7.1 Zählweisen von Matches

Zunächst halten wir fest, dass sich Matches auf verschiedene Arten zählen lassen, wenn man es mit einer Menge von Patterns zu tun hat, die unter Umständen auch noch unterschiedlich lang sein können. Wir können die Anzahl der Vorkommen eines Musters P in einem Text T auf (mindestens) drei verschiedene Arten zählen:

- überlappende Matches; entspricht der Anzahl der Paare (i, j), so dass $T[i \dots j] \in P$, entspricht der Standard-Definition.
- Endpositionen von Matches; entspricht der Anzahl der j, für die es mindestens ein i gibt, so dass $T[i \dots j] \in P$.
- nichtüberlappende Matches; gesucht ist eine maximale Menge von Paaren (i, j), die Matches sind, so dass die Intervalle $[i, j]$ für je zwei verschiedene Paare disjunkt sind. Wir werden diese Zählweise nicht behandeln. Man kann sich aber fragen, wie man die einzelnen Algorithmen modifizieren muss, um nichtüberlappende Matches zu erhalten.

7.2 NFA: Shift-And-Algorithmus

Gesucht ist zunächst ein NFA, der $\Sigma^* P = \cup_{i=1}^{k} \Sigma^* P^k$ akzeptiert, also immer dann akzeptiert, wenn gerade ein Wort aus der Menge P gelesen wurde.

Einen solchen NFA zu konstruieren ist sehr einfach: Wir schalten die NFAs zu den einzelnen Strings parallel, d.h. es gibt so viele Zusammenhangskomponenten mit je einem Start- und Endzustand wie es Patterns gibt.

Genauer: Wir verwenden als Zustandsmenge $Q := \{ (i, j) \mid 1 \leq i \leq k, -1 \leq j < |P^i| \}$. Zustand (i, j) repräsentiert das Präfix der Länge $j+1$ von String $P^i \in P$. (In der Praxis zählen wir diese Paare in irgendeiner Reihenfolge auf.)

Wir bauen den Automaten so auf, dass stets alle Zustände aktiv sind, die zu einem Präfix gehören, das mit dem Suffix (gleicher Länge) des bisher gelesenen Textes übereinstimmt. Entsprechend ist $Q_0 = \{ (i, -1) \mid 1 \leq i \leq k \}$ die Menge der Zustände, die jeweils das leere Präfix representieren und $F = \{ (i, |P^i| - 1) \mid 1 \leq i \leq k \}$ die Menge der Zustände, die jeweils einen ganzen String repräsentieren. Ein Übergang von Zustand (i, j) nach $(i, j + 1)$ ist beim Lesen des „richtigen“ Zeichens $P^i[j + 1]$ möglich. Die Startzustände bleiben immer aktiv, was wir durch Selbstübergänge, die für alle Zeichen des Alphabets gelten, erreichen. Abbildung 7.1 illustriert diese Konstruktion an einem Beispiel.

Um die Implementierung von Shift-And anzupassen, hängen wir alle Strings nebeneinander in ein Register. Nun müssen wir jedoch genau buchhalten, welche Bits akzeptierende bzw. „Start“-Zustände repräsentieren. Man muss nur die Funktion, die die Masken berechnet, umschreiben; P ist jetzt ein Container von Strings.

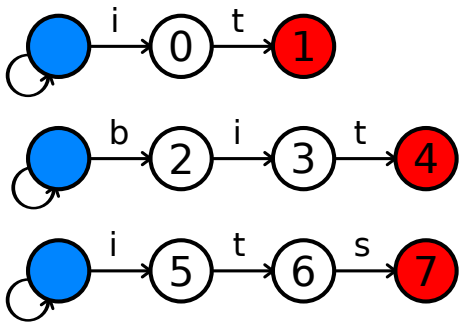


Abbildung 7.1: NFA zum parallelen Finden der Wörter aus der Menge { `it`, `bit`, `its` }. Die Startzustände sind blau, die akzeptierenden Zustände rot dargestellt. Auch wenn es mehrere Zusammenhangskomponenten gibt, handelt es sich um *einen* Automaten.

```
1  def ShiftAnd_set_masks(P):
2      """for a set P of patterns, returns (mask,ones,accepts), where
3      mask is a function such that mask(c) returns the bit-mask for c,
4      ones is the bit-mask of states after start states, and
5      accept is the bit-mask for accept states."""
6      mask, ones, accept, bit = dict(), 0, 0, 1
7      for p in P:
8          ones |= bit
9          for c in p:
10             if c not in mask: mask[c]=0
11             mask[c] |= bit
12             bit *= 2
13         accept |= (bit//2)
14     return (dict2function(mask,0), ones, accept)
15
16 def ShiftAnd(P,T):
17     (mask, ones, accept) = ShiftAnd_set_masks(P)
18     return  ( (None, i+1, None)
19             for (i,_) in ShiftAnd_with_masks(T, mask, ones, accept)
20     )
```

Der eigentliche Code der Funktion `ShiftAnd_with_masks` bleibt unverändert und wird direkt aus dem Modul `simplepattern`, das die Routinen für einfache Patterns enthält, importiert. Die `ShiftAnd`-Funktion gibt wie alle Funktionen in diesem Modul Tripel zurück (statt Paaren wie für einfache Patterns). Das Problem bei Mengen besteht allerdings darin festzustellen, welche(s) Pattern(s) genau gefunden wurde! Man müsste noch einmal analysieren, welche(s) Bit(s) gesetzt waren. Dies herauszufinden kann schnell die Einfachheit und Effizienz von Shift-And zu Nichte machen. Daher beschränkt man sich auf das Identifizieren der Endpositionen, ignoriert das von `ShiftAnd_with_masks` zurückgegebene Bitmuster und gibt nur die Endposition aus. Startposition und welches Pattern passt bleibt unbekannt. Hiermit können wir also nur die Endpositionen von Matches bekommen (Zählweise 2).

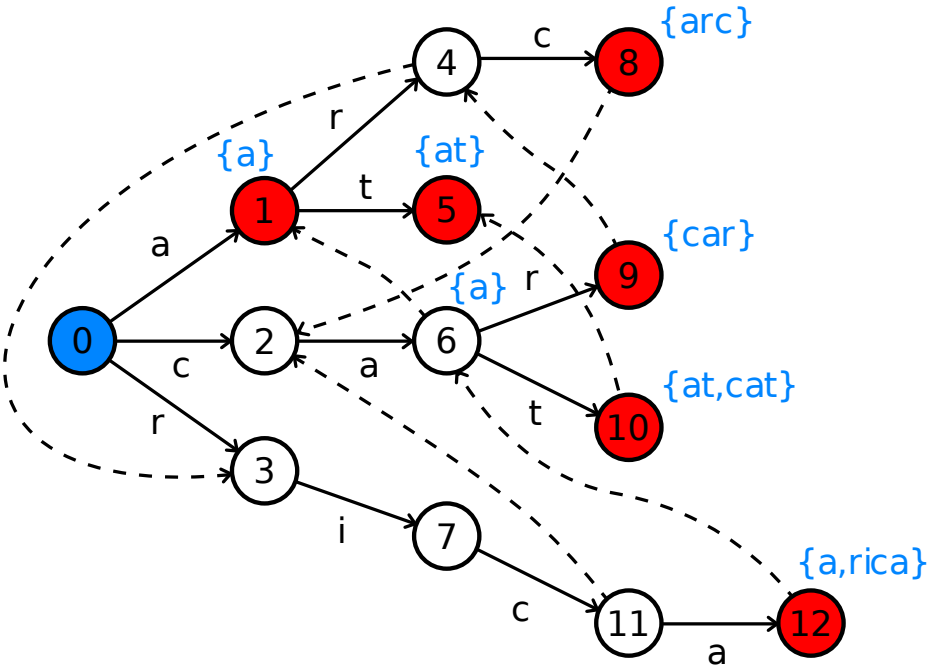


Abbildung 7.2: Trie über der Patternmenge $P = \{\texttt{cat}, \texttt{car}, \texttt{arc}, \texttt{rica}, \texttt{at}, \texttt{a}\}$. Zusätzlich ist die *lps*-Funktion durch gestrichelte Pfeile dargestellt (Pfeile in die Wurzel sind der Übersichtlichkeit halber weggelassen). Der Startzustand ist blau hinterlegt; die Zustände, die explizit Wörtern aus P entsprechen, sind rot dargestellt. Jeder Knoten ist mit der Menge der auszugebenden Wörter annotiert (in blau). All diese Komponenten zusammen bilden den Aho-Corasick-Automaten.

7.3 Aho-Corasick-Algorithmus

Offensichtlich kann man den KMP-Algorithmus (bzw. seine Variante mit DFAs) für jedes Pattern einzeln laufen lassen. Das führt zu einer Laufzeit von $\mathcal{O}(kn+m)$. In diesem Abschnitt wollen wir der Frage nachgehen, ob das auch in $\mathcal{O}(n+m)$ Zeit möglich ist.

Die Grundidee ist die selbe wie beim KMP-Algorithmus: wir führen im Verlauf des Algorithmus darüber Buch, was das längste Suffix des bisher gelesenen Textes ist, das ein Präfix eines Patterns in der Menge P ist. Wir benötigen also eine Datenstruktur, die uns in konstanter Zeit ermöglicht, ein weiteres Zeichen zu lesen und dabei diese Invariante erhält. Das ist mit einem *Trie* (Kunstwort aus *tree* und re*trie*val) zu bewerkstelligen. Dies führt auf den Algorithmus von **?**.

7.1 Definition (Trie)**.** Ein *Trie* über einer endlichen Menge von Wörtern $S \subset \Sigma^+$ ist ein kantenbeschrifteter Baum über der Knotenmenge Prefixes(S) mit folgender Eigenschaft: Der Knoten s ist genau dann ein Kind von t, wenn $s = ta$ für ein $a \in \Sigma$; die Kante $t \to s$ ist dann mit a beschriftet. Wir identifizieren also jeden Knoten v mit dem String, den man erhält, wenn man die Zeichen entlang des eindeutigen Pfades von der Wurzel bis zu v abliest.

Um nach einer Menge P von Wörtern gleichzeitig zu suchen, konstruieren wir also den Trie

zu P. Nach dem Lesen eines jeden Textzeichens befinden wir uns in dem Knoten, der dem längsten Suffix des Textes entspricht, das ein Präfix eines Wortes aus P ist. Wenn wir nun ein neues Zeichen a lesen und eine ausgehende Kante exisiert, die mit a beschriftet ist, dann folgen wir dieser Kante. Wir können den Trie zu einem nicht-deterministischen Automaten machen, indem wir einen Selbstübergang für alle Zeichen $a \in \Sigma$ im Startzustand anfügen. Um einen deterministischen Automaten zu erhalten, könnten wir wieder die aus der Automatentheorie bekannte Teilmengen-Konstruktion anwenden.

Eine hier effizientere(!) Möglichkeit, den äquivalenten deterministischen Automaten zu erhalten, besteht darin, wieder eine *lps*-Funktion einzuführen. Diese hilft uns, den richtigen Zielknoten zu finden, falls keine entsprechend beschriftete weiterführende Kante im NFA existiert.

Wir definieren

$$lps(q) := \text{argmax}\,\{\,|p| \mid p \in \text{Prefixes}(P),\ |p| < |q|, p = q[|q| - |p| \ldots]\,\}\,.$$

Im Unterschied zum KMP-Algorithmus sind q und $lps(q)$ hier keine Zahlen, sondern Strings. Also: $lps(q)$ verweist auf den Knoten, der zum längsten Präfix p eines Patterns in P gehört, so dass dieses Präfix gleichzeitig ein Suffix von q (aber nicht q selbst) ist.

Berechnung der *lps*-Funktion:

1. Nummeriere Knoten durch Breitensuche, beginnend beim Startknoten ε, der die Ordnungszahl 0 bekommt (siehe auchg Abbildung 7.2.
2. Es ist $lps[\varepsilon]$ nicht definiert.
3. Tiefe 1: $lps[c] = \varepsilon$ für alle c der Tiefe 1
4. Tiefe $j \geq 2$: In Knoten xa mit $x \neq \varepsilon$, setze $v := x$. Prüfe, ob Knoten $lps[v]$ eine ausgehende Kante a hat. Wenn ja, dann setze $lps[xa] := lps[v]a$. Wenn nein, gehe über zu $v := lps[v]$ bis entweder $lps[v]$ nicht mehr existiert oder es eine ausgehende Kante a von v gibt. Das Vorgehen ist analog zur lps-Berechnung bei KMP.

Um die Konstruktion des Aho-Corasick-Automaten zu vervollsändigen, benötigen wir für jeden Knoten noch die Menge der Wörter, die im gelesenen Text enden, wenn wir diesen Knoten erreichen. Wir nennen diese Abbildung $Q = \text{Prefixes}(P) \to 2^P$ die Ausgabefunktion. Die Ausgabefunktion kann in $\mathcal{O}(m)$ Zeit durch Rückverfolgen der *lps*-Links gewonnen werden.

Den Aho-Corasick-Algorithmus kann man wie den KMP-Alogrithmus auf Basis der *lps*-Funktion laufen lassen; dann dauert die Verarbeitung jedes Zeichens amortisiert $\mathcal{O}(1)$.

Zur Implementierung beschaffen wir uns zunächst eine Klasse `ACNode`, die einen Knoten des Aho-Corasick-Automaten implementiert.

```
1 class ACNode():
2
3     def __init__(self, parent=None, letter=None, depth=0, label=""):
4         self.targets = dict()  # children of this node
5         self.lps = None        # lps link of this node
6         self.parent = parent   # parent of this node
7         self.letter = letter   # letter between parent and this node
8         self.out = []          # output function of this node
```

```
        if parent == None:
            self.depth = depth # number of chars from root to this node
            self.label = label # string from root to this node
        else:
            self.depth = parent.depth + 1
            self.label = parent.label + letter

    def delta(self,a):
        """transit to next node upon processing character a"""
        q = self
        while q.lps != None  and  a not in q.targets:
            q = q.lps
        if a in q.targets: q = q.targets[a]
        return q

    def bfs(self):
        """yields each node below and including self in BFS order"""
        Q = collections.deque([self])
        while len(Q) > 0:
            node = Q.popleft()
            yield node
            Q.extend(node.targets.values())
```

Die Klasse `ACNode` besitzt folgende Attribute: Die Kinder des Knotens/Zustands werden im Dictionary `targets` gespeichert. Der Buchstabe an der Kante, die zu diesem Zustand führt, wird in `letter` gespeichert. Die lps-Links und Ausgabefunktion sind als `lps` und `out` verfügbar. Zusätzlich stellt ein Knoten noch die Funktion `bfs()` zur Verfügung, die alle Knoten des entsprechenden Teilbaums in der Reihenfolge einer Breitensuche zurückliefert. Dies geschieht durch Verwendung einer FIFO-Schlange (realisiert mit einer `deque` des `collections`-Moduls). Jeder Knoten kann über die Funktion `delta()` unter Berücksichtigung seines lps-Links darüber Auskunft geben, in welchen Zustand er beim Lesen eines bestimmten Zeichens übergeht. Beachtenswert ist die Analogie der `delta`-Funktion zu der des KMP-Algorithmus, nur dass wir auf eine explizite Nummerierung verzichtet haben.

Wir sind jetzt in der Lage, den AC-Automaten zur Menge P aufzubauen, indem wir zunächst das Trie-Gerüst erstellen und dann mit einem BFS-Durchlauf die lps-Funktion und die Ausgabefunktion erstellen.

```
def AC_build(P):
    """build AC autmaton for list of patterns P, return its root node."""
    # Build a root for the trie
    root = ACNode()
    # Build the trie, pattern by pattern
    for (i,p) in enumerate(P):
        node = root
        for a in p:
            if a in node.targets:
                node = node.targets[a]
            else:
                newnode = ACNode(parent=node, letter=a)
                node.targets[a] = newnode
                node = newnode
        node.out.append(i)
```

```
16      # Walk through trie in BFS-order to build lps
17      for node in root.bfs():
18          if node.parent == None: continue
19          node.lps = node.parent.lps.delta(node.letter) if node.depth>1 \
20                      else root
21          node.out.extend(node.lps.out)
22      return root
```

Der Rest ist nun einfach: Der eigentliche Algorithmus startet in der Wurzel, liest nacheinander jedes Textzeichen und führt die Übergangsfunktion `delta` des aktuellen Knoten mit dem gerade gelesenen Zeichen aus. Danach wird die Ausgabeliste des neuen Zustands durchgegangen und alle relevanten Muster mit Start- und Endposition ausgegeben. Die Hauptfunktion `AC` erstellt den AC-Automaten und ruft mit dessen Wurzel den eben beschriebenen Algorithmus auf.

```
1  def AC_with_automaton(P, T, root):
2      q = root
3      for (i,c) in enumerate(T):
4          q = q.delta(c)
5          for x in q.out:
6              yield (i-len(P[x])+1, i+1, x)
7
8  def AC(P,T):
9      ac = AC_build(P)
10     return AC_with_automaton(P, T, ac)
```

Die Gesamtlaufzeit (Aufbau des Automaten plus Durchmustern des Textes) ist $\mathcal{O}(m+n)$. Um die Laufzeit jedes einzelnen Schritts beim Durchmustern des Textes auf $\mathcal{O}(1)$ zu beschränken (z.B. für Realzeitanwendungen), müssen wir den Automaten explizit konstruieren, d.h. für jede Kombination aus Knoten und Buchstaben den Zielknoten vorberechnen. Das geschieht genauso wie bei KMP, indem wir für jeden Knoten alle `delta`-Werte in einem `dict()` speichern und dies in eine Funktion einwickeln.

7.4 Positions-Gewichts-Matrizen (PWMs) als Modelle für Transkriptionsfaktorbindestellen

In diesem Abschnitt lernen wir eine Möglichkeit kennen, eine (potenziell sehr große) Menge von Strings kompakt als ein Pattern zu beschreiben. Als Motivation dienen uns Transkriptionsfaktorbindestellen der DNA (siehe auch Anhang **??**). Transkriptionsfaktoren (Proteine, die die Transkription von DNA beeinflussen; TFs) binden physikalisch an bestimmte Stellen der DNA. Diese Stellen heißen konsequenterweise Transkriptionsfaktorbindestellen (TFBSs). Sie lassen sich durch das dort vorhandene Sequenzmotiv charakterisieren. Zum Beispiel binden die TFs der „Nuclear factor I“ (NF-I) Familie als Dimere an das DNA-IUPAC-Motiv `5'-TTGGCNNNNNGCCAA-3'`. Viele der Bindemotive, insbesondere wenn das Protein als Dimer bindet, sind identisch zu ihrem reversen Komplement. Ein solches Bindemotiv ist aber nicht exakt zu verstehen; schon am Beispiel sieht man, dass bestimmte Positionen nicht exakt vorgegeben sein müssen. Manche Stellen sind variabler als andere, bzw. die Stärke der Bindung verringert sich um so stärker, je unterschiedlicher die DNA-Sequenz zum Idealmotiv ist.

7.4.1 Definition vom PWMs

Wie beschreibt man nun das Idealmotiv einer TFBS und die Unterschiede dazu? Man könnte einfach die Anzahl der unterschiedlichen Symbole zählen. Es stellt sich aber heraus, dass dies zu grob ist. Stattdessen gibt man jedem Nukleotid an jeder Position eine Punktzahl (Score) und setzt beispielsweise einen Schwellenwert fest, ab dem eine hinreichend starke Bindung nachgewiesen wird. Alternativ kann man mit Methoden der statistischen Physik den Score-Wert in eine Bindungswahrscheinlichkeit umrechnen. Als Modell für Transkriptionsfaktorbindestellen haben sich daher Positions-Gewichts-Matrizen (position weight matrices, PWMs) etabliert.

7.2 Definition (PWM)**.** Eine Positions-Gewichts-Matrix (PWM) der Länge m über dem Alphabet Σ ist eine reellwertige $|\Sigma| \times m$-Matrix $S = (s_{c,j})_{c\in\Sigma, j\in\{0,\ldots,m-1\}}$. Jedem String $x \in \Sigma^m$ wird durch die PWM S ein Score zugeordnet, nämlich

$$\text{score}(x) := \sum_{j=0}^{m-1} S_{x[r],j}. \tag{7.1}$$

Man liest also anhand von x die entsprechenden Score-Werte der Matrix aus und addiert sie.

7.3 Beispiel (PWM)**.** In diesem Abschnitt verwenden wir ausschließlich das DNA-Alphabet $\{\texttt{A},\texttt{C},\texttt{G},\texttt{T}\}$. Sei

$$S = \begin{pmatrix} 0 & 1 & 2 \\ \hline -23 & 17 & -6 \\ -15 & -13 & -3 \\ -16 & 2 & -4 \\ 17 & -14 & 5 \end{pmatrix}.$$

Dann ist $\text{score}(\texttt{TGT}) = 17 + 2 + 5 = 24$. Die zu diesem Pattern am besten passende Sequenz (maximale Punktzahl) ist `TAT` mit Score 39. ♡

Die Matrizen für viele Transkriptionsfaktoren in verschiedenen Organismen sind bekannt und in öffentlichen oder kommerziellen Datenbanken wie JASPAR (`http://jaspar.genereg.net/`) oder TRANSFAC (`http://www.gene-regulation.com/pub/databases.html`) verfügbar. Die Matrizen werden anhand von beobachteten und experimentell verifizierten Bindestellen erstellt; auf die genaue Schätzmethode gehen wir in Abschnitt 7.4.3 ein. Zunächst betrachten wir aber das Pattern-Matching-Problem mit PWMs.

7.4.2 Pattern-Matching mit PWMs

Wir gehen davon aus, dass eine PWM S der Länge m, ein Text $T \in \Sigma^n$ und ein Score-Schwellenwert $t \in \mathbb{R}$ gegeben sind. Sei x_i das Fenster der Länge m, das im Text an Position i beginnt: $x_i := T[i \ldots i+m-1]$. Gesucht sind alle Positionen i, so dass $\text{score}(x_i) \geq t$ ist.

Naive Verfahren. Der naive Algorithmus besteht darin, das Fenster x_i für $i = 0, \ldots, n-m$ über den Text zu schieben, in jedem Fenster $\text{score}(x_i)$ gemäß (7.1) zu berechnen und mit t zu vergleichen. Die Laufzeit ist offensichtlich $\mathcal{O}(mn)$.

Eine Alternative besteht darin, alle 4^m möglichen Wörter der Länge m aufzuzählen, ihren Score zu berechnen, und aus den Wörtern mit Score $\geq t$ einen Trie und daraus den Aho-Corasick-Automaten zu erstellen. Mit diesem kann der Text in $\mathcal{O}(n)$ Zeit durchsucht werden. Das ist nur sinnvoll, wenn $4^m \ll n$.

Lookahead Scoring. Besser als der naive Algorithmus ist, den Vergleich eines Fensters abzubrechen, sobald klar ist, dass entweder der Schwellenwert t nicht mehr erreicht werden kann oder in jedem Fall überschritten wird. Dazu müssen wir wissen, welche Punktzahl im maximalen (minimalen) Fall noch erreicht werden kann. Wir berechnen also zu jeder Spalte j das Score-Maximum $M[j]$ über die verbleibenden Spalten:

$$M[j] := \sum_{k>j} \max_{c \in \Sigma} S_{c,k}.$$

Diese Liste der Maxima wird nur einmal für die PWM vorberechnet.

Beim Bearbeiten eines Fensters x kennen wir nach dem Lesen von $x[j]$ den partiellen Score

$$\text{score}_j(x) := \sum_{k=0}^{j} S_{x[k],k}.$$

Ist nun $\text{score}_j(x) + M[j] < t$, so kann t nicht mehr erreicht werden, und die Bearbeitung des Fensters wird erfolglos abgebrochen; ansonsten wird das nächste Zeichen evaluiert.

Abhängig von der Höhe des Schwellenwertes t lassen sich auf diese Weise viele Fenster nach wenigen Vergleichen abbrechen. Dieselbe Idee hilft bei einer Trie-Konstruktion, nur die Strings aufzuzählen, die tatsächlich den Trie bilden, statt aller 4^m Strings.

Permuted Lookahead Scoring. Lookahead Scoring ist vor allem dann effizient, wenn anhand der ersten Positionen eine Entscheidung getroffen werden kann, ob der Schwellenwert t noch erreichbar ist. Bei PWMs die Motiven entsprechen, bei denen weit links jedoch viele mehrdeutige Zeichen stehen (zum Beispiel `ANNNNGTCGT`; in den `N`-Spalten wären alle Werte in der PWM gleich, zum Beispiel Null) hilft diese Strategie nicht viel. In welcher Reihenfolge wir aber die Spalten einer PWM (und die zugehörigen Textfensterpositionen) betrachten, spielt keine Rolle. Wir können also die Spalten vorab so umsortieren, dass „informative“ Spalten vorne stehen. Mit derselben Permutation muss dann in jedem Textfenster gesucht werden. Wie kann man eine gute Permutation π finden? Das Ziel ist, die erwartete Anzahl der verarbeiteten Textzeichen pro Fenster zu minimieren.

Sei $\text{score}_j^\pi(x) := \sum_{k=0}^{j} S_{x[\pi_k],\pi_k}$ der partielle Score bis Position j einschließlich nach Umsortieren mit der Permutation π, wenn das Fenster mit Inhalt x betrachtet wird.

Sei Y^π die Zufallsvariable, die nach Umsortierung mit der Permutation π die Anzahl der Textzeichen in einem Fenster zählt, die gelesen werden müssen, bis entweder die Entscheidung getroffen werden kann, dass der Schwellenwert t nicht erreicht wird, oder das Fenster

vollständig abgearbeitet wurde. Für $0 \leq j < m-1$ ist man genau dann mit Spalte j fertig ($Y^\pi = j+1$), wenn $\text{score}_j^\pi(X) + M[j] < t$, aber noch $\text{score}_k^\pi(X) + M[k] \geq t$ für alle $k < j$. Dabei steht X für ein „zufälliges“ Textfenster der Länge m, also ein Textfenster, das der Hintergrundverteilung der Nukleotide des betrachteten Organismus folgt. Nach Spalte $m-1$ ist man in jedem Fall fertig; dann hat man m Zeichen gelesen ($Y^\pi = m$).

Nach Definition des Erwartungswertes ist also

$$\begin{aligned}\mathbb{E}[Y^\pi] &= \sum_{j=0}^{m-2} (j+1) \cdot \mathbb{P}(Y^\pi = j+1) + m \cdot \mathbb{P}(Y^\pi = m) \\ &= \sum_{j=0}^{m-2} (j+1) \cdot \mathbb{P}(\text{score}_k^\pi(X) + M[k] \geq t \text{ für } 0 \leq k < j \text{ und } \text{score}_j^\pi(X) + M[j] < t) \\ &\quad + m \cdot \mathbb{P}(\text{score}_k^\pi + M[k] \geq t \text{ für alle } 0 \leq k < m-1).\end{aligned}$$

Prinzipiell können diese Wahrscheinlichkeiten und damit der Erwartungswert für jede Permutation π berechnet werden. Das ist jedoch insbesondere für lange TFBSen zu aufwändig.

Daher behilft man sich mit Heuristiken. Ein gutes Argument ist wie folgt: Aussagekräftig sind insbesondere Spalten, in denen Scores stark unterschiedlich sind (also das Maximum sehr viel größer ist als der zweitgrößte Wert oder als der Mittelwert der Spalte.

7.4.3 Schätzen von PWMs

Eine PWM wird aus (experimentell validierten) Beispielsequenzen einer TFBS erstellt. Die erste vereinfachende Annahme ist, dass man alle Positionen unabhängig voneinander betrachten kann. An jeder Position j wird also gezählt, wie oft jedes Zeichen $c \in \Sigma$ beobachtet wurde. Dies liefert eine *Zählmatrix* $(N_{c,j})$, in der die „horizontalen“ Abhängigkeiten zwischen den einzelnen Positionen verlorengegangen sind, d.h. es ist nicht möglich, die Beispielsequenzen aus der Matrix zu rekonstruieren.

Teilt man die Zählmatrix elementweise durch die Gesamtzahl der Beobachtungen, erhält man eine *Wahrscheinlichkeitsmatrix* (auch *Profil*) $(P_{c,j})$. Hier summieren sich die Spalten zum Wert 1. Bei wenig Beispielsequenzen kommt es vor, dass an manchen Positionen manche Nukleotide gar nicht beobachtet wurden; dort ist $N_{c,j} = P_{c,j} = 0$. Nun muss es aber nicht unmöglich sein, dass man in Zukunft eine entsprechende Beobachtung macht. Eine Wahrscheinlichkeit von 0 bedeutet jedoch ein unmögliches Ereignis. Daher werden vor der Division (oder Normalisierung) zu allen Einträgen sogenannte *Pseudocounts* hinzugezählt (z.B. jeweils 1). Dies lässt sich auch lerntheoretisch und mit Methoden der Bayes'schen Statistik begründen. Man spricht dabei von *Regularisierung*. Im Ergebnis enthält das Profil nun kleine Wahrscheinlichkeiten für nicht beobachtete Ereignisse und Wahrscheinlichkeiten, die in etwa proportional zur beobachteten Häufigkeit sind, für beobachtete Ereignisse.

Der Sequenzbereich, in dem man sucht, wird normalerweise eine bestimmte Verteilung von Nukleotiden aufweisen, die nicht die Gleichverteilung ist. Angenommen, das Profil enthält an manchen Positionen mit großer Wahrscheinlichkeit ein G. Wenn nun auch die durchsuchte DNA-Region GC-reich ist, ist es nicht so überraschend ein G an dieser Stelle zu sehen wie in einer AT-reichen Region. Die Bewertung der Übereinstimmung eines Fensters x sollte daher immer relativ zum globalen „Hintergrund“ erfolgen. Wir gehen also davon

aus, dass ein Hintergrundmodell q gegeben ist, das jedem Symbol eine bestimmte Wahrscheinlichkeit zuordnet. Eine GC-reiche Region könnte dann (in der Reihenfolge `ACGT`) durch $q = (0.2, 0.3, 0.3, 0.2)$ beschrieben werden.

Die PWM erhält man nun als *log-odds* Matrix zwischen Profil und Hintergrundmodell, d.h. man betrachtet an jeder Stelle j das Verhältnis zwischen den Wahrscheinlichkeiten für Symbol c im Profil und im Hintergrundmodell, also $P_{c,j}/q_c$. Ist dieses Verhältnis größer als 1, dann steht das Zeichen an Position j für eine gute Übereinstimmung zur PWM. Äquivalent dazu betrachtet man den Logarithmus des Verhältnisses im Vergleich zu 0. Damit ist die PWM $S = (S_{c,j})$ durch

$$S_{c,j} := \log \frac{P_{c,j}}{q_c}$$

gegeben. Die Addition der Scores in einem Fenster entspricht einer Multiplikation der Wahrscheinlichkeiten über alle Positionen (Unabhängigkeitsannahme).

7.4.4 Sequenzlogos als Visualisierung von PWMs

Eine Matrix von Scores ist schlecht zu lesen. Man möchte aber natürlich wissen, wie Sequenzen, die einen hohen Score erreichen, typischerweise aussehen. Eine Möglichkeit besteht in der Angabe, eines *IUPAC-Konsensus-Strings*: Man wählt in jeder Spalte das Nukleotid mit höchstem Score aus, oder auch das IUPAC-Symbol für die 2/3/4 Nukleotide mit höchsten Scores, wenn die Unterschiede nicht sehr groß sind.

Eine ansprechendere Visualisierung gelingt durch sogenannte Sequenzlogos. Jede Position wird durch einen Turm oder Stapel aller Nukleotide dargestellt. Die Gesamthöhe des Turms ist proportional zur mit den Profilwahrscheinlichkeiten gewichteten Summe aller Scores in dieser Spalte. Diese beträgt

$$h_j = \sum_c P_{c,j} S_{c,j} = \sum_c P_{c,j} \log \frac{P_{c,j}}{q_c}.$$

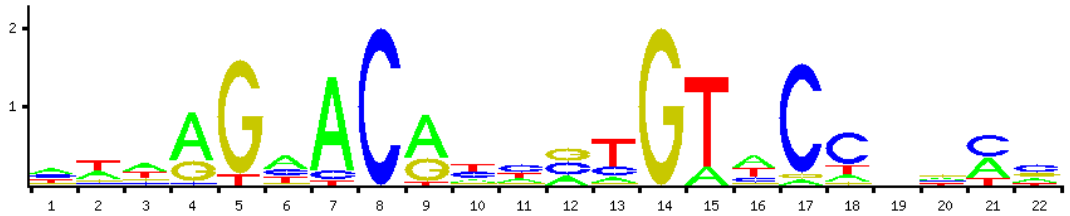
Dieser Ausdruck ist in der Informationstheorie auch bekannt als die *relative Entropie* zwischen der Verteilung $P_{\cdot,j}$ und der Verteilung q. Man kann beweisen, dass sie immer nichtnegativ ist.

Für die Visualisierung wird q oft einfach als Gleichverteilung auf dem DNA-Alphabet angenommen, so dass $q_c \equiv 1/4$. Nimmt man dann noch Logarithmen zur Basis 2 (dabei spricht man dann von Bits), so ist

$$h_j = \sum_c P_{c,j} \log_2 \frac{P_{c,j}}{1/4} = \sum_c P_{c,j}(2 + \log_2 P_{c,j}) = 2 + \sum_c P_{c,j} \log_2 P_{c,j}.$$

Da die Summe immer negativ ist, folgt $0 \leq h_j \leq 2$ [Bits].

Innerhalb dieser Gesamthöhe erhält dann jedes Symbol Platz proportional zu $P_{c,j}$. Die Höhe von Symbol c an Position j in der Visualisierung ist also $H_{c,j} = h_j \cdot P_{c,j}$. Ein Beispiel ist in Abbildung 7.3 gezeigt.



Oben: Sequenzlogo. Unten: Zugehörige Zählmatrix.

```
A  [ 9  9 11 16  0 12 21  0 15  4  5  6  3  0  4 11  1  3  6  6 10  5 ]
C  [ 7  2  3  1  0  6  2 24  0  9 11  9  5  0  0  5 22 16  7  5 11 11 ]
G  [ 2  3  3  7 22  1  0  0  8  2  2  9  1 24  0  1  1  1  5  9  0  6 ]
T  [ 6 10  7  0  2  5  1  0  1  9  6  0 15  0 20  7  0  4  6  4  3  2 ]
```

Abbildung 7.3: Sequenzlogo des Androgen-Transkriptionsfaktors der Ratte aus der JASPAR-Datenbank (`http://jaspar.genereg.net/cgi-bin/jaspar_db.pl?ID=MA0007.1&rm=present&collection=CORE`). Die x-Achse ist die Position (0 bis 21, dargestellt als 1 bis 22). Die y-Achse repräsentiert die Höhe in Bits.

7.4.5 Wahl eines Schwellenwerts

Beim Pattern-Matching mit einer PWM muss man sich für einen Score-Schwellenwert t entscheiden. Je größer t gewählt wird, desto weniger Strings erreichen diesen Schwellenwert, und die Suche wird spezifischer (und bei permuted lookahead scoring auch schneller). Wie aber sollte t gewählt werden?

Es gibt mehrere Kriterien, die sinnvoll sind, die sich aber teilweise widersprechen.

- Alle der in die PWM eingegangenen Beispielsequenzen sollten gefunden werden.
- Die Wahrscheinlichkeit, dass ein zufälliges Textfenster ein Match ist, sollte klein sein, vielleicht 0.01. Damit findet man dann (im Mittel) alle 100 Positionen einen „zufälligen“ Match.
- Die Wahrscheinlichkeit, dass ein nach dem Profil erstellter String ein Match ist, sollte groß sein, vielleicht 0.95.

Um t zu wie oben vorgegebenen Wahrscheinlichkeiten passend zu wählen, muss man in der Lage sein, die Score-Verteilung einer PWM unter verschiedenen Modellen (unter dem Hintergrundmodell q und unter dem PWM-Modell P selbst) zu berechnen. Dies ist effizient möglich, aber wir gehen an dieser Stelle nicht darauf ein. (**TODO: Vereweis auf Kapitel Sequenzstatistik.**)

KAPITEL 8

Weitere Planungen

Weitere Algorithmen zum Patternmatching (Suffix-, Factor-based). Analyse von Sunday. – geht nicht mit PAA?!

Exkurs: Abelian Pattern Matching, Analyse.

Reguläre Ausdrücke, Matching, mit Fehlern.

Approximative Suche: Alignment (Varianten inkl. Best Match), NFA-Ansatz mit epsilon-Uebergaengen Alignment mit Scores: NW / SW. Dotplots als Visualisierung. affine Gapkosten. Konvexe Gapkosten? linearer Speicherbedarf: Hirshberg.

Datenbanksuche. Statistik.

Indexdatenstrukturen für Teilstrings: Suffixbäume, Arrays, Konstruktion (Ukkonen, Skew, BwtWalk) Exaktes Pattern Matching auf Index-Datenstrukturen. BWT, auch: Backward Search (statt Affix Trees). Q-gram-Index, gapped q-Grams, seeded Alignment, filter, sensitivity.

Next generation sequencing. Technologien. Color space. Read mapping.

Kompression, bzip2. Compression Boosting.

Multiples Alignment. Heuristiken, exakte Algorithmen, Praxis. SP-Score, Komplexität, Carillo-Lipman-Schranke. Star Alignment. Progressives Alignment (CLUSTAL/MUSCLE).

HMMs: Definition, Algorithmen, Spracherkennung, Modellierung von Proteinfamilien, Gene Finding. Anwendung aus Fachprojekt? Massenspektren für Proteinfamilien. Viterbi, Forward, Baum-Welch.

DNA-Design, Oligo-Stabilität,

RNA: RNA-Faltung (Nussinov, Energiemodell bei Zuker). Erkennen von miRNAs?

Subsequenz-Kombinatorik?

Gesten als Sequenzen?

Weitere Iterationen der Vorlesung

Sequenzstatistik: Compound-Poission Approximation. Fehlerschranken?

ANHANG A

Molekularbiologische Grundlagen

Das Ziel der Bioinformatik ist das Lösen von biologischen Problemen mit Hilfe von Informatik, Mathematik und Statistik. Um die zu Grunde liegenden Fragestellungen verständlich zu machen, geben wir in diesem Kapitel eine kurze Einführung in einige wichtige Beriffe und Techniken der molekularen Genetik. Für eine weitergehende Beschäftigung mit dem Thema sei das Buch von **?** empfohlen. Dieses Buch hat auch beim Schreiben dieses Kapitels als Quelle gedient.

A.1 Desoxyribonukleinsäure (DNA)

Die Entdeckung der molekularen Struktur der *Desoxyribonukleinsäure (DNA)* zählt zu den wichtigen wissenschaftlichen Durchbrüchen des 20. Jahrhunderts (**???**)[1]. DNA ist ein Doppelhelix-förmiges Molekül, dessen Bestandteile zwei antiparallele Stränge aus einem Zucker-Phosphat-Rückgrat und Nukleotid-Seitenketten sind. Die „Sprossen“ der Leiter werden aus zwei komplementären Basen gebildet; Adenin (`A`) ist immer mit Thymin (`T`) gepaart und Cytosin (`C`) immer mit Guanin (`G`). Chemisch gesehen bilden sich Wasserstoffbrücken zwischen den jeweiligen Partnern; das Ausbilden dieser Bindungen bezeichnet man auch als Hybridisierung. Abstrakt können wir ein DNA-Molekül einfach als eine Sequenz (String) über dem 4-Buchstaben-Alphabet $\{\,\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\,\}$ auffassen; die Sequenz des zweiten Strangs erhalten wir durch Bildung des reversen Komplements. Da DNA fast immer doppelsträngig auftritt, ist die DNA-Sequenz `AAGCCT` äquivalent zu ihrem reversen Komplement `AGGCTT`.

DNA enthält durch die Abfolge der Basen also Informationen. In jeder bekannten Spezies (von Bakterien über Pflanzen bis hin zu Tieren und Menschen) wird DNA als Träger der

[1] Zur Rolle Rosalind Franklins bei der Bestimmung der DNA-Struktur sei auf die Artikel von **?** und **?** hingewiesen.

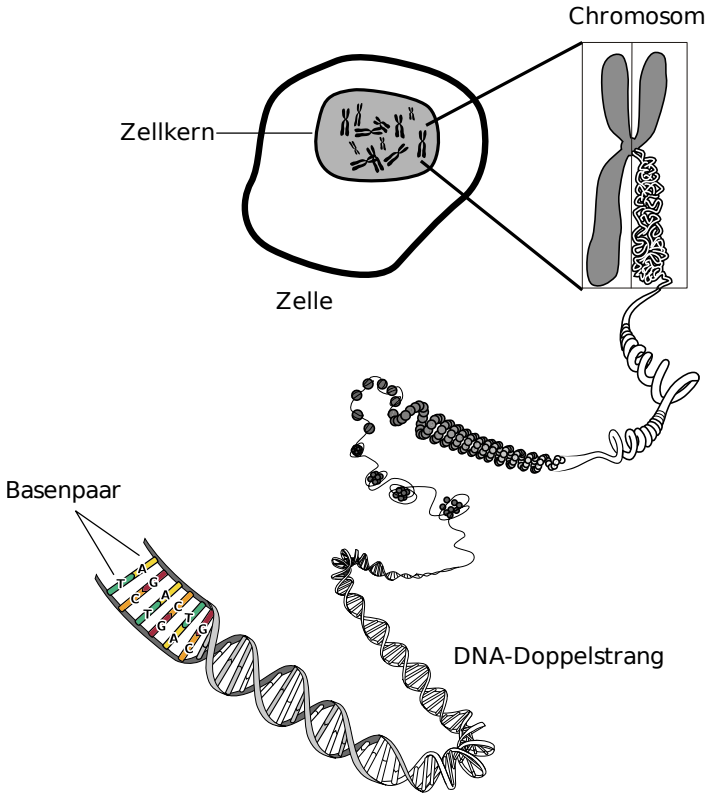


Abbildung A.1: Dieses Bild zeigt schematisch die Organisation einer eukaryotischen Zelle. Man sieht, dass sich mit Zellkern verschiedene Chromosomen befinden. Jedes Chromosom besteht aus einem langen DNA-Doppelstrang, der in verschiedenen Organisationsebenen immer weiter aufgerollt ist. Ein DNA-Strang besteht aus einem Zucker-Phosphat-Rückgrat und einer Folge verschiedener Nukleobasen (farbig). Diese bilden Paare mit den Basen des reversen Strangs. (Dieses Bild ist gemeinfrei. Quelle: `http://de.wikipedia.org/w/index.php?title=Datei:Chromosom.svg`)

Erbinformationen verwendet. Das bedeutet, dass die kodierte Buchstabenfolge vererbt wird und den „Bauplan“ für das jeweilige Individuum enthält. Die gesamte DNA-Sequenz eines lebenden Organismus bezeichnet man als *Genom*. In (fast) allen Zellen eines Organismus befindet sich eine Kopie des Genoms. Man unterscheidet Lebewesen, bei denen die Zellen einen Zellkern besitzen, die *Eukaryoten*, und solche bei denen das nicht der Fall ist, die *Prokaryoten*. Bakterien sind zum Beispiel Prokaryoten, während alle Tiere und Pflanzen Eukaryoten sind. Im folgenden behandeln wir nur eukaryotische Zellen.

Das Genom besteht in der Regel aus mehreren DNA-Molekülen. Diese Teilstücke sind mit Hilfe spezieller Proteine sehr eng „aufgerollt“. Den gesamten Komplex aus einem langen DNA-Faden und Strukturproteinen bezeichnet man als *Chromosom*. In (fast) jeder Zelle eines Organismus befindet sich im Zellkern ein kompletter Satz an Chromosomen. Abbildung A.1 illustriert diese Sachverhalte.

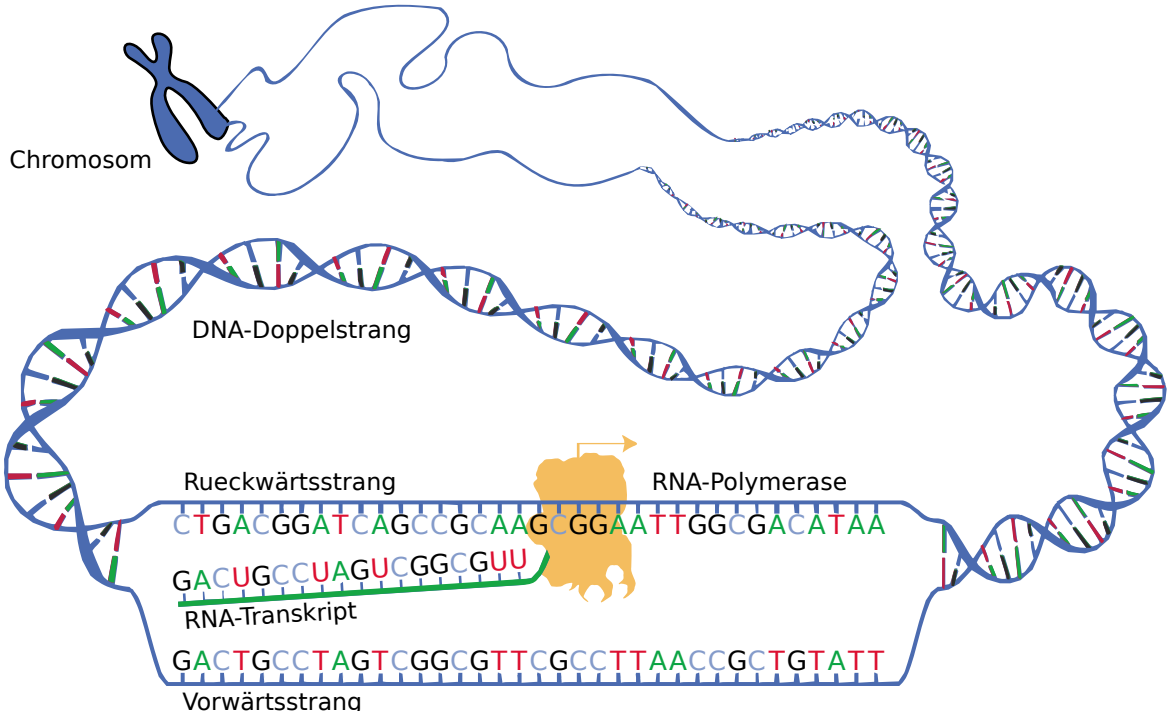


Abbildung A.2: RNA-Polymerase erstellt eine Kopie des Vorwärtsstrangs, indem RNA erzeugt wird, die revers-komplementär zum Rückwärtsstrang ist. (Dieses Bild ist gemeinfrei. Quelle: `http://de.wikipedia.org/w/index.php?title=Datei:DNA_transcription.svg`)

A.2 Ribonukleinsäure (RNA)

Die *Ribonukleinsäure (RNA)* ist ein der DNA sehr ähnliches Kettenmolekül. Neben chemischen Unterschieden im Rückgrat des Moleküls besteht der wichtigste Unterschied darin, das RNA (fast) immer als Einzelstrang auftritt. Ein weiterer Unterschied ist, dass RNA statt der Base Thymin (T) die Base Uracil (U) enthält. Wir können einen RNA-Strang somit als String über dem Alphabet $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}$ auffassen.

Ein Vorgang von zentraler Bedeutung ist die Abschrift von DNA in RNA. Diesen Prozess bezeichnet man als *Transkription*. Dabei lagert sich ein spezielles Protein, die *RNA-Polymerase*, an den DNA-Doppelstrang an und trennt die Bindungen zwischen den gegenüberliegenden Strängen temporär und örtlich begrenzt auf. An dem nun freiliegenden Teil des Rückwärtsstrangs wird ein zum diesem komplementärer RNA-Strang hergestellt. Die Reaktion schreitet in Rückwärtsrichtung fort; die RNA-Polymerase „rutscht“ dabei an der DNA entlang. Die Bindung zwischen RNA und DNA-Strang wird durch die RNA-Polymerase sofort wieder getrennt. Die beiden DNA-Stränge binden wieder aneinander. Zurück bleibt ein Kopie der DNA in Form eines RNA-Moleküls und der unveränderte DNA-Strang. Eine schematische Darstellung findet sich in Abbildung A.2.

Wichtig ist, dass immer nur ganz bestimmte Regionen der DNA abgeschrieben werden. Üblicherweise bezeichnet man einen solchen Abschnitt als *Gen*. Eine präzise, allgemein ak-

Name	Abkürzung (3 Buchstaben)	Abkürzung (1 Buchstabe)
Alanin	Ala	A
Arginin	Arg	R
Asparagin	Asn	N
Asparaginsäure	Asp	D
Cystein	Cys	C
Glutamin	Gln	Q
Glutaminsäure	Glu	E
Glycin	Gly	G
Histidin	His	H
Isoleucin	Ile	I
Leucin	Leu	L
Lysin	Lys	K
Methionin	Met	M
Phenylalanin	Phe	F
Prolin	Pro	P
Serin	Ser	S
Threonin	Thr	T
Tryptophan	Trp	W
Tyrosin	Tyr	Y
Valin	Val	V

Tabelle A.1: Übersicht über die in Proteinen verwendeten Aminosäuren und ihre ein- bzw. dreibuchstabigen Abkürzungen.

zeptierte Definition des Begriffs Gen gibt es allerdings nicht. Die Diskussion um diesen Begriff ist in vollem Gange (**??**).

A.3 Proteine

Neben den Nukleinsäuren (also DNA und RNA) sind die *Proteine* eine weitere Klasse von biologisch wichtigen Kettenmolekülen. Proteine machen in der Regel ungefähr die Hälfte der Trockenmasse (also des Teils, der nicht Wasser ist) einer Zelle aus. Sie übernehmen sehr unterschiedliche Funktionen: sie fungieren als Enzyme, Antikörper, Membranproteine, Transkriptionsfaktoren, etc. Die Liste ließe sich noch weit fortsetzen. Kurz gesagt: Proteine sind extrem wichtig für das funktionieren von Zellen und ganzen Organismen.

Die Bestandteile von Proteinen sind *Aminosäuren*. Jede Aminosäure enthält eine Carboxyl- und eine Aminogruppe. Zwei Aminosäuren können mit Hilfe einer sogenannten Peptidbindung miteinander verbunden werden. Dabei verbindet sich die Carboxylgruppe der ersten Aminosäure mit der Aminogruppe der zweiten Aminosäure. Das resultierende Molekül (ein sogenanntes Dipeptid) hat also auf der einen Seite eine freie Aminogruppe und auf der anderen Seite eine freie Carboxylgruppe. Das Ende mit der freien Aminogruppe bezeichnet man als N-Terminus und das Ende mit der freien Carboxylgruppe als C-Terminus. Dort können nun weitere Aminosäuren über Peptidbindungen angekoppelt werden um eine lange Ketten – ein Protein – zu bilden. Die meisten Proteine sind 50 bis 2000 Aminosäuren lang.

Aminosäure	Codons
Alanin	GCA, GCC, GCG, GCU
Arginin	AGA, AGG, CGA, CGC, CGG, CGU
Asparagin	AAC, AAU
Asparaginsäure	GAC, GAU
Cystein	UGC, UGU
Glutamin	CAA, CAG
Glutaminsäure	GAA, GAG
Glycin	GGA, GGC, GGG, GGU
Histidin	CAC, CAU
Isoleucin	AUA, AUC, AUU
Leucin	CUA, CUC, CUG, CUU, UUA, UUG
Lysin	AAA, AAG
Methionin	AUG
Phenylalanin	UUC, UUU
Prolin	CCA, CCC, CCG, CCU
Serin	AGC, AGU, UCA, UCC, UCG, UCU
Threonin	ACA, ACC, ACG, ACU
Tryptophan	UGG
Tyrosin	UAC, UAU
Valin	GUA, GUC, GUG, GUU
Stopp-Signal	UAA, UAG, UGA

Tabelle A.2: Genetischer Code: Zuordnung von Aminosäuren zu Codons. Wird bei der Translation ein Stopp-Codon erreicht, endet der Prozess.

Die Vielfalt unter den Proteinen entsteht durch die Kombination von 20 verschiedenen Aminosäuren (siehe Tabelle A.1). Die Aminosäuren unterscheiden sich durch die sogenannten *Seitenketten*. Das sind chemische Komponenten, die neben der Amino- und Carboxylgruppe an dem zentralen Kohlenstoffatom der Aminosäure befestigt sind. Warum im Laufe der Evolution genau diese 20 Aminosäuren als Komponenten aller Proteine ausgewählt wurden ist bisher ungeklärt, denn chemisch lassen sich noch viel mehr Aminosäuren herstellen. Fest steht, dass diese Auswahl sehr früh im Laufe der Evolution stattgefunden haben muss, denn sie ist allen bekannten Spezies gemein.

Die Herstellung von Proteinen erfolgt durch die *Translation* von *mRNAs*. Die Abkürzung mRNA steht für „messenger RNA“ (Boten-RNA). Sie rührt daher, dass die mRNA als Zwischenprodukt bei der Herstellung von Proteinen dient. Nachdem ein Gen in eine mRNA transkribiert wurden (vgl. Abschnitt A.2), wird diese mRNA aus dem Zellkern exportiert und danach in ein Protein übersetzt. Dabei kodieren immer drei Basenpaare der RNA eine Aminosäure. Ein solches Tripel nennt man *Codon*. In Tabelle A.2 ist zu sehen, welche Codons jeweils welche Aminosäure kodieren. Diese Zuordnung bezeichnet mal als *genetischen Code*. Da es 20 Aminosäuren gibt, aber $4^3 = 64$ mögliche Codons, gibt es Aminosäuren, die durch mehrere Codons dargestellt werden.

Die Reihenfolge der Aminosäuren bestimmt die Struktur eines Proteins; deshalb nennt man sie auch *Primärstruktur*. Nach der Herstellung eines Proteins verbleibt es allerdings

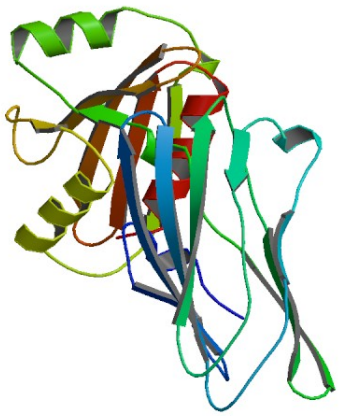


Abbildung A.3: Schematische Darstellung eines Proteins. Jede α-Helix ist als Spirale, jedes β-Faltblatt als Pfeil dargestellt. (Dieses Bild ist gemeinfrei. Quelle: `http://en.wikipedia.org/wiki/File:PBB_Protein_AP2A2_image.jpg`)

nicht in seiner Form als langer Strang, sondern faltet sich. Man kann sich das Vorstellen wie ein Faden, der sich „verknäult“. Dies geschiet jedoch nicht zufällig, sondern ist bestimmt von den Eigenschaften der Aminosäuren-Seitenketten. So gibt es zum Beispiel Aminosäuren, die hydrophob sind; andere sind hydrophil; wieder andere sind positiv/negativ geladen. So wirken verschiedene Kräfte zwischen den einzelnen Aminosäuren und der umgebenden (wässrigen) Zellflüssigkeit sowie zwischen den Aminosäuren untereinander. Diese Kräfte zwingen das Protein in eine ganz bestimmte dreidimensionale Struktur. Das Annehmen dieser Struktur heißt *Proteinfaltung*. Dabei gibt es bestimmte „Bausteine“, die in vielen Proteinen vorkommen, nämlich die α-Helix und das β-Faltblatt. Diese Elemente nennt man *Sekundärstruktur*, während die vollständige 3D-Struktur als *Tertiärstruktur* bezeichnet wird (siehe Abbildung A.3). Es kommt vor, dass mehrere Proteine sich zu einem Proteinkomplex zusammenschliessen. Ein solcher Komplex heißt dann *Quatärstruktur*.

A.4 Das zentrale Dogma der Molekularbiologie

Wir wollen die wichtigsten Prozesse noch einmal zusammenfassen: Abschnitte der DNA (die Gene) werden transkribiert; das Resultat ist mRNA. Aus jeder mRNA wird bei der Translation ein Protein hergestellt. Somit kann man ein Gen als „Bauanleitung für ein Protein“ bezeichnen. Anders ausgedrückt werden in der DNA kodierte Informationen in RNA und von dort in Proteine übersetzt. Dieses Prinzip ist so wichtig, dass man es als *zentrales Dogma der Molekularbiologie* bezeichnet.

Man muss jedoch anmerken, dass es auch Ausnahmen von dieser Regel gibt. So weiss man heute, dass zum Beispiel auch RNA in DNA übersetzt werden kann. Ein Prinzip, dass sich Retroviren zu Nutze machen. Dennoch hat das zentrale Dogma nichts an seiner Wichigkeit eingebüßt, denn es beschreibt nach wie vor einen sehr wichtigen, wenn nicht den wichtigsten, Prozess in Zellen.

A.5 Genregulation

Jede Zelle eines Organismus enthält eine exakte Kopie des Genoms. Wie ist es dann möglich, dass es so verschiedene Zellen wie Haut-, Leber, Nerven- oder Muskelzellen gibt? Die Antwort liegt in der *Regulation der Gene*. Die meisten Gene werden in unterschiedlichen Geweben unterschiedlich stark verwendet; d. h. es kann zum Beispiel sein, dass ein bestimmtes Protein in Nervenzellen in sehr großer Zahl hergestellt wird, während man es in Hautzellen gar nicht antrifft. Man sagt dann, das dazugehörige Gen ist unterschiedlich stark *exprimiert*.

Es gibt viele verschiedene Mechanismen, mit denen eine Zelle die Genexpression steuert. Ein wichtiger Mechanismus, auf den wir uns zunächst beschränken wollen, ist die Regulation der Transkription durch *Transkriptionsfaktoren*. Das sind spezielle Proteine, die an DNA binden können. Diese Bindung erfolgt sequenzspezifisch, das bedeutet ein bestimmter Transkriptionsfaktor bindet nur an eine bestimmte DNA-Sequenz. Solche Sequenzen befinden sich üblicherweise vor Genen auf der DNA in den sogenannten *Promoter-Regionen*. Ein DNA-Abschnitt an den ein Transkriptionsfaktor binden kann bezeichnet man als *Transkriptionsfaktorbindestelle*. Sobald sich ein Transkriptionsfaktor an eine Bindestelle angelagert hat, beeinflusst er die Transkription des nachfolgenden Gens. Es gibt sowohl Transkriptionsfaktoren, die das Ablesen eines Gens hemmen, als auch solche, die es fördern.

ANHANG B

Molekularbiologische Arbeitstechniken

Voraussetzung für die moderne Molekularbiologie sind zahlreiche experimentelle Techniken zur DNA-Bearbeitung:

Kopieren mit PCR (polymerase chain reaction, Polymerasekettenreaktion): Zyklen von (Denaturierung, Priming, Erweiterung) und Zugabe von dNTP führen zu exponentieller Vermehrung der DNA.

Kopieren durch Klonierung in Vektoren: Einfügen von DNA-Abschnitten in Vektoren (DNA sich vermehrender Organismen, z.B. Viren, Bakterien)

Cut+Paste mit Restriktionsenzymen: Restriktionsenzyme sind Proteine, die einen DNA-Doppelstrang an charakteristischen Sequenz-Motiven zerschneiden Beispiele: PvuII (CAG|CTG) oder BamHI (G|GATCC); der Strich (|) kennzeichnet die Schnittstelle. Zusammenkleben durch Hybridisierung und Ligation. Viele der Schnittstellen sind palindromisch, d.h. revers komplementär zu sich selbst. Wenn nicht symmetrisch geschnitten wird, ergeben sich klebende Enden (sticky ends), so dass man die DNA-Fragmente in anderer Reihenfolge wieder zusammensetzen kann.

Längenbestimmung mit Gel-Elektrophorese: DNA ist negativ geladen, DNA-Fragmente wandern zu einem positiv geladenen Pol eines elektrischen Feldes durch ein Gel. Das Gel wirkt als Bremse; längere Fragmente sind langsamer. Die DNA wird radioaktiv oder mit fluoreszierenden Farbstoffen sichtbar gemacht. Die Position der sichtbaren Banden im Gel erlaubt eine relativ genaue Längenermittlung der DNA-Fragmente

Test auf Präsenz eines DNA-Abschnitts: mittels DNA-Sonden und Hybridisierung (z.B. Microarrays)

ANHANG C

Genomprojekte und Sequenziertechnologien

Ziel eines Genomprojekts: Bestimmung der DNA-Sequenz des gesamten Genoms (Nukleotidsequenzkarte); das Genom wird *sequenziert*.

Vorgehen bei Genomprojekten bis ca. 2004

1. Isolieren der DNA aus Zellkern
2. Zerschneiden der DNA mit Restriktionsenzymen (Enzym oder Enzymkombination, so dass viele Fragmente der Längen 300–1000 entstehen)
3. Einbringen der DNA-Fragmente in Bakterien oder Hefe durch Rekombination
4. Vermehrung dieser Organismen = Klonierung der DNA-Fragmente
5. Erstellung einer Klonbibliothek
6. evtl. Auswahl geeigneter Klone (physikalische Karte)
7. Sequenzierung der Klone mittels Sanger Sequencing
8. Zusammensetzen der überlappenden Fragmente per Computer

Man unterscheidet dabei zwei wesentliche Sequenzierstragetien:

- Ende 80er / Anfang 90er Jahre: Sequenzieren ist teuer: Kartierung der Klone, sorgfältige Auswahl ist billiger als mehr zu sequenzieren (klonbasierte Sequenzierung) –
- Ende der 90er Jahre: Sequenzierung ist relativ billig(er), alles wird sequenziert (whole genome shotgun). Problem, das Genom aus den Stücken der Länge 100-1000 zusammenzusetzen, ist schwieriger.

Bis ca. 2004 wurde nahezu ausschließlich *Sanger Sequencing* verwendet.

- TODO

Neue Sequenziertechnologien ab 2005 erlauben höheren Durchsatz und deutlich geringere Kosten.

- 454
- Illumina/Solexa
- SOLiD
- ...

Literaturverzeichnis

M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, Mar. 2004. ISSN 15708667. doi: 10.1016/S1570-8667(03)00065-0. URL `http://portal.acm.org/citation.cfm?id=985384.985389`.

A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi: 10.1145/360825.360855. URL `http://portal.acm.org/citation.cfm?id=360855`.

B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell*. Garland Science, 2007. ISBN 0815341059.

A. N. Arslan, O. Eğecioğlu, and P. A. Pevzner. A new approach to sequence comparison: normalized sequence alignment. *Bioinformatics*, 17(4):327–337, Apr 2001.

R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977. URL `http://portal.acm.org/citation.cfm?id=359842.359859`.

N. Christianini and M. W. Hahn. *Introduction to Computational Genomics – A Case Studies Approach*. Cambridge University Press, 2006.

R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1999. ISBN 0521629713.

R. Franklin and R. Gosling. Molecular configuration in sodium thymonucleate. *Nature*, 171: 740–741, 1953.

M. B. Gerstein, C. Bruce, J. S. Rozowsky, D. Zheng, J. Du, J. O. Korbel, O. Emanuelsson, Z. D. Zhang, S. Weissman, and M. Snyder. What is a gene, post-ENCODE? History and updated definition. *Genome Research*, 17(6):669–681, 2007. ISSN 1088-9051. doi: 10.1101/gr.6339607.

D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. ISBN 0521585198.

D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975. ISSN 0001-0782. doi: 10.1145/360825.360861. URL `http://doi.acm.org/10.1145/360825.360861`.

R. N. Horspool. Practical fast searching in strings. *Software-Practice and Experience*, 10: 501–506, 1980.

A. Klug. Rosalind Franklin and the discovery of the structure of DNA. *Nature*, 219:808–844, 1968.

D. E. Knuth, J. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977. doi: 10.1137/0206024. URL `http://link.aip.org/link/?SMJ/6/323/1`.

B. Maddox. The double helix and the 'wronged heroine'. *Nature*, 421:407–408, 2003.

G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002. ISBN 0521813077.

S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, Mar. 1970. ISSN 0022-2836. doi: 10.1016/0022-2836(70)90057-4. URL `http://www.sciencedirect.com/science/article/B6WK7-4DN8W3K-7X/2/0d99b8007b44cca2d08a031a445276e1`.

S. Prohaska and P. Stadler. "Genes". *Theory in Biosciences*, 127(3):215–221, 2008.

D. Sankoff and J. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. CSLI - Center for the Study of Language and Information, Stanford, CA, 1999. ISBN 1575862174. Reissue Edition.

T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, Mar. 1981. ISSN 0022-2836. doi: 10.1016/0022-2836(81)90087-5. URL `http://www.sciencedirect.com/science/article/B6WK7-4DN3Y5S-24/2/b00036bf942b543981e4b5b7943b3f9a`.

D. M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33 (8):132–142, 1990. doi: 10.1145/79173.79184. URL `http://portal.acm.org/citation.cfm?id=79184`.

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

E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi: 10.1007/BF01206331. URL `http://dx.doi.org/10.1007/BF01206331`.

J. Watson and F. Crick. A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

M. Wilkins, A. Stokes, and H. Wilson. Molecular structure of deoxypentose nucleic acids. *Nature*, 171:738–740, 1953.