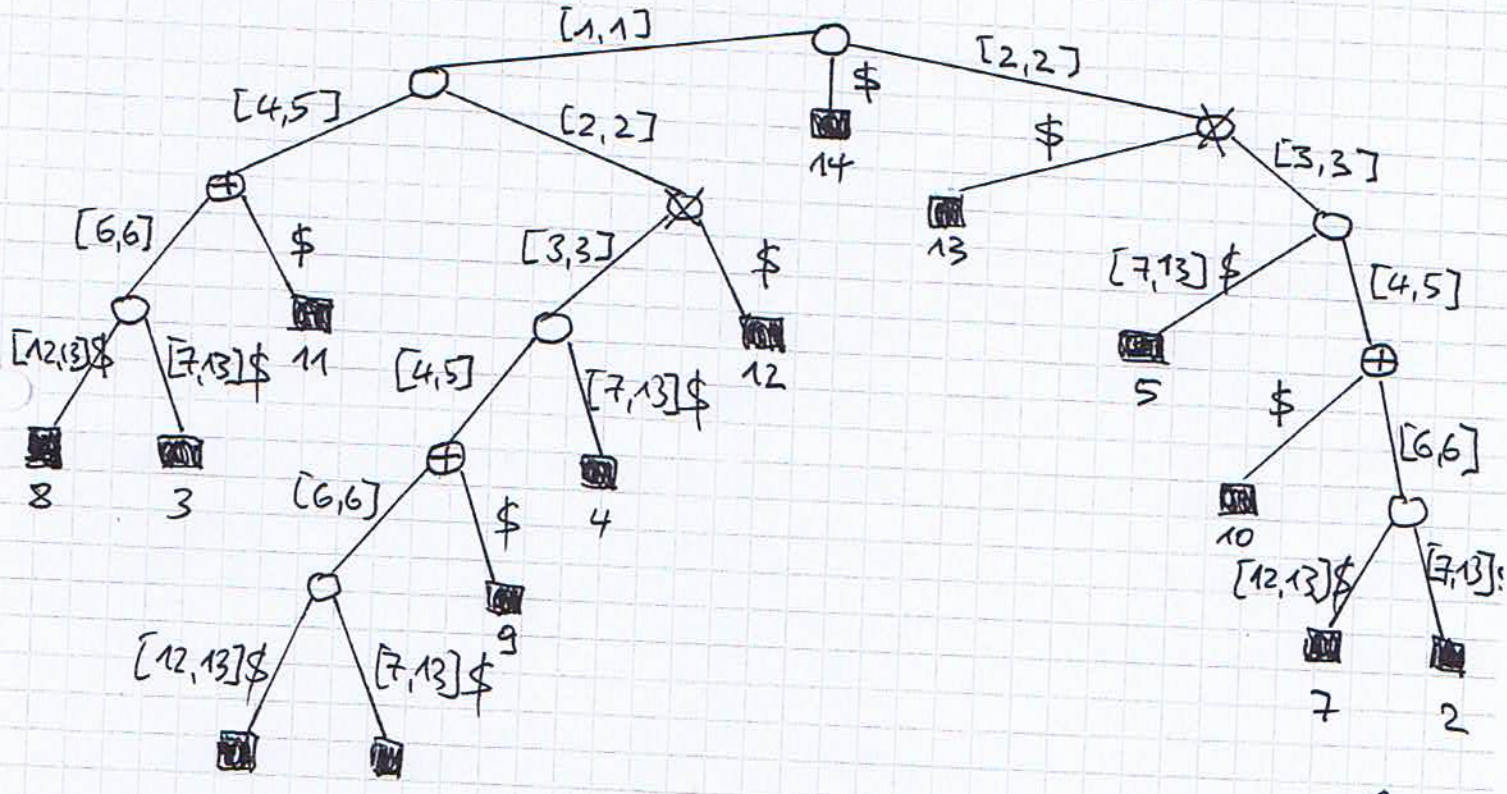
2. Kompakte auf Suffixe basierende Strukturen

In vielen praktischen Anwendungen ist der benötigte Speicherplatz eine kritische Randbedingung. Suffixbäume haben die Tendenz, isomorphe Teilbäume, die sich nur durch die Nummerierung ihrer Blätter unterscheiden, zu enthalten.

Beispiel:

Suffixbaum für x = abaababaabaabab$

1 2 3 4 5 6 7 8 9 10 11 12 13 14



Teilbaum, dessen Wurzel mit x markiert ist, kommt zweimal vor. Teilbaum, dessen Wurzel mit + markiert ist, kommt dreimal vor.

Ziel:

Entwicklung von Datenstrukturen, die aufgrund derartiger Übereinstimmungen Platz einsparen.

2.1 Gerichtete, azyklische Wortgraphen

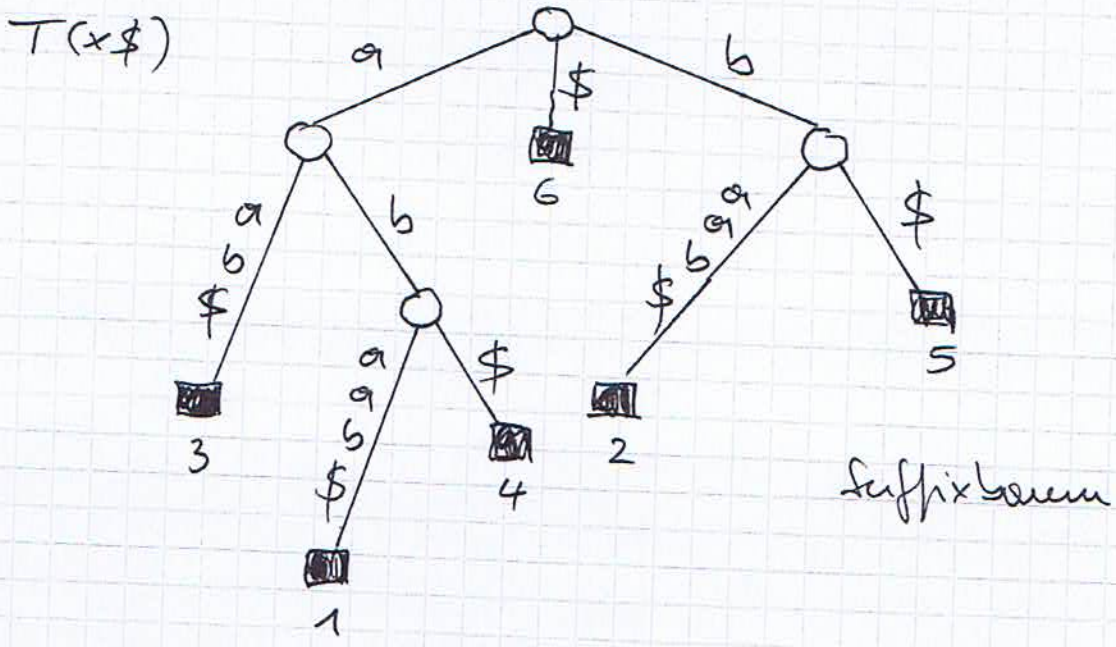
A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, J. Seiferas, The smallest automaton recognizing the subwords of a text, TCS 40 (1985), 31-55.

Der gerichtete azyklische Wortgraph eines Textes ist das Resultat einer Kompressionstechnik, die die oben beschriebene Redundanzen eliminiert. Gegeben einen Suffixbaum für einen Text x kann dieser in Linearzeit konstruiert werden.

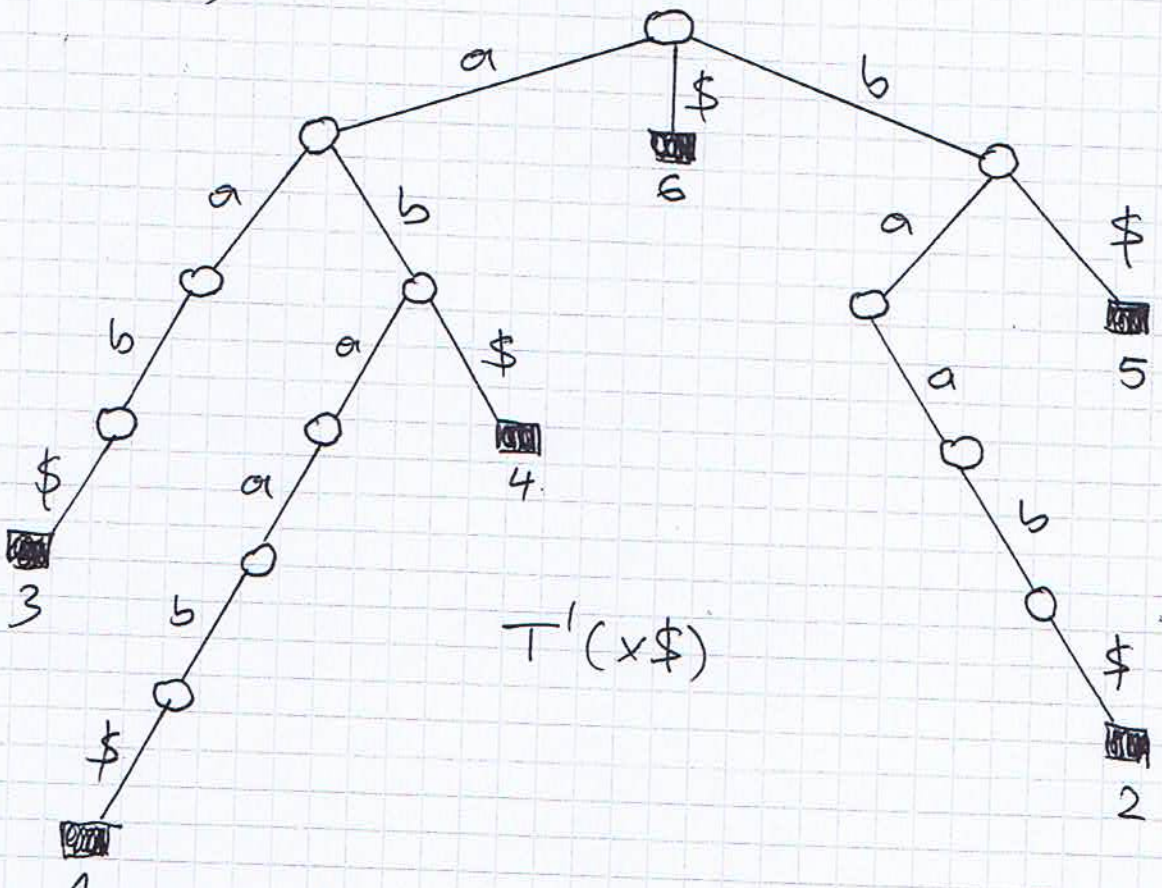
Die Entwicklung einer Methode zur Elimination von isomorphen Teilbäumen in einem Suffixbaum setzt die Analyse der Umstände, in denen derartige isomorphe Teilbäume in Suffixbäume vorkommen, voraus. Hierzu sorgen wir durch anhängen des Sonderzeichens $, dass der resultierende Textstring x$ präfixfrei ist. Die Analyse ist einfacher, wenn wir diese nicht direkt am Suffixbaum, sondern an dem korrespondierenden Trie, der für jeden Teilstring von x$ einen Knoten enthält, vornehmen.

Beispiel(*):

$x\$ = abaab\$$



Suffixbaum



Zu $T(x\$)$ korrespondierendes Trie.

◇

Sei u ein nichtleerer Teilstring von $x\$$. Dann bezeichnet Π_u die Folge von Positionen in $x\$$, in denen ein Vorkommen von u endet.

Beispiel (*):

$$\Pi_{ab} = 2, 5$$

◇

Wir vereinbaren

$$\Pi_\varepsilon := 1, 2, \ldots, n+1,$$

wobei $n := |x|$.

Bemerkung:

Die Terminatorenfolgen spiegeln die Struktur des Suffixbaumes $T(x\$)$ wider. Sei u ein nichtleerer Teilstring von $x\$$, der in exakt k Positionen $j_1, j_2, \ldots, j_k$ in x vorkommt. Dann gilt:

- $\Pi_u = j_1 + |u| - 1, j_2 + |u| - 1, \ldots, j_k + |u| - 1.$
- Die Positionen $j_1, j_2, \ldots, j_k$ sind die Nummern derjenigen Blätter in zum Teilstring u korrespondierenden Teilbaum von $T(x\$)$.

Π_ε enthält die Nummern aller Blätter, die zum Teilstring ε korrespondieren; d.h., die Nummern aller Blätter in $T(x\$)$.

Für die Konstruktion des gerichteten, azyklischen Wortgraphen interessieren wir uns für Teilstrings, die identische Terminatorenfolgen haben.

Seien u_1 und u_2 zwei Teilstrings von $x\$$. Falls $\Pi_{u_1} = \Pi_{u_2}$, dann heißen u_1 und u_2 <u>Π-äquivalent</u>.

<u>Übung</u>
Zeigen Sie, dass die Π-Äquivalenz eine Äquivalenzrelation ist.

<u>Lemma 2.1</u>
Seien u_1 und u_2 zwei nichtleere Teilstrings von $x\$$ mit $|u_1| \leq |u_2|$. Die Teilstrings u_1 und u_2 sind genau dann Π-äquivalent, wenn u_1 nur als Suffix von u_2 in $x\$$ vorkommt.

<u>Beweis:</u>

„$\Rightarrow$“

<u>Annahme:</u> $\Pi_{u_1} = \Pi_{u_2}$

$\Rightarrow$

u_1 und u_2 terminieren in exakt denselben Positionen von $x\$$.

Also impliziert $|u_1| \leq |u_2|$, dass u_1 stets ein Suffix von u_2 ist.

„⇐“

Wenn u_1 stets ein Suffix von u_2 ist, dann müssen die Terminatorenfolgen von u_1 und u_2 gleich sein.

∎

Lemma 2.2

Seien u_1 und u_2 zwei Teilstrings von $x\$$ mit $|u_1| \leq |u_2|$. Dann gilt

$$\Pi_{u_1} \cap \Pi_{u_2} = \emptyset \quad \text{oder} \quad \Pi_{u_2} \subseteq \Pi_{u_1}.$$

Beweis:

Falls $u_1 = \varepsilon$, dann gilt offensichtlich $\Pi_{u_2} \subseteq \Pi_{u_1}$.

Annahme: $u_1 \neq \varepsilon$.

$\Pi_{u_1} \cap \Pi_{u_2} \neq \emptyset \Rightarrow u_1$ ist ein Suffix von u_2.

Dann ist aber auch jedes Element von Π_{u_2} in Π_{u_1} enthalten. Also gilt $\Pi_{u_2} \subseteq \Pi_{u_1}$.

∎

Ein Knoten u in $T'(x\$)$ repräsentiert den zur Kantenmarkierung des Pfades von der Wurzel zum Knoten u korrespondierenden String $m(u)$.
T'_u bezeichnet den Teilbaum mit Wurzel u von $T'(x\$)$.

Lemma 2.3

Seien u_1 und u_2 zwei beliebige Knoten in $T'(x\$)$ T'_{u_1} und T'_{u_2} sind genau dann isomorph, wenn

u_1 und u_2 Π-äquivalent sind.

Beweis:

Falls $u_1 = u_2$, dann sind $T'_{u_1} = T'_{u_2}$ und $m(u_1) = m(u_2)$, so dass die Behauptung trivialerweise erfüllt ist.

Annahme: $u_1 \neq u_2$

Da nur für die Wurzel w $m(w) = \varepsilon$ und T'_w zu keinem echten Teilbaum isomorph sein kann, können wir $m(u_1) \neq \varepsilon$ und $m(u_2) \neq \varepsilon$ annehmen.

„$\Leftarrow$“

Annahme: $\Pi_{m(u_1)} = \Pi_{m(u_2)}$.

O.B.d.A. können wir $|m(u_1)| \leq |m(u_2)|$ annehmen.

Lemma 2.1 $\Rightarrow$ $m(u_1)$ kommt nur als Suffix von $m(u_2)$ in x vor.

$\Rightarrow$ T'_{u_2} und T'_{u_1} sind isomorph.

„$\Rightarrow$“

Annahme: T'_{u_1} und T'_{u_2} sind isomorph.

$\Rightarrow$

$\exists$ Suffix $m(u_1)x'\$$ von $x\$$

$\Leftrightarrow$

$\exists$ Suffix $m(u_2)x'\$$ von $x\$$.

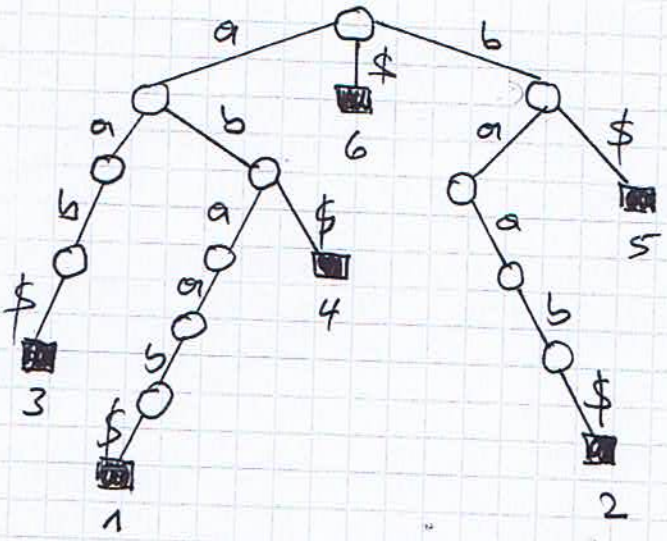
$m(u_1)$ und $m(u_2)$ müssen in x stets in derselben Position enden.

$\Rightarrow$

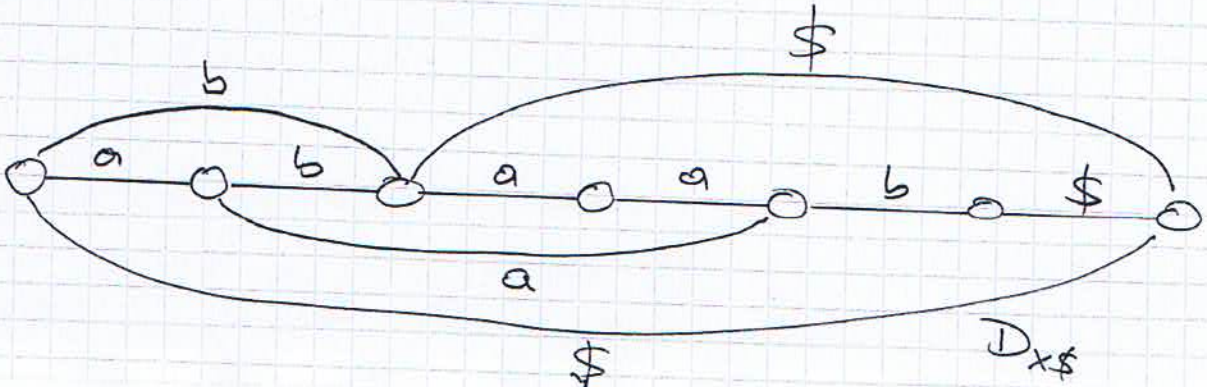
$$\Pi_{m(u_1)} = \Pi_{m(u_2)}$$

Der gerichtete, azyklische Wortgraph $D_{x\$}$ repräsentiert den minimalen endlichen Automaten, der exakt die Suffixe des zugrundeliegenden Textstrings akzeptiert. Dieser kann aus dem zum Suffixbaum $T(x\$)$ korrespondierenden Trie $T'(x\$)$ konstruiert werden. Wir erhalten diesen Graphen, indem wir in $T'(x\$)$ isomorphe Teilbäume nur einmal realisieren.

Beispiel (*): $x\$ = abaab\$$



$T'(x\$)$



Frage: Wie groß ist $D_{x\$}$ im worst case?

Der nachfolgende Satz beantwortet diese Frage.

Satz 2.1

Sei $x\$$ ein String der Länge $n+1$. Dann enthält der gerichtete azyklische Wortgraph $D_{x\$}$ höchstens $N \leq 2n+1$ Knoten und höchstens $N+n-1 \leq 3n$ Kanten.

Beweis:

Die Knoten in $D_{x\$}$ korrespondieren zu einem oder mehreren Knoten in $T'(x\$)$. Falls ein Knoten in $D_{x\$}$ zu mehreren Knoten in $T'(x\$)$ korrespondiert, dann sind die dazugehörigen Teilstrings in x π-äquivalent. Für Knoten u in $D_{x\$}$ bezeichne $\ell(u)$ stets den kürzesten solchen Teilstring.

Der Startknoten s von $D_{x\$}$ korrespondiert zur Wurzel des Tries $T'(x\$)$.

$\Rightarrow$

$\ell(s) = \varepsilon$.

Konstruktion und Lemma 2.3 $\Rightarrow$

Für unterschiedliche Knoten u_1 und u_2 von $D_{x\$}$ gilt:

$\ell(u_1)$ und $\ell(u_2)$ sind nicht π-äquivalent

Somit impliziert Lemma 2.2

$\Pi_{k(u_1)} \cap \Pi_{k(u_2)} = \emptyset$ oder

$\left(\Pi_{k(u_1)} \not\subseteq \Pi_{k(u_2)} \text{ oder } \Pi_{k(u_2)} \not\subseteq \Pi_{k(u_1)} \right)$.

Idee:

Abarbeitung von $D_{x\$}$ in topologischer Reihenfolge mehrt gleichzeitiger Konstruktion eines Baumes $B_{x\$}$, der exakt die Knoten aus $D_{x\$}$ enthält. Eine obere Schranke für die Anzahl der Knoten in $B_{x\$}$ ergibt dann eine obere Schranke für die Anzahl der Knoten in $D_{x\$}$.

Durchführung:

- Der Startknoten s von $D_{x\$}$ wird die Wurzel von $B_{x\$}$.
- Sei u derjenige Knoten in $D_{x\$}$, der gerade betrachtet wird.

  ⇝

  u wird zum Sohn desjenigen Knotens v im aktuellen Baum, für den gilt:

  i) $k(v)$ ist Suffix von $k(u)$ und

  ii) $|k(v)|$ ist maximal unter allen Knoten im aktuellen Baum, die i) erfüllen.

Da $\varepsilon = k(s)$ Suffix von $k(u)$ ist, existiert der Knoten v.

Eigenschaften von $B_{x\$}$:

1) Jeder Knoten korrespondiert zu einer nichtleeren Teilmenge von $\{1, 2, \ldots, n+1\}$.

2) Die Vereinigung der korrespondierenden Teilmengen der Söhne eines Knotens v ist in der zu v korrespondierenden Teilmenge enthalten.

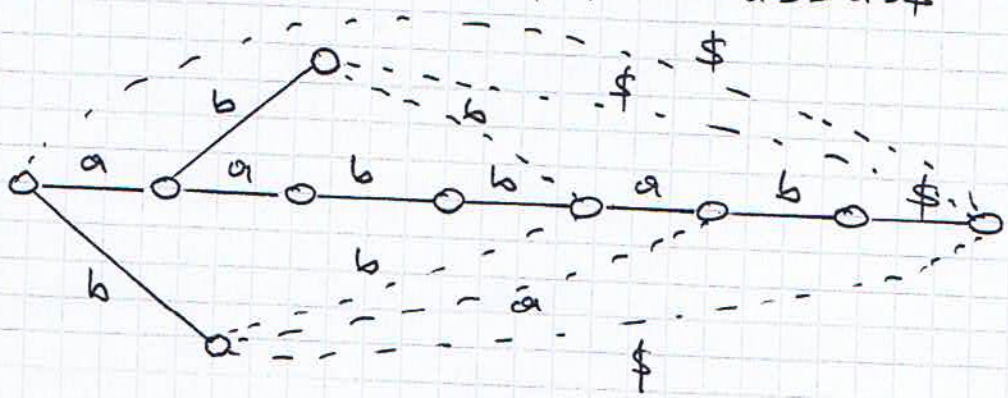
3) Befinden sich u und v auf zwei unterschiedlichen Pfaden von der Wurzel zu einem Blatt, dann sind die korrespondierenden Teilmengen disjunkt.

Grundstudium $\Rightarrow$

$B_{x\$}$ enthält $\leq 2n+1$ Knoten.

Zum Beweis der oberen Schranke für die Anzahl der Knoten in $D_{x\$}$ betrachten wir einen überspannenden Baum $ST_{x\$}$ von $D_{x\$}$, der den mit $x\$$ markierten längsten Pfad enthält.

Beispiel: $D_{aabbab\$}$, $ST_{aabbab\$}$



Nacheinander fügen wir die übrigen Kanten von $D_{x\$}$ hinzu. Da das Hinzufügen einer solchen Kante die Anzahl der Pfade von s zum Endknoten von $D_{x\$}$ mindestens um eins erhöht und die Anzahl solches Pfade n ist, ist die Anzahl der Kanten in $D_{x\$}$

$$\leq (N-1) + n.$$

■

Ziel:
Konstruktion von $D_{x\$}$ in $O(|x\$|)$ Zeit.

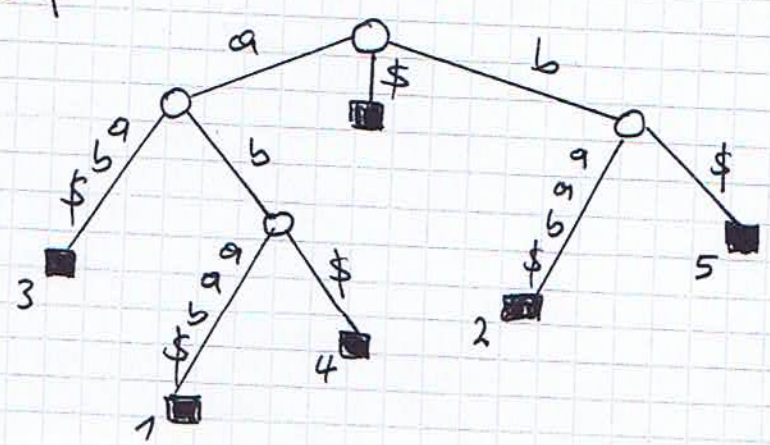
Bemerkung:
Da $T'(x\$)$ in $|x|$ quadratisch viele Knoten enthalten kann, führt der oben beschriebene Weg nicht notwendigerweise zum Ziel.

Idee:
Konstruktion von $D_{x\$}$ direkt aus dem Suffixbaum $T(x\$)$.

Betrachten wir hierzu nochmal das Beispiel (*).

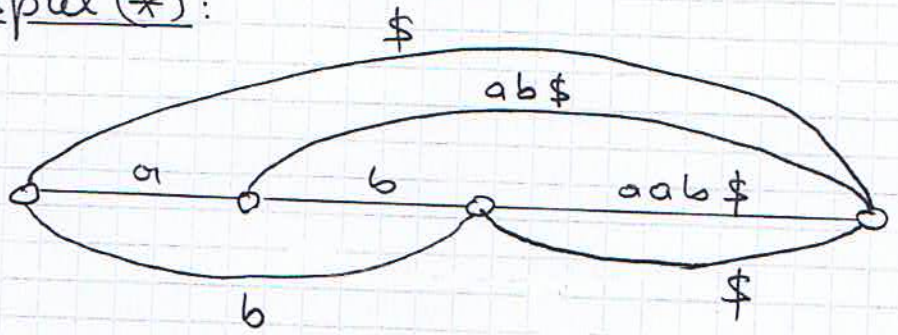
Beispiel (*):
$x\$ = abaab\$$

$T(x\$)$



Wenn wir die Methode von oben direkt auf $T(x\$)$ anwenden und dabei die Kantenmarkierungen als einzelnes Symbol interpretieren, dann erhalten wir einen gerichteten azyklischen Graphen. Dieser kann dann in den gerichteten azyklischen Wortgraphen transformiert werden.

Beispiel (*):



◇

Die Frage, die sich nun stellt ist die folgende:

Wie findet man in $T(x\$)$ effizient die isomorphe Teilbäume?

Folgendes Lemma gibt uns eine Antwort auf diese Frage:

Lemma 2.4

Seien u_1 und u_2 zwei Knoten in $T(x\$)$ mit $|m(u_1)| < |m(u_2)|$. Dann sind die Teilbäume mit Wurzel u_1 und Wurzel u_2 genau dann isomorph, wenn folgendes erfüllt ist:

i) Beide Teilbäume enthalten dieselbe Anzahl von Blättern.

ii) Es gibt von u_2 nach u_1 einen Pfad über Suffixzeiger.

Beweis:

Für $v \in T(x\$)$ bezeichne T_v den Teilbaum mit Wurzel v von $T(x\$)$.

„$\Rightarrow$"

Annahme: T_{u_1} und T_{u_2} sind isomorph.

$\Rightarrow$

T_{u_1} und T_{u_2} besitzen dieselbe Anzahl von Blättern.

Falls $m(u_1)$ ein Suffix von $m(u_2)$ ist, dann folgt aus unseren bisherigen Überlegungen bzgl. Suffixzeiger, dass es einen Pfad von u_2 nach u_1 über Suffixzeiger gibt.

z.z. $m(u_1)$ ist ein Suffix von $m(u_2)$.

Annahme: $m(u_1)$ ist kein Suffix von $m(u_2)$.

Wir werden nun beweisen, dass dann T_{u_1} und T_{u_2} nicht isomorph sind.

Betrachten wir irgendein Vorkommen von $m(u_2)$ in $x\$$, d.h.

$$x\$ = \alpha\, m(u_2)\, \beta\$.$$

$\Rightarrow$

in T_{u_2} gibt es einen Pfad von u_2 zu einem

Blatt mit Markierung $B\$$

Da $m(u_1)$ kein Suffix von $m(u_2)$ ist, kann es in T_{u_1} keinen Pfad von u_1 zu einem Blatt mit Markierung $B\$$ geben.

$\Rightarrow$

T_{u_1} und T_{u_2} sind nicht isomorph.

$\Rightarrow$

Obige Annahme ist falsch und somit $m(u_1)$ ein Suffix von $m(u_2)$.

"$\Leftarrow$"

Annahme:

T_{u_1} und T_{u_2} besitzen dieselbe Anzahl von Blättern und es gibt einen Pfad von u_2 nach u_1 über Suffixzeiger.

Wir beweisen, dass T_{u_1} und T_{u_2} isomorph sind, mittels Induktion über die Länge ℓ des Pfades von u_2 nach u_1 über Suffixzeiger.

$\ell = 1$: Es gibt einen Suffixzeiger von u_2 nach u_1

$\Rightarrow$

$m(u_2) = a\,m(u_1)$ für ein $a \in \Sigma$.

$\Rightarrow$

$\Pi(u_2) \subseteq \Pi(u_1)$

$\Rightarrow$

Für jeden Pfad P von u_2 zu einem

Blatt mit Markierung $m(P)$ in T_{u_2} gibt es in T_{u_1} den entsprechenden Pfad von u_1 zu einem Blatt mit derselben Struktur und derselben Markierung.

Da die Anzahl der Blätter in T_{u_1} und T_{u_2} gleich ist, gibt es in T_{u_1} keine weitere Pfade

$\Rightarrow$

T_{u_1} und T_{u_2} sind isomorph.

Annahme: Behauptung ist wahr für Pfade der Länge ℓ, $\ell \geq 1$.

$\ell \rightsquigarrow \ell+1$:

Betrachte Pfad $P = u_2 = v_1, v_2, \dots, v_{\ell+1}, v_{\ell+2} = u_1$ von u_1 nach u_1 über Suffixzeiger.

Beobachtung:
Die Anzahl der Blätter in T_{v_i}, $1 \leq i \leq \ell+2$ ist auf P monoton wachsend.

Da die Anzahl der Blätter in T_{v_1} und in $T_{v_{\ell+2}}$ gleich ist, haben alle Teilbäume T_{v_i}, $1 \leq i \leq \ell+2$, dieselbe Anzahl von Blätter.

Induktionsannahme $\Rightarrow$

T_{v_1} und $T_{v_{\ell+1}}$ sind isomorph.

Induktionsanfang $\Rightarrow$

$T_{v_{\ell+1}}$ und $T_{v_{\ell+2}}$ sind isomorph.

Da die Isomorphie von Bäumen eine Äquivalenz-relation ist, sind somit T_{u_2} und T_{u_1} isomorph ■

Somit haben wir die Voraussetzungen für die Entwicklung eines Algorithmus zur Berechnung des gerichteten azyklischen Wortgraphen aus dem Suffixbaum in Linearzeit geschaffen.

Übung:
Entwickeln Sie einen Algorithmus, der für einen Textstring x$ aus dem Suffixbaum T(x$) den gerichteten azyklischen Wortgraphen $D_{x\$}$ in Zeit $O(|x\$|)$ berechnet.

2.2 Suffixarrays

Sei $x = a_1 a_2 \dots a_n \in \Sigma^n$ ein String über dem Alphabet Σ. Sei auf Σ eine totale Ordnung definiert. Dann definieren wir die lexikographische Ordnung von Strings über Σ auf die übliche Art und Weise.

Beispiel:

x = Mississippi

Dann ist die lexikographische Ordnung der Suffixe von Mississippi die folgende:

11 : i
8 : ippi
5 : issippi
2 : ississippi
1 : Mississippi
10 : pi
9 : ppi
7 : sippi
4 : sissippi
6 : ssippi
3 : ssissippi

Die Zahl links vom Suffix gibt seine Anfangsposition im String an.

◊

Idee:

Entwickle eine Datenstruktur, die die binäre Suche unterstützt.

↝

Ein Suffixarray für den String x ist ein Feld Pos der Anfangspositionen der Suffixe von x in lexikographischer Ordnung.

Beispiel:

Suffixarray für Mississippi:

Pos = [11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]

◊

Eigenschaften: ($n := |x|$)

1) Speicherplatz:

- Hängt nicht von der Alphabetgröße ab.
- Wesentlich geringer als für den Suffixbaum

2) Konstruktionszeit:

- direkte Konstruktion: $O(n \cdot \log n)$
- unter Verwendung des Suffixbaumes: $O(n)$

3) Musterstringsuche: (alle Vorkommen)

- naiv: $O(m \cdot \log n)$, wobei $m := |y|$ ← Musterstring
- einfach beschleunigt:
  in Praxis $O(m + \log n)$ aber
  im worst case $O(m \cdot \log n)$
- klever beschleunigt:
  im worst case $O(m + \log n)$.

Wir werden uns zunächst überlegen, wie wir in $O(n)$ Zeit das Suffixarray mit Hilfe eines gegebenen Suffixbaumes konstruieren können. Die Idee hierbei ist, dass wir während der Vorbereitungsphase mehr Speicherplatz verwenden dürfen. Der für den Suffixbaum benötigte Speicherplatz wird nach der Konstruktion des Suffixarrays wieder freigegeben.

Annahme:
Suffixbaum $T(x)$ ist gegeben.

Idee:
Führe auf $T(x)$ eine lexikographische Tiefensuche durch; d.h., von innerem Knoten v aus betrete als nächste Kante diejenige, deren Markierung unter allen noch nicht betretenen ausgehenden Kanten von v mit dem lexikographisch kleinsten Buchstaben beginnt.

Übung
a) Arbeiten Sie die Konstruktion des Suffixarrays Pos eines Textes x mittels lexikographischer Tiefensuche auf einem Suffixbaum $T(x)$ aus.
b) Beweisen Sie die Korrektheit Ihres Verfahrens.

Als nächstes werden wir Verfahren zur Musterstringsuche in einem Text x unter Verwendung des Suffixarrays Pos entwickeln.

Annahme:
Musterstring y, $|y| = m$ soll in Textstring x, $|x| = n$ gesucht werden, wobei das Suffixarray Pos gegeben ist.

Beobachtung:
Die Anfangspositionen aller Vorkommen von y in x sind in Pos aufeinanderfolgend in einem Block gruppiert.

Ziel:

Ermittlung des kleinsten Index min und des größten Index max, so dass y Präfix des in Pos[min] sowie des in Pos[max] stehenden Suffixes ist.

Idee:

Suche zunächst min und dann max mittels binärer Suche auf dem Suffixarray Pos.

Durchführung:

- Bezeichne stets L und R die linke und die rechte Grenze des aktuellen Suchintervalls. D.h., zu Beginn gilt

$$L = 1 \quad \text{und} \quad R = n.$$

- Bezeichne $\leq_{lex}$ die lexikographische Ordnungsrelation.
- Vergleiche in jedem Schritt y und Pos[M], wobei $M := \lceil \frac{R+L}{2} \rceil$ bezüglich ihrer lexikographischen Ordnung.

if $y <_{lex} Pos[M]$

then $R := \lceil \frac{R+L}{2} \rceil$

else

if $Pos[M] <_{lex} y$

then $L := \lceil \frac{R+L}{2} \rceil + 1$

else $max := \lceil \frac{l+L}{2} \rceil$

fi

fi.

Übung:
Arbeiten Sie die Berechnung von min und max mittels Binärsuche auf Pos aus.

benötigte Zeit:

- worst case: $O(m \cdot \log n)$
- Diese Zeit wird benötigt, wenn die mit y verglichenen Suffixe meistens lange Präfixe mit y gemeinsam haben.

Frage: Können unnötige Vergleiche vermieden werden?

Bezeichne l bzw. r stets die Länge des gemeinsamen Präfixes von Pos[L] und y bzw von Pos[R] und y.

$\Rightarrow$

Pos[M] und y haben einen gemeinsamen Präfix der Länge $\geq \min\{l, r\}$.

$\Rightarrow$

Der Vergleich von Pos[M] mit y könnte in der Position $\min\{l, r\} + 1$ begonnen werden.

benötigte Zeit:

In Praxis: $O(m + \log n)$

worst case: $\Omega(m \log n)$.

Ziel:

Modifikation der obigen Idee, dass auch im worst case die Zeit $O(m + \log n)$ benötigt wird.

Die Überprüfung eines Symbols in y heißt redundant, falls dieses Symbol bereits vorher überprüft worden ist. Falls es gelingt die Anzahl der redundanten Überprüfungen pro Iteration des binären Suche auf eins zu begrenzen, dann erreichen wir obiges Ziel.

Für i und j bezeichne $Lcp(i,j)$ die Länge des längsten gemeinsamen Präfixes von $Pos[i]$ und $Pos[j]$.

Beispiel:

x = Mississippi

$lcp(3,4) = 4$, da

$Pos[3]$ = issippi

$Pos[4]$ = ississippi

◊

Frage:
Welche Überprüfungen sind bei einfacher Beschleunigung redundant?

Antwort:
Die Überprüfung von Symbolen in den Positionen von $\min\{\ell, r\} + 1$ bis $\max\{\ell, r\}$ sind alle redundant.

Idee:
Verwende $Lcp(L, M)$ und $Lcp(M, R)$ für jedes auftretenden Tripel (L, M, R) um diese Überprüfungen zu vermeiden.

Wir nehmen an, dass jeder der benötigten Werte in konstanter Zeit verfügbar ist. Später werden wir uns überlegen, wie wir das erreichen.

Durchführung:

- Falls $\ell = r$, dann können derartige redundante Überprüfungen nicht stattfinden, so dass der Vergleich zwischen y und $Pos[M]$ in Position

$$\min\{\ell, r\} + 1 = \ell + 1 = r + 1$$

gestartet werden kann.

- Annahme: $\ell \neq r$.

Zunächst betrachten wir den Fall $\ell > r$.

Zur Ermittlung der neuen unteren (und oberen) Grenze des Such-intervals vergleichen wir $Lcp(L, M)$ mit ℓ. Drei Fälle können eintreten.

$\underline{Lcp(L,M) > \ell}$:

D.h., dass das gemeinsame Präfix von $Pos[L]$ und $Pos[M]$ länger ist als das gemeinsame Präfix von y und $Pos[L]$.

$\Rightarrow$

Gemeinsame Präfix von y und $Pos[M]$ hat Länge ℓ.

Da $y >_{ex} Pos[L]$ und $Pos[L]$ und $Pos[M]$ auch in der $\ell+1$-ten Position gleich sind, folgt

$Pos[M] <_{ex} y$

$\Rightarrow$

M ist die neue untere Grenze des Such-intervalls. ℓ und r bleiben unverändert.

$\underline{Lcp(L,M) < \ell}$:

D.h., das gemeinsame Präfix von $Pos[L]$ und $Pos[M]$ ist kürzer als das gemeinsame Präfix von y und $Pos[L]$.

$\Rightarrow$

$y <_{ex} Pos[M]$

$\Rightarrow$

M ist die neue obere Grenze des Suchintervalls.
l bleibt unverändert. r wird zu Lcp(L, M).

Lcp(L, M) = l:

D.h., das gemeinsame Präfix von y und Pos[M] hat Länge ≥ l.

⇝

Beginnend in Position l+1 vergleicht der Algorithmus lexikographisch y und Pos[M].

In Abhängigkeit des Resultats ändert sich die untere oder obere Grenze des Suchintervalls und dementsprechend der korrespondierende Wert l oder r.

Der Fall $l < r$ geht analog.

Übung:
Arbeiten Sie obigen Algorithmus aus.

Satz 2.2
Mustershingsuche mit "Klevere" Beschleunigung hat Laufzeit $O(m + \log n)$.

Beweis:

- Konstruktion ⇒ l und r wachsen monoton.
- Jede Iteration der binären Suche
  - überprüft kein Symbol von y oder
  - überprüft Symbole von y.

Nur in den Fällen $\ell = r$ und $Lcp(L, M) = \ell$ werden Symbole in y und Pos[M] lexikographisch miteinander verglichen.

Gestartet wird in Position $\max\{\ell, r\} + 1$.

Annahme:
In der aktuellen Iteration werden k Symbole überprüft.

$\Rightarrow$

$k-1$ der Überprüfungen waren Matches

$\Rightarrow$

$\max\{\ell, r\}$ erhöht sich um $k-1$, da entweder ℓ oder r zu diesem Wert übergeht.

$\Rightarrow$

Zu Beginn jeder Iteration gilt:

Das Symbol in Position $\max\{\ell, r\} + 1$ in y wurde bereits überprüft (mit Ausgang Mismatch). Jedes in y nachfolgende Symbol wurde bisher nicht überprüft.

$\Rightarrow$

Pro Iteration wird höchstens eine redundante Überprüfung durchgeführt.

$\Rightarrow$

Anzahl der redundante Überprüfungen insgesamt:

$\leq \log n$.

Weiterhin gilt:

i) Gesamtaufwand pro Iteration in den anderen Fällen: $O(1)$

ii) Anzahl der nicht redundanten Überprüfungen insgesamt: $\leq m$

$\Rightarrow$

Gesamtaufwand: $O(m + \log n)$

Als nächstes werden wir uns eine effiziente Berechnung der benötigten Lcp-Werte überlegen.

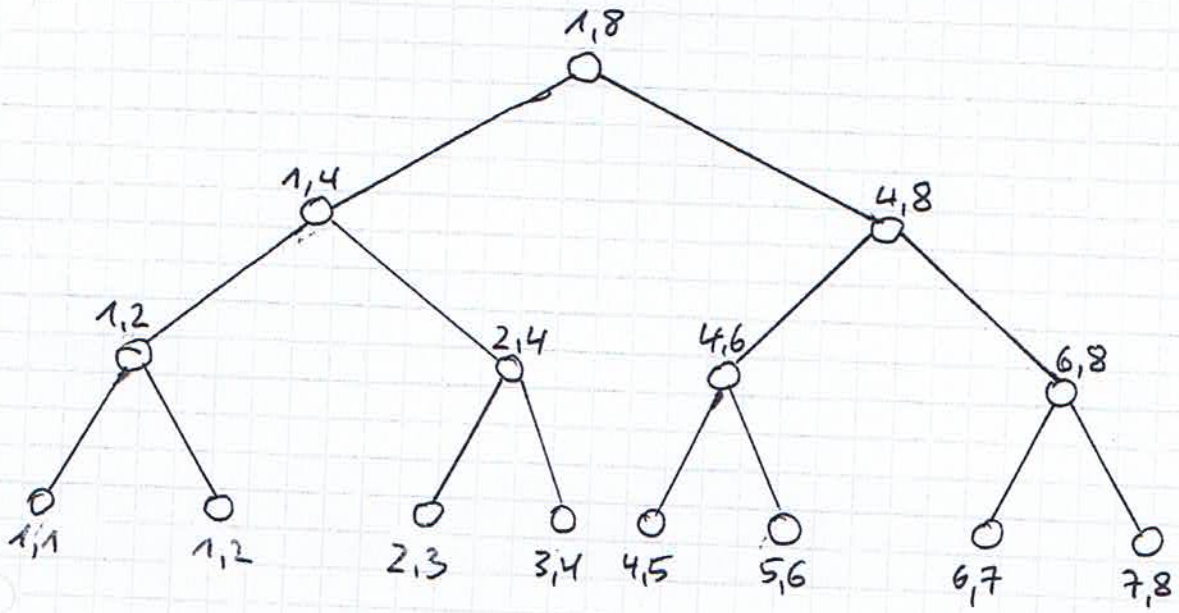
Frage:
Welche Lcp-Werte werden eventuell benötigt?

Für die Beantwortung dieser Frage nehmen wir der Einfachheit halber an, dass n eine Potenz von zwei ist.

Sei B ein vollständig binärer Baum mit n Blättern. Jeder Knoten in B ist mit einem Paar (i,j), $1 \leq i \leq j \leq n$ wie folgt markiert:

- Markierung der Wurzel: $(1,n)$
- Ein innerer Knoten mit Markierung (i,j) hat zwei Söhne mit Markierungen $(i, \lfloor \frac{i+j}{2} \rfloor)$ und $(\lfloor \frac{i+j}{2} \rfloor, j)$.

Beispiel: $n = 8$



Die Markierungen der Knoten geben jeweils die untere und die obere Grenze des korrespondierenden Such-Intervalls $[L, R]$ an.

Da die Anzahl der Knoten $2n-1$ ist, werden höchstens die zu den Markierungen dieser Knoten korrespondierenden Lcp-Werte benötigt.

Annahme:
An jeden inneren Knoten v in $T(x\$)$ ist $|m(v)|$ gespeichert.

Beobachtung:

- Bezeichne $v(i,j)$ denjenigen inneren Knoten in $T(x\$)$, in dem sich die Pfade von dem zu $Pos[i]$ bzw. $Pos[j]$ korrespondierenden Blatt zur Wurzel treffen. Dann gilt:

$$Lcp(i,j) = |m(v(i,j))|.$$

$\Rightarrow$

Die benötigten Lcp-Werte können während der lexikographischen Tiefensuche auf $T(x\$)$ mit berechnet werden.

Übung:
Erweitern Sie die lexikographische Tiefensuche, so dass auch die benötigten Lcp-Werte mit berechnet werden.

3. Approximatives Stringmatching

Ziel:

Durchführung von approximatives String-matching; d.h., finden von zu einem Musterstring "ähnliche" Teilstrings in einem Textstring.

weitere Literatur:

N. Blum, Speeding up dynamic programming without omitting any optimal solution and some applications in molecular biology, J. of Algorithms 35 (2000), 129-168.

Approximativer Stringmatching sucht nicht notwendigerweise exakte Vorkommen eines Musterstrings im Textstring, sondern nahezu exakte Vorkommen.

Es stellt sich nun folgende Frage:

Was bedeutet "nahezu exakt"?

Ziel:
Präzisierung von "nahezu exakt" nebst exakte Formulierung der Aufgabenstellung.

3.1 Die Editierdistanz zweier Sequenzen

Seien $x = x_1 x_2 \dots x_n$ und $y = y_1 y_2 \dots y_m$ zwei Strings über einem endlichen Alphabet Σ.

Beispiel:

Alphabete in genetischen Sequenzen

- $\Sigma = \{A, G, C, T\}$,
  falls x eine DNA-Sequenz ist.
- $\Sigma = \{$ alanine, arginine, ..., valine $\}$,
  falls x eine Proteinsequenz ist.

◇

Genetische Sequenzen können in der evolutionären Geschichte durch Mutationen geändert werden. Dies möchten wir nun modellieren.

Modell:

Mutation $\hat{=}$ $(x,y) \in \Sigma^+ \times \Sigma^+$

Interpretation: Sequenz x wird durch Sequenz y ersetzt.

$M \subseteq \Sigma^+ \times \Sigma^+$ Menge der betrachteten Mutationen

$c: M \to \mathbb{R}$ Kostenfunktion.

Eine Folge $S := s_1, s_2, \ldots, s_t$ heißt genau dann Mutationenfolge über M, wenn es Strings $z_0, z_1, \ldots, z_t \in \Sigma^+$ gibt, so dass $(z_{i-1}, z_i) = s_i \in M$ für alle $1 \leq i \leq t$. Wir sagen, S transformiert z_0 nach z_t. $z_0, z_1, \ldots, z_t$ heißt M-Ableitung von z_t aus z_0

$$c(S) := \sum_{i=1}^{t} c(s_i)$$

sind die Kosten der Mutationenfolge S.

M heißt genau dann vollständig, wenn für alle $(x,y) \in \Sigma^+ \times \Sigma^+$ eine M-Ableitung von y aus x existiert.

Der evolutionäre Abstand oder auch Editier-abstand $d_M(x,y)$ der Sequenzen x und y ist definiert durch

$$d_M(x,y) := \min\{ c(S) \mid S \text{ transformiert } x \text{ nach } y \}.$$

Frage: Was ist eine vernünftige Wahl für M und c?

Anforderungen an M:

1) M muss Anforderungen aus der Molekularbiologie genügen, d.h., die Wirklichkeit möglichst exakt widerspiegeln.
2) M muss Anforderungen aus der Informatik genügen, d.h., die resultierenden Probleme müssen praktisch mit dem Computer lösbar sein.

Üblicherweise werden nur Mutationen betrachtet, die als lokale Operationen beschreibbar sind. Dann ist es möglich, die Menge aller betrachteten Mutationen durch eine endliche Menge von Operationen zu beschreiben.

M_1, eine erste Wahl für M:

M_1 enthält folgende vier Typen von Operationen:

a) Streichen eines Symbols aus x.
b) Einfügen eines Symbols in x.
c) Ersetzen eines Symbols durch ein anderes in x.
d) Substitution eines Symbols durch sich selbst in x.

Obwohl eine Substitution keine Mutation modelliert, ist es aus Gründen der Darstellung sinnvoll, sie auch in die Operationenmenge aufzunehmen. Da wir Mutationen positive Kosten zuordnen, haben Substitutionen Kosten ≤ 0.

Formal ist eine Operation in M_1 ein Paar $(a,b) \in (\Sigma \cup \{-\})^2 \setminus \{(-,-)\}$, $- \notin \Sigma$, wobei

$$(a,b) = \begin{cases} \text{Streichen von } a & \text{falls } a \in \Sigma,\ b = - \\ \text{Einfügen von } b & \text{falls } a = -,\ b \in \Sigma \\ \text{Ersetzen von } a \text{ durch } b & \text{falls } a \in \Sigma,\ b \in \Sigma \setminus \{a\} \\ \text{Substitution von } a \text{ für } a & \text{falls } a = b \in \Sigma \end{cases}$$

Bemerkung:

- M_1 ist vollständig
- Die Kostenfunktion c ist nur sinnvoll, falls

$$c(a,b) < c(a,-) + c(-,b) \quad \forall\, a \in \Sigma,\ b \in \Sigma \setminus \{$$

Eine in der Molekularbiologie übliche Darstellungsweise einer Ableitung einer Sequenz y aus einer Sequenz x ist das korrespondierende, nachfolgend definierte Alignment $A(x,y)$.

Ein Alignment $A(x,y)$ zwischen x und y besteht aus einer zweizeiligen Matrix, so dass gilt:

a) Die erste Zeile besteht aus x, möglicherweise mit Nullsymbolen - durchsetzt.

b) Die zweite Zeile besteht aus y möglicherweise mit Nullsymbolen - durchsetzt.

c) Es gibt keine Spalte, die nur aus Nullsymbolen - besteht.

Eine Spalte $\begin{bmatrix} a \\ b \end{bmatrix}$ beschreibt die Operation (a,b).
Die Kosten $c_{M_1}(A(x,y))$ eines Alignments $A(x,y)$ sind definiert durch

$$c_{M_1}(A(x,y)) := \sum_{\begin{bmatrix} a \\ b \end{bmatrix} \in A(x,y)} c_{M_1}(a,b),$$

wobei $c_{M_1}: M_1 \to \mathbb{R}$ die Kostenfunktion bezüglich der Operationenmenge M_1 ist. Dabei werden gleiche Spalten in unterschiedlichen Positionen des Alignments in obiger Summe als verschiedene Spalten behandelt.

Beobachtung:
Ein Alignment $A(x,y)$ beschreibt eine Menge von Ableitungen von y aus x. Zwei dieser Ableitungen unterscheiden sich lediglich in der Anordnung der Operationen. Demzufolge haben alle diese Ableitungen dieselben Kosten $c_{M_1}(A(x,y))$.

Ziel:
Entwicklung eines Verfahrens zur Berechnung der Editierdistanz zweier gegebenen Strings.

Sei $x = x_1 x_2 \dots x_n$ ein String über ein Alphabet Σ. Dann bezeichnet

$$x^j := \begin{cases} \varepsilon & \text{falls } j = 0 \\ x_1 x_2 \dots x_j & \text{falls } j > 0. \end{cases}$$

Folgender Satz liefert uns eine Rechenvorschrift zur Berechnung der Editierdistanz zweier Strings.

Satz 3.1

Seien $x = x_1 x_2 \dots x_n$ und $y = y_1 y_2 \dots y_m$ zwei Strings. Dann gilt für $0 \le j \le n$, $0 \le i \le m$:

$$d_{M_1}(x^j, y^i) = \min \begin{cases} d_{M_1}(x^{j-1}, y^i) + c_{M_1}(x_j, -) \\ d_{M_1}(x^j, y^{i-1}) + c_{M_1}(-, y_i) \\ d_{M_1}(x^{j-1}, y^{i-1}) + c_{M_1}(x_j, y_i) \end{cases}$$

Beweis:

Oben haben wir uns überlegt, dass für die Kosten eines Alignments die Reihenfolge, in der die Operationen durchgeführt werden, keine Rolle spielt. Ein optimales Alignment von x^j und y^i kann die Operation (x_j, y_i) enthalten oder nicht.

Ein bestes Alignment, das (x_j, y_i) enthält, erhalten wir, indem wir ein optimales Alignment für x^{j-1} und y^{i-1} nehmen und als letzte Spalte die Spalte $\begin{bmatrix} x_j \\ y_i \end{bmatrix}$ hinzufügen.

Falls ein optimales Alignment (x_j, y_i) nicht enthält, dann muss dieses die Operation $(x_j, -)$ oder die Operation $(-, y_i)$ enthalten.

Obige Minimumsbildung wählt unter diesen drei Alternativen eine optimale aus.

$\Rightarrow$

Behauptung. ∎

Obiger Satz impliziert die Korrektheit des folgenden Algorithmus zur Berechnung der Editierdistanz zweier Strings.

Algorithmus EdDisM_1

Eingabe: Strings $x = x_1 x_2 \cdots x_n$, $y = y_1 y_2 \cdots y_m \in \Sigma^+$, Kostenfunktion c_{M_1}.

Ausgabe: Editierdistanz $d_{M_1}(x, y)$

Methode:

$e(x^0, y^0) := 0$;
for j from 1 to n
  do
    $e(x^j, y^0) := e(x^{j-1}, y^0) + c_{M_1}(x_j, -)$
  od;
for i from 1 to m
  do
    $e(x^0, y^i) := e(x^0, y^{i-1}) + c_{M_1}(-, y_i)$
  od;

for j from 1 to n

do

for i from 1 to m

do

$$e(x^j, y^i) := \min \begin{cases} e(x^{j-1}, y^i) + c_{M_1}(x_j, -) \\ e(x^j, y^{i-1}) + c_{M_1}(-, y_i) \\ e(x^{j-1}, y^{i-1}) + c_{M_1}(x_j, y_i) \end{cases}$$

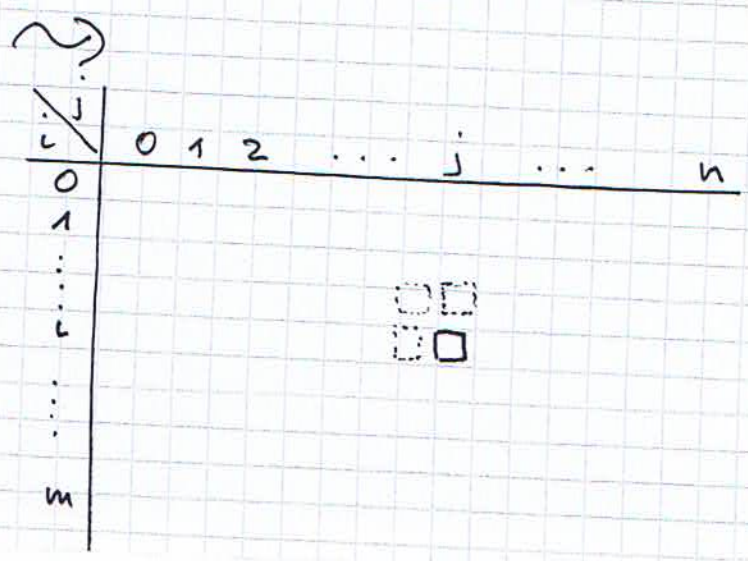
od

od ;

$d_{M_1}(x, y) := e(x^n, y^m)$.

Fragen:

- Wie implementieren wir den Algorithmus EDDisM_1? Welche Datenstruktur verwenden wir?
- Wie erhalten wir ein oder alle zu optimalen Ableitungen korrespondierenden Alignments?
- Was ist die Zeit- und die Platzkomplexität des Algorithmus?



$(n+1) \times (n+1)$ - Matrix Γ

In Position $[i,j]$ der Matrix wird der Wert $e(x^i, y^j)$ eingetragen.

Ziel:
Berechnung aller zu den optimalen Ableitungen korrespondierenden Alignments.

Beobachtung:
Es genügt, sich zusätzlich zum Wert $e(x^i, y^j)$ genau diejenigen der drei Alternativen zu merken, die zum Minimum führen.

Hierzu betrachten wir den zur obigen Matrix korrespondierenden Editiergraphen $E_{m,n}(x,y) := (V, E)$, wobei

$$V := \{ [i,j] \mid 0 \le i \le m, 0 \le j \le n \} \text{ und}$$

$$\begin{aligned} E := \{ &([i,j],[i,j+1]), ([i,j],[i+1,j]), \\ &([i,j],[i+1,j+1]) \mid 0 \le i < m, 0 \le j < n \} \\ \cup \{ &([m,j],[m,j+1]) \mid 0 \le j < n \} \\ \cup \{ &([i,n],[i+1,n]) \mid 0 \le i < m \}. \end{aligned}$$

Jede Kante $e := ([i,j],[i',j'])$ korrespondiert auf folgende Art und Weise zu einer Editieroperation $op(e)$: