Kapitel 5

Textalgorithmen

5.3 Der Algorithmus von Aho und Corasick

Der Zweck des hier vorgestellten Algorithmus ist, in einem Text $S = S[1..n]$ über einem Alphabet Σ nach Mustern aus einer endlichen Menge $\Pi = \{P_1, \ldots, P_r\} \subseteq \Sigma^+$ zu suchen, und alle Vorkommen zu melden. Der Algorithmus wurde 1975 von Alfred V. Aho und Margaret J. Corasick vorgestellt[1].

1. Schritt: Preprocessing/Vorverarbeitung von Π.
Die Rechenzeit ist O(Gesamtlänge aller Suchmuster).

2. Schritt: Durchlauf durch S. Der Algorithmus gibt für jede Stelle i und jedes Muster P_t das Paar (t, i) aus, falls $S[i..i + |P_t| - 1] = P_t$.
Die Rechenzeit ist $O(n +$ Länge der Ausgabe$)$, unabhängig von Π.

(Die Rechenzeit könnte länger als $O(n)$ sein, wenn die Ausgabe umfangreicher als n ist. Dies kann durchaus passieren, wenn viele Muster an der gleichen Stelle vorkommen, zum Beispiel wenn $\Pi = \{a, a^2, a^3, a^4, \ldots, a^r\}$ und $S = a \ldots a$ ist. Wir nehmen an, dass $\varepsilon \notin \Pi$ ist, da das Suchwort ε trivialerweise an jeder Stelle des Textes S vorkommt.)

Wir diskutieren den Algorithmus anhand eines konzeptuellen Baums $T_{\mathrm{Pref}(\Pi)}$, der als

[1] Alfred V. Aho, Margaret J. Corasick, Efficient String Matching: An Aid to Bibliographic Search. Commun. ACM 18(6): 333-340 (1975), `https://doi.org/10.1145/360825.360855`.

Knoten die Menge aller Präfixe

$$\text{Pref}(\Pi) := \{w \mid \exists t \in \{1, \ldots, r\} \colon w \text{ ist Präfix von } P_t\}$$

von Mustern in Π hat. Knoten $w \in \text{Pref}(\Pi)$ hat die Knoten wa als Kinder, für die $wa \in \text{Pref}(\Pi)$ ist. Die Knoten, die in Π sind (Blätter oder nicht) werden markiert. Es ist klar, dass $\text{Pref}(\Pi)$ maximal $1 + |P_1| + \cdots + |P_r|$ Elemente hat. Mit $\text{length}(\Pi)$ bezeichnen wir $|P_1| + \cdots + |P_r|$, die Gesamtlänge aller Muster in Π.

Beispiel: $\Pi = \{P_1, \ldots, P_7\} = \{\text{bei}, \text{beide}, \text{beine}, \text{eis}, \text{eid}, \text{ein}, \text{nein}\}$. Der entsprechende Baum ist in Abb. 5.1 dargestellt.

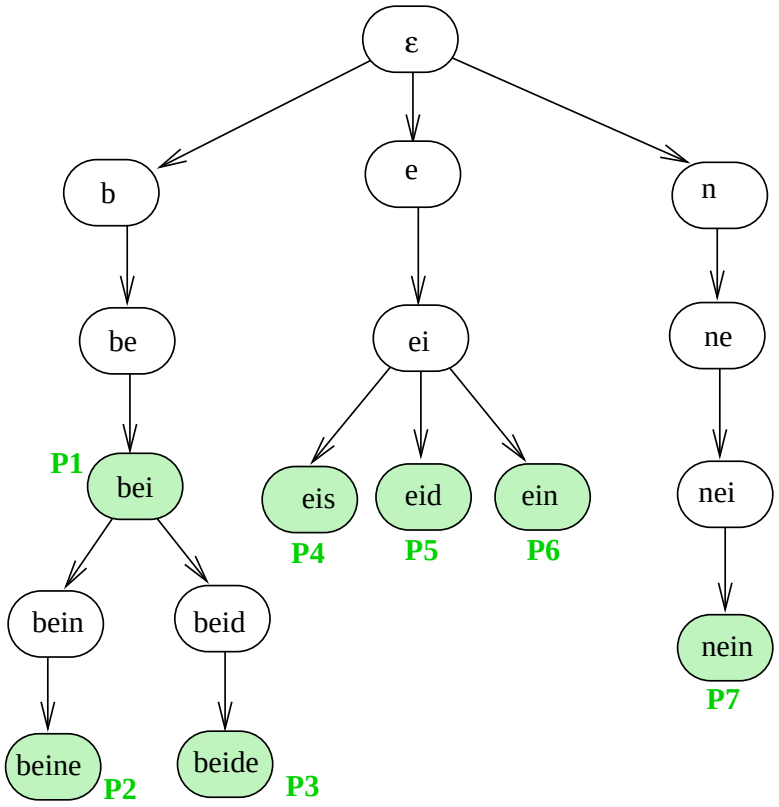


Abbildung 5.1: Baum der Präfixe für eine Menge Π

Wir verändern diesen Baum etwas, ohne dass Information verlorengeht: Wir schreiben den Buchstaben a an die Kante von w nach wa. Die ursprüngliche Beschriftung eines

Knotens ist dann das Wort, das an den Kanten von der Wurzel zu diesem Knoten abzulesen ist. Wir nummerieren die Knoten (im Wesentlichen beliebig, zum Beispiel in Präorder-Reihenfolge) durch, die Wurzel erhält dabei die Nummer 0. Die Markierung der Knoten, die Wörtern aus Π entsprechen, wird beibehalten. Zudem gibt es einen künstlichen Knoten mit Nummer -1, der außerhalb des Baums sitzt. Die Menge $\{0, 1, \ldots, |\mathrm{Pref}(\Pi)|-1\}$ der Baumknoten heißt $V_\Pi = V$, der Knoten -1 kommt hinzu. Das Wort $w \in \mathrm{Pref}(\Pi)$, das dem Knoten v entspricht, wollen wir $w(v)$ nennen. Ein Beispiel ist in Abb. 5.2 angegeben.

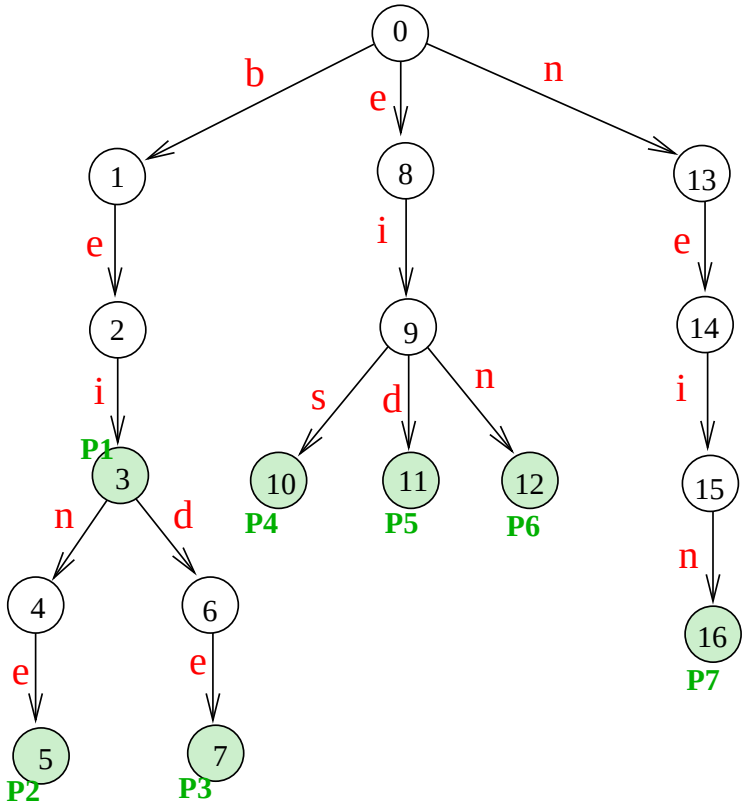


Abbildung 5.2: Baum (Trie) T_Π. Die Muster stehen an den Wegen. Es gilt beispielsweise $w(0) = \varepsilon$, $w(4) =$ bein, $w(9) =$ ei, usw.

Wir nehmen an, dass dieser Baum T_Π vorliegt, und dass man in ihm in konstanter Zeit pro Schritt wie folgt navigieren kann: (i) zu v seinen Vorgänger $p(v)$ und den Buchstaben a an der Kante $(p(v), v)$ finden; (ii) zu Knoten v und Buchstabe a den

a-Nachfolger von v finden, falls er existiert. (Man benötigt hier eigentlich eine Datenstruktur „Trie“, deren Konstruktion eigenen Aufwand erfordert. Beschrieben wird sie in vielen Büchern zu Datenstrukturen, beispielsweise im Buch von Ottmann und Widmayer.)

Der Platzbedarf für T_Π ist $O(|V|) = O(|\text{Pref}(\Pi)|) = O(\text{length}(|\Pi|)$. Wir können uns vorstellen, dass wir zwischen Knoten in T_Π Zeiger setzen können. Dies wird einfach durch die Angabe von Knotennummern realisiert.

Diesen Baum wollen wir so präparieren, dass ein Textsuchalgorithmus im Stil von KMP (mit f_{bord} anstelle von f_{KMP}) möglich ist. Dabei soll der Platzbedarf $O(\text{length}(\Pi))$ bleiben, ganz gleich wie groß Σ ist.

Anmerkung: Alternativ kann man aus T_Π einen endlichen Automaten mit Zustandsmenge V bauen, der für jedes $v \in V$ und jeden Buchstaben $a \in \Sigma$ den Nachfolgezustand $\delta(v, a)$ angibt. Die Tabelle für δ erfordert Platz $O(\text{length}(\Pi) \cdot |\Sigma|)$, was für kleine Alphabete eventuell akzeptabel, für große Alphabete normalerweise nicht erwünscht ist.

Wir definieren eine Fehlerfunktion $f\colon V \to V \cup \{-1\}$. In der abstrakten Beschreibung des Baumes, in der die Knoten Präfixe sind, soll die Fehlerfunktion zu w das längste *echte* Suffix von w finden, das in Pref(Π) ist. *Beispiele*: f(nei) „=“ ei, f(eid) „=“ ε, f(beine) „=“ ne, usw. (Natürlich ist $f(\varepsilon)$ „=“ -1.) Wir wollen aber f in der T_Π-Version des Baumes definieren, bei der die Knoten durch Nummern gegeben sind.

Was wir noch brauchen, ist für jeden Knoten $v \in V$ die Information, ob $w(v)$ ein echtes Suffix hat, das in Π ist. (Ob $w(v)$ selbst in Π ist, steht schon an dem Knoten.) Diese Information wird gegebenenfalls mit einem Wert $g(v) = g_\Pi(v) \in \{1, \ldots, |V|-1\}$ angegeben, der auf das längste echte Suffix von $w(v)$ zeigt, *das in* Π *ist*, falls so etwas existiert. Falls $w(v)$ kein echtes Suffix in Π hat, setzen wir $g_\Pi(v) = 0$. Beispielsweise ist $w(4) =$ bein, und dieses Wort hat $w(12) =$ ein als Suffix. Es wird also $g_\Pi(4) = 12$ sein.

Fehlerfunktion f und „gefunden“-Funktion g werden in einer Vorverarbeitungsprozedur berechnet, die der Berechnung der Randfunktion f_{bord} ähnelt.

Algorithmus 5.3.1 (**AC: Berechnung der Fehlerfunktion**).
AC-Preprocessing(Π)
Eingabe: Menge Π von Mustern, gegeben als T_Π;
Wenn $w(v) = P_t$ für ein t, dann ist v mit „P_t“ markiert;
Ausgabe: `f[0..`$|V|-1$`]` und `g[0..`$|V|-1$`]` als Tabellen;
(1) `g[0]` $\leftarrow 0$;
(2) `f[0]` $\leftarrow -1$;
(3) **for** jedes Kind v der Wurzel **do** `f[`v`]` $\leftarrow 0$; `g[`v`]` $\leftarrow 0$;
(4) **for** die restlichen Knoten v in Levelorder (ebenenweise!) **do**
(5) $u \leftarrow p(v)$; // Vorgänger; wir wissen: $u \neq 0$
(6) $a \leftarrow$ Buchstabe an Kante (u, v);
(7) $z \leftarrow$ `f[`u`]`;
(8) **while** $z \neq -1 \wedge z$ hat kein a-Kind **do** $z \leftarrow$ `f[`z`]`;
(9) **if** $z = -1$ **then** `f[`v`]` $\leftarrow 0$ **else** `f[`v`]` $\leftarrow$ das a-Kind von z;
(10) **if** `f[`v`]` ist mit „P_t“ markiert **then** `g[`v`]` $\leftarrow$ `f[`v`]` **else** `g[`v`]` $\leftarrow$ `g[f[`v`]]`.

Was passiert in diesem Algorithmus? Wir ignorieren zunächst die Funktion g. Zeilen (1) und (2) setzen `f[`v`]` auf den richtigen Wert für die Knoten, die zu ε und zu den einbuchstabigen Präfixen gehören. Die levelweise Verarbeitung der anderen Knoten stellt sicher, dass bei Bearbeitung des Knotens v die Knoten für alle Wörter, die kürzer als $w(v)$ sind, schon bearbeitet sind. Wir bearbeiten v. Zeilen (5)–(7) identifizieren den Vorgänger u (der nicht die Wurzel ist) und den Buchstaben a an der Kante von u nach v. Die **while**-Schleife sucht das längste echte Suffix von $w(u)$, so dass der zugehörige Knoten einen a-Nachfolger hat. (Man kann sich leicht überlegen, dass der Knoten für das längste echte Suffix von $w(v) = w(u)a$ genau der a-Nachfolger dieses Knotens sein muss.) Um dies durchzuführen, benutzt die Schleife die Funktion f auf Knoten weiter oben im Baum, wo sie (nach Induktionsvoraussetzung) schon korrekt berechnet ist. Wenn die **while**-Schleife mit $z = -1$ endet, dann gibt es kein echtes Suffix von $w(u)$ in Pref(Π), das einen a-Nachfolger hat, also hat $w(v)$ kein echtes Suffix in Pref(Π), und es ist $f(v) = 0$. Wenn ein passendes z gefunden wird, dann ist $f(v)$ der a-Nachfolger dieses Knotens.

Noch ein Wort zur Berechnung der g-Funktion (folgend einem Hinweis eines Studierenden): Wir wollen alle Suffixe von $w(v)$ finden, die in Π liegen. (Man beachte, dass dies eine andere Anforderung ist als das längste Suffix in Pref(Π).) Wenn v mit P_t markiert ist, ist dies für v der Fall. Es kann aber auch sein, dass ein *echtes* Suffix w' von $w(v)$ zu Π gehört. Weil $w(f(v))$ das längste Suffix von $w(v)$ in Π *überhaupt* ist, muss dann w' (echtes oder unechtes) Suffix von $w(f(v))$ sein. Wir können annehmen

(nach Induktionsvoraussetzung), dass Knoten $z = f(v)$ schon die richtige Information hat: Dieser Knoten könnte selbst mit einem P_t markiert sein (dann lassen wir $g(v)$ darauf verweisen) oder mit $g(z) \neq 0$ zu dem längsten Suffix von $w(f(v))$ führen, *das in* Π *ist*. – Man kann sich nun überlegen, dass für jeden Knoten v das iterierte Verfolgen der g-Verweise (v, $g(v)$, $g(g(v))$, ..., bis 0 erreicht ist) zu den Knoten für alle Suffixe von $w(v)$ führt, die in Π liegen. Bei jedem Iterationsschritt wird ein neues solches Suffix gefunden. Der Zeitaufwand für das Durchlaufen dieser Kette ist linear in der Anzahl der betreffenden Suffixe.

Satz 5.3.2. *Algorithmus 5.3.1 hat Zeitbedarf* $O(\mathit{length}(\Pi))$. *Er berechnet korrekt die Fehlerfunktion* f *auf den Knoten des Baum* T_Π *und die „gefunden"-Funktion* g.

Beweis: Der Beweis der *Korrektheit* ist praktisch der gleiche wie bei der Berechnung der Randfunktion (s. Übung). Man benutzt induktiv, dass die $f(v')$-Werte für v' in Levels oberhalb von v schon korrekt berechnet sind, und führt eine Unterinduktion durch, die die folgende Invariante benutzt:

$w(z)$ ist echtes Suffix von $w(u)$; es gibt kein längeres echtes Suffix w' von $w(u)$ derart dass $w'a \in \mathrm{Pref}(\Pi)$ ist.

Laufzeitanalyse: Sei P_t eines der Muster, die zu einem Blatt gehören. Man betrachtet den Zeitaufwand für die Knoten v auf dem Weg im Baum, der zu diesem Muster gehört. Hierzu wendet man die Technik aus der Zeitanalyse des KMP-Algorithmus an, auch wenn es keine explizite Variable `q` gibt. Wir benutzen einen gedachten Zähler q, der anfangs auf 0 steht. Wenn der nächste Knoten v auf dem Weg bearbeitet wird, erhöhen wir q um 1. Wenn in der **while**-Schleife in Zeile (8) die Operation $z \leftarrow \mathtt{f[}z\mathtt{]}$ ausgeführt wird, verringern wir q um die Differenz der beiden Werte $|w(z)|$ und $|w(\mathtt{f[}z\mathtt{]})|$, also mindestens um 1. Man sieht dann leicht, dass stets $q \geq |w(f(v))|$ ist, für den Knoten, dessen Bearbeitung eben abgeschlossen wurde. Das heißt: Am Ende ist $q \geq 0$. Genau wie in der KMP-Analyse folgt daraus, dass die Anzahl der Durchläufe durch die **while**-Schleife bei der Bearbeitung der Knoten auf dem P_t-Weg nicht größer als $|P_t|$ ist. Die gesamte Anzahl der Durchläufe ist also nicht größer als $|P_1| + \cdots + |P_t| = \mathrm{length}(\Pi)$. □

Wir führen den Algorithmus am in Abb. 5.2 gegebenen Baum durch.

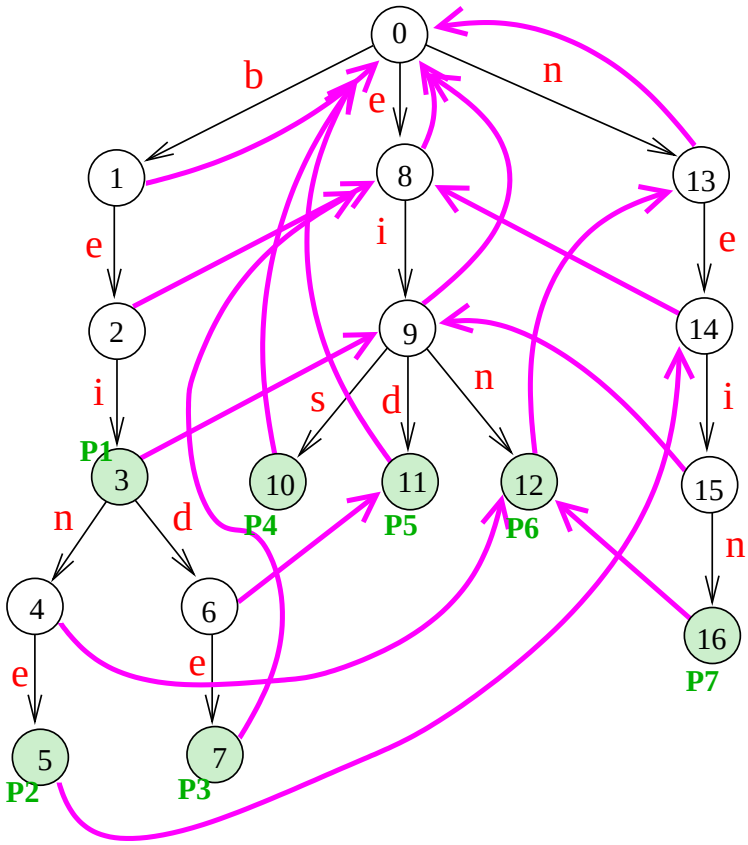
$f(0) \leftarrow -1$ Wurzel
$f(1) \leftarrow 0$ Kind der Wurzel
$f(8) \leftarrow 0$ Kind der Wurzel
$f(13) \leftarrow 0$ Kind der Wurzel
Knoten 2: $u = 1$, $a =$ e, $z = f(1) = 0$,Knoten $z = 0$ hat e-Nachfolger $8 =: f(2)$
Knoten 9: $u = 8$, $a =$ i, $z = f(8) = 0$, Knoten $z = 0$ hat keinen i-Nachfolger:
$z \leftarrow f(z) = -1$, Schleifenabbruch, $f(9) \leftarrow 0$
Knoten 14: $u = 13$, $a =$ e, $z = f(13) = 0$, Knoten $z = 0$ hat e-Nachfolger $8 =: f(14)$
Knoten 3: $u = 2$, $a =$ i, $z = f(2) = 8$, Knoten $z = 8$ hat i-Nachfolger $9 =: f(3)$
Knoten 10: $u = 9$, $a =$ s, $z = f(9) = 0$, Knoten $z = 0$ hat keinen s-Nachfolger:
$z \leftarrow f(z) = -1$, Schleifenabbruch, $f(10) \leftarrow 0$
Knoten 11: $u = 9$, $a =$ d, $z = f(9) = 0$, Knoten $z = 0$ hat keinen d-Nachfolger:
$z \leftarrow f(z) = -1$, Schleifenabbruch, $f(11) \leftarrow 0$
Knoten 12: $u = 9$, $a =$ n, $z = f(9) = 0$, Knoten $z = 0$ hat n-Nachfolger $13 =: f(12)$
Knoten 15: $u = 14$, $a =$ i, $z = f(14) = 8$, Knoten $z = 8$ hat i-Nachfolger $9 =: f(15)$
Knoten 4: $u = 3$, $a =$ n, $z = f(3) = 9$, Knoten $z = 9$ hat n-Nachfolger $12 =: f(4)$
12 ist mit „P_6“ markiert, daher: $g(4) \leftarrow 12$
Knoten 6: $u = 3$, $a =$ d, $z = f(3) = 9$, Knoten $z = 9$ hat d-Nachfolger $11 =: f(6)$
11 ist mit „P_5“ markiert, daher: $g(6) \leftarrow 11$
Knoten 16: $u = 15$, $a =$ n, $z = f(15) = 9$, Knoten $z = 9$ hat n-Nachfolger $12 =: f(16)$
12 ist mit „P_6“ markiert, daher: $g(16) \leftarrow 12$
Knoten 5: $u = 4$, $a =$ e, $z = f(4) = 12$, Knoten $z = 12$ hat keinen e-Nachfolger;
$z \leftarrow f(12) = 13$, Knoten $z = 13$ hat e-Nachfolger $14 =: f(5)$
Knoten 7: $u = 6$, $a =$ e, $z = f(6) = 11$, Knoten $z = 11$ hat keinen e-Nachfolger;
$z \leftarrow f(11) = 0$, Knoten $z = 0$ hat e-Nachfolger $8 =: f(7)$.

Die resultierende Funktion f ist in Abb. 5.3 graphisch dargestellt. Dabei wurden der Übersichtlichkeit halber die „g-Pfeile“ für $g(4) = g(16) = 12$ und $g(6) = 11$ nicht dargestellt. Alle anderen $g(v)$-Werte sind 0.

Mit Hilfe der Fehlerfunktion f und der „gefunden“-Funktion g können wir nun in einem gegebenen Text nach Vorkommen von Mustern aus Π suchen.

Der Algorithmus benutzt den Baum T_Π mit seiner Fehlerfunktion ähnlich wie einen endlichen Automaten. Wir bezeichnen die Knoten v in diesem Zusammenhang daher auch als „Zustände“.

Abbildung 5.3: Die Fehlerfunktion f



Startzustand/-knoten: $v_0 = 0$.

Die Buchstaben $S[i]$ werden in Runden $i = 1, \ldots, n$ bearbeitet, nach Runde i ist ein Knoten v_i erreicht.

Invariante: Am Ende von Runde i gilt Folgendes:
Die Inschrift auf dem Weg zu v_i ist das längste Präfix in Pref(Π), das Suffix von $S[1..i]$ ist.

Nehmen wir an, Runde $i-1$ ist zuende, und Knoten $v = v_{i-1}$ ist erreicht, mit $p_{i-1} = w(v_{i-1})$.

Sei $a = S[i]$ der nächste Buchstabe.

1. Fall: $p_{i-1}a \in \text{Pref}(\Pi)$.
(Das erkennt man daran, dass v einen a-Nachfolger v' hat.)
$v_i \leftarrow v'$.
Wenn v' mit einem P_t markiert ist: Paar $(t, i - |P_t| + 1)$ in die Ausgabe;
wenn $g(v') > 0$: Verfolge die Kette $g(v'), g(g(v')), \ldots$; jeder so erreichte Knoten u ist mit einem Muster P_t markiert, gib $(t, i - |P_t| + 1)$ aus.

2. Fall: $p_{i-1}a \notin \text{Pref}(\Pi)$.
(Das erkennt man daran, dass v keinen a-Nachfolger v' hat.)
Setze $v' \leftarrow v$, und führe folgende Schleife aus:
while $v' \neq -1 \wedge v'$ hat kein a-Kind **do** $v' \leftarrow f(v')$.
(Man sucht auf diese Weise das längste Suffix w' von $w(v)$ mit $w'a \in \text{Pref}(\Pi)$, falls so etwas existiert.)
Fall 2a: v' ist ein Knoten mit a-Kind. Setze $v_i \leftarrow v'$.
Fall 2b: $v' = -1$. Setze $v_i \leftarrow 0$.

Algorithmus 5.3.3 (**Aho-Corasick, mehrere Muster**).
AC-Textsuche(`S[1..n]`, Π)
Eingabe: Π: Menge von Mustern;
`S[1..n]`: Text;
Vorberechnet: Baum T_Π, Knoten haben Nummern $0, \ldots, m$; 0 ist Wurzel;
Fehlerfunktion f und „gefunden“-Funktion g;
Ausgabe: Menge `A` von Paaren; initialisiert: `A` $\leftarrow \emptyset$;

```
(1)  int v ← 0;
(2)  for i from 1 to n do
(3)    while v ≥ 0 ∧ v hat kein S[i]-Kind do v ← f[v];
(4)    if v ≥ 0 then v ← S[i]-Kind von v else v ← 0;
(5)    u ← v;
(6)    repeat
(7)      if u ist mit P_t markiert then A ← A ∪ {(t, i − |P_t| + 1)};
(8)      u ← g[u]
(9)    until u = 0.
```

Wir führen den Algorithmus an der Beispieleingabe

$$S[1..21] = \text{esbeidebeineineisbiss}$$

durch. Wenn für einen Knoten v kein g-Wert angegeben ist, ist dieser 0. Wenn für einen Knoten v nicht vermerkt ist, dass er mit P_t markiert ist, ist er unmarkiert.

Startzustand $v = 0$;

$S[1] = \text{e}$	von $v = 0$ zu e-Kind $v = 8$
$S[2] = \text{s}$	$v = 8$ hat kein s-Kind; $v \leftarrow f(8) = 0$;
	$v = 0$ hat kein s-Kind; $v \leftarrow f(0) = -1$;
	Schleifenabbruch: setze $v \leftarrow 0$;
$S[3] = \text{b}$	von $v = 0$ zu b-Kind $v = 1$;
$S[4] = \text{e}$	von $v = 1$ zu e-Kind $v = 2$;
$S[5] = \text{i}$	von $v = 2$ zu i-Kind $v = 3$; mit $P_1 =$ bei markiert, Ausgabe $(1, 3)$;
$S[6] = \text{d}$	von $v = 3$ zu d-Kind $v = 6$; $g(6) = 11$, mit $P_5 =$ eid markiert, Ausgabe $(5, 4)$;
$S[7] = \text{e}$	von $v = 6$ zu e-Kind $v = 7$; mit $P_3 =$ beide markiert, Ausgabe $(3, 3)$;
$S[8] = \text{b}$	$v = 7$ hat kein b-Kind; $v \leftarrow f(7) = 8$;
	$v = 8$ hat kein b-Kind; $v \leftarrow f(8) = 0$;
	von 0 zu b-Kind $v = 1$;
$S[9] = \text{e}$	von $v = 1$ zu e-Kind $v = 2$;
$S[10] = \text{i}$	von $v = 2$ zu i-Kind $v = 3$; mit $P_1 =$ bei markiert, Ausgabe $(1, 8)$;
$S[11] = \text{n}$	von $v = 3$ zu n-Kind $v = 4$; $g(4) = 12$, mit $P_6 =$ ein markiert, Ausgabe $(6, 9)$;
$S[12] = \text{e}$	von $v = 4$ zu e-Kind $v = 5$; mit $P_2 =$ beine markiert, Ausgabe $(2, 8)$;
$S[13] = \text{i}$	$v = 5$ hat kein i-Kind; $v \leftarrow f(5) = 14$;
	von $v = 14$ zu i-Kind $v = 15$;
$S[14] = \text{n}$	von $v = 15$ zu n-Kind $v = 16$; mit $P_7 =$ nein markiert, Ausgabe $(7, 11)$;
	$g(16) = 12$, mit $P_6 =$ ein markiert, Ausgabe $(6, 12)$; $g(12) = 0$;
$S[15] = \text{e}$	$v = 16$ hat kein e-Kind; $v \leftarrow f(16) = 12$;
	$v = 12$ hat kein e-Kind; $v \leftarrow f(12) = 13$;
	von $v = 13$ zu e-Kind $v = 14$;
$S[16] = \text{i}$	von $v = 14$ zu i-Kind $v = 15$;
$S[17] = \text{s}$	$v = 15$ hat kein s-Kind; $v \leftarrow f(15) = 9$;
	von $v = 9$ zu s-Kind $v = 10$; mit $P_4 =$ eis markiert, Ausgabe $(4, 15)$;
$S[18] = \text{b}$	$v = 10$ hat kein b-Kind; $v \leftarrow f(10) = 0$;
	von $v = 0$ zu b-Kind $v = 1$;
$S[19] = \text{i}$	$v = 1$ hat kein i-Kind; $v \leftarrow f(1) = 0$;
	$v = 0$ hat kein i-Kind; $v \leftarrow f(0) = -1$;
	Schleifenabbruch: setze $v \leftarrow 0$;
$S[20] = \text{s}$	$v = 0$ hat kein s-Kind; $v \leftarrow f(0) = -1$;
	Schleifenabbruch: setze $v \leftarrow 0$;
$S[21] = \text{s}$	$v = 0$ hat kein s-Kind; $v \leftarrow f(0) = -1$;
	Schleifenabbruch: setze $v \leftarrow 0$;

Satz 5.3.4. *Algorithmus 5.3.3 hat Zeitbedarf $O(n+$ Länge der Ausgabe$)$. Er berechnet korrekt sämtliche Stellen in S, an denen ein Muster aus Π vorkommt.*

Beweis: Die **Zeitanalyse** ist praktisch identisch zu der des KMP-Algorithmus. Wenn v der aktuelle Zustand ist, ist $q = |w(v)|$ (also das Level von Knoten v) die geeignete Maßzahl (die man als Potential auffassen kann). Anfangs ist $q = 0$, am Ende ist $q \geq 0$. Mit jedem gelesenen Buchstaben wird q um 1 erhöht, mit jedem Durchlauf durch den Rumpf der **while**-Schleife wird q strikt erniedrigt. Daher kann es maximal n solche Durchläufe geben.

Für die Korrektheitsanalyse zeigt man durch Induktion über die Schleifendurchläufe die folgenden Invarianten:

Äußere Schleife: Nach Zeile (5) gilt: $w(v_i)$ ist das längste Suffix von $S[1..i]$, das in Pref(Π) vorkommt.

Für die **while**-Schleife, in Runde i: $w(v)$ ist Suffix von $S[1..i-1]$, und es gibt in Pref(Π) kein Suffix w' von $S[1..i]$ mit Länge $> |w(v)| + 1$.

(Die Argumentation ist dabei sehr ähnlich zu der im Beweis der Korrektheit des KMP-Algorithmus.) □

Übungsaufgabe: Erstelle Baum T_Π, Fehlerfunktion f und „gefunden"-Funktion g für $\Pi = \{\text{dein}, \text{ein}, \text{herein}, \text{rein}, \text{sein}, \text{dasein}, \text{in}\}$.

Wende dann den Suchalgorithmus auf das Eingabewort $S =$ „deinhereinseindasein" an. Beachte besonders die Wirkungsweise der g-Funktion an der Stelle, wo das Teilwort „deinherein" gelesen worden ist.