Mathematische Strukturen

zur Musteridentifizierung

auf mehrdimensionalen Zeichenfeldern

Diplomarbeit

bei Dr. Michael Pfender

von

Jochen Burghardt

Technische Universitaet Berlin

Fachbereich Mathematik

September 1985

INHALT:
======

0.1 Einleitung
0.2 Literatur
0.3 Definition der benoetigten Grundbegriffe
0.3.1 Strings
0.3.2 Baeume
0.3.3 Ordnungsrelationen

1 Der eindimensionale Fall
1.1 Ausgangspunkt
1.2 Uebersicht ueber die Algorithmen in der Literatur
1.3 Kurzdarstellung der wichtigsten Algorithmen
1.3.0 Straight-Forward-Pattern-Matching
1.3.1 Knuth-Morris-Pratt
1.3.2 Aho-Corasick
1.3.3 Rosenberg
1.3.4 Karp-Miller-Rosenberg
1.3.5 desgl. fuer Arrays
1.4 Theorie der Position-Identifier
1.5 Die Prefix-Tree-Algorithmen
1.5.6 Weiners Alg.D
1.5.7 Weiners Alg.B
1.5.8 Weiners Alg.C
1.5.9 McCreight
1.6 Der von mir entwickelte Algorithmus
1.7 Korrektheit und Aufwand
1.8 Der Algorithmus fuer den kompaktifizierten Bi-Tree
1.9 Anwendungsbeispiele

2 Der mehrdimensionale Fall
2.1 Einleitung
2.2 Minimal-Identifier
2.3 Zeichenfelder mit Positionen
2.4 Position-Identifier
2.5 Die Sortierstruktur
2.6 S-Father-Zeiger
2.7 Bloecke
2.8 Zusammenhang zwischen xdulin und uvor
2.9 Das Richtungskonzept
2.10 Das End-Abschnitts-Konzept

3 Ein Algorithmus fuer den mehrdimensionalen Fall
3.1 Beschreibung des Algorithmus
3.1.1 Die verwendete Datenstruktur
3.1.2 Der Algorithmus
3.1.3 Ablaufbeispiel
3.2 Korrektheit und Aufwand
3.3 Literatur
3.4 Ausblick

4 Anhang
4.0 Schreibweisen
4.1 Aufwandsklassen fuer Algorithmen
4.2 Ein Satz aus der Kombinatorik
4.3 Implementierung des eindimensionalen Algorithmus
4.4 Implementierung des eindimensionalen Algorithmus (kompaktifiziert)

KAPITEL 0.1 Einleitung

Mit Hilfe von "Prefix-Tree-Algorithmen" lassen sich eine ganze Reihe interessanter und wichtiger Probleme der Musteridentifizierung bei eindimensionalen Zeichenketten (Strings) loesen.

Im ersten Teil dieser Arbeit werden verschiedene Algorithmen zur Musteridentifizierung (Pattern-Matching und Repetition-Finding) vorgestellt und verglichen. Dabei wird der Schwerpunkt auf die Prefix-Tree-Algorithmen und ihre unterschiedlich komplexen und effizienten Datenstrukturen gelegt. Es wird der Begriff der Minimal-Identifier eines Strings eingefuehrt und damit werden wichtige Aussagen ueber Form und Lage der Position-Identifier - der Grundelemente fuer den Prefix-Tree - gezeigt.

Es werden zwei Prefix-Tree-Algorithmen vorgestellt, die im Gegensatz zu den in der Literatur bekannten Algorithmen "online" arbeiten, d.h. nach jedem Einlesen eines Zeichens sofort ein vorlaeufiges Ergebnis liefern. Schliesslich werden verschiedene Anwendungen der Prefix-Trees vorgestellt.

Das Ziel dieser Arbeit ist, einen verallgemeinerten Prefix-Tree-Algorithmus anzugeben, der nicht nur eindimensionale Strings verarbeiten kann. Dazu ist gleichzeitig eine Klasse von "Zeichenfeldern" zu definieren, deren Eigenschaften hinreichend fuer das Funktionieren eines solchen Algorithmus' sind, wobei umgekehrt der Algorithmus auf diesen Eigenschaften beruht.

Im zweiten Teil der Arbeit wird eine solche Klasse von Zeichenfeldern definiert, und es werden die Eigenschaften nachgewiesen, die fuer den konzipierten Algorithmus benoetigt werden. Als Orientierung dienen dabei die im ersten Teil dargestellten Erfahrungen mit dem eindimensionalen String-Fall. Fuer eine genauere Uebersicht ueber die Struktur des zweiten Teils sei daher auf Kapitel 2.1 verwiesen.

Im dritten Teil schliesslich wird der Algorithmus vorgestellt und einige Anwendungen skizziert.

Diese Arbeit wurde mit Hilfe eines Textverarbeitungssystems erstellt, das mir von der Firma MSB Systemberatung Bardohl freundlicherweise zur Verfuegung gestellt wurde.

Ich danke Frau Roswitha Bardohl, Herrn Bernhard Bardohl und Herrn Dr. Michael Pfender fuer ihre wertvolle Hilfsbereitschaft und Unterstuetzung.

KAPITEL 0.2 Literatur

AHO 74 A.V.Aho, J.E.Hopcroft, J.D.Ullman,
"The Design and Analysis of Computer Algorithms",
Addison-Wesley, Reading/MA, 1974
Kap 9. "Pattern matching Algorithms"
Regulaere Sprachen (1.3.2),
Darstellung von Weiners Alg.B (1.5.7)

AHO 75 A.V.Aho, M.J.Corasick,
"Efficient String Matching: An Aid to Bibliographic Search",
in: Communications of the ACM, Jun. 1975, Bd. 18, Nr. 6
Aho-Corasick-Algorithmus (1.3.2)

BAK 78 T.P.Baker,
"A Technique for extending rapid exact-match String Matching to Arrays of more than one Dimension",
in: SIAM Journal of Computing, Nov. 1978, Bd. 7, Nr. 4
Knuth-Morris-Pratt-Algorithmus fuer Arrays (3.3)

BIR 77 R.S.Bird,
"Two Dimensional Pattern Matching",
in: Information Processing Letters, Okt. 1977, Bd. 6, Nr. 5
(wie BAK 78)

BOO 80 K.S.Booth,
"Lexicographically least circular Substrings",
in: Information Processing Letters, Jul. 1980, Bd. 10, Nr. 4+5
Eine Anwendung des Knuth-Morris-Pratt-Algorithmus fuer die Graphen-Theorie

EHR 72 H.Ehrig, M.Pfender,
"Kategorien und Automaten",
de Gruyter, Berlin, 1972
Theorie der Endlichen Automaten (1.3.2)

FIS 74-1 R.A.Wagner, M.J.Fischer,
"The String-to-String Correction Problem",
in: Journal of the ACM, Jan. 1974, Bd. 21, Nr. 1
Loest das File-Compare-Problem, wenn keine mehrfachen Kopien erlaubt sind (1.9.6) in der Zeit O(|v|*|w|)

FIS 74-2 M.J.Fischer, M.S.Paterson
"String-Matching and Other Products",
in: SIAM-AMS-Proceedings, 1974, Bd. 7
Gute Beschreibung des Knuth-Morris-Pratt-Algorithmus (1.3.1),
stellt Matching-Probleme als Spezialfall von "linearen Produkten" von Vektoren dar und loest mit Hilfe des Schoenhage-Strassen-Algorithmus das Matching-Problem fuer Strings mit Luecken (2.3.1) in der Zeit
$O(|v|*(\log(|w|))^2*\log(\log(|w|)))$

HIR 75 D.S.Hirschberg,
"A Linear Space Algorithm for Computing Maximal Common Subsequences",
in: Communications of the ACM, Jun. 1975, Bd. 18, Nr. 6
(wie FIS 74-1)

INO 83 M.Toda, K.Inoue, I.Takanami,
"Two-Dimensional Pattern Matching by Two-Dimensional on-line Tessellation Acceptors",
in: Theoretical Computer Science, 1983, Bd. 24
(wie BAK 78)

KAR 72 R.M.Karp, R.E.Miller, A.L.Rosenberg,
"Rapid Identification of Repeated Patterns in Strings, Trees and Arrays",
in: ACM, 4th Symposium on Theory of Computing, Mai 1972
Karp-Miller-Rosenberg-Algorithmus (1.3.4+5)

KNU 73 D.E.Knuth,
"The Art of Computer Programming",
Volume 3, "Sorting and Searching",
Kap. 6.2.3, "Balanced Trees",
AVL-Trees (1.5.7)

KNU 77 D.E.Knuth, J.H.Morris, V.R.Pratt,
"Fast Pattern Matching in Strings",
in: SIAM Journal of Computing, Jun. 1977, Bd. 6, Nr. 2
Knuth-Morris-Pratt-Algorithmus (1.3.1)

MCC 76 E.M.McCreight,
"A Space-Economical Suffix Tree Construction Algorithm",
in: Journal of the ACM, Apr. 1976, Bd. 23, Nr. 2
McCreight-Algorithmus (1.5.9),
Aenderungen des Prefix-Trees bei nachtraeglichen Aenderungen des Eingabestrings

ROS 72 A.L.Rosenberg,
"One-Pass Discovery of Repeated Substrings of a String"
in: IBM Technical Disclosure Bulletin, Mrz. 1972, Bd. 14, Nr. 10
Rosenberg-Algorithmus (1.3.3)

WEI 73-1 P.Weiner,
"Linear Pattern Matching Algorithms",
in: Conf. Record, IEEE 14th Annual Symposium on Switching and Automata Theory, 1973
Weiner-Algorithmen D, B, C (1.5.6-8)

WEI 73-2 P.Weiner,
"The File Transmission Problem", (nur Abstract)
in: AFIPS Conference Proceedings, Jun. 1973, Bd. 42
Beschreibt das File-Transmission-Problem und stellt einen Loesungs-Algorithmus vor (1.9.5)

Ein Grossteil der hier verwendeten Literatur wurde durch eine Datenbank-Anfrage der Universitaetsbibliothek der TU Berlin nachgewiesen.

KAPITEL 0.3 Definitionen der benoetigten Grundbegriffe

0.3.1 Strings

1. Def: Ein Alfabet ist eine endliche oder hoechstens abzaehlbare Menge A, auf der eine totale Ordnung definiert ist.

Bsp: $A = \{a,b,c,\ldots,x,y,z\}$ mit $a<b<c<\ldots<x<y<z$

2. Def: Das freie Monoid ueber A ist die kleinste Menge M, fuer die gilt:
M ist Monoid,
d.h. auf M ist eine assoziative Operation
$* : M \times M \to M$
und ein neutrales Element \$ definiert [1],
ex. $i: A \to M$ injektiv (i heisst Einbettung von A in M),
fa. Monoide M' und fa. $f: A \to M'$
ex. genau ein Monoid-Homomorphismus
$eval: M \to M'$
mit $eval(i(a)) = f(a)$ fa. $a \in A$

Das freie Monoid ueber A wird mit A* bezeichnet, die Elemente von A* heissen Zeichenketten oder Strings von Zeichen aus A.
Fuer $n \in N$ und $a \in A$ sei $a^n := a * \ldots * a$ (n Faktoren)

3. Satz: $A^* = \{\$\} \cup \{(a1,\ldots,an) \mid n \in N \setminus \{0\},\ a1,\ldots,an \in A\}$

Bew: A* Monoid:
$(a1,\ldots,an) * (b1,\ldots,bm) := (a1,\ldots,an,b1,\ldots,bm)$
$(a1,\ldots,an) * \$:= \$ * (a1,\ldots,an) := (a1,\ldots,an)$
definiert eine assoziative Operation mit \$ als neutralem Element
$i(a) := (a)$ ist injektive Einbettung von A in A*
fa. M' Monoid und $f: A \to M'$ waehle
$eval(\$) := \$'$
$eval((a1,\ldots,an)) := f(a1) *' \ldots *' f(an)$
dann ist eval Monoid-Homomorphismus und $eval(i(a))=f(a)$
und eval ist dadurch eindeutig bestimmt

A* ist das kleinste Monoid:
denn A* muss \$ enthalten und mit $a1,\ldots,an \in A$ auch
$(a1), \ldots, (an)$ und damit auch $(a1,\ldots,an) = (a1) * \ldots * (an)$

----

[1] normalerweise mit dem griechischen Buchstaben Epsilon bezeichnet

4. Def: sei v,w ∈ A*,
v heisst Teilstring von w :<==> ex. u1,u2 ∈ A* mit
w = u1*v*u2
v heisst Prefix (Anfangsstueck) von w :<==> ex. u2 ∈ A* mit
w = v*u2
fuer u2 ≠ $ heisst v echter Prefix von w
v heisst Suffix (Endstueck) von w :<==> ex. u1 ∈ A* mit
w = u1*v
fuer u1 ≠ $ heisst v echter Suffix von w

|v| heisst Laenge von v und wird definiert durch:
|$| := 0,
|(a1,...,an)| := n
i.a. schreibt man a statt i(a) fuer a ∈ A.
Strings werden in Beispielen oft der Anschaulichkeit halber mit
ihren Positionen angegeben, d.h. als
a1 a2 a3 ... an
1 2 3 ... n

Bem: Ist u,v,w ∈ A*,
u Prefix von w, v Prefix von w,
so folgt u Prefix von v oder v Prefix von u.
Analog fuer Suffix

5. Bem: ist w ∈ A*, |w| = n,
so gibt es zu 1 <= i,j <= n genau ein v ∈ A*
und genau je ein u,u' ∈ A* mit
|u| = i-1, |v| = j-i+1, |u'| = n-j und u*v*u' = w

6. Def: Das v aus Lemma 5 wird auch mit w[i...j] bezeichnet,
manchmal auch mit w[i...k...j] wenn i <= k <= j (als
suggestive Schreibweise);
weiter sei w[i] := ai , wenn w=(a1,...,an)
und w[i...i-1] := $ fa. i
Wenn irgendwo ein Ausdruck von einer der obigen Formen steht,
wird immer vorausgesetzt, dass die Grenzen vernuenftig sind,
d.h., dass 1 <= i,j,k <= n

7. Satz: (Lexikografische Ordnung auf A*)
Ist < die totale Ordnungsrelation auf A ,
so kann sie zu einer totalen Ordnungsrelation auf A*
fortgesetzt werden durch die rekursive Definition:
u < v :<==> u = $, v = b*v'
oder
u = a*u', v = b*v', (a<b oder a=b, u'<v')
(wobei u,v,u',v' ∈ A*, a,b ∈ A)

Bew: Totalitaet, Irreflexivitaet, Asymmetrie und Transitivitaet zeigt
man leicht durch Induktion ueber |u| (Fallunterscheidung u=$ oder
u=a*u')

8. Satz: Fuer u,v,u',v',t ∈ A* gilt
u = t*u', v = t*v' ==> (u < v <==> u' < v')

Bew: Induktion ueber |t| :
t = $ trivial
t = a*t' fuer a ∈ A, t' ∈ A*:
u < v <==> a*t'*u' < a*t'*v'
<==> a<a oder a=a, t'*u'<t'*v'
<==> t'*u' < t'*v'
<==> u' < v' (Ind.vor.)

9. Def: Fuer u,v ∈ A* heisst t ∈ A* maximaler gemeinsamer Prefix von u und v
:<==> u = t*u', v = t*v' fuer geeignete u',v' ∈ A*
und (u'=$ oder v'=$ oder u'[1]≠v'[1])
Nach 0.3.3.4 (s.u.) ex. stets ein maximaler gemeinsamer Prefix

10. Satz: Fuer u,v ∈ A* gilt:
u < v <==> u ist echter Prefix von v
oder ex. t ∈ A*, a,b ∈ A mit
u = t*a*u', v = t*b*v', a < b

Bew: "<==": u echter Prefix von v
==> v = u*u' fuer ein u' ≠ $, u' ∈ A*
==> $ < u'
==> u = u*$ < u*u' = v (nach 8.)
und: u = t*a*u', v = t*b*v', a < b
==> a*u' < b*v'
==> u < v (nach 8.)

"==>": sei t maximaler gemeinsamer Prefix von u und v,
sei u = t*u', v = t*v' (nach 9.)
Fall 1: u' = $, dann ist u=t Prefix von v
Fall 2: u' = a*u" fuer ein a ∈ A, u" ∈ A*,
dann folgt:
u < v ==> a*u" < v' (nach 8.)
==> v' ≠ $, etwa v' = b*v" und
a<b oder a=b, u"<v"
==> a<b, denn a≠b nach Def von t

0.3.2 Baeume

1. Def: Ein Baum T ist ein gerichteter markierter endlicher Graph, mit einem ausgezeichneten Knoten root (Wurzel), in den keine Kante hineinfuehrt und von dem aus zu jedem anderen Knoten genau ein gerichteter Weg fuehrt. Sei A das Alfabet der Kantenmarkierungen.

Die Zahl der Kanten, die aus einem Knoten herausfuehren, heisst der Grad des Knotens, der Grad von T ist das Maximum der Grade seiner Knoten. Ein Knoten n, aus dem keine Kanten herausfuehren, heisst Blatt von T.

Zu jedem Knoten n ausser root gibt es genau einen Knoten nf, von dem eine gerichtete Kante zu n fuehrt (sonst gaebe es von root zwei Wege zu n: ueber nf und ueber nf'), nf heisst Vaterknoten (father) von n,

nf = father(n),

Dies definiert eine Abbildung father: (|T|\{root}) --> |T| .

Gibt es von einem Knoten n1 einen gerichteten Weg zu einem Knoten n2 (oder ist n1 = n2), so heisst n2 ein Nachkomme von n1 . Ist M die Menge aller Nachkommen von n1 , so heisst T|M (die Einschraenkung von T auf die Knotenmenge M) Unterbaum von T mit der Wurzel n1 .

Die Hoehe (oder Tiefe) eines Knotens n im Baum ist die Zahl der Kanten des Weges von root nach n. Die Hoehe des Baumes ist die Hoehe seines hoechsten Knotens.

Zu jedem Knoten n kann man den String aus den Kantenmarkierungen des Weges von root nach n berechnen. sei l(v) ∈ A die Markierung der Kante v, sei vf(n) die Kante von father(n) nach n, dann definiere:

```
label(n) := { $                              wenn n=root
            { label(father(n)) * l(vf(n))    sonst
```

Man zeigt die Wohldefiniertheit leicht durch Induktion ueber die Hoehe von n.

In dieser Arbeit wird - sofern nicht ausdruecklich anders vereinbart - von einem Baum zusaetzlich verlangt, dass aus keinem Knoten zwei gleich markierte Kanten herausfuehren.

Unter dieser Voraussetzung gibt es eine partielle Abbildung

son': |T|xA --> (|T|\{root})

son'(n,a) = m, wenn vom Knoten n eine mit a markierte Kante in den Knoten m hineinfuehrt.

Man kann daraus eine totale Abbildung

son: |T|xA --> ((|T|\{root}) u {NIL})

machen, indem man son(n,a):=NIL setzt fuer alle (n,a), auf denen son' undefiniert ist. (Dabei sei NIL ∉ |T|)

Ebenso setzt man i.a. father(root):=NIL

Damit gilt

father(son(n,a)) = n

und

l(vf(son(n,a))) = a fuer son(n,a) ≠ NIL

(nach Def.)

Weiter kann man definieren, was es heisst, einen String aus A* im Baum von der Wurzel aus zu verfolgen:

```
follow($)   :=  root
follow(u*a) :=  { son(follow(u),a)   falls follow(u) ≠ NIL
                { NIL                sonst
```

(mit a ∈ A, u ∈ A*)
Durch Induktion ueber |u| zeigt man wieder leicht die Wohldefiniertheit.

follow kann auch allgemein fuer gerichtete zyklenfreie Graphen mit Wurzel definert werden, wenn fuer jeden Knoten die Markierungen der herausfuehrenden Kanten disjunkt sind.

2. Lem: Es gilt:

(a) follow(label(n)) = n fa. n ∈ |T|
(b) label(follow(u)) = u fa. u ∈ A* mit follow(u) ≠ NIL
(c) follow ist injektiv auf dem Urbild von |T|
(d.h. sofern follow(u) ≠ NIL)
(d) follow ist surjektiv
(e) label ist injektiv
(f) |label(n)| ist die Hoehe von n

Bew: (a) Induktion ueber die Hoehe von n
(b) Induktion ueber |u|
(c) folgt aus (b)
(d) folgt aus (a)
(e) folgt aus (a)
(f) Induktion ueber die Hoehe von n

3. Bem: label ist nicht surjektiv
(da |T| endlich, |A*| unendlich)

4. Bem: Ein Baum definiert eine totale Ordnung fuer seine Blaetter, sofern in jedem Knoten die ausgehenden Kanten total geordnet sind:
definiere fuer n1,n2 ∈ |T|, n1,n2 Blatt
j(n1,n2) als die Wurzel des kleinsten Unterbaums von T, der n1 und n2 enthaelt,
und definiere n1 < n2 :<==> die Kante, mit der der Weg von j(n1,n2) nach n1 beginnt, ist frueher als die Kante, mit der der Weg nach n2 beginnt

0.3.3 Ordnungsrelationen

1. Def: Sei < irreflexive Ordnungsrelation auf einer Menge G,
sei M c G endliche Menge, P Praedikat auf G,
dann heisst y ∈ M ein minimales Element in M mit der Eigenschaft P
:<==> P(y) und fa. z ∈ M mit z < y ist nicht P(z)

2. Def: Sei <= reflexive Ordnungsrelation auf einer Menge G,
sei M c G endliche Menge, P Praedikat auf G,
dann heisst y ∈ M ein minimales Element in M mit der Eigenschaft P
:<==> P(y) und fa. z ∈ M mit z <= y ist
(nicht P(z) oder z=y)

3. Bem: Sei < irreflexive Ordnungsrelation auf G, M c G endlich,
P Praedikat auf G, dann gilt
ex. x ∈ M mit P(x) ==> ex. ein minimales Element y ∈ M
mit P(y), und (y<x oder y=x)

4. Satz: Sei <= reflexive Ordnungsrelation auf G, M c G endlich,
P Praedikat auf G, dann gilt
ex. x ∈ M mit P(x) ==> ex. ein minimales Element y ∈ M
mit P(y), und y <= x

```
****************************************
* TEIL 1      Der eindimensionale Fall *
****************************************
```

KAPITEL 1.1 Ausgangspunkt

Ausgangspunkt der Arbeit war die Frage, wie es moeglich ist, mit Hilfe eines Musteridentifizierungs-Algorithmus einen Strom von Eingabedaten (eine Zeichenkette / String) so abzuspeichern, dass

1. nach dem Einlesen die Frage beantwortet werden kann, ob ein bestimmter String als Teilstring vorkam und in welchem Kontext (d.h. welche Zeichen davor und dahinter kamen), und
2. diese Frage schon waehrend des Einlesens staendig fuer die neu eingelesenen Zeichen beantwortet wird (d.h. gleiche Strings werden miteinander assoziiert).

Beispiel:
Der Algorithmus sollte nach dem Einlesen des Strings

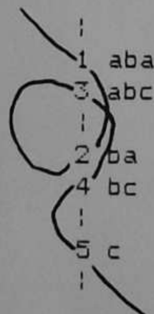
```
abcbbbabcaaa
123456789012
000000000111
```

sagen koennen, dass der Teilstring bbab an der Position 5 vorkommt, und weiter sollte er schon waehrend dem Einlesen von ...abc... bei Position 7 feststellen, dass dieser Teilstring schon an Position 1 vorkam.

Die hier verfolgte Idee zur Implementierung besagt anschaulich, den Eingabstring um eine Messlatte aus den Zeichen des Alfabets zu wickeln, sodass an einer Stelle der Messlatte alle Vorkommen eines bestimmten Teilstrings zu finden sind.

Beispiel:

```
     ababc
     12345
```



Mathematisch heisst das, zu einem gegebenen Eingabestring
w = w[1...n] eine Permutation s von {1,...,n} zu finden, sodass
w[s(i)...n] < w [s(i+1)...n]   fa. 1 <= i <= n-1

Im Beispiel ist:

i	s(i)	w [s(i)...n]
1	1	ababc
2	3	abc
3	2	babc
4	4	bc
5	5	c

In der Tabelle der w[s(i)...n] ist es durch das "binaere Suchverfahren" (binary search) moeglich, nach einem beliebigen Teilstring zu suchen und zu entscheiden, ob er vorkommt, an welchen Stellen er vorkommt und in welchem Kontext (Frage 1).
Ausserdem sind die assoziierten Positionen (an denen also gleiche Strings beginnen) unter benachbarten Werten von i zu finden (im Beispiel: i=1,2).
Wird die Tabelle schon waehrend des Einlesens des Eingabe-Strings schrittweise aufgebaut (jeweils fuer den bisher schon eingelesenen Teil des Eingabestrings), so ist auch Frage 2 geloest. (Natuerlich braucht die Tabelle der w [s(i)...n] nicht so explizit konstruiert werden wie im obigen Beispiel, fuer die binaere Suche genuegt die Angabe von w (als Zeichenfolge) und s (als Zahlenfolge.)

KAPITEL 1.2 Uebersicht ueber die Algorithmen in der Literatur

In der Literatur existieren relativ wenig Arbeiten ueber dieses Gebiet, sodass es von Vorteil ist, sich auch einen ungefaehren Ueberblick ueber Arbeiten zu verwandten Problemen und insbesondere zu Teil-Problemen zu verschaffen. Es lassen sich verschiedene Teilaspekte des oben angegebenen Problems und entsprechende Loesungsalgorithmen unterscheiden:

- Frage 1 fuehrt zu der Fragestellung: Gegeben 2 Strings v und w, kommt v in w als Teilstring vor und wo ? Diese Fragestellung wird in der Literatur i.a. als Pattern-Matching-Problem (fuer Strings) bezeichnet, v heisst auch Pattern.

Die Pattern-Matching-Algorithmen lassen sich noch weiter unterteilen. Es ist moeglich, dass ein Algorithmus eine Vorverarbeitungsphase fuer w hat (mit Zeitaufwand [1] mindestens O(|w|), da w eingelesen werden muss), nach der dann in einer Suchphase v eingelesen und die Matching-Frage beantwortet wird (in kuerzerer Zeit als O(|w|)), und dass man die Suchphase fuer verschiedene v wiederholen kann, ohne die Vorverarbeitungsphase wiederholen zu muessen. Ebenso ist es denkbar, eine Vorverarbeitungsphase fuer v zu haben (Der Begriff einer Vorverarbeitungsphase fuer v UND w ist nicht mehr sinnvoll).

Die Pattern-Matching-Algorithmen lassen sich also danach klassifizieren, ob sie eine Vorbereitungsphase fuer v, fuer w, oder gar keine haben. Die Algorithmen sind entsprechend vor allem fuer Anwendungen mit konstantem v und haeufig wechselndem w [2], bzw. umgekehrt [3], bzw. fuer beide gleich gut (schlecht) geeignet.

- Frage 2 des Problems fuehrt zu der Fragestellung: Gegeben ein String w , welche Teilstrings kommen in w mehrfach vor und an welchen Stellen ? (Repetition-Finding)

Man kann aus jedem Repetition-Finding-Algorithmus (vorausgesetzt, er liefert ALLE Positionen, an denen Wiederholungen vorkommen) einen Pattern-Matching-Algorithmus machen, indem man die Repetitions in v#w sucht (wenn $\# \notin A$). Jeder String, der mehrmals vorkommt und u.a. an Position 1, und der die Laenge |v| hat, ist ein Vorkommen von v in w. Dabei gibt es allerdings keine Aufteilung in Vorbereitungs- und Suchphase, denn fuer veraendertes v oder w muss v#w wieder ganz neu eingelesen werden (ausser, der Repetition-Finding-Algorithmus gestattet inkrementelle Aenderungen, was bei keinem der hier angegebenen der Fall ist [4]).

----

[1] siehe Anhang 2

[2] Z.B. wenn der String w mit Hilfe einer begrenzten Zahl von Ersetzungsregeln (deren linke Seiten die v's sind) sukzessive umgeformt werden soll.

[3] Z.B. wenn in einem grossen, sich selten aendernden Datenbestand bestimmte Datensaetze mit v als Schluessel gesucht werden.

[4] McCreight untersucht diese Moeglichkeit fuer seinen Algorithmus, aber dieser ist sowieso Repetition-Finding- UND Pattern-Matching-Algorithmus vgl. 1.5.9).

Nach den oben genannten Kriterien lassen sich die in der Literatur vorhandenen Algorithmen wie folgt klassifizieren:

Kap	Algorithmus	Aufwand[1]	Aufg	VorV	Onl
1.3.0	Straight Forward Matching	O(|v|*|w|)	P	-	+
1.3.1	Knuth-Morris-Pratt	O(|v|+|w|)	P	-	+
1.3.2	Aho-Corasick	O(|v|+|w|)	P	-	+
1.3.3	Rosenberg	$O(\|w\|*\|A\|^d)$	R	/	+
1.3.4	Karp Miller Rosenberg	O(|w|*log(d))	R	/	-
1.3.5	desgl. fuer Arrays	$O(\|w\|^2)$	R	/	-
1.5.6	Weiner Alg.D	$O(\|w\|^2)$	P+R	w	-
1.5.7	Weiner Alg.B (sfather)	$O(\|w\|^2)$	P+R	w	-
1.5.8	Weiner Alg.C (kompakt)	O(|w|)	P+R	w	-
1.5.9	McCreight	O(|w|)	P+R	w	-
1.6	eigener Alg.	$O(\|w\|^2)$	P+R	w	+
1.8	eigener Alg. (kompakt)	O(|w|)	P+R	w	+

Legende:
Spalte "Aufg" (fuer "Aufgabe"): "P"=Pattern Matching, "R"=Repetition Finding
Spalte "VorV" (fuer "Vorverarbeitung"): "-" = keine Vorverarbeitung,
"w" = Vorverarb. fuer w, "/" = entfaellt fuer Rep.Finding
Spalte "Onl" (fuer "Online"): gibt an, ob der Algorithmus schon waehrend des Einlesens von w ein (vorlaeufiges) Ergebnis liefert ("+") oder ob er erst w ganz eingelesen haben muss ("-")
Spalte "Aufwand": A = Alfabet, d = Laenge der gesuchten Repetitions,
|v| steht bei Alg.2 fuer die Summe der Laenge aller Patterns,
|w| steht bei Alg.5 fuer die Kantenlaenge eines Quadrats

----
[1] siehe Anhang 2

KAPITEL 1.3 Kurzdarstellung der wichtigsten Algorithmen

Die hier gewaehlte Reihenfolge ist keineswegs die, in der die Algorithmen historisch entstanden sind, und die hier dargestellten Zusammenhaenge zwischen einzelnen Algorithmen haben nicht unbedingt dazu gefuehrt, dass der eine aus dem anderen entstanden ist, sondern sie dienen dazu, im Nachhinein die verschiedenen Loesungen zu strukturieren und ansatzweise gemeinsame Methoden herauszuarbeiten.

Die wesentlichste Methode, die sich praktisch bei allen Algorithmen (ausser 1.3.0, 4+5, 6) in irgendeiner Form wiederfinden laesst, ist folgende:
wenn fuer die Bearbeitung einer Position (in w) mehrere Zeichen eingelesen wurden, wird die dadurch erhaltene Information auch fuer die Bearbeitung der naechsten Position(en) genutzt, sodass jedes Zeichen nur einmal eingelesen werden braucht.
In 1.3.1 wird das durch die Tabelle F(j) ermoeglicht, in 2. durch die Failure-Funktion, in 3. durch die Uebergangsfunktion, schliesslich in 7.-11. durch die s-Tree-Zeiger (insbesondere s-father-Zeiger in 9.-11.) .

1.3.0 Straight-Forward-Pattern-Matching-Algorithmus

Er vergleicht v und w Zeichen fuer Zeichen, bis Uebereinstimmung gefunden wird oder w abgearbeitet ist.

```
Beispiel:   w = aaabbbcbabbbccc
            v =      bcba
```

v wird solange laengs w verschoben, bis bei Position 6 vollstaendige Uebereinstimmung gefunden wird.

Dieser Algorithmus, ohne Vorverarbeitung von v oder w und ohne irgendwelche Kniffe, ist der einfachste und wohl auch der weitverbreitetste Algorithmus zur Loesung des Matching-Problems.

Sein Nachteil ist der hohe Zeitaufwand in solchen Faellen, wo sehr oft echte Prefixe von v in w vorkommen, und so an vielen Positionen von w viele Zeichen von v verglichen werden, bevor ein Mis-Match festgestellt wird. Der Zeitaufwand im Fall
$w = a^{2n}b$, $v = a^{n}b$
betraegt $(n+1)^2$ Vergleiche, bevor das einzige Vorkommen von v in w gefunden wird (nach KNU 77).

1.3.1 Der Knuth-Morris-Pratt-Algorithmus (KNU 77, FIS 74-2),

der meistzitierte Pattern-Matching-Algorithmus in der Literatur, vermeidet genau diesen Nachteil, indem er aus v eine Tabelle erstellt, aus der im Mis-Match-Fall hervorgeht, um wieviel Positionen v relativ zu w nach rechts verschoben werden kann, um dort die Suche fortzusetzen.

Beispiel: v = abcaba
Wenn v mit w verglichen wurde und die ersten 5 Zeichen uebereinstimmen, das 6. aber nicht, so weiss man folgendes ueber die Situation:

```
     w = ...abcabx... (x ≠ a)
     v =    abcaba
                 ^
                (Vergleichsposition)
```

Das Ergebnis der ersten 5 Vergleiche liefert genug Information ueber w, um sagen zu koennen, dass v mindestens um 3 Positionen nach rechts verschoben werden muss, bevor ein Match ueberhaupt moeglich ist, und dass dann die ersten beiden Zeichen von v auf jeden Fall matchen:

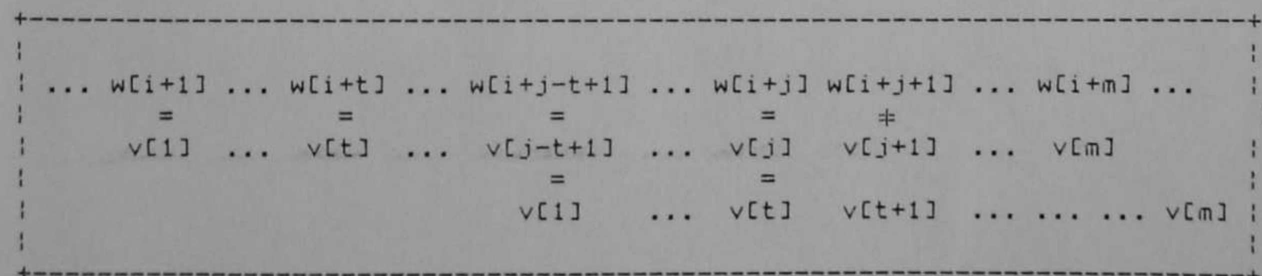
```
     w = ...abcabx...
     v =       abcaba
                 ^
```

Die Tabelle wurde also fuer Position 6 eine 3 enthalten.
In KNU 77 wird ein umfangreiches Beispiel fuer das Matchen mit Hilfe der Tabelle durchgespielt.

Durch die Tabelle erreicht der Knuth-Morris-Pratt-Algorithmus einen Zeitaufwand von O(|w|)+O(Aufwand zur Erstellung der Tabelle), denn in jedem Schritt wird entweder die Vergleichsposition oder das Pattern nach rechts verschoben, beides geht aber hoechstens |w| mal.

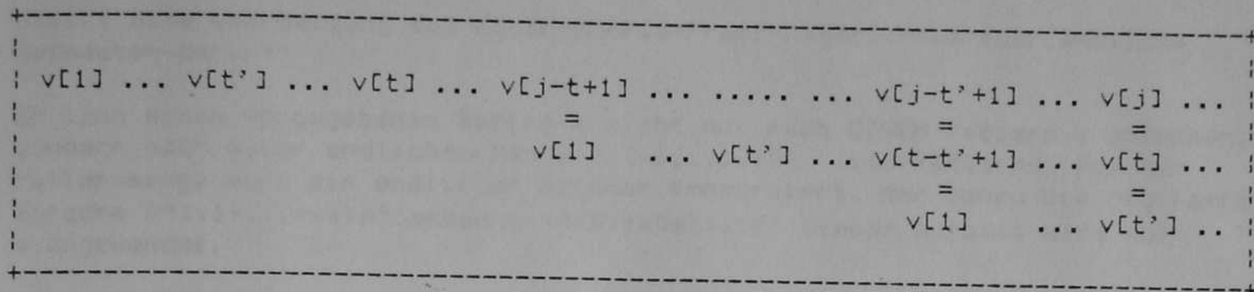
Konstruktion der Tabelle (nach der sehr guten Beschreibung in FIS 74-2):

Ist bei v[j] der erste Mis-Match festgestellt worden, d.h. ist
   v[1...j] = w[i+1...i+j] ,   aber   v[j+1] ≠ w[i+j+1] ,
so muss t abhaengig von j so bestimmt werden, dass t maximal ist mit
   v[1...t] = v[j-t+1...j] ,   (t>=0)
dann kann v um j-t Positionen verschoben werden, sodass an die Stelle von v[j-t+1...j] jetzt v[1...t] kommt:



(Im Bild ist t < j-t+1 dargestellt, es ist aber genauso j-t+1 <= t moeglich)
Im Beispiel oben ist j=5 und t=3.

Sei also
   $P(j) := \max \{t \mid 0 <= t < j$ und $v[1...t] = v[j-t+1...j]\}$
Es ist klar, dass $P(j) < j$,
und $P(P(j))$ liefert das zweitgroesste t mit $v[1...t] = v[j-t+1...j]$:

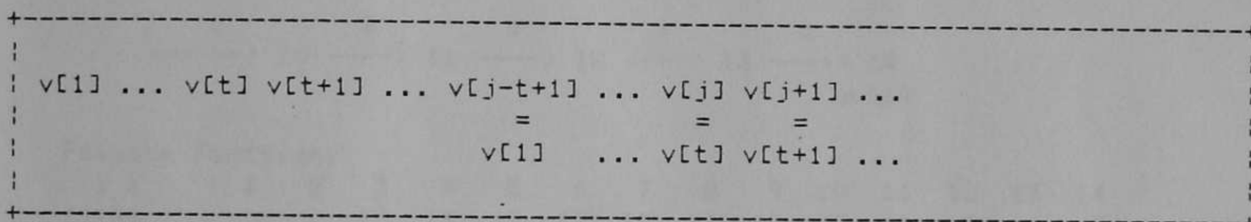


Allgemein liefert $P^{(k)}(j)$ das k.groesste t mit $v[1...t] = v[j-t+1...j]$
(wobei $P^{(k)}$ die k-malige Anwendung von P bedeute).
Damit laesst sich P rekursiv berechnen:

$$P(0) = 0$$

$$P(j+1) = \begin{cases} P^{(k)}(j)+1 & \text{wenn } k>0 \text{ minimal ist mit } v[P^{(k)}(j)+1] = v[j+1] \\ 0 & \text{wenn kein solches } k \text{ existiert} \end{cases}$$

Anschaulich (mit $t = P^{(k)}(j)$):



Der Zeitaufwand fuer die Erstellung der Tabelle betraegt $O(|v|)$, der Gesamtaufwand fuer den Knuth-Morris-Pratt-Algorithmus betraegt also $O(|v|+|w|)$. Eine Vorverarbeitungsphase im Sinne von 1.2 von einer Suchphase zu unterscheiden, ist nicht sinnvoll (obwohl es naheliegend scheint, die Erstellung der Tabelle der $P(j)$ als Vorverarbeitung zu sehen), denn da $|v| <= |w|$, ist der Zeitaufwand fuer die Suchphase nicht geringer (d.h. ob bei veraendertem w die Erstellung der Tabelle aus v -unnoetigerweise- wiederholt wird oder nicht, macht keinen Unterschied in der Zeitaufwandsklasse).

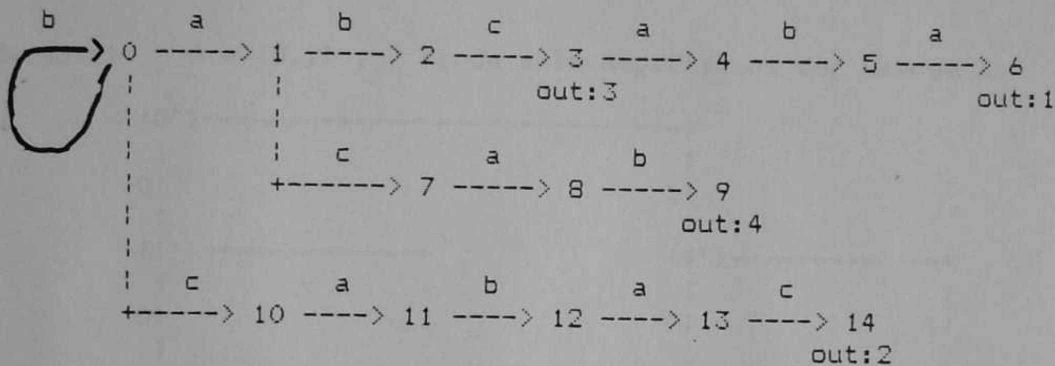
1.3.2 Der Algorithmus von Aho und Corasick (AHO 75)

stellt eine Umarbeitung des Knuth-Morris-Pratt-Algorithmus fuer endliche Automaten dar. [1]

Er kann einen vorgegebenen String w nicht nur nach EINEM Pattern v absuchen, sondern nach einer endlichen Menge {v1,...,vk} von Patterns. Aus der Patternmenge wird ein endlicher Automat konstruiert, der genau die regulaere Sprache A*(v1+...+vk)A* erkennt (A:Alfabet). [2] Dieser Automat wird auf w angewendet.

Beispiel:

Patterns: abcaba , cabac , abc , acab (A={a,b,c})



Failure Function:

i :	1	2	3	4	5	6	7	8	9	10	11	12	13	14
f(i):	0	0	10	11	12	13	10	11	12	0	1	2	1	7

Der Automat startet im Zustand 0 und geht bei Eingabe eines Zeichens in den entsprechenden Folgezustand ueber, sofern eine Kante vom augenblicklichen Zustand ausgeht, die mit dem eingelesenen Zeichen markiert ist. Existiert eine solche Kante nicht (Mis-Match-Fall), so geht er in den Zustand ueber, der sich aus der Tabelle f (Failure Function) ergibt. f(j) entspricht der Funktion P(j) bei Knuth-Morris-Pratt und wird auch nach einem analogen Rekursionsschema berechnet.

Beispiel fuer die Arbeitsweise:

```
Zustand:  0   10   11   12   2 3   4   5   6   13 14 7
Eingabe:    c    a    b    c     a   b   a   c         ...
                             ^ ^           ^     ^  ^ ^
                             f |           |     f  | f      (f=Failure)
                               |           |        |
                            out:3        out:1    out:2      (Pattern erkannt)
```

Bezueglich Vorverarbeitung gilt das beim Knuth-Morris-Pratt-Algorithmus Gesagte entsprechend.

----

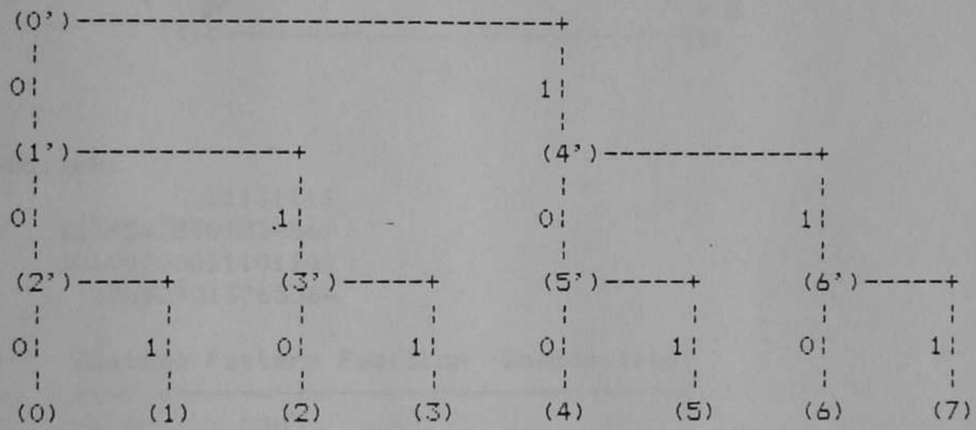
[1] Fuer die Theorie der endlichen Automaten siehe z.B. EHR 72

[2] Fuer die Theorie der regulaeren Sprachen siehe z.B. AHO 74

1.3.3 Der Algorithmus von Rosenberg

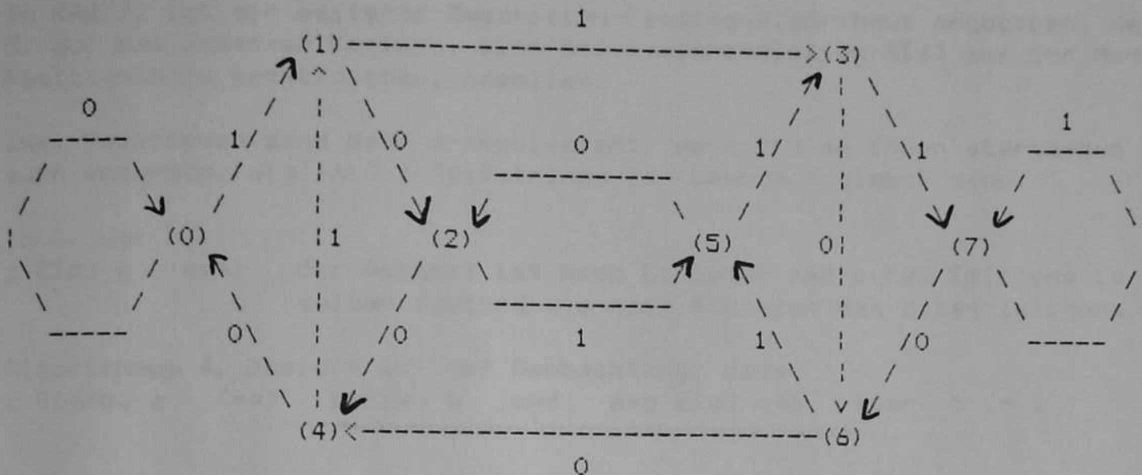
Wenn man den Aho-Corasick-Algorithmus mit allen Strings einer bestimmten Laenge als Patterns betrachtet, erhaelt man den Rosenberg-Algorithmus (KAR 72, ROS 72). Der Algorithmus loest nicht das Pattern-Matching-Problem (das bei dieser Patternmenge natuerlich trivial ist), sondern findet alle mehrfach vorkommenden Strings einer bestimmten Laenge. Dementsprechend wird jetzt bei Erreichen eines Endknotens die aktuelle Position nicht ausgegeben, sondern im Knoten "deponiert". Nachdem der gesamte Eingabestring eingelesen ist, steht dann bei jedem Endknoten die Liste aller Positionen, an denen das (dem Knoten) entsprechende Pattern vorkommt. Durch die exponentiell mit der Patternlaenge wachsende Anzahl von Patterns (und damit von Knoten) ist der Algorithmus auf sehr kurze Laengen (und kleine Alphabete) begrenzt.

Beispiel: A = {0,1} , finde alle Repetitions der Laenge 3



i:	0'	1'	2'	3'	4'	5'	6'	0	1	2	3	4	5	6	7
f(i):	/	/	/	/	/	/	/	2'	3'	5'	6'	2'	3'	5'	6'

Wenn man die Failure-Uebergaenge mit den nachfolgenden jeweils moeglichen Lese-Uebergaenge zusammenfasst, erhaelt man die von Rosenberg angegebene Form:



Beispiel-Ablauf:

```
                 11111111
Position:   12345678901234567
Eingabe:    00110100011101100
Zustand:       136524013765364
```

Ergebnis:

Zustand	Pattern	Position (Endposition)
0	000	9
1	001	3,10
2	010	7
3	011	4,11,15
4	100	8,17
5	101	6,14
6	110	5,13,16
7	111	12

Man kann auch auf die Darstellung der Uebergangsfunktion als Graph verzichten und den Folgezustand jeweils neu berechnen:

$$neu := (alt * 2 + c) \pmod{2^3}$$

(mit alt, neu: alter, neuer Zustand, c: Eingabezeichen), in jedem Fall benoetigt man jedoch Speicherplatz in der Groessenordnung 2^3 (allgemein: $|A|^d$) zum "Deponieren" der Positionen. Abhaengig von der Laenge des Eingabestrings w waechst der Aufwand allerdings nur linear.

1.3.4 Der Algorithmus von Karp, Miller und Rosenberg

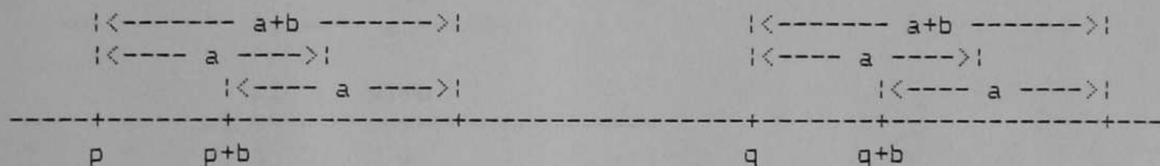
In KAR 72 ist ein weiterer Repetition-Finding-Algorihmus angegeben, der wie 3. auf dem Gedanken basiert, eine Aequivalenzrelation E[d] auf der Menge der Positionen zu konstruieren, naemlich:

Zwei Positionen sind dann d-aequivalent, wenn die an ihnen startenden (oder auch endenden, wie in 3.) Teilstrings der Laenge d gleich sind.

In 3. war
p E[d] q <==> der Automat ist nach Einlesen des p.ten Zeichens im selben Zustand wie nach Einlesen des q.ten Zeichens.

Algorithmus 4. basiert auf der Beobachtung, dass
p E[a+b] q <==> p E[a] q und p+b E[a] q+b fuer b <= a

Anschaulich:



Da die Relation E[1] trivial ist (p E[1] q <==> w [p] = w [q]), kann E[1] und daraus nacheinander E[2] = E[1+1], E[4], E[8], ... usw. berechnet werden. Werden nur Repetitions einer bestimmten festen Laenge d gesucht, so kann E[d] berechnet werden (z.B. E[7] = E[4+3], E[3] = E[2+1]), werden Repetitions maximaler Laenge gesucht, so kann E[d] fuer d maximal mit E[d] ≠ Id bestimmt werden (Id bezeichne die Diagonale von {1,...,¦w¦}x{1,...,¦w¦}). In jedem Fall besteht eine Aequivalenzklasse von E[d] aus allen Positionen, an denen der gleiche String startet.

Da der Zeitaufwand fuer die Berechnen von E[a+b] aus E[a] (fuer b<=a) O(¦w¦) betraegt, braucht der Algorithmus einen Gesamtaufwand von O(¦w¦*log(d)).

Beispiel:

```
               111111
      123456789012345
      abbaabbbabaabba

E[1]: 122112221211221
E[2]: 12341223134123
E[4]: 123456789341
E[8]: 12345678
```

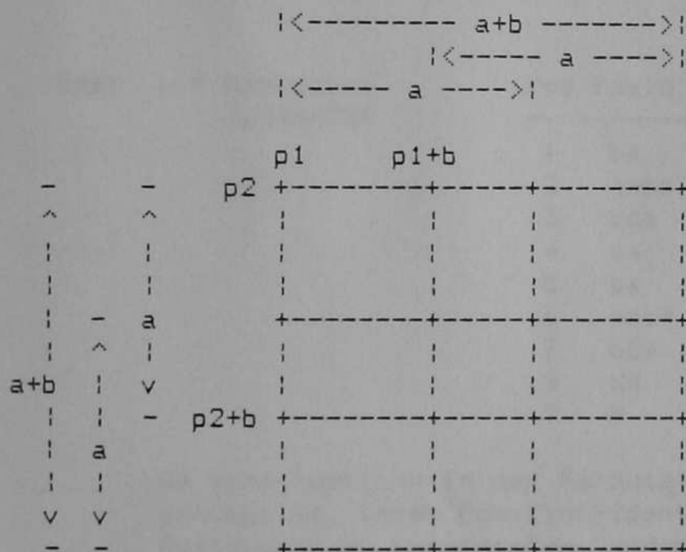
1.3.5 Dieselbe Methode funktioniert auch fuer Arrays:

Sei (p1,p2) E[d] (q1,q2),
wenn die beiden d*d-Quadrate mit den linken oberen Ecken bei Position (p1,p2) und bei Position (q1,q2) gleich sind.

Es gilt fuer b <= a :

```
        (p1,p2) E[a+b] (q1,q2)
    <==>
         (p1,p2) E[a] (q1,q2)       und
       (p1,p2+b) E[a] (q1,q2+b)     und
       (p1+b,p2) E[a] (q1+b,q2)     und
     (p1+b,p2+b) E[a] (q1+b,q2+b)
```

Anschaulich:



Da E[1] wieder trivial ist, lassen sich E[2], E[4], E[8], usw. genauso nacheinander konstruieren wie im String-Fall. Der Aufwand betraegt $O(n^2*\log(d))$ fuer E[d] und ein n*n-Eingabefeld.

Bevor die Prefix-Tree-Algorithmen hier behandelt werden, sollen die wichtigsten theoretischen Begriffe und Zusammenhaenge im folgenden Kapitel erlaeutert werden.

KAPITEL 1.4 Theorie der Position-Identifier

Im Folgenden sei stets $v, w \in A^*$, $w=w[1...n]$, $v=v[1...m]$

1. Def: v heisst Position-Identifier (kurz: Posid) fuer eine Position i in w

   :<==> v kommt nur an der Position i in w vor,
   (d.h. $v=w[i...j]$ und $v \neq w[k...l]$ fa. $k \neq i$)
   und jeder echte Prefix von v kommt mehrmals in w vor

   Schreibweise: $v = I^w(i)$,
   wenn w feststeht, schreibt man statt $I^{w[1...k]}(i)$ auch $I^k(i)$
   und statt $I^w(i)$ auch $I(i)$

   Der Begriff des Position-Identifiers geht vermutlich auf Weiner zurueck (WEI 73-1).

   Bsp: w = babcaabc#
   123456789

Pos	Posid	B	M	N
1	ba	1	*	b
2	abca	2		abc
3	bca	2		
4	ca	2	*	
5	aa	5	*	a
6	abc#	6		abc
7	bc#	6		
8	c#	6		
9	#	6	*	

   (Spalten B, M, N siehe unten, Def 5., 8., 15.)

   Um eine Position in der Permutation aus Kapitel 1.1 einzuordnen, genuegt es, ihren Position-Identifier mit denen der anderen Positionen zu vergleichen, andererseits reicht es nicht, einen kuerzeren String zu vergleichen, da dieser (nach Def.) auf jeden Fall in w mehrfach vorkommt und somit keine Position eindeutig identifizieren kann.

   Die Permutation im obigen Beispiel: (sei # < a < b < c)

i	s(i)	Posid
1	9	#
2	5	aa
3	6	abc#
4	2	abca
5	1	ba
6	3	bca
7	7	bc#
8	8	c#
9	4	ca

2. Bem: (a) kein Posid ist echter Prefix eines anderen
(b) jeder String v, der nur einmal in w vorkommt, hat einen Posid als Prefix
(c) zu einer Position gibt es hoechstens einen Posid
(d) kommt das letzte Zeichen von w nur einmal in w vor, so hat jede Position von w genau einen Posid
(e) der Posid von i ist der kuerzeste String, der an der Position i startet und nur einmal in w vorkommt
(f) jeder String w enthaelt einen Posid fuer die Position 1

Bew: (a) waere $I^w(i) * u = I^w(j)$ fuer $u \neq \$$,
so wuerde $I^w(i)$ mehrmals in w vorkommen, naemlich bei i und bei j ($j \neq i$, da sonst $I^w(i) = I^w(j)$)
(b) kommt v nur einmal in w vor, etwa an der Position i,
so ist $I^w(i)$ Prefix von v oder v echter Prefix von $I^w(i)$,
letzteres ist aber nach Def von $I^w(i)$ ausgeschlossen
(c) folgt aus (a)
(d) folgt aus (b),
denn w[i...n] kommt dann nur einmal in w vor fa. i
(e) nach Def
(f) folgt aus (b), da w nur einmal in w vorkommt

3. Lem: $|I(i-1)| \leq |I(i)| + 1$
oder anders formuliert: $|I(i-1)| - 1 \leq |I(i)|$

Bew: sei I(i-1) = w[i-1...j],
dann ist w[i-1...j-1] mehrfach in w
==> w[i...j-1] mehrfach in w
==> w[i...j-1] echter Prefix von I(i)
==> $j-i = j-1-i+1 = |w[i...j-1]| < |I(i)|$
==> $j-i+1 \leq |I(i)|$
==> $|I(i-1)| = |w[i-1...j]| = j-i+2 \leq |I(i)|+1$

Bem: Man kann also alle Positionen i von w danach klassifizieren, ob $|I(i)| > |I(i-1)|-1$ oder $|I(i)| = |I(i-1)|-1$ gilt.
Eine genauere Untersuchung fuehrt zu interessanten Ergebnissen (Satz 10ff).

4. Lem: $|I(i)|-j \leq |I(i+j)|$
oder anders formuliert: $|I(i-j)| \leq |I(i)|+j$ fuer $j \leq i$
oder noch anders: $|I(j)|-(i-j) \leq |I(i)|$ fuer $j \leq i$

Bew: folgt aus 3. durch Induktion ueber j

5. Def: sei $B := B[w] := \{1\} \cup \{i \mid 2 \leq i \leq n,\ |I(i-1)| < |I(i)|+1\}$
B heisst Menge der Block-Beginn-Positionen (von w)

fuer $i \in B[w]$ heisst
$b(i) := b^w(i) := \{j \mid j \geq i,\ |I(j)| = |I(i)|-(j-i)\}$
Block von i
(fuer $i \notin B[w]$ ist $b^w(i)$ nicht definiert)

Bsp: Im Beispiel zu Def 1. ist der Block $b(i)$ durch ein i in der Spalte B gekennzeichnet, $B = \{1, 2, 5, 6\}$,
$b(1) = \{1\}$, $b(2) = \{2, 3, 4\}$, $b(5) = \{5\}$, $b(6) = \{6, 7, 8, 9\}$

6. Lem:
(a) $b(i) \cap B = \{i\}$ fa. $i \in B$
(b) $|b(i)| <= |I(i)|$ fa. $i \in B$
(c) zu jedem $j \in \{1,...,n\}$ ex. $i \in B$ mit $j \in b(i)$
(d) fuer $i \neq j$, $i,j \in B$ ist $b(i) \cap b(j) = \{\}$
(e) $j \in b(i)$, $i <= k <= j$ ==> $k \in b(i)$
(f) $j \in b(i)$ ==> $I(j)$ ist Suffix von $I(i)$

Bew:
(a) $i \in b(i)$, $i \in B$ nach Def.
waere $k \in B$, $k \in b(i)$, $k \neq i$,
dann: $k \in B$ ==> $k=1$ (geht nicht wegen $k \in b(i)$ ==> $k > i >= 1$)
oder
$|I(k-1)| < |I(k)|+1$
(und da $k \in b(i)$:) $= |I(i)|-(k-i)+1$
nach 4. gilt andererseits:
$|I(k-1)| >= |I(i)|-(k-i)+1$
W.!

(b) fuer $j-i >= |I(i)|$ wuerde die Bedingung fuer $j \in b(i)$ lauten:
$|I(j)| = |I(i)|-(j-i) <= |I(i)|-|I(i)| = 0$
d.h. $b(i) \subset \{i, i+1, ... , i+|I(i)|-1\}$

(c) fuer $j=1$ ist $j \in b(1)$, da $1 \in B$
fuer $j>1$ mit $|I(j)| > |I(j-1)|-1$
ist $j \in b(j)$, da $j \in B$
fuer $j>1$ mit $|I(j)| = |I(j-1)|-1$
sei $i < j$ maximal mit $|I(i)| > |I(i-1)|-1$,
falls kein solches i ex., sei $i=1$
dann ist fa. k mit $i < k <= j$
$|I(k)| = |I(k-1)|-1$ (da i maximal),
also $|I(j)| = |I(j-1)|-1 = ... = |I(i)|-(k-i)$
d.h. $j \in b(i)$ bzw. $j \in b(1)$

(d) waere $k \in b(i)$, $k \in b(j)$,
so waere $|I(i)|-(k-i) = |I(k)| = |I(j)|-(k-j)$,
also $|I(i)| = |I(j)|-(i-j)$,
d.h. $j \in b(i)$
d.h. $j = i$ nach (a)

(e) $|I(i)|-(k-i) = |I(i)|-(j-i)+(j-k) = |I(j)|+(j-k)$
$>= |I(k)| >= |I(i)|-(k-i)$

(f) sei $I(i) = w[i...k]$,
dann ist wegen $|I(j)| = |I(i)|-(j-i) = k-i+1-(j-i) = k-j+1$
$I(j) = w[j...k]$

7. Def: $E := E[w] := \{i \mid i+1 \in B[w]\} \cup \{|w|\}$
heisst Menge der Block-End-Positionen

8. Def: $M := M[w] := \{I^w(i) \mid i \in E[w]\}$
heisst Menge der Minimal-Identifier (kurz: Minids) von w

Bsp: Im Beispiel zu Def 1. ist $E = \{1, 4, 5, 9\}$
und $M = \{I(1), I(4), I(5), I(9)\} = \{ba, ca, aa, \#\}$
(in der Spalte M gekennzeichnet).

9. Lem: $i \in E \iff i = \max b(j)$ fuer ein $j \in B$

Bew: "==>": nach 6.(c),(d) ex. zu i ein $j \in B$ mit $i \in b(j)$,
da $i \in E$, ist $i+1 \in B$ (fuer $i=|w|$ ist die Beh klar)
also $|I(i+1)| > |I(i)|-1 = |I(j)|-(i-j)-1$
$= |I(j)|-(i+1-j)$
also $i+1 \notin b(j)$
nach 6.(e) folgt $i = \max b(j)$

"<==": ist $b(j)$ der Block, der $|w|$ enthaelt, so ist
$i = \max b(j) = |w|$, also $i \in E$
ist $b(j)$ ein anderer Block, so gilt:

$$|I(i)| = |I(j)|-(i-j) \quad \text{da } i \in b(j)$$
$$|I(i+1)| > |I(j)|-(i+1-j) \quad \text{da } i+1 \notin b(j)$$
$$= |I(j)|-(i-j)-1$$
$$= |I(i)|-1$$

also $i+1 \in B$, d.h. $i \in E$

10. Satz: zu jedem Posid $I^w(i)$ von i in w
ex. ein $I^w(m) \in M[w]$ sodass $I^w(m)$ Suffix von $I^w(i)$
und m und i liegen im gleichen Block

Bew: nach 6.(c) ex. ein $k \in B$ mit $i \in b(k)$
setze $m := \max b(k)$,
nach 9. folgt $m \in E$, also $I(m) \in M$
nach 6.(f) ist $I(i)$ Suffix von $I(k)$ und $I(m)$ Suffix von $I(k)$,
also $I(m)$ Suffix von $I(i)$, da $m >= i$

11. Satz: Ein Substring v von w kommt genau dann mehrmals in w vor,
wenn er keinen der Minimal-Identifier als Substring enthaelt

Bew: "==>": klar, da sonst einer der Minimal-Identifier auch mehrmals
vorkommen wuerde
"<==": kaeme v nur einmal in w vor, etwa an der Position i,
dann waere nach 2.(b) $I(i)$ Prefix von v,
also waere nach 10. ein Minimal-Identifier in v

12. Satz: kein Minimal-Identifier ist echter Substring eines anderen

Bew: Ann: I(m) = u*I(m')*u' mit u*u' ≠ $, oBdA. m< m',
waere u' ≠ $, so waere ein echter Prefix des Posids I(m) nur
einmal in w (naemlich u*I(m')), das geht nicht,
also u' = $, d.h. I(m) = u*I(m')
also |I(m')| = |I(m)|-|u| = |I(m)|-(m'-m)
sei nun m ∈ b(k) (k ex. nach 6.(c)), so gilt
|I(m')| = |I(m)|-(m'-m) = |I(k)|-(m-k)-(m'-m) = |I(k)|-(m'-k)
wegen k <= m < m' folgt m' ∈ b(k),
W.! zu m = max b(k) nach 9.

13. Korr: (a) kein Posid ist echter Substring eines Minimal-Identifiers
(b) jeder echte Substring eines Minimal-Identifers kommt mehrfach vor

Bew: (a) aus 12. und 10.
(b) aus 12. und 11.

14. Satz: Die Menge M der Minimal-Identifier ist durch 11. eindeutig
charakterisiert in folgendem Sinn:
ist M'={v1,...,vk} eine Menge von Substrings von w, sodass
1. kein vi Substring eines anderen vj ist (i≠j) und
2. M' die Behauptung von Satz 11. erfuellt, d.h.
fa. u Substring von w gilt
(u mehrmals in w <==> u enthaelt keines der vi)
dann ist M' = M

Bew: M' c M: Ann: ex. vi ∈ M', vi ∉ M, oBdA. sei vi = v1
d.h. fuer kein i ∈ E ist v1 = I(i)
Fall 1: v1 ≠ I(j) fa. 1 <= j <= n
Fall 2: v1 = I(j) fuer ein j ∉ E
Fall 1: sei v1 = w[i1...i2], sei I(i1) = w[i1...i3]
ist i2 < i3, so kommt v1 mehrmals vor,
mit u:=v1 ergibt sich ein W. zur Vor.!
ist i3 < i2, so folgt mit u := I(i1) n.Vor.,
da u einmal in w:
u enthaelt eines der vi,
d.h. v1 = u*w[i3+1...i2] enthaelt eines der
vi echt, W.!
Fall2: nach 10. ex. ein Minimal-Identifier I(l)
mit v1 = I(j) = w[j...l-1]*I(l)
da v1 kein Minimal-Identifier ist,
ist w[j...l-1] ≠ $,
d.h. u := I(l) enthaelt nicht v1,
n.Vor. folgt (da u einmal in w):
u enthaelt eines der v2,...,vk
d.h. v1 enthaelt eines der v2,...,vk , W.!

M c M': sei I(m) ∈ M, n.Vor. folgt I(m) enthaelt (oBdA.) v1
waere I(m) ≠ v1, d.h. v1 echter Substring von I(m),
so waere v1 mehrmals in w nach 13.(b),
W.! zur Vor. mit u:=v1,
also I(m) = v1

15. Def: N := N[w] := {w[i...k-1] | w[i...k] = $I^w(i)$ fuer i ∈ B[w]}
heisst die Menge der maximalen Mehrfachstrings (vgl. 16.)

Bsp: Im Beispiel zu Def 1. ist N = {b, abc, a, abc}
(in der Spalte N aufgelistet).

16. Satz: fa. Substrings v von w gilt:
v kommt in w mehrfach vor
<==> ex. u ∈ N[w] mit v Substring von u

Bew: nach 11. genuegt es zu zeigen, dass N alle laengstmoeglichen Substrings von w enthaelt, die keinen Minimal-Identifier enthalten

(a) u ∈ N enthaelt keinen Minid,
da u (nach Def des Posids) mehrmals in w vorkommt (mit 11.)

(b) kein Element u ∈ N kann noch vergroessert werden,
denn w[i-1...k-1] enthaelt den Posid von i-1 als Prefix:
da I(i-1) = w[i-1...l] mit
mit l-(i-1)+1 = |I(i-1)| < |I(i)|+1 = k-i+1+1 = k-(i-1)+1
also l < k
und w[i...k] ist Posid von i;
sowohl w[i-1...k-1] als auch w[i...k] kommen also nur einmal in w vor

(c) gaebe es noch einen Substring v von w, der mehrfach vorkommt aber mit keinem u ∈ N ein Zeichen gemeinsam hat (und damit nach (b) Substring von u ist),
dann sei v = w[j1...j2], j1 ∈ b(j3), I(j3) = w[j3...j4],
sei u := w[j3...j4-1], u ∈ N
nach 6.(b) folgt j3 <= j1 <= j4
ist j1 < j4, so hat v und u das Zeichen w[j1] gemeinsam, W.!
ist j1 = j4, so ist j1 = max b(j3) (da j1 ∈ b(j3))
==> j1 ∈ E (nach 8.)
==> I(j1) = w[j1] (da I(j1) Suffix von I(j3))
==> v umfasst einen Minimal-Identifier, W.!

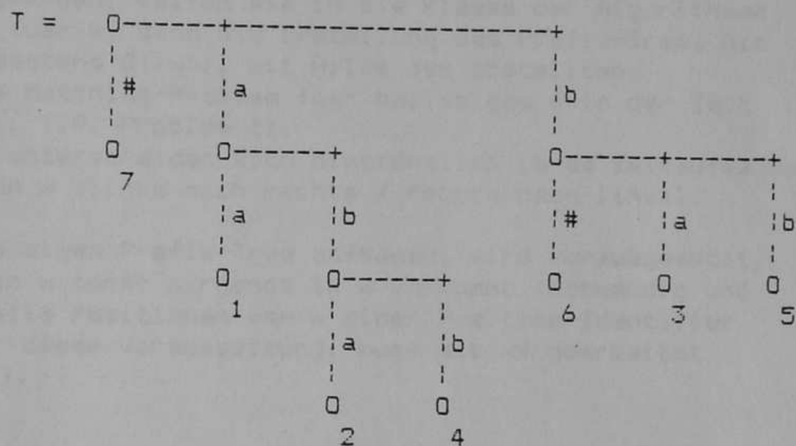
Als Datenstruktur fuer die naechsten Algorithmen soll jetzt noch der Prefix-Tree definiert werden.

17. Def: Sei $w \in A^*$, $\# \notin A$, T heisst Prefix-Tree zu w#
:<==> T ist Tree vom Grad hoechstens $|A|+1$,
label liefert eine 1-1-Zuordnung von der Menge aller Blaetter zur Menge aller Posids von w#,
die Blaetter von T sind mit den Positionen markiert, fuer die die Posids stehen (d.h. Blatt n ist mit Position i markiert, wenn label(n)=I(i))
fuer $i<j$ ist die Markierung der i.ten Kante eines Knotens lexikografisch vor der j.ten Kante des Knotens

Bsp: w = aababb#
1234567

Pos	Posid
1	aa
2	aba
3	ba
4	abb
5	bb
6	b#
7	#

T =



Der Prefix-Tree zum Beispiel zu Def 1. ist im Kapitel 1.5.6 (Weiners Alg.D) abgebildet.

KAPITEL 1.5 Die Prefix-Tree-Algorithmen

Waehrend die bisher skizzierten Algorithmen nur einzelne Teilprobleme loesten, sind die Algorithmen, die mit Hilfe eines Prefix-Trees arbeiten [1], sowohl Matching- als auch Repetition-Finding-Algorithmen. Die in dieser Arbeit vorgestellten eigenen Algorithmen basieren wesentlich auf den Ideen von Weiner. Der Prefix-Tree, der von dem Weiner-Algorithmus aus einem Eingabe-String w aufgebaut wird, repraesentiert genau die Permutation (Ordnungsrangfolge) s aus Kapitel 1.1. Wie dort skizziert, ist mit deren Hilfe sowohl das Matching- als auch das Repetition-Finding-Problem loesbar.

Als Matching-Algorithmen gesehen, fallen sie in die Klasse der Algorithmen mit Vorverarbeitungsphase fuer w, denn die Erstellung des Prefix-Trees hat einen Zeitaufwand von mindestens $O(|w|)$, mit Hilfe des erstellten Prefix-Trees kann dann das Matching-Problem fuer beliebiges v in der Zeit $O(|v|)$ geloest werden (vgl. 1.9, Problem 1).
Die einzelnen Algorithmen unterscheiden sich hinsichtlich ihres Zeitaufwandes und der Lesereihenfolge von w (links nach rechts / rechts nach links).

Fuer alle Algorithmen, die einen Prefix-Tree aufbauen, wird vorausgesetzt, dass das letzte Zeichen von w sonst nirgends in w vorkommt (notwendig und hinreichend dafuer, dass alle Positionen von w einen Position-Identifier besitzen; erfuellt w nicht diese Voraussetzung, muss mit w# gearbeitet werden, wobei $\# \notin A$ sei).

----

[1] das sind die Algorithmen von Weiner (WEI 73-1, AHO 74), von McCreight (MCC 76) sowie die hier vorgestellten eigenen Algorithmen

1.5.6 Weiners Alg.D

Weiner gibt zunaechst einen einfachen Algorithmus ("Alg.D") zur Konstruktion des Prefix-Trees von w [i...n] aus w [i+1...n] an. Damit koennen zu gegebenem w = w [1...n] nacheinander die Prefix-Trees von w[n], w[n-1...n], w[n-2...n], ..., w[1...n] konstruiert werden.

Der Algorithmus verfolgt Zeichen fuer Zeichen von root aus den Weg w[i] w[i+1] ...
soweit er im Prefix-Tree fuer w[i+1...n] schon vorhanden ist (d.h. bis zum laengsten Prefix von w[i...n], der in w[i+1...n] nochmals vorkommt.

Fall 1:
- ist der so erreichte Knoten n1 ein Blatt (d.h. ist u1 = label(n1) gleich einem Position-Identifier I^{i+1}(j0) einer Position j0 in w[i+1...n]), so ist I^{i+1}(j0) kein Posid in w[i...n] mehr.
In diesem Fall muss I^{i+1}(j0) = w[j0...j1] soweit nach rechts verlaengert werden, bis es nicht mehr auch als Prefix von w[i...n] vorkommt, um I^{i}(j0) zu erhalten, und u1 = w[i...i1] entsprechend.
D.h. ausgehend von n1 wird ein neuer Weg gebildet durch Anhaengen von Kanten, die mit w[j1+j] (j=1,2,...) markiert sind, solange w[j1+j] = w[i1+j].
Ist w[j1+j] $\neq$ w[i1+j] , so ist I^{i}(i) = w[i...i1+j] und I^{i}(j0) = w[j0...j1+j] , unter der Baum wird durch Anhaengen der entsprechenden beiden Knoten zum Prefix-Tree fuer w[i...n].

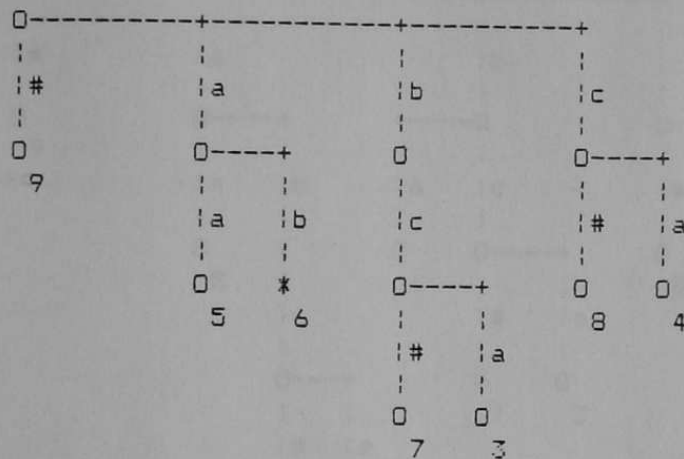
Fall 2:
- ist der erreichte Knoten n1 kein Blatt, so kommt zwar u1 = w[i...i1] in w[i+1...n] vor (also in w[i...n] doppelt), nicht aber u1*w[i1+1] ,
d.h. I^{i}(i) = u1*w[i1+1], und es genuegt, an n1 ueber eine Kante mit w[i1+1] den Knoten fuer I^{i}(i) anzuhaengen, um den Prefix-Tree fuer w[i...n] zu erhalten.

```
                                                 Posid in
                                                 w[i+1...n]
                                               |<---------->|

... w[i] w[i+1] ... w[i1] ... w[i1+j] ... w[j0] ... w[j1] ... w[j1+j] ...

   |<---------------------------->|        |<---------------------->|
          Posid in w[i...n]                    Posid in w[i...n]
```

Beispiel:

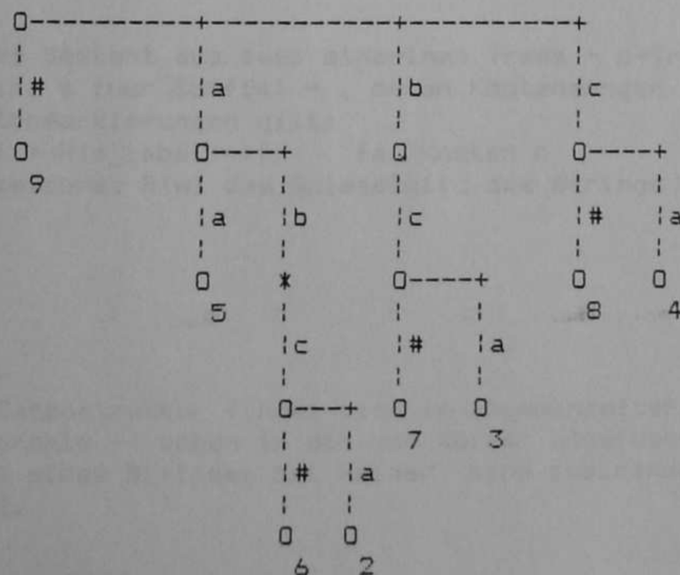
w = babcaabc# (Entspricht dem Beispiel zu Def 1.4.1)
123456789

Prefix-Tree fuer w[3...9] = bcaabc# :
3456789



Konstruktion des Prefix-Trees fuer w[2...9] = abcaabc#
23456789

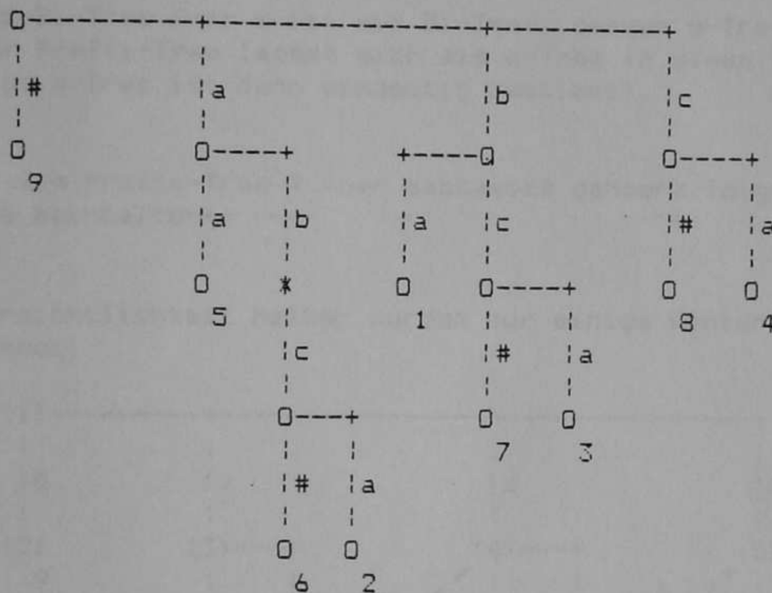
Bestimme u1 = ab , n1 = * ist Blatt (Fall 1), w[2...3] = ab = w[6...7]
w[2...4] = abc = w[6...8] , haenge Kante mit c an,
aber w[2...5] = abca ≠ abc# = w[6...9] ,
haenge zwei Kanten an (mit a und mit #) ,
ergibt den Prefix-Tree fuer w[2...9] :



Konstruktion des Prefix-Trees fuer w[1...9] = babcaabc#
123456789

Bestimme u1 = b , n1 = Q ist kein Blatt (Fall 2),
w[1] = b = w[3] (= w[7]) ,
aber w[1...2] = ba ≠ bc = w[3...4]
(≠ bc = w[7...8]) ,
also ist ba Posid von 1, bca und bc# bleiben Posids von 3 und 7

Prefix-Tree fuer w[1...9] :



Der Nachteil dieses Algorithmus' ist, dass bei jeder Konstruktion des naechsten Prefix-Trees immer wieder von der Wurzel aus der Weg von u1 verfolgt werden muss, wodurch sich ein Zeitaufwand von $O(|w|^2)$ ergibt. Dies fuehrt zu Weiners Idee vom Bi-Tree [1].

Ein Bi-Tree besteht aus zwei einzelnen Trees - p-Tree und s-Tree genannt (p fuer Prefix, s fuer Suffix) - , deren Knotenmengen identisch sind und fuer deren Kantenmarkierungen gilt:
p_label(n) = R(s_label(n)) fa. Knoten n
(Dabei bezeichnet R(w) das Spiegelbild des Strings w).

----

[1] Diese Datenstruktur findet sich in abgewandelter Form - als Konzept der Anti-Isomorphie - schon in dem von Weiner angefuehrten Papier KAR 72. p-Tree und s-Tree eines Bi-Trees bei Weiner sind zueinander anti-isomorph im Sinne von KAR 72.

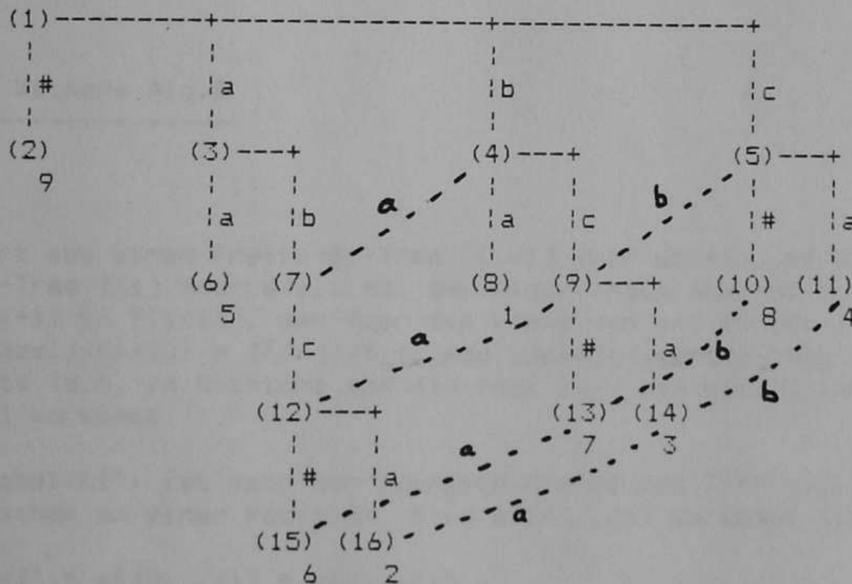
Die Namen Prefix- und Suffix-Tree kommen daher, dass
p_label(p_son(n,a)) = p_label(n)*a fa. Knoten n und Kantenmarkierungen a
(d.h. p_label(n) ist Prefix von p_label(p_son(n,a)))
und
s_label(s_son(n,a)) = a*s_label(n) fa. n und a
(d.h. s_label(n) ist Suffix von s_label(s_son(n,a)))

Insbesondere diese letzte Eigenschaft ist fuer die Konstruktion des Prefix-(Bi-)Trees von Nutzen, da man dadurch leicht von den Prefixen von w[i-1...n] zu Prefixen von w[i...n] uebergehen kann (entspricht dem Uebergang von n zu s_son(n,w[i]) im Bi-Tree).

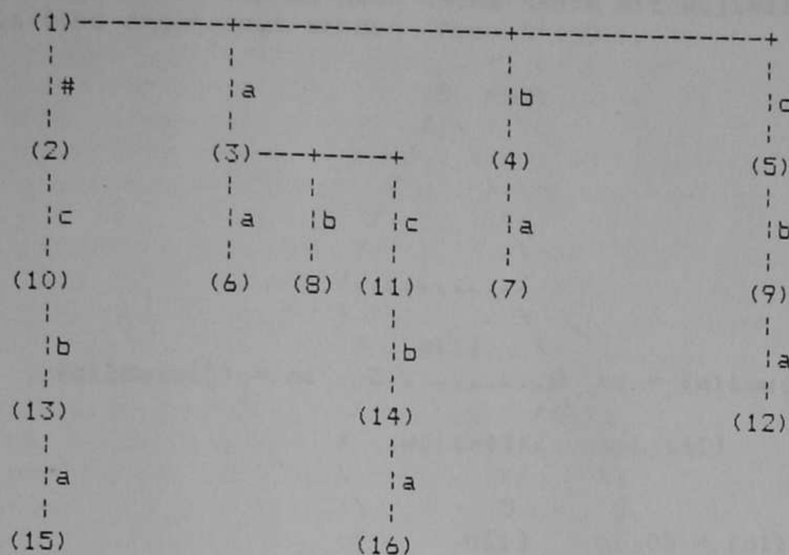
Ein Prefix-Bi-Tree fuer w ist ein Bi-Tree, dessen p-Tree Prefix-Tree fuer w ist (jeder Prefix-Tree laesst sich als p-Tree in einen Bi-Tree einbetten, der zugehoerige s-Tree ist dann eindeutig bestimmt).

Beispiel: Zum Prefix-Tree T fuer babcaabc# gehoert folgender Bi-Tree (der T als p-Tree beinhaltet):

p-Tree:
(der Uebersichtlichkeit halber wurden nur einige Kanten des s-Trees eingezeichnet)



s-Tree:
(der Uebersichtlichkeit halber getrennt vom p-Tree gezeichnet)



1.5.7 Weiners Alg.B

konstruiert aus einem Prefix-Bi-Tree T[i+1] fuer w[i+1...n] einen Prefix-Bi-Tree T[i] fuer w[i...n]. Der Algorithmus startet mit n1' bei dem Knoten n[i+1] in T[i+1] , der fuer den Posid von i+1 in w[i+1...n] steht (d.h. p_label(n[i+1]) = I^{i+1}(i+1)), und verfolgt den (p-)Weg solange rueckwaerts (d.h. in Richtung auf die root zu), bis w[i]*p_label(n1') in w[i+1...n] vorkommt [1].

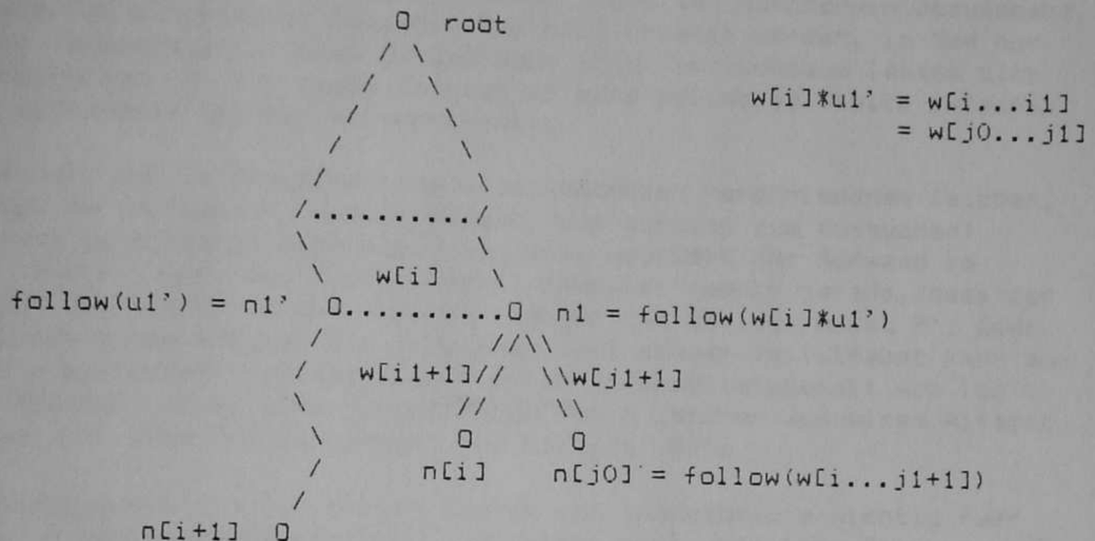
u1' = p_label(n1') ist dann der laengste Prefix von I^{i+1}(i+1), sodass w[i]*u1' schon an einer Position j0 in w[i+1...n] vorkommt (in w[i...n] also mehrmals).
Sei w[i]*u1' = w[j0...j1] = w[i...i1] ,
dann ist w[j0...j1+1] = w[i]*u1'*w[j1+1] neuer Posid fuer j0 in w[i...n]
und w[i...i1+1] = w[i]*u1'*w[i1+1] ist Posid fuer i in w[i...n].

Ausgehend vom Knoten n1' kann jetzt durch den s-Tree direkt uebergegangen werden zu n1 := p_follow(w[i]*u1') , denn n1 = s_son(n1',w[i]).

----

[1] Um das feststellen zu koennen hat jeder Knoten n einen Vektor Ln, dessen j.te Komponente Ln(j) angibt, an welcher Position w[j]*p_label(n) in w[i+1...n] vorkommt (falls mehrmals, wird die rechteste Position angegeben, falls gar nicht, NIL). Die Suche geht also solange, bis Ln1'(w[i]) ≠ NIL .

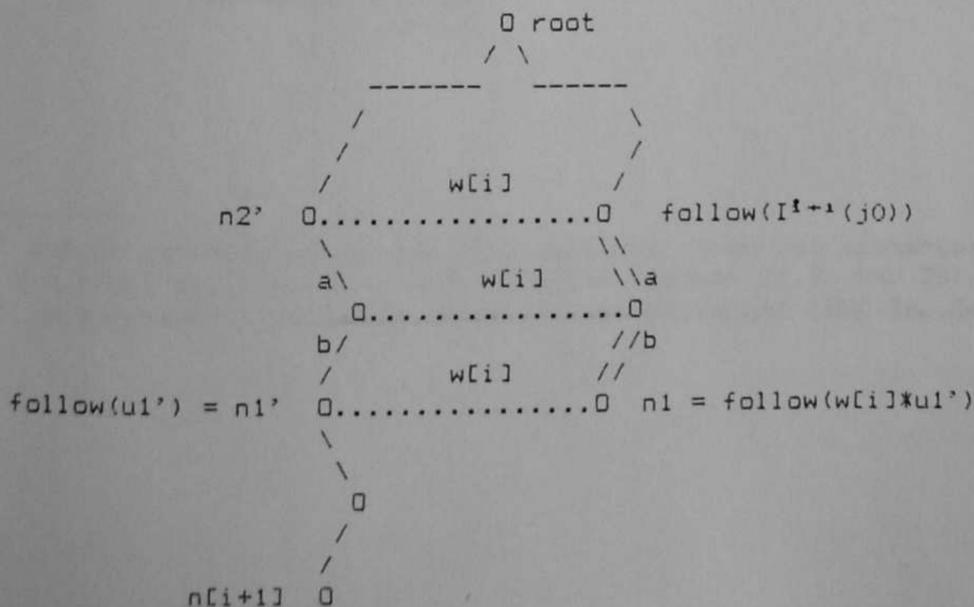
Fall 1:
- Ist n1 in T[i+1] schon vorhanden (d.h. ≠ NIL), so braucht dort nur noch eine Kante mit w[i1+1] angehaengt werden, die zu n[i] fuehrt, dem Knoten fuer $I^{i}(i)$ in T[i]. Falls von n1 noch keine Kante mit w[j1+1] ausgeht, muss ebenfalls eine angehaengt werden (fuer $I^{i}(j0)$).



Legende: einfache Striche ----- in T[i+1] vorhanden
doppelte Striche ===== wird in T[i] neu eingetragen
Punkte ..... s-Tree-Zeiger

Fall 2:
- Ist n1 noch nicht in T[i+1] vorhanden, d.h. ist $I^{i+1}(j0)$ echter Prefix von $I^{i}(j0)$ = w[i]*u1'*w[j1+1] , so muss der (p-)Pfad von root zu p_follow($I^{i+1}(j0)$) ergaenzt werden zu einem (p-)Pfad zu p_follow(w[i]*u1'*w[j1+1]).

Dazu wird mit n2' von n1' aus soweit im (p-)Baum rueckwaerts gegangen, bis s_son(n2',w[i]) ≠ NIL wird. Von dort aus wird der gleiche Pfad wieder vorwaerts gegangen, indem Schritt fuer Schritt ein neuer Knoten konstruiert wird, bis man schliesslich bei n1 angelangt. Dann weiter wie in Fall 1.



Der Algorithmus in der von Weiner angegebenen Form geht von einem festen endlichen Alfabet aus, sodass die Zahl der von einem Knoten ausgehenden Kanten durch |A|+1 begrenzt ist. Dadurch ist es moeglich, die Kanten durch ein Array von Zeigern auf die son-Knoten zu implementieren.

Ist die Groesse des Alfabets unendlich (bzw. nicht von vornherein abzusehen), so kann das Array durch einen binaeren Suchbaum ersetzt werden, in dem nur die wirklich verwendeten Zeichen eingetragen sind (im Suchbaum laesst sich jederzeit nachtraeglich ein neues Zeichen an eine beliebige Stelle zwischen zwei schon vorhandene Zeichen einsortieren).

Sei t die Anzahl der im Eingabestring w vorkommenden verschiedenen Zeichen, dann betraegt der Aufwand [1] zum Eintragen (und genauso zum Aufsuchen) eines Zeichens im binaeren Suchbaum O(log(t)), waehrend der Aufwand im Array-Fall konstant war. Der Faktor log(t) bedeutet jedoch gerade, dass der Aufwand NICHT abhaengig von der "realen Laenge" des Strings w ist [2]. Denn ein Zeichen aus einem Alfabet von t Zeichen (mit diesem Teilalfabet kann man n.Vor. fuer w auskommen) repraesentiert einen Informationsgehalt von log(t) Bits (nach Shannon). D.h. eine Zeichenfolge von n Zeichen aus einem Alfabet von t Zeichen hat eine "reale Laenge" von n*log(t) Bits.

Diese Unabhaengigkeit von der realen Laenge ist insbesondere wichtig fuer Algorithmen im mehrdimensionalen Fall, wo statt nach einzelnen Zeichen nach Zeichenfeldern (im Rechteck-Fall: nach 1-dimensionalen Strings) sortiert wird.

Der Zeitaufwand dieses Algorithmus' ist O(Anzahl der erzeugten Knoten), denn in der Schleife von Fall 2 wird in jedem Schritt auch ein neuer Knoten erzeugt; und in der Schleife zur Bestimmung von n1' wird jedesmal der Wert von Ln1'(w[i]) von NIL auf i gesetzt (und spaeter nicht mehr veraendert), das geht aber bei jedem Knoten hoechstens t mal, insgesamt also hoechstens t*(Anzahl der erzeugten Knoten), wobei t = |A|+1 konstant ist (A:Alphabet).

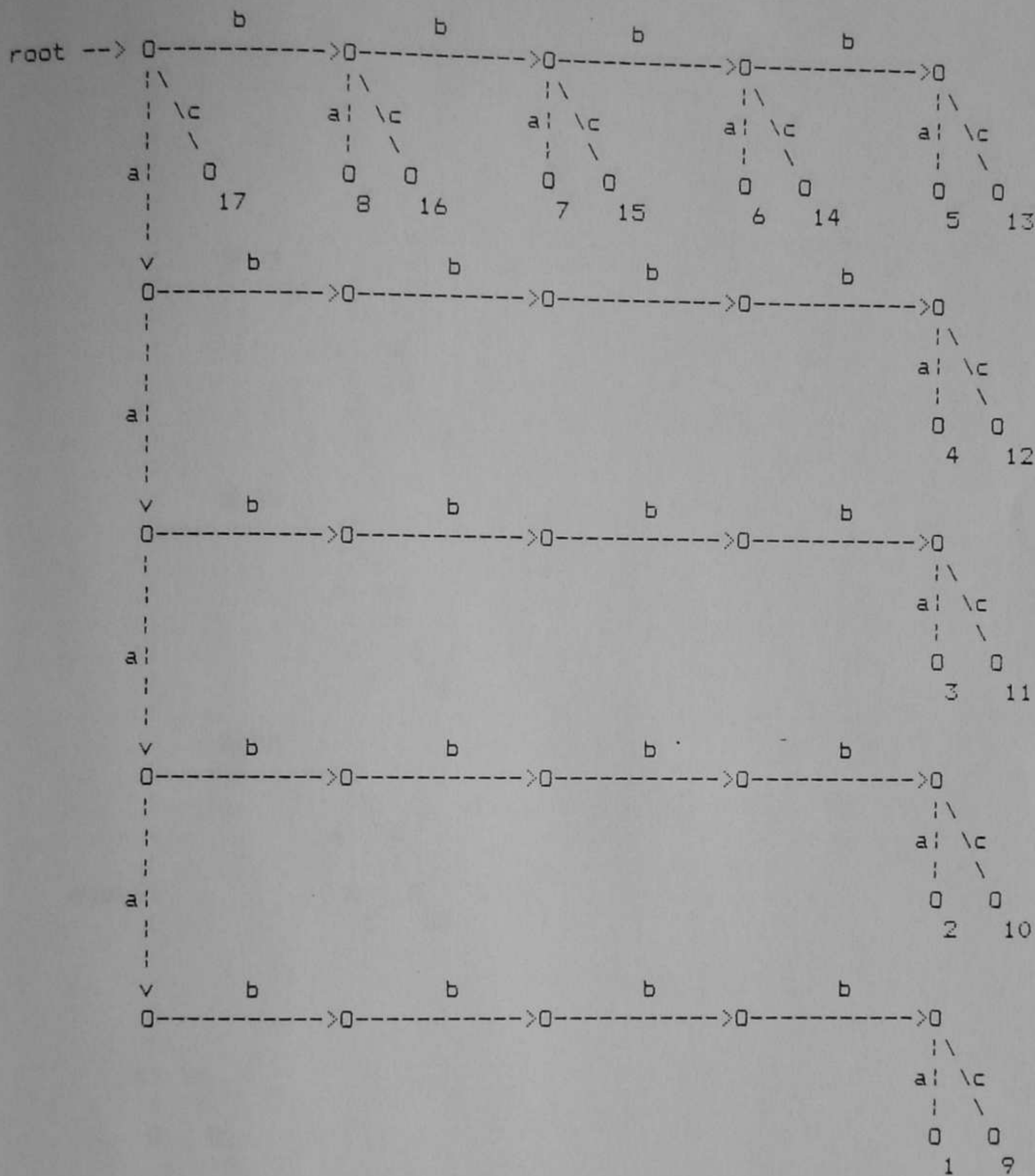
Allerdings kann die Zahl der Knoten quadratisch mit der Laenge von w wachsen, z.B. wenn $w = a^n b^n a^n b^n c$ ist.

```
Beispiel: w = aaaabbbbaaaabbbbc
              12345678901234567
                       11111111
```

----

[1] sofern man AVL-Baeume benutzt, denn bei ihnen ist sichergestellt, dass sie nicht zu linearen Listen entarten koennen (z.B. KNU 73)

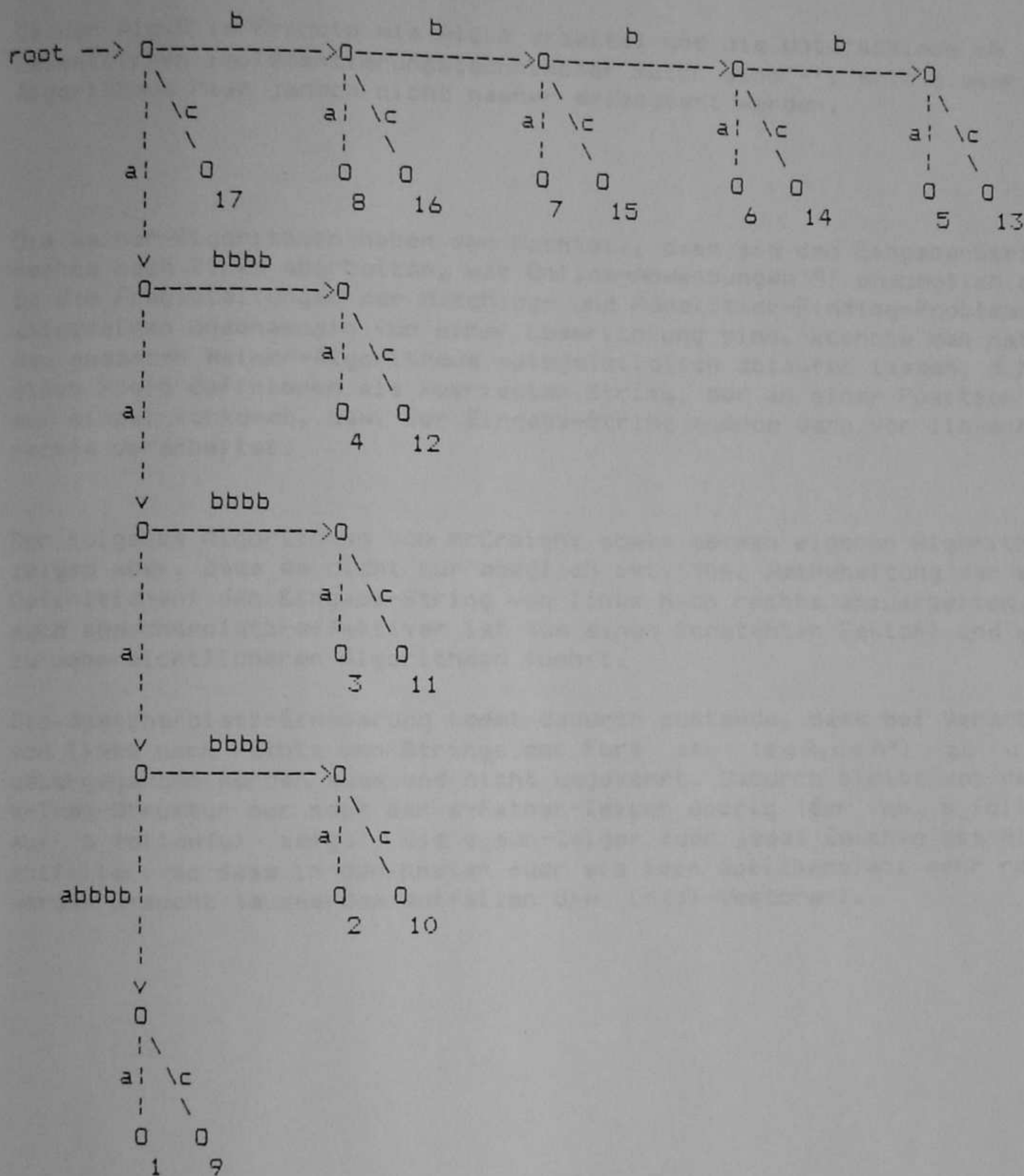
[2] im Gegensatz zu den Ausfuehrungen von McCreight (MCC 76, Seite 268)



1.5.8 Weiners Alg.C

Weiners zweite Idee war daher, die Baeume zu kompaktifizieren, d.h. einen Pfad, von dessen Knoten jeweils nur eine Kante ausgeht, zu einer (Super-)Kante zusammenzufassen. Von jedem (Super-)Knoten gehen dann mindestens zwei (Super-)Kanten aus.

Der Baum aus obigem Beispiel sieht in kompaktifizierter Darstellung so aus:



Die Markierungen der kompaktifizierten Kanten bestehen in Wirklichkeit nur aus der Start- und End-Position des Markierungsstrings, da sonst der Aufwand doch wieder $O(|w|^2)$ werden wuerde.

Man zeigt leicht durch Induktion ueber die Hoehe des Baumes (Laenge des laengsten Pfades in (Super-)Knoten), dass ein Baum mit n Blaettern hoechstens 2*n-1 Knoten hat, wenn kein Knoten genau einen Sohn hat. D.h. die Zahl der Knoten eines kompaktifizierten Prefix-Bi-Trees fuer w waechst nur linear mit |w|. Tatsaechlich gelingt es Weiner, seinen Alg.B fuer kompaktifizierte Bi-Trees (nur der p-Tree ist kompaktifiziert) umzuschreiben zu Alg.C, der mit einem linearen Zeitaufwand auskommt.

Da der Alg.C im Prinzip wie Alg.B arbeitet und die Unterschiede im Wesentlichen implementierungstechnischer Natur sind [1], soll dieser Algorithmus hier jedoch nicht naeher erlaeutert werden.

Die Weiner-Algorithmen haben den Nachteil, dass sie den Eingabe-String w von rechts nach links abarbeiten, was Online-Anwendungen [2] unmoeglich macht. Da die Fragestellungen der Matching- und Repetition-Finding-Probleme im allgemeinen unabhaengig von einer Leserichtung sind, koennte man natuerlich den gesamten Weiner-Algorithmus spiegelbildlich ablaufen lassen, d.h. z.B. einen Posid definieren als kuerzesten String, der an einer Position ENDET und nur einmal vorkommt, usw. Der Eingabe-String wuerde dann von links nach rechts verarbeitet.

Der folgende Algorithmus von McCreight sowie meinen eigenen Algorithmen zeigen aber, dass es nicht nur moeglich ist, (bei Beibehaltung der alten Definitionen) den Eingabe-String von links nach rechts abzuarbeiten, sondern auch speicherplatz-effektiver ist (um einen konstanten Faktor) und m.E. auch zu uebersichtlicheren Algorithmen fuehrt.

Die Speicherplatz-Einsparung kommt dadurch zustande, dass bei Verarbeitung von links nach rechts von Strings der Form $a*u$ $(a \in A, u \in A^*)$ zu u uebergegangen werden muss und nicht umgekehrt. Dadurch bleibt von der s-Tree-Struktur nur noch der s-Father-Zeiger uebrig (der von s_follow(a*u) auf s_follow(u) zeigt), die s_son-Zeiger fuer jedes Zeichen des Alphabets entfallen, so dass in den Knoten fuer sie kein Speicherplatz mehr reserviert werden braucht (ausserdem entfallen die Ln(j)-Vektoren).

----

[1] vor allem: Umgang mit s-Zeigern auf Knoten, die der (p-)Kompaktifizierung "zum Opfer gefallen" sind; Aufsplitten einer kompaktifizierten Kante, um an einem Zwischenknoten eine Verzweigung anzubringen.

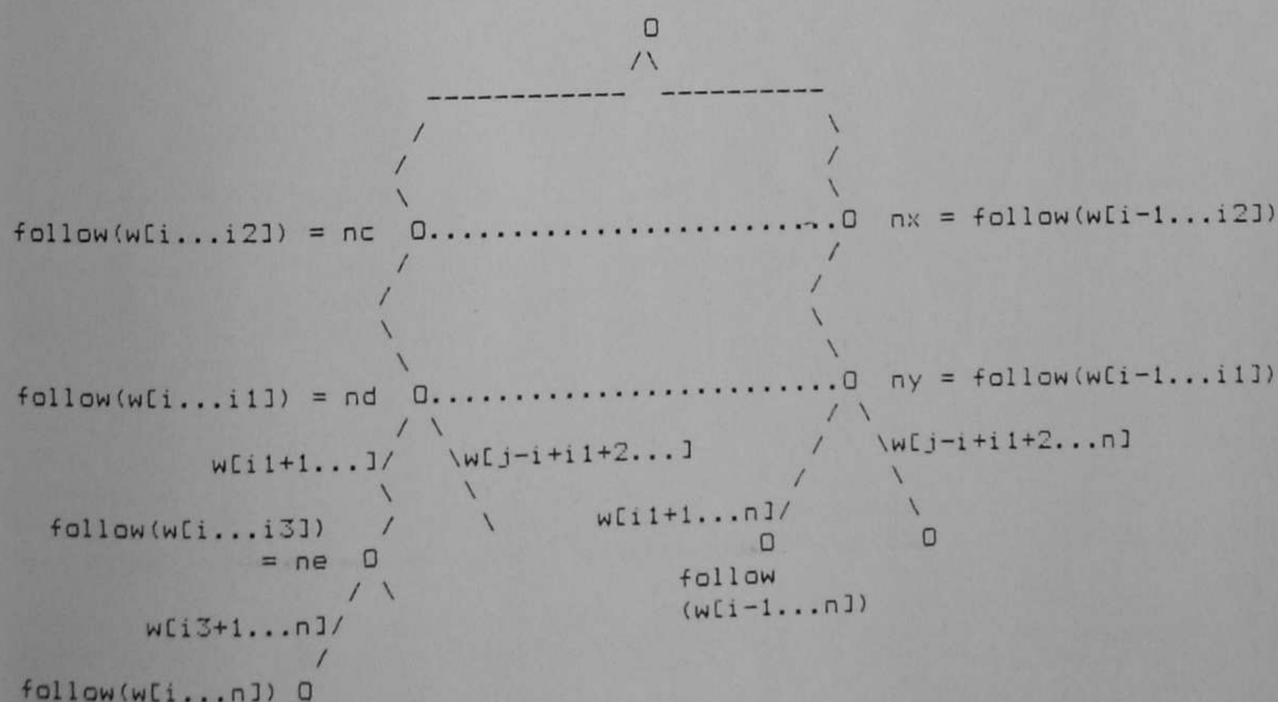
[2] d.h. Anwendungen, bei denen waehrend der Eingabe schon (Matching-)Fragen ueber den bisher eingegebenen String gestellt werden, siehe Anwendungen in Kapitel 1.9 (z.B. Problem 5), sowie Frage 2 in Kapitel 1.1 .

1.5.9 Der Algorithmus von McCreight (MCC 76)

arbeitet wie Weiners Alg.C mit kompaktifizierten Bi-Trees (wobei vom s-Tree wie gesagt nur die s-Father-Zeiger verwendet werden, bei McCreight als Suffix-Links bezeichnet; jeder Nicht-Blatt-Knoten hat ein Suffix-Link). Statt mit Posids arbeitet er mit Suffixen, d.h. ist w = w[1...n] der Eingabe-String, so wird fuer die Position i nicht der Posid w[i...i1], sondern der Suffix w[i...n] in den Baum eingetragen. Das macht keinen Unterschied im Aufwand, denn jeder String w[i...i1...j] mit j>i1 kommt nach Definition nur einmal in w vor, d.h. die Kanten fuer n[i1] - n[i1+1], n[i1+1] - n[i1+2], ... n[n-1] - n[n] (mit n[j]:=follow(w[i...j])) sind alle zu einer (Super-)Kante n[i1] - n[n] zusammengefasst. Der einzige Unterschied zwischen Posid und Suffix ist also, dass von p_follow (w[i...i1-1]) einmal eine Kante mit der Markierung (i1,i1) und einmal eine Kante mit der Markierung (i1,n) ausgehen wuerde.

Andererseits entfaellt durch das Arbeiten mit Suffixen der Fall, dass ein alter Posid verlaengert werden muss (siehe Fall 1 in Alg.6, Fall 2 in Alg.7).

Der Algorithmus geht aus von einem Baum T[1], der nur aus der Wurzel besteht, und traegt nacheinander die Suffixe w[1...n], w[2...n], ... , w[n] ein. Seien in T[i-1] schon die Suffixe w[1...n], ... , w[i-1...n] eingetragen, sei w[i-1...i1] der laengste Prefix von w[i-1...n] , der auch an einer frueheren Position j<i-1 vorkommt, (w[i-1...i1] = w[j...j-i+i1+1]), sei ny = follow(w[i-1...i1]) (ist definiert, denn von ny gehen zwei Kanten aus, fuer w[i1+1...n] und fuer w[j-i+i1+2...n]). ny ist im letzten Schritt (Konstruktion von T[i-1] aus T[i-2]) gerade eingesetzt oder zumindest aufgesucht worden.



Der Algorithmus verfolgt den Weg von ny rueckwaerts (d.h. Richtung root), bis er auf einen Knoten nx trifft, der schon in T[i-2] vorhanden war und daher schon einen Suffix-Link $\neq$ NIL hat (sei label(nx) = w[i-1...i2] fuer ein i2 <= i1). Von nx aus verfolgt er den Suffix-Link und kommt zu einem Knoten nc mit label(nc) = w[i...i2], von nc verfolgt er den Weg mit der Markierung von nx nach ny (d.h. w[i2+1...i1]) und kommt zum Knoten nd mit label(nd) = w[i...i1]. Eventuell muss nd durch Aufspalten einer Kante eingefuegt werden. Ein Aufspalten ist gerechtfertigt, denn von nd aus muessen Wege fuer w[j-i+i1...n] (alt) und w[i1+1...n] (neu) abgehen. Von ny nach nd kann jetzt ein Suffix-Link eingesetzt werden.

Von nd aus wird dann w[i1+1...n] verfolgt bis zum letzten Knoten ne, der auf diesem Weg schon im Baum vorhanden ist. Eventuell muss ne durch Aufspalten einer Kante eingefuegt werden. Sei label(ne) = w[i...i3], dann wird von ne aus eine Kante fuer w[i3+1...n] und ein Blatt eingefuegt. ne ist gleichzeitig das ny fuer den naechsten Schritt. Falls schon nd neu eingefuegt wurde, ist ne = nd (denn der alte Weg von nd aus beginnt dann mit w[j-i+i1+2] $\neq$ w[i1+1]), also wird hoechstens ein Knoten eingefuegt, und der ist ny fuer den naechsten Schritt, erhaelt also dort einen Suffix-Link.

McCreight weist nach, dass der Algorithmus (wie Weiners Alg.C) den Zeitaufwand O(|w|) hat. Der Algorithmus eignet sich nicht fuer Online-Anwendungen, da er in jedem Schritt schon w[i...n] zur Verfuegung haben muss.

KAPITEL 1.6 Der von mir entwickelte Algorithmus

Der von mir entwickelte Algorithmus liest den Eingabe-String w von links nach rechts und baut dazu einen Prefix-Tree mit zusaetzlichen s-Father-Zeigern auf. Er sortiert nacheinander die Positionen 1,2,...,n in den Baum ein (d.h. bildet einen Weg, der dem Posid der Position entspricht), dabei werden zum Einsortieren einer Position i mehrere Zeichen "Vorgriff" benoetigt (naemlich soweit, bis w[i...j] Posid von i in w[1...j] ist).

Sei w[1...i-1] eingelesen, sei i1<=i-1 maximal, so dass jede Position <= i1 einen Posid in w[1...i-1] hat [1], und sei der Posid von i1 gerade w[i1...i-1]. (Fuer i-1 = 1 und fuer i-1 = n ist das der Fall, beide Male ist i1 = i-1.) Der Algorithmus verfolgt den Weg w[i1+1...i-1...j] im Baum - und liest dabei w[i...j] Zeichen fuer Zeichen ein - , solange bis i1+1 einen Posid in w[1...j1] hat (naemlich w[i1+1...j1]).

```
                                  Posid wird gesucht
                                  ¦      letztes eingelesenes Zeichen
                                  ¦                              ¦
¦<--- jede Pos hat Posid --->¦    v                              v

w[1] ... ... ... ... ... w[i1] w[i1+1] ... w[i-1] ... ... w[j-1] w[j] ...

¦<---- alle Posids liegen in diesem Bereich --->¦
```

Dabei sei die Position i1+1 jeweils bis zu einem Knoten nc im Baum einsortiert, d.h. label(nc) = w[i1+1...j-1] [1], w[j] wird eingelesen [1], es gibt vier Moeglichkeiten [2]:

Fall 1: son(nc,w[j]) ≠ NIL,

dieser Fall tritt ein, wenn w[i1+1...j-1] mehrmals in w[1...i-1] vorkommt (echter Prefix eines Posids in w[1...i-1]) und w[i1+1...j] wenigstens einmal dort vorkommt (Prefix eines Posids in w[1...i-1]),
verfolge den (vorhandenen) Weg weiter:
setze nc := son(nc,w[j])

----

[1] bei der Implementierung wurden folgende Namen verwendet:
j: readpos, i1: sortpos, nc: curnode, w[j]: curch, w[i3+1]: existch

[2] Die Numerierung erfolgt hier entsprechend der bei der Implementierung des Algorithmus gewaehlten Numerierung; die Faelle werden in der Reihenfolge 1, 4, 3, 2 vorgestellt.

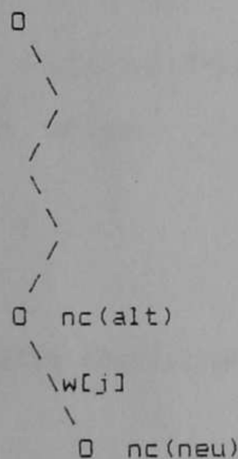
Fall 4: son(nc,w[j]) = NIL, aber nc ist kein Blatt

dann kommt w[i1+1...j-1] mehrmals vor in w[1...j], aber w[i1+1...j] kommt dort gar nicht vor, ist also Posid fuer i1+1 in w[1...j].
Haenge an nc eine Kante zu einem Knoten ne, die mit w[j] markiert ist, an, bilde den s-Father-Zeiger von ne (siehe unten),

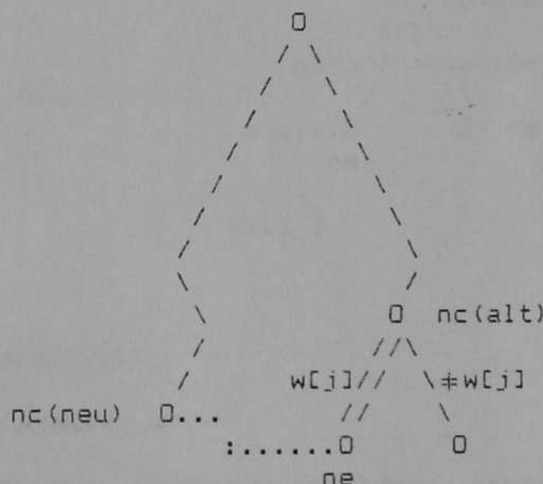
(x1):

finde alle weiteren dadurch schon einsortierten Positionen, d.h.
erhoehe i1 um 1, setze nc := s-Father(ne),
solange nc Blatt ist und i1 die Position von nc ist,
(dann ist nc als s-Father-Zeiger von ne eingesetzt worden und kam vorher nicht im Baum vor, d.h. label(nc) = w[i1+1...j] ist Posid fuer den laufenden Wert von i1+1):
    erhoehe i1 um 1 und setze nc := s-Father(nc)

Bricht die Schleife ab, so ist nc nicht mehr Posid fuer i1+1, d.h. alle Positionen von 1 bis i1 haben einen Posid in w[1...j], die Position i1+1 ist schon bis nc im Baum einsortiert (label(nc) = w[i1+1...j])



Fall 1



Fall 4

Fall 1 oder 4 tritt immer dann ein, wenn w[i1+1...j-1] mehrmals in w[1...i-1] vorkommt. Nach dem folgenden Lemma gilt sogar:
wenn Fall 1 oder 4 nicht eintritt, kommt w[i1+1...j-1] hoechstens einmal in w[1...j-2] vor, etwa w[i1+1...j-1] = w[i2...i3], und mindestens einmal, da sonst schon w[i1+1...j-1] Posid waere.

Fall 3: nc ist Blatt, w[i3+1] = w[j],

d.h. w[i1+1...j] mehrmals in w[1...j],
dann verfolge den Weg weiter: haenge an nc eine neue Kante mit w[j] zu einem Knoten ne an und konstruiere den s-father-Zeiger von ne (siehe unten),
setze nc := ne

Fall 2: nc ist Blatt, w[i3+1] $\neq$ w[j],

dann ist w[i1+1...j] Posid fuer i1+1 in w[1...j]
(denn w[i1+1...j-1] ist mehrmals in w[1...j], naemlich genau an den Positionen i2 und i1+1, aber w[i1+1...j] ist einmal in w[1...j], ebenso w[i2...i3+1]),
haenge an nc eine Kante mit w[i3+1] zu einem Knoten nd und eine mit w[j] zu einem Knoten ne an,
bilde die s-Father-Zeiger von nd und ne (siehe unten),
weiter wie bei (x1) in Fall 4

```
        O                                                  O
         \                                                / \
          \                                              /   \
           \                                            /     \
           /                                           /       \
          /                                           /         \
          \                                          /          /
           \                                        /           \
            O  nc(alt)                             /             O  nc(alt)
           //                                      \            //\\
         //w[j]=w[i3+1]                            /       w[j]//  \\w[i3+1]
         //                               nc(neu) O.....       //    \\
        O  nc(neu)                                    :.....O         O  nd
                                                            ne

     Fall 3                                              Fall 2
```

Damit sind alle moeglichen Faelle abgehandelt.

Die Konstruktion eines s-Father-Zeigers fuer jeden neu gebildeten Knoten ne erfolgt durch ein Unterprogramm:

Ist nc der (p-)Father von ne (und die Kante von nc nach ne mit a markiert), so hat nc bereits einen s-Father-Zeiger, der auf einen Knoten nd zeige. Dann ist label(ne) = label(nc)*a , b*label(nd) = label(nc) fuer geeignetes $b \in A$, also b*label(nd)*a = label(ne), d.h. wenn es einen Knoten nf = (p-)son(nd,a) im Baum gibt, so ist er der s-Father von ne und der Zeiger wird entsprechend eingetragen. Wenn nf noch nicht existiert, wird er neu in den Baum eingetragen und rekursiv fuer ihn ein s-Father-Zeiger gebildet. (Das Unterprogramm terminiert, da label(nf) kuerzer ist als label(nc).)

```
  follow(u) = nd  O.................O  nc = follow(b*u)
                   \                 \
                    \                 \
                   a \                 \ a
                      \                 \
                       \                 \
     follow(u*a)=nf     O.................O  ne = follow(b*u*a)
```

Das Programm wurde unter OS9/6809 in der (Algol-aehnlichen) Sprache C implementiert (siehe Programmauflistung, Anhang 3).

Lemma:
wenn u := w[i1+1...j-1] mehrmals in w[1...j-2] vorkommt, dann kommt es nicht in w[i-j+i1+2...j-2] vor (sondern nur in w[1...i-1])

Bew:
sonst wuerde u an einer Position i2 mit i-j+i1+2<=i2<=i1 beginnen und haette somit den Posid von i2 (bzgl. w[1...i-1]) als Prefix (denn der Posid endet spaetestens bei i-1, aber u fruehestens bei i),

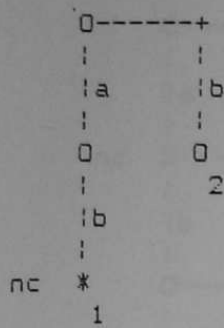
da u mehrmals vorkommt, kommt es noch an einer anderen Position i3<=i1 vor
ist i3>=i-j+i1+2,
so hat u auch den Posid von i3 als Prefix, W.!
ist i3<i-j+i1+2,
so kommt der Posid von i2 mehrmals in w[1...i] vor (naemlich an i3 und an i2), W.!

```
w[1] ... w[i-j+i1+2] ... w[i1] w[i1+1] ... w[i-1] w[i] ... w[j-2] w[j-1] ...
                                |<----------- Laenge von u ------------>|
        |<------------- Laenge von u -------------->|
        |<---------------- u kommt nicht vor ----------------->|
|<---------------- u kommt vor ---------------->|
```

Beispiel fuer den Ablauf des Algorithmus:

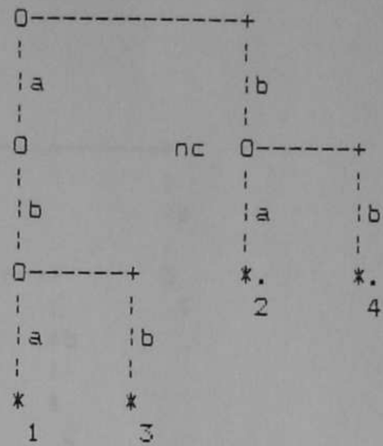
```
w = ababbaab#                 Legende:  *    neu eingetragener Knoten
    123456789                           *.   neu eingetragener sfather-Knoten
                              bei der Angabe von  w = w[1...j]
                              steht jeweils zwischen  w[i1]  und  w[i1+1]
                              ein Strich '
```

```
                                                         0------+
   nc  0                 nc  0------+                    |      |
        1                    |      |                    |a     |b
                             |a     |b                   |      |
                             |      |                 nc 0      0
                             *      *                     1      2
                              1      2

     w = a'                w = ab'                    w = ab'a
         1                     12                         12 3

                          (Fall 2)                   (Fall 1)
```



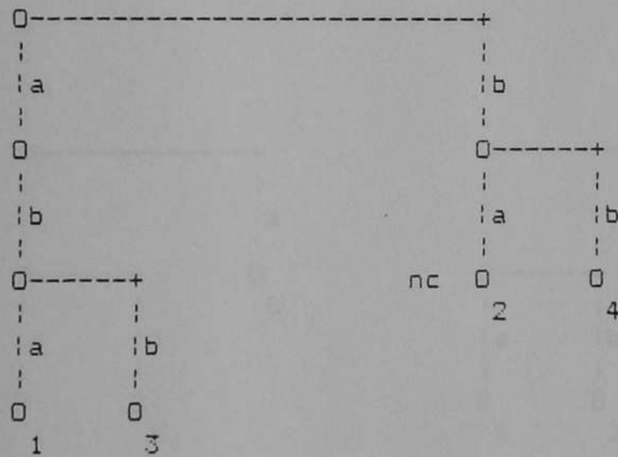
w = ab'ab
12 34

(Fall 3)



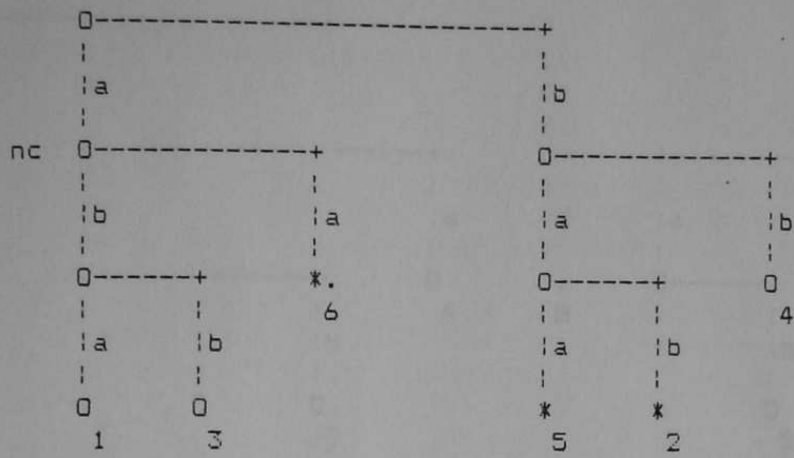
w = abab'b
1234 5

(Fall 2)



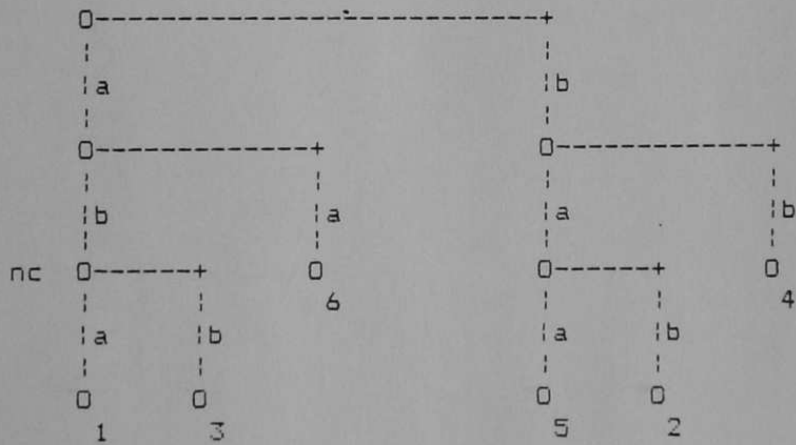
w = abab'ba
1234 56

(Fall 1)



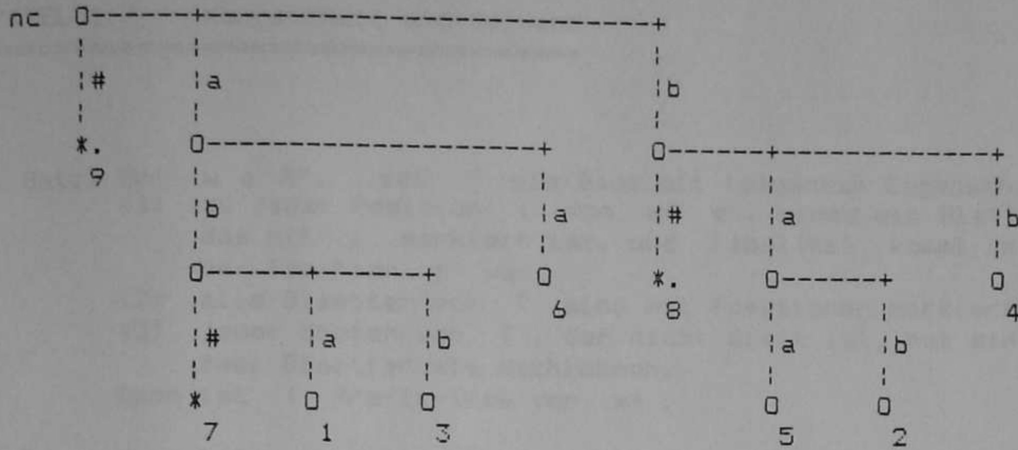

```
w = ababba'a
    123456 7
```

(Fall 2)




```
w = ababba'ab
    123456 78
```

(Fall 1)



w = ababbaab#'
123456789

(Fall 4)

KAPITEL 1.7 Korrektheit und Aufwand

1. Satz: Sei $w \in A^*$, sei T ein Baum mit folgenden Eigenschaften:
   (1) zu jeder Position i von w# ex. genau ein Blatt ni von T das mit i markiert ist, und label(ni) kommt in w# an der Position i vor,
   (2) alle Blaetter von T sind mit Positionen markiert, und
   (3) jeder Knoten von T , der nicht Blatt ist, hat mindestens zwei Blaetter als Nachkommen.

   Dann ist T Prefix-Tree von w# .

Bew: Nach Def 1.4.17 bleibt zu zeigen, dass eine 1-1-Zuordnung zwischen Blaettern von T und Posids von w# existiert.

Sei ni Blatt von T,
nach (2) ist ni mit einer Position i von w# markiert,
nach (1) kommt label(ni) in w# an der Position i vor,
sei nf = father(ni) , er hat nach (3) mindestens ein weiteres Blatt nk als Nachkommen, das mit der Position k markiert sei
($k \neq i$ nach (1)),
sei label(ni) = u*a mit $u \in A^*$, $a \in A$,
dann ist label(nf) = u
und u kommt in w# an den Positionen i und k vor (nach 2.)
aber u*a kommt in w# nur an der Position i vor:
  denn kaeme u*a bei Position j vor,
  so sei nach (1) nj das Blatt, das mit j markiert ist,
  dann sind u*a und label(nj) Prefixe von w[i...n],
  also label(ni) = u*a Prefix von label(nj) oder label(nj) Prefix von u*a,
  nach 3. ist aber weder label(ni) echter Prefix von label(nj) noch label(nj) echter Prefix von label(ni),
  also label(ni) = label(nj)
  d.h. ni = nj, nach 1. folgt i = j

Sei jetzt umgekehrt u*a Posid der Position i in w#
($u \in A^*$, $a \in A$),
sei ni das Blatt in T , das mit i markiert ist (nach (1)),
dann folgt nach den obigen Ausfuehrungen label(ni) ist Posid der Position i in w#,
also label(ni) = u*a,
d.h. follow(u*a) = ni ist Blatt in T

2. Lem: fa. nu, nv $\in$ |T| gilt
   nu Nachkomme von nv <==> label(nv) Prefix von label(nu)

Bew: Induktion ueber die Zahl der Knoten zwischen nv und nu:
nu = nv : klar
die Beh. gelte fuer Knoten nu' und nv', die hoechtens n Kanten auseinanderliegen, seien nu und nv Knoten, die n+1 Kanten auseinanderliegen,

"==>": sei nu' := father(nu) (nu=root ist trivial)
nu' ist Nachkomme von nv ,
dann ist label(nv) Prefix von label(nu') (Ind.vor.)
Prefix von label(nu)

"<==": sei label(nu) = u*a fuer u ∈ A*, a ∈ A
(label(nu)=$ ist trivial)
sei nu' = father(nu),
dann ist label(nu') = u,
und label(nv) Prefix von label(nu')
nach Ind.vor. folgt nu' Nachkomme von nv
also nu Nachkomme von nv

3. Korr: Ist nu Blatt von T, nv ∈ |T| beliebig,
so ist label(nu) nicht echter Prefix von label(nv)

Bew: sonst waere nach 2. nv echter Nachkomme von nu,
W.! zu nu Blatt

Im folgenden sollen die Eigenschaften (1) bis (3) aus Satz 1 fuer den vom Algorithmus erzeugten Baum nachgewiesen werden (vgl. Satz 11).

4. Lem: Alle Blaetter, die der Algorithmus erzeugt, sind mit einer Position markiert

Bew: klar (vgl. Anhang 3)

5. Lem: Wurde vom Algorithmus ein Blatt ni mit der Position i markiert, so kommt label(ni) in w# an der Position i vor

Bew: Induktion ueber die Anzahl j der Schleifendurchlaeufe:
j=0: klar (root hat die Position 1)
j-1 ⟶ j: Fall 1: keine neuen Knoten werden erzeugt
Fall 4: fuer den neu erzeugten Knoten ne gilt
label(ne) = w[i1+1...j], (nach 8.)
er wird mit der Position i1+1 markiert
Fall 3: die alte Position wird in den neu gebildeten Knoten ne uebernommen,
label(ne) = w[i1+1...j-1]*w[j]
Fall 2: fuer nd gilt label(nd) = w[i2...i3+1],
er wird mit i2 , der Position von nc , markiert,
fuer ne gilt label(ne) = w[i1+1...j-1]*w[j],
er wird mit i1+1 markiert
Unterprogramm zur Konstruktion eines s-father-Zeigers:
evtl. wird nf , der sfather-Knoten von nc , neu gebildet,
ist label(nc) = w[p...q] , so ist
label(nf) = w[p+1...q],
nf wird mit p+1 markiert, wenn nc mit p markiert ist

6. Lem: Zu jedem Zeitpunkt im Algorithmus-Ablauf gilt:
zu jeder Position i <= i1 ex. genau ein Blatt ni ,
das mit i markiert ist

Bew: Induktion ueber i1 :
Zu Beginn des Algorithmus ist i1 = 1 und der Baum besteht nur aus der Wurzel, die mit der Position 1 markiert ist.
i1 wird nur in Fall 2 oder 4 erhoeht, wobei jeweils ein neues Blatt mit dem neuen Wert von i1 gebildet wird.
Wird ein Blatt durch Anhaengen eines Knotens zum Nicht-Blatt (in Fall 3 und 2), so wird dessen Position an das (bzw. eines der) neu angehaengten Blaetter weitergegeben.

7. Lem: Unmittelbar nachdem der Algorithmus Fall 2 oder Fall 4 durchlaufen hat, gilt:
jeder Knoten, der nicht Blatt ist, hat mindestens zwei Nachkommen

Bew: Induktion ueber die Anzahl der Durchlaeufe von Fall 2 oder 4:
Zu Beginn besteht der Baum nur aus dem Blatt root.
Fall 4: gilt nach Ind.vor., da nur ein neues Blatt angehaengt wurde, und dadurch kein neuer Nicht-Blatt-Knoten erzeugt wurde
Fall 2: nach 9. haengen alle Blaetter, die der Algorithmus seit dem letzten Durchlauf von Fall 2 oder 4 erzeugt hat, an einer s-father-Kette mit dem Startknoten nc ,
an nc und damit auch an alle Blaetter dieser s-father-Kette werden jeweils zwei Blaetter nd und ne (bzw. deren sfather-Knoten) angehaengt,

8. Lem: Im Algorithmus gilt
label(nc) = w[i1+1...j-1]

Bew: Induktion ueber die Anzahl j der Schleifendurchlaeufe:
Zu Beginn gilt die Beh., da nc = root, i1 = 1, j = 2,
Sei nc der Wert vor Verarbeitung des Zeichens w[j] ,
sei nc' der Wert danach.
Fall 1: label(nc') = label(nc)*w[j] = w[i1+1...j] ,
i1 bleibt unveraendert
Fall 3: wie Fall 1
Fall 2: label(nc') = label(nc)*w[j] = w[i1+1...j] ,
in der Schleife ab [1] wird i1 um 1 erhoeht genau dann wenn nc' = sfather(nc') gesetzt wird
Fall 4: wie Fall 2

9. Lem: Tritt der Algorithmus in Fall 2 ein, so haengen alle Blaetter, die er seit dem letzten Durchlauf von Fall 2 oder 4 erzeugt hat, an einer sfather-Kette mit dem Startknoten nc

Bew: neue Knoten werden nur in Fall 3 erzeugt,
dabei wird immer nc auf das neu erzeugte Blatt gesetzt,
und alle Knoten, die vom Unterprogramm zur Konstruktion eines sfather-Zeigers neu gebildet werden, haengen an einer sfather-Kette mit dem Start-Knoten nc

10. Lem: zu jedem Knoten nu = follow(a*u) (a ∈ A, u ∈ A*)
setzt der Algorithmus einen sfather-Zeiger von nu nach
nu' = follow(u) ein

Bew: nach jeder Erzeugung eines neuen Knotens wird das Unterprogramm zur Konstruktion eines sfather-Zeigers (ggf. rekursiv) aufgerufen

11. Satz: Der Algorithmus baut aus einem Eingabestring w# eine Datenstruktur mit folgenden Eigenschaften auf (Prefix-Bi-Tree):
(a) u ist Prefix eines Posids von w
<==> ex. ein Knoten nu im Baum mit label(nu)=u
(b) von jedem Knoten nu=follow(a*u) (a ∈ A, u ∈ A*)
fuehrt ein s-father-Zeiger zum Knoten nu'=follow(u)

Bew: (b) nach Lemma 10.

(a) mit Satz 1. nach den Lemmata 4., 5., 6. und 7.,
denn beim Einlesen des Zeichens # tritt Fall 2 oder 4 ein,
da # nicht in w und somit auch noch nirgends in T als
Kantenmarkierung vorkommt

12. Bem: Der Zeitaufwand des Algorithmus ist - wie bei Weiner -
O(Anzahl der erzeugten Knoten),
denn nachdem ein Zeichen eingelesen wurde, wird
einer der Faelle 1 bis 4 abgearbeitet, dann wird das naechste
Zeichen eingelesen. Jeder der Faelle hat aber einen konstanten
Zeitaufwand, wenn man vom Unterprogramm zur Konstruktion des
s-Father-Zeigers absieht.

Der Aufwand ist also
O(|w|) + (Aufwand fuer alle Aufrufe des Unterprogramms).
Das Unterprogramm erzeugt jedesmal, wenn es sich
rekursiv selbst aufruft, einen Knoten, hat also pro erzeugten
Knoten einen konstanten Aufwand.

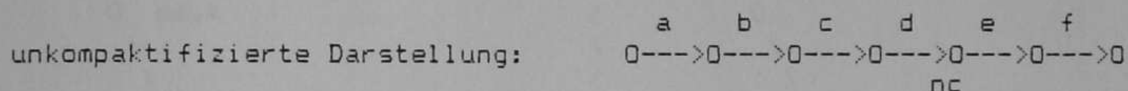
D.h.
(Aufwand fuer alle Aufrufe des Unterprogramms)
= O(Anzahl der erzeugten Knoten) ,
damit ist der Gesamtaufwand:
O(|w|) + O(Anzahl der erzeugten Knoten)
= O(Anzahl der erzeugten Knoten)

KAPITEL 1.8 Der Algorithmus fuer den kompaktifizierten Bi-Tree

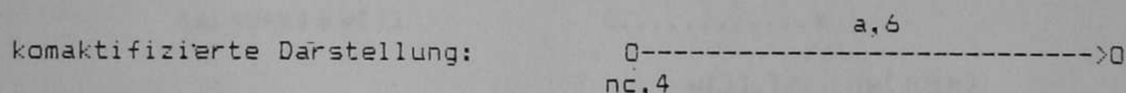
Der in Kapitel 1.6 vorgestellte Algorithmus kann analog zu Weiners Alg.C so umgeschrieben werden, dass er einen kompaktifizierten Bi-Tree erzeugt (vgl. 1.5.8). Der prinzipielle Ablauf bleibt dabei der gleiche (wie in 1.6), es aendert sich im wesentlichen nur die Darstellung des Trees.

Zusaetzlich zu dem Zeiger nc auf den gerade bearbeiteten (Super-)Knoten wird jetzt durch eine natuerliche Zahl k angegeben, an welcher "Stelle" innerhalb der gerade bearbeiteten Super-Kante sich der Algorithmus befindet. Wurde bisher nc auf den naechsten Knoten gesetzt, so wird jetzt, sofern dieser naechste Knoten wegen der Kompaktifiziertheit nicht mehr existiert, nc beibehalten und k um 1 erhoeht.
w[j0] gebe das erste Zeichen der Markierung der aktuellen Kante an (es wird nur dieses erste Zeichen und die Laenge der Kantenmarkierung abgespeichert).

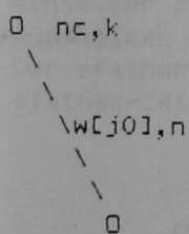
unkompaktifizierte Darstellung:



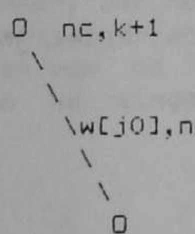
komaktifizierte Darstellung:



Damit spalten sich die Faelle 1 bis 4 aus Kapitel 1.6 jeweils in mehrere Unterfaelle auf:



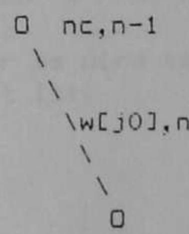
(alt)



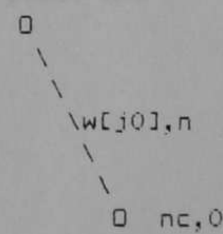
(neu)

Fall 1a+1b

Das Ende der aktuellen Kante ist noch nicht erreicht (0 <= k < n-1).



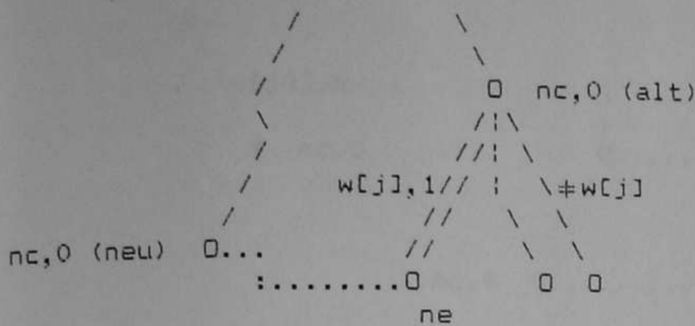
(alt)



(neu)

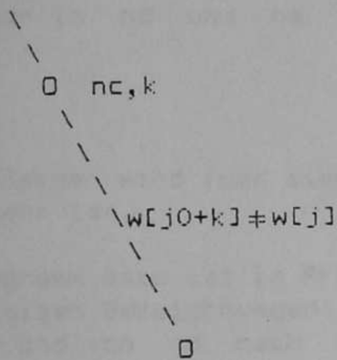
Fall 1c

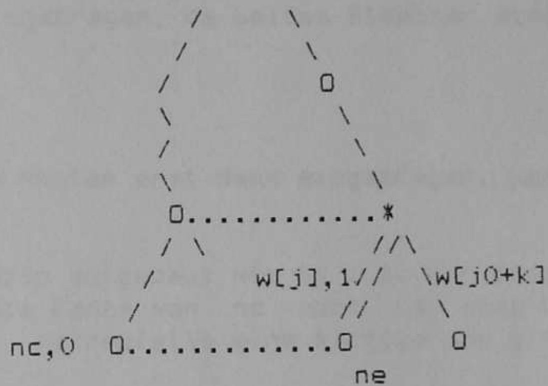
Das Ende der aktuellen Kante ist erreicht.



Fall 4a

Es gibt schon einen Knoten, an dem eine Abzweigung fuer w[j] eingesetzt werden kann.

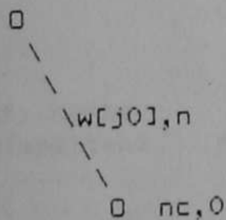


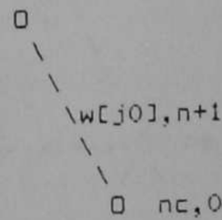


(alt) (neu)

Fall 4b

Die aktuelle Kante muss aufgespalten werden, um eine Abzweigung fuer w[j] einsetzen zu koennen. Der neu eingesetzte Knoten * ist Nicht-Blatt und muss einen sfather-Zeiger erhalten.
Der sfather-Knoten von nd wird gebildet, aber es wird kein sfather-Zeiger in nd eingetragen, da es Blatt ist.

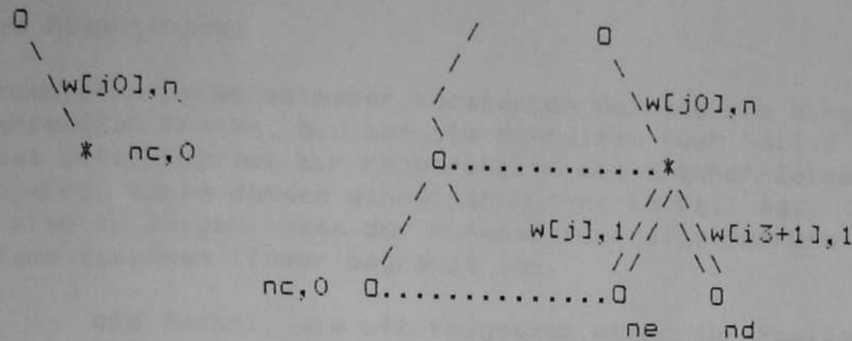




(alt) (neu)

Fall 3

Die Laenge der aktuellen Kante wird einfach um 1 erhoeht.

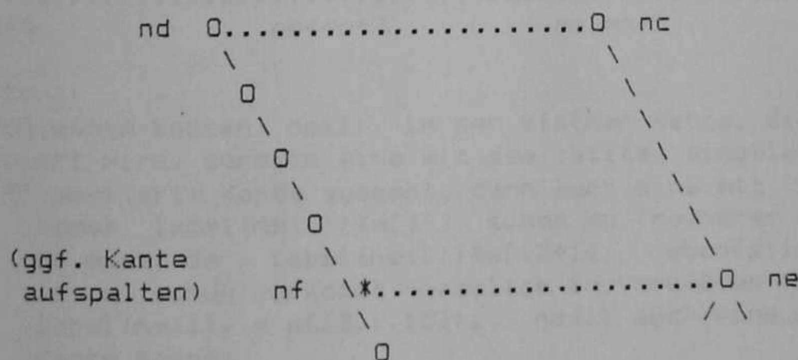


Fall 2

Fuer w[j] und fuer w[i3+1] wird je eine neue Kante der Laenge 1 angehaengt. Dadurch wird der Knoten * zum Nicht-Blatt und muss einen sfather-Zeiger erhalten.
Die sfather-Knoten von nd und ne werden gebildet, aber es werden keine sfather-Zeiger in nd und ne eingetragen, da beides Blaetter sind.

Ein sfather-Zeiger wird fuer einen Knoten erst dann eingetragen, wenn er kein Blatt mehr ist.

Das Unterprogramm dazu ist im Prinzip aufgebaut wie in 1.6, nur dass jetzt (mit den dortigen Bezeichnungen) die Kante von nc nach ne eine Laenge k >= 1 hat und von nd nach nf keinesfalls eine einzige und gleichlange Kante verlaufen muss.
Der Weg von nd nach nf kann aus mehreren Kanten bestehen, wobei ausserdem die letzte Kante zu lang sein kann und dann aufgespalten werden muss, um nf einsetzen zu koennen.
Ein Aufspalten ist gerechtfertigt, denn der sfather-Zeiger von ne wird nur dann konstruiert, wenn ne kein Blatt ist, also mindestens zwei Nachfolger hat. Damit hat aber auch nf mindestens zwei Nachfolger.



Aufwand des Algorithmus:

Alle Programmteile haben entweder konstanten Aufwand pro eingelesenes Zeichen oder pro erzeugten Knoten, bis auf die Schleifen fuer Fall 2 und Fall 4b, in denen das Unterprogramm zur Konstruktion des sfather-Zeigers [1] aufgerufen wird, sowie dessen einzelnen Aufruf in Fall 4a.
Es bleibt also zu zeigen, dass der Aufwand fuer alle Aufrufe dieses Unterprogramm zusammen linear begrenzt ist.

Sei	m'	die Anzahl, wie oft insgesamt einer der Faelle 2, 4a oder 4b eintritt (m' <= |w|);
sei	r(i)	die Position, die als letztes eingelesen wurde, wenn zum i.ten mal einer der Faelle 2, 4a oder 4b eintritt (1 <= i <= m', 1 <= r(i) <= |w|);
sei	a(i)	die Anzahl der Aufrufe des Unterprogramms in diesem Fall;
sei	b(i,j)	die Anzahl der Knoten, die bei einem Aufruf (naemlich dem j.ten) des Unterprogramms durchlaufen werden (1 <= j <= a(i));
sei	c(i,j)	die Laenge der Kante von nc nach ne beim j.ten Aufruf im i.ten Eintreten eines der Faelle 2, 4a, 4b (1 <= j <= a(i)), sei c(i,a(i)+1) die Laenge der Kante von nd(i) nach nf(i) .

Beispiel: a(i)=4

j	1	2	3	4	5
b(i,j)	5	3	3	2	
c(i,j)	9	6	5	4	4

```
                                                 nd1 0................0 nc1
                                                     :                :
                                                     0                :
                                                     :                :
                                                     0                :
                                                     :                :
                                   nd2 0.............0 nc2            :
                                       :             :                :
                     nd3 0.............0 nc3         :                :
nd4                      :             :             :                :
  0.................0 nc4              :             :                :
  :                 :                  :5            :6               :9
  :4                :4                 :             :                :
  0.................0..................0.............0................0
     nf4          ne4=nf3          ne3=nf2        ne2=nf1            ne1
```

Es gilt:
Wenn von einem Knoten ne(i) in der sfather-Kette, die nach und nach konstruiert wird, bereits eine mit dem letzten eingelesenen Zeichen w[j'] [2] markierte Kante ausgeht, dann auch eine mit w[i3+1] markierte.
Denn: kommt label(ne(i))*w[j'] schon an frueherer Stelle in w vor, so muss, da label(ne(i))*w[i3+1] ebenfalls an frueherer Stelle in w schon vorkommt (naemlich im Vorkommen von label(ne(1)) = w[i2...i3]), ne(i) auch eine mit w[i3+1] markierte Kante haben.

----

[1] In der Implementierung in Anhang 4 heisst das Unterprogramm "findsfather".

[2] In 1.6 ist die Position des letzten eingelesenen Zeichens mit j bezeichnet, hier wird j' dafuer verwendet, um Verwechslungen mit dem Index j zu vermeiden. Die Variablen i2 und i3 behalten aber hier dieselbe Bedeutung wie in 1.6.

Und:
Die Schleifen in Fall 2 und 4b, in denen das Unterprogramm aufgerufen wird, laufen jeweils hoechstens solange, bis ein Knoten ne(i) bereits eine mit w[j'] markierte Kante (und damit auch eine mit w[i3+1] markierte Kante) hat,
denn dieses ne(i) ist kein Blatt mehr, hat also bereits einen sfather-Zeiger eingetragen.

Damit gilt,
wenn i1' den Wert von i1 vor dem i.ten Eintritt in einen der Faelle 2, 4a oder 4b bezeichnet,
und i1" den Wert von i1 nach dem i.ten Eintritt:
a(i) <= i1" - i1'
denn i1 wird in den Faellen 2, 4a und 4b solange erhoeht, bis ein ne(i) gefunden wird, das bereits eine mit w[j'] markierte Kante hat.
also a(1) + ... + a(m') <= |w|

Weiter gilt fuer 1 <= j <= a(i)-1, wie aus der obigen Skizze ersichtlich,
b(i,j) - 2 + c(i,j+1) <= c(i,j)
d.h. b(i,j) <= c(i,j) - c(i,j+1) + 2

Also ist der Aufwand fuer die Bearbeitung des i.ten Eintritts von Fall 2,4a oder 4b:

$$\sum_{j=1}^{a(i)} b(i,j) \ <= \ (\sum_{j=1}^{a(i)} c(i,j) - c(i,j+1) + 2)$$

$$= \ 2*a(i) + c(i,1) - c(i,a(i)+1)$$

Schliesslich ist
c(i+1,1) - c(i,a(i)+1) <= r(i+1) - r(i)
denn nachdem der i.te Eintritt von Fall 2,4a,4b abgeschlossen ist, wird der Weg von nf(i) weiter nach unten verlaengert (in Fall 1 oder 3), bis nach r(i+1)-r(i) Schritten der (i+1).te Eintritt von Fall 2,4a,4b erfolgt. In diesen r(i+1)-r(i) Schritten kann die Laenge der Kante von nd nach nf aber hoechtens um r(i+1)-r(i) erhoeht werden.

Aus alledem folgt der lineare Zeitaufwand des Algorithmus:

$$\sum_{i=1}^{m'} \sum_{j=1}^{a(i)} b(i,j) \ <= \ 2 * (\sum_{i=1}^{m'} a(i)) \ + \ (\sum_{i=1}^{m'} c(i,1) - c(i,a(i)))$$

$$<= \ 2*|w| + c(1,1) + (\sum_{i=1}^{m'-1} c(i+1,1) - c(i,a(i))) - c(m',a(m'))$$

$$<= \ 2*|w| + r(1) + (\sum_{i=1}^{m'-1} r(i+1) - r(i))$$

$$<= \ 2*|w| + r(m')$$

$$<= \ 3*|w|$$

KAPITEL 1.9 Anwendungsbeispiele

Mit Hilfe des Prefix-Trees kann eine ganze Reihe von Matching- und Repetition-Finding-Problemen IN LINEARER ZEIT [1] geloest werden. Die wichtigsten Probleme und ihre Zusammenhaenge sollen hier kurz dargestellt werden.

1. Das Pattern-Matching-Problem (gegeben v und w, finde alle Positionen, an denen v in w vorkommt; evtl. sind auch mehrere v gegeben)

2. Finde den laengsten Prefix von v, der in w vorkommt

3. Gegeben w, finde einen laengsten Teilstring von w, der zweimal vorkommt

4. Gegeben w, finde einen laengsten Teilstring von w, der k mal vorkommt

5. Das File-Transmission-Problem (WEI 73-2): gegeben Strings v und w, stelle v moeglichst kurz dar unter Verwendung von "Referenzen auf w", d.h. wenn z.B. v[i...j] = w[k...l] ist, so kann v dargestellt werden als v[1...i-1] * (k,l) * v[j+1...m] , wobei das Zahlenpaar (k,l) eine Referenz auf w ist und als ein Zeichen gezaehlt wird.
   Die anschauliche - und praktische - Bedeutung davon ist folgende:
   Soll ein Datenstrom v von einem Ort zum anderen uebertragen werden und verfuegen Sender und Empfaenger ueber den gleichen Datenbestand w, so kann die Uebertragung durch die Verwendung von Referenzen auf w stark beschleunigt werden.
   Entsprechendes gilt auch fuer die Speicherplatzeinsparung, wenn v mit Referenzen auf w abgespeichert wird.

6. Das File-Compare-Problem: gegeben Strings v und w, finde die Unterschiede zwischen v und w heraus, bzw. finde heraus, wie w aus v entstanden ist, an welchen Stellen Zeichen eingefuegt wurden, an welchen Stellen Zeichen geloescht wurden.
   Dieses Problem ist sehr verwandt mit 5., auch hier geht es um die Darstellung durch Referenzen auf w.

7. Gegeben Strings v und w, finde einen laengsten gemeinsamen Teilstring t von v und w. (t kann an beliebigen Stellen in v und w vorkommen, muss also nicht Prefix oder Suffix sein.)

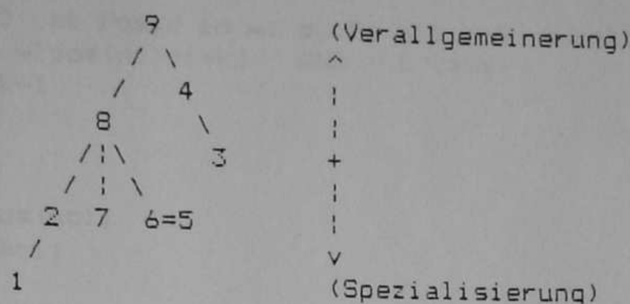
8. Gegeben Strings v und w, finde zu jeder Position i von v den laengsten Prefix von v[i...m], der in w vorkommt.

9. Gegeben Strings v und w, finde zu jeder Position i von v den laengsten Prefix von v[i...m], der in w genau (mindestens) k mal vorkommt.

----

[1] d.h. sind v=v[1...m] und w=w[1...n] gegeben, in der Zeit O(m+n), ist nur w gegeben, in der Zeit O(n)

Zwischen diesen Problemen gibt es folgende Zusammenhaenge:



2\. loest 1.: v kommt genau dann in w vor, wenn der laengste gefundene Prefix v selbst ist
8\. loest 2.: nimm die Loesung zu Position i=1
8\. loest 7.: jeder gefundene Prefix von maximaler Laenge (unter allen gefundenen Prefixen) ist eine Loesung
8\. loest 6.: beginne bei i:=1,
ist v[i...j] = w[k...l] der laengste Prefix von v[i...m], der in w vorkommt,
so ersetze v[i...j] durch (k,l) ,
setze i:=j+1 und wiederhole den Schritt,
bis i=m
(ist der laengste Prefix v[i...j] = $, so erhoehe i um 1 und versuche den naechsten Prefix)
9\. loest 8.: trivial
4\. loest 3.: trivial
9\. loest 4.: waehle w:=v

Als Beispiel fuer die Anwendung des Prefix-Trees soll ein Algorithmus fuer Problem 8. vorgestellt werden:

Sei der Prefix-Tree fuer w bereits konstruiert.
Fuer jede Position i wird der laengste Prefix bestimmt und dessen Startposition in prefstart[i] sowie dessen Laenge in preflength[i] abgespeichert.
pos(nc) liefere dabei die Position, mit der der Knoten nc markiert ist (ist nc kein Blatt, so sei nc mit einer der Positionen seiner Nachkommen markiert),
depth(nc) liefere die Tiefe des Knotens nc im Baum.

```
nc := root;
j := k := 1;
FOR  i := 1  TO  m
REPEAT /* bestimme den laengsten Prefix von  v[i...m], der in w vorkommt */
       /* es gilt   label(nc) = v[i...j-1]                               */
       /* und       v[i...k-1] kommt in w vor
       WHILE  son(nc,v[j]) ≠ NIL   AND   j <= m
       REPEAT nc := son(nc,v[j]);
              j := j+1
       ENDREPEAT;
       IF   j > k
       THEN k := j
       ENDIF;
```

```
/* (x1): */
      IF   leaf(nc)
      THEN /* v[i...j-1] ist Posid in w, suche mit k in w selbst weiter */
           WHILE v[k] = w[pos(nc)-i+k]   AND   k <= m
           REPEAT k := k+1
           ENDREPEAT
      ENDIF;
/* (x2): */
      prefstart[i] := pos(nc);
      preflength[i] := k-i;
      IF   nc ≠ root
      THEN nc := sfather(nc)
      ENDIF
ENDREPEAT;
```

Korrektheit:
Vor dem ersten Eintritt in die FOR-Schleife gilt offensichtlich die Invariante: label(nc) = v[i...j-1] und v[i...k-1] kommt in w vor.
Gilt die Invariante vor dem i.ten Eintritt, so gilt an der Stelle (x1) ebenfalls label(nc) = v[i...j-1] ,
und bei (x2) gilt: v[i...k-1] kommt in w vor, v[i...k] aber nicht,
also ist v[i...k-1] der gesuchte laengste Prefix, und seine Daten werden in prefstart[i] und preflength[i] eingetragen.
(Wenn nc kein Blatt ist, gilt j = k .)
Ist nc=root, so ist der laengste Prefix v[i...k-1] = $ und die Invariante gilt auch fuer i+1,
andernfalls kommt mit v[i...k-1] auch v[i+1...k-1] in w vor und der laengste Prefix von v[i+1...m] muss mindestens v[i+1...k-1] enthalten. Durch Uebergang zum sfather-Knoten von nc gilt die Invariante auch in diesem Fall auch fuer i+1.
Nach Ablauf der aeusseren FOR-Schleife enthalten prefstart und preflength also die Daten der laengsten Prefixe fuer jede Position von v.

Aufwand:
Der Aufwand fuer die Konstruktion des Prefix-Trees [1] betraegt O(n), der Aufwand fuer die Suche darin betraegt O(m), denn in jedem Schritt wird j oder k um 1 erhoeht und beide koennen hoechstens den Wert m annehmen.
Die Verwendung der sfather-Zeiger auch fuer die Suche ist der wesentlichste Grund fuer die erreichte Schnelligkeit.

Da die von mir entwickelten Algorithmen zur Konstruktion des Prefix-Trees online arbeiten, ist es erstmals moeglich, schon waehrend der Eingabe von w den Algorithmus zu Problem 8 ablaufen zu lassen. Damit sind die Probleme 1, 2, 5, 6, 7, 8 online loesbar.

----
[1] Der Algorithmus fuer Problem 8 wurde der Uebersichtlichkeit halber fuer einen nicht-kompaktifizierten Prefix-Tree geschrieben. Ihn fuer einen kompaktifizierten Prefix-Tree umzuschreiben, ist lediglich ein - einfaches - technisches Problem.

Geht man durch den Prefix-Tree, nachdem er konstruiert wurde, und traegt in jeden Knoten noch zusaetzlich die Anzahl der Blaetter ein, die er als Nachkommen hat (das geht mit einem Zeitaufwand O(n), allerdings nicht mehr online), so laesst sich der Such-Algorithmus leicht modifiziert zur Loesung des Problems 9. verwenden, indem nur solange der Weg von v[i...m] im Baum verfolgt wird, wie die Knoten mit einem Wert =k (oder einem Wert >=k) markiert sind.

Wegen des hohen Speicherplatz- [1] und Zeit- [2] Aufwandes wird man keinen Prefix-Tree-Algorithmus fuer ein Problem benutzen, fuer das auch ein anderer Algorithmus existiert (z.B. fuer 1, 2: Aho-Corasick-Algorithmus, fuer 3: Rosenberg-Algorithmus, falls moeglich, vgl. Kap. 1.3).

Zur Loesung des Matching-Problems wird man einen Prefix-Tree-Algorithmus nur dann einsetzen, wenn eine Vorverarbeitung fuer w geboten ist (vgl. Kap 1.2), z.B. zum Suchen in einem Datenbestand, der sich praktisch nie aendert.

Bei McCreight finden sich auch Ueberlegungen, wie bei einer nachtraeglichen Aenderung des Eingabestrings der Prefix-Tree entsprechend geaendert werden kann. Er zeigt, dass kleine Aenderungen des Eingabestrings auch nur kleine Aenderungen des Trees erforderlich machen [3] und gibt an, wie die Aenderungen ausgefuehrt werden muessen.
Damit laesst sich ein Prefix-Tree-Algorithmus als Kern eines Editors vorstellen, der gegenueber den herkoemmlichen Editoren den Vorteil eines extrem schnellen Such-Kommandos haette.

Die typischen Anwendungen von Prefix-Tree-Algorithmen liegen eher bei den Problemen 5, 6, 7, 8 (Knuth sagt z.B., dass er ausser dem Weiner-Algorithmus keinen kenne, der Problem 7 in der Zeit O(|v|+|w|) loest (KNU 77)).

----

[1] der gesamte Prefix-Tree fuer w# muss im Speicher bereitgehalten werden, waehrend z.B. beim Knuth-Morris-Pratt-Algorithmus jeweils nur ein Zeichen von w im Speicher sein muss

[2] Auch wenn es mit Hilfe des Aufwandsklassenkalkuels nicht moeglich ist, den Aufwand zur Erstellung eines kompaktifizierten Prefix-Bi-Trees vom Knuth-Morris-Pratt-Algorithmus' zu unterscheiden (beide Male Aufwand des Knuth-Morris-Pratt-Algorithmus' zu unterscheiden (beide Male O(|w|)), ist doch offensichtlich, dass die Zeit fuer einen elementaren Schritt im ersten Fall ungleich groesser ist als im zweiten.

[3] Mit Hilfe der Minid-Theorie ist das unmittelbar anschaulich: alle Minids u ausserhalb des geaenderten Bereichs bleiben erhalten, ausser wenn
- ein echter Teilstring v von u im geaenderten Bereich vorkam und jetzt nur noch in u vorkommt (statt u ist dann das kleinste v mit dieser Eigenschaft Minid), oder
- im geaenderten Bereich jetzt u nochmals vorkommt.

Damit bleiben auch fast alle Posids erhalten.

Folgendes Beispiel soll andeuten, was mit Hilfe eines Prefix-Trees moeglich ist - wenn auch klar ist, dass das Beispiel sehr stark vereinfachend ist und keineswegs fuer alle Probleme, die sich bei automatischer Fehlerkorrektur stellen, eine Loesung skizziert.

Skizze eines Programms zur automatischen Fehlerkorrektur:

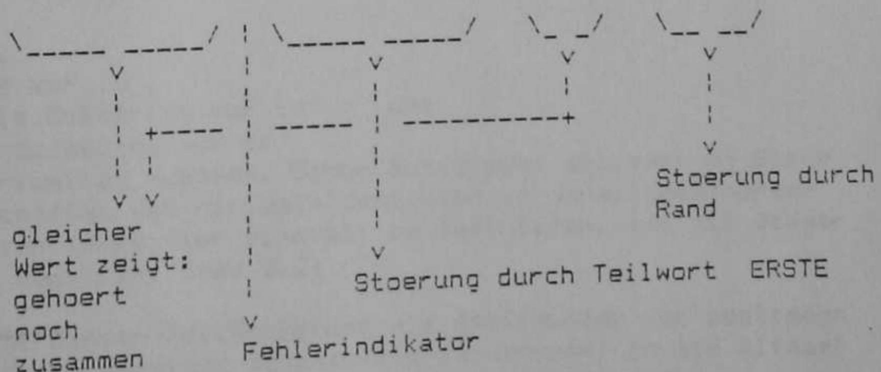
Von dem Eingabestring w wird waehrend des Einlesens Schritt fuer Schritt der Prefix-Tree aufgebaut und fuer jedes neue Wort wird gesucht, ob es schon im Prefix-Tree (und damit schon frueher in w) vorkam. Kam es noch nicht vor, wohl aber ein sehr aehnliches Wort, so liegt es nahe, dass es sich um einen (Tipp-)Fehler handelt (besonders wenn w sehr lang ist und damit die Wahrscheinlichkeit sehr gross ist, dass das Wort schon vorgekommen sein muesste).
Mit Hilfe des Algorithmus zu Problem 8. lassen sich in der Zeit O(m), wenn das zu untersuchende Wort v=v[1...m] ist, eine Reihe von Informationen gewinnen, die zur Fehlererkennung und -korrektur nuetzlich sind.

Beispiel:
in w kommen schon die Worte ERSTE FEHLERQUELLE vor.

```
ERSTE FEHLERQUELLE
123456789012345678
          11111111
```

eingelesen wurde das Wort FEHLERSTELLE , das in w noch nicht vorkomme und hier als Beispiel fuer einen Tippfehler benutzt wird.
Betrachte die folgenden Zahlenwerte:

i	1	2	3	4	5	6	7	8	9	10	11	12
v[i]	F	E	H	L	E	R	S	T	E	L	L	E
preflength[i]	6	5	4	3	5	4	3	2	4	3	2	1
prefstart[i]	7	8	9	10	1	2	3	4	15	16	10	1
preflength[i] + i	7	7	7	7	10	10	10	10	13	13	13	13
prefstart[i] - i	6	6	6	6	4	4	4	4	6	6	-1	-11



Wert gibt die Position des ersten Zeichens des Fehlers an (ebenso die 13 fuer i=9...12)

In diesem Beispiel laesst sich also der Fehler erkennen und das richtige Wort rekonstruieren:
das erste fehlerhafte Zeichen steht an Position 7.
die an Position 1 bis 4 und wieder bei 9 bis 10 beginnenden maximalen Prefixe verweisen alle auf dasselbe (in w vorkommende) Wort "FEHLERQUELLE", das auch an Position 9 und 10 noch mit v uebereinstimmt.

```
****************************************
*  TEIL 2     Der mehrdimensionale Fall  *
****************************************
```

KAPITEL 2.1 Einleitung

Musteridentifizierungsprobleme treten nicht nur bei eindimensionalen Strings auf.
Z.B. tritt das File-Transmission-Problem (1.9.5) in analoger Weise bei einer Datenreduktion bei der Uebertragung von (Fernseh-)Bildern auf;
bei der Optimierung von Computerprogrammen tritt das Repetition-Finding-Problem fuer Trees auf: naemlich beim Finden gemeinsamer Ausdruecke, die nur einmal berechnet werden brauchen:
bei Expertensystemen wird ein Pattern-Matching-Algorithmus fuer Graphen benoetigt, um festzustellen, welche Regeln sich im augenblicklichen Zustand anwenden lassen.

Die Frage stellt sich, ob es in solchen Faellen eine Entsprechung der Prefix-Tree-Algorithmen gibt. Das wird sicherlich auch von der Struktur der Eingabedaten (rechteckige Arrays, Trees, usw.) abhaengen. Um einen aehnlich schnellen Algorithmus wie in 1.5.8+9 und 1.8 zu entwickeln, ist es also nicht nur notwendig, die einzelnen Elemente der eindimensionalen Prefix-Tree-Algorithmen zu verallgemeinern, sondern man muss parallel dazu auch die Anforderungen an die Struktur der Eingabedaten untersuchen, die ein solcher Algorithmus stellt. Dies soll im zweiten Teil dieser Arbeit geschehen.

Das Ziel ist es, ausgehend von den Prefix-Tree-Algorithmen im eindimensionalen Fall einen Algorithmus zu formulieren, der nicht mehr an lineare Strings gebunden ist, sondern moeglichst allgemeine "mehrdimensionale Zeichenfelder" zu einer mit dem Prefix-Tree vergleichbaren Datenstruktur verarbeiten kann. Dazu gehoert insbesondere die Definition der Anforderungen an ein solches mehrdimensionales Zeichenfeld.

Hierbei wird zunaechst ausgegangen von den nicht naeher spezifizierten Grundrelationen
   "ist Substring von",
   "kommt als Substring vor in",
   "kommt nur einmal als Substring vor in"   und
   "kommt mehrmals als Substring vor in"
die bestimmte Axiome erfuellen muessen. Schon auf dieser allgemeinen Ebene lassen sich die Eigenschaften der Minimal-Identifier in verallgemeinerter Form nachweisen. Position-Identifier sinnvoll zu definieren, ist auf dieser Ebene allerdings nicht moeglich. (Kap 2.2)

In einem zweiten Schritt werden Zeichenfelder als Abbildungen von bestimmen Mengen von "Positionen" (zulaessigen Zeichenfeld-Strukturen) in ein Alfabet definiert (wie im eindimensionalen Fall die Schreibweise mit den Positionsnummern unterhalb eines Strings schon sehr suggestiv darstellte). Die obengenannten Relationen werden als Relationen zwischen solchen Abbildungen definiert. Es wird nachgewiesen, dass die Relationen die im ersten Schritt formulierten Axiome erfuellen. (Kap 2.3)

Ein zentrales Problem im mehrdimensionalen Fall ist, dass sich viele Konstruktionen aus dem eindimensionalen Fall nicht oder nicht eindeutig durchfuehren lassen. Schon im einfachsten zweidimensionalen Fall, wenn man beliebige Rechtecke von Zeichen als Zeichenfelder zulaesst, gibt es i.a. keine eindeutigen Position-Identifier und keine eindeutigen maximalen gemeinsamen Prefixe zweier Zeichenfelder mehr. Anstelle der Posids muss also eine andere Konstruktion gefunden werden, auf der die von einem Algorithmus aufzubauende Datenstruktur [1] basiert. (Kap 2.4)

Weiter ist es nicht moeglich, fuer diese Datenstruktur weiterhin mit einem Baum, der fuer seine Blaetter eine totale Ordnungsrelation definiert, auszukommen. Man kann nicht erwarten, z.B. zweidimensionale Zeichenfelder linear sortieren zu koennen und darin die Struktur der Zeichenfelder wiederzufinden. Die hier verwendete Ordnungsstruktur setzt sich zusammen aus einer Komponente, in die nur die Zeichenfeld-Struktur eingeht, und einer, in die nur die "Belegung" der Strukturelemente (der einzelnen Positionen also) mit Zeichen des Alfabets eingeht. (Kap 2.5)

Die angestrebte Verwendung der sfather-Zeiger, die im eindimensionalen Fall wesentlicher Grund fuer die Effizienz der Prefix-Tree-Algorithmen war, macht es im mehrdimensionalen Fall notwendig, eine Art Rangfolge unter den einzelnen Positionen einer Zeichenfeld-Struktur zu definieren. Ist eine Position bzgl. dieser Rangfolge unmittelbar vor einer anderen, so ist es sinnvoll, einen sfather-Zeiger zwischen den Knoten der beiden Positionen anzulegen. (Kap 2.6)

Die Eigenschaften, die im eindimensionalen Fall zur Definition der Bloecke gefuehrt haben, finden sich im wesentlichen auch im mehrdimensionalen Fall wieder. (Kap 2.7)

Schliesslich werden weitere verschaerfende Anforderungen an Zeichenfelder eingefuehrt und untersucht:

So wird definiert, was es heissen soll, ein Zeichenfeld in eine bestimmte Richtung auszudehnen. Im eindimensionalen Fall laesst sich ein Zeichenfeld nur in eine Richtung ausdehnen (wenn der Startpunkt beibehalten wird), im Rechteck-Fall in zwei (nach rechts und nach unten), usw. Dieses Konzept, das eine ganze Reihe wichtiger Ergebnisse im Hinblick auf die Verallgemeinerung des Algorithmus liefert, wird in Kapitel 2.8 und 2.9 untersucht.

Zum anderen wird der Endabschnitt eines Zeichenfelds definiert (eine spezielle Art von Suffix) und untersucht. (2.10)

----

[1] die Verallgemeinerung des Prefix-Trees also

KAPITEL 2.2 Minimal-Identifier
================================

Im ersten Teil (Kap. 1.4) waren die Relationen "v ist Substring von w", "v kommt einmal in w vor", "v kommt mehrmals in w vor" zentrale Begriffe. Wieweit sich aus diesen Begriffen, wenn man von allen Eigenschaften der Strings, die sich nicht damit darstellen lassen, abstrahiert, schon brauchbare Saetze ableiten kann, soll in diesem Kapitel untersucht werden. Dabei wird sich zeigen, dass man damit immerhin die Saetze ueber die Minimal-Identifer aus dem ersten Teil (1.4.11,12,14) verallgemeinern kann.

1. Def: Gegeben sei eine Menge ZF von Objekten ("Zeichenfeldern") mit den Relationen (fuer v, w ∈ ZF):
   v kommt in w vor (kurz: v in w),
   v kommt mehrmals in w vor (kurz: v mehrmals in w, v min w)
   mit den folgenden Eigenschaften:
   (a) "in" ist reflexive partielle Ordnung,
   (b) v mehrmals in w ==> v in w
   (c) "mehrmals in" ist irreflexiv
   (d) u mehrmals in v, v in w ==> u mehrmals in w
   (e) u in v, v mehrmals in w ==> u mehrmals in w
   (f) zu jedem w ∈ ZF gibt es hoechstens endlich viele v mit v in w

2. Def: v kommt nur einmal in w vor (kurz: v einmal in w, v ein w)
   :<==> v in w, v nicht mehrmals in w

   Bem: statt v in w, v ≠ w schreibt man kurz v xin w (v echt in w), analog statt v ein w, v ≠ w kurz v xein w (v einmal echt in w)

   Bei allen im Lauf der Arbeit definierten Relationen wird entsprechend verfahren: Voranstellen eines "x" vor den Namen der Relation schliesst die Diagonale aus.

3. Lem: (a) u in v, u einmal in w ==> v einmal in w
   (b) "mehrmals in" ist irreflexive partielle Ordnung
   (c) v einmal in w ==> v in w

   Bew: (a) nach 1e waere sonst u mehrmals in w
   (b) irreflexiv: nach 1c ist u einmal in u, also nach 2 u nicht mehrmals in u
   transitiv: u mehrmals in v, v mehrmals in w
   ==> u in v, v mehrmals in w (1b)
   ==> u mehrmals in w (1e)
   asymmetrisch: u mehrmals in v, v mehrmals in u
   ==> u mehrmals in u (da transitiv)
   W.! zur Irreflexivitaet
   (c) nach 2

4\. Def: Sei v,w ∈ ZF,
v heisst Minimal-Identifier von w (kurz: Minid)
:<==> v einmal in w,
fa. u xin v ist u mehrmals in w
(d.h. v ist ein minimales Element (bzgl. "in") mit der Eigenschaft, nur einmal in w vorzukommen)

M[w] := {u | u ist Minimal-Identifier von w}
M[w,v] := {u | u ist Minimal-Identifier von w, u in v}

5\. Satz: (Existenz von Minimalidentifiern)
Sei v,w ∈ ZF, v einmal in w,
dann ex. ein Minimal-Identifier u von w mit u in v

Bew: nach 0.3.3.4 mit M0 := {x | x in w} endlich, P(x) :<==> x ein w

6\. Korr: M[w] ≠ {} fa. w ∈ ZF

Bew: nach 5., da w einmal in w

7\. Lem: (a) M[w,v] ⊂ M[w] fa. v in w
(b) M[w] ist endliche Menge

Bew: (a) unmittelbar nach Def 4.
(b) da v ∈ M[w] ==> v in w (und mit 1f)

8\. Satz: Sei v,w ∈ ZF, v in w, dann gilt
v einmal in w <==> M[w,v] ≠ {}

Bew: "<==": u ∈ M[w,v] ==> u einmal in w, u in v
==> v einmal in w (3a)

"==>": nach 5.

9\. Korr: Sei v,w ∈ ZF, v in w, dann gilt
v einmal in w <==> v enthaelt einen Minimal-Identifier von w

Bew: folgt unmittelbar aus Satz 8. und Def 4.

10\. Lem: u in v ==> M[w,u] ⊂ M[w,v]

Bew: x ∈ M[w,u] ==> x in u, x Minimal-Identifier von w
==> x in v, x Minimal-Identifier von w
==> x ∈ M[w,v]

11\. Lem: fa. u,v ∈ M[w] gilt u nicht xin v

Bew: sonst folgt nach Def von v:
u mehrmals in w, W.! zur Def von u

12. Lem: $u \in M[w]$, u in v $\Longrightarrow$ $u \in M[w,v]$

Bew: klar nach Def 4.

13. Satz: (Charakterisierung der Minimal-Identifier)
Sei $w \in ZF$, $MO \subset ZF$, dann gilt
$MO = M[w]$
$\Longleftrightarrow$
MO erfuellt folgende Eigenschaften:
(a) fa. $v \in MO$ ist v in w
(b) fa. $u,v \in MO$ ist u nicht xin v
(c) fa. v in w ist
(v einmal in w $\Longleftrightarrow$ ex. $u \in MO$ mit u in v)

Bew: "$\Longrightarrow$": (a) folgt nach Def 4.,
(b) folgt nach Lemma 11.
(c) folgt nach Korr 9.

"$\Longleftarrow$": $MO \subset M[w]$:
sei $v \in MO$, nach (c) ist v einmal in w (mit $u:=v$),
wuerde ein u xin v ex. mit u einmal in w,
so ex. nach (c) ein $v' \in MO$ mit v' in u
also v' in u xin v, W.! zu (b)
$M[w] \subset MO$:
sei $v \in M[w]$, d.h. v einmal in w,
nach (c) ex. $u \in MO$ mit u in v,
waere $u \neq v$, so folgte nach Def von v: u mehrmals in w,
W.!, also $v = u \in MO$

KAPITEL 2.3 Zeichenfelder mit Positionen

Es ist offensichtlich, dass man nicht auf der Abstraktionsebene von Kapitel 2.2 bleiben kann, will man Eigenschaften von Zeichenfeldern untersuchen, die eine Verallgemeinerung des Prefix-Tree-Algorithmus ermoeglichen. Man braucht zumindest den Begriff der Position, den des Zeichens, das an einer Position steht, sowie eine Art von Orientierung, damit Begriffe wie Prefix und Suffix (ebenfalls zentral im ersten Teil) definiert werden koennen. In diesem Kapitel werden diese Begriffe definiert und es wird gezeigt, dass sie eine Konkretisierung der in 2.2 gegebenen Definition eines Zeichenfeldes sind (Satz 7).

1. Def: Gegeben sei eine Menge P von Positionen,
eine Menge ZS von endlichen Teilmengen von P
(ZS heisst die Menge der zulaessigen Zeichenfeld-Strukturen),
und ein abzaehlbares Alfabet A,
$ZF := \{f \mid \text{ex. } Xf \in ZS \text{ mit } f: Xf \to A\}$ heisst Menge der zulaessigen Zeichenfelder,
$f \in ZF$ heisst Zeichenfeld, Xf heisst zugehoerige Zeichenfeld-Struktur (kurz: Struktur von f),
ein $p \in Xf$ heisst Position von f (oder auch von Xf).
Jedes Zeichenfeld f versieht also eine entsprechende ZF-Struktur mit Labeln (oder Belegungen) aus dem Alfabet A.
Ist $f \in ZF$, so sei $Xf \in ZS$ die zugehoerige ZF-Struktur,
d.h. $X: ZF \longrightarrow ZS, \quad f \longmapsto dom(f)$

Auf ZS sei ein Aequivalenzrelation $\equiv$ definiert, die nur Mengen von gleicher Maechtigkeit identifiziert
(d.h. $Xf \equiv Xg \Rightarrow |Xf| = |Xg|$).

Auf der Menge aller aequivalenten Paare sei eine Funktion Iso definiert mit Wertebereich in den partiellen Abbildungen von P in sich selbst,
$Iso: \{(Xf,Xg) \mid Xf \equiv Xg\} \to part(P \to P)$
mit folgenden Eigenschaften:

(a) $Iso(Xf,Xg): Xf \to Xg$ ist bijektiv
(b) $Iso(Xf,Xf) = Id(Xf)$ (Identitaet von Xf)
(c) $Iso(Xf,Xg) = Iso(Xg,Xf)^{-1}$ (inverse Abbildung)
(d) $Iso(Xg,Xh) * Iso(Xf,Xg) = Iso(Xf,Xh)$ (Komposition)
(e) $Xg1 \equiv Xg2, \quad Xf1 \subset Xg1, \quad$ sei $Xf2 := Iso(Xg1,Xg2)[Xf1]$
$\Rightarrow Xf2 \in ZS, \quad Xf1 \equiv Xf2,$
$Iso(Xf1,Xf2) = Iso(Xg1,Xg2)|Xf1$
(f) $Xf \equiv Xg, \quad Xe \subset Xf \cap Xg, \quad Iso(Xf,Xg)[Xe] = Xe$
$\Rightarrow Xf = Xg$
(diese Bedingung ist hinreichend fuer 2.2.1e, vgl. Satz 8)

Auf ZS sei eine Funktion st definiert mit Wertebereich in P:
$st: ZS \longrightarrow P, \quad Xf \longmapsto st(Xf) \in Xf$
mit der Eigenschaft:

(g) $Xf \equiv Xg \Rightarrow st(Xg) = Iso(Xf,Xg)(st(Xf))$

st (fuer "Start") liefert zu jeder ZF-Struktur eine ausgezeichete Anfangsposition. (Entspricht im eindimensionalen Fall der Anfangsposition eines Strings.)

Bsp: (a) Sei P = Z^n die Menge aller n-Tupel von ganzen Zahlen,
P ist abelsche Gruppe bzgl. der punktweisen Addition,
auf P sei die lexikografische Ordnung definiert
(d.h. (p1,...,pn) < (q1,...,qn) <==>
pi = qi fuer 1 <= i <= j-1, pj < qj)
sei min Xf das Minimum der Menge Xf bzgl. dieser Ordnung,
sei A beliebig,
P' sei die Menge der endlichen Teilmengen von P,
sei ZS c P' beliebig mit der Eigenschaft
(x) Xf = {p1,...,pm} ∈ ZS, p ∈ P
==> Xf+p := {p1+p,...,pm+p} ∈ ZS
definiere Xf ≡ Xg :<==> ex. p ∈ P mit Xg = Xf+p
(p ist eindeutig dann bestimmt durch p = (min Xg) - (min Xf))
und Iso(Xf,Xg)(q) := q+p

erfuellt trivialerweise Def 1a-d
erfuellt e: sei Xg2 = Xg1+p, dann ist
Xf2 := Iso(Xg1,Xg2)[Xf1] = Xf1+p ≡ Xf1, Xf2 ∈ ZS,
Iso(Xf1,Xf2)(q) = q+p = Iso(Xg1,Xg2)(q)
erfuellt f: sei Xg = Xf+p, Xe = Iso(Xf,Xg)[Xe] = Xe+p,
==> min Xe = min (Xe+p) = (min Xe) + p
==> p = (0,...,0)
==> Xf = Xg
erfuellt g: definiere st(Xf) := min Xf,
ist dann Xg = Xf+p, so folgt
st(Xg) = min Xg = (min Xf) + p = Iso(Xf,Xg)(st(Xf))

D.h. man kann eine beliebige Menge von n-dimensionalen "Figuren" vorgeben, die beliebig verschoben werden duerfen, und hat damit eine Menge von Zeichenfeld-Strukturen.

Insbesondere fuer n=2:

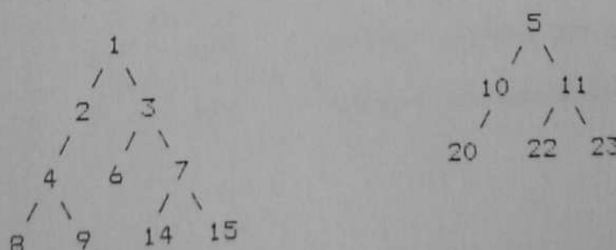
(b) ZS := {[a,b]x[c,d] | a,b,c,d ∈ Z} die Menge der Rechtecke,
Startpunkte sind die linken oberen Ecken,

(c) beliebige Teilmengen von (b) mit der Eigenschaft (x),
das ist insbesondere wichtig fuer Gegenbeispiele, z.B. in 2.5.9, und wird dementsprechend in dieser Arbeit meist dafuer benutzt, zumal es sich auch leicht geometrisch veranschaulichen laesst,

(d) beliebige Vereinigungen von Rechtecken mit gleicher linker oberer Ecke,

Ein weiteres wichtiges Beispiel passt nicht in den Rahmen von (a):

(e) Sei P := N \ {0} die Menge der positiven natuerlichen Zahlen,
Xf c P stellt einen binaeren Baum dar
:<==> fa. a ∈ Xf mit a > min Xf ist (a//2) ∈ Xf
wobei a//2 den ganzzahligen Anteil der Division von a durch 2 darstellt,
z.B. {1,2,3,4,6,7,8,9,14,15}, {5,10,11,20,22,23}



i kann 2*i und 2*i+1 als Soehne haben,
setze st(Xf) := min Xf,
ZS := {Xf | Xf c P, Xf endlich, Xf binaerer Baum}
Iso(Xf0,Xf1)(p) := p - $p0*2^{d(p)}$ + $p1*2^{d(p)}$
mit p0 = st(Xf0), p1 = st(Xf1),
d(p) := [lb(p)] (ganzzahliger Anteil des Logarithmus von p zur Basis 2; entspricht der Tiefe von p im Baum)
und Xf1 ≡ Xf2 :<==> Iso(Xf1,Xf2) ist bijektive Abbildung von Xf1 auf Xf2
erfuellt ebenfalls Def 1a-g

(f) Strings mit Luecken
faellt noch unter (a)
mit n = 1, ZS besteht aus allen endlichen Teilmengen von Z
Als Anwendung kommen Matching-Probleme mit "Don't care Symbols" in Frage, fuer die es bislang noch keine befriedigend schnellen Loesungsalgorithmen gibt (z.B. FIS 74-1, FIS 74-2, HIR 75).

2. Def: Sei f,g ∈ ZF,
f ≡ g :<==> Xf ≡ Xg und f = g*Iso(Xf,Xg)

3. Lem: Damit laesst sich die Aequivalenzrelation auf ZS zu einer auf ZF hochziehen, d.h. ≡ auf ZF ist Aequivalenzrelation.

Bew: Reflexivitaet, Symmetrie und Transitivitaet folgen aus den entsprechenden Eigenschaften von ≡ auf ZS und aus 1.b-d, exemplarisch die Transitivitaet:
f ≡ g, g ≡ h ==> Xf ≡ Xg ≡ Xh,
f = g * Iso(Xf,Xg)
= h * Iso(Xg,Xh) * Iso(Xf,Xg)
= h * Iso(Xf,Xh) (1.d)
==> f ≡ h

4. Def: Sei f,g ∈ ZF, dann heisst
f direkt in g (kurz: f din g)
:<==> Xf c Xg, f = g|Xf
f einmal direkt in g (kurz: f dein g)
:<==> Xf c Xg, f = g|Xf,
und fa. h ∈ ZF mit h ≡ f, Xh c Xg, h = g|Xh
ist schon Xh = Xf (gleich, nicht nur aequivalent)
f direkt linksbuendig in g (kurz: f dlin g)
:<==> Xf c Xg, f = g|Xf, st(Xf) = st(Xg)
(d.h. f direkt in g und st(Xf)=st(Xg))
f in g
:<==> ex. f' ≡ f mit f' direkt in g
f einmal in g (kurz: f ein g)
:<==> ex. f' ≡ f mit f' einmal direkt in g
f linksbuendig in g (kurz: f lin g)
:<==> ex. f' ≡ f mit f' direkt linksbuendig in g

```
f mehrmals in g   (kurz: f min g)
     :<==>   ex.   f',f" ∈ ZF   mit   f ≡ f' ≡ f",
             Xf',Xf" ⊂ Xg,   f' = g|Xf',   f" = g|Xf",
             aber   Xf' ≢ Xf"
             (obwohl natuerlich Xf' ≡ Xf" nach Def 2.)

Xf direkt in Xg   (kurz: Xf din Xg)
     :<==>   Xf ⊂ Xg
Xf direkt linksbuendig in Xg   (kurz: Xf dlin Xg)
     :<==>   Xf ⊂ Xg,   st(Xf) = st(Xg)
```

```
Bem:  f dlin g    ==>    Xf dlin Xg,
      f din g     ==>    Xf din Xg
      f dein g   <==>    f din g,    fa. h mit   h ≡ f,   h din g
                                     ist   Xh = Xf
      f min g    <==>    ex.   f',f" ∈ ZS   mit   f ≡ f' ≡ f",
                                     f',f" din g,   Xf' ≢ Xf"
```

Bsp: Als Zeichenfelder seien beliebige Rechtecke zulaessig, sei

```
           1 2 3 4            1 2 3            2 3 4           3
     f =   +-------     g =   +-----     h =   +-----     i =  +-
          1|a b c a          1|a b c          2|a b c         2|b
          2|a a b c          2|a a b          3|a a b         3|a
          3|c a a b                                           4|c
          4|b c c a
```

```
dann ist   Xf = [1,4]x[1,4],   st(Xf) = (1,1),
           Xg = [1,2]x[1,3],   st(Xg) = (1,1),
           Xh = [2,3]x[2,4],   st(Xh) = (2,2),
           Xi = [2,4]x[3,3],   st(Xi) = (2,3),
und es gelten folgende Beziehungen:
           Xg ≡ Xh,
           g ≡ h,
           g din f,   sogar   g dlin f,
           h din f,   nicht   h dlin f,   wohl aber   h lin f,
           g min f,
           i dein f
```

```
5. Def:  Fuer   f ∈ ZF   sei   F := {g | g ≡ f}   die Aequivalenzklasse
         von  f,
         analog   XF  die Aequivalenzklasse von  Xf,
         sei      _ZF_ := {F | f ∈ ZF},     _ZS_ := {XF | Xf ∈ ZS},
         fuer   F,G ∈ _ZF_   sei
         F IN G   :<==>   f in g,
         F EINMAL IN G   :<==>   f einmal in g,
         F MEHRMALS IN G   :<==>   f mehrmals in g,
         F LINKSBUENDIG IN G   :<==>   f linksbuendig in g
```

6. Satz: Die Definitionen der Relationen IN, EINMAL IN, MEHRMALS IN und LINKSBUENDIG IN auf _ZF_ sind repraesentantenunabhaengig.

```
Bew:  IN:
      sei    f1 in g1,    f1 ≡ f2,    g1 ≡ g2,
      ==>    ex.   f1' ≡ f1    mit    Xf1' c Xg1,    f1' = g1|Xf1'          (4)
      ==>    mit   Xf2' := Iso(Xg1,Xg2)[Xf1']    ist
             Xf2' ∊ ZS,        Xf2' c Xg2,       Xf2' ≡ Xf1',
             Iso(Xf1',Xf2') = Iso(Xg1,Xg2)|Xf1
      setze   f2' := g2|Xf2',                                               (1e)
      dann ist   f2' ≡ f1',     denn    Xf2' ≡ Xf1'    und
                 f2' * Iso(Xf1',Xf2')
            =    (g2|Xf2') * Iso(Xf1',Xf2')
            =    (g2 * Iso(Xf1',Xf2')) | Xf1'
            =    (g2 * Iso(Xg1,Xg2)|Xf1') | Xf1'
            =    (g1|Xf1') | Xf1'
            =    g1 | Xf1'
            =    f1'
      ausserdem ist    Xf2' c Xg2    und    f2' = g2|Xf2',
      also (da   f2' ≡ f1' ≡ f1 ≡ f2)    f2 in g2

      LIN:
      ist bei   IN   ausserdem    st(Xf1') = st(Xg1),    so ist
                 st(Xf2')
            =    Iso(Xf1',Xf2')(st(Xf1'))                                   (1g)
            =    Iso(Xg1,Xg2)(st(Xf1'))
            =    Iso(Xg1,Xg2)(st(Xg1))
            =    st(Xg2)

      MEHRMALS IN:
      gilt bei   IN   ausserdem    ex.    f1" ∊ ZF    mit    f1" ≡ f1',
             f1" = g1|Xf1",      Xf1" c Xg1,      Xf1' ≠ Xf1",
      so setze    f2" := g2|Xf2",     wobei  Xf2" := Iso(Xg1,Xg2)[Xf1"]
      dann ist    f1" = f2" * Iso(Xf1",Xf2")        (folgt analog wie bei IN)
      und nach 1.e    Xf2" c Xg2
      weiter ist    Iso(Xf1',Xf2') = Iso(Xg1,Xg2)|Xf1' ,
      also          Iso(Xf2',Xf1') = Iso(Xg2,Xg1)|Xf2' ,
      analog ist    Iso(Xf2",Xf1") = Iso(Xg2,Xg1)|Xf2" ,
      damit muss    Xf2' ≠ Xf2"    sein, denn sonst waere
      Xf1'    =    Iso(Xf2',Xf1')[Xf2']
              =    Iso(Xg2,Xg1)[Xf2']
              =    Iso(Xg2,Xg1)[Xf2"]                (da Xf2' = Xf2" nach Ann)
              =    Iso(Xf2",Xf1")[Xf2"]
              =    Xf1"                              W.!

      EINMAL IN:
      gelte bei   IN   ausserdem    fa. h1 mit    h1 ≡ f1,    Xh1 c Xg1,
             h1 = g1|Xh1      ist    Xh1 = Xf1',
      sei   h2   gegeben mit    h2 ≡ f2,    Xh2 c Xg2,    h2 = g2|Xh2,
      zeige:    Xh2 = Xf2'
      setze    Xh1 := Iso(Xg2,Xg1)[Xh2] ,     nach 1.e ist    Xh1 ∊ ZS   und
             Xh1 c Xg1,     Xh2 ≡ Xh1,     Iso(Xh1,Xh2) = Iso(Xg1,Xg2)|Xh1,
      setze   h1 := g1|Xh1,
      dann ist    h1 ≡ f1,
             denn    Xh1 ≡ Xh2 ≡ Xf2                    (folgt wie bei   IN)
             und     h2 * Iso(Xh1,Xh2) = h1
```

```
da ausserdem   Xh1 c Xg1   und   h1 = g1|Xh1,   folgt   Xh1 = Xf1',
also auch
Xh2   =   Iso(Xh1,Xh2)[Xh1]
      =   Iso(Xg1,Xg2)[Xh1]
      =   Iso(Xg1,Xg2)[Xf1']
      =   Iso(Xf1',Xf2')[Xf1']
      =   Xf2'
```

Bem: Satz 6 legt es an sich nahe, ab jetzt nur noch in Aequivalenzklassen von Zeichenfeldern zu rechnen. Dies ist jedoch nicht immer moeglich, naemlich dann nicht, wenn die relative Lage von Zeichenfeldern zueinander fixiert werden soll (z.B. Def 2.4.3). Daher wird im folgenden weiter auf der "unteren" Ebene gerechnet, d.h. mit "in", "lin" usw. und mit "din", "dlin" usw.
Die Ergebnisse lassen sich dann mit Satz 6 sofort auf die Ebene der Aequivalenzklassen "hochziehen".

7. Satz: Die Menge _ZF_ und die Relationen IN, MEHRMALS IN und EINMAL IN erfuellen die Eigenschaften aus Def 2.2.1+2

Bew: 2.2.2 nach Def von mehrmals in und Repraesentantenunabhaengigkeit

```
(a)   IN  ist reflexiv:
      f in f    ==>   F IN F
      antisymmetrisch:
      F IN G,   G IN F   ==>   f in g,   g in f
      ==>   ex. f' ≡ f,   g' ≡ g   mit   Xf' c Xg,    f' = g|Xf',
            Xg' c Xf,   g' = f|Xg'
      ==>  (wegen f'≡f ==> Xf'≡Xf ==> |Xf'|=|Xf| , analog fuer g)
            |Xf'| <= |Xg| = |Xg'| <= |Xf| = |Xf'| ,
      ==>   |Xf'| = |Xg| = |Xg'| = |Xf|
      ==>   Xf' = Xg,     (da   Xf' c Xg,   |Xf'| = |Xg|)
            f' = g|Xf' = g|Xg = g
      ==>   f ≡ f' = g
      ==>   F = G
      transitiv:
      F IN G,   G IN H   ==>   f in g,   g in h
      ==>   ex. f' ≡ f,   g' ≡ g   mit
            Xf' c Xg,   f' = g|Xf',   Xg' c Xh,   g' = h|Xg'
      nach 1.e ist   Xf" := Iso(Xg,Xg')[Xf']   ∈ ZS ,   und
         Xf" c Xg',   Xf" ≡ Xf',   Iso(Xf',Xf") = Iso(Xg,Xg')|Xf'
      setze   f" := g'|Xf",   dann ist   f" ≡ f',   denn
              f" * Iso(Xf',Xf")
           =  f" * Iso(Xg,Xg')|Xf'
           =  (g'|Xf") * Iso(Xg,Xg')|Xf'
           =  (g' * Iso(Xg,Xg')) | Xf'
           =  g | Xf'
           =  f'
      weiter ist   Xf" c Xg' c Xh
      und   f" = g'|Xf' = (h|Xg')|Xf' = h|Xf"
      also (da   f" ≡ f' ≡ f):   f in h

(b)   MEHRMALS IN  c  IN :   klar nach Def
```

(c) Wegen Def 2.2.2 genuegt es zu zeigen, dass EINMAL IN reflexiv ist.
F EINMAL IN F, da f einmal in f:
mit f' := f ist f' ≡ f, Xf' c Xf, f' = f|Xf',
und fa. h mit h ≡ f, Xh c Xf, h = f|Xh folgt:
h ≡ f ==> Xh ≡ Xf ==> |Xh|=|Xf| ==> Xh = Xf = Xf'

(d) F MEHRMALS IN G, G IN H ==> F MEHRMALS IN H
f min g ==> ex. f1,f2 mit f1 ≡ f2 ≡ f, f1,f2 din g,
Xf1 ≠ Xf2,
g in h ==> ex. g' ≡ g mit g' din h,
setze Xf1' := Iso(Xg,Xg')[Xf1], f1' := h|Xf1',
nach 1e ist Xf1' ∈ ZS, Xf1' ≡ Xf1,
also f1' ≡ f1, da
f1' * Iso(Xf1,Xf1')
= (h|Xf1') * Iso(Xf1,Xf1')
= (g'|Xf1') * Iso(Xf1,Xf1')
= g|Xf1
= f1,
und es ist f1' din h,
analog setze Xf2' := Iso(Xg,Xg')[Xf2], f2' := h|Xf2',
dann ist genauso f2' ≡ f2, f2' din h,
schliesslich ist
Xf1' = Iso(Xg,Xg')[Xf1] ≠ Iso(Xg,Xg')[Xf2] = Xf2'
(denn Xf1 ≠ Xf2 und Iso(...) ist bijektiv),
also f min h

(e) F IN G, G MEHRMALS IN H ==> F MEHRMALS IN H
f in g ==> ex. f' ≡ f mit Xf' c Xg, f' = g|Xf'
g min h ==> ex. g1' ≡ g2' ≡ g mit Xg1',Xg2' c Xh,
g1' = h|Xg1', g2' = h|Xg2', Xg1' ≠ Xg2'
nach 1.e ist Xf1' := Iso(Xg,Xg1')[Xf'] ∈ ZS und
Xf1' ≡ Xf', Iso(Xf',Xf1') = Iso(Xg,Xg1')|Xf'
analog ist Xf2' := Iso(Xg,Xg2')[Xf'] ∈ ZS und
Xf2' ≡ Xf', Iso(Xf',Xf2') = Iso(Xg,Xg2')|Xf'
setze f1' := g1'|Xf1', f2' := g2'|Xf2',
dann ist f1' ≡ f', da Xf1' ≡ Xf' und
f1' * Iso(Xf',Xf1')
= (g1'|Xf1') * Iso(Xg,Xg1')|Xf'
= g|Xf'
= f'
weiter ist Xf1' c Xg1' c Xh
und f1' = g1'|Xf1' = (h|Xg1')|Xf1' = h|Xf1'
analog findet man f2' ≡ f', Xf2' c Xh, f2' = h|Xf2'
wegen 1.f folgt schliesslich auch Xf1' ≠ Xf2',
denn sonst waere Xg1' ≡ Xg2', Xf1' = Xf2' c Xg1' n Xg2',
Iso(Xg1',Xg2')[Xf1'] = Xf2' = Xf1'
nach 1.f folgt Xg1' = Xg2', W.!

8. **Satz:** Fuer die Forderung aus 2.2.1e ist folgende Bedingung notwendig und hinreichend (leichte Abschwaechung von 1f):

Xf ≡ Xg,   Xe ⊂ Xf ∩ Xg,   Iso(Xf,Xg)[Xe] = Xe
ex. Xh mit   Xf, Xg ⊂ Xh
==>   Xf = Xg

Bew: hinreichend: im Beweis von 7. (Punkt (e)) ist (mit den dortigen Bezeichnungen)   Xg1', Xg2' ⊂ Xh,   d.h. es kann auch mit der Abschwaechung geschlossen werden

notwendig: wuerde Xg1, Xg2, Xf, Xh existieren mit
Xg1 ≡ Xg2,   Xg1 ≠ Xg2,   Xf ⊂ Xg1 ∩ Xg2,
Xg1, Xg2 ⊂ Xh,   Xf = Iso(Xg1,Xg2)[Xf],
dann liesse sich folgendes Gegenbeispiel konstruieren:

```
+-----------------------------+
| +------------------+      Xh|
| |      "g"      Xg1|     "g"|
| |                  |        |
| |   +--------------|------+ |
| |   |     +-----+  |   Xg2| |
| |   |     |   Xf|  |      | |
| |   |     |     |  |      | |
| |   | "g" |"f"  |  |   "g"| |
| |   |     +-----+  |      | |
| +---|--------------+      | |
|     |                     | |
|     +---------------------+ |
+-----------------------------+
```

setze   h: Xh --> A,
x +--> {"f"   x ∈ Xf
{"g"   sonst
fuer "f", "g" ∈ A

setze   g := h|Xg1,   f := h|Xf,
dann ist nach Def   f direkt in g
und   g mehrmals in h,   denn:
setze   g' := h|Xg2,
wegen   Xf = Iso(Xg2,Xg1)[Xf]
ist   x ∈ Xf   <==>
Iso(Xg2,Xg1)(x) ∈ Xf
fa. x ∈ Xg2,
also   g'(x) = "f"   <==>
g(Iso(Xg2,Xg1)(x)) = "f",
d.h.   g' = g * Iso(Xg2,Xg1),
insgesamt also   g' ≡ g,

aber   f einmal in h, denn   f in h   und:
ist   f' ≡ f   mit   Xf' ⊂ Xh,   f' = h|Xf',
so folgt   f'(x)=f(Iso(Xf',Xf)(x))="f"   fa. x ∈ Xf',
also muss nach Def von h gelten: Xf' ⊂ Xf,
d.h.   Xf' = Xf   (wegen gleicher Maechtigkeit)

KAPITEL 2.4 Position-Identifier

In diesem Kapitel soll die Verallgemeinerung des Position-Identifiers definiert werden. Man kann nicht mehr damit rechnen, dass es zu einer Position einen eindeutig bestimmten Position-Identifier gibt, wie das Beispiel zu Def 8 zeigt. Deshalb wird hier unterschieden zwischen Einzel-Positions-Identifiern (alle minimalen Zeichenfelder, die nur einmal vorkommen) und Gesamt-Positions-Identifiern (minimale Zeichenfelder, die alle Einzel-Posids enthalten). Um die letzteren konstruieren zu koennen und Bedingungen fuer ihre Eindeutigkeit finden zu koennen (Korrollar 12), werden Vereinigungen (und Durchschnitt) von Zeichenfeldern eingefuehrt.

1\. Def: $PZS := \{Xf \mid Xf \subset P,\ Xf \text{ endlich}\}$ heisst Menge aller Pseudo-Zeichenfeld-Strukturen,

$PZF := \{f \mid f: Xf \to A \text{ fuer ein } Xf \in PSZ\}$ heisst Menge aller Pseudo-Zeichenfelder

2\. Bem: Es ist $ZS \subset PZS$ und $ZF \subset PZF$

3\. Def: Sei $f1, f2, g \in ZF$, $f1, f2$ direkt in g, dann heisst
$(f1 \cap f2): (Xf1 \cap Xf2) \to A,\quad x \mapsto f1(x) \quad (=f2(x))$
Durchschnitt von f1 und f2,

$$(f1 \cup f2): (Xf1 \cup Xf2) \to A,\quad x \mapsto \begin{cases} f1(x) & \text{fuer } x \in Xf1 \setminus Xf2 \\ f1(x) \ (= f2(x)) & \text{fuer } x \in Xf1 \cap Xf2 \\ f2(x) & \text{fuer } x \in Xf2 \setminus Xf1 \end{cases}$$

Vereinigung von f1 und f2,

4\. Bem: Vereinigung und Durchschnitt sind i.a. nur Pseudo-Zeichenfelder, es gilt $f1 \cup f2 = f2 \cup f1$ und $f1 \cap f2 = f2 \cap f1$ (nach Def)

Bsp: Seien g, h und i wie im Beispiel zu Def 2.3.4, dann ist

```
              3                   2 3 4                 1 2 3
h ∩ i =      +-      h ∪ i =     +-----      g ∪ i =   +-----
             2|b                 2|a b c               1|a b c
             3|a                 3|a a b               2|a a b
                                 4|  c                 3|    a
                                                       4|    c
```

5. Def: ZS heisst abgeschlossen gegenueber Vereinigung
:<==> die Vereinigung zweier ZF-Strukturen mit nicht-leerem Durchschnitt ist stets wieder eine ZF-Struktur

ZS heisst halb abgeschlossen gegenueber Vereinigung
:<==> die Vereinigung zweier ZF-Strukturen mit nicht-leerem Durchschnitt liegt stets wieder in einer ZF-Struktur
<==> ($X_f, X_g \in ZS, \quad X_f \cap X_g \neq \{\}$
==> ex. $X_h \in ZS$ mit $X_f \cup X_g \subset X_h$)

ZS heisst eindeutig halb abgeschlossen gegenueber Vereinigung
:<==> es gibt stets genau ein minimales X_h in der vorigen Def
<==> zu $X_f, X_g \in ZS$ mit $X_f \cap X_g \neq \{\}$
ex. genau ein $X_h \in ZS$ mit
($X_f \cup X_g \subset X_h$ und fa. $X_m \subset X_h, \quad X_m \neq X_h$,
ist nicht $X_f \cup X_g \subset X_m$)

ZS heisst zusammenhaengend halb abgeschlossen gegenueber Vereinigung (kuerzer: z-halb abgeschlossen gegen Vereinigung)
:<==> die Vereinigung zweier ZF-Strukturen mit nicht-leerem Durchschnitt liegt stets in einer zusammenhanegenden ZF-Struktur (siehe Def 2.6.8)

ZS heisst abgeschlossen gegenueber Durchschnitt
:<==> der Durchschnitt zweier ZF-Strukturen ist stets wieder eine ZF-Struktur oder leer

ZS heisst halb abgeschlossen gegenueber Durchschnitt
:<==> der Durchschnitt zweier ZF-Strukturen enthaelt stets wieder eine ZF-Struktur oder ist leer
<==> ($X_f, X_g \in ZS, \quad X_f \cap X_g \neq \{\}$
==> ex. $X_h \in ZS$ mit $X_h \subset X_f \cap X_g$)

ZS heisst eindeutig halb abgeschlossen gegenueber Durchschnitt
:<==> es gibt stets genau ein maximales X_h in der vorigen Def
<==> zu $X_f, X_g \in ZS$ mit $X_f \cap X_g \neq \{\}$
ex. genau ein $X_h \in ZS$ mit
($X_h \subset X_f \cap X_g$, und fa. X_m mit $X_h \subset X_m, \quad X_h \neq X_m$,
ist nicht $X_m \subset X_f \cap X_g$)

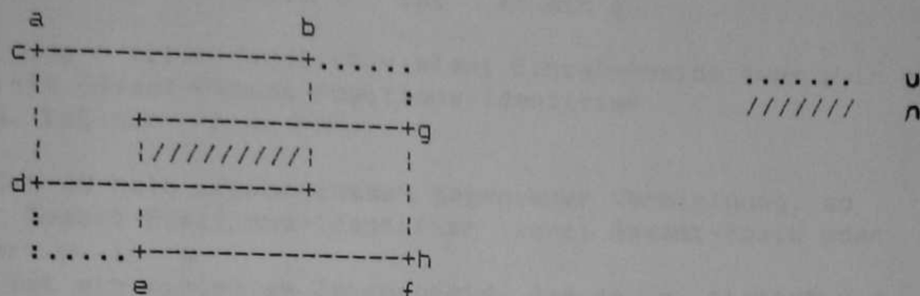
Bsp: Sei ZS wie im Beispiel (a) zu Def 2.3.1, dann gilt:
- ZS ist halb abgeschlossen gegen Vereinigung, wenn zu jeder natuerlichen Zahl k ein $X_f \in ZS$ ex. mit $[0,k]^n \subset X_f$ (d.h. X_f umfasst einen n-dimensionalen Wuerfel der Kantenlaenge k);
- ZS ist halb abgeschlossen gegen Durchschnitt, wenn eine (und damit alle) einelementige Menge in ZS existiert.

(Beide Bedingungen sind hinreichend, aber nicht notwendig.)

- Rechtecke sind eindeutig halb abgeschlossen gegen Vereinigung und Durchschnitt:

[a,b]x[c,d] u [e,f]x[g,h] c [min(a,e),max(b,f)]x[min(c,g),max(d,h)]

Durchschnitt entsprechend mit min und max vertauscht



- Strings mit Luecken sind abgeschlossen gegen Vereinigung.
- Trees sind eindeutig halb abgeschlossen gegen Vereinigung.

6. Lem: Sei ZS abgeschlossen gegenueber Vereinigung,
dann gilt (a) f1,f2 direkt in g ==> (f1 u f2) direkt in g
und (b) f direkt in (f u g)

Bew: (a)
f1,f2 direkt in g
==> Xf1 c Xg, Xf2 c Xg, f1 = g|Xf1, f2 = g|Xf2
==> (mit f := (f1 u f2) ∈ ZF n.Vor.)
Xf = Xf1 u Xf2 c Xg, und fa. x ∈ Xf ist:

f(x) = {f1(x) fuer x ∈ Xf1 = {g(x) fuer x ∈ Xf1 = g(x)
{f2(x) sonst {g(x) sonst

d.h. f = g|Xf
==> (f1 u f2) direkt in g

(b)
f direkt in (f u g) klar nach Def

7. Def: Sei SP := st[ZS] die Menge aller Positionen, an denen eine ZF-Struktur starten kann (wenn keine Verwechslung moeglich ist, auch als Menge der Startpositionen bezeichnet).

Bsp: Um Trees mit Kantenmarkierungen als Zeichenfelder definieren zu koennen, braucht man auch Positionen, die zwar Markierungen tragen koennen, aber an denen keine ZF-Struktur starten kann.
Denn ein Tree soll stets mit der Wurzel starten, aber nie mit einer Kante. Also wird man SP als Menge aller Knoten definieren und ZS \ SP als Menge aller Kanten.

8. Def: Sei $g \in ZF$, $p \in (Xg \cap SP)$,
$f \in ZF$ heisst Einzel-Positions-Identifier (kurz: Einzel-Posid oder EPosid) fuer p in g
:<==> st(f) = p, f dein g,
fa. f' mit f' xdlin f ist f' min g

Die Vereinigung h aller (endlich vielen) Einzel-Posids fuer p in g heisst Gesamt-Pseudo-Positions-Identifier
i.a. ist nur $h \in PZF$

Ist ausserdem ZS halb abgeschlossen gegenueber Vereinigung, so heisst h' Gesamt-Positions-Identifier (kurz: Gesamt-Posid oder GPosid) fuer p in g
:<==> h' ist ein minimales Zeichenfeld, das an p startet und dessen Struktur Xh' die Struktur Xh des Gesamt-Pseudo-Positions-Identifiers enthaelt
<==> $Xh \subset Xh'$, $h' = g|Xh'$, $Xh' \in ZS$, $st(Xh') = p$,
es ex. kein $Xh'' \in ZS$ mit $Xh \subset Xh'' \subset Xh'$,
$Xh \neq Xh'' \neq Xh'$
(h' ist nicht immer eindeutig bestimmt, vgl. Bsp zu Korr 12)

Bsp: Als Zeichenfelder seien beliebige Rechtecke zugelassen.

```
        1 2 3 4        Einzelposids        2 3 4       2
f =    +-------        fuer (1,2):        +-----  ,   +-
       1|a b c b                          1|b c b     1|b
       2|b c a a                                      2|c
       3|a a b b
       4|b a a a       Pseudo-Gesamt-      2 3 4
                       Posid:             +-----
                                          1|b c b
                                          2|c

                       Gesamt-Posid:       2 3 4
                                          +-----
                                          1|b c b
                                          2|c a a
```

Wie sehen die Position-Identifier bei Strings mit Luecken aus ?

Beh: Sind als Zeichenfelder Strings mit Luecken zugelassen (vgl. Bsp (f) zu 2.3.1), ist $f \in ZF$, $p \in Xf$,
sind g1,...,gn alle Minimal-Identifier von f, die bei p beginnen,
sind h1,...,hm alle Minimal-Identifier von f, die rechts von p beginnen und wo h'(i) := f | ({p} u Xhi) keines der gj enthaelt,
dann
sind g1,...,gn, h'(1),...,h'(m)
alle Einzel-Posids von p in f

Bew: alle gi sind Einzel-Posid nach Satz 2.7.2 (s.u.),

alle h'(i) sind Einzel-Posids:
st(h'(i)) = p, da min Xhi > p,
h'(i) einmal in f, da hi in h'(i), hi Minid,
ist h xlin h'(i) fuer ein h,
so ist hi nicht in h, und h enthaelt auch keinen anderen Minid (denn der muesste echt in hi liegen oder bei p beginnen, beides ist ausgeschlossen),
also h mehrmals in f (nach 2.2.9 mit 2.3.7)

es gibt keine weiteren Einzel-Posids:
ist h Einzel-Posid von p, aber kein Minid (sonst ist h = gi fuer ein i), so ex. nach 2.7.1 ein Suffix h' von h (Def 2.5.20), der Minid ist,
es ist h' $\neq$ h, da sonst h Minid,
also beginnt h' rechts von p und gehoert damit zu den hi : h' = hi fuer ein i,
also ist Xh'(i) = {p} u Xhi c Xh,
und da h'(i) = f ¦ ({p} u Xhi) einmal in f,
und h minimal mit h einmal in f, st(h) = p,
muss schon Xh'(i) = Xh sein, also h = h'(i)

Allerdings hat f $\in$ ZF i.a. $O(|Xf|^2)$ viele Minids, z.B.

```
f =  a...cdad.bca..b          ("." deutet eine Luecke an)
     123456789012345
              111111
```

hat als Teil-Zeichenfelder:	kommt vor ab Position:	
a	1, 7, 12	
a...c	1, 7	
a....d	1	(Minid)
a.....a	1	(Minid)
a......d	1	(Minid)
a........b	1	(Minid)
a.........c	1	(Minid)
a..........a	1	(Minid)
a.............b	1	(Minid)
c	5, 11	
cd	5	(Minid)
c.a	5	(Minid)

... usw.

Insgesamt hat f gerade $2^{|Xf|}-1$ Teil-Zeichenfelder,
denn jede nicht-leere Teilmenge von Xf ist auch ZF-Struktur;
davon sind fast alle zweielementigen Teil-Zeichenfelder
(deren gibt es |Xf| * (|Xf|+1) / 2) Minids.

9. Lem: Ist ZS halb abgeschlossen gegen Vereinigung, p $\in$ SP, g $\in$ ZF,
f' ein Gesamt-Posid fuer p in g,
f1 Einzel-Posid fuer p in g,
dann ist f1 dlin f'

Bew: Sei f der Gesamt-Pseudo-Posid fuer p in g, dann ist
Xf1 c Xf c Xf', f'¦Xf1 = g¦Xf1 = f1 und st(f') = p = st(f1)

10. Lem: ZS eindeutig halb abgeschlossen gegen Vereinigung
$\Longrightarrow$ zu $Xf, Xg \in ZS$ ex. Xh mit
$Xf \cup Xg \subset Xh$ und fa. Xk mit $Xf \cup Xg \subset Xk$ ist $Xh \subset Xk$
(d.h. es ex. ein kleinstes Element Xh)

Bew: Sei $Xf \cup Xg \subset Xk$,
sei Xm eine minimale ZF-Struktur (bzgl. $\subset$) in Xk mit
$Xf \cup Xg \subset Xm$,
dann gilt $Xf \cup Xg \subset Xm$, fa. $Xm' \subset Xm$, $Xm' \neq Xm$
ist nicht $Xf \cup Xg \subset Xm'$,
wegen der Eindutigkeit von Xh folgt $Xm = Xh$,
also $Xh = Xm \subset Xk$

11. Satz: Sei ZS eindeutig halb abgeschlossen gegenueber Vereinigung,
dann ex. zu $Xf1, \ldots, Xfn \in ZS$ genau ein minimales (bzgl. $\subset$)

$Xf' \in ZS$ mit $\bigcup_{i=1}^{n} Xfi \subset Xf'$

Bew: Induktion ueber n:
n=1: waehle $Xf' := Xf1$
n=2: klar, da ZS eindeutig halb abgeschlossen gegen Vereinigung
n-->n+1:
nach Ind.vor. ex. genau ein minimales $Xf'' \in ZS$ mit

$$\bigcup_{i=1}^{n-1} Xfi \subset Xf''$$

zu Xf'' und Xfn ex. nach Vor. genau ein minimales Xf'

mit $\bigcup_{i=1}^{n} Xfi = (\bigcup_{i=1}^{n-1} Xfi) \cup Xfn \subset Xf'' \cup Xfn \subset Xf'$

Sei nun $\bigcup_{i=1}^{n} Xfi \subset Xh$, dann ist $\bigcup_{i=1}^{n-1} Xfi \subset Xh$,

also nach 10. $Xf'' \subset Xh$,
wegen $Xfn \subset Xh$ folgt weiter $Xf'' \cup Xfn \subset Xh$,
also $Xf' \subset Xh$
d.h. Xf' ist die einzige minimale ZF-Struktur mit

$$\bigcup_{i=1}^{n} Xfi \subset Xf'$$

12. Korr: Sei ZS eindeutig halb abgeschlossen gegen Vereinigung, dann gibt es zu jedem $f \in ZF$ und jeder Position $p \in Xf$ hoechstens einen Gesamt-Positions-Identifier

Bew: folgt direkt aus den Definitionen und Satz 11.

Bsp: Als Zeichenfelder seien Rechtecke, aber keine Quadrate, zulaessig.

```
      1 2 3 4    Einzelposids     1 2        1     Pseudo-       1 2
 f =  +-------   fuer (1,1)      +---   ,   +-     Gesamt-       +---
     1|a b a c   in f:          1|a b      1|a     Posid:       1|a b
     2|b c c c                             2|b                  2|b
     3|b b b b
                 Gesamt-      1 2 3       1 2
                 Posids:     +-----   ,  +---
                            1|a b a     1|a b
                            2|b c c     2|b c
                                        3|b b
```

ZS ist nicht eindeutig halb abgeschlossen gegen Vereinigung !

KAPITEL 2.5 Die Sortierstruktur

Die zentrale Datenstruktur im Weiner-Algorithmus war der Prefix-Bi-Tree, mit dessen Hilfe die verschiedenen Position-Identifier nach der lexikografischen Ordnung einsortiert wurden. Wurde fuer einen Eingabestring f der Prefix-Bi-Tree konstruiert, so galt fuer jeden beliebigen String g :

g kommt einmal in f vor ==> ein Prefix von g (naemlich der Posid) kommt im Tree vor,

und

g kommt mehrmals in f vor ==> g kommt im Tree vor.

Im mehrdimensionalen Fall kann man nicht erwarten, diese Eigenschaft ebenfalls mit einem Baum (abgesehen von zusaetzlichen Hilfs-Zeigern), der fuer seine Knoten eine totale Ordnung darstellt, erreichen zu koennen. Schon im zweidimensionalen Fall, wenn als Zeichenfelder Rechtecke zugelassen sind, gibt es keine totale Ordnungsrelation mehr, die mit "lin" vertraeglich ist (siehe Gegenbeispiel unten, Bem 3).

Daher ist es notwendig, eine neue Sortierstruktur zu definieren (in Abhaengigkeit von der durch ZS, ≡ und Iso vorgegebenen Struktur der Zeichenfelder): Die hier vorgeschlagene Sortierstruktur zu einem Eingabe-Zeichenfeld f entsteht, indem man zu jeder Position einen MPMR auswaehlt und deren saemtliche Prefixe in die Sortierstruktur aufnimmt, wobei ein Zeiger zwischen zwei Prefixen dazugenommen wird, wenn der eine ein maximaler echter Prefix des anderen ist.

Mit dieser Sortierstruktur bleibt die Eigenschaft des Prefix-Bi-Trees erhalten (Satz 23).

1. Def: Eine irreflexive Ordnungsrelation < heisst vertraeglich mit lin
:<==> (a) p < q < r, p xlin r ==> p xlin q,
und (b) p xlin r ==> p < r

Bem: Diese Bedingung ist notwendig, damit beim Sortieren einer Menge von Zeichenfeldern nach der Relation < solche Zeichenfelder benachbart werden, von denen das eine ein Prefix des anderen ist.
Im eindimensionalen Fall z.B.:
p := "aba" und r := "ababbc",
es wird verlangt, dass z.B. q := "acaa" nicht dazwischen sortiert wird.

2. Satz: Die lexikografische Ordnung im eindimensionalen Fall (Satz 0.3.1.7) ist mit "ist Prefix von" vertraeglich.

Bew: (a) sei p < r < q, p xlin q, zeige p xlin r :
sei p ≠ $ (sonst trivial), p ≠ r, da < irreflexiv
sei t der maximale gemeinsame Prefix von p und r,
p = t * p', r = t * r',
sei q = p * q', dann folgt

```
        p < r    ==>  p' < r'
                 ==>  p' = $                (also  p = t lin t*r' = r,   qed.)
                      oder
                      p' = a * p",    r' = b * r"         fuer   a,b ∈ A
                      a < b   oder   a = b,   p" < r"
                      (da   a ≠ b   nach Def von  t, folgt weiter:)
                 ==>  a < b
                 ==>  q = p * q' = t * a * p" * q'
                                 < t * b * r"        = r ,   W.!
                 also muss gelten   p' = $     bzw.   p lin r

    (b)  sei   p xlin q,    zeige   p < q :
         q = p * q'   ==>   (p < q   <==   $ < q'
                                     <==   q' ≠ $
                                     <==   q ≠ p)
                      ==>    p < q
```

3. Bem: Gegenbeispiel im zweidimensionalen Fall
(Rechtecke als Zeichenfelder):
Angenommen, es gaebe eine totale Ordnungsrelation, die mit lin
vertraeglich ist.

```
Wegen              a   xlin   ab   xlin   ab
                                          ba

und                a   xlin   a    xlin   ab
                              b           ba

gilt dann          a    <     ab    <     ab
                                          ba

und                a    <     a     <     ab
                              b           ba
```

```
wenn  <  total sein soll, muss entweder     ab   <   a    gelten
                                                     b

oder umgekehrt, beides widerspricht aber der Vertraeglichkeit:

     ab  <  a  <  ab  ,   ab  xlin  ab  ==>  ab  xlin  a
            b     ba                ba                 b

und umgekehrt.
```

4. Def: Sei f,g ∈ ZF, dann heisst

f direkt unmittelbar linksbuendig in g (kurz: f dulin g)

```
:<==>  f dlin g,    fa.   h ∈ ZF   gilt
                    f dlin h dlin g   ==>   f = h   oder   g = h
```

f unmittelbar linksbuendig in g (kurz: f ulin g)

```
:<==>  f lin g,     fa.   h ∈ ZF   gilt
                    f lin h lin g    ==>   f ≡ h   oder   g ≡ h
```

Xf direkt unmittelbar linksbuendig in Xg

```
:<==>  Xf dlin Xg,   fa.   Xf dlin Xg dlin Xg
                     ist   Xh = Xf    oder Xh = Xg
```

Bem: f dulin g ==> Xf dulin Xg

Bsp: Als Zeichenfelder seien beliebige Rechtecke zulaessig.

```
 ab           abc          abc
 ba   ulin    bac   ulin   bac
                           cbc
```

5\. Def: Fuer $f \in ZF$ heisst
dPrf(f) := {g | g dlin f} Menge der direkten Prefixe von f
(g dlin f heisst auch direkter Prefix von f)
Prf(f) := {g | g lin f} Menge der Prefixe von f

Fuer $M \subset ZF$ heisst
$$dPrf(M) := \bigcup_{f \in M} dPrf(f)$$
Menge aller direkten Prefixe von M, und
$$Prf(M) := \bigcup_{f \in M} Prf(f)$$
Menge aller Prefixe von M.

Es ist $dPrf(f) \subset Prf(f)$ und $dPrf(M) \subset Prf(M)$.

6\. Def: $f \in ZF$ heisst Anfangs-Zeichenfeld (kurz: Anfangs-ZF)
:<==> f ist minimales Element bzgl. lin
<==> f ist minimales Element bzgl. dlin

AZF := {f | f ist Anfangs-Zeichenfeld}
AZS := {Xf | $f \in$ AZF}
$Xf \in$ AZS heisst Anfangs-ZF-Struktur (kurz auch: Anfangs-ZS)

7\. Bem: Die $Xf \in$ AZS sind gerade die minimalen Elemente (bzgl. $\subset$), die an einer vorgegebenen Position starten.

8\. Satz: fa. $f \in ZF$ ist $dPrf(f) \cap AZF \neq \{\}$

Bew: Sei g minimales Element bzgl. dlin mit der Eigenschaft g dlin f (ex. nach 0.3.3.4, da f dlin f und dPrf(f) endlich), dann ist $g \in dPrf(f)$ wegen g dlin f, und $g \in AZF$, denn waere etwa h xdlin g, so folgt h dlin f, W.! zur Minimalitaet von g

9\. Bem: i.a. enthaelt $dPrf(f) \cap AZF$ mehrere Elemente,
z.B. im Rechteck-Fall, wenn aa und a/a (a über a) als Zeichenfelder zugelassen sind, nicht aber a

10. Lem: fa. g,h ∈ ZF mit g dlin h gilt:
ex. eine Kette g = g1 xdulin g2 xdulin ... xdulin gn = h

Bew: setze g1 := g,
ist g1,...,gj bereits gefunden mit
g = g1 xdulin g2 xdulin ... xdulin gj xdlin h,
so koennen zwei Faelle eintreten:

Fall 1: ex. g' ∈ ZF mit gj xdlin g' xdlin h,
dann sei g(j+1) minimales Element (bzgl. dlin) mit
gj xdlin g(j+1) dlin g' ,
dann ist gj xdulin g(j+1) ,
denn gj dlin g" dlin g(j+1)
==> nicht gj xdlin g" (sonst Widerspruch zur Minimalitaet von g(j+1) wegen g" dlin g(j+1) dlin g')
==> gj = g"
damit kann die Kette um das Element g(j+1) verlaengert werden

Fall 2: ex. kein solches g', dann folgt:
gj dlin g' dlin h ==> gj = g' oder g' = h
d.h. gj xdulin h , die Kette bricht ab, weil sie vollstaendig ist

Die Kette muss irgendwann abbrechen, da alle gj verschieden sind und nur endlich viele gj in h existieren.

11. Lem: Sei f ∈ ZF, g ∈ dPrf(f),
dann ex. eine Kette von Zeichenfeldern aus dPrf(f):
g1 xdulin g2 xdulin ... xdulin gi = g xdulin ... xdulin gn = f
mit g1 ∈ AZF

Bew: nach 10. folgt:
g ∈ dPrf(f) ==> g dlin f ==> ex. g = g1 xdulin ... xdulin gn = f,
die gj liegen in dPrf(f), da gj dlin f
und nach 8. ex. g1 ∈ dPrf(f) n AZF , also g1 dlin g
==> ex. g1 xdulin g2 xdulin ... xdulin gi = g
da g dlin f, ist auch gj dlin f fa. j, d.h. gj ∈ dPrf(f)

12. Satz: Sei f,g ∈ ZF, g din f, dann gilt
g einmal direkt in f <==> ex. g' lin g mit g' EPosid von f

Bew: "<==": sonst waere nach 2.2.1e g' mehrmals in f

"==>": nach 0.3.3.4 ex. ein minimales Element g' in g (bzgl. lin)
mit g' einmal in f, d.h. g' ist EinzelPosid

13. Bem: (1)

Im eindimensionalen Fall enthielt der Posid einer Position i schon automatisch alle Strings, die bei i starten und mehrmals vorkommen. Im mehrdimensionalen Fall gilt das nicht mehr automatisch, wie folgende Gegenbeispiele zeigen:

Als Zeichenfelder seien beliebige Rechtecke zugelassen.
Betrachte das Zeichenfeld:

```
w=  a a b a a d
    a a c a a e
    d e f b c f
```

EinzelPosids fuer Pos 1/1:

```
a a b  ,  a
          a
          d
```

darin kommt mehrfach vor:

```
v=  a a
    a a
```

GesamtPosid fuer 1/1:

```
a a b
a a c
d e f
```

v kommt zwar im GesamtPosid vor, aber in keinem EinzelPosid.

Seien jetzt als Zeichenfelder beliebige Vereinigungen von Rechtecken mit gleicher Startposition zugelassen.
Die EinzelPosids fuer w sind wie oben, der GesamtPosid fuer 1/1 ist jetzt:

```
a a b
a
d
```

v kommt also weder in einem Einzel- noch im GesamtPosid vor.

Betrachte jetzt das Zeichenfeld:

```
a a a b
a a a c
a a a d
e f g h
```

EinzelPosids fuer 1/1:

```
a a a b  ,  a  ,  a a a
            a     a
            a     a
            e
```

darin kommt mehrfach vor:

```
a a a  ,  a a
a a a     a a
          a a
```

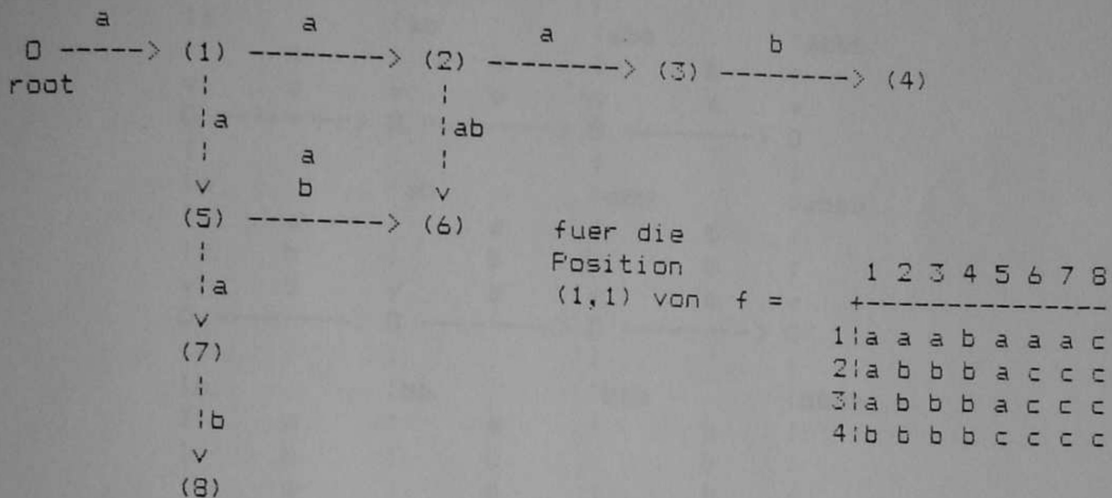
GesamtPosid fuer 1/1:

```
a a a b
a
a
e
```

Es gibt also mehrere maximale mehrfach vorkommende Zeichenfelder, trotzdem liegt keines von ihnen in einem EinzelPosid oder im GesamtPosid.

Will man die Eigenschaften des Prefix-Trees erhalten (siehe Einleitung dieses Kapitels), so muss man also ausdruecklich dafuer sorgen, dass auch alle mehrfach vorkommenden Zeichenfelder in die Sortierstruktur aufgenommen werden. Sie sind nicht mehr alle automatisch als Prefixe von (Einzel- oder Gesamt-) Posids beruecksichtigt.

(2)
Wuenschenswert ist, dass von einem verallgemeinerten Prefix-Tree-Algorithmus pro Position nur EIN Zeichenfeld abgespeichert wird, d.h. z.B. nicht:

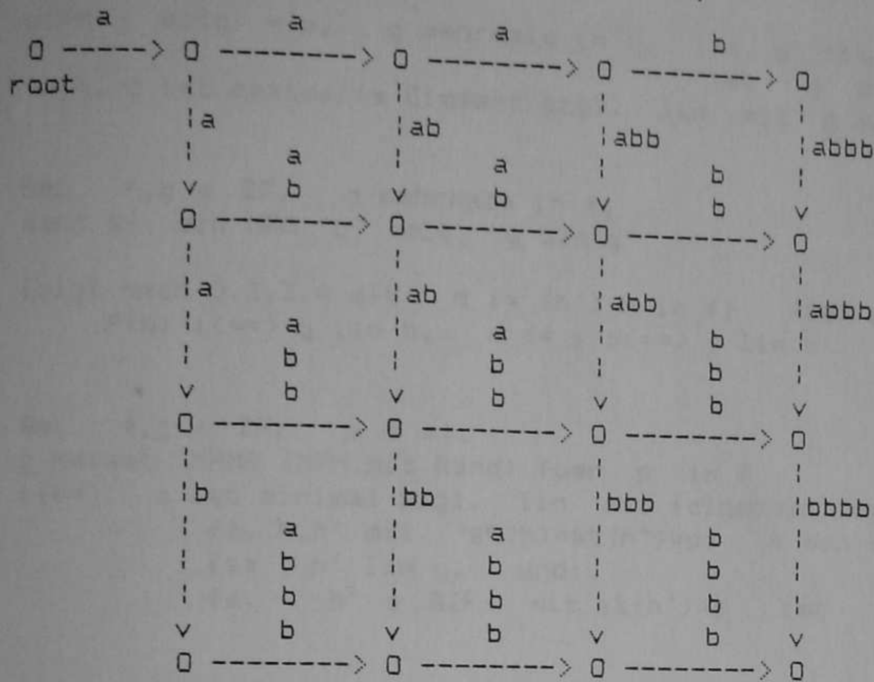


Legende: jeder Knoten entspricht einem Zeichenfeld;
der Knoten, auf den die mit a markierte Kante von root aus zeigt, steht fuer das Anfangs-ZF a ;
ein waagerechter mit einer Spalte markierter Pfeil steht fuer das Anhaengen dieser Spalte;
entsprechend steht ein mit einer Zeile markierter senkrechter Pfeil fuer das Anhaengen dieser Zeile;
die Knoten (1) bis (8) stehen also fuer:

```
(1)  a          (2)  aa        (3)  aaa       (4)  aaab

(5)  a          (6)  aa
     a               ab

(7)  a
     a
     a

(8)  a
     a
     a
     b
```

(Die Knoten (4), (6) und (8) entsprechen den Einzelposids von Position (1,1) von f)

sondern:



Denn bei der Suche, ob z.B.

```
   1 2 3 4
  +-------
 1¦a a a c
 2¦a b b b
```

in f liegt, wuerde man sonst

```
   1 2 3 4             1 2
  +-------      und   +---
 1¦a a a c           1¦a a
                     2¦a b
```

als eindeutig finden und muesste fuer BEIDE (und bei anderen Beispielen fuer sehr viel mehr) Zeichenfelder pruefen, ob die restlichen Zeichen mit dem vorgegebenen Zeichenfeld uebereinstimmen. Im Fall, dass nur ein Zeichenfeld abgespeichert wird, wuerde man dies sofort feststellen koennen

```
                     1 2 3                          c
                                                    b
(vom Knoten fuer    +-----    gibt es keinen Zeiger  ----->  ).
                   1¦a a a
                   2¦a b b
```

(3)
Diese Ueberlegungen und die weitere Ueberlegung bei McCreight, dass es nicht unbedingt notwendig ist, mit Posids zu arbeiten, fuehren zu folgenden Definitionen:

14. Def: Sei f,g ∈ ZF, p ∈ Xf,
g heisst MPM (maximales positionsgebundenes Mehrfach-Zeichnfeld)
fuer p in f
:<==> st(g) = p, g mehrmals in f, fa. g' mit g xlin g' in f
ist g' einmal in f
(d.h. g ist maximales Element bzgl. lin mit g min f)

15. Satz: Sei f,g ∈ ZF, g mehrmals in f,
dann ex. ein MPM g' mit g lin g'

Bew: folgt nach 0.3.3.4 mit M := {h | h in f} (ist endlich),
P(h) :<==> g lin h, x <= y :<==> y lin x

16. Def: Sei f,g ∈ ZF, p ∈ Xf,
g heisst MPMR (MPM mit Rand) fuer p in f
:<==> g ist minimal bzgl. lin mit folgender Eigenschaft:
fa. h,h' mit st(h)=st(h')=p, h min f, h xdulin h'
ist h' lin g, und:
fa. h' ∈ AZF mit st(h')=p ist h' lin g

Schon mit Blick auf den Algorithmus sei hier definiert:

17. Def: Sei f,g ∈ ZF, p ∈ Xf,
sei <= eine reflexive Ordnungsrelation auf SP,
g heisst vorlaeufiger MPMR (MPM mit Rand) fuer p in f
bzgl. den Positionen <= q
:<==> g ist minimal bzgl. lin mit folgender Eigenschaft:
fa. h,h' mit st(h)=st(h')=p, h xdulin h',
h min h' bzgl. den Positionen <= q
ist h' lin g, und:
fa. h' ∈ AZF mit st(h')=p ist h' lin g

dabei heisst
h min h' bzgl. den Positionen <= q
:<==> ex. h1,h2 mit h1 ≡ h2 ≡ h, Xh1 ≠ Xh2,
st(Xh1), st(Xh2) <= q,
h1, h2 in h'

18. Lem: Sei p0 groesstes Element von f bzgl. <=
(d.h. q <= p0 fa. q ∈ Xf), dann gilt:
g MPMR fuer p in f
<==> g vorlaeufiger MPMR fuer p in f bzgl. den Positionen <= p0

Bew: klar, da
h min h' bzgl. den Positionen <= p0 <==> h min h'

19. Lem: Ist g MPMR fuer p in f, h' EinzelPosid fuer p in f,
so ist h' lin g

Bew: sei h xdulin h', dann ist h min f, also h' lin g
existiert kein h mit h xdulin h', so ist h' ∈ AZF,
also ebenfalls h' lin g

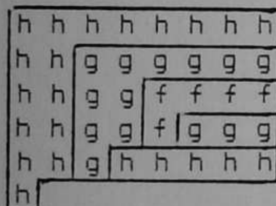
20. Def: Sei Xf,Xg ∈ ZS, p ∈ Xf, dann heisst
Xg ein Suffix von Xf ab p
:<==> Xg din Xf, st(Xg)=p, ex. kein Xh mit Xg xlin Xh in Xf
(d.h. Xg ist maximales Element bzgl. lin mit st(Xg)=p, Xg in f)
Xg heisst Suffix von Xf
:<==> Xg ist Suffix von Xf ab p fuer ein p ∈ Xf

21. Bem: (a) zu einer Position kann es mehrere Suffixe geben
(b) "ist Suffix von" ist reflexiv und antisymmetrisch, aber nicht notwendig transitiv
(c) hinreichende Bedingungen fuer Eindeutigkeit und fuer Transitivitaet siehe unten, 2.10.10 und 2.10.11

Bew: (a) enthaelt ZS alle Rechtecke, die keine Quadrate sind,
so hat Xf := [1,4]x[2,4] ab p := (2,2)
zwei Suffixe, naemlich [2,3]x[2,4] und [2,4]x[2,3]

(b) Antisymmetrie:
ist Xg Suffix von Xf ab p, Xf Suffix von Xg ab q,
so folgt: Xf din Xg din Xf ==> Xf = Xg, p=st(Xg)=st(Xf)=q

(b) Gegenbeispiel fuer Transitivitaet:
als Zeichenfelder seien nur solche Figuren zugelassen, die aus einem Rechteck entstehen, indem die erste Spalte um eine Position nach unten verlaengert wird:



Xf (angedeutet durch "f") ist Suffix von Xg ("g" oder "f"), Xg ist Suffix von Xh, aber Xf ist nicht Suffix von Xh (es koennte noch in jeder Spalte um eine Position nach unten verlaengert werden)

22. Lem: Sei jeder Suffix von f einmal in f,
ist g ein MPMR fuer p in f,
h mehrmals in f mit st(h) = p,
so ist h lin g

Bew: sei h" maximal mit h lin h" min f (ex. nach 0.3.3.4),
da h" n.Vor. nicht Suffix von f, ex. ein h' in f mit
h" xdulin h', also h' einmal in f ==> h' lin g (da g MPMR)
==> h lin h" lin h' lin g

23. Satz: Sei f,g ∈ ZF, jeder Suffix von f sei einmal in f,
sei M c ZF derart, dass fuer jede Position p von f ein MPMR
MPMR in M enthalten ist, dann gilt:
(a) g einmal in f ==> ex. g' lin g mit g' ∈ Prf(M)
(b) g mehrmals in f ==> g ∈ Prf(M)

Bew: sei g" ∈ M ein MPMR fuer st(g) in f, (ex. nach 24.)
(a) nach Satz 12 ex. ein EinzelPosid g' mit g' lin g,
nach 19. ist g' lin g", also g' ∈ Prf(M)
(b) nach 22. ist g lin g", also g ∈ Prf(M)

24. Lem: Ist $f \in ZF$ und jeder Suffix von f einmal in f, so ex. zu jeder Position p von f ein Einzel-Posid und ein MPMR

Bew: nach 0.3.3.4, denn jeder einmal vorkommende Suffix erfuellt die Bedingungen, bzgl. derer ein Einzelposid sowie ein MPMR minimal sind (Def 2.4.8, 2.5.16)

KAPITEL 2.6 S-Father-Zeiger

In diesem Kapitel wird eine (partielle) Rangfolge unter den Positionen definiert als Voraussetzung fuer eine Verwendung von sfather-Zeigern.

1. Def: p,q ∈ SP seien Startpositionen, dann heisst
   p vor q :<==> ex. Xf ∈ ZS mit st(Xf) = p, q ∈ Xf
   (anschaulich: die Startposition kommt vor jeder anderen)
   p neben q :<==> p vor q und q vor p
   p unmittelbar vor q (kurz: p uvor q)
   :<==> p vor q und fa. r mit p vor r vor q
   ist r neben p oder r neben q

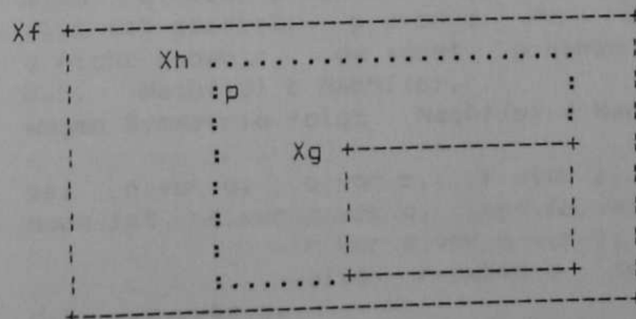
   Nachf(p) := {q | p unmittelbar vor q, p nicht neben q}
   heisst die Menge der Nachfolger von p

Bsp: (vgl. Bsp zu 2.3.1, (b) und (e))
   Im Rechteck-Fall ist:
   (a,b) vor (c,d) <==> a <= c, b <= d,
   neben = Id,
   (a,b) uvor (c,d) <==> (a,b) = (c,d) oder (a,b) = (c+1,d)
   oder (a,b) = (c,d+1),

   Nachf(a,b) = {(a+1,b), (a,b+1)};

   im Tree-Fall ist:
   p vor q <==> ex. i mit $2^i*p <= q < 2^i*(p+1)$,
   neben = Id,
   p uvor q <==> q = 2*p oder q = 2*p+1,
   Nachf(p) = {2*p, 2*p+1}

2. Def: ZS hat die Eigenschaft der Links-Erweiterbarkeit
   :<==> fa. p ∈ SP, Xf,Xg ∈ ZS mit p vor st(Xg)
   ex. Xh mit st(Xh)=p, Xg Suffix von Xh,
   und: war p ∈ Xf, Xg c Xf, so ist auch Xh c Xf
   (anschaulich: Xg wurde nach links erweitert bis zum Punkt p)
   ZS heisst dann auch links-erweiterbar.



3. Lem: Sei ZS links-erweiterbar, dann gilt:
(a) "neben" ist Aequivalenzrelation,
(b) "vor" ist reflexive Ordnungsrelation auf den Aequivalenzklassen von "neben"

Bew: (1) "vor" ist reflexiv: klar nach Def, und da p ∈ SP

(2) "vor" ist transitiv: sei p vor q, q vor r,
dann ist st(Xf)=p, q ∈ Xf, st(Xg)=q, r ∈ Xg
fuer geeignete Xf,Xg ∈ ZS,
n.Vor. laesst sich Xg nach links bis q erweitern,
d.h. ex. Xf' mit st(Xf')=p, Xg Suffix von Xf'
==> p vor r, denn st(Xf')=p, r ∈ Xg c Xf'

(3) "neben" ist reflexiv: folgt aus (1)

(4) "neben" ist symmetrisch: nach Def

(5) "neben" ist transitiv: folgt aus (2)

aus (3), (4), (5) folgt (a)

(6) "vor" laesst sich auf den Aequivalenzklassen von "neben" definieren:
sei p neben p', q neben q', p vor q, dann ist
p' vor p vor q vor q', d.h. "vor" ist repraes.unabhaengig
Reflexivitaet und Transitivitaet von "vor" bleiben erhalten

(7) Antisymmetrie von "vor" auf den Aequivalenzklassen:
p vor q, q vor p ==> p neben q ==> [p] = [q]

Fuer den Rest des Kapitels sei stets ZS links-erweiterbar.

4. Lem: (a) p neben q ==> Nachf(p) = Nachf(q)
(b) p neben q, p ∈ Nachf(r) ==> q ∈ Nachf(r)

Bew: (a) sei p vor q, q vor p, p uvor r, dann ist
q vor p vor r, und fa. s mit q vor s vor r ist
p vor q vor s vor r,
d.h. wegen p uvor r: p neben s oder r neben s,
also q neben p neben s oder r neben s
also ist gezeigt: p uvor q ==> q uvor r,
q nicht neben r, da sonst p neben q neben r
d.h. Nachf(p) c Nachf(q),
wegen Symmetrie folgt Nachf(p) = Nachf(q)

(b) sei p vor q, q vor p, r uvor p,
dann ist r vor p vor q, und fa. s mit r vor s vor q ist
r vor s vor q vor p,
also r neben s oder q neben p neben s,
d.h. r uvor q,
und q nicht neben r, da sonst p neben q neben r,
also q ∈ Nachf(r)

5. Def: Nachf laesst sich somit auch auf den Aequivalenzklassen bzgl. "neben" definieren:
Nachf([p]) := Nachf(p)

Nach 4a ist die Def repraesentantenunabhaengig,
nach 4b umfasst Nachf([p]) stets eine oder mehrere Aequivalenzklassen vollstaendig.

6. Lem: p neben q ==> p uvor q

Bew: p neben q ==> p vor q,
und fa. r mit p vor r vor q gilt
p vor r vor q vor p, d.h. r neben p

7. Lem: [q] c Nachf([p]) <==> q ∈ Nachf(p)

Bew: "==>": q ∈ [q] c Nachf([p]) = Nachf(p)
"<==": q ∈ Nachf([p]), r neben q ==> r ∈ Nachf([p]) (4b)

8. Def: Sei Xf ∈ ZS, p,q ∈ Xf,
Xf heisst nicht zusammenhaengend zwischen p und q
:<==> ex. r ∈ ZS mit p xvor r xvor q,
aber ex. kein r ∈ Xf mit p xvor r xvor q
Xf heisst nicht zusammenhaengend, wenn es solche p und q gibt,
andernfalls heisst Xf zusammenhaengend.

Bsp: Xf = [1,2]x[1,2] u [1,2]x[5,6] ist nicht zusammenhaengend zwischen (1,2) und (1,5) und zwischen (2,2) und (2,5)

9. Def: (Nachfolge-Graph)
Sei Xf ∈ ZS,
der Nachfolge-Graph von Xf wird wie folgt definiert:
- die Knoten sind die Aequivalenzklassen von Positionen von Xf n SP,
- von einem Knoten [p] fuehrt ein gerichteter Pfeil zu einem Knoten [q], wenn [q] c Nachf([p])
- von [p] fuehrt ein gerichteter Pfeil nach [q], wenn Xf zwischen p und q nicht zusammenhaengend ist
- es existiert ein ausgezeichneter Knoten
  root := [st(Xf)]

10. Lem: (a) der Nachfolge-Graph von Xf enthaelt alle Aequivalenzklassen von (Start-)Position von Xf,
(b) der Nachfolge-Graph enthaelt keine Zyklen,
(c) in root fuehren keine Pfeile rein,
(d) jeder Knoten laesst sich von root aus durch einen gerichteten Weg erreichen,
(e) [p] vor [q] <==> von [p] fuehrt ein gerichteter Weg nach [q]

Bew: (a) nach Def

(e) "<==": nach Def fuehrt nur dann ein Pfeil
von [r] nach [s], wenn [r] vor [s],
wegen der Transitivitaet von "vor" folgt die Beh.
"==>": Induktion ueber die Anzahl n der verschiedenen
Aequivalenzklassen [r] von Positionen von Xf,
die zwischen [p] und [q] liegen koennen
(d.h. fuer die gilt [p] xvor [r] xvor [q]):
n=0: dann ist nach Def [p] uvor [q],
==> ex. ein Pfeil von [p] nach [q]
oder [p] = [q]
die Beh. gelte fuer alle n < n0, zwischen [p] und [q]
existierten gerade n0 verschiedene Aequivalenzklassen
von Positionen von Xf [r1], ... , [rn0]
Fall 1: ex. r ∈ Xf mit [p] xvor [r] xvor [q],
dann liegen zwischen [p] und [r] hoechstens
n0-1 verschiedene Aequivalenzklassen, ebenso
zwischen [r] und [q], d.h. nach Ind.vor.
ex. ein gerichteter Weg von [p] nach [r] und von
von dort nach [q]
Fall 2: ex. kein r ∈ Xf mit [p] xvor [r] xvor [q],
ist [p] uvor [q], so folgt die Beh wie im
Fall n=0,
sonst ist Xf nicht zusammenhaengend bei p und q,
d.h. ex. ein Pfeil von [p] nach [q]
Da in Xf nur endlich viele Positionen liegen, kann es
zwischen [p] und [q] auch nur endlich viele verschiedene
Aequivalenzklassen von Positionen von Xf geben.

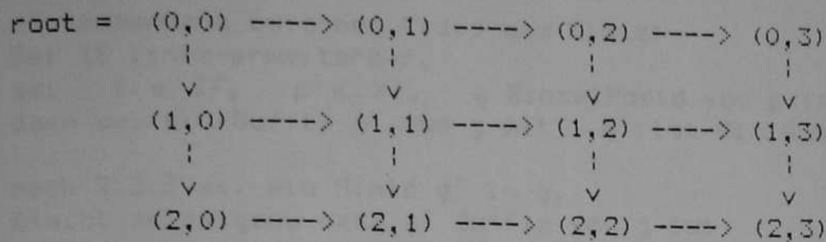
(b) mit (e) wuerde sonst folgen:
[p] xvor [q] vor ... vor [p] (ein Zyklus muesste
mindestens zwei verschiedene Knoten enthalten nach Def)
==> [p] = [q], W.!

(d) folgt aus (e), da [st(Xf)] vor [p] fa. p ∈ Xf ∩ SP

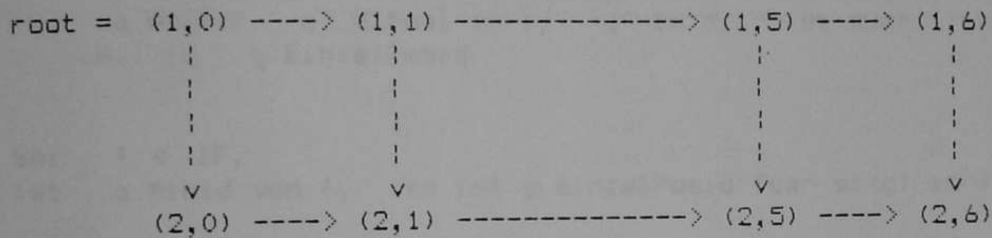
(c) folgt aus (b) und (d)

Bsp: Der Nachfolgegraph von Xf gibt die Struktur auf den Positionen von Xf wieder.

Z.B. Xf = [0,2]x[0,3] hat als Nachfolgegraph:



Ein nicht-zusammenhaengendes Beispiel (vgl. Bsp zu Def 8):
Xf = [1,2]x[1,2] u [1,2]x[5,6] hat als Nachfolgegraph



KAPITEL 2.7 Bloecke
======================

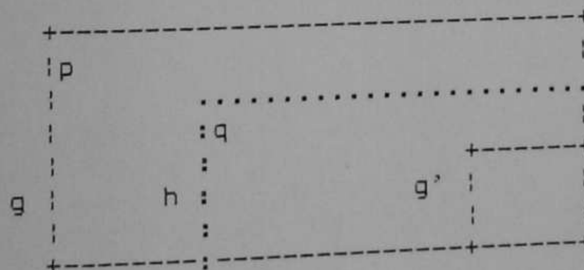
1. Satz: (Zusammenhang zwischen Posid und Minid)
Sei ZS links-erweiterbar,
sei f $\in$ ZF, p $\in$ Xf, g EinzelPosid von p in f,
dann ex. ein Suffix g' von g mit g' ist Minid bzgl. f

Bew: nach 2.2.5 ex. ein Minid g' in g,
bleibt zu zeigen, dass g' Suffix von g ist
Ann: ex. g" mit g' xlin g" in g,
dann laesst sich g' nach links bis p erweitern
(denn p vor st(g'), da st(g)=p, st(g') $\in$ Xg),
d.h. ex. h mit st(h)=p, g' Suffix von h, h in g,
da g' Suffix von h, aber nach Ann g' nicht Suffix von g,
ist h $\neq$ g, also h xlin g,
da weiter g' einmal in f, g' in h, ist auch h ein f,
W.! zu g EinzelPosid

2. Satz: Sei f $\in$ ZF,
ist g Minid von f, so ist g EinzelPosid fuer st(g) in f

Bew: g Minid in f ==> g ein f, fa. g' xin g ist g' min f
==> g EinzelPosid fuer st(g) in f

3. Satz: Sei ZS links-erweiterbar,
sei f $\in$ ZF, p,q $\in$ Xf $\cap$ SP, g Einzelposid fuer p in f,
sei g' ein Suffix von g, der Minid ist (nach 1.),
sei p vor q vor st(g'),
dann ex. ein EinzelPosid h fuer q in f mit g' ist Suffix von h



Bew: q vor st(g') ==> g' laesst sich nach links bis q erweitern
d.h. ex. h mit st(h)=q, g' Suffix von h, h in g
==> h einmal in f, da g' in h
bleibt zu zeigen, dass kein h' xlin h ex. mit h' ein f
sonst wuerde sich h' nach links erweitern lassen bis p
(denn st(g)=p, st(h')=q $\in$ Xg),
d.h. ex. g" mit st(g")=p, h' Suffix von g", und g" lin g,
weiter ist g" $\neq$ g, denn h' ist Suffix von g", aber h' ist kein Suffix von g (denn h' xlin h in g)

also g" xlin g,
ausserdem ist g" einmal in f, da h' in g", h' ein f,
W.! zu g EinzelPosid

```
Bsp:          1 2 3 4 5
       f =   +---------
             1|a b c a c
             2|a a b c c
             3|c a a c c
             4|c b c a a

     Einzelposids          1 2 3 4       1 2 3          1
     fuer (1,1):          +-------       +-----        +-
                         1|a b c a     1|a b c        1|a
                                       2|a a b        2|a
                                                      3|c

     Einzelposids            2 3 4 5       2 3           2
     fuer (1,2):            +-------       +---         +-
                           1|b c a c     1|b c         1|b
                                         2|a b         2|a
                                                       3|a

     Einzelposids              3 4 5        3
     fuer (1,3):              +-----        +-
                             1|c a c      1|c
                                          2|b

     darin                 1 2 3 4          3           1
     Minids               +-------          +-          +-
     (u.a.):             1|a b c a        1|c         1|a
                                          2|b         2|a
                                                      3|c
                               3 4 5
                              +-----
                             1|c a c
```

KAPITEL 2.8 Zusammenhang zwischen xdulin und xuvor

In Kapitel 2.9 soll ein Konzept von Richtungen eingefuehrt werden. Dazu bieten sich im Prinzip zwei Definitionsmoeglichkeiten an:

- Einmal koennte man zu einer Position p jeder unmittelbaren Nachfolgeposition q (d.h. mit p xuvor q) eine Richtung r zuordnen ("von p in Richtung r kommt man nach q"). Im Rechteck-Beispiel erhaelt man so die Richtungen "nach rechts" und "nach unten", im Tree-Beispiel erhaelt man "linker Sohn" und "rechter Sohn".
  Diese Definition von Richtung ist wichtig fuer die sfather-Zeiger, denn ein solcher zeigt vom MPMR von p zum MPMR von $q \in$ Nachf(p) .

- Zum andern koennte man zu einer ZF-Struktur Xf jeder unmittelbaren Erweiterung Xg (d.h. mit Xf xdulin Xg) eine Richtung r zuordnen ("Xf erweitert in Richtung r ergibt Xg"). Im Rechteck-Beispiel erhaelt man wieder die Richtungen "nach rechts" und "nach unten", waehrend man im Tree-Beispiel zu jedem Blatt zwei Richtungen und zu jedem Knoten mit nur einem Sohn eine Richtung erhaelt (d.h. die Anzahl der Richtungen ist abhaengig von Xf).
  Diese Definition von Richtung ist wichtig fuer das Auffinden der Einzel-Posids bzw. MPMRs, da dabei beginnend mit einem Anfangs-ZF das Zeichenfeld solange in allen Richtungen erweitert wird, bis ein Einzel-Posid bzw. MPMR gefunden ist.
  Im Kapitel 2.9 wird diese Art von Richtung untersucht werden.

In diesem Kapitel (2.8) soll ansatzweise der Zusammenhang zwischen beiden Richtungsbegriffen untersucht werden. Dabei sind nur recht schwache Zusammenhaenge nachzuweisen (Satz 9). Das Thema wird jedoch in Kapitel 2.10 - mit staerkeren Voraussetzungen - wieder aufgenommen (ab Lemma 12). Dort wird dann das Gegenstueck zu Satz 9 gezeigt (Satz 2.10.15) und in Satz 2.10.16 die Grundlage fuer die Konstruktion von sfather-Zeigern im Algorithmus geschaffen.

1. Bem: In den vergangenen Kapiteln hat sich gezeigt, dass man mit den Definitionen aus 2.3.1 zwar eine ganze Reihe von Konstruktionen mit Zeichenfeldern durchfuehren kann, die jedoch haeufig nicht alle gewuenschten Eigenschaften haben (z.B. 2.5.21b). Man benoetigt - wie schon in Kapitel 2.3 gegenueber Kapitel 2.2 - zusaetzliche Anforderungen an Zeichenfelder im Hinblick auf den angestrebten Algorithmus. Doch ist jetzt, nachdem die grundlegenden Konstruktionen auf Zeichenfeldern definiert sind, nicht mehr so zwingend klar, welche weiteren Eigenschaften sinnvoll, welche notwendig, welche unsinnig sind [1].

----

[1] Z.B. scheint es naheliegend zu fordern:
f lin h, g lin h ==> f lin g oder g lin f
Damit folgt aber:
zu jedem f ex. hoechstens ein g mit f xdulin g
(denn f xdulin g1,g2 ==> (oBdA.) g1 lin g2 ==> g1 ≡ g2)
d.h. man schliesst durch diese Forderung alle vom eindimensionalen Stringfall wesentlich verschiedenen Modelle aus!

Daher soll mit zusaetzlichen Forderungen insofern flexibel umgegangen werden, als sie nicht zentral zu einem neuen Katalog fuer einen eingeschraenkten Zeichenfeld-Typ zusammengefasst werden, sondern bei jedem Satz die notwendigen Zusatzforderungen explizit als Voraussetzungen aufgefuehrt werden, wie dies z.T. schon in frueheren Kapiteln praktiziert wurde.
Dieses Vorgehen erleichtert es nicht nur, einzelne Forderungen auszutauschen, falls es sich spaeter einmal als sinnvoll erweisen sollte, oder wegzulassen, wenn eine allgemeinere Version des Algorithmus sie nicht mehr benoetigt, es macht auch die Untersuchung der Zusammenhaenge zwischen ihnen einfacher.

2. Def: ZS hat die Eigenschaft der beliebigen Verschiebbarkeit
:<==> fa. $Xf \in ZS$, $p \in Xf \cap SP$, $p' \in P \cap SP$
ex. $Xf' \in ZS$ mit $Xf' \equiv Xf$ und $Iso(Xf,Xf')(p) = p'$
ZS heisst dann auch (beliebig) verschiebbar.
Anschaulich: Xf wird so verschoben, dass p mit p' zur Deckung kommt.

Bsp: Im Beispiel (a) zu Def 2.3.1 ist ZS stets beliebig verschiebbar:
waehle $Xf' := Xf + (p'-p)$

3. Def: ZS heisst AZS-eindeutig
:<==> fa. $p \in SP$ ex. hoechstens ein $Xf \in AZS$
mit $st(Xf) = p$

4. Bem: Sei ZS AZS-eindeutig, dann gilt
(a) fa. $p \in SP$ ex. genau ein $Xf \in AZS$ mit $st(Xf)=p$
(b) $Xf \in AZS$, $Xg \in ZS$, $st(Xf)=st(Xg)$ ==> Xf lin Xg

Bew: (a) nach 2.5.8 ex. zu jedem $p \in SP$ mindestens ein AZS
(b) sei $Xh \in ZS$ minimal mit $Xh \subset Xg$ ==> $Xh \in AZS$
==> $Xh = Xf$ ==> $Xf \subset Xg$

5. Lem: $Xf \in AZS$, $Xg \equiv Xf$ ==> $Xg \in AZS$

Bew: ist $Xg' \subset Xg$, $st(Xg')=st(Xg)$ fuer ein $Xg' \in ZS$,
so folgt nach 2.3.1e $Xf' := Iso(Xg,Xf)[Xg'] \in ZS$,
$st(Xf') = Iso(Xg',Xf')(st(Xg')) = Iso(Xg,Xf)(st(Xg)) = st(Xf)$
und $Xf' \subset Xf$,
also $Xf' = Xf$, da $Xf \in AZS$
==> $Xg' = Iso(Xf,Xg)[Xf'] = Iso(Xf,Xg)[Xf] = Xg$

6. Satz: Sei ZS beliebig verschiebbar und AZS-eindeutig,
dann sind alle Elemente von AZS aequivalent bzgl. $\equiv$

Bew: Sei $Xf,Xg \in AZS$, sei $p := st(Xf)$, $q := st(Xg)$,
Ann: nicht $Xf \equiv Xg$,
==> $Xf \not\equiv Xg$ ==> $p \neq q$ (wegen AZS-Eindeutigkeit)
wegen der Verschiebbarkeit ex. zu p und st(Xg) ein $Xf' \equiv$
Xf' mit $Xf' \equiv Xf$ und $Iso(Xf,Xf')(st(Xf)) = st(Xf')$
$st(Xg) = Iso(Xf,Xf')(p)$

$Xf' \in AZS$ nach 5.
nach 4b ist $Xf' \subset Xg$, da $Xf' \in AZS$
und $Xg \subset Xf'$, da $Xg \in AZS$,
also $Xg = Xf' \equiv Xf$

7. Lem: Sei $f,g \in ZF$,
f xdulin g, $p \in Xg \setminus Xf$, ==> st(f) xvor p

Bew: nach Def von vor

8. Lem: Zu $Xf, Xg \in ZS$ mit Xf xdulin Xg
ex. $p \in Xf$, $q \in Xg \setminus Xf$ mit p xvor q

Bew: Sei p := st(Xf), $q \in Xg \setminus Xf$ beliebig, dann folgt die Beh.

9. Satz: Zu $Xf, Xg \in ZS$ mit Xf xdulin Xg, Xg zusammenhaengend
ex. $p \in Xf$, $q \in Xg \setminus Xf$ mit p xuvor q

Bew: nach 8. ex. $p' \in Xf$, $q' \in Xg \setminus Xf$ mit p' xvor q'
sei p maximal bzgl. vor mit $p \in Xf$, p vor q'
sei q minimal bzgl. vor mit $q \in Xg \setminus Xf$, p vor q
dann ist p xuvor q, denn ist p vor r vor q (vor q'), so:
Fall 1: $r \in Xf$ ==> p neben r, da p maximal
Fall 2: $r \in Xg \setminus Xf$ ==> q neben r, da q minimal
Fall 3: $r \notin Xg$ ==> W.! zu g zusammenhaengend

Mit den staerkeren Voraussetzungen von Kapitel 2.10 werden zu diesem Thema weitere Saetze gezeigt (2.10.12-16).

KAPITEL 2.9 Das Richtungskonzept

1. Bem: Um den Prefix-Tree-Algorithmus aus dem ersten Teil zu verallgemeinern, muss der eindimensionale Fall im allgemeinen Fall wiedergefunden werden. Dazu wird der Begriff der Richtung axiomatisch gefordert und definiert, was es heisst, ein Zeichenfeld in eine bestimmte Richtung auszudehnen. Im eindimensionalen Fall gab es nur eine Richtung, in die ein String ausgedehnt werden konnte. Jetzt sollen zu gegebenem Xf die Richtungen von Xf korrespondieren mit den Xg mit Xf xdulin Xg, d.h. mit den moeglichen Resultaten eines elementaren "Ausweitungsschritts" (Def 2).

Waehrend die bisherige Darstellungsart von Zeichenfeldern als "deskriptiv" bezeichnet werden kann, da sie nicht durch Operationen und Gleichungen, sondern durch die Menge ZS selbst und die Aequivalenzrelation ≡ beschrieben wurden, kann die Darstellung durch Richtungen als "operativ" bezeichnet werden. Das Erweitern eines Zeichenfelds in einer Richtung entspricht der Multiplikation auf A* im eindimensionalen Fall (0.3.1.2).

Die deskriptive Darstellung hat den Vorzug, dass Modelle relativ leicht definiert werden koennen. (Das wird deutlich, wenn man den Versuch macht, "Strings mit Luecken" durch Konkatenations-Operationen zu definieren [1], und dies mit der deskriptiven Darstellung im Beispiel zu 2.3.1f vergleicht.)
Der Nachteil ist dafuer, dass die Struktur der Menge ZF sich nicht direkt aus der Darstellung ergibt.

2. Def: ZS heisst Zeichenfeld-Strukturmenge mit Richtungen
:<==> zu ZS ist eine Menge R von Richtungen gegeben, die folgende Eigenschaften hat:
(a) fa. Xf ∈ ZS ex. eine endliche Menge Rf c R,
sodass zu jedem r ∈ Rf genau ein Xg ∈ ZS ex.
mit Xf xdulin Xg
(Schreibweise: Xg = Xf * r ,
fuer Xf * r * .... * r kurz: $Xf * r^i$)
(i Faktoren)
(b) fa. Xf,Xg ∈ ZS mit Xf c Xg ist Rf c Rg
(c) fa. Xf ∈ ZS, r1,r2,s1,s2 ∈ R gilt
Xf * r1 * r2 = Xf * s1 * s2 <==>
r1=s1, r2=s2 oder r1=s2, r2=s1

----

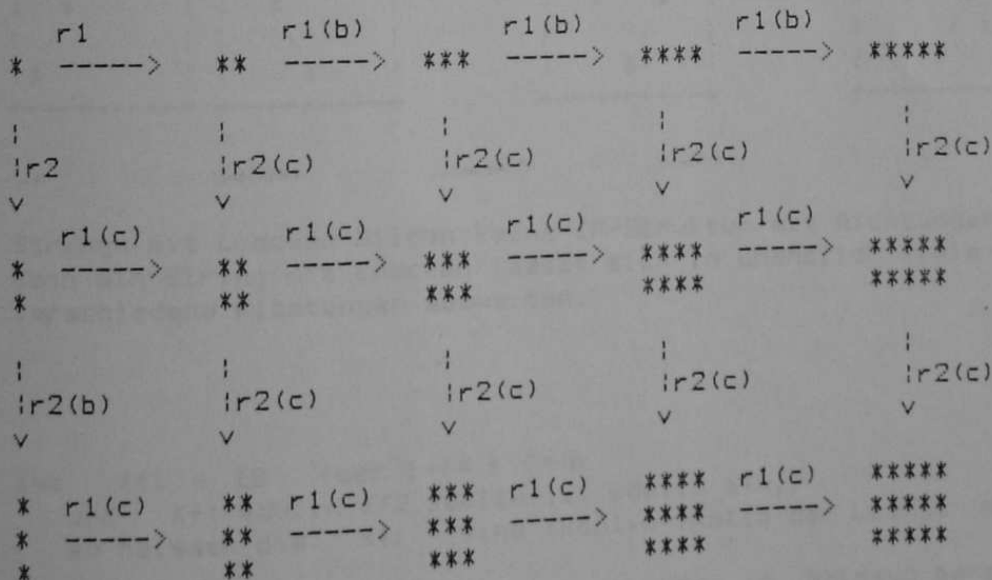
[1] etwa: $u *^i v$ = haenge v an der i.ten Luecke von u an,
Bsp: abc...def $*^1$ xy = abcxy.def
abc...def $*^2$ xy = abc.xydef
abc...def $*^3$ xy ist nicht definiert
abc...def $*^4$ xy = abc...defxy
abc...def $*^5$ xy = abc...def.xy
abc...def $*^1$ xyz...pqr = abcxyzdefpqr

3. Bem: Rechtfertigung:

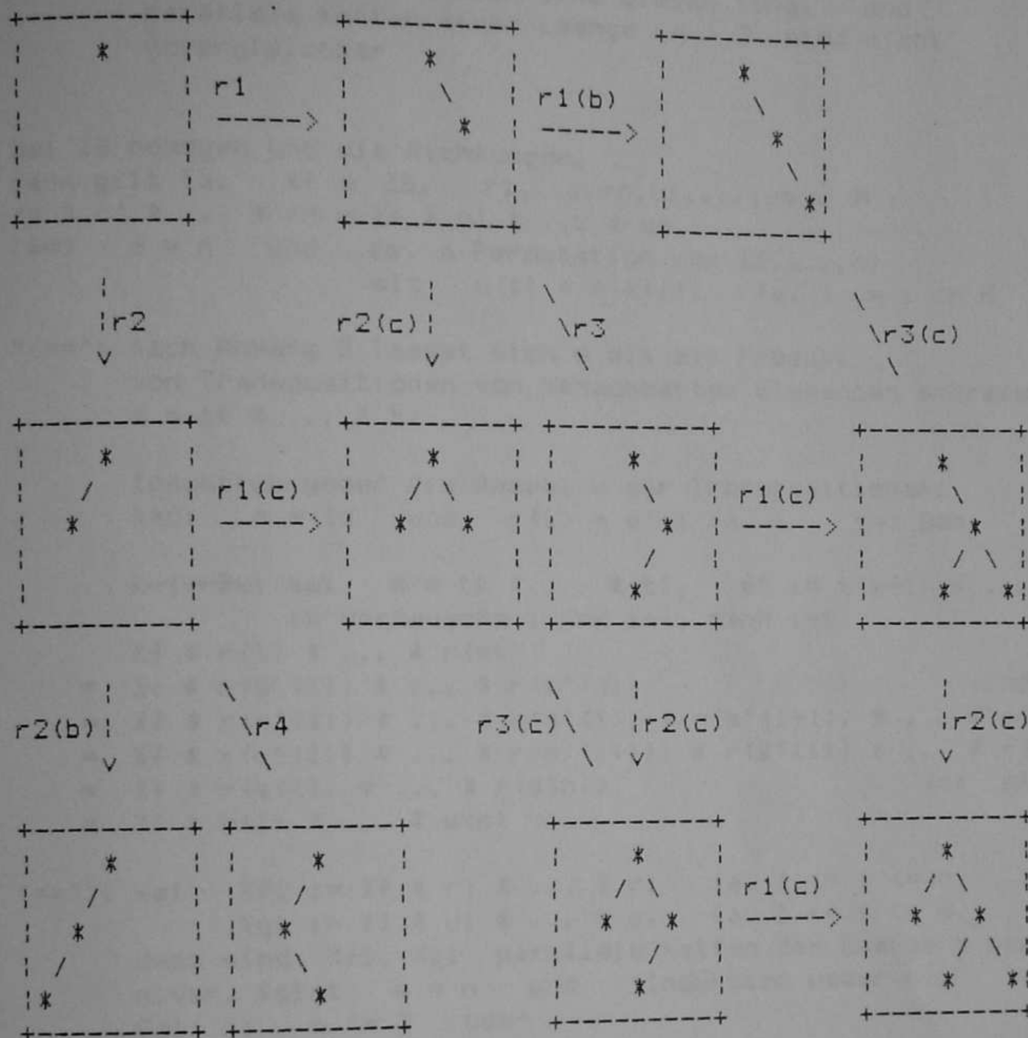
(b)
Es soll moeglich sein, ein Zeichenfeld in einer Richtung in mehreren Schritten zu "erweitern" (z.B. solange, bis es nur einmal in einem anderen vorkommt), ohne dass die Richtung ploetzlich "verschwindet".

(c)
Die Erweiterungsschritte sollen "prim" sein, d.h. sich nicht weiter zerlegen lassen. Das motiviert "==>", dann waere
Xf * r1 * r2 = Xf * s1 * s2   mit   s1 ≠ r1 ≠ s2 ≠ r2 ≠ s1,
so ist es naheliegend, kleinere Unterschritte t1,...,tn anzunehmen, mit denen etwa
s1 = t1 * ... * ti,   s2 = t(i+1) * ... * tn,
r1 = t1 * ... * tj,   r2 = t(j+1) * ... * tn
fuer geeignete   1 <= i,j <= n   waere.
Andererseits ("<==") sollen die Erweiterungsschritte "unabhaengig" voneinander sein, d.h. es soll egal sein, in welcher Reichenfolge sie ausgefuehrt werden. Im Idealfall sind die Mengen der neu dazukommenden Positionen disjunkt.

Bsp: (in Klammer jeweils die Regel, nach der die Benennung der Richtung erfolgt ist)
Rechtecke:



Binaere Baeume (nur skizziert):



Strings mit Luecken bilden keine ZF-Struktur mit Richtungen, denn ein String mit Luecken laesst sich in unendlich viele verschiedene Richtungen ausweiten.

4. Def: Ist Xfi ∈ ZS fuer 1 <= i <= n
und Xf1 xdulin Xf2 xdulin ... xdulin Xfn,
so heissen die Xfi eine (xdulin-)Kette der Laenge n

Zwei Ketten Xfi, Xgi der Laenge n bzw. m heissen parallel,
wenn Xf1 = Xg1 und Xfn = Xgm,

sie heissen unvergleichbar,
wenn Xfi unvergleichbar mit Xgj bzgl. c fa. 1<i<n, 1<j<m
(d.h. Xf1, Xg1, Xfn, Xgm koennen durchaus vergleichbar sein)

5. Def: ZS heisst homogen
:<==> alle parallelen Ketten sind gleich lang, und
parallele Ketten einer Laenge n > 3 sind nicht
unvergleichbar

6. Satz: Sei ZS homogen und mit Richtungen,
dann gilt fa. Xf ∈ ZS, r1,...,rn,u1,...,um ∈ R :
Xf * r1 * ... * rn = Xf * u1 * ... * um
<==> m = n und ex. s Permutation von {1,...,n}
mit u(i) = r(s(i)) fa. 1 <= i <= n

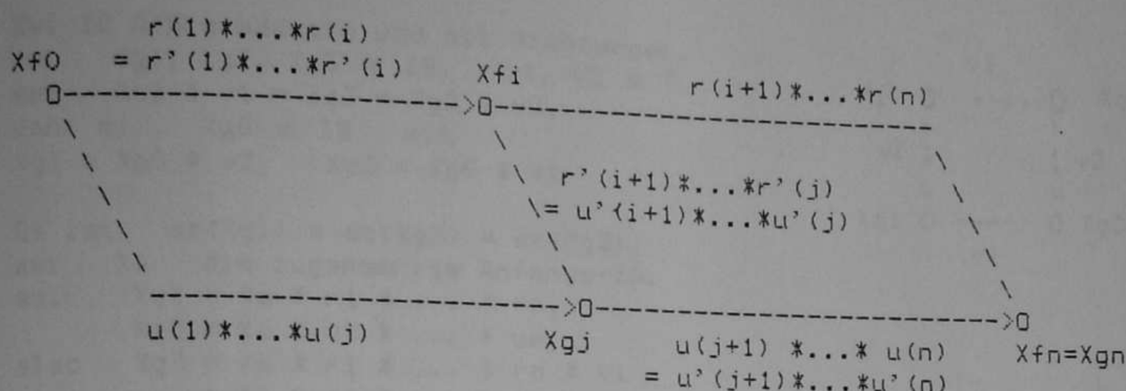
Bew: "<==": nach Anhang 2 laesst sich s als ein Produkt
von Transpositionen von benachbarten Elementen schreiben:
s = tk * ... * t1

Induktion ueber die Anzahl k der Transpositionen:
k=0: s = Id und r(i) = u(i) fa. i ==> Beh.

k-1 ⟶ k: sei s = tk * ... * t1, s' := t(k-1) * ... * t1,
tk vertausche i und i+1, dann ist
Xf * r(1) * ... * r(n)
= Xf * r(s'(1)) * ... * r(s'(n)) (Ind.vor.)
= Xf * r(s'(1)) * ... * r(s'(i)) * r(s'(i+1)) * ... * r(s'(n))
= Xf * r(s'(1)) * ... * r(s'(i+1)) * r(s'(i)) * ... * r(s'(n))
= Xf * r(s(1)) * ... * r(s(n)) (da s=tk*s')
= Xf * u(1) * ... * u(n)

"==>": sei Xfi := Xf * r1 * ... * ri fa. 1 <= i <= n,
Xgi := Xf * u1 * ... * ui fa. 1 <= i <= m,
dann sind Xfi, Xgi parallele Ketten der Laenge n bzw. m,
n.Vor. folgt m = n und (Induktion ueber n):
Fall 1: n <= 2 oder
Fall 2: n = 3 oder
Fall 3: ex. Xfi, Xgj mit (oBdA.) Xfi c Xgj

im Fall 1 ist nichts zu zeigen,
im Fall 2 folgt die Beh. nach Def 2c,
im Fall 3 ex. nach 2.5.10 eine Kette zwischen Xfi und Xgj,
d.h. nach Def 2a ex. r'(i+1), ... , r'(k) mit
Xf * r(1) * ... * r(i) * r'(i+1) * ... * r'(k)
= Xfi * r'(i+1) * ... * r'(k)
= Xgj
= Xf * u(1) * ... * u(j)
nach Ind.vor. (da k=j<n) ex. s' Permutation von
{1,...,j} mit u(l) = r'(s'(l)) fa. 1 <= l <= j
(wobei r'(l) := r(l) fuer 1 <= l <= i sei)
andererseits ist
Xfi * r(i+1) * ... * r(n)
= Xf * r(1) * ... * r(n)
= Xf * u(1) * ... * u(n)
= Xgj * u(j+1) * ... * u(n)
= Xfi * r'(i+1) * ... * r'(j) * u(j+1) * ... * u(n)
nach Ind.vor. (da i+1 > 1) ex. s" Permutation von
{1,...,n-i} mit u'(l+i) = r(s"(l)+i) fa. 1 <=l <=n-i
(wobei u'(l) := r'(l) fuer i+1 <= l <= j
und u'(l) := u(l) fuer j+1 <= l <= n sei)



definiere nun:

```
         / s'(l)              fuer  1 <= l <= j  und  s'(l) <= i
s(l) := {  s"(s'(l)-i)+i      fuer  1 <= l <= j  und  i+1 <= s'(l) <= j
         \ s"(l-i)+i          fuer  j+1 <= l <= n
```

dann ist in jedem Fall u(l) = r(s(l)), denn:
1.Zeile: u(l) = r'(s'(l)) = r(s'(l)) = r(s(l))
2.Zeile: u(l) = r'(s'(l)) = u'(s'(l)) = u'((s'(l)-i)+i)
= r(s"(s'(l)-i)+i) = r(s(l))
3.Zeile: u(l) = u'(l) = u'((l-i)+i) = r(s"(l-i)+i)
= r(s(l))

s ist fuer alle l ∈ {1,...,n} definiert
und liefert Werte im Bereich {1,...,n} :
im 1.Fall: 1,...,j und im 2.+3.Fall: i+1,...,n
s ist surjektiv auf {1,...,n}:
sei 1 <= h <= n, finde l mit h = s(l)
ist h <= i, so ex. l ∈ {1,...,j} mit h = s'(l)
also s(l) = s'(l) = h
ist h >= i+1, so ex. l" ∈ {1,...,n-i} mit
h-i = s"(l")
ist l" <= j-i, so ex. zu (l"+i) ∈ {i+1,...,j}
ein l ∈ {1,...,j} mit l"+i = s'(l),
s(l) = s"(s'(l)-i)+i = s"(l"+i-i)+i = s"(l")+i
= h-i+i = h
ist l" > j-i, so ist j+1 <= l"+i <= n,
setze l := l"+i, dann ist
s(l) = s(l"+i) = s"(l")+i = h-i+i = h

damit ist s auch injektiv (da Quell- und Zielmenge endlich und gleichmaechtig), also die gesuchte Permutation

7. Bem: Ist f ∈ ZF gegeben, g in f, r ∈ Rg,
dann laesst sich g * r definieren durch
g * r := f | (Xg * r)
Wenn im folgenden nicht nur ZF-Strukturen, sondern auch Zeichenfelder selbst erweitert werden, so ist das in diesem Sinne zu verstehen.
Das f , auf das Bezug genommen wird, ist stets aus dem Zusammenhang eindeutig ersichtlich.

8. Satz: Sei ZS AZS-eindeutig und mit Richtungen,
sei Xg1, Xg2, Xg3 ∈ ZS, v1, v2 ∈ R,
sei Xg1 * v1 = Xg3 = Xg2 * v2,
dann ex. Xg0 ∈ ZS mit
Xg1 = Xg0 * v2, Xg2 = Xg0 * v1

```
             v1
Xg0  0 .....> 0  Xg2
     :        ¦
v2   :        ¦ v2
     v        v
Xg1  0 -----> 0  Xg3
             v1
```

Bew: Es ist st(Xg1) = st(Xg3) = st(Xg2),
sei Xa die zugehoerige Anfangs-ZS,
sei Xg1 = Xa * r1 * ... * rn,
Xg2 = Xa * u1 * ... * um,
also Xg3 = Xa * r1 * ... * rn * v1
= Xa * u1 * ... * um * v2,
nach 6. folgt m = n und es ex. eine Permutation s von {1,...,n+1}
mit u(i) = r(s(i)), dabei sei r(n+1) = v1, u(n+1) = v2,
sei j := s(n+1), d.h. r(j) = r(s(n+1)) = u(n+1) = v2,
sei t Transposition von {1,...,n+1}, die j und n vertauscht,
also r(t(n)) = r(j) = v2,
setze Xg0 := Xa * r(t(1)) * ... * r(t(n-1)),
dann ist Xg0 * v2 = Xa * r(t(1)) * ... * r(t(n-1)) * r(t(n))
= Xa * r(1) * ... * r(n) (nach 6.)
= Xg1
und Xg0 * v1 =
= Xa * r(t(1)) * ... * r(t(j)) * ... * r(t(n-1)) * r(t(n+1))
= Xa * r(1) * ... * r(j-1) * r(j+1) * ... * r(n+1)
denn r(t(n)) = r(j) fehlt, aber alle anderen
Faktoren von r(1) bis r(n+1) sind vorhanden
= Xa * r(s(1)) * ... * r(s(s^{-1}(j)-1)) *
* r(s(s^{-1}(j)+1)) * ... * r(s(n+1))
d.h. der Faktor r(j) = r(s(s^{-1}(j))) = r(s(n+1)) fehlt
= Xa * u(1) * ... * u(n)
= Xg2

Bem: In allen Faellen, in denen die Erweiterung zweier verschiedener
Zeichenfelder Xg1,Xg2 (in verschiedenen Richtungen) dasselbe
Ergebnis liefert, muss dies im Algorithmus festgestellt und
entsprechend in die Sortierstruktur eingetragen werden. Satz 8
zeigt eine Methode auf, wie man zu gegebenem
Xg1, Xg3 = Xg1 * v1 alle entsprechenden Xg2 findet:
Xg1, Xg3 = Xg1 * v1 alle entsprechenden Xg2 findet:
suche alle Xg0 xdulin Xg1, ist Xg1 = Xg0 * v2,
so ist (mit Xg2 := Xg0 * v1) Xg2 * v2 = Xg3 .
Nach dieser Methode arbeitet das Unterprogramm
"enter additional ptr links" (s. Kap. 3.1).

9. Lem: Sei ZS AZS-eindeutig und mit Richtungen,
sei f ∈ ZF, a ∈ AZF, r ∈ Ra mit
h := a * r^i mehrmals in f, h * r einmal in f,
dann gilt fa. k,k' ∈ ZF mit k xdulin k', st(k)=st(a),
k min f, k' ein f :

k laesst sich darstellen als k = a * t1 * ... * tm
mit r kommt hoechstens i mal unter den t1,...,tm vor

Bew: waere r i+1 mal unter den t1,...,tm,
oBdA. (wegen Kommutativitaet) t1 = ... = t(i+1) = r
dann folgt: k = a * t1 * ... * t(i+1) * t(i+2) * ... * tm
= a * r^{i+1} * t(i+2) * ... * tm
= h * r * t(i+2) * ... * tm
W.!, denn h * r ein f ==> k ein f

10. Lem: Sei ZS AZS-eindeutig und mit Richtungen, sei r ∈ R,
sei f ∈ ZF, g MPMR in f, a ∈ AZF mit st(a) = st(g),
es gelte fa. k,k' ∈ ZF mit k xdulin k' lin g :
k laesst sich darstellen als k = a * t1 * ... * tm
mit r kommt hoechstens i mal unter den t1,...,tm vor
dann ist g = a * u1 * ... * un
mit r kommt hoechstens i+1 mal unter den u1,...,un vor

Bew: Ann: (oBdA.) u1 = ... = u(i+2) = r
dann enthaelt aber schon g' := a * u2 * ... * un
alle k' mit k xdulin k', k min f, k' ein f, st(k)=st(g),
(sei k' = k * t(m+1)), denn:
g = k' * v1 * ... * vm' (vj ex., da k' lin g)
= a * t1 * ... * t(m+1) * v1 * ... * vm'
andererseits ist
g = a * r^{i+2} * u(i+3) * ... * un
da unter den t1,...,tm hoechstens i Faktoren r vorkommen,
koennen unter den t1,...,t(m+1) hoechstens i+1 vorkommen,
damit muss nach Satz 6 mindestens ein vj = r sein,
oBda. vm' = r,
also g' * r = g = k' * v1 * ... * v(m'-1) * r
==> g' = k' * v1 * ... * v(m'-1) (nach 15)
==> k' lin g' W.!

11. Satz: Sei ZS homogen, AZS-eindeutig und mit Richtungen,
sei f,g ∈ ZF, r ∈ R, g MPMR in f, g ∉ AZF, dann gilt
ex. h ∈ ZF mit h mehrmals in f, h * r einmal in f,
g = h * r * u1 * ... * un mit r ∉ {u1,...,un}

Bew: Sei a das AZF zu st(g),
setze h := a * r^{i} mit i maximal, sodass h min f
(waere schon a ein f, so waere g = a ∈ AZF, W.!)
dann ist h * r ein f,
also nach Def des MPMR h*r lin g, d.h. g = h * r * u1*...*un
bleibt zu zeigen, dass r ∉ {u1,...,un}
nach 9. gilt fa. k,k' mit k xdulin k', k min f, k' ein f,
st(k) = st(h)
k laesst sich darstellen als k = a * t1 * ... * tm
mit r kommt hoechstens i mal unter den t1,...,tm vor
nach 10. folgt dann r ∉ {u1,...,un}

12. Satz: Sei ZS AZS-eindeutig und mit Richtungen,
sei p ∈ SP, sei Xa,Xg1,...,Xgn ∈ ZS mit st(Xgi) = p fa. i
sei st(Xa) = p, (r(i,j) ∈ R)
sei Xgi = Xa * r(1,i) * ... * r(li,i) fa. i
dann ex. genau ein kleinstes Xh ∈ ZS, das alle Xgi enthaelt,
und wenn r(j1,i) ≠ r(j2,k) fa. i ≠ k, j1,j2 beliebig,
so ist weiter Xh = Xa * r(1,1) * ... * r(l1,1) *
* r(1,2) * ... * r(l2,2) * ...
... * r(n,1) * ... * r(n,ln)

Bew: Induktion ueber n:
n=1: trivial
n-->n+1: nach Ind.vor. ex. genau ein kleinstes Xh' , das alle
Xg1,...,Xg(n-1) enthaelt,
jedes Xh , das alle Xg1,...,Xgn enthaelt, muss Xh' und
Xgn enthalten
d.h. wenn die Beh. fuer n=2 gilt, so auch fuer n+1

n=2: sei Xg1 = Xa * r1 * ... * rn, Xg2 = Xa * u1 * ... * um
oBdA. sei ri ≠ sj fa. i,j
(sonst waehle Xa' := Xa * ri = Xa * sj,
Xg1,Xg2 entsprechend)
waehle Xh := Xa * r1 * ... * rn * u1 * ... * um,
dann ist Xg1 lin Xh, Xg2 lin Xh,
und ist fuer ein Xk ∈ ZS Xg1 lin Xk, Xg2 lin Xk,
so folgt Xk = Xa * r1 * ... * rn * t1 * ... * tn'
= Xa * u1 * ... * um * v1 * ... * vm'
d.h. die ui kommen unter den ti vor
==> Xh = Xa * r1 * ... * rn * u1 * ... * um
lin Xk

13. Korr: Sei ZS AZS-eindeutig und mit Richtungen, sei f ∈ ZF,
dann ex. hoechstens ein MPMR zu jeder Position von f

Bew: folgt direkt aus 12.:
seien g1,...,gn alle ZF, die an p starten und einmal in f
vorkommen und wo zu gi' mehrmals in f fuer ein gi' xdulin gi

14. Lem: Sei ZS mit Richtungen,
sei f ∈ ZF, g MPMR von f, h min f, h lin g,
dann ist g = h * u1 * ... * un,
wobei unter den s1,...,sn alle Richtungen r von h vorkommen

Bew: fa. h' = h * r ist nach Def des MPMR h' lin g

15. Lem: Sei ZS AZS-eindeutig und mit Richtungen,
r1,...,rn,s1,...,sn,r ∈ Ra,
sei Xa ∈ AZS,
sei Xa * r1 * ... * rn * r = Xa * s1 * ... * sn * r
sei Xa * r1 * ... * rn * r = Xa * s1 * ... * sn
dann ist auch Xa * r1 * ... * rn = Xa * s1 * ... * sn

Bew: nach 8. ex. zu Xg1 := Xa * r1 * ... * rn
und Xg2 := Xa * s1 * ... * sn
ein Xg0 mit Xg1 = Xg0 * r, Xg2 = Xg0 * r
==> Xg1 = Xg2

KAPITEL 2.10 Das End-Abschnitts-Konzept

1. Bem: Die Relation "ist Suffix von" spielt eine wichtige Rolle bei der Untersuchung von Zeichenfeldern; die Eigenschaften, die fuer diese Relation gelten, sind im Gegensatz dazu noch sehr unzureichend (vgl. 2.5.21b). In diesem Kapitel wird eine einfache Bedingung untersucht, die sehr schoene Eigenschaften von "ist Suffix von" liefert. Ausserdem liefert sie weitere Ergebnisse zu dem in Kapitel 2.8 begonnenen Thema.

2. Def: Sei ZS links-erweiterbar, $Xf \in ZS$, $p \in Xf$,
dann heisst $Xf/p := \{q \mid q \in Xf,\ p \text{ vor } q\}$
End-Abschnitt von Xf ab p

i.a. ist Xf/p nur Pseudo-ZF-Struktur

3. Def: ZS heisst End-Abschnitts-vollstaendig
:<==> ZS links-erweiterbar und
fa. $Xf \in ZS$, $p \in Xf$ ist $Xf/p \in ZS$

Bsp: Rechtecke, Trees und Strings mit Luecken (vgl. Bsp zu 2.3.1) sind endabschnitts-vollstaendig:
$([a,b]\times[c,d]) / (p,q) = [p,b]\times[q,d]$,
ist T Baum, so ist T/p der Unterbaum von T mit der Wurzel p,
ist Xf String mit Luecken, so ist $Xf/p = \{q \mid q \in Xf,\ q >= p\}$

Fuer den Rest des Kapitels 2.10 wird angenommen, dass ZS Endabschnitts-vollstaendig sei.

4. Lem: Ist $Xf \in ZS$, $p,q \in Xf$, p neben q,
so ist $Xf/p = Xf/q$

Bew: $r \in Xf/p$ ==> $r \in Xf$, p vor r
==> $r \in Xf$, q vor r
==> $r \in Xf/q$
analog zeigt man $Xf/q \subset Xf/p$

5. Lem: st(Xf/p) neben p

Bew: $st(Xf/p) \in Xf$ ==> p vor st(Xf/p) (Def Xf/p)
$p \in Xf/p$ ==> st(Xf/p) vor p (Def vor)

Bem: Es laesst sich also zwar fuer jedes $p \in Xf$ die Konstruktion Xf/p durchfuehren, man erhaelt dadurch aber nicht immer eine ZF-Struktur, die bei p startet.

6. Lem: Xf/p ist Suffix von Xf

Bew: Ann: ex. Xh mit Xf/p xlin Xh in Xf
==> ex. $q \in Xh$ mit $q \notin Xf/p$
==> nicht p vor q
==> st(Xh) = st(Xf/p) nicht vor q (da $q \in Xf$)
W.! zur Def von vor (nach 5.)
(da $q \in Xh$)

7. Lem: Xf/p ist der einzige Suffix in Xf ab st(Xf/p)

Bew: Ann: ex. Xg mit $Xg \neq Xf/p$, st(Xg) = st(Xf/p) = p (oBdA.)
==> ex. $q \in Xg \setminus (Xf/p)$
==> st(Xg) = p nicht vor q
W.! zur Def von vor (da $q \in Xf$)
(da $q \in Xg$)

8. Lem: (Xf/p)/q = Xf/q fuer p vor q

Bew: sei Xg := Xf/p,
$r \in Xg/q$ ==> $r \in Xg$, q vor r
==> $r \in Xf$, q vor r
==> $r \in Xf/q$

$r \in Xf/q$ ==> $r \in Xf$, q vor r
==> $r \in Xf$, p vor r, q vor r
==> $r \in Xf/p = Xg$, q vor r
==> $r \in Xg/q$

Bem: Die Lemmata 6, 7 und 8 sagen aus, dass auf den Positionen, an denen ein Xf/p startet, der Suffix eindeutig bestimmt ist (naemlich Xf/p) und die Relation "ist Suffix von" transitiv ist.

9. Lem: Es gebe fa. $Xf \in ZS$, $p \in Xf$ genau einen Suffix Xg in Xf ab p
dann gilt: Xh in Xf, st(Xh) = p ==> Xh lin Xg

Bew: Sei Xh' maximal mit Xh lin Xh' in Xf,
dann ist Xh' Suffix von Xf ab p,
also Xh' = Xg
==> Xh lin Xh' = Xg

10. Satz: neben = Id <==> fa. $Xf \in ZS$, $p \in Xf$ ex. genau ein Suffix in Xf ab p

Bew: "==>": aus der Endabschnitts-Vollstaendigkeit folgt die
Existenz und Eindeutigkeit des Suffix' ab allen Positionen
st(Xf/p) nach Lemma 6+7,
ist neben = Id, so folgt nach 5: st(Xf/p) = p
und damit gilt die Beh.

"<==": Ann: ex. p,q ∈ SP mit p neben q, p ≠ q,
sei st(Xf/p) = p', dann ist p' neben q, p' neben p,
und p' ≠ p oder p' ≠ q,
in jedem Fall ex. also ein p' mit p' = st(Xf/p'),
p' neben q, p' ≠ q,
deshalb sei oBdA. p' = p,
sei Xg Suffix von Xf ab q,
dann ist Xg ≠ Xf/p = Xf/q (da st(Xf/p) ≠ st(Xg))
Xf/p laesst sich nach links bis q erweitern
(da q vor st(Xf/p)), d.h. ex. Xg' mit st(Xg') = q,
Xf/p Suffix von Xg', und Xg' ⊂ Xf
nach 9. folgt Xg' lin Xg (da q ∈ Xf)
also Xf/p in Xg' in Xg (da Xg Suffix in Xf)
umgekehrt laesst sich Xg nach links bis p erweitern
(da p vor q = st(Xg)), d.h. ex. Xg" mit st(Xg") = p,
Xg Suffix von Xg", und Xg" in Xf
nach 9. folgt Xg" lin Xf/p (da p ∈ Xf)
also Xg in Xg" in Xf/p (da Xf/p Suffix in Xf)
d.h. Xg = Xf/p, W.!

11. Satz: neben = Id, (ZS Endabschnitts-vollstaendig)
==> "ist Suffix von" ist transitiv

Bew: Xg Suffix von Xf, Xh Suffix von Xg
==> Xg = Xf/p, Xh = Xg/q (n.Vor. mit 5.)
==> Xh = (Xf/p)/q = Xf/q (nach 8., da st(Xg)=p, q ∈ Xg)
==> Xh Suffix von Xf

Es folgen einige Saetze, die inhaltlich noch zum Kapitel 2.8 gehoeren, aber erst unter der Bedingung der Endabschnitts-Vollstaendigkeit bewiesen werden koennen.

12. Lem: Sei neben = Id,
sei Xf,Xf' ∈ ZS, p,q ∈ Xf, Xf ≡ Xf', dann gilt
p vor q ==> Iso(Xf,Xf')(p) vor Iso(Xf,Xf')(q)

Bew: st(Xf/p) = p, q ∈ Xf/p n.Vor., Xf/p ⊂ Xf
==> st(Iso(Xf,Xf')[Xf/p]) = Iso(Xf,Xf')(st(Xf/p))
= Iso(Xf,Xf')(p)
q ∈ Xf/p ==> Iso(Xf,Xf')(q) ∈ Iso(Xf,Xf')[Xf/p]
also Iso(Xf,Xf')(p) vor Iso(Xf,Xf')(q)

13. Korr: Sei neben = Id,
sei Xf,Xf' ∈ ZS, p,q ∈ Xf, Xf ≡ Xf', dann gilt
p vor q <==> Iso(Xf,Xf')(p) vor Iso(Xf,Xf')(q)

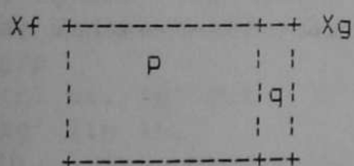
Bew: folgt aus 12., da Iso(Xf,Xf') = Iso(Xf',Xf)$^{-1}$

14. Korr: Sei neben = Id, Xf,Xf' ∈ ZS, Xf ≡ Xf',
dann gilt
(a) Xf zusammenhaengend <==> Xf' zusammenhaengend,
(b) ist Xf zusammenhaengend, p,q ∈ Xf, so gilt und
p uvor q <==> Iso(Xf,Xf')(p) uvor Iso(Xf,Xf')(q)

Bew: (a) "==>":
sei p, q ∈ Xf', p vor r vor q fuer ein r ∈ SP, dann ist
Iso(Xf',Xf)(p) vor Iso(Xf',Xf)(r) vor Iso(Xf',Xf)(q)
==> Iso(Xf',Xf)(r) ∈ Xf (13)
==> r ∈ Xf'

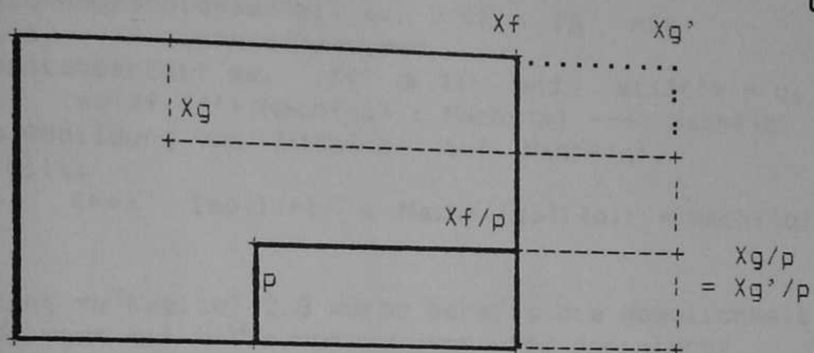
(b)
p uvor q ==> p vor q,
(p vor r vor q ==> p = r oder q = r)
==> Iso()(p) vor Iso()(q), (13 ==>)
(Iso()(p) vor r' vor Iso()(q)
==> p vor Iso()$^{-1}$(r') vor q (13 <==)
da r' ∈ Xf' wegen
Xf' zusammenhaengend
==> p = Iso()$^{-1}$(r') oder q = Iso()$^{-1}$(r')
==> Iso()(p) = r' oder Iso()(q) = r'
)
==> Iso()(p) uvor Iso()(q)
umgekehrt analog

15. Satz: Sei neben = Id, ZS AZS-eindeutig,
verschiebbar und halb abgeschlossen gegen Vereinigung,
sei p,q ∈ SP, dann gilt
p xvor q ==> ex. Xf,Xg ∈ ZS mit Xf xdulin Xg,
p ∈ Xf, q ∈ Xg \ Xf



Bew: Sei Xf' ∈ AZS beliebig,
sei p' maximal bzgl. vor , p' ∈ Xf',
sei Xf" ≡ Xf' mit Iso(Xf',Xf")(p') = p
dann ist p maximal bzgl. vor in Xf" nach 13,14
p xvor q ==> ex. Xh mit st(Xh)=p, q ∈ Xh,
sei Xg" minimal bzgl. dlin mit st(Xg") = st(Xf"), q ∈ Xg",
(Xg" ex., da st(Xf") vor p vor q)
es ist Xf" dlin Xg" nach 2.8.5 und 2.8.4b,
und Xf" ≠ Xg", da q ∈ Xg", q ∉ Xf"
(wegen p xvor q, aber p maximal in Xf" bzgl. vor)
nach 2.5.10 ex. eine Kette
Xf" = Xf0 xdulin Xf1 xdulin ... xdulin Xfn = Xg",
(i ex., da q ∉ Xf0, q ∈ Xfn)
sei i maximal mit q ∉ Xfi, Xg := Xf(i+1) :
dann ist mit Xf := Xfi,
p ∈ Xf, q ∈ Xg \ Xf, Xf xdulin Xg

16. Satz: Vor: (a) ZS Endabschnitts-vollstaendig, (also auch links-
erweiterbar) und mit Richtungen,
(b) neben = Id,
(c) fa. Xf,Xg ∈ ZS, p ∈ Xf n Xg n SP
mit st(Xf) = st(Xg), Xf/p xlin Xg/p
ex. Xg' ∈ ZS mit Xf xlin Xg' und Xg'/p = Xg/p
und: wenn Xf,Xg c Xh, so auch Xg' c Xh



Beh: fa. Xh,Xf ∈ ZS, p ∈ Xf n SP gilt
(a) Xf xdulin Xh ==> Xf/p dulin Xh/p, und
(b) Xf/p ≠ Xh/p <==> ex. q ∈ Xh \ Xf mit p vor q

Bew: (a)
Xf/p in Xh/p: q ∈ Xf/p ==> p vor q, q ∈ Xf
==> p vor q, q ∈ Xh
==> q ∈ Xh/p
Xf/p lin Xh/p: da st(Xf/p) = p = st(Xh/p) n.Vor. (b)
Xf/p ulin Xh/p:
waere Xf/p xlin Xg" xlin Xh/p fuer ein Xg" ∈ ZS,
so laesst sich Xg" nach links bis st(Xf) erweitern,
(da st(Xf) = st(Xh) vor p)
d.h. ex. Xg mit Xg" Suffix von Xg, st(Xg) = st(Xf), Xg in Xh,
wegen der Endabschnittsvollstaendigkeit und neben=Id folgt
Xg" = Xg/p
n.Vor. (c) ex. Xg' mit Xf xlin Xg', Xg'/p = Xg/p, Xg' in Xh,
also Xg' lin Xh,
Xg' ≠ Xh, da Xg/p = Xg'/p Suffix in Xg',
aber nicht in Xh (da Xg/p = Xg" xlin Xh/p)
also Xf xlin Xg' xlin Xh, W.! zu Xf dulin Xh

(b)
wegen Xf/p lin Xh/p gilt:
Xf/p ≠ Xh/p <==> ex. q ∈ (Xh/p) \ (Xf/p)
<==> ex. q ∈ Xh mit p vor q, q ∉ Xf

Bsp: Rechtecke, Trees und Strings mit Luecken erfuellen die
Voraussetzung (c) in Satz 16:
bei Rechtecken setze Xg' := Xg;
bei Trees setze Xg' := Xf ∪ Xg,
dann ist Xg' ∈ ZS, da st(Xg) = st(Xf),
nach dem Beweis von Lemma 19 gilt dann (c);
bei Strings mit Luecken gilt (c) nach Lemma 19.

17. Lem: Sei ZS beliebig verschiebbar und z-halb abgeschlossen gegen Vereinigung (sowie endabschnitts-vollstaendig),
sei p,q ∈ SP mit Nachf(p),
dann ist auch Nachf(q) endlich
und es gilt |Nachf(p)| = |Nachf(q)|

Bew: Zu jedem ri ∈ Nachf(p) ex. Xfi ∈ ZS mit
st(Xfi) = p, ri ∈ Xfi,
wegen der Z-Halb-Abgeschlossenheit ex. Xf ∈ ZS mit
Xfi c Xf fa. i, Xf zusammenhaengend,
wegen der Verschiebbarkeit ex. Xf' ≡ Xf mit st(Xf') = q,
dann definiert Iso(Xf,Xf')|Nachf(p) : Nachf(p) ---> Nachf(q)
eine bijektive Abbildung von Nachf(p) auf Nachf(q),
denn nach 14b gilt:
ri ∈ Nachf(p) <==> Iso()(ri) ∈ Nachf(Iso()(p)) = Nachf(q)

18. Bem: In der Einleitung zu Kapitel 2.8 wurde bereits die Moeglichkeit erwaehnt, Richtungen mit Hilfe von xuvor zu definieren.
Lemma 17 sagt in diesem Zusammenhang, dass zu jeder Position gleich viele solcher Richtungen existieren.

19. Lem: Ist ZS abgeschlossen gegen Vereinigung und Endabschnitts-vollstaendig,
so gilt schon die Voraussetzung (c) in Satz 16.

Bew: Setze Xg' := Xf u Xg,
dann ist Xf lin Xg' nach 2.4.6a,
Xg'/p = Xg/p, denn:

```
    q ∈ Xg/p   ==>  q ∈ Xg,     p vor q
               ==>  q ∈ Xg',    p vor q
               ==>  q ∈ Xg'/p
    q ∈ Xg'/p  ==>  (q ∈ Xf   oder  q ∈ Xg),   p vor q
               ==>  q ∈ Xf/p   oder  q ∈ Xg/p
               ==>  q ∈ Xg/p                 (da Xf/p in Xg/p)
Xf ≠ Xg',  denn sonst waere   Xf/p = Xg'/p = Xg/p,   W.!
```

```
***************************************************************
* TEIL 3     Ein Algorithmus fuer den mehrdimensionalen Fall  *
***************************************************************
```

KAPITEL 3.1 Beschreibung des Algorithmus

3.1.1 Die verwendete Datenstruktur

Als Voraussetzung fuer den Algorithmus sind alle bisher definierten zusaetzlichen Bedingungen an ZS erforderlich, naemlich:

ZS mit Richtungen (2.9.2),
es gebe nur endlich viele s-Richtungen (analog zur Def 2.9.2 sei zu p ∈ SP jedem q ∈ Nachf(p) eine s-Richtung r zugeordnet, vgl. die Einleitung zu Kap 2.8 sowie Lemma 2.10.17),
homogen (2.9.5),
beliebig verschiebbar (2.8.2),
AZS-eindeutig (2.8.3)
Endabschnitts-vollstaendig (2.10.3), also auch linkserweiterbar
und neben = Id .

Ausserdem sei

- eine totale Ordnung <= definiert mit
  p vor q ==> p <= q
  (d.h. <= ist eine Verfeinerung von vor);
- sei eine totale Ordnung auf der Menge alle Zeichenfeld-Zuwaechse definiert (d.h. auf der Menge
  {h|(Xh\Xg) | g,h ∈ ZF, h xdulin g}); sowie
- eine totale Ordnung auf der Menge aller Anfangs-Zeichenfelder.

Diese Voraussetzungen werden fuer den Rest des Kapitels als gueltig angenommen.

Ist f ∈ ZF das Eingabe-Zeichenfeld und existiert zu jeder Position von f ein MPMR, so laesst sich die Sortierstruktur wie folgt charakterisieren:

(a) es gibt einen ausgezeichneten Knoten root,
(b) fuer jedes Anfangs-ZF g in f gibt es einen Knoten ng = follow(g) und einen Pfeil von root nach ng
(c) g ∈ dPrf(h) fuer ein MPMR h von f
  <==> ex. ein korrespondierender Knoten ng = follow(g)
(d) ex. ein Pfeil.
  von einem Knoten ng zu einem Knoten nh
  wenn g xdulin h,
  der Pfeil ist mit h|(Xh\Xg) markiert
(e) von einem Knoten ng ex. ein sfather-Zeiger zu einem anderen Knoten nh , wenn st(g) xuvor st(h) und h Prefix des MPMR von st(h)

Ein Beispiel fuer eine Sortierstruktur siehe 3.1.3 .

3.1.2 Der Algorithmus

Sei f wie in 3.1.1 gegeben. Der Algorithmus geht die Positionen von f in der durch die totale Ordnung <= gegebenen Sortierreihenfolge durch, konstruiert fuer jede Position einen vorlaeufigen MFMR und traegt ihn (und seine Prefixe) in die Sortierstruktur ein.

Gab es beim eindimensionalen Algorithmus (Kap 1.6) einen Zeiger nc auf den gerade verarbeiteten Knoten, so gibt es jetzt eine endliche Menge cthis solcher Zeiger, die - wie nc - angeben, wieweit die laufende Position schon in die Struktur einsortiert ist.

Ist ng ein Knoten, auf den ein Zeiger aus cthis zeigt, und r eine Richtung von g = label(ng), so koennen bezueglich der Ausweitung von g in Richtung r die analogen vier Faelle wie in Kap 1.6 auftreten:

(Sei h := g * r = f!(Xg*r))

Fall 1:
fuer h ist schon ein Knoten nh in der Sortierstruktur vorhanden
(erkennbar daran, dass von ng eine Kante ausgeht, die mit f!(Xh\Xg) markiert ist),
dann verfolge den vorhandenen Weg weiter (in Richtung r):
nimm nh zu cthis dazu [1]

Fall 4:
fuer h ist noch kein Knoten vorhanden, aber es gibt einen in Richtung r,
dann ist h ein (vorlaeufiger) Einzel-Posid von f,
trage einen Knoten nh fuer h in die Sortierstruktur ein,
nimm nh zu cthis dazu [2]

----

[1] Aus technischen Gruenden werden die neu zu cthis aufgenommenen Knoten zunaechst nach cnext aufgenommen; sind alle (alten) Knoten aus cthis abgearbeitet, wird cnext komplett nach cthis uebernommen.

[2] Im eindimensionalen Fall konnte man aufhoeren, weiter auszuweiten, wenn man (wie in Fall 4) einen Einzel-Posid erreicht hatte. Jetzt muss man trotzdem weiter ausweiten, solange, bis sicher ist, dass es nicht noch einen anderen Einzel-Posid gibt, der in Richtung r weiter ausgedehnt ist als h (vgl. Lemma 2.9.10). Diese Information erhaelt der Algorithmus aus einem Array rk , das fuer jeden Knoten und fuer jede Richtung r angibt, ob in dieser Richtung noch weiter ausgeweitet werden muss. Einzelheiten siehe 3.2.3-6 .

Fall 3:
in Richtung r ist ueberhaupt kein weiterer Knoten vorhanden,
und die Zuwaechse sind gleich (d.h. $g'*r \equiv g*r$, wobei g' das
Zeichenfeld ist, das an der aktuell einzusortierenden Position beginnt und
fuer das gilt $g' \equiv g$, aber natuerlich $g' \neq g$),
dann muss ein neuer Knoten nh fuer $g*r$ erzeugt werden und zu cthis
dazugenommen werden.

Fall 2:
in Richtung r ist noch kein Knoten vorhanden,
und die Zuwaechse von g' und g sind verschieden
(d.h. $g \equiv g'$, aber nicht $g*r \equiv g'*r$),
$h = g*r$ und $h' := g'*r$ sind dann Einzel-Posids von ihren Positionen.
Fuer beide muss je ein Knoten nh und nh' [1] in die Sortierstruktur
aufgenommen werden,
ausserdem werden beide zu cthis dazugenommen.

Schliesslich muessen noch - was im eindiemsionalen Fall nicht notwendig
war - diejenigen Knoten verarbeitet werden, die Zeichenfeldern entsprechen,
die zwar schon nur einmal in f vorkommen, die aber trotzdem noch weiter
ausgeweitet werden muessen, um den MPMR zu erhalten (vgl. die Bemerkung dazu
in Fall 4).
Dazu existiert im Algorithmus noch ein

Fall 5,
der genau diesen Fall erledigt (d.h. einen Knoten fuer $h := g*r$ in die
Sortierstruktur und nach cthis dazunimmt).

Wird ein neuer Knoten nh in die Sortierstruktur eingetragen, so genuegt es
i.a. nicht, nur von ng aus eine Kante nach nh einzusetzen;
es muss von allen nk aus mit $h = k*r'$ eine entsprechende Kante
eingesetzt werden.
Dies geschieht im Unterprogramm "enter additional ptr links", das genau
nach dem Ergebnis von Satz 2.9.8 arbeitet (vgl. auch die dortige Bemerkung).
Fuer das Unterprogramm "enter additional ptr links" werden die Zeiger von
cthis vom vorigen Durchlauf aufbewahrt (im Array clast).

Die sfather-Zeiger fuer einen neuen Knoten nh werden von dem Unterprogramm
"enter all sfather nodes" aufgrund der Ergebnisse von Satz 2.10.16
konstruiert.

----

[1] im Algorithmus: nhs , ebenso hs fuer h' , usw.

Dabei ist zu beachten:
Ist q ∈ Nachf(p), so ist nicht immer der MPMR von q Suffix des MPMR von p.
Beispiel:
Das Zeichenfeld: hat als MPMRs:

```
                       1 1 1
  1 2 3 4 5 6 7 8 9 0 1 2          1 2 3 4 5 6        2 3 4 5 6
 +-----------------------         +-----------       +---------
 1|a a a a a c a a a a a d        1|a a a a a c      1|a a a a c
 2|a b c c c c a b d d d d        2|a b c c c c      2|b c c c c
 3|c c c c c c d d d d d d        3|c c c c c c      3|c c c c c
 4|d d d d d d d b c c c d
 5|d d d d d d d d d d d d         1 2 3              2 3 4 5 6
                                  +-----             +---------
                                  2|a b c            2|b c c c c
                                  3|c c c            3|c c c c c
```

Der MPMR von (2,1) ist kein Suffix des MPMR von (1,1)

Deshalb darf das Unterprogramm "enter all sfather nodes", das waehrend der Berabeitung von p schon den MPMR von q teilweise aufbaut, nicht einfach fuer jeden Knoten fuer p einen fuer q (als sfather) erzeugen, sondern muss damit aufhoeren, wenn die Grenzen des MPMR von q erreicht sind.
Deshalb wird vom Unterprogramm ebenfalls ueber das Array rk Buch gefuehrt.
Dazu muss festgestellt werden, wenn hs nur einmal vorkommt (also nhs erst waehrend der Bearbeitung der gerade aktuellen Position in die Sortierstruktur eingesetzt wurde). Dies geschieht durch die Zuweisung
  nhs.rk[r] := (nhs.startpos) = follow sdir(curpos.sr)) .

Im eindimensionalen Fall wurde unmittelbar nach der fertigen Bearbeitung einer Position zu deren (sfather-)Nachfolger uebergegangen.
Dies ist jetzt natuerlich nicht mehr immer moeglich, denn i.a. hat eine Position p mehrere (sfather-)Nachfolger q ∈ Nachf(p) .
Deshalb muss jetzt ein sfather-Zeiger auf eine Nachfolger-Positionen solange abgespeichert werden, bis diese Position mit dem Einsortieren "an der Reihe" ist. Dies geschieht mit Hilfe der Arrays cthisinit und clastinit .

Im folgenden eine Darstellung des Algorithmus in einer Quasi-Programmiersprache (in Anlehnung an Algol68 und Elan).

Legende:

ROW [1 : 0 FLEX] definiert in Algol68 ein Array mit flexibler oberer
  Grenze, die zu Beginn den Wert 0 hat und je nach Bedarf beliebig weit
  herauf - und auch wieder herunter - gesetzt werden kann.
[x1, x2, x3] steht in Algol68 fuer ein Array mit den Grenzen 1 und 3
  und den Elementen x1, x2 und x3;
[] steht also fuer ein leeres Array (mit den Grenzen 1 und 0);
  entsprechend steht
(x1, x2, x3) fuer eine Struktur mit den Elementen x1, x2 und x3 .
FOR r := ALL DIRs OF g soll eine Schleife darstellen, bei der r der
  Reihe nach alle Richtungen des Zeichenfelds g annimmt; entsprechend
FOR sr := ALL SDIRs (sr soll alle moeglichen s-Richtungen durchlaufen,
  von denen es nur endlich viele gibt).

```
/* ----- Datenstrukturen ------------------------------------------------- */
TYPE POS;    /* Positionen*/
TYPE DIR;    /* Richtungen (bzgl. xdulin) */
TYPE SDIR;   /* Richtungen (bzgl. vor/sfather) */
TYPE PZS;    /* Pseudo-ZS = Menge von Positionen,  zwei aequivalente
                PZS koennen trotzdem verschieden sein */
TYPE NODE = STRUCT
       (POS startpos,
        ROW [1:0 FLEX] REF NODE pred,     /* Rueckwaertszeiger */
        ROW [1:0 FLEX] INT sucdir,        /* zeigt auf das erste Feld
                                             zu einer Richtung in suc */
        ROW [1:0 FLEX] REF NODE suc,      /* Vorwaertszeiger */
        ROW [1:0 FLEX] PZS suclab,        /* Markierung (Abmessungen) */
        ROW [1:0 FLEX] REF NODE sfather,
        ROW [1:0 FLEX] BOOL rk,           /* Rand komplett */
        PZS label
       );

TYPE CURLIST = ROW[1:0 FLEX] CURELEM;

TYPE CURELEM = STRUCT
        (BOOL once,                       /* ZF kommt nur einmal vor */
         ROW [1:0 FLEX] BOOL rk,
         ROW [1:0 FLEX] REF NODE suc, pred,
         PZS label,
         REF NODE node
        );

REF NODE root;
POS startpos, curpos;
POS CONST nopos;
DIR CONST nodir;
SDIR CONST  nosdir;

ROW [1:2] CURLIST clast, cthis, cnext;
ROW [1:0 FLEX] CURLIST cthisinit, clastinit;


/* ----- Zugriffsroutinen ------------------------------------------------ */

REF NODE PROC new node (POS p, PZS lab);
              /* erzeugt einen neuen Knoten mit  startpos = p ,
                 label = lab,
                 pred = sucdir = suc = suclab = sfather = [],
                 rk bleibt noch undefiniert, wird spaeter eingetragen */

INT PROC compare label (PZS lab1, lab2);
              /* liefert  -1, 0, +1  zurueck fuer
                 Belegung(lab1) < , = , > Belegung(lab2),
                 implementiert daruch den Zugriff auf das Eingabe-ZF */

POS PROC next pos (POS p);
              /* implementiert die Sortierreihenfolge der Positionen */
```

```
POS PROC follow sdir (POS p, SDIR sr);
                /* liefert die Startposition des sfather-ZFs in s-Richtung
                   sdir , ausgehend von  p */

PZS PROC extend (PZS lab, DIR r);
                /* liefert die Abmessungen der Erweiterung
                   von  lab  in Richtung  r  */

PZS PROC growth (PZS lab, DIR r);
                /* liefert die Abmessungen des Zuwachsstuecks bei der
                   Erweiterung von  lab  in Richtung  r ,  es gilt
                   extend(l,r) = l u growth(l,r)    */

PZS PROC initial lab (POS p);
                /* liefert das AZS mit der Startposition  p  zurueck */

PROC put suc ptr (REF NODE source, dest, DIR r, PZS lab);
                /* sortiert in die  suc-Tabelle  von  source  einen Zeiger
                   auf den Knoten  dest  ein, traegt in die  pred-Tabelle
                   von  dest  einen Zeiger auf  source  ein */

REF NODE PROC get suc ptr from lab (REF NODE nn, DIR r, PZS lab);
                /* sucht, ob in der  suc-Tabelle von  nn  ein Eintrag mit
                   Richtung  r  und einem PZS, das (im Eingabe-ZF) genauso
                   belegt ist wie  lab ,  existiert (d.h. nicht die Ab-
                   messungen werden verglichen, sondern die Belegungen)
                   und liefert dann den zugehoerigen  ptr  zurueck,
                   sonst  NIL  */

REF NODE PROC get suc ptr from dir (REF NODE nn, DIR r);
                /* sucht, ob in der  suc-Tabelle  von nn  ueberhaupt ein
                   Eintrag mit Richtung  r  vorkommt, liefert den  ptr
                   des ersten vorkommenden zurueck, sonst  NIL  */

REF NODE PROC get pred ptr from dir (REF NODE nn, DIR r);
                /* liefert den  ptr-Eintrag  in der  pred-Tabelle von
                   nn , dessen zugehoerige Richtung  r  ist, sonst  NIL */

REF NODE PROC get sfather ptr from sdir (REF NODE nn, SDIR sr);
                /* liefert den  ptr-Eintrag in der  sfather-Tabelle von
                   nn , dessen zugehoerige s-Richtung  sr  ist */

PROC put sfather ptr (REF NODE nn, sfnn, SDIR sr);
                /* traegt in die  sfather-Tabelle von  nn  bei der
                   Richtung  sr  einen Zeiger auf  sfnn  ein */

PROC append (CURLIST list, CURELEM c, cpred, DIR r)
                /* fuegt  c  zu  list  hinzu, setzt   c.pred[r] := cpred
                   und   cpred.suc[r] := c    */

CURELEM PROC copy curelem (CURELEM c)
                /* liefert eine Kopie von  c  */

BOOL PROC is azs (PZS lab)
                /* gibt an, ob  lab  eine Anfangs-ZF-Struktur ist oder
                   nicht */
```

```
PZS PROC trunc (PZS lab, POS p)
                /* liefert den Endabschnitt   lab/p   */

BOOL PROC equal sfather (PZS f, h, DIR r, SDIR sr)
                /* fuehrt den Test nach Satz 2.10.16 (b) durch fuer
                   h = f*r,   p = follow sdir(f.startpos,sr),
                   und bestimmt so, ob  f und  h verschiedene oder gleiche
                   Zeichenfelder als sfather in Richtung  sr  haben */


/* ----- Initialisierung ------------------------------------------------- */

PZS g, h, hs, dh, dhs;
REF NODE ng, nh;
DIR r;
INT i, j;
CURELEM cthis1icopy;
ROW [1:0 FLEX] BOOL rkfalse, rktrue;
              /* es sei   rkfalse[i] = FALSE,   rktrue[i] = TRUE   fa. i */

curpos := startpos;
root := new node (nopos, initial lab(nopos));

/* lies das erste Zeichenfeld  g  und sortiere es ein */
g := initial lab (curpos);
ng := new node (curpos, g);
put suc ptr (root, ng, nodir, g);
enter all sfather nodes (ng);
clast[1] := cnext[1] := clast[2] := cthis[2] := cnext[2] := [];
cthis[1] := [ (TRUE, rktrue, g, ng) ];


/* ----- Gesamt-Schleife ------------------------------------------------- */

REPEAT /* Gesamt-Schleife */
  REPEAT /* Schleife fuer eine Position */
    BOOL cthis exhausted := TRUE;
    FOR j := 1 TO 2
    REPEAT
      FOR i := 1 TO UPB cthis[j]
      REPEAT /* Verarbeite das i.te Element von cthis[j] */
        g := cthis[j][i].label;
        ng := cthis[j][i].node;
        FOR r := ALL DIRs OF g
        REPEAT
          IF   NOT cthis[j][i].rk[r]
          THEN /* erweitere  g  in Richtung  r  */
               cthis exhausted := FALSE;
               h := extend (g,r);
               dh := growth (g,r);
               hs := extend (ng.label,r);
               dhs := growth (ng.label,r);
```

```
/* Fallunterscheidung */
IF   cthis[j][i].once
THEN /* Fall 5 */
     nh := new node (curpos, h);
     enter additional ptr links (nh, ng, r, dh);
     calc rk (nh, r);
     enter all sfather nodes (nh);
     cthis[j][i].suc := nh;
     append (cnext[j], (TRUE, nh.rk, h, nh), cthis[j][i],
                                                          r)
ELIF (nh := get suc ptr from lab (ng,r,dh))  ≠  NIL
THEN /* Fall 1 */
     append (cnext[j], (FALSE, rkfalse, h, nh),
                                          cthis[j][i], r);
     nh.rk := rkfalse;
     ng.rk := rkfalse
ELIF get suc ptr from dir (ng,r) ≠ NIL
THEN /* Fall 4 */
     nh := new node (curpos, h);
     enter additional ptr links (nh, ng, r, dh);
     calc rk (nh,r);
     enter all sfather nodes (nh);
     cthis[j][i].suc := nh;
     append (cnext[j], (TRUE, nh.rk, h, nh), cthis[j][i],
                                                          r)
ELIF compare label (dhs,dh) = 0
THEN /* Fall 3 */
     nh := new node (ng.startpos, hs);
     ng.rk := rkfalse;
     cthis[j][i].suc := nh;
     append (cnext[j], (FALSE, rkfalse, h, nh),
                                          cthis[j][i], r);

     enter additional ptr links (nh, ng, r, dh);
     enter all sfather nodes (nh)
ELSE /* compare label (dhs,dh) ≠ 0 */
     /* Fall 2 */
     nh := new node (curpos, h);
     nhs := new node (ng.startpos, hs);
     enter additional ptr links (nh, ng, r, dh);
     calc rk (nh,r);
     enter additional ptr links (nhs, ng, r, dhs);
     enter all sfather nodes (nh);
     enter all sfather nodes (nhs);
     cthis1icopy := copy curelem (cthis[1][i]);
     append (cthis[2], cthis1icopy, NIL, nodir);
     append (cnext[1], (TRUE,nh.rk, h, nh), cthis[1][i],
                                                          r);
                                                          
     append (cnext[2], (TRUE, nhs.rk, hs, nhs),
                                          cthis1icopy, r)

ENDIF /* Fallunterscheidung */
ENDIF /* NOT cthis[j][i].rk[r] */
ENDREPEAT /* FOR r := ALL DIRs OF g */
ENDREPEAT; /* FOR i := 1 TO UPB cthis[j] */
```

```
        clast[1] := cthis[1];
        cthis[1] := cnext[1];
        cnext[1] := [];
        clast[2] := cthis[2];
        cthis[2] := cnext[2];
        cnext[2] := []
     ENDREPEAT /* j := 1 TO 2 */
  UNTIL cthis exhausted
  ENDREPEAT; /* Schleife fuer eine Position */
  curpos := next pos (curpos);
  IF   curpos = nopos
  THEN EXIT /* letzte Position wurde bereits verarbeitet */
  ENDIF;
  cthis[1] := cthisinit [curpos];
  clast[1] := clastinit [curpos];
  IF   cthis[1] = []
  THEN /* keine sfather-Zeiger vorhanden */
       IF   ( nh := get suc ptr from lab (root,nodir,initial lab(curpos)) )
            =   NIL
       THEN nh := new node (curpos, initial lab(curpos));
            put suc ptr (root, nh, nodir, initial lab(curpos));
            enter all sfather nodes (nh)
            append (cthis[1], (TRUE, rktrue, initial lab(curpos), nh), NIL,
                                                                     nodir)
       ELSE append (cthis[1], (FALSE, rkfalse, initial lab(curpos), nh),
                                                                NIL, nodir)

       ENDIF
  ENDIF;
  lcast[2] := cthis[2] := cnext[2] := cnext[1] := []
ENDREPEAT; /* Gesamt-Schleife */




/* ----- Unterprogramme ------------------------------------------------------ */



PROC calc rk (REF NODE nh, DIR r);
                      /* berechnet den Wert von   nh.rk[r]    */
  DIR r2;
  nh.rk[r] := TRUE;
  FOR r2 := ALL DIRs OF nh.pred
  REPEAT
     IF   r2 ≠ r   AND   get pred ptr (nh,r2) ≠ NIL
     THEN nh.rk[r] := nh.rk[r] AND get pred ptr (nh,r2).rk[r]
     ENDIF
  ENDREPEAT
ENDPROC;
```

```
PROC enter additional ptr links (REF NODE nh, nf, DIR r, PZS lab);
                     /* berechnet fuer den neu gebildeten Knoten  nh  alle
                        zusaetzlichen Zeiger (die von anderen Knoten als
                        nf  auf ihn zeigen muessen) und traegt alle
                        Zeiger ein */
  DIR rr;
  REF NODE nfs, ng;
  put suc ptr (nf,nh,r,lab);                       /*      ng 0     (clast) */
  IF  is azs (nh.label)  OR  is azs (nf.label)     /*       / \             */
  THEN EXIT                                        /*      r/   \rr          */
  ENDIF;                                           /*      /     \          */
  FOR rr := ALL DIRs OF nf.pred                    /* nfs 0       0 nf      */
  REPEAT                                           /*     \\     / (cthis)  */
    IF   rr ≠ r                                    /*    rr\\   /r          */
    THEN ng := get pred ptr from dir (nf,rr);      /*       \\ /            */
         nfs := get suc ptr from dir (ng,r);       /*     nh  0    (cnext)  */
         put suc ptr (nfs,nh,rr,growth(nfs.label,rr))
    FI
  ENDREPEAT
ENDPROC;



PROC enter all sfather nodes (REF NODE nn):
                      /* berechnet fuer den neu gebildeten Knoten  nn
                         alle sfather-Zeiger, bildet evtl. neue sfather-
                         Knoten iterativ selbst: fuellt die  ROWs
                         cthisinit  und  clastinit  mit den berechneten
                         sfather-Knoten */
  CURLIST sfthis, sfnext;
  REF NODE nh, nf, nhs, nfs;
  DIR r;
  SDIR sr;
  PZS fs;
  INT i;
  BOOL nhs valid, nhs new;

  sfthis := [ (nn.startpos, nn.rk, nn.label, nn) ];
  sfnext := [];
  WHILE sfthis ≠ []
  REPEAT
    FOR i := 1 TO UPB sfthis
    REPEAT /* bearbeite das i.te Element von sfthis */
      nh := sfthis[i].node;
      FOR sr := ALL SDIRs
      REPEAT
        nhs valid := FALSE;
        FOR r := ALL DIRs OF nh.pred
        REPEAT
          nf := get pred ptr from dir (nh,r);
          nfs := get sfather ptr from sdir (nf, sr);
```

```
                IF   nf = root
                THEN put sfather ptr (nh, root, sr);
                     GOTO skip
                ELIF nfs = root AND NOT is asz (nf.label)
                THEN GOTO skip
                ELIF equal sfather (nf.label, nh.label, r, sr)
                THEN put sfather ptr (nh, nfs, sr);
                     GOTO skip
                ELIF NOT nfs.rk[r]
                THEN
                     fs := nfs.label;
                     dfs := trunc (growth(f,r,h), followsdir(nh.startpos,sr));
                     IF   NOT nhs valid
                     THEN
/* nfs    nf */               nhs := get suc ptr from lab (nfs,r,dfs);
/* O<.....O */                IF   nhs = NIL
/* !  sr  ! */                THEN nhs := new node (nfs.startpos, extend(fs,r));
/* !      ! */                     nhs new := TRUE
/* !r    r! */                ELSE nhs new := FALSE
/* v      v */                ENDIF; /* nhs = NIL */
/* O<.....O */                nhs.rk[r] := (nhs.startpos=follow sdir(curpos,sr));
/* nhs    nh */               nhs valid := TRUE
                     ENDIF; /* NOT nhs valid */
                     IF   nhs new
                     THEN put suc ptr (nfs, nhs, r, dfs)
                     ENDIF;
                     put sfather ptr (nh,nhs,sr);
                     cnfs := (FALSE, nfs.rk, nfs.label, nfs);
                     append (clastinit[nfs.startpos], cnfs, NIL, nodir)
                ENDIF /* NOT nfs.rk[r] */
            ENDREPEAT; /* r */
            IF   nhs new
            THEN FOR r2 := ALL DIRs OF nhs.pred
                 REPEAT
                    calc rk (nhs, r2)
                 ENDREPEAT;
                 APPEND (sfnext, (nhs new, nhs.rk, nhs.label, nhs), NIL,
                                                                    nodir)

            ENDIF;
skip:
        ENDREPEAT /* sr */
        APPEND (cthisinit[nhs.startpos], (nhs new, nhs.rk, nhs.label,nhs),
                                                                   cnfs, r)

    ENDREPEAT; /* i */
    sfthis := sfnext;
    sfnext := []
  ENDREPEAT /* while sfthis ≠ [] */
ENDPROC;
```

3.1.3 Ablaufbeispiel

Die Sortierreihenfolge sei
(1,1), (1,2), (2,1), (1,3), (2,2), (3,1), ...
... (1,n), (2,n-1), ... , (n,1), (1,n+1), ...
Die Zuwaechse (Zeilen oder Spalten) werden lexikografisch sortiert.

Im folgenden ein Protokoll des Ablaufs des Algorithmus bei der Bearbeitung des Zeichenfelds:

```
      1 2 3 4               1 2 3      2 3       3
f =  +--------   MPMRs:    +------    +---      +-
    1|a b c #             1|a b c    1|b c     1|c
    2|b a b #             2|b a b    2|a b
    3|d b a #             3|d b a    3|b a
    4|# # # #
                            1 2 3      2 3 4     3 4
                           +------    +------   +---
                          2|b a b    2|a b #   2|b #
                          3|d b a    3|b a #   3|a #
                                     4|# # #   4|# #

                            1          2 3 4     3 4
                           +-         +------   +---
                          3|d        3|b a #   3|a #
                                     4|# # #   4|# #
```

Die fertig konstruierte Sortierstruktur ist im Anschluss daran abgebildet.

Legende:
Die Nummern der Knoten beziehen sich auf die untenstehende Abbildung der fertigen Sortierstruktur.

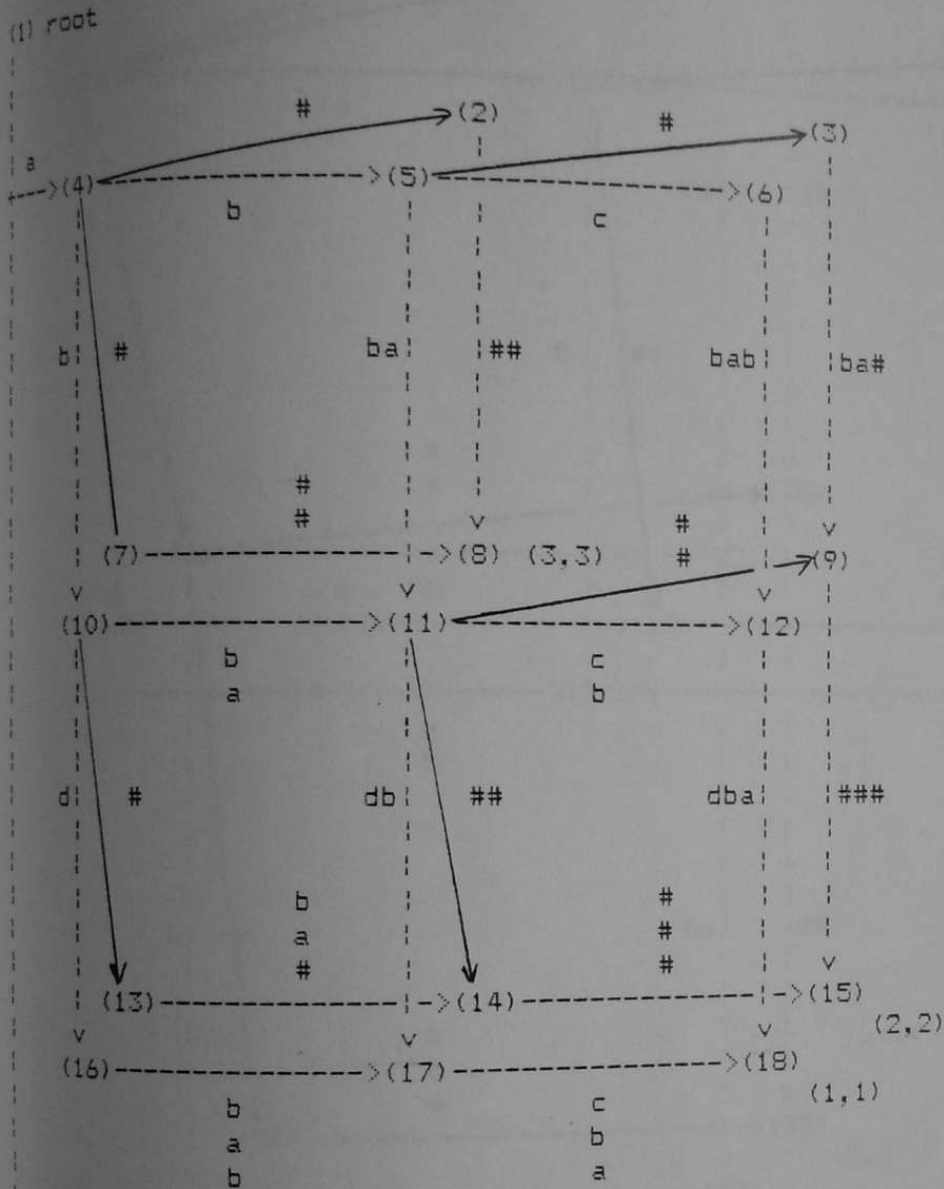
```
(11)+-  <--- abc ---  (22)    Vom Knoten (22) zum (evtl. neu erzeugten)
                                 Knoten (11) wird eine mit der Zeile  abc
                                 markierte Kante gezogen; (11) erhaelt die
                                 rk-Werte:   rk[-]=TRUE(+), rk[|]=FALSE(-)
(11)++  <-- /abc/ --  (22)    Von (22) wird eine mit der Spalte
                                    a
                                    b
                                    c
                                 markierte Kante nach (11) gezogen;
                                 (11) erhaelt die rk-Werte   rk[-]=rk[|]=TRUE
sf((11),|) = (22)             Der sfather-Zeiger von (11) in s-Richtung  |
                                 zeigt auf (22)
sf((11),|) =   /              Fuer (11) in s-Richtung  |  existiert kein
                                 sfather-Zeiger
sf((11),|) =   #              Der sfather-Zeiger von (11) in s-Richtung  |
                                 zeigt auf einen Knoten fuer ein Zeichenfeld,
                                 das mit  #  beginnt (diese Knoten sind hier
                                 nicht beruecksichtigt)
```

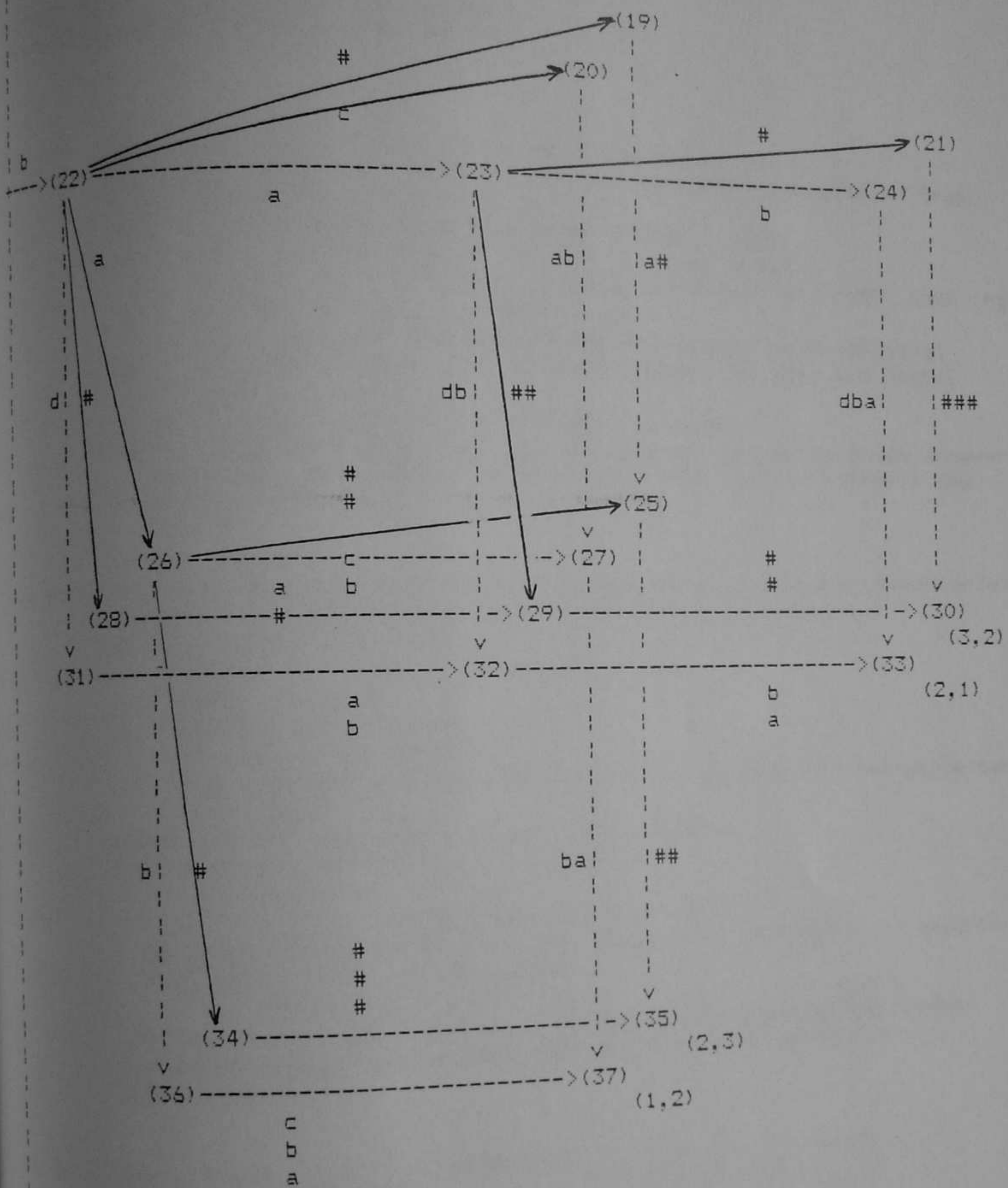
Doppelte Linien ===== kennzeichnen einen neuen Eintritt in die "Schleife fuer eine Position"; einfache Linien ----- kennzeichnen einen neuen Eintritt in die Schleife "FOR i := 1 TO UPB cthis[j]" .

```
(1)     root
=============================
curpos = (1,1)    "a"
(4)++   <--- a -----    (1)        sf( (4),-) =  (1),     sf( (4),!) =  (1)
=============================
curpos = (1,2)    "b"
(22)++  <--- b -----    (1)        sf((22),-) =  (1),     sf((22),!) =  (1)
=============================
curpos = (2,1)    "b"
ng = (1),     r = -,   Fall 1
-----------------------------
ng = (22),    r = -,   Fall 2
(20)+-  <--- c -----   (22)        sf((20),-) = (38),     sf((20),!) =  (1)
(23)+-  <--- a -----   (22)        sf((23),-) =  (4),     sf((23),!) =  (1)
                                   sf((38),-) =  (1),     sf((38),!) =  (1)

ng = (22),    r = !,   Fall 2
(26)-+  <--- a -----   (22)        sf((26),-) =  (1),     sf((26),!) =  (4)
(31)-+  <--- c -----   (22)        sf((31),-) =  (1),     sf((31),!) = (39)
                                   sf((39),-) =  (1),     sf((39),!) =  (1)

-----------------------------
ng = (20),    r = -,   komplett
ng = (20),    r = !,   Fall 5
(27)++  <--- ab ----   (20)        sf((27),-) =   / ,     sf((27),!) =  (5)
(27)     <-- /cb/ ---  (26)        sf( (5),-) = (22),     sf( (5),!) =  (1)
ng = (23),    r = -,   komplett
ng = (23),    r = !,   Fall 5
(32)++  <--- cb ----   (23)        sf((32),-) = (10),     sf((32),!) =   /
(32)     <-- /ab/ ---  (31)
ng = (26),    ex. schon
ng = (31),    ex. schon
=============================
curpos = (1,3)    "c"
cthisinit = (38)++
=============================
curpos = (2,2)    "a"
cthisinit = (5)--, (10)--
ng =  (5),    r = -,   Fall 2                              sf( (3),!) = (19)
 (3)+-  <--- # -----    (5)        sf( (3),-) =  (1),     sf( (6),!) = (20)
 (6)+-  <--- c -----    (5)        sf( (6),-) =  (1),
ng =  (5),    r = !,   Fall 3                              sf((11),!) = (26)
(11)--  <--- ba ----    (5)        sf((11),-) = (23),
(11)     <-- /ba/ ---  (10)
ng = (10),    r = -,   ex. schon
ng = (10),    r = !,   Fall 2                              sf((13),!) =  (1)
(13)-+  <--- # -----   (10)        sf((13),-) = (28),     sf((16),!) =  (1)
(16)-+  <--- d -----   (10)        sf((16),-) = (31),     sf((28),!) =   #
                                   sf((28),-) =  (1),
```

```
----------------------------
ng = (3),    r = -,   komplett
ng = (3),    r = !,   Fall 5
(9)+-  <--- ba# ---    (3)      sf( (9),-) = (25),    sf( (9),!) = (21)
(9)    <-- /##/ ---   (11)      sf((25),-) =   # ,    sf((25),!) =  (2)
                                sf((21),-) =  (2),    sf((21),!) =  (1)
                                sf( (2),-) =   # ,    sf( (2),!) =  (1)

ng = (6),    r = -,   komplett
ng = (6),    r = !,   Fall 5
(12)+- <--- bab ---    (6)      sf((12),-) = (27),    sf((12),!) = (24)
(12)   <-- /cb/ ---   (11)      sf((24),-) =  (5),    sf((24),!) =  (1)
ng = (11),   ex. schon
ng = (13),   r = -,   Fall 5
(14)-+ <-- /ba#/ --   (13)      sf((14),-) = (34),    sf((14),!) = (29)
(14)   <-- ## -----   (11)      sf((34),-) =  (1),    sf((34),!) =  (7)
                                sf((29),-) =   / ,    sf((29),!) =   #
                                sf( (7),-) =  (1),    sf( (7),!) =   #

ng = (13),   r = !,   komplett
ng = (16),   r = -,   Fall 5
(17)-+ <-- /bab/ --   (16)      sf((17),-) = (36),    sf((17),!) = (32)
(17)   <--- db ----   (11)      sf((36),-) =  (1),    sf((36),!) = (10)
ng = (16),   r = !,   komplett
----------------------------
ng = (9),    r = -,   komplett
ng = (9),    r = !,   Fall 5
(15)++ <--- ### ---    (9)      sf((15),-) = (35),    sf((15),!) = (30)
(15)   <-- /###/ --   (14)      sf((35),-) =   # ,    sf((35),!) =  (8)
                                sf((30),-) =  (8),    sf((30),!) =   #
                                sf( (8),-) =   # ,    sf( (8),!) =   #

ng = (12),   r = -,   komplett
ng = (12),   r = !,   Fall 5
(18)++ <--- dba ---   (12)      sf((18),-) = (37),    sf((18),!) = (33)
(18)   <-- /cba/ --   (17)      sf((37),-) =   / ,    sf((37),!) = (11)
                                sf((33),-) = (11),    sf((33),!) =   /

ng = (14),   ex. schon
ng = (17),   ex. schon
============================
curpos = (3,1)    "d"
cthisinit = (39)++
============================
curpos = (2,3)    "b"
cthisinit = (35)++
============================
curpos = (3,2)    "b"
cthisinit = (30)++
============================
curpos = (3,3)    "a"
cthisinit =  (8)++
============================
```

Insgesamt wurde also folgende Sortierstruktur erzeugt:





c
--->(38)
(1,3)

d
--->(39)
(3,1)

KAPITEL 3.2 Korrektheit und Aufwand

1. Def: Sei f ∈ ZF, jeder Suffix von f sei einmal in f,
eine Sortierstruktur fuer f ist ein endlicher gerichteter Graph
mit folgenden Eigenschaften:
(a) es gibt einen ausgezeichneten Knoten root,
(b) fuer jedes Anfangs-ZF g in f gibt es einen
Knoten ng = follow(g) und einen Pfeil von root nach ng
(c) g ∈ dPrf(h) fuer ein MPMR h von f
<==> ex. ein korrespondierender Knoten ng = follow(g)
(d) von einem Knoten ng zu einem Knoten nh ex. ein Pfeil,
wenn g xdulin h,
der Pfeil ist mit h!(Xh\Xg) markiert
(e) von einem Knoten ng ex. ein sfather-Zeiger zu einem anderen
Knoten nh , wenn st(g) xuvor st(h) und h Prefix des
MPMR von st(h)

2. Def: Als Voraussetzung fuer den Algorithmus sind alle bisher definierten
zusaetzlichen Bedingungen an ZS erforderlich, naemlich:
ZS mit Richtungen (2.8.2),
es gebe nur endlich viele s-Richtungen,
homogen (2.8.5),
beliebig verschiebbar (2.9.2),
AZS-eindeutig (2.9.3)
Endabschnitts-vollstaendig (2.10.3), also auch linkserweiterbar
und neben = Id .
Ausserdem sei, wie schon in Kap 3.1 erwaehnt,
- eine totale Ordnung <= definiert mit
p vor q ==> p <= q
(d.h. <= ist eine Verfeinerung von vor);
- sei eine totale Ordnung auf der Menge alle Zeichenfeld-Zuwaechse
definiert (d.h. auf der Menge); sowie
{h!(Xh\Xg) ! g,h ∈ ZF, h xdulin g}
- eine totale Ordnung auf der Menge aller Anfangs-Zeichenfelder.
Diese Voraussetzungen werden fuer den Rest des Kapitels als
gueltig angenommen.

3. Def: Sei f,g,h ∈ ZF, f' MPMR in f, h lin f', r ∈ Rh
h heisst r-komplett
:<==> f' = h * s1 * ... * sn mit r ∉ {s1,...,sn}

4. Lem: Sei f,g,h ∈ ZF, f' MPMR in f, r ∈ Rg, p = st(h),
h einmal in f, h = g * r, dann ist
(h r-komplett <==> h' r-komplett fa. h' xdulin h, h' ≠ g)

Bew: sei h = a * r' * t1 * ... * tn mit r ∉ {t1,...,tn},
sei f' = h * u1 * ... * um,
"<==": Fall 1: n > 0,
Ann: (oBdA.) u1 = r,
dann ist mit h' := a * r' * t2 * ... * tn
f' = h' * t1 * r * u2 * ... * um,
also h' nicht r-komplett. W.!

Fall 2: n=0, also h = a * r^i,
dann ist fa. k,k' mit k xdulin k', k min f,
k' ein f, st(k) = st(h),
(sei k = a * r^j * v1 * ... * vn'
mit r ∉ {v1,...,vn'})
j <= i-1, (denn sonst waere h lin k ==> k ein f)
nach Lemma 2.8.10 folgt h r-komplett

"==>": sei h' xdulin h, h' ≠ g, dann ist (oBdA.)
h' = a * r^i * t2 * ... * tn
(denn a * r^{i-1} * t1 * ... * tn = g)
also f' = h' * t1 * u1 * ... * um
mit r ∉ {t1,u1,...,um} da h r-komplett

5. Lem: h ist MPMR <==> h ist r-komplett fa. r ∈ Rh
Bew: folgt direkt aus Def 3

Bem: Die Aussagen ueber Komplettheit und MPMR in 3, 4 und 5
gelten beneso fuer vorlaeufige MPMRs und vorlauefige Komplettheit.
Lemma 2.9.10 gilt ebenso fuer "vorlaeufig".
Die Umformulierungen sind trivial, nur bei Lemma 4 muss man etwas
vorsichtig sein. Dessen "vorlaeufige" Entsprechung lautet:
Sei f,g,h ∈ ZF, f' vorlaeufiger MPMR in f (<= q)
r ∈ Rg, p = st(h),
h vorlaeufig einmal in f (<= q), h = g * r, dann ist
(h vorlaeufig r-komplett (<= q) <==>
h' vorlaeufig r-komplett (<= q) fa. h' xdulin h, h' ≠ g)

6. Lem: Fuer jeden Knoten nh in der Sortierstruktur und jede Richtung
r von h gilt:
nh.rk[r] = TRUE <==> h ist (vorlaeufig) r-komplett

Bew: Induktion ueber die Anzahl der Durchlaeufe der
"Schleife fuer eine Position":
Beim ersten Eintritt gilt die Beh., da nur root vorhanden ist.
Gilt die Beh. zu Beginn der Schleife, so auch am Ende:
in Fall 1 und 3 ist h min f, also nach Lemma 2.8
h nicht r-komplett, und entsprechend
nh.rk[r] = FALSE
in Fall 2, 4 und 5 ist h ein f, also wird nh.rk[r] von
"calc rk" korrekt berechnet nach Lemma 4
in der Routine "enter all sfather nodes" wird nur dann ein
neuer Knoten nhs erzeugt, wenn hs ein f,
also ist auch nhs.rk[r] korrket,
d.h. fuer alle Knoten, die waehrend eines Schleifendurchlaufs
dazukommen, gilt die Bedingung ebenfalls.

7. Lem: Zu Beginn jedes Durchlaufs der "Schleife fuer eine Position" gilt:
(a) Jeder Knoten in der Sortierstruktur entspricht einem Prefix eines MPMRs des Eingabe-ZFs, und
(b) zu jedem Prefix f" eines vorlaeufigen MPMRs (bzgl. den Positionen < curpos)
(b1) gibt es einen Knoten in der Sortierstruktur oder/und
(b2) es gibt einen Knoten unter denen in cthis , der einem Prefix von f" entspricht

Bew: Induktion ueber die Anzahl der Durchlaeufe der "Schleife fuer eine Position":
Beim ersten Eintritt gilt die Beh., da nur root vorhanden ist und es noch keine vorlaeufigen MPMRs gibt.
Fuer jeden Knoten ng aus cthis , der einem echten Prefix eines vorlaeufigen MPMRs f' entspricht, ex. nach dem Durchlauf der Schleife mindestens ein nh in cnext mit
g xdulin h lin f'
denn: da g nicht vorlaeufiger MPMR ist, ist nach Lemma 5
ng.rk[r] = FALSE fuer ein r
also wird nh mit h := g * r in cnext aufgenommen
(h lin f' nach 4, 3)

Damit gilt (b):
galt zu Beginn des vorigen Schleifendurchlaufs (b2) fuer f",
so gilt es jetzt immer noch,
galt (b1), so gilt es ebenfalls noch, da keine Knoten geloescht werden.

Ausserdem gilt (a), denn alle neu erzeugten Knoten nh entprechen einem Prefix eines vorlaeufigen MPMRs und damit auch eines (endgueltigen) MPMRs.

8. Lem: Fuer alle Knoten ng in cthis ist die Anzahl der Faktoren des entsprechenden ZFs die gleiche,
d.h. g = a * t1 * ... * tn mit a ∊ AZF,
n ist unabhaengig von g

Bew: Induktion ueber die Anzahl der Durchlaeufe der "Schleife fuer eine Position":
ist ng in cthis und hat n-1 Faktoren,
so werden nur Knoten der Form nh mit h = g * r (also mit n Faktoren) in cnext aufgenommen.

9. Korr: Alle Zeichenfelder g , die einem Knoten aus cthis entsprechen, sind unvergleichbar

Bew: sonst waere g1 xlin g2, ==> g2 hat mehr Faktoren als g1, W.!

10. Lem: Bei jedem Eintritt in die Gesamt-Schleife gilt:
(a), (d), (e) aus Def 1., sowie:

(f) jeder Knoten in der (vorlaeufigen) Sortierstruktur hat einen sfather-Eintrag fuer jede s-Richtung, in der ein sfather existiert,

(g) g ∈ dPrf(h) fuer einen vorlaeufigen MPMR h von f (bzgl. curpos) mit st(h) <= curpos
<==> ex. ein Knoten ng = follow(g)

(h) fuer jedes Anfangs-ZF g in f mit st(g) <= curpos
ex. ein Knoten ng = follow(g)

Bew: (a) trivial

Induktion ueber die Anzahl der "Gesamtschleifen"-Durchlaeufe:

(d) zu zeigen ist, dass "put suc ptr" nur aufgerufen wird mit ng, nh wenn g xdulin h

Initialisierung: klar,
Fallunterscheidung: klar, da h = extend (g,r)
enter all sfather nodes: klar, da fs xdulin hs
enter additional ptr links:
klar, sofern enter additional ptr links nur mit ng, nh aufgerufen wird mit g xdulin h,
denn nach Ind.vor. ist mit source xdulin dest auch nfs xdulin dest
enter additional ptr links wird nur in der Fallunterscheidung aufgerufen, mit h = extend (g.r)

(e),(f)
da zu jedem neuen Knoten ein sfather-Zeiger gesetzt wird (enter all sfather nodes wird jedesmal nach new node aufgerufen),
bleibt zu zeigen, dass enter all sfather nodes tatsaechlich alle sfather-Zeiger eintraegt:
nach Ind.vor. ex. zu nf und jeder s-Richtung sr schon ein sfather-Knoten nfs , und daraus wird nhs konstruiert;
fuer evtl. neu konstruierte Knoten werden iterativ ebenfalls die sfather-Knoten konstruiert;
dadurch, dass fuer jede s-Richtung nur ein Knoten nhs gebildet wird, werden fuer nhs auch alle pred-Zeiger eingetragen;
die Routine terminiert, da alle (auch indirekten) sfather-Zeichenfelder im urspruenglichen Zeichenfeld dest vorkommen und somit nur endlich viele sind;

(g) wird die "Schleife fuer eine Position" beendet, so ist nh.rk[r] = TRUE fa. r ∈ Rh,
d.h. h ist vorlaeufiger MPMR, (Lemma 5)
in cthis befindet sich dann nur noch nh (Lemma 8),
aus Lemma 7, (b) folgt dann:
zu jedem Prefix eines vorlaeufigen MPMRs (bzgl. den Positionen <= curpos) gibt es einen Knoten in der Sortierstruktur.
zusammen mit Lemma 7. (a) folgt die Beh.

(h) folgt sofort aus (g), denn jedes Anfangs-ZF ist Prefix des vorlaeufigen MPMRs seiner Position

11. Satz: Der Algorithmus aus Kapitel 3.1 baut aus jedem $f \in ZF$, dessen saemtliche Suffixe einmal in f sind, eine Sortierstruktur auf.

Bew: sind alle Positionen verarbeitet, so sind alle vorlaeufigen MPMRs auch (endgueltige) MPMRs, damit folgen die Bedingungen von Def 1 direkt aus Lemma 10: (b) folgt aus (h), (c) folgt aus (g)

12. Satz: Der Aufwand des Algorithmus ist proportional zur Anzahl der erzeugten Kanten.

Bew: Alle Teile des Algorithmus ausser Fall 1 haben einen konstanten Aufwand pro erzeugte Kante.
In Fall 1 ist durch die Verwendung der sfather-Zeiger sichergestellt, dass die gelesenen Zeichen nicht (fuer das Einsortieren einer anderen Position) nochmals eingelesen werden muessen (vgl. Satz 2.7.3). Dadurch ist der Aufwand in Fall 1 linear begrenzt.

KAPITEL 3.3 Literatur

In der Literatur gibt es verschiedene Verallgemeinerungen von eindimensionalen Algorithmen auf einzelne Modelle von Zeichenfeldern (Rechtecke, Trees, usw.).

In 1.3.5 wurde bereits eine Verallgemeinerung des Karp-Miller-Rosenberg-Algorithmus fuer Arrays vorgestellt.

Eine Verallgemeinerung des Knuth-Morris-Pratt-Algorithmus fuer Arrays sei hier kurz skizziert (BAK 78, BIR 77, INO 83):

```
Sind    v =  v(1,1)  ... v(1,m2)       und     w =  w(1,1)  ... w(1,n2)
               :           :                          :           :
             v(m1,1) ... v(m1,m2)                   w(n1,1) ... w(n1,n2)
```

gegeben (v als Pattern, w als Array, in dem gesucht werden soll), so wird zunaechst jede Zeile v(k,1) ... v(k,m2) einzeln in w gesucht (mit Hilfe des eindimensionalen Knuth-Morris-Pratt-Algorithmus, indem man die Zeilen von w nacheinander durchsucht).
Damit laesst sich aus w ein Array w' gewinnen, wobei
w'(i,j) = k bedeutet, dass die k.te Zeile von v an der Position (i,j) in w vorkommt.
Danach wird in w' (wieder mit Hilfe des eindimensionalen Knuth-Morris-Pratt-Algorithmus) nach einer Spalte

```
1
2
:
:
m2
```

gesucht (bzw. entsprechend, falls nicht alle Zeilen von v verschieden sind).
Man kann den Algorithmus auf verschiedene Arten verbessern, z.B. die explizite Konstruktion von w' vermeiden.

In der Literatur existieren noch verschiedene weitere Algorithmen, von denen hier nur auf den in KAR 72 angegebenen Repetition-Finding-Algorithmus fuer Trees verwiesen sei. Eine Entsprechung zum eindimensionalen Prefix-Tree-Algorithmus fuer irgendein (nicht-eindimensionales) Modell von Zeichenfeldern findet sich jedoch nicht.

KAPITEL 3.4 Ausblick

In dieser Arbeit wurde der Versuch unternommen, einen verallgemeinerten Prefix-Tree-Algorithmus fuer eine gewisse Klasse von "Zeichenfeldern" zu entwerfen und gleichzeitig diese Klasse durch die Anforderungen, die der Algorithmus an sie stellt, zu charakterisieren.

Es wurde ein solcher Algorithmus skizziert, der ueber die elementare Definition des Begriffs "Zeichenfeld" in 2.3.1 hinaus eine ganze Reihe zusaetzlicher Eigenschaften verlangt. Zu untersuchen waere, wieweit alle diese Eigenschaften wirklich notwendig sind und wieweit sie wichtige Modelle von Zeichenfeldern ausschliessen. Insbesondere das Richtungskonzept (bzw. beide Richtungskonzepte) sollten daraufhin geprueft werden.

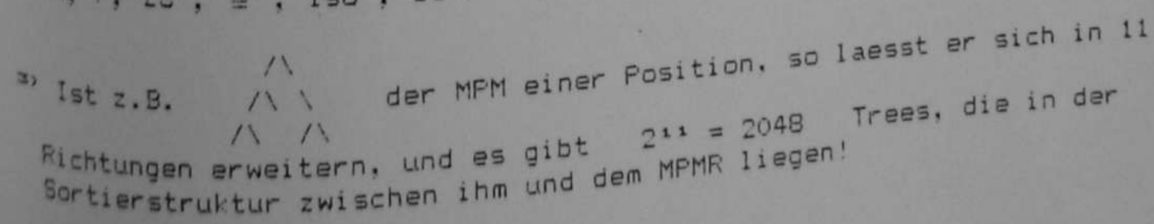
Auch sollten die Zusammenhaenge zwischen den zusaetzlichen Eigenschaften selbst weiter untersucht werden, um den Anteil der modell-abhaengigen Komponenten des Algorithmus [1] zu reduzieren. Dazu gehoert auch, zu untersuchen, wie sich ein Modell aus einem anderen gewinnen laesst (als triviales Beispiel: beliebige Vereinigungen von Rechtecken aus Rechtecken) und wie z.B. die MPMRs des zweiten Modells aussehen, wenn die des ersten bekannt sind. Mit anderen Worten, es waeren bestimmte Abgeschlossenheitseigenschaften zu untersuchen [2].

Andererseits gibt es auch am Algorithmus selbst noch viel zu verbessern. Zunaechst einmal muesste er wirklich implementiert werden und der skizzierte Korrektheitsbeweis ausgefuehrt werden. Damit der Algorithmus auch fuer solche Modelle (wie z.B. Trees), bei denen eine Zeichenfeld-Struktur i.a. sehr viele Richtungen hat, praktisch einsetzbar ist, ist eine kompaktifizierte Darstellung unerlaesslich [3]. Dabei waere zu untersuchen, ob man damit - wie im eindimensionalen Fall - einen linearen Zeitaufwand erreichen kann.

----

[1] in 3.1.2 in Form der Zugriffroutinen vorausgesetzt

[2] Z.B. stellt sich die Frage: wenn $(A, P, ZS, \equiv, Iso, st)$ die Definition 2.3.1 erfuellt und $ZS' := \{Xf \cup Xg \mid Xf, Xg \in ZS, st(Xf)=st(Xg)\}$, ob man $\equiv'$, Iso' und st' geeignet waehlen kann, sodass auch $(A, P, ZS', \equiv', Iso', st')$ die Definition 2.3.1 erfuellt.



[3] Ist z.B. [Baum] der MPM einer Position, so laesst er sich in 11 Richtungen erweitern, und es gibt $2^{11} = 2048$ Trees, die in der Sortierstruktur zwischen ihm und dem MPMR liegen!

ANHANG 0 Schreibweisen

ex.	Existenzquantor ("es existiert")
fa.	Allquantor ("fuer alle")
W.!	Widerspruch
t1,...,tn	Wenn moeglich, wird der Index auf dieselbe Zeile geschrieben, notfalls werden Klammern gesetzt, z.B. fuer t(n-1)
xlin	Ist R der Name einer reflexiven Ordnungsrelation, so wird die zugehoerige irreflexive Ordnungsrelation (die aus R durch Herausnehmen der Diagonale entsteht) mit xR bezeichnet. Also z.B. p xlin q <==> p lin q, p ≠ q

ANHANG 1 Aufwandsklassen fuer Algorithmen

Will man Aussagen ueber den Zeit- oder Speicherplatz-Aufwand eines Algorithmus machen, die unabhaengig von einer konkreten Implementierung auf einem bestimmten Computer sein sollen, so bedient man sich i.a. des Kalkuels der Aufwandsklassen.

Ist f: N --> N eine (berechenbare) Funktion, so ist
 O(f) := {f' | f':N-->N, ex. c,n0 ∈ N mit f'(n) <= c*f(n) fa. n>=n0}
die Aufwandsklasse von f. Statt O(f) wird meist O(f(n)) geschrieben.

Die Definition beruecksichtigt, dass man durch unterschiedlich schnelle Computer und unterschiedlich gute Compilierungen durchaus verschiedene Implementierungen desselben Algorithmus erhalten kann, die sich um einen konstanten Faktor c im Aufwand unterscheiden. Das n0 in der Definition beruecksichtigt, dass man den Algorithmus fuer endlich viele Eingaben immer noch effizienter machen kann, indem man z.B. das Resultat vorausberechnet und in einer Tabelle bereithaelt.

Eine Aufwandsklasse gibt eine obere Schranke fuer den Aufwand des Algorithmus in Abhaengigkeit von der Groesse der Eingabe (z.B. Anzahl der Zeichen des Eingabestrings) an. Aufwand kann sich auf Zeit- oder Speicherplatz-Aufwand beziehen. Ist der Zeitaufwand eines Algorithmus O(t(n)) und sein Speicherplatzaufwand O(s(n)), so gilt O(s(n)) ⊂ O(t(n)) , denn um s(n) Speicherzellen vollzuschreiben, braucht man mindestens s(n) Zeiteinheiten.

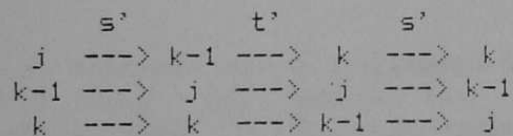
Die wichtigsten Eigenschaften der Aufwandsklassen:
 O(f(n)+g(n)) = O(f(n)) wenn c,n0 ex. mit c*f(n) >= g(n) fa. n>=n0
 g(n) ∈ O(f(n)) , c ∈ N ==> c*g(n) ∈ O(f(n))
 g1(n),g2(n) ∈ O(f(n)) ==> (g1+g2)(n) ∈ O(f(n))
 $O(\log^a(n)) = O(\log^b(n))$ fa. a,b>0
(Wegen der letzten Eigenschaft schreibt man i.a. nur O(log(n)))

ANHANG 2 Ein Satz aus der Kombinatorik

Satz: Jede Permutation s von {1,...,n} laesst sich als Produkt tk * ... * t1 von Transpositionen von benachbarten Elementen darstellen.

Bew: Bekannt ist, dass sich jede Permutation als Produkt von Transpositionen (von moeglicherweise nicht benachbarten Elementen) darstellen laesst,
oBdA. sei daher s schon Transposition,
s vertausche j und k (sei k >= j),

Induktion ueber k-j :
k-j = 1: dann sind j und k schon benachbart
k-j-1 ⟶ k-j: s' vertausche j und k-1,
nach Ind.vor. ist s' schon Produkt von Transpositionen von benachbarten Elementen
t' vertausche k-1 und k,
dann ist s = s' * t' * s',
denn:



(alle anderen Elemente bleiben unveraendert)

```
/*
ANHANG 3    Ein Algorithmus fuer den eindimensionalen Fall
=====================================================================
*/


/* implementiert unter OS9 auf EKF/6809 */
/* Programmiersprache:  C               */


#include <stdio.h>
#include <std.h>

#define maxnode       256        /* max. Anzahl der Knoten */
#define alfasize      8          /* Groesse des verwendeten Alfabets */
#define alfastart     ('a'-1)    /* erster Char des Alfabets */
#define maxstrlgth    100        /* max. String-Laenge */
#define maxdepth      20         /* max. Depth in Tree */
#define nopos         (-1)       /* illegal Position in String */
#define nochar        (-1)       /* illegal Char Rank */
#define NIL           0          /* illegal Pointer */
#define eoschar       0          /* End-of-String Indicator Char Rank */


int str [maxstrlgth];      /* enthaelt den eingelesenen String */
int curch;                 /* curch = str[readpos] */
int existch;               /* existch = str[curnode->pos+curdepth] */
int readpos,               /* String str eingelesen bis readpos */
    sortpos;               /* fertig einsortiert bis sortpos */
int curdepth;              /* (p-)Tiefe von curnode, (p-)Root=0 */
struct bitree *curnode;    /* zeigt auf die Stelle im Tree, bis zu der */
                           /* str[sortpos+1] schon einsortiert ist     */
struct bitree *a,          /* (p-)Root */
        *asfather;         /* sfather von a; aus technischen Gruenden noetig */
int i;



/* ******************************************************************* */
/*                                                                     */
/* Tree-Zugriffs-Routinen                                              */
/* ******************************************************************* */


struct bitree {struct bitree *son [alfasize];   /* p-Tree */
               struct bitree *father;           /* p-Tree */
               struct bitree *sfather;          /* s-Tree */
               int pos;
               int nodeid;
              };


static struct bitree nodestore [maxnode];
static int lastnode = -1;
```

```
struct bitree *newnode (father, fatherchar, sfather, pos)
struct bitree *father, *sfather;
             /* liefert einen neuen Knoten als Sohn von  father  mit einer
                mit  fatherchar  markierten Kante,  sfather und Position
                werden ebenfalls gesetzt */
int fatherchar, pos;
   {
   struct bitree *res;
   int i;
   if (++lastnode >= maxnode)
      error ("Memory overflow");
   res = &nodestore[lastnode];
   res->father  = father;
   res->sfather = sfather;
   res->pos     = pos;
   res->nodeid  = lastnode;
   for (i=0; i<alfasize; ++i)
      res->son[i] = NIL;
   if (father != NIL)
      father->son[fatherchar] = res;
   return (res);
   }


BOOL leaf (node)
struct bitree *node;
             /* gibt an, ob  node  ein Blatt ist oder nicht */
   {
   int i;
   BOOL res = TRUE;
   for (i=0; i<alfasize; ++i)
      if (node->son[i] != NIL)
         {
         res = FALSE;
         break;
         }
   return (res);
   }


int rank (ch)
char ch;
             /* rechnet Zeichen von der externen in die interne Darstellung
                um */
   {
   int res;
   if ((res=ch-alfastart) < 0   ||   res > alfasize)
      error ("Illegal Char");
   return (res);
   }
```

```
char unrank (i)
int i;
            /* rechnet Zeichen von der internen in die externe Darstellung
               um */
  {
  return ((char)(i+alfastart));
  }


error (msg)
char *msg;
            /* bricht das Programm mit einer Fehlermeldung ab */
  {
  int i;
  printf ("\n\nAlg.B Error: %s\n\n", msg);
  printf ("\n");
  for (i=0; i<=readpos; ++i)
     printf ("%c", unrank (str[i]));
  printf ("\n");
  for (i=0; i<=readpos; ++i)
     printf ("%d", i % 10);
  printf ("\n");
  for (i=0; i<=readpos; ++i)
     printf ("%d", i / 10);
  printf ("\n");
  dumptree (a);
  exit (errno);
  }


int prdepth = 0;
char cont [maxdepth];


dumptree (t)
struct bitree *t;
            /* druckt den Baum  t  aus */
  {
  printf ("\n");
  for (i=0; i<maxdepth; ++i)
     cont [i] = '|';
  if (t != NIL)
     dt (t, TRUE);
  else printf ("NIL");
  printf ("\n");
  }


dt (t, last)
struct bitree *t;
BOOL last;
            /* (Hilfs-Unterprogramm fuer  dumptree ) */
  {
  int i, j, lastbranch;
```

```
  printf ("(%d) sf=", t->nodeid);
  if (t->sfather == NIL)
    printf ("NIL");
  else
    printf ("%d", t->sfather->nodeid);
  if (t->pos != nopos)
    printf (" pos=%d", t->pos);
  cont [prdepth]   = (last    ? ' ' : '|');
  cont [prdepth+1] = (leaf(t) ? ' ' : '|');
  printf ("\n");
  for (j=0; j<=prdepth+1; ++j)
    printf ("%c   ", cont[j]);
  for (i=alfasize-1; i>=0; --i)
     if (t->son[i] != NIL)
        {
        lastbranch = i;
        break;
        }
  ++prdepth;
  for (i=0; i<=lastbranch; ++i)
     if (t->son[i] != NIL)
        {
        printf ("\n");
        for (j=0; j<prdepth; ++j)
           printf ("%c   ", cont[j]);
        printf ("+-%c-+ ", unrank (i));
        dt (t->son[i], i==lastbranch);
        }
  cont [prdepth] = '|';
  --prdepth;
  }



/* ******************************************************************** */
/*                                                                      */
/* Algorithmus                                                          */
/* ******************************************************************** */


struct bitree *insertnode (fa, fach, po)
struct bitree *fa;          /* father */
int fach, po;               /* father char, pos */
          /* erzeugt einen neuen Knoten als Sohn von  fa  mit einer mit
             fach  markierten Kante,
             erzeugt automatisch den sfather-Zeiger (ggf. werden rekursiv
             mehrere sfather-Knoten erzeugt) */
  {
  struct bitree *sfa;    /* sfather */
  if ((sfa = fa->sfather->son[fach])   ==   NIL)
     { /* build sfa recursively */
     sfa = insertnode (fa->sfather, fach, po+1);
     fa->sfather->pos = nopos;
     }
  return (newnode (fa, fach, sfa, po));
  }
```

```
followsfatherchain ()
          /* verfolgt nach dem fertigen Einsortieren einer Position die
             sfather-Kette und ueberspringt alle nachfolgenden Positionen,
             die ebenfalls schon vollstaendig einsortiert sind */
   {
   while (leaf (curnode)   &&   curnode->pos+curdepth-1==readpos)
      {
      curnode = curnode->sfather;
      --curdepth;
      ++sortpos;
      }
   }


main ()
          /* Hauptprogramm */
{
asfather = newnode (NIL, nochar, NIL, nopos);
a        = newnode (NIL, nochar, NIL, nopos);
a->sfather = asfather;
a->father  = asfather;
for (i=0; i<alfasize; ++i)
   asfather->son[i] = a;
curnode  = a;
curdepth = 0;

str [0] = curch = rank (getchar());  /* sortiere den ersten Char ein */
a->pos = 0;
if (curch == eoschar)
   exit (0);
sortpos  = 0;        /* der erste Char des Strings hat die Position 0 */


str [1] = curch = rank (getchar());
for (readpos = 1;   ;   str[++readpos] = curch = rank(getchar()))
   {
   if (curnode->son[curch] != NIL)

      { /* === Fall 1 === */
      curnode = curnode->son[curch];
      ++curdepth;
      }

   else /* son[curch] == NIL */
      {
      if (leaf (curnode))
         {
         if ((existch=str[curnode->pos+curdepth]) != curch)

            { /* === Fall 2 === */
            insertnode (curnode, existch, curnode->pos);
            curnode->pos = nopos;
            insertnode (curnode, curch, sortpos+1);
```

```
            ++sortpos;
            curnode = curnode->son[curch]->sfather;
            curdepth = curdepth;
            followsfatherchain();
            }

        else /* existch == curch */

            { /* === Fall 3 === */
            i = curnode->pos;
            curnode->pos = nopos;
            curnode = insertnode (curnode, curch, i);
            ++curdepth;
            }

        } /* leaf (curnode) */
      else /* no leaf, but son[curch]==NIL */

        { /* === Fall 4 === */
        insertnode (curnode, curch, sortpos+1);
        ++sortpos;
        curnode = curnode->son[curch]->sfather;
        followsfatherchain();
        }

      } /* son[curch]==NIL */
    if (curch == eoschar)
      break;
    } /* for */


 /* hier gilt:   curnode==a, curdepth==0, sortpos==readpos */


 printf ("\n");
 for (i=0; i<=readpos; ++i)
   printf ("%c", unrank (str[i]));
 printf ("\n");
 for (i=0; i<=readpos; ++i)
   printf ("%d", i % 10);
 printf ("\n");
 for (i=0; i<=readpos; ++i)
   printf ("%d", i / 10);
 printf ("\n");
 dumptree (a);
 } /* main */
```

```
/*   Ein Algorithmus fuer den eindimensionalen Fall (kompaktifiziert)
                                                          Anh.4-1 */


/*
ANHANG 4    Ein Algorithmus fuer den eindimensionalen Fall (kompaktifiziert)
================================================================================
*/


/* implementiert unter OS9 auf EKF/6809 */
/* Programmiersprache: C                */


#include <stdio.h>
#include <std.h>


#define maxnode        256          /* max. Anzahl der Knoten */
#define alfasize       8            /* Groesse des verwendeten Alfabets */
#define alfastart      ('a'-1)      /* erster Char des Alfabets */
#define maxstrlgth     100          /* max. String-Laenge */
#define maxdepth       20           /* max. Depth in Tree */
#define nopos          (-1)         /* illegal Position in String */
#define nochar         (-1)         /* illegal Char Rank */
#define NIL            0            /* illegal Pointer */
#define eoschar        0            /* End-of-String Indicator Char Rank */


int str [maxstrlgth];      /* enthaelt den eingelesenen String */
int curch;                 /* curch = str[readpos] */
int existch;               /* existch = str[curnode->pos+curdepth] */
int readpos,               /* String str eingelesen bis readpos */
    sortpos;               /* fertig einsortiert bis sortpos */
int curdepth;              /* (p-)Tiefe von curnode, (p-)Root=0 */
int curbrpos;              /* Position innerhalb des Branch-Labels */
int curbrch;               /* erster Char des aktuellen Branch-Labels */
struct bitree *curnode;    /* zeigt auf die Stelle im Tree, bis zu der */
                           /* str[sortpos+1] schon einsortiert ist     */
                           /* wird bei der sfather-Konstruktion verwendet */
struct bitree *sf;         /* ------------------ " --------------------- */
int sfdepth, epos;         /* ------------------------------------------ */
struct bitree *a,          /* (p-)Root */
          *asfather;       /* sfather von a; aus technischen Gruenden noetig */
int i;



/* *************************************************************** */
/*                                                                  */
/* Tree-Zugriffs-Routinen                                           */
/* *************************************************************** */


struct bitree {struct bitree *son [alfasize];     /* p-Tree */
               int sonlgth [alfasize];            /* Lgth des Branch-Labels */
               struct bitree *father;             /* p-Tree */
               struct bitree *sfather;            /* s-Tree */
               int pos;
               int nodeid;
              };
```

```
/*    Ein Algorithmus fuer den eindimensionalen Fall (kompaktifiziert)
                                                               Anh.4-2 */


static struct bitree nodestore [maxnode];
static int lastnode = -1;



struct bitree *newnode (father, fatherchar, pos)
struct bitree *father;
int fatherchar, pos;
          /* liefert einen neuen Knoten als Sohn von  father  mit einer
             mit  fatherchar  markierten Kante,  Pos wird ebenfalls
             gesetzt,
             sfather wird auf NIL gesetzt */
  {
  struct bitree *res;
  int i;
  if (++lastnode >= maxnode)
     error ("Memory overflow");
  res = &nodestore[lastnode];
  res->father  = father;
  res->sfather = NIL;
  res->pos     = pos;
  res->nodeid  = lastnode;
  for (i=0; i<alfasize; ++i)
     {
     res->son[i] = NIL;
     res->sonlgth[i] = 0;
     }
  if (father != NIL)
     {
     father->son[fatherchar] = res;
     father->sonlgth[fatherchar] = 1;
     }
  return (res);
  }



BOOL leaf (node)
struct bitree *node;
          /* gibt an, ob  node  ein Blatt ist oder nicht */
  {
  int i;
  BOOL res = TRUE;
  for (i=0; i<alfasize; ++i)
     if (node->son[i] != NIL)
        {
        res = FALSE;
        break;
        }
  return (res);
  }
```

```
/*   Ein Algorithmus fuer den eindimensionalen Fall (kompaktifiziert)
                                                              Anh.4-3 */


int branchchar (n)
struct bitree *n;
             /* liefert das (erste) Zeichen, mit dem die Kante vom Vater
                von  n  nach  n  markiert ist */
  {
  int i;
  struct bitree *nf;
  if (n == NIL)
    error ("branchchar (NIL)");
  nf = n->father;
  for (i=0; i<alfasize; ++i)
    if (nf->son[i] == n)
      return (i);
  error ("branchchar not found");
  }



int rank (ch)
char ch;
             /* rechnet Zeichen von der externen in die interne
                Darstellung um */
  {
  int res;
  if ((res=ch-alfastart) < 0   ||   res >= alfasize)
    error ("Illegal Char");
  return (res);
  }



char unrank (i)
int i;
             /* rechnet Zeichen von der internen in die externe
                Darstellung um */
  {
  return ((char)(i+alfastart));
  }



error (msg)
char *msg;
             /* bricht das Programm mit einer Fehlermeldung ab */
  {
  int i;
  printf ("\n\nAlg.C Error: %s\n\n", msg);
  printf ("\n");
  for (i=0; i<=readpos; ++i)
    printf ("%c", unrank (str[i]));
  printf ("\n");
  for (i=0; i<=readpos; ++i)
    printf ("%d", i % 10);
  printf ("\n");
  for (i=0; i<=readpos; ++i)
    printf ("%d", i / 10);
  printf ("\n");
```

```
   dumptree (a);
   exit (errno);
   }


int prdepth = 0;
char cont [maxdepth];



dumptree (t)
struct bitree *t;
           /* druckt den Baum  t  aus */
   {
   printf ("\n");
   for (i=0; i<maxdepth; ++i)
      cont [i] = '|';
   if (t != NIL)
      dt (t, TRUE);
   else printf ("NIL");
   printf ("\n");
   }



dt (t, last)
struct bitree *t;
BOOL last;
           /* (Hilfs-Unterprogramm fuer  dumptree ) */
   {
   int i, j, lastbranch;
   printf ("(%d) sf=", t->nodeid);
   if (t->sfather == NIL)
      printf ("NIL");
   else
      printf ("%d", t->sfather->nodeid);
   if (t->pos != nopos)
      printf (" pos=%d", t->pos);
   cont [prdepth]   = (last    ? ' ' : '|');
   cont [prdepth+1] = (leaf(t) ? ' ' : '|');
   printf ("\n");
   for (j=0; j<=prdepth+1; ++j)
      printf ("%c   ", cont[j]);
   for (i=alfasize-1; i>=0; --i)
      if (t->son[i] != NIL)
         {
         lastbranch = i;
         break;
         }
   ++prdepth;
   for (i=0; i<=lastbranch; ++i)
      if (t->son[i] != NIL)
         {
         printf ("\n");
         for (j=0; j<prdepth; ++j)
            printf ("%c   ", cont[j]);
```

```
        printf ("+%c,%1d+ ", unrank (i), t->sonlgth[i]);
        dt (t->son[i], i==lastbranch);
        }
   cont [prdepth] = '|';
   --prdepth;
   }



/* ************************************************************************ */
/* Algorithmus                                                              */
/* ************************************************************************ */


struct bitree *splitbranch (fa, brch, brpos)
struct bitree *fa;
int brch, brpos;
            /* spaltet die vom Knoten  fa  ausgehende mit  brch  markierte
               (Super-)Kante nach  brpos  Mikro-Knoten auf und setzt dort
               einen neuen (Super-)Knoten ein */
   {
   struct bitree *new, *save;
   int existch, totlgth;
   save = fa->son[brch];
   totlgth = fa->sonlgth[brch];
   new = newnode (fa, brch, save->pos);
   existch = str[save->pos+brpos];
   new->son[existch] = save;
   new->sonlgth[existch] = totlgth-brpos;
   fa->sonlgth[brch] = brpos;
   return (new);
   }


struct bitree *findsfather (n, ndepth)
struct bitree *n;
int ndepth;
            /* sucht den sfather-Knoten von  n , ggf. wird ein neuer
               Knoten erzeugt
               (es wird aber NICHT ggf. auch dessen sfather-Knoten rekursiv
               erzeugt !)
               ndepth  gibt die Tiefe von  n  im Baum an */
   {
   struct bitree *res;
   int pathlgth, brchpos, brch;
   if (n->sfather != NIL)
      return (n->sfather);
   brch = branchchar (n);
   pathlgth = n->father->sonlgth[brch];
   res = n->father->sfather;
   brchpos = n->pos + ndepth - pathlgth;
```

```
  while (pathlgth > 0)
    {
    if (res->son[brch] == NIL)
      {
      if (leaf (res))
        res->father->sonlgth[branchchar(res)] += pathlgth;
      else
        {
        newnode (res, brch, n->pos+1);
        res->sonlgth[brch] = pathlgth;
        res = res->son[brch];
        }
      pathlgth = 0;
      }
    else
      {
      if (res->sonlgth[brch] > pathlgth)
        {
        res = splitbranch (res, brch, pathlgth);
        pathlgth = 0;
        }
      else
        {
        brchpos  += res->sonlgth[brch];
        pathlgth -= res->sonlgth[brch];
        res = res->son[brch];
        brch = str[brchpos];
        }
      }
    } /* while */
  n->sfather = res;
  return (res);
  }




main ()
           /* Hauptprogramm */
{
asfather = newnode (NIL, nochar, nopos);
a        = newnode (NIL, nochar, nopos);
asfather->sfather = asfather;
a->sfather = asfather;
a->father  = asfather;
for (i=0; i<alfasize; ++i)
  {
  asfather->son[i]     = a;
  asfather->sonlgth[i] = 1;
  }
curnode  = a;
curbrpos = 0;
curdepth = 0;
```

```
/*    Ein Algorithmus fuer den eindimensionalen Fall (kompaktifiziert)
                                                          Anh.4-7 */


str [0] = curch = rank (getchar());   /* sortiere den ersten Char ein */
a->pos = 0;
if (curch == eoschar)
   exit (0);
sortpos  = 0;        /* der erste Char des Strings hat die Position 0 */


str [1] = curch = rank (getchar());
for (readpos = 1;   ;   str[++readpos] = curch = rank(getchar()))
   {
   if (curbrpos == 0)
      {
      if (curnode->son[curch] != NIL)

         { /* === Fall 1a === */
         curbrch = curch;
         ++curdepth;
         if (curnode->sonlgth[curch] == 1)
            curnode = curnode->son[curch];
         else
            curbrpos = 1;
         } /* 1a */

      else /* son[curch] == NIL */
         {
         if (leaf (curnode))
            {
            if ((existch=str[curnode->pos+curdepth]) != curch)

               { /* === Fall 2 === */
               newnode (curnode, existch, curnode->pos);
               newnode (curnode, curch, sortpos+1);

               /* konstruiere die sfather-Kette von curnode: */
               sf = findsfather (curnode, curdepth);
               sfdepth = curdepth-1;
               while (sf->sfather == NIL)
                  {
                  sf = findsfather (sf, sfdepth);
                  --sfdepth;
                  }

               /* konstruiere alle sfather-Knoten fuer
                  curnode->son[existch] ,  ohne sie zu verketten: */
               sf = curnode->sfather;
               epos = curnode->pos;
               while (sf->son[existch] == NIL)
                  {
                  newnode (sf, existch, ++epos);
                  sf = sf->sfather;
                  }

               /* konstruiere alle sfather-Knoten fuer
                  curnode->son[curch] ,  ohne sie zu verketten: */
               sf = curnode->sfather;
               ++sortpos;
```

```
        while (sf->son[curch] == NIL)    /* incl. followsfatherchain */
           {
           newnode (sf, curch, ++sortpos);
           sf = sf->sfather;
           --curdepth;
           }
        curbrch = curch;
        if (sf->sonlgth[curch] == 1)
           curnode = sf->son[curch];
        else
           {
           curnode = sf;
           curbrpos = 1;
           }
        } /* 2 */

     else /* existch == curch */

        { /* === Fall 3 === */
        if (curnode == a)
           {
           curnode = newnode (curnode, curch, curnode->pos);
           curbrch = curch;
           }
        else
           ++(curnode->father->sonlgth[curbrch]);
        ++curdepth;
        } /* 3 */

     } /* leaf (curnode) */
   else /* no leaf, but curch branch doesnt exist */

     { /* === Fall 4a === */
     newnode (curnode, curch, sortpos+1);
     sf = findsfather (curnode, curdepth);
     /* sf->sfather != NIL, da curnode mindestens 2 sons hatte und */
     /* daher sfather(curnode) ebenfalls                            */
     ++sortpos;

     /* konstruiere alle sfather-Knoten fuer
        curnode->son[curch] ,  ohne sie zu verketten: */
     while (sf->son[curch] == NIL)
        {
        newnode (sf, curch, ++sortpos);
        sf = sf->sfather;
        --curdepth;
        }
     curnode = sf->son[curch];
     curbrch = curch;
     curbrpos = curbrpos;
     } /* 4a */

    } /* son[curch]==NIL */
  } /* curbrpos == 0 */
```

```
  else
    { /* curbrpos != 0 */
    existch   =   str  [curnode->son[curbrch]->pos + curdepth];
    if (existch == curch)
      {
      ++curdepth;
      if (curbrpos < curnode->sonlgth[curbrch]-1)

        /* === Fall 1b === */
        ++curbrpos;

      else

        { /* === Fall 1c === */
        curnode = curnode->son[curbrch];
        curbrpos = 0;
        } /* 1c */

      } /* existch == curch */
    else

      { /* === Fall 4b === */
      curnode = splitbranch (curnode, curbrch, curbrpos);
      newnode (curnode, curch, sortpos+1);

      /* konstruiere die sfather-Kette fuer den gerade neu eingesetzten
        Knoten */
      sf = findsfather (curnode, curdepth);
      sfdepth = curdepth-1;
      while (sf->sfather == NIL)
        {
        sf = findsfather (sf, sfdepth);
        --sfdepth;
        }

      /* konstruiere alle sfather-Knoten fuer
        curnode->son[curch] ,  ohne sie zu verketten */
      ++sortpos;
      sf = curnode ->sfather;
      while (sf->son[curch] == NIL)   /* incl. follow sfather chain */
        {
        newnode (sf, curch, ++sortpos);
        sf = sf->sfather;
        --curdepth;
        }
      curnode = sf->son[curch];
      curbrch = curch;
      curbrpos = 0;
      } /* 4b */

    } /* curbrpos != 0 */
  if (curch == eoschar)
    break;
  } /* for */


/* hier gilt:   curnode==a, curdepth==0, sortpos==readpos */
```

```
printf ("\n");
for (i=0; i<=readpos; ++i)
   printf ("%c", unrank (str[i]));
printf ("\n");
for (i=0; i<=readpos; ++i)
   printf ("%d", i % 10);
printf ("\n");
for (i=0; i<=readpos; ++i)
   printf ("%d", i / 10);
printf ("\n");
dumptree (a);
} /* main */
```

INDEX:
=====

1.5 .... #
0.3.1.2 .... $
0.3.1.2 .... *
2.3.1 .... ≡
2.3.2 .... ≡
2.3.5 .... _ZF_
2.3.5 .... _ZS_
2.4.3 .... ∩
2.4.3 .... ∪
0.3.1.4 .... |v|
2.3.1 .... A
2.5.6 .... AZF
2.5.6 .... AZS
0.3.1.2 .... A*
1.3.2 .... Aho-Corasick-Algorithmus
2.3.1 .... Alfabet
2.5.6 .... Anfangs-ZF
2.5.6 .... Anfangs-ZF-Struktur
2.5.6 .... Anfangs-Zeichenfeld
0.3.1.4 .... Anfangsstueck
1.3.2 .... Automat
1.4.5 .... B
1.4.5 .... B[w]
0.3.2.1 .... Baum
2.3.1 .... Belegung
1.5.6 .... Bi-Tree
0.3.2.1 .... Blatt
1.4.5 .... Block
1.4.5 .... Block-Beginn-Position
1.4.7 .... Block-End-Position
2.4.3 .... Durchschnitt
1.4.7 .... E
2.3.5 .... EINMAL IN
2.4.8 .... EPosid
1.4.7 .... E[w]
2.4.8 .... Einzel-Posid
2.4.8 .... Einzel-Positions-Identifier
0.3.1.4 .... Endstueck
2.3.5 .... F
1.9.6 .... File-Compare-Problem
1.9.5 .... File-Transmission-Problem
2.4.8 .... GPosid
2.4.8 .... Gesamt-Posid
2.4.8 .... Gesamt-Positions-Identifier
2.4.8 .... Gesamt-Pseudo-Positions-Identifier
0.3.2.1 .... Grad
0.3.2.1 .... Hoehe
1.4.1 .... I(i)
2.3.5 .... IN
1.4.1 .... I*(i)
1.4.1 .... I~(i)
2.3.1 .... Iso
1.3.4 .... Karp-Miller-Rosenberg-Algorithmus
1.3.1 .... Knuth-Morris-Pratt-Algorithmus
2.3.5 .... LINKSBUENDIG IN
2.3.1 .... Label
0.3.1.4 .... Laenge
2.6.2 .... Links-Erweiterbarkeit

1.4.8 .... M
2.3.5 .... MEHRMALS IN
2.5.14 .... MPM
2.5.16 .... MPM mit Rand
2.5.16 .... MPMR
2.2.4 .... M[w,v]
1.4.8 .... M[w]
2.2.4 .... M[w]
1.5.9 .... McCreight-Algorithmus
2.2.4 .... Minid
1.4.8 .... Minid
1.4.8 .... Minimal-Identifier
2.2.4 .... Minimal-Identifier
1.4.15 .... N
0.3.2.1 .... NIL
1.4.15 .... N[w]
2.6.5 .... Nachf([p])
2.6.1 .... Nachf(p)
2.6.9 .... Nachfolge-Graph
2.6.1 .... Nachfolger
2.3.1 .... P
2.4.1 .... PZF
2.4.1 .... PZS
1.2 .... Pattern
1.2 .... Pattern-Matching-Problem
1.4.1 .... Posid
2.3.1 .... Position
1.4.1 .... Position-Identifier
0.3.1.4 .... Prefix
1.7.11 .... Prefix-Bi-Tree
1.5.6 .... Prefix-Bi-Tree
1.4.17 .... Prefix-Tree
2.5.5 .... Prf(M)
2.5.5 .... Prf(f)
2.4.1 .... Pseudo-Zeichenfeld
2.4.1 .... Pseudo-Zeichenfeld-Struktur
2.9.2 .... R
1.2 .... Repetition-Finding
2.9.2 .... Rf
2.9.2 .... Richtung
1.3.3 .... Rosenberg-Algorithmus
1.3.2 .... Sprache
2.3.1 .... Start
1.3.0 .... Straight-Forward-Pattern-Matching-Algorithmus
0.3.1.2 .... String
2.3.1 .... Struktur
1.2 .... Suchphase
0.3.1.4 .... Suffix
2.5.20 .... Suffix
1.5.9 .... Suffix-Link
0.3.1.4 .... Teilstring
0.3.2.1 .... Tiefe
0.3.2.1 .... Vaterknoten
2.4.3 .... Vereinigung
1.2 .... Vorverarbeitungsphase
1.5.7 .... Weiners Alg.B
1.5.8 .... Weiners Alg.C
1.5.6 .... Weiners Alg.D
2.3.5 .... XF
2.3.1 .... Xf

2.2.1 .... ZF
2.3.1 .... ZF
2.3.1 .... ZS
2.2.1 .... Zeichenfeld
2.3.1 .... Zeichenfeld
2.3.1 .... Zeichenfeld-Struktur
2.9.2 .... Zeichenfeld-Strukturmenge mit Richtungen
0.3.1.2 .... Zeichenkette

0.3.1.2 .... a^n
2.4.5 .... abgeschlossen gegenueber Durchschnitt
2.4.5 .... abgeschlossen gegenueber Vereinigung
1.4.5 .... b(i)
2.5.5 .... dPrf(f)
2.3.4 .... dein
2.3.4 .... din
2.3.4 .... direkt in
2.3.4 .... direkt linksbuendig in
2.5.4 .... direkt unmittelbar linksbuendig in
2.5.5 .... direkter Prefix
2.3.4 .... dlin
2.5.4 .... dulin
2.2.2 .... echt in
0.3.1.4 .... echter Prefix
0.3.1.4 .... echter Suffix
2.3.4 .... ein
2.2.2 .... ein
2.4.5 .... eindeutig halb abgeschlossen gegenueber Durchschnitt
2.4.5 .... eindeutig halb abgeschlossen gegenueber Vereinigung
2.3.4 .... einmal direkt in
2.2.2 .... einmal echt in
2.3.4 .... einmal in
2.2.2 .... einmal in
1.3.2 .... endlicher Automat
0.3.2.1 .... father
0.3.2.1 .... follow(u)
0.3.1.2 .... freies Monoid
2.4.5 .... halb abgeschlossen gegenueber Durchschnitt
2.4.5 .... halb abgeschlossen gegenueber Vereinigung
2.3.4 .... in
2.2.2 .... kommt einmal vor
2.2.1 .... kommt mehrmals vor
2.2.1 .... kommt vor in
1.5.8 .... kompaktifiziert
0.3.2.1 .... label(n)
0.3.1.7 .... lexikografische Ordnung
2.3.4 .... lin
2.6.2 .... links-erweiterbar
2.3.4 .... linksbuendig in
1.4.15 .... maximaler Mehrfachstring
0.3.1.9 .... maximaler gemeinsamer Prefix
2.5.14 .... maximales positionsgebundenes Mehrfach-Zeichenfeld
2.2.1 .... mehrmals in
2.3.4 .... mehrmals in
2.2.1 .... min
2.3.4 .... min
2.5.17 .... min bzgl. den Positionen
0.3.3.1 .... minimales Element
0.3.3.2 .... minimales Element
2.6.1 .... neben
2.6.8 .... nicht zusammenhaengend
1.5.6 .... p-Tree

1.3.2 .... regulaere Sprache
2.6.9 .... root
0.3.2.1 .... root
1.5.6 .... s-Tree
0.3.2.1 .... son
2.3.1 .... st
2.5.4 .... ulin
2.5.4 .... unmittelbar linksbuendig in
2.6.1 .... unmittelbar vor
2.6.1 .... uvor
0.3.2.1 .... verfolgen
2.5.1 .... vertraeglich mit lin
2.6.1 .... vor
2.5.17 .... vorlaeufiger MPMR
0.3.1.6 .... w[i...j]
0.3.1.6 .... w[i...k...j]
0.3.1.6 .... w[i]
2.2.2 .... xein
2.2.2 .... xin
2.6.8 .... zusammenhaengend