1.5 Priority Queues

Operationen:

- insert (i, h): Füge das Element i in die Priority Queue h, die i nicht bereits enthält, ein.
- deletemin (h): Streiche ein Element mit minimalem Schlüssel aus h und gib dieses Element aus. Falls h leer ist, dann gib "empty" aus.
- makepq (S): Konstruiere eine neue Priority Queue zur Speicherung der Menge S.

zusätzliche Operationen:

- findmin (h): wie deletemin(h), nur finde statt streiche
- delete (i, h): Streiche das Element i aus der Priority Queue h.
- union(h_1, h_2): Kreiere eine neue Priority Queue, die alle Elemente der disjunkten PQ h_1 und h_2 enthält. h_1 und h_2 existieren anschließend nicht mehr.

Verwendung von balancierten Bäumen:

Operation	Aufwand
insert (i,h) deletemin (h) delete (i,h)	$O(\log n)$
findmin (h)	$O(\log n)$ oder $O(1)$ bei Verwendung eines zusätzlichen Zeigers.
makepq (S)	$O(\lvert S\rvert \cdot \log \lvert S\rvert)$
union (h_1, h_2)	$O(\min\{\lvert h_1\rvert(1+\log\lvert h_2\rvert), \lvert h_2\rvert(1+\log\lvert h_1\rvert)\})$

Ziel: effizientere Lösung.

Ein Baum heißt heapgeordnet, falls auf jedem Pfad von einem Blatt zur Wurzel die Größe der Schlüssel monoton fällt.

1.5.1 d-Heaps

- geeignet, falls keine union-Operationen durchzuführen sind.

Ein d-näer Baum der Höhe t heißt vollständig, falls gilt:

1) Jeder Knoten der Tiefe $< t-1$ hat d Söhne.

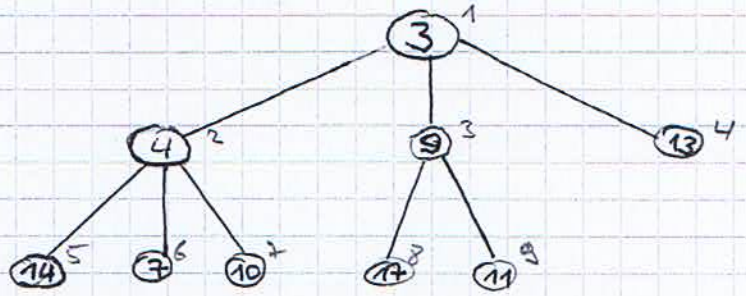
2) Maximal ein Knoten der Tiefe $t-1$ hat $\ell \notin \{0, d\}$ Söhne.

3) Die Anzahl der Söhne der Knoten der Tiefe $t-1$ ist "von links nach rechts" monoton fallend.

Ein <u>d-Heap</u> ist ein heapgeordneter vollständiger d-närer Baum.

<u>Beispiel</u>: $d = 3$, $S = \{3, 4, 7, 9, 10, 11, 13, 14, 17\}$

$n := |S|$



Regelmäßige Struktur von d-Heaps

$\rightsquigarrow$

Realisierung ohne Zeiger in einem Array ist möglich, nämlich:

- Nummeriere die Knoten zeilenweise von links nach rechts und speichere sie im Array in den korrespondierenden Positionen.

3	4	9	13	14	7	10	17	11
1	2	3	4	5	6	7	8	9

$\Rightarrow$

- Söhne des j-ten Knotens stehen in den Feld-komponenten

$$d(j-1)+2,\ d(j-1)+3,\ \dots,\ \min\{dj+1, n\}.$$

- Der Vater des j-ten Knotens steht an der Stelle $\left\lceil \frac{j-1}{d} \right\rceil$.

<u>Bezeichnungen:</u>

- $h = (V, E)$ d-Heap mit n Knoten
- $f(v)$, $v \in V \setminus \{\text{Wurzel}(h)\}$ Vater von v
- $k(v)$ der in v gespeicherte Schlüssel
- $\text{num}(v)$ Nummer von v in obiger Nummerierung

- <u>insert (x, h)</u>:

  insert (2, h) anhand des Beispiels erklären

  <u>Aufwand</u>: $O(\log_d n)$

- <u>delete (x, h)</u>:

  <u>Annahme</u>: Knoten i mit $k(i) = x$ ist bekannt.

  delete (4, h) anhand des Beispiels erklären. Beispiel erweitern, so dass delete-Operation, nach der nach oben Heap in Ordnung gebracht werden muss, erklärt werden kann und solche erklären.

  <u>Aufwand</u>: $O(d \cdot \log_d n)$. (4 mit Schlüssel, die Stellen

- <u>deletemin (h)</u>:

  (1) Gib $k(1)$ aus;

  (2) delete $(k(1), h)$.

  <u>Aufwand</u>: $O(d \cdot \log_d n)$.

- <u>findmin (h)</u>:

  (1) Gib $k(1)$ aus.

  <u>Aufwand</u>: $O(1)$.

- <u>makepq (S)</u>:

  (1) Konstruiere einen vollständigen d-nären Baum T mit $n := |S|$ Knoten;

  (2) Fülle T rückwärts auf und halte dabei die Heapordnung aufrecht (d.h., lasse gegebenenfalls den neuen Schlüssel nach unten sinken).

  anhand des Beispiels erläutern

  <u>Aufwand</u>:

  - Für jeden Knoten, der Wurzel eines Teilbaumes der Höhe j ist: $O(j \cdot d)$.

  - Anzahl solcher Knoten:

    $j = \lceil \log_d n \rceil$ $\qquad 1 \leq \frac{n}{d^{\lceil \log_d n \rceil - 1}}$

$j = \lceil \log_d n \rceil - 1 \qquad d \leq \frac{n}{d^{\lceil \log_d n \rceil - 2}}$

$\vdots$

$j = \lceil \log_d n \rceil - i \qquad d^i \leq \frac{n}{d^{\lceil \log_d n \rceil - (i+1)}}$

$\Longrightarrow$

Gesamtaufwand =

$$O\left(\sum_{i=0}^{\lceil \log_d n \rceil} \frac{n(i+1)d}{d^i} \right)$$

$$= O\left(d \cdot n \cdot \underbrace{\sum_{i=0}^{\lceil \log_d n \rceil} \frac{(i+1)}{d^i}}_{O(1)} \right) = O(d \cdot n)$$

- <u>union (h_1, h_2)</u>:
  - nicht in logarithmischer Zeit.
  - möglich mittels

    $|h_1|$ insert-Operationen $(|h_1| \leq |h_2|)$

    <u>Aufwand</u>: $O(|h_1| \cdot \log_d |h_2|)$

    oder

    einer makepq-Operation

    <u>Aufwand</u>: $O(d(|h_1| + |h_2|))$

<u>Bemerkung</u>:

Datenstrukturen, die die Durchführung einer union-Operation in logarithmischer Zeit ermöglicht. Linksbaum AuD. 4=

2. Durchmusterung von Graphen

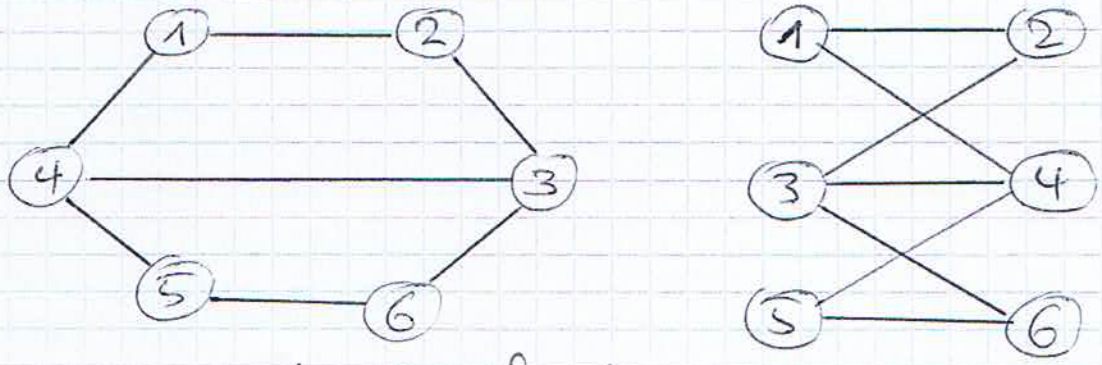
2.1 Graphentheoretische Grundlagen

<u>Graph</u> $G = (V, E)$ $\quad$ V Menge der Knoten
E Menge der Kanten

- ungerichtet: $e \in E$ ungeordnetes Paar $\{v, w\}$ $\quad v, w \in V$
- gerichtet: geordnetes Paar (v, w)

- bipartit: $V = A \dot{\cup} B$, $E \subseteq A \times B$
  Schreibweise: $G = (A, B, E)$

<u>Beispiel</u>:



derselbe Graph

$v, w \in V$.

<u>Pfad</u> P von v nach w: Folge von Knoten und Kanten:

$$P = [v = v_0], (v_0, v_1), v_1, (v_1, v_2), v_2, \ldots, v_{k-1}, (v_{k-1}, v_k), [v_k = w]$$

$(v_i, v_{i+1}) \in E$ für $0 \leq i < k$.

Länge von P: Anzahl der Kanten $|P|$

- <u>einfach</u>: $v_i \neq v_j$ für $0 \leq i < j \leq k$.
- <u>Kreis</u>: $v_0 = v_k$ und $|P| \geq 1$
- <u>kreisfrei (azyklisch)</u>: G enthält keinen Kreis.

Sei $G = (V, E)$ ungerichtet, $u \in V$

<u>Grad</u> $\deg(u)$ von u:

- Anzahl der zu u inzidenten Kanten

$\deg(u) = |\{(u,v) \mid (u,v) \in E\}|$

Sei $G = (V, E)$ gerichtet, $v \in V$

<u>Eingangsgrad</u> $\mathrm{indeg}(v)$

- Anzahl der eingehenden Kanten

$\mathrm{indeg}(v) = |\{(u,v) \mid (u,v) \in E\}|$

<u>Ausgangsgrad</u> $\mathrm{outdeg}(v)$

- Anzahl der ausgehenden Kanten

$\mathrm{outdeg}(v) = |\{(v,u) \mid (v,u) \in E\}|$

$G' = (V', E')$ <u>Teilgraph</u> von $G = (V, E)$, falls

$V' \subseteq V$ und $E' \subseteq E \cap (V' \times V')$

G' <u>induzierter Teilgraph</u> von G, falls

$V' \subseteq V$ und $E' = E \cap (V' \times V')$

Beisp.:

G <u>zusammenhängend</u>

$\nexists\ V_1, V_2 \subseteq V$ mit:

- $V_1 \cup V_2 = V$
- $V_1, V_2 \neq \emptyset$, und $V_1 \cap V_2 = \emptyset$
- $E \cap V_1 \times V_2 = \emptyset$

  d.h., keine Kante zwischen V_1 und V_2.

<u>Zusammenhangskomponenten</u> von G:

- maximaler zusammenhäng. induzierter Teilgraph
- $G' = (V', E')$ Zusammenhangskomponente

  $\Rightarrow (V' \cup \{u\}, \overline{E})$ $u \in V \setminus V'$ ist nicht zusammenhängend

Ein <u>ungerichteter Baum</u> ist ein kreisfreier zusammenhängender ungerichteter Graph.

<u>ungerichteter Wald</u> ungerichteter Graph, dessen Zusammenhangskomponenten Bäume sind.

- ungerichtet wird normalerweise weggelassen.
- alternative Definitionen. (Siehe A u D, S. 46-48)

~~Satz 2.1~~

Sei $T = (V, E)$ ~~ein ungerichteter Graph mit $|V| = n \geq 2$. Folgende Eigenschaften sind äquivalent:~~

$\Rightarrow$ $\exists$ Kante $e \in P_2 \setminus P_1$ od. $e \in P_1 \setminus P_2$.

Nach Entfernen von e ist T noch wie vor zshgd.

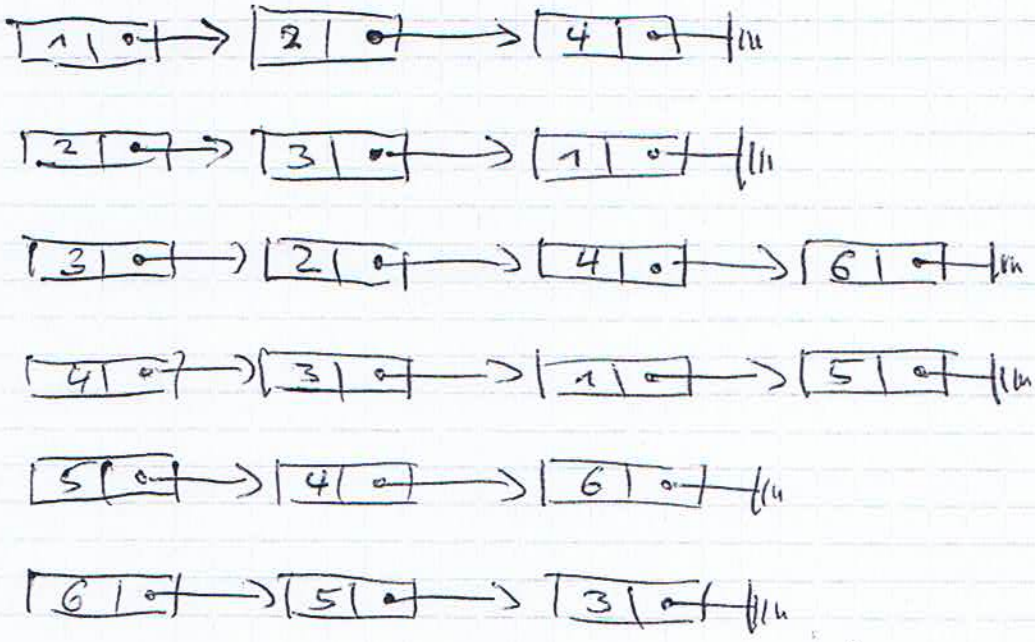
$\underline{G \Rightarrow 1.}$ ✓

<u>Darstellung von Graphen:</u> $G = (V, E)$

- <u>Nachbarschaftslisten</u>

$|V|$ verkettete Listen. Für $v \in V$ enthält Liste(v) genau dann $w \in V$, wenn $(v, w) \in E$.

<u>Bsp.:</u>



<u>Platzbedarf:</u> gerichtet: $|V| + |E|$

ungerichtet: $|V| + 2 \cdot |E|$

<u>Zugriff auf Kante:</u> $\mathcal{O}(\max_v \deg(v))$

Nachbarschaftsmatrix.

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	1	0	0	0
3	0	1	0	1	0	1
4	1	0	1	0	1	0
5	0	0	0	1	0	1
6	0	0	1	0	1	0

$(n \times n)$-Matrix $B = (b_{ik})_{nn}$

$$b_{ik} = \begin{cases} 1 & \text{falls } (i,k) \in E \\ 0 & \text{sonst} \end{cases}$$

Platzbedarf: $|V|^2$

Zugriff auf Kante: $\Theta(1)$

2.2 Graphdurchmusterungsmethoden

Menge Q enthält alle bisher besuchten Knoten, die noch nicht vollständig bearbeitet sind.

N[v] Nachbarschaftsliste für den Knoten v.

Algorithmus ALGORITHMISCHE SUCHE

Eingabe: Graph $G = (V,E)$ repräsentiert durch seine Nachbarschaftslisten, Startknoten s

Ausgabe: Hängt von der Anwendung ab. Die von s erreichbaren Knoten sind markiert

Methode:

```
for alle v ∈ V
  do
    H[v] := N[v]
  od;
Q := {s};
markiere s;
while Q ≠ ∅
  do
    wähle v ∈ Q;
    if H[v] ≠ ∅
      then
        wähle v' ∈ H[v];
        H[v] := H[v] \ {v'}
        if v' nicht markiert
          then
            Q := Q ∪ {v'};
            markiere v'
        fi
      else
        Q := Q \ {v}
    fi
  od.
```

Aufwand:

Knoten $v \in V$: $\leq (\text{outdeg}(v) + 1)$-mal behandelt

Kante $e \in E$: höchstens einmal betrachtet.

Annahme:

"wähle $v \in Q$" bzw. "wähle $v' \in H[v]$" kann jeweils in konstanter Zeit durchgeführt werden.

$\rightsquigarrow$

insgesamt $O(|V| + |E|)$ Zeit.

Frage: Wie verwalten wir die Menge Q?

- Q durch einen Keller $\rightsquigarrow$ Tiefensuche (depth first search)
- Q durch eine Schlange $\rightsquigarrow$ Breitensuche (breadth first search)

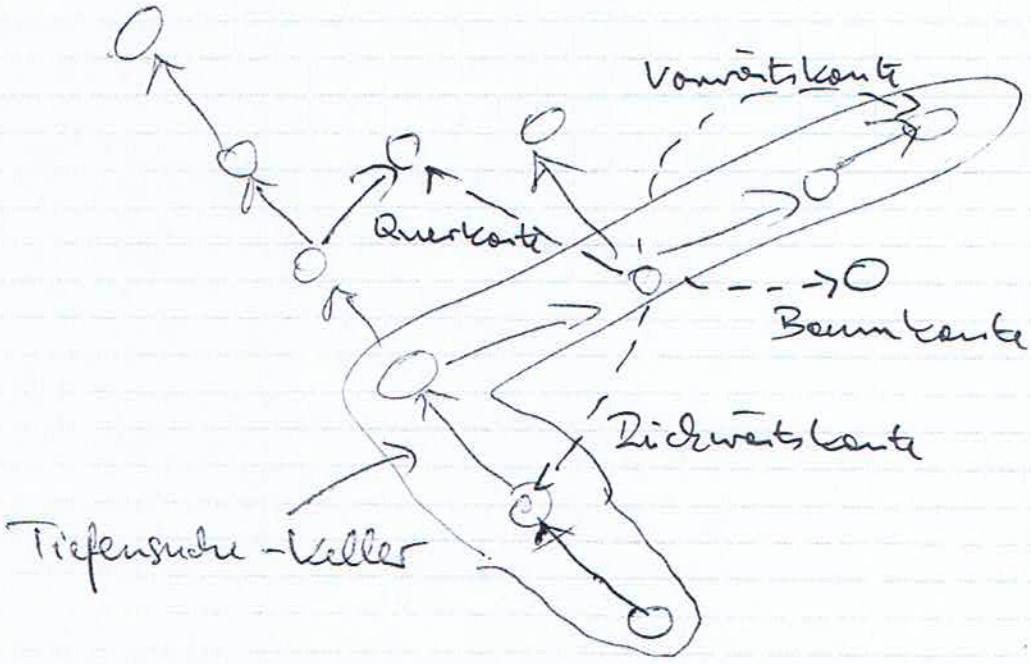
Eigenschaften von Tiefensuche:

Wählen von $v' \in H[v]$ $\exists$ Betrachten der Kante $(v, v') \in E$.

- Baumkanten: Kanten, die während der Suche zu neuen Knoten führen (d.h., v' ist noch nicht markiert).
- Rückwärtskanten: Kanten, die bzgl. Baumkanten von Nachfolgern zu Vorgängern führen (insbesondere $v' \in Q$).

<u>Vorwärtskanten</u>: Kanten, die bzgl. Baum-Kanten von Vorgängern zu Nachfolgern führen und keine Baum-Kanten sind (insb. $v' \notin Q$).

<u>Querkanten</u>: Kanten, die Knoten miteinander verbinden, die bzgl. Baumkanten weder Vorgänger noch Nachfolger miteinander sind (insb. $v' \notin Q$).



Es gilt:

1. Einfügen des Knotens v' in Q $\equiv$ Operation PUSH(v') (Baumkante)

2. Streichen des Knotens v aus Q $\equiv$ POP (alle Kante $(v, v') \in E$ sind bereits betrachtet.)

3. Tiefensuche-Keller Q enthält immer einen einfachen Pfad von s zum obersten Knoten in Q.

Satz 2.1

Sei $G = (V, E)$ ein gerichteter Graph und $s, t \in V$. Dann konstruiert eine Tiefensuche auf G mit Startknoten s immer einen einfachen Pfad von s nach t, falls solcher existiert.

Beweis:

Ann.:

$\exists$ einfacher Pfad

$$P = (s = v_0), v_1, v_2, \dots, (v_k = t),$$

aber Tiefensuche auf G mit Startknoten s konstruiert keinen einfachen Pfad von s nach t.

Sei v_i der erste Knoten auf P, für den die Operation PUSH(v_i) niemals ausgeführt wird.

Da PUSH(v_k) niemals ausgeführt wird, existiert v_i.

Da Tiefensuche die Operation PUSH(v_0) durchführt, ist PUSH(v_{i-1}) ausgeführt worden.

$\rightsquigarrow$ Übungsaufgabe

Irgendwann ist die Kante (v_{i-1}, v_i) betrachtet worden.

$\Rightarrow$ (v_{i-1}, v_i) keine Baumkante, da ansonsten PUSH(v_i) aufgrund der Betrachtung

$\Rightarrow$ (v_{i-1}, v_i) ist Vorwärts-, Rückwärts- oder Querkante. Dann ist aber die Operation PUSH(v_i) bereits bei Betrachtung der Kante

durchgeführt worden.

Topologische Suche

$\forall v \in V$ Tabelle $NZB[v]$

- zu Beginn: $NZB[v] := indeg(v)$
- Betrachtung einer eingehenden Kante von v
  $\rightsquigarrow$ $NZB[v] := NZB[v] - 1$
- Falls $NZB[v] = 0$, dann
  $Q := Q \cup \{v\}$.

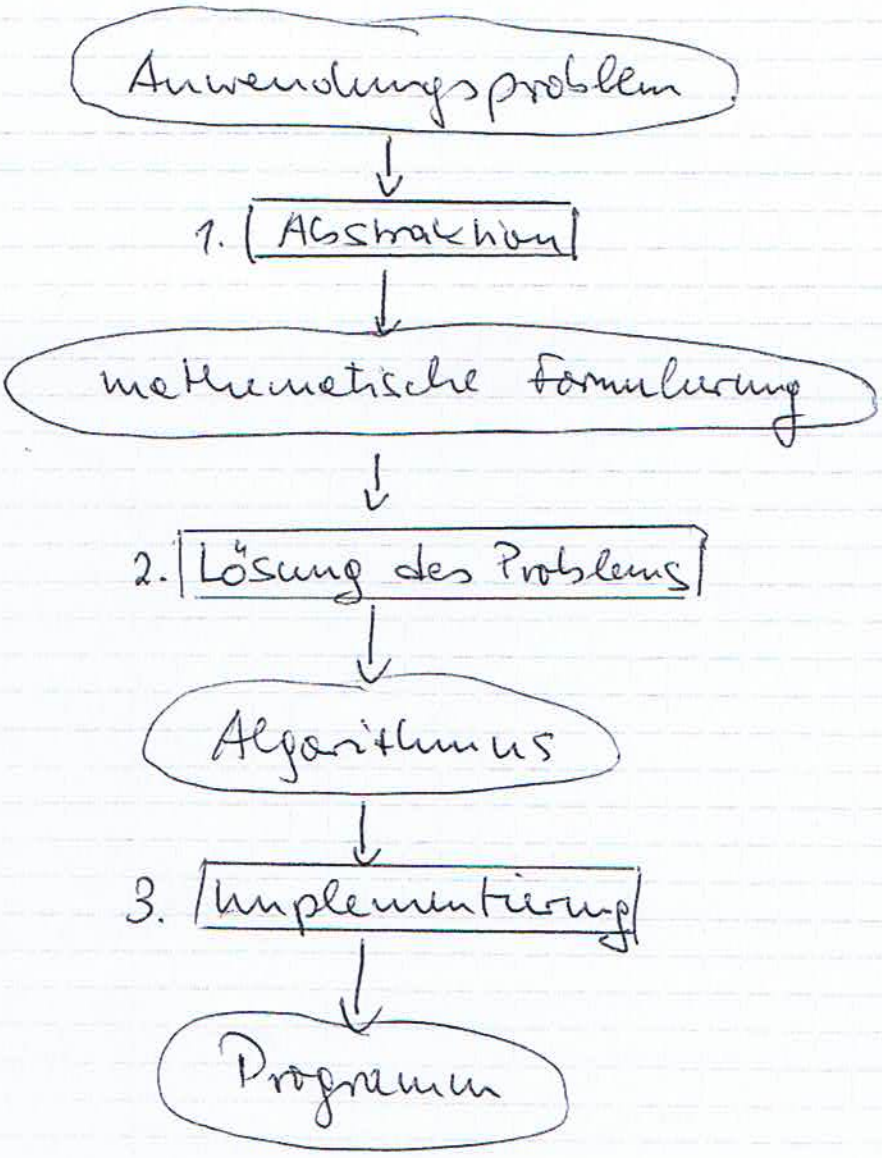
Rückwärts-Tiefensuche, Rückwärts-Breitensuche

Rückwärts-Topologische Suche

entgegen Kantenrichtung.

3. Über den Entwurf von Algorithmen

übliche Vorgehensweise:



Ziel: Beschäftigung mit Schritt 2.

2. Problemanalyse nebst Überprüfung der Anwendbarkeit der Paradigmen.
⇒
Entwicklung des konkreten Algorithmus

3.1 Divide and Conquer

gegeben: Problem P der Größe n mit

- Teilprobleme haben dieselbe Struktur.

Divide-and-Conquer-Paradigma:

1. Divide: Teile das Problem in eine Anzahl von Teilproblemen auf.
2. Conquer: Falls die Teilproblemgröße klein genug ist, dann löse die Teilprobleme direkt. Andernfalls löse die Teilprobleme rekursiv auf dieselbe Art und Weise.
3. Combine: Kombiniere die Lösungen der Teilprobleme zu einer Gesamtlösung des ursprünglichen Problems.

Voraussetzungen für die Anwendbarkeit:

- Aufteilbarkeit in ≥ 2 Teilprobleme
- effiziente Kombinierbarkeit der Teillösungen zu einer Gesamtlösung

Beispiel: Mergesort.

gegeben: Folge $x_1, x_2, \ldots, x_n$ von n Zahlen

gesucht: Permutation $y_1, y_2, \ldots, y_n$ der Eingabe, so daß $y_1 \leq y_2 \leq \ldots \leq y_n$

Annahme: $n = 2^k$ für ein $k \in \mathbb{N}_0$

Strategie:

1. Divide: Teile die n-elementige Folge in zwei Teilfolgen, bestehend aus jeweils $n/2$ Elementen

2. Conquer: Unter Verwendung von Mergesort, sortiere die beiden Teilfolgen rekursiv. Folge der Länge 1 ist sortiert.

3. Combine: Mische die beiden sortierten Folgen zu einer sortierten Folge.

Prozedur MERGE(A, p, q, r), wobei

- A ein Feld
- p, q, r Indizes von Feldelementen mit $p \leq q \leq r$ und
- Teilfelder $A[p..q]$ und $A[q+1..r]$ aufsteigend sortiert

sind.

Resultat: Beide Teilfelder werden zu einem einzelnen Teilfeld gemischt, das dann das aktuelle Teilfeld $A[p..r]$

ersetzt.

MERGESORT (A, p, r)

- sortiert das Teilfeld $A[p..r]$.
- Falls $p \geq r$, dann besitzt das Teilfeld maximal ein Element und ist daher sortiert.
- Falls $p < r$, dann berechnet der Divide-Schritt den Index q, der $A[p..r]$ in zwei gleichgroße Teilfelder $A[p..q]$ und $A[q+1..r]$ aufteilt.

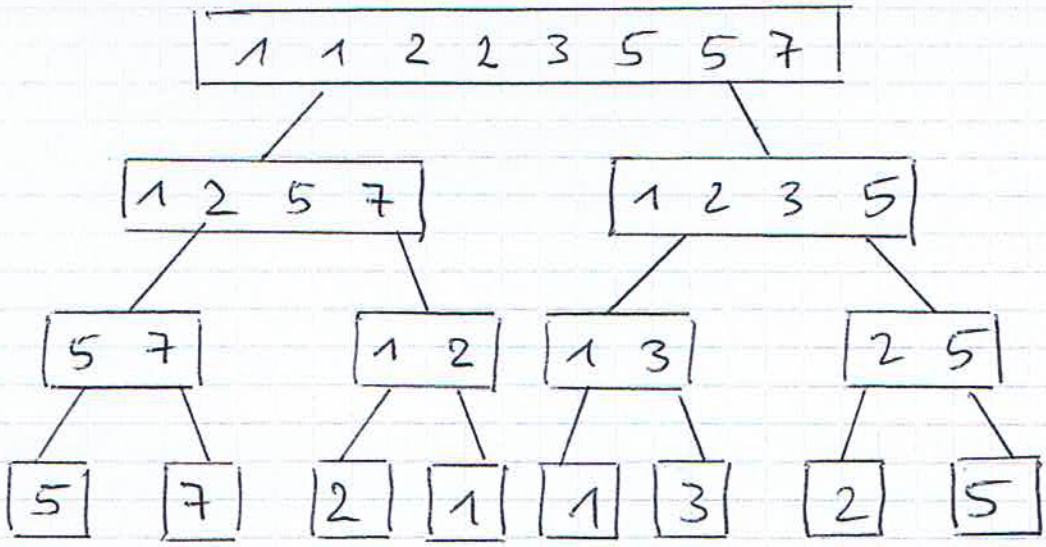
⇝ rekursiven Prozedur

```
MERGESORT (A, p, r)

    if p < r
    then
            q := ⌊(p+r)/2⌋
            MERGESORT(A, p, q)
            MERGESORT(A, q+1, r)
            MERGE (A, p, q, r)
    else
            Gib A[p..r] aus
    fi.
```

Beispiel:

$A = \langle 5, 7, 2, 1, 1, 3, 2, 5 \rangle$.



<u>Aufwand:</u>

$T(n)$ benötigte Zeit für ein Problem der Größe n.

<u>Annahme:</u>
- $T(c) = O(1)$ für eine Konstante c
- Divide-Schritt teilt ein Problem der Größe n in a gleichgroße Probleme der Größe $\frac{n}{b}$.

Rekurrenzgleichung:

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n \leq c \\ a \cdot T(\frac{n}{b}) + D(n) + C(n) & \text{sonst} \end{cases}$$

$D(n)$: Kosten für Aufteilen

$C(n)$: Kosten für Combine.

<u>Mergesort:</u> $c = 1$, $a = b = 2$ und

$D(n) + C(n) = \Theta(n)$,

was folgende Rekursionsgleichung ergibt.

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n \leq 1 \\ 2 \cdot T(n/2) + \Theta(n) & \text{sonst.} \end{cases}$$

Falls n eine Potenz von 2 ist, dann erhalten wir:

$$T(n) = 2^{\log n} \cdot T(1) + \sum_{i=1}^{\log n} 2^{i-1} \cdot \Theta\left(\frac{n}{2^{i-1}}\right)$$

$$= 2^{\log n} \cdot T(1) + \sum_{i=1}^{\log n} \Theta(n)$$

$$= \Theta(n) + \Theta(n \cdot \log n)$$

$$= \Theta(n \cdot \log n).$$

<u>Bemerkung:</u>

Herleitung der allgemeinen Rekursionsgleichung (und mehr):

Siehe
Cormen, Leiserson, Rivest, Stein: "Introduction to Algorithms", Chapter 4.

3.2 Dynamische Programmierung

Idee:

Lösung des Gesamtproblems durch Kombination von Lösungen von Teilproblemen

aber:

auch anwendbar, wenn die Teilprobleme nicht disjunkt sind, sondern wiederum gemeinsame Teilprobleme haben. ($\Leftrightarrow$ divide-and-conquer)

Anwendbarkeit setzt voraus:

Optimalitätsprinzip:

Jede Teillösung einer optimalen Lösung, die Lösung eines Teilproblems ist, ist selbst eine optimale Lösung des betreffenden Teilproblems.

Dynamische Programmierung berechnet für immer größer werdenden Teilprobleme optimale Lösungen, bis schließlich eine optimale Lösung des Gesamtproblems berechnet ist.

Beispiel (optimale binäre Suchbäume)

gegeben: statische, sortierte Menge $S = \{x_1, x_2, \ldots, x_n\}$, d.h., $x_1 < x_2 < \ldots < x_n$.

Zugriffsverteilung $(\alpha_0, \beta_1, \alpha_1, \ldots, \beta_n, \alpha_n)$ bezüglich der Operation Zugriff (σ), wobei

$$\alpha_j = \begin{cases} \text{prob}(a < x_1) & \text{falls } j = 0 \\ \text{prob}(x_j < a < x_{j+1}) & \text{falls } 0 < j < n \\ \text{prob}(x_n < a) & \text{falls } j = n \end{cases}$$

und

$$\beta_i = \text{prob}(a = x_i) \quad \text{für } 1 \leq i \leq n.$$

Da $(\alpha_0, \beta_1, \alpha_1, \beta_2, \ldots, \beta_n, \alpha_n)$ eine Wahrscheinlichkeitsverteilung ist, gilt:

$$\alpha_j \geq 0 \quad \text{für } 0 \leq j \leq n$$
$$\beta_i \geq 0 \quad \text{für } 1 \leq i \leq n$$

und

$$\sum_{i=1}^{n} \beta_i + \sum_{j=0}^{n} \alpha_j = 1.$$

<u>Bezeichnungen</u>: (knotenorientierte Speicherung)

- T Suchbaum für die Menge S
- b_i^T Tiefe des Knotens x_i in T
- $a_j^T = \begin{cases} \text{Tiefe des Blattes } (-\infty, x_1) & \text{falls } j = 0 \\ \text{Tiefe des Blattes } (x_j, x_{j+1}) & \text{falls } 0 < j < n \\ \text{Tiefe des Blattes } (x_n, \infty) & \text{falls } j = n \end{cases}$
- <u>gewichtete Pfadlänge</u> P^T des Suchbaumes T:

$$P^T := \sum_{i=1}^{n} \beta_i (1 + b_i^T) + \sum_{j=0}^{n} \alpha_j \cdot a_j^T$$

Bemerkung:

Die mittlere Suchzeit einer Folge von Zugriffsoperationen ist proportional zur gewichteten Pfadlänge des Suchbaumes.

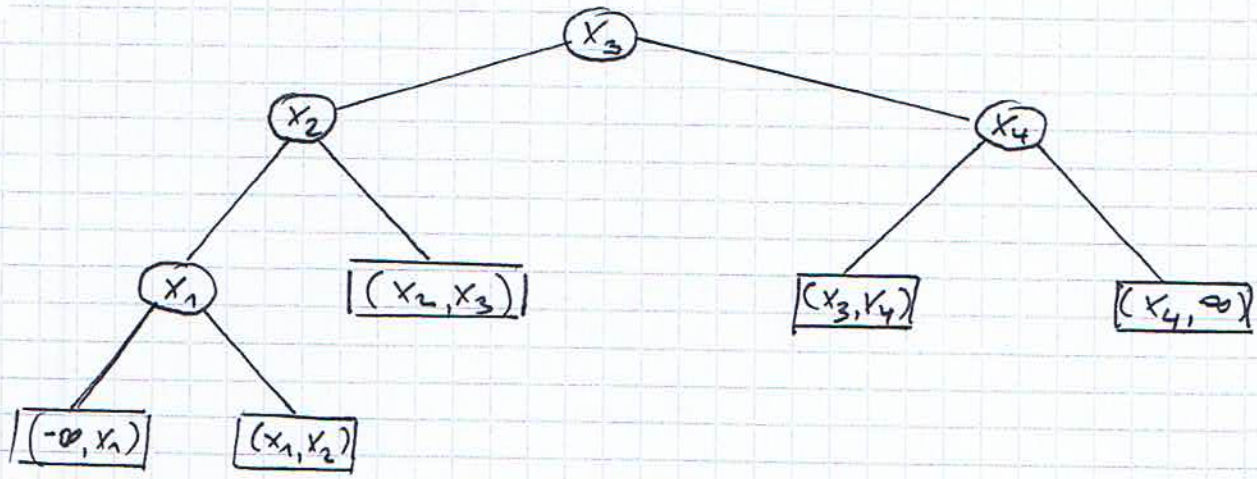
Sei $S = \{x_1, x_2, \ldots, x_n\}$ eine sortierte Menge mit der Zugriffsverteilung $(\alpha_0, \beta_1, \alpha_1, \ldots, \beta_n, \alpha_n)$. Ein Suchbaum T für S heißt optimal, wenn seine gewichtete Pfadlänge P^T minimal ist. D.h.,

$$P^T = \min \{ P^{T'} \mid T' \text{ ist Suchbaum für } S \}.$$

Wir bezeichnen einen optimalen Suchbaum mit T_{opt} und seine gewichtete Pfadlänge mit P_{opt}.

Beispiel:

Betrachte $S = \{x_1, x_2, x_3, x_4\}$ mit Zugriffsverteilung $(1/6, 1/24, 0, 1/8, 0, 1/8, 1/8, 0, 5/12)$. Dann ist folgender Suchbaum optimal:



◊

gesucht: optimaler Suchbaum T_{opt} für S.

Ziel:

Entwicklung eines effizienten Algorithmus zur Berechnung eines optimalen Suchbaumes T_{opt} für eine gegebene sortierte Menge $S = \{x_1, x_2, \ldots, x_n\}$ und gegebener Zugriffsverteilung $(\alpha_0, \beta_1, \alpha_1, \ldots, \beta_n, \alpha_n)$.

Idee:

Verwendung von dynamischer Programmierung

Da die Verwendung von dynamischer Programmierung die Gültigkeit des Optimalitätsprinzips voraussetzt, überprüfen wir zunächst, ob dieses vorliegend in der Tat erfüllt ist.

Beobachtung:

Falls in T_{opt} ein Teilbaum nicht optimal für die korrespondierenden Wahrscheinlichkeiten (oder auch Gewichten) ist, dann könnten wir diesen durch einen besseren Teilbaum ersetzen, was dann auch zur Verbesserung des gesamten Suchbaum führen würde. Dies wäre ein Widerspruch zur Optimalität von T_{opt}.

$\Rightarrow$

In T_{opt} ist jeder Teilbaum ein bezüglich der korrespondierenden Wahrscheinlichkeiten optimaler Suchbaum.

$\rightsquigarrow$

Dynamische Programmierung ist anwendbar.

<u>Bezeichnungen:</u> $(0 \le i \le j \le n)$

- $c(i,j)$ Kosten eines optimalen Suchbaumes mit Gewichten $(\alpha_i, \beta_{i+1}, \alpha_{i+1}, \dots, \beta_j, \alpha_j)$
- $w(i,j) := \sum_{\ell=i+1}^{j} \beta_\ell + \sum_{\ell=i}^{j} \alpha_\ell$.

Betrachten wir x_k mit $i < k \le j$.

<u>Frage:</u>

Was sind die kleinstmöglichen Kosten eines Suchbaumes mit Wurzel x_k und Gewichten $(\alpha_i, \beta_{i+1}, \alpha_{i+1}, \dots, \beta_j, \alpha_j)$?

<u>Antwort:</u>

$$w(i,j) + c(i,k-1) + c(k,j)$$

Also gilt

1. $c(i,i) = 0$
2. $c(i,j) = w(i,j) + \min_{i<k\le j} \{c(i,k-1) + c(k,j)\}$

für $i<j$.

~~Sei für $i<j$~~

~~$R(i,j) := \{ k' \mid i < k' \le j$ und $c(i,k'-1) + c(k',j) = \min_{i<k\le j} \{c(i,k-1) + c(k,j)\}\}$~~

Idee:

Unter Verwendung obiger Rekursionsgleichung für $c(i,j)$ berechne sukzessive für $j-i = 1, 2, \ldots, n$ die Werte $c(i,j)$.

Durchführung:

Der Algorithmus konstruiert optimale Suchbäume $t(i,j)$ für die Gewichte $(\alpha_i, \beta_{i+1}, \alpha_{i+1}, \ldots, \beta_j, \alpha_j)$. Hierzu werden drei Felder

$c[i,j]$	$, 0 \leq i \leq j \leq n$	Kosten von $t(i,j)$
$r[i,j]$	$, 0 \leq i < j \leq n$	Wurzel von $t(i,j)$
$w[i,j]$	$, 0 \leq i \leq j \leq n$	Gesamtgewicht von $t(i,j)$

berechnet.

Das Resultat des Algorithmus wird schließlich durch das Feld r spezifiziert.

Algorithmus OPTSB

Eingabe: Zugriffsverteilung $(\alpha_0, \beta_1, \alpha_1, \ldots, \beta_n, \alpha_n)$

Ausgabe: $T_{opt} = t(0,n)$

Methode:

(1) **for** i **from** 0 **to** n
do
$c[i,i] := 0;$
$w[i,i] := \alpha_i;$

```
        for j from i+1 to n
            do
                w[i,j] := w[i,j-1] + β_j + α_j
            od
        od ;
(2) for j from 1 to n
    do
        c[j-1,j] := w[j-1,j] ;   (* Berechnung der
        r[j-1,j] := j             1-Knoten optimale
    od ;                          Suchbäume *)
(3) for d from 2 to n
    do
        for j from d to n
            do
                i := j-d ;
                c(i,j) := w[i,j] + min_{i<k≤j} { c[i,k-1] + c[k,j] }

                r[i,j] := k_0 ,  für ein k_0, das obiges
                                 Minimum annimmt.
            od
    od .
```

Übung:

Beweisen Sie die Korrektheit des Algorithmus OPTSB.

Aufwandsanalyse:

- Drei Felder der Größe n^2 werden benötigt

$\Rightarrow$

Algorithmus OptSB kann derart implementiert werden, dass $O(n^2)$ Platz ausreicht.

- benötigte Zeit:

(1) $O(n^2)$

(2) $O(n)$

(3)

innere for-Schleife:

- Block wird $(n-d)+1$-mal ausgeführt.
- Jede Ausführung des Blockes benötigt $O(d)$ Zeit

insgesamt: $O(n-d)\,d$ Zeit.

äußere for-Schleife:

führt innere for-Schleife für jedes $2 \leq d \leq n$ aus.

$\Rightarrow$

Gesamtzeit:

$$\sum_{d=2}^{n} (n-d+1)\,d = O(n^3).$$

<u>Bemerkung:</u>

Eine verfeinerte Implementierung, die gewisse Eigenschaften von optimalen binären Suchbäumen ausnutzt, ermöglicht die benötigte Zeit auf $O(n^2)$ zu reduzieren. (siehe z.B. Kurt Mehlhorn, Datenstrukturen und effiziente Algorithmen, S. 147–153)

Untere Schranke ist in jedem anderen Blatt > 6

$\Rightarrow$

~~Umdrehen 0, 1, 3, 2, 0 ist optimal.~~

3.3 Greedyalgorithmen

- Eine einmal getroffene Wahl wird nicht mehr rückgängig gemacht (Daher greedy = gierig)
- Ein Greedyalgorithmus wählt stets diejenige Alternative, die im Moment am besten geeignet erscheint.

Huffman-Codes

Alphabet Σ

binäre Codierung φ von Σ injektive Abb.

$$\varphi : \Sigma \mapsto \{0,1\}^+$$

$\varphi(\Sigma)$ Binärcode für Σ

präfixfrei, falls $\varphi(a)$ kein Präfix von $\varphi(b)$ $\forall\, a,b \in \Sigma,\ a \neq b$.

Sei $t \in \Sigma^+$ ein Text. $\varphi(\Sigma)$ präfixfreier Code

Definiere für $t = a_1 a_2 \dots a_n \in \Sigma^n$

$$\varphi(t) = \varphi(a_1)\varphi(a_2)\cdots\varphi(a_n)$$

Ein präfixfreier Binärcode $\varphi(\Sigma)$ heißt <u>optimal</u> für den Text $t \in \Sigma^+$, falls $\varphi(t)$ minimale Länge hat. D.h.,

$$|\varphi(t)| = \min\{|\varphi'(t)| \mid \varphi'(\Sigma) \text{ präfixfreier Code für } \Sigma\}.$$

<u>Ziel:</u>

Entwicklung eines Greedyalgorithmus, der für einen gegebenen Text $t \in \Sigma^+$ einen optimalen präfixfreien Binärcode $\varphi(\Sigma)$ konstruiert.

<u>Beobachtung:</u>

Ein präfixfreier Binärcode für Σ bzgl. des Textes $t \in \Sigma^+$ kann durch einen Binärbaum $T_\Sigma(t)$ darart repräsentiert werden, so daß gilt:

1) $T_\Sigma(t)$ besitzt $|\Sigma|$ Blätter, die jeweils mit einem (paarweise verschiedenen) Symbol aus Σ beschriftet sind.

2) Wenn wir die linke ausgehende Kante eines inneren Knotens mit 0 und die rechte ausgehende Kante mit 1 beschriften, dann ist $\varphi(a)$ für alle $a \in \Sigma$ gleich der Beschriftung des Pfades von der Wurzel zu dem Blatt, das a enthält.

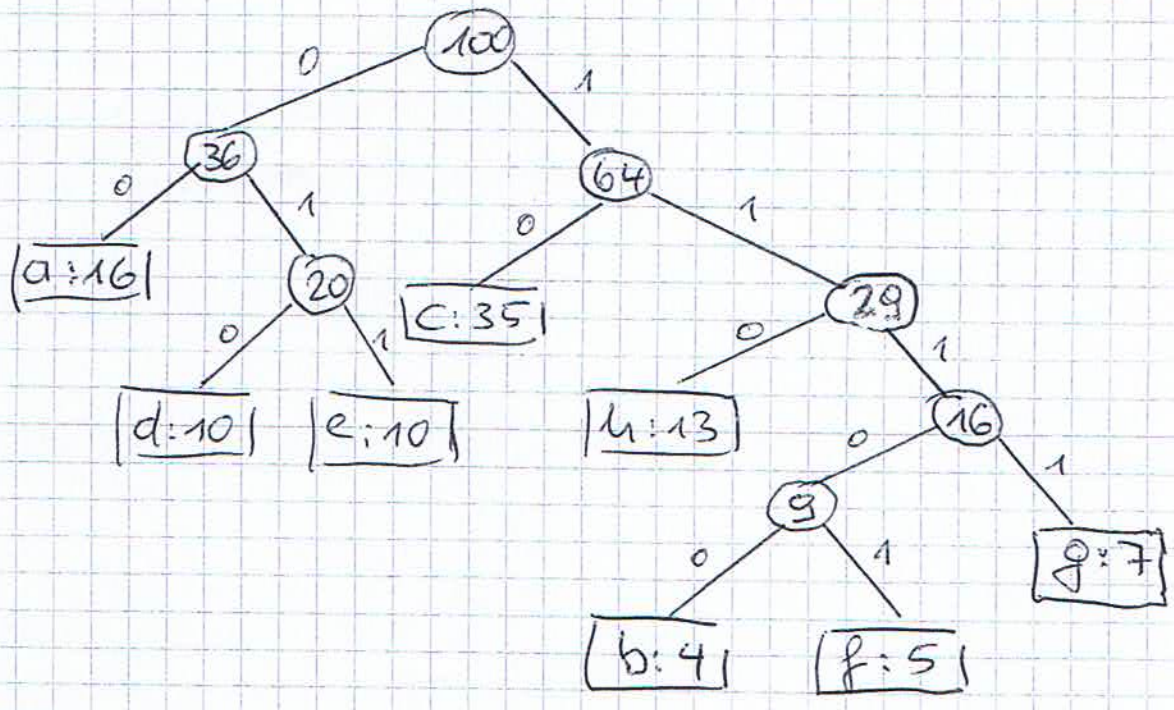
$T_\Sigma(t)$ ist ein Trie, der eine präfixfreie Menge $S \subset \{0,1\}^+$ repräsentiert. Dabei gilt $|S| = |\Sigma|$ und jedes Element in S korrespondiert zu einem Element in Σ.

Bsp.:

Häufigkeiten der Symbole in einem gegebenen Text t:

Symbol	a	b	c	d	e	f	g	h
Häufigkeit	16	4	35	10	10	5	7	13

Beobachtung:
Für die Güte einer Codierung spielen lediglich die Häufigkeiten der Symbole und nicht der konkrete Text t eine Rolle.



$T_\Sigma(t)$ bzgl. opt. präf. Binärcode.

<u>Bezeichnungen:</u> $c \in \Sigma$

- $f_t(c)$ Häufigkeit von c in Text t
- gegeben $T_\Sigma(t)$ für $\varphi(\Sigma)$

$C(T_\Sigma(t))$ # der für $\varphi(t)$ benötigten Bits

Es gilt:

Kosten von $T_\Sigma(t)$

$$C(T_\Sigma(t)) = \sum_{c \in \Sigma} f_t(c) \cdot d_T(c) ,$$

wobei $d_T(c)$ Tiefe des Blattes, das c enthält, ist.

$\rightsquigarrow$

Zu lösen ist folgendes Problem:

<u>gegeben:</u> Alphabet Σ, Text $t \in \Sigma^+$
<u>gesucht:</u> Binärbaum $T_\Sigma(t)$ mit minimalen Kosten.

<u>Algorithmus</u> OPTCODE

<u>Eingabe:</u> Σ, $f_t(c)$ $\forall c \in \Sigma$.

<u>Ausgabe:</u> Binärbaum $T_\Sigma(t)$ mit minimalen Kosten.

<u>Methode:</u>

(1) $n := |\Sigma|$; $Q := \emptyset$;

(2) <u>for</u> alle $c \in \Sigma$

<u>do</u>

Kreiere Blatt b mit Markierung c;

$f(b) := f_t(c)$;

$Q := Q \cup \{b\}$

<u>od</u>;

(3) <u>for</u> $i := 1$ <u>to</u> $n-1$

<u>do</u>

Kreiere neuen Knoten z;

Entferne x mit $f(x)$ minimal aus Q;

Mache x zum linken Sohn von z;

Entferne y mit $f(y)$ minimal aus Q;

Mache y zum rechten Sohn von z;

$f(z) := f(x) + f(y)$;

füge z in Q ein

<u>od</u>;

(4) Gib den Binärbaum mit Wurzel in Q aus.

<u>Satz 3.1</u>

Der Algorithmus OPTCODE berechnet einen Binärbaum $T_\Sigma(t)$ minimaler Kosten.

<u>Beweis:</u>

Für $1 \leq i \leq n-1$

Q_i bezeichnet die Menge Q unmittelbar vor dem i-ten Durchlauf der for-Schleife.

Beobachtung:

- Q_1 enthält die Wurzeln von n Bäumen, die jeweils nur aus der Wurzel bestehen. mit paarweise verschiedenen Buchstaben markiert

$\Longrightarrow$

Jeder Binärbaum $T_\Sigma(t)$ minimaler Kosten enthält jeden Baum, dessen Wurzel in Q_1 ist als Teilbaum.

Annahme:

Ausgabe ist kein Binärbaum minimaler Kosten.

$\Longrightarrow$

Baum, dessen Wurzel in Q_n ist, ist nicht optimal und somit in keinem Binärbaum $T_\Sigma(t)$ minimaler Kosten enthalten.

Betrachte $i_0 \geq 1$ minimal, so daß

1. Es gibt einen Binärbaum $T_\Sigma'(t)$ minimaler Kosten, der jeden Baum, dessen Wurzel in Q_{i_0} ist, enthält.

2. Es gibt keinen ---- $T_\Sigma(t)$ ------ Q_{i_0+1}, enthält.

Sei z ein tiefster Knoten in $T_\Sigma'(t)$, der nicht in einem Baum, dessen Wurzel in Q_{i_0} ist,

enthalten ist. Dann sind sein linker Sohn z_1 und sein rechter Sohn z_2 in Q_{i_0}.

Seien

x und y die während des (i_0+1)-ten Durchlaufs der for-Schleife (3) gewählten Knoten.

Konstruktion $\Rightarrow$

x und y sind in $T'_\Sigma(t)$.

- Tausche die Teilbäume mit Wurzeln z_1 und x in $T'_\Sigma(t)$.

  Da $f(x)$ minimal und z ein tiefster Knoten in $T'_\Sigma(t)$, der nicht in einem Baum, dessen Wurzel in Q_{i_0} ist, enthalten ist

  $\Rightarrow$

  Kosten können sich nicht erhöht haben.

- Tausche die Teilbäume mit Wurzeln z_2 und y.

  Genauso $\Rightarrow$ keine Kostenerhöhung.

$\Rightarrow$

resultierender Baum ist optimal und enthält jeden Baum, dessen Wurzel in Q_{i_0+1} ist.

$\lightning$

Übung: Entwicklung einer $O(n \log n)$ Implementierung

4. Automatentheorie und formale Sprachen

Teil 2 : Die Syntaxanalyse.

stellt fest, ob das gegebene Programm syntaktisch korrekt ist.

zzgl. (falls korrekt)

Syntaxstruktur durch Ableitungsbaum

Falls nicht korrekt

$\rightsquigarrow$ · Fehlermeldung

· Fortsetzung der Syntaxanalyse an der "frühest möglichen" Stelle mit dem Ziel

möglichst viele Fehler zu lokalisieren und zu melden.

Definition der Syntax durch cfg.

4.1 Kontextfreie Grammatiken

Eine kontextfreie Grammatik (cfg) ist ein 4-Tupel $G = (V, \Sigma, P, S)$, wobei

1. das Vokabular V eine nichtleere, endliche Menge,
2. $\emptyset \neq \Sigma \subseteq V$ das Terminalalphabet,
3. $S \in V \setminus \Sigma =: N$ das Startsymbol und
4. das Produktionen- oder Regelsystem eine endliche Teilmenge von $N \times V^*$

sind.

- Elemente von N — Nichtterminale oder Variablen
- Elemente von P — Produktionen oder Regeln

Schreibweisen:

$$(A, \alpha) \in P \quad \hat{=} \quad A \xrightarrow[G]{} \alpha \quad \text{oder} \quad A \to \alpha$$

- Vereinbarung:

Großbuchstaben	für	Nichtterminale
Kleinbuchstaben Alphabetanfang	für	Terminale
Kleinbuchstaben Alphabetende	für	Worte aus Σ^*
griechische Buchstaben	für	Strings aus V^*

- $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_t \rightsquigarrow A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_t$

  $\alpha_1, \alpha_2, \ldots, \alpha_t$ <u>Alternativen</u> für A

- $\alpha, \beta \in V^*$: $\alpha \underset{G}{\Rightarrow} \beta$ (α <u>produziert</u> β <u>direkt</u>, β <u>ist aus α direkt ableitbar</u>)

  genau dann, wenn

  $\exists \gamma, \delta, \rho \in V^*, A \in N : \alpha = \gamma A \delta, \ \beta = \gamma \rho \delta$

  und $A \to \rho \in P$.

- $\overset{*}{\underset{G}{\Rightarrow}}$ reflexive, transitive Hülle von $\underset{G}{\Rightarrow}$

  $\rightsquigarrow$

  $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ (<u>α produziert β</u> oder <u>β ist aus α ableitbar</u>)

  genau dann, wenn

  $\exists\, \alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_m$ mit

  $$\alpha \underset{G}{\Rightarrow} \alpha_1 \underset{G}{\Rightarrow} \alpha_2 \underset{G}{\Rightarrow} \alpha_3 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m \underset{G}{\Rightarrow} \beta.$$

  <u>Ableitung</u> von β aus α.

  <u>Länge</u> der Ableitung $\hat{=}$ Anzahl der verwendete Produktionen,

  also $m+1$

- $L(G) = \{ w \in \Sigma^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w \}$ ist die <u>durch G generierte Sprache</u>

- $L \subseteq \Sigma^*$ heißt <u>kontextfrei</u>, falls $L = L(G)$ für eine kontextfreie Grammatik G.

- G und G' <u>äquivalent</u>, falls $L(G) = L(G')$

<u>Bsp.:</u>

$G_0 = (V, \Sigma, P, S)$, wobei

$N = \{S\}$, $\Sigma = \{a, b\}$

$P = \{ S \to aSb \mid \varepsilon \}$

Es gilt:

$L(G_0) = \{ a^n b^n \mid n \geq 0 \}$. $\quad a^0 = \varepsilon$

<u>Bezeichnungen</u>

$x \in L(G)$ <u>Satz</u> von G

$\alpha \in V^*$ mit $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ <u>Satzform</u>

$A \in N$ <u>nützlich</u>, falls $\exists\, \alpha, \beta \in V^*, w \in \Sigma^*$ mit $S \overset{*}{\underset{G}{\Rightarrow}} \alpha A \beta \overset{*}{\underset{G}{\Rightarrow}} w$.

Andernfalls <u>nutzlos</u>.

G <u>reduziert</u>, falls alle Nichtterminale in N nützlich sind.

Ziel:

Entwicklung eines Verfahrens zur Transformation einer beliebigen cfg G in eine äquivalente reduzierte cfg.

Idee

(1) Entferne ~~alle~~ jede Variablen, aus ~~denen~~ der nicht ein Terminalwort ableitbar ist.

(2) Entferne alle Symbole, die nicht in einer Satzform der Grammatik vorkommen.

Lemma 4.1

Sei $G = (V, \Sigma, P, S)$ eine cfg mit $L(G) \neq \emptyset$. Dann kann eine äquivalente cfg $G' = (V', \Sigma, P', S)$, so daß $\forall A \in V' \setminus \Sigma =: N'$ gilt: $\exists w \in \Sigma^*: A \overset{*}{\underset{G'}{\Rightarrow}} w$, konstruiert werden.

Beweis:

· iterative Berechnung von N'

(1) $N' := \{A \in N \mid A \to w, w \in \Sigma^* \in P\}$

(2) Solange $B \to X_1 X_2 \cdots X_t \in P$ mit $X_i \in \Sigma \cup N'$, $1 \leq i \leq t$ füge B zu N' hinzu.

(3) P' besteht aus allen Produktionen in P, die nur Symbole aus $N' \cup \Sigma \cup \{\varepsilon\}$ enthalten

Algorithmus PROD

Eingabe: cfg $G = (V, \Sigma, P, S)$

Ausgabe: cfg $G' = (V', \Sigma, P', S)$ mit $L(G') = L(G)$
und $\forall A \in V' \setminus \Sigma \; \exists w \in \Sigma^*$ mit $A \overset{*}{\underset{G'}{\Rightarrow}} w$

Methode:

(1) $OLDN := \emptyset$;
(2) $NEWN := \{ A \in N \mid A \to w \in P$ für ein $w \in \Sigma^* \}$;
(3) <u>while</u> $OLDN \neq NEWN$
<u>do</u>
$OLDN := NEWN$;
$NEWN := OLDN \cup \{ A \mid A \underset{G}{\to} \alpha$ für ein $\alpha \in (\Sigma \cup OLDN)^* \}$
<u>od</u>;
(4) $V' := NEWN \cup \Sigma$,
(5) $P' := \{ A \to \alpha \in P \mid A\alpha \in V'^* \}$.

Korrektheit kann leicht mittels Induktion über die Länge der Ableitung bewiesen werden.

∎

Lemma 4.2.
Sei $G = (V, \Sigma, P, S)$ eine cfg. Dann kann eine äquivalente cfg $G' = (V', \Sigma, P', S)$ konstruiert werden, so dass $\forall X \in V'$ gilt:
$\exists \alpha, \beta \in V'^* : S \overset{*}{\underset{G'}{\Rightarrow}} \alpha X \beta$

Beweis:

- iterative Berechnung von V'
- V'' enthält Symbole, für die bereits bekannt ist, dass sie zu V' gehören, die jedoch, um gegebenenfalls weitere Symbole in V' einzufügen, noch zu bearbeiten sind.

Algorithmus ERREICH

Eingabe: cfg $G = (V, \Sigma, P, S)$

Ausgabe: äquivalente cfg (V', Σ', P', S), so dass $\forall X \in V'$ $\alpha, \beta \in V^*$ mit $S \overset{*}{\underset{G'}{\Rightarrow}} \alpha X \beta$ existieren.

Methode:

(1) $V' := \emptyset$; $V'' := \{S\}$;

(2) <u>while</u> $\exists A \in V''$

<u>do</u>

Wähle solchen $A \in V''$;

$V' := V' \cup \{A\}$; $V'' := V'' \setminus \{A\}$;

<u>for</u> alle $X \in V \setminus (V' \cup V'')$ mit

X Zeichen in Alternative für A

<u>do</u>

<u>if</u> $X \in N$

<u>then</u>

$V'' := V'' \cup \{X\}$

<u>else</u>

$V' := V' \cup \{X\}$

<u>od</u> <u>fi</u>

(3) $\Sigma' := V' \cap \Sigma$;

$P' := \{ A \to \alpha \in P \mid A\alpha \in V'^* \}$.

Korrektheit kann leicht mittels Induktion über die Länge der Ableitung bewiesen werden.

$\blacksquare$

$\rightsquigarrow$

<u>Satz 4.1</u>

Für jede nichtleere kontextfreie Sprache L existiert eine reduzierte cfg $G = (V, \Sigma, P, S)$ mit $L(G) = L$.

<u>Annahme:</u>

Im folgenden ist eine gegebene cfg stets reduziert.

Sei $G = (V, \Sigma, P, S)$ eine cfg. Ein <u>Ableitungs-baum</u> in G ist ein geordneter Baum T, für den gilt:

1. Jeder Knoten in T ist mit einem Symbol aus $V \cup \{\varepsilon\}$ markiert.
2. Die Markierung der Wurzel ist S.
3. Die Markierung eines inneren Knotens v ist immer aus N.

4. Hat ein Knoten v die Markierung A und sind $v_1, v_2, \ldots, v_k$ die direkten Nachfolger von v in dieser Reihenfolge mit den Markierungen $X_1, X_2, \ldots, X_k$, dann gilt $A \to X_1 X_2 \ldots X_k \in P$.

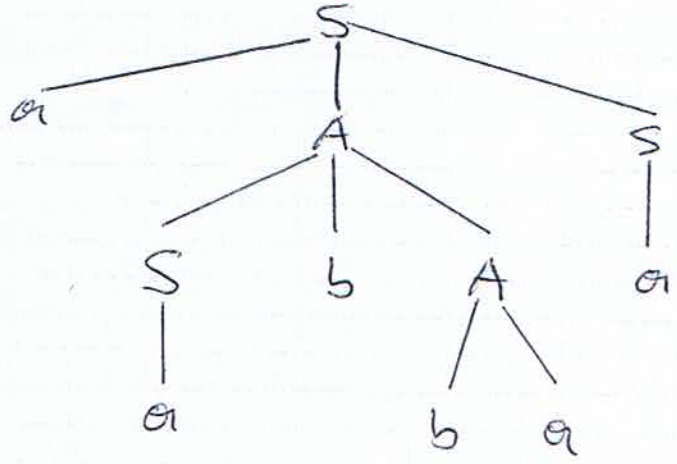
5. Hat ein Knoten v die Markierung ε, dann ist v ein Blatt und v ist der einzige Sohn seines Vaters.

<u>Bsp.</u>

$G = (V, \Sigma, P, S)$ mit

$N = \{S, A\}$, $\Sigma = \{a, b\}$ und

$P = \{ S \to aAS \mid a \,,\; A \to SbA \mid SS \mid ba \}$



Ableitungsbaum in G.

Bezeichnungen

A-Baum Unterbaum mit Wurzelmarkierung A

Resultat eines A-Baumes Markierungen der Blätter von links nach rechts.

Satz 4.2

Sei $G = (V, \Sigma, P, S)$ eine cfg. Dann gilt $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ genau dann, wenn es einen Ableitungsbaum mit Resultat α gibt.

Beweis:

Beh.:

$\forall A \in N, \alpha \in V^*$ gilt genau dann $A \overset{*}{\underset{G}{\Rightarrow}} \alpha$, wenn es einen A-Baum T in G mit Resultat α gibt.

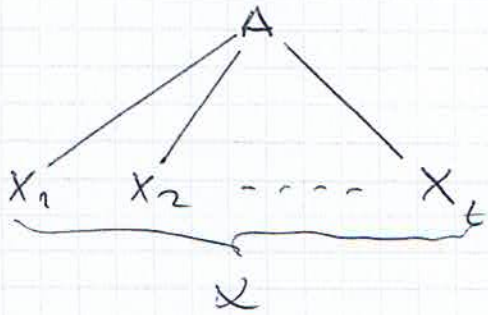
„$\Leftarrow$"

Sei α Resultat eines A-Baumes T.

Beweis $A \overset{*}{\underset{G}{\Rightarrow}} \alpha$ durch Induktion über die Anzahl der inneren Knoten des A-Baumes T.

Anfang: 1 innerer Knoten

Struktur



Def. A-Baum $\Rightarrow$ $A \to \alpha \in P$

$\Rightarrow$ $A \underset{G}{\Rightarrow} \alpha$.

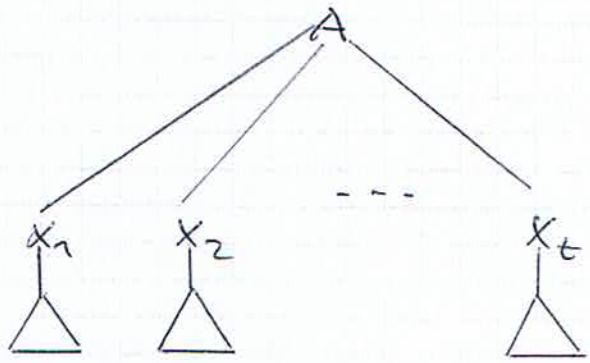
Sei $k \geq 1$.

<u>Annahme:</u>

Beh. ist wahr $\forall$ X-Bäume, $X \in N$ mit $\leq k$ inneren Knoten.

Sei α Resultat eines A-Baumes mit $(k+1)$ inneren Knoten.

Struktur:



Def. $\Rightarrow$

$$A \to X_1 X_2 \cdots X_t \in P$$

<u>Beobachtung:</u>

i-te Sohn von A kein Blatt $\Rightarrow$

$X_i \in N$ ist Wurzel eines X_i-Baumes mit $\leq k$ inneren Knoten.

Sei

$$\alpha_i = \begin{cases} X_i & \text{falls } X_i \text{ Blatt} \\ \text{Resultat des } X_i\text{-Baumes} & \text{sonst.} \end{cases}$$

$\Rightarrow$

$$\alpha = \alpha_1 \alpha_2 \dots \alpha_t .$$

ind. Voraussetzung $\Rightarrow$

$$X_i \overset{*}{\underset{G}{\Rightarrow}} \alpha_i \qquad \text{falls } X_i \text{ kein Blatt}$$

$\Rightarrow$

$$A \underset{G}{\Rightarrow} X_1 X_2 \dots X_t \overset{*}{\underset{G}{\Rightarrow}} \alpha_1 X_2 X_3 \dots X_t \overset{*}{\underset{G}{\Rightarrow}} \dots \overset{*}{\underset{G}{\Rightarrow}} \alpha_1 \alpha_2 \dots \alpha_t$$

„$\Leftarrow$" analog ∎

Sei $G = (V, \Sigma, P, S)$ eine cfg. $x \in L(G)$ heißt <u>mehrdeutig</u>, falls es mehr als einen Ableitungsbaum für x in G existiert. G heißt <u>mehrdeutig</u>, falls ein mehrdeutiges $x \in L(G)$ existiert.

<u>Bezeichnungen:</u>

Sei $S = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \dots \Rightarrow \alpha_t = \alpha$ eine Ableitung von α aus S. Falls in jeder Satzform der Ableitung immer die am weitesten links stehende Variable durch eine ihrer Alternativen ersetzt wird, dann heißt $\alpha_1, \alpha_2, \dots, \alpha_t$ <u>Linksableitung</u> von α aus S.

D.h., für $1 \leq i \leq t-1$ gilt:

$\alpha_i = w_i A_i \beta_i$, $\alpha_{i+1} = w_i \delta_i \beta_i$, wobei

$w_i \in \Sigma^*$, $A_i \to \delta_i \in P$.

<u>Schreibweise:</u> $S \overset{*}{\underset{lm}{\Rightarrow}} x$ ("lm" für "leftmost").

<u>Bsp:</u>

$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS$
$\Rightarrow aabbaa$

zum obigen Ableitungsbaum korrespondierende Linksableitung

<u>Linkssatzform</u> Satzform in Linksableitung

analog:

<u>Rechtsableitung</u>, <u>Rechtssatzform</u>, $S \overset{*}{\underset{rm}{\Rightarrow}} x$

4.2. <u>Kellerautomaten</u>

- cfg G generiert $x \in L(G)$
- Benötigt wird Akzeptor für $L(G)$.

Klasse der kontextfreie Sprachen bildet eine echte Obermenge der Klasse der regulären Mengen.

<u>Ziel:</u>

Erweiterung der endlichen Automaten, so dass die erweiterte Klasse von Automaten exakt die Klasse der kontextfreien Sprachen akzeptiert.

Betrachte

$$L = \{ ww^R \mid w \in \{a,b\}^* \}, \text{ wobei}$$
$$w^R = a_n a_{n-1} \dots a_1 \text{ für } w = a_1 a_2 \dots a_n.$$

L ist nicht regulär (überzeugen Sie sich mit Hilfe von Satz 4.4.)

L ist kontextfrei, da $L = L(G)$ für cfg $G = (V, \Sigma, P, S)$ mit $N = \{S\}$, $\Sigma = \{a,b\}$ und $P = \{ S \to aSa \mid bSb \mid \varepsilon \}$.

<u>Beobachtung:</u>

- Jeder Automat, der Strings aus L bei einmaligem Lesen der Eingabe akzeptiert, muß sich an die erste Hälfte der Eingabe "erinnern".

$\rightsquigarrow$

Benötigt Fähigkeit, Information beliebiger Länge zu speichern.

- Für obiges Beispiel würde hierfür ein Keller genügen.

erklären.

und auch erklären, warum der Automat nichtdeterministisch ist.

Ein <u>nichtdeterministischer Kellerautomat</u> M ist ein 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, wobei