

Notizen zu Algorithmen II

Jens Ochsenmeier

16. Februar 2018

Inhaltsverzeichnis

1 Randomisierte Algorithmen	• 5
2 Approximationsalgorithmen	• 9
3 Stringology	• 15
3.1 Strings sortieren	• 15
3.2 Pattern Matching	• 16
3.3 Datenkompression	• 23
4 Range Minimum Queries	• 25
4.1 Lösung 1 – $\langle O(n), O(\log n) \rangle$	• 26
4.2 Lösung 2 – $\langle O(n \log n), O(1) \rangle$	• 26
4.3 Lösung 3 – $\langle O(n \log \log n), O(1) \rangle$	• 27
4.4 Lösung 4 – $\langle O(n), O(1) \rangle$	• 27
4.5 Lowest Common Ancestor	• 28
5 Burrows-Wheeler-Transformation	• 33
5.1 Konstruktion	• 33
5.2 Beobachtungen	• 34
5.3 Rücktransformation	• 34
5.4 Was bringt die BWT?	• 37
5.5 Kompression	• 37
5.6 Suche in der Burrows-Wheeler-Transformation	• 38
5.7 Wavelet Trees	• 39
5.8 Exkurs — Succinct Data Structures	• 40

Inhaltsverzeichnis

6 Geometrische Algorithmen	• 45
6.1 Grundlegende Definitionen	• 45
6.2 Streckenschnitte	• 46
6.3 Konvexe Hülle	• 48
6.4 Kleinste einschließende Kugel	• 49
6.5 Range Search	• 49
7 Online-Algorithmen	• 51
7.1 Übersicht	• 51
7.2 Job-Scheduling	• 54
7.3 Skiausleihe	• 55
7.4 Speicherverwaltung	• 55
7.5 Auswahl von Experten	• 58
8 Parallel Algorithmen	• 61
8.1 Einleitung	• 61
8.2 Nachrichtenengkoppelte Parallelrechner	• 62
9 Fortgeschrittene Datenstrukturen	• 69
9.1 Adressierbare Prioritätslisten	• 69
9.2 Pairing Heaps	• 71
9.3 Fibonacci-Heaps	• 73
10 Kürzeste Wege	• 75
10.1 Allgemeine Definitionen	• 75
10.2 Dijkstras Algorithmus	• 76
10.3 Monotone ganzzahlige Prioritätslisten	• 76
10.4 All-Pairs Shortest Paths	• 78
10.5 Distanz zu Zielknoten	• 80
11 Anwendungen von DFS	• 83
11.1 Starke Zusammenhangskomponenten	• 83

1

Randomisierte Algorithmen

Neue Grundlagen

- **Zufallszahlen:** `randInt($c : \mathbb{N}$)` liefere zufälliges $n \in \{0, 1, \dots, c - 1\}$
- **Nichtdeterminismus:** Mehrere Ausführungen des *gleichen* randomisierten Algorithmus für *gleiche* Eingabe evtl. verschieden!
- **Zufallsvariablen:** Für feste Eingabe sind Laufzeit, Speicherplatzbedarf, Ergebnis Zufallsvariablen

Recap Wahrscheinlichkeitstheorie

- **Elementarereignisse:** Menge Ω
- **σ -Algebra:** Menge von Ereignissen $E \subseteq 2^\Omega$ (Potenzmenge von Ω)
 - *diskret*, wenn $E = 2^\Omega$ und $|\Omega| \leq |\mathbb{N}|$
- **Wahrscheinlichkeitsmaß:** $\Pr : E \rightarrow [0, 1]$ (σ -additiv, $\Pr(\Omega) = 1$)
- **Zufallsvariable:** $X : \Omega \rightarrow \mathbb{R}$ mit Eigenschaften ...
 - Eigenschaften erfüllt, falls σ -Algebra diskret (hier immer der Fall)
- **Schreibweisen:**
 - $\Pr(X \leq x)$ statt $\Pr(\{\omega \mid X(\omega) \leq x\})$
 - $\Pr(X = x)$ statt $\Pr(\{\omega \mid X(\omega) = x\})$
- **Beispiel:** Würfeln
 - $\Omega = \{1, 2, 3, 4, 5, 6\}$
 - $E = 2^\Omega$ diskret, z.B. $E \ni g = \{2, 4, 6\}$ (gerade Zahl gewürfelt)

- fairer Würfel: $\forall w \in \Omega : \Pr(\omega) = \frac{1}{6}$
- Beispiel: $X(\omega) = \begin{cases} 0, & \text{falls } \omega \text{ Produkt zweier Primfaktoren} \\ 1, & \text{sonst} \end{cases}$
- z.B. $\Pr(X = 1) = \Pr(\{1, 2, 3, 5\}) = \frac{2}{3}$
- **Erwartungswert** von X: $E(X) = \sum_{x \in \mathbb{R}} x \Pr(X = x)$
 - $E(X + Y) = E(X) + E(Y)$
 - *unabhängig*, falls $\forall x, y \in \mathbb{R} : \Pr(X = y \wedge Y = y) = \Pr(X = x) \cdot \Pr(Y = y)$
 - falls X, Y unabhängig: $E(X \cdot Y) = E(X) \cdot E(Y)$
- **Indikator-Zufallsvariable**: 0 und 1 einzige mögliche Funktionswerte

Ausführung randomisierter Algorithmen

- randomisierter Algorithmus R , Eingabe x
- $\Omega = \{\text{"Programmlauf"} \text{ von } R \text{ für Eingabe } x\}$
 - *Programmlauf*: Folge der durchlaufenen globalen Speicherzustände
- $E = 2^\Omega$
- Mögliche Zufallsvariablen:
 - $X(\text{Prog.lauf})$ = Anzahl Schritte
 - $X(\text{Prog.lauf})$ = Ausgabe
- **Unbekannte Laufzeit**: Gewollt? Mögliche Quantifizierung?
- **Variierende Ausgaben**: Oft gewollt (Erzeugung zufälliger Objekte, Optimierung)
- **Fehlerhafte Ausgaben**: Fehlerwahrscheinlichkeit?
- *Quantifizierung*
- **Vorteile**: manchmal
 - leichter zu formulieren/implementieren
 - "schneller", "besser"
 - einzige Möglichkeit

Markov-/Chebyshev-Ungleichung

- **Markov**: ZV $Y \geq 0$, Erwartungswert μ_Y . Dann gilt $\forall t, k \in \mathbb{R}_+$:

$$\Pr(Y \geq t) \leq \frac{\mu_Y}{t} \text{ bzw } \Pr(Y \geq k\mu_Y) \leq \frac{1}{k}$$

- **Chebyshev**: ZV X mit EW μ_X , Standardabweichung σ_X . Dann gilt $\forall t \in \mathbb{R}_+$:

$$\Pr(|X - \mu_X| \geq t\sigma_X) \leq \frac{1}{t^2} \text{ bzw } \Pr(|X - \mu_X| \geq t) \leq \frac{\sigma_X^2}{t^2}$$

Chernoff-Schranken

- X_1, \dots, X_n unabhängige Indikator-ZV mit $\Pr(X_i = 1) = p_i$
- $X = \sum_{i=1}^n X_i$ und $\mu = \mathbf{E}(X) = \sum p_i$
- Dann gilt für alle $0 \leq \delta$:

$$\Pr(X \geq (1 + \delta)\mu) \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}} \right)^\mu$$

- Dann gilt für alle $1 > \delta \geq 0$, also $0 < 1 - \delta \leq 1$:

$$\Pr(X \leq (1 - \delta)\mu) \leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}} \right)^\mu$$

- **Vereinfachung:** Betrachte $f(\delta) = \delta - (1 \pm \delta) \ln(1 \pm \delta)$ statt $\left(\frac{e^\delta}{(1 \pm \delta)^{1 \pm \delta}} \right)^\mu$
 - je nach Bereich, aus dem δ stammt, kann man $f(\delta)$ nach oben durch $-\frac{\delta^2}{c}$ abschätzen

- **Korollar:** Für $1 > \delta \geq 0$ gilt:

$$\Pr(X \leq (1 - \delta)\mu) \leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}} \right)^\mu \leq e^{-\delta^2 \frac{\mu}{2}}$$

Und-Oder-Bäume

- **Aufbau:** T_k vollständiger binärer Baum, Höhe $2k$
 - *innere Knoten*: abwechselnd mit \wedge und \vee markiert
 - *Wurzel* von T_1 : \wedge -Knoten mit zwei nachfolgenden \vee -Knoten
 - $T_{k-1} \rightarrow T_k$: Blätter werden durch Kopien von T_{k-1} ersetzt
 - *Knoten*: T_K hat $n = 4^k$ Blätter (im Folgenden x_1, \dots, x_{4^k})
- **Auswertung:**
 - *gegeben*: Werte x_1, \dots, x_n an den Blättern
 - *gesucht*: Wert $T_k(x_1, \dots, x_n)$ an der Wurzel
 - Berechnung: bottom-up durch Besuch aller Blätter + Berechnung der inneren Knoten
 - *Frage*: geht es besser?
- **Problem:** für jeden *deterministischen* Algorithmus A und jedes $k \geq 1$ gibt es eine Bitfolge x_1, \dots, x_n , sodass A zur Berechnung von T_k alle 4^k Blätter besucht
- **Aber:** jede Liste x_1, \dots, x_{4^k} enthält Teilfolge x_1, \dots, y_{2^k} , die schon Wurzelwert festlegt!
 - diese y_i sind schwer zu finden \sim **Randomisierung**?

- **Satz:** Für jede randomisierte UOB-Auswertung gilt: Für jede Folge x_1, \dots, x_{4^k} ist Erwartungswert für die Anzahl besuchter Blätter $\leq 3^k = n^{\log_4 3} \approx n^{0,792\dots}$. Das ist beweisbar optimal.

Erdős-Rényi-Zufallsgraphen

- **Initialisierung:** $G = (\{1, \dots, n\}, \emptyset)$

- **Aufbauen:**

```

for i ← 1 to n - 1 do
    for j ← i + 1 to n do
        add {i, j} to E with probability p = p(n)
return (V, E)

```

- **Eigenschaften:**

- Wahrscheinlichkeit für bestimmten Graphen mit n Knoten und m Kanten:

$$p^m(1-p)^{\binom{n}{2}-m}$$

- Erwartete Kantenzahl: $p\left(\frac{n}{2}\right)$
- erwarteter Knotengrad: $p(n-1)$
- Wahrscheinlichkeit für Knotengrad d : $\binom{n-1}{d} p^d (1-p)^{n-1-d}$

2

Approximationsalgorithmen

Suchprobleme

- Für Probleminstanz $I \in M_I$ gibt es Menge $M_S(I)$ möglicher Lösungen $S \in M_S(I)$
- **Zielfunktion** $f : \bigcup_{I \in M_I} M_S(I) \rightarrow \mathbb{R}_+$
- **Minimierungsproblem:** $\forall I \in M_I \exists f^*(I) := \min\{f(S) : S \in M_S(I)\}$
 - gegeben: I
 - gesucht: S mit $f(S) = f^*(I)$
- **Maximierungsproblem:** analog

Suchprobleme — Approximation

- Lösungen S mit $f(S) = f^*(I)$ oft schwer zu finden (z.B. **NP-schwer**)
- **Auswege:**
 - *naiv*: alle möglichen Lösungen betrachten \rightsquigarrow oft zu aufwändig
 - *ad-hoc-Heuristiken*: Qualität der Antwort eventuell unklar
 - *Approximationsalgorithmus A*:
 - $f(A(I))$ “möglichst nah an” $f^*(I)$
 - **Optimierungsaufgabe**
 - möglichst polynomiale Laufzeit
 - Garantie für Lösungsgüte

Job Scheduling

- **Problem:** m Maschinen sollen n Jobs abarbeiten, möglichst alle gleichzeitig fertig
 - Maschinen M_1, \dots, M_m
 - Jobs J_1, \dots, J_n
 - Lösung $S : \{1, \dots, n\} \rightarrow \{1, \dots, m\}$
 - Last von Maschine i : $L_i = \sum_{S(j)=i} t_j$
 - Zielfunktion: minimiere Makespan $L_{\max} = \max_i L_i$ (Wann ist letzte Maschine fertig?)
 - einfacher Fall: identische Maschinen, unabhängige Jobs
 - Problem ist **NP-hart**
- **Approximationsfaktor:** Approximationsalgorithmus A für Minimierungsproblem mit Zielfunktion f erzielt *Approximationsfaktor* ρ , falls

$$\forall I \in M_I : \frac{f(A(I))}{f(I)} \leq \rho$$
 - einfacher Fall: $\rho = 1 \rightsquigarrow A$ liefert stets optimale Lösung

Turing-Reduzierbarkeit

- **Definition:** Suchproblem Π **NP-schwer** oder **NP-hart**, falls ein **NP-vollständiges** Entscheidungsproblem L existiert mit $L \leq_T^p \Pi$, d.h.
 - *Orakel-Turingmaschine* für L mit Orakel für Π (eine Befragung = 1 Schritt) in polynomieller Laufzeit
 - ~ Turing-Reduktion in Polynomialzeit
 - ~ Wenn Π polynomiell lösbar ist, dann auch L

Allgemeines Travelling Salesman-Problem

- **Probleminstanz:** Graph $G = (V, E := V \times V)$, Längenfunktion $c : E \rightarrow \mathbb{Z}_+$
- **Zielfunktion** für Permutation π von V :

$$f(\pi) = \sum_{i=1}^{n-1} c(\pi(i), \pi(i+1)) + c(\pi(n), \pi(1))$$

$$\rightsquigarrow f^*(G, c) = \min_{\pi} f(\pi)$$
- **Gesucht:** Permutation π mit minimalem $f(\pi) = f^*(G, c)$

Allgemeines TSP — Approximation

- **Gegeben:**
 - $a \geq 1$
 - *Probleminstanz* (wie oben)
 - *Zielfunktion* (wie oben)
- **Gesucht:** Permutation π mit $f(\pi) \leq a \cdot f^*(G, c)$
- **Satz:** Für jedes $a \geq 1$ ist TSP- a -Approximations-Suchproblem **NP**-hart.

Entscheidungsproblem — Hamiltonkreis

- **Probleminstanz:** Graph $G = (V, E)$
- **Frage:** Gibt es Hamilton-Kreis in G ?
 - ≈ Permutation π derart, dass $\pi(1), \dots, \pi(n), \pi(1)$ Kreis
- Problem ist **NP**-vollständig

Metrisches TSP

- **Definition:** Wie TSP, aber *Dreiecksungleichung* wird von c verlangt:

$$\forall x, y, z \in V : c(x, y) + c(y, z) \geq c(x, z)$$

- **Satz:** Für Instanzen des Problems kann man in Polynomialzeit eine 2-Approximation berechnen

Pseudopolynomielle Laufzeit

- **Laufzeitabhängigkeit:** Laufzeit $t(I)$ für Eingabe I abhängig von Größe $n(I)$ der Repräsentation von I
- **Binäre Codierung:** Codierung von $k \in \mathbb{N}$ braucht $n(I) = n_2(I) = \Theta(\log_2 k)$ Bits
- **Unäre Codierung:** Codierung von $k \in \mathbb{N}$ braucht $n(I) = n_1(I) = k$ Bits
- **Polynomielle Laufzeit** $t(n)$, wenn

$$\exists \text{ Polynom } p(n) \forall I : t(I) \leq p(n_2(I))$$

- **Pseudopolynomielle Laufzeit** $t(n)$, wenn

$$\exists \text{ Polynom } p(n) \forall I : t(I) \leq p(n_1(I))$$

- **Achtung!**

2 Approximationsalgorithmen

- *Pseudopolynomielle Laufzeit*: Tatsächliche Form der Eingabe?
- Nicht verwechseln mit *quasipolynomieller Laufzeit* $t(n) = 2^{O((\log n)^c)}$ (Konstante $c > 0$)

Knapsack

- **Probleminstanz:**

- Gegenstände $M = \{1, \dots, n\}$
- Maximalgröße $W \in \mathbb{N}$ (Rucksackgröße)
- Größen $w_i \in \mathbb{N}$ (oBdA jedes $w_i \leq W$)
- Profite $p_i \in \mathbb{N}$
- ~> Gegenstand i hat Größe w_i und Profit p_i

- **Lösungen:** Teilmenge $M' \subseteq M$ mit

$$w(M') = \sum_{i \in M'} w_i \leq W$$

- **Gesucht:**

- Teilmengen mit möglichst großem Profit
- Zielfunktion $f(M') = p(M') = \sum_{i \in M'} p_i$
- Maximierungsproblem $f^*(I)$

- **Codierung** ($\hat{P} := \sum_i p_i$)

Bestandteil	$u_2(T)$	$u_1(T)$
$\{1, \dots, n\}$	$\log n$	n
W	$\log W$	W
$\langle w_1, \dots, w_n \rangle$	$n \log W$	nW
$\langle p_1, \dots, p_n \rangle$	$n \log \hat{P}$	$n\hat{P}$
insgesamt	$\Theta(n \log W + n \log \hat{P})$	$\Theta(nW + n\hat{P})$

- **Schwere:** NP-schwer

- bei “normaler” Messung der Eingabegröße
- aber: pseudopolynomielle Laufzeit erreichbar
- ~> schwach NP-schwer

- **Pseudopolynomielle Laufzeit** durch *dynamische Programmierung*:

$$C(i, P) := \min\{w(M') : M \subseteq \{1, \dots, i\} \wedge p(M') \geq P\}$$

falls ein solches M' existiert, ∞ sonst.

$$C(1, P) = \begin{cases} w_1, & \text{falls } p_1 \geq P \\ \infty, & \text{sonst} \end{cases}$$

$$C(i+1, P) = \min\{C(i, P), w_{i+1} + C(i, P - p_{i+1})\}$$

```
DYNPROGKNAPSACK(n, W, {w1, ..., wn}, {p1, ..., pn})
for P ← 1 to P do
    C(1, P) ← ...
    for i ← 1 to n - 1 do
        for P ← 1 to P do
            C(i + 1, P) ← min{C(i, P), wi+1 + C(i, P - pi+1)}
return max{P : C(n, P) ≤ W}
```

- Erweiterung: in $C(i, P)$ speichern, welche Objekte der $1, \dots, i$ benutzt werden
- ↪ Skalarprodukt Objekt-Bitvektor und Profit ist maximaler Profit

(Voll) Polynomielle Approximationsschemata

- Vorgaben:

- $\left\{ \begin{array}{l} \text{Minimierungs} \\ \text{Maximierungs} \end{array} \right\}$ -Problem $\Pi = \{D, S, f\}$
 - Eingabemenge D
 - $S \ni S_I$ Menge der für Eingabe $I \in D$ gültigen Lösungen
 - Bewertungsfunktion $f : S_I \rightarrow \mathbb{N}$
- Algorithmus $\mathcal{A}(I, \varepsilon)$ mit $\varepsilon \in \mathbb{R}_+, I \in \Pi_D$

- PTAS \mathcal{A} (pol. Approx.-Schema), falls $\forall \varepsilon > 0 \exists$ Polynom $p(n) : \forall I \in \Pi_D :$

1. $f(\mathcal{A}(I, \varepsilon)) \leq \left(\frac{1+\varepsilon}{1-\varepsilon}\right) f^*(I)$
2. Laufzeit $t(I, \varepsilon) \leq p(n_2(I))$

- FPTAS \mathcal{A} (voll pol. Approx.-Schema), falls \exists Polynom $p(n, x) : \forall I \in \Pi_D, \varepsilon > 0 :$

1. $f(\mathcal{A}(I, \varepsilon)) \leq \left(\frac{1+\varepsilon}{1-\varepsilon}\right) f^*(I)$
2. Laufzeit $t(I, \varepsilon) \leq p(n_2(I), \frac{1}{\varepsilon})$

- Jedes FPTAS ist PTAS, aber nicht umgekehrt

Knapsack — FPTAS

- **Implementierung:**

```
EPSAPPROXKNAPSACK( $\varepsilon, n, W, w, p$ )
   $P \leftarrow \max_i p_i$ 
   $K \leftarrow \varepsilon \frac{P}{n}$ 
  each  $p'_i \leftarrow \lfloor \frac{p_i}{K} \rfloor$ 
   $x' \leftarrow \text{DYNPROGKNAPSACK}(n, W, w, p')$ 
  return  $x'$ 
```

- **Analyse:**

- x^* optimale Lösung für ursprüngliches p , x' optimale Lösung für p'
- $px^* = \sum_{i:x_i=1} p_i = \max.$ Profit des Originalproblems
- *Frage:* Wie gut ist px' im Vergleich zu px^* ?
 - $px' \geq (1 - \varepsilon)px^*$
- **Laufzeit:** $O(n^3 \frac{1}{\varepsilon})$
 - dyn. Programmierung für p' Problem dominiert → Laufzeit $n\hat{P}'$
 - $n\hat{P}' = n \sum i : x'_i = 1 p'_i \leq n^2 \max_i p'_i = n^2 \left\lfloor \frac{\hat{P}}{K} \right\rfloor = n^2 \left\lfloor \frac{\hat{P}n}{\varepsilon P} \right\rfloor \leq n^3 \frac{1}{\varepsilon}$

3

Stringology

Inhalt dieses Kapitels:

- Strings sortieren
- Patterns suchen
- Datenkompression

3.1 Strings sortieren

Naive Sortierverfahren, wie sie aus der Vorlesung “Algorithmen 1” bekannt sind, sind beim Sortieren von Strings ineffizient, deswegen gibt es für das Sortieren von Strings andere Algorithmen. Ein solcher ist der **Multikey Quicksort**-Algorithmus:

```
MKQSORT (S: String Seq, l:N): String Seq
assert ∀e,e' ∈ S : e[1...l - 1] = e'[1...l - 1]
if |S| ≤ 1 then return S
pick p ∈ S randomly
return concatenation of
    MKQSORT ((e ∈ S : e[l] < p[l]),l),
    MKQSORT ((e ∈ S : e[l] = p[l]),l + 1),
    MKQSORT ((e ∈ S : e[l] > p[l]),l)
```

Abbildung 3.1. Pseudocode-Implementierung des Multikey-Quicksort-Algorithmus.

3 Stringology

Dieser Algorithmus sortiert eine String-Sequenz und nimmt an, dass die ersten $l - 1$ Buchstaben bereits sortiert wurden.

Zuerst wird ein zufälliges Pivotelement gewählt. Danach wird die übergebene Sequenz an Strings wird in drei Teilsequenzen geteilt:

1. Sequenz an Strings, deren l -ter Buchstabe kleiner ist als der l -te Buchstabe des Pivotelements.
2. Sequenz an Strings, deren l -ter Buchstabe derselbe ist wie der l -te Buchstabe des Pivotelements.
3. Sequenz an Strings, deren l -ter Buchstabe größer ist als der l -te Buchstabe des Pivotelements.

Auf die erste und dritte Teilsequenz wird der Algorithmus nun rekursiv mit dem selben Parameter l ausgeführt, da die Buchstaben an der l -ten Position nicht übereinstimmen (müssen) — auf die zweite Teilsequenz wird der Algorithmus rekursiv mit dem Parameter $l + 1$ ausgeführt, weil hier die l -ten Buchstaben aller Wörter in der Sequenz gleich sind.

Die Laufzeit des Algorithmus ist in $O(|S| \log |S| + d)$, wobei d die Summe der eindeutigen Präfixe der Strings in S ist.

3.2 Pattern Matching

Hinweis: In diesem Abschnitt sind Arrays 1-basiert.

In diesem Abschnitt wird es darum gehen, alle oder zumindest ein Vorkommen eines **Patterns** $P = p_1 \dots p_m$ in einem gegebenen **Text** $T = t_1 \dots t_n$ zu finden. Im Allgemeinen ist $n \gg m$, also der Text wesentlich länger als das Pattern, das wir in ihm suchen.

Naives Pattern Matching

Das naive Vorgehen ist, an jeder Position von T zu schauen, ob an dieser das gesuchte Pattern vorkommt. Offensichtlich ist dieser Algorithmus in $O(nm)$, da im schlimmsten Fall für jede Position des Textes das gesamte Pattern durchlaufen werden muss. Dieser Algorithmus kann folgendermaßen implementiert werden:

```

NAIVEPATTERNMATCH ( $P, T$ )
 $i, j := 1$ 
while  $i \leq n - m + 1$ 
  while  $j \leq m \wedge t_{i+j-1} = p_j$  do  $j++$ 
  if  $j > m$  then return “ $P$  occurs at pos  $i$  in  $T$ ”
   $i++$ 
 $j := 1$ 

```

Abbildung 3.2. Pseudocode-Implementierung des naiven Pattern-Matching-Algorithmus.

Knuth-Morris-Pratt

Ein anderer Algorithmus zum Finden von Patterns in einem gegebenen Text ist der **Knuth-Morris-Pratt-Algorithmus**. Dieser hat sogar optimale Laufzeit, nämlich $O(n + m)$.

Idee dieses Algorithmus ist es, das Pattern eleganter nach vorne zu verschieben, wenn es einen Mismatch zwischen Text und Pattern gibt. Hierfür brauchen wir ein Hilfswerkzeug:

Für einen String S mit Länge k sei $\alpha(S)$ die Länge des längsten Präfixes von $S_{1\dots k-1}$, das auch Suffix von $S_{2\dots k}$ ist.¹

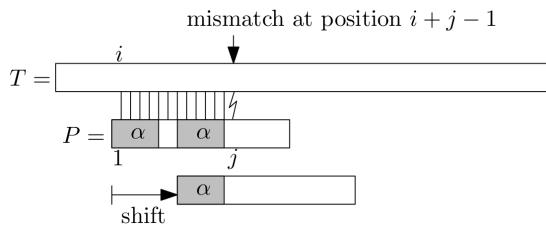


Abbildung 3.3. Idee beim Verschieben des Patterns: α wurde bereits gematcht. Früher als mit dem bereits gematchten Suffix kann das nächste Vorkommen von P nicht auftauchen, also kann man P direkt um $j - 1 - \alpha$ verschieben.

Der Algorithmus besteht aus zwei Teilen:

1. **Border-Array berechnen** ($O(m)$). Damit die oben erläuterten Verschiebungen nachher effizient durchgeführt werden können, berechnen wir für das leere Wort

¹ Wir lassen absichtlich bei Betrachtung des Präfixes den letzten und bei Betrachtung des Suffixes den ersten Buchstaben weg, damit $\alpha(S) = 0$ ist, wenn $|S| = k = 1$ ist.

und jeden Buchstaben in P einen α -Wert. Diese Werte ergeben das **Border-Array**:

$$\text{border}[j] = \begin{cases} -1, & \text{falls } j = 1 \\ \alpha(P_{1\dots j-1}), & \text{sonst} \end{cases}.$$

P	a	n	a	n	a	s
border	-1	0	0	1	2	3

Abbildung 3.4. Beispiel für das Border-Array eines Patterns.

2. **Pattern matchen** ($O(n)$). Nun verwenden wir das erstellte Border-Array, um Vorkommnisse von P in T zu finden. Wir starten sowohl im Text als auch im Pattern an Position 1 und fangen an zu matchen. Kommt es an Position $1 \leq j \leq m$ des Patterns zu einem Mismatch, so können wir P direkt um $j - \text{border}[j] - 1$ verschieben. In Pseudocode sieht das so aus:

```

KMPMatch (P,T)
i,j := 1
while i ≤ n − m + 1
    while j ≤ m ∧ ti+j-1 = pj do j++
    if j > m then return “P occurs at pos i in T”
    i += j − border[j] + 1
    j := max{1, border[j] + 1}

```

Eine Ausführung des Algorithmus kann also folgendermaßen aussehen:

$$\begin{array}{l} \text{Beispiel: } \\ \begin{array}{r} 12345 \\ P = \underline{\underline{\overline{ahaq}}} \\ -10011 \end{array} \quad \begin{array}{c} \overline{T} = ahahaahahahaq \\ \begin{array}{ccccccccc} & 1 & 4 & an & | & 1 & ah & aq \\ & \downarrow \\ ah & ah \end{array} \\ j=5 \end{array} \end{array}$$

Abbildung 3.5. Beispiel für das Verwenden des Knuth-Morris-Pratt-Algorithmus.

Suffix-Arrays

Im Folgenden werden Arrays wieder mit Position 0 beginnen. Wir verwenden des Weiteren folgende Festlegungen:

- Ein **String** ist ein Array von Buchstaben,

$$S[0 \dots n) \coloneqq S[0 \dots n - 1] \coloneqq [S[0], \dots, S[n - 1]].$$

- Das **Suffix** S_i sei der Substring $S[i \dots n]$ von S .

- Wir setzen an das Ende jedes Strings ausreichend viele **Endmarkierungen**: $S[n] := S[n + 1] := \dots := 0$. 0 sei per Definition kleiner als alle anderen vorkommenden Zeichen.

Das **Suffix-Array** eines Strings lässt sich nun folgendermaßen konstruieren:

1. Bilde die Menge aller Suffixe S_i ($i = 0, \dots, n - 1$) des Strings.
2. Sortiere die Menge aller Suffixe des Strings (z.B. mit **Multikey Quicksort**).

0	banana	5	a
1	anana	3	ana
2	nana	1	anana
3	ana	0	banana
4	na	4	na
5	a	2	nana

Abbildung 3.6. Beispiel für die Konstruktion des Suffix-Arrays des Strings "banana".

Mithilfe dieses Suffix-Arrays lassen sich später viele Suchprobleme in Linearzeit lösen. Beispielsweise ist die Suche nach dem längsten Substring, der (eventuell mit Überschneidung) zweimal im Text vorkommt, linear — dafür muss nach Berechnung des Suffix-Arrays der längste String gefunden werden, der Präfix von zwei Strings im Suffix-Array ist (im Beispiel oben wäre das "ana").

Berechnung des Suffix-Arrays in Linearzeit

Das Suffix-Array eines Strings lässt sich in Linearzeit berechnen.² Hier soll lediglich das Prinzip erläutert werden, genauere Angaben gibt es im Paper.

Wir betrachten den String

$$T[0, n) = \underset{0}{x} \underset{1}{a} \underset{2}{b} \underset{3}{b} \underset{4}{a} \underset{5}{d} \underset{6}{a} \underset{7}{b} \underset{8}{b} \underset{9}{a} \underset{10}{d} \underset{11}{o}.$$

Unser Ziel ist das Suffix-Array

$$SA = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0).$$

Wir gehen wie folgt vor:

0. **Suffixe wählen.** Sei

$$B_k = \{i \in [0, n] : i \mod 3 = k\}$$

und $C = B_1 \cup B_2$ sowie S_C die Menge der entsprechenden Suffixe. C ist also die Menge aller Positionen in T , an denen Suffixe mit einer nicht durch 3 teilbaren Länge beginnen. Hier ist $C = \{1, 4, 7, 10, 2, 5, 8, 11\}$.

² Kärkkäinen, Sanders, Burkhardt: Linear Work Suffix Array Construction

- 1. Gewählte Suffixe sortieren.** Wir fügen am Ende von T beliebig viele \emptyset hinzu und bilden zuerst für $k = 1, 2$ die Strings

$$R_k = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] \cdots [t_{\max B_k} t_{\max B_k + 1} t_{\max B_k + 2}].$$

Der Charaktere von R_k sind also Tripel. Das letzte Tripel ist immer eindeutig, weil $t_{\max B_k + 2} = 0$. Sei $R = R_1 \odot R_2$.

Hier ist

$$R = [\text{abb}][\text{ada}][\text{bba}][\text{do}\emptyset][\text{bba}][\text{dab}][\text{bad}][\text{o}\emptyset\emptyset].$$

Die Ordnung der Suffixe von R stimmt mit der Ordnung der Suffixe S_i überein, deswegen genügt es, die Suffixe von R zu sortieren.

Wir sortieren R nun, indem wir die einzelnen Charaktere von R sortieren und durch ihren Rang in R ersetzen:

$$\text{SA}_R = (8, 0, 1, 6, 4, 2, 5, 3, 7).$$

Nun weisen wir jedem Suffix einen Rang zu. Dazu sei $\text{rank}(S_i)$ der Rang von S_i in C . Für $i \in B_0$ sei $\text{rank}(S_i)$ nicht definiert.

Hier ist $\text{rank}(S_i) = \perp 1 4 \perp 2 6 \perp 5 3 \perp 7 8 \perp 0 0$.

- 2. Restliche Suffixe sortieren.** Jeder Suffix $S_i \in S_{B_0}$ sei dargestellt durch $(t_i, \text{rank}(S_{i+1}))$. Da wir alle anderen Suffixe oben schon sortiert haben ist $\text{rank}(S_{i+1})$ hier stets definiert.

Offensichtlich ist

$$S_i \leq S_j \Leftrightarrow (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})),$$

also lassen sich die Paare Radix-sortieren.

Hier ist

$$S_{12} < S_6 < S_9 < S_3 < S_0, \quad \text{weil } (\emptyset, 0) < (\text{a}, 5) < (\text{a}, 7) < (\text{b}, 2) < (\text{x}, 1).$$

- 3. Zusammenführen.** Das Zusammenführen erfolgt vergleichsbasiert. Beim Vergleichen von $S_i \in S_C$ mit $S_j \in S_{B_0}$ unterscheiden wir zwei Fälle:

$$i \in B_1 : \quad S_i \leq S_j \Leftrightarrow (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1}))$$

$$i \in B_2 : \quad S_i \leq S_j \Leftrightarrow (t_i, t_{i+1}, \text{rank}(S_{i+2})) \leq (t_j, t_{j+1}, \text{rank}(S_{j+2}))$$

Hier ist z.B. $S_1 < S_6$ weil $(\text{a}, 4) < (\text{a}, 5)$ und $S_3 < S_8$ weil $(\text{b}, \text{a}, 6) < (\text{b}, \text{a}, 7)$.

Suchen in Suffix-Arrays

Um ein Pattern in einem String zu finden, zu dem man das Suffix-Array konstruiert hat, muss man lediglich ein Suffix finden, dass das gesuchte Pattern als Präfix hat. Man kann so beispielsweise mit binärer Suche in $O(m \log n)$ ein Vorkommen von P in T finden.

Nutzen wir eine zusätzliche Struktur, das **LCP-Array** — dieses speichert in $\text{LCP}[i]$ die Länge des längsten gemeinsamen Präfixes von $\text{SA}[i]$ und $\text{SA}[i - 1]$ — so können wir die Suchzeit auf $O(m + \log n)$ reduzieren.

0	banana	5	a	0	a
1	anana	3	ana	1	ana
2	nana	1	anana	3	anana
3	ana	0	banana	0	banana
4	na	4	na	0	na
5	a	2	nana	2	nana

Abbildung 3.7. Suffixe, Suffix-Array und LCP-Array des Strings “banana”.

Um das LCP-Array berechnen zu können brauchen wir das **invertierte Suffix-Array**. Dieses gibt Aufschluss darüber, wo im Suffix-Array ein bestimmter Suffix steht. Offensichtlich ist $\text{SA}^{-1}[\text{SA}[i]] = i$.

Der Algorithmus sieht folgendermaßen aus ($O(n)$):

```
CALCULATELCPARRAY (SA-1, SA)
h := 0, LCP[1] := 0
for i = 1, ..., n do
    if SA-1[i] ≠ 1 then
        while ti+h = tSA-1[i]-1+h do h++
        LCP[SA-1[i]] := h
    h := max(0, h - 1)
```

Suffix-Bäume

Noch anschaulicher, allerdings wesentlich platzverbrauchender, sind **Suffix-Bäume** von Strings. Sie sind formal der *komprimierte Trie der Suffixe* und lassen sich (wenn auch sehr kompliziert) in $O(n)$ berechnen.

Bevor wir den Suffix-Baum eines Strings bilden hängen wir hinten an den String noch einen Charakter dran, der nicht im Alphabet des Strings vorkommt. Das hat den Vorteil, dass anschließend alle Suffixe in einem Blatt des Baums enden.

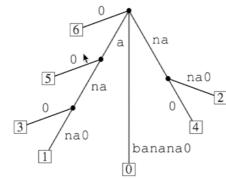


Abbildung 3.8. Beispiel für den Suffix-Baum des Strings “banana”.

“Naiv” ist die Erstellung des Suffixbaums in $O(n^2)$. Man kann ihn aber auch aus Suffix-Array und LCP-Array in Linearzeit konstruieren. Dazu hängt man die Suffixe sukzessive in der Tiefe ein, die ihr LCP-Wert angibt:

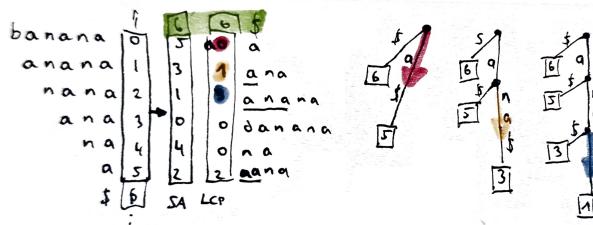


Abbildung 3.9. Sukzessive Konstruktion des Suffix-Baums aus Suffix- und LCP-Array. Zuerst hängt man $\$$ und das erste Suffix (dessen LCP-Wert immer 0 ist) an die Wurzel. Anschließend nutzt man den LCP-Wert des darauffolgenden Suffixes (hier 1), um festzulegen, wo der Suffix zum Baum hinzugefügt werden muss (durch Pfeile gekennzeichnet).

Die Suche in einem Suffix-Baum ist relativ simpel — man muss lediglich den entsprechenden Kanten entlanglaufen, alle Vorkommen des Patterns liegen im entsprechenden Teilbaum.

Zur Angabe der Komplexitäten sind zwei Fälle zu unterscheiden:

1. Die ausgehenden Kanten sind als Arrays der Größe $|\Sigma|$ gespeichert. Dann ist die Suchzeit in $O(m)$ und der Gesamtplatzbedarf in $O(n|\Sigma|)$.
2. Die ausgehenden Kanten sind als Arrays gespeichert, deren Größe proportional zur Anzahl der Kinderknoten ist. Dann ist die Suchzeit in $O(m \log |\Sigma|)$ und der Gesamtplatzbedarf in $O(n)$.

3.3 Datenkompression

Eine Anwendung der Suffix-Arrays und -Trees ist die **Datenkompression**. Inhalt dieser Vorlesung wird ausschließlich die *verlustfreie Textkompression* sein.

Wörterbuchbasierte Textkompression

Für besonders große Datenbestände bietet sich eine **wörterbuchbasierte Textkompression** an. Grundidee ist, $\Sigma^* \subseteq \Sigma^*$ zu wählen und $S \in \Sigma^*$ durch $S' = \langle s'_1, \dots, s'_k \rangle \in \Sigma^{l*}$ zu ersetzen, sodass $S = s'_1 \cdot \dots \cdot s'_k$ ist. Problem ist der hohe zusätzliche Platzbedarf für das Wörterbuch.

Lempel-Ziv-Kompression

Die **Lempel-Ziv-Kompression** baut das Wörterbuch *on the fly* bei Codierung und Decodierung, sodass dieses nicht explizit gespeichert werden muss.

```

NAIVE LZCOMPRESS( $\langle s_1, \dots, s_n \rangle$ ,  $\Sigma$ )
 $D := \Sigma$  // init dictionary
 $p := s_1$  // current string
for  $i := 2$  to  $n$  do
    if  $p \cdot s_i \in D$  then  $p := p \cdot s_i$ 
    else
        output code for  $p$ 
         $D := D \cup p \cdot s_i$ 
         $p := s_i$ 
    output code for  $p$ 
NAIVE LZDECODE( $\langle c_1, \dots, c_k \rangle$ )
 $D := \Sigma$ 
output decode( $c_1$ )
for  $i := 2$  to  $k$  do
    if  $c_i \in D$  then
         $D := D \cup \text{decode}(c_{i-1}) \cdot \text{decode}(c_i)[1]$ 
    else
         $D := D \cup \text{decode}(c_{i-1}) \cdot \text{decode}(c_{i-1})[1]$ 
    output decode( $c_i$ )

```

Abbildung 3.10. Kompressions- und Dekodierungs-Algorithmus für die Lempel-Ziv-Kompression.

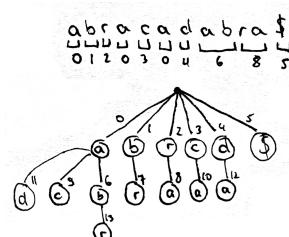


Abbildung 3.11. Beispiel für das Bilden der Lempel-Ziv-Kompression des Wortes "abracadabra". Der Baum wird *on the fly* berechnet und nicht übergeben, sondern nur die komprimierte Information und das Alphabet.

4

Range Minimum Queries

Eine **range minimum Query** gibt für ein array A ($|A| =: n$) die Position des kleinsten Elements zwischen zwei Begrenzern $1 \leq l < r \leq n$ zurück:

$$\text{rmq}_A(l, r) = (\arg \min_{l \leq k \leq r} A[k])$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	2	4	7	1	9	3	5	7	4	6	4	3	1	4	8

$\text{rmq}(5, 9) = 6$

Abbildung 4.1. Beispiel einer range minimum query.

Ziel dieses Kapitels wird sein, einen Algorithmus anzugeben, mit dem eine RMQ-Abfrage in konstanter Zeit beantwortet werden kann, nachdem eine $2n + o(n)$ große Datenstruktur in Linearzeit vorbereitet wurde.

Ein naiver Ansatz, um eine range minimum query auszuführen, ist, einfach das Array zu durchlaufen und das Minimum zu speichern (und wenn nötig zu aktualisieren). Dafür ist keine Vorbereitungarbeit nötig (also $O(1)$) und die Abfrage ist in $O(n)$. Wir notieren

$$\langle O(1), O(n) \rangle.$$

4.1 Lösung 1 — $\langle O(n), O(\log n) \rangle$

Baut man einen binären Suchbaum über das Array auf, so lässt sich die Komplexität der Abfrage auf $O(\log n)$ reduzieren.

Hierzu betrachtet man die größtmöglichen Knoten, die vollständig im Abfrageintervall liegen (grün dargestellt), und berechnet das Minimum dieser.

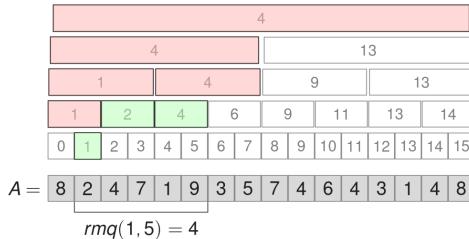


Abbildung 4.2. Die grauen Felder stellen die Knoten des binären Suchbaums dar, sie beinhalten die Position des Arrays, an der der Teilbaum, dessen Wurzel sie sind, den minimalen Wert annimmt. Die größten Knoten, die vollständig im Intervall liegen, sind grün markiert..

4.2 Lösung 2 — $\langle O(n \log n), O(1) \rangle$

Wir reduzieren nun die Zeit, die zum Bearbeiten der rmq benötigt wird, auf $O(1)$, indem wir für jedes $A[i]$ ein Array $M_i[0, \log n]$ vorberechnen. Es sei

$$M_i[j] = \text{rmq}_A(i, i + 2^j - 1).$$

Idee ist es nun, $\text{rmq}_A(l, r)$ aus der Überdeckung des Intervalls durch zwei Zweierpotenzen zu berechnen.

Wir suchen dafür $2^{\lfloor l-r \rfloor}$, also die größte Zweierpotenz, die kleiner ist als die Länge des Intervalls. Offensichtlich ist diese Zweierpotenz mehr als halb so groß wie das Intervall, also ist $\text{rmq}_A(l, r)$ entweder das Minimum der ersten oder zweiten überdeckenden Zweierpotenz.

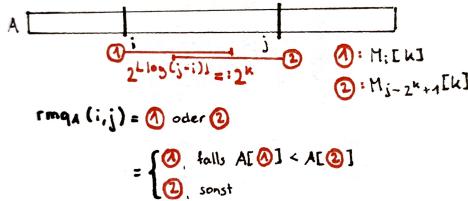


Abbildung 4.3. Funktionsweise des zweiten rmq-Algorithmus.

4.3 Lösung 3 – $\langle O(n \log \log n), O(1) \rangle$

Wir wenden folgende Prozedur an:

1. A in $t = \frac{n}{\log n}$ Blöcke B_0, \dots, B_{t-1} der Größe $\log n$ unterteilen.
2. Array $S[0, t-1]$ mit $S[i] = \min \{x \in B_i\}$ erstellen, rmq-Struktur nach Lösung 2 für S berechnen.
3. Für jeden Block B_i rmq-Struktur nach Lösung 2 berechnen.

Diese Prozedur liegt in $O(n \log \log n)$.

Soll nun $\text{rmq}_A(l, r)$ bestimmt werden, so geht das folgendermaßen:

1. Bestimme die Blöcke $l \in B_{l'}$ und $r \in B_{r'}$.
2. Berechne $m = \text{rmq}_S(l'+1, r'-1)$. Wir nutzen also die Struktur über S , um die rmq-Werte der Blöcke zwischen den beiden Grenzblöcken zu berechnen.
3. Es seien k_0, k_1, k_2 die rmq_A -Resultate in den Blöcken l', r' und m
4. $\text{rmq}_A(l, r) = \arg \min \{A[k_0], A[k_1], A[k_2]\}$.

4.4 Lösung 4 – $\langle O(n), O(1) \rangle$

Zur Konstruktion des Algorithmus mit linearem Platzverbrauch benötigen wir kartesische Bäume.

Kartesischer Baum

Der **kartesische Baum** eines Arrays A ist folgendermaßen definiert:

- Wurzel des Baums ist das (linkste) kleinste Element m des Arrays, mit Position als Label.

- Die Wurzel hat zwei Kindknoten — die Wurzel des linken und die Wurzel des rechten Subarrays von A (jeweils von m aus).
- Rekursion.

Implementierung

Die Implementierung funktioniert so:

1. Partitioniere das Array in Blöcke der Größe s — jeder Block entspricht einem kartesischen Baum.
2. Berechne alle s^2 Möglichkeiten aller $\frac{1}{s+1} \binom{2s}{s}$ möglichen kartesischen Bäume der Größe s in einer Tabelle P . Diese benötigt $O(2^{2s}s^2)$ Speicher, also braucht P für $s = \frac{\log n}{4}$ nur $o(n)$ Speicher.
3. Berechne nach Lösung 2 die Struktur für das Array A' , welche aus den blockweisen Minima von A besteht ($O(n)$ Zeit, $O(n)$ Platz).

4.5 Lowest Common Ancestor

Gegeben sei ein Baum T mit Wurzel, $|T| = n$. Es sei

$$\text{LCA}_T(v, w)$$

der **lowest common Ancestor** der Knoten v, w in T , also der Knoten $a \in T$, der

- Vorgänger von v und w ist und
- maximalen Abstand zur Wurzel hat.

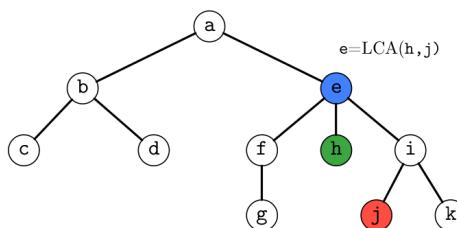


Abbildung 4.4. lowest common ancestor der Knoten h und j ist e .

Von RMQ zu LCA

Lemma 4.5.1. Gibt es eine $\langle f(n), g(n) \rangle$ -Lösung für RMQ, so gibt es eine Lösung in

$$\langle f(2n - 1) + O(n), g(2n - 1) + O(1) \rangle$$

für LCA.

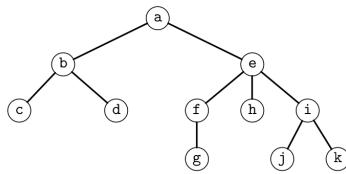
Beweis. Sei

- T der kartesische Baum des Arrays A ,
- $E[0, \dots, 2n - 2]$ das Knoten-Array, die in einer DFS-Euler-Tour von T besucht wurden,
- $L[0, \dots, 2n - 2]$ die entsprechende Tiefe der Knoten in E ,
- $R[0, \dots, n - 1]$ ein Array mit $R[i] = \min\{j : E[j] = i\}$ für jeden Knoten $i \in T$, also wo in der Euler-Tour der Knoten i das erste Mal auftaucht.

Dann ist

$$\text{LCA}_T(v, w) = E[\text{RMQ}_L(\min\{R[v], R[w]\}, \max\{R[v], R[w]\})].$$

□



$i =$	0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
$E =$	a b c b d b a e f g f e h e i j i k i e a
$L =$	0 1 2 1 2 1 0 1 2 3 2 1 2 1 2 3 2 3 2 1 0
	a b c d e f g h i j k
$R =$	0 1 2 4 7 8 9 12 14 15 17

Abbildung 4.5. Konstruktion der Hilfsarrays für LCA.

Achtung: Die Tiefe zweier Knoten, die nacheinander bei der Euler-Tour besucht wurden, kann sich höchstens um 1 unterscheiden, also

$$(L[i] - L[i + 1]) \in \{-1, 1\}.$$

Wir müssen also nur RMQs über Arrays lösen, bei denen sich zwei nacheinanderfolgende Elemente um 1 unterscheiden. Das ist das sogenannte **±1-RMQ**.

Wir werden nun im Folgenden ausnutzen, dass wir zum Lösen des LCA-Problems nur ± 1 -RMQs betrachten müssen.

LCA in $\langle O(n), O(1) \rangle$ auf $4n + o(n)$ Bits

Wir verwenden folgende Konstruktion:

- **Kartesischen Baum bauen.** Wir gehen das Array A von links nach rechts durch und fügen die Elemente sukzessive dem Baum hinzu. Dabei sind stets die Regeln für einen kartesischen Baum einzuhalten, also
 - Wurzel des Baums ist das (linkste) kleinste Element m des Arrays.
 - Die Wurzel hat zwei Kindknoten – die Wurzel des linken und die Wurzel des rechten Subarrays von A (jeweils von m aus).
 - Rekursion.
- **Dummy-Blätter hinzufügen.** An jeden Knoten des Baums hängen wir ein Dummy-Blatt.
- **DFS.** Wir konstruieren eine Sequenz an Klammern, indem wir ein DFS durchführen. Traversieren wir einen Knoten das erste Mal, so setzen wir “(“ beim zweiten Mal “)”. In einem Blatt drehen wir um und setzen daher direkt beide Klammern.

Wir können die Klammernfolge in $4n$ Bits speichern, indem wir “(“ durch 1 und “)” durch 0 repräsentieren.

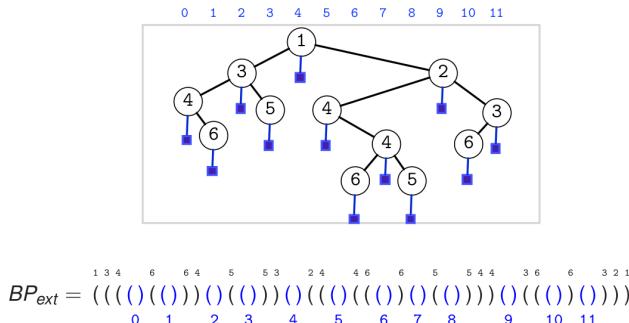


Abbildung 4.6. Konstruktion des kartesischen Baums mit Dummy-Blättern (blau) für das Array “463514645236”. Anschließend wird die Klammernfolge gebildet. In blau hervorgehoben sind die “Wendepunkte”, also die Dummy-Blätter, und ihr Index.

- **RMQ berechnen.** Zur Berechnung von $\text{rmq}_A(l, r)$ brauchen wir drei Hilfsfunktionen:
 - $\text{rank}(\text{pos}, \text{char}, BP_{ext})$ gibt die Anzahl an Vorkommisse von char (z.B. “)”) bis pos in BP_{ext} an.
 - $\text{excess}(i) = \text{rank}(i + 1, (, BP_{ext}) - \text{rank}(i + 1,), BP_{ext})$
 - $\text{select}(j, \text{char}, BP_{ext})$ gibt die Position des j -ten Vorkommens von char in BP_{ext} an.

Wir können nun den RMQ-Wert folgendermaßen berechnen:

```
RMQ ( $A, l, r$ )
lpos := select( $l + 1, (), \text{BP}_{\text{ext}}$ )
rpos := select( $r + 1, (), \text{BP}_{\text{ext}}$ )
return rank( $\text{rmq}_{\text{excess}}^{\pm 1}(\text{lpos}, \text{rpos} + 1), (), \text{BP}_{\text{ext}}$ )
```

LCA in $\langle O(n), O(1) \rangle$ auf $2n + o(n)$ Bits

Die Anzahl an benötigten Bits lässt sich im Vergleich zum vorhergehenden Ansatz noch weiter verkleinern. Wir transformieren dazu den kartesischen Baum — der ja ein Binärbaum ist — in einen allgemeinen Baum. Dazu gehen wir folgendermaßen vor:

1. Füge einen Elternknoten zur Wurzel des Baums hinzu (der hinzugefügte Knoten ist also die neue Wurzel).
2. Nehme von jedem Knoten v — von der neuen Wurzel ausgehend — den rechten Kindknoten w , und füge alle Knoten auf dem linkesten Pfad von w ausgehend zu den Kindknoten von v hinzu.

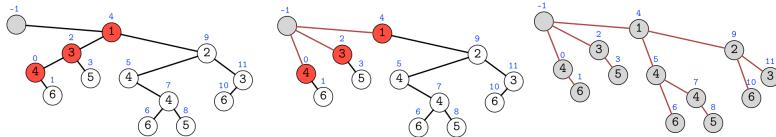


Abbildung 4.7. Kartesischer Baum mit neuer Wurzel. Die roten Knoten in der ersten Grafik sind die Knoten, die zu v — hier die neue Wurzel — hinzugefügt werden. Durch Hinzufügen dieser Knoten zu den Kindknoten von v hat der Baum die Struktur in Grafik 2. Nun wird dieser Prozess rekursiv auf die drei roten Knoten ausgeführt. Ist die Rekursion vollständig abgeschlossen sieht der Baum wie in der dritten Grafik aus.

Diese Transformation lässt sich bei Bedarf auch rückgängig machen; man kann also, wenn nötig, wieder den Binärbaum konstruieren.

Nun lässt sich wieder die Klammerreihe BP von oben bauen. Mit dieser können wir nun RMQ-Abfragen lösen:

```
RMQ ( $A, l, r$ )
lpos := select( $l + 2, (), \text{BP}$ )
rpos := select( $r + 2, (), \text{BP}$ )
return rank( $\text{rmq}_{\text{excess}}^{\pm 1}(\text{lpos} - 1, \text{rpos}), (), \text{BP} - 1$ )
```


5

Burrows-Wheeler- Transformation

Die **Burrows-Wheeler-Transformation** erzeugt eine sinnvolle Permutation des eingegebenen Strings; sie gruppiert Zeichen mit ähnlichem Kontext nahe beieinander. Die Struktur der Permutation beinhaltet alle Informationen, die benötigt werden, um eine Rücktransformation durchzuführen, es sind also keine Zusatzinformationen nötig. Hin- und Rücktransformation geht in $O(n)$. Sie wird hauptsächlich zur Vorverarbeitung statischer Texte genutzt, um sie komprimieren, indizieren und in ihnen suchen zu können.

5.1 Konstruktion

Sei $T = \text{lalalangng\$}$ der gegebene String (mit angehängtem \\$-Zeichen), $n = |T|$ und $T^{(i)}$ die i -te Permutation von T (durch i mal den vordersten Buchstaben nehmen und hinten anhängen). Man erhält die Burrows-Wheeler-Transformation von T so:

1. Schreibe $T^{(1)}$ bis $T^{(n)}$ untereinander.
2. Sortiere $T^{(1)}$ bis $T^{(n)}$.
3. Die letzte Spalte ist T^{BWT} ($= L$), die Burrows-Wheeler-Transformation von T .

5 Burrows-Wheeler-Transformation

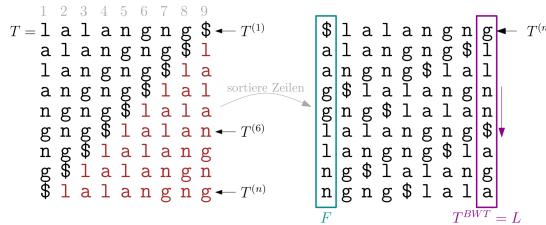


Abbildung 5.1. Konstruktion der Burrows-Wheler-Transformation von $T = 1alalangng\$$, $T^{BWT} = g1lnn\$ aga$. Da T^{BWT} die letzte Spalte ist schreibt man oft auch L stattdessen. Die erste Spalte wird auch F genannt.

Naiv benötigt die Berechnung von T^{BWT} $O(n^2 + n \log n)$ Schritte. Die Berechnungszeit lässt sich aber auf $O(n)$ reduzieren.

5.2 Beobachtungen

Folgende Eigenschaften lassen sich feststellen:

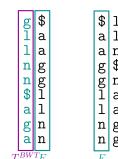
- Die Zeilen der oben konstruierten Matrix enthalten die sortierten Suffixe von T (vom Zeilenstart bis $\$$ gehend).
- Die Zeichen der letzten Spalte (also T^{BWT}) sind also die Zeichen, die vor dem zu ihrer Zeile gehörenden Suffix stehen. Formaler ist $T^{BWT}[i]$ das Zeichen vor dem i -ten Suffix in T :

$$T^{BWT}[i] = L[i] = T[\text{SA}[i] - 1] = T^{(\text{SA}[i])}[n]$$

Da wir mithilfe des **DC3-Algorithmus** das Suffix-Array in Linearzeit berechnen können, können wir auch die Burrows-Wheeler-Transformation in Linearzeit bestimmen.

5.3 Rücktransformation

Wir können aus einer vorliegenden T^{BWT} einfach F — also die erste Spalte der Matrix — konstruieren, indem wir die Buchstaben von T^{BWT} sortieren. Hängen wir nun T^{BWT} und F hintereinander, so haben wir bereits Buchstabenpaare, die so auch in T auftreten. Sortieren wir nun die beiden Spalten (also die Buchstabenpaare) lexikografisch, so erhalten wir die auf F folgende Spalte. Durch diesen Prozess lässt sich die gesamte Matrix und somit T rekonstruieren.



Diese Art der Rücktransformation benötigt $O(n^2 \log n)$ Schritte. Im Folgenden werden wir die Rücktransformation auf Linearzeit reduzieren. Dazu benötigen wir **Last-to-front mapping**:

$\text{LF}[i] :=$ Position in L , an der Vorgänger von $L[i]$ steht

Da die Spalten der BWT-Matrix zyklisch sind, ist der Vorgänger von $L[i]$ derjenige Buchstabe, der in $F[i]$ steht, also

$\text{LF}[i] =$ Position, an der $L[i]$ in F steht

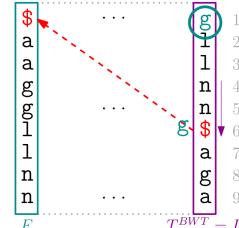


Abbildung 5.2. Gesucht ist der Vorgänger von \$. Stellen wir uns T^{BWT} ein zweites Mal links von F vor, so sehen wir, dass es g ist.

Wir erhalten folgenden Zusammenhang:

$$\text{LF}[i] = j \Leftrightarrow T^{(\text{SA}[j])} = (T^{(\text{SA}[i])})^{(n)}.$$

Weitere Überlegungen zur Rücktransformation

Wir können desweiteren folgende Beobachtungen an T^{BWT} machen:

- Gleiche Zeichen haben gleiche Reihenfolge in F und L .
- Falls $L[i] = L[j]$ für $i < j$, dann ist $\text{LF}[i] < \text{LF}[j]$.

Grund dafür ist, dass die Zeilen der BWT-Matrix lexicographisch sortiert sind.

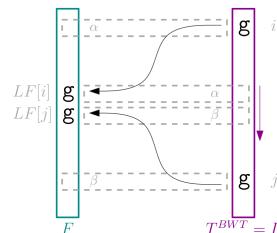


Abbildung 5.3. Präfixe α und β und wie sie in der Matrix vorkommen.

Wir können also LF rein aus T^{BWT} berechnen. Dazu brauchen wir nur zwei Hilfsfunktionen:

- $C(a) := \# \text{ Zeichen } < a$
- $\text{occ}[i] := \# \text{ Zeichen } = L[i] \text{ in } L[1 \dots i]$

Nun können wir $\text{LF}[i]$ darstellen als

$$\text{LF}[i] = C(L[i]) + \text{occ}[i]$$

und können somit LF in $O(n)$ berechnen, da sich C und occ in Linearzeit berechnen lassen.

Implementierung

Zuerst berechnen wir LF. Hier sieht die Implementierung so aus:

1. Initialisiere occ und h . h sei ein Array, das zählt, wie oft ein bestimmter Buchstabe vorkommt, damit wir nachher C gescheit berechnen können.
2. Laufe durch $L = T^{\text{BWT}} (i = 1 \dots n)$
 - $h(L[i])++$
 - $\text{occ}(L[i]) = h(L[i])$
3. Konstruiere C aus h : $C(\$) = 0, C(\alpha) = C(\alpha - 1) + h(\alpha - 1)$ (α ist ein Buchstabe, $\alpha - 1$ sein Vorgänger)
4. $\text{LF}[i] = C(L[i]) + \text{occ}[i]$

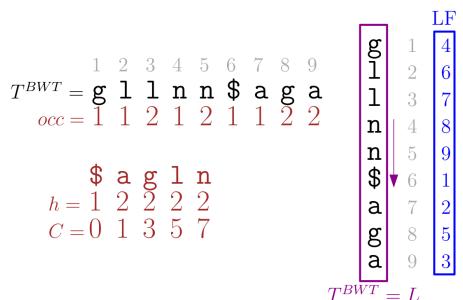
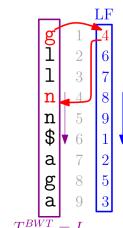


Abbildung 5.4. Beispiel des Algorithmus zur Berechnung von LF nach Durchführung.

Nun kann T von rechts nach links berechnet werden:

1. $T[n] = \$ \Rightarrow \text{LF}[\cdot] = 1$. Das ist unabhängig von T so.
2. $L[1] = g \Rightarrow T[n-1] = g \Rightarrow \text{LF}[1] = 4$
3. $L[4] = n \Rightarrow T[n-2] = n \Rightarrow \dots$

Also geht auch die Rücktransformation in $O(n)$.



5.4 Was bringt die BWT?

Die Vorteile der Burrows-Wheeler-Transformation sind nicht direkt erkennbar — sie nutzt dieselben Zeichen wie T und benötigt den gleichen Platz.

Allerdings wird die *Komprimierung stark vereinfacht*, weil Zeichen mit ähnlichem Kontext gruppiert werden. Besonders gut funktioniert sie auf Texten mit vielen gleichen Substrings, wie beispielsweise einem englischen Fließtext. Zur Vereinfachung von *Indexierung* und *Suche* steuert sie auch bei, weil Vorgänger von Suffixen einfach bestimmt werden können.

Im Folgenden werden wir uns die Burrows-Wheeler-Transformation im Kontext von *Kompression* und *Suche* anschauen.

5.5 Kompression

Wir schauen uns zwei Kompressionsmöglichkeiten an: die *move to front*-Kodierung und die Huffman-Kodierung

MTF-Kodierung

Idee der **MTF-Kodierung** ist es, lokale Redundanz zu nutzen und so kleine Zahlen für gleiche Zeichen, die nahe beieinander liegen, zu verwenden. Die Umsetzung funktioniert so:

1. Initialisiere Y mit Alphabet von T^{BWT} .
2. Durchlaufe T^{BWT} ($i = 1, \dots, n$)
 - Generiere $R[1, \dots, n]$, wobei $R[i]$ die Position von $T^{\text{BWT}}[i]$ in Y codiert.
 - Schiebe $T^{\text{BWT}}[i]$ an den Anfang von Y .

$$\begin{array}{ccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
 T^{\text{BWT}} = & g & l & l & n & n & a & g & a & \$ \\
 R = & 3 & 4 & & & & & & &
 \end{array}
 \qquad
 \begin{array}{cccccc}
 & 1 & 2 & 3 & 4 & 5 \\
 \$ & a & g & l & n \\
 g & \$ & a & l & n \\
 Y = & \textcolor{teal}{1} & g & \$ & a & n
 \end{array}$$

Abbildung 5.5. Es wurde hier gerade die 4 eingefügt und deswegen 1 in Y nach vorne genommen. Als nächstes muss 1 codiert (≥ 1) und Y anschließend nicht verändert werden, weil 1 ja eh schon ganz vorne steht.

Huffman-Kodierung

Die **Huffman-Kodierung** erzeugt präfixfreie Codes variabler Länge. Der Ablauf ist:

1. Notiere vorkommende Symbole und ihre jeweiligen Häufigkeiten. Sie sind die Blätter des (binären) Huffman-Baumes.
2. Verknüpfe die zwei seltensten Knoten in einem neuen Knoten. Die Häufigkeit des neuen Knotens ist die Summe der Häufigkeiten seiner Kinder. Dies erfolgt nun iterativ.
3. Die Wurzel hat relative Häufigkeit 1 (bzw. absolute Häufigkeit $|T|$).
4. Beschrifte die Kanten zwischen einem Knoten und seinen beiden Kindern mit 0 und 1. Der Pfad von der Wurzel zu einem bestimmten Blatt ergibt den Code des Symbols, zu dem das Blatt gehört.

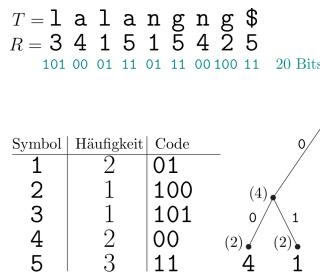


Abbildung 5.6. MTF-Kodierung R von T , die Häufigkeit der in R vorkommenden Symbole und die mit dem Huffman-Baum erzeugten Codes.

5.6 Suche in der Burrows-Wheeler-Transformation

Wir möchten nun in T^{BWT} nach einem Pattern P suchen. Hier sei

$$P = \text{bar} \quad \text{und}$$

$$T = \text{abracadabrabarbara\$} \quad \text{und somit}$$

$$\text{BWT} = \text{arrd\$rcbbraaaaaabba}$$

Wir benötigen dazu zwei Hilfsmittel:

- Das Array C beinhaltet für jeden eindeutigen Buchstaben in $t \in T$ die Position des ersten Suffixes im Suffix-Array, das mit t beginnt.

- $\text{rank}(i, X, \text{BWT})$ gibt an, wie oft ein Buchstabe X in $\text{BWT}[0, \dots, i - 1]$ vorkommt.

Wir suchen nun nach P in BWT. Wir suchen rückwärts, starten also mit "r". Dazu ermitteln wir alle Suffixe, die mit r starten. Wir nutzen dazu C und rank :

- Initiales Intervall: $[\text{sp}_0, \text{ep}_0] = [0, \dots, n - 1]$.
- Ermittle Intervall der Suffixe, die mit r starten:
 - $\text{sp}_1 = C[r] + \text{rank}(\text{sp}_0, r, \text{BWT}) = 15 + \text{rank}(0, r, \text{BWT}) = 15 + 0 = 15$
 - $\text{ep}_1 = C[r] + \text{rank}(\text{ep}_0 + 1, r, \text{BWT}) - 1 = 15 + 4 - 1 = 18$

i	BWT	$T[SA[i]..n - 1]$
0	a	\$
1	r	a\$
2	r	abarbara\$
3	d	abrabarbara\$
4	\$	abracadabrabarbara\$
5	r	acadabrabarbara\$
6	c	adabrabarbara\$
7	b	ara\$
8	b	arbara\$
9	r	bara\$
10	a	barbara\$
11	a	brabarbbara\$
12	a	bracadabrabarbara\$
13	a	cadabrabarbara\$
14	a	dabrabarbara\$
15	a	ra\$
16	b	rabarbara\$
17	b	racadabrabarbara\$
18	a	rbara\$

Abbildung 5.7. Intervall $[\text{sp}_1, \text{ep}_1]$.

Zusammenfassung

Wir brauchen also nur C und R , um Abfragen zu Existenz und Anzahl eines Patterns machen zu können. Die Ausführungszeit ist in $O(m \cdot t_{\text{rank}})$, wobei t_{rank} die Ausführungszeit einer rank-Operation ist.

Als nächstes werden wir uns damit beschäftigen, wie wir die rank-Operation implementieren können. Wir werden dazu *wavelet trees*¹ verwenden.

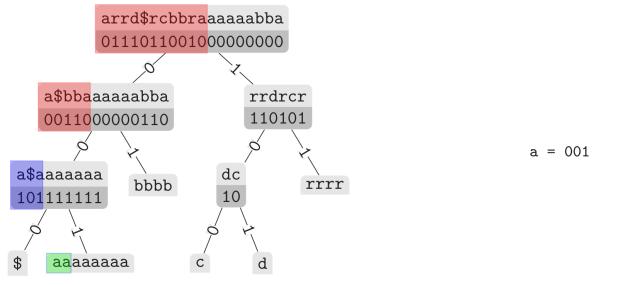
5.7 Wavelet Trees

Wavelet Trees erlauben ein schnelles Berechnen der rank-Operation. Dazu wird in einem Baum codiert, ob ein bestimmter Buchstabe des Strings (hier des BWT) in der oberen oder der unteren Hälfte des Alphabets liegt. So werden die Buchstaben des BWT einem Kindknoten zugeordnet, wo auf dem jeweiligen Teilalphabet erneut eine

¹ Grossi & Vitter, 2003

5 Burrows-Wheeler-Transformation

Zweiteilung stattfindet. Dieser Prozess wiederholt sich so lange, bis in jedem Blatt nur Zeichen einer Art stehen.



$$\text{rank}(11, a, \text{WT}) = \text{rank}(\text{rank}(\text{rank}(11, 0, b_c) = 5, 0, b_0) = 3, 1, b_{00}) = 2$$

Abbildung 5.8. Wavelet Tree einer BWT und zugehörige Berechnung von $\text{rank}(11, a, \text{BWT})$.

Abfragen können auf einem Wavelet Tree in konstanter Zeit durchgeführt werden. Der Wavelet Tree selbst benötigt $o(n)$ viel Platz, genauer

$$O\left(\frac{n}{\log n} + \frac{n \log \log n}{\log n} + \sqrt{n} \log n \log \log n\right).$$

Die Konstruktion des Wavelet-Trees ist nicht zwingend an die Unterteilung des BWT zwei lexikographische Teilalphabete gebunden. Beispielsweise lässt sich eine Unterteilung in zwei Teilwörter auch durch die Häufigkeit der Buchstaben konstruieren, wodurch ein **Huffman-Wavelet-Tree** entsteht.

5.8 Exkurs — Succinct Data Structures

Eine extrem platzeffiziente Datenstruktur, **succinct data structure** genannt, benötigt nur wenig mehr Platz als die informationstheoretische untere Grenze, unterstützt allerdings Operationen zeiteffizient.

Sei L die informationstheoretische untere Schranke, die zur Repräsentierung einer Klasse von Objekten benötigt wird. Eine Datenstruktur, die Operationen trotzdem zeiteffizient unterstützt, heißt

- *implicit*, falls sie $L + O(1)$ Bits Platz braucht (also nur konstant mehr als die informationstheoretische untere Schranke, z.B. Heap),
- *succinct*, falls sie $L + o(L)$ Bits Platz braucht (also nur sublinear mehr als die informationstheoretische untere Schranke, z.B. Baum),
- *compact*, falls sie $O(L)$ Bits Platz braucht.

Binäräume – succinct

Es gibt $C_n = \frac{1}{n+1} \binom{2n}{n}$ Binäräume auf n Knoten. Um einen Binärbaum auf n Knoten speichern zu können, brauchen wir $\log C_n = 2n + o(n)$ bits.² Wir wollen folgende Operationen unterstützen:

- $\text{parent}(v)$
- $\text{leftchild}(v)$
- $\text{rightchild}(v)$

Eine mögliche Kodierung wäre, zu jedem Knoten des Baumes, der kein Blatt ist, so viele imaginäre Knoten hinzuzufügen, dass der jeweilige Knoten zwei Kinder hat (also entweder, 0, 1 oder 2 imaginäre Knoten). Diesen Prozess führt man durch, bis alle Blätter des Baumes dieselbe Tiefe haben. Wir können nun alle Knoten durchnummieren und pro Knoten in einem Bit-Array speichern, ob der Knoten real (=1) ist oder nicht (=0).

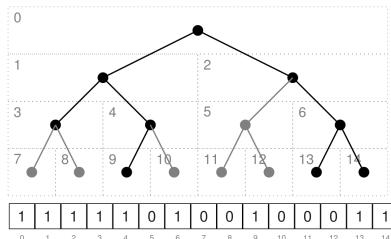


Abbildung 5.9. Die schwarzen Knoten sind die realen Knoten des Baumes, die grauen die imaginären.

Die geforderten Operationen können hier sehr einfach implementiert werden:

- $\text{parent}(v) = \left\lfloor \frac{v-1}{2} \right\rfloor$ (für $v \neq$ Wurzel)
- $\text{leftchild}(v) = 2v + 1$
- $\text{rightchild}(v) = 2v + 2$

Problem dieses Ansatzes ist, dass für einen Binärbaum mit einer Maximaltiefe d stets 2^d Bits benötigt werden.

Jacobson schlug 1989 einen besseren Algorithmus vor:

1. Alle Knoten des Binärbaums mit 1 markieren.
2. Kinder jedes Knotens im Baum mit imaginären Knoten auf 2 ergänzen.
3. Bit-Markierungen wie im vorhergehenden Algorithmus ablesen.

² Das kann mithilfe der Sterling-Approximation gezeigt werden.

5 Burrows-Wheeler-Transformation

Benötigt wird eine weitere Operation $\text{rank}(i, \text{type}, b)$, die für Knoten i des ergänzten Baums b zurückgibt, um den wievielten Knoten des Typs type es sich handelt.
Wir erhalten folgendes Resultat:

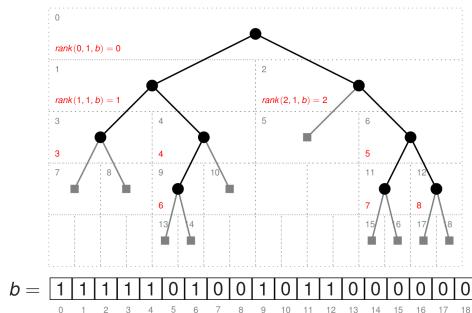


Abbildung 5.10. Jacobson-Kodierung eines Binärbaums.

Wir können hiermit also einen Binärbaum mit einem Bit-Array der Länge $2n + 1$ (mit n gesetzten Bits) repräsentieren. Die gewünschten Operationen funktionieren hier so:

- $\text{leftchild}(v) = 2\text{rank}(v) + 1$
- $\text{rightchild}(v) = 2\text{rank}(v) + 2$
- $\text{parent}(v) = \text{auch in konstanter Zeit möglich}$ ³

Der totale Platzverbrauch inklusive rank ist also $2n + o(n)$ Bits.

Bäume – succinct: LOUDS

Die Abkürzung **LOUDS** steht für **level order unary degree sequence**.

Die Implementierung funktioniert so:

1. Füge über der Wurzel des Baums eine Pseudo-Wurzel hinzu und verbinde sie nur mit der alten Wurzel.
2. Der Ausgangsgrad wird zu jedem Knoten unär⁴ codiert dazugeschrieben.

Die **LOUDS-Sequenz** ist nun die Konkatenation der Knotenmarkierungen (sortiert nach Level).

³ Übungsaufgabe!

⁴ Ausgangsgrad x : x mal 0, hintendran immer noch eine 1.

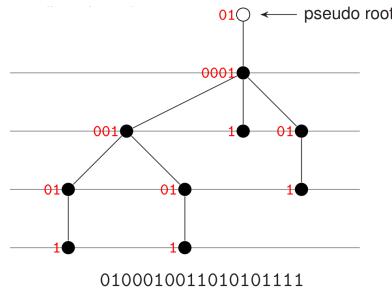


Abbildung 5.11. Allgemeiner Baum mit Pseudo-Wurzel, Knotenmarkierungen und LOUDS-Sequenz.

Die Konstruktion verursacht, dass — abgesehen von der Wurzel — jeder Knoten zweimal in der LOUDS-Sequenz vorkommt: einmal als \emptyset in der Kinder-Liste seines Elternknotens und einmal als terminierende 1 in seiner eigenen Kinder-Liste.

Der gesamte Platzverbrauch hiervorn ist $2n + 1 + o(n)$ Bits. Außerdem lassen sich alle gewünschten Operationen in konstanter Zeit implementieren:

```
ISLEAF( $v$ )
id := rank( $v, 0$ , LOUDS)
 $p$  := select(id + 2, 1, LOUDS)
return LOUDS[ $p - 1$ ] == 1
```

```
OUTDEGREE( $v$ )
if ISLEAF( $v$ ) then return 0
id := rank( $v, 0$ , LOUDS)
return select(id + 2, 1, LOUDS) -
       select(id + 1, 1, LOUDS) - 1
```

```
CHILD( $v, i$ )
if  $i <$  OUTDEGREE( $v$ ) then
    return ⊥
id := rank( $v, 0$ , LOUDS)
return select(id + 1, 1, LOUDS) +  $i$ 
```

```
PARENT( $v$ )
if ISROOT( $v$ ) then
    return ⊥
pid := rank( $v, 1$ , LOUDS)
return select(pid, 0, LOUDS)
```


6

Geometrische Algorithmen

Inhalt dieses Kapitels:

- Plane-Sweep-Algorithmus
- Konvexe Hülle
- Kleinste einschließende Kugel
- Range Search

6.1 Grundlegende Definitionen

Wir nennen $p \in \mathbb{R}^d$ einen **Punkt**. $p.i$ stelle die i -te Komponente von p dar. Für $d \in \{2, 3\}$ schreiben wir $p.x, p.y, p.z$ statt $p.1, p.2, p.3$.

Für zwei Punkte a, b definieren wir

$$\overline{ab} := \{\alpha \cdot a + (1 - \alpha) \cdot b : \alpha \in [0, 1]\}$$

als das **Segment** zwischen a und b .

Ein **Polygon** ist eine Menge an Segmenten, gegeben als Punktemenge $P = p_1, \dots, p_n$ mit $p_i \in \mathbb{R}^d$, $p_n = p_1 \cdot \overline{p_i, p_{i+1}}$ für $i = 1, \dots, n - 1$ ist der **Umriss** des Polygons.

Ist für alle $a, b \in P$ auch $\overline{ab} \in P$, so nennen wir P **konvex**.

6.2 Streckenschnitte

Bei diesem Problem sind n Strecken $S = \{s_1, \dots, s_n\}$ gegeben und wir wollen alle Schnittpunkte dieser, also $\bigcup_{s, t \in S} s \cap t$ berechnen.

Naiv lassen sich diese Streckenschnitte in $O(n^2)$ berechnen:

```
foreach  $\{s, t\} \subseteq S$  do
    if  $s \cap t \neq \emptyset$  then output  $\{s, t\}$ 
```

Dieser Algorithmus ist für große Datenmengen offensichtlich zu langsam.

Idee ist nun, dass eine (waagerechte) **Sweep-Line** von oben nach unten läuft. Dabei speichern wir Segmente, die l schneiden, und finden deren Schnittpunkte. Invariante ist, dass Schnittpunkte oberhalb von l korrekt ausgegeben wurden.

Orthogonale Streckenschnitte

Zuerst betrachten wir die Vereinfachung, dass nur orthogonale Segmente (also parallel zur x - oder y -Achse existieren).

```
 $T := \langle \rangle$  SortedSequence of Segment
invariant  $T$  stores vertical segments intersecting  $l$ 
 $Q := \text{sort}(\{(y, s) : \exists \text{ hor-seg } s \text{ at } y \vee \exists \text{ ver-seg } s \text{ starting/ending at } y\})$ 
foreach  $(y, s) \in Q$  in descending order do
    if  $s$  is ver-seg and starts at  $y$  then  $T.\text{insert}(s)$ 
    elif  $s$  is ver-seg and ends at  $y$  then  $T.\text{remove}(s)$ 
    else // horizontal segment  $s = (x_1, y)(x_2, y)$ 
        foreach  $t = \overline{(x, y_1)(x, y_2)} \in T$  with  $x \in [x_1, x_2]$  do output  $\{s, t\}$ 
```

Hier sind T und Q die einzigen komplexen Datenstrukturen, die wir benötigen, also sortierte Listen an Segmenten (T geordnet nach x -Wert, Q nach y).

insert und remove gehen in $O(\log n)$, die rangeQuery für ein Segment in $O(\log n + k_s)$ (bei k_s Schritten mit horizontalem Segment s). Insgesamt haben wir also

$$O(n \log n + \sum_s k_s) = O(n \log n + k).$$

Verallgemeinerung

Wir verallgemeinern jetzt den Spezialfall von oben, verwenden allerdings folgende Vereinfachungen: Es gebe keine horizontalen Segmente und Überschneidungen sind

immer nur zwischen zwei Segmenten, nicht mehr. Außerdem soll es keine Überlappungen geben, die Anzahl an Schnitten zwischen zwei Segmenten ist also immer entweder 0 oder 1.

Wir verwenden wieder T als nach x geordnete Liste der Strecken, die l schneidet. Außerdem verwenden wir *Ereignisse* — diese sind Änderungen von T , also das Starten und Enden von Segmenten sowie Schnittpunkte.

Einen Schnitttest müssen wir nur dann durchführen, wenn zwei Segmente an einem Ereignispunkt in T benachbart sind.

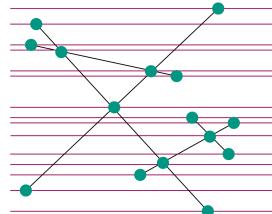


Abbildung 6.1. Die grünen Punkte stellen die Ereignisse dar. Außerdem ist l zum Zeitpunkt der Ereignisse dargestellt.

Zur Implementierung brauchen wir nun einige Zusatzmethoden:

findNewEvent ermittelt, ob es einen Schnitt zwischen zwei Segmenten s und t gibt.

```
FINDNEWEVENT( $s, t$ )
if  $s$  and  $t$  cross at  $y' < y$  then
     $Q$ .insert(( $y'$ , intersection, ( $s, t$ )))
```

Die Event-Handler werden kümmern sich um die Handhabung der drei möglichen Event-Types.

```
HANDLEEVENT( $y, \text{intersection}, (a, b), T, Q$ )
output( $s \cap t$ )
 $T$ .swap( $a, b$ )
prev := pred( $b$ )
next := succ( $a$ )
findNewEvent(prev,  $b$ )
findNewEvent( $a$ , next)
```

```
HANDLEEVENT( $y, \text{start}, s, T, Q$ )
 $h := T$ .insert( $s$ )
prev := pred( $h$ )
next := succ( $h$ )
findNewEvent(prev,  $h$ )
findNewEvent( $h$ , next)
```

```
HANDLEEVENT( $y, \text{finish}, s, T, Q$ )
 $h := T$ .locate( $s$ )
prev := pred( $h$ )
next := succ( $h$ )
 $T$ .remove( $s$ )
findNewEvent(prev, next)
```

Nun können wir den Algorithmus implementieren.

```

 $T := ()$  SortedSequence of Segment
invariant  $T$  stores relative order of segments intersecting  $l$ 
 $Q := \text{MaxPriorityQueue}$ 
 $Q := Q \cup \{(\max\{y, y'\}, \text{start}, s) : s = \overline{(x, y)(x', y')} \in S\}$ 
 $Q := Q \cup \{(\min\{y, y'\}, \text{finish}, s) : s = \overline{(x, y)(x', y')} \in S\}$ 
while  $Q \neq \emptyset$  do
     $(y, \text{type}, s) := Q.\text{deleteMax}$ 
    handleEvent( $y, \text{type}, s, T, Q$ )

```

Dieser Algorithmus benötigt $O(n \log n)$ zur Initialisierung und $O((n + k) \log n)$ für die Event-Schleife, insgesamt also $O((n + k) \log n)$.

6.3 Konvexe Hülle

Wir werden uns in diesem Abschnitt mit dem folgenden Problem beschäftigen:

Gegeben sei eine Punktmenge $P = \{p_1, \dots, p_n\} \subset \mathbb{R}^2$. Gesucht ist ein konvexes Polygon C mit Eckpunkten $\in P$, sodass alle Punkte von P in C liegen.

Zuerst sortieren wir P lexikographisch. Das bedeutet, dass

$$p > q \Leftrightarrow p.x > q.x \vee (p.x = q.x \wedge p.y > q.y).$$

Wir berechnen ohne Einschränkung nur die obere Hülle, also die Hülle um die Punkte oberhalb von $\overline{p_1 p_n}$.

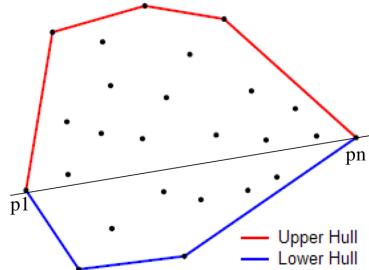


Abbildung 6.2. Obere Hülle.

Wir können beobachten, dass die obere Hülle ausschließlich Abbiegungen nach rechts macht (und die untere nur Abbiegungen nach links). Um damit arbeiten zu können müssen wir Abbiegungen definieren:

Definition 6.3.1 (Abbiegung). Für eine Punktmenge $P = \{p_1, \dots, p_n\}$ ist eine **Abbiegung nach rechts** an Stelle i vorhanden, falls p_{i+1} rechts von $\overline{p_{i-1} p_i}$ liegt.

Das konstruieren der oberen Hülle nennt sich auch **Graham's Scan**.¹

```
UPPERHULL( $p_1, \dots, p_n$ )
 $L := \langle p_n, p_1, p_2 \rangle$ : Stack of Point
invariant  $L$  is upper hull of  $\langle p_n, p_1, \dots, p_i \rangle$ 
for  $i := 3$  to  $n$  do
    while  $\neg$ rightTurn( $L$ .secondButLast,  $L$ .last,  $p_i$ ) do  $L$ .pop
     $L := L \circ \langle p_i \rangle$ 
return  $L$ 
```

Der Algorithmus selbst läuft in $O(n)$, weswegen das Sortieren dominiert und das ganze in $O(n \log n)$ liegt.

6.4 Kleinste einschließende Kugel

In diesem Abschnitt ist eine Punktmenge $P := \{p_1, \dots, p_n\} \subset \mathbb{R}^d$ gegeben und eine Kugel K mit minimalem Radius gesucht, sodass $P \subset K$. Wir verwenden einen Algorithmus in $O(n)$ nach Welzl.²

Q sei zu Beginn leer. Wir fügen Punkte derart zu Q hinzu, dass durch die Punkte in Q ein Ball aufgespannt wird, in dem alle $p \in P$ liegen.

```
sEB( $P, Q$ )
if  $|P| = 0 \vee |Q| = d + 1$  then return ball( $Q$ )
 $x := p \in P$  picked at random
 $B := \text{sEB}(P \setminus \{x\}, Q)$ 
if  $x \in B$  then return  $B$ 
return  $\text{sEB}(P \setminus \{x\}, Q \cup \{x\})$ 
```

6.5 Range Search

Beim **Range Search** (Bereichssuche) haben wir wieder eine Menge $P = \{p_1, \dots, p_n\} \subset \mathbb{R}^2$ gegeben. Wir erhalten als Anfrage nun ein achsenparalleles Rechteck

$$Q := [x, x'] \times [y, y'].$$

Gesucht ist nun entweder $P \cap Q$ (*range reporting*) oder $k := |P \cap Q|$ (*range counting*).

Wir werden range counting in $O(\log n)$ und range reporting in $O(k + \log n)$ lösen.

Dazu ist $O(n \log n)$ Vorverarbeitungszeit und $O(n)$ Platz notwendig.

¹ Graham 1972, Andrew 1979

² Welzl, 1991

Range Search im Eindimensionalen

Zuerst machen wir einen range search in einer Dimension. Dazu konstruieren wir im Voraus einen binären Suchbaum. Dieser codiert pro Blatt das größte Element des linken Teilbaums. So können die beiden Grenzen leicht gefunden werden.

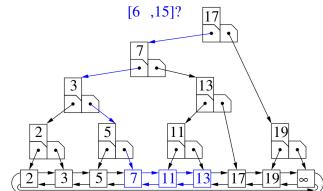


Abbildung 6.3. Ermittlung des ersten Elements, das größer als 6 ist.

Range Search im Zweidimensionalen – Erster Ansatz

Ein naives Verfahren ist es, die Punkte aus P in zwei Arrays A_x und A_y zu speichern, wobei die Punkte in A_x nach x -Wert und in A_y nach y -Wert sortiert sind.

Man bestimmt nun

$$k_x := \text{count}(x_0, x_1) \text{ in } A_x \quad (\text{alle Punkte mit } x_0 < x < x_1 \text{ und})$$

$$k_y := \text{count}(y_0, y_1) \text{ in } A_y \quad (\text{alle Punkte mit } y_0 < y < y_1).$$

Nun müssen noch $\min\{k_x, k_y\}$ Punkte gematcht werden, um zu überprüfen, dass der jeweilige Punkt sowohl im x - als auch im y -Wert im Rechteck liegt.

Insgesamt braucht dieser Ansatz also $O(\log n) + O(\min\{k_x, k_y\})$ Zeit, ist also nicht wirklich brauchbar.

Range Search im Zweidimensionalen – Zweiter Ansatz

Wir konstruieren nun einen balancierten Binärbaum mithilfe der x -Koordinaten. Anschließend berechnen wir die $O(\log n)$ Teilbäume, die zusammen alle Punkte mit $x_0 \leq x \leq x_1$ enthalten. Nun müssen diese nur noch nach y -Koordinate gefiltert werden. Wir speichern dazu in jedem Knoten die Punkte ab, die in seinem Teilbaum liegen, sortiert nach y -Wert.

Insgesamt können wir so den Algorithmus auf $O(\log^2 n + k)$ drücken.

Mithilfe von Wavelet Trees lässt sich die Zeit auf $O(\log n)$ reduzieren.

7

Online-Algorithmen

Inhalt dieses Kapitels:

- Einführung in Online-Algorithmen
- Beispiel: Job-Scheduling
- Beispiel: Skiausleihe
- Beispiel: Speicherverwaltung
- Beispiel: Auswahl von Experten

Online-Algorithmen werden verwendet, wenn Eingabegrößen nicht im Vornerein bekannt sind. Viele Algorithmen, die wir bisher diskutiert haben, benötigen Vorarbeitszeit, um optimale Ergebnisse liefern zu können, und sind daher nicht auf Probleme anwendbar, wo die Eingabe in serieller Natur vorliegt. Die bisher diskutierten Algorithmen werden daher auch **Offline-Algorithmen** genannt.

7.1 Übersicht

Wir werden uns im Folgenden Konzepte von Online-Algorithmen anhand von Beispielen anschauen. Zuerst definieren wir, was ein Online-Algorithmus formal ist.

- **Eingabe:** Folge von Anforderungen (*requests*)

$$\sigma = (r_1, \dots, r_n) \in R^n$$

- r_i muss auf Anforderung bearbeitet werden, es entsteht eine Antwort

$$a_i = g_i(r_1, \dots, r_i) \in A.$$

Es sind keine Informationen über die Zukunft verfügbar (r_{i+1}, \dots). Antworten sind unwiderruflich.

Der konkrete Algorithmus ist also durch g_1, \dots eindeutig festgelegt.

- **Kosten** $\text{cost}_n : R^n \times A^n \rightarrow \mathbb{R}_{>0}$
- **Ausgabe** für $\sigma \in R^n$ ist also

$$\text{alg}[\sigma] = (g_1(r_1), g_2(r_1, r_2), \dots, g_n(r_1, \dots, r_n)) \in A^n$$

und die Kosten

$$\text{alg}(\sigma) = \text{cost}_n(\sigma, \text{alg}[\sigma])$$

Kompetitive Analyse

Um einschätzen zu können, wie gut ein Online-Algorithmus ist, vergleichen wir ihn mit einem optimalen Offline-Algorithmus — das ist die **Kompetitive Analyse** dieses Online-Algorithmus.

Definition 7.1.1 (c-kompetitiv). Ein Online-Algorithmus ist für ein Optimierungsproblem auf Input σ **c-kompetitiv**, falls es einen Offline-Algorithmus OPT gibt, der nach einem Kostenmaß $\text{OPT}(\sigma)$ eine Optimallösung berechnen kann, sodass für den betrachteten Online-Algorithmus ALG und alle Eingabesequenzen σ gilt:

$$\text{ALG}(\sigma) \leq c \cdot \text{OPT}(\sigma) + \alpha.$$

Ist die zusätzliche Konstante $\alpha \leq 0$, dann heißt ALG **strikt c-kompetitiv**.

Unterschied ist, dass man bei nicht-strikter Kompetitivität Ausnahmen erlaubt.

Wir können nun den **Wettbewerbsfaktor** (*competitive ratio*) für den Algorithmus ALG definieren als

$$c_{\text{alg}} = \sup \left\{ \frac{\text{alg}(\sigma)}{\text{opt}(\sigma)} : \sigma \in R^+ \right\}.$$

Ist alg ein strikt c -kompetitiver Online-Algorithmus und

$$C = \{c : \text{alg ist strikt } c\text{-kompetitiv}\},$$

so ist

$$c_{\text{alg}} = \inf C \quad \text{und} \quad c_{\text{alg}} \in C.$$

Im Folgenden geben wir eine kurze Übersicht über die Beispiele, die wir im Folgenden behandeln werden.

Job-Scheduling

Dieses Beispiel ist dem Beispiel im Kapitel “Approximationsalgorithmen” sehr ähnlich.
Wir haben

- **Maschinen** M_1, \dots, M_m
- **Anfrage:** Job J_i , benötigt Zeit $t_i \geq 0$
- **Antwort:** Zuordnung von J_i zu M_j

Wieder stellt sich die Frage, wie sich der Makespan (also das Intervall zwischen Start und Fertigstellen der letzten Maschine) minimieren lässt. Diese Entscheidung muss hier für jedes J_i ohne Kenntnis über die zukünftigen Jobs passieren.

Skiausleihe

Die Situation ist hier folgende: Man befindet sich im Skीurlaub und solange das Wetter gut ist, lautet jeden Morgen die **Anforderung** “Ski ausleihen!”. Sobald das Wetter allerdings schlecht ist, lautet die **Anforderung** “Heimfahren!”.

Es sind zwei **Antworten** möglich:

- Ski für einen Tag ausleihen: Kosten k Euro
- Ski kaufen: Kosten K Euro ($K \gg k$)

Was muss nun getan werden, um die Gesamtkosten klein zu halten? Diese Entscheidung muss ohne Kenntnis des zukünftigen Wetters gemacht werden!

Speicherverwaltung

“Kosten minimieren” bedeutet im Speicherverwaltungskontext meist, die Anzahl an Cache Misses zu minimieren. Welche Verdränungsstrategie minimiert die Kosten hier am besten? Und wie kann man am besten zu verdrändende Seiten auswählen, ohne Kenntnis über zukünftige Anforderungen zu haben?

Auswahl von Experten

Hier gibt es mehrere Runden, jede läuft wie folgt ab:

1. Jeder von n Experten gibt zu einer Frage eine Ja/Nein-Empfehlung ab. Diese sind im Allgemeinen nicht richtig.
2. Man trifft seine eigene Ja/Nein-Entscheidung zu derselben Fragestellung.
3. Es wird mitgeteilt, welche Entscheidung richtig gewesen wäre.

Wie lässt sich hier die Anzahl an Fehlentscheidungen minimieren?

Selbstorganisierende Datenstrukturen

Hier haben wir eine einfach verkettete Liste und erhalten als **Anforderung** das Element x in der Liste. Die dabei entstehenden Kosten sind x . Als **Reaktion** kann entweder

- ohne weitere Kosten das angefragte Element weiter nach vorne gerückt werden, oder
- mit Kosten von 1 zwei aufeinanderfolgende Elemente vertauscht werden.

Welche Listen-Verwaltung minimiert hier die Kosten? Wie kann ohne Kenntnis über zukünftige Anforderungen sinnvoll umgeordnet werden?

7.2 Job-Scheduling

Wir nehmen den listScheduling-Approximationsalgorithmus und bauen ihn in einen Online-Algorithmus um:

```
LISTSCHEDULING( $n, m, t_1 \dots n$ )
each  $L_i := 0$  // load of machine  $1 \leq i \leq m$ 
each  $S_j := 0$  // machine for job  $1 \leq j \leq n$ 
for each  $j$  in range( $1, n$ ) do
    pick  $k$  from  $\{i : L_i \text{ is currently minimal}\}$ 
     $S_j := k$ 
     $L_k := L_k + t_j$ 
return  $S$ 
```

Frühere Analysen ergeben, dass der Wettbewerbsfaktor höchstens 2 ist. Der Online-Algorithmus sieht nun so aus:

```

LISTSCHEDULING( m      )
each  $L_i := 0$  // load of machine  $1 \leq i \leq m$ 

for each  $t_j$  in  $\sigma$  do
  pick  $k$  from  $\{i : L_i \text{ is currently minimal}\}$ 
   $a_j := k$ 
   $L_k := L_k + t_j$ 
  assign job to machine  $a_j$ 

```

7.3 Skiausleihe

Wir haben oben bereits diskutiert, was hier das Problem ist.

Die Optimalkosten sind offensichtlich

- Falls Urlaubsdauer $t \leq \frac{K}{k}$ Tage: t mal ausleihen \Rightarrow Kosten tk
- Falls Urlaubsdauer $t > \frac{K}{k}$ Tage: Ski sofort kaufen \Rightarrow Kosten K

Der Wettbewerbsfaktor ist hier 2. Optimal ist es, an Tag $\frac{K}{k}$ die Ski zu kaufen.

7.4 Speicherverwaltung

Wir werden uns die folgenden Speicherverwaltungsprobleme anschauen:

- *Offline* (lfd ist optimal)
- *Deterministisch* (bestenfalls k -kompetitiv)
- *Deterministisch*: lru ist k -kompetitiv
- *Resource Augmentation*: (h, k) -Seitenwechsel
- *Randomisiert* (randMark ist $2H_k$ -kompetitiv)

Wir betrachten hier stets einen Speicher der Größe K und einen Cache der Größe k ($K \gg k$).

Offline – lfd ist optimal

lfd (*longest forward distance*) verdrängt den Eintrag, der am weitesten in der Zukunft benötigt wird. Dieser Algorithmus ist offensichtlich optimal aber auch offensichtlich nicht möglich.

3	1	5	3	2	4	1
1	4	2				
3	1	5	3	2	4	1
1	3	2				

Abbildung 7.1. Eingabe und Cache vor und nach Bearbeitung der ersten Anforderung. Es wird die 4 ersetzt, weil sie erst am weitesten in der Zukunft wieder benötigt wird.

Deterministisch — bestenfalls k -kompetitiv

Die vier üblichen Algorithmen sind

- fifo (*first in first out*),
- lifo (*last in first out*),
- lru (*least recently used*),
- lfu (*least frequently used*).

lifo und lfu sind nicht kompetitiv, lru und fifo sind k -kompetitiv. In der Praxis wird üblicherweise lru verwendet.

Es gilt folgender Satz:

Satz 7.4.1. Jeder deterministische Online-Algorithmus für das Seitenwechselproblem mit Cachegröße k hat einen Wettbewerbsfaktor $c \geq k$. k ist also eine untere Schranke für den Wettbewerbsfaktor.

Deterministisch: lru ist k -kompetitiv

Wir können zeigen, dass lru k -kompetitiv ist.

Resource Augmentation

Wir betrachten das (h, k) -Seitenwechselproblem: hier wird der Online-Algorithmus alg_k mit der Cachegröße k mit lfd_h mit Cachegröße $h < k$ verglichen.

Wir benötigen folgende Definition:

Definition 7.4.2. Ein Online-Algorithmus heißt **konservativ**, falls er bei Anderungsfolgen mit höchstens k verschiedenen Seiten höchstens k Cache-Misses hat. Beispiele hierfür sind lru und fifo.

Es gilt:

Jeder konservative Online-Algorithmus ist $\frac{k}{k - h + 1}$ -kompetitiv.

Randomisiert: randMark ist $2H_k$ -kompetitiv

Bei einem randomisierten Online-Algorithmus zur Speicherverwaltung ist die Zahl der Cache-Misses eine *Zufallsvariable*

$$f_R(r_1, \dots, r_n).$$

Hier sind gegebenenfalls andere Sichtweisen auf c -kompetitivität sinnvoll.

Zur Analyse verwenden wir sogenannte **Widersacher**. Diese bekommen als Eingabe die gewünschte Länge n und R und erzeugen eine "schlimme" Anforderungsfolge der Länge n . Diese müssen sie aber auch selbst verarbeiten.

Wir unterscheiden folgende Widersachertypen:

- **unwissender Widersacher** W (*oblivious adversary*): kein Wissen über erzeugte Zufallsbits, erzeugt für (R, n) immer gleiches (r_1, \dots, r_n) .
- **adaptiver Widersacher** W' (*adaptive adversary*): arbeitet gegen eine konkrete Abarbeitung von R , kennt die von R bei der Abarbeitung von (r_1, \dots, r_i) erzeugten Zufallsbits und folglich auch immer den aktuellen Cache-Zustand von R .

Wir werden im Folgenden nur unwissende Widersacher analysieren und vergleichen sie mit $\text{opt}(r_1, \dots, r_n)$.

Definition 7.4.3. R ist c -kompetitiv gegen unwissende Widersacher, wenn es ein von n unabhängiges b gibt, sodass für jede Anforderungsfolge (r_1, \dots, r_n) gilt:

$$\mathbb{E}[f_R(r_1, \dots, r_n)] - c \cdot \text{opt}(r_1, \dots, r_n) \leq b.$$

Wir betrachten nun den randMark -Algorithmus.¹

```
(Cache: cache[i], Markierungsbits mark[i], 1 ≤ i ≤ k)
for i := 1 to k do mark[i] := 0 // alle Markierungen auf 0
while ∃ weitere Anforderungen do
    r := nächste Anforderung
    if memory[r] ∉ cache then
        if ∀mark[i] ≡ 1 then ∀mark[i] := 0 // alles 1 ~ neue Phase ~ alles 0
        i := zufälliges j mit mark[j] ≡ 0
        cache[i] := memory[r]
    else i := Index mit cache[i] ≡ memory[r]
    mark[i] := 1
```

Dieser Algorithmus ist $2H_k$ -kompetitiv gegen unwissende Widersacher.

¹ Fiat et al., 1991

7.5 Auswahl von Experten

Es gibt mehrere Runden mit folgender Struktur:

1. Jeder von n Experten gibt zu einer Frage eine Ja/Nein-Empfehlung ab. Diese sind im Allgemeinen nicht richtig.
2. Man trifft seine eigene Ja/Nein-Entscheidung zu derselben Fragestellung.
3. Es wird mitgeteilt, welche Entscheidung richtig gewesen wäre.

Ziel ist:

- **Anforderung:** k -Tupel aus Ja/Nein-Empfehlungen der Experten
- **Antwort:** Eigene Ja/Nein-Entscheidung
- **Kosten:** Anzahl eigener Fehlentscheidungen
- **Ziel:** Die eigenen Kosten sollen möglichst nah an den Kosten der besten Experten liegen

Es ist leicht einzusehen, dass immer die Antwort des Experten zu wählen, der bisher am meisten korrekte Antworten produziert hat, schlecht ist — man kann eine Expertenmenge derart konstruieren, dass diese Strategie immer irrt.

Weighted Majority Algorithm

Der **Weighted Majority Algorithm** weist zu Beginn jedem Experte i ein Gewicht $w_i := 1$ zu.

In jeder Runde wird die eigene Entscheidung nun folgendermaßen konstruiert:

1. Experten geben Empfehlungen $x_i \in \{\text{ja}, \text{nein}\}$
2. Eigene Entscheidung fällen:

$$\begin{cases} \text{ja}, & \text{falls } \sum_{i, x_i \equiv \text{ja}} w_i \geq \sum_{i, x_i \equiv \text{nein}} w_i \\ \text{nein}, & \text{falls } \sum_{i, x_i \equiv \text{ja}} w_i < \sum_{i, x_i \equiv \text{nein}} w_i \end{cases}$$

3. Expertengewichte anpassen:

$$w_i^{t+1} = \begin{cases} w_i^t, & \text{falls } x_i \text{ richtig war} \\ \frac{w_i^t}{2}, & \text{falls } x_i \text{ falsch war} \end{cases}$$

wma ist $\frac{1}{\log_2(4/3)}$ -kompetitiv. Genauer:

$$\text{wma}(\sigma) \leq \frac{1}{\log_2(4/3)} (\text{optExpert}(\sigma) + \log_2 k).$$

Verallgemeinerung

Wir betrachten nun nicht mehr nur den Fall, dass Antworten richtig oder falsch sein können, sondern gewichten sie mit Faktoren zwischen 0 und 1. In der t -ten Runde läuft ab:

1. **Experten:** geben Empfehlungen x_1, \dots, x_k
2. Eigene Wahl der Empfehlung
3. Mitteilung, welche Entscheidung richtig gewesen wäre
4. Beurteilung der Empfehlungen durch "Noten" $c_i^t \in [0, 1]$
5. **Kosten:** Note der gewählten Empfehlung

Wir verwenden hierfür einen randomisierten Algorithmus $\text{randWMA}_\varepsilon$:

- Initial:
 - Es sei $\varepsilon \in (0, \frac{1}{2})$
 - Experte i hat Gewicht w_i
 - alle $w_i := 1$
- In jeder Runde t :
 1. wähle Empfehlung von Experte i mit Wahrscheinlichkeit

$$p_i = \frac{w_i}{\sum_1^k w_i}$$

2. Benotung
3. setze jedes $w_i := w_i(1 - \varepsilon c_i^t)$

Ist k die Anzahl der Experten und $0 < \varepsilon < \frac{1}{2}$ und $\text{randWMA}_\varepsilon$ nach einer Anzahl an Runden Gesamtkosten von K hat und die besten Experten Empfehlungen mit (minimalen) Gesamtkosten K_{opt} gegeben haben, so ist

$$K \leq (1 + \varepsilon) \cdot K_{\text{opt}} + \frac{\ln n}{\varepsilon}.$$

8

Parallele Algorithmen

8.1 Einleitung

Indem man Parallelverarbeitung verwendet, kann man sowohl Ressourcen einsparen als auch Ressourcenrestriktionen brechen:

- **Zeitersparnis:** Arbeiten p Computer an einem Problem, so sind sie bis zu p mal so schnell.
- **Kommunikationsersparnis:** Fallen Daten verteilt an, so kann man sie auch verteilt (vor)verarbeiten.
- **Energieersparnis:** Zwei Prozessoren mit halber Taktfrequenz brauchen weniger Energie als ein voll getakteter Prozessor.
- **Speicherbeschränkung:** Mehr Prozessoren haben mehr Hauptspeicher, mehr Cache,...

Es gibt sehr viele Modelle der Parallelverarbeitung, wir werden hier allerdings nur zwei Standardmodelle diskutieren:

- **Prozessornetzwerke mit Nachrichtenkopplung.**

Prozessoren sind hier “normale CPUs” mit lokalen Speicher. Gemeinsam mit seinem lokalen Speicher wird eine lokale CPU auch *processing element* (PE) genannt. Der Datenaustausch passiert über ein Netzwerk in Form von Nachrichten zwischen zwei Prozessoren.

- **Parallele Registermaschinen mit Speicherkopplung.**

Auch hier wird mit normalen CPUs gearbeitet, allerdings haben diese keinen lokalen Speicher, sondern sind über ein Netzwerk mit Speichermodulen verbunden, auf welche alle CPUs zugreifen können. Der Datenaustausch erfolgt über das Netzwerk zwischen einer CPU und einem Speicher.

Auf nachrichtengekoppelte Rechner werden wir genauer eingehen.

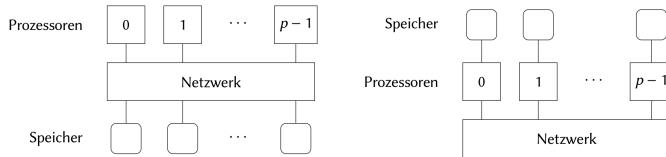


Abbildung 8.1. Vergleich zwischen *parallelen Registermaschinen mit Speicherkopplung* (links) und einem *Prozessornetzwerk mit Nachrichtenkopplung* (rechts).

Nachrichtenkopplung vs. Speicherkopplung

Neben den oben erläuterten Vorteilen von Parallelverarbeitung haben beide Modelle auch Nachteile:

- **Nachrichtenkopplung:**

- Zwei Prozessoren werden zum Datentransport benötigt. Das passt dem Empfänger nicht immer.
- Parallelismus muss explizit programmiert werden.

- **Speicherkopplung:**

- *Skalierbarkeit*: Ist eine große Anzahl an Prozessoren sinnvoll?
- *Kostenmaß* bei Speicherzugriffskonflikten?

Als eine gute Strategie hat es sich erwiesen, den Entwurf für einen verteilten Speicher durchzuführen, da dieser einen viel breiteren Bereich abdecken kann. Die Implementierung erfolgt dann gegebenenfalls für einen gemeinsamen Speicher.

8.2 Nachrichtengekoppelte Parallelrechner

Modell

- **Netzwerk**: Vollständig verknüpftes Punkt-zu-Punkt-Netzwerk
 - voll-duplex
 - Nachrichten überholen sich nicht

- **Prozessoren:** können jeweils maximal gleichzeitig

- eine Nachricht an einen beliebigen Empfänger senden (`send(smsg,to)`)
- eine Nachricht von einem beliebigen Sender empfangen (`rmsg := recv(from)`)
- *oder* beides gleichzeitig (`rmsg := sendRecv(smsg, to, from)`)

Als *Kostenmodell* für das Senden oder Empfangen von l Bytes verwenden wir

$$T_{\text{comm}}(l) = T_{\text{start}} + l \cdot T_{\text{byte}},$$

wobei in der Praxis meist $T_{\text{byte}} \ll T_{\text{start}}$. Ignoriert wird hier unter anderem der “Abstand” zwischen Sender und Empfänger.

Als *Programmiermodell* verwenden wir **SPMD** (*single program multiple data*). Alle PEs führen hier dasselbe Programm aus, unterschieden wird lediglich durch “Ränge” der PEs (paarweise verschiedene PE-Nummern).

Parallele Reduktion

Im Folgenden gehen wir über die grundlegenden Werkzeuge, die wir benötigen, um parallele Programme analysieren zu können.

Definition 8.2.1 (Reduktion). Sei \otimes eine binäre, assoziative Operation auf einer Menge M .

Für $x = (x_0, \dots, x_{p-1}) \in M$ definieren wir

$$R_{\otimes}(x) = \bigotimes_{i < p} x_i = x_0 \otimes \cdots \otimes x_{p-1}.$$

Nun gilt folgender Satz:

Satz 8.2.2. Wenn \otimes eine binäre, assoziative Operation ist und p Elemente x_0, \dots, x_{p-1} auf p PEs verteilt sind, dann kann man $\bigotimes_{i < p} x_i$ in Zeit $O(\log p)$ auf PE 0 berechnen.

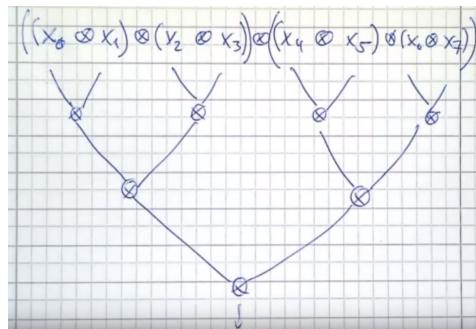


Abbildung 8.2. Idee hinter obigem Satz.

Sequenziell wäre die optimale Laufzeit $O(n)$ gewesen. Auf $p = n$ PEs hätte man nach obigem Satz eine Laufzeit von $O(\log n)$ erreicht, also eine Beschleunigung von $O\left(\frac{n}{\log n}\right)$. "Ideal" wäre eine Beschleunigung von $O(p) = O(n)$ gewesen.

Wie kann man die Beschleunigung noch weiter verbessern?

Parallele Reduktion mit $p < n$

Verwenden wir nun $p < n$ viele PEs, um eine Reduktion auf n Elementen durchzuführen, so erhält jedes PE n/p Datenelemente. Die PEs berechnen zuerst die Reduktion der lokalen Elemente, und anschließend wird auf diesen Reduktionen eine parallele Reduktion durchgeführt. Laufzeit hierfür ist $O(n/p) + O(\log p)$ und die Beschleunigung somit

$$\frac{O(n)}{O(n/p + \log p)}.$$

Ist $p \in O(n/\log n)$, so ist die Beschleunigung $O(p)$.

Man kann also durch Verringerung der Prozessorzahl p die Beschleunigung in die Nähe von p bringen. Dieses Prinzip nennt man **Brent's Prinzip**.

Analyse paralleler Programme

Wir interessieren uns in erster Linie für

- Laufzeit,
- Beschleunigung und
- verrichtete Arbeit.

Es ist

- $T_{\text{par}}(I, p)$ die parallele Laufzeit der Probleminstanz I , bearbeitet mit p PEs,
- $T_{\text{seq}}(I)$ die sequenzielle Laufzeit der Probleminstanz I mit dem “besten bekannten Algorithmus”.

Im Allgemeinen ist $T_{\text{seq}}(I) < T_{\text{par}}(I, 1)$. Man erhält aber leicht Gleichheit, indem man den parallelen Algorithmus einfach den sequenziellen ausführen lässt, falls nur ein PE zur Verfügung steht.

Definition 8.2.3 (Speedup). Wir definieren den **Speedup** als

$$S(I, p) = \frac{T_{\text{seq}}(I)}{T_{\text{par}}(I, p)}.$$

Vergröbert für alle Probleminstanzen der Größe n :

$$S(n, p) = \inf \{S(I, p) : n = |I|\}.$$

Zur Berechnung des Speedups können wir praktischerweise folgende Spezialfälle benutzen, die für Instanzen I, I' gleicher Größe n gelten:

- $T_{\text{par}}(I, p) = T_{\text{par}}(I', p) = T_{\text{par}}(n, p)$,
- $T_{\text{seq}}(I) = T_{\text{seq}}(I') = T_{\text{seq}}(n)$

Simuliert man p Prozessoren durch einen Prozessor, so sehen wir, dass ein sequenzieller Algorithmus nie langsamer als $O(p \cdot T_{\text{par}}(I, p))$ sein kann, weswegen immer $\frac{T_{\text{seq}}(I)}{T_{\text{par}}(I, p)} \in O(p)$ ist und $S(n, p) \in O(p)$ ebenfalls.

Definition 8.2.4 (Effizienz). Wir definieren die **Effizienz** eines parallelen Algorithmus als

$$E(n, p) := \frac{S(n, p)}{p}.$$

Es ist $S(n, p) \in O(p)$ und daher $E(n, p) \in O(1)$. Tatsächlich ist sogar $E(n, p) > 1$ möglich (sogenannter *superlinearer Speedup*).

Es ist

$$E(n, p) \in \frac{T_{\text{par}}(n, p)}{p \cdot T_{\text{seq}}(n)}.$$

Definition 8.2.5 (Arbeit). Wir definieren **Arbeit** als

$$W(n, p) := p \cdot T_{\text{par}}(n, p).$$

Mit obiger Definition gilt

$$E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{seq}}(n)}{T_{\text{par}}(n, p) \cdot p} = \frac{T_{\text{seq}}(n)}{W(n, p)}.$$

Wir können nun die parallele Reduktion mit $p < n$ von vorhin genauer analysieren:

- Es werden zuerst $\frac{n}{p}$ Elemente sequenziell und anschließend p partielle Summen reduziert.
- **Laufzeit:** $T_{\text{par}}(n, p) = O(n/p) + O(\log p)$
- **Beschleunigung:** $S(n, p) = \frac{O(n)}{O(n/p + \log p)}$
- **Effizienz:** $E(n, p) = \frac{O(n)}{O(n + p \log p)}$

Wir betrachten noch die zwei Sonderfälle:

- $p = n$:
 - $S(n, n) \in O\left(\frac{n}{\log n}\right)$
 - $E(n, n) \in O\left(\frac{1}{\log n}\right)$
- $p \in O\left(\frac{n}{\log n}\right)$:
 - $S(n, p) = O(p)$
 - $E(n, p) = O(1)$

Wir erkennen hier die größere Effizienz für kleinere p (nach Brents Prinzip).

Parallel Präfixsummen

Wir verwenden \otimes wie oben definiert. Wir definieren nun

$$P_{\otimes}(x) = y = (y_0, \dots, y_{p-1})$$

$$\text{mit } y_i = \bigotimes_{k \leq i} x_k.$$

Es ist also $y_0 = x_0$ und $y_{i+1} = y_i \otimes x_{i+1}$.

Präfixsummen – Hyperwürfel-Algorithmus

Wir machen es uns hier einfach und legen fest, dass $p = n = 2^d$ (für $d \in \mathbb{N}$) und \otimes kommutativ.

Jede PE erhält nun eine “Koordinate” $0 \leq i \leq 2^d - 1$. Diese wird als Bitvektor

$$i = (i_{d-1} \dots i_0) \quad \text{mit} \quad i_j \in \{0, 1\}$$

repräsentiert. Hier ist i_k das k -te Bit von rechts in i .

Wir erlauben Kommunikation zwischen zwei PE i und i' nur, wenn die Hammingdistanz ihrer Bitvektoren 1 ist, sie sich also nur in einer Stelle unterscheiden. Wir erhalten so einen *Hyperwürfel* aus PEs.

Wir berechnen nun Präfixsummen auf einem solchen Hyperwürfel:

```
PREFIXSUM(x, ⊗)
y := x // auf PE i liegt x_i
s := x // für Summe von Elementen in Unterwürfeln
for k := 0 to d - 1 do
    s' := SENDRECV(s, i ⊕ 2k, i ⊕ 2k)
    s := s ⊗ s'
if ik ≡ 1 then y := y ⊗ s'
// auf PE i liegt yi = x0 ⊗ ⋯ ⊗ xi
```

Wir erhalten eine Laufzeit

$$T_{\text{prefix}} \in O((T_{\text{start}} + l \cdot T_{\text{byte}}) \cdot \log p).$$

Diese Laufzeit ist nicht optimal. Wie man sie optimieren kann wird in der Vorlesung "Parallele Algorithmen" näher erläutert.

Paralleles Sortieren

Hier gibt es zwei verschiedene Aufgabenvarianten:

1. Alle n Elemente liegen zu Beginn auf PE 0. Deswegen muss jedes Element von PE 0 mindestens einmal angefasst werden. Die Laufzeit ist daher in $\Omega(n)$.
2. Je n/p Elemente liegen zu Beginn auf PE i . Dieser Fall ist wesentlich interessanter.

Zunächst betrachten wir den einfachen Fall $p = n$ (Prozessor i hat Eingabeelement x_i). Wir behalten die Grundidee von Quicksort bei:

- Wir wählen ein Element pv als Pivot.
- Elemente werden umverteilt:
 - kleiner als pv: auf Prozessoren mit kleineren Rängen
 - größer als pv: auf Prozessoren mit großen Rängen
- Parallele Rekursion.

Wir schauen uns den **Theoretiker-Quicksort** an:

```

// Teil 0: Vorbereitungen
THEOQSORT0(x)
i := commRank()
p := commSize() // hier hat PE i Element xi
theoQSort(x, 0, p - 1)

// Teil 1: kleine Elemente zählen
THEOQSORT(x, ...)
if i ≡ 0 then ipv := randInt(0, p)
ipv := bcast(ipv, 0)
pv := bcast(x, ipv)
small := (x ≤ pv)
j := prefixSum(small, +)
p' := bcast(j, p - 1)

// Teil 2: Datenumverteilung und Rekursion
if small ≡ 1 then send(x, j - 1)
else send(x, p' + i - j)
x := recv(any)
recursive theoQSort of “left”/“right” part

```

Die erwartete Rekursionstiefe ist in $O(\log p)$, die Zeit jeweils in $O(T_{\text{start}} \log p)$. Insgesamt ist die erwartete Zeit also in $O(T_{\text{start}} (\log p)^2)$.

9

Fortgeschrittene Datenstrukturen

Wir werden uns in diesem Kapitel mit Prioritätslisten beschäftigen. Es gibt noch viele weitere fortgeschrittene Datenstrukturen, z.B.

- monotone ganzzahlige Prioritätslisten (später im Kapitel “kürzeste Wege”)
- perfektes Hashing
- Suchbäume mit fortgeschrittenen Operationen
- externe Prioritätslisten (später im Kapitel “Externe Algorithmen”)
- Geometrische Datenstrukturen (siehe Kapitel “Geometrische Algorithmen”)

9.1 Adressierbare Prioritätslisten

Eine **adressierbare Prioritätsliste** muss folgende Funktionen implementieren:

BUILD ($\{e_1, \dots, e_n\}$)	$M := \{e_1, \dots, e_n\}$
SIZE	return $ M $
INSERT (e)	$M := M \cup \{e\}$
MIN	return $\min M$
DELETEMIN	$e := \min; M := M \setminus \{e\}; \text{return } e$
REMOVE ($h : \text{Handle}$)	$e := h; M := M \setminus \{e\}; \text{return } e$
DECREASEKEY ($h : \text{Handle}, k : \text{Key}$)	$\text{key}(h) := k$
MERGE (M')	$M := M \cup M'$

Adressierbare Prioritätslisten haben viele Anwendungen, beispielsweise im *Dijkstra-Algorithmus* für kürzeste Wege oder in der Graphpartitionierung. Allgemein lassen sich adressierbare Prioritätslisten gut bei Greedy-Algorithmen verwenden, bei denen sich die Prioritäten (begrenzt) ändern.

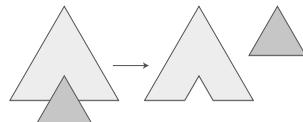
Datenstruktur

Als grundlegende Datenstruktur wird ein Wald heap-geordneter Bäume verwendet. Hier wird also der Binary Heap verallgemeinert

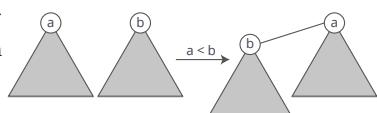
- Baum \rightarrow Wald
- zwei Kindknoten \rightarrow beliebig viele Kindknoten

Wir verwenden folgende grundlegende Operationen zur Bearbeitung solcher Wälder:

- **cut**: Teilbaum ausschneiden und als neuen Baum speichern



- **link**: Baum 2 mit größerem Wurzelknoten als Baum 1 an Baum 1 als Kindknoten des Wurzelknotens anhängen



- **union**: $\text{union}(a, b) = \text{link}(\min(a, b), \max(a, b))$

Dijkstras Algorithmus

Dijkstras Algorithmus kann die Distanz zwischen einem Startknoten s und jedem anderen Knoten des Graphen berechnen.

```

DIJKSTRA(s : Node, T : Tree)
// Initialisieren: Distanz zu jedem Knoten ist  $\infty$ , zu Startknoten 0.
d =  $\langle \infty, \dots, \infty \rangle$ 
d[s] = 0
// Startknoten zu PQ hinzufügen.
Q.insert(s)

```

- $d = \langle \infty, \dots, \infty \rangle$

Zu Beginn ist die Distanz zu jedem Knoten ∞ .

- $\text{parent}[s] = s, d(s) = 0$

Der Startknoten wird initialisiert.

- $Q.insert(s)$

Prioritätsliste wird mit Startknoten initialisiert.

9.2 Pairing Heaps

Pairing Heaps müssen folgende Funktionen implementieren:

```

INSERTITEM(h : Handle)
newTree(h)

NEWTREE(h : Handle)
forest := forest  $\cup \{h\}$ 
if *h < min then minPtr := h

DECREASEKEY(h : Handle, k : Key)
key(h) := k
if h not a root then cut(h) else updateMinPtr(h)

DELETEMIN() : Handle
m := minPtr
forest := forest \ {m}
foreach child h of m do newTree(h)
pairwiseRootUnion()
updateMinPtr()
return m

PAIRWISEROOTUNION()
// see picture

MERGE(o : AdressablePQ)
if *minPtr > *(o.minPtr) then minPtr := o.minPtr
forest := forest  $\cup o.\text{forest}$ 
o.forest :=  $\emptyset$ 

```

Einige Funktionalitäten lassen sich gut veranschaulichen:

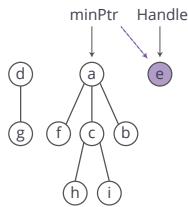


Abbildung 9.1. newTree: Element e wird hinzugefügt (ggf. auch ein ganzer Baum) und — falls nötig — der minPtr angepasst.

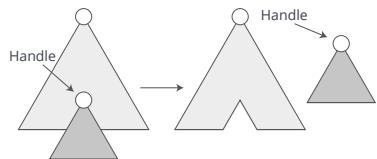


Abbildung 9.2. decreaseKey: Durch Herabsetzen des Keys wird eventuell die Heap-Eigenschaft verletzt, deswegen wird der Teilbaum ausgeschnitten und als neuer Baum hinzugefügt.

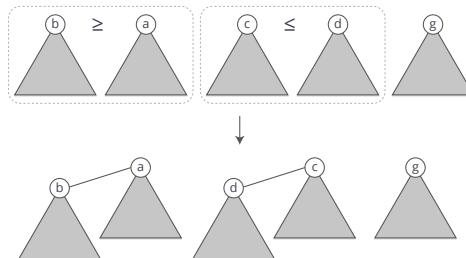


Abbildung 9.3. pairwiseRootUnion fügt jeweils zwei Bäume des Waldes zu einem größeren Baum zusammen, indem die beiden Wurzeln miteinander verglichen werden und eine der beiden Wurzeln Kindknoten der anderen wird.

Repräsentation

Meistens speichert man Pairing Heaps als doppelt verkettete Liste der Wurzeln. Die Baum-Items können beispielsweise folgendermaßen gespeichert werden:

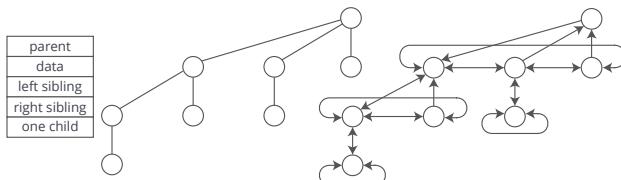


Abbildung 9.4. Speichern der Baum-Items. Man kann hier noch Speicherplatz einsparen, indem man left sibling und parent zusammenfasst. Allerdings muss man dann alle Geschwisterknoten traversieren, damit man zum Elternknoten kommen kann, da nur der linkeste Kindknoten den Elternknoten speichert.

Analyse

- `insert` und `deleteMin` gehen in $O(1)$.
- `deleteMin` und `remove` gehen jeweils in $O(\log n)$ amortisiert.
- `decreaseKey` ist schwieriger zu analysieren, geht aber amortisiert in $O(\log \log n) \leq T \leq O(\log n)$ und ist in der Praxis sehr schnell.

Wir werden als nächstes *Fibonacci-Heaps* verwenden, um noch mehr Leistung rauszukitzeln.

9.3 Fibonacci-Heaps

Mithilfe von Fibonacci-Heaps erhalten wir eine amortisierte Komplexität von $O(\log n)$ für `deleteMin` und $O(1)$ für alle anderen Operationen.

Fibonacci-Heaps speichern ein paar Zusatzinformationen pro Knoten ab, wodurch neue Hilfsfunktionen kreiert werden können, die diese Beschleunigung ermöglichen:

- **Rank** eines Knotens: Anzahl direkter Kinder
- **Mark**: Knoten, die ein Kind verloren haben, werden markiert
- **Vereinigung nach Rank**: Union nur für gleichrangige Wurzeln
- **Kaskadierende Schnitte**: Knoten, die beide markiert sind (also ein Kind verloren haben), werden geschnitten

Repräsentation

Die Repräsentation ist analog zu der von Pairing Heaps, die Wurzeln werden wieder als doppelt verkettete Liste gespeichert und die Baum-Items als Parameterliste:

parent
data, rank, mark
left sibling
right sibling
one child

Funktionalität

`insert` und `merge` werden wie gehabt implementiert. `decreaseKey` verwendet die neue `cascadingCut`-Methode und `deleteMin` Union-by-Rank. Wir beschleunigen Union-by-Rank, indem wir ein Feld pro Rank bereitstellen und Knoten in diese

Felder eintragen. Sollte das passende Feld bereits belegt sein, so wird der neue Knoten mit dem Knoten, der sich im Feld befindet gelinkt und entsprechend verschoben.

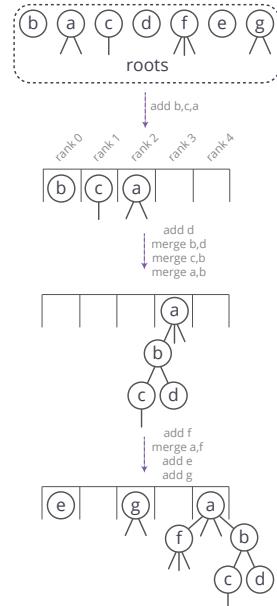
```

DELETEMIN() : Handle
    m := minPtr
    forest := forest \ {m}
    foreach child h of m do newTree(h)
    while ∃ a, b ∈ forest : rank(a) ≡ rank(b) do
        union(a, b)
        updateMinPtr()
    return m

DECREASEKEY(h : Handle, k : Key)
    key(h) := k
    cascadingCut(h)

CASCADINGCUTh : Handle
    assert h is not a root
    p := parent(h)
    unmark(h)
    cut(h)
    if p is marked then
        cascadingCut(p)
    else mark(p)

```



Eine amortisierte Analyse von `deleteMin` ergibt $\Omega(\log n)$ für vergleichsbasiertes `deleteMin`.

Abbildung 9.5. Fast Union-by-Rank.

10

Kürzeste Wege

Wir betrachten einen Graph $G = (V, E)$ mit Kantengewicht $c : E \rightarrow \mathbb{R}$ und Anfangsknoten $s \in V$.

Gesucht ist die Länge $\mu(v)$ des **kürzesten Pfades** von s nach v für alle $v \in V$, wobei

$$\mu(v) := \min \{c(p) : p \text{ ist Pfad von } s \text{ nach } v\}$$

und

$$c(\langle e_1, \dots, e_k \rangle) := \sum_{i=1}^k c(e_i).$$

Oft suchen wir auch eine “geeignete” Repräsentation des kürzesten Pfades.

10.1 Allgemeine Definitionen

Wir benutzen im Allgemeinen zwei Knotenarrays:

- $d[v]$ = aktuelle (= vorläufige) Distanz von s nach v .
Invariante: $d[v] \geq \mu(v)$.
- $\text{parent}[v]$ = Vorgänger von v auf (vorläufigem) kürzesten Pfad von s nach v .
Invariante: Dieser Pfad bezeugt $d[v]$.

Initial ist

- $d[s] = 0$, $\text{parent}[s] = s$ und
- $d[v] = \infty$, $\text{parent}[v] = \perp$.

Kern ist das *Relaxieren* der Kanten $(u, v) \in E$:

```
if  $d[u] + c(u, v) < d[v]$  then // z.B. wenn  $d[v] \equiv \infty$ 
     $d[v] := d[u] + c(u, v)$ 
    parent[v] := u
```

Die oben genannten Invarianten werden dadurch nicht verletzt. $d[v]$ kann sich also problemlos mehrmals ändern.

10.2 Dijkstras Algorithmus

Dijkstras Algorithmus ist der wohl einfachste Algorithmus, um dieses Problem zu lösen. In Pseudocode:

```
// d und parent initialisieren
// alle Knoten als ungescannt setzen
while  $\exists$  non-scanned node  $u$  with  $d[u] < \infty$  do
     $u :=$  non-scanned node  $v$  with minimal  $d[v]$ 
    relax all edges  $(u, v)$  out of  $u$ 
    set  $u$  scanned
```

Ist $v \in V$ von s aus erreichbar, so wird v irgendwann gescannt. Wird v gescannt, so ist $\mu(v) = d[v]$.

Am Ende definiert d die optimalen Entfernung und parent die zugehörigen Wege. Dieser Algorithmus wurde bereits in Algorithmen I ausführlich diskutiert.

Analyse

Es ist

$$T_{\text{Dijkstra}} = O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))).$$

Nutzen wir also Fibonacci-Heaps, so kriegen wir

$$T_{\text{DijkstraFib}} = O(m + n \log n).$$

10.3 Monotone ganzzahlige Prioritätslisten

Wir beobachten, dass Dijkstras Algorithmus die Prioritätsliste *monoton* benutzt — *insert* und *decreaseKey* benutzen nämlich Distanzen der Form $d[u] + c(e)$. Die Werte nehmen also ständig zu.

Sind alle Kantengewichte $\in [0, C]$, so gilt

$$\forall v \in V : d[v] \leq (n - 1)C.$$

Ist insbesondere \min der letzte Wert, der aus Q entfernt wurde, so sind in Q *immer* nur Knoten mit Distanzen im Intervall $[\min, \min + C]$.

Bucket-Queue

Eine **Bucket-Queue** ist ein zyklisches Array B von $C + 1$ doppelt verketteten Listen. Knoten der Distanz $d[v]$ werden in $B[d[v] \bmod (C + 1)]$ gespeichert.

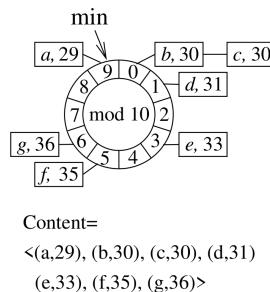


Abbildung 10.1. Bucket-Queue mit $C = 9$.

Folgende Operationen werden auf einer Bucket Queue implementiert:

- **Initialisierung:** $C + 1$ leere Listen werden angelegt, $\min = 0$.
- **insert(v):** fügt v in $B[d(v) \bmod (C + 1)]$ ein
 $\Rightarrow O(1)$
- **decreaseKey(v):** schiebt v von seiner Liste nach $B[d(v) \bmod (C + 1)]$
 $\Rightarrow O(1)$
- **deleteMin:** fängt bei $B[\min \bmod (C + 1)]$ an; falls leer, $\min := \min + 1$, \circlearrowleft
 $\Rightarrow O(nC)$

Mit Bucket-Queues kriegen wir also den Dijkstra-Algorithmus auf $O(m + \maxPathLength)$ gedrückt.

Radix-Heaps

Radix-Heaps sind eine Variante von Bucket Queues, die Buckets von -1 bis K für $K = 1 + \lfloor \log C \rfloor$ benutzt. Wie vorhin schon betrachtet sei \min die zuletzt aus Q entfernte Distanz und somit

$$\forall v \in Q : d[v] \in [\min, \dots, \min + C].$$

Wir betrachten die *binäre Repräsentation* der möglichen Distanzen in Q . Wir speichern v in Bucket $B[i]$, falls sich $d[v]$ und \min zuerst an der i -ten Stelle unterscheiden (Ausnahmen: $B[K]$ falls $i > K$, $B[-1]$ falls sie sich nicht unterscheiden).

Das definiert die **Most Significant Digit** (kurz MSD) — das ist die Position der höchstwertigen Ziffer in der Binärdarstellung von a und b , an der sich die beiden unterscheiden. $\text{msd}(a, b)$ kann mit Maschinenbefehlen sehr schnell berechnet werden.

Wir nutzen folgende *Radix-Heap-Invariante*:

v ist gespeichert in Bucket $B[i]$, wo $i = \min\{\text{msd}(\min, d[v]), K\}$

Wir können nun `deleteMin` folgendermaßen implementieren:

```

DELETEMIN(): Element
if  $B[-1] = \emptyset$  then
     $i := \min\{j \in 0 \dots K : B[j] \neq \emptyset\}$ 
    move min  $B[i]$  to  $B[-1]$  and to  $\min$ 
    foreach  $e \in B[i]$  do // exactly here the invariant is violated!
        move  $e$  to  $B[\min\{\text{msd}(\min, d[v]), K\}]$ 
return  $B[-1].\text{popFront}()$ 

```

Die `deleteMin`-Laufzeit ist $O(K)$, insgesamt lässt sich also die Laufzeit des Dijkstra-Algorithmus hierdurch auf $O(m + n \log C)$ drücken.

10.4 All-Pairs Shortest Paths

Herausforderung in diesem Abschnitt ist es, nicht die Abstände zu einem festgelegten Startknoten zu berechnen, sondern zwischen allen Knotenpaaren $(u, v) \in V^2$ für $G = (V, E)$. Zusätzlich erlauben wir negative Kantenkosten, allerdings keine negativen Kreise.

Wir werden zwei verschiedene Lösungen erhalten:

1. n mal den *Bellman-Ford-Algorithmus* ausführen
 $\Rightarrow O(n^2 m)$
2. *Knotenpotentiale* verwenden
 $\Rightarrow O(nm + n^2 \log n)$

Bellman-Ford-Algorithmus

Der **Bellman-Ford-Algorithmus** wurde bereits in Algorithmen I behandelt, deswegen hier nur eine kurze Wiederholung. Wie Dijkstras Algorithmus findet er den kürzesten Pfad zwischen einem festgelegten Startknoten und allen anderen Knoten des Graphen, unterstützt aber auch negative Kantengewichte.

1. Distanzen initialisieren: $d[s] = 0, d[v] = \infty$ für alle anderen Knoten
2. Von s ausgehend alle Knoten des Graphen durchgehen und pro Knoten v die ausgehenden Kanten betrachten.
 - Eintragen, ob v von s in $< \infty$ erreicht werden kann.
 - Ist $d[v] + c(v, u) < d[u]$ für einen Nachbar von v ? Wenn ja, $d[u]$ aktualisieren.

Der zweite Schritt wird maximal $|V| - 1$ mal wiederholt. Sollte sich schon davor bei einem Durchgang nichts mehr ändern, so kann man aufhören.

Die Laufzeit des Bellman-Ford-Algorithmus ist $O(nm)$, nutzt man ihn als Basis für All-Pairs Shortest Paths kriegt man also $O(n \cdot nm) = O(n^2 m)$.

Knotenpotentiale

Jeder Knoten erhält ein Potential $\text{pot}(v)$. Mit diesen Knotenpotentialen lassen sich die **reduzierten Kosten** $\bar{c}(e)$ für eine Kante $e = (u, v) \in E$ als

$$\bar{c}(e) = \text{pot}(u) + c(e) - \text{pot}(v)$$

definieren.

Ist p ein Pfad von u nach v mit Kosten $c(p)$, dann ist

$$\bar{c}(p) = \text{pot}(u) + c(p) - \text{pot}(v).$$

Ist p' ein anderer u - v -Pfad, dann gilt

$$c(p) \leq c(p') \Leftrightarrow \bar{c}(p) \leq \bar{c}(p').$$

Wir berechnen die gewünschten Informationen nun so:

1. Wir fügen einen *Hilfsknoten* s zu G hinzu.
2. Wir fügen (s, v) für alle $v \in V \setminus \{s\}$ mit Kosten 0 hinzu.
3. Berechne die kürzesten Pfade von s aus mit Bellman-Ford.
4. Definiere $\text{pot}(v) := \mu(v)$ für alle $v \in V$.

Die reduzierten Kosten sind jetzt alle nicht-negativ, also können wir Dijkstra benutzen und ggf. s wieder entfernen.

5. Für eine beliebige Kante $(u, v) \in E$ gilt

$$\mu(u) + c(e) \geq \mu(v) \text{ und deshalb } \bar{c}(e) = \mu(u) + c(e) - \mu(v) \geq 0.$$

```

neuen Knoten  $s$  und alle Kanten  $s, v$  hinzufügen //  $O(n)$ 
pot :=  $\mu := \text{BELLMANFORDSSSP}(s, c)$  //  $O(nm)$ 
foreach  $x \in V$  do
     $\bar{\mu}(x, \cdot) := \text{DIJKSTRASSSP}(x, \bar{c})$ 

// zurück zur ursprünglichen Kostenfunktion
foreach  $e = (v, w) \in V^2$  do //  $O(n^2)$ 
     $\mu(v, w) := \bar{\mu}(v, w) + \text{pot}(w) - \text{pot}(v)$ 

```

Die Gesamlaufzeit beträgt also $O(nm + n^2 \log n)$.

10.5 Distanz zu Zielknoten

Wir haben bisher zwei Fälle diskutiert:

1. Abstände aller Knoten zu einem Startknoten s
2. Abstände zwischen allen Knotenpaaren $\{u, v\} \in V^2$

Als nächstes schauen wir uns an, wie man den kürzesten Pfad zwischen einem Startknoten s und einem Zielknoten t ermittelt.

Trick 0 – Dijkstra abbrechen

Am einfachsten ist es, einfach Dijkstra abzubrechen, wenn t aus Q entfernt wird. Das spart “im Schnitt” die Hälfte des Scans.

Bidirektionale Suche

Idee ist hier, abwechselnd von s und t aus zu suchen. Von s aus sucht man auf $G = (V, E)$, von t aus auf dem zugehörigen Rückwärtsgraphen $G^r = (V, E^r)$.

Die vorläufige kürzeste Distanz wird in jedem Schritt gespeichert:

$$d[s, t] = \min \{d[s, t], d_{\text{forward}}[u] + d_{\text{backward}}[u]\}$$

Abgebrochen wird, wenn die Suche einen Knoten scannt, der in die andere Richtung bereits gescannt wurde.

A^{}-Suche*

Idee der A^* -Suche ist es, “in die Richtung” des Ziels zu suchen. Dazu benötigen wir eine Funktion $f(v)$, die für alle $v \in V$ die eigentliche Funktion $\mu(v, t)$ schätzen kann.

Anschließend können wir $\text{pot}(v) = f(v)$ setzen und $\bar{c}(u, v) = c(u, v) + f(v) - f(u)$.

$f(v)$ muss diese Eigenschaften haben:

- **Konsistenz:** $c(e) + f(v) \geq f(u)$ ($\forall e = (u, v)$).

Die reduzierten Kosten dürfen also nicht negativ sein.

- $f(v) \leq \mu(v, t)$ ($\forall v \in V$).

Dann ist $f(t) = 0$ und wir können aufhören wenn t aus Q entfernt wird.

Ist p ein beliebiger Pfad von s nach t , so ist $d[t] \leq c(p)$ (alle Kanten auf p seien relativiert). Wie finden wir jetzt aber so eine Funktion $f(v)$?

Wir benötigen eine Heuristik für $f(v)$.

- Betrachtet man eine Strecke im Straßennetzwerk, so kann $f(v)$ beispielsweise der euklidische Abstand $\|v - t\|_2$ sein. Damit erhält man eine deutliche, aber keine überragende Beschleunigung.

- **Landmarks** sind deutlich geeigneter, benötigen allerdings Vorberechnung.

Man wähle eine *Landmarkmenge* L . Berechne und speichere $\mu(v, l)$ für alle $l \in L, v \in V$.

Während einer Query suche man jetzt ein Landmark $l \in L$ "hinter" dem Ziel und benutze die untere Schranke

$$f_l(v) = \mu(v, l) - \mu(t, l)$$

Vorteile sind, dass Landmarks konzeptuell einfach sind, eine erhebliche Beschleunigung bringen (um den Faktor 20) und mit anderen Techniken kombinierbar sind. *Allerdings* ist die Landmarkauswahl schwierig und der Platzverbrauch sehr groß (besonders für große V).

11

Anwendungen von DFS

11.1 Starke Zusammenhangskomponenten

Zusammenhangskomponenten in einem ungerichteten Graph sind Teilgraphen, in denen es zwischen je zwei beliebigen Knoten einen Pfad gibt.

In gerichteten Graphen sind **starke Zusammenhangskomponenten** Teilgraphen $G \subseteq H$, in denen ebenfalls gilt: für jedes Knotenpaar $u, v \in G$ gibt es einen u - v -Pfad *und* einen v - u -Pfad.

Insbesondere werden starke Zusammenhangskomponenten durch Zyklen erzeugt (dann kann man einfach im Kreis laufen von einem Knoten zum anderen). Das bedeutet im Umkehrschluss, dass der **Schrumpfgraph** – das ist der Graph, den man erhält, indem man jede starke Zusammenhangskomponente als einen Knoten zusammenfasst – zyklusfrei ist.

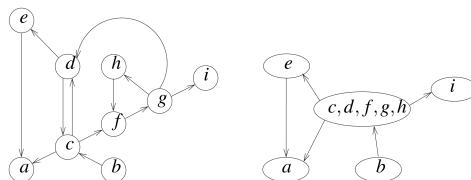


Abbildung 11.1. Gerichteter Graph G und zugehöriger Schrumpfgraph G_S .

Tiefensuchschema

Um später SCCs ermitteln zu können konstruieren wir ein Tiefensuchschema:

```

unmark all nodes
init()
foreach  $s \in V$  do
    if  $s$  is not marked then
        mark  $s$ 
        root( $s$ )
        DFS( $s, s$ )

DFS( $u, v : \text{Node}$ )
    foreach  $(v, w) \in E$  do
        if  $w$  is marked then
            traverseNonTreeEdge( $v, w$ )
        else
            traverseTreeEdge( $v, w$ )
            mark  $w$ 
            DFS( $v, w$ )
    backtrack( $u, v$ )

```

Dieses Tiefensuchschema kann auf unterschiedliche Graphtraversierungsprobleme angepasst werden.

Wir verwenden nun zwei Arrays zum Zwischenspeichern unserer Resultate:

- `oNodes` speichert die bereits besuchten Knoten,
- `oReps` speichert die Repräsentanten der einzelnen SCCs.

Beim Durchlaufen des Graphen werden die Knoten mit `dfsNum` inkrementell durchnummeriert.

Außerdem gibt es drei Invarianten, die wir im Folgenden nicht verletzen dürfen:

1. Kanten von abgeschlossenen Knoten gehen zu abgeschlossenen Knoten
2. Offene Komponenten S_1, \dots, S_k bilden einen Pfad in G_C^s .
3. Repräsentanten partitionieren die offenen Komponenten bezüglich ihrer `dfsNum`.

Für das Finden von SCCs brauchen wir folgende Implementierungen für die rot gekennzeichneten Prozessen:

- `root(s)`:

```

oReps.push( $s$ )
oNodes.push( $s$ )

```

Hierdurch wird eine neue offene Komponente gebildet und s als besucht gekennzeichnet.

- **traverseTreeEdge(v, w):**

```
oReps.push(w)
oNodes.push(w)
```

Hier wird $\{w\}$ als neue offene Komponente angelegt.

- **traverseNonTreeEdge(v, w):**

```
if  $w \in oNodes$  then
    while  $w.dfsNum < oReps.top.dfsNum$  do oReps.pop
```

Ist $w \notin oNodes$ ist w abgeschlossen und die Kante somit uninteressant. Ist w allerdings in $oNodes$, so werden die auf dem Kreis befindlichen SCCs kollabiert.

- **backtrack(u, v):**

```
if  $v \equiv oReps.top$  then
    oReps.pop
repeat
     $w := oNodes.pop$ 
    component[w] := v
until  $w = v$ 
```

Damit haben wir alles was wir brauchen, um die Suche nach SCCs durchführen zu können. Wir kriegen sie sogar in $O(m + n)$, also in Linearzeit, hin!

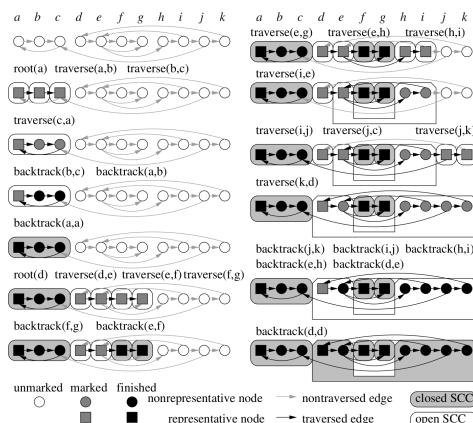


Abbildung 11.2. Kompletter Durchlauf des Algorithmus.