Entwurf und Analyse von Algorithmen
WS 1999/2000

Prof. Dorothea Wagner

Ausarbeitung:

Dagmar Handke
Matthias Weisgerber
Anja Kristina Koch

Version vom 19. Januar 2001

Inhaltsverzeichnis

Kapitel 1

Divide-and-Conquer Verfahren und das Prinzip der Rekursion

Der Literaturtip. Ein gutes Buch unter vielen zum Gesamtthema der Vorlesung ist [CLR90]. Es enthält auch einführende Kapitel zum Algorithmusbegriff sowie zur Laufzeitanalyse von Algorithmen und zu Wachstumsraten von Funktionen (O, Ω, Θ). Ein weiteres empfehlenswertes Buch ist [OW93]. Für das Kapitel „Divide-and-Conquer Verfahren und das Prinzip der Rekursion“ speziell empfehlen wir [CLR90].

Ein grundlegendes algorithmisches Prinzip besteht darin, ein Problem zu lösen, indem man es in Probleme (meist desselben Typs) kleinerer Größe oder mit kleineren Werten aufteilt, diese löst und aus den Lösungen eine Lösung für das Ausgangsproblem konstruiert.

1.1 Ein Beispiel aus der Informatik-Grundvorlesung

Merge Sort ist ein Verfahren zum Sortieren einer Folge von n Werten, auf denen es eine Ordnung $\leq$ gibt. Merge Sort sortiert eine Folge der Länge n, indem es zunächst halb so lange Teilfolgen sortiert und aus diesen die sortierte Folge der Länge n „zusammmenmischt“. Dies kann, wie im Folgenden beschrieben, auf iterative oder auf rekursive Weise durchgeführt werden.

1.1.1 Iterativ bzw. bottom-up

Iteratives Merge-Sort

1. sortiere zunächst Teilfolgen der Länge zwei und mische sie zu sortierten Folgen der Länge vier zusammen;
2. mische je zwei sortierte Folgen der Länge vier zu sortierten Folgen der Länge acht zusammen;
3. u.s.w.

1.1.2 Rekursiv

Sehr oft werden Divide-and-Conquer Verfahren jedoch rekursiv angewandt. Das bedeutet: Das Verfahren ruft sich selbst wieder auf, angewandt auf Eingaben kleinerer Länge bzw. mit kleineren Werten.

Merge Sort **rekursiv formal.**

Eingabe: Array A mit n Elementen, die an den Stellen $A[p]$ bis $A[r]$ stehen.

Ausgabe: n Elemente sortiert in Ergebnisarray B.

```
Merge Sort$(A; p, r)$
  1. Falls $p < r$ gilt, dann setze $q := \lfloor \frac{p+r}{2} \rfloor$
  2. $\left.\begin{array}{l}\text{Merge Sort}(A; p, q)\\ \text{Merge Sort}(A; q+1, r)\end{array}\right\}$ rekursive Aufrufe
  3. $B :=$ Merge$(A; p, q, r)$.
```

Die Hauptarbeit besteht hier bei Merge Sort, wie auch bei den meisten Divide-and-Conquer Verfahren, im „Zusammensetzen“ der Teillösungen, bei Merge Sort also in Merge.

Merge **informell:** Durchlaufe A einerseits von $A[p]$ bis $A[q]$ und andererseits von $A[q+1]$ bis $A[r]$ und vergleiche die Elemente jeweils paarweise. Das kleinere der Elemente wird „weggeschrieben“ (in ein Ergebnisarray) und in dem entsprechenden Teil von A um eine Position weitergegangen. Ist man bei $A[q]$ oder $A[r]$ angelangt, wird der restliche Teil des anderen Teilarrays von A an das Ergebnisarray angehängt. Die Einträge aus dem Ergebnisarray werden an die Stellen $A[p]$ bis $A[q]$ kopiert.

Wir haben gezeigt, daß Merge eine Laufzeit hat, die linear in der Länge der Eingabe ist, also in $\mathcal{O}(n)$ ist. Aus der rekursiven Beschreibung für Merge Sort können wir als Laufzeit $T(n)$ insgesamt ablesen:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \underbrace{k \cdot n}_{\text{für Merge}} \quad \text{mit } k \geq 0 \text{ Konstante.}$$

Solche Rekursionsgleichungen sind typisch für rekursiv beschriebene Divide-and-Conquer-Algorithmen. Wie schätzt man eine Laufzeitfunktion, die als Rekursionsgleichung (bzw. -ungleichung) gegeben ist, möglichst gut ab?

Zur Erinnerung: Bei MERGE SORT ist $T(n) \in \mathcal{O}(n \cdot \log n)$. Der Beweis wird induktiv geführt, vgl. Informatik II: Algorithmen und Datenstrukturen.

1.2 Methoden zur Analyse von Rekursionsabschätzungen

Substitutionsmethode: Wir vermuten eine Lösung und beweisen deren Korrektheit induktiv.

Iterationsmethode: Die Rekursionsabschätzung wird in eine Summe umgewandelt, und dann mittels Techniken zur Abschätzung von Summen aufgelöst.

Meistermethode: Man beweist einen allgemeinen Satz zur Abschätzung von rekursiven Ausdrücken der Form

$$T(n) = a \cdot T(n/b) + f(n), \quad \text{wobei } a \geq 1 \text{ und } b > 1.$$

Technische Details. Normalerweise ist die Laufzeitfunktion eines Algorithmus nur für ganze Zahlen definiert. Entsprechend steht in einer Rekursionsabschätzung eigentlich $T(\lfloor n/b \rfloor)$ oder $T(\lceil n/b \rceil)$. Außerdem haben Laufzeitfunktionen $T(n)$ die Eigenschaft, daß zwar $T(n) \in \mathcal{O}(1)$ ist für kleine n, allerdings oft mit großer $\mathcal{O}$-Konstante. Meistens kann man diese Details jedoch vernachlässigen.

1.2.1 Die Substitutionsmethode

Beispiel.

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n.$$

Dies ist etwa die Laufzeitfunktion von MERGE SORT, von der wir wissen, daß sie in $\mathcal{O}(n \log n)$ liegt.

Wir beweisen, daß $T(n) \leq c \cdot n \cdot \log n$ für geeignetes $c > 0$, c konstant ist. Dazu nehmen wir als Induktionsvoraussetzung an, daß die Abschätzung für Werte kleiner n gilt, also insbesondere

$$\begin{aligned} T(\lfloor n/2 \rfloor) &\leq c \cdot \lfloor n/2 \rfloor \cdot \log(\lfloor n/2 \rfloor) \text{ und} \\ T(\lceil n/2 \rceil) &\leq c \cdot \lceil n/2 \rceil \cdot \log(\lceil n/2 \rceil). \end{aligned}$$

Im Induktionsschritt erhalten wir also:

$$\begin{aligned}T(n) &\leq c \cdot \lfloor n/2 \rfloor \cdot \log \lfloor n/2 \rfloor + c \cdot \lceil n/2 \rceil \cdot \log \lceil n/2 \rceil + n\\ &\leq c \cdot \lfloor n/2 \rfloor \cdot (\log n - 1) + c \cdot \lceil n/2 \rceil \cdot \log n + n\\ &\leq c \cdot n \cdot \log n - c \cdot \lfloor n/2 \rfloor + n\\ &\leq c \cdot n \log n \ \text{ für } \ c \geq 3, n \geq 2.\end{aligned}$$

Für den Induktionsanfang muß natürlich noch die Randbedingung bewiesen werden, z.B. für $T(1) = 1$. Dies ist oft problematisch. Es gilt beispielsweise für kein $c > 0$, daß $T(1) = 1 \leq c \cdot 1 \cdot \log 1 = 0$ ist. Da uns bei Laufzeitfunktionen allerdings asymptotische Abschätzungen genügen, d.h. Abschätzung für $n \geq n_0$, können wir auch mit der Randbedingung $T(4)$ starten, d.h.

$$T(4) = 2 \cdot T(2) + 4 = 4 \cdot T(1) + 8 = 12 \leq c \cdot 4 \cdot \log 4 = c \cdot 8 \ \text{ für } \ c \geq 2.$$

Wie kommt man an eine gute Vermutung? Es gibt keine allgemeine Regel für die Substitutionsannahme, aber einige heuristische „Kochrezepte“. Lautet die Rekursionsgleichung etwa

$$T(n) = 2 \cdot T(n/2 + 17) + n,$$

so unterscheidet sich die Lösung vermutlich nicht substantiell von der Lösung für obige Rekursion, da die Addition von 17 im Argument von T für hinreichend große n nicht so erheblich sein kann. In der Tat ist hier wieder $T(n) \in \mathcal{O}(n \log n)$, und dies kann wieder mit Induktion bewiesen werden (vgl. Übung).

Manchmal läßt sich zwar die korrekte Lösung leicht vermuten, aber nicht ohne weiteres beweisen. So ist vermutlich

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1 \in \mathcal{O}(n).$$

Versucht man jedoch zu beweisen, daß $T(n) \leq c \cdot n$ für geeignetes $c > 0$ ist, so erhält man im Induktionsschritt

$$\begin{aligned}T(n) &\leq c \cdot \lfloor n/2 \rfloor + c \cdot \lceil n/2 \rceil + 1\\ &= c \cdot n + 1,\end{aligned}$$

aber kein $c > 0$ erfüllt $T(n) \leq c \cdot n$.

Trick. Starte mit der schärferen Vermutung $T(n) \leq c \cdot n - b$ für $c > 0$ und einer geeigneten Konstanten $b \geq 0$. Dann gilt

$$\begin{aligned} T(n) &\leq c \cdot \lfloor n/2 \rfloor - b + c \cdot \lceil n/2 \rceil - b + 1 \\ &\leq c \cdot n - 2b + 1 \\ &\leq c \cdot n - b \text{ für } b \geq 1. \end{aligned}$$

Ein weiterer Trick, der oft funktioniert, ist die Variablenersetzung.

Beispiel.

$$T(n) = 2 \cdot T(\lfloor \sqrt{n} \rfloor) + \log n.$$

Setze $m = \log n$, also $n = 2^m$, dann ergibt sich

$$T(2^m) = 2 \cdot T(\lfloor 2^{m/2} \rfloor) + m \leq 2 \cdot T(2^{m/2}) + m.$$

Setzt man nun $S(m) := T(2^m)$, so gilt

$$S(m) \leq 2 \cdot S(m/2) + m \in \mathcal{O}(m \log m).$$

Rückübersetzung von $S(m)$ nach $T(m)$ ergibt dann

$$T(n) = T(2^m) = S(m) \in \mathcal{O}(m \log m) = \mathcal{O}(\log n \log \log n).$$

1.2.2 Die Iterationsmethode

Eine naheliegende Methode zur Auflösung einer Rekursionsgleichung ist deren iterative Auflösung.

Beispiel.

$$\begin{aligned} T(n) &= 3 \cdot T(\lfloor n/4 \rfloor) + n \\ &= n + 3 \cdot (\lfloor n/4 \rfloor + 3 \cdot T(\lfloor n/16 \rfloor)) \\ &= n + 3 \cdot (\lfloor n/4 \rfloor + 3 \cdot (\lfloor n/16 \rfloor + 3 \cdot T(\lfloor n/64 \rfloor))) \\ &= n + 3 \cdot \lfloor n/4 \rfloor + 9 \cdot \lfloor n/16 \rfloor + 27 \cdot T(\lfloor n/64 \rfloor). \end{aligned}$$

Wie weit muß man die Rekursion iterativ auflösen, bis man die Randbedingungen erreicht?

Der i-te Term ist $3^i \cdot \lfloor n/4^i \rfloor$. Die Iteration erreicht im Argument 1, wenn $\lfloor n/4^i \rfloor \le 1$, d.h. wenn $i \ge \log_4 n$. Dann ergibt sich

$$\begin{aligned}
T(n) &\le n + 3 \cdot n/4 + 9 \cdot n/16 + 27 \cdot n/64 + \ldots + 3^{\log_4 n} \cdot c_1 \\
&\ \text{für } c_1 \ge 0 \text{ konstant} \\
&\le n \cdot \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + c_1 \cdot n^{\log_4 3}, \quad \text{da } 3^{\log_4 n} = n^{\log_4 3} \\
&= 4n + c_1 \cdot n^{\log_4 3} \in \mathcal{O}(n), \quad \text{da } \log_4 3 < 1.
\end{aligned}$$

Die Iterationsmethode führt oft zu aufwendigen (nicht umbedingt trivialen) Rechnungen. Durch iterative Auflösung einer Rekursionsabschätzung kann man jedoch manchmal zu einer guten Vermutung für die Substitutionsmethode gelangen.

1.3 Der Aufteilungs-Beschleunigungssatz

Es gibt einen sehr allgemeinen Satz über das asymptotische Wachstum rekursiv beschriebener Laufzeitfunktionen. Wir werden eine vereinfachte Form des Satzes ausführlich beweisen. Der Beweis des allgemeinen Satzes geht im Prinzip genauso; er ist nur technisch aufwendiger.

Satz 1.1 (allgemeine Form) *Seien $a \ge 1$ und $b > 1$ Konstanten, $f(n)$ eine Funktion in n und $T(n)$ über nichtnegative ganze Zahlen definiert durch*

$$T(n) = a \cdot T(n/b) + f(n),$$

wobei n/b für $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$ steht.

Dann gilt

(i) $T(n) \in \Theta(n^{\log_b a})$, falls $f(n) \in \mathcal{O}(n^{\log_b a - \varepsilon})$ für eine Konstante $\varepsilon > 0$.

(ii) $T(n) \in \Theta(n^{\log_b a} \cdot \log n)$, falls $f(n) \in \Theta(n^{\log_b a})$.

(iii) $T(n) \in \Theta(f(n))$, falls $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ für eine Konstante $\varepsilon > 0$ und $a \cdot f(n/b) \le c \cdot f(n)$ für eine Konstante $c < 1$ und für $n \ge n_0$.

Beispiel. MERGE SORT

$$b = 2, \quad a = 2, \quad f(n) \in \Theta(n) = \Theta(n^{\log_2 2})$$

Aus Fall (ii) folgt $T(n) \in \Theta(n \cdot \log n)$.

Wir schränken den Satz auf den Fall ein, daß $f(n) \in \mathcal{O}(n)$ ist, und beweisen ihn nur für Potenzen von b, d.h. $n = b^q$. Letztere Einschränkung wird aus technischen Gründen vorgenommen; man kann für $n \neq b^q$ die Analyse unter Betrachtung der nächsten Potenzen von b, d.h. $n_2 = b^{q+1}$ für $n_1 = b^q < n < n_2$ durchführen. Die Einschränkung auf $f(n) \in \mathcal{O}(n)$ vereinfacht den Beweis insofern, daß man nicht ausführlich $f(n)$ gegenüber $a \cdot T(n/b)$ abschätzen muß.

Satz 1.2 (eingeschränkte Form) *Seien* $a \geq 1, b > 1, c_1 > 0$ *und* $c_2 > 0$ *Konstanten und* $T(n)$ *über nichtnegative ganze Zahlen definiert durch*

$$\begin{aligned} c_1 \leq T(1) &\leq c_2 \text{ und} \\ a \cdot T(n/b) + c_1 n \leq T(n) &\leq a \cdot T(n/b) + c_2 \cdot n. \end{aligned}$$

Für $n = b^q$ *gilt*

(i) $T(n) \in \Theta(n)$, *falls* $b > a$.

(ii) $T(n) \in \Theta(n \cdot \log n)$, *falls* $a = b$.

(iii) $T(n) \in \Theta(n^{\log_b a})$, *falls* $b < a$.

Beweis. Durch Induktion über q beweisen wir daß

$$T(n) \leq c_2 \cdot n \cdot \sum_{i=0}^{q} (a/b)^i.$$

Für $q = 0$ ergibt sich $T(1) \leq c_2$. Die Behauptung gelte also für $q > 0$. Betrachte $q + 1$:

$$\begin{aligned} T(b^{q+1}) &\leq a \cdot T(b^q) + c_2 \cdot b^{q+1} \\ &\leq a \cdot \Big(c_2 \cdot b^q \cdot \sum_{i=0}^{q} (a/b)^i\Big) + c_2 \cdot b^{q+1} \\ &= c_2 \cdot b^{q+1} \cdot \Big(\sum_{i=0}^{q} (a/b)^{i+1} + 1\Big) \\ &= c_2 \cdot b^{q+1} \cdot \Big(\sum_{i=1}^{q+1} (a/b)^i + 1\Big) \\ &= c_2 \cdot b^{q+1} \cdot \sum_{i=0}^{q+1} (a/b)^i. \end{aligned}$$

Analog läßt sich auch folgern

$$T(n) \geq c_1 \cdot n \cdot \sum_{i=0}^{q} (a/b)^i.$$

Fall $b > a$. Dann ist $a/b < 1$ und es gibt Konstante $k_1, k_2 > 0$, so daß

$$c_1 \cdot n \cdot k_1 \leq T(n) \leq c_2 \cdot n \cdot k_2, \quad \text{d.h. } T(n) \in \Theta(n).$$

Fall $b = a$.

$$\begin{aligned} T(n) &= T(b^q) \leq c_2 \cdot b^q \cdot \sum_{i=0}^{q} 1^i = c_2 \cdot b^q \cdot (q+1) \text{ und} \\ T(n) &\geq c_1 \cdot b^q \cdot (q+1), \text{ also } T(n) \in \Theta(n \log n). \end{aligned}$$

Fall $b < a$.

$$\begin{aligned} T(n) &= T(b^q) \leq c_2 \cdot b^q \cdot \sum_{i=0}^{q} (a/b)^i = c_2 \cdot \sum_{i=0}^{q} a^i \cdot b^{q-i} \\ &= c_2 \cdot \sum_{i=0}^{q} a^{q-i} \cdot b^i = c_2 \cdot a^q \cdot \sum_{i=0}^{q} (b/a)^i. \end{aligned}$$

Dann gibt es Konstante $k_1, k_2 > 0$, so daß

$$c_1 \cdot \underbrace{a^{\log_b n}}_{=n^{\log_b a}} \cdot k_1 \leq T(n) \leq c_2 \cdot \underbrace{a^{\log_b n}}_{=n^{\log_b a}} \cdot k_2$$

und damit $T(n) \in \Theta(n^{\log_b a})$.

■

Bemerkung. Gelten in der Voraussetzung von Satz 1.2 nur die Beschränkungen nach oben (unten), so gilt immer noch $T(n) \in \mathcal{O}(n)$ (bzw. $T(n) \in \Omega\,(n)$).

1.3.1 Beispiel: Matrix-Multiplikation nach Strassen

(Informatik II, Übung)

Problem. Gegeben zwei $(n \times n)$-Matrizen A, B. Berechne $A \cdot B$ mit möglichst wenig Operationen.

Herkömmlich: $C = A \cdot B = (a_{ij}) \cdot (b_{ij}) = (c_{ij})$ mit $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$. Die Laufzeit ist insgesamt in $\mathcal{O}(n^3)$.

Ein Divide-and-Conquer Ansatz für die Matrixmultiplikation ist:

$$A \cdot B = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} = \begin{pmatrix} c_1 & c_2 \\ c_3 & c_4 \end{pmatrix} = C,$$

wobei die a_i, b_i, c_i $(n/2 \times n/2)$-Matrizen sind. Dann berechnet sich C durch

$$\left.\begin{array}{l} c_1 = a_1 b_1 + a_2 b_3 \\ c_2 = a_1 b_2 + a_2 b_4 \\ c_3 = a_3 b_1 + a_4 b_3 \\ c_4 = a_3 b_2 + a_4 b_4 \end{array}\right\} \begin{array}{c} \text{8 Matrixmultiplikationen} \\ \text{von } (n/2 \times n/2)\text{-Matrizen und} \\ \text{4 Additionen von } (n/2 \times n/2)\text{-Matrizen} \end{array}$$

Addition zweier $(n \times n)$-Matrizen ist in $\Theta(n^2)$. Damit ergibt sich

$$\begin{array}{rcl} T(n) & = & 8 \cdot T(n/2) + c \cdot n^2, \quad c > 0 \text{ Konstante}, \\ c \cdot n^2 & \in & \mathcal{O}(n^{\log_2 8 - \varepsilon}) = \mathcal{O}(n^{3-\varepsilon}) \text{ mit } 0 < \varepsilon = 1 \\ \Longrightarrow T(n) & \in & \Theta(n^{\log_2 8}) = \Theta(n^3) \quad \text{(leider!)} \end{array}$$

Für die Laufzeit ist offensichtlich die Anzahl der Multiplikationen „verantwortlich“. Das heißt: Wenn man die Anzahl der Multiplikationen der $(n/2 \times n/2)$-Matrizen auf weniger als 8 reduzieren könnte bei ordnungsmäßig gleicher Anzahl von Additionen erhält man eine bessere Laufzeit.

Es gibt einen Divide-and-Conquer Ansatz von Strassen, der nur 7 Multiplikationen bei 18 Additionen verwendet. Als Laufzeit ergibt sich dann

$$\begin{array}{rcl} T(n) & = & 7 \cdot T(n/2) + c \cdot n^2 \\ \Longrightarrow T(n) & \in & \Theta(n^{\log_2 7}). \end{array}$$

Schema:

$$\left.\begin{array}{l} c_1 = P_5 + P_4 - P_2 + P_6 \\ c_2 = P_1 + P_2 \\ c_3 = P_3 + P_4 \\ c_4 = P_5 + P_1 - P_3 - P_7 \end{array}\right\} \text{ mit } \left\{\begin{array}{l} P_1 = a_1 \cdot (b_2 - b_4) \\ P_2 = (a_1 + a_2) \cdot b_4 \\ P_3 = (a_3 + a_4) \cdot b_1 \\ P_4 = a_4 \cdot (b_3 - b_1) \\ P_5 = (a_1 + a_4) \cdot (b_1 + b_4) \\ P_6 = (a_2 - a_4) \cdot (b_2 + b_4) \\ P_7 = (a_1 - a_3) \cdot (b_1 + b_3) \end{array}\right.$$

1.3.2 Beispiel: Das Auswahlproblem (Selection)

Gegeben. Eine Folge von n Elementen, auf denen es eine Ordnung $\leq$ gibt, und $i \in \{1, \ldots, n\}$.

Problem. Gib das i-te (i-kleinste) Element aus.

Wir nehmen an, daß alle Elemente verschieden sind. (Falls gleiche Elemente auftreten, müßte zwischen diesen eine künstliche Ordnung festgelegt werden.) Dann ist x das i-te Element genau dann, wenn x größer als genau $i-1$ Elemente der Folge ist. Ein einfacher Sortier-Algorithmus macht folgendes:

1. Sortiere die Elemente in aufsteigender Reihenfolge und gib das i-te der sortierten Folge aus.

Mit MERGE SORT beispielsweise ergibt sich eine Laufzeit in $\Theta(n \log n)$. Ein Verfahren mit linearer worst-case-Laufzeit ist SELECT(n, i):

1. Teile die n Elemente in $\lfloor n/5 \rfloor$ Gruppen mit jeweils 5 Elementen, und einer weiteren Gruppe mit den restlichen n mod 5 Elementen auf.
2. Bestimme aus jeder der $\lceil n/5 \rceil$ Gruppen das mittlere Element, wobei in der letzten Gruppe das größere der beiden mittleren Elemente genommen wird, falls diese gerade Kardinalität hat.
3. Rufe SELECT rekursiv auf, um das mittlere Element m der $\lceil n/5 \rceil$ mittleren Elemente zu bestimmen, d.h. SELECT$(\lceil n/5 \rceil, \lceil n/10 \rceil)$. Wenn $\lceil n/5 \rceil$ gerade ist, wird wiederum das größere Element genommen.
4. Teile die Folge aller Elemente in die Teilfolge A_1 der Elemente $\leq m$, und die Teilfolge A_2 der Elemente $> m$ auf. Sei m das k-te Element unter allen n Elementen.
5. Rufe SELECT rekursiv auf, um nun das i-te Element in der entsprechenden Teilfolge zu finden. D. h. falls $i \leq k$ ist, dann rufe SELECT(A_1, i) auf, und sonst SELECT$(A_2, i - k)$.

Laufzeitanalyse von SELECT. Es gilt: Die Anzahl der Elemente $> m$ ist mindestens $3n/10 - 6$. Begründung: Mindestens die Hälfte der mittleren Elemente aus den $\lceil n/5 \rceil$ Gruppen (Schritt 2) sind $> m$. Also steuert die Hälfte der $\lceil n/5 \rceil$ Gruppen jeweils 3 Elemente zu A_2 bei, außer evtl. der letzten Gruppe und der Gruppe, die m enthält, d.h.

$$|A_2| \geq 3 \cdot \left(\frac{1}{2} \cdot \lceil n/5 \rceil - 2\right) \geq \frac{3}{10}n - 6$$
$$\Longrightarrow |A_1| \leq \frac{7}{10}n + 6.$$

Entsprechend ist die Anzahl der Elemente kleiner als m ebenfalls mindestens $3 \cdot (\frac{1}{2} \cdot \lceil n/5 \rceil - 2)$, d.h.

$$|A_1| \geq \frac{3}{10}n - 6 \Longrightarrow |A_2| \leq \frac{7}{10}n + 6.$$

Der rekursive Aufruf von SELECT in 5. wird also auf höchstens $\frac{7}{10}n + 6$ Elemente angewandt.

- Die Schritte 1., 2., 4. sind in $\mathcal{O}(n)$.
- Schritt 3. benötigt $T(\lceil n/5 \rceil)$ Zeit.
- Schritt 5. benötigt $T(\frac{7}{10}n + 6)$ Zeit; dabei ist $T(n)$ die Laufzeit von SELECT(n, i) mit $1 \leq i \leq n$.

Zunächst stellen wir fest, daß ab $n > 20$ gilt $\frac{7}{10}n + 6 < n$. Wir erhalten also als Abschätzung für die Laufzeit:

$$T(n) \le \begin{cases} c_0 \cdot n & \text{für } n \le n_0, \\ & n_0 > 20 \text{ geeignet gewählt} \\ T(\lceil n/5 \rceil) + T(\frac{7}{10}n + 6) + c_1 \cdot n & \text{für } n > n_0. \end{cases}$$

Wir beweisen $T(n) \in \mathcal{O}(n)$ durch Substitution. Sei also $T(n) \le c \cdot n$ für eine Konstante $c > 0$ und alle $n \le n_0$. Für $n > n_0$ folgt dann per Induktion

$$\begin{aligned} T(n) &\le c \cdot \lceil n/5 \rceil + c \cdot \left(\frac{7}{10}n + 6\right) + c_1 \cdot n \\ &\le c \cdot n/5 + c + c \cdot \frac{7}{10}n + c \cdot 6 + c_1 \cdot n \\ &= 9 \cdot c \cdot \frac{n}{10} + 7 \cdot c + c_1 \cdot n. \end{aligned}$$

Damit $9 \cdot c \cdot \frac{n}{10} + 7 \cdot c + c_1 \cdot n \le c \cdot n$ ist, muß c so gewählt werden können, daß $c_1 \cdot n \le c \cdot \underbrace{\left(\frac{n}{10} - 7\right)}_{>0 \text{ nötig}}$. Dazu muß $n > 70$ gelten. In der Abschätzung sollten wir also $n_0 > 70$ wählen.

Kapitel 2

Amortisierte Analyse von Algorithmen

> **Der Literaturtip.** Weiterführendes zu diesem Kapitel ist in dem Buch [CLR90] zu finden.

Die **amortisierte Laufzeitanalyse** ist eine Analysemethode, bei der die Zeit, die benötigt wird, um eine Folge von Operationen auszuführen, über alle auszuführenden Operationen gemittelt wird. Diese Analysemethode kann angewandt werden, um zu beweisen, daß die mittleren Kosten einer Operation klein sind, wenn man über eine Folge von Operationen mittelt, obwohl eine einzelne Operation sehr aufwendig sein kann. Im Unterschied zu der „average-case“-Analyse werden keine Wahrscheinlichkeitsannahmen gemacht. Zur Erinnerung: Der average-case ist das „Mittel“ über alle Eingaben. Bei Gleichverteilungsannahme ist dies wirklich der Mittelwert, ansonsten der Erwartungswert.

Analyse-Techniken für die amortisierte Analyse:

1. **Ganzheitsmethode:** Bestimme eine obere Schranke $T(n)$ für die Gesamtkosten von n Operationen. Das ergibt $\frac{T(n)}{n}$ als die amortisierten Kosten für eine Operation.

2. **Buchungsmethode (accounting):** Bestimme die amortisierten Kosten jeder Operation. Verschiedene Operationen können verschiedene Kosten haben. Es werden frühe Operationen in der betrachteten Folge höher veranschlagt, und die zu hoch veranschlagten Kosten jeweils als Kredit für festgelegte Objekte gespeichert. Dieser Kredit wird dann für spätere Operationen verbraucht, die niedriger veranschlagt werden, als sie tatsächlich kosten.

3. **Potentialmethode:** Ähnlich wie bei der Buchungsmethode werden die amortisierten Kosten jeder einzelnen Operation berechnet. Dabei werden möglicherweise wieder einzelne Operationen höher veranschlagt und später andere Operationen niedriger veranschlagt. Der Kredit wird als Potentialenergie insgesamt gespeichert, anstatt einzelnen Objekten zugeordnet zu werden.

Beachte dabei: Der Kredit wird nur für die Analyse eingeführt.

Wir wollen die drei Methoden am Beispiel „STACK mit MULTIPOP“ erläutern. Betrachte dazu eine Datenstruktur mit „Last in – First out“), einen STACK S. Ein „normaler“ Stack umfaßt zwei Operationen:

PUSH(S, x): lege Objekt x auf STACK S

POP(S): gib oberstes Objekt vom STACK aus und entferne es vom STACK.

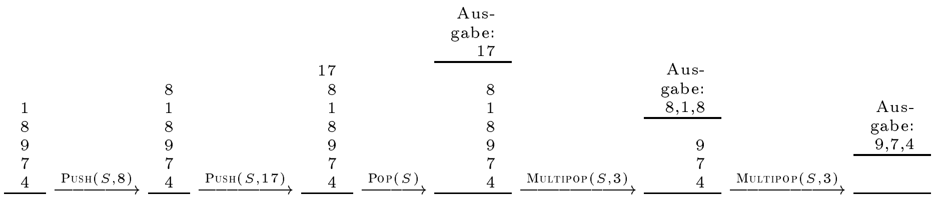
Beide Operationen sind in $\Theta(1)$. Veranschlage daher die Kosten pro Operation mit 1. Dann sind die Gesamtkosten einer Folge von insgesamt n PUSH- und POP-Operationen in $\Theta(n)$. Betrachte nun eine weitere Operation:

MULTIPOP(S, k): gib die obersten k Objekte vom STACK aus und entferne sie vom STACK, bzw. falls der STACK weniger als k Objekte enthält, gib alle Objekte aus, bis der STACK leer ist.

Eine andere Beschreibung von MULTIPOP(S, k):

```
solange $S \neq \emptyset$ und $k \neq 0$, führe aus:
    POP($S$)
    $k := k - 1$.
```

Beispiel.



Die Laufzeit einer Operation MULTIPOP(S, k) ist linear in der Anzahl der ausgeführten POP-Operationen, also $\mathcal{O}(\min(s, k))$, wobei $s = |S|$ ist und wobei die Kosten im worst-case betrachtet werden.

Für eine Folge von n PUSH-, POP-, und MULTIPOP-Operationen, angewandt auf einen zu Beginn leeren STACK, ergibt sich:

worst-case Aufwand einer MULTIPOP-Operation ist $\mathcal{O}(n)$

→ worst-case Aufwand einer beliebigen STACK-Operation ist $\mathcal{O}(n)$

→ Für n STACK-Operationen ergibt sich dann im worst-case $\mathcal{O}(n^2)$.

2.1 Die Ganzheitsmethode

Eine einzelne MULTIPOP-Operation kann teuer sein, d.h. sie kann Kosten n haben. Andererseits gilt: Ein Objekt kann höchstens einmal wieder gePOPt werden für jedes Mal, wo es gePUSHt wurde. Bei dieser Rechnung sind die POPs von MULTIPOP inbegriffen. Es sind also höchstens so viele Kosten für POPs möglich wie für PUSHs, d.h. höchstens n. Der Gesamtaufwand ist also in $\mathcal{O}(n)$, und damit ist der amortisierte Aufwand einer einzelnen Operation in der Folge der n STACK-Operationen in $\mathcal{O}(1)$.

2.2 Die Buchungsmethode

Es werden verschiedene „Gebühren“ für verschiedene Operationen veranschlagt. Die Gebühr pro Operation ist dann der amortisierte Aufwand der Operation. Falls der amortisierte Aufwand einer einzelnen Operation höher ist als der wirkliche Aufwand, so wird die Differenz als Kredit speziellen Objekten zugeordnet. Dieser Kredit kann später benutzt werden, um Aufwand, der höher als der amortisierte Aufwand ist, auszugleichen. Der amortisierte Aufwand muß sorgfältig veranschlagt werden. Der Kredit darf niemals negativ sein, denn der gesamte amortisierte Aufwand einer Folge von Operationen muß immer eine obere Schranke für den wirklichen Aufwand sein.

Aktueller Aufwand bei STACK-Operationen

PUSH(S, x):	1
POP(S):	1
MULTIPOP(S, k):	$\min(\lvert S\rvert, k)$

Definiere amortisierten Aufwand (Gebühr)

PUSH(S, x):	2
POP(S):	0
MULTIPOP(S, k):	0

Beispiel. Tabletts in der Mensa.

Wenn wir ein Tablett auf den STACK legen, bezahlen wir zwei Einheiten. Eine Einheit wird sofort verwendet, um die wirklichen Kosten der PUSH-Operation zu bezahlen. Die zweite Einheit bleibt auf dem Tablett liegen. Wenn das Tablett gePOPt wird, egal ob durch POP oder MULTIPOP, wird die Einheit auf dem Tablett verwendet, um die wirklichen Kosten für POP zu bezahlen. Offensichtlich ist der Kredit niemals negativ. Bei einer Folge von n PUSH-, POP- und MULTIPOP-Operationen benötigen wir höchstens $2 \cdot n$ amortisierte Gesamtkosten, da für maximal n PUSH-Operationen je zwei als amortisierte Kosten veranschlagt werden. Der Gesamtaufwand ist also in $\mathcal{O}(n)$.

2.3 Die Potentialmethode:

Es wird aus zu hoch veranschlagten Kosten ein Potential aufgespart, das später verbraucht werden kann. Starte mit einer Datenstruktur D_0, auf der n Operationen ausgeführt werden.

Für $i = 1, 2, \ldots, n$ seien
c_i: die tatsächlichen Kosten der i-ten Operation
D_i: die Datenstruktur nach der i-ten Operation (angewandt auf D_{i-1}).

Definiere eine Potentialfunktion $\mathcal{C} : D_i \mapsto \mathcal{C}(D_i)$ „Potential von D_i“ und den amortisierten Aufwand $\widehat{c_i}$ der i-ten Operation bzgl. $\mathcal{C}$ als

$$\widehat{c_i} := c_i + \mathcal{C}(D_i) - \mathcal{C}(D_{i-1})$$

Die amortisierten Kosten sind also die tatsächlichen Kosten plus dem Zuwachs an Potential entsprechend der Operation. Das heißt:

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c_i} &= \sum_{i=1}^{n} (c_i + \mathcal{C}(D_i) - \mathcal{C}(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \mathcal{C}(D_n) - \mathcal{C}(D_0).\end{aligned}$$

Wenn $\mathcal{C}$ so definiert wird, daß $\mathcal{C}(D_n) \geq \mathcal{C}(D_0)$ ist, dann sind die amortisierten Kosten eine obere Schranke für die Gesamtkosten. Im allgemeinen ist nicht unbedingt im vorhinein klar, wieviele Operationen ausgeführt werden. Falls $\mathcal{C}(D_i) \geq \mathcal{C}(D_0)$ für alle i, so ist allerdings garantiert, daß (wie bei der Buchungsmethode) im voraus immer genug Kredit angespart wurde. Oft ist es günstig, $\mathcal{C}$ so zu wählen, daß $\mathcal{C}(D_0) = 0$ ist und zu zeigen, daß $\mathcal{C}(D_i) \geq 0$ ist für alle i.

Intuitiv ist klar: Falls $\mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) > 0$ ist, dann wird durch die amortisierten Kosten $\widehat{c_i}$ eine überhöhte Gebühr für die i-te Operation dargestellt, d.h. daß das Potential wächst. Falls $\mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) < 0$ ist, so ist $\widehat{c_i}$ zu tief veranschlagt.

Beispiel. STACK mit MULTIPOP.

Definiere $\mathcal{C}$ als Anzahl der Objekte auf dem STACK. Dann ist $\mathcal{C}(D_0) = 0$. Da die Anzahl der Objekte auf einem STACK nie negativ ist, gilt für alle i $\mathcal{C}(D_i) \geq \mathcal{C}(D_0)$. $\mathcal{C}$ ist also gut gewählt, da die amortisierten Gesamtkosten $\sum_{i=1}^{\ell} c_i + \mathcal{C}(D_\ell) - \mathcal{C}(D_0)$ zu jedem Zeitpunkt ℓ eine obere Schranke für die wirklichen Gesamtkosten sind. Betrachte den amortisierten Aufwand für die verschiedenen STACK-Operationen:

1. Die i-te Operation sei ein PUSH, auf einen STACK mit s Objekten angewandt:

   $$c_i = 1 \text{ und } \mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) = (s+1) - s = 1.$$

   Daraus folgt

   $$\widehat{c_i} = c_i + \mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) = 1 + 1 = 2.$$

2. Die i-te Operation sei ein MULTIPOP(S, k), auf einen STACK S mit s Objekten und $k' := \min(|S|, k)$ angewandt:

   $$c_i = k' \text{ und } \mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) = (s - k') - s = -k'.$$

   Daraus folgt

   $$\widehat{c_i} = c_i + \mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) = k' - k' = 0.$$

3. Die i-te Operation sei ein POP, angewandt auf einen STACK mit s Objekten:

   $$c_i = 1 \text{ und } \mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) = s - 1 - s = -1.$$

   Daraus folgt

   $$\widehat{c_i} = 1 - 1 = 0.$$

Also ist $\sum_{i=1}^{n} \widehat{c_i} \leq 2n$; der amortisierte Gesamtaufwand ist damit im worst-case in $\mathcal{O}(n)$.

Bemerkung. Ein weiteres Beispiel, an dem sich die Ideen der amortisierten Analyse gut erklären lassen, sind **dynamische Tabellen**: Objekte sollen in einer Tabelle abgespeichert werden, wobei Objekte eingefügt bzw. gelöscht werden können. Zu Beginn einer Folge von Operationen vom Typ Einfügen und Löschen habe die Tabelle die Größe h. Wenn eine Operation Einfügen zu einem Zeitpunkt auf die Tabelle angewandt wird, an dem die Tabelle voll ist, soll eine Tabellenexpansion vorgenommen werden, in der die Tabelle verdoppelt wird. Die Tabellenexpansion (angewandt auf die Tabelle der Größe k) habe wirkliche Kosten k. Entsprechend werde eine Tabellenkontraktion durchgeführt, in der die Tabelle halbiert wird, wenn sie nur noch sehr dünn besetzt ist, etwa höchstens zu $\frac{1}{4}$. Die Tabellenkontraktion angewandt auf eine Tabelle mit k Elementen hat dann wiederum Kosten k. Wie groß ist der amortisierte Aufwand für eine Folge von n Operationen vom Typ Einfügen bzw. Löschen?

Kapitel 3

Das Union-Find-Problem

Das Union-Find-Problem ist ein grundlegendes Problem, das sehr viele Anwendungen in sehr unterschiedlichen Bereichen hat. Neben seiner grundsätzlichen Bedeutung zeichnet es sich durch die Tatsache aus, daß es zwar einen ausgesprochen einfachen (einfach zu implementierenden) Algorithmus zu seiner Lösung gibt, die Analyse dieses Algorithmus jedoch „beeindruckend“ schwierig ist und überraschenderweise eine fast lineare (aber eben nur fast) Laufzeit ergibt.

Definition 3.1 *Das* Union-Find-***Problem*** *besteht darin, eine Folge disjunkter Mengen zu verwalten, die sich infolge von Vereinigungsoperationen „laufend“ ändert.*

Gegeben sei eine endliche Grundmenge M. Es soll möglichst effizient eine beliebige Abfolge von Operationen folgenden Typs ausgeführt werden:

> Makeset(x): *Führe eine neue einelementige Menge $\{x\}$ ein, die zuvor noch nicht existiert hat; $x \in M$.*
>
> Union($S_i, S_j; S_k$): *Bilde eine neue Menge $S_k := S_i \dot{\cup} S_j$ aus bisher vorhandenen (disjunkten) Mengen S_i und S_j durch Vereinigung. Entferne S_i und S_j.*
>
> Find(x): *Für $x \in M$ gib diejenige Menge der bisher entstandenen Mengen an, welche x enthält.*

3.1 Drei Ansätze

Beispiel.

$$S_1 = \{1,3,5,7\},\ S_2 = \{2,4,8\},\ M = \{1,2,3,4,5,6,7,8,9,10\}$$

3.1.1 1. Ansatz

Benutze ein Array A, das die Zuordnung von x zum Index der Menge, die x enthält, angibt.

x	1	2	3	4	5	6	7	8	9	...
$A[x]$	1	2	1	2	1	–	1	2	–	...

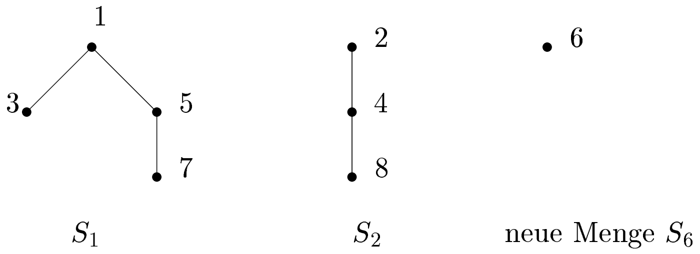
- MAKESET(x): Schreibe in $A[x]$ „neuen Index“, etwa x.
  Beipiel: MAKESET(6), $A[6] := 6$. Aufwand ist in $\mathcal{O}(1)$.
- FIND(x): Gib $A[x]$ aus. Aufwand ist in $\mathcal{O}(1)$.
- UNION($S_i, S_j; S_k$):
  ```
  1. Für $x \in M$ führe aus
  2.     falls $A[x] = i$ oder $A[x] = j$ setze $A[x] := k$
  ```
  Aufwand ist in $\mathcal{O}(|M|)$.

Wenn also eine beliebige Folge von n Operationen vom Typ MAKESET, FIND und UNION ausgeführt wird, so ist der Aufwand dafür in $\mathcal{O}(n^2)$, falls $|M| \in \mathcal{O}(n)$. Wir setzen im folgenden voraus: $|M| \in \mathcal{O}(n)$.

3.1.2 2. Ansatz

Repräsentiere Mengen durch Wurzelbäume, d.h. jede Menge ist eine „Struktur“ Baum, dessen Knoten die Elemente der Menge sind. Diese Bäume entstehen durch die Operationen MAKESET und UNION.

Beispiel.



Als Mengenindex diene jeweils die Wurzel des Baumes. Dann werden zwei Mengen vereinigt, indem der eine Baum an die Wurzel des anderen gehängt wird. Als Mengenindex der neuen Menge diene der Index der Menge, an deren Wurzel angehängt wurde.

Beispiel.

$$\begin{array}{lcl} \textsc{Union}(S_1, S_2; S_2) & \approx & \textsc{Union}(1,2;2) \text{ und} \\ \textsc{Union}(S_2, S_6; S_6) & \approx & \textsc{Union}(2,6;6) \end{array}$$

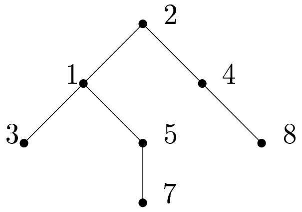


Abbildung 3.1: Resultat von $\textsc{Union}(S_1, S_2; S_2)$

$\textsc{Find}$ wird für ein Element x ausgeführt, indem von dem entsprechenden Knoten aus durch den Baum bis zu dessen Wurzel gegangen wird.

Die **Repräsentation der Bäume** erfolgt durch ein Array $\textsc{Vor}$, in dem in $\textsc{Vor}[x]$ der *Vorgänger* von x im Baum abgelegt ist. Dabei setzen wir $\textsc{Vor}[x] := 0$, wenn x eine Wurzel ist.

x	1	2	3	4	5	6	7	8	9	...
$\textsc{Vor}[x]$	2	0	1	2	1	0	5	4	–	...

- $\textsc{Find}(x)$:

```
Setze $j := x$
Solange $\textsc{Vor}[j] \neq 0$ (und definiert), führe aus
    $j := \textsc{Vor}[j]$
Gib $j$ aus.
```

  Der Aufwand ist in $\mathcal{O}(n)$.

- $\textsc{Union}(i, j; j)$:

```
$\textsc{Vor}[i] := j$.
```

- $\textsc{Makeset}(x)$:

```
setze $\textsc{Vor}[x] := 0$.
```

  Der Aufwand für $\textsc{Union}$ und $\textsc{Makeset}$ ist in $\mathcal{O}(1)$.

Gesamtaufwand. Bei einer Folge von n Operationen vom Typ MAKESET, UNION und FIND ist der Gesamtaufwand in $\mathcal{O}(n^2)$:

Betrachte

$\Theta(n)$ MAKESET (z.B $n/4$), gefolgt von

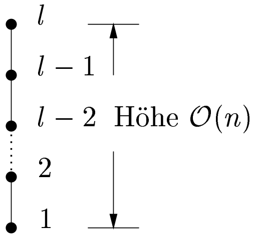
$\Theta(n)$ UNION, gefolgt von

$\Theta(n)$ FIND.

Eine Folge von t verschiedenen FIND hat Aufwand von mindestens

$$\sum_{i=1}^{t} i \in \Theta(t^2) = \Theta(n^2)$$

(weil $t \in \Theta(n)$). Operationen vom Typ FIND sind also besonders teuer, da durch UNION „sehr hohe“ Bäume entstehen können.



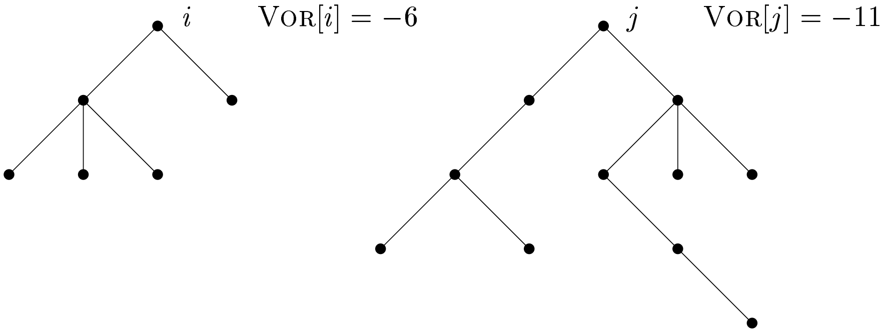
3.1.3 3. Ansatz

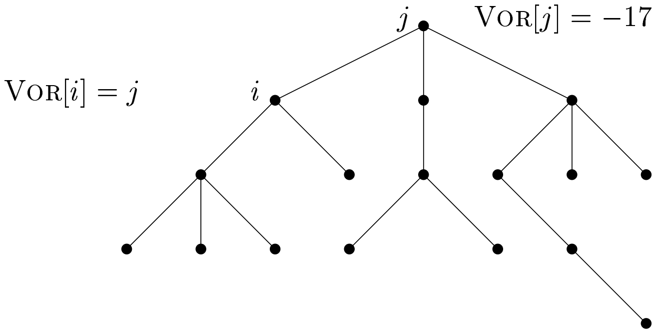
Wie nehmen zwei Modifikationen am zweiten Ansatz vor.

- Modifikation 1: ein Ausgleich bei UNION: „weighted UNION rule“ oder „balancing“.
- Modifikation 2: „Pfadkompression“ bei FIND.

Modifikation 1: „weighted UNION“. Bei der Ausführung von UNION wird immer der „kleinere“ Baum an den „größeren“ Baum gehängt. „Kleiner“ bzw. „größer“ bezieht sich hier einfach auf die Anzahl der Knoten (Kardinalität der entsprechenden Menge). Um zu entscheiden, welcher Baum kleiner bzw. größer ist, wird in VOR$[x]$ für Wurzeln x jeweils die Knotenzahl des Baums als negative Zahl gespeichert. Das negative Vorzeichen kennzeichnet die Wurzeleigenschaft und der Betrag die Kardinalität der Menge.

Beispiel. „weighted UNION“: UNION(i, j)





Formal: UNION(i, j)

Es gilt:

- $\text{VOR}[i] = -\#(\text{Knoten im Baum i})$
- $\text{VOR}[j] = -\#(\text{Knoten im Baum j})$

```
1. Setze $z := \text{VOR}[i] + \text{VOR}[j]$
2. Falls $|\text{VOR}[i]| < |\text{VOR}[j]|$ dann
3.         setze $\text{VOR}[i] := j$ und $\text{VOR}[j] := z$,
4.         ansonsten setze $\text{VOR}[j] := i$ und $\text{VOR}[i] := z$.
```

Der Aufwand ist in $\Theta(1)$.

Läßt sich nun etwas über die Höhe von Bäumen aussagen, die durch „weighted UNION“ entstanden sind?

Lemma 3.1 *Entsteht durch eine Folge von* MAKESET *und weighted* UNION *über einer Menge M ein Baum T mit $|T| = t$ (T enthält t Knoten), so ist $h(T) \leq \log_2 t$, wobei $h(T)$ die Höhe von T ist, also die maximale Anzahl von Kanten auf einem einfachen Weg von der Wurzel von T zu einem Blatt von T.*

Beweis. Induktion über die Anzahl der UNION-Operationen.

Solange keine UNION-Operation ausgeführt wurde, haben alle Bäume Höhe 0 und einen Knoten, d.h. es gilt für alle T:

$$h(T) = 0 \leq \log_2 1 = 0.$$

Betrachte die n-te UNION-Operation, etwa UNION(i, j), wobei T_i Baum mit Wurzel i und T_j Baum mit Wurzel j vor Ausführung der Operation UNION(i, j) sei. O.B.d.A. sei $|T_j| \geq |T_i|$, dann wird durch UNION(i, j) T_i an die Wurzel j von T_j angehängt, und es entsteht der Baum $T_{\text{UNION}(i,j)}$ mit

$$h(T_{\text{UNION}(i,j)}) = \max(h(T_j), h(T_i) + 1).$$

- Falls $h(T_j) > h(T_i) + 1$, dann ist

  $$h(T_{\text{UNION}(i,j)}) = h(T_j) \leq \log_2(|T_j|) \leq \log_2(|T_{\text{UNION}(i,j)}|).$$

- Falls $h(T_j) \leq h(T_i) + 1$, dann ist

  $$h(T_{\text{UNION}(i,j)}) = h(T_i) + 1 \leq \log_2(|T_i|) + 1 \leq \log_2(|T_{\text{UNION}(i,j)}|),$$

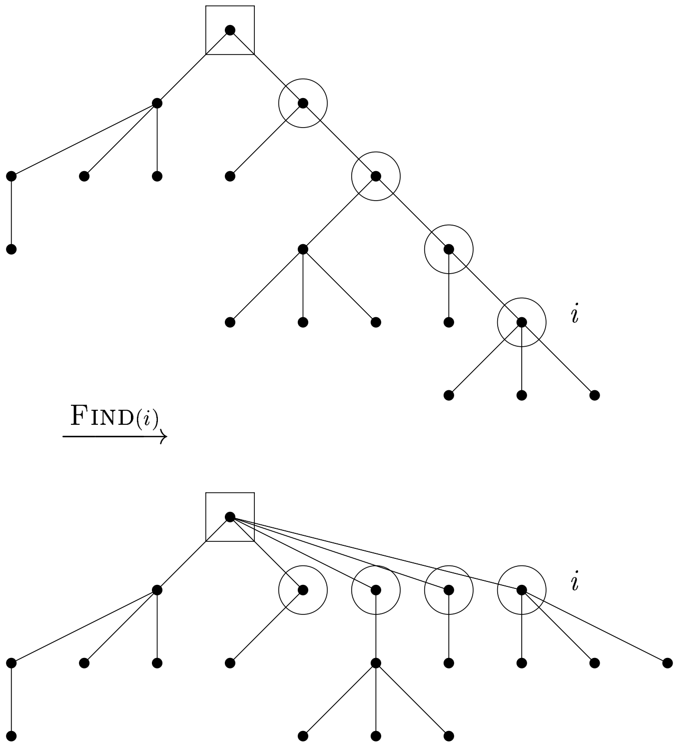
  da $|T_{\text{UNION}(i,j)}| \geq 2 \cdot |T_i|$.

■

Folgerung. In einem durch weighted UNION-Operation entstandenen Baum kann jede FIND-Operation in $\mathcal{O}(\log n)$ Zeit ausgeführt werden. Für eine Folge von n MAKESET-, UNION- und FIND-Operationen ergibt sich dann ein Gesamtaufwand von $\mathcal{O}(n \log n)$.

Modifikation 2: Pfadkompression. Bei der *Pfadkompression* werden bei der Ausführung einer Operation FIND(i) alle Knoten, die während dieses FIND durchlaufen werden, zu direkten Nachfolgern der Wurzel des Baumes gemacht, in dem Knoten i liegt.

Beispiel. Beispiel für Pfadkompression.



Auch bei FIND mit Pfadkompression hat eine Operation FIND einen worst-case Aufwand von $\mathcal{O}(\log n)$. Der Gesamtaufwand ist also weiterhin in $\mathcal{O}(n \log n)$. Eine genauere amortisierte Analyse ergibt jedoch einen besseren Gesamtaufwand von $\mathcal{O}(n * G(n))$, wobei $G(n)$ sehr, sehr langsam wächst – wesentlich langsamer als $\log n$. $G(n)$ ist für alle praktisch relevanten Werte n „klein“, d.h. $G(n)$ verhält sich „praktisch“ wie eine Konstante.

Definition 3.2

$$G(n) := min\{y : F(y) \geq n\},$$

wobei $F(0) := 1$ *und* $F(y) = 2^{F(y-1)}$ *für* $y > 0$.

Wie schnell wächst $F(n)$ bzw. $G(n)$?

$F(0)$	$F(1)$	$F(2)$	$F(3)$	$F(4)$	$F(5)$
		$= 2^2$	$= 2^4$	$= 2^{16}$	$= 2^{65536}$
$= 1$	$= 2$	$= 4$	$= 16$	$= 65536$	

Für alle praktisch relevanten Werte von n ist also $G(n) \leq 5$.

Satz 3.1 (Hopcroft & Ullman 1973) *Der Gesamtaufwand für eine Folge Q von n Operationen vom Typ* MAKESET, weighted UNION *und* FIND *mit Pfadkompression ist in $\mathcal{O}(n \cdot G(n))$.*

Beweis. Wir trennen den Aufwand für die MAKESET- und UNION-Operationen von dem Aufwand für die FIND-Operationen.

Der Gesamtaufwand für alle MAKESET- und alle UNION-Operationen ist in $\mathcal{O}(n)$.

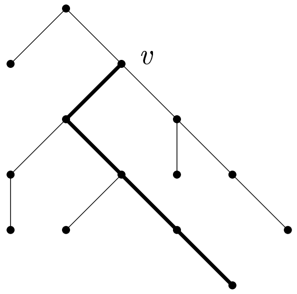
Der Gesamtaufwand für alle FIND-Operationen ist proportional zu der Anzahl der durch FIND-Operationen bewirkten „Knotenbewegungen“ (Zuteilung eines neuen Vorgängers). Eine solche Knotenbewegung hat Aufwand $\mathcal{O}(1)$.

Die Knotenbewegungen werden in zwei Klassen A und B aufgeteilt. Dazu werden *Knotengruppen* $\gamma_1, \ldots, \gamma_{G(n)+1}$ gebildet, für die sich die jeweilige Anzahl an Knotenbewegungen „leicht“ aufsummieren läßt.

Zur Definition der γ_j.

Der **Rang** $r(v)$ eines Knotens v sei definiert als die Höhe des Unterbaums mit Wurzel v im Baum T, der nach Ausführung aller MAKESET- und UNION-Operationen aus Q *ohne* die FIND-Operationen entstehen würde (und v enthält). Nenne diese Folge von Operationen Q'. Wenn also T_v der Unterbaum von T mit Wurzel v ist, so ist $r(v) = h(T_v)$.

Beispiel. $r(v) = 4$



Aus dem Lemma folgt, daß $r(v) \leq \log n$ ist. Für $j \geq 1$ definiere

$$\gamma_j := \{v : \log^{(j+1)} n < r(v) \leq \log^{(j)} n\},$$

wobei

$$\log^{(j)} n := \begin{cases} n & \text{falls } j = 0, \\ \log(\log^{(j-1)} n) & \text{falls } j > 0 \text{ und } \log^{(j-1)} n > 0, \\ \text{undefiniert} & \text{falls } j > 0 \text{ und } \log^{(j-1)} n \leq 0 \text{ oder} \\ & \log^{(j-1)} n \text{ undefiniert} \end{cases}$$

und

$$\gamma_j := \begin{cases} \emptyset & \text{wenn } \log^{(j)} n \text{ undefiniert}, \\ \{v : r(v) = 0\} & \text{falls } \log^{(j)} n = 0. \end{cases}$$

γ_j heißt die *j-te Ranggruppe*, und es gilt

$$\begin{aligned} \gamma_1 &= \{v : \log^{(2)} n < r(v) \leq \log n\} \\ &\vdots \\ \gamma_{G(n)+2} &= \emptyset, \end{aligned}$$

da $\log^{(G(n))} n \leq 1$ und daher $\log^{(G(n)+2)} n$ undefiniert ist.

Beispiel. Ranggruppen für $n = 66000$.

$$\begin{aligned} \gamma_1 =& \{v : \log^{(2)} 66000 < r(v) \leq \log 66000\} \\ =& \{v : 4 < r(v) \leq 16\} \\ \gamma_2 =& \{v : \log 4 < r(v) \leq 4\} = \{v : 2 < r(v) \leq 4\} \\ \gamma_3 =& \{v : \log 2 < r(v) \leq 2\} = \{v : 1 < r(v) \leq 2\} \\ \gamma_4 =& \{v : 0 < r(v) \leq 1\} \\ \gamma_5 =& \{v : \log 0 < r(v) \leq 0\} = \{v : r(v) = 0\} \end{aligned}$$

Dann ist $G(66000) = 5$ und $\gamma_{G(66000)+1} = \emptyset$.

Die Klasse aller Knotenbewegungen, die durch FIND-Operationen ausgeführt werden, wird aufgeteilt in zwei disjunkte Klassen A und B.

- Klasse A: diejenigen Knotenbewegungen, die für Knoten ausgeführt werden, deren Vorgänger einer anderen Ranggruppe angehört (Vorgänger zum Zeitpunkt der entsprechenden FIND-Operation).
- Klasse B: diejenigen Knotenbewegungen, die für Knoten ausgeführt werden, deren Vorgänger zu derselben Ranggruppe gehört.

Bei der nun folgenden *amortisierten Analyse* werden bei Knotenbewegungen aus A die Kosten der entsprechenden FIND-Operation zugeordnet und bei Knotenbewegungen aus B „dem bewegten Knoten". In letzterem Fall wird dann argumentiert, daß der Aufwand, der sich aus der „Anzahl der Knoten, auf die Bewegungen aus B angewandt werden, mal Anzahl der Bewegungen pro solchen Knoten" ergibt, in $\mathcal{O}(n \cdot G(n))$ ist.

Hilfsbehauptung 1 *Zu jedem Zeitpunkt von Q gilt für jeden Knoten v, daß für alle w, die Vorgänger von v sind, $r(w) > r(v)$ ist.*

Beweis. Wenn irgendwann v ein Nachfolger von w wird, so gehört v in dem Baum, der nach Ausführung aller Operationen von Q' (also ohne die FIND) entstehen würde, zum Unterbaum von w. ■

Folgerung. Zu jedem Zeitpunkt von Q sind für jeden Knoten v die Ränge der Knoten auf dem Weg von v zur Wurzel seines Baumes monoton wachsend.

Zu A: Betrachte die Ausführung von FIND(v). Es liegen auf dem Weg von v zur Wurzel des entsprechenden Baumes höchstens $G(n)$ Knoten, deren direkter Vorgänger in einer anderen Ranggruppe liegt, da wir maximal $G(n)+1$ Ranggruppen haben, und die Ränge auf dem Weg monoton wachsend sind. Die Operation FIND(v) verursacht also höchstens $G(n)$ Knotenbewegungen aus A.

Zu B: Wir beweisen eine weitere Hilfsbehauptung, um $|\gamma_j|$ abschätzen zu können.

Hilfsbehauptung 2 *Es gibt höchstens $n/2^r$ Knoten vom Rang r.*

Beweis. Betrachte zu Knoten v vom Rang r den Unterbaum T_v aus dem Baum zu Q'. Da $h(T_v) \leq \log|T_v|$ und $h(T_v) = r(v) = r$ ist, gibt es mindestens 2^r Knoten in T_v. Da Knoten gleichen Ranges disjunkte Unterbäume (bzgl. Q') haben, folgt die Behauptung. ■

Da es höchstens $n/2^r$ Knoten vom Rang r gibt, ist

$$\begin{aligned}
|\gamma_j| &\leq \sum_{i=\lceil \log^{(j+1)} n \rceil}^{\lfloor \log^{(j)} n \rfloor} \frac{n}{2^i} \leq \frac{n}{2^{\log^{(j+1)} n}} \underbrace{\Big(\sum_{i=0}^{\infty} \frac{1}{2^i}\Big)}_{\leq 2} \\
&\leq \frac{2n}{2^{\log^{(j+1)} n}} = \frac{2n}{2^{\log(\log^{(j)} n)}} \\
&= \frac{2n}{\log^{(j)} n}.
\end{aligned}$$

Ein Knoten aus γ_j kann höchstens $\log^{(j)} n$ mal bewegt werden, bevor er einen Vorgänger erhält, der in einer anderen Ranggruppe γ_i mit $i < j$ liegt. Damit ist die Gesamtzahl an Knotenbewegungen der Klasse B für Knoten der Ranggruppe γ_j höchstens

$$\log^{(j)} n \cdot \frac{2n}{\log^{(j)} n} = 2n.$$

Da insgesamt höchstens $G(n)+1$ nichtleere Ranggruppen existieren, ist der Gesamtaufwand für Knotenbewegungen der Klasse B also in $\mathcal{O}(n \cdot G(n))$.

Insgesamt ist dann der Gesamtaufwand für alle durch Find ausgelösten Knotenbewegungen ebenfalls in $\mathcal{O}(n \cdot G(n))$. ■

3.1.4 Bemerkungen

Eine genauere Analyse führt zu einem Aufwand aus $\mathcal{O}(m \cdot \alpha(m,n))$ für m Find, Union und Makeset-Operationen mit n Elementen (Tarjan, 1975). Dabei ist

$$\alpha(m,n) := \min\{i \geq 1 : A(i, \lfloor \tfrac{m}{n} \rfloor) > \log n\}$$

und

$$\begin{aligned} A(1,j) &= 2^j \text{ für } j \geq 1 \\ A(i,1) &= A(i-1,2) \text{ für } i \geq 2 \\ A(i,j) &= A(i-1, A(i,j-1)) \text{ für } i,j \geq 2. \end{aligned}$$

Die Funktion A heißt *Ackermann-Funktion* und wächst noch stärker als F (iterative Zweierpotenz). Andererseits wurde bewiesen, daß der Aufwand für eine Folge von m Find und n Union- und Makeset-Operationen im Allgemeinen auch in $\Omega(m \cdot \alpha(m,n))$ liegt (Tarjan, 1979).

Für spezielle Folgen von Union-, Find- und Makeset-Operationen, über die man vorab „gewisse strukturelle Informationen“ hat, ist der Aufwand in $\mathcal{O}(m)$ (Gabow & Tarjan, 1985).

Insgesamt lassen sich sehr oft Algorithmen oder Teile von Algorithmen als Folgen von Union-, Find- und Makeset-Operationen auffassen und sind damit „fast“ in linearer Zeit durchführbar. Oft tritt sogar der Fall auf, daß die Folge „spezielle Struktur“ hat und wirklich in Linearzeit ausführbar ist.

3.2 Anwendungsbeispiele für UNION-FIND

3.2.1 Der Algorithnus von **Kruskal** für MST

MST: Minimum Spanning Tree, siehe auch Kapitel 4.1.

Eingabe: Knotenliste V, Kantenliste E mit Kantengewichten.

Ausgabe: Datenstruktur: Menge GRÜN.

Kruskal

```
1. Setze GRÜN := $\emptyset$.
2. Sortiere $E$ entsprechend Gewicht „aufsteigend“.
   Die sortierte Liste sei SORT($E$).
3. Für $v \in V$ führe aus: MAKESET($v$).
4. Für $\{v, w\} \in$ SORT($E$) führe aus
5.       FIND($v$) und FIND($w$)
6.       Falls FIND($v$) $\neq$ FIND($w$) führe aus
7.            UNION(FIND($v$), FIND($w$)) und GRÜN := GRÜN $\cup \{\{v, w\}\}$.
8. Gib GRÜN aus.
```

Wenn SORT(E) bereits vorliegt, dann ist die Laufzeit in $\mathcal{O}(|E| \cdot \alpha(|E|, |V|))$.

3.2.2 Das OFFLINE-MIN-Problem

Gegeben sei eine Menge $M = \emptyset$. Führe ein Folge Q von n Operationen vom Typ

$$\begin{array}{ll} \textsc{Insert}[i]: & M := M \cup \{i\} \\ \textsc{Extract-Min}: & M := M \setminus \{\min M\} \end{array}$$

aus, wobei o.B.d.A. i eine positive ganze Zahl ist. Eine Operation INSERT[i] trete für jedes i höchstens einmal in der Folge auf.

Zu einer Folge Q wollen wir alle i finden, die durch eine Operation EXTRACT-MIN entfernt werden und die entsprechende EXTRACT-MIN-Operation angeben. (Daher Bezeichnung „Offline“: Q ist vorher bekannt.) Wir lösen das Problem durch eine Folge von UNION- und FIND-Operationen.

Bemerkung. Eine Folge von n Operationen vom Typ UNION und FIND, beginnend mit einer Menge disjunkter Teilmengen einer Menge mit $\mathcal{O}(n)$ Elementen, ist natürlich ebenfalls in $\mathcal{O}(n \cdot G(n))$ (bzw. $\mathcal{O}(n \cdot \alpha(n, n))$) ausführbar.

Schreibe die Folge Q als

$$Q_1 E Q_2 E \dots E Q_{k+1},$$

wobei Q_j für $1 \leq j \leq k+1$ nur aus INSERT-Operationen besteht (möglicherweise $Q_j = \emptyset$); E stehe für eine EXTRACT-MIN-Operation. Die Anzahl der EXTRACT-MIN sei also k.

Für den UNION-FIND-Algorithmus initialisieren wir (paarweise disjunkte) Mengen

$$j := \{i : \textsc{Insert}[i] \text{ liegt in } Q_j\}$$

für $1 \leq j \leq k+1$. Benutze Arrays PRED (predecessor) und SUCC (successor) zur Erzeugung doppelt verketteter Listen der Werte j, für die eine Menge j existiert.

Zu Beginn sei $\textsc{Pred}[j] = j-1$ für $2 \leq j \leq k+1$ und $\textsc{Succ}[j] = j+1$ für $1 \leq j \leq k$.

$$\begin{tikzcd} 1 \arrow[r, bend left] & 2 \arrow[l, bend left] \arrow[r, bend left] & \cdots \arrow[l, bend left] \arrow[r, bend left] & k \arrow[l, bend left] \arrow[r, bend left] & k+1 \arrow[l, bend left] \end{tikzcd}$$

Formale Beschreibung der OFFLINE-MIN-Prozedur.

```
OFFLINE-MIN
Für $i = 1$ bis $n$ führe aus
    Setze $j :=$ FIND[$i$]
    Falls $j \leq k$, dann
        Schreibe „$i$ ist entfernt worden im $j$-ten EXTRACT-MIN “
        UNION($j$, SUCC[$j$]; SUCC[$j$]),
        SUCC[PRED[$j$]] := SUCC[$j$],
        PRED[SUCC[$j$]] := PRED[$j$].
```

Beispiel. Folge Q: $\underbrace{4,3}_{Q_1}, E, \underbrace{2}_{Q_2}, E, \underbrace{1}_{Q_3}, E, \underbrace{\quad}_{Q_4}$ mit $k = 3$.

Mengen: $1 = \{4,3\}, 2 = \{2\}, 3 = \{1\}, 4 = \emptyset$.

i=1: $\textsc{Find}[1] = 3; 3 \leq 3 \leadsto$ „1 ist im 3-ten EXTRACT-MIN entfernt worden.“
$\leadsto 1 = \{4,3\}, 2 = \{2\}, 4 = \{1\}$,

$\textsc{Succ}[2] = 4$, $\textsc{Pred}[4] = 2$

i=2: $\textsc{Find}[2] = 2; 2 \leq 3 \leadsto$ „2 ist im 2-ten EXTRACT-MIN entfernt worden“
$\leadsto 1 = \{4,3\}, 4 = \{1,2\}$,

$\textsc{Succ}[1] = 4$, $\textsc{Pred}[4] = 1$

i=3: FIND[3] $= 1; 1 \leq 3 \rightsquigarrow$ „3 ist im 1-ten EXTRACT-MIN entfernt worden.“
$\rightsquigarrow 4 = \{4, 3, 1, 2\}$,
SUCC[0] = 4, PRED[4] = 0

i=4: FIND[4] $= 4; 4 > 3 \rightsquigarrow$ „4 ist nicht gelöscht worden.“

Bemerkung. Der Algorithmus OFFLINE-MIN ist sogar in $\mathcal{O}(n)$, da die UNION-FIND-Folge zu den Spezialfällen gehört, die in Linearzeit ausführbar sind.

3.2.3 Äquivalenz endlicher Automaten

Definition 3.3 *Ein endlicher Automat $\mathcal{A}$ besteht aus einem Alphabet Σ, einer endlichen Zustandsmenge Q, einem Anfangszustand $q_0 \in Q$, einer Menge von Endzuständen $F \subseteq Q$ und der „Zustandsübergangsfunktion“ $\delta : Q \times \Sigma \longrightarrow Q$. Σ^* bezeichne die Menge aller Worte endlicher Länge über Σ (inkl. dem „leeren Wort“ ϵ). Dann läßt sich δ erweitern zu $\delta : Q \times \Sigma^* \longrightarrow Q$ durch*

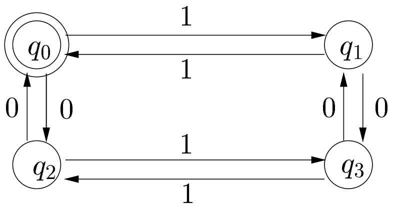
$$\begin{aligned} \delta(q, \epsilon) &:= q \text{ und} \\ \delta(q, wa) &:= \delta(\delta(q, w), a) \end{aligned}$$

für alle $w \in \Sigma^*$ und $a \in \Sigma$ und $q \in Q$.

Der Automat $\mathcal{A}$ ***akzeptiert*** *ein Wort w genau dann, wenn $\delta(q_0, w) \in F$. Die Sprache der Worte, die von $\mathcal{A}$ akzeptiert werden, heißt $L(\mathcal{A})$. Zwei Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$ heißen* äquivalent, *wenn $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ ist. Schreibe dann $\mathcal{A}_1 \equiv \mathcal{A}_2$.*

Beispiel.

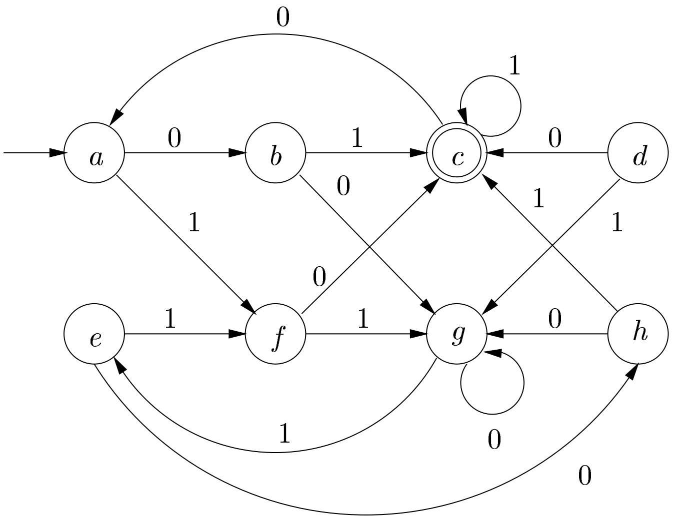
$\Sigma = \{0, 1\}, Q = \{q_0, q_1, q_2, q_3\}, F = \{q_0\}$,
$L(\mathcal{A}) := \{w \in \Sigma^* :$ Anzahl der 0 in w und Anzahl der 1 in w sind gerade$\}$.



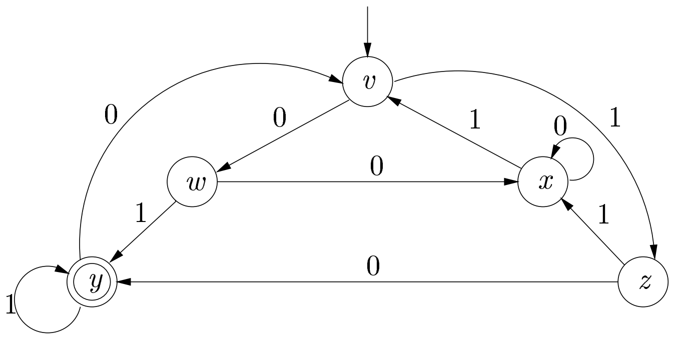
Wir wollen möglichst „effizient“ für zwei beliebige endliche Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$ über dem Alphabet Σ entscheiden, ob sie äquivalent sind, d.h., ob $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ ist.

Beispiel. $\Sigma = \{0, 1\}$.

$\mathcal{A}_1 : Q_1 = \{a, b, c, d, e, f, g, h\}$, Anfangszustand a, Endzustandsmenge $\{c\}$.



$\mathcal{A}_2 : Q_2 = \{v, w, x, y, z\}$, Anfangszustand v, Endzustandsmenge $\{y\}$.



Sind $\mathcal{A}_1$ und $\mathcal{A}_2$ äquivalent? Die Antwort ist „ja“. Wie kann man das aber testen?

Wir benutzen zur Entscheidung, ob $\mathcal{A}_1 \equiv \mathcal{A}_2$, den Begriff *äquivalenter Zustände*: schreibe $q \equiv q'$. Aus der Theoretischen Informatik ist dieser Begriff für Zustände desselben Automaten $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ bekannt. Dort wurde definiert, daß

$$q \equiv q' \text{ für } q, q' \in Q, \text{ wenn für alle } w \in \Sigma^* \text{ gilt}$$
$$\delta(q, w) \in F \iff \delta(q', w) \in F.$$

Entsprechend definiere für zwei endliche Automaten $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ und $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$, daß für Zustände $q_1 \in Q_1, q_2 \in Q_2$ mit o.B.d.A. $Q_1 \cap Q_2 = \emptyset$ gilt:

$$q_1 \equiv q_2 \text{ wenn } \delta_1(q_1, w) \in F_1 \iff \delta_2(q_2, w) \in F_2 \text{ für alle } w \in \Sigma^*.$$

Dann gilt offensichtlich: $\mathcal{A}_1 \equiv \mathcal{A}_2 \iff s_1 \equiv s_2$.

Es ist leicht zu sehen (siehe auch Theoretische Informatik), daß gilt:

$$q \equiv q' \text{ für } q, q' \in Q \text{ genau dann, wenn } \delta(q, a) \equiv \delta(q', a) \text{ für alle } a \in \Sigma.$$

Außerdem ist $q \not\equiv q'$ für alle $q \in F, q' \in Q \setminus F$ (betrachte das leere Wort ϵ).

Entsprechend gilt auch:

> $q_1 \equiv q_2$ für $q_1 \in Q_1, q_2 \in Q_2$ genau dann, wenn $\delta(q_1, a) \equiv \delta(q_2, a)$
> für alle $a \in \Sigma$.

Außerdem ist $q_1 \not\equiv q_2$ für alle $q_1 \in F_1$ und $q_2 \in Q_2 \setminus F_2$ (bzw. $q_1 \in Q_1 \setminus F_1$ und $q_2 \in F_2$).

Diese Eigenschaften benutzen wir nun, um zu testen, ob $\mathcal{A}_1 \equiv \mathcal{A}_2$, d.h. $s_1 \equiv s_2$. Wir nehmen an, daß $s_1 \equiv s_2$, und folgern daraus die Äquivalenz weiterer Zustände. Auf diese Weise erhalten wir eine Partition von $Q_1 \cup Q_2$ in Klassen „angeblich äquivalenter“ Zustände. Enthält eine dieser Klassen sowohl einen Endzustand als auch einen Nichtendzustand, so kann auch nicht $s_1 \equiv s_2$ gelten.

Vorgehensweise informell. In einem STACK S werden Paare von Zuständen (q_1, q_2) gehalten, die „angeblich“ äquivalent sind, deren Nachfolger $(\delta_1(q_1, a), \delta_2(q_2, a))$ usw. noch nicht betrachtet wurden. Zu Beginn enthält S das Paar (s_1, s_2). Um eine Partition von $Q_1 \cup Q_2$ in Klassen „angeblich“ äquivalenter Zustände zu berechnen, wird eine Folge von UNION- und FIND-Operationen benutzt. Beginnend mit den einelementigen Teilmengen von $Q_1 \cup Q_2$ werden jeweils die Mengen vereinigt, die q_1 bzw. q_2 enthalten für ein Paar (q_1, q_2), das als „angeblich“ äquivalent nachgewiesen wurde. Dann ist $\mathcal{A}_1 \equiv \mathcal{A}_2$ genau dann, wenn die Partition von $Q_1 \cup Q_2$, mit der das Verfahren endet, keine Menge enthält, die sowohl einen Endzustand als auch einen Nichtendzustand als Element enthält.

Formale Beschreibung.

Äquivalenz endlicher Automaten

```
$S := 0$
Für alle $q \in Q_1 \cup Q_2$ führe aus
    $\textsc{Makeset}(q)$;
$\textsc{Push}((s_1, s_2))$;
Solange $S \neq \emptyset$ führe aus
    $(q_1, q_2) := \textsc{Pop}(S)$;
    falls $\textsc{Find}[q_1] \neq \textsc{Find}[q_2]$ führe aus
        $\textsc{Union}(\textsc{Find}[q_1], \textsc{Find}[q_2])$
        Für alle $a \in \Sigma$: $\textsc{Push}(\delta_1(q_1, a), \delta_2(q_2, a))$.
```

Laufzeitanalyse. Sei $n := |Q_1| + |Q_2|$ und $|\Sigma| = k$. Da zu Beginn die Partition aus n Mengen besteht, werden höchstens $n - 1$ Union ausgeführt. Die Anzahl der Find ist proportional zur Anzahl der Paare, die insgesamt auf den Stack gelegt werden. Dies sind höchstens $k \cdot (n - 1) + 1$ Paare, da nur nach jeder Union-Operation jeweils c Paare auf S gelegt werden. Wird $|\Sigma| = k$ als Konstante angenommen, so ist die Laufzeit also in $\mathcal{O}(n \cdot G(n))$ (bzw. $\mathcal{O}(n \cdot \alpha(n, n))$).

Beispiel. Test auf Äquivalenz der Automaten von oben (S. 32).

Nach Makeset-Operationen: $\{a\}, \{b\}, \ldots, \{v\}, \ldots, \{z\}$

- $S : (a, v)$

  $\textsc{Find}(a) = \{a\} \neq \{v\} = \textsc{Find}(v)$

  $\xrightarrow{\textsc{Union}} \{a, v\}, \{b\}, \{c\}, \ldots, \{w\}, \{x\}, \{y\}, \{z\}$

- $S : (b, w), (f, z)$

  $\textsc{Find}(f) = \{f\} \neq \{z\} = \textsc{Find}(z)$

  $\xrightarrow{\textsc{Union}} \{a, v\}, \{b\}, \{c\}, \ldots, \{f, z\}, \ldots, \{w\}, \{x\}, \{y\}$

- $S : (b, w), (c, y), (g, x)$

  $\textsc{Find}(g) = \{g\} \neq \{x\} = \textsc{Find}(x)$

$\xrightarrow{\textsc{Union}} \{a,v\},\{b\},\{c\},\ldots,\{f,z\},\{g,x\}\ldots,\{w\},\{y\}$

- $S:(b,w),(c,y),(g,x),(e,v)$

  $\textsc{Find}(e)=\{e\}\neq\{a,v\}=\textsc{Find}(v)$

  $\xrightarrow{\textsc{Union}} \{a,e,v\},\{b\},\{c\},\{d\},\{f,z\},\{g,x\},\{h\},\{w\},\{y\}$

- $S:(b,w),(c,y),(g,x),(h,w),(f,z)$

  $\textsc{Find}(f)=\{f,z\}=\textsc{Find}(z)$

- $S:(b,w),(c,y),(g,x),(h,w)$

  $\textsc{Find}(h)=\{h\}\neq\{w\}=\textsc{Find}(w)$

  $\xrightarrow{\textsc{Union}} \{a,e,v\},\{b\},\{c\},\{d\},\{f,z\},\{g,x\},\{h,w\},\{y\}$

- $S:(b,w),(c,y),(g,x),(g,x),(c,y)$

  $\textsc{Find}(c)=\{c\}\neq\{y\}=\textsc{Find}(y)$

  $\xrightarrow{\textsc{Union}} \{a,e,v\},\{b\},\{c,y\},\{d\},\{f,z\},\{g,x\},\{h,w\}$

- $S:(b,w),(c,y),(a,v),(g,x),(g,x),(a,v),(c,y)$

- $S:(b,w)$

  $\textsc{Find}(b)=\{b\}\neq\{h,w\}=\textsc{Find}(w)$

  $\xrightarrow{\textsc{Union}} \{a,e,v\},\{b,h,w\},\{c,y\},\{d\},\{f,z\},\{g,x\}$

- $S=\emptyset$, Mengen $\{a,e,v\},\{b,h,w\},\{c,y\},\{d\},\{f,z\},\{g,x\}$.

Die Endzustände c und y sind nicht mit Nicht-Endzuständen zusammen in einer Menge, also ist $\mathcal{A}_1\equiv\mathcal{A}_2$.

Kapitel 4

Aufspannende Bäume minimalen Gewichts

Der Literaturtip. Der „Algorithmus von Kruskal“ ist in [Kru56] beschrieben. Bereits 1930 wurde von Jarník ein Algorithmus veröffentlicht (in tschechisch), der dem „Algorithmus von Prim“ entspricht. Später ist er unabhängig voneinander von Prim [Pri57] und Dijkstra [Dij59] wiederentdeckt worden. Die Färbungsmethode wird in [Tar83] von Tarjan beschrieben. Dort werden auch andere Varianten der Färbungsmethode angegeben. Folgen von UNION– und FIND–Operationen und Datenstrukturen vom Typ HEAP, sowie die effiziente Implementationen der Algorithmen von Kruskal und Prim unter Benutzung dieser Konzepte sind ebenfalls genauer in [Tar83] beschrieben. Matroide und deren Zusammenhang mit aufspannenden Bäumen sind etwa in [Jun94] zu finden.

4.1 Einführung

Wir benutzen folgende Grundbegriffe der Graphentheorie. Bezeichne das Paar $G = (V, E)$ einen **ungerichteten Graphen** mit endlicher **Knotenmenge** V und **Kantenmenge** $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$. Ein **Weg** in G ist eine Folge $v_1, v_2, \ldots, v_k$ von Knoten aus V, in der zwei aufeinanderfolgende Knoten durch eine Kante aus E verbunden sind. Ein Graph $G = (V, E)$ heißt **zusammenhängend**, wenn es zwischen je zwei Knoten $u, v \in V$ einen Weg in G gibt. Ein zusammenhängender Graph $B = (V(B), E(B))$ heißt **Baum**, wenn es zwischen je zwei Knoten aus $V(B)$ *genau einen* Weg in B gibt. Ein zusammenhängender Teilgraph $B = (V(B), E(B))$ von $G = (V, E)$, $E(B) \subseteq E$, heißt **aufspannend**, wenn $V(B) = V$.

Problem 4.1 ***Das MST–Problem.***[1]
Gegeben sei ein zusammenhängender Graph $G = (V, E)$ und eine Gewichtsfunktion $c : E \longrightarrow I\!R$. Finde einen aufspannenden Teilgraph $B = (V, E')$ von G, mit $E' \subseteq E$, der ein Baum ist und bezüglich c minimales Gewicht hat. Das heißt so, daß

$$c(B) = \sum_{\{u,v\} \in E'} c(\{u, v\})$$

minimal über alle aufspannenden Bäume in G ist.

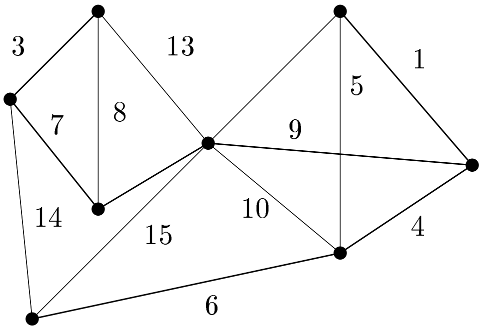


Abbildung 4.1: Ein aufspannender Baum minimalen Gewichts.

Motivation. Das MST–Problem ist ein Grundproblem der algorithmischen Graphentheorie, das viele Anwendungen hat, etwa beim Entwurf eines Netzwerkes, das geographisch verteilte Komponenten möglichst günstig miteinander verbinden soll, um beispielsweise Kommunikationsmöglichkeiten oder Infrastruktur zur Verfügung zu stellen.

Wir wollen effiziente Algorithmen zur Lösung des MST–Problems entwerfen. Alle aufspannenden Bäume in einem Graphen zu ermitteln und einen kostenminimalen daraus auszuwählen, ist sicher keine „effiziente“ Vorgehensweise: Man kann unter Benutzung der sogenannten „Prüfer–Korrespondenz“ beweisen, daß es in einem vollständigen Graphen mit n Knoten n^{n-2} aufspannende Bäume gibt. (Dies ist der Satz von Cayley.) Dazu zeigt man, daß es eine bijektive Abbildung zwischen der Menge aller aufspannenden Bäume über n Knoten und der Menge aller Worte der Länge $n-2$ über dem Alphabet $\{1, \ldots, n\}$ gibt.

Im nächsten Abschnitt formulieren wir eine allgemeine Vorgehensweise zur Lösung des MST–Problems, die alle bisher bekannten Algorithmen für das MST–Problem verallgemeinert. Diese allgemeine Methode, genannt

---

[1] MST steht für **M**inimal **S**panning **T**ree

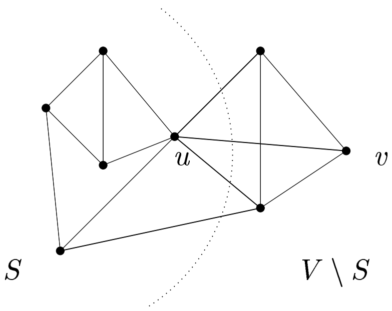
„Färbungsmethode“, ist von R. E. Tarjan eingeführt worden. Wir werden sehen, daß zwei klassische Algorithmen zur Lösung des MST–Problems, der „Algorithmus von Kruskal“ und der „Algorithmus von Prim“, nur spezielle Varianten der Färbungsmethode sind. Für diese beiden Algorithmen werden wir effiziente Implementationen skizzieren. Die Färbungsmethode kann als ein „Greedy–Verfahren“ im allgemeinen Sinne aufgefaßt werden. Es werden auf der Basis der bisher konstruierten Teillösung Kanten in die Lösung aufgenommen oder aus der Lösung ausgeschlossen. Diese Entscheidungen werden nachträglich nicht mehr rückgängig gemacht. Die Optimalität von Greedy–Verfahren basiert auf einer kombinatorischen Struktur, die im letzten Abschnitt kurz behandelt wird.

4.2 Die Färbungsmethode von Tarjan

Die Färbungsmethode färbt Kanten nacheinander *grün* oder *rot*. Am Ende bildet die Menge der grünen Kanten einen aufspannenden Baum minimalen Gewichts. Die Färbungen der Kanten sind Anwendungen von Regeln, einer **grünen Regel** oder einer **roten Regel.** Um diese Regeln zu formulieren, benötigen wir die Begriffe „Schnitt“ und „Kreis“ in einem Graphen.

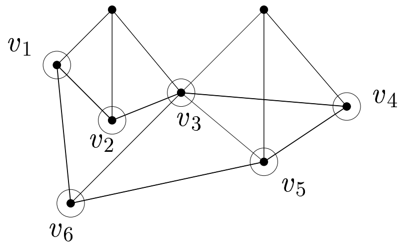
Definition 4.2 *Ein* ***Schnitt*** *in einem Graphen* $G = (V, E)$ *ist eine Partition* $(S, V \setminus S)$ *der Knotenmenge* V*. Eine Kante* $\{u, v\}$ ***kreuzt*** *den Schnitt* $(S, V \setminus S)$*, falls* $u \in S$ *und* $v \in V \setminus S$ *ist. Oft wird auch die Menge der Kanten, die den Schnitt* $(S, V \setminus S)$ *kreuzt, mit diesem Schnitt identifiziert.*

Beispiel. Ein Schnitt $(S, V \setminus S)$.



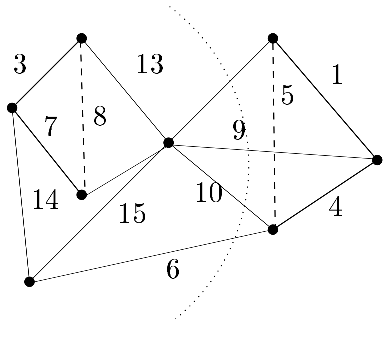
Definition 4.3 *Ein* ***Kreis*** *in einem Graphen* $G = (V, E)$ *ist eine Folge* $v_1, \ldots, v_k = v_1, k > 3$*, von Knoten aus* G*, in der zwei aufeinanderfolgende Knoten durch eine Kante verbunden sind, und kein Knoten außer dem Anfangs- und Endknoten zweimal auftritt.*

Beispiel. Die Folge $v_1, v_2, v_3, v_4, v_5, v_6, v_1$ ist ein Kreis.



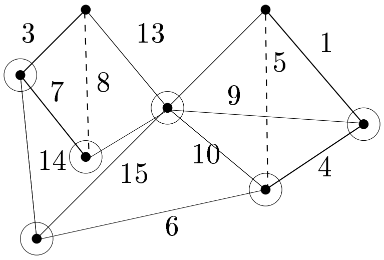
Grüne Regel. Wähle einen *Schnitt* in G, der von keiner grünen Kante gekreuzt wird. Unter allen ungefärbten Kanten, die diesen Schnitt kreuzen, wähle eine Kante *minimalen Gewichts* und färbe sie grün.

Beispiel. Eine Anwendung der grünen Regel: Die gestrichelten Kanten sind bereits rot, die breiten Kanten grün gefärbt. Die Kante mit Gewicht 6 wird grün gefärbt.



Rote Regel. Wähle einen *Kreis*, der keine rote Kante enthält. Unter allen ungefärbten Kanten, die auf diesem Kreis liegen, wähle eine Kante *maximalen Gewichts* und färbe sie rot.

Beispiel. Eine Anwendung der roten Regel: Wieder sind die gestrichelten Kanten bereits rot, die breiten Kanten grün gefärbt. Wir wählen den Kreis aus obigem Beispiel. Die Kante mit Gewicht 14 wird rot gefärbt.



Die Färbungsmethode von Tarjan

1. **Solange** noch eine der beiden Regeln anwendbar ist
2. wende die grüne oder die rote Regel an.

Diese Methode ist nichtdeterministisch. Im allgemeinen gibt es in einem Schleifendurchlauf verschiedene Wahlmöglichkeiten. Einerseits kann die Regel, welche angewandt wird, gewählt werden, andererseits die Kante, auf die die gewählte Regel angewandt wird. Die Behauptung ist, daß am Ende die Menge aller grün gefärbten Kanten einen aufspannenden Baum minimalen Gewichts in G induziert. Um die Korrektheit des Verfahrens zu beweisen, also die Behauptung, daß am Ende gerade die Menge aller grün gefärbten Kanten einen aufspannenden Baum minimalen Gewichts in G induziert, formulieren wir die folgende „Invariante“ für die Färbungsmethode.

Färbungsinvariante. Es gibt einen aufspannenden Baum minimalen Gewichts, der *alle grünen* Kanten und *keine rote* Kante enthält.

Wir werden beweisen, daß die Färbungsmethode die Färbungsinvariante erhält. Ist erst dann keine der beiden Regeln mehr anwendbar, wenn alle Kanten gefärbt sind, so folgt die Korrektheit des Färbungsalgorithmus aus der Färbungsinvariante.

Satz 4.1 Satz über die Färbungsinvariante
Die Färbungsmethode angewandt auf einen zusammenhängenden Graphen erhält die Färbungsinvariante. Nach jedem Färbungsschritt gibt es also einen aufspannenden Baum minimalen Gewichts, der alle grünen Kanten und keine rote Kante enthält.

Beweis. Wir beweisen den Satz über die Färbungsinvariante durch eine Induktion über die Anzahl m der Färbungsschritte.

Induktionsanfang ($m = 0$). Alle Kanten sind ungefärbt und jeder aufspannende Baum minimalen Gewichts erfüllt die Färbungsinvariante. Da der Graph zusammenhängend ist, existiert mindestens ein aufspannender Baum.

Induktionsschluß ($m \longrightarrow m+1$). Für den $(m+1)$–ten Färbungsschritt sind zwei Fälle zu unterscheiden. Er ist entweder eine Anwendung der grünen Regel oder eine Anwendung der roten Regel. Wir betrachten die Kante e, auf die der $(m+1)$–te Färbungsschritt angewandt wurde.

Fall 1: Der $(m+1)$–te Färbungsschritt ist eine Anwendung der grünen Regel auf die Kante e. Nach Induktionsvoraussetzung existiert nach dem m–ten Färbungsschritt ein aufspannender Baum B minimalen Gewichts, der alle grünen Kanten und keine rote Kante enthält. Ist e in B enthalten, so erfüllt B auch nach dem $(m+1)$–ten Färbungsschritt diese Bedingung.

Ist e nicht in B enthalten, so betrachte den Schnitt, auf den die grüne Regel im $(m+1)$–ten Färbungsschritt angewandt wurde. Da B zusammenhängend und aufspannend ist, muß es in B einen Weg geben, der die Endknoten von e enthält und mindestens eine Kante e', die den betrachteten Schnitt kreuzt. Da B keine rote Kante enthält und die grüne Regel auf den Schnitt mit e angewandt wird, ist e' ungefärbt. Wegen der Wahl von e ist $c(e') \geq c(e)$.

Durch Wegnahme von e' und Hinzunahme von e entsteht aus B dann wieder ein Baum B'. Dieser Baum B' ist wiederum ein aufspannender Baum minimalen Gewichts, der die Färbungsinvariante erfüllt.

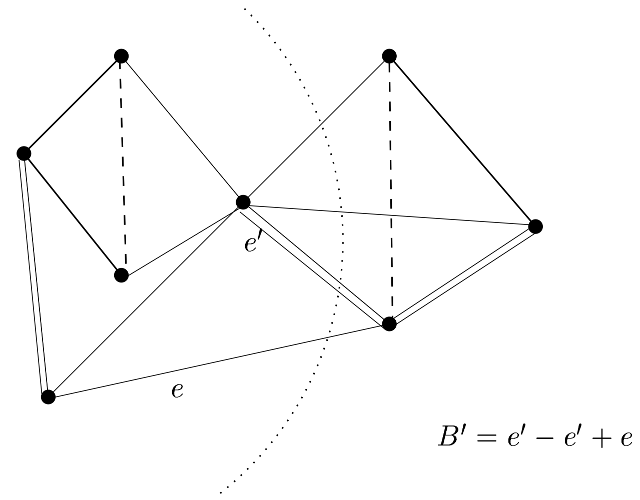


Abbildung 4.2: Die gestrichelten Kanten sind rot, die breiten Kanten grün gefärbt. Die grünen Kanten bilden zusammen mit den doppelt gezeichneten Kanten den Baum B. Durch Austausch von e' und e erhalten wir aus B einen aufspannenden Baum B' mit kleinerem Gewicht, der wiederum die Färbungsinvariante erfüllt.

Fall 2: Der $(m+1)$–te Färbungsschritt ist eine Anwendung der roten Regel auf die Kante e. Sei B wieder der nach Induktionsvoraussetzung nach dem m–ten Färbungsschritt existierende aufspannende Baum minimalen Gewichts, der die Färbungsinvariante erfüllt. Falls e nicht in B ist, so erfüllt B auch nach dem $(m+1)$–ten Färbungsschritt die Färbungsinvariante.

Ist e in B enthalten, so zerfällt B nach Wegnahme von e in zwei Teilbäume, deren Knotenmengen einen Schnitt im Graphen induzieren, der von e gekreuzt wird. Betrachte den Kreis, auf den die rote Regel im $(m+1)$–ten Färbungsschnitt angewandt wurde. Auf diesem Kreis liegt eine Kante $e' \neq e$, die ebenfalls den Schnitt kreuzt und nicht rot gefärbt ist. Die Kante e' ist auch nicht grün gefärbt, da nach Definition des Schnittes nicht beide, e und e', zu B gehören können. Da die rote Regel im $(m+1)$–ten Färbungsschritt auf e angewandt wird, ist $c(e) \geq c(e')$.

Der Baum B', der aus B durch Wegnahme von e und Hinzunahme von e' entsteht, ist dann wieder ein aufspannender Baum minimalen Gewichts, der die Färbungsinvariante erfüllt. ■

Satz 4.2 *Die Färbungsmethode färbt alle Kanten eines zusammenhängenden Graphen rot oder grün.*

Beweis. Falls die Färbungsmethode endet bevor alle Kanten gefärbt sind, so existiert einerseits eine Kante e, die nicht gefärbt ist, andererseits ist weder die rote noch die grüne Regel anwendbar. Da die Färbungsinvariante erfüllt ist, induzieren die grünen Kanten eine Menge von „grünen Bäumen“ (wobei jeder Knoten als „grün“ aufgefaßt wird).

Fall 1. Beide Endknoten der ungefärbten Kante e liegen in demselben grünen Baum. Dann bildet e zusammen mit dem Weg in diesem Baum, der die Endknoten von e verbindet, einen Kreis, auf den die rote Regel anwendbar ist.

Fall 2. Die Endknoten von e liegen in verschiedenen grünen Bäumen. Dann existiert ein Schnitt, der von e gekreuzt wird, und auf den die grüne Regel anwendbar ist. Betrachte dazu einfach einen der Schnitte, die durch die beiden grünen Bäume, in denen die Endknoten von e liegen, induziert wird. ■

4.3 Der Algorithmus von Kruskal

Der Algorithmus von Kruskal läßt sich nun einfach als eine Variante der Färbungsmethode formulieren.

Eingabe: Graph $G = (V, E)$

Ausgabe: Aufspannender Baum minimalen Gewichts.

Algorithmus von Kruskal

```
1. Sortiere die Kanten nach ihrem Gewicht in nicht–absteigender Reihenfolge.
2. Durchlaufe die sortierten Kanten der Reihe nach, und wende folgenden Färbungsschritt an:
3.     Wenn beide Endknoten der Kante in demselben grünen Baum liegen, so färbe sie rot;
4.     ansonsten färbe sie grün.
```

Der Algorithmus endet, wenn alle Kanten durchlaufen sind. Der Algorithmus von Kruskal ist offensichtlich eine spezielle Version der Färbungsmethode, denn jeder Färbungsschritt ist eine Anwendung der grünen oder roten Regel. Betrachte dazu die nächste Kante e in der sortierten Kantenfolge.

Färbe rot: Wird die Kante e rot gefärbt, so liegen ihre beiden Endknoten in demselben grünen Baum. Sie schießt also einen Kreis, der keine rote Kante enthält und e als einzige ungefärbte Kante. Damit ist dieser Färbungsschritt eine Anwendung der roten Regel.

Färbe grün: Wird die Kante e grün gefärbt, so liegen ihre beiden Endknoten nicht in demselben grünen Baum. Damit induziert sie einen Schnitt, der von keiner anderen grünen Kante gekreuzt wird. Wegen der Sortierung der Kanten ist dieser Färbungsschritt eine Anwendung der grünen Regel.

In einer Implementation des Algorithmus von Kruskal müssen zwei Teilschritte effizient realisiert werden: die Sortierung der Kanten entsprechend ihrem Gewicht und die Organisation der grünen Bäume in einer Form, die den Test „beide Endknoten einer Kante liegen in demselben grünen Baum“ unterstützt. Die Sortierung der Kanten kann in Laufzeit $\mathcal{O}(|E| \log |E|) = \mathcal{O}(|E| \log |V|)$ vorgenommen werden. Die Organisation der (sich verändernden) grünen Bäume wird als Folge von UNION- und FIND-Operationen realisiert:

- FIND: Finde die grünen Bäume, in denen die beiden Endknoten der zu färbenden Kante liegen.
- UNION: Vereinige die beiden grünen Bäume, in denen die beiden Endknoten einer Kante liegen, die grün gefärbt wird.

Wenn eine sortierte Kantenliste gegeben ist, so kann diese Folge von UNION- und FIND-Operationen in Laufzeit $\mathcal{O}(|E| \cdot \alpha(|E|, |V|))$ realisiert werden, wobei $\alpha(|E|, |V|)$ die sehr langsam wachsende Funktion aus Abschnitt 3.1.4, S. 29 ist. Die Gesamtlaufzeit wird also durch das Sortieren dominiert. In Situationen, in denen die Kanten bereits sortiert vorliegen, oder die Kantengewichte so sind, daß sie „schneller“ als in $\mathcal{O}(|E| \log |V|)$ sortiert werden können, ist auch die Gesamtlaufzeit in $\mathcal{O}(|E| \cdot \alpha(|E|, |V|))$.

4.4 Der Algorithmus von Prim

Der Algorithmus von Prim läßt sich ebenfalls als spezielle Variante der Färbungsmethode formulieren.

Eingabe: Graph $G = (V, E)$

Ausgabe: Aufspannender Baum minimalen Gewichts.

Algorithmus von Prim

```
1. Wähle einen beliebigen Startknoten und betrachte diesen als einen
   “grünen Baum”.
2. Wiederhole den folgenden Färbungsschritt $(|V| - 1)$–mal:
3.     Wähle eine ungefärbte Kante minimalen Gewichts, die genau
       einen Endknoten in dem grünen Baum hat, der den Startkno-
       ten enthält, und färbe sie grün.
```

Der Algorithmus von Prim ist offensichtlich eine spezielle Version der Färbungsmethode, denn jeder Färbungsschritt ist eine Anwendung der grünen Regel. Der Schnitt, auf den die grüne Regel angewandt wird, wird jeweils von den Knoten des grünen Baumes induziert, der den Startknoten enthält. Nach genau $|V| - 1$ solcher Färbungsschritte ist nur noch die rote Regel anwendbar, denn zu Beginn können wir die $|V|$ Knoten als $|V|$ disjunkte grüne Bäume auffassen. Mit jedem Färbungsschritt reduziert sich die Anzahl der disjunkten grünen Bäume um genau einen. Nach $|V| - 1$ Färbungsschritten muß also genau ein grüner Baum übrig sein.

Implementation des Algorithmus von Prim. Die Idee der Implementation besteht darin, Kanten, die „Kandidaten“ dafür sind, grün gefärbt zu werden, geeignet zu verwalten. Dazu sei zu einem beliebigen Zeitpunkt des Algorithmus B der grüne Baum, der den Startknoten enthält. Ein Knoten u *begrenzt* B genau dann, wenn u nicht in B ist, aber eine Kante $\{u, w\}$ in G existiert, für die w in B ist. Zu jedem Knoten u, der B begrenzt, betrachte eine Kante mit minimalem Gewicht, die u und B verbindet und nenne sie *hellgrün*. Die hellgrünen Kanten sind Kandidaten als nächstes grün zu werden, und die grünen und hellgrünen Kanten bilden B und seine begrenzenden Knoten.

In einer Implementation des Algorithmus von Prim besteht eine Anwendung des Färbungsschritts aus:

- Wahl einer hellgrünen Kante minimalen Gewichts. Diese Kante wird grün gefärbt und damit ein neuer Knoten u in B eingefügt.
- Betrachte alle Kanten $\{u, w\}$. Falls w nicht in B ist und zu keiner hellgrünen Kante inzident ist, färbe $\{u, w\}$ hellgrün; falls w nicht in B, aber zu einer hellgrünen Kante $\{w, x\}$ größeren Gewichts inzident ist, so färbe diese rot und $\{u, w\}$ hellgrün; ansonsten färbe $\{u, w\}$ rot.

Basierend auf dieser Idee ist der Algorithmus von Prim in Laufzeit $\mathcal{O}(|V^2|)$ direkt realisierbar. Benutzt man etwa einen Heap zur Verwaltung der hellgrünen Kanten, so kann der Algorithmus in $\mathcal{O}(m \cdot \log_{2+m/n} n)$ realisiert werden (wobei $m = |E|$ und $n = |V|$). Falls $m \in \Omega(n^{1+\epsilon})$ ist, ist dies in $\mathcal{O}(m/\epsilon)$. Der Algorithmus von Prim ist daher gut geeignet für „dichte“ Graphen, also für Graphen mit $|E| \in \Omega(|V|^{1+\epsilon}), \epsilon > 0$. Er ist schlechter als der Algorithmus von Kruskal, wenn die Kanten vorsortiert sind.

4.5 Greedy–Verfahren und Matroide

Wir haben in dem Korrektheitsbeweis zur Färbungsmethode mehrfach ein *Austauschargument* benutzt. Dabei haben wir aus einem aufspannenden Baum minimalen Gewichts durch Austausch einer Kante des Baumes durch eine geignete Kante, die nicht zum Baum gehört, wieder einen aufspannenden Baum minimalen Gewichts konstruiert. Dieses Austauschargument basiert auf einer interessanten strukturellen Eigenschaft von Bäumen in Graphen. Die Mengen von Kanten eines Graphen, welche Bäume induzieren, bilden eine spezielle kombinatorische Struktur, ein „Matroid“.

Definition 4.4 *Ein Mengensystem $\mathcal{U}$ über einer endlichen Menge M heißt* ***Unabhängigkeitssystem****, wenn*

- $\emptyset \in \mathcal{U}$
- $I_1 \in \mathcal{U}, I_2 \subseteq I_1$ *induziert* $I_2 \in \mathcal{U}$

Matroide sind nun Unabhängigkeitssysteme, die eine gewisse Austauscheigenschaft erfüllen.

Definition 4.5 *Ein Unabhängigkeitssystem* $\mathcal{U}$ *ist ein* ***Matroid****, falls es folgende „Austauscheigenschaft" erfüllt:*

- *Falls* $I, J \subseteq M$, $I, J \in \mathcal{U}$ *mit* $|I| = |J|$*, dann existiert ein* $e \in I$ *und ein* $e' \in J$ *mit* $(I \setminus \{e\}) \cup \{e'\} \in \mathcal{U}$.

Man kann nun beweisen, daß ein Greedy–Verfahren genau dann eine optimale Lösung über einem Mengensystem liefert, wenn das Mengensystem ein Matroid ist. Offensichtlich bilden die Mengen von Kanten eines Graphen, welche Mengen von Bäumen induzieren, ein Unabhängigkeitssystem über der Kantenmenge E. Denn einerseits induziert die leere Menge einen Baum, andererseits induziert jede Teilmenge einer Kantenmenge, die eine Menge von Bäumen induziert, ebenfalls eine Menge von Bäumen. Mit dem Austauschargument, das wir im Korrektheitsbeweis der Färbungsmethode angewandt haben, kann man beweisen, daß dieses Unabhängigkeitssystem sogar ein Matroid bildet.

Kapitel 5

Schnitte in Graphen und Zusammenhang

Der Literaturtip. Schnitte in Graphen und Zusammenhang werden in [SW94, NI92, FIN93] beschrieben.

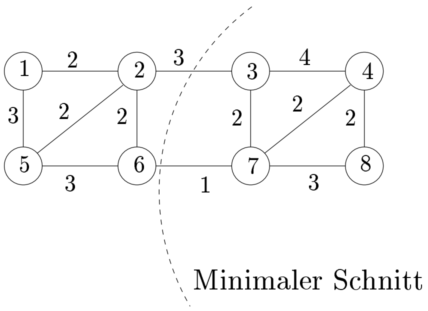
5.1 Schnitte minimalen Gewichts: MinCut

Problem. MinCut. Gegeben sei ein Graph $G = (V, E)$ mit einer Kantengewichtsfunktion $c : E \longrightarrow \mathbb{R}_0^+$. Finde einen *nichttrivialen Schnitt* $(S, V \backslash S)$ *minimalen Gewichts in* G, d.h. finde $S \subseteq V, \emptyset \neq S \neq V$, so daß

$$c(S, V \setminus S) := \sum_{\substack{\{u,v\} \in E, \\ u \in S, \\ v \in V \setminus S}} c(\{u, v\})$$

minimal wird. $(S, V \backslash S)$ wird **minimaler Schnitt** genannt.

Beispiel.



Bemerkung

> Mit einem Flußalgorithmus (Ford & Fulkerson, Goldberg & Tarjan) kann man zu gegebenen s und t einen minimalen s–t–Schnitt bestimmen. Einen minimalen Schnitt allgemein kann man also durch $|V|^2$ Durchläufe eines Flußalgorithmus (für alle möglichen $s, t \in V$) berechnen oder sogar effizienter durch $|V| - 1$ Durchläufe, indem man ein s festhält. In diesem Kapitel wollen wir einen minimalen Schnitt ohne Anwendung von Flußmethoden berechnen. Der hier behandelte Algorithmus ist gleichzeitig (etwas) effizienter als $|V|-1$–maliges Anwenden des effizientesten bekannten Flußalgorithmus.

Wir benutzen folgende Definitionen:

Definition 5.1 *Zu $S \subseteq V$ und $v \notin S$ sei*

$$c(S,v) := \sum_{\substack{\{u,v\} \in E \\ u \in S}} c(\{u,v\}).$$

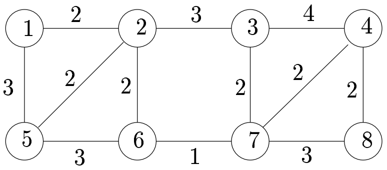
Den Knoten $v \in V \setminus S$, für den $c(S,v)$ maximal wird, nennen wir auch den ***am stärksten mit S verbundenen Knoten****.*

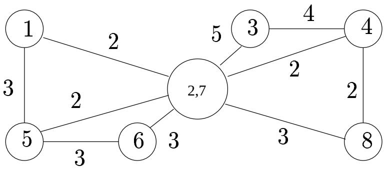
Definition 5.2 (Verschmelzen zweier Knoten) *Seien $s, t \in V$. Dann werden s und t* ***verschmolzen****, indem ein neuer Knoten $x_{s,t}$ eingeführt wird, s und t gelöscht werden, alle Nachbarn von s und t zu Nachbarn von $x_{s,t}$ werden und gegebenenfalls Kantengewichte von Kanten, die inzident zu s oder t waren, addiert werden. Falls zuvor $\{s,t\}$ eine Kante war, wird diese ebenfalls gelöscht.*

Formal. Sei $G = (V,E)$, $c : E \longrightarrow \mathbb{R}_0^+$, $s,t \in V$, $s \neq t$. Durch Verschmelzen von s und t wird G in $G' = (V', E')$ und die Kantengewichtsfunktion in $c : E' \longrightarrow \mathbb{R}_0^+$ transformiert mit

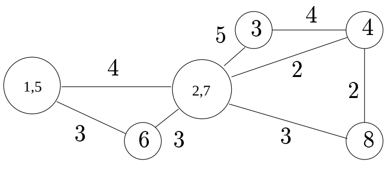
$$\begin{aligned}
V' &:= \big(V \setminus \{s,t\}\big) \cup \{x_{s,t}\} \text{ mit } x_{s,t} \notin V \\
E' &:= E \setminus \{\{u,v\} \in E : u = s \text{ oder } u = t\} \cup \\
& \{\{x_{s,t}, v\} : \{s,v\} \in E \text{ oder } \{t,v\} \in E \text{ und } v \in V \setminus \{s,t\}\} \\
c(\{x_{s,t}, v\}) &:= \begin{cases} c(\{s,v\}), & \text{falls } \{s,v\} \in E \text{ und } \{t,v\} \notin E \\ c(\{t,v\}), & \text{falls } \{t,v\} \in E \text{ und } \{s,v\} \notin E\ . \\ c(\{s,v\}) + c(\{t,v\}), & \text{falls } \{s,v\}, \{t,v\} \in E. \end{cases}
\end{aligned}$$

Beispiel.





verschmelze 2 mit 7



verschmelze 1 mit 5

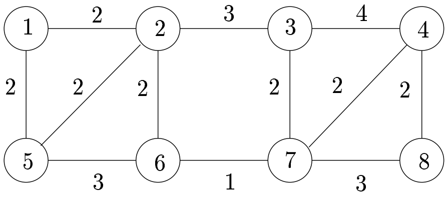
5.1.1 Der Algorithmus von **Stoer & Wagner**

Der folgende Algorithmus wurde von Stoer & Wagner (1994) veröffentlicht und basiert teilweise auf Ideen von Nagamochi & Ibaraki (1992).

Idee. Der Algorithmus besteht aus $|V| - 1$ Phasen. In der i–ten Phase wird in einem Graph G_i ein Schnitt berechnet – der **Schnitt der Phase** i. G_i entsteht aus dem Graphen G_{i-1}, der der vorherigen Phase zugrunde lag, durch Verschmelzen „geeigneter Knoten“ s und t. Der Schnitt der Phase i $(S_i, V_i \setminus S_i)$ wird mit einer Prozedur berechnet, die dem Algorithmus von PRIM für MST entspricht. Ausgehend von einem Startknoten a wird in jedem Schritt der am stärksten mit S_i verbundene Knoten zu S_i hinzugefügt, wobei zu Beginn $S_i := \{a\}$ ist. Die zu verschmelzenden „geeigneten Knoten“ s und t der Phase i sind die beiden letzten Knoten, die zu S_i hinzugefügt werden. Schnitt der Phase i ist $S_i := (V \setminus \{t\}, \{t\})$. Ergebnis des ganzen Algorithmus ist der minimale Schnitt aller Schnitte der Phasen i $(1 \leq i \leq |V| - 1)$.

Beispiel. Der Startknoten sei 2.

- 1. Phase



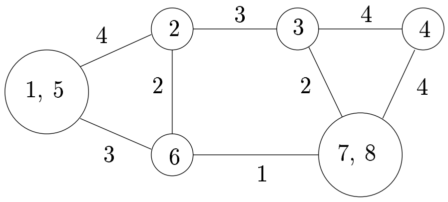
  - $G_1 := G$.
  - $S_1 := \{2\}$
  - $S_1 = \{2,3\}$.
  - $S_1 = \{2,3,4\}$
  - $S_1 = \{2,3,4,7\}$
  - $S_1 = \{2,3,4,7,8\}$
  - $S_1 = \{2,3,4,7,8,6\}$
  - $S_1 = \{2,3,4,7,8,6,5\} \Longrightarrow s = 5$
  - $S_1 = V_1,$ also $t = 1$
  - Schnitt der ersten Phase ist also $(V_1 \setminus \{1\}, \{1\})$ mit Gewicht 5.

- 2. Phase

width width width width width width width

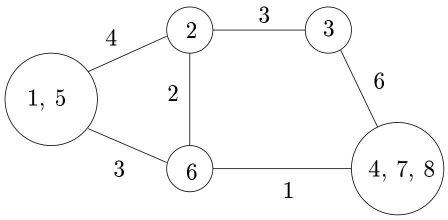
  - $S_2 := \{2\}$
  - $S_2 = \{2, \{1,5\}\}$
  - $S_2 = \{2, \{1,5\}, 6\}$
  - $S_2 = \{2, \{1,5\}, 6, 3\}$
  - $S_2 = \{2, \{1,5\}, 6, 3, 4\}$
  - $S_2 = \{2, \{1,5\}, 6, 3, 4, 7\} \Longrightarrow s = 7$
  - $S_2 = V_2,$ also $t = 8$
  - Schnitt der zweiten Phase ist also $(V_2 \setminus \{8\}, \{8\})$ mit Gewicht 5.

- 3. Phase



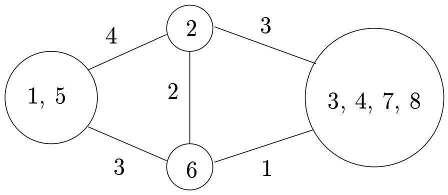
  - $S_3 := \{2\}$
  - $S_3 = \{2, \{1,5\}\}$
  - $S_3 = \{2, \{1,5\}, 6\}$
  - $S_3 = \{2, \{1,5\}, 6, 3\}$
  - $S_3 = \{2, \{1,5\}, 6, 3, 4\} \Longrightarrow s = 4$
  - $S_3 = V_3, t = \{7,8\}$
  - Schnitt der dritten Phase ist also $(V_3 \setminus \{7,8\}, \{7,8\})$ mit Gewicht 7.

- 4. Phase



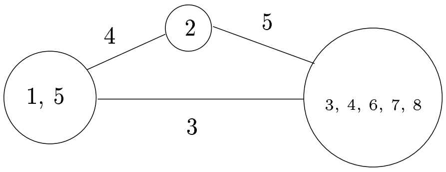
  - $S_4 := \{2\}$
  - $S_4 = \{2, \{1,5\}\}$
  - $S_4 = \{2, \{1,5\}, 6\}$
  - $S_4 = \{2, \{1,5\}, 6, 3\} \Longrightarrow s = 3$
  - $S_4 = V_4, t = \{4,7,8\}$
  - Schnitt der vierten Phase ist also $(V_4 \setminus \{4,7,8\}, \{4,7,8\})$ mit Gewicht 7.

- 5. Phase



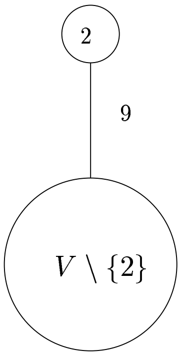
  - $S_5 := \{2\}$
  - $S_5 = \{2, \{1,5\}\}$
  - $S_5 = \{2, \{1,5\}, 6\} \Longrightarrow s = 6$
  - $S_5 = V_5, t = \{3,4,7,8\}$
  - Schnitt der fünften Phase ist also $(V_5 \setminus \{3,4,7,8\}, \{3,4,7,8\})$ mit Gewicht 4.

- 6. Phase



  - $S_6 := \{2\}$
  - $S_6 = \{2, \{3,4,6,7,8\}\} \Longrightarrow s = \{3,4,6,7,8\}$
  - $S_6 = V_6, t = \{1,5\}$
  - Schnitt der sechsten Phase ist also $(V_6 \setminus \{1,5\}, \{1,5\})$ mit Gewicht 7.

- 7. Phase



  - $S_7 := \{2\} \Longrightarrow S = 2$
  - $S_7 = V_7, t = V \setminus \{2\}$
  - Schnitt der siebten Phase ist also $(V_7 \setminus \{2\}, \{2\})$ mit Gewicht 9.

Der minimale Schnitt unter allen Schnitten der Phasen ist der Schnitt der fünften Phase mit Gewicht 4. Dieser Schnitt $(V_5 \setminus \{3,4,7,8\}, \{3,4,7,8\})$ in G_5 **induziert in G** den Schnitt $(\{1,2,5,6\}, \{3,4,7,8\})$.

Formale Beschreibung des Algorithmus.

Prozedur: MINSCHNITTPHASE(G_i, c, a)

```
1. Setze $S := \{a\}$, $t := a$
2. Solange $S \neq V_i$ ist, führe aus
3.     Bestimme Knoten $v \in V \setminus S$ mit $c(S, v)$ maximal und setze
       $S := S \cup \{v\}$
4.     Setze $s := t$ und $t := v$
5. Speichere $(V_i \setminus \{t\}, \{t\})$ als SCHNITT–DER–PHASE
6. Konstruiere aus $G_i$ Graph $G_{i+1}$ durch Verschmelzen von $s$ und $t$.
```

Algorithmus: MIN–SCHNITT(G, c, a)

```
1. Setze $G_1 := G$
2. Für $i = 1$ bis $|V| - 1$ führe aus
3.     MINSCHNITTPHASE($G_i, c, a$)
4.     Falls der SCHNITT–DER–PHASE kleiner ist als MIN–SCHNITT (der
       bisher minimale SCHNITT–DER–PHASE), dann
5.         speichere SCHNITT–DER–PHASE als MIN–SCHNITT
6. Gib MIN–SCHNITT aus.
```

Laufzeit. Die Prozedur MINSCHNITTPHASE kann genauso wie der Algorithmus von Prim implementiert werden, wobei nur anstatt eines Minimums jeweils ein Maximum berechnet und am Ende G_{i+1} konstruiert werden muß. Mit einem HEAP kann Prozedur MINSCHNITTPHASE in $\mathcal{O}(|E| + |V| \log |V|)$ ausgeführt werden. Dazu werden die Knoten außerhalb der Menge S in einem HEAP verwaltet, in dessen Wurzel das Element mit maximalem Schlüsselwert steht. Der Schlüsselwert von $v \in V \setminus S$ ist jeweils $c(S, v)$ zur aktuellen Menge S. Wird ein Knoten zu S eingefügt, so wird die Wurzel des HEAP gelöscht, die Schlüsselwerte der Knoten im HEAP jeweils ggf. entsprechend erhöht und Knoten an die richtige Stelle im HEAP bewegt. Ein Aufruf von MINSCHNITTPHASE erfordert also maximal $|V|$-mal DELETEMAX und höchstens $|E|$-mal „erhöhe Schlüsselwert“ mit entsprechender Knotenbewegung (im HEAP ausgeführt).

Dies ist in $\mathcal{O}(|V| \log |V| + |E|)$ möglich. Damit ist der Aufwand für MINSCHNITT insgesamt in $\mathcal{O}(|V|^2 \log |V| + |V||E|)$.

Korrektheit des Algorithmus. Für $s, t \in V$, $s \neq t$ nenne den Schnitt $(S, V \setminus S)$ mit $s \in S$ und $t \in V \setminus S$ einen ***s*–*t*–Schnitt**. Ein s–t–Schnitt **trennt** Knoten u und v, wenn $u \in S$ und $v \in V \setminus S$.

Lemma 5.1 *Zu $G = (V, E)$ und $c : E \longrightarrow \mathbb{R}_0^+$ gilt für die Prozedur* MINSCHNITTPHASE(G, c, a) *mit beliebigem $a \in V$, daß der berechnete* SCHNITT-DER-PHASE *minimal ist unter allen s–t–Schnitten, wobei s und t vorletzter bzw. letzter betrachteter Knoten ist.*

Beweis. Sei $(S, V \setminus S)$ SCHNITT-DER-PHASE, s und t vorletzter und letzter betrachteter Knoten. MINSCHNITTPHASE betrachtet die Knoten aus V entsprechend einer „linearen Ordnung“, die mit a beginnt und mit s und t endet. Betrachte einen beliebigen s-t-Schnitt $(S', V \setminus S')$. Wir zeigen, daß $c(S', V \setminus S') \geq c(S, V \setminus S)$ ist. Nenne einen Knoten $v \in V$ **aktiv** (bzgl. S'), wenn v und der Knoten, der unmittelbar vor v von MINSCHNITTPHASE zu S hinzugefügt wurde, von S' getrennt werden. Zu $v \in V \setminus \{a\}$ sei S_v die Menge aller Knoten, die vor v zu S hinzugefügt wurden und $S'_v := S' \cap (S_v \cup \{v\})$.

Für alle aktiven Knoten v gilt

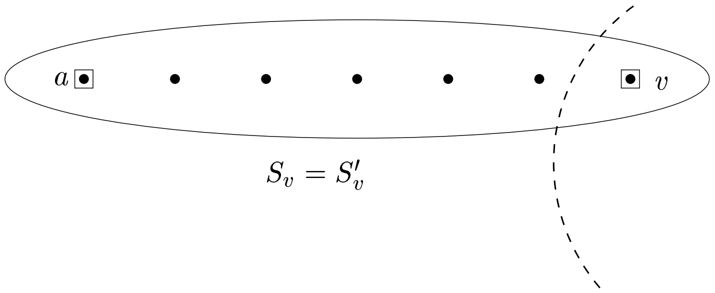
$$c(S_v, v) \leq c(S'_v, V \setminus S' \cap (S_v \cup \{v\})).$$

Dies ist der durch S' induzierte Schnitt in dem durch $S_v \cup \{v\}$ induzierten Graphen.

Der Beweis wird durch Induktion über die aktiven Knoten v in der Reihenfolge, in der sie zu S hinzugefügt werden, geführt.

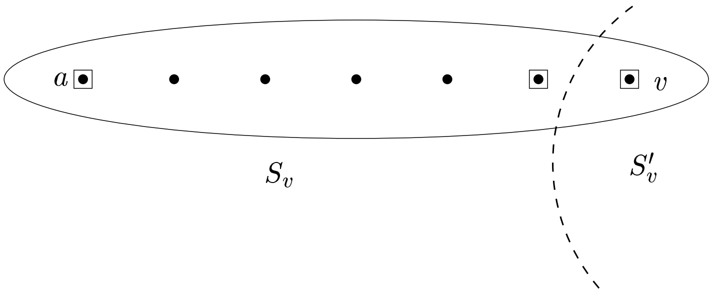
Induktionsanfang. Sei v erster aktiver Knoten.

- $v \notin S'$:



Es gilt: $S'_v = S_v$ und $V \setminus S' \cap (S_v \cup \{v\}) = \{v\}$.

- $v \in S'$:

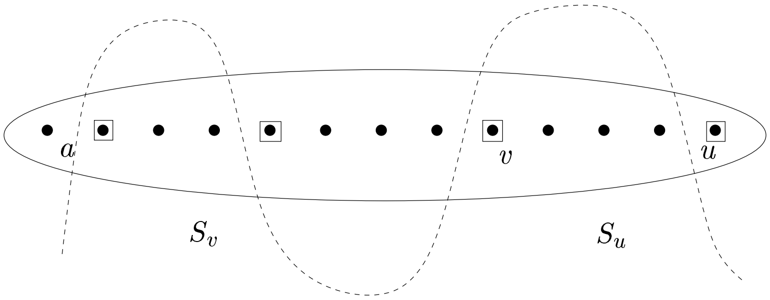


Es gilt: $S'_v = \{v\}$ und $V \setminus S' \cap (S_v \cup \{v\}) = S_v$.

In beiden Fällen gilt:

$$c(S_v, v) = c(S'_v, V \setminus S' \cap (S_v \cup \{v\}).$$

Induktionsschritt. Angenommen, die Behauptung gelte für alle aktiven Knoten bis zum Knoten v und u sei der nächste aktive Knoten.



Dann folgt

$$c(S_u, u) = c(S_v, u) + c(S_u \setminus S_v, u).$$

Da v vor u zu S hinzugefügt wurde, gilt

$$c(S_v, u) \leq c(S_v, v).$$

Außerdem ist nach Induktionsannahme

$$c(S_v, v) \leq c(S'_v, V \setminus S' \cap (S_v \cup \{v\})).$$

Alle Kanten zwischen $S_u \setminus S_v$ und u verbinden die verschiedenen Seiten von S'. Daher tragen sie zu $c(S'_u, V \setminus S' \cap (S_u \cup \{u\}))$ bei. Es gilt also

$$\begin{aligned} c(S_u, u) &\leq c(S'_v, V \setminus S' \cap (S_v \cup \{v\})) + c(S_u \setminus S_v, u) \\ &\leq c(S'_u, V \setminus S' \cap (S_u \cup \{u\})). \end{aligned}$$

Da t ein aktiver Knoten ist bzgl. $(S', V \setminus S')$, folgt

$$c(S, V \setminus S) = c(S_t, t) \leq c(S'_t, V \setminus S' \cap (S_t \cup \{t\})) = c(S', V \setminus S').$$

■

Satz 5.1 *Der minimale Schnitt von allen Ergebnissen der* $|V|-1$ *Ausführungen von* MINSCHNITTPHASE *ist ein minimaler nichttrivialer Schnitt in* $G = (V, E)$ *mit* $|V| \geq 2$.

Beweis. Induktion über $|V|$.
$|V| = 2$ ist trivial.
Sei $|V| \geq 3$. Betrachte Phase 1: Falls G einen nichttrivialen minimalen Schnitt hat, der s und t (vorletzter bzw. letzter Knoten der Phase 1) trennt, so ist der SCHNITT-DER-PHASE 1 nach Lemma ein minimaler nichttrivialer Schnitt. Wenn es jedoch keinen nichttrivialen minimalen Schnitt in G gibt, der s und t trennt, so müssen in jedem minimalen Schnitt s und t in derselben Menge liegen. Der Graph G', der aus G durch Verschmelzen von s und t entsteht, hat also einen Schnitt, der gleichzeitig einen minimalen Schnitt in G induziert. Es genügt also, einen minimalen Schnitt von G' zu bestimmen. Diesen bestimmt der Algorithmus nach Induktionsannahme mit dem Durchlaufen der Phasen 2 bis $|V|-1$, da $G' = (V', E')$ die Ungleichung $|V'| < |V|$ erfüllt. ■

Bemerkung. Bei allgemeinerer Kantengewichtsfunktion $c : E \longrightarrow \mathbb{R}$ (d.h. negative Gewichte sind zugelassen) ist das MINCUT-Problem im allgemeinen $\mathcal{NP}$-schwer. Ebenso ist das „duale“ MAXCUT-Problem im allgemeinen $\mathcal{NP}$-schwer.

Das ungewichtete MINCUT-Problem ist äquivalent zu folgendem grundlegenden Graphenproblem:

> Was ist die minimale Anzahl an Kanten in $G = (V, E)$, deren Wegnahme einen unzusammenhängenden Graphem induziert?

Der entsprechende Wert wird auch **Kanten-Zusammenhangs-Zahl** genannt.

Kapitel 6

Algorithmen für grundlegende geometrische Probleme

> **Der Literaturtip.** Zum Thema „Algorithmen für grundlegende algorithmische Probleme“ siehe auch [CLR90].

In diesem Kapitel beschäftigen wir uns mit grundlegenden Techniken zur Behandlung von Fragestellungen, denen als Eingabe geometrische Objekte zugrundeliegen. Solche geometrischen Objekte sind etwa Mengen von Punkten, Mengen von Liniensegmenten oder die Eckpunkte eines Polygons. Typische algorithmische Aufgaben sind:

- Schneiden sich gewisse Liniensegmente?
- Welches Punktepaar unter angegebenen Punkten liegt am nächsten zueinander?
- Finde die konvexe Hülle (d.h. das kleinste umschließende konvexe Polygon) einer Punktemenge.

Wir werden uns auf solche Fragen im zweidimensionalen Raum, d.h. in der Ebene, beschränken. Ein **geometrisches Objekt** ist dort typischerweise durch eine **Menge von Punkten** p_i repräsentiert, wobei $p_i = (x_i, y_i)$ durch seine Koordinaten $x_i, y_i \in \mathbb{R}$ festgelegt ist.

Beispielsweise ist ein **Polygon** P mit n Ecken durch eine Folge von Punkten $\langle p_0, p_1, \ldots, p_{n-1} \rangle$ repräsentiert, wobei diese in der Reihenfolge ihres Auftretens auf dem Rand von P gegeben sind.

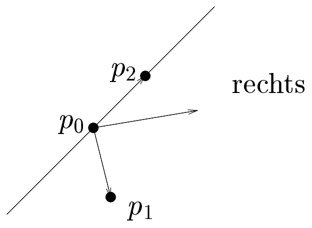
6.1 Einige Grundbegriffe

Definition 6.1 *Eine* ***konvexe Kombination*** *zweier verschiedener Punkte* $p_1 = (x_1, y_1)$ *und* $p_2 = (x_2, y_2)$ *ist ein Punkt* $p = (x, y)$*, für den es ein* $\alpha \in [0,1]$ *gibt, so dass* $x = \alpha x_1 + (1-\alpha)x_2$ *und* $y = \alpha y_1 + (1-\alpha)y_2$*. Schreibe auch* $p = \alpha p_1 + (1-\alpha)p_2$*. Jeder beliebige Punkt* p*, der auf einer Geraden liegt, die durch* p_1 *und* p_2 *gegeben ist und zwischen* p_1 *und* p_2 *oder auf einem der beiden Punkte liegt, ist eine konvexe Kombination aus* p_1 *und* p_2*.*

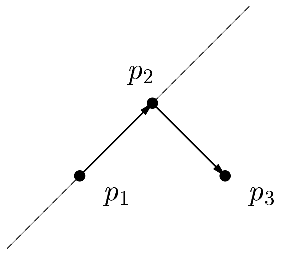
Definition 6.2 *Die Menge aller Punkte, die konvexe Kombination aus* p_1 *und* p_2 *sind, heißt das* ***Liniensegment*** $\overline{p_1p_2}$*; dabei sind* p_1 *und* p_2 *die Endpunkte von* $\overline{p_1p_2}$*. Ist die Reihenfolge von* p_1 *und* p_2 *von Bedeutung, so sprechen wir von dem* ***gerichteten Liniensegment*** $\overrightarrow{p_1p_2}$*.*

Zunächst behandeln wir folgende Fragen:

1. Gegeben zwei gerichtete Liniensegmente $\overrightarrow{p_0p_1}$ und $\overrightarrow{p_0p_2}$. Ist $\overrightarrow{p_0p_1}$ „rechts von“ (im Uhrzeigersinn) $\overrightarrow{p_0p_2}$ in bezug auf p_0?



2. Gegeben zwei Liniensegmente $\overline{p_1p_2}$ und $\overline{p_2p_3}$. Wenn wir von p_1 über p_2 nach p_3 entlang der Liniensegmente laufen – biegen wir dann in p_2 rechts (oder links) ab?



3. Schneiden sich die Segmente $\overline{p_1p_2}$ und $\overline{p_3p_4}$?

Dazu werden wir Methoden angeben, die in $\mathcal{O}(1)$ arbeiten und nur auf Addition, Subtraktion und Multiplikation und Vergleichen beruhen, also insbesondere keine Division verwenden. Folgende „straight forward“-Methode, zu entscheiden, ob sich zwei Liniensegmente schneiden, würde Division verwenden:

> Berechne die Geradengleichungen $y = mx + b$ für die Geraden, auf denen die Segmente liegen, finde deren Schnittpunkt und überprüfe, ob er auf beiden Segmenten liegt.

Sind Liniensegmente fast parallel, so ist diese Methode sehr anfällig für Berechnungsungenauigkeiten der Division auf einem Rechner.

Zu Frage 1.

Für Liniensegmente $\overrightarrow{p_0p_1}$ und $\overrightarrow{p_0p_2}$ mit $p_0 = (0,0)$,$p_1 = (x_1, y_1), p_2 = (x_2, y_2)$ ist das **Kreuzprodukt** $p_1 \times p_2$ definiert als

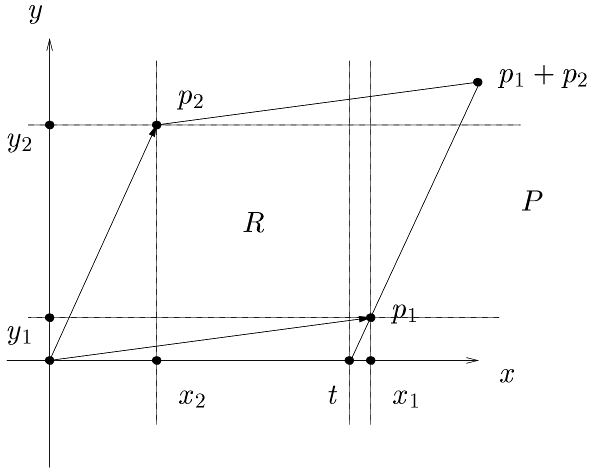
$$p_1 \times p_2 := \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1y_2 - x_2y_1 = -(p_2 \times p_1).$$

Wir erläutern nun, dass $\overrightarrow{p_0, p_1}$ **rechts von** $\overrightarrow{p_0, p_2}$ g.d.w. $p_1 \times p_2 > 0$.

Behauptung. $p_1 \times p_2$ beschreibt die Fläche P des Parallelogramms mit Eckpunkten $(0,0)$, p_1, p_2 und $p_1 + p_2$. Insbesondere ist $P = |p_1 \times p_2|$.

- **Fall 1:** $\overrightarrow{p_0p_1}$ rechts von $\overrightarrow{p_0p_2}$. Wir betrachten nur den Fall

$$x_1, x_2, y_1, y_2 \geq 0.$$



Es gilt $P = R$, wobei R die Fläche des Rechtecks mit Eckpunkten $(0,0)$, $(0,y_2)$, (t,y_2), $(t,0)$, also $R = y_2 t = P$. Nun gilt

$$\begin{aligned} \frac{y_2}{x_2} &= \frac{y_1}{x_1 - t} \\ \iff t &= x_1 - y_1\frac{x_2}{y_2} \end{aligned}$$

$$\Longrightarrow P = R = x_1y_2 - y_1x_2 = p_1 \times p_2 = |p_1 \times p_2| > 0$$

- **Fall 2:** $\overrightarrow{p_0p_1}$ links von $\overrightarrow{p_0p_2}$. Es gilt $P = R$, wobei R die Fläche des Rechtecks mit Eckpunkten $(0,0)$, $(0,y_1)$, (t,y_1), $(t,0)$ ist, d.h. $P = R = ty_1$. Dann gilt

$$\begin{aligned} \frac{y_1}{x_1} &= \frac{y_2}{x_2 - t} \\ \iff t &= x_2 - y_2\frac{x_1}{y_1} \end{aligned}$$

$$\Longrightarrow P = R = y_1x_2 - y_2x_1 = -p_1 \times p_2 = |p_1 \times p_2|$$

Das Vorzeichen von $p_1 \times p_2$ liefert also eine Aussage darüber, ob $\overrightarrow{p_0p_1}$ rechts oder links von $\overrightarrow{p_0p_2}$ liegt.

Definition 6.3 *$\overrightarrow{p_0,p_1}$ und $\overrightarrow{p_0p_2}$ heißen **kollinear**, falls $p_1 \times p_2 = 0$.*

Um nun zu entscheiden, ob $\overrightarrow{p_0p_1}$ für beliebiges p_0 rechts von $\overrightarrow{p_0p_2}$ liegt, muss p_0 als „Ursprung" aufgefasst werden. Das heißt, es muss getestet werden, ob

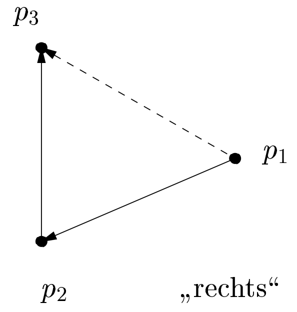
$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) > 0$$

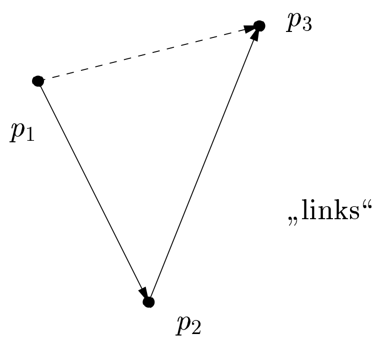
ist, wobei

$$p_1 - p_0 := (x_1 - x_0, y_1 - y_0) \text{ und } p_2 - p_0 := (x_2 - x_0, y_2 - y_0).$$

Zu Frage 2.

Eine andere Formulierung dieser Frage ist: In welche Richtung verläuft der Winkel $\sphericalangle p_1p_2p_3$ zu gegebenen p_1, p_2, p_3?





Diese Frage kann man ohne Berechnung der Winkel durch Betrachtung des Kreuzprodukts beantworten. Auf dem Weg von p_1 über p_2 nach p_3 biegen wir in p_2 nach rechts ab genau dann, wenn $\overrightarrow{p_1p_3}$ rechts von $\overrightarrow{p_1p_2}$ liegt, d.h. genau dann, wenn $(p_3 - p_1) \times (p_2 - p_1) > 0$ ist.

Zu Frage 3.

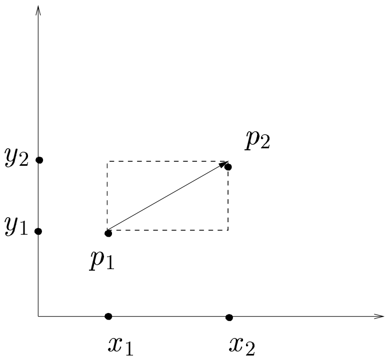
Ein „schneller Test“ , ob $\overline{p_1p_2}$ und $\overline{p_3,p_4}$ sich überhaupt schneiden können, ist der Test: schneiden sich die „umschließenden Rechtecke“ von $\overline{p_1p_2}$ und $\overline{p_3p_4}$ (notwendige Bedingung)?

Definition 6.4 *Das* ***umschließende Rechteck (bounding box)*** *eines geometrischen Objekts ist das kleinste umschließende Rechteck, das dieses Objekt enthält und dessen Seiten parallel zur x– bzw. y–Achse sind.*

Das umschließende Rechteck $(\widehat{p_1}, \widehat{p_2})$ zu $\overline{p_1p_2}$ ist beschrieben durch den linken unteren Punkt $\widehat{p_1} := (\widehat{x_1}, \widehat{y_1})$ und den rechten oberen Punkt $\widehat{p_2} := (\widehat{x_2}, \widehat{y_2})$, wobei

$$\begin{aligned}
\widehat{x_1} &:= \min(x_1, x_2),\\
\widehat{y_1} &:= \min(y_1, y_2),\\
\widehat{x_2} &:= \max(x_1, x_2),\\
\widehat{y_2} &:= \max(y_1, y_2)
\end{aligned}$$

mit $p_1 = (x_1, y_1)$ und $p_2 = (x_2, y_2)$.



Die Bounding Box von $\overline{p_1p_2}$.

Zwei Rechtecke $(\widehat{p_1}, \widehat{p_2})$ und $(\widehat{p_3}, \widehat{p_4})$ **schneiden** sich genau dann, wenn die Bedingungen

$$\begin{array}{rcl} \widehat{x_2} & \geq & \widehat{x_3}, \\ \widehat{x_4} & \geq & \widehat{x_1}, \\ \widehat{y_2} & \geq & \widehat{y_3}, \\ \widehat{y_4} & \geq & \widehat{y_1} \end{array}$$

gleichzeitig wahr sind. Ist der „schnelle Test“ positiv, so wird überprüft, ob sich die Liniensegmente gegenseitig „überqueren“ .

Definition 6.5 *$\overline{p_1p_2}$ **berührt** eine gegebene Gerade durch $\overline{p_3p_4}$,*

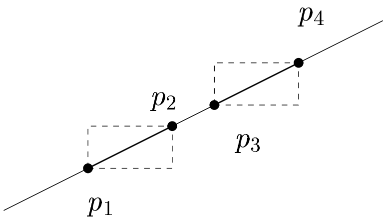
- *falls p_1 auf der einen Seite der Geraden und p_2 auf der anderen Seite der Geraden liegt (die Gerade **trennt** p_1 und p_2)*
- *oder falls p_1 oder p_2 auf der Geraden liegt.*

Zwei Liniensegmente schneiden sich genau dann, wenn ihre umschließenden Rechtecke sich schneiden und jedes der beiden Segmente die Gerade durch das jeweils andere Segment berührt.

Offensichtlich berührt $\overline{p_3p_4}$ die Gerade durch $\overline{p_1p_2}$ (p_3, p_4 nicht auf der Geraden) genau dann, wenn $\overrightarrow{p_1p_3}$ und $\overrightarrow{p_1p_4}$ auf verschiedenen Seiten von $\overrightarrow{p_1p_2}$ liegen, also wenn $(p_3 - p_1) \times (p_2 - p_1)$ und $(p_4 - p_1) \times (p_2 - p_1)$ verschiedene Vorzeichen haben. Sonderfall: eines der Produkte ist Null. In diesem Fall liegen p_3 oder p_4 auf der Geraden durch $\overline{p_1p_2}$.

Bemerkung. Sowohl der Test, ob Bounding Boxes sich schneiden, als auch der Test, ob ein Liniensegment eine Gerade berührt, sind ohne Division in $\mathcal{O}(1)$ durchführbar. Also ist der Test, ob zwei Liniensegmente, sich schneiden, in $\mathcal{O}(1)$ möglich.

Im folgenden Fall gilt, dass beide Liniensegmente $\overline{p_1p_2}$ und $\overline{p_3p_4}$ jeweils die Gerade durch das andere berühren, sich aber nicht schneiden.



Dieser Fall wird beim „schnellen Test“, ob die umschließenden Rechtecke sich schneiden, erkannt.

Bemerkung. Beim „schnellen Test“ müssen nur Vergleiche ausgeführt werden, keine arithmetischen Operationen.

6.2 Die Sweep Line Methode

Problem. Gegeben seinen n Liniensegmente. Schneiden sich zwei dieser Segmente?

Wir werden einen Algorithmus mit Laufzeit $\mathcal{O}(n \log n)$ entwerfen, der dieses Problem löst. Dieser Algorithmus beruht auf der SWEEP LINE Methode, einer grundlegenden Methode der Algorithmischen Geometrie.

Definition 6.6 *Eine* ***Sweep Line*** *ist eine imaginäre vertikale Linie, die eine Menge geometrischer Objekte von links nach rechts passiert.*

Die Sweep Line wird also entlang der x–Achse als „Zeitachse“ bewegt: sie ist durch die x–Koordinate bestimmt. Wir machen zur Vereinfachung des Problems folgende Voraussetzungen:

(i) Kein Liniensegment verläuft vertikal,

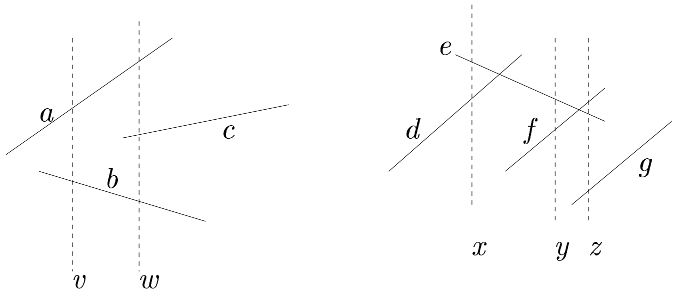
(ii) in einem Punkt schneiden sich höchstens zwei Liniensegmente.

Wir können wegen (i) dann davon ausgehen, dass ein Liniensegment, das eine gegebene Sweep Line schneidet, diese in genau einem Punkt schneidet. Zwei Liniensegmente s_1 und s_2 sind **vergleichbar bezüglich Sweep Line** x, falls sie beide x schneiden und sich nicht gegenseitig an der x–Koordinate x schneiden. Es gilt dann

$s_1 <_x s_2$, falls s_1 unterhalb von s_2 in x,

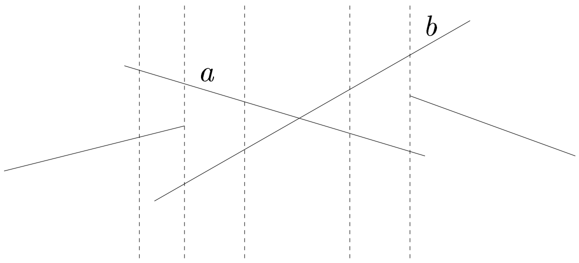
$s_2 <_x s_1$, falls s_1 oberhalb von s_2 in x.

Für jedes x ist $<_x$ eine totale Ordung auf den Liniensegmenten, die x schneiden und sich nicht gegenseitig schneiden.



$$a >_v b, \quad a >_w c >_w b, \quad d <_x e, \quad e >_y f, \quad f >_z e >_z g$$

Die Geraden e und f schneiden sich: die relative Ordnung wird vertauscht, d.h. $e >_y f$ und $f >_z e$. Da sich nie drei Liniensegmente in demselben Punkt schneiden, muss es für zwei Liniensegmente a und b, die sich schneiden, Sweep Lines x und y mit $a <_x b$ und $a >_y b$ geben, so dass a und b an x und y in der entsprechenden Ordnung direkt aufeinander folgen.



a schneidet b, zugehörige Sweep Lines

Sweep Line Algorithmen verwenden typischerweise zwei Datenmengen:

1. den *Sweep Line Zustand*, welcher die Beziehung zwischen Objekten, die die Sweep Line schneiden, wiedergibt und
2. den *Event Point Schedule*, eine Folge von x–Koordinaten, die, von rechts nach links angeordnet, für den Sweep Line Algorithmus die „Haltepositionen“ angeben. Solche Haltepunkte heißen **Event Point**. Änderungen am Sweep Line Zustand treten nur an Event Points auf.

6.2.1 Der Algorithmus von Shamos & Hoey (1975)

Im folgenden Algorithmus sind gerade die Endpunkte der Liniensegmente die Event Points. Diese Event Points werden entsprechend der x–Koordinate von links nach rechts sortiert. Ein Segment wird in den Sweep Line Zustand eingefügt, wenn von der Sweep Line dessen linker Endpunkt passiert wird und entfernt, wenn sein rechter Endpunkt erreicht wird. Wenn zwei Liniensegmente zum ersten mal im Sweep Line Zustand unmittelbar aufeinander folgen, wird überprüft, ob sie sich schneiden.

Der Sweep Line Zustand ist eine sich dynamisch ändernde Ordnung T, für die wir folgende Operationen ausführen müssen:

- INSERT(T, s), DELETE(T, s)
- ABOVE(T, s): gibt das Liniensegment aus, das in T unmittelbar oberhalb von s liegt – das nächst größere Segment in T.
- BELOW(T, s): gibt das Liniensegment aus, das in T unmittelbar unterhalb von s liegt – das nächst kleinere Segment in T.

Diese Operationen können für n Segmente mit Speziellen Suchbäumen, z.B. AVL–Bäumen in $\mathcal{O}(\log n)$ pro Operation ausgeführt werden (vgl. Theoretische Informatik).

Formale Beschreibung des Algorithmus.

Eingabe: Eine Menge von Liniensegmenten S.

```
INTERSECT($S$)
Setze $T := \emptyset$
Sortiere die Endpunkte der Liniensegmente aus S von links nach rechts;
  bei Gleichheit ordne sie entsprechend ihrer $Y$–Koordinate.
Falls zwei Endpunkte verschiedener Liniensegmente in dieser Ordnung
  identisch sind, so gib aus TRUE, ansonsten führe aus.
    für jeden Endpunkt $p$ in der sortierten Liste führe aus:
    falls $p$ linker Endpunkt von Segment $S$ dann
        INSERT($T, s$)
        falls ABOVE($T, s$) existiert und $s$ schneidet oder BELOW($T, s$)
            existiert und $s$ schneidet dann gib aus TRUE
    falls $p$ rechter Endpunkt von Segment $s$
        falls ABOVE($T, s$) und BELOW($T, s$) beide existieren und sich
            schneiden dann gib aus TRUE.
    DELETE($T, s$)
Gib aus FALSE.
```

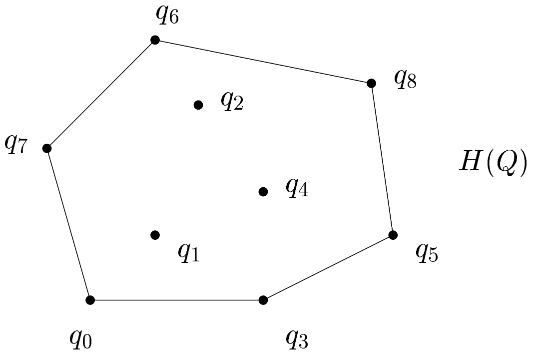
Laufzeit. Die $2n$ Endpunkte können in $\mathcal{O}(n \log n)$ sortiert werden. Es werden maximal $2n$ Endpunkte p durchlaufen, und für jeden von diesen wird eine konstante Anzahl von Operationen Insert, Delete, Above und Below ausgeführt, welche jeweils in $\mathcal{O}(\log n)$ möglich sind. Außerdem werden jeweils eine konstante Anzahl von Tests auf „Schneiden“ ausgeführt, welche in $\mathcal{O}(1)$ möglich sind. Intersect hat also eine Laufzeit von $\mathcal{O}(n \log n)$.

Korrektheit. Einen formalen Korrektheitsbeweis für Intersect wollen wir hier nicht führen, da die Korrektheit ziemlich offensichtlich ist. Die einzige Möglichkeit, dass Intersect nicht korrekt arbeiten würde, wäre die, dass Intersect False ausgibt, obwohl sich zwei Liniensegmente s_1, s_2 schneiden. Diese Möglichkeit kann leicht zum Widerspruch geführt werden.

6.3 Die konvexe Hülle einer Punktmenge

Problem. Gegeben sind n Punkte $\{q_0, \ldots, q_{n-1}\} =: Q$, $|Q| \geq 3$. Berechne die konvexe Hülle $H(Q)$ von Q, d.h. das minimale konvexe Polygon, dessen Eckpunkte aus Q sind, so dass alle Punkte aus Q im Inneren oder auf dem Rand von $H(Q)$ liegen. (Insbesondere ist also ein Punkt des Randes von $H(Q)$, der nicht aus Q ist, nicht aus $H(Q)$.)

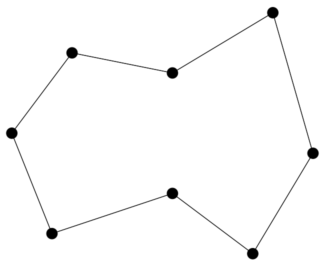
Beispiel.

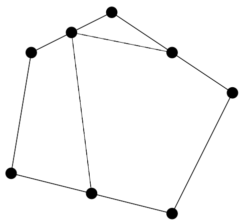


Definition 6.7 *Ein **Polygon** ist durch eine Folge von Punkten $\langle p_1, \ldots, p_n \rangle$ gegeben, die p_i sind in der Reihenfolge ihres Auftretens auf dem Rand indiziert. $\overline{p_i p_{i+1}}$ heißt (i modulo n)-te Polygonkante.*

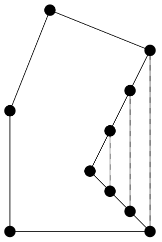
Definition 6.8 *Ein Polygon heißt **einfach**, wenn sich keine zwei Polygonkanten in einem Punkt ungleich ihrer Endpunkte schneiden.*

Definition 6.9 *Ein einfaches Polygon heißt* **konvex** *g.d.w. für je zwei Punkte q_i, q_j aus dem Inneren (inkl. Rand) des Polygons gilt, dass $\overline{q_iq_j}$ vollständig im Inneren (inkl. Rand) des Polygons liegt.*





konvex



nicht konvex

Wir werden zwei Algorithmen behandeln, die die konvexe Hülle von n Punkten in $\mathcal{O}(n \log n)$ bzw. $\mathcal{O}(nh)$ ($h = \#(\text{Eckpunkte})$) bestimmen.

6.3.1 Der GRAHAM SCAN (1972)

Idee.

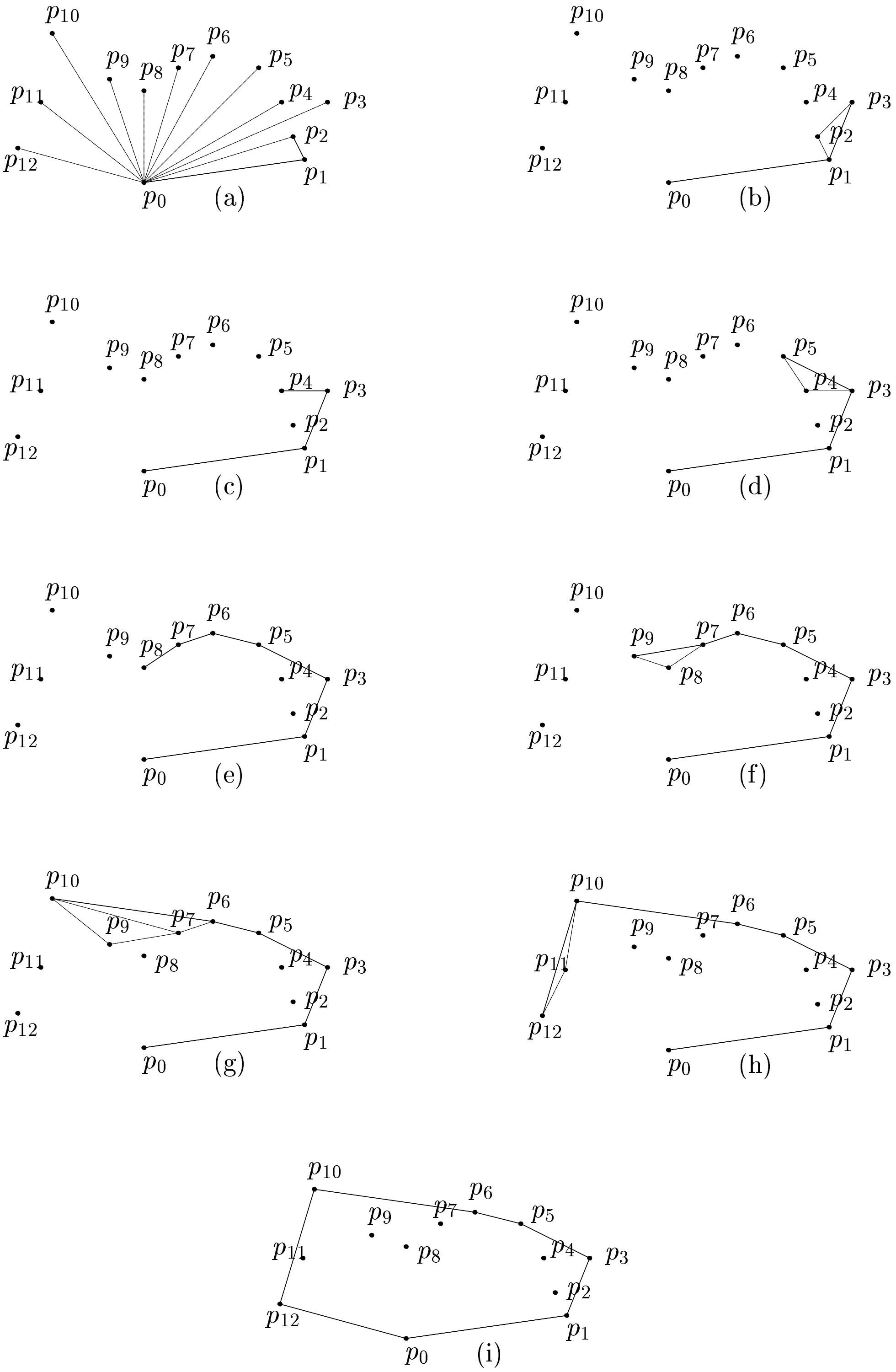
1. Bestimme Punkt p_0 „unten links" aus Q.
2. Ordne die Punkte q_i aus Q nach dem Winkel zwischen der Horizontalen durch p_0 und $\overline{p_0q_i}$.
3. Durchlaufe die Punkte $p_0, p_1, \ldots, p_{n-1}$ in dieser Reihenfolge und konstruiere sukzessive die konvexe Hülle $H(Q)$ nach folgender Regel:

   > Wird beim Durchlaufen der Liniensegmente $\overline{pq}$ und $\overline{qr}$ bei q nicht nach links abgebogen, so ist q nicht aus dem Rand von $H(Q)$.

Als Datenstruktur benutzen wir einen STACK S mit den Operationen

- PUSH,
- POP,
- TOP(S): gibt das oberste Element von S an, ohne S zu verändern,
- NEXT–TO–TOP(S): gib das zweitoberste Element von S an, ohne S zu verändern.

Am Ende des GRAHAM SCAN liegen in S (von unten nach oben) die Eckpunkte von $H(Q)$ in der Reihenfolge, in der sie im Gegenuhrzeigersinn auf dem Rand von $H(Q)$ liegen.


p_0
p_1
p_2
p_3
p_4
p_5
p_6
p_7
p_8
p_9
p_{10}
p_{11}
p_{12}
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)

Formale Beschreibung des Algorithmus.

```
GRAHAM SCAN($Q$)
1. Sei $p_0$ ein Punkt mit kleinster $y$–Koordinate in $Q$ und unter solchen
   der mit kleinster $x$–Koordinate [unten links in Q].
2. Sortiere die anderen $n-1$ Punkte $q_i$ aus $Q$ gemäß dem Winkel zwischen
   der Horizontalen durch $p_0$ und $\overline{p_0 q_i}$ im Gegenuhrzeigersinn. Falls
   mehrere Punkte denselben Winkel zu $p_0$ haben, entferne alle bis auf
   denjenigen, der am weitesten von $p_0$ entfernt ist. Die sortierte Folge
   sei $\langle p_1, \ldots, p_m \rangle$
3. PUSH($p_0, S$), PUSH($p_1, S$), PUSH($p_2, s$)
4. Für $i = 3$ bis $m$ führe aus:
5.     solange die Folge der Liniensegmente $\overline{\text{NEXT–TO–TOP}(S)\text{TOP}(S)}$,
       $\overline{\text{TOP}(S)p_i}$ nicht nach links abbiegt in TOP($S$), führe aus
6.         POP($S$)
7.     PUSH($p_i, S$)
8. Gib $S$ aus.
```

Laufzeit. 1. kann in $\mathcal{O}(n)$ berechnet werden. 2. ist in $\mathcal{O}(n \log n)$ ausführbar. Der Rest ist insgesamt in $\mathcal{O}(n)$: alle Durchläufe der „solange"-Schleife sind in $\mathcal{O}(n)$, da die Anzahl der Durchläufe nicht größer als die Gesamtzahl der durchgeführten PUSH sein kann. Diese ist in $\mathcal{O}(n)$.

Bemerkung. Wir machen hier eine amortisierte Laufzeitanalyse wie bei MULTIPOP.

Satz 6.1 GRAHAM SCAN *berechnet die konvexe Hülle einer Punktmenge* Q.

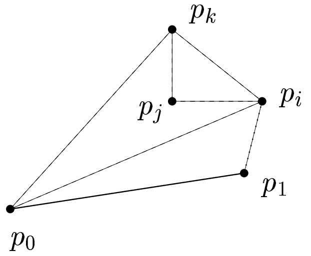
Beweis. Wir beweisen zwei Invarianten für GRAHAM SCAN:

1. Wird ein Punkt von S entfernt (POP(S)), so ist er nicht Eckpunkt der konvexen Hülle von Q.

2. Der aktuelle Inhalt von S bestimmt stets ein konvexes Polygon.

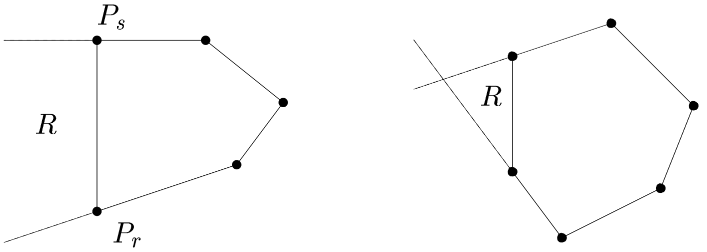
Da jeder Punkt aus Q genau einmal auf S gelegt wird, folgt daraus die Behauptung.

Invariante 1 ist erfüllt, denn angenommen, p_j wird von S entfent, dann gibt es p_i und p_k, so dass $\overline{p_i p_j}$, $\overline{p_j p_k}$ keine Linksbiegung macht. Dann muss p_j im Inneren oder auf dem Rand des Dreiecks $p_0 p_i p_k$ liegen, kann also nicht Teil von $H(Q)$ sein.

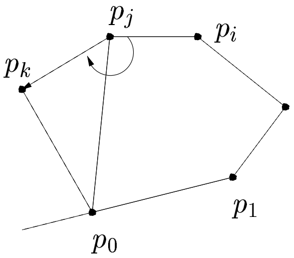


Invariante 2 ist zunächst für $p_0p_1p_2$ erfüllt. Wir betrachten die durchgeführten Veränderungen von S (Pop und Push) und zeigen, dass eine jede solche Veränderung ein konvexes Polygon wieder in ein konvexes Polygon überführt.

Zunächst ist klar, dass Pop, also Entfernen eines Punktes aus der Menge der Eckpunkte eines konvexen Polygons, wieder eine Punktefolge ergibt, die ein konvexes Polygon angibt. Um zu zeigen, dass die ausgeführten Push jeweils ein konvexes Polygon P in ein konvexes Polygon P' überführt, betrachte die Region R, die durch eine Seite eines konvexen Polygons P und die Verlängerung der beiden anliegenden Seiten bestimmt ist.



Wird ein Punkt aus einer solchen Region R zu einem konvexen Polygon P hinzugefügt, so ist das resultierende Polygon P' wieder konvex. Sei nun p_k ein Punkt, für den Push(p_k, S) ausgeführt wird, d.h. der zu dem durch S bestimmten Polygon P hinzugefügt wird. Dann gibt es p_i, p_j so, dass $\overline{p_ip_j}$, $\overline{p_jp_k}$ nach links abbiegt. Dann liegt p_k links von $\overrightarrow{p_ip_j}$. Wegen der Reihenfolge, in der die Punkte betrachtet werden, liegt p_k links von $\overrightarrow{p_0p_j}$. Und wegen der Wahl von p_0 liegt p_k auch links von $\overrightarrow{p_0p_1}$. Also liegt p_k in der durch $\overline{p_0p_j}$ und den beiden angrenzenden Seiten von P bestimmten Region.

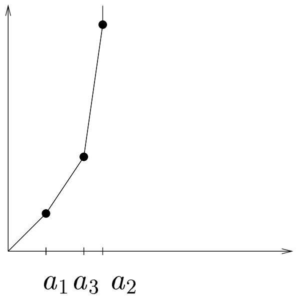


■

Satz 6.2 *Die Laufzeit zur Berechnung der konvexen Hülle einer Menge von* n *Punkten ist in* $\Omega(n \log n)$.

Beweis. Natürlich ist die Laufzeit $T(n)$ zur Bestimmung der konvexen Hülle in $\Omega(n)$, da jeder Punkt zumindest einmal „angesehen“ werden muss. Denn würde man einen Punkt p nicht betrachten, so könnte er außerhalb des berechneten Polygons sein.

Wir benutzen nun die Tatsache, dass Sortieren basierend auf Vergleichen in $\Omega(n \log n)$ ist. Angenommen, es gäbe einen Algorithmus, der die konvexe Hülle in $T(n)$ bestimmt, mit $T(n) \in \Omega(n)$ und $T(n) \in o(n \log n)$.[1] Betrachte nun n verschiedene Zahlen $a_1, \ldots, a_n$. Konstruiere daraus in $\mathcal{O}(n)$ die Punktemenge $A := \{(a_i, a_i^2) : i = 1, \ldots, n\}$.



Bestimme die konvexe Hülle von A in $T(n)$ Zeit und das Minimum $a_{\min}$ der a_i in $\mathcal{O}(n)$. Lese dann anhand von $H(A)$ die Sortierung der a_i ab, startend von $a_{\min}$ im Gegenuhrzeigersinn.

Jeder Punkt aus A liegt auf dem Rand von $H(A)$, also findet man so die Sortierung der a_i in Zeit $f(n) \in \mathcal{O}(T(n) + n)$ im Widerspruch zur unteren Schranke $\Omega(n \log n)$ für Sortieren. ■

6.3.2 Algorithmus JARVIS' MARCH (1973)

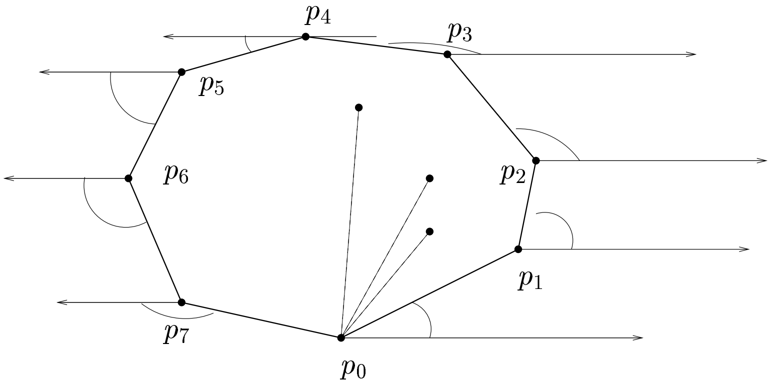
Der nun folgende Algorithmus zur Bestimmung der konvexen Hülle von Jarvis geht nach dem Prinzip des „Geschenke einpacken“ vor. Die Laufzeit ist in $\mathcal{O}(nh)$, wobei h die Anzahl der Punkte ist, die die konvexe Hülle festlegen (Eckpunkte). Ist $h \in o(\log n)$, so ist JARVIS' MARCH besser als GRAHAM SCAN.

---

[1] $o(g(n)) := \{f(n)$: für jede Konstante $c > 0$ existieren Konstanten $n_0 > 0$ so, dass für alle $n \geq n_0$ gilt $0 \leq f(n) < cg(n)\}$. Es ist beispielsweise $f(n) = 2n \in o(n \log n)$, $5n \log n \in o(n^2)$, aber $f(n) = 2n^2 \notin o(n^2)$.

Idee. Ausgehend von dem „untersten Punkt“ p_0 (wie bei GRAHAM SCAN) werden zunächst die Punkte p_{i+1} mit kleinstem Winkel zur Horizontalen durch den Vorgängerpunkt p_i „nach rechts, in positiver Richtung“ berechnet. (Begriff: „Rechtskette“).

Wenn der oberste Punkt p_k erreicht ist, werden von diesem ausgehend die Punkte p_{i+1} mit kleinstem Winkel zur Horizontalen durch den Vorgängerpunkt p_i „nach links, in negativer Richtung“ berechnet. (Begriff: „Linkskette“).



Formale Beschreibung des Algorithmus.

```
JARVIS' MARCH($Q$)
1. Sei $p_0$ der „unterste“ Punkt aus $Q$. Setze $i := 0$.
2. Solange es einen Punkt oberalb von $p_i$ gibt:
3.     Bestimme den Punkt $p_{i+1}$ unter allen Punkten oberhalb von $p_i$,
       der den kleinsten Winkel zur Horizontalen durch $p_i$ nach rechts
       hat.
4.     $i := i + 1$
5. Solange $p_i \neq p_0$:
6.     Bestimme den Punkt $p_{i+1}$ unter allen Punkten unterhalb von $p_i$,
       der den kleinsten Winkel zur Horizontalen durch $p_i$ nach links
       hat.
7.     $i := i + 1$
```

Laufzeit. Ein Durchlauf von 2., 3. bzw. 5., 6. kann in $\mathcal{O}(n)$ realisiert werden durch $\mathcal{O}(n)$ „Vergleiche“, die (wie bei Frage 1) in $\mathcal{O}(1)$ durchführbar sind. Wenn h Anzahl der Punkte, die die konvexe Hülle von Q festlegen, ist, so ist also JARVIS' MARCH insgesamt in $\mathcal{O}(nh)$ ausführbar.

Kapitel 7

String–Matching

> **Der Literaturtip.** Zum Thema „String-Matching“ siehe auch [CLR90].

Gegeben. Zwei Folgen P und T von Buchstaben oder Ziffern (oder von beidem), wobei die Länge von P kleiner ist als die Länge von T.

Problem. Finde alle Vorkommen von P in T.

Beispiel. Text T, Pattern (Muster) P.

T	a	c	b	c	c	a	b	c	b	c	b	c	a	c	b
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)
P	c	b	c												

Vorkommen: 2, 8, 10

Motivation. Wo kann String-Matching eingesetzt werden?

- Textverarbeitung;
- Editoren;
- Suche in DNA-Folgen.

Wenn „Text“ T im Array $T[1 \dots n]$ und „Muster“ P in Array $P[1 \dots m]$ steht, so sollen also alle s ($0 \leq s \leq n - m$) ausgegeben werden, für die gilt

$$T[s + j] = P[j] \text{ für alle } 1 \leq j \leq m.$$

Für diese s gilt: Das Muster P taucht mit Verschiebung s in T auf.

Algorithmen. Es existieren verschiedene Algorithmen zur Lösung dieses Problems. Zunächst die Frage: Welche Laufzeit können wir bestenfalls für einen solchen Algorithmus erwarten?

Wir suchen also eine untere Schranke für die Laufzeit. Um das Problem zu lösen, müssen wir den gesamten Text T und das gesamte Muster P kennen. Wenn also $n = |T|$ die Länge von T und die $m = |P|$ die Länge von P ist, so benötigen wir zumindest $\Omega(n + m)$ Schritte (falls wir keinerlei Vorwissen über T und P haben).

Zunächst der „naive“ Ansatz:

> Teste für alle $n - m + 1$ möglichen s die Bedingung
> $T[s + j] = P[j],\ 1 \leq j \leq m$.

Beispiel. $|T| = 18$ und $|P| = 3$.

	a	**c**	**b**	**c**	c	a	b	**c**	**b**	**c**	a	b	**c**	**b**	**c**	**b**	**c**	c
$s = 0$	c	b	c															
$s = 1$		**c**	**b**	**c**	←													
$s = 2$			c	b	c													
$s = 3$				c	b	c												
$s = 4$					c	b	c											
$s = 5$						c	b	c										
$s = 6$							c	b	c									
$s = 7$								**c**	**b**	**c**	←							
$s = 8$									c	b	c							
$s = 9$										c	b	c						
$s = 10$											c	b	c					
$s = 11$												c	b	c				
$s = 12$													**c**	**b**	**c**	←		
$s = 13$														c	b	c		
$s = 14$															**c**	**b**	**c**	←
$s = 15$																c	b	c

P taucht mit Verschiebung 1, 7, 12 und 14 in T auf.

Formale Beschreibung des Algorithmus

Naiver String–Matcher(T, P)

```
1. Setze $n :=$ Länge$[T]$ und $m :=$ Länge$[P]$
2. Für $s = 0$ bis $n - m$ führe aus
3.     Falls $P[1 \dots m] = T[s + 1 \dots s + m]$ ist, dann
4.     gib aus: „Muster P taucht mit Verschiebung $s$ in $T$ auf.
```

Diese Prozedur hat eine worst-case-Laufzeit von $\Theta((n - m + 1) \cdot m)$. Der worst-case tritt beispielsweise ein, wenn ein Text $T = a\ \dots\ a$ (n-mal) und ein Muster $P = a \ \dots \ a$ (n-mal) verglichen wird. In diesem Fall werden für alle shifts s $(0 \leq s \leq n-m)$ für alle j $(1 \leq j \leq m)$ die Vergleiche $P[j] = T[s+j]$ gemacht.

7.1 Der Algorithmus von **Rabin** & **Karp** (1981)

Wie wir sehen, hat dieser Algorithmus eine Laufzeit von $\mathcal{O}((n-m+1)\cdot m)$ im worst-case und unter gewissen Vorbedingungen eine average-case-Laufzeit von $\mathcal{O}(n+m)$.

Sei o.B.d.A. Alphabet $\Sigma = \{0, 1, 2, \dots, 9\}$.

Bemerkung. Im allgemeinen Fall kann man Strings als Zahl in „d-ärer Darstellung“ auffassen, mit $d = |\Sigma|$.

Wir betrachten also einen String der Länge k als **Dezimalzahl** der Länge k. Bezeichne mit p den zugehörigen Dezimalwert zu $P[1 \dots m]$ und mit t_s den zugehörigen Dezimalwert zu $T[s+1, \dots, s+m]$. Dann gilt

$$t_s = p \ \text{ g.d.w. } \ P[j] = T[s+j] \text{ für alle } 1 \leq j \leq m.$$

Falls also p in Zeit $\mathcal{O}(m)$ berechnet werden kann und alle t_i insgesamt in Zeit $\mathcal{O}(n)$, dann könnten alle möglichen Vorkommen von P in T in Zeit $\mathcal{O}(n)$ berechnet werden durch Vergleichen aller möglichen t_p mit p, wobei die Zeit für einen Vergleich mit $\mathcal{O}(1)$ gezählt wird. Dies ist allerdings nicht unbedingt „realistisch“:

Problem. p und t_s können sehr große Zahlen sein.

Zunächst kann p in Zeit $\mathcal{O}(|m|)$ mit dem sogenannten **Horner-Schema** berechnet werden:

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \ldots + 10(P[2] + 10P[1]) \ldots))$$

Genauso kann t_0 in Zeit $\mathcal{O}(m)$ aus $T[1 \dots m]$ berechnet werden und die restlichen $t_1, t_2, \dots, t_{n-m}$ in Gesamtzeit $\mathcal{O}(n-m)$ mittels

$$t_{s+1} = 10 \cdot (t_s - 10^{m-1} T[s+1]) + T[s+m+1].$$

Kommentar:

Streichen der ersten Ziffer von t_s:	$-10^{m-1} \cdot T[s+1]$
Anhängen der letzten Ziffer an t_s:	$+T[s+m+1]$

Beispiel. Sei $m = 5$, $t_s = 31415$, $t_{s+1} = 14152$, dann erhalten wir t_{s+1} aus t_s mittels

$$14152 = 10 \cdot (31415 - 10000 \cdot 3) + 2.$$

Für dieses Verfahren benötigen wir nur vorberechnete 10^{m-1} (geht z.B. in $\mathcal{O}(\lg m)$).

Nun zu dem Problem, dass t_s und p sehr groß sein können: Falls der String P die Länge m hat, so ist p gerade m Ziffern lang. Die Annahme, dass eine Operation auf p einen konstanten Zeitaufwand hat, ist also unrealistisch.

Trick. Berechne p und die t_s modulo einer geeigneten Zahl q.

Natürlich können die Berechnungen von p, t_0 und den t_{s+1} aus den t_s alle auch modulo q in $\mathcal{O}(n+m)$ Zeit vorgenommen werden. Typischerweise wird q als Primzahl gewählt und zwar so, dass $10q$ genau in ein Computerwort passt.

Beispiel.

$$\begin{aligned} 1\,4\,1\,5\,2 &\equiv ((31415 - 3 \cdot 10000) \cdot 10 + 2) (\bmod 13) \\ &\equiv ((7 - 3 \cdot 3) \cdot 10 + 2) (\bmod 13) \\ &\equiv 8 (\bmod 13) \end{aligned}$$

Beispiel. Berechnung für eine längere Folge.
Sei $T := 2\,3\,5\,9\,0\,2\,3\,1\,4\,1\,5\,2\,6\,7\,3\,9\,9\,2\,1$, $P := 3\,1\,4\,1\,5$, $m = 5$, $q := 13$.

Berechne:

$$T = \underbrace{2\,3\,5\,9\,0}_{8}\ 2\,3\,1\,4\,1\,5\,2\,6\,7\,3\,9\,9\,2\,1$$

$$T = 2\ \underbrace{3\,5\,9\,0\,2}_{9}\ 3\,1\,4\,1\,5\,2\,6\,7\,3\,9\,9\,2\,1$$

$$T = 2\,3\ \underbrace{5\,9\,0\,2\,3}_{3}\ 1\,4\,1\,5\,2\,6\,7\,3\,9\,9\,2\,1$$

$\vdots$

$$T = 2\,3\,5\,9\,0\,2\ \underbrace{3\,1\,4\,1\,5}_{7}\ 2\,6\,7\,3\,9\,9\,2\,1$$

$\vdots$

$$T = 2\,3\,5\,9\,0\,2\,3\,1\,4\,1\,5\,2\ \underbrace{6\,7\,3\,9\,9}_{7}\ 2\,1$$

$\vdots$

$\Longrightarrow 8\ 9\ 3\ 11\ 0\ 1\ 7\ 8\ 4\ 5\ 10\ 11\ 7\ 9\ 11$

Beobachtung. Durch die Modulo-Berechnung geht die Eindeutigkeit verloren! D.h. es gibt „Ausschnitte“ t_s aus t, für die $t_s \equiv p (\text{mod } q)$, aber $T[s+1 \dots s+m] = P[1 \dots m]$. Es muss also jedesmal, wenn $t_s(\text{mod } q) \equiv p(\text{mod } q)$ gilt, der Test $T[s+j] = P[j]$ für alle $1 \leq j \leq m$ gemacht werden.

Formale Beschreibung des Algorithmus

Sei Σ Alphabet mit $|\Sigma| = d$, T ein Text, P ein Muster, Länge von $[P] \leq$ Länge von $[T]$ und q eine geeignete Primzahl.

```
RABIN-KARP-MATCHER($T, P, d, q$)
 1. $n :=$ Länge$[T]$;
 2. $m :=$ Länge$[P]$;
 3. $h := d^{m-1} \text{ mod } q$;
 4. $p := 0$;
 5. $t := 0$.
 6. Für $i = 1$ bis $n$ führe aus:
 7.       $p := (d \cdot p + P[i])(\text{mod } q)$
 8.       $t := (d \cdot t + T[i])(\text{mod } q)$
 9. Für $s = 0$ bis $n - m$ führe aus:
10.       Falls $p = t_s$ dann
11.             falls $P[1 \dots m] = T[s+1 \dots s+m]$ dann
12.                   gib aus: „$P$ taucht an Stelle $s$ bis $s+m$ in $T$ auf“
13.       Falls $s < n - m$ dann
14.             $t := (d \cdot (t - T[s+1] \cdot h) + T[s+m+1])(\text{mod } q)$
```

Die Laufzeit ist im worst-case ebenfalls in $\mathcal{O}((n-m+1) \cdot m)$. Denn: wenn beispielsweise $P = a^m$, $T = a^n$, dann wird jeder der $n - m + 1$ „shifts“ verifiziert.

Die Berechnungen in (3) und (6)-(8) benötigen $\mathcal{O}(m)$ Zeit. Es kann allerdings davon ausgegangen werden, dass in vielen Anwendungen nur selten (10) erfüllt ist, d.h. explizit $P[1 \dots m] = T[s+1 \dots ; s+m]$ getestet wird.

Gehen wir davon aus, dass sich die Berechnung mod q wie eine „Zufallsabbildung“ von Σ^* nach $\mathbb{Z}_q$ verhält. Dann ist die erwartete Anzahl von „unzulässigen“ Übereinstimmungen $t_s \text{ mod } q = p \text{ mod } q$ in $\mathcal{O}(\frac{n}{q})$, da die Wahrscheinlichkeit, dass ein beliebiges t_s äquivalent zu $p \text{ mod } q$ ist, mit $\frac{1}{q}$ angenommen werden kann. Damit ist die „average-case“-Laufzeit des Rabin-Karp-Algorithmus in $\mathcal{O}(m+n+m \cdot (v + \frac{n}{q}))$, wobei v die Anzahl der zulässigen Übereinstimmungen ist.

Falls die Anzahl der zulässigen Übereinstimmungen klein ist, etwa konstant, und $q > m$, so ist die average-case-Laufzeit in $\mathcal{O}(m+n)$.

7.2 String-Matching mit endlichen Automaten

Notation. Ein String w heißt **Präfix** eines String x, falls $x = wy$ für ein y.

Wir behandeln eine Methode, die einen sogenannten „endlichen Automaten“ aufbaut, der einen Text T nach allen Vorkommen des Musters P durchläuft. Dieser endliche Automat betrachtet jeden Buchstaben aus T genau einmal mit jeweils konstanter Zeit. Damit ist der Gesamtaufwand für String-Matching $\Theta(n)$ – nach Aufbau des Automaten. Allerdings ist der Aufwand für den Aufbau des Automaten abhängig von der Anzahl aller möglichen Buchstaben; d.h. falls T und P Strings über dem Alphabet Σ sind, ist der Aufwand für den Aufbau des Automaten abhängig von $|\Sigma|$ (grobe Idee).

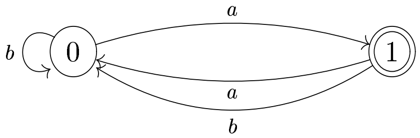
Definition 7.1 *Ein* ***endlicher Automat*** $\mathcal{A}$ *ist ein Tupel* $(Q, q_0, A, \Sigma, \delta)$, *wobei*

Q	*endliche Menge von* ***Zuständen****;*
$q_0 \in Q$	***Startzustand****;*
$A \subseteq Q$	*Menge von* ***akzeptierenden Zuständen****;*
Σ	*endliches* ***Eingabealphabet****;*
$\delta : Q \times \Sigma \longrightarrow Q$	***Übertragungsfunktion****.*

Die Menge aller Strings, die nur aus Buchstaben aus Σ bestehen, nennen wir Σ^*.

Beispiel. $\Sigma = \{a, b\},\ Q = \{0, 1\},\ q_0 = 0,\ A = \{1\}$,

$$\delta : \begin{array}{lcl} (0,a) & \longrightarrow & 1 \\ (0,b) & \longrightarrow & 0 \\ (1,a) & \longrightarrow & 0 \\ (1,b) & \longrightarrow & 0 \end{array}$$



Der Automat „akzeptiert “ die folgenden Strings:

$$a,\ ba,\ bba,\ \ldots$$
$$aa,\ aaba,\ babba,\ \ldots$$

Ein endlicher Automat $\mathcal{A}$ induziert eine Funktion $\mathcal{C}$, die **Zustandsfunktion** mit $\mathcal{C} : \Sigma^* \longrightarrow Q$, so dass $\mathcal{C}(w)$ der Zustand von $\mathcal{A}$ ist, in dem $\mathcal{A}$ beim Lesen von w mittels δ endet, d.h. $\mathcal{C}$ ist definiert durch:

$$\begin{array}{rcl} \mathcal{C}(a) & = & \delta(q_0, a) \text{ für } a \in \Sigma, \\ \mathcal{C}(wa) & = & \delta(\mathcal{C}(w), a) \text{ für } w \in \Sigma^*, a \in \Sigma. \end{array}$$

$\mathcal{A}$ **akzeptiert** w genau dann, wenn $\mathcal{C}(w) \in A$.

Zu jedem Muster P kann ein entsprechender „String Matching Automat“ $\mathcal{A}_p$ konstruiert werden. Definiere dazu als „Hilfsfunktion“ die **Suffix-Funktion** suf_p durch:

$$\begin{array}{rcl} \text{suf}_p & : & \Sigma^* \longrightarrow \{0, 1, \ldots, |P| = m\} \\ \text{suf}_P(w) & := & \begin{cases} \max\{k : P[1 \ \ldots \ k] \text{ ist Suffix von } w\}, \\ \ \text{falls solch ein } k \ (1 \leq k < m) \text{ existiert;} \\ 0 \text{ sonst.} \end{cases} \end{array}$$

Beispiel. $P = ab$; $\text{suf}_p(ccaca) = 1$, $\text{suf}_p(ccab) = 2$, $\text{suf}_p(abcc) = 0$.

Der String-Matching-Automat $\mathcal{A}_p$ zu P ist definiert durch

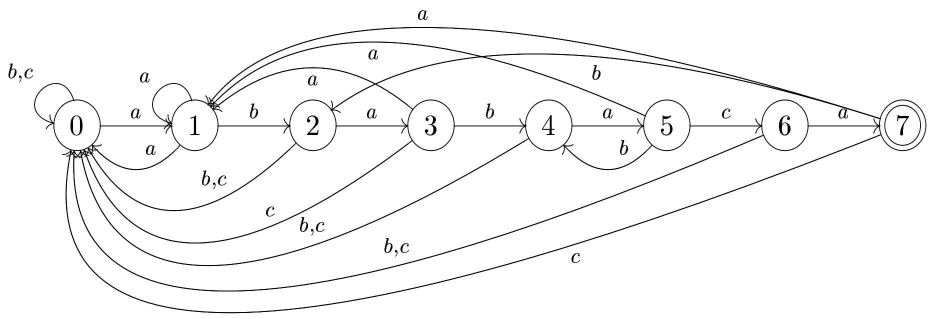
$$Q = \{0, 1, \ldots, m\},\ q_0 = 0,\ A = \{m\}$$
$$\delta : Q \times \Sigma \longrightarrow Q$$
$$\delta(q, a) = \text{suf}_p(P[1 \ \ldots \ q]a)$$

Begründung für diese Definition.

Wenn der Automat $\mathcal{A}_p$ den Text T bearbeitet, so ist er nach i Schritten im Zustand $\mathcal{C}(T[1 \ \ldots \ i]) = q$, wobei $q = \text{suf}_p(T[1 \ \ldots \ i])$ die Länge des längsten Suffix von $T[1 \ \ldots \ i]$ ist, der auch Präfix von P ist. Falls der nächste Buchstabe in T, d.h. $T[i+1]$, gleich a ist, so sollte der nächste Zustand $\text{suf}_p(T[1 \ \ldots \ i+1]) = \text{suf}_p(T[1 \ \ldots \ i]a)$ sein.

Wir werden sehen, dass zur Berechnung des längsten Suffix von $T[1 \ \ldots \ i]a$, der auch Präfix von P ist, die Berechnung des längsten Suffix von $P[1 \ \ldots \ q]a$, der auch Präfix von P ist, ausreicht. Das heißt: in jedem Zustand muss $\mathcal{A}_p$ nur die Länge des längsten Präfix von P kennen, der ein Suffix von dem ist, was bisher gelesen wurde.

Beispiel. $P = a\ b\ a\ b\ a\ c\ a \quad \Sigma = \{a, b, c\}$



Zugehörige Übergangstafel:

	$\Sigma = a$	b	c
$q = 0$	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	7	0	0
7	1	2	0

Textbeispiele.

i	1	2	3	4	5	6	7	8	9	10	11	12
T	a	b	a	b	a	b	a	c	a	b	a	b
$(T[1\ \ldots\ i])$	1	2	3	4	5	4	5	6	7	2	3	4
T'	c	c	a	b	a	b	a	b	a	c	a	c
$(T'[1\ \ldots\ i])$	0	0	1	2	3	4	5	4	5	6	7	0

Der String-Matching Automat $\mathcal{A}_p$ zu P wird durch die Übergangsfunktion δ bestimmt. Diese wird duch folgende Prozedur berechnet:

```
ÜBERGANGSFUNKTION($P, \Sigma$)
  Setze $m :=$ Länge[$P$].
  Für $q = 0$ bis $m$ führe aus:
      Für alle $a \in \Sigma$ führe aus:
          $k := \min(m, q + 1)$.
          Solange $P[1\ \ldots\ k]$ kein Suffix von $P[1\ \ldots\ q]a$ ist, setze
          $k := k - 1$;
          setze $\delta(q, a) := k$.
```

Laufzeit: $\mathcal{O}(m \cdot |\Sigma| \cdot m^2)$. Die Prozedur kann verbessert werden, so dass die Laufzeit in $\mathcal{O}(m \cdot |\Sigma|)$ ist.

Mit dem String-Matching-Automat $\mathcal{A}_p$ bzw. der Übergangsfunktion δ kann das String-Matching-Problem mit folgender Prozedur gelöst werden:

```
ENDLICHER-AUTOMAT-MATCHER($T, \delta, m$) zu $\mathcal{A}_p$
1. Setze $n :=$ Länge[$T$] und $q := 0$.
2. Für $i = 1$ bis $n$ führe aus:
3.     Setze $q := \delta(q, T[i])$.
4.     Falls $q = m$ ist dann
5.         setze $s := i - m$ und gib aus
           „$P$ taucht mit Verschiebung $s$ in $T$ auf.“
```

Laufzeit: $\mathcal{O}(n)$.

Korrektheit. Um zu zeigen, dass diese Methode korrekt ist, machen wir uns zunächst klar, dass $\mathrm{suf}_p(T[1\ \dots\ i]a) = \mathrm{suf}_p(P[1\ \dots\ q]a)$ ist, wobei $q = \mathrm{suf}_p(T[1\ \dots\ i])$. Es gilt sogar allgemein:

Für jedes $w \in \Sigma^*$ *und* $a \in \Sigma$ *ist* $\mathit{suf}_p(wa) = \mathit{suf}_p(P[1\ \dots\ q]a)$, *wobei* $q = \mathit{suf}_p(w)$.

$$\begin{array}{cccccccccccccccc}
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & a & (wa) \\
 & & & & & | & | & | & | & | & | & | & | & | & & \\
 & & & & P[& 1 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & q] & a & \\
 & & & & & & & & & | & | & | & | & | & | & \\
 & & & & & & & & P[& 1 & \bullet & \bullet & \bullet & \bullet & r] & (r = \mathrm{suf}_p(wa))
\end{array}$$

Sei $w = w_1\ \dots\ w_\ell$, dann ist

$$\begin{aligned}
w_\ell &= P[q], \\
w_{\ell-1} &= P[q-1], \\
&\ \vdots \\
w_{\ell-m+1} &= P[1].
\end{aligned}$$

Falls nun $r = \max\big\{k : P[1\ \dots\ k] \text{ Suffix von } wa\big\}$, dann ist

$$\begin{aligned}
P[r] &= a, \\
P[r-1] &= w_\ell = P[q], \\
P[r-2] &= w_{\ell-1} = P[q-1], \\
&\ \vdots \\
P[1] &= w_{\ell-r+1} = P[q-r+1],
\end{aligned}$$

d.h.

$$r = \max\left\{k : P[1\ \dots\ k] \text{ Suffix von } P[1\ \dots\ q]a\right\} = \mathrm{suf}_p(P[1\ \dots\ q]a).$$

Korrektheit. Der ENDLICHER-AUTOMAT-MATCHER ist korrekt, wenn für jedes i $(1 \leq \imath \leq n = |T|)$ gilt:

> Falls $\mathcal{A}_p$ bei Abarbeitung von $T[1\ \dots\ i]$ im Zustand m endet, dann taucht P mit Verschiebung $i - m$ in T auf, d.h.
>
> $$P[1\ \dots\ m] = T[i - m + 1\ \dots\ i].$$

Es gilt sogar:

Satz 7.1 *Falls $\mathcal{C}$ Zustandsfunktion von $\mathcal{A}_p$ ist und $T[1\ \dots, n]$ Input in $\mathcal{A}_p$, so gilt*

$$\mathcal{C}(T[1\ \dots\ i]) = suf_p(T[1\ \dots\ i])$$

für alle $i = 1, \dots, n$.

Beweis. Für $i = 1$ gilt $\mathcal{C}(T[1]) = \delta(q_0, T[1]) = \mathrm{suf}_p(T[0])$. Sei für $i \geq 2$ $q := \mathcal{C}(T[1\ \dots\ i-1])$, dann gilt

$$\begin{aligned} \mathcal{C}(T[1\ \dots\ i]) &= \mathcal{C}(T[1\ \dots\ i-1]\ T[i]) = \delta(q, T[i]) && \text{(nach Def. von } \mathcal{C}) \\ &= \mathrm{suf}_p(P[1\ \dots\ q]\ T[i]) && \text{nach Def. von } \delta) \\ &= \mathrm{suf}_p(T[1\ \dots\ i-1]\ T[i]) && \text{(bereits klar gemacht)} \\ &= \mathrm{suf}_p(T[1\ \dots\ i]). \end{aligned}$$

■

Wenn also $\mathcal{A}_p$ bei Abarbeitung von $T[1\ \dots\ i]$ im Zustand j $(1 \leq j \leq m)$ endet, dann ist $P[1\ \dots\ j] = T[i - j + 1\ \dots\ i]$.

Zusatzinformation. Die Ideen, die im String-Matching-Verfahren mit endlichen Automaten stecken, können zu einem Algorithmus weiter entwickelt werden, der optimale Laufzeit von $\mathcal{O}(n+m)$ hat (Knuth-Morris-Pratt, 1977). Dabei wird nicht extra der Automat $\mathcal{A}_p$ berechnet, sondern die entsprechenden „Informationen“ geschickt direkt aus P gezogen.

7.3 Das Verfahren von **Boyer & Moore**

- „Heuristische“ Vorgehensweise für die Entscheidung, welche Verschiebungen betrachtet werden.
- In der Praxis sehr effizient.

Idee. Text und Muster werden verglichen, wobei das Muster von hinten nach vorne durchlaufen wird. Wird eine Nichtübereinstimmung zwischen Text und Muster gefunden, so wird diese Information bei der Bestimmung der Verschiebung, die als nächstes betrachtet wird, verwendet.

Beispiel.

T:	a	b	a	a	**c**	a	b	b	c	c	a	b	a	b	a	a	c	a	b
					$\not\uparrow$														
P:	c	a	b	a	**b**	**a**													

In T taucht bei der gefundenen Nichtübereinstimmung ein c auf. Die nächste Verschiebung, bei der überhaupt „eine Chance “ auf Übereinstimmung zwischen P und T besteht, sollte so sein, dass unter diesem „schlechten Buchstaben“ c in T auch in P ein c steht.

- Verschiebung um 4 nach rechts, da das rechteste Vorkommen von c in P „um 4 weiter vorne“ ist.

T:	a	b	a	a	c	a	b	b	c	**c**	a	b	a	b	a	a	c	a	b
										$\not\uparrow$									
P:					c	a	b	a	b	**a**									

- Verschiebung um 5 nach rechts, da das rechteste Vorkommen von c in P „um 5 weiter vorne“ ist.

T:	a	b	a	a	c	a	b	b	c	c	a	b	a	b	a	a	c	a	b
P:										**c**	**a**	**b**	**a**	**b**	**a**				

- Übereinstimmung. P taucht mit Verschiebung 9 in T auf. Betrachte danach Verschienung um 1 nach rechts.

T:	a	b	a	a	c	a	b	b	c	c	a	b	a	b	**a**	a	c	a	b
															$\not\uparrow$				
P:											c	a	b	a	**b**	**a**			

- Verschiebung um -1, da das rechteste Vorkommen von a in P „um 1 weiter hinten“ ist. Diese Verschiebung ist nicht sinnvoll!

Vorgehensweise. Für die „schlechter-Buchstabe“-Heuristik wird vorab das rechteste Vorkommen eines jeden Buchstaben des Alphabets Σ im Muster P berechnet.

Definiere zu P Funktion $r_p : \Sigma \longrightarrow \{1, \ldots, m\}$, wobei für $a \in \Sigma$:

$$r_p(a) = \begin{cases} 0, & \text{falls } a \text{ in } P[1 \ \ldots \ m] \text{ nicht vorkommt,} \\ k, & \text{falls } a = P[k] \text{ und } a \neq P[i] \text{ für alle } k < i \leq m \end{cases}$$

(k ist rechteste Position in P, an der a vorkommt).

Aus $r_p(a)$ kann die Verschiebung berechnet werden.

- **Fall 1.** Falls die erste Nichtübereinstimmung bei $T[s+j] = a \neq P[j]$ eintritt und $r_p(a) < j$, so verschiebe um $j - r_p(a)$.
- **Fall 2.** Falls erste Nichtübereinstimmung bei $T[s+j] = a \neq P[j]$ eintritt und $r_p > j$, so verschiebe um 1.

Das heißt: als nächste Verschiebung nach der gerade betrachteten Verschiebung s wird in Fall 1 die Verschiebung $s + j - r_p(a)$ und in Fall 2 die Verschiebung $s + 1$ betrachtet. Dabei wird der Test von hinten nach vorne ausgeführt, d.h. als erstes wird $T[s + j - r_p(a) + m]$ bzw. $T[s + 1 + m]$ mit $P[m]$ verglichen.

Die „rechtesten Vorkommen“ $r_p(a)$ für $a \in \Sigma$ können folgendermaßen berechnet werden:

```
RECHTESTES-VORKOMMEN($\Sigma, P$)
1. Für $a \in \Sigma$ setze $r_p(a) := 0$.
2. Für $j = 1$ bis $m$ setze $r_p(P[j]) := j$.
3. Gib die Werte $r_p(a)$ für $a \in \Sigma$ aus.
```

Die Laufzeit ist in $\mathcal{O}(|\Sigma| + m)$.

Unter Benutzung von r_p kann dann String-Matching durch das folgende Verfahren SCHLECHTER-BUCHSTABE (oder (Boyer-Moore-1) gelöst werden:

```
SCHLECHTER-BUCHSTABE-VERFAHREN($T, P, r_p$)
1. Setze $n :=$ Länge[$T$] und $m :=$ Länge[$P$],
2. setze $s := 0$.
3. Solange $s \leq n - m$ ist, führe aus:
4.         $j := m$.
5.         Solange $j > 0$ und $P[j] = T[s + j]$ führe aus:
6.                 $j := j - 1$.
```

```
    Falls $j = 0$ dann
        gib aus „$P$ tritt mit Verschiebung $s$ in $T$ auf“,
        setze $s := s + 1$;
    sonst:
        Falls $r_p(T[s + j]) < j$
            setze $s := s + j - r_p(T[s + j])$;
        ansonsten
            setze $s := s + 1$.
```

Die Laufzeit ist im worst-case $\mathcal{O}(m \cdot (n - m + 1))$, wobei der worst-case wieder im Fall $P = a \ \ldots \ a$ (m-mal) und $T = a \ \ldots \ a$ (n-mal) auftritt. Zusätzlich muss der Aufwand für die Berechnung von r_p gerechnet werden.

Das Verfahren kann noch weiter „optimiert“ werden, indem man weitere Informationen über P nutzt.

Beobachtung.

T	a	b	a	a	c	a	b	b	c	c	a	b	a	b	a	c	a	c	b
P				c	c	a	b	a	c	b	**a**	**b**	**a**						

Der Algorithmus SCHLECHTER-BUCHSTABE induziert hier eine Verschiebung um 1. Beachte nun außerdem, dass nur dann eine Chance auf Übereinstimmung zwischen P und T besteht, wenn die Verschiebung so gewählt wird, dass der „gute Suffix“ a b a von P, den wir gerade auch in T „gefunden haben“, auf eine Übereinstimmung „weiter vorne in P“ trifft.

- Suche rechtestes Vorkommen des „guten Suffix“ weiter vorne in P.
- Verschiebung um 5 nach rechts.

Vorgehensweise. Für die GUTER-SUFFIX-Heuristik wird vorab für jede Position j im Muster P das rechteste k bestimmt ($0 \leq k \leq m$), so dass $P[j + 1 \ \ldots \ m]$ Suffix von $P[1 \ \ldots \ k]$ ist oder $P[1 \ \ldots \ k]$ Suffix von $P[j + 1 \ \ldots \ m]$.

Beispiel.

T	a	a	b	a	a	c	a	b	b	c	a	c	a	b	a	b	a	c	c	a	b	a	b
											$\not\uparrow$												
P			a	b	a	c	a	b	a	a	c	**c**	**a**	**b**	**a**								

($m = 13,\ j = 9$.)

c a b a $= P[10\ \ldots\ 13]$ ist Suffix von $P[1\ \ldots\ 7]$.

T	a	a	b	a	a	c	a	b	b	c	a	c	a	b	a	b	a	c	c	a	b	a	b
																$\not\uparrow$							
P										a	b	a	c	a	b	a	**a**	**c**	**c**	**a**	**b**	**a**	

($j = 7$.)

a c c a b a $= P[8\ \ldots\ 13]$ ist für kein $k < m$ Suffix von $P[1\ \ldots\ k]$, aber $P[1\ 2\ 3] =$ a b a ist Suffix von $P[8\ \ldots\ 13]$.

Definiere zu P Funktion $s_p\ : \{1, \ldots, m-1\} \longrightarrow \{0, 1, \ldots, m-1\}$, wobei

$$\begin{aligned} s_p(j) :=\ & \max\big\{k \in \{0, 1, \ldots, m-1\}, \\ & P[j+1\ \ldots\ m] \text{ Suffix von } P[1\ \ldots\ k] \text{ oder} \\ & P[1\ \ldots\ k] \text{ Suffix von } P[j+1\ \ldots\ m]\big\}. \end{aligned}$$

Falls dann

$$\begin{aligned} T[s+j] &\neq P[j], \\ \text{aber } T[s+j+1\ \ldots\ s+m] &= P[j+1\ \ldots\ m] \\ \text{und } s_p(j) &= k, \end{aligned}$$

so verschiebe um $m-k$, d.h. betrachte als nächstes die Verschiebung $s+m-k$.

Bemerkung. $s_p(j)$ kann für P in $\mathcal{O}(m)$ Zeit vorausberechnet werden.

Das GUTER-SUFFIX-VERFAHREN (BOYER-MOORE-2)

Das Verfahren kann analog zu SCHLECHTER-BUCHSTABE-Verfahren formuliert werden. Beide Verfahren können kombiniert werden, indem man als Verschiebung „das Beste“ aus SCHLECHTER BUCHSTABE und GUTER SUFFIX wählt, d.h. falls

$$P[j] \neq T[s+j], \text{ aber } P[j+1\ \ldots\ m] = T[s+j+1\ \ldots\ s+m],$$

so wähle als Verschiebung das Maximum von $s+m-s_p(j)$ und $s+j-r_p(P[j])$. Da GUTER SUFFIX immer eine Verschiebung nach rechts induziert, muss Fall 2 ($r_p > j$) bei SCHLECHTER BUCHSTABE nicht betrachtet werden.

Laufzeit. Das Boyer-Moore(T, P, r_p, s_p)-Verfahren hat ebenfalls worst-case-Laufzeit $\mathcal{O}(m(n-m+1))$ bei vorausberechnetem r_p und s_p.

Kapitel 8

Parallele Algorithmen

Der Literaturtip. Zu parallelen Algorithmen vgl. [GR90].

Beim Entwurf paralleler Algorithmen ist in stärkerem Maß als bei sequentiellen Algorithmen das zugrundeliegende Berechnungsmodell von besonderer Bedeutung. Das Hauptmerkmal paralleler Algorithmen ist, dass anstatt eines Prozessors mehrere Prozessoren gleichzeitig, mehr oder weniger unabhängig voneinander, Operationen ausführen können. Mögliche Berechnungsmodelle, und auch wirkliche Parallelrechner, unterscheiden sich etwa darin, wie unabhängig die Prozessoren voneinander sind (gleichartige bzw. unterschiedliche Operationen in einem parallelen Schritt; Kommunikation kostet mehr oder weniger viel Berechnungszeit, etc.)

Ein sinnvolles Berechnungsmodell für den Entwurf von parallelen Algorithmen auf relativ hohem Abstraktionsniveau und für theoretische Betrachtungen wie etwa Komplexitätsbetrachtungen ist die PRAM (Parallel Random Access Machine), eine parallele Version der RAM (siehe Einführung). Das Konzept einer PRAM, das am meisten zitiert und für Algorithmen zugrundegelegt wird, stammt von Fortune & Wyllie (1978).

8.1 Das PRAM Modell

- unbegrenzte Prozessorenzahl;
- unbegrenzter globaler Speicher, auf den alle Prozessoren Zugriff haben;
- Speicherzellen des globalen Speichers wie bei RAM;
- Operationen, die jeder einzelne Prozessor ausführen kann wie bei RAM;
- Jeder Prozessor hat einen eigenen lokalen Speicher, der ebenfalls unbegrenzt ist.

Jeder Prozessor darf nur auf seinen eigenen lokalen Speicher zugreifen, nicht aber auf den lokalen Speicher eines anderen Prozessors. Alle Prozessoren dürfen auf den globalen Speicher zugreifen.

Problem. Was passiert, wenn mehrere Prozessoren gleichzeitig aus derselben Speicherstelle des globalen Speichers lesen wollen oder dieselbe Speicherzelle des globalen Speichers beschreiben wollen? Alle der folgenden Kombinationen sind denkbar und werden als Modelle untersucht bzw. beim Algorithmus zugrundegelegt:

gleichzeitiges Lesen erlaubt CR (**concurrent read**)	gleichzeitiges Lesen verboten ER (**exclusive read**)
gleichzeitiges Schreiben erlaubt CW (**concurrent write**)	gleichzeitiges Schreiben verboten EW (**exclusive write**)

Wir wollen uns hier auf die CREW-PRAM beschränken.

8.2 Komplexität von parallelen Algorithmen

Wir betrachten wieder die Laufzeit im worst-case. Beim Ablauf eines parallelen Algorithmus werden N Operationen, die gleichzeitig von N Prozessoren ausgeführt werden, als ein (paralleler) Berechnungsschritt gezählt. Dann ist die **Laufzeit** $T_{\mathcal{A}}(n)$ eines parallelen Algorithmus $\mathcal{A}$ definiert als

Definition 8.1 $T_{\mathcal{A}}(n) := \max\{$*Anzahl der Berechnungsschritte von* $\mathcal{A}$ *bei Eingabe des Problembeispiels* I*, wobei* I *ein Problembeispiel der Größe* n *ist*$\}$.

Bemerkung. Die Berechnung beginnt, wenn der erste Prozessor aktiv wird, also irgendeine Operation ausführt, und endet, nachdem der letzte Prozessor inaktiv geworden ist, also keine Operation mehr ausführt. Die Prozessoren arbeiten synchron.

Für die Güte eines parallelen Algorithmus ist neben der Laufzeit (und dem Speicherplatzbedarf, den wir hier nicht betrachten) die Anzahl der benötigten Prozessoren relevant. Die **Prozessorenzahl** $P_{\mathcal{A}}(n)$ eines parallelen Algorithmus $\mathcal{A}$ ist definiert als

Definition 8.2 $P_{\mathcal{A}}(n) := \max\{$*Anzahl an Prozessoren, die während des Ablaufs von* $\mathcal{A}$ *bei Eingabe des Problembeispiels* I *gleichzeitig aktiv ist, wobei* I *ein Problembeispiel der Größe* n *ist*$\}$.

(Dies entspricht also wieder einer worst-case-Abschätzung).

Ein paralleler Algorithmus $\mathcal{A}$ steht natürlich in Konkurrenz zu sequentiellen Algorithmen. Dabei interessiert uns vor allem:

$$\text{speed-up}(\mathcal{A}) := \frac{\begin{array}{c}\text{worst-case Laufzeit des schnellsten}\\ \text{bekannten sequentiellen Algorithmus}\end{array}}{\begin{array}{c}\text{worst-case Laufzeit des}\\ \text{parallelen Algorithmus } \mathcal{A}\end{array}}$$

Je größer der speed-up($\mathcal{A}$) ist, umso besser ist natürlich der Algorithmus $\mathcal{A}$. Im Idealfall hofft man natürlich, einen speed-up von N zu erreichen, wenn etwa $P_{\mathcal{A}}(n) =: N$ (für alle n). Bei PRAM-Algorithmen geht man allerdings meist davon aus, dass die Prozessorenzahl abhängig von der Problemgröße ist (wie aus der Definition 8.2 von $P_{\mathcal{A}}(n)$ ersichtlich). Man betrachtet also bei $T_{\mathcal{A}}$, $P_{\mathcal{A}}$, speed-up($\mathcal{A}$), usw. asymptotisches Verhalten.

Ein Qualitätsmaß für einen parallelen Algorithmus $\mathcal{A}$, in welches also sinnvoll Laufzeit und Prozessorenzahl eingehen, sind die **Kosten** $C_{\mathcal{A}}$ von $\mathcal{A}$:

$$C_{\mathcal{A}}(n) := T_{\mathcal{A}}(n) \cdot P_{\mathcal{A}}(n)$$

Sind asymptotisches Wachstum von $C_{\mathcal{A}}(n)$ und die schärfste asymptotische untere Schranke für die Laufzeit eines sequentiellen Algorithmus gleich, so nennt man $\mathcal{A}$ **kostenoptimal**. Kennt man keine gute untere Schranke, so betrachtet man die **Effizienz** $E_{\mathcal{A}}$ von $\mathcal{A}$:

$$E_{\mathcal{A}}(n) := \frac{\begin{array}{c}\text{worst-case Laufzeit des schnellsten}\\ \text{bekannten sequentiellen Algorithmus}\end{array}}{\text{Kosten des parallelen Algorithmus } \mathcal{A}}$$

Man kann davon ausgehen, dass $E_{\mathcal{A}}(n) \leq 1$, denn sonst könnte man aus $\mathcal{A}$ einen sequentiellen Algorithmus ableiten mit (asymptotisch) besserer Laufzeit, als die Laufzeit des schnellsten bekannten sequentiellen Algorithmus, indem man einen Prozessor alle Operationen eines parallelen Berechnungsschritts von $\mathcal{A}$ hintereinander ausführen lässt. Dann ist die Laufzeit $\approx C_{\mathcal{A}}(n)$.

8.3 Die Komplexitätsklassen

Die am meisten untersuchte Klasse in Zusammenhang mit parallelen Algorithmen ist die Klasse $\mathcal{NC}$:

Definition 8.3 (Nick's Class nach Nicholas Pippinger) *Die Klasse* $\mathcal{NC}$ *ist die Klasse der Probleme, die durch einen parallelen Algorithmus* $\mathcal{A}$ *mit polylogarithmischer Laufzeit und polynomialer Prozessorenzahl gelöst werden kann, d.h.* $T_{\mathcal{A}}(n) \in \mathcal{O}((\log_n)^{k_1})$ *mit* k_1 *Konstante, und* $P_{\mathcal{A}}(n) \in \mathcal{O}(n^{k_2})$ *mit* k_2 *Konstante.*

Eine wichtige offene Frage ist: Gilt $\mathcal{P} = \mathcal{NC}$? Vermutung ist: „nein“. Wie bei der Frage, ob $\mathcal{P} = \mathcal{NP}$ ist, gibt es auch hier ein Vollständigkeitskonzept. Parallele Laufzeitkomplexität steht in engem Zusammenhang zu sequentieller Speicherplatzkomplexität.

Definition 8.4 (Steve's Class nach Stephan Cook) *Die Klasse* $\mathcal{SC}$ *ist die Klasse der Probleme, die durch einen sequentiellen Algorithmus mit polylogarithmischem Speicherplatzbedarf und polynomialer Laufzeit gelöst werden kann.*

Die offene Frage ist hier: $\mathcal{NC} = \mathcal{SC}$?

Einschub. Verschiedene PRAM-Modelle wie CREW-PRAM und EREW-PRAM, oder ein Modell, bei dem zu Beginn der Berechnung eine beliebige Anzahl von Prozessoren aktiv sind gegenüber einem Modell, bei dem zu Beginn nur ein Prozessor aktiv ist und alle anderen erst „aufgeweckt“ werden müssen, unterscheiden sich in der Laufzeit im wesentlichen nur um einen Faktor $\log N$ (bei N Prozessoren, die gleichzeitig lesen wollen, bzw. aktiviert werden sollen). Dies lässt sich durch den folgenden Übertragungsalgorithmus, bei dem ein Wert an N Prozessoren auf einer EREW-PRAM übertragen wird, veranschaulichen.

Formale Beschreibung des Algorithmus BROADCAST.

Eingabe: Wert m.
Datenstruktur: Array A der Länge N im globalen Speicher.

```
BROADCAST($N$)
  $P_1$ liest $m$,
  kopiert $m$ in den eigenen Speicher
  und schreibt $m$ in $A[1]$.
  Für $i = 0$ bis $\lceil \log N \rceil - 1$ führe aus:          (MERGEWITHHELP)
      Für alle $j$ ($2^i + 1 \leq j \leq 2^{i+1}$) führe parallel aus:
          $P_j$ liest $m$ aus $A[j - 2^i]$,
          kopiert $m$ in den eigenen Speicher
          und schreibt $m$ in $A[j]$.
```

Die Laufzeit ist in $\mathcal{O}(\log N)$.

8.4 Parallele Basisalgorithmen

8.4.1 Berechnung von Summen

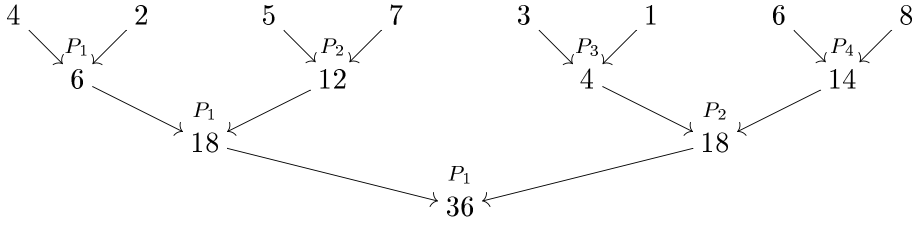
Ziel ist die Berechnung der Summe (bzw. Produkt, Minimum, Maximum, allgemein n-fache binäre Operation) aus n Werten $a_1, \ldots a_n$.

Formale Beschreibung des Algorithmus SUMME.

Eingabe: n Werte $a_1, \ldots, a_n$, o.B.d.A. sei $n = 2^m$.
Ausgabe: $\sum_{i=1}^{n} a_i$

```
SUMME($a_1, \ldots, a_n$)
  Für $i = 1$ bis $m$ führe aus:
      Für alle $j$, $1 \leq j \leq \frac{n}{2^i}$ führe parallel aus:
          Prozessor $P_j$ berechnet $a_j := a_{2j-1} + a_{2j}$.
  Gib $a_1$ aus.
```

Beispiel: Die Berechnung einer Summe.



Es ist $P_{\mathcal{A}}(n) = n/2$, $T_{\mathcal{A}}(n) \in \mathcal{O}(\log n)$, $C_{\mathcal{A}} \in \mathcal{O}(n \log n)$. Der Algorithmus $\mathcal{A}$ ist also nicht kostenoptimal.

Bemerkung. Das logische ODER aus n boooleschen Variablen (0 und 1) kann ebenso auf einer CREW-PRAM in $\mathcal{O}(\log n)$ berechnet werden. Auf einer CRCW-PRAM könnte man mit n Prozessoren das logische ODER aus n Werten in $\mathcal{O}(1)$ bestimmen, wenn concurrent-write aufgelöst würde, indem mehrere Prozessoren an dieselbe Speicherzelle schreiben dürften g.d.w. sie denselben Wert schreiben wollen:

```
1. Für alle $i$, $1 \le i \le n$ führe parallel aus:
2.     Prozessor $P_i$ liest $i$-ten Eingabewert $x_i$.
3.     Falls $x_i = 1$:
4.         $P_i$ schreibt Wert 1 in die Speicherzelle „1“
           des globalen Speichers.
```

Geht man davon aus, dass zu Beginn der Berechnung in allen Speicherzellen des globalen Speichers 0 steht, so hat das Ergebnis des logischen ODER den Wert 1 g.d.w. am Ende in Speicherzelle „1“ eine 1 steht.

Zurück zum Algorithmus $\mathcal{A}$ zur Berechnung von SUMME. Durch Rescheduling lässt sich aus $\mathcal{A}$ ein ein paralleler Algorithmus $\mathcal{A}'$ gewinnen, der Kosten $C_{\mathcal{A}'}(n) \in \mathcal{O}(n)$ hat. Dazu werden anstatt n Prozessoren $\lceil \frac{n}{\log n} \rceil$ Prozessoren verwendet. Die $\frac{n}{2}$ Operationen, die im ersten Durchlauf von $\mathcal{A}$ von $\frac{n}{2}$ Prozessoren in einem parallelen Berechnungsschritt ausgeführt werden, werden stattdessen von $\lceil \frac{n}{\log n} \rceil$ in höchstens $\frac{\log n}{2}$ parallelen Berechnungsschritten ausgeführt, bzw. allgemein im i-ten Durchlauf $\frac{n}{2^i}$ Operationen in höchstens $\frac{\log n}{2^i}$ Berechnungsschritten. Damit ergibt sich insgesamt

$$T_{\mathcal{A}}(n) \le c \cdot \sum_{i=1}^{\log n} \frac{\log n}{2^i} = c \cdot \log n \underbrace{\sum_{i=1}^{\log n} \frac{1}{2^i}}_{\le 1} \in \mathcal{O}(\log n).$$

Es gilt $P_{\mathcal{A}'}(n) = \lceil n / \log n \rceil$, also $C_{\mathcal{A}'} \in \mathcal{O}(n)$.

8.4.2 Berechnung von Präfixsummen

Wir berechnen die Präfixsummen $\sum_{i=1}^{k} a_i$ $(1 \le k \le n)$ aus den n gegebenen Werten $a_1, \ldots, a_n$ (bzw. Produkt, Minimum, Maximum). Aus technischen Gründen nennen wir die Eingabewerte $a_n, a_{n+1}, \ldots, a_{2n-1}$; es sei wieder o.B.d.A. $n = 2^m$. Das Verfahren besteht aus zwei Phasen. In der ersten wird SUMME$(a_n, \ldots, a_{2n+1})$ ausgeführt. In der zweiten Phase werden aus dem Ergebnis von SUMME und aus Zwischenergebnissen des Verfahrens die $\sum_{i=n}^{k} a_i$ $(n \le k \le 2n-1)$ als Differenzen geeignet berechnet.

Formale Beschreibung des Algorithmus

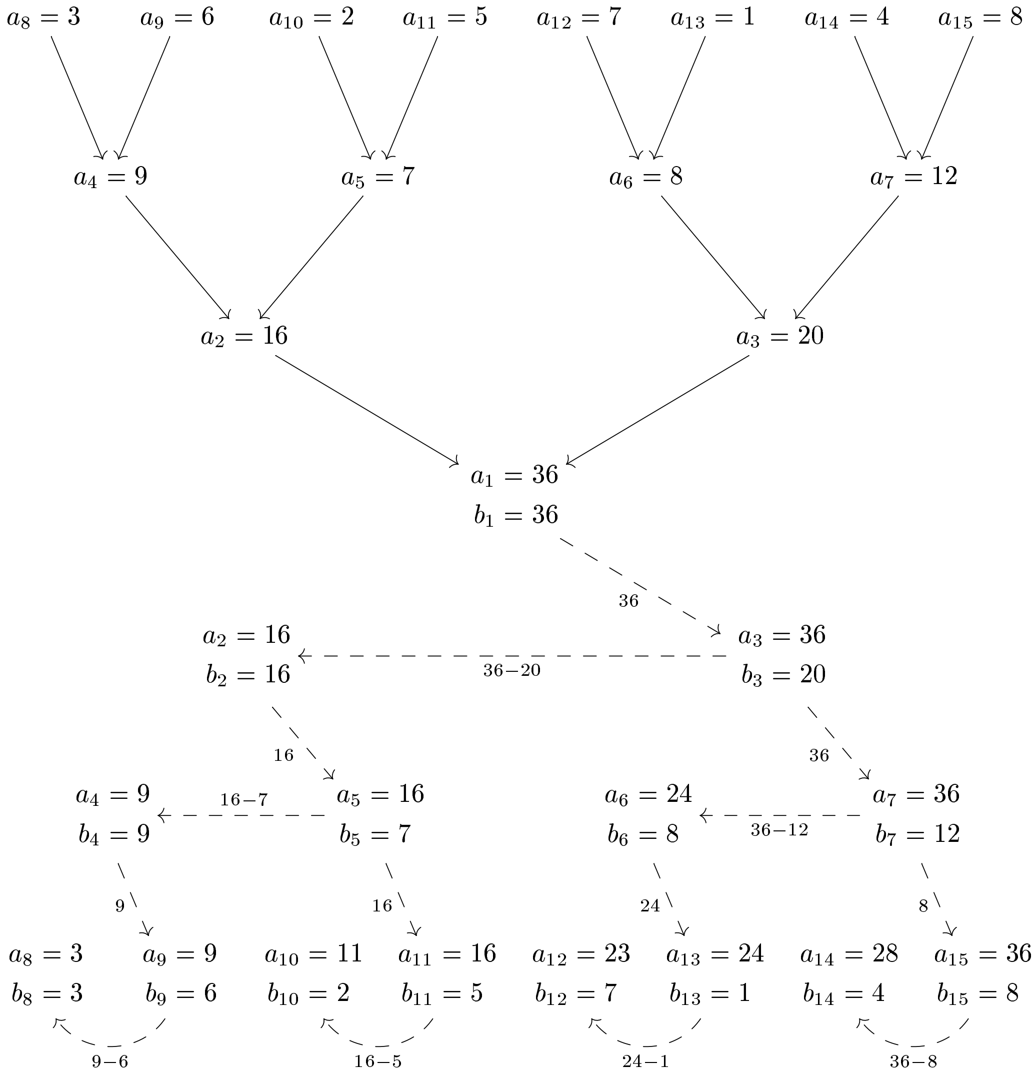
Eingabe: n Werte $a_n, \ldots, a_{2n-1}$; $n = 2^m$,
Ausgabe: Die n Präfixsummen $\sum_{i=1}^{k} a_i$ für $1 \leq k \leq n$.

```
PRÄFIXSUMME($a_n, \ldots, a_{2n-1}$)
Für $j = m-1$ bis 0 führe aus:
    Für alle $i$ ($2^j \leq i \leq 2^{j+1}-1$) führe parallel aus:
        $a_i := a_{2i} + a_{2i+1}$
    Setze $b_1 := a_1$.
Für $j = 1$ bis $m$ führe aus:
    Für alle $i$ ($2^j \leq i \leq 2^{j+1}-1$) führe parallel aus:
        Falls $i$ ungerade ist, setze $b_i := b_{\frac{i-1}{2}}$.
        Falls $i$ gerade ist, setze $b_i := b_{\frac{i}{2}} - a_{i+1}$.
Das Ergebnis steht in $b_n, \ldots, b_{2n-1}$.
```

Durch Induktion lässt sich leicht zeigen, dass $b_{2^j+\ell} = a_{2^j} + \ldots + a_{2^j+\ell}$ ist für $\ell + 0, \ldots, 2^j - 1$.

Komplexität. Offensichtlich ist die Laufzeit in $\mathcal{O}(\log n)$. Die Prozessorenzahl ist in $\mathcal{O}(n)$, es sind maximal $n/2$ Prozessoren gleichzeitig aktiv. Durch Rescheduling kann die Prozessorenzahl wieder auf $\mathcal{O}(n/\log n)$ bei Laufzeit $\mathcal{O}(\log n)$ reduziert werden, womit die Kosten wieder optimal sind.

Beispiel.



8.4.3 Die Prozedur List Ranking oder Short Cutting

In einer linearen Liste (oder allgemeiner einem Wurzelbaum, in dem jedes Element i nur seinen direkten Vorgänger kennt) sollen alle Elemente das erste Element in der Liste bzw. die Wurzel kennenlernen. Dies kann in $\mathcal{O}(\log n)$, wobei n die Länge der Liste ist, bzw. in $\mathcal{O}(\log h)$, wobei h die Höhe des Wurzelbaumes ist, mit n Prozessen (n Anzahl der Elemente im Baum) realisiert werden. Es sei Vor[root]=root.

Formale Beschreibung des Algorithmus.

Eingabe: Lineare Liste bzw. Wurzelbaum der Höhe h mit n Elementen.
Ausgabe: Vorgänger $\textsc{Vor}[i]$ für jedes Element $1 \leq i \leq n$.

```
List Ranking($n, h$)
Für $j = 1$ bis $\lceil \log h \rceil$ führe aus:
    Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
        Setze $\textsc{Vor}[i] := \textsc{Vor}[\textsc{Vor}[i]]$.
```

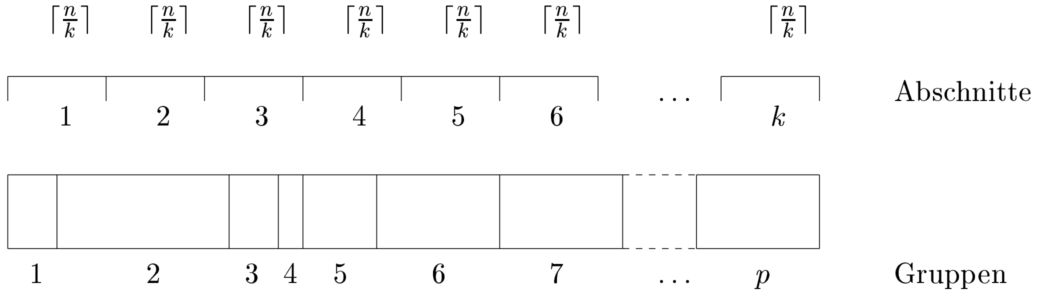
8.4.4 Binäroperationen einer partitionierten Menge mit K Prozessoren

Gegeben. n Werte, die in p Gruppen aufgeteilt sind.

Problem. Mit K Prozessoren sollen die p Werte bestimmt werden, die sich als Binärverbindungen (also Summe, Minimum etc.) der Werte der p Gruppen ergeben. Die Laufzeit soll $t(n)$ sein, mit

$$t(n) \in \begin{cases} \lceil \frac{n}{K} \rceil - 1 + \log K, & \text{falls } n > K \\ \log n, & \text{sonst.} \end{cases}$$

Mit $K \geq n$ Prozessoren können die p Binärverbindungen wie üblich in $\mathcal{O}(\log n)$ berechnet werden. Falls $K < n$ ist, teile die n Werte in K Abschnitte auf. Jeder Abschnitt erhält einen Prozessor.



Die Werte eines Abschnitts gehören entweder alle zu derselben Gruppe oder zu verschiedenen Gruppen. Gehören alle Werte zu derselben Gruppe, so wird die Binärverbindung aus diesen berechnet, ansonsten, falls sie zu m Gruppen gehören, werden m solche Binärverbindungen berechnet. Alle diese werden mit den K Prozessoren jeweils sequentiell in höchstens $\lceil \frac{n}{K} \rceil - 1$ Schritten berechnet. Gehören alle Werte einer Gruppe zu demselben Abschnitt, so sind nach diesen $\lceil \frac{n}{K} \rceil - 1$ Schritten ihre Binärverbindungen endgültig berechnet. Für jeden Abschnitt sind jedoch höchstens 2 (bzw. 1)

Ergebnisse noch nicht endgültig, sondern müssen mit anderen zusammengefasst werden. Falls also für jede Gruppe G_i $(1 \leq i \leq p)$ die Anzahl der noch zusammenzufassenden Teilergebnisse n_i ist, so gilt

$$\sum_{i=1}^{p} n_i \leq 2K - 2.$$

Ordne den Gruppen G_i $(1 \leq i \leq p)$ jeweils $\lfloor \frac{n_i}{2} \rfloor$ Prozessoren zu, dann können diese in $\log n_i$ Zeit die endgültigen Ergebnisse bestimmen. Da $n_i \leq K$ gilt, ist dies in $\log K$. Die Anzahl an Prozessoren ist ausreichend, da

$$\sum_{i=1}^{p} \lfloor \tfrac{n_i}{2} \rfloor \leq \frac{1}{2} \cdot (2K - 2) < K.$$

8.5 Ein paralleler Algorithmus für die Berechnung der Zusammenhangskomponenten

Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit $V = \{1, \ldots, n\}$. Bestimme durch einen CREW-PRAM-Algorithmus die Zusammenhangskomponenten von G.

Idee. Zunächst wird jeder Knoten als eine aktuelle Zusammenhangskomponente aufgefasst. Der Algorithmus besteht aus maximal $\lceil \log n \rceil$ Phasen, wobei in jeder Phase gewisse aktuelle Zusammenhangskomponenten zu neuen Zusammenhangskomponenten vereinigt werden. In einer Phase wird zunächst für alle Knoten parallel die aktuelle Zusammenhangskomponente (ungleich der eigenen) kleinster Nummer gewählt, die einen Nachbarknoten enthält. Dann wird für alle Knoten i parallel die aktuelle Zusammenhangskomponente kleinster Nummer gewählt unter allen zuvor gewählten Zusammenhangskomponenten, die benachbart sind zu einem Knoten, der in derselben Komponente wie i liegt. Jede Komponente kann nun mit der so bestimmten Zusammenhangskomponente kleinster Nummer zusammengefasst werden. Da bei diesem Schritt Ketten von Zusammenhangskomponenten entstehen können, muss in einer anschließenden Berechnung für alle Zusammenhangskomponenten, die zuvor zusammengefasst worden sind, eine gemeinsame neue Nummer bestimmt werden. Dies wird mit List Ranking realisiert. In jeder Phase wird die Zahl der aktuellen Zusammenhangskomponenten, die echte Subgraphen von Zusammenhangskomponenten von G sind, halbiert. Daher endet der ALgorithmus tatsächlich nach maximal $\lceil \log n \rceil$ Phasen.

Der Algorithmus ZUSAMMENHANG(G)
(Chandra, Hirschberg & Sarwarte 1979)

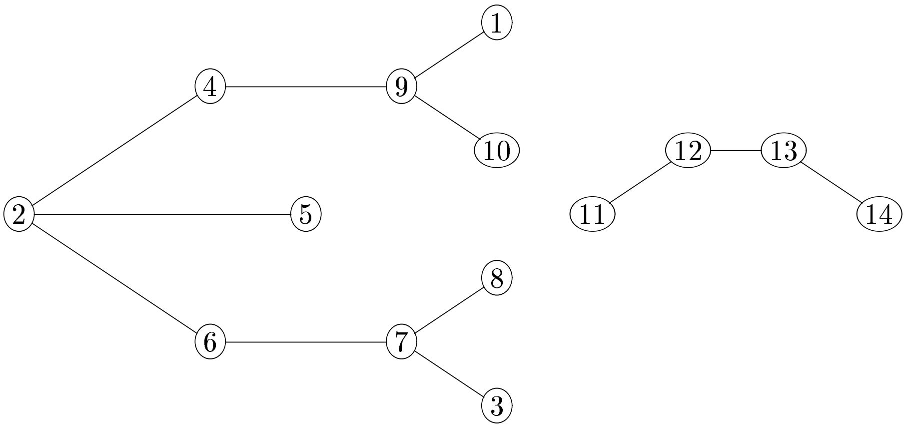
Eingabe: $G = (V, E)$, $V = \{1, \ldots, n\}$
Ausgabe: Zu $i \in V$ steht in $K[i]$ kleinster Knoten, der in derselben Zusammenhangskomponente wie i in G liegt.
Datenstruktur:

- Array K der Länge n ($K[i]$ enthält Nummer der aktuellen Zusammenhangskomponente, in der i liegt.)
- Array N der Länge n ($N[i]$ enthält die Nummer der aktuellen Zusammenhangskomponente kleinster Nummer, in der ein Nachbar von i bzw. eines Knoten aus $K[i]$ liegt).

```
ZUSAMMENHANG($G$)
 1. Für alle $i$, $1 \leq i \leq n$ führe parallel aus:
 2.      $K[i] := i$.
 3. Für $\ell = 1$ bis $\lceil \log n \rceil$ führe aus:
 4.      Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
 5.           $N[i] := \min\{K[j] : \{i, j\} \in E$ und $K[i] \neq K[j]\}$.
 6.           Falls kein solches $j$ existiert, setze $N[i] := K[i]$.
 7.      Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
 8.           $N[i] := \min\{N[j] : K[i] = K[j], N[j] \neq K[j]\}$.
 9.      Für alle $i$ $(1 \leq i \leq n)$ mit $N[N[i]] = K[i]$ führe parallel aus:
10.           $N[i] := \min\{N[i], K[i]\}$.
11.      Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
12.           $K[i] := N[i]$.
13.      Für $m = 1$ bis $\lceil \log n \rceil$ führe aus:
14.           Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
15.                $K[i] := K[K[i]]$.
```

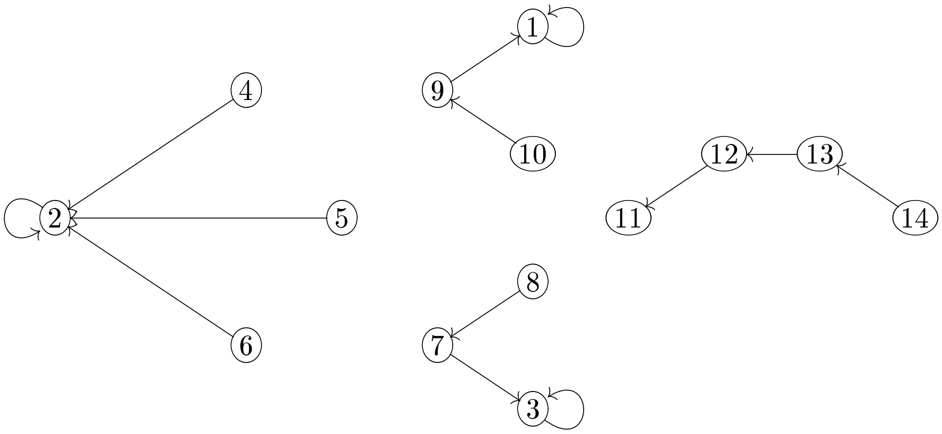
Komplexität: Die Laufzeit ist in $\mathcal{O}(\log^2 n)$: Schleife 3. hat Tiefe $\log n$ und innerhalb von 3. werden mehrere Minimum-Berechnungen vom Aufwand $\mathcal{O}(\log n)$ vorgenommen. 13. ist eine Ausführung von LIST RANKING und benötigt $\mathcal{O}(\log n)$. Die benötigte Prozessorenzahl ist zunächst n^2. Durch Rescheduling kann diese auf $\lceil \frac{n^2}{\log n} \rceil$ gesenkt werden, womit die Kosten in $\mathcal{O}(n^2 \log n)$ sind. Dies ist nicht kostenoptimal.

Beispiel. Wir betrachten folgenden Graphen:

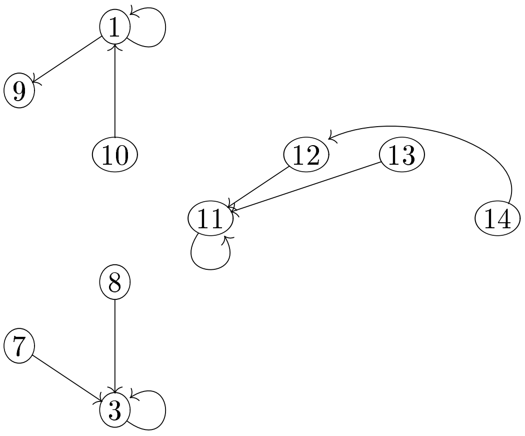


Dann ist $n = 14$, $\lceil \log n \rceil = 4$. Der Algorithmus ZUSAMMENHANG geht wie folgt vor:

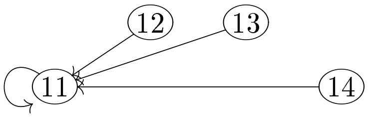
```
 1. $K = [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14]$
 3.     $\ell = 1$
 4.         $N = [9\ 4\ 7\ 2\ 2\ 2\ 3\ 7\ 1\ 9\ 12\ 11\ 12\ 13]$
 7.         $N = [9\ 4\ 7\ 2\ 2\ 2\ 3\ 7\ 1\ 9\ 12\ 11\ 12\ 13]$
 9.         $N = [1\ 2\ 3\ 2\ 2\ 2\ 3\ 7\ 1\ 9\ 11\ 11\ 12\ 13]$
11.         $K = [1\ 2\ 3\ 2\ 2\ 2\ 3\ 7\ 1\ 9\ 11\ 11\ 12\ 13]$
```



```
13.         $m = 1$
14.             $K = [1\ 2\ 3\ 2\ 2\ 2\ 3\ 3\ 1\ 1\ 11\ 11\ 11\ 12]$
```



```
13.         $m = 2$
14.             $K = [1\ 2\ 3\ 2\ 2\ 2\ 3\ 3\ 1\ 1\ 11\ 11\ 11\ 11]$
```



```
13.         Für $m = 3, 4$ ergibt sich keine Änderung mehr.
 3.     $\ell = 2$
 4.         $N = [1\ 2\ 3\ 1\ 2\ 3\ 2\ 3\ 2\ 1\ 11\ 11\ 11\ 11]$
 7.         $N = [2\ 1\ 2\ 1\ 1\ 1\ 2\ 2\ 2\ 2\ 11\ 11\ 11\ 11]$
 9.         $N = [1\ 1\ 2\ 1\ 1\ 1\ 2\ 2\ 1\ 1\ 11\ 11\ 11\ 11]$
11.         $K = [1\ 1\ 2\ 1\ 1\ 1\ 2\ 2\ 1\ 1\ 11\ 11\ 11\ 11]$
13.         $m = 1$
14.             $K = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 11\ 11\ 11\ 11]$
13.         Für $m = 2, 3, 4$ ergibt sich keine Änderung mehr.
 3.     Für $\ell = 3, 4$ ergibt sich keine Änderung mehr.
```

8.6 Modifikation zur Bestimmung eines MST

Der Algorithmus zur Bestimmung der Zusammenhangskomponenten kann geeignet modifiziert werden, so dass er in $\mathcal{O}(\log^2 n)$ Zeit mit n^2 bzw. $\lceil \frac{n^2}{\log n} \rceil$ Prozessoren einen MST in einem gewichteten Graphen bestimmt.

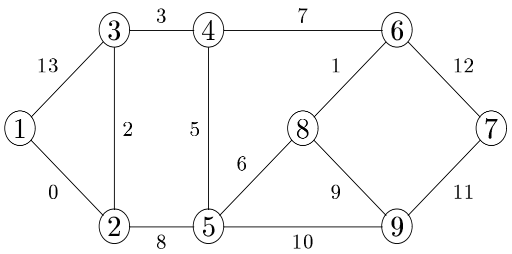
8.6.1 Der Algorithmus

Gegeben. Graph $G = (V, E)$, $V = \{1, \ldots, n\}$ und Kantengewichte c_{ij} für $(i,j) \in V \times V$, wobei $c_{ij} = \infty$, falls $\{i,j\} \notin E$. Zusätzlich werden im Algorithmus Datenstrukturen T und S benutzt. T ist ein Array, das die zum MST gehörenden Kanten enthält. S ist ein Array der Länge n, in dem an der Stelle $S[i]$ die Nummer desjenigen Knoten steht, der in derselben Komponente wie i ist und eine Kante minimalen Gewichts zu einem Knoten einer anderen Komponente hat. In dem Array N steht nun an der Stelle $N[i]$ die Nummer desjenigen Knotens, zu dem es von i aus eine Kante minimalen Gewichts gibt und der in einer anderen Komponente als i liegt.

Im Algorithmus ZUSAMMENHANG werden die Schritte 4. - 12. ersetzt. Der neue Algorithmus sieht damit aus wie folgt:

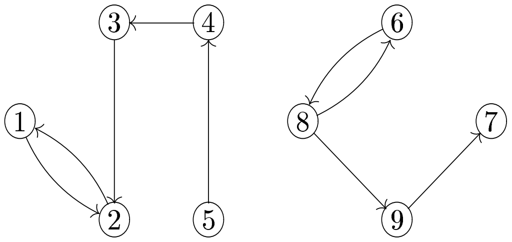
```
MST($G$)
 1. Für alle $i$, $1 \leq i \leq n$ führe parallel aus:
 2.     $K[i] := i$.
 3. Für $\ell = 1$ bis $\lceil \log n \rceil$ führe aus:
 4.     Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
 5.         Finde $k \in V$ mit $K[i] \neq K[k]$ und
            $c_{ik} = \min\{c_{ij} : 1 \leq j \leq n, K[i] \neq K[j]\}$,
 6.         setze $N[i] := k$.
 7.     Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
 8.         Finde $t \in V$ mit $K[i] = K[t]$ und
            $c_{tN[t]} = \min\{c_{jN[j]} : 1 \leq j \leq n, K[i] = K[j]\}$,
 9.         setze $N[i] := N[t]$ und $S[i] := t$.
10.     Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
11.         Falls $N[N[i]] = S[i]$ und $K[i] < K[N[i]]$:
12.             Setze $S[i] := 0$.
13.     Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
14.         Falls $S[i] \neq 0$ und $K[i] = i$ und $c_{N[i],S[i]} \neq \infty$:
15.             Setze $T := T \cup \{N[i], S[i]\}$.
16.         Falls $S[i] \neq 0$:
17.             Setze $K[i] = K[N[i]]$.
18.     Für $m = 1$ bis $\lceil \log n \rceil$ führe aus:
19.         Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
20.             $K[i] := K[K[i]]$.
```

Beispiel. Wir betrachten folgenden Graphen:

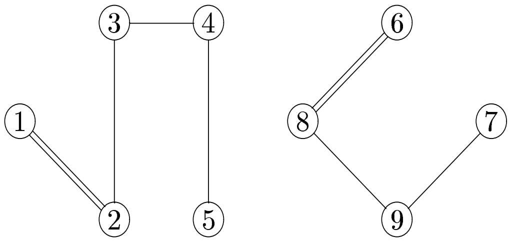


Der Algorithmus geht nun wie folgt vor:

1. $K := [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$
3. $\quad \ell = 1$
4. $\qquad N := [2\ 1\ 2\ 3\ 4\ 8\ 9\ 6\ 8]$
7. $\qquad N := [2\ 1\ 2\ 3\ 4\ 8\ 9\ 6\ 8]$
   $\qquad S := [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$



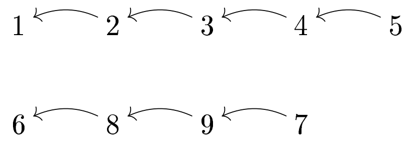
10. $\qquad S := [0\ 2\ 3\ 4\ 5\ 0\ 7\ 8\ 9]$
14. $\qquad T := \{\{1,2\},\{2,3\},\{3,4\},\{4,5\},\{6,8\},\{7,9\},\{8,9\}\}$
16. $\qquad K := [1\ 1\ 2\ 3\ 4\ 6\ 9\ 6\ 8]$



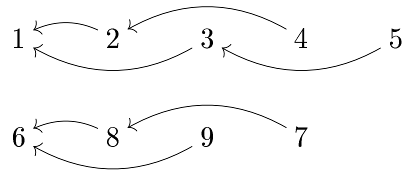
18. $\qquad m = 1$
19. $\qquad\quad K := [1\ 1\ 1\ 2\ 3\ 6\ 8\ 6\ 6]$
18. $\qquad m = 2$
19. $\qquad\quad K := [1\ 1\ 1\ 1\ 1\ 6\ 6\ 6\ 6]$
18. $\qquad$ Für $m = 3$ und $m = 4$ ergibt sich keine Änderung mehr.

Die Schritte 18–20 lassen sich wie folgt darstellen:

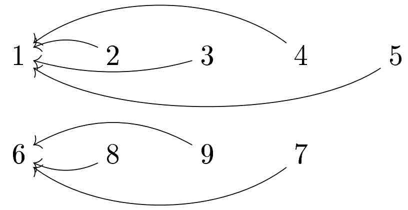
- $m = 0$.



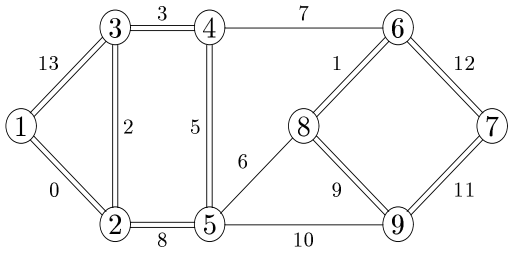
- $m = 1$.



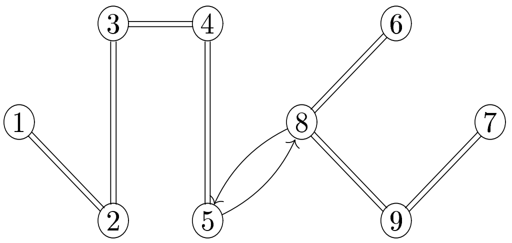
- $m = 2$.



- Ergebnis:

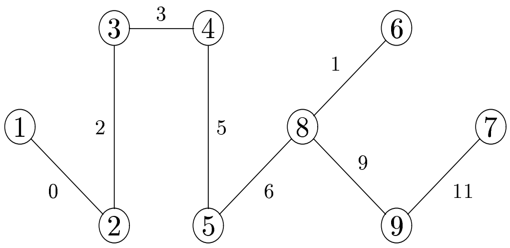


3. $\ell = 2$
4. $N := [6\ 6\ 6\ 6\ 8\ 4\ 1\ 5\ 5]$
7. $N := [8\ 8\ 8\ 8\ 8\ 5\ 5\ 5\ 5]$
   $S := [5\ 5\ 5\ 5\ 5\ 8\ 8\ 8\ 8]$



10. $S := [0\ 0\ 0\ 0\ 0\ 8\ 8\ 8\ 8]$
14. $T := [\{1,2\}\{2,3\}\{3,4\}\{4,5\}\{5,8\}\{6,8\}\{7,9\}\{8,9\}]$
16. $K := [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]$
18. Für $m = 1$ ergibt sich keine Veränderung mehr.

Endergebnis:



8.6.2 Reduzierung der Prozessorenzahl auf $\frac{n^2}{\log^2 n}$

Idee. Schritt 4. bis 15. müssen jeweils nur für die Repräsentanten der Komponenten ausgeführt werden, nicht für alle Knoten. Die Anzahl der Komponenten halbiert sich jedoch mindestens in jeder Phase. Bei Vorgabe von $\lceil \frac{n^2}{\log^2 n} \rceil$ Prozessoren wird der Algorithmus an zwei Stellen modifiziert:

1. Minimum in Schritt 4. und 7. wird mit nur $\lceil \frac{n}{\log^2 n} \rceil$ Prozessoren ausgeführt;
2. nach jeder Phase wird die Adjazenzmatrix des Graphen neu aufgestellt, d.h. für alle Knoten i und j werden die Kanten $\{i, j\}$ auf die Repräsentanten der Komponenten von i bzw. j übertragen.

Realisierung von 1. Binäroperation aus n Werten mit K Prozessoren.

Gegeben. n Werte $a_1, \dots, a_n$.

Problem. Die Summe (bzw. Produkt, Minimum, etc) aus $a_1, \dots, a_n$ soll mit K Prozessoren in Zeit $\mathcal{O}(t(n))$ bestimmt werden, wobei

$$t(n) \leq \begin{cases} \lceil n/K \rceil - 1 + \log K, & \text{falls } \lceil n/2 \rceil > K \\ \log n, & \text{sonst.} \end{cases}$$

Für $K \geq \lceil n/2 \rceil$ ist das, wie bereits bewiesen, in $\mathcal{O}(\log n)$ Zeit möglich.

Sei $K < \lceil n/2 \rceil$: Teile $a_1, \ldots, a_n$ in K Gruppen auf, wobei jede Gruppe $\lceil n/K \rceil$ bzw. die letzte Gruppe $n - (K-1) \cdot \lceil n/K \rceil$ Werte enthält. Jeder Gruppe wird ein Prozessor zugeteilt, der die Binärportion der Werte seiner Gruppe sequentiell in maximal $\lceil n/K \rceil - 1$ Schritten bestimmt. Die Binäroperation aus diesen K Ergebnissen wird dann in $\log K$ Zeit mit K Prozessoren parallel bestimmt.

Im Algorithmus ergibt sich dann eine Gesamtlaufzeit von

$$T(n) \in \mathcal{O}(\frac{n}{k} + \log n \cdot \log k)$$

für die Schritte 4. und 7. mit $K \leq \lceil n/2 \rceil$ Prozessoren: Da sich die Anzahl der zu behandelnden Repräsentanten in jeder Phase halbiert und nach $\ell = \log n - \lceil \log K \rceil$ höchstens $2K$ Repräsentanten übrig sind, gilt:

$$\begin{aligned}
T(n) &\leq \sum_{k=0}^{\ell-1} (\lceil \tfrac{n}{2^k \cdot K} \rceil - 1 + \log K) + \sum_{k=\ell}^{\log n - 1} (\log(\frac{n}{2^k}) \\
&\leq \frac{2n}{K} + \log n \cdot \log K - \log^2 K + \sum_{k=\ell}^{\log n - 1} \log n - k \\
&\in \mathcal{O}(\frac{n}{K} + \log n \cdot \log K).
\end{aligned}$$

Die letzte Ungleichung gilt, da $\sum_{k=0}^{\infty} \frac{1}{2^k} \leq 2$.

Realisierung von 2. Um die Adjazenzmatrix des Graphen nach jeder Phase geeignet neu aufzustellen, werden die Knoten der Komponenten entsprechend der Nummer ihrer Repräsentanten angeordnet und jeweils an den Anfang einer solchen Knotengruppe ihr Repräsentant geschrieben. Zum Ermitteln der Einträge für alle Repräsentantenpaare muss jeweils für jede Knotengruppe die ODER-Verbindung der entsprechenden Einträge berechnet werden.

Dafür werden jeder Komponente $K = \lceil \frac{n}{\log^2 n} \rceil$ Prozessoren zugeordnet, um die ODER-Verbindungen für alle Adjazenzen von Knoten der Komponenten zu bilden. Dies ist eine Binärverbindung einer partitionierten Menge und kann mit k Prozessoren in $\mathcal{O}(\lceil \frac{n}{k} \rceil - 1 + \log k)$ berechnet werden (siehe 8.4.4)

Analog wie bei 1. ist die Gesamtlaufzeit über alle Phasen für die Neuberechnungen der Adjazenzmatrix mit K Prozessoren in $\mathcal{O}(\frac{n}{K} + \log n \cdot \log K)$, also bei $K = \lceil \frac{n}{\log^2 n} \rceil$ in $\mathcal{O}(\log^2 n)$.

8.7 ParallelSelect: Parallelisierung von Select zur Bestimmung des k-ten Elements aus n Elementen

Das Verfahren ParallelSelect benutzt p Prozessoren (wobei $p \leq n$) auf einer CREW-PRAM. Sei S die Menge der gegebenen n Elemente. Falls $|S| = n < 5$ ist, wird das k-te Element durch Sortieren bestimmt. Ansonsten wird S in p Teilfolgen der Länge höchstens $\lceil \frac{n}{p} \rceil$ aufgeteilt, und jeder Prozessor bestimmt mittels Select in $\mathcal{O}(\lceil \frac{n}{p} \rceil)$ das mittlere Element seiner Teilfolge. Beachte, dass Select wieder „ganz normal" in Teilfolgen der Länge 5 unterteilt. Die mittleren Elemente bilden die Menge M, deren mittleres Element m (d.h. das $\lceil \frac{|M|}{2} \rceil$-kleinste) durch einen rekursiven Aufruf von ParallelSelect bestimmt wird. S wird dann wieder in Teilfolgen

$$\begin{aligned} &S_< = \{x \in S : x \leq m\} \text{ und} \\ &S_> = \{x \in S : x > m\} \end{aligned}$$

geteilt. (Voraussetzung sei wieder, dass alle Elemente verschieden sind.) Falls $|S_<| \geq k$ ist, wird dann ParallelSelect rekursiv auf $S_<$ mit k aufgerufen, ansonsten auf $S_>$ mit $k - |S_<|$. Da ParallelSelect kostenoptimal arbeiten soll, können $S_<$ und $S_>$ nicht durch sequentielles Durchlaufen der Menge S und Vergleichen mit m bestimmt werden. Stattdessen unterteilen die p Prozessoren jeweils parallel ihre Teilfolgen S^i $(1 \leq i \leq p)$ in

$$\begin{aligned} &S^i_< = \{x \in S^i : x \leq m\} \text{ und} \\ &S^i_> = \{x \in S^i : x > m\}. \end{aligned}$$

$S_<$ und $S_>$ werden dann aus diesen $S^i_<$ und $S^i_>$ zusammengesetzt, indem die Prozessoren gleichzeitig in verschiedene Positionen eines Arrays $S_<$ im gleichen Speicher ihre $S^i_<$ schreiben. Die Positionen werden zuvor mittels Prefix-Summe berechnet.

Formale Beschreibung von PARALLELSELECT$(S, k; p)$

Eingabe: S mit $|S| = n$, k, p.

Datenstruktur: Arrays $S_<$ und $S_>$ und M im globalen Speicher. Arrays S^i, $S^i_<$ und $S^i_>$ in den lokalen Speichern für $1 \leq i \leq p$.

```
Falls $n < 5$:
    Prozessor $p_1$ liest Folge $S$, sortiert $S$ und gibt das $k$-kleinste Element aus.
Falls $n \geq 5$ führe aus:
    Prozessor $p_1$ berechnet aus $n$ und $p$ die Zahl $\ell$ mit $p = n^{1-\ell}$.
    Für alle $i$ $(1 \leq i \leq p)$ führe parallel aus:
        Prozessor $p_i$ berechnet $(i-1) \cdot \lfloor n^\ell \rfloor$ und $i \cdot \lfloor n^\ell \rfloor - 1$ und kopiert die Elemente aus $S$, die zwischen Position $(i-1) \cdot \lfloor n^\ell \rfloor$ und Position $i \cdot \lfloor n^\ell \rfloor - 1$ stehen, nach $S^i$.
        Prozessor $p_i$ berechnet $n' := n^{1-\ell} \cdot \lfloor n^\ell \rfloor$ und kopiert das Element aus $S$, das an Position $n' + i$ steht, nach $S^i$, falls vorhanden.
        Prozessor $p_i$ führt SELECT$(S^i, \lceil \frac{1}{2} n^\ell \rceil)$ aus und schreibt das Ergebnis $m_i$ an die $i$-te Position von $M$.
    Prozessor $p_1$ berechnet $p' := \lceil |M|^{1-\ell} \rceil$ (beachte $|M| = p = n^{1-\ell}$).
    PARALLELSELECT$(M, \lceil \frac{|M|}{2} \rceil; p')$ Ergebnis sei $m$.
    Für alle $i$ $(1 \leq i \leq p)$ führe parallel aus:
        Prozessor $p_i$ berechnet
        $S^i_< = \{x \in S^i : x \leq m\}, |S^i_<| =: a_i$
        $S^i_> = \{x \in S^i : x > m\}, |S^i_>| =: b_i$.
        PRÄFIX-SUMME$(a_1, \ldots, a_p)$
        PRÄFIX-SUMME$(b_1, \ldots, b_p)$
        Die Ergebnisse seien $A_1, \ldots, A_p$ und $B_1, \ldots, B_p$.
        Setze $A_0 = B_0 = 0$.
    Für alle $i$ $(1 \leq i \leq p)$ führe parallel aus:
        Prozessor $p_i$ schreibt $S^i_<$ an die Positionen $(A_{i-1} + 1)$ bis $A_i$ von $S_<$ und $S^i_>$ an die Positionen $(B_{i-1} + 1)$ bis $B_i$ von $S_>$.
    Prozessor $p_1$ berechnet aus $A_p = |S_<|$ die Zahl $s_1 := \lceil A_p^{1-\ell} \rceil$ und aus $B_p = |S_>|$ die Zahl $s_2 := \lceil B_p^{1-\ell} \rceil$.
    Falls $|S_<| \geq k$ führe aus:
        PARALLELSELECT$(S_<; k; s_1)$.
    Ansonsten:
        Prozessor $p_1$ berechnet $k' := k - |S_<|$.
        PARALLELSELECT$(S_>; k'; s_2)$.
```

Laufzeit. Die Laufzeit von PARALLELSELECT ist in $\mathcal{O}(n^\ell)$, wobei $p = n^{1-\ell}$ die Anzahl der Prozessoren ist.

Schritt 1:	$\mathcal{O}(1)$	
Schritt 4:	$\mathcal{O}(1)$	
Schritt 6:	$\mathcal{O}(n^\ell)$	
Schritt 7:	$\mathcal{O}(1)$	
Schritt 8:	$\mathcal{O}(n^\ell)$	
Schritt 9:	$\mathcal{O}(1)$	
Schritt 10:	$t(n^{1-\ell})$,	wobei $t(n)$ Laufzeit von ParallelSelect(n) ist.
Schritt 12:	$\mathcal{O}(n^\ell)$	
Schritt 13:	$\mathcal{O}(\log n^{1-\ell})$	
Schritt 16:	$\mathcal{O}(n^\ell)$	
Schritt 17:	$\mathcal{O}(1)$	
Schritt 19:	$\leq t(\frac{3}{4}n + \frac{n^{1-\ell}}{4})$	
Schritt 21–22:	$\leq t(\frac{3}{4}n + \frac{n^{1-\ell}}{4})$	

Die letzten beiden gelten wegen

$$S_< \leq n - \lceil \tfrac{n^{1-\ell}}{2} \rceil \cdot \lceil \tfrac{\lceil n^\ell \rceil + 1}{2} \rceil \leq \frac{3}{4}n + \frac{n^{1-\ell}}{4}.$$

Denn wenn m das $\lceil \frac{|M|}{2} \rceil$-kleinste Element von M ist, so gibt es $\lceil \frac{|M|}{2} \rceil$ Elemente in M, die größer als m sind, und für jedes dieser Elemente existieren mindestens $\lceil \frac{\lceil n^\ell \rceil - 1}{2} \rceil$ Elemente in S, die größer sind. Damit ergibt sich für geeignete Konstanten c_1, c_2:

$$t(n) \leq c_1 \cdot n^\ell + t(n^{1-\ell}) + t(\frac{3}{4}n + \frac{n^{1-\ell}}{4}) + c_2 \cdot \log n^{1-\ell}.$$

Wähle Konstante c_3 so, dass $c_2 \cdot \log n^{1-l} \leq c_3 \cdot n^\ell$ ist, und setze $c_4 := c_3 + c_1$; dann ergibt sich per Induktion über n:

$$\begin{array}{rcl} t(n) & \leq & c_4 \cdot n^\ell + c \cdot (n^{1-\ell})^\ell + c \cdot (\frac{3}{4}n + \frac{n^{1-\ell}}{4})^\ell \\ & \leq & c_4 \cdot n^\ell + c \cdot \frac{1}{n^{\ell^2}} \cdot n^\ell + c \cdot (\frac{3}{4}n)^\ell + c \cdot (\frac{n^{1-\ell}}{4})^\ell, \quad \text{da}\ell < 1 \\ & \leq & n^\ell(\frac{c_4}{c} + \frac{1}{n^{\ell^2}} + (\frac{3}{4})^\ell + \frac{1}{4^\ell \cdot n^{\ell^2}}) \\ & \leq & n^\ell(\frac{c_4}{c} + \frac{2}{n^{\ell^2}} + (\frac{3}{4})^\ell). \end{array}$$

Für festes ℓ ($0 \leq \ell \leq 1$) ist $(\frac{3}{4})^\ell = 1 - \varepsilon$, wobei $\varepsilon > 0$ ist. Das bedeutet, für geeignetes c und genügend großes n ist

$$\frac{c_4}{c} + \frac{2}{n^{\ell^2}} + (\frac{3}{4})^\ell \leq 1$$

und damit $t(n) \leq c \cdot n^\ell$.

Beispiel. Sei $n = 17$, $p = 4$, $k = 7$. **Gesucht:** das 7.-kleinste Element.

Vorgehen des Algorithmus.

$S := [24\ \ 3\ \ 12\ \ 21\ \ 7\ \ 6\ \ 13\ \ 20\ \ 23\ \ 14\ \ 1\ \ 4\ \ 15\ \ 16\ \ 2\ \ 22\ \ 19]$

7.

$$\begin{aligned} S^1 &= [24\ \ 3\ \ 12\ \ 21\ \ 19] \\ S^2 &= [7\ \ 6\ \ 13\ \ 20] \\ S^3 &= [23\ \ 14\ \ 1\ \ 4] \\ S^4 &= [15\ \ 16\ \ 2\ \ 22] \end{aligned}$$

8. $M = [19\ \ 13\ \ 14\ \ 16]$

9. $p' = 2$

10. $m = 14$

11.

$$\begin{array}{ll} S^1_< = [3\ \ 12] & S^1_> = [24\ \ 21\ \ 19] \\ S^2_< = [7\ \ 6\ \ 13] & S^2_> = [20] \\ S^3_< = [14\ \ 1\ \ 4] & S^3_> = [23] \\ S^4_< = [2] & S^4_> = [15\ \ 16\ \ 22] \end{array}$$

16.

$$\begin{aligned} S_< &= [3\ \ 12\ \ 7\ \ 6\ \ 13\ \ 14\ \ 1\ \ 4\ \ 2] \\ S_> &= [24\ \ 21\ \ 19\ \ 20\ \ 23\ \ 15\ \ 16\ \ 22] \end{aligned}$$

17. $|S_<| = 9$, $|S_>| = 8$, $s_1 = 3$, $(s_2 = 3)$

Aufruf von $\textsc{ParallelSelect}(S_<; 7; 3)$

7.

$$\begin{aligned} S^{1,1} &= [3\ \ 12\ \ 7] \\ S^{1,2} &= [6\ \ 13\ \ 14] \\ S^{1,3} &= [1\ \ 4\ \ 2] \end{aligned}$$

8. M^1=[7 13 2]

10. $m = 7$

11.

$$\begin{array}{ll} S_<^{11} = [3\ \ 7] & S_>^{11} = [12] \\ S_<^{12} = [6] & S_>^{12} = [13\ \ 14] \\ S_<^{13} = [1\ \ 4\ \ 2] & \end{array}$$

16.

$$\begin{aligned} S_<^1 &= [3\ \ 7\ \ 6\ \ 1\ \ 4\ \ 2] \\ S_>^1 &= [12\ \ 13\ \ 14] \end{aligned}$$

20. $|S_<^1| = 6 < 7$, $k' = 7 - 6 = 1$

ParallelSelect mit $S_<^1 = [12\ \ 13\ \ 14]$ und $k = 1$ liefert sofort das Ergebnis:

Das gesuchte 7.-kleinste Element ist die 12.

8.8 Ein paralleler Algorithmus für das Scheduling-Problem für Jobs mit festen Start- und Endzeiten

Gegeben. n Jobs J_i mit Startzeiten s_i und Endzeiten t_i $(1 \leq i \leq n)$, o.B.d.A. $s_i \neq s_j$, $t_i \neq t_j$ für $i \neq j$.

Gesucht. Ein optimaler Schedule, d.h. eine Zuordnung der J_i auf einer minimalen Anzahl an Maschinen, so dass sich keine zwei Jobs auf derselben Maschine überlappen.

Sequentiell kann das Problem in $\mathcal{O}(n \log n)$ Zeit wie folgt gelöst werden:

```
1. Sortiere die $s_1$, $t_1$, $s_2$, $t_2, \ldots, s_n$, $t_n$ in nichtabsteigender Reihenfolge,
   wobei $t_j$ vor $s_k$ kommt, falls $t_j = s_k$.
2. Schreibe sortierte Folge in Array $U$.
3. Setze $\ell := 1$ und $S[i] := i$ für $1 \leq i \leq n$.
4. Für $k = 1$ bis $2n$ führe aus:
5.     Falls $U[k] = s_j$,
6.         setze $M[j] := S[\ell]$, $S[\ell] := 0$ und $\ell := \ell + 1$;
7.     ansonsten, falls $U[k] = t_j$
8.         setze $S[\ell - 1] := M[j]$ und $\ell := \ell - 1$.
```

Dabei realisiert S einen Stack, auf dem die freien Maschinen liegen; $M[j]$ gibt an, welcher Maschine Job J_j zugeordnet wird.

Die parallele Version des Algorithmus besteht aus drei Phasen.

1. Phase. Die s_i, t_i werden mittels eines parallelen Sortieralgorithmus sortiert und in ein Array U der Länge $2n$ geschrieben. Für alle Jobs J_i wird parallel die Anzahl a_i der Maschinen berechnet, die unmittelbar nach der Startzeit von J_i belegt sind sowie die Anzahl b_i der Maschinen, die unmittelbar vor Beendigung von J_i belegt sind. Dazu wird ein Array L der Länge $2n$ angelegt mit

$$L[k] = \begin{cases} 1, & \text{falls } U[k] \text{ Startzeit} \\ -1, & \text{falls } U[k] \text{ Endzeit.} \end{cases}$$

Wenn nun $\sum_{j=1}^{k} L[j] = p_k$ für $1 \leq k \leq 2n$, dann ist

$$\begin{aligned} a_i &= p_k, && \text{falls } U[k] \text{ Startzeit } s_i \text{ und} \\ b_i &= p_k + 1, && \text{falls } U[k] \text{ Endzeit } t_i. \end{aligned}$$

2. Phase. Für alle parallelen Jobs J_i wird der unmittelbare Vorgänger auf derselben Maschine berechnet (falls er existiert). Dies ist gerade der Job J_ℓ, der als letzter Job vor bzw. zum Zeitpunkt s_i geendet hat und für den $a_i = b_\ell$ ist.

3. Phase. Durch List Ranking wird der erste Vorgänger eines jeden Jobs J_i auf derselben Maschine berechnet und den Jobs die entsprechende Maschinennummer zugeordnet.

Zum Schluss enthält $M[i]$ die Nummer der Maschine, der der Job J_i zugeordnet wird. Die Schritte 3. bis 20. entsprechen der ersten Phase, die Schritte 21. bis 26. der zweiten Phase und 27. bis 31. der dritten Phase. Der Algorithmus benötigt $\mathcal{O}(\log n)$ Zeit bei $\frac{n^2}{\log n}$ Prozessoren, ist also nicht kostenoptimal.

Formale Beschreibung des Algorithmus PARALLEL SCHEDULING **(Dekel & Sahni 1983)**

Eingabe: n Jobs J_i, Startzeiten s_i, Endzeiten t_i $(1 \leq i \leq n)$.

Ausgabe: $M[i]$ enthält Nummer der Maschine, der Job J_i zugeordnet wird.

Datenstruktur: Arrays U und L der Länge $2n$.

```
PARALLEL SCHEDULING
Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
    Setze $U[i] := s_i$ und $U[n+i] := t_i$.
Für alle $i$, $j$ $(1 \leq i, j \leq 2n)$ führe parallel aus:
    Falls $U[i] < U[j]$, oder $U[i] = U[j]$ und $U[i]$ Endzeit, $U[j]$ Startzeit:
        Setze $r_{ij} := 1$,
    ansonsten
        setze $r_{ij} := 0$.
Für alle $j$ $(1 \leq j \leq 2n)$ führe parallel aus:
    Berechne $\Pi(j) := \sum_{i=1}^{2n} r_{ij}$,
    setze $U[\Pi(j)+1] := U[j]$.
Für alle $k$ $(1 \leq k \leq 2n)$ führe parallel aus:
    Falls $U[k]$ Startzeit:
        Setze $L[k] := 1$,
    ansonsten
        setze $L[k] := -1$.
    Berechne $p_k = \sum_{\ell=1}^{k} L[\ell]$.
    Falls $U[k] = s_j$:
        Setze $a_j := p_k$,
    ansonsten, falls $U[k] = t_j$
        setze $b_j := p_k + 1$.
Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
    Finde $k$ mit $t_k = \max\{t_\ell : t_\ell \leq s_i, b_\ell = a_i\}$.
    Falls $k$ existiert:
        Setze $\textsc{Vor}[i] := k$,
    ansonsten
        setze $\textsc{Vor}[i] := i$.
Für alle $\ell = 1$ bis $\lceil \log n \rceil$ führe aus:
    Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
        Setze $\textsc{Vor}[i] := \textsc{Vor}[\textsc{Vor}[i]]$.
Für alle $i$ $(1 \leq i \leq n)$ führe parallel aus:
    Setze $M[i] := a_{\textsc{Vor}[i]}$.
```

Beispiel.

i	1	2	3	4	5	6	7	8
s_i	5	0	1	2	4	3	7	6
t_i	8	4	5	7	9	6	10	12

Vorgehen des Algorithmus.

2. $U = [5\ \ 0\ \ 1\ \ 2\ \ 4\ \ 3\ \ 7\ \ 6\ \ 8\ \ 4\ \ 5\ \ 7\ \ 9\ \ 6\ \ 10\ \ 12]$

5.

$$
r_{ij}:\quad
\begin{array}{r|cccccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
3 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
4 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
5 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
6 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
10 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
11 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
14 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

9. $\Pi = [7\ \ 0\ \ 1\ \ 2\ \ 5\ \ 3\ \ 11\ \ 9\ \ 12\ \ 4\ \ 6\ \ 10\ \ 13\ \ 8\ \ 14\ \ 15]$

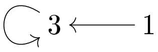
7. $U = [0\ \ 1\ \ 2\ \ 3\ \ 4\ \ 4\ \ 5\ \ 5\ \ 6\ \ 6\ \ 7\ \ 7\ \ 8\ \ 9\ \ 10\ \ 12]$

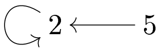
9. $L = [1\ \ 1\ \ 1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ -1\ \ -1\ \ -1\ \ -1]$

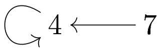
11. $\begin{array}{ll} a_j\colon & 4\ \ 1\ \ 2\ \ 3\ \ 4\ \ 4\ \ 4\ \ 4 \\ b_j\colon & 4\ \ 4\ \ 4\ \ 4\ \ 3\ \ 4\ \ 2\ \ 1 \end{array}$

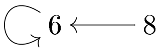
14. $\textsc{Vor} = [3\ \ 2\ \ 3\ \ 4\ \ 2\ \ 6\ \ 4\ \ 6]$

15. $\ell = 1$.

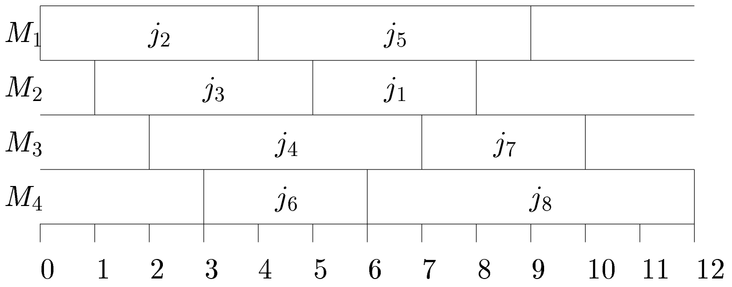








19.



Literaturverzeichnis

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw–Hill, 1990. ISBN 0-262-03141-8, 0-07-013143-0. Unibib KN: kid 112/c67, lbs 830/c67.

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. *BIT*, 1:269–271, 1959.

[FIN93] András Frank, Toshihide Ibaraki, and Hiroshi Nagamochi. On sparse subgraphs preserving connectivity properties. *J. of Graph Theory*, 17:275–281, 1993.

[GR90] Alan Gibbons and Wojciech Rytter. *Efficient parallel algorithms.* Cambridge University Press, 1990. Unibib KN: kid 112/g41a, kid 112/g41.

[Jun94] Dieter Jungnickel. *Graphen, Netzwerke und Algorithmen.* BI-Wissenschaftsverlag, 1994. Unibib KN: kid 114/j96a(3), kid 604/j96, lbs 840/j96.

[Kru56] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[NI92] Hiroshi Nagamochi and Toshihide Ibaraki. A Linear-Time Algorithm for Finding a Sparse k-Connected Spanning Subgraph of a k-Connected Graph. *Algorithmica*, 7:583–596, 1992.

[OW93] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen.* B.I.–Wissenschaftsverlag, 1993. Unibib KN: kid 112/o98(2), lbs 830/o99.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Tech. J.*, 36:1389–1401, 1957.

[SW94] Mechtild Stoer and Frank Wagner. A Simple Min Cut Algorithm. In Jan v. Leeuwen, editor, *Second European Symposium on Algorithms, ESA'94*, pages 141–147. Springer-Verlag, Lecture Notes in Computer Science, vol. 855, 1994. Unibib KN: kid 100:i/l22-855.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms.* SIAM, 1983. ISBN 0-89871-187-8. Unibib KN: kid 112/t17.