Effiziente Algorithmen
Vorlesung im WS 1997/98

Christian Schindelhauer

23. Februar 1998

Inhaltsverzeichnis

Kapitel 1

Algorithmen und Effizienz

Wir untersuchen die Effzienz eines Algorithmus auf der Grundlage des theoretischen Maschinenmodel der *random access machine* (`RAM`) mit beschränkter Wortlänge und dem Einheitskostenmodel.

1.1 Das RAM-Modell

Definition 1 *Eine **RAM** besteht aus einem Prozessor und wahlfrei adressierbaren **Speicherplätzen** $M_0, \ldots, M_{s-1}$ und internen variablen **Registern** $\mathrm{IR} = \{A, B, F, V, W, X, Y, Z\}$. Jedes dieser Speicher kann ganzzahlige Werte aus $\{0, \ldots, 2^{w-1}\}$ speichern. Man bezeichnet w als die **Wortlänge** der RAM.*

Zu Beginn der Berechnung befindet sich die Eingabe $x \in \{0,1\}^n$ verteilt in den Speicherplätzen $M_1, \ldots, M_n$. Speicherzelle M_0 enthält die Länge der Eingabe n. Das Ergebnis befindet sich in den Speicherzellen.

Der Prozessor besitzt:

- die internen Register $\mathrm{IR} = \{A, B, F, V, W, X, Y, Z\}$ haben folgende Bedeutung:

Register	Bezeichnung	Auswirkung
A	Adressregister	dient zur Adressierung einer Speicherzelle.
B	Befehlszeilenregister	gibt die abzuarbeitende Befehlszeile an.
F	*flag*/Bedingungs-Register	speichert letzten Vergleich.
V, W, X, Y, Z	Operationsregister	Hier finden arithmetische Operationen statt.

- das Programm, welches aus folgenden Befehlen bestehen kann für $U \in \mathrm{IR}$, $U' \in \{\texttt{pid}\} \cup \mathrm{IR}$, $c \in \{0, \ldots, p\}$, $\sigma \in \{<, \leq, =, >, \geq, \neq\}$ und $b \in \mathbb{N}$:

Befehl	Wirkung auf Register	Befehlszeilenregister
`READ U`	$U \leftarrow M_A$	$B \leftarrow B+1$
`WRITE U`	$M_a \leftarrow U$	$B \leftarrow B+1$
`COPY U' U`	$U \leftarrow U'$	$B \leftarrow B+1$
`LOAD U c`	$U \leftarrow c$	$B \leftarrow B+1$
`COMP` σ	$F \leftarrow \begin{cases} 1\,, & \text{falls } X\sigma Y \\ 0\,, & \text{sonst} \end{cases}$	$B \leftarrow B+1$
`JUMP b`		$B \leftarrow \begin{cases} b\,, & \text{falls } F=1 \\ B+1\,, & \text{sonst} \end{cases}$
`NOP`		$B \leftarrow B+1$
`STOP`	Maschine hält.	

- Zusätzlich gibt es noch folgende arithmetische Operationen. Grundsätzlich wird bei jeder arithmetischen Operation das Befehlszeilenregister B um eins erhöht. Sei w die Wortlänge der RAM.

Befehl	Auswirkung	
`ADD`	$X \leftarrow \begin{cases} Y+Z\,, & \text{falls } Y+Z \leq 2^w \\ X+Y-2^w\,, & \text{sonst} \end{cases}$	$W \leftarrow \begin{cases} 0\,, & \text{falls } Y+Z \leq 2^w \\ 1\,, & \text{sonst} \end{cases}$
`SUB`	$X \leftarrow \begin{cases} Y-Z\,, & \text{falls } Y \geq Z \\ 2^w-Y+Z\,, & \text{sonst} \end{cases}$	$W \leftarrow \begin{cases} 0\,, & \text{falls } Y \geq Z \\ 1\,, & \text{sonst} \end{cases}$
`MULT`	$X \leftarrow Y \cdot Z \bmod 2^w$	$W \leftarrow \lfloor Y \cdot Z/2^w \rfloor$
`DIV`	$X \leftarrow \begin{cases} Y \bmod Z\,, & \text{falls } Z \neq 0 \\ 0\,, & \text{sonst} \end{cases}$	$W \leftarrow \begin{cases} \lfloor Y/Z \rfloor\,, & \text{falls } Z \neq 0 \\ 0\,, & \text{sonst} \end{cases}$

Jeder Auswertung einer Befehlszeile kostet eine Recheneinheit. Die höchste Adresse eines gelesenen oder beschriebenen Speicherzelle M_s ergibt den Speicherplatzverbrauch. Dieses Zeit- und Speicherverbrauchsmessung wird **uniformes Kostenmaß** genannt.
Es findet auch Verwendung beim Modell einer RAM mit unbeschränkter Wortlänge. Damit aber diese Ressource nicht über die Maßen eingesetzt werden kann, betrachtet man hierbei auch das **logarithmische Kostenmaß**. Jede Schritt einer Berechnung wird hierbei mit dem binären Logarithmus der größten dabei vorkommenden Speicherwerts zeitlich verrechnet. Der verwendete Platz entspricht nun dem höchsten Aufwand während der gesamte Berechnung den Speicherinhalt als Binärstring darzustellen.
Zeichen werden in einer RAM als Zahlen abgespeichert, während Zeichenketten x als durchgehende Folgen von Speicherzellen $M_k \dots M_{k+|x|-1}$ mit den Buchstaben $x[1] \dots x[n]$ dargestellt werden.

1.2 Zeit und Platz

Dieses Berechnungsmodell modelliert die Prozessorebene eines Rechner. Natürlich eignet es sich nicht zur anschaulichen Darstellung von Algorithmen. Eine Modellierung ist aber notwendig, um die Ressourcen **Zeit** und **Platz** zu fundieren.

Definition 1 *Ein Problem P kann in **Zeit** $T(n)$ und mit **Speicherplatz** $S(n)$ gelöst werden, wenn es eine RAM gibt, die auf jeder Eingabe der (Nenn-)Größe n mit $S(n)$ Speicherzellen in $T(n)$ Schritten die, dem Problem entsprechende, Lösung berechnet.*

+

Für einen Algorithmus, den wir in der Regel nur abstrakt beschreiben werden als prozedural orientiertes Computer-Esperanto, wird in Gedanken eine RAM konstruiert, um dann das Laufzeitverhalten zu bestimmen.
Wir werden sehen, daß dieser Abstraktionsschritt in der Regel nicht schwer fällt. Schließlich werden wir die asymptotische Darstellung von Laufzeitverhalten verwenden (O-Notation). Im allgemeinen gelten in der Regel die folgenden Faustregeln:

> Jeder Programmschrit in einem Algorithmus bedeutet konstante Zeit.
>
> Der Speicherverbrauch eines Pseudo-Pascal-Programms entspricht dem der zugehörigen RAM.

1.3 Notation der Zeichenketten

Wenn wir algorithmische Probleme auf Zeichenkette (Strings) betrachten, gehen wir prinzipiell von einem **endlichen Alphabet** Σ (Menge von Buchstaben) aus. Als **binäres Alphabet** wird $\{0,1\}$ bezeichnet.
Eine **Zeichenkette** x (synomym Wort, Text oder String) ist ein endliches Tupel von Buchstaben $x[1], x[2], \ldots, x[n] \in \Sigma$. Hierbei ist $|x| = n$ die Länge der Zeichenkette. Das **leere Wort** $\boldsymbol{\lambda}$ ist eine Zeichenkette ohne einen einzigen Buchstaben (Man beachte: λ ist kein Buchstabe). Σ^* bezeichnet die Menge aller Zeichenketten über dem Alphabet Σ. Σ^n ist die Menge aller Zeichenketten der Länge n.
Als elementare Operationen auf Zeichenketten betrachten wir das Produkt $x\ y$, die das Aneinanderhängen zweier Zeichenketten bezeichnet. z ist ein **Präfix** einer Zeichenkette x, falls es eine Zeichenkette y existiert, sodaß $z\ y = x$. Wir schreiben dann

$$\boldsymbol{z} \sqsubset \boldsymbol{x}\ .$$

Analog definieren wir das Prädikat **Suffix**:

$$\boldsymbol{y} \sqsupset \boldsymbol{x} \qquad \Longleftrightarrow \qquad \exists z \in \Sigma^* \quad zy = x\ .$$

Mit $x[i,j]$ bezeichnen wir das Teilwort $x[i] \ldots x[j]$ aus x.

Kapitel 2

Stringsuche in einem Text

Stringsuche in einem Text ***string searching***
Geg.: Ein Suchstring x der Länge $|x| = m$ und ein Text t der Länge n.
Ges.: Falls vorhanden, die erste Position im Text t in der x vorkommt,
d.h. das kleinste i mit $t[i, i+m-1] = x[1, m]$.

Wenn wir Zeichenketten algorithmisch auf Identität vergleichen, legen wir folgenden Algorithmus zugrunde:

```
x = y                                          \* Infixoperator für Zeichenketten *\
input: x, y ∈ Σ^n
{ i ← 1
  while i ≤ n ∧ x[i] = y[i] do i ← i + 1
  return i = n + 1
}
```

Der folgende Algorithmus löst das Problem der Stringsuche:

```
Brute-Force-Search
input: x, t ∈ Σ*
{ m ← |x|
  n ← |t|
  i ← 1
  while i ≤ n − m ∧ ¬(x = t[i, i + m − 1]) do i ← i + 1
  return i
}
```

Beispiel 2.0.1: Der **Brute-Force-**Algorithmus führt auf diesen Beispiel folgende Buchstabenvergleiche aus.

```
x = a b a b a c a
t = a b a b a b a c c a b a b a c a
    a b a b a c
```

```
a
  a b a b a c a
    a
      a b a b
        a
          a b
            a
              a
                a b a b a c a
```

Fakt 1 *Der* ***Brute-Force-Search****-Algorithmus hat eine Laufzeit von* $O(n \cdot m)$.

Aufgabe 2.0.1: 3
Beweisen Sie die Richtigkeit dieses Fakts.

Dieses Ergebnis bezieht sich auf die ***worst-case***-Laufzeit. Das heißt: Wenn man alle Wörter x und t der Länge m und n betrachtet, dann erfüllt der Algorithmus für jedes dieser Wörter diese Eigenschaft[1].

Aufgabe 2.0.2: 6
Geben sie ein Beispiel für Eingaben x und t der Länge m und n an, für die der Algorithmus **Brute-Force-Search** tatsächlich die *worst-case*-Laufzeit benötigt.

Dieser Algorithmus erfreut sich allgemeiner Beliebtheit. Diese Problem läßt aber weitaus effizienter lösen.

2.1 Stringsuche mit endlichen Automaten

Die Leitidee ist jeden Buchstaben des Texts t nur einmal zu verarbeiten. Man kann die Information, die erfolgreiche frühere Vergleiche ergaben, benutzen, um das Pattern an eine neue Stelle zu verschieben.

Finite-Automaton-Matcher
input: $x, t \in \Sigma^*$
{ **var** *shift*: **function** $\{1 \ldots m\} \times \Sigma \to \mathbb{N}$
  $m \leftarrow |x|$
  $n \leftarrow |t|$
  shift $\leftarrow$ **Init-Shift**(x)
  $i \leftarrow 1$
  $j \leftarrow 1$
  while $i \leq n \ \wedge \ j \leq m$ **do**
    { $j \leftarrow$ *shift*$(j, t[i])$
      $i \leftarrow i + 1$
    }
  return $i - m$
}

[1] Tatsächlich ist dieser Algorithmus im Erwartungswert wesentlich schneller: Wählt man das Pattern x zufällig, so hat dieser Algorithmus eine erwartete Laufzeit von $O(n)$. Beweis zur Übung.

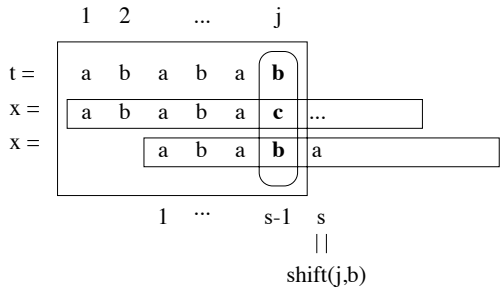

Abbildung 2.1: j ist die Anzahl der gerade bearbeiteten Zeichen, $j-1$ davon waren korrekt. j und das Zeichen c definieren die Schiebeoperation. $\mathit{shift}(j,b)$ bezeichnet die Position, des als nächstes zu untersuchenden Zeichens in x.

Beispiel 2.1.2:

```
t = a b a b a b a c c a b a b a c a
    | | | | | #
x = a b a b a c a
          ->
            | | #
        a b a b a c a
                  -> # | | | | | | |
                     a b a b a c a
```

Die Information, die man zur Berechnung der Verschiebung des Suchstrings benötigt, ist endlich (für ein festes x). Man kann sie durch einen endlichen deterministischen Automaten (DFA) mit $m+1$ Zuständen modellieren (siehe Abbildung 2.2).

Abbildung 2.1 zeigt einen typischen Verlauf der Stringsuche. Hierbei bezeichnet j die Länge des betrachteten Präfix ($j-1$ Zeichen stimmten bereits überein). Nun berechnet sich also die nächste zu betrachtende Position des Suchtextes x als:

$$\mathit{shift}(j,c) \;=\; \begin{cases} j+1\,, & \text{falls } x[j]=c\,, \\ 1+\max\{l \le j \mid x[1,l] \sqsupset x[1,j-1]c\}\,, & \text{sonst}\,. \end{cases}$$

Man erkennt, daß der erste Fall nur ein Sonderfall des zweiten ist. Es gilt also

$$\mathit{shift}(j,c) \;=\; 1+\max\{l \le j \mid x[1,l] \sqsupset x[1,j-1]c\}\,.$$

shift gibt also die um eins vergrößerte Länge l des größten Präfix von x, der eine Endung von $x[1,j-1]c$ darstellt.

Folgender Algorithmus berechnet die Übergangsfunktion eines Suchautomaten. Die Nummer des Folgezustands gibt die neue Position im Suchwort x an.

+

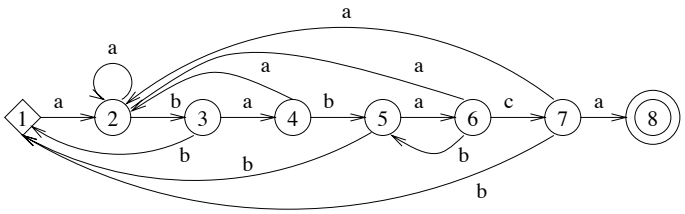

Abbildung 2.2: Dieser DFA sucht nach dem Wort *ababaca*. Alle nicht angegebenen Pfeile mit Beschriftung c zeigen auf Zustand 1.

```
Init-Shift
input: x ∈ Σ*
{ var shift:   array {1...m} × Σ of integer
      suffix:  array {0...m} of integer

  m ← |x|
  suffix[1] ← 1
  for all c ∈ Σ
        shift[1, c] ← 1
  shift[1, x[1]] ← 2

  for j ← 2 to m
     { suffix[j] ← shift[suffix[j − 1], x[j]]
       for all c ∈ Σ
             { if x[j] = c then shift[j, c] ← j + 1
                           else shift[j, c] ← shift[suffix[j − 1], c]
             }
  return shift
}
```

Hierbei berechnet

$$suffix(j) \;=\; 1 + \max\{i < j \mid x[1,i] \;\sqsupset\; x[1,j]\}\ .$$

Diese Funktion ermöglicht die effiziente Aufstellung der Übergangsfunktion des Automaten. Denn es gilt:

Lemma 1

$$\forall i \geq 2 \qquad suffix(j) \;=\; shift(suffix(j-1), x[j])\ , \tag{2.1}$$

$$\forall c \neq x[j] \qquad shift(j, c) \;=\; shift(suffix(j-1), c)\ . \tag{2.2}$$

Aufgabe 2.1.3: 7
Beweisen Sie die Richtigkeit dieses Lemmas.

Lösung:
Sei $s = suffix(j-1) = 1 + \max\{i < j-1 \mid x[1,i] \;\sqsupset\; x[1,j-1]\}$. Damit ist s maximal (aber nicht trivial) gewählt, so daß gilt

$$x[1, s-1] \;\sqsupset\; x[1, j-1]\ .$$

+

- Gleichung (2.1): Einsetzen der Definitionen in die rechte Seite

$$\begin{aligned} \mathit{shift}(\mathit{suffix}(j-1), x[j]) &= 1 + \max\{l \le s \mid x[1,l] \sqsupset x[1,s-1]x[j]\} \\ &= 1 + \max\{l \le s \mid x[1,l] \sqsupset x[1,s-1]x[j] \sqsupset x[1,j]\} \\ &= 1 + \max\{l < j \mid x[1,l] \sqsupset x[1,s-1]x[j] \sqsupset x[1,j]\} \\ &= 1 + \max\{l < j \mid x[1,l] \sqsupset x[1,j]\} \\ &= \mathit{suffix}(j) \end{aligned}$$

- Gleichung (2.2): Sei $c \neq x[j]$

$$\begin{aligned} \mathit{shift}(\mathit{suffix}(j-1), c) &= 1 + \max\{l \le s \mid x[1,l] \sqsupset x[1,s-1]c\} \\ &= 1 + \max\{l \le s \mid x[1,l] \sqsupset x[1,s-1]c \sqsupset x[1,j-1]c\} \\ &= 1 + \max\{l < j \mid x[1,l] \sqsupset x[1,j-1]c\} \\ &= 1 + \max\{l \le j \mid x[1,l] \sqsupset x[1,j-1]c\} \end{aligned}$$

  Die letzte Zeile folgt aus $c \neq x[j]$.

Aus diesem Lemma folgt sofort die Richtigkeit des Initialisierungalgorithmus.

Beispiel 2.1.3:
Init-Shift berechnet auf Eingabe *ababaca* folgendes:

$x[i]$	a	b	a	b	a	c	a
i	1	2	3	4	5	6	7
$\mathit{shift}[i,a]$	2	2	4	2	6	2	8
$\mathit{shift}[i,b]$	1	3	1	5	1	5	1
$\mathit{shift}[i,c]$	1	1	1	1	1	7	1
$\mathit{suffix}[i]$	1	1	2	3	4	1	1

Den zugehörigen Automat zeigt Abbildung 2.2.

Aufgabe 2.1.4: 2
Vollziehen Sie die Berechnung von Hand nach, die **init-shift** auf Eingabe *otontototont* vornimmt.

Theorem 1 *Stringsuche mit endlichen Automaten benötigt eine Laufzeit von* $O(n+m\cdot|\Sigma|)$ *und neben dem Speicherplatz für die Eingabe den Platz* $m\cdot|\Sigma|+O(1)$.

Damit ist dieser Algorithmus für große Alphabete (wie z.B. ASCII) kaum brauchbar, da das Anfertigen der Suchtabelle zu viel Zeit und Platz benötigt.

2.2 Der Algorithmus von Knuth, Morris und Pratt

Folgender Algorithmus arbeitet nach dem selben Grundprinzip. Es gelingt hier aber die Kodierung des endlichen Automaten geschickt in der Tabelle *next* abzuspeichern:

$$\mathit{next}(i) = \max\{j < i \mid x[1,j] \sqsupset x[1,i]\} \qquad (= \mathit{suffix}(j) - 1) \ .$$

+

```
KMP-Matcher
input: x, t ∈ Σ*
{ m ← |x|
  n ← |t|
  next ← Compute-Next(x)
  i ← 1
  j ← 0
  while i ≤ n ∧ j < m do
        { while j > 0 ∧ x[j+1] ≠ t[i] do j ← next[j]      ≙ Shift
          if x[j+1] = c then j ← j+1
          i ← i+1
        }
  return i − m
}
```

Die Berechnung der Tabelle *next* übernimmt der folgende Algorithmus:

```
Compute-Next
input: x ∈ Σ*
{ var next:     array {1...m} of integer
  m ← |x|
  next[1] ← 0
  k ← 0
  for i ← 2 to m
      { while k > 0 ∧ x[k+1] ≠ x[i] do k ← next[k]
        if x[k+1] = x[i] then k ← k+1
        next[q] ← k
      }
  return next
}
```

Nun gilt folgendes Lemma ($next^e(j)$ steht hier für die e-fach iterierte Anwendung von *next*: $next^0(i) = i$; $next^{e+1}(i) = next(next^e(i))$):

Lemma 2 *Sei $c \neq x[j]$, dann gilt*

$$\exists e \quad shift(j, z) \;=\; next^e(j-1)+1\ .$$

Aufgabe 2.2.5: 7
Beweisen Sie die Korrektheit des Lemmas.

Aus diesem Lemma folgt die Richtigkeit des KMP-Algorithmus direkt aus der Korrektheit von **Finite-Automaton-Matcher.**
Zur Laufzeit:

- Bei der Berechnung von **Compute-Next** gilt:
  Die Laufzeit ist proportional zu m plus der Anzahl der Veränderungen der Variablen k. k wird höchstens m mal um eins vergrössert; ansonsten nur verkleinert. Damit ist die Laufzeit $O(m)$.

+

- Die **while**-Schleife im Programmteil von **KMP-Matcher**, die den Aufruf von **shift** realisiert, ist auch unkritisch:
  In jedem Schleifendurchlauf wird j verringert. j kann aber bei der Suche durch den Text höchstens n-mal um eins erhöht werden. Damit ist die Laufzeit, die in dieser Schleife vergeht, beschränkt durch $O(n)$.

Theorem 2 *Der Algorithmus von Knuth, Morris und Pratt benötigt Zeit $O(n+m)$ und neben der Eingabe Platz $m + O(1)$.*

Dieser Algorithmus bietet schon das optimale *worst-case*-Zeitverhalten. In [Rive 77] wurde eine untere Schranke von $\Omega(n+m)$[2] bewiesen.

Aufgabe 2.2.6: 1
Implementieren Sie nur unter Zuhilfenahme der Prozedur **next** die Prozedur **shift**, welche die gleiche Funktionalität, wie die gleichnamige Tabelle haben soll.

2.3 Der Algorithmus von Rabin und Karp

Der folgende Algorithmus erlaubt eine weitere Einsparung des Speicherplatzes. Hierzu wird auf jeden Teilstring $t[i+1, i+m]$ des Texts t eine **Hashfunktion** h angewendet und das Ergebnis verglichen mit den Hashwert $h(x)$. Sind diese gleich, werden die entsprechenden Strings auf Gleichheit untersucht.

Der Witz an dieser Realisierung ist, daß bei geschickter Wahl der Hashfunktion aus $h(t[j+1, j+m])$ in konstant vielen Schritten der Wert $h(t[j+2, j+m+1])$ berechnet werden kann. Hierzu wählt man für geeignete Zahlen $d, q \in N$ die Funktion $h_{d,q} : \Sigma^m \to \mathbb{N}$ als

$$h_{d,q}(x) \;=\; \sum_{i=1}^{m} x[i] \cdot d^{m-i} \mod q \;.$$

Der Funktionswert von $h(t[2, m+1])$ berechnet sich nun aus $h_0 = h(t[1, m])$ als

$$d \cdot (h_0 - t[1] \cdot d^{m-1}) + t[m+1] \bmod q \;.$$

Das ergibt folgenden Algorithmus

[2] Natürlich ist im allgemeinen $m < n$, so daß eigentlich die asymptotische Laufzeit von $\Omega(n)$, resp. $O(n)$ gemeint ist. Das Anhängsel m dient zur Verdeutlichung der Laufzeitunterschiede bei der Analyse des Patterns und hat hauptsächlich historische Gründe.

+

```
Rabin-Karp-Matcher
input: t, x ∈ Σ*, d, q ∈ ℕ
{ n ← |t|
  m ← |x|
  h ← d^(m−1)
  p ← 0
  t_h ← 0
  for i ← 1 to m do
      { p ← (d · p + x[i]) mod q
        t_h ← (d · t_h + t[i]) mod q
      }
  i ← m
  found ← false
  while i ≤ n − m ∧ ¬found do
        { if p = t_h then
            if x[1, m] = t[i + 1, i + m] then found ← true
          t_h ← (d · (t_h − h · t[s + 1]) + t[s + m + 1]) mod q
          i ← i + 1
        }
  return i − 1
}
```

Theorem 3 *Die* worst-case*-Laufzeit des Rabin-Karp-Algorithmus beträgt $O(n \cdot m)$. Der über die Eingabe hinaus benötigte Speicherplatz ist konstant. Die* ***erwartete Laufzeit*** *des Rabin-Karp-Algorithmus für zufällig gewählte Eingabe d ist $O(n + m \cdot (1 + n/q))$.*

So erhält man für $q > n$ einen effizienten Algorithmus, der im Erwartungswert in Zeit $O(n + m)$ mit konstanten Speicherplatz die Stringsuche löst.

2.4 Der Algorithmus von Boyer und Moore

Betrachten wir als Beispieltext t und Suchstring x:

```
t = BONGOBONGOBONGOBONGOABBA
x = ABBA
```

Wenn wir beim Vergleich von x mit $t[1, m]$ diesmal nicht an der linken Position sondern mit $x[4]$ = `G` beginnen, fällt auf, daß `G` nicht im Suchtext vorhanden ist. Somit können wir das Suchwort um m Stellen nach rechts verschieben. Und das aufgrund eines einzigen Vergleichs!

Dies ist die Leitidee des Algorithmus von Boyer und Moore. Beim Anlegen des Suchstrings x an den Text wird immer von rechts nach links nach Buchstaben gefahndet, die nicht in x vorhanden sind.

```
t = BONGOBONGOBONGOBONGOABBA
       1   2   3   45   6987
x = ABBA   |   |   ||  10|||
        ABBA   |   ||   ||||
            ABBA   ||   ||||
                ABBA|   ||||
                 ABBA   ||||
                     ABBA|||
                        ABBA
```

Boyer-Moore-Matcher
input: $x, t \in \Sigma^*$
{ $n \leftarrow |t|$
  $m \leftarrow |x|$
  last-occur $\leftarrow$ **Compute-Last-Occurrence-Function**(x, Σ)
  good-suffix $\leftarrow$ **Compute-Good-Suffix-Function**(x)
  $i \leftarrow 0$
  $j \leftarrow m$
  while $i \leq n - m \wedge j > 0$ **do**
        { $j \leftarrow m$
          while $j > 0 \ \wedge \ x[j] = t[i + j]$ **do**
                $j \leftarrow j - 1$
          if $j > 0$ **then**
            $i \leftarrow i + \max(j - \textit{last-occur}(t[i + j]), \textit{good-suffix}(j))$
        }
  return $i + 1$
}

Die Funktion zur Berechnung von *last-occur* des am weitesten rechts vorkommenden Buchstaben c in x

$$\textit{last-occur}(c) \ = \ \max\{j \mid x[j] = c \ \vee \ j = 0\}$$

berechnet folgendes Programm:

Compute-Last-Occurence-Function
input: $x \in \Sigma^*$
{ $m \leftarrow |x|$
  for all $c \in \Sigma$
        last-occur$[c] \leftarrow 0$
  for $j \leftarrow 1$ **to** m
        last-occur$[x[j]] \leftarrow j$
  return *last-occur*
}

Die effiziente Berechnung der Funktion *good-suffix*, welche die Verschiebung des Suchstrings nach rechts beschreibt, gestaltet sich etwas komplizierter. Wir führen die Notation $a \sim b$ ein als:

$$a \sim b \quad \Longleftrightarrow \quad a \sqsupset b \vee b \sqsupset a \, .$$

+

Sei j die erste Stelle in x (von rechts gesehen), für die gilt $x[j] \neq t[i+j]$. Dann beschreiben alle Werte von $m-k$, wobei $x[1,k] \sim x[j+1,m]$ alle sinnvollen Werte für *good-suffix*(j).

Beispiel 2.4.4: $x[1,k] \sim x[j+1,m]$

1. $x[1,k] \supset x[j+1,m]$. Sei $x =$ `ababacaba` und $j = 5$.

```
        j=5
         |
a b a b a c a b a
          + + + +
            a b a . . .
                |
               k=3
```

   $k = 3$: $x[1,3] \supset x[6,9]$

2. $x[j+1,m] \supset x[1,k]$. Sei $x =$ `ccbabaab` und $j = 6$.

```
          j=6
           |
c c b a b a a b
            + +
      c c b a b ...
              |
             k=5
```

   $k = 5$: $x[6,8] \supset x[1,5]$

Somit erhält man die gewünschte Funktion *good-suffix* durch

$$\textit{good-suffix}(j) \;=\; s(j) \;:=\; m - \max\{k \mid 0 \leq k < m \;\wedge x[j+1,m] \sim x[1,k]\}\ .$$

Wir möchten diese Funtion s mit Hilfe von $\pi(j) = \textit{next}(j) = \max\{k < j \mid x[1,k] \supset x[1,j]\}$ (siehe KMP-Algorithmus) berechnen.

Fakt 2

$$\forall j \leq m \quad s(j) \leq m - \pi(m)$$

Sei $w = \pi(m)$. Dann ist $x[1,w] \supset x$. Da aber auch $x[j+1,m] \supset x$, folgt $x[1,w] \supset x[j+1,m]$, was den Fakt beweist.
Somit gilt also

$$s(j) = m - \max\{k \mid \pi(m) \leq k < m \;\wedge\; x[j+1,m] \sim x[1,k]\}\ .$$

Falls $x[1,k] \supset x[j+1,m]$ impliziert dies $k \leq \pi(m)$. Somit kann man s vereinfachen zu

$$s(j) = m - \max\{\pi(m)\} \;\cup\; \{k \mid \pi(m) \leq k < m \;\wedge\; x[j+1,m] \supset x[1,k]\}\ .$$

Sei nun $x^{\text{rev}} = x[m]x[m-1]\ldots x[1]$ und

$$\pi^{\text{rev}}(j) = \textit{next}^{\text{rev}}(j) = \max\{k \leq j \mid x^{\text{rev}}[1,k] \supset x^{\text{rev}}[1,j]\}\ .$$

Dann gilt folgendes Lemma:

Lemma 3 *Sei* $k_{\max} = \max\{k \leq m \mid x[j-1] \sqsupset x[1,k]\}$. *Dann gilt*

$$\pi^{rev}(l) = m - j \quad , \quad \textit{für } l = m - k_{\max} + m - j \ .$$

Beweis: Aus $x^{\text{rev}}[1, m-j] \sqsupset x^{\text{rev}}[1,l]$ folgt $\pi^{\text{rev}}(l) \geq m-j$.
Annahme: $p := \pi^{\textbf{rev}}(l) > m - j$
Es gilt auf Grund der Definition von π^{rev}:

$$\begin{array}{rrcl} & x^{\text{rev}}[1,p] & \sqsupset & x^{\text{rev}}[1,l] \\ \iff & x^{\text{rev}}[1,p] & = & x^{\text{rev}}[l-p+1,l] \\ \iff & x[m-p+1,m] & = & x[m-l+1, m-l+p] \\ \overset{l=2m-k_{\max}-j}{\iff} & x[m-p+1,m] & \sqsupset & x[1, k_{\max}-m+j+p] \end{array}$$

Nun gilt, da $p > m-j \Longrightarrow j+1 > m-p+1$ und da $x[j+1,m] \sqsupset x[m-p+1,m]$, aufgrund der Transitivität von $\sqsupset$:

$$x[j+1,m] \sqsupset x[1, k_{\max} + \underbrace{j+p-m}_{\geq 1}] \ .$$

Dies ist ein Widerspruch zur Maximalität von $k_{\max}$. Die Annahme ist also falsch. ∎
Nun ist also

$$\begin{array}{rcl} s(j) & = & m - \max\{\pi(m)\} \cup \{m - l + \pi^{\text{rev}}(l) \mid 1 \leq l \leq m \ \wedge \ j = m - \pi^{\text{rev}}(l)\} \\ & = & \min\{m - \pi(m)\} \cup \{l - \pi^{\text{rev}}(l) \mid 1 \leq l \leq m \ \wedge \ j = m - \pi^{\text{rev}}(l)\} \end{array}$$

Daraus folgt die Richtigkeit des folgenden Algorithmus.

```
Compute-Good-Suffix-Function
input: x ∈ Σ*
{ π ← Compute-Next(x)
  π^rev ← Compute-Next(reverse(x))
  for j ← 0 to m − 1
      good-suffix[j] ← m − π(m)
  for l ← 1 to m − 1
      { j ← m − π^rev(l)
        if good-suffix[j] > l − π^rev[l] then
           good-suffix[j] ← l − π^rev[l]
      }
  return good-suffix
}
```

Theorem 4 *Die Laufzeit des Boyer-Moore-Algorithmus ist im* worst-case $O(m \cdot n + |\Sigma|)$. *Der neben der Eingabe benötigte Speicherplatz beträgt* $O(m + |\Sigma|)$.

Damit bietet dieser Algorithmus im *worst-case* keinen Vorteil gegenüber dem **Brute-Force**-Verfahren. Die Vorteile zeigen sich aber im Erwartungswert[3].

[3]Beachten Sie, daß das KMP-Verfahren und das Rabin-Karp-Verfahren auch im *best-case* keine Beschleunigung im Erwartungswert haben (vorausgesetzt die Fundstellen befinden sich genügend weit hinten im Text).

+

Theorem 5 *Ist jeder Buchstabe des Textes gleich wahrscheinlich und enthält x nur einen konstanten Anteil aller Buchstaben aus Σ, so hat der Algorithmus von Boyer und Moore eine erwartete Laufzeit von $O(n/m + m + |\Sigma|)$.*

Beweis: Sei $p = \frac{|\{x[1],...,x[m]\}|}{|\Sigma|} < 1$ und sei $q = 1 - p$. Dann wird der Boyer-Moore-Algorithmus beim Lesen eines neuen Buchstabens aus t mit Wahrscheinlichkeit q den Wert i um m Schritte erhöhen. Ansonsten nehmen wir an, daß der Algorithmus i erst in m Schritten um 1 erhöht.
Der erwartete Fortschritt im Text ist also $q \cdot m + \frac{p}{m}$. Damit ist der erwartete Fortschritt nach t Schritten $t \cdot (q \cdot m + \frac{p}{m})$.
Der erwartete Fortschritt nach Lesen von $\frac{n}{q \cdot m}$ Buchstaben ist also

$$n + \frac{p \cdot n}{m^2} > n \ .$$

Elementare Methoden aus der Wahrscheinlichkeitstheorie ergeben nun das Theorem. ∎ Für andere realistischere Wahrscheinlichkeitsmodelle kann die gleiche erwartete Laufzeit bewiesen werden. Darüber hinaus gibt es eine Reihe von Varianten der hier vorgestellten Stringsuche-Algorithmen, die eine verschiedene Eigenschaften kombinieren [Ste 91].

Aufgabe 2.4.7: 1
Geben Sie ein Beispiel an für

$$x \sqsupset y \not\Leftrightarrow y \sqsupset x \ .$$

Aufgabe 2.4.8: 6
Konstruieren Sie Eingaben, für die der Boyer-Moore-Algorithmus die *worst-case*-Laufzeit annimmt.

Aufgabe 2.4.9: 10
Geben Sie einen Algorithmus an, der die erwartete Laufzeit des Boyer-Moore-Verfahren und die *worst-case*-Laufzeit des Knuth-Morris-Pratt-Algorithmus besitzt (Hinweis: Überlegen Sie sich vorher im allgemeinen, ob es ein Programm gibt, das immer die bessere Laufzeit zweier Algorithmen besitzt, welche dasselbe Problem lösen).

Kapitel 3

Die Datenstruktur disjunkter Mengen

Wir betrachten eine Menge $\mathcal{S}$ endlicher disjunkte Mengen $S_1, \ldots, S_k \subseteq \mathcal{U}$ mit folgenden Eigenschaften:

- Zu jedem Zeitpunkt kommt ein Element $x \in \mathcal{U}$ dieser Mengen ausschließlich in einer dieser Mengen vor. Wir nennen daher die zu x gehörende Menge S_x.
- Jede Menge hat ein Element als Repräsentanten. Der Repräsentant der Menge S_x heißt $r(x)$.
- Neue Elemente können in das System nur eingefügt werden, indem man sie als ein-elementige Mengen definiert.
- Die Mengen können sich dynamisch verändern. Sie können sogar aufhören zu existieren.

Folgende elementaren Operationen stehen zur Verfügung

1. **Make-Set**(x)
   erzeugt die neue ein-elementige Menge S_x. Es wird vorausgesetzt, daß x in keiner der bestehenden Mengen vorhanden ist.
2. **Union**(x, y)
   Die Mengen S_x und S_y werden vereinigt. Es entsteht hierbei eine neue Menge. Die alten Mengen S_x und S_y existieren nach dieser Operation nicht mehr, da sie sonst die Bedingung der Disjunktheit verletzen würden.
3. **Find-Set**(x)
   gibt den Repräsentanten der Menge S_x aus.

   Aufgabe 3.0.10: 5
   Finden Sie eine eigene Datenstruktur, die dieses Problem löst. Welche Laufzeiten haben Ihre elementaren Operationen.

3.1 Notation der Graphen

Wir unterscheiden zwei Arten von Graphen: **ungerichtete** und **gerichtete** Graphen.
Ein **ungerichteter Graph** $G = (V, E)$ besteht aus einer Knotenmenge V und einer Teilmenge E der Menge aller zwei-elementiger Teilmengen von V:

$$E \subseteq \{\{u, v\} \mid u, v \in V\} .$$

Ein **gerichteter Graphen** $G = (V, E)$ mit Knotenmenge V besitzt dagegen als Kantenmenge eine Knotentupelmenge:

$$E \subseteq \{(u, v) \mid u, v \in V\} .$$

Für eine gerichtete Kante von (u, v) schreiben wir auch $u \to v$. Dies schließt auch **Schlingen** ein: Das sind Kanten mit gleichen Start- und Zielknoten. Die Adjazensmenge $Adj(u)$ eines Knoten u ist die Menge aller Knoten v mit $u \to v$:

$$Adj(u) := \{v \in V \mid (u, v) \in E\}$$

Ein **Pfad** $p = (v_1, \ldots, v_k)$ mit $v_i \in V$ ist ein Weg von v_1 bis v_k entlang der Kanten (in Kantenrichtung):

$$\forall i \quad (v_i, v_{i+1}) \in E$$

Als Notation für einen Pfad p von v_1 nach v_2 schreiben wir $v_1 \overset{p}{\rightsquigarrow} v_2$. Ein **Kreis** $k = (v_1, \ldots, v_k, v_1)$ ist ein Pfad mit gleichen Start- und Endknoten.
Ein **gerichteter azyklischer Graph** (*directed acyclic graph* — ***DAG***) ist ein gerichteter Graph ohne Kreise. Ein Graph ist **zusammenhängend**, falls zwischen zwei Knoten es immer einen Pfad im Graph gibt.

3.2 Zusammenhängende Komponenten eines Graphen

Wir betrachten hier ungerichtete Graphen. Ein Teilgraph K (Komponente) eines Graphen $G = (V, E)$ ist zusammenhängend, wenn jeder Knoten des Teilgraphes K über einen Weg in G mit jedem anderen in K erreichbar ist.
Für einen gegebenen Graph wollen wir jetzt eine Prozedur **Same-Component** zur Verfügung stellen, die auf Eingabe zweier Knoten ausgibt, ob sie zu dem gleichen zusammenhängenden Teilgraph angehören.
Wir implementieren diese Prozedur mit der oben eingeführten disjunkte-Mengendatenstrukur. Die Initialisierung berechnet **Connected-Component.** V bezeichnet hier die Knotenmenge eines Graphen und E die Menge der Kanten.

```
Connected-Component
input: Graph G = (V,E)
{ for all v ∈ V
        Make-Set(v)
  for all {u,v} ∈ E
        if Find-Set(u) ≠ Find-Set(v) then
          Union(u,v)
}
```

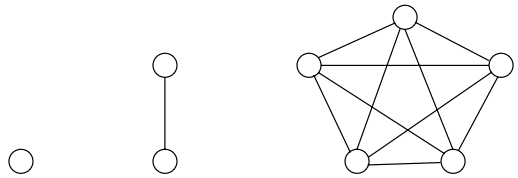

Abbildung 3.1: Ein ungerichteter ziemlich unzusammenhängender Graph.

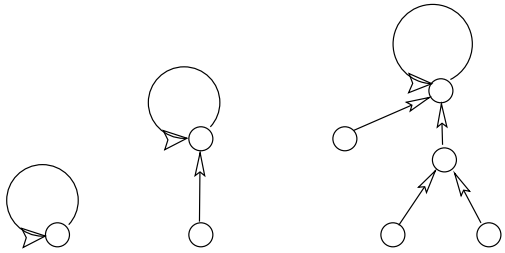

Abbildung 3.2: Ein Wald ist eine Menge Bäume (paßt zum Graph).

Folgende Prozedur entscheidet nun ob u und v in einem zusammenhängenden Teilgraph sind.

Same-Component
input: u, v
{ **return Find-Set**(u) = **Find-Set**(v)
}

3.3 Effiziente Realisierung der Datenstruktur

Jedes Element $x \in \mathcal{U}$ besitzt einen Zeiger $p(x)$, der auf sich oder andere Elemente zeigt. Die Zeigerstruktur stellt einen **aufwärts gerichteten Wald** dar, d.h. sie besteht aus einer Menge von Bäumen, deren Kanten (Zeiger) alle in Richtung Wurzel zeigen (siehe Abbildung 3.2). Die Wurzel referenziert hierbei auf sich selbst. Jeder Baum steht für eine disjunkte Menge; Die Elemente stehen weiterhin eins zu eins für die Elemente aus $\mathcal{U}$.

Ferner wird in der Wurzel w eines jeden Baums die Größe des Baums abgespeichert. Dies geschieht nicht explizit, vielmehr wird der binäre Logarithmus der Größe in der Variable *rank*(w) approximiert. Diese Information wird genutzt, damit bei der Vereinigung zweier Bäume, der kleinere Baum unter den größeren gehängt wird. Wir werden später bei der Analyse diese Art der Größenabschätzung noch zu nutzen wissen.

Folgende Prozeduren implementieren nun die disjunkte-Mengendatenstruktur.

+

```
Make-Set
input: x ∈ 𝒰
{ p(x) ← x
  rank(x) ← 0
}
```

```
Union
input: x, y
{ Link(Find-Set(x), Find-Set(y))
}
```

Link verkettet zwei Bäume miteinander.

```
Link
input: x, y
{ if rank(x) > rank(y) then
    p(y) ← x
  else if rank(x) < rank(y) then
    p(x) ← y
  else   \* rank(x) = rank(y) *\
    { p(x) ← y
      rank(y) ← rank(y) + 1
    }
}
```

```
Find-Set
input: x
{ if x ≠ p(x) then
    p(x) ← Find-Set(p(x))
  return p(x)
}
```

Find-Set versetzt den Zeiger von x direkt auf den Repräsentanten $r(x)$. Dies geschieht aber auch für jedes Element, das auf dem Weg zwischen x und $r(x)$ lag. Diesen Effekt nennen wir **Pfadkompression.**

3.4 Laufzeit des Verfahrens

Die Laufzeit weicht von der linearen Funktion asymptotisch leicht ab. Um diese Abweichung genau beschreiben zu können, definieren wir die (modifizierte[1]) Ackermannfunktion A für alle $i \geq 2$ und $j \geq 2$ als:

[1] Im Zusammenhang mit der `Loop`-Sprache wird eine andere Version der Ackermannfunktion A' gewählt. Es gilt hierbei $A(i-1,j) \leq A'(i,j) \leq A(i,j)$, so daß hier kein wesentlicher Unterschied besteht.

$$\begin{aligned} A(1,1) &:= 2 \\ A(1,j) &:= 2^j \\ A(i,1) &:= A(i-1,2) \\ A(i,j) &:= A(i-1,A(i,j-1)) \end{aligned}$$

Folgende Tabelle gibt einige Werte dieser Funktion wieder

	$A(i,1)$	$A(i,2)$	$A(i,3)$	$A(i,4)$
$A(1,j)$	2	4	8	16
$A(2,j)$	4	16	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$
$A(3,j)$	16	$2^{2^{\cdot^{\cdot^{\cdot^2}}}\}16}$	$2^{2^{\cdot^{\cdot^{\cdot^2}}}\}2^{2^{\cdot^{\cdot^{\cdot^2}}}\}16}}$	$2^{2^{\cdot^{\cdot^{\cdot^2}}}\}2^{2^{\cdot^{\cdot^{\cdot^2}}}\}2^{2^{\cdot^{\cdot^{\cdot^2}}}\}16}}}$

Wir definieren die (modifizierte) Umkehrfunktion α der Ackermannfunktion.

$$\alpha(m,n) := \min\left\{ i \geq 1 \mid A\left(i, \left\lfloor \frac{m}{n} \right\rfloor\right) > \log_2 n \right\} .$$

Theorem 6 *Die Hintereinanderausführung von m Datenoperationen des Typs* **Make-Set**, **Find-Set** *und* **Union** *auf einer Menge von $|\mathcal{U}| = n$ Elementen benötigt die Laufzeit*

$$O(m \cdot \alpha(m,n)) .$$

Aufgabe 3.4.11: 2
Geben Sie eine obere Schranke für $m \cdot \alpha(m,n)$ an, falls m und n die Zahl 10^{80} (geschätzte Anzahl aller Atome im Universum) nicht überschreiten.

Wir werden Theorem 6 nicht beweisen (für einen Beweis siehe [Tarj 83, Tarj 75]). Vielmehr beweisen wir einen schwächeren Satz, dessen Ergebnis sich für praktische Belange kaum vom obigen unterscheidet. Hierzu definieren wir die iterierte Logarithmusfunktion itlog n als

$$\text{itlog } n := \min\left\{ i \geq 0 \;\middle|\; 2^{2^{\cdot^{\cdot^{\cdot^2}}}\}i-1} \geq n \right\} .$$

Theorem 7 *Die Hintereinanderausführung von m Datenoperationen des Typs* **Make-Set**, **Find-Set** *und* **Union** *auf einer Menge von $|\mathcal{U}| = n$ Elementen benötigt die Laufzeit*

$$O(m \cdot \text{itlog } n) .$$

Aufgabe 3.4.12: 2
Geben Sie eine obere Schranke für $m \cdot$itlog n an, falls m und n die Zahl 10^{80} (geschätzte Anzahl aller Atome im Universum) nicht überschreiten.

+

Eigenschaften von *rank*

Wir betrachten jetzt immer einen beliebigen Schnappschuß nach Beendigung einer der vier oben beschriebenen Prozeduren und untersuchen die Eigenschaften der Variablen *rank*.

Lemma 4 *1. Für alle Knoten x gilt*

$$rank(x) \leq rank(p(x)) \ .$$

2. Falls x keine Wurzel ist ($x \neq p(x)$), gilt

$$rank(x) < rank(p(x)) \ .$$

3. Der Wert rank(x) ist zu Beginn 0 und wächst mit der Zeit monoton an bis schließlich $x \neq p(x)$. Danach verändert sich rank(x) nicht mehr.

4. Der Wert rank($p(x)$) ist eine mit der Zeit monoton wachsende Funktion.

Aufgabe 3.4.13: 4
Beweisen Sie das Lemma.

Sei **size(x)** die Anzahl aller Knoten des Baumes mit der Wurzel x.

Lemma 5 *Für alle Baumwurzeln x gilt*

$$\text{size}(x) \ \geq \ 2^{rank(x)} \ .$$

Beweis: Induktion über die Anzahl der **Link**-Operationen. Die Operation **Find-Set** verändert weder die Größe eines Baumes noch den Wert *rank*(x) der Wurzel x.
Induktionsverankerung: Vor der ersten **Link**-Operation sind alle Werte *rank*(x) $= 0$. Die Bäume bestehen nur aus einzelnen Knoten.
Induktionsschritt: Angenommen, das Lemma gilt vor der nächsten Ausführung von **Link**. Dann sei *rank* der Wert davor und *rank'* der danach.
Falls *rank*(x) $\neq$ *rank*(y), wird der größere Wert als Wurzelwert *rank'*(w) übernommen. Die Größe des Baumes ergibt sich aus der Summe von $|S_x|$ und $|S_y|$. Damit gilt das Lemma für diesen Fall.
Falls *rank*(x) $=$ *rank*(y) ergibt sich die neue Baumgröße als size(x) + size(y). Der neue Rang der Wurzel ist *rank*(x) $+ 1$. Daher gilt:

$$2^{rank'(x)} = 2^{rank(x)+1} \geq \text{size}(x) + \text{size}(y) = \text{size}'(x) \ .$$

∎

Lemma 6 *Für alle $r \geq 0$ gilt: Es gibt höchstens $\frac{n}{2^r}$ Knoten x mit rank(x) $= r$.*

Beweis: Betrachten wir ein festes $r > 0$. Damit ein Rang der Größe r überhaupt entstehen kann, müssen zwei Bäume mit Wurzelrang $r-1$ vereinigt werden. Nach Lemma 5 sind aber dann 2^r Knoten in den Bäumen daran beteiligt, die der Rangerzeugung anderer Wurzeln mit Rang r nicht mehr zur Verfügung stehen, da eventuelle Väter der Wurzel höheren Rang erhalten werden (Lemma 4).
Andererseits ist in diesem Baum (in diesem Moment) auch kein weiterer Knoten des Rangs r vorhanden. Da insgesamt nur n Elemente vorhanden sind und für die Erzeugung eines Rangs r 2^r Knoten notwendig sind, können höchstens $n/2^r$ Knoten den Rang r erhalten. ∎

+

Lemma 7 *Für alle $x \in \mathcal{U}$ gilt*

$$rank(x) \leq \log n \ .$$

Beweis folgt direkt aus dem Lemma davor.

Aufgabe 3.4.14: 5
Konstruieren Sie eine Folge von **Make-Set**, **Link** und **Find-Set**-Befehlen, die einen Wald aufbauen, in der ein Knoten Rang 4 besitzt.

Aufgabe 3.4.15: 7
Versuchen Sie dasselbe mit einer Folge von **Make-Set**, **Union** und **Find-Set**-Befehlen.

Lemma 8 *Eine Folge von m* **Make-Set**, **Union** *und* **Find-Set**-*Befehlen kann in eine Folge von m'* **Make-Set**, **Link** *und* **Find-Set**-*Befehlen umgewandelt werden, so daß, falls die zweite Folge Laufzeit T benötigt, die erste Laufzeit $O(T)$ besitzt.*

Hier wird jeder **Union**-Befehl durch den entsprechenden Aufruf von **Link** und **Find-Set** ersetzt.

Theorem 8 *Eine Folge von m* **Make-Set**, **Link** *und* **Find-Set**-*Operationen auf einer Menge $\mathcal{U}$ von $n = |\mathcal{U}|$ Elementen benötigt höchstens eine Laufzeit von $O(m \cdot \text{itlog } n)$ Schritten.*

Beweis: Wir untersuchen hierzu den für jede der drei Operationen erforderlichen Aufwand separat:

1. **Make-Set**:

   Jede **Make-Set**-Operation benötigt konstanten Aufwand. Da zu Beginn keine Menge vorhanden ist, höchstens m solche Operationen vorkommen können und eine **Make-Set**-Operation Vorbedingung zur Verwaltung von Elementen ist, können wir die Größe n aller Elemente durch m beschränken: $n \leq m$.

2. **Link**:

   Jede Ausführung von **Link** (ist nur erlaubt auf den Wurzeln zweier Bäume) kostet konstanten Aufwand.

3. **Find-Set**:

   Die Analyse dieser Operation ist schwieriger. Hierzu heften wir den enstehenden Laufzeitaufwand dieser Operation an die beteiligten Elemente — wir belasten ein Element mit **Kosten** — und addieren dann am Schluß diese Werte. Dieser Buchhaltungstrick erlaubt es uns, Laufzeitspitzen einzelner Operationen (auf besonders tiefen Knoten) auf andere umzulegen und ermöglicht es uns so, die Effizienz des Verfahren zu beweisen. Dieses Vorgehen nennt man eine **amortisierte Laufzeitanalyse.**

   Betrachten wir nun eine Situation im Wald, nachdem schon einige **Make-Set**, **Link** und **Find-Set**-Operationen ausgeführt wurden. Jetzt steht eine **Find-Set**-Operation an. Uns interessieren besonders die dem Knoten anhaftenden *rank*-Werte. Diese unterteilen wir in Blöcke.

+

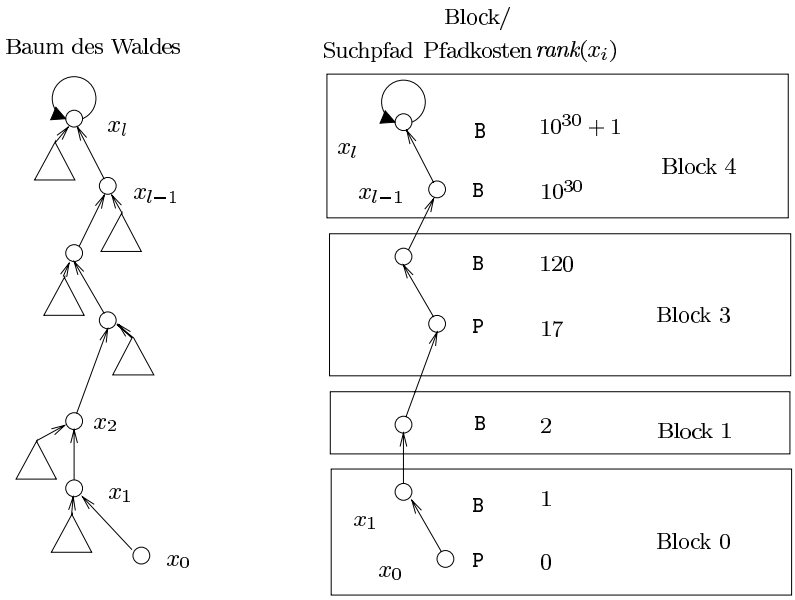

Abbildung 3.3: Verteilung von Block- und Pfadkosten auf einem Suchpfad.

Block	*rank*-Werte enthaltener Knoten
0	0,1
1	2
2	3,4
3	5,6,..., 16
4	$17, 18, \ldots, 2^{16}$
$\vdots$	$\vdots$
i	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}i-2 + 1, \ldots, 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}i-1$

In Block j sind also Knoten x mit $rank(x) \in [B(j-1)+1, B(j)]$, wobei $B : \mathbb{N} \to \mathbb{N}$ definert wird als:

$$B(j) := \begin{cases} -1\,, & j = -1 \\ 1\,, & j = 0 \\ 2\,, & j = 1 \\ 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}j-1\,, & j \geq 2 \end{cases}$$

Sei nun x_0 die Eingabe von **Find-Set** und $(x_1, x_2, \ldots x_l)$ der Pfad zur Wurzel x_l des Baumes. Wir nennen $x_0, x_1 \ldots, x_l$ den **Suchpfad** von x_0 im Baum. Der Aufwand dieser **Find-Set** Operation ist also $O(l)$. Wir werden jetzt diesen Aufwand, wie in Abbildung 3.3 gezeigt, auf die Knoten des Suchpfades aufteilen.

Jeder Knoten des Suchpfades wird Kosten im Wert von **einer Einheit** erhalten. Wir unterscheiden dabei in der Qualität aber **Blockkosten** und **Pfadkosten**:

+

- Ein Knoten x_i erhält einen Blockkosteneintrag, wenn er Wurzel oder Sohn der Wurzel ist oder wenn sein Vater einem anderen Block als x_i angehört. Dies ist äquivalent zu

$$p(x_i) = x_l \quad \vee \quad \text{itlog } \mathit{rank}(x_i) < \text{itlog } \mathit{rank}(x_{i+1}) \ .$$

- Alle anderen Knoten des Suchpfades erhalten einen Pfadkosteneintrag.

Wir werden Blockkosten und Pfadkosten jetzt getrennt abschätzen:

(a) Blockkosten
Die Anzahl der Blöcke ist beschränkt durch $\max_x \text{itlog } \mathit{rank}(x)$. Nach Lemma 7 gilt $\mathit{rank}(x) \leq \log n$. Da höchstens soviel Blockkosten pro Ausführung einer **Find-Set**-Operation entstehen, wie die Anzahl der Blöcke auf dem Suchpfad ist plus zwei für Wurzel und Sohn, ist der Blockkostenaufwand pro **Find-Set**-Operation beschränkt durch

$$\max_x \text{itlog } \mathit{rank}(x) \ \leq \ \text{itlog } \log n \ \leq \ \text{itlog } n \ .$$

(b) Pfadkosten
Falls ein Knoten x_i, der weder Wurzel noch Sohn einer Wurzel ist, einen Blockkosteneintrag erhalten hat, werden ihm niemals wieder Pfadkosten berechnet.
Diese Eigenschaft gilt, weil nach Lemma 4 die Differenz $\mathit{rank}(p(x)) - \mathit{rank}(x)$ mit der Zeit monoton zunimmt. Sobald x nicht mehr Wurzel ist, bleibt $\mathit{rank}(x)$ konstant. Dann erst kann diese Differenz ungleich 0 sein, und folglich kann nur noch $\mathit{rank}(p(x))$ seinen Wert ändern. Somit nimmt auch die Funktion $\text{itlog } \mathit{rank}(p(x)) - \text{itlog } \mathit{rank}(x)$ monoton zu.
Falls also der Vater eines Knoten x_i jemals zu einem fremden Block gehört (was äquivalent zu $\text{itlog } \mathit{rank}(p(x)) - \text{itlog } \mathit{rank}(x) > 0$ ist) werden zukünftige Väter (oder eben dieser) immer fremd zu x_i bleiben.

Wie oft kann x_i in Block j Pfadkosten erhalten, bis er ein Vater in einem fremden Block besitzt (und damit nie wieder Pfadkosten erhält) ?
Das kann $B(j) - B(j-1) - 1$ mal geschehen.
Begründung: Der Rang des Vaters von x_i nimmt mit jeder **Find-Set**-Operation streng monoton zu (Lemma 4). Danach ist der minimale Abstand 1, der maximale $B(j) - B(j-1) - 1$ und mit jedem Pfadkosteneintrag vergrößert sich dieser.
Definieren wir also $\boldsymbol{N(j)}$ als die Anzahl aller Knoten x, die im gesamten Verlauf aller m Operationen jemals einen *rank*-Wert besitzen, der in Block j liegt. Nach Lemma 6 gilt also

$$N(j) \leq \sum_{r=B(j-1)+1}^{B(j)} \frac{n}{2^r} \ .$$

Für $j = 0$ ergibt dies

$$N(0) \leq \frac{3n}{2} = \frac{3}{2}\frac{n}{B(0)} \ .$$

Für $j \geq 1$ läßt sich $N(j)$ wie folgt abschätzen:

$$N(j) \ \leq \ \frac{n}{2^{B(j-1)+1}} \underbrace{\sum_{r=0}^{B(j)-(B(j-1)+1)} \frac{1}{2^r}}_{<2}$$

$$\begin{aligned} &< \frac{n}{2^{B(j-1)}} \\ &= \frac{n}{B(j)} \end{aligned}$$

Damit ist $N(j) \leq \frac{3}{2}\frac{n}{B(j)}$ für alle j gültig.

Sei nun $\boldsymbol{P(n)}$ die Summe aller Pfadkosten, die im Verlauf aller m Operationen verteilt werden. Dann gilt

$$P(n) \leq \underbrace{\sum_{j=0}^{\text{itlog } n}}_{\substack{\text{Summe über}\\ \text{alle Blöcke.}}} \underbrace{\frac{3}{2}\frac{n}{B(j)}}_{\substack{\text{Maximale Anzahl}\\ \text{aller Knoten in}\\ \text{Block } j.}} \underbrace{(B(j) - B(j-1) - 1)}_{\substack{\text{Maximale Pfadkosten, die}\\ \text{einem Knoten in Block } j\\ \text{berechnet werden können.}}}$$

$$\begin{aligned} &\leq \sum_{j=0}^{\text{itlog } n} \frac{3}{2} \cdot \frac{n}{B(j)} B(j) \\ &\leq \frac{3}{2} \cdot n \cdot (\text{itlog } n + 1) \ . \end{aligned}$$

Die Gesamtkosten sind also

$$\text{Blockkosten} + \text{Pfadkosten} = O(m \cdot \text{itlog } n + \tfrac{3}{2} \cdot n \cdot (\text{itlog } n + 1)) = O(m \cdot \text{itlog } n) \ .$$

∎

Aufgabe 3.4.16: 7

Kann dem Sohn einer Wurzel Pfadkosten berechnet werden, nachdem er einmal einen Blockkosteneintrag erhielt? Begründen Sie Ihre Antwort!

Kapitel 4

Kürzeste Wege in Graphen

4.1 Der kürzesteste Weg von einem Startpunkt

Für das Wegeproblem in gerichteten Graphen $G = (V, E)$ betrachtet man eine **Gewichtungsfunktionen** $w : E \to \mathbb{R}$. Das **Gewicht** (die Länge) eines Pfads $p = (v_1, v_2, \ldots, v_k)$ berechnet sich aus der Summe der Kanten des Pfads:

$$w(p) := \sum_{i=0}^{k-1} w((v_i, v_{i+1})) \ .$$

Die Funktion $\delta : V \times V \to \mathbb{R}$ bezeichnet Entfernung zwischen zwei Knoten (siehe auch Abb. 4.1):

$$\delta(u, v) := \begin{cases} \min\{w(p \mid u \overset{p}{\rightsquigarrow} v\} \ , & \text{falls ein Pfad von } u \text{ nach } v \text{ existiert,} \\ \infty \ , & \text{sonst.} \end{cases}$$

Wir betrachten also folgendes Problem

> **Kürzeste Wege von einem Startknoten *(single source shortest path)***
> Geg.: Graph $G = (V, E)$, Startknoten s, Gewichtungsfunktion $w : E \to \mathbb{R}$
> Ges.: Für jeden Knoten $g \in V$ ein kürzester Pfad p (falls vorhanden),
> d.h. $s \overset{p}{\rightsquigarrow} g$, mit $w(p) = \delta(s, g)$.

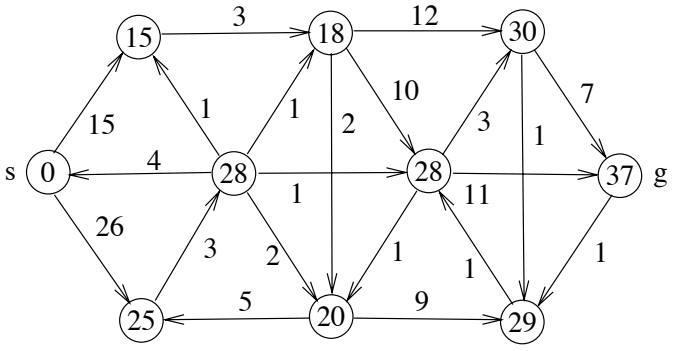

Abbildung 4.1: Graph mit positiven Kantengewichten und Entfernung $\delta(s, u)$

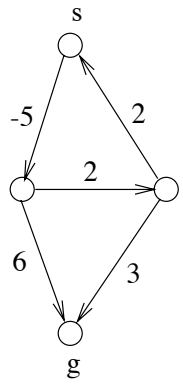

Abbildung 4.2: Der kürzeste Weg kann manchmal sehr lang sein.

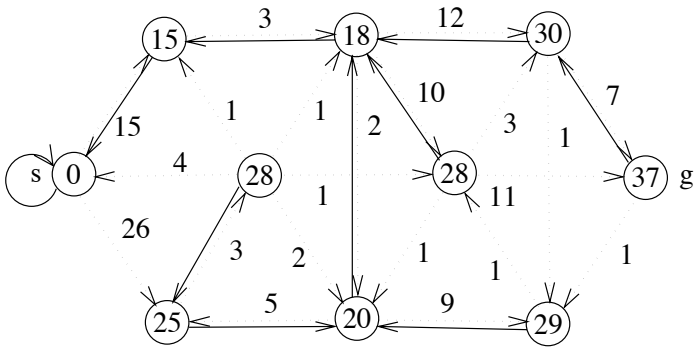

Abbildung 4.3: Graph mit kürzesten Wege-Baum

Negative Kantengewichte können zu einigen Problemen führen. Betrachten Sie die Abb. 4.2. Hier gibt es einen Kreis k mit negativen Gewicht $w(k) < 0$. Damit gibt es keinen kürzesten Weg: Man kann den Kreis beliebig oft entlangfahren und man erhält immer kürzere (negative) Entfernungen vom Startpunkt s.
Wir betrachten daher für den Anfang nur das Problem des kürzesten Weges mit nichtnegativen Kantengewichten: $w : E \to \mathbb{R}_0^+$.

Darstellung der Lösung Wir werden eine Funktion $d : V \to \mathbb{R}_0^+$ berechnen, die nach Abarbeitung der Algorithmen die Entfernung jedes Knoten zum Startknoten darstellt. Für den kürzesten Weg berechnen wir für jeden Knoten einen Wert $\pi : V \to V$, der angibt über welchen Knoten dieser Knoten am kürzesten erreichbar ist.
Der kürzeste Weg von $s \overset{p}{\rightsquigarrow} g$ ergibt sich dann also aus $v_1 = \pi(s)$, $\forall i : v_{i+1} = \pi(v_i)$ als $p = (g, v_k, \ldots, v_1, s)$ (Abb. 4.3).
Interpretiert man die Abbildung $\pi(u) = v$ als Kanten $u \to v$, so ergibt der resultierende Graph (V, π) einen aufwärts gerichteten Baum, den **kürzesten Wege-Baum.**

4.1.1 Der Algorithmus von Dijkstra

Wir benötigen im Vorfeld einige Rechenregel für das Symbol ∞. Für jede Zahl z gilt $\infty + z = \infty - z = \infty + \infty = \infty$. Genauso gilt $-\infty + z = -\infty - z = -\infty - \infty = -\infty$. Ferner gilt für alle

Zahlen z: $-\infty < z < \infty$. Sowohl $\infty - \infty$, als auch $\frac{\infty}{\infty}$ sind nicht definiert.

```
Init-Single-Source
input: G = (V, E), s ∈ V
{ for all v ∈ V do
          { d(v) ← ∞
            π(v) ← v
          }
  d(s) ← 0
}
```

Folgende Prozedur überprüft, ob die Kante (u, v) den Weg von u nach v abkürzen kann.

```
Relax
input: u, v ∈ V
{ if d(v) > d(u) + w(u, v) then
    { d(v) ← d(u) + w(u, v)
      π(v) ← u
    }
  }
```

Dijkstras Algorithmus wendet diese lokale Optimierung gemäß einer *greedy*-Strategie an (Algorithmen mit *greedy*-Strategie werden auch als **gierigen Algorithmen** bezeichnet). Aus der Menge der nicht besuchten Knoten wird immer der nächstgelegene genommen und bewertet.

```
Dijkstra
input: G = (V, E), s ∈ V, w : E → ℝ₀⁺
{ Init-Single-Source(G, w)
  S ← ∅
  Q ← V
  while Q ≠ ∅ do
        { u ← Element aus Q mit minimalen Wert d(u)
          S ← S ∪ {u}
          Q ← Q \ {u}
          for all v ∈ Adj(u) do
                Relax(u, v)
        }
}
```

Laufzeit des Verfahrens Entscheidend für die Effizienz dieses Verfahrens ist die Verwendung einer effizienten Datenstruktur zur Berechnung eines minimalen Elements in Q. Wir setzen voraus, daß Wörterbücher mit einer logarithmischen Zeitkomplexität von Einfüge-, Lösch- und

+

Suchoperationen bekannt sind. Verwendet man solche eine effiziente Datenstruktur so ergibt sich eine Laufzeit von $O(|E|\log|V|)$ wie man folgender Tabellen entnimmt:

Anzahl	Operation	Laufzeit einer Operation
1	Initialisierung mit $\|V\|$ Elementen	$\|V\|$
$\|V\|$	Löschen eines Elements	$\log\|V\|$
$\|V\|$	Suchen des kleinsten Elements	$\log\|V\|$
$\|E\|$	Verringern des Werts eines Elements	$\log\|V\|$
	Summe aller Laufzeiten:	$O((\|E\|+\|V\|)\log\|V\|)$

Es gibt aber eine für dieses Problem besser angepaßte Datenstruktur: **Fibonacci-Heaps** (siehe hierzu [CLR 90] ,[FT 87]). Diese Datenstruktur stellt nicht die voll Wörterbuchfunktion zur Verfügung. Als Suchkriterium steht nur die Minimumfunktion zur Verfügung. Dafür kann man aber in konstanter amortisierter Zeit den Schlüsselwert eines Elements verringern:

Anzahl	Operation	Laufzeit einer Operation
1	Initialisierung mit $\|V\|$ Elementen	$\|V\|$
$\|V\|$	Löschen eines Elements	$\log\|V\|$
$\|V\|$	Suchen des kleinsten Elements	$\log\|V\|$
$\|E\|$	Verringern des Werts eines Elements	$O(1)$
	Summe aller Laufzeiten:	$O(\|V\|\log\|V\|+\|E\|)$

Somit folgt unter Verwendung dieser Datenstruktur.

Theorem 9 *Der Algorithmus von Dijkstra hat eine Laufzeit von* $O(|V|\log|V|+|E|)$.

Korrektheit von Dijkstras Algorithmus

Lemma 9 *Falls* $p=(s,\ldots,u,v)$ *ein kürzester Pfad von* s *nach* v *ist, gilt*

$$\delta(s,v)=\delta(s,u)+w(u,v)\ .$$

Beweis:
Annahme $\delta(s,v)>\delta(s,u)+w(u,v)$:
Sei p' ein kürzester Pfad von s nach u ($s \overset{p'}{\rightsquigarrow} u$). Dann könnte man aus den kürzesten Pfad p' den kürzesten Pfad p von s nach v weiter verkürzen. Ein Widerspruch.
Annahme $\delta(s,v)<\delta(s,u)+w(u,v)$:
Sei p'' der Teilpfad von p von s nach u. Dann gilt $\delta(s,u)\leq w(p'')$ und $w(p)=w(p'')+w(u,v)=\delta(s,v)$. Wieder ein Widerspruch. ∎

Theorem 10 *Dijkstras Algorithmus arbeitet korrekt für nichtnegative Gewichtungsfunktionen.*

Beweis: Wir beweisen die Korrektheit für positive Gewichte. Die Verallgemeinerung auf nichtnegative Gewichte sei dem Leser zur Übung überlassen. Folgende Schleifeninvarianten gelten für die (äußere) **while**-Schleife:

$$\forall v\in V \quad d(v) \geq \delta(s,v)\ , \tag{4.1}$$

$$\forall v\in S \quad d(v) = \delta(s,v)\ , \tag{4.2}$$

$$\forall v\in S\ \forall u\in Adj(v) \quad d(v)-d(u) \leq w(u,v)\ , \tag{4.3}$$

$$\forall u\notin S \quad \delta(s,u) > \max_{v\in S}\delta(s,v)\ . \tag{4.4}$$

+

Wir beweisen die Richtigkeit dieser Invarianten. Aus (4.2) und aus $S = V$ nach Beendigung der Schleife folgt sofort die Richtigkeit des Algorithmus.
Zu Beginn sind sie erfüllt, da $S = \{s\}$ und $d(s) = 0$.
Sei u das Element, das als minimales Element aus Q gewählt wird. Ein minimaler Pfad von s nach u sei $p = (s, \ldots, v, u)$.
Angenommen $v \notin S$, dann sei x der erste der Vorgänger von v in p, der in S liegt und $y \notin S$ dessen Nachfolger in p. Nun gilt

$$\delta(s, y) = \delta(s, x) + w(x, y) = d(x) + w(x, y) \geq d(y) \ .$$

Aufgrund der Wahl von u gilt $d(u) \leq d(y)$ und für das Gewicht von p gilt

$$w(p) \geq \delta(s, y) + w(v, u) \geq d(y) + w(v, u) \geq d(u) + w(v, u) > d(u) \ .$$

Also ist p nicht der kürzeste Weg. Ein Widerspruch.
Somit gilt $v \in S$ und nach (4.2) und (4.3).

$$d(u) \leq d(v) + w(v, u) = \delta(s, v) + w(v, u) = \delta(s, u)$$

Da $d(u) \geq \delta(s, u)$ (4.1), gilt also nach Einfügen von u in S die Invariante (4.2):

$$d(u) \ = \ \delta(s, u) \ .$$

Somit ist u auch gemäß δ optimal gewählt worden, dies ergibt die Invariantenbedingung (4.4) für den nächsten Durchlauf.
Die Invarianten (4.1) und (4.3) werden nach dem Einfügen von u durch die Aufrufe **Relax** auf allen von u benachbarten Knoten sichergestellt. ∎

Aufgabe 4.1.17: 5
Vervollständigen Sie den Beweis.

Aufgabe 4.1.18: 6
Hein Blöd ist in der Lage, Dijkstras Algorithmus auch auf einem Graph mit negativen Kantengewichten einzusetzen. Hierzu bestimmt er zuerst die Kante mit den kleinsten negativen Gewicht $-g$ und addiert nun auf jede Kante diesen Wert g hinzu. Nun wendet Hein Dijkstras Algorithmus an.

Geben Sie einen Graph und eine Gewichtung ohne negativ gewichtete Kreise an, bei der Hein Blöds Algorithmus scheitert.

Aufgabe 4.1.19: 8
Wieviele Pfade (ohne mehrfach besuchte Knoten) können zwischen zwei Knoten in einem Graph mit n Knoten maximal vorkommen.

Aufgabe 4.1.20: 5
Geben Sie einen Algorithmus an, der alle Pfade zwischen zwei Knoten eines Graphen ausgibt.

4.1.2 Der Algorithmus von Bellman und Ford

löst das Problem der kürzesten Wege auch für negative Kantengewichte, wenn kein Kreis mit negativen Gewicht existiert:

+

Bellman-Ford
input: $G = (V, E)$, $s, g \in V$, $w : E \to \mathbb{R}_0^+$
{ **Init-Single-Source**(G, w)
  loop $|V| - 1$ **times**
    for all $(u, v) \in E$ **do**
      Relax(u, v)
  for all $(u, v) \in E$ **do**
    if $d(v) > d(u) + w(u, v)$ **then return false**
  return true
}

Dieser Algorithmus gibt **true** genau dann aus, wenn keine Kreis mit negativen Kantengewicht erreichbar ist.

Theorem 11 *Der Bellman-Ford-Algorithmus hat eine Laufzeit von* $O(|V| \cdot |E|)$

Aufgabe 4.1.21: 2
Beweisen Sie das Theorem.

Damit ist die Laufzeit selbst bei Graphen mit geringer Kantenanzahl wesentlich höher als die des Algorithmus von Dijkstra.

Korrektheit

Lemma 10 *Wenn ein von s erreichbarer negativ gewichteter Kreis in G vorhanden ist, gibt der Bellman-Ford-Algorithmus* **false** *aus.*

Beweis: Sei $k = (u_1, \ldots, u_\ell, u_1)$ ein Kreis aus G mit $w(k) < 0$. Ferner sei $p = (v_1 = s, v_2, \ldots, v_m = u_1)$ ein Pfad von $s \overset{p}{\rightsquigarrow} u_1$, wobei o.B.d.A. $\{v_1, \ldots, v_m\} \cap \{u_1, \ldots, u_\ell\} = \{u_1\}$.
Nach dem i-ten **loop**-Schleifendurchlauf gilt $d(v_{i+1}) \in \mathbb{Z}$. Nach dem $i + m - 1$-ten **loop**-Schleifendurchlauf gilt $d(u_{i+1}) \in \mathbb{Z}$. (Beweis jeweils durch vollständige Induktion). Demnach sind nach $|V| - 1$ **loop**-Schleifendurchläufen alle d-Einträge auf den Knoten des Kreises k endlich.
Es gilt: $\sum_{i=1}^{\ell} w(u_i, u_{i+1)} < 0$, da k ein negativ gewichteter Kreis ist.
Angenommen nach Beendigung des Programms gelte für alle i: $d(u_{i+1}) \leq d(u_i) + w(u_i, u_{i+1})$.
Dann folgt daraus

$$\begin{aligned} \sum_{i=1}^{\ell} d(u_i) &\leq \sum_{i=1}^{\ell} d(u_i) + \sum_{i=1}^{\ell} w(u_i, u_{i+1}) \\ \Longrightarrow \quad \sum_{i=1}^{\ell} w(u_i, u_{i+1)} &\geq 0 \end{aligned}$$

∎

Lemma 11 *Wenn kein negativ gewichteter Kreis von s in G erreichbar ist, gibt der Bellman-Ford-Algorithmus* **true** *aus und berechnet für alle Knoten v:*

$$d(v) = \delta(s, v) \ .$$

+

Beweis: Für einen Knoten $v \in V$ sei $p = (s = u_1, u_2, \ldots, u_\ell = v)$ ein kürzester Weg $s \overset{p}{\rightsquigarrow} v$. Dann gilt nach Lemma 9

$$\delta(s, u_j) = \sum_{i=1}^{j-1} w(u_i, u_{i+1}) \ .$$

Durch vollständige Induktion beweisen wir jetzt: $\delta(s, u_j) = d(u_j)$ nach dem $j-1$-ten **loop**-Schleifendurchlauf:

Induktionsverankerung: Vor der ersten **loop**-Schleife gilt $d(s) = 0$.

Induktionsschritt: Nach dem $j-1$-ten **loop**-Schleifendurchlauf gilt $\delta(s, u_j) = d(u_j)$.

In der inneren Schleife wird u.a. **Relax**(u_i, u_{i+1}) aufgerufen. Nach dieser Operation gilt

$$\begin{aligned} d(u_{i+1}) &\leq d(u_i) + w(u_i, u_{i+1}) \\ &= \delta(s, u_{i+1}) \end{aligned}$$

Natürlich kann $d(u_{i+1})$ nicht kleiner als $\delta(s, u_{i+1})$ sein, da sonst p nicht minimal war.

Wir verifizieren nun, das der Algorithmus auch noch **true** ausgibt: Für alle Kanten $(u, v) \in E$ gilt

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

Somit gilt $d(v) \leq d(u) + w(u, v)$ und der Algorithmus gibt **true** aus. ∎

Aufgabe 4.1.22: 5
Modifizieren Sie den Algorithmus so, daß $d(v) = -\infty$ gesetzt wird, falls ein negativ gewichteter Kreis auf dem Weg zu v existiert.

Aufgabe 4.1.23: 5
Finden Sie einen Graphen der fehlerhaft vom Bellman-Ford-Algorithmus mit nur $|V| - 2$ **loop**-Schleifendurchläufen bearbeitet wird.

4.2 Kürzeste Wege zwischen allen Paaren

Statt des kürzesten Weges ausgehend von einem Knoten wollen wir nun eine Matrix bestimmen, die die kürzesten Entfernungen zwischen allen Knoten-Paaren darstellt.

Kürzeste Wege zwischen allen Paaren *(all pair shortest paths)*
Geg.: Graph $G = (V, E)$, Matrix $W = (w(u, v))_{u,v \in V}$
Ges.: Matrix $\delta = (\delta(u, v))_{u,v \in V}$

Natürlich kann man dieses Problem auf das Problem kürzester Wege von einer Quelle reduzieren. Dieser Ansatz ergibt aber nur eine Laufzeit von $O(|V|^4)$.

Aufgabe 4.2.24: 4
Warum?

+

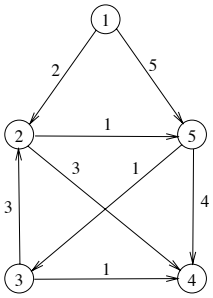

Abbildung 4.4: Noch ein Graph.

4.2.1 Kürzeste Wege mit höchstens m-mal Umsteigen

Die Matrix $D^{(1)} = (d_{ij}^{(1)})$ beschreibt den kürzestens Weg zwischen allen Paaren über höchstens eine Kante:

$$d_{ij}^{(1)} = \begin{cases} 0\,, & \text{falls } i = j \\ w(v_i, v_j)\,, & \text{falls } (v_i, v_j) \in E \\ \infty\,, & \text{sonst} \end{cases}$$

Diese Matrix entspricht der Gewichtsmatrix W, wobei alle Werte auf der Diagonalen ersetzt werden durch $d_{ii}^{(1)} = 0$. $D^{(k)}$ bezeichnet nun die Länge des kürzesten Pfads zwischen allen Knoten, der höchstens k Kanten besitzt.

Init-D
input: $W = (w(v_i, v_j))_{1 \le i \le n, 1 \le j \le n}$
{ $D^{(1)} \leftarrow W$
for $i \leftarrow 1$ **to** n **do** $D^{(1)}(i,i) \leftarrow 0$
return $D^{(1)}$
}

Beispiel 4.2.5: Die zu Abb. 4.4 zugehörige Gewichtsmatrix ist

$$W = \begin{pmatrix} \infty & 2 & \infty & \infty & 5 \\ \infty & \infty & \infty & 3 & 1 \\ \infty & 3 & \infty & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 1 & 4 & \infty \end{pmatrix}$$

Daraus bestimmen sich die Matrizen $D^{(1)}$ und $D^{(2)}$ als

$$D^{(1)} = \begin{pmatrix} 0 & 2 & \infty & \infty & 5 \\ \infty & 0 & \infty & 3 & 1 \\ \infty & 3 & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 4 & 0 \end{pmatrix} \qquad D^{(2)} = \begin{pmatrix} 0 & 2 & 6 & 5 & 3 \\ \infty & 0 & 2 & 3 & 1 \\ \infty & 3 & 0 & 1 & 4 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 4 & 1 & 2 & 0 \end{pmatrix}$$

+

Man erkennnt leicht, daß sich $D^{(m)}$ aus $D^{(m-1)}$ wie folgt errechnet:

$$\begin{aligned} d_{ij}^{(m)} &= \min\left(d_{ij}^{(m-1)}, \min_{1\le k\le n}\left\{d_{ik}^{(m-1)} + w(v_k, v_j)\right\}\right) \\ &= \min_{1\le k\le n}\left\{d_{ik}^{(m-1)} + d_{kj}^{(1)}\right\} \end{aligned}$$

Der folgende Algorithmus führt diese Berechnung auf der gesamten Matrix aus. Hier sei $A = D^{(m-1)}$, $B = D^{(1)}$ und $C = D^{(m)}$

```
Extend-Shortest-Path
input: A = (a_ij)_{1≤i≤n,1≤j≤n}
       B = (b_ij)_{1≤i≤n,1≤j≤n}
{ for i ← 1 to n do
      for j ← 1 to n do
          { c_ij ← ∞
            for k ← 1 to n do
                c_ij ← min(c_ij, a_ik + b_kj)
          }
  return C = (c_ij)_{1≤i≤n,1≤j≤n}
}
```

Aufgabe 4.2.25: 6
Wann verändert sich der Wert auf der Diagonalen, was für Auswirkungen hat es und wie kann man dies ausnutzen?

Offensichtlich hat dieser Algorithmus starke Ähnlichkeit mit dem Standardalgorithmus zur Multiplikation zweier Matrizen:

```
Matrix-Multiplikation
input: A = (a_ij)_{1≤i≤n,1≤j≤n}
       B = (b_ij)_{1≤i≤n,1≤j≤n}
{ for i ← 1 to n do
      for j ← 1 to n do
          { c_ij ← 0
            for k ← 1 to n do
                c_ij ← c_ij + a_ik b_kj
          }
  return C = (c_ij)_{1≤i≤n,1≤j≤n}
}
```

Folgender Algorithmus berechnet nun die Lösung dieses Problems, indem sukzessive immer mehr Umsteigestellen, d.h. immer kürzere Pfade mit mehr Knoten, berücksichtigt werden. Wenn $A \times B$ die Operation von **Extend-Shortest-Path**(A, B) beschreibt berechnet, Algorithmus **Silly-All-Pair-Shortest-Path** also:

$$(\cdots((D^{(1)} \underbrace{\times D^{(1)}) \times D^{(1)}) \cdots \times D^{(1)}}_{m\text{-mal}}) \; .$$

+

```
Silly-All-Pairs-Shortest-Path
input: W = (w(v_{i,j}))_{1≤i≤n,1≤j≤n}
{ D^(1) ← Init-D(W)
  for m ← 2 to n − 1 do
      D^(m) ←Extend-Shortest-Paths(D^(m−1), D^(1))
  return D^(n)
}
```

Effizienter ist der Algorithmus **All-Pairs-Shortest-Path.** Er berechnet den Wert $B^{(n)}$ durch iteriertes Quadrieren: $B^{(2m)} = B^{(m)} \times B^{(m)}$.

```
All-Pairs-Shortest-Path
input: W = (w(v_{i,j}))_{1≤i≤n,1≤j≤n}
{ D^(1) ← Init-D(W)
  m ← 1
  while m ≤ n do
        { D^(2m) ← Extend-Shortest-Paths(D^(m), D^(m))
          m ← 2m
        }
  return D^(m)
}
```

Theorem 12 *Der Algorithmus* **All-Pairs-Shortest-Path** *löst das Problem der kürzesten Wege zwischen allen Paaren in Zeit* $O(|V|^3 \log |V|)$.

Aufgabe 4.2.26: 3
Vergleichen Sie diese Laufzeit mit einem Algorithmus, der das Verfahren von Dijkstra $|V|$-mal verwendet. Welches Verfahren ist wann vorzuziehen?

4.2.2 Der Floyd-Warshall-Algorithmus

Die Matrix $D^{[k]}$ gibt die Matrix des Gewichts der kürzesten Pfade an, die nur als innere Knoten (Umsteigestellen) nur die ersten k Knoten $v_1, \ldots, v_k$ beinhalten dürfen.
Nehmen wir an, wir hätten die Matrix $D^{[k]}$ der kürzesten Entfernungen für einen Graphen $G = (V, E)$ schon berechnet. Nun wollen wir den nächsten Knoten v_{k+1} als inneren Knoten berücksichtigen. Wie kann man jetzt die entsprechende Matrix $D^{[k+1]}$ berechnen? Für zwei Knoten v_i, v_j gibt es nur zwei Möglichkeiten:

1. Unter Berücksichtigung der ersten $k+1$ Knoten als innere Knoten eines Pfades findet sich kein kürzerer Weg als unter Berücksichtigung der ersten k.

2. Der Knoten v_{k+1} kommt im jetzt kürzeren Pfad $v_i \overset{p}{\rightsquigarrow} v_j$ mit $p = (v_i, \ldots, v_{k+1}, \ldots, v_j\}$ zwischen v_i und v_j genau einmal vor. Da aber alle Knoten des Pfades p aus der Menge $\{v_1, \ldots, v_k, v_{k+1}\}$ sind, kann man die Entfernungen der Pfade $v_i \overset{p_1}{\rightsquigarrow} v_{k+1}$ und $v_{k+1} \overset{p_2}{\rightsquigarrow} v_j$ in der Matrix $D^{[k]}$ finden.

+

Damit ergibt sich für $D^{[0]} = D^{(1)}$ folgende Rekursionsgleichung zur Berechnung von $D^{[k+1]}$ ($k \geq 0$):

$$d_{ij}^{[k+1]} = \min\left(d_{ij}^{[k]}, d_{ik}^{[k]} + d_{kj}^{[k]}\right) .$$

Der entsprechende Algorithmus lautet:

Floyd-Warshall
input: $W = (w(v_i, v_j))_{1 \leq i \leq n, 1 \leq j \leq n}$
{ $D^{[0]} \leftarrow$ **Init-D**(W)
  for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
      for $j \leftarrow 1$ **to** n **do**
        $d_{ij}^{[k]} \leftarrow \min\left(d_{ij}^{[k-1]}, d_{ik}^{[k-1]} + d_{kj}^{[k-1]}\right)$
  return $\left(d_{ij}^{[n]}\right)_{1 \leq i \leq n, 1 \leq j \leq n}$
}

Theorem 13 *Der Floyd-Warshall-Algorithmus löst das Problem der kürzesten Wege in Zeit* $O(|V|^3)$.

Beispiel 4.2.6: Der Floyd-Warshall-Algorithmus für den Graphen aus Abb. 4.4.

$$D^{[0]} = \begin{pmatrix} 0 & 2 & \infty & \infty & 5 \\ \infty & 0 & \infty & 3 & 1 \\ \infty & 3 & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 4 & 0 \end{pmatrix} \qquad D^{[1]} = \begin{pmatrix} 0 & 2 & \infty & \infty & 5 \\ \infty & 0 & \infty & 3 & 1 \\ \infty & 3 & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 4 & 0 \end{pmatrix}$$

$$D^{[2]} = \begin{pmatrix} 0 & 2 & \infty & \mathbf{5} & \mathbf{3} \\ \infty & 0 & \infty & \mathbf{3} & 1 \\ \infty & 3 & 0 & 1 & \mathbf{4} \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & 4 & 0 \end{pmatrix} \qquad D^{[3]} = \begin{pmatrix} 0 & 2 & \infty & 5 & 3 \\ \infty & 0 & \infty & 3 & 1 \\ \infty & 3 & 0 & 1 & 4 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \mathbf{4} & 1 & \mathbf{2} & 0 \end{pmatrix}$$

$$D^{[4]} = \begin{pmatrix} 0 & 2 & \infty & 5 & 3 \\ \infty & 0 & \infty & 3 & 1 \\ \infty & 3 & 0 & 1 & 4 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 4 & 1 & 2 & 0 \end{pmatrix} \qquad D^{[5]} = \begin{pmatrix} 0 & 2 & \mathbf{4} & 5 & 3 \\ \infty & 0 & \mathbf{2} & 3 & 1 \\ \infty & 3 & 0 & 1 & 4 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 4 & 1 & 2 & 0 \end{pmatrix}$$

Aufgabe 4.2.27: 6
Modifizieren Sie den Algorithmus von Floyd und Warshall so, daß er den transitiven Abschluß eines Graphen berechnet. Der resultierende Algorithmus soll dabei nur Matrizen mit Booleschen Werten benutzen!

Für die Lösungsmatrix T des transitiven Abschlußes gilt $T(i,j) = 1$, wenn ein Weg in G von Knoten v_i zu v_j existiert. Ansonsten ist $T(i,j) = 0$.

Aufgabe 4.2.28: 5
Konstruieren Sie einen Algorithmus, der aus der Matrix $D^{[n]}$ für zwei gegebene Knoten den kürzesten Weg berechnet. Welche Laufzeit hat Ihr Verfahren?

+

Aufgabe 4.2.29: 8
Konstruieren Sie einen Algorithmus, der für jeden Knoten den kürzesten Kreis berechnet, in dem der Knoten enthalten ist. Welche Laufzeit hat Ihr Verfahren?

Aufgabe 4.2.30: 9
Geben Sie für einen gegebenen Graphen einen Algorithmus an, der für zwei gegebene Knoten das Problem des längsten Weges löst. Der längste Weg zwischen zwei Knoten ist ein Pfad p ohne Mehrfachvorkommen eines Knoten mit maximalem Gewicht $w(p)$. Welche Laufzeit hat Ihr Verfahren?

Kapitel 5

Rechnen mit Matrizen

5.1 Strassens Algorithmus für Matrixmultiplikation

Wir haben bereits einen einfachen Algorithmus für Matrixmultiplikation in Zeit $O(n^3)$ für $n \times n$-Matrizen kennengelernt. Für sehr große Matrizen kann diese Aufgabe effizienter gelöst werden. Gegeben seien zwei $n \times n$ Matrizen A, B für gerades n. Gesucht ist das Produkt $C = AB$. Die $(n/2) \times (n/2)$-Teilmatrizen der Matrix A und B werden mit a, b, c, d, e, f, g, h wie folgt bezeichnet:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \qquad B = \begin{pmatrix} e & g \\ f & h \end{pmatrix}.$$

Das Produkt $C = AB$ habe die $(n/2) \times (n/2)$-Teilmatrizen r, s, t, u:

$$A \cdot B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} r & s \\ t & u \end{pmatrix} = C.$$

Damit gilt

$$\begin{aligned} r &= ae + bf, \\ s &= ag + bh, \\ t &= ce + df, \\ u &= cg + dh. \end{aligned}$$

Sei $T(n)$ der Aufwand, um 2 $n \times n$-Matrizen zu multiplizieren. Dann hätte ein Verfahren, das auf den letzten vier Gleichungen basiert und diese rekursiv via *divide and conquer* anwendet, die Laufzeit:

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2),$$

was $T(n) = O(n^3)$ implizieren würde und keine Verbesserung wäre. Der Algorithmus von Strassen kommt aber mit nur 7 Matrixmultiplikationen von Teilmatrizen der Größe $(n/2) \times (n/2)$ aus und erreicht damit eine asymptotisch geringere Laufzeit:

Hierzu werden die Matrizen $A_1, B_1, \ldots, A_7, B_7$ wie folgt berechnet:

$$\begin{array}{lcllcll} A_1 & = & a & , & B_1 & = & g-h \quad , \\ A_2 & = & a+b & , & B_2 & = & h \quad , \\ A_3 & = & c+d & , & B_3 & = & e \quad , \\ A_4 & = & d & , & B_4 & = & f-e \quad , \\ A_5 & = & a+d & , & B_5 & = & e+h \quad , \\ A_6 & = & b-d & , & B_6 & = & f+h \quad , \\ A_7 & = & a-c & , & B_7 & = & e+g \quad . \end{array}$$

Nun berechnet man rekursiv nach Strassens Verfahren die 7 Matrixprodukte $P_1, \ldots, P_7$ dieser Teilmatrizen aus:

$$\forall i \in \{1, \ldots, 7\} \qquad P_i \;=\; A_i \cdot B_i \;.$$

Die gesuchten Teilmatrizen r, s, t, u ergeben sich nun aus

$$\begin{array}{rcl} r & = & P_5 + P_4 - P_2 + P_6 \;, \\ s & = & P_1 + P_2 \;, \\ t & = & P_3 + P_4 \;, \\ u & = & P_5 + P_1 - P_3 - P_7 \;. \end{array}$$

Theorem 14 *Der Matrixmultiplikationsalgorithmus von Strassen hat auf Eingabe zweier $n \times n$-Matrizen Laufzeit $O(n^{\log_2 7})$.*

Beweis: Jede der Additionen und Subtraktionen der Teilmatrizen kostet Aufwand $O(n^2)$. Damit gibt sich folgende Rekursionsgleichung für die Laufzeit dieses Algorithmus

$$T(n) \;\leq\; 7 \cdot T(n/2) + c \cdot n^2 \;,$$

für eine geeignet gewählte $c \geq 0$. Wenn n keine Zweierpotenz ist, so ergänzen wir die Matrizen durch Anfügen von Zeilen und Spalten mit 0-Einträgen zu einer $n' \times n'$-Matrix, wobei n' die nächstgrößere Zweierpotenz ist.
Die Laufzeit von $n = 2^k$ ergibt sich also aus

$$\begin{array}{rcl} T(2^k) & \leq & 7 \cdot T(2^{k-1}) + cn^2 \\ & \leq & 7^2 \cdot T(2^{k-2}) + 7^2 \cdot T(2^{k-2}) + cn^2(1 + \frac{7}{4}) \\ & \leq & 7^3 \cdot T(2^{k-3}) + 7^3 \cdot T(2^{k-3}) + cn^2(1 + \frac{7}{4} + \frac{49}{16}) \\ & & \ldots \\ & \leq & 7^k \cdot T(1) + cn^2(1 + \frac{7}{4} + \frac{7^2}{4^2} + \frac{7^3}{4^3} + \ldots + \frac{7^{k-1}}{4^{k-1}}) \\ & = & 7^{\log_2 n} + 2cn^2 \sum_{i=0}^{k-1} (\frac{7}{4})^i \\ & = & n^{\log_2 7} + 2cn^2 \dfrac{(\frac{7}{4})^k - 1}{\frac{7}{4} - 1} \\ & \leq & n^{\log_2 7} + \frac{8}{3} cn^2 (\frac{7}{4})^{\log_2 n} \\ & = & n^{\log_2 7} + \frac{8}{3} cn^2 n^{\log_2 7 - 2} \\ & = & (\frac{8}{3} c + 1) n^{\log_2 7} \end{array}$$

Somit ist $T(n) = O(n^{\log_2 7})$. ∎

Aufgabe 5.1.31: 3
Beweisen Sie die Richtigkeit von Strassens Verfahren.

Aufgabe 5.1.32: 6
Bestimmen Sie die Konstante c von Strassens Algorithmus möglichst genau! Für welche n ist Strassens Algorithmus schneller als das Standardverfahren?

Aufgabe 5.1.33: 8
Gegeben sei für Konstanten $\alpha, \beta < 1$ und $\alpha, \beta, \delta > 0$ die Rekursionsgleichung

$$S(n) \leq \alpha S(\lfloor \beta \cdot n \rfloor) + O(n^\delta) \ .$$

In welcher Größenordnung wächst S (d.h. $S \in O(?)$) siehe [RR 90]?

5.2 Berechnung der Umkehrmatrix

Wir betrachten im folgenden quadratische $n \times n$-Matrizen mit reellen Zahleneinträgen.

Geg.: Nichtsinguläre $n \times n$ Matrix A.
Ges.: Umkehrmatrix A^{-1} mit $A \cdot A^{-1} = E_n$.

E_n ist hierbei die $n \times n$ Einheitsmatrix $\begin{pmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{pmatrix}$. Eine Matrix ist **singulär**, falls sie keine Umkehrmatrix besitzt (und sobald eine Umkehrmatrix existiert, ist diese eindeutig). Die Berechnung der Umkehrmatrix ist mindestens so schwierig wie die Multiplikation zweier Matrizen, was folgendes Theorem zeigt.

Theorem 15 *Wenn ein Algorithmus eine $n \times n$-Matrix in Zeit $I(n)$ invertieren kann, so können zwei $n \times n$-Matrizen in Zeit $O(I(3n))$ multipliziert werden.*

Beweis: Seien die Matrizen A und B gegeben. Hieraus ergibt sich die $(3n) \times (3n)$-Matrix D wie folgt:

$$D = \begin{pmatrix} E_n & A & 0 \\ 0 & E_n & B \\ 0 & 0 & E_n \end{pmatrix} .$$

Die Umkehrmatrix D^{-1} ergibt sich dann als

$$D^{-1} = \begin{pmatrix} E_n & -A & \boldsymbol{A \cdot B} \\ 0 & E_n & -B \\ 0 & 0 & E_n \end{pmatrix} .$$

∎

Aufgabe 5.2.34: 1
Beweisen Sie die Korrektheit der Umkehrmatrix.

Aufgabe 5.2.35: 8
Finden Sie Funktionen f, für die gilt $f(3n) \notin O(f)$. Kann so eine Funktion auch die zusätzliche Eigenschaft $f \in O(n^3)$ erfüllen?

+

Wir werden im folgenden zeigen, daß die Berechnung der Umkehrmatrix nicht schwieriger ist als die Multiplikation zweier Matrizen. Hierfür benötigen wir aber noch einige Begriffe aus der linearen Algebra.

Definition 2 *Die* **Transposition A^T** *einer $m \times n$ Matrix $A = (a_{ij})_{1 \le i \le n, 1 \le j \le m}$ ist die an der Diagonalen gespiegelte $n \times m$ Matrix $A^T = (a_{ji})_{1 \le j \le m, 1 \le i \le n}$. Eine Matrix A ist* **symmetrisch,** *gdw. $A = A^T$. Eine Matrix A ist* **positiv definit,** *gdw.*

$$\forall x \neq 0 \; : \quad x^T \cdot A \cdot x > 0 \; .$$

Folgende Eigenschaften symmetrischer positiv definiter Matrizen werden wir später benötigen:

Theorem 16 *Es gilt für jede symmetrische positiv definite Matrix*

$$A \; = \; \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}$$

1. *A ist nicht singulär.*
2. *Die $k \times k$-Teilmatrix A_k ist positiv definit.*
3. *Das* **Schur-Komplement** *$S = C - BA_k^{-1}B^T$ ist symmetrisch und positiv definit.*

Beweis:

1. Wir setzen folgendes voraus:

   Eine Matrix ist genau dann singulär, wenn sich ein Teilvektor der Matrix als Linearkombination der anderen Teilvektoren darstellen läßt.

   Wäre A singulär, dann gäbe es einen Vektor $x \neq 0$, so daß $A \cdot x = 0$. Damit würde aber auch $x^T \cdot A \cdot x = 0$ gelten. Somit wäre A nicht positiv definit.

2. Wir betrachten einen beliebigen k-stelligen Vektor x:

$$\begin{aligned} x^T \cdot A_k \cdot x &= (x^T 0) \cdot \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix} \cdot \begin{pmatrix} x \\ 0 \end{pmatrix} \\ &= (x^T 0) \cdot A \cdot \begin{pmatrix} x \\ 0 \end{pmatrix} \; > \; 0 \; , \end{aligned}$$

   da A positiv definit ist.

3. Die Symmetrie von S zu zeigen, sei dem Leser zur Übung überlassen. Für einen Vektor x seien y der k-stellige Teilvektor und z der Restvektor: $x^T = (y^T z^T)$. Damit gilt

$$\begin{aligned} x^T \cdot A \cdot x &= (y^T z^T) \cdot \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix} \cdot \begin{pmatrix} y \\ z \end{pmatrix} \\ &= y^T A_k y + y^T B^T z + z^T B y + z^T C z \\ &= (y + A_k^{-1} B^T z)^T A_k (y + A_k^{-1} B^T z) + z^T (C - B A_k^{-1} B^T) z \; . \end{aligned}$$

   Für ein $z \neq 0$ wählen wir nun $y = -A_k^{-1} B^T z$, was den linken Term verschwinden und folgenden Term stehen läßt:

$$z^T (C - B A_k^{-1} B^T) z = z^T S z \; .$$

   Da $z \neq 0$ keiner Einschränkung unterliegt, ist also S positiv definit.

Theorem 17 *Für jede nicht singuläre Matrix A ist $A^T \cdot A$ positiv definit und symmetrisch.*

Beweis: Die Symmetrie ist leicht einzusehen. Für die positive Definitheit sei $x \neq 0$. Dann gilt

$$x^T(A^TA)x \;=\; (Ax)^T(Ax) \;=\; ||Ax||^2 \;\geq\; 0\,.$$

Beachten Sie, daß $||Ax||^2$ die Summe aller Quadrate des Vektors Ax ist. Wäre $Ax = 0$, dann wäre A singulär, da x eine Linearkombination der Teilvektoren von A definieren würde, deren Ergebnis 0 ist. ∎

Theorem 18 *Falls zwei $n \times n$ Matrizen in Zeit $M(n)$ multipliziert werden können, kann das Inverse einer $n \times n$-Matrix in Zeit $O(M(2n))$ berechnet werden.*

Beweis: Sei $A' = A^TA$. Damit ist A' positiv definit und symmetrisch. Nun sei

$$A' \;=\; \begin{pmatrix} B & C^T \\ C & D \end{pmatrix}\,.$$

Sei $S = D - CB^{-1}C^T$ das Schur-Komplement von A'. Dann gilt

$$A'^{-1} \;=\; \begin{pmatrix} B^{-1} + B^{-1}C^TS^{-1}CB^{-1} & -B^{-1}C^TS^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}\,.$$

B^{-1} und S^{-1} existieren, wie zuvor schon bewiesen. Ferner gilt:

$$\begin{aligned} B^{-1}C^T &= (CB^{-1})^T \\ B^{-1}C^TS^{-1} &= (S^{-1}CB^{-1})^T \end{aligned}$$

Damit müssen folgende vier Multiplikationen von Teilmatrizen der Größe $(n/2)\times(n/2)$ ausgeführt werden:

$$\begin{aligned} C &\cdot B^{-1} \\ (CB^{-1}) &\cdot C^T \\ S^{-1} &\cdot (CB^{-1}) \\ (CB^{-1})^T) &\cdot (S^{-1}CB^{-1}) \end{aligned}$$

Die Berechnung der Umkehrmatrizen B^{-1}, S^{-1} erfolgt rekursiv nach dem gleichen Verfahren (Auf den ersten Schritt mit der Multiplikation der transponierten Matrix kann aber verzichtet werden).
Die Laufzeit $I_r(n)$ der Rekursion ergibt sich also für $n = 2^k$ als

$$\begin{aligned} I_r(n) &\leq 2 \cdot I(n/2) + 4 \cdot M(n) + O(n^2) \\ &= 2 \cdot I(n/2) + O(M(n)) \\ &= O(M(n))\,. \end{aligned}$$

Ist n nicht als Zweierpotenz darstellbar, so wählt man die nächstgrößere Zweierpotenz (ist höchstens doppelt so groß wie n). Die $(2^k) \times (2^k)$-Matrix A' ergänzt man durch Eintragen von Einsen auf der Diagonalen, die anderen Positionen werden mit Null ergänzt.
Die gesuchte Matrix A^{-1} ergibt sich aus $A'^{-1} \cdot A^T$, da

$$(A'^{-1} \cdot A^T) \cdot A \;=\; ((A^T \cdot A)^{-1} \cdot A^T) \cdot A \;=\; (A^TA)^{-1}(A^TA) \;=\; E_n$$

Die gesamte Laufzeit $I(n)$ ist daher $I(n) \leq I_r(2n) + M(n) \leq O(M(2n))$. ∎

+

Aufgabe 5.2.36: 5
Beweisen Sie mittels einer Matrixmultiplikation, daß das Inverse von A' im obigen Beweis richtig angegeben wurde.

Hieraus ergibt sich direkt ein Algorithmus, der die Umkehrmatrix mit Hilfe von Strassens Matrixmuliplikationsalgorithmus in Laufzeit $O(n^{\log_2 7})$ berechnet. Bis heute ist es offen wie schnell man Matrizen multiplizieren kann. Der asymptotisch schnellste bekannte Algorithmus ist von Coppersmith und Winograd [CW 87] und hat eine Laufzeit von $O(n^{2.376})$.

Kapitel 6

Die diskrete Fouriertransformation

Wir betrachten im folgenden **Polynome**

$$A(x) \;=\; \sum_{j=0}^{n} a_j x^j \;,$$

wobei $a_0, \ldots, a_n$ die **Koeffizienten** und n der **Grad** des Polynoms ist. Polynome sind gegen Summe $A(x)+B(x)$ und Produkt $A(x)\cdot B(x)$ abgeschlossen, wobei sich beim Produkt der Grad verdoppelt.
Wenn die Addition zweier Koeffizienten in konstanter Zeit[1] möglich ist, können Polynome in linearer Zeit addiert werden. Wir werden uns im folgenden mit der Frage beschäftigen, wieviel Zeit man benötigt um zwei Polynome zu multiplizieren.

Produkt zweier Polynome
Geg.: zwei Polynome A, B vom Grade n.
Ges.: das Produkt $A(x)\cdot B(x) \;=\; \sum_{j=0}^{n} a_j x^j \;\cdot \sum_{j=0}^{n} b_j x^j$.

Bezeichne $C(x) = A(x)\cdot B(x)$, wobei

$$C(x) \;=\; \sum_{j=0}^{2n} c_j x^j \;.$$

Dann gilt für die Koeffizienten c_j:

$$c_j \;=\; \sum_{k=0}^{j} a_k \cdot b_{j-k} \;,$$

wenn wir die Koeffizienten höheren Grades als $a_{n+1} = b_{n+1} = a_{n+2} = \cdots = 0$ definieren.

Aufgabe 6.0.37: 2
Stellen Sie das Produkt zweier Polynome A, B als Multiplikation zweier Matrizen $A'B'$ dar, wobei A' aus A und B' aus B gewonnen wurde. Welchen Zeitbedarf hat das Matrixprodukt?

[1] Wenn die Wortbreite der RAM also ausreicht.

6.1 Polynomdarstellung durch Stützstellen

Theorem 19 *Für unterschiedliche Zahlen $x_0, \ldots, x_n$ und beliebige Zahlen $y_0, \ldots, y_n$ gibt es immer genau ein Polynom $A(x)$ vom Grad n, so daß für alle $i \in \{0, \ldots, n\}$ gilt:*

$$A(x_i) \;=\; y_i \;.$$

Dieser Satz sollte aus der Analysis schon bekannt sein. Die Formel von **Lagrange** liefert das Lösungspolynom

$$A(x) \;=\; \sum_{k=0}^{n} y_k \cdot \frac{\prod_{j \neq k}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)} \;.$$

Die Angabe von mindestens $n+1$ Stützstellen ist also eine andere Möglichkeit ein Polynom vom Grad n zu eindeutig darzustellen. Diese Darstellung hat sogar einige Vorteile. In der Stützstellendarstellung läßt sich die Addition, Subtraktion und sogar Multiplikation zweier Polynome in linearer Zeit berechnen:
Sei $C(x) = A(x) + B(x)$. Dann ergeben sich für die Stützstellen $u_i = A(x_i)$ und $v_i = B(x_i)$ die Stützstellen von C als $(x_i, u_i + v_i)$, d.h. $C(x_i) = A(x_i) + B(x_i)$.
Sei $D(x) = A(x) \cdot B(x)$. Dann ergibt sich die Stützstellendarstellung von D als $(x_i, u_i \cdot v_i)$, d.h. $C(x_i) = A(x_i) + B(x_i)$. Zur eindeutigen Darstellung von D sind hier aber mindestens $2n+1$ Werte $x_0, \ldots x_{2n}$ notwendig, da $D(x)$ den Grad $2n$ besitzen kann.

Aufgabe 6.1.38: 3
Geben Sie einen Algorithmus an, der auf Eingabe eines Polynoms A und einer Zahl x den Wert $A(x)$ in linearer Zeit berechnet.

Aufgabe 6.1.39: 5
Geben Sie einen Algorithmus an, der auf Eingabe von $n+1$ Stützstellen das zugehörige Polynom A bestimmt. Welche Laufzeit hat ihr Verfahren?

6.2 Schnelle Berechnung des Produkts zweier Polynome

Berechnet man das Produkt zweier Polynome direkt, so benötigt man Laufzeit $O(n^2)$. Die Stützstellenrepräsentation des Produkts zweier Polynome läßt sich jedoch aus den entsprechenden Repräsentationen der Polynome in linearer Zeit berechnen. Leider kostet die Umwandlung in dieses Repräsentation und die Rückumwandlung für allgemeine Positionen $x_0, \ldots, x_n$ Laufzeit $O(n^2)$. Für eine spezielle Wahl der Werte $x_0, \ldots, x_n$ ist diese Umformung effizienter möglich. Hierzu verallgemeinern wir das Problem auf komplexe Zahlen.
Eine **komplexe m-te Einheitswurzel** ist eine komplexe Zahl ω mit

$$\omega^m \;=\; 1 \;.$$

Es gibt genau m komplexe m-te Einheitswurzeln (siehe Abb. 6.1):

$$\omega_m^k \;=\; e^{2\pi i k/m} \;=\; \cos(2\pi k/m) \;+\; i \cdot \sin(2\pi k/m)$$

für $k \in \{0, \ldots, m-1\}$, wobei die **Basiswurzel** $\omega_m = e^{2\pi i/m}$ ist. Diese Zahlen auf dem Einheitskreis bilden bezüglich der Multiplikation eine Gruppe. Insbesondere gilt

$$\omega_m^j \cdot \omega_m^k \;=\; \omega_m^{(j+k) \bmod m} \;.$$

+

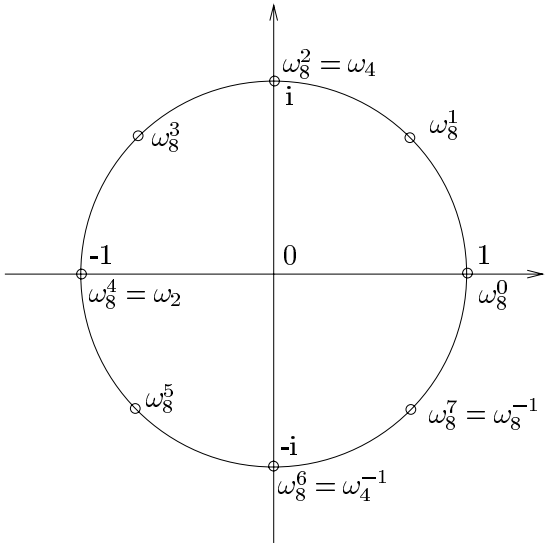

Abbildung 6.1: Die achten Einheitswurzeln.

Der Vektor $y = (y_0, \ldots, y_{n-1})$ wird **diskrete Fouriertransformierte** des Polynoms $A(x) = \sum_{j=0}^{n-1} a_j x^j$ genannt, wenn

$$y_k = A(\omega_n^k) \ .$$

Fast-Pol-Mul
input: $A = (a_0, \ldots, a_n)$
$B = (b_0, \ldots, b_n)$
{ $m \leftarrow \min\{2^k \mid 2^k \geq 2n+1\}$
$(u_0, \ldots, u_{m-1}) \leftarrow \mathbf{FFT}(a_0, \ldots, a_{m-1}) \quad = (A(\omega_m^0), \ldots, A(\omega_m^{m-1}))$
$(v_0, \ldots, v_{m-1}) \leftarrow \mathbf{FFT}(b_0, \ldots, b_{m-1}) \quad = (B(\omega_m^0), \ldots, B(\omega_m^{m-1}))$
for $j \leftarrow 0$ **to** $m-1$ **do** $z_i \leftarrow u_i \cdot v_i$
$(c_0, \ldots, c_{m-1}) \leftarrow \mathbf{inv\text{-}FFT}(z_0, \ldots, z_{m-1})$
return $C = (c_0, \ldots, c_{2n})$
}

Die Funktion **FFT** wandelt ein Polynom in die Fouriertransformierte um, d.h. berechnet die **diskrete Fourier-Transformation.** Die Rückumwandlung wird durch die Funktion **inv-FFT** berechnet.

6.3 Die schnelle Fouriertransformation

Die schnelle Fouriertransformation (*Fast Fourier Transform* — FFT) nutzt die besonderen Eigenschaften der Einheitswurzeln aus und benötigt statt einer Laufzeit von $O(n^2)$ nur $O(n \log n)$. Sei ein Polynom $A(x)$ mit Grad $n = 2^k - 1$ gegeben. Dann definieren wir

$$A^{[0]}(x) \quad = \quad a_0 + a_2 x + \ldots + a_{n-2} x^{n/2-1} \ ,$$

+

$$A^{[1]}(x) \quad = \quad a_1 + a_3 x + \ldots + a_{n-1} x^{n/2-1} \ .$$

Nun gilt: $A(x) \ = \ A^{[0]}(x^2) + xA^{[1]}(x^2)$. Folgender Algorithmus nutzt diese Zerlegung effizient aus:

```
FFT
input: A = (a_0, ..., a_{n-1})
       n = 2^k
{ if n = 1 then return (a_0)
    else {
          y^[0] ← FFT((a_0, a_2, ..., a_{n-2}))
          y^[1] ← FFT((a_1, a_3, ..., a_{n-1}))
          for k ← 0 to n/2 − 1 do
              {y_k       ← y_k^[0] + ω_n^k y_k^[1]
               y_{k+n/2} ← y_k^[0] − ω_n^k y_k^[1]
              }
          return y = (y_0, ..., y_{n-1})
    }
}
```

Lemma 12 *Der Algorithmus* **FFT** *berechnet die diskrete Fouriertransformation korrekt in Zeit* $O(n \log n)$.

Beweis: der Korrektheit durch vollständige Induktion:
Falls der Grad des Polynoms 0 ist, arbeitet der Algorithmus korrekt. Nehmen wir nun an, daß für $n' = n/2$ **FFT** korrekt arbeitet. So erhalten wir für $k \in \{0, \ldots, n/2 - 1\}$ als Ergebnis

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k) = A^{[0]}(\omega_n^{2k}) \ , \\ y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k) = A^{[0]}(\omega_n^{2k}) \ . \end{aligned}$$

Dies folgt aus $\omega_{n/2}^k = e^{4\pi i k/n} = \omega_n^{2k}$. Damit gilt für y_k:

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k \cdot A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) \end{aligned}$$

Für $y_{k+n/2}$ gilt:

$$\begin{aligned} y_{k+n/2} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + e^{\pi i} e^{2\pi i k/n} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+n/2} A^{[1]}(\omega_n^{2k+n}) \\ &= A(\omega_n^{k+n/2}) \end{aligned}$$

Effizienz:
Die Potenzen der Einheitswurzeln werden in einer Tabelle abgelegt. Ihre Berechnung kostet daher einmalig lineare Zeit.

+

Neben dem rekursiven Aufruf wird nur lineare Laufzeit benötigt. Die Laufzeit $T(n)$ des Algorithmus ergibt sich also rekursiv aus

$$T(n) \;=\; 2T(n/2) + O(n) \;=\; O(n \log n)\,.$$

∎

Aufgabe 6.3.40: 4
Geben Sie Imaginärteil und Realteil von ω_{16} durch Wurzeln explizit als reelle Zahlen an (z.B.: $\omega_8 = \frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$.).

Lösung:
Zur Berechnung der 2^k-ten Einheitswurzel kann folgende Formel rekursiv verwendet werden. Sei $\omega_n = x_n + iy_n$. Dann gilt

$$\omega_{2n} = \sqrt{\frac{1+x_n}{2}} + i\,\sqrt{\frac{1-x_n}{2}}\,.$$

Damit ist der Aufwand, um die Einheitswurzeln $\omega_n^0, \ldots, \omega_n^{n-1}$ auf ℓ gültigen Stellen zu berechnen beschränkt durch $2\log n$ Wurzeloperationen (z.B. durch Newtonsche Iterationsverfahren) und n Multiplikationen mit entsprechender gültiger Stellenanzahl $\ell + \log n$. Diese Operationen fallen gegenüber der diskreten Fouriertransformation nicht ins Gewicht.

6.4 Die inverse Fouriertransformation

Die diskrete Fouriertransformation von $(a_0, \ldots, a_{n-1})$ zu $(y_0, \ldots, y_{n-1})$ wird auch durch das Matrixprodukt[2]

$$V_n \cdot a = y$$

dargestellt, wobei

$$V_n \;=\; \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{-2} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{-3} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \omega_n^{-3} & \cdots & \omega_n^1 \end{pmatrix}$$

Für die Umkehrmatrix gilt folgendes Theorem:

Theorem 20

$$V_n^{-1} \;=\; \frac{1}{n} \cdot \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \omega_n^{-3} & \cdots & \omega_n^1 \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \omega_n^{-6} & \cdots & \omega_n^2 \\ 1 & \omega_n^{-3} & \omega_n^{-6} & \omega_n^{-9} & \cdots & \omega_n^3 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^1 & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{-1} \end{pmatrix}$$

[2] V_n heißt für beliebige Basen x statt ω_n Vandermonde-Matrix, daher V_n.

+

Beweis: Wir multiplizieren die Matrizen $V_n \cdot V_n^{-1}$ und erhalten in Zeile j und Spalte k folgenden Eintrag:

$$\sum_{\nu=0}^{n-1} \frac{\omega_n^{\nu j} \cdot \omega_n^{-\nu k}}{n} = \sum_{\nu=0}^{n-1} \frac{\omega_n^{\nu(j-k)}}{n} .$$

1. $j = k$:

$$\sum_{\nu=0}^{n-1} \frac{\omega_n^{\nu(j-k)}}{n} = \sum_{\nu=0}^{n-1} \frac{1}{n} = 1 .$$

2. $j \neq k$:

$$\sum_{\nu=0}^{n-1} \frac{\omega_n^{\nu(j-k)}}{n} = \frac{1}{n} \sum_{\nu=0}^{n-1} (\omega_n^{(j-k)})^{\nu} = \frac{1}{n} \frac{\overbrace{\omega_n^{n(j-k)}}^{=1} - 1}{\omega_n^{j-k} - 1} = 0 .$$

Es gilt also $V_n \cdot V_n^{-1} = E_n$. ∎
Der Algorithmus zur Berechnungen der inversen Fouriertransformierten gleicht dadurch dem **FFT**-Algorithmus.

```
inv-FFT
input: y = (y_0, ..., y_{n-1})
       n = 2^k
{ if n = 1 then return (y_0)
    else {
        a^[0] ← inv-FFT((y_0, y_2, ..., y_{n-2}))
        a^[1] ← inv-FFT((y_1, y_3, ..., y_{n-1}))
        for k ← 0 to n/2 − 1 do
            {a_k       ← (a_k^[0] + ω_n^{-k} a_k^[1])/2
             a_{k+n/2} ← (a_k^[0] − ω_n^{-k} a_k^[1])/2
            }
        return A = (a_0, ..., a_{n-1})
    }
}
```

Aufgabe 6.4.41: 7
Beweisen Sie die Korrektheit von **inv-FFT**.

Da die Laufzeit von **inv-FFT** der von **FFT** entspricht, erhalten wir folgenden Satz:

Theorem 21 *Der Algorithmus* **Fast-Pol-Mul** *berechnet das Produkt zweier Polynome in Zeit* $O(n \log n)$.

Analysiert man die rekursiven Programme genauer, zeigt sich, daß Datenflußgraphen der Größe $O(n \log n)$ und der Tiefe $O(\log n)$ diese Berechnung darstellen können. Die Topologie der Berechnung läßt sich mit sogenannten *Butterfly*-Graphen[3] darstellen. Abb. 6.2 und 6.3 zeigen den Datenflußgraphen für die schnelle Fouriertransformation mit $n = 8$ und $n = 16$.

[3] *butterfly* = Schmetterling

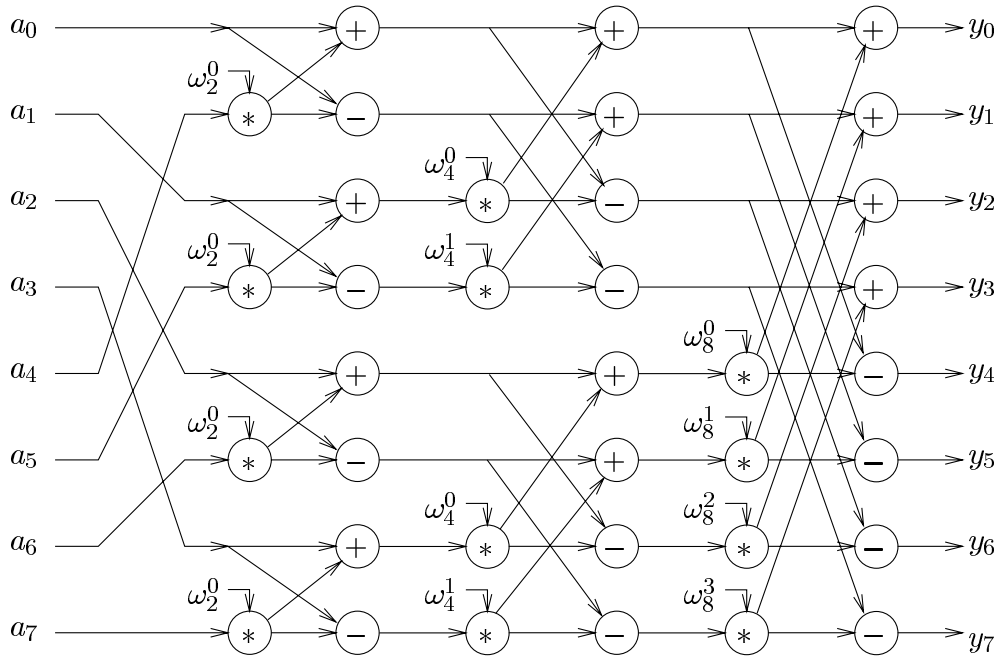

Abbildung 6.2: Die diskrete Fouriertransformation für $n = 8$.

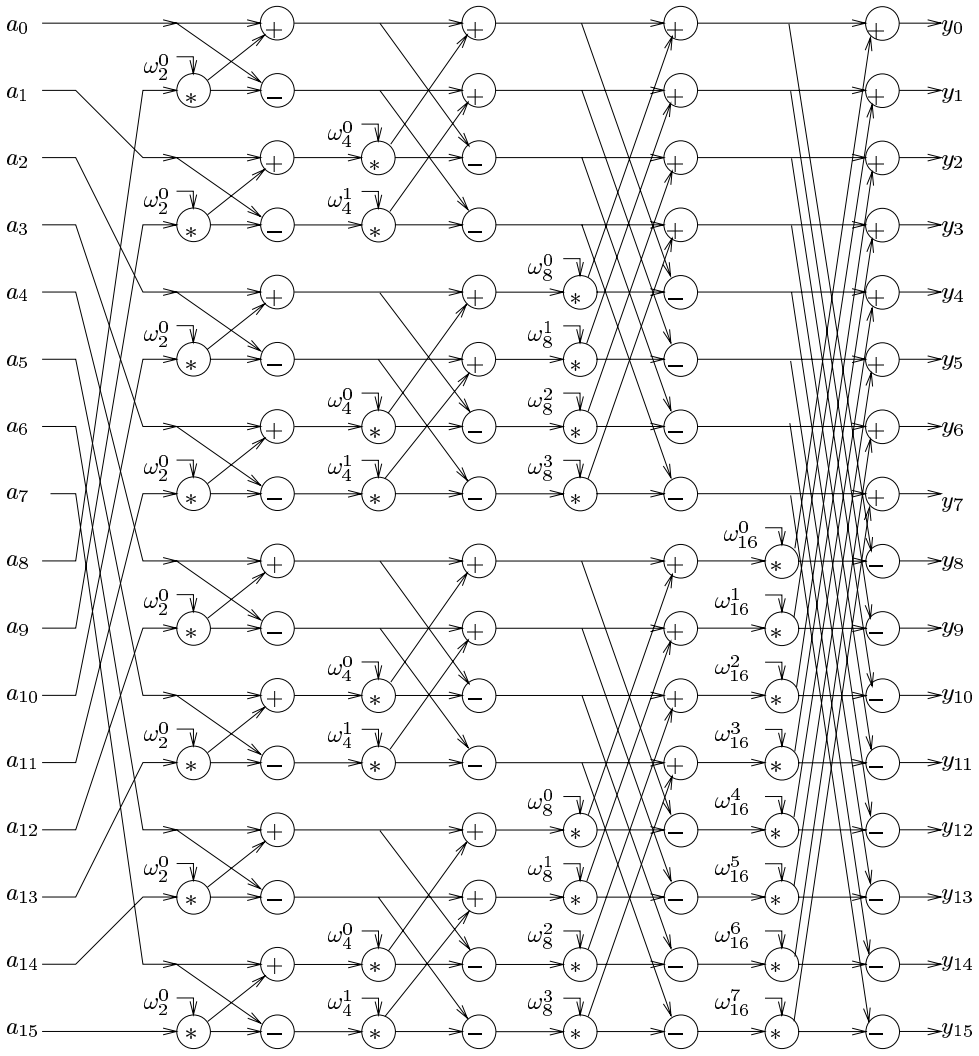

Abbildung 6.3: Die diskrete Fouriertransformation für $n = 16$.

Kapitel 7

Multiplikation und Division

Das Rechenmodell, das wir allen effizienten Algorithmen bis jetzt zugrunde gelegt haben, sieht einen Multiplikationsbefehl für Operanden deren Bitdarstellung durch die Wortbreite beschränkt ist vor. Viele Anwendungen benötigen aber arithmetische Operationen, welche wesentlich größere Zahlen betrachten. So werden beispielsweise in der Kryptographie mindestens 200-(dezimal)-stellige Primzahlen miteinander multizipliert, um einen sicheren Schlüssel für das RSA-Kryptosystem zu erzeugen.

Produkt zweier n-stelliger Zahlen
Geg.: Basis $b = 2^w$ und zwei n-stellige Zahlen $u, v \in \{0, 2^{nw} - 1\}$
Ges.: das Produkt $a \cdot b$ zur Basis 2^w

Aufgabe 7.0.42: 4
Geben Sie einen Algorithmus an der beliebig lange Zahlen in quadratischer Zeit multipliziert.

7.1 Der Algorithmus von Karazuba

Seien $(u_0, \ldots, u_{n-1})$ die Ziffern von u zur Basis b und ebenso $(v_0, \ldots, v_{n-1})$ für v (d.h. $u = \sum_{i=0}^{n-1} u_i b^i$). Wie identifizieren diese Tupeldarstellung mit der eigentlichen Zahl.
Nun definieren wir U_0, U_1, V_0, V_1 wie folgt:

$$\begin{aligned} U_0 &= (u_0, \ldots, u_{n/2-1}) , \\ U_1 &= (u_{n/2}, \ldots, u_{n-1}) , \\ V_0 &= (v_0, \ldots, v_{n/2-1}) , \\ V_1 &= (v_{n/2}, \ldots, v_{n-1}) . \end{aligned}$$

Dann gilt:

$$u = M \cdot U_1 + U_0 \qquad \text{und} \qquad v = M \cdot V_1 + V_0 ,$$

wobei $M = b^{n/2}$. Damit ergibt sich für das Produkt von u und v:

$$u \cdot v = (M^2 + M)U_1 \cdot V_1 + M(U_1 - U_0) \cdot (V_0 - V_1) + (M + 1)U_0 \cdot V_0 .$$

Damit kann man eine Multiplikation von zwei n-stelligen Zahlen zerlegen in drei Multiplikation von $n/2$-stelligen Zahlen zerlegen. Folgender Algorithmus benutzt diesen Ansatz:

Karazuba
input: $u = s_u \cdot (u_0, \ldots, u_{n-1})$
$v = s_v \cdot (v_0, \ldots, v_{n-1})$
$s_u, s_v \in \{-1, 1\}$
$u_0, \ldots, u_{n-1}, v_0, \ldots, v_{n-1} \in \{0, b-1\}$
{ **if** $n = 0$ **then return** $u \cdot v$
else {
$U_0 \leftarrow (u_0, \ldots, u_{n/2-1})$
$U_1 \leftarrow (u_{n/2}, \ldots, u_{n-1})$
$V_0 \leftarrow (v_0, \ldots, v_{n/2-1})$
$V_1 \leftarrow (v_{n/2}, \ldots, v_{n-1})$
$DU \leftarrow$ **add**$(U_1, -U_0)$
$DV \leftarrow$ **add**$(V_0, -V_1)$
$P_1 \leftarrow$ **Karazuba**(U_1, V_1)
$P_2 \leftarrow$ **Karazuba**(DU, DV)
$P_3 \leftarrow$ **Karazuba**(U_0, V_0)
$S_1 \leftarrow$ **right-shift**(P_1, n)
$S_2 \leftarrow$ **right-shift**$(P_1, n/2)$
$S_3 \leftarrow$ **right-shift**$(P_2, n/2)$
$S_4 \leftarrow$ **right-shift**$(P_3, n/2)$
$S_5 \leftarrow P_3$
$S \leftarrow$ **add**$(S_1, S_2, S_3, S_4, S_5)$
return $s_u \cdot s_v \cdot S$
}
}

Aufgabe 7.1.43: 6
Programmieren Sie die Prozeduren **right-shift** und **add.** Achten Sie auf Vorzeichen und auf das Laufzeitverhalten!

Theorem 22 *Der Algorithmus von Karazuba multipliziert n-stellige Zahlen in Zeit $O(n^{\log_2 3})$.*

Beweis: Die Korrektheit folgt sofort aus den vorab ausgeführten Überlegungenen. Zur Effizienz sei $T(n)$ die Laufzeit des Algorithmus. Addition und Verschiebeoperationen benötigen nur lineare Zeit. Damit ergibt sich die rekursive Gleichung:

$$T(n) \leq 3T(n/2) + O(n)$$

Somit ist (siehe Übungsaufgabe 5.33) $T(n) \in O(n^{\log_2 3})$. ∎

Beispiel 7.1.7: Wir multiplizieren $13579246 \cdot 77777777$. Als Basis legen wir $n = 10$ zugrunde. Dies ergibt also:

+

$$
\begin{array}{rcl|rcl|rcl|rcl}
\multicolumn{12}{c}{\mathbf{1357 \cdot 2468}} \\
U_0 & = & 57 & U_1 & = & 13 & V_0 & = & 68 & V_1 & = & 24 \\
U_1 - U_0 & = & -44 & V_0 - V_1 & = & 44 & M & = & 100 & & &
\end{array}
$$

$$
\begin{array}{rcl|rcl|rcl|rcl}
\multicolumn{12}{c}{13 \cdot 24} \\
U_0 & = & 3 & U_1 & = & 1 & V_0 & = & 4 & V_1 & = & 2 \\
U_1 - U_0 & = & -2 & V_0 - V_1 & = & 2 & M & = & 10 & & & \\
u \cdot v & = & \multicolumn{9}{l}{110 \cdot 2 + 10 \cdot (-4) + 11 \cdot 12 = \mathbf{312}}
\end{array}
$$

$$
\begin{array}{rcl|rcl|rcl|rcl}
\multicolumn{12}{c}{-44 \cdot 44} \\
U_0 & = & 4 & U_1 & = & 4 & V_0 & = & 4 & V_1 & = & 4 \\
U_1 - U_0 & = & 0 & V_0 - V_1 & = & 0 & M & = & 10 & & & \\
u \cdot v & = & \multicolumn{9}{l}{-(110 \cdot 16 + 10 \cdot (0) + 11 \cdot 16) = \mathbf{-1936}}
\end{array}
$$

$$
\begin{array}{rcl|rcl|rcl|rcl}
\multicolumn{12}{c}{57 \cdot 68} \\
U_0 & = & 7 & U_1 & = & 5 & V_0 & = & 8 & V_1 & = & 6 \\
U_1 - U_0 & = & -2 & V_0 - V_1 & = & 2 & M & = & 10 & & & \\
u \cdot v & = & \multicolumn{9}{l}{110 \cdot 30 + 10 \cdot (-4) + 11 \cdot 56 = \mathbf{3876}}
\end{array}
$$

$$
\begin{array}{rcl}
u \cdot v & = & 10100 \cdot 312 + 100 \cdot (-1936) + 101 \cdot 3876 = \mathbf{3349076}
\end{array}
$$

7.2 Anwendung der schnellen Fouriertransformation

Eine n-stellige Zahl A in Dezimaldarstellung ist gegeben als ein Polynom zur Basis $r = 10$:

$$A = \sum_{\nu=0}^{n} a_\nu \cdot r^\nu \ .$$

Multiplizieren wir zwei Zahlen A und B, so führen wir eine Polynommultiplikation von $A(r) \cdot B(r)$, ohne r explizit einzusetzen. Dieses Produkt berechnen wir mittels der diskreten Fouriertransformation. Hierbei ergeben sich folgende Propleme:

1. Die Koeffizienten des Produktpolynoms sind nicht unbedingt mehr Zahlen aus der Menge $\{0, \ldots, r-1\}$.
2. Wie multipliziert man komplexe Zahlen mittels ganzer Zahlen?

Lemma 13 *Wenn die Koeffizienten zweier Polynome A, B vom Grad n aus $\{0, \ldots, r-1\}$ gewählt werden, so ist im Produkt $C(x) = A(x) \cdot B(x)$ jeder Koeffizient nicht größer als $n \cdot (r-1)^2$.*

Beweis: Wie wir bereits gesehen haben berechnen sich die Koeffizienten von $C(x) = A(x) \cdot B(x)$ als

$$c_j = \sum_{k=0}^{j} a_k \cdot b_{j-k} \ ,$$

Da a_k, b_k Ziffern aus $\{0, \ldots, r-1\}$ sind, ist $c_j \in \mathbb{N}$ und folgendermaßen nach oben beschränkt:

$$c_j \leq j \cdot (x-1)^2 \leq n \cdot r^2 \ .$$

∎

Stellt man c_j wieder zur Basis r dar, so hat sie $\lceil \log_r n \rceil + 2 \leq \left\lceil \frac{\log n}{\log_2 r} \right\rceil + 2$ Stellen (vgl. die beiden gegebenen Zahlen A und B haben n Ziffern).
Sind also die Koeffizienten c_j bestimmt worden, so kann man sie in $\lceil \log_r n \rceil + 2$ Zahlen mit $2n$ Bits kombinieren. Diese Addition dieser Zahlen benötigt einen zeitlichen Aufwand von $O(n \log n)$.

+

Beispiel 7.2.8: Nach dieser Methode multiplizieren wir nun

$$987654321 \cdot 135792468\,.$$

Es gilt

$$\begin{aligned}
c_0 &= 1\cdot 8 = 8\\
c_1 &= 1\cdot 6 + 2\cdot 8 = 22\\
c_2 &= 1\cdot 4 + 2\cdot 6 + 3\cdot 8 = 40\\
c_3 &= 1\cdot 2 + 2\cdot 4 + 3\cdot 6 + 4\cdot 8 = 60\\
c_4 &= 1\cdot 9 + 2\cdot 2 + 3\cdot 4 + 4\cdot 6 + 5\cdot 8 = 89\\
c_5 &= 1\cdot 7 + 2\cdot 9 + 3\cdot 2 + 4\cdot 4 + 5\cdot 6 + 6\cdot 8 = 127\\
c_6 &= 1\cdot 5 + 2\cdot 7 + 3\cdot 9 + 4\cdot 2 + 5\cdot 4 + 6\cdot 6 + 7\cdot 8 = 166\\
c_7 &= 1\cdot 3 + 2\cdot 5 + 3\cdot 7 + 4\cdot 9 + 5\cdot 2 + 6\cdot 4 + 7\cdot 6 + 8\cdot 8 = 210\\
c_8 &= 1\cdot 1 + 2\cdot 3 + 3\cdot 5 + 4\cdot 7 + 5\cdot 9 + 6\cdot 2 + 7\cdot 4 + 8\cdot 6 + 9\cdot 8 = 255\\
c_9 &= 2\cdot 1 + 3\cdot 3 + 4\cdot 5 + 5\cdot 7 + 6\cdot 9 + 7\cdot 2 + 8\cdot 4 + 9\cdot 6 = 220\\
c_{10} &= 3\cdot 1 + 4\cdot 3 + 5\cdot 5 + 6\cdot 7 + 7\cdot 9 + 8\cdot 2 + 9\cdot 4 = 197\\
c_{11} &= 4\cdot 1 + 5\cdot 3 + 6\cdot 5 + 7\cdot 7 + 8\cdot 9 + 9\cdot 2 = 188\\
c_{12} &= 5\cdot 1 + 6\cdot 3 + 7\cdot 5 + 8\cdot 7 + 9\cdot 9 = 195\\
c_{13} &= 6\cdot 1 + 7\cdot 3 + 8\cdot 5 + 9\cdot 7 = 130\\
c_{14} &= 7\cdot 1 + 8\cdot 3 + 9\cdot 5 = 76\\
c_{15} &= 8\cdot 1 + 9\cdot 3 = 35\\
c_{16} &= 9\cdot 1 = 9
\end{aligned}$$

Somit muß nur noch folgende Addition ausgeführt werden.

	c_{16}	c_{15}	c_{14}	c_{13}	c_{12}	c_{11}	c_{10}	c_9	c_8	c_7	c_6	c_5	c_5	c_3	c_2	c_1	c_0
	9	35	76	130	195	188	197	220	255	210	166	125	89	60	40	22	8
		9	5	6	0	5	8	7	0	5	0	6	5	9	0	0	2	8
		3	7	3	9	8	9	2	5	1	6	2	8	6	4	2	0	0
+			1	1	1	1	2	2	2	1	1	0	0	0	0	0	0	0
1	3	4	1	1	6	0	1	7	7	7	9	4	5	4	2	2	8	

Also gilt

$$987654321 \cdot 135792468 \;=\; 134116017779454228\,.$$

Präzision

Lemma 14 *Bei der Addition oder Subtraktion zweier Zahlen a, b gegeben mit Genauigkeit $\pm\epsilon$ kann das Ergebnis mit Genauigkeit $\pm 2\epsilon$ angegeben werden.*

Beweis:

$$(a+\epsilon)+(b+\epsilon)-(a-\epsilon)+(b-\epsilon) = 4\epsilon$$

∎

Lemma 15 *Multipliziert man zwei Zahlen a, b gegeben mit Genauigkeit $\pm\epsilon$ so kann das Produkt mit Genauigkeit $\pm\epsilon(a+b)$ angegeben werden.*

Beweis:

$$(a+\epsilon)(b+\epsilon)-(a-\epsilon)(b-\epsilon) = \epsilon\cdot 2(a+b)$$

∎

+

Beispiel 7.2.9:

$$\begin{aligned}(13 \pm 0,5) + (12 \pm 0,5) &= (25 \pm 1)\\ (1 \pm 0,5) \cdot (9 \pm 0,5) &= (9.25 \pm 5)\\ (2,0 \pm 0,05) \cdot (3,0 \pm 0,05) &= (6,0025 \pm 0.25)\end{aligned}$$

Um die Berechnung reeller Zahlen nachzuempfinden, rechnen wir nur mit ℓ gültigen Nachkommastellen. Dies ergibt einen Fehler von $\pm r^{-\ell}$. Damit verringert sich in der Darstellung zur Basis r bei jeder Rechenoperateion die Anzahl der gültigen Stellen nur um eine Ziffer.

Aufgabe 7.2.44: 6
Eine Ziffernposition $k \in \{s, \ldots, t\}$ einer Zahl z in Darstellung $z = \sum_{j=s}^{t} a_j x^j \pm \epsilon$ heißt gültig, wenn $k \geq \log_x \epsilon + 1$. Beweisen Sie, daß eine Addition oder Multiplikation zweier Zahlen mit der gleichen Anzahl p gültiger Stellen, eine Zahl ergibt, deren Anzahl gültiger Stellen q nur um eins kleiner ist als p.

7.3 Der Multiplikationsalgorithmus

Wir fordern, daß zu Beginn der Berechnung die Anzahl gültiger Stellen ℓ der Einheitswurzeln mindestens die Größe $\ell \geq 6 \log n + 4$ besitzt. Als Stellenzahl der Basis wählen wir $k = \log n$. Die Basis r ist daher 2^k.

Theorem 23 *Eine RAM mit Wortlänge w kann zwei n-stellige Binärzahlen a, b mittels des Algorithmus* **Mult-FFT** *in Zeit $O(n)$ multiplizieren, wenn $w \geq 6 \log n + 4$.*
Es gibt eine Konstante c, so daß beliebige Zahlen durch **Mult-FFT** *in Zeit $O(c^{\text{itlog } n} \cdot n \cdot \log n \cdot \log\log n \cdot \log\log\log n \cdots)$ multipliziert werden.*

Beweis: **Korrektheit**
Abb. 6.2 und 6.3 zeigen, daß die schnelle Fouriertransformation aus $2 \log m$ Schichten paralleler Additionen und Multiplikationen besteht. Wenn nun für die Einheitswurzeln Approximationen mit ℓ gültigen Stellen eingesetzt werden, verringern sich mit jedem parallelen Berechnungsschritt die Anzahl gültiger Stellen um eine Stelle.
Die komplexe Multiplikation kann in zwei parallelen Berechnungsschritten dargestellt werden; vier parallele Multiplikationen und danach zwei parallele Additionen:

$$(a + bi)(c + di) = (ac - bd) + i(ad + bc) \ .$$

Die komplexe Addition entspricht einen Berechnungsschritt aus zwei parallelen reellen Additionen.
Insgesamt sind also zur Berechnung der diskreten Fouriertransformierten $2 \log m$ parallele reellwertige Multiplikationsschritte und $\log m$ reell-wertige Additionen. Die Berechnung des eigentlichen Produkts ist eine parallele Multiplikation. Die Berechnung der inversen Fouriertransformation hat die gleiche Eigenschaften wie die der Fouriertransformation.

+

Mult-FFT
input: $A = \sum_{i=0}^{n-1} a_i 2^i$
$B = \sum_{i=0}^{n-1} b_i 2^i$
w: Wortlänge der RAM
{ $\ell \leftarrow 6 \log n + 4$
if $\ell \leq w$ **then set-precision**(w)
else set-precision(ℓ)
$k \leftarrow \log n$
$r \leftarrow 2^k$
$m \leftarrow \lceil \frac{n}{k} \rceil$
for $j \leftarrow 0$ **to** $m-1$ **do**
{ $a'_j \leftarrow \sum_{i=0}^{k-1} a_{i+kj} 2^i$
$b'_j \leftarrow \sum_{i=0}^{k-1} b_{i+kj} 2^i$
}
$(u_0, \ldots, u_{m-1}) \leftarrow$ **FFT**$(a'_0, \ldots, a'_{m-1})$
$(v_0, \ldots, v_{m-1}) \leftarrow$ **FFT**$(b'_0, \ldots, b'_{m-1})$
for $j \leftarrow 0$ **to** $m-1$ **do** $z_i \leftarrow u_i \cdot v_i$
$(c_0, \ldots, c_{m-1}) \leftarrow$ **inv-FFT**$(z_0, \ldots, z_{m-1})$
for $j \leftarrow 0$ **to** $m-1$ **do**
$s_{j \bmod (k+2)} \leftarrow s_{j \bmod (k+2)} + \lfloor c_j + \frac{1}{2} \rfloor \cdot r^j$
$S \leftarrow 0$ **for** $j \leftarrow 0$ **to** $\log k + 1$ **do**
$S \leftarrow S + s_j$
return S
}

Damit also mindestens $\lceil \log_r m \rceil + 2$ gültige Stellen am Ende der Berechnung vorhanden sind, muß für die gültige Stellenanzahl ℓ der komplexen Einheitswurzel ω_n^k gelten:

$$\ell \;\geq\; \lceil \log_r m \rceil + 6 \log m + 3.$$

Nun ist $k = \log n$, $r = 2^k$, $m = \lceil \frac{n}{k} \rceil$. Somit muß gelten

$$\ell \geq \underbrace{\left\lceil \frac{\log m}{k} \right\rceil + 6 \log m + 3}_{\leq 6 \log n + 4}.$$

Diese Wahl wurde im Algorithmus umgesetzt.

Effizienz

Um zwei Zahlen mit m Ziffern (zur Basis r) zu multiplizieren, müssen $m(2 \log m + 1)$ komplexe Multiplikationen mit ℓ-binärstelligen Real- und Imaginärteil berechnet werden. Dies sind insgesamt $2n(2 \log n + 1)$ reell-wertige Multiplikationen. Ist dies innerhalb der Wortlänge w der berechnenden Maschine möglich ($w \geq \ell$), so kostet eine Multiplikation nur einen Schritt. Die Addition der Koeffizienten c_j kann unter geschickter Ausnutzung der Wortlänge der RAM in linearer Zeit berechnet werden. Man erhält so die folgende Laufzeit $T'(n)$ für den Algorithmus:

$$T'(n) \;=\; O(m \log m) + O(n) = O(n)\ .$$

Ist die Wortlänge w kleiner als die notwendige Präzision ℓ, wendet man dieses Verfahren rekursiv an. So erhält man im ersten Schritt $4m \log m + 2m$ Multiplikationen von zwei ℓ-stelligen Zahlen.

+

Die Laufzeit $T(n)$ ergibt sich also aus folgender Rekurstionsgleichung:

$$T(n) \leq (4m \log m + 2m) \cdot T(\ell) + O(n \log n) \ .$$

Der Aufwand $O(n \log n)$ wird benötigt, um die abschließende Addition der Koeffizienten zu berechnen. Wir setzen $m = \frac{n}{\log n}$ und $\ell = 7 \log n$ ein und erhalten:

$$T(n) \leq 6n \cdot T(7 \log n) + O(n \log n) \ .$$

Damit erhält man die gesuchte Laufzeit für $c = 42$. ∎
Verwendet man statt komplexer Zahlen den Restklassenring bezüglich einer Zahl $2^e + 1$ für die diskrete Fouriertransformation, kann man die asymptotische Laufzeit für die Multiplikation noch weiter verbessern auf $O(n \log n \log \log n)$ (Verfahren von Schönhage und Strassen [SS 71]).

Aufgabe 7.3.45: 8
Wie schnell ist der Algorithmus **Mult-FFT**, wenn für die Wortlänge der RAM gilt: $w \in O(\log n)$?

Aufgabe 7.3.46: 6
Verifizieren oder Falsifizieren Sie folgeden Aussagen ($c \in N$):

$$\begin{aligned} c^{\text{itlog}} &\in O(\log\log\log\log n) \\ c^{\text{itlog}}\ n \log n \log\log n (\log\log\log n)^2 &\in O(n \log^2 n) \\ c^{\text{itlog}} &\in O(\text{itlog}\ n) \end{aligned}$$

7.4 Division

Genauso wie bei der Berechnung der Quadratwurzel kann bei der Division das Newtonsche Iterationsverfahren verwendet werden. Jede Division a/b kann als Multiplikation $a\frac{1}{b}$ aufgefaßt werden, so daß wir uns darauf beschränken werden den Kehrwert einer Zahl $v \in]\frac{1}{2}; 1[$ zu berechnen.

Reciprocal
input: $v = (0, v_1 v_1 v_2, \ldots)_2$, $v_i \in \{0,1\}$
$n \in \mathbb{N}$
{ $z \leftarrow \frac{1}{4} \left\lfloor \frac{32}{4v_1 + 2v_2 + v_3} \right\rfloor$
for $k \leftarrow 0$ **to** $\log n$ **do**
$z \leftarrow \frac{(2z - vz^2)2^{2^{k+1}} + \frac{1}{2}}{2^{2^{k+1}}}$
return z
}

Aufgabe 7.4.47: 1
Berechnen Sie mit Hilfe dieses Algorithmus die Binärdarstellung von $\frac{1}{5}$.

Theorem 24 *Wenn $M(n)$ der zeitliche Aufwand für eine Multiplikation zweier n-stelliger Binärzahlen ist, dann berechnet der Algorithmus* **Reciprocal** *den Kehrwert einer Zahl $v \in]\frac{1}{2}; 1[$ bis auf n Binärstellen in Zeit $O(M(n))$.*

+

Beweis: **Korrektheit**
durch vollständige Induktion über k. Es wird angenommen, daß vor jedem Schleifendurchlauf gilt

$$z = \frac{1}{v} \pm 2^{-e} .$$

Für den Induktionsschluß gilt also Sei $z_0 = \frac{1}{v} + x$ für $x \in [2^{-e}, 2^e]$. Dann berechnet sich der nächste Wert als der gerundete Wert von z_1.

$$\begin{aligned} z_1 &= 2z_0 - vz_0^2 \\ &= \frac{1}{v} - v(z_0 - \frac{1}{v})^2 \\ &= \frac{1}{v} - vx^2 \\ \Longrightarrow \qquad z_1 &\in \left[\frac{1}{v} - 2^{-2e}; \frac{1}{v}\right] \end{aligned}$$

Mit jeder Schleife verdoppelt sich also die Anzahl gültiger Stellen.
Effizienz
Innerhalb der Schleife sind zwei n-stellige Multiplikationen und eine Subtraktion zu berechnen. Damit ergibt sich für den zeitlichen Aufwand $T(n)$ zur Berechnung des n-stelligen Kehrwerts:

$$T(n) \leq O(n) + 2 \cdot \left(M(n) + M\left(\frac{n}{2}\right) + M\left(\frac{n}{4}\right) + \cdots\right).$$

Für alle Laufzeitschranken, die wir bisher für die Multiplikation in Betracht gezogen haben, gilt also $T(n) = O(M(n))$. ∎

Aufgabe 7.4.48: 3
Beweisen Sie die Induktionsverankerung des Korrektheitsbeweises per Fallunterscheidung.

Aufgabe 7.4.49: 9
Zeigen Sie, daß die Berechnung der k-ten Wurzel auf n Stellen Genauigkeit nur um eine multiplikative Konstante aufwendiger ist als die Multiplikation zweier n-stelliger Zahlen.

Kapitel 8

Lineare Programmierung

Das lineare Programmierungsproblem[1] ist eines der grundlegenden Optimierungsprobleme. Es ist definiert als

Lineare Programmierung (LP) ***(linear programming)***
Geg.: $m \times n$-Matrix A,
m-dim. Vektor b
n-dim. Vektor c
Ges.: n-dim. Vektor $x = (x_1, \ldots, x_n)$ mit den Eigenschaften:
$x \geq 0$, d.h. $\forall j \; x_j \geq 0$
$Ax = b$, d.h. $\sum_{j=1}^{n} \sum_{i=1}^{m} a_{ij} x_j = b_j$
$z = c^T x$ ist minimal, d.h. $z = \sum_{j=1}^{n} c_j x_j$.

Viele praktisch interessante Problem können als lineares Programm formuliert werden. Hier nur ein Beispiel:

Beispiel 8.0.10: Das Diätproblem.
Gesucht ist eine möglichst sparsame Diät, die die minimale tägliche Grundversorgung an Kohlenhydraten, Eiweiß, Fetten, Vitaminen, Mineralstoffe, etc. gewährleistet. Natürliche Nahrungsmittel bieten in der Regel aber von jeden etwas.

Die Nahrungsmittel seien durchnumeriert (Brot = 1, Wirsing = 2, ...) und sei c_j der Energiegehalt von 100g des j-ten Nahrungsmittel. b_i ist der tägliche Minimalbedarf an Grundnahrungsstoff Nummer i. Nun ist also die Menge x_j an Nahrungsmittel j gesucht, die dieser sparsamsten Diät genüg.

Hierzu muß $\sum_{j=0}^{m} c_j x_j$ minimiert werden und die Gleichungen

$$\begin{aligned} \sum_{j=0}^{n} a_{ij} x_j &\geq b_j, \quad i = 1, 2, \ldots, m \; , \\ x_j &\geq 0, \quad i = 1, 2, \ldots, n \end{aligned}$$

eingehalten werden.

Mitunter werden lineare Programme also in unterschiedlichen Formen betrachtet wie z.B.:

[1] Da streikbedingt eine Vorlesungausgefallen ist, konnten nur die wichtigsten Sätze (oftmals ohne Beweise) vorgestellt werden. Für eine adäquate Darstellund dieses Gebiets sei auf [NKT 89] verwiesen.

Lineare Programmierung (LP2) ***(linear programming)***
Geg.: $m \times n$-Matrix A,
m-dim. Vektor b
n-dim. Vektor c
Ges.: n-dim. Vektor x
$x \geq 0$
$Ax \leq b$
$c^T x$ ist maximal.

Lemma 16 LP *und* **LP2** *können ineinander umgeformt werden, wobei die Problemstellung sich nur um einen konstanten Faktor vergrößert.*

Beweis: Maximierungsprobleme können durch Vorzeichenumkehr von c in Minimierungsprobleme umgewandelt werden und umgekehrt. Wir setzen nun voraus, daß beide Problemstellungen Minimierungsprobleme sind.

- Sei ein lineares Programm der Form **LP2** gegeben, wobei $Ax \leq b$:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} .$$

Somit muß jede der folgenden Ungleichungen für $i \in \{1, \ldots, m\}$ eingehalten werden:

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i .$$

Wir fügen nun m neue Variablen $x_{n+1}, \ldots, x_{n+m}$ hinzu, so daß $x' = (x_1, \ldots, x_{m+n})^T$. und ergänzen die Matrix A wie folgt:

$$A' = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & 1 & & 0 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & & 1 & \\ \vdots & \vdots & \ddots & \vdots & & & \ddots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} & 0 & & 1 \end{pmatrix}$$

Der Vektor b bleibt unverändert. Der Vekter $c' = (c_1, \ldots, c_n, 0, \ldots, 0)$. Betrachten wir nun das Gleichungssystem $A'x' = b$ mit $x' \geq 0$. Zur Optimierung von $c'^T x'$ tragen die neu angehängten freien Variablen nichts bei. Damit können sie frei gewählt werden bis auf die beiden Einschränkungen:

$$x_{n+i} \geq 0 \quad \wedge \quad x_{n+i} + \sum_{j=1}^{n} a_{ij} x_j = b_i .$$

Dies ist aber äquivalent zu

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i .$$

+

- Ist ein Programm der Form **LP** gegeben, wird die Gleichung $Ax = b$ durch zwei Ungleichungen $Ax \leq b$ und $-Ax \leq -b$ ersetzt:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \\ -a_{1,1} & -a_{1,2} & \cdots & -a_{1,n} \\ -a_{2,1} & -a_{2,2} & \cdots & -a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ -a_{m,1} & -a_{m,2} & \cdots & -a_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} b_1 \\ \vdots \\ b_m \\ -b_1 \\ \vdots \\ -b_m \end{pmatrix} .$$

  Der Vektor c bleibt unverändert.

∎

Es genügt also nur das Problem **LP** zu lösen, welches wir im folgenden als **kanonische Form** bezeichnen.

8.1 Geometrische Interpretation

Definition 3

- *Eine Menge $C \subseteq \mathbb{R}^n$ ist **konvex**, gdw.*

$$\forall x, y \in C \; \forall \lambda \in [0;1] \; : \qquad z(y) = \lambda x + (1-\lambda)y) \in C \; .$$

- *x ist eine **konvexe Kombination** aus $x_1, \ldots, x_N$ gdw.*

$$x = \sum_{i=1}^{N} \lambda_i x_i \quad \wedge \quad \forall i \;\; \lambda_i \geq 0 \quad \wedge \quad \sum_{i=1}^{N} \lambda_i = 1 \; .$$

- *$C \subseteq \mathbb{R}^n$ ist ein **Kegel** (Konus), gdw.*

$$\forall x \in C \; \forall \lambda \in \mathbb{R}_0^+ \quad \lambda x \in C \; .$$

- *Die Menge $H = \{x \in \mathbb{R}^n \mid a^T x = \beta\}$ mit $a \in \mathbb{R}^n$, $a \neq 0$, $\beta \in \mathbb{R}^n$ ist eine **Hyperebene**. Die Menge $\overline{H} = \{x \in \mathbb{R}^n \mid a^T x \leq b\}$ ist ein **geschlossener Halbraum**.*

Eine Hyperebene ist eine $n-1$-dimensionale affine Menge. Der Normalvektor dieser Hyperebene ist a.

Definition 4

- *Ein **konvexer Polyeder** ist eine Menge, die durch Schnitt endlich vieler geschlossener Halbräume entsteht. Falls der Polyeder nicht leer ist und beschränkt, heißt er **konvexer Polytop**.*

- *Die **Dimension** eines Teilraums S ist gleich der maximalen Anzahl unabhängiger Vektoren in S.*

- *Eine **unterstützende Hyperebene** einer konvexen Menge C hat die Eigenschaft $H \cap C \neq \emptyset$ und $C \subseteq H$, wobei C in einer der beiden geschlossen Halbräume von H liegt.*

+

- *Sei P ein d-dimensionaler konvexer Polyeder und H eine unterstützende Hyperebene, dann definiert $F = P \cap H$ eine* **Seite** *von P:*
  - *Eine Seite mit Dimension 0 heißt* **Knoten** *(Ecke).*
  - *Eine Seite mit Dimension 1 heißt* **Kante***.*
  - *Eine Seite mit Dimension $d-1$ heißt* **Seitenfläche***.*
- *Eine* **Richtung** *eines konvexen Polyeders P ist ein Vektor $d \in \mathbb{R}^n$, so daß $\forall x_0 \in P$ der Strahl $\{x \in \mathbb{R}^n \mid \exists \lambda \in \mathbb{R}_0^+ \; x = x_0 + \lambda d\}$ in P liegt.*

Lemma 17 *Ein Punkt $x \in P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$ ist ein Knoten, gdw. wenn die Spalten von A, die zu positiven Komponenten von x korrespondieren, linear unabhängig sind.*

Aufgabe 8.1.50: 7
Sei P der durch die Matrix A beschriebene n-dimensionale konvexe Polyeder. Wieviele Ecken, Kanten und Seitenflächen kann dieser maximal besitzen?

Definition 5 *Sei B eine nicht-singuläre $m \times m$-Matrix zusammengesetzt aus m (linear unabhängigen) Spalten von A. Alle Komponenten von x, die nicht dieser Auswahl entsprechen, werden auf 0 gesetzt. Die Gleichung $Ax = b$ wird für die ausgewählten Komponenten von x, genannt* **Basisvariablen***, gelöst. Dieses Ergebnis wird dann* **Basislösung** *genannt bezüglich der* **Basis** *B. Falls die Basislösung $x \geq 0$ ist, wird sie* **zulässige Basislösung** *genannt.*

Fakt 3 *Ein Punkt x ist ein Knoten eines Polyeders P gdw. wenn x eine zuläsige Basislösung zu einer Basis B ist.*

Fakt 4 *Ein Polyeder P hat nur eine endliche Anzahl an Knoten.*

Lemma 18 *Jeder Punkt $x \in P$ kann dargestellt werden als*

$$x = \sum_{i \in I} \lambda_i v_i + d \; ,$$

wobei $\{v_i \mid i \in I\}$ die Menge der Knoten von P ist,

$$\sum_{i \in I} \lambda_i = 1 \; , \quad \forall i \in I \; \lambda_i \geq 0$$

und d entweder eine Richtung von P ist oder $d = 0$.

Fakt 5 *Falls P ein Polytop ist, dann kann jeder Punkt $x \in P$ als konvexe Kombination der Knoten dargestellt werden.*

Beispiel 8.1.11: Betrachten Sie Abb. 8.1. Gegeben ist die Schnittlinie der Ebenen $x_1 - x_2 = 1$ und $3x_1 + x_2 + x_3 = 9$. Gesucht ist der Punkt $x \geq 0$ auf der Schnittlinie mit größter x_3-Koordinate (optimaler Punkt ist nicht eingezeichnet).

Das Lineare Optimierungsproblem $A \cdot x = b$ mit Vektor c ergibt sich also aus

$$\underbrace{\begin{pmatrix} 1 & -1 & 0 \\ 3 & 1 & 1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 1 \\ 9 \end{pmatrix}}_{b} ,$$

+

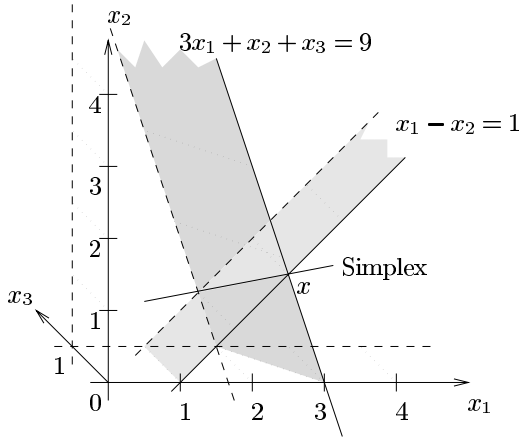

Abbildung 8.1: Ein einfaches Beispiel für eine lineare Optimierung.

wobei für $x > 0$ die Zehl $c^T x = (0\ 0\ -1)\, x$ minimiert werden muß.

Wenn wir als Basisvariablen die ersten beiden Koordinaten x_1, x_2 wählen, erhalten wir

$$B = \begin{pmatrix} 1 & -1 \\ 3 & 1 \end{pmatrix} \qquad \text{und} \qquad B^{-1} = \frac{1}{4}\begin{pmatrix} 1 & 1 \\ -3 & 1 \end{pmatrix} .$$

Theorem 25 *Sei P der durch die Matrix A beschriebene konvexe Polyeder. Wenn P nicht leer ist, dann wird der kleinste Wert von $z = c^T x$ für $x \in P$ entweder in einem Knoten von P angenommen oder es gibt keine untere Schranke in P.*

8.2 Der Simplexalgorithmus

Die Idee dieses Algorithmus ist: Finde eine zulässige Lösung. Diese stellt einen Knoten im Polyeder P dar. Nun folgen wir einer der Abwärtskanten zu einer besseren Lösung (Knoten). Dies wird solange fortgesetzt bis wir die optimale Lösung gefunden haben, dann führen alle Kanten aufwärts, oder eine Abwärtskante ins Unendliche führt (dann gibt es keine untere Schranke für $c^T x$).
O.B.d.A. bilden die ersten m Komponenten von x eine Basis. Dann sei:

$$\begin{aligned} x &= \begin{pmatrix} x_B \\ x_N \end{pmatrix} = \begin{pmatrix} B^{-1}b \\ 0 \end{pmatrix} , \\ A &= (B \quad N) , \\ c &= \begin{pmatrix} c_B \\ c_N \end{pmatrix} . \end{aligned}$$

Definition 6 *Falls mindestens eine Basisvariable 0 ist, dann heißt die Basislösung* **degeneriert**, *andernfalls* **nicht-degeneriert.**

+

Falls eine Lösung nicht degeneriert ist, dann liegt die Lösung im Schnitt von genau m Hyperebenen. Sei

$$M = \begin{pmatrix} B & N \\ 0 & E_{n-m} \end{pmatrix} .$$

Die Zeilen der Matrix M stellen nun die Normalen der n Ebenen auf der x liegt dar. Dann ist

$$M^{-1} = \begin{pmatrix} B^{-1} & -B^{-1}N \\ 0 & E_{n-m} \end{pmatrix} .$$

Sei e_q die q-te Spalte der $n \times n$-Einheitsmatrix. Dann definieren wir

$$\eta_q = M^{-1} e_q .$$

Der Vektor η_q ist nun parallel zu $n-1$ Hyperebenen (die nicht der q-ten Zeile von A entsprechen. Damit stellt

$$x(s) = x + s\eta_q .$$

für $s \geq 0$ eine Kante des Polyeders dar.
Das Gefälle der Kante, genannt **relative Kosten** $\overline{c_j}$ berechnet sich als

$$\overline{c_j} = c^T \eta_j = c_j - c_B^T B^{-1} a_j ,$$

wobei a_j die j-te Spalte der Matrix A ist. Meistens wird beim Simplexverfahren die Kante gewählt, die am steilsten ist. Der Winkel, der am stumpfesten zu c ist, berechnet sich als

$$\Phi_q = \cos^{-1}\left(\frac{c^T \eta_q}{||c|| \cdot ||\eta_q||}\right) .$$

Lemma 19 *Sei x eine zulässige Basislösung gegeben, dann kann jeder Punkt $y \in P$ dargestellt werden als*

$$y = x + \sum_{j=m+1}^{n} y_j \eta_j ,$$

wobei $\forall j \in \{m+1, \ldots, n\} \; : \; y_j \geq 0$ und η_j die j-te Spalte von M^{-1} ist.

Lemma 20 *Eine zulässige Basislösung ist genau dann eine optimale Lösung, wenn alle entsprechenden relativen Kosten keine negative Komponenten besitzen.*

Um einer Kante zu folgen, erhöht man s in $x(s) = x + s\eta_q$ solange, bis eine der Basisvariablen 0 wird. Sei

$$w = B^{-1} a_q ,$$

dann ist $x(s) \geq 0$ gdw. $x_B - sw \geq 0$ und $s \geq 0$. Somit gilt

Lemma 21 *Wenn $\overline{c_q} < 0$ und $w \leq 0$, dann ist das lineare Programmsystem nicht beschränkt. Dann ist nämlich $x(s)$ zulässig für alle $s \geq 0$ und $\lim_{s\to\infty} c^T x(s) = -\infty$. $d = \eta_q$ ist dann eine Richtung mit $c^T d = \overline{c_q} < 0$.*

Wenn wir der q-ten Kante bis zur nächsten Ecke des Polyeders folgen wollen, berechnet sich s wie folgt:

$$s = \min\left\{\frac{x_i}{w_i} \mid w_i > 0 \; \wedge \; 1 \leq i \leq m\right\} = \frac{x_p}{w_p} .$$

+

Beispiel 8.2.12: Bezugnehmend auf das obige Beispiel ist

$$B^{-1}b = \frac{1}{4}\begin{pmatrix} 1 & 1 \\ -3 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 9 \end{pmatrix} = \begin{pmatrix} \frac{5}{2} \\ \frac{3}{2} \end{pmatrix}$$

eine zulässige Basislösung. Der Punkt $x^T = (\frac{5}{2}\ \frac{3}{2}\ 0)$ ist in Abb. 8.1 eingezeichnet. Nun gilt

$$M = \begin{pmatrix} 1 & -1 & 0 \\ 3 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{und} \quad M^{-1} = \begin{pmatrix} \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ -\frac{3}{4} & \frac{1}{4} & -\frac{1}{4} \\ 0 & 0 & 1 \end{pmatrix} .$$

η_3 ergibt sich nun als

$$\eta_3 = M^{-1}e_3 = \begin{pmatrix} \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ -\frac{3}{4} & \frac{1}{4} & -\frac{1}{4} \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -\frac{1}{4} \\ -\frac{1}{4} \\ 1 \end{pmatrix}$$

Die relativen Kosten $\overline{c_3}$ ergeben sich nun als $c^T\eta_3 = (0\ 0\ -1)\eta_3 = -1$. Da diese negativ sind, kann fortgefahren werden. Wir gehen also entlang dieser Kante (die hier sogar dem gesamten Simplex entspricht) bis eine der beiden Koordinaten x_1 oder x_2 den Wert 0 annimmt.

Hierzu berechnen wir w als

$$w = B^{-1}a_3 = \frac{1}{4}\begin{pmatrix} 1 & 1 \\ -3 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{4} \\ \frac{1}{4} \end{pmatrix} .$$

Die nächste Position auf der Kante berechnet sich mittels s:

$$s = \min(\frac{x_1}{w_1}, \frac{x_2}{w_2}) = \min(10, 6) = 6 .$$

Somit ist die nächste zu betrachtende Lösung x':

$$x' = x + s\eta_3 = \begin{pmatrix} \frac{5}{2} \\ \frac{3}{2} \\ 0 \end{pmatrix} + 6\begin{pmatrix} -\frac{1}{4} \\ -\frac{1}{4} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 6 \end{pmatrix} .$$

Da der Simplex nur aus einer Kante besteht, ist also x' nun die optimale Lösung.

+

Simplex-Algorithmus
input: $m \times n$-Matrix A,
m-dim. Vektor b
n-dim. Vektor c
{ $I_B \leftarrow$ eine Menge $\{j_1, \ldots, j_m\}$ von m Positionen deren
Spaltenvektoren in A unabhängig sind.
$B \leftarrow (a_{j_1}, \ldots, a_{j_m})$
$x \leftarrow B^{-1}b$
stop $\leftarrow$ **false**
while $\neg$*stop* **do**
{ $c_B \leftarrow (c_{j_1}, \ldots, c_{j_m})$
for all $j \notin I_B$ **do** $\overline{c_j} \leftarrow c_j - c_B B^{-1} a_j$
optimal $\leftarrow \bigwedge_{j \notin I_B} \overline{c_j} \geq 0$
stop $\leftarrow$ *optimal*
if $\neg$*stop* **then**
{ $V \leftarrow \{j \notin I_B \mid \overline{c_j} < 0\}$
$q \leftarrow$ beliebiges Element aus V
$w \leftarrow B^{-1} a_q$
stop $\leftarrow (w \leq 0)$
if $\neg$*stop* **then**
{ Bestimme j_p so daß $\frac{x_{j_p}}{w_p} = \min_{1 \leq i \leq m}\{\frac{x_{j_i}}{w_i} \mid w_i \geq 0\}$
$s \leftarrow \frac{x_{j_p}}{w_p}$
$x_q \leftarrow s$
for all $i \in \{1, \ldots m\}$ **do** $x_{j_i} \leftarrow x_{j_i} - s w_i$
$B \leftarrow$ ersetze Spalte q durch Spalte j_p.
$I_B \leftarrow (I_B \setminus \{q\}) \cup \{j_p\}$
$j_p \leftarrow q$
}
}
}
if *optimal* **then return** x
else return keine untere Schranke
}

Theorem 26 *Der Simplex-Algorithmus löst das lineare Programmierungsproblem* **LP2** *in polynomieller Zeit, wenn die längste Abwärtskantenfolge polynomiell viele Kanten hat und keine der Lösungen degeneriert ist.*

Die Anzahl der Knoten eines Polyeders ist beschränkt durch $\binom{n}{m}$. Leider sind Kantenzüge konstruierbar, die eine überpolynomielle Anzahl von Knoten ablaufen, bevor der Simplexalgorithmus die optimale Lösung findet. In der Praxis kommen solche Eingaben aber kaum vor.
Es existiert noch ein weiteres sehr bekanntes Verfahren, die **Ellipsoid-Methode.** Hier werden um die Menge möglicher optimaler Lösungen immer engere Ellipsen gelegt und so wortwörtlich die Lösung immer enger eingekreist. Dieses Verfahren hat nachweislich polynomielle Laufzeit (wenn die Lösungen nicht zu dicht aneinander liegen).

Aufgabe 8.2.51: 9
Was passiert, wenn eine Lösung degeneriert ist? Modifizieren Sie den Simplexalgorithmus, damit dieser Fall nicht zu Problemen führt.

Aufgabe 8.2.52: 5
Sei ℓ die Anzahl der Knoten, die der Simplexalgorithmus auf seinen Weg zur optimalen Lösung besucht. Geben sie die Größenordnung des Zeitaufwand des Simplexalgorithmus in Abhängigkeit von n, m und ℓ an.

Aufgabe 8.2.53: 7
Klassifizieren sie die Eingaben, in der eine Basis nicht gefunden werden kann.

Kapitel 9

Organisation

9.1 Skript

Während dieser Veranstaltung ist ein Skript ausgegeben worden. Dies ist das überarbeitete und fehlerverringerte Skript, das nach Beendigung der Vorlesung ausgegeben wird. Die Reihenfolge wurde etwas verändert. An einigen Stellen wurden neue Passagen eingefügt. Die Literaturliste ist kommentiert und ergänzt worden. Allen Teilnehmern wird empfohlen die Übungsaufgaben des Skripts (Schwierigkeitsgrad 1-10 angegeben) zu bearbeiten, da deren Lösung das Verstehen des Vorlesungsstoffs erleichtert.
Dieses Skript ist erhältlich per WWW (`http://www.itheoi.mu-luebeck.de/pages/schindel/` oder kann geheftet im Sekretariat abgeholt werden. Prfungsrelevant ist ausschließlich der Stoff, der in der Vorlesung behandelt wurde.

9.2 Prüfung und Sprechstunden

Diese Veranstaltung ist gemäß Prüfungsordnung eine vertiefende zweistündige Vorlesung. Die erfolgreiche Teilnahme an dieser Vorlesung wird nach einer mündlichen Prüfung bescheinigt.
Die Sprechstunden sind während der vorlesungsfreien Zeit nach Vereinbarung (Ort: alte Seefahrtschule, Wallstraße 40, Raum 4). Vereinbarung per Telefon 70 30-414 oder via E-Mail `schindel@informatik.mu-luebeck.de`.
Anmeldungen und Terminabsprachen für die mündliche Prüfung zu dieser Veranstaltung können persönlich als auch schriftlich per E-Mail (`schindel@informatik.mu-luebeck.de`) vorgenommen werden.
Ein Termin für eine mündliche Prüfung zur Erlangung einer Vorlesungsbescheinigung kann in der Sprechstunde vereinbart werden. Die mündliche Prüfung gliedert sich in einen schriftlichen Teil, in der innerhalb einer halben Stunde eine Aufgabe zu lösen ist und einen mündlichen Teil, in der Fragen zur Vorlesung gestellt werden.
Zur Vorbereitung wird dringend empfohlen, die Sprechstunden vorher zu besuchen und einige Übungsaufgaben aus den Skript zu rechnen.

Literaturverzeichnis

Lehrbücher

[CLR 90] Cormen, Leiserson, Rivest, „Introduction to Algorithms“, MIT Press, 1990.

Diese Buch kann uneingeschränkt empfohlen werden (auch zum Kauf). Es werden weite Teile des Stoffs dieser Veranstaltung beschrieben. Auch werden viele Problemstellungen behandelt, die nicht in dieser Veranstaltung beschrieben werden konnten.

[Sed 91] Sedgewick, „Algorithms“, Addison-Wesley, 1991.

Diese Monographie ist auch in Deutsch erhältlich, daneben gibt es modifiezierte Fassungen, wie etwa „Algorithms in C++“ oder Algorithms in „Pascal“. Das Themengebiet diese Buches ist enger umfaßt, hat aber durch die Angabe expliziter Programmbeispiele höheren Praxisbezug.

[Knu 69] Knuth, „The Art of Computer Programming“, Vol.1-2, 1968, 1969.

Ein Klassiker und umfassendes Standardwerk. Leider etwas in die Jahre gekommen. Vom Stil ist es für das Selbststudium von Studenten sehr komprimiert und daher schwierig zu verstehen. Für Studenten, die sich tiefer mit dem Gebiet der Algorithmen auseinandersetzen müssen, sind diese beiden Bande ein Muß.

[AHU 74] Aho, Hopcroft, Ullman, „The Design and Analysis of Computer Algorithms“, Addison-Wesley, 1974.

Noch ein Klassiker. Etwas aufführlicher als Knuths Werk, aber nicht so umfassend. Dieses Buch umfaßt das Grundlagenwissen für effiziente Algorithmen. Genauso ein Muß wie Knuths Bände.

[PB 84] Purdom, Brown, „The Analysis of Algorithms“, Holt, Rinehart, Winston, 1984.

[Koz 91] Kozen, „The Design and Analysis of Algorithms“, Springer, 1991.

Neuere aktualisierte Bücher, die sich an [AHU 74] orientieren.

Vertiefende Literatur

[Ste 91] Graham A Stephen, „String Searching Algorithms“, World Scientific, 1994.

Für alle Leser, die mehr über Stringsuche im Text erfahren wollen.

[NKT 89] Nemhauser, Rinnooy Kan, Todd, „Optimization“, Handbooks in Operations Research and Management Science, Vol. 1, Elsevier, 1989.

Hier wird das Simplex-Verfahren mit Varianten anschaulich und ausführlich beschrieben. Weitergehende Strategien der linearen Optimierung werden besprochen.

[RR 90] Reischuk, „Einführung in die Komplexitätstheorie“, Teubner, 1990, 2. Auflage in Vorbereitung, erscheint 1998.

Das Standardwerk der Komplexitätstheorie[1].

[Tarj 83] Tarjan, „Data Structures and Network Alogirhms“, Society for Industrial and Applied Mathematics, 1983.

Originalartikel

[CW 87] Coppersmith, Winograd, „Matrix multiplication via arithmetic progressions“, Proceedings of the 19th Annual ACM Symposium on Theory of Computing, S. 1-6, 1987

[FT 87] Fredman, Tarjan, „Fibonacci heaps and their uses in improved network optimization algorithms“, Journal of the ACM, Vol. 37(2), S. 209-221, 1987.

[Rive 77] Rivest, „On the worst-case behaviour of string searching algorithms“, Journal an Computing Vol.6, No. 4, S. 669-74, 1977.

[SS 71] Schönhage, Strassen, „Schnelle Multiplikation großer Zahlen“, Journal on Computing 7, S. 281-292, 1971.

[Tarj 75] Tarjan, „Efficiency of a good but not linear set union algorithm“, Journal of the ACM, Vol. 22(2), S. 215-225, 1975.

[1] Nicht zu verwechseln mit dem Gebiet der effizienten Algorithmen.