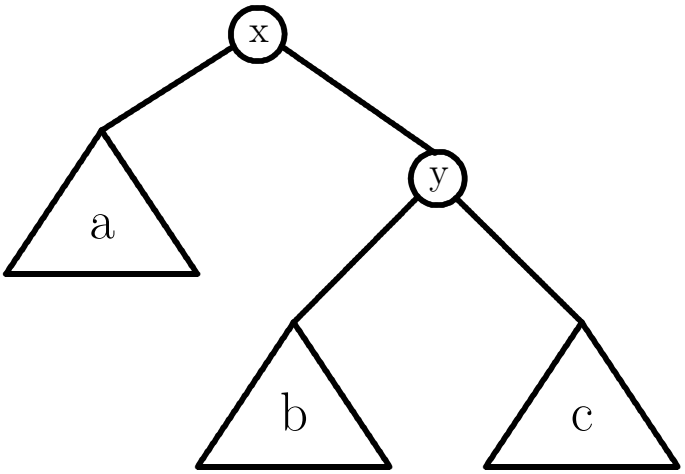
HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Effiziente Algorithmen und Datenstrukturen

Skript zum Modul
“Algorithmen und Datenstrukturen”
(Studiengang AIN)



Prof. Dr. Wolfgang Rülling

Inhaltsverzeichnis

1 Einleitung

Das Modul **Algorithmen und Datenstrukturen** im zweiten Semester des Studiengangs AIN (Algemeine Informatik) beinhaltet die beiden Themengebiete **Effiziente Algorithmen und Datenstrukturen** und **Objektorientierte Programmierung in Java**. Dabei werden grundlegende Programmierkenntnisse, wie sie im ersten Semester vermittelt wurden als bekannt vorausgesetzt.
Im vorliegenden Skript geht es speziell darum, wie man gute Programme entwirft. Ähnlich wie man nach dem Erlernen einer Fremdsprache noch lange keine anspruchsvolle Literatur in dieser Sprache verfassen kann, werden elementare Programmierkenntnisse nicht ausreichen, um gute Software zu erstellen. Es reicht nicht aus, die "Vokabeln" (=Schlüsselwörter), die "Grammatik" (=Syntax) und die Bedeutung (=Semantik) der Anweisungen einer Programmiersprache zu lernen. Zu den weiteren wichtigen Voraussetzungen gehört z.B. ein guter Programmierstil, Techniken der strukturierten Programmierung, der modulare Aufbau großer Programmsysteme, u.s.w. Dies sind wichtige Kenntnisse, die in *Software-Engineering*- bzw. *Software-Technik*- Veranstaltungen vermittelt werden.
In unserer Vorlesung geht es dagegen statt um *Programme* primär um *Algorithmen*. Bevor man sich nämlich die Arbeit macht, einen Algorithmus (d.h. eine "für Menschen verständliche Arbeitsanweisung") als Programm zu formulieren, sollte man sich davon überzeugen, dass der Algorithmus tatsächlich die gewünschten Eigenschaften besitzt. Beispielsweise wird ein "langsamer" Algorithums immer zu einem relativ langsamen Computerprogramm führen. Will man in der Praxis ein vorhandenes Computerprogramm beschleunigen, so sollte man, bevor man Umformulierungen oder Optimierungen des Programms vornimmt, erst untersuchen, ob überhaupt ein geeigneter Algorithmus implementiert wurde. Die Implementierung eines schnelleren Verfahrens wird in der Regel einen viel größeren Einfluss auf die Laufzeit haben, als lokale Programmoptimierungen, die man gegebenenfalls noch zusätzlich vornehmen kann.
In der Vorlesung werden wir deshalb versuchen, die Qualität von Algorithmen zu beurteilen. Dabei werden wir uns als Qualitätsmerkmale auf die *Laufzeit* und den *Speicherplatzbedarf* beschränken. Außerdem werden wir uns nur für "sehr große Probleme" interessieren. Während es nämlich beispielsweise beim Sortieren von 10 Zahlen völlig irrelvant ist, mit welchen Methoden man die Daten sortiert, weil das Ergebnis immer sehr schnell vorliegen wird, ergeben sich bei sehr großen Datenmengen bei den verschiedenen Sortierverfahren extreme Laufzeitunterschiede. Daher ist es nach der Programmierung keineswegs ausreichend die Programme anhand von kleinen Beispielen auszuprobieren. Vielmehr sollte man auch untersuchen, wie sich die Programme bei großen Aufgaben verhalten.
Als Beispiel denke man etwa an das exakte Rechnen mit extrem großen

Zahlen mit z.B. mehreren hundert Dezimalstellen. Es gewinnt derzeit immer mehr an Bedeutung, weil übliche Daten-Verschlüsselungsmethoden oft darauf basieren, dass es schwierig ist, große Zahlen in Primfaktoren zu zerlegen. Wenn es irgendwie gelingt, viel schneller mit großen Zahlen umzugehen, werden Passwörter leichter "geknackt" werden können und viele Sicherungssysteme (EC-Karten, Telefonkarten, u.s.w) werden unbrauchbar werden.
Es gibt also viele Gründe sich damit zu beschäftigen, wie man effizient mit großen Datenmengen umgehen kann. In der Vorlesung werden wir uns daher mit folgenden Themenbereichen beschäftigen.

Typische gute Algorithmen: Anhand typischer Beispiele werden wir schnelle Algorithmen kennenlernen. Die dabei kennengelernten Methoden können dann bei sehr verschiedenen praxisrelevanten Problemen eingesetzt werden.

Effizienter Zugriff auf Daten: Es stellt sich heraus, dass die Geschwindigkeit guter Algorithmen häufig nur durch einen besonders geschickten Zugriff auf die Daten erreicht wird. Beispielsweise wird man es sich nicht leisten können, relevante Daten bei jedem Zugriff "suchen" zu müssen.

Spezielle Baumstrukturen: Wir werden sehen, dass bei der Datenspeicherung Baumstrukturen eine besondere Rolle spielen. Auf sie wird deshalb näher eingegangen.

Arbeiten mit Graphen: Als Verallgemeinerung von Baumstrukturen lernen wir Graphen kennen. In der Praxis lassen sich überraschend viele Problemstellungen mit Hilfe von Graphen formulieren, so dass Graphenalgorithmen eine sehr große Bedeutung haben.

Programmiertechniken: Bei der Vorstellung der Algorithmen werden auch besondere häufig verwendete Techniken zur Realisierung schneller Programme vorgestellt. Dazu gehört beispielsweise das Arbeiten mit verketteten Daten und das Schreiben rekursiv arbeitender Programme.

2 Laufzeitabschätzungen von Algorithmen

Um die Qualität von Algorithmen zu beurteilen, müssen wir ihre Ausführungszeit bestimmen. Prinzipiell könnte man dabei so vorgehen, dass man die Algorithmen zunächst durch Computerprogramme realisiert und dann deren Laufzeiten anhand von Beispielanwendungen bestimmt. Das wäre aber viel zu aufwendig, da wir uns ja die Programmierarbeit für unnötig schlechte Algorithmen sparen wollen. Wir müssen also die Laufzeit bereits vor der Programmierung abschätzen. Dies kann beispielsweise dadurch geschehen, dass wir einfach zählen, wieviele elementare Anweisungen bei der Ausführung

ausgeführt werden müssen. Durch Gewichtung mit den jeweiligen CPU-Zeiten pro Anweisung könnte man daraus sogar recht genaue Ausführungszeiten für einen vorgegebenen Rechnertyp ermitteln. Allerdings gibt man sich in der Praxis mit recht groben Abschätzungen zufrieden, bei denen die Geschwindigkeit der Rechner, die sich ohnehin jährlich erhöht, gar nicht eingeht. Um die Vorgehensweise zu erläutern, betrachten wir ein erstes sehr einfaches Beispiel.

2.1 Beispiel: Einfaches Verfahren zur Berechnung von x^n für reelles x und positives ganzes n

Um den Algorithmus zur Berechnung von x^n zu formulieren, benutzen wir zur Vereinfachung (und Übung) die Notation der Programmiersprache JAVA (bzw. C++). Beachten Sie aber bitte, dass es hier nicht auf programiertechnische Details, sondern nur auf eine allgemein verständliche Darstellung des Verfahrens ankommt. In späteren Beispielen werden wir daher nur eine JAVA (bzw. C++)-ähnliche Notation verwenden und Details weglassen. (Tatsächlich könnten wir auch jede andere Sprache benutzen.)

```
double hoch (double x, int n)
{ // berechnet x**n
  double y=1;
  for (int i=1; i<=n; ++i)
     y = y*x;
  return y;
}
```

Da uns die Laufzeit für große Aufgabenstellungen interessiert, macht es keinen Sinn, eine Zeitangabe für ein konkretes Zahlenbeispiel, wie etwa $x = 2$ oder $n = 3$, anzugeben. Deshalb müssen wir folgende allgemeinere Fragestellung bearbeiten:
Welche Laufzeit hat dieser Algorithmus in Abhängigkeit von n?
Wir zählen zunächst die auszuführenden Anweisungen separat nach verschiedenen Befehlstypen

• Zuweisungen:	2+1+(n+1)+n+1
• Additionen:	n
• Multiplikationen:	n
• Testabfragen:	n+1
Gesamt:	$5 \cdot n + 6$

Die genaue Zählung der Anweisungen ist etwas schwierig. Beispielsweise wurden bei den Zuweisungen die Initialisierung von `y`, $n + 1$ Wertzuweisungen an `i` (bis $i > n$ gilt), n Zuweisungen an `y` nach den Multiplikationen und schließlich die Ergebnisrückgabe mit dem `return`-Statement gezählt. Außerden sollte man auch die Übergabe der Parameter `x` und `n` mitzählen, da bei

call by value-Parametern Kopien angelegt werden. Ähnliche Ungenauigkeiten treten auch bei den anderen Anweisungstypen auf.
Bei der Summation aller Anweisungen gehen wir davon aus, dass sämliche Anweisungstypen mit gleicher Geschwindigkeit ausgeführt werden können. In der Praxis kann es jedoch sein, dass die Multiplikation länger dauert, weil beispielsweise kein Co-Prozessor vorhanden ist. Außerdem wird der Zugriff auf eine Konstante in der Regel schneller sein, als der Zugriff auf eine Variable.
Tatsächlich kommt es für unsere Zwecke nicht auf die genaue Zählweise an, da wir uns ohnehin nur für die größenordnungsmäßige Laufzeit in Abhängigkeit von n interessieren.

2.2 Spezielle Notationen

Um das größenordnungsmäßige Verhalten von Funktionen darzustellen, führen wir eine besondere Notation ein.

Es gilt
$$T(n) = \mathrm{O}\left(f(n)\right) \qquad \text{(gelesen: “groß O von } f(n)\text{”)}$$
falls es eine Konstante $c > 0$ gibt, so dass für hinreichend große n gilt:
$$T(n) \leq c \cdot f(n)$$

$T(n) = \mathrm{O}(f(n))$ bedeutet also, dass die Funktion $f(n)$ für große n bis auf einen konstanten Faktor eine obere Schranke für $T(n)$ ist. Man spricht daher auch von einer *asymptotischen worst case* Abschätzung.

Für die Funktion `hoch` mit der Laufzeit $T_{\text{hoch}}(n) = 5 \cdot n + 6$ ergibt sich nach dieser Definition $T_{\text{hoch}}(n) = 5n + 6 = \mathrm{O}(n)$. D.h. die Abschätzung hat ergeben, dass die Laufzeit T_{hoch} größenordnungsmäßig linear mit n wächst. Diese O-Notation führt offensichtlich dazu, dass konstante Faktoren und konstante Summanden vernachlässigt werden können.
Dazu ein paar weitere Beispiele:

$$\begin{array}{lcl}
3 \cdot n^2 + 5 \cdot n - 19 & = & \mathrm{O}(n^2) \\
13 \cdot n^3 - 4 \cdot n^2 + 2 \cdot n^5 & = & \mathrm{O}(n^5) \\
\mid \sin n \mid & = & \mathrm{O}(1) \\
n \cdot \mid \sin n \mid & = & \mathrm{O}(n) \\
n^2 \cdot \mid \sin n \mid + n^3 & = & \mathrm{O}(n^3) \\
n + \log_2 n & = & \mathrm{O}(n) \\
n \cdot \log_2 n + 3 \cdot n & = & \mathrm{O}(n \cdot \log n) \\
n^2 \cdot \log_2 n + \underbrace{n \log_2 n^3}_{3n \log_2 n} & = & \mathrm{O}(n^2 \cdot \log n)
\end{array}$$

Manchmal interessiert man sich auch für untere Schranken.

Es gilt

$$T(n) = \Omega\left(f(n)\right)$$

falls es ein $c > 0$ gibt, so dass für ein hinreichend großes n folgendes gilt:

$$T(n) \geq c \cdot f(n)$$

Auch hierzu ein paar Beispiele:

$$\begin{array}{lcll} 3 \cdot n^2 + 5 \cdot n - 19 & = & \Omega(n^2) & \\ 13 \cdot n^3 - 4 \cdot n^2 + 2 \cdot n^5 & = & \Omega(n^5) & \\ |\sin n| & = & \Omega(1) & \text{statt } \Omega(0) \\ n \cdot |\sin n| & = & \Omega(1) & \text{statt } \Omega(0) \\ n^2 \cdot |\sin n| + n^3 & = & \Omega(n^3) & \\ n + \log_2 \cdot n & = & \Omega(n) & \end{array}$$

$$T_1(n) = \begin{cases} 5n^2 + 3n & : \text{für gerade n} \\ 4n + 2 & : \text{für ungerade n} \end{cases}$$

$$\begin{array}{lcl} T_1(n) & = & \mathrm{O}(n^2) \\ T_1(n) & = & \Omega(n) \end{array}$$

$$\begin{array}{lcl} T_2(n) & = & n + n^2 \cdot |\sin n| \\ T_2(n) & = & O(n^2) \\ T_2(n) & = & \Omega(n) \end{array}$$

Im Fall $T(n) = O(f(n))$ und $T(n) = \Omega(f(n))$ sind T und f von der gleichen Ordnung. Dafür schreibt man:

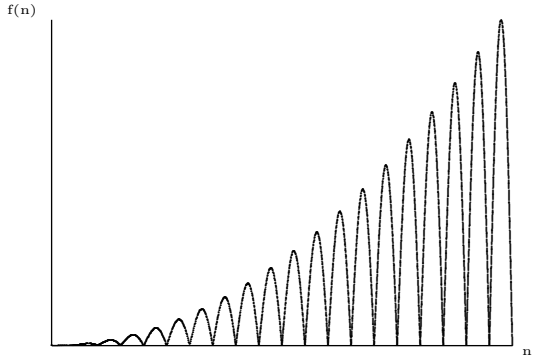
$$T(n) = \Theta\left(f(n)\right)$$

Beispiel:

$$\begin{array}{lcl} n + |\sin n| & = & \mathrm{O}(n) \\ n + |\sin n| & = & \Omega(n) \\ n + |\sin n| & = & \Theta(n) \end{array}$$

Für zwei Beispiele sollen die Laufzeitabschätzungen auch noch graphisch dargestellt werden.

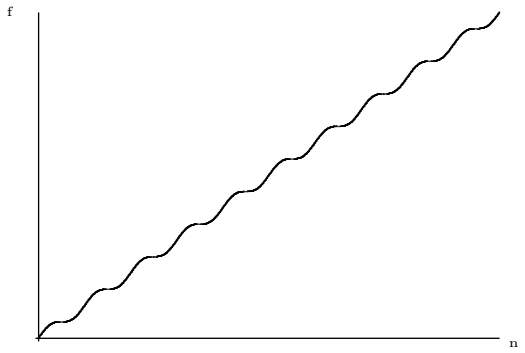
Beispiel: $f(n) = n^2 \cdot |sin(n)|$



Hier gilt

$$\begin{array}{rcl} f(n) & \leq & n^2 \cdot 1 \quad \Rightarrow \quad f(n) = \mathrm{O}(n^2) \\ f(n) & \geq & 0 \qquad \Rightarrow \quad f(n) = \Omega(1) \end{array}$$

Beispiel: $f(n) = n + sin(n)$



$$\begin{array}{rcl} f(n) & \leq & n+1 \quad \Rightarrow \quad f(n) = \mathrm{O}(n) \\ f(n) & \geq & n-1 \quad \Rightarrow \quad f(n) = \Omega(n) \\ & & \Rightarrow \quad f(n) = \Theta(n) \end{array}$$

2.3 Verbessertes Verfahren zur Berechnung von x^n für reelles x und positives ganzes n

Die bisherige Realisierung der Funktion `hoch` hatte eine lineare Laufzeit in n. Einfache Optimierungsversuche werden lediglich Änderungen der Konstanten in der Funktion $T_{\text{hoch}} = 5n+6$ bewirken, so dass sich an der größenordnungsmäßigen Laufzeit nichts ändern wird.
Wenn wir eine größenordnungsmäßige Verbesserung der Laufzeit wünschen, müssen wir offensichtlich eine andere Berechnungsmethode, d.h. einen anderen Algorithmus benutzen. Im folgenden wird deshalb der "square-and-multiply"-Algorithmus vorgestellt.

```
double hoch (double x, int n)
{ if (n==0)
    return 1;
  else
    if (n%2==0)
      return (sqr(hoch(x,n/2)));
    else
       return (x * sqr(hoch(x,(n-1)/2)));
}
```

Dieses rekursive Verfahren basiert auf der *Divide-and-Conquer*-Methode, bei der man ein großes Problem in mehrere kleine Probleme zerlegt, die dann separat gelöst werden. Will man beispielsweise x^{18} berechnen, so kann man statt dessen auch zunächst das kleinere Problem x^9 lösen und dessen Ergebnis quadrieren. (Quadrieren= eine zusätzliche Multiplikation). x^9 vereinfacht man entsprechend zu $x^9 = x * \text{sqr}(x^4)$. Mit dieser Methode reduzieren sich die 18 Multiplikationen von

$$x^{18} = \underbrace{x \cdot x \cdot \ldots \cdot x}_{\text{18-mal}}$$

auf 5 Multiplikationen im folgenden Ausdruck:

$$x^{18} = \Big(x \cdot \Big(\big((x^2)^2\big)^2\Big)\Big)^2$$

Die Laufzeitabschätzung für die rekursive Funktionsrealisierung ist allerdings nicht ganz einfach. Die verschachtelten `if`-Anweisungen führen zu entsprechenden Fallunterscheidungen für die Laufzeit und der rekursive Aufruf der Funktion `hoch` führt zu einer rekursiven Darstellung der Laufzeit:

$$T(n) = \begin{cases} 4 & \text{für } n = 0 \\ 10 + T(\frac{n}{2}) & \text{für gerade } n > 0 \\ 13 + T(\frac{n-1}{2}) & \text{für ungerade } n > 0 \end{cases}$$

Diese Funktion T können wir etwas vereinfachen und gleichzeitig nach oben abschätzen, so dass wir eine obere Schranke $\tilde{T}(n) \geq T(n)$ erhalten.

$$\tilde{T}(n) = \begin{cases} 4 & \text{für } n = 0 \\ 13 + \tilde{T}(\lfloor \frac{n}{2} \rfloor) & \text{für } n > 0 \end{cases}$$

Dabei wurden die beiden Fälle "n gerade, $n > 0$" und "n ungerade, $n > 0$" einfach mit der ungünstigeren Laufzeit zusammengefasst. Außerdem wurde der rekursive Aufruf von $T(\ldots)$ durch die Abschätzung nach oben $\tilde{T}(\ldots)$ ersetzt.

Anmerkung: Die *Gauß-Klammern* im Ausdruck $\lfloor \frac{n}{2} \rfloor$ bedeuten, dass nach der Berechnung des Bruchs auf die nächste ganze Zahl *abgerundet* werden soll. Entsprechend steht "$\lceil \ldots \rceil$" für das *Aufrunden*.

Die Rekursion lässt sich leicht auflösen. Dazu betrachten wir zunächst ein paar Zahlenbeispiele.

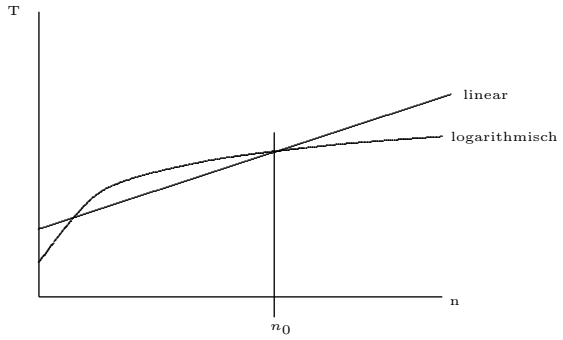
$$\begin{aligned}
\tilde{T}(10) &= 13 + \tilde{T}(5) \\
&= 13 + 13 + \tilde{T}(2) \\
&= 13 + 13 + 13 + \tilde{T}(1) \\
&= 13 + 13 + 13 + 13 + 4 = 4 \cdot 13 + 4 \\
\tilde{T}(15) &= 13 + \tilde{T}(7) \\
&= 13 + 13 + \tilde{T}(3) \\
&= 13 + 13 + 13 + 13 + 4 = 4 \cdot 13 + 4 \\
\tilde{T}(16) &= 13 + \tilde{T}(8) \\
&= 5 \cdot 13 + 4
\end{aligned}$$

Offensichtlich gilt also $\tilde{T}(n) = 13 * (\lfloor \log_2 n \rfloor + 1) + 4$ und damit $T(n) = O(\log n)$.
Mit der *Divide and Conquer* Technik haben wir also die lineare Laufzeit der Funktion `hoch` auf eine bessere logarithmische Laufzeit reduziert.
Wir sehen uns den Laufzeitunterschied an einem Zahlenbeispiel an.

Laufzeitvergleich:

n	5n +6	$13 \cdot(\lfloor \log_2 n \rfloor +1)+4$
1	11	17
2	16	30
4	26	43
100	506	95
1000	5006	134

Während bei kleinen Argumenten n das alte Verfahren schneller ist, ergibt sich für große Argumente ein offensichtlicher Geschwindigkeitsvorteil für das neue Verfahren. An diesem Phänomen ändert sich auch dann nichts, wenn wir annehmen, dass die Konstanten nicht korrekt abgeschätzt wurden. Die logarithmische Laufzeit ist immer größenordnungsmäßig besser als die lineare Laufzeit. In der Praxis kann man (experimentell) einen Wert n_0 bestimmen, so dass das neue Verfahren für $n > n_0$ besser ist.



2.4 Schnelle Realisierung der Funktion `hoch` ohne Rekursion

Die obige rekursive Beschreibung der Funktions `hoch` ergab sich direkt aus der verwendeten *Divide-and-Conquer*-Technik. Dies bedeutet aber nicht, dass man gute Laufzeiten nur durch rekursive Programmierung erreichen kann. Tatsächlich lässt sich die gleiche Laufzeit ohne Rekursion erreichen. Zur Vervollständigung ist im folgenden eine nicht-rekursive Implementierung der Funktion `hoch` mit Laufzeit $O(\log(n))$ angegeben.

```
double hoch (double x, int n) {
  double y=1;
  while (n>0) {
    if (n%2==1)
      y *= x;
    n = n/2;
    x = x*x;
  };
  return y;
}
```

Aus dieser nicht-rekursiven Implementierung erkennt man direkt die logarithmische Laufzeit, da die `while`-Schleife so oft durchlaufen wird, wie man n halbieren muss, bis sich $n = 0$ ergibt.

$$T = \text{const} + \text{const} \cdot (\lfloor \log_2 n \rfloor + 1) = O(\log n).$$

Im allgemeinen liegt bei Verwendung der *Divide and Conquer* Technik eine rekursive Beschreibung des Algorithmus vor. Typischerweise erhält man dann für die Laufzeit eine rekursive Darstellung der folgenden Form:

$$T(n) = \begin{cases} a & \text{für n} = 1 \\ b \cdot T(\frac{n}{c}) + d & \text{für n} > 1 \end{cases}$$

Dabei wird ein Problem der Größe n in Teilprobleme der Größe $\frac{n}{c}$ zerlegt. Die Anzahl dieser Teilprobleme ist b und d ist der Aufwand für das Zusammensetzen der Lösung. Im Beispiel der Funktion `hoch` galt $c = 2$ und $b = 1$ (einmal muss man $x^{\frac{n}{2}}$ ausrechnen). Bei der Auflösung dieser Rekursion sind 2 Fälle zu unterscheiden:

$$\begin{aligned} b = 1 &: \text{ dann gilt } T(n) = O(\log n) \\ b > 1 &: \text{ dann gilt } T(n) = O(n^{\log_c b}) \end{aligned}$$

In der Praxis braucht man oft zum Zusammensetzen der Teillösungen einen höheren als konstanten Aufwand. Bei linearem Aufwand ergibt sich folgende Formel:

$$T(n) = \begin{cases} a & \text{für } n = 1 \\ b \cdot T(\frac{n}{c}) + d \cdot n & \text{für } n > 1 \end{cases}$$

dann gilt

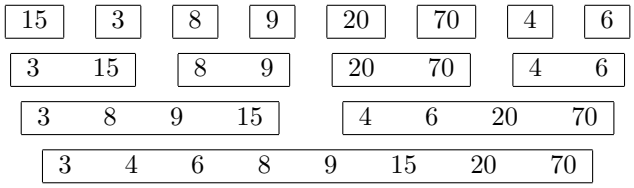
$$T(n) = \begin{cases} O(n) & \text{falls } b < c \\ O(n \log n) & \text{falls } b = c \\ O(n^{\log_c b}) & \text{falls } b > c \end{cases}$$

2.5 Divide-and-Conquer Technik beim Sortieren

Wir wollen nun als eine weitere Anwendung der *Divide and Conquer* Technik ein Sortierverfahren betrachten. Das Verfahren geht so vor, dass die zu sortierende Folge in zwei (etwa) gleich große Teilfolgen aufgeteilt wird, die dann getrennt sortiert werden. Anschließend werden die beiden sortierten Teilfolgen zusammengemischt. Dieses Verfahren nennt man *merge_sort* (*Sortieren durch Mischen*).

(15 , 3 , 8 , 9 , 20 , 70 , 4 , 6)	
↓	Aufteilung in zwei Teilfolgen
(15 , 3 , 8 , 9) (20 , 70 , 4 , 6)	
↓	Sortieren der Teilfolgen (rekursiv)
(3 , 8 , 9 , 15) (4 , 6 , 20 , 70)	
↓	Zusammensetzen der Teilfolgen
(3 , 4 , 6 , 8 , 9 , 15 , 20 , 70)	

Zum Sortieren der Teilfolgen kann man wiederum das gleiche Verfahren (rekursiv) benutzen, bis man so kurze Teilfolgen erhält, die einfach zu sortieren sind. Im Beispiel soll die Aufteilung bis zu Folgen der Länge 1 durchgeführt werden, da diese naturgemäß bereits sortiert sind.



Für die Implementierung dieses Verfahrens, gehen wir davon aus, dass die zu sortierenden Daten in einem Feld $x[0] \ldots x[n-1]$ (oder allgemeiner im Bereich $x[l] \ldots x[r]$) stehen.

```
void merge_sort(int l, int r)
// sortiert den Bereich x[l]..x[r]
{ if (r>l)
   {int m= (l+r)/2;
    merge_sort( l, m );  // linken Teil sortieren
    merge_sort( m+1, r); // rechten Teil sortieren
    // Zusammenmischen der beiden Teilfelder
    int[] z = new int[r-l+1]; // Hilfsfeld z
    int i=l; int j=m+1; int k=0;
    while ((i<=m) && (j<=r))
      if ( x[i] < x[j] )
        z[k++] = x[i++];
      else
        z[k++] = x[j++];
    while (i<=m)   z[k++] = x[i++];
    while (j<=r)   z[k++] = x[j++];
    for (k=0; k<=r-l; k++)
      x[l+k]=z[k];
   }
}
```

Als Laufzeit ergibt sich folgende rekursive Darstellung in Abhängigkeit von der Anzahl $n = r - l + 1$ der zu sortierenden Daten.

$$T(n) = \begin{cases} \text{const} & \text{für } n = 1 \\ 2 \cdot T(\frac{n}{2}) + \underbrace{d \cdot n}_{\text{Zusammenmischen}} & \text{für } n > 1 \end{cases}$$

Mit $b = c = 2$ ergibt sich aus obigen Formeln $T(n) = \mathrm{O}(n \log n)$. Dies ist auch die optimale Laufzeit für das Sortieren von n Daten.

3 Multiplikation großer Zahlen

Wir wollen uns nun überlegen, wie man zwei extrem große Zahlen möglichst schnell multiplizieren kann. Dabei setzen wir voraus, dass die Zahlen so groß sind, dass die Multiplikationshardware des Rechners nicht verwendbar ist. Als Anwendung denke man beispielsweise an Verschlüsselungsprobleme, bei denen Dezimalzahlen mit über hundert Ziffern verarbeitet werden müssen.

3.1 Die Schulmethode

Zunächst betrachten wir als allgemein bekannten Multiplikations-Algorithmus die sogenannte "Schulmethode" für n-stellige Dezimalzahlen.

$$
\begin{array}{r}
(a_{n-1}, a_{n-2}, \cdots, a_1, a_0) \cdot (b_{n-1}, b_{n-2}, \cdots, b_1, b_0) \\
\hline
(a_{n-1}, a_{n-2}, \ldots, a_1, a_0) \cdot (b_0) \\
(a_{n-1}, a_{n-2}, \ldots, a_1, a_0) \cdot (b_1) \\
\vdots \\
(a_{n-1}, a_{n-2}, \ldots, a_1, a_0) \cdot (b_{n-2}) \\
(a_{n-1}, a_{n-2}, \cdots, a_1, a_0) \cdot (b_{n-1}) \\
\hline
\sum_{i=0}^{n-1} a\ b_i \cdot 10^i \\
\hline\hline
\end{array}
$$

Formelmäßig kann das Verfahren für Dezimalzahlen folgendermaßen dargestellt werden.

$$c = \sum_{i=0}^{n-1} a \cdot b_i \cdot 10^i$$

Man beachte, dass die Methode auch für andere Zahlsysteme verwendbar ist. Beispielsweise sieht die Multiplikation $13_{\text{dez}} * 5_{\text{dez}} = 65_{\text{dez}}$ im Binärsystem nach der Schulmethode folgendermaßen aus.

0	1	1	0	1		$*$		1	0	1
					0	1	1	0	1	
				0	0	0	0	0		
			0	1	1	0	1			
			1	0	0	0	0	0	1	

Die allgemeine Formel für die Berechnung im Binärsystem lautet.

$$c = \sum_{i=0}^{n-1} a \cdot b_i \cdot 2^i$$

Welche Laufzeit hat das Verfahren, wenn zwei Binärzahlen mit jeweils n Stellen multipliziert werden sollen?
Wenn wir davon ausgehen, dass die Shift-Operationen durch eine geschickte Indizierung implementiert werden können und die Multiplikation mit 0 bzw. 1 nur einer `if`-Anweisung entspricht, besteht der Aufwand nur im Summieren der Terme $a \cdot b_i$. Dies sind n Terme mit jeweils n Binärziffern, so dass sich insgesamt

$$T(n) = O(n^2)$$

ergibt. D.h. die Multiplikation hat quadratische Laufzeit in der Länge der Zahldarstellung.

3.2 Karatsuba-Algorithmus

Wir betrachten nun einen schnelleren Algorithmus zu Multiplikations zweier n-stelliger (Binär-)Zahlen. Zur Vereinfachung der Darstellung setzen wir voraus, dass $n = 2^k$ gilt, d.h., dass n eine 2er-Potenz ist. Die Grundidee besteht darin, die n-stelligen Zahlen in die höherwertigen Bits und die niederwertigen Bits zu zerlegen und dann die n-stellige Multiplikation auf die Multiplikation von Zahlen der Länge $\frac{n}{2}$ zurückzuführen. Es wird also die *Divide-and-Conquer*-Methode benutzt.

$$a = \underbrace{a_{n-1}, a_{n-2}, \cdots, a_{\frac{n}{2}}}_{a_H}, \underbrace{a_{\frac{n}{2}-1}, \cdots, a_2, a_1, a_0}_{a_L}$$

$$\begin{aligned} a = &\quad a_H \cdot 2^{\frac{n}{2}} + a_L \\ b = &\quad b_H \cdot 2^{\frac{n}{2}} + b_L \\ a \cdot b = &\quad (a_H \cdot b_H)2^n + (a_H \cdot b_L + a_L \cdot b_H)2^{\frac{n}{2}} + (a_L \cdot b_L) \end{aligned}$$

Auf diese Weise würden wir eine n-stellige Multiplikation durch 4 Multiplikationen der Länge $\frac{n}{2}$ ersetzen. Es geht aber auch noch etwas geschickter. Wegen

$$(a_H \cdot b_L + a_L \cdot b_H) = (a_H + a_L) \cdot (b_H + b_L) - a_H \cdot b_H - a_L \cdot b_L$$

können wir sogar mit 3 Multiplikation auskommen:

$$\begin{aligned} x = &\quad a_H \cdot b_H; \\ y = &\quad a_L \cdot b_L; \\ z = &\quad (a_H + a_L) \cdot (b_H + b_L); \\ \text{Ergebnis} = &\quad x \cdot 2^n + (z - x - y) \cdot 2^{\frac{n}{2}} + y \end{aligned}$$

Wir brauchen also nur 3 Multiplikationen (für Zahlen der Länge $\frac{n}{2}$) und einige Additionen und Shift-Operationen. Daraus lässt sich sofort eine rekursive Darstellung für die Gesamtlaufzeit des *Karatsuba*-Verfahrens herleiten.

$$T(n) = \begin{cases} \text{const} & \text{für } n \leq k_0,\ n \text{ klein genug} \\ \underbrace{3 \cdot T\left(\frac{n}{2}\right)}_{\text{Berechnung von x,y,z}} + \underbrace{\mathrm{O}(n)}_{\text{Additionen}} & \text{sonst} \end{cases}$$

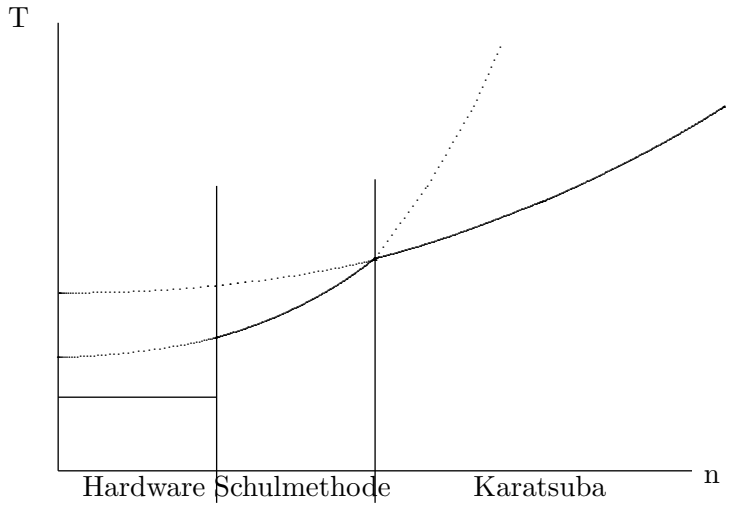
Es gilt der Fall b = 3 und c = 2 bei linearem Aufwand zum Zusammensetzen der Teillösungen. Als Lösung erhalten wir also

$$T(n) = O(n^{\log_2 3}) = O(n^{1{,}59})$$

Das ist eine Verbesserung gegenüber dem Schulalgorithmus, bei dem wir die Laufzeit $T(n) = O(n^2)$ haben.

1. Hinweis: Je nach Implementierung sollte man die kritische Datenlänge n_0 bestimmen, ab der der *Karatsuba*-Algorithmus der Schulmethode überlegen ist. Für $n < n_0$ wird dann die Schulmethode und bei $n \geq n_0$ die Karatsuba-Zerlegung benutzt. (Es gilt z.B. $n_0 \approx 100$)

2. Hinweis: Noch geschickter ist eine Unterscheidung in drei Fälle: Bei sehr kleinen Daten benutzt man eine Hardwaremultiplikation, bei etwas größeren die Schulmethode und bei großen Daten die rekursive Zerlegung.



4 Multiplikation großer Matrizen

Einen ganz ähnlichen Zerlegungstrick kann man beispielsweise bei der Matrizenmultiplikation benutzen.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

Die übliche Rechenmethode geht folgendermaßen vor:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \ldots + a_{in} b_{nj}$$

Auf diese Weise braucht man	$4 \cdot 2$	Multiplikationen
	4	Additionen
Bei $n \times n$ - Matrizen ergeben sich :	n^3	Multiplikationen
	$(n^3 - n^2)$	Additionen

Insgesamt ergibt sich also die Laufzeit $T(n) = \mathrm{O}(n^3)$

Beim "Multiplikationsalgorithmus nach *Strassen*" benutzt man für das Produkt von 2×2-Matrizen folgendes Verfahren:

$$\begin{aligned}
h_1 &= (a_{12} - a_{22}) \cdot (b_{21} + b_{22}) \\
h_2 &= (a_{11} + a_{22}) \cdot (b_{11} + b_{22}) \\
h_3 &= (a_{11} - a_{21}) \cdot (b_{11} + b_{12}) \\
h_4 &= (a_{11} + a_{12}) \cdot b_{22} \\
h_5 &= a_{11} \cdot (b_{12} - b_{22}) \\
h_6 &= a_{22} \cdot (b_{21} - b_{11}) \\
h_7 &= (a_{21} + a_{22}) \cdot b_{11} \\
c_{11} &= h_1 + h_2 - h_4 + h_6 \\
c_{12} &= h_4 + h_5 \\
c_{21} &= h_6 + h_7 \\
c_{22} &= h_2 - h_3 + h_5 - h_7
\end{aligned}$$

Die Methode braucht also 7 Multiplikationen und 18 Additionen. Man kann die Methode auf $n \times n$-Matrizen erweitern, indem man die Matrizen jeweils in 4 Quadranten aufteilt.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

Statt also eine Multiplikation mit $n \times n$-Matrizen durchzuführen, kann man 7 Multiplikationen von $\frac{n}{2} \times \frac{n}{2}$ - Matrizen benutzen. Für die Laufzeitabschätzung ergibt sich.

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + \underbrace{18n^2}_{\text{Additionen}}$$

Man erhält schließlich (mit $n = 2^k$):

$$T(n) = \mathrm{O}(7^k) = \mathrm{O}(7^{\log_2 n}) = \mathrm{O}(n^{\log_2 7}) = O(n^{2{,}81})$$

Das ist besser als die Laufzeit $\mathrm{O}(n^3)$ der klassischen Methode.

5 Datenspeicherung in binären Bäumen

Häufig verwendet man zur effizienten Speicherung von Daten Baumstrukturen. Diese Technik soll zunächst in den Abschnitten 5.1 und 5.2 eingeführt werden. Anschließend wird im Abschnitt 5.3 ausführlich darauf eingegangen, wie die Baumstrukturen durch Balancierungsmaßnahmen zu schnellen Zugriffszeiten auf die Daten führen.

5.1 Aufgabenstellung

In einem Wörterbuch sollen Paare (a, b) von Wörtern (Daten) gespeichert werden. Folgende Operationen sollen unterstützt werden.

insert: Einfügen eines neuen Paares (a, b)

translate: Suche nach einem gegebenen Wert a und Ausgabe des dazugehörenden Wertes b

list: Ausgabe aller Paare (a, b) alphabetisch nach den a-Komponenten sortiert

Wie kann man ein Wörterbuch verwalten, so dass diese Operationen alle möglichst schnell ausgeführt werden können?
Als Anwendungsbeispiele für Wörterbücher denke man etwa an eine "Liste" von Rechnerbenutzern mit ihren Passwörtern (braucht das Betriebssystem, um das Einloggen von Anwendern zu kontrollieren) oder an eine "Liste" von Befehlen mit den dazugehörigen Programmen (braucht das Betriebssystem bei der Analyse eingegebener Kommandos).

Lösungsmöglichkeiten

Feld: Man könnte die Paare (a, b) in einem großen Feld speichern und neue Einträge immer hinten anfügen. Wenn man sich dabei die Anzahl der bereits eingetragenen Paare merkt, ist die neue Einfügeposition für *insert* immer bekannt. Das Einfügen geht also extrem schnell. Allerdings muss man beim *translate*-Befehl das ganze Feld durchsuchen. Bei n Eintragungen braucht man dazu im Mittel $\frac{n}{2}$ und im ungünstigsten Fall n Datenvergleiche. Bei großen Wörterbüchern dauert das sehr lange. Die alphabetische Ausgabe ist noch aufwendiger.

sortiertes Feld: Wenn man die Daten sortiert im Feld vorliegen hätte, dann wäre die alphabetische Ausgabe kein Problem mehr und auch die Suche nach einem Eintrag (*translate*) ginge recht schnell. Statt bei der Suche sequentiell alle Einträge zu prüfen, könnte man den mittleren Eintrag im Feld benutzen und sofort entscheiden, ob der gesuchte Wert a in der linken oder der rechten Hälfte des Feldes liegt. Durch Wiederholung erreicht man eine Art "Intervallschachtellung" und braucht

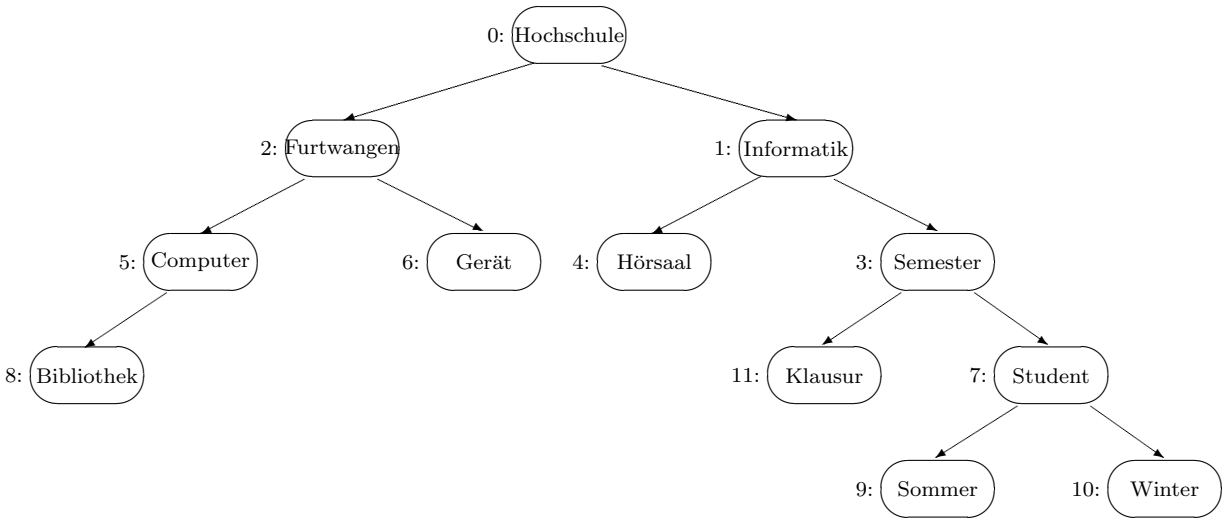
bei n Eintragungen höchstens $\log_2 n$ Vergleiche. Allerdings ist jetzt das Einfügen eines neuen Paares schwierig, weil alle dahinterliegenden Einträge um eine Einheit verschoben werden müssen. Im Mittel sind dies $\frac{n}{2}$ zu verschiebende Werte.

sortierte Liste: Bei einer sortierten Liste benutzt man Zeiger, um von einem Eintrag auf den nächsten zu zeigen. Der Vorteil besteht darin, dass man beim Einfügen keine Elemente verschieben muss, sondern das neue Paar irgendwo speichern kann und nur die Verkettungszeiger ändern muss. Der Nachteil ist jedoch, dass man jetzt keine Möglichkeit mehr hat, direkt auf einen "mittleren" Eintrag der Liste zuzugreifen.

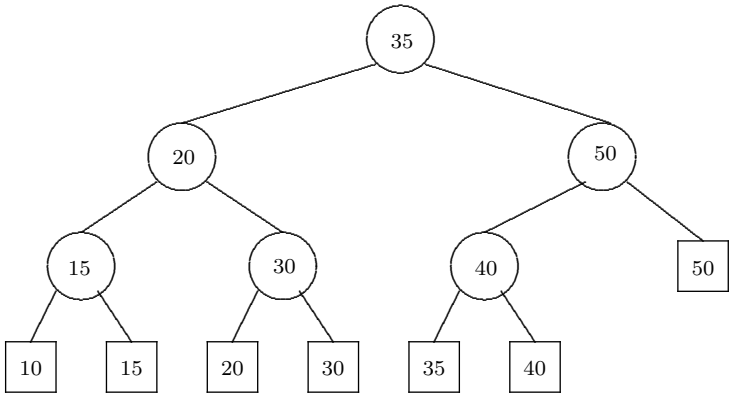
5.2 Lösungsansatz

Wir werden als Lösungsansatz eine *Baumstruktur* verwenden, bei der sämtliche Operationen schnell ausführbar sind. Um die Idee zu verdeutlichen, betrachten wir ein einfaches Beispiel, bei dem nacheinander die folgenden Wörter $a_0 \dots a_{11}$ in das Wörterbuch eingetragen werden sollen. Die Komponenten b_i werden zunächst zur Vereinfachung weggelassen.

0	Hochschule	6	Gerät
1	Informatik	7	Student
2	Furtwangen	8	Bibliothek
3	Semester	9	Sommer
4	Hörsaal	10	Winter
5	Computer	11	Klausur



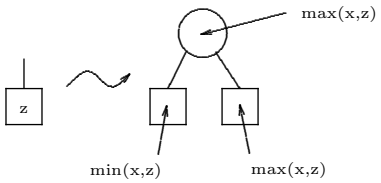
Sofern die Wörter in einer zufälligen Reihenfolge eingegeben werden, ist zu erwarten, dass der Baum relativ gleichmäßig gefüllt ist. Die einzelnen Operationen sollten dann relativ schnell sein, weil z.B. bei der Suche nach einem Eintrag nur die Höhe des Baumes (etwa $\log_2 n$ Stufen) zu durchlaufen ist. Die Entscheidung die Suche nach links oder rechts weiterzuführen, entspricht jeweils etwa einer Halbierung des Suchraums.
Für den Fall, dass die Wörter in einer ungünstigen Reihenfolge eingegeben werden und der Baum ungleichmäßig wird, verschlechtert sich die Laufzeit für die einzelnen Operationen. Um dies zu vermeiden, sollen im Abschnitt 5.3 balancierte Bäume betrachtet werden.
Zuvor modifizieren wir die Baumdarstellung allerdings noch etwas, um auch ein einfaches Löschen von Einträgen zu ermöglichen. Während das Entfernen von Blättern recht einfach ist, weil der Rest des Baumes nicht betroffen ist, würde das Entfernen von inneren Knoten dazu führen, dass der Baum auseinanderfällt. Als Lösung des Problemes werden wir im folgenden die Daten grundsätzlich nur noch in den Blättern des Baumes speichern. Die folgende Skizze zeigt dazu ein Beispiel. Als Daten verwenden wir diesmal Zahlenwerte.



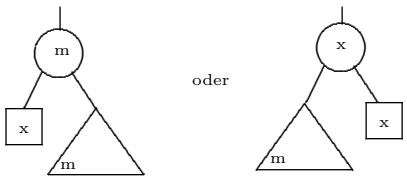
Um auf die Daten zuzugreifen, durchläuft man den Baum bei der Wurzel beginnend bis zum gewünschten Blatt. Dazu müssen natürlich Hilfsinformation in den internen Knoten vorhanden sein, mit denen entschieden werden kann, ob die Suche im linken oder rechten Teilbaum fortgesetzt werden kann. Beispielsweise können wir dazu in jedem inneren Knoten einen Wert m ablegen, der den kleinsten Wert des rechten Teilbaums darstellt. Ist der gesuchte Wert x kleiner, geht man nach links, anderenfalls nach rechts (siehe obige Skizze).
Das Einfügen oder Entfernen von Einträgen kann dann folgendermaßen erfolgen.

Einfügen von x: Der Baum wird bei der Wurzel beginnend nach dem Wert x durchsucht. Die Suche endet an einem Blatt mit Eintrag z. Im Fall $z = x$ ist der gewünschte Eintrag bereits vorhanden und es ist nichts weiter

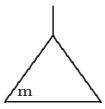
zu tun. Anderenfalls ist das Blatt mit Wert z gemäß folgender Skizze durch einen neuen inneren Knoten mit zwei Blättern zu ersetzen. Das linke Blatt erhält den Wert $\min(x, z)$ und das rechte Blatt den Wert $\max(x, z)$.



Entfernen von x: Der Baum wird bei der Wurzel beginnend nach dem Wert x durchsucht. Ist er nicht vorhanden, so ist nichts zu tun. Anderenfalls ist das erreichte Blatt mit Wert x das linke oder rechte Blatt eines inneren Knotens gemäß folgender Skizze.



Man entfernt das Blatt und ersetzt seinen Vaterknoten durch dessen anderen Teilbaum, d.h. man ersetzt den Vaterknoten durch den "Bruder von x" (siehe folgende Skizze).



Bei dieser Vorgehensweise gibt es noch ein kleines Problem. Es kann nämlich vorkommen, dass während der Suche nach dem Blatt x ein innerer Knoten mit Hilfsinformation x erreicht wird. Diese Hilfsinformation muss natürlich nach dem Entfernen des Blattes x aktualisiert werden. Deshalb muss man sich diesen inneren Knoten merken und den Hilfswert zum Schluss durch den Wert m (= kleinster Wert des Bruders von x) gemäß obiger Skizze ersetzen. Man kann sich leicht davon überzeugen, dass jeder Wert höchstens einmal als Hilfsinformation auftritt, so dass diese Zusatzaktion tatsächlich nur einen Knoten betrifft.

Bei diesen Operationen ändert sich natürlich die Balancierung des Baumes, so dass gegebenenfalls Korrekturschritte erforderlich werden. Bevor wir uns damit genauer befassen, soll der "Balance"-Begriff definiert werden.

5.3 Balancierte Bäume

Definition: Sei ein T ein *binärer Baum* mit linkem Teilbaum T_l und rechtem Teilbaum T_r.

Dann heißt $\rho(T) = \frac{|T_l|}{|T_l|+|T_r|}$ die *Balance* von T.

Dabei bedeutet $|T|$ die Anzahl der Blätter von T.

Ein Baum heißt von *beschränkter Balance* α, wenn für jeden Unterbaum T' von T gilt:

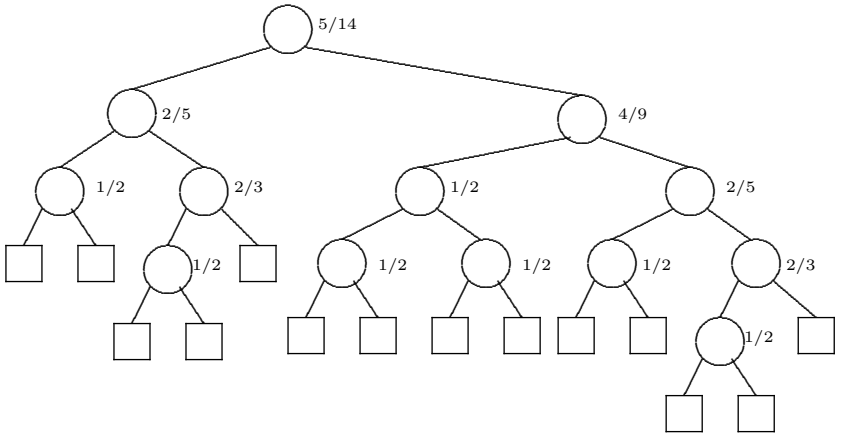
$$\alpha \leq \rho(T') \leq 1 - \alpha \quad \text{mit } 0 < \alpha \leq \frac{1}{2}$$

Die Bedeutung dieser Definition ergibt sich aus folgendem Satz.

Satz: Bäume von *beschränkter Balance* haben eine logarithmische Höhe.

Man beachte dabei, dass der Balancewert α für die Bäume fest vorgegeben sein muss. $\alpha = 0.5$ steht für eine perfekte Balancierung und mit kleineren Werten wird die Balancierung immer schlechter, was sich bei der größenordungsmäßigen Höhe in einem schlechteren Faktor auswirkt.

Beispiel: Ermittlung der Balance eines Baumes



In diesem Beispiel liegen die Werte zwischen $\frac{1}{3}$ und $\frac{2}{3}$. Der Baum ist also von *beschränkter Balance* mit $\alpha = \frac{1}{3}$.

Beim Einfügen oder Löschen von Daten in einem balancierten Baum ändern sich natürlich die Balancewerte auf dem verwendeten Suchpfad. Deshalb durchläuft man anschließend den Pfad von unten nach oben und ermittelt dabei die neuen Balance-Werte. Solange diese Werte im Bereich $[\alpha, 1 - \alpha]$ liegen, ist nichts weiter zu tun. Falls die Balance eines Knotens ungültig

wird, muss man den Schaden durch Rebalancierung beheben. Dazu gibt es im wesentlichen 2 Transformationen. Es handelt sich dabei um die *einfache Rotation nach links* und die *Doppelrotation nach links*. Beide Methoden werden benutzt, wenn sich an einem Knoten zuviele Blätter im rechten Teilbaum befinden. Als Ergebnis werden einige der Blätter nach links gebracht. Die gleichen Methoden gibt es natürlich auch nach rechts. Aus Symmetriegründen werden hier jedoch nur die Rotationen nach links erläutert.

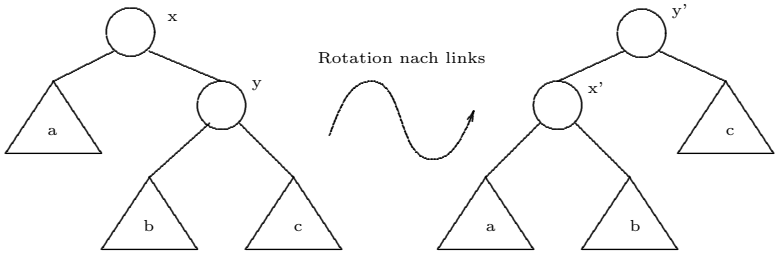
5.3.1 Einfache Rotation

Durch die *einfache Rotation nach links* ändern sich die Balancewerte folgendermaßen:

$$\begin{aligned} \text{vorher:} \quad \rho(x) &= \frac{|a|}{|a|+|b|+|c|} \\ \rho(y) &= \frac{|b|}{|b|+|c|} \end{aligned}$$

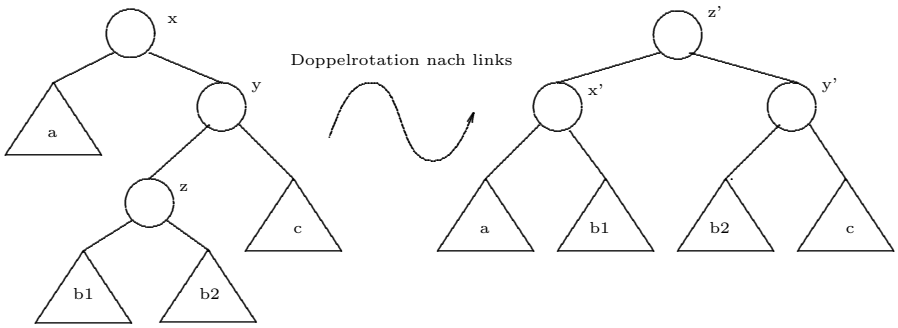
$$\begin{aligned} \text{nacher:} \quad \rho(x') &= \frac{|a|}{|a|+|b|} \\ \rho(y') &= \frac{|a|+|b|}{|a|+|b|+|c|} \end{aligned}$$

Die durchzuführende Transformation ist durch die folgende Skizze gegeben.



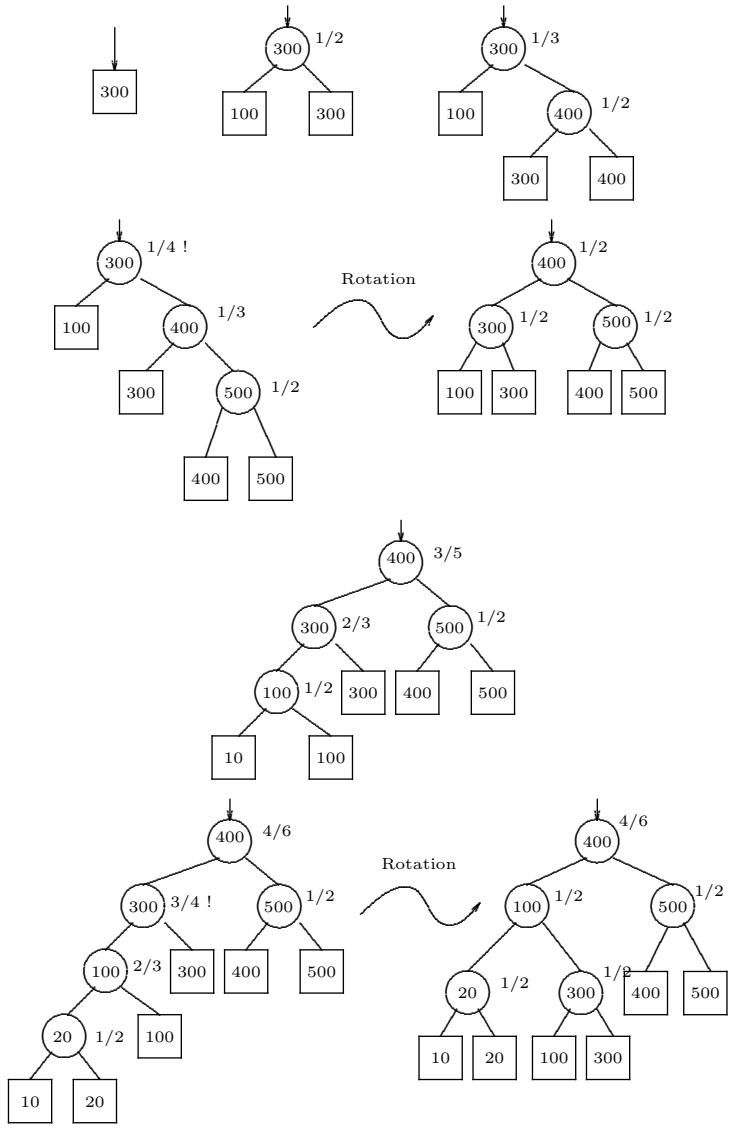
5.3.2 Doppelrotation

Eine andere Transformation ist die Doppelrotation nach links. Diese Operation benutzt man, wenn bei der einfachen Rotation zuviele Blätter in den linken Teilbaum gelangen.



5.4 Rebalancierungsbeispiele

Im folgenden sollen in einen anfangs leeren Baum nacheinander die Werte 300, 100, 400, 500, 10, 20 eingefügt werden. Die entstehenden Bäume sollen von beschränkter Balance $\alpha = \frac{1}{3}$ sein.



Beobachtung: Die Rotation und Doppelrotation braucht konstante Zeit. Man braucht sie relativ selten. Bei einer Einfüge-Operation sind maximal $\log n$ Rotationen (auf einem Pfad) erforderlich.

$\Rightarrow$ die Rebalancierung ändert die größenordnungsmäßige Laufzeit beim Einfügen und Streichen nicht.

5.5 Implementierung

Im folgenden soll kurz skizziert werden, wie man die Rebalancierungen in Java implementieren kann. Zunächst betrachten wir eine mögliche Realisierung von Knoten.

```
class Knoten { int wert;
               Knoten lsohn, rsohn; //Söhne
               int blaetter; // Anzahl der Blätter unterhalb des Knotens
             };
```

Der Eintrag `wert` ist entweder der zu speichernde Wert, falls der Knoten ein Blatt ist, oder es ist der minimale Wert des rechten Teilbaums, falls der Knoten ein innerer Knoten ist. Die Verweise von inneren Knoten auf Sohnknoten sind durch Referenzvariablen dargestellt. Zur Berechnung der Balancewerte soll in jedem Knoten die Anzahl der unter ihm vorkommenden Blätter gespeichert sein. Den Balancewert `rho` eines inneren Knotens kann man dann beispielsweise mit folgender Funktion berechnen.

```
double rho (Knoten x)
// liefert den Balancewert des Knotens x
{ double links=  x.lsohn.blaetter;
  double rechts= x.rsohn.blaetter;
  return ( links / (links+rechts) );
}
```

Eine einfache Rotation nach links kann dann gemäß der oben angegebenen Skizze einfach durch "Umhängen von Zeigern" realisiert werden. Die Implementierung sieht folgendermaßen aus.

```
void rotation_nach_links(Knoten& x)
// Fuehrt am Knoten x eine einfache Rotation nach links durch
{ Knoten y = x.rsohn;
  Knoten a = x.lsohn;
  Knoten b = y.lsohn;
  Knoten c = y.rsohn;
  x.rsohn = b;
  y.lsohn = x;
  x.blaetter = a.blaetter + b.blaetter;
  y.blaetter = a.blaetter + b.blaetter + c.blaetter;
}
```

Ähnlich einfach können auch die Doppelrotation nach links und die entsprechenden Transformationen nach rechts implementiert werden.

6 Hashing

In diesem Kapitel wollen wir uns mit einer besonders schnellen Methode der Datenspeicherung beschäftigen. Sie soll sowohl das Einfügen, als auch den Zugriff auf die Daten sehr schnell realisieren. Als Anwendungsbeispiele denke man z.B. an Password-Tabellen. Beim Einloggen in einen Rechner muss überprüft werden, ob der Benutzer das richtige zum Login-Namen passende Password eingegeben hat. Ein zweites Beispiel ist die Zuordnung von Befehlsnamen zu ausführbaren Programmen. Bei jedem auf der Betriebssystemebene eingegebenen Kommando sollte der Rechner möglichst schnell das dazugehörige auszuführende Programm im Verzeichnisbaum finden.
Beide Anwendungsbeispiele haben gemeinsam, dass im "Normallfall" einer korrekten Eingabe ein sehr schneller Datenzugriff gewünscht wird, während es im Fall einer fehlerhaften Eingabe (falsches Password, bzw. ungültiger Befehlsname) nicht auf eine schnelle Systemantwort ankommt, weil ohnehin eine Fehlermeldung generiert wird, auf die der Benutzer reagieren muss.
Eine weitere Gemeinsamkeit beider Anwendungsbeispiele besteht darin, dass sich die Größe des Datenbestands nicht stark ändert und insbesondere nur selten Einträge entfernt werden müssen.
Das *Hash-Verfahren* zur Speicherung von Daten benutzt eine sogenannte *Hash-Funktion*, um die Daten auf ein Feld (die Hash-Tabelle) zu verteilen. Zur Vereinfachung der Darstellung benutzen wir im folgenden als Daten Zahlenwerte statt Zeichenketten und verzichten darauf, bei den Daten die jeweils dazugehörige Zusatzinformation anzugeben.

Beispiel: Hash-Tabelle der Größe m = 5
Speicherung der Daten 3, 15, 22, 24
Wir benutzen die Hash-Funktion $h(x) = x \bmod 5$ und es entsteht folgende Hash-Tabelle

0:	15
1:	
2:	22
3:	3
4:	24

Dabei wird der Wert x wird an der Stelle $h(x) = x \bmod 5$ der Tabelle abgespeichert.

Die zugrundeliegende Idee ist dabei, die Daten durch einfaches Ausrechnen der *Hash-Funktion* wiederzufinden. Da man jedoch mit relativ kleinen Hash-Tabellen arbeiten möchte (nicht viel größer als die Anzahl der zu speichernden Daten) kommt es zu Konflikten, wenn verschiedene Daten auf der gleichen Adresse abgelegt werden sollen.

$$x \neq y \quad \text{aber} \quad h(x) = h(y)$$

Um dieses Problem zu lösen, gibt es verschiedene Techniken.

6.1 Hashing mit Verkettung

Die Hashtabelle ist ein Feld von verketteten Listen und die i-te Liste enthält alle Daten x mit $h(x) = i$.

```
0:  15  →  20
1:
2:  22
3:   3
4:  24
```

In diesem Fall muss man beim "Einfügen", "Suchen" oder "Entfernen" eines Elements x immer die bei $h(x)$ gespeicherte Liste durchsuchen.
Wenn wir insgesamt s Elemente gespeichert haben, kann ein weiterer Zugriff im ungünstigsten Fall die Laufzeit $O(s)$ haben (wenn alle Daten die gleiche Hashadresse haben). Im Mittel sollte der Zugriff aber viel schneller sein. Um das abzuschätzen, machen wir folgende Annahme.

1. alle möglichen x Werte sind gleich wahrscheinlich
2. die Hash-Funktion verteilt die Werte gleichmäßig

Wenn wir in der Tabelle der Größe m bereits n Daten gespeichert haben, dann berechnen wir den Quotienten

$$\beta = \frac{n}{m}$$

als den *Belegungsfaktor* der Hash-Tabelle.

Satz: Wenn wir nacheinander n "Einfüge-", "Suche-" oder "Lösche-" Operationen auf einer anfangs leeren Hash-Tabelle durchführen, gilt für die erwartete Zeitkomplexität der n-Operationen

$$T(n) = \mathrm{O}\left((1 + \frac{\beta}{2}) \cdot n\right)$$

wobei $\beta = \frac{n}{m}$ der maximale Belegungsfaktor der Tabelle ist.
Die mittlere Zeitkomplexität pro Operation ist also

$$1 + \frac{\beta}{2}$$

Beispiel: Aufwand beim Hashing mit Verkettung

Belegungsfaktor	0.5	0.6	0.7	0.8	0.9
Zugriffszeit im Experiment	1.19	1.25	1.28	1.34	1.38
in der Theorie	1.25	1.30	1.35	1.40	1.45

6.2 Hashing mit offener Adressierung

Beim Hashing mit offener Adressierung berechnet man für ein abzuspeicherndes Element x nacheinander mehrere Hash-Adressen $h(x,0), h(x,1), \ldots$ bis ein leerer Platz in der Tabelle gefunden ist oder x in der Tabelle gefunden ist.
Eine beliebte Wahl für $h(x,i)$ besteht darin, zwei einfache Hashfunktionen zu kombinieren.

$$\text{z.B.:} \quad h(x,i) = [(x \bmod 7) + i \cdot (1 + x \bmod 4)] \bmod m$$

$$\text{oder allgemein:} \quad h(x,i) = [h_1(x) + i \cdot h_2(x)] \bmod m$$

Beispiel: $m = 7$, Einfügen von 3, 17, 6, 9, 15, 13

$$\begin{array}{lcl} h(3,i) & = & (3 + i \cdot 4) \bmod 7 \text{ funktioniert für } i = 0 \\ h(17,i) & = & (3 + i \cdot 2) \bmod 7 \text{ funktioniert für } i = 1 \\ h(6,i) & = & (6 + i \cdot 3) \bmod 7 \text{ funktioniert für } i = 0 \\ h(9,i) & = & (2 + i \cdot 2) \bmod 7 \text{ funktioniert für } i = 0 \\ h(15,i) & = & (1 + i \cdot 4) \bmod 7 \text{ funktioniert für } i = 0 \\ h(13,i) & = & (6 + i \cdot 2) \bmod 7 \text{ funktioniert für } i = 4 \end{array} \quad \rightarrow$$

0:	13
1:	15
2:	9
3:	3
4:	
5:	17
6:	6

Hashing mit offener Adressierung braucht keinen zusätzlichen Speicherplatz, verhält sich aber miserabel, wenn der Belegungsfaktor nahe an 1 kommt. Außerdem unterstützt es nicht das Entfernen von Einträgen, weil dabei Kollisionsketten unterbrochen werden können. Entfernt man beispielsweise in obiger Tabelle den Eintrag 6 auf Adresse 6, so wird man als Folge bei der Suche nach dem Eintrag 13 bereits mit $i = 0$ auf ein leeres Feld stoßen und deshalb annehmen, dass der Wert 13 nicht mehr eingetragen ist.

Satz: Falls der Belegungsfaktor $\beta = \frac{n}{m}$ kleiner als 1 ist, ist die mittlere Dauer einer Einfügeoperation

$$\frac{m+1}{m-n+1} \approx \frac{1}{1-\beta}$$

Für die mittlere Zeit einer erfolgreichen Suche gilt

$$O(\frac{1}{\beta} \cdot ln\frac{1}{1-\beta})$$

Beispiel: Aufwand beim Hashing mit offener Adressierung

β	0.5	0.7	0.9	0.95	0.99	0.999
Einfügen $\frac{1}{1-\beta}$	2	3.3	10	20	100	1000
erfolgreiche Suche: $\frac{1}{\beta}\ln(\frac{1}{1-\beta})$	1.38	1.70	2.55	3.15	4.65	6.9

Um in der Praxis günstige Zugriffzeiten zu haben, muss man die Hashtabelle ausreichend groß wählen. Sollte der Belegungsfaktor trotzdem zu groß werden, ist es sinnvoll, alle Daten in eine größere Hashtabelle zu kopieren. Die dazu erforderliche Rechenzeit spart man bei den weiteren Zugriffen wieder ein.

Beispiel: "Umkopieren" einer Hashtabelle in eine neue größere Tabelle.
Alte Tabelle der Größe $m = 7$, neue Tabelle der Größe $m = 23$, neue Hash-Funktion: $h(x, i) = [x \bmod 23 + i \cdot (1 + x \bmod 7)] \bmod 23$

0	13
1	15
2	9
3	3
4	
5	17
6	6

$\Rightarrow$

0	
1	
2	
3	3
4	
5	
6	6
7	
8	
9	9
10	
11	
12	
13	13
14	
15	15
16	
17	17
18	
19	
20	
21	
22	

7 Suchen in Zeichenfolgen

In diesem Abschnitt sollen verschiedene Verfahren vorgestellt werden, mit deren Hilfe man vorgegebene Muster (Pattern) in Zeichenketten finden kann. Als typische Anwendung denke man beispielsweise an "Suchmaschinen", mit denen vorgegebene Datenbestände nach Stichworten durchsucht werden. Als "Texte" sollen allerdings neben den normalen ASCI-Zeichenketten auch Strings über anderen Alphabeten zugelassen werden. Beispielsweise könnte man auch bestimmte Bitfolgen in Binärdateien suchen wollen.

Problem: Gegeben sei eine Zeichenfolge $p = (p_0, p_1, p_2, \ldots, p_{m-1})$ der Länge m und ein Text $t = (t_0, t_1, t_2, \ldots, t_{n-1})$ der Länge n. Man finde die erste Position, an der der String p in t auftritt.

Beispiel: $t =$ "Dies ist ein Beispieltext", $p =$ "spiel".

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
D	i	e	s		i	s	t		e	i	n		B	e	i	s	p	i	e	l	t	e	x	t
s	p	i	e	l																				
	s	p	i	e	l																			
		s	p	i	e	l																		
			s	p	i	e	l																	
																s	p	i	e	l				

Die Lösung würde in diesem Beispiel $i = 16$ lauten.

Anmerkung: Wenn wir uns nicht mit dem Sonderfall, dass p überhaupt nicht in t vorkommt, beschäftigen wollen, können wir den String p an t anhängen und dann das erste Vorkommen von p in dem zusammengesetzten Text tp suchen.

7.1 Einfaches Suchverfahren

Ein erstes einfaches Verfahren zur Lösung des Problems kann darin bestehen, dass wir den String p von der Position 0 bis zur Position $n - m$ am String t vorbeischieben und dabei jeweils die Gleichheit der m Zeichen prüfen.
Im günstigsten Fall, dass das erste Zeichen p_0 des gesuchten Strings nur einmal in t auftritt, erkennt man die fehlerhaften Positionierungen von p bereits durch den Vergleich eines Zeichens. In diesem Fall brauchen wir insgesamt höchstens n Vergleiche von Zeichen (bzw.$n + m$, wenn wir in tp suchen).
Treten jedoch Präfixe von p sehr oft in t auf, so wächst der Vergleichsaufwand erheblich. Im ungünstigsten Fall brachen wir dann $n \cdot m$ Vergleiche.
Dazu betrachten wir als Beispiel das Suchen in Binärmustern.

Beispiel: $t =$ "10100101001101011", $p =$ "1011".

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	1	0	0	1	0	1	0	0	1	1	0	1	0	1	1
$\underline{1}$	$\underline{0}$	$\underline{1}$	1													
	1	0	1	1												
		$\underline{1}$	$\underline{0}$	1	1											
			1	0	1	1										
				1	0	1	1									
					$\underline{1}$	$\underline{0}$	$\underline{1}$	1								
...																
													$\underline{1}$	$\underline{0}$	$\underline{1}$	$\underline{1}$

Diesen offensichtlich unbefriedigenden Algorithmus könnten wir etwa folgendermaßen implementieren.

```
char t[n]="10100101001101011";
char p[m]="1011";
...
int search ( char t[], char p[], int n, int m)
{
 int i=0;
 while (i<n)
  {
   int j=0;
   while ( (t[i+j] == p[j]) && (j<m) )
     j++;
   if (j==m)
     return i;
   i++;
  }
}
```

7.2 Algorithmus von `Knuth-Morris-Pratt`

Eine Verbesserung der Laufzeit lässt sich dadurch erreichen, dass man bei einem Fehlversuch an Position i, der vielleicht k Vergleiche benötigt hat, die durch die k Vergleiche gewonnenen Informationen ausnutzt. Sucht man etwa an der Stelle i in t beginnend den String $p = 1011$ und findet man die erste Abweichung im vierten Zeichen von p, so kann man p sofort um zwei Zeichen nach rechts verschieben, weil an der Stelle $i + 1$ in t das Zeichen $p_1 = 0$ steht, das nicht mit $p_0 = 1$ übereinstimmt.

t=	... 1 0 1 ? ...	
p=	1 0 1 1	Versuch an Position i
p=	1 0 1 1	Versuch an Position $i + 2$

Das Zeichen t_{i+2} stimmt dann bereits mit p_0 überein, so dass der Vergleich zwischen t und p sogar mit dem zweiten Zeichen p_1 von p fortgesetzt werden kann. Für den Ablauf des Suchalgorithmus bedeutet dies, dass wir im Text t nicht zurückpositionieren müssen. Überaschenderweise ist diese Situation keineswegs ein Sonderfall, sondern es zeigt sich, dass man immer ohne Zurückpositionieren in t auskommen kann. Als Konsequenz kommt die Suche immer mit $n + m$ Vergleichen aus.
Der so arbeitende Algorithmus stammt von den Autoren *Knuth*, *Morris* und *Pratt*. Um ihn zu formulieren, wollen wir zunächst genauer untersuchen, welche Position von p als nächstes betrachtet werden muss, wenn beim vorigen Anlegen an t die Zeichen p_0,. . .,p_{j-2} übereingestimmt haben. D.h., wenn erst das Zeichen `p[j]` fehlerhaft war.
Dazu notieren wir die bekannten Zeichen von t, also die ersten j Zeichen von p. Diese Folge nennen wir q. Wir schreiben unter das Wort q nocheinmal um eine Position nach rechts verschoben wieder das Wort q und verschieben das untere Wort soweit nach rechts, bis der überlappenden Bereich beider Wörter übereinstimmt, bzw. bis sich die Wörter nicht mehr überlappen.

Beispiel: Wenn beispielsweise im String p = "10101011" die ersten 5 Zeichen gepasst haben, d.h. $j = 5$ und $q = 10101$, dann ergibt sich folgende Situation.

t= . . . 1 0 1 0 1 ? ? . . .	String q
1 0 1 0 1	q um eine Position versetzt passt nicht
1 0 1 $\underline{0}$ 1	q um zwei Positionen versetzt passt zu t

Das nächste zu prüfende Zeichen ist dann p_3.

Die nächste in p zu prüfende Position halten wir in einem Vektor `next[j]` fest. Beispielsweise bedeutet `next[j]=3`, dass als wir an der Stelle $p_{\texttt{next[j]}} = p_3$ mit den Vergleichen weitermachen können, wenn, zuvor im Zeichen `p[j]` ein Fehler entdeckt wurde.
Vor dem Start des eigentlichen Suchalgorithmus muss das Feld `next` für den Suchstring p einmal berechnet werden. Wir geben als Beispiel die Berechnung für $p = 10100111$ an.

Beispiel: Berechnung des Feldes `next` für $p = 10100111$. Für $j = 1$ hat nur ein Zeichen gepasst:

t= ... 1 ? ? ? ... String q
$\underline{1}$ q um eine Position versetzt

Es ergibt sich `next`[1] = 0

Für $j = 2$ haben 2 Zeichen gepasst:

t= ... 1 0 ? ? ? ... String q
$\underline{1}$ 0 q um zwei Positionen versetzt

Es ergibt sich `next`[2] = 0

$j = 3$:

t= ... 1 0 1 ? ? ? ... String q
1 $\underline{0}$ 1

Es ergibt sich `next`[3] = 1

$j = 4$:

t= ... 1 0 1 0 ? ? ? ? ? ? ...
1 0 $\underline{1}$ 0

Es ergibt sich `next`[4] = 2

$j = 5$:

t= ... 1 0 1 0 0 ? ? ? ? ? ...
$\underline{1}$ 0 1 0 0

Es ergibt sich `next`[5] = 0

$j = 6$:

t= ... 1 0 1 0 0 1 ? ? ? ? ...
1 $\underline{0}$ 1 0 0 1

Es ergibt sich `next`[6] = 1

$j = 7$:

t= ... 1 0 1 0 0 1 1 ? ? ? ? ...
1 $\underline{0}$ 1 0 0 1

Es ergibt sich `next`[7] = 1

$j >= 8$ ist irrelevant, weil in diesem Fall bereits alle 8 Zeichen von p in t übereingestimmt haben und deshalb die Suche abgebrochen wird.

Mit Hilfe des Feldes `next`, kann der Suchalgorithmus folgendermaßen durchgeführt werden.

```
int kmp_search ( char t[], char p[], int n, int m )
{
 int i=0;
 int j=0;
 int next[m];
 init_next(next,p,m); // Initialisierung von next

 do
  {
   if (t[i] == p[j])
     {i++; j++;}
   else
     if (j==0)    // Das erste Zeichen hat nicht gepasst
       i++;       // in t weiterschalten
     else
       j=next[j]; // p entsprechend der Uebereinstimmung verschieben
  }while ((j<m) && (i<n));
 if (j>=m)
    return i-m; // gefunden
 else
     return -1; // nicht gefunden
}
```

Eine schöne Eigenschaft dieses Verfahrens besteht darin, dass der String t sequentiell abgearbeitet wird und nie zurückpositioniert wird. Deshalb eignet sich das Verfahren besonders gut, wenn t ein sequentiell eintreffender Datenstrom ist, der nicht gespeichert werden soll. Kann man in t beliebig vor und zurückpositionieren, gibt es weitere Verbesserungsmöglichkeiten.

7.3 Algorithmus von `Boyer-Moore`

Beim Algorithmus von *Boyer-Moore* geht man im Prinzip genauso vor, führt aber die Vergleiche mit dem Muster p immer von rechts nach links durch. Dazu wird wieder ein Feld `next` verwendet, das die maximal mögliche Verschiebung des Musters p angibt. Sucht man beispielsweise das Muster $p = 010111$ und haben die hinteren drei Zeichen gepasst, nicht aber das vierte von rechts, so kann man p weil 111 nicht mehr in p vorkommt, sofort um 6 Stellen nach rechts verschieben.
Alle bisher behandelten Verfahren haben jedoch eine Information vernachlässigt. Der tatsächliche Wert des ersten nicht passenden Zeichens in t wurde nicht analysiert. Unter der Annahme, dass das Alphabet relativ groß ist und p relativ kurz ist, kann man dabei das Muster p nach einem Widerspruch meist viel weiter nach rechts verschieben. Kommt dieses Zeichen beispielsweise überhaupt nicht in p vor, so kann man p um die Gesamtlänge von p verschieben. Ansonsten zumindest bis zum nächsten Auftreten des Zeichens in p.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
D	i	e	s		i	s	t		e	i	n		B	e	i	s	p	i	e	l	t	e	x	t
s	p	i	e	l̲																				
					s	p	i	e	l̲															
						s	p	i	e	l̲														
								s	p	i	e	l̲												
													s	p	i	e	l̲							
																s̲	p̲	i̲	e̲	l̲				

Beim ersten Anlegen von p vergleicht man beispielsweise das Zeichen `l` mit einem Leerzeichen (‘ ’). Da dieses Zeichen überhaupt nicht in p auftritt, kann p um 5 Zeichen nach rechts verschoben werden. Beim zweiten Anlegen trifft das `l` auf ein `e`. Dieser Widerspruch führt dazu, dass p nur um eine Position nach rechts verschoben werden kann, weil dann erstmals das `e` des Textes zum Muster p passt.
Wir müssen also für jedes Zeichen des Alphabets festlegen, wie weit man p verschieben kann, wenn das letzte Zeichen von p nicht passt.

`l`:	0
`e`:	1
`i`:	2
`p`:	3
`s`:	4
alle anderen Zeichen:	5

Damit haben wir jetzt zwei Möglichkeiten kennengelernt, das Muster p zu verschieben:

1. Der analysierte Suffix von p bestimmt die Verschiebung
2. Das erste unterschiedliche Zeichen bestimmt die Verschiebung

Die Idee von *Boyer* und *Moore* besteht nun darin, beide Methoden zu kombinieren, indem man jeweils die größere der beiden Verschiebungen benutzt. Auch dazu noch ein Demonstrationsbeispiel.

Beispiel: Im Text $t = 12331201231123$ soll das Muster $p = 1123$ gesucht werden.

1	2	3	3	1	2	0	1	2	3	1	1	2	3
1	1	2	3										
				1	1	2	3						
						1	1	2	3				
										1	1	2	3

7.4 Algorithmus von *Rabin-Karb*

Ein ganz anderer Ansatz zur Suche des Strings p in einem Text t benutzt ein Hashverfahren. Dazu berechnet man jeweils für m aufeinanderfolgende Zeichen des Textes eine Hashadresse $h(t_i, t_{i+1}, \ldots, t_{i+m-1})$ und vergleicht diese Adresse mit der Hashadresse $h(p_0, p_1, \ldots, p_{m-1})$ des gesuchten Strings. Meistens wird die Adresse unterschiedlich sein und man geht im Text um eine Position weiter. Nur wenn die berechnete Hashadresse stimmt, überprüft man den Text genauer, um festzustellen, ob p tatsächlich gefunden wurde. Der Vorteil der Methode besteht darin, dass man den Vergleich von Zeichenketten der Länge m auf den Vergleich zweier Zahlen reduziert. Dabei wählt man den Zahlbereich der Hashadressen möglichst groß, so dass die Gefahr von Kollisionen (verschiedene Strings mit gleicher Haschadresse) gering ist. Man beachte, dass dies nicht zu einem hohen Speicherplatzbedarf führt, weil überhaupt keine Hashtabelle angelegt wird.
Damit die Methode praktikabel ist, wählt man die Hashfunktion so, dass sich das Verschieben um ein Zeichen nach rechts mit geringem Rechenaufwand realisieren lässt.
Beispiel:

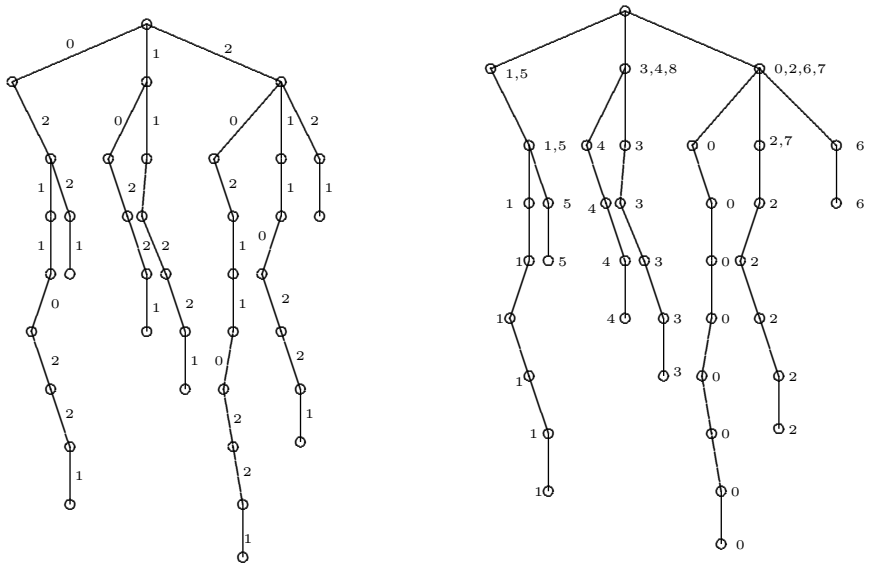
$$h(x_0, x_1, \ldots, x_{m-1}) = \left(\sum_{i=0}^{m-1} x_i \cdot d^{m-1-i} \right) \bmod p$$

Bei dieser Definition ergibt sich:

$$h(t_{i+1}, t_{i+2}, \ldots, t_{i+m}) = \left(\left(h(t_i, t_{i+1}, \ldots, t_{i+m-1}) - t_i \cdot d^{m-1} \right) \cdot d + t_{i+m} \right) \bmod p$$

7.5 Verwendung von Suffix-Trees (Tries)

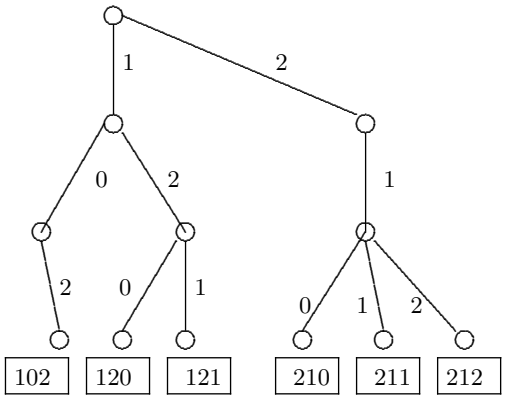
Die bisher vorgestellten Verfahren beruhten alle darauf, das Suchwort p vorzuverarbeiten und anschließend einen beliebigen Text nach p zu durchsuchen. Als Alternative kann man auch eine Vorverarbeitung von t vornehmen, um in einem festen Text besonders schnell nach beliebigen Wörtern suchen zu können. Dazu verwendet man sogenannte Suffix-Trees, in denen sämtliche in t vorkommenden Zeichenketten in einer Baumstruktur dargestellt werden. Zur Vereinfachung soll dies am Beispiel des Alphabets $\Sigma = \{0, 1, 2\}$ dargestellt werden. Der Text $t = 202110221$ entspricht folgendem Suffix-Baum.



Dieser Baum (linke Abbildung) entsteht dadurch, dass man nacheinander alle Suffixe von t als bei der Wurzel beginnenden Pfade einträgt. (Also die Wörter 1, 21, 221, 0221, ..., 202110221.) Anschließend entspricht jedes Teilwort aus t einem bei der Wurzel des Baumes beginnenden Pfad. Will man die Teilwörter auch noch in t lokalisieren, dann kann man jeden Knoten des Tries mit den entsprechenden Positionen beschriften (rechte Abbildung).
Sucht man beispielsweise den String `21`, so folgt man bei der Wurzel beginnend den mit `2` und `1` beschrifteten Kanten und gelangt an den Knoten mit den Positionsangaben "2,7". Der String kommt also an den Positionen 2 und 7 in t vor. Offensichtlich kann man also in Zeit $O(m)$ alle Vorkommen eines Musters $p = (p_0, \ldots, p_{m-1})$ in t entdecken. Kommt ein Muster nicht vor, so fehlt im Baum eine der benötigten Kanten.
Für die Implementierung eines Tries legt man in jedem Knoten ein Feld mit Verweisen auf die Nachfolger an. Dieses Feld muss mit den Zeichen des Alphabets Σ indiziert werden.
Der hohen Geschwindigkeit beim Suchvorgang steht ein sehr hoher Speicherplatzbedarf gegenüber. Im folgenden werden wir deshalb untersuchen, wie

sich Tries platzsparender implementieren lassen. Dazu werden wir jedoch als übersichtlicheres Beispiel die Speicherung von *Mengen von Wörtern* betrachten. Dazu tragen wir statt den Suffixes eines Strings t, einfach alle Wörter einer gegebenen Menge S ein.

Beispiel: Darstellung der Menge $S = \{102, 120, 121, 210, 211, 212\}$ in einem Trie.



7.5.1 Platzsparende Darstellung von Tries

Zunächst betrachten wir eine einfache Implementierung eines Tries über dem Alphabet $\Sigma = \{0, 1, 2, \ldots, k-1\}$. Die Knoten sollen dann einfach als Felder von Verweisen dargestellt werden. Außerdem sollen die Knoten (überflüsigerweise) noch einen Verweis auf den durch sie dargestellten String beeinhalten.

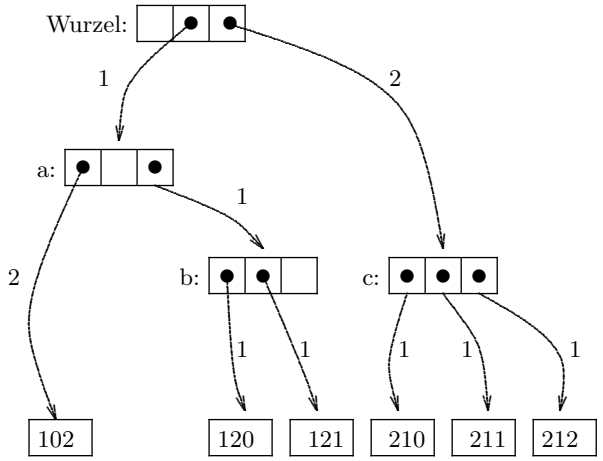
```
struct Knoten
   { Knoten* next[k];
     char*   Inhalt;
   };
```

Um für einen String `x[0]`, ..., `x[m - 1]` der Länge m zu prüfen, ob er zur Menge S gehört, können wir dann folgendes Programmstück benutzen.

```
v=Wurzel;   // v stellt einen Verweis auf die Wurzel dar
for (int i=0; i<m; i++)
   v = (*v).next[x[i]];

if (x== (*v).Inhalt) // Stringvergleich
  cout << x << "gehoert zur Menge" << endl;
else
  cout << x << "gehoert nicht zur Menge" << endl;
```

Dabei setzen wir voraus, dass die nicht benutzten Verweise im Baum alle auf einen zusätzlichen Hilfsknoten zeigen und wir deshalb nicht auf die Gültigkeit der Pointer testen müssen.
Die Laufzeit dieses Programmstücks beträgt $T = \mathrm{O}(m)$. Der benötigte Speicherplatz des Tries für eine Menge S mit n Wörtern der Länge m beträgt im ungünstigsten Fall $\mathrm{O}(n \cdot m \cdot k)$, weil der Trie $n \cdot m$ Knoten haben kann.
Wir versuchen nun den Speicherplatz zu reduzieren, indem wir nicht mehr alle Knoten speichern, sondern nur noch solche, an denen es Verzweigungen gibt. man beachte dabei, dass bei unseren bisherigen Beispielen von Suffix-Trees sehr viele Knoten ohne Verzweigungen existieren.



In diesem Fall muss man bei den Kanten speichern, um wieviele Stufen sie nach unten laufen. Diese Zahl gibt an, mit welchem Zeichen der Eingabe die nächste Verzweigung durchgeführt werden muss. Wir speichern diese Werte in den jeweiligen Ausgangsknoten in der Komponente `delta`.

```
struct Knoten
   { Knoten* next[k];
     int     delta[k];
     char*   Inhalt;
   };
```

Der Suchvorgang sieht dann folgendermaßen aus.

```
v=Wurzel;   // v stellt einen Verweis auf die Wurzel dar
int i=0;
while (i<m)
 {
   int j= x[i]; // j ist das zu untersuchende Zeichen
   int d= (*v).delta[j];
   v = (*v).next[j];    // Folgeknoten
   i += d;
 }

if (x== (*v).Inhalt) // Stringvergleich
  cout << x << "gehoert zur Menge" << endl;
else
  cout << x << "gehoert nicht zur Menge" << endl;
```

Die hier verwendete Baumstruktur wird auch als *komprimierter Trie* bezeichnet.

Satz: Für einen komprimierten Trie zur Darstellung einer Menge mit n Wörtern der Länge m über einem Alpahbet der Größe k gilt:

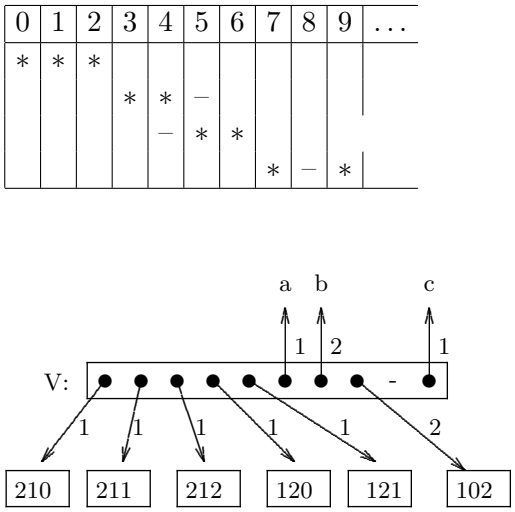
- Die Operationen *Suche*, *Einfügen* und *Löschen* brauchen die Laufzeit $T = \mathrm{O}(m)$.
- Der Speicherplatzbedarf beträgt $\mathrm{O}(k \cdot n)$.
- Die mittlere Laufzeit (bei zufälligen Elementen) beträgt: $\mathrm{O}\left(\frac{\log n}{\log k}\right)$

Oftmals braucht man weder "Einfügen" noch "Löschen", sondern benutzt nur die Suche. In diesem Fall bezeichnet man den Trie als *statisch* (weil er sich nicht ändert). Ein *statischer Trie* kann viel platzsparender dargestellt werden. Um dies zu demonstrieren, betrachten wir nochmal die Verweisvektoren aus dem obigen Beispiel des komprimierten Tries.
In der folgenden Tabelle ist jeder Verweis durch ein "$*$" und die nicht benutzten Einträge durch "–" dargestellt.

c:	$*$	$*$	$*$
b:	$*$	$*$	–
Wurzel:	–	$*$	$*$
a:	$*$	–	$*$

Dabei sind die Zeilen nach der Anzahl der benutzten Verweise geordnet. Wir werden die Verweise nun platzsparend mit Hilfe der *Overlay-Technik* in einem Feld `V` speichern und die Anfangsposition der Vektoren in einem Feld `offset` darstellen:

Knoten	`offset`
Wurzel:	4
a:	7
b:	3
c:	0



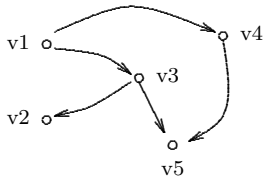
Dabei werden die Vektoren im Feld V nebeneinander möglichst weit links eingetragen, so dass alle Verweise auf verschiedene Positionen gelangen. Wenn wir nun beispielsweise den i-ten Sohn des Knotens a zugreifen wollen, dann geht das folgendermaßen.

$$\texttt{V}[\texttt{offset}[\texttt{a}] + \texttt{i}]$$

Damit ist V eine komprimierte Darstellung des Baumes, bei der man Speicherplatz für unbenutzte Variablen spart. Die gleiche Komprimierungsmethode benutzt man auch für die `delta`-Werte der Verweise.

8 Graphen

Ein *Graph* G besteht aus einer Menge V von *Knoten* (vertex) und einer Menge $E \subseteq V \times V$ von *Kanten* (edge).



$$\begin{aligned} G &= (V, E) \\ V &= \{v_1, v_2, v_3, v_4, v_5\} \\ E &= \{(v_1, v_4), (v_1, v_3), (v_3, v_2)(v_3, v_5), (v_4, v_5)\} \end{aligned}$$

Um einen Graphen im Rechner darzustellen, kann man folgendermaßen vorgehen:

1. Nummeriere die Knoten und setze $V = \{1, 2, 3, \ldots, n\}$; $n = |V|$.
   Die Kanten werden dann durch Zahlenpaare dargestellt.

   ```
   struct Kante {
                int a,b;
               };
   struct Graph {
                int n;  // Knotenanzahl
                list E; // verkettete Liste von Kanten
               };
   ```

   Die Kantenliste könnte etwa durch eine Verkettung von Kanten oder mit Hilfe eines Feldes dargestellt sein.

2. Als Alternative kann man einen Graphen auch in Form einer Matrix beschreiben:

   ```
   boolean G[n][n];
   ```

   Dabei soll `G[i][j] == true` bedeuten, dass es im Graphen G eine Kante vom Knoten i zum Knoten j gibt.
   Im Beispiel ergibt sich

i/j	1	2	3	4	5
1	0	0	1	1	0
2	0	0	0	0	0
3	0	1	0	0	1
4	0	0	0	0	1
5	0	0	0	0	0

   Eine solche Matrix wird auch `Adjazenzmatrix` genannt. (Zwei Knoten heißen `adjazent` (oder benachbart), wenn sie durch eine Kante verbunden sind.

Die bisher vorgestellten Beispiele stellen `gerichtete` Graphen dar. Bei `ungerichteten` Graphen kann man jede Kante in beide Richtungen durchlaufen. Die Adjazenzmatrix ist dann symmetrisch.

$$(v_i, v_j) \in E \Rightarrow (v_j, v_i) \in E$$

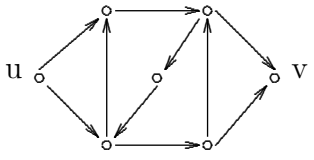
Im obigen Beispiel ergäbe sich ohne Kantenorientierung beispielsweise die folgende Matrix.

i/j	1	2	3	4	5
1	0	0	1	1	0
2	0	0	1	0	0
3	1	1	0	0	1
4	1	0	0	0	1
5	0	0	1	1	0

Beide Darstellungsarten haben Vor- und Nachteile. Beispielsweise ist der Speicherplatzbedarf bei der verketteten Liste linear in der Anzahl der Kanten, während er bei der Matrixdarstellung quadratisch in der Anzahl der Knoten ist. Bei Graphen mit relativ wenigen Kanten, ist deshalb die Listendarstellung platzsparender.
Ein Vorteil der Matrix ist, dass man leicht für zwei Knoten a und b prüfen kann, ob die Kante (a, b) existiert. Bei der Listendarstellung muss dazu die Liste sequentiell durchlaufen werden. Will man jedoch alle Kanten nacheinander bearbeiten, so ist die Listendarstellung geschickter, weil man bei der Matrix n^2 Einträge überprüfen muss, um alle Kanten zu finden.

Problem: Gegeben sei ein Graph $G = (V, E)$ und zwei Knoten $u, v \in V$. Über wieviele Kanten muss man laufen, um von u zu v zu gelangen? Ist v überhaupt von u aus erreichbar?

Wie aufwendig ist es, diese Frage zu beantworten, wenn der Graph n Knoten und m Kanten hat?



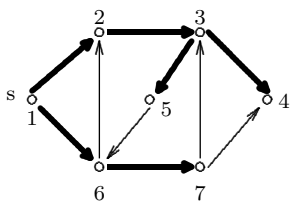
8.1 "Breadth-first-search"-Algorithmus (BFS)

Die Grundidee, von einem Startknoten s aus kürzeste Wegen zu anderen Knoten zu finden, besteht darin, den Graphen bei u beginnend "in der Breite" zu durchsuchen. Anschaulich bedeutet dies, dass man zunächst alle von u ausgehenden Kanten gleichzeitig ausprobiert und anschließend von den erreichten Knoten wiederum alle Kanten gleichzeitig betrachtet. Das Verfahren kann abgebrochen werden, sobald der Zielknoten erreicht wird. Da auf diese Weise kurze Wege vor langen Wegen ausprobiert werden, ist garantiert, dass der gefundene Weg von u nach v auch der kürzest mögliche Weg ist.
Das Verfahren eignet sich auch dazu, die Entfernungen von einem gegebenen Startknoten zu allen anderen Knoten des Graphen zu ermitteln. Im folgenden wird eine präzise Darstellung des Algorithmus gegeben.

"Breadth-first-search"-Algorithmus
Gegeben : Graph $G = (V, E)$ und ein Knoten $s \in V$.
Gesucht : Die kürzesten Abstände von s aus zu allen anderen Knoten.

```
L    ← Liste, die nur das Element s enthält;
d[v]← undefiniert für alle v ∈ V \ {s};
d[s]← 0;

while L ≠ ∅
    {
     entferne das erste Listenelement v aus L;
     Für alle von v ausgehenden Kanten (v, w):
       if (d[w] == undefiniert)
          {
           d[w] ← d[v] + 1;
           füge w am Ende der Liste L an;
          }
    }
```



Die kürzesten Wege von s aus bilden einen aufspannenden Baum

i	$d[i]$
1	0
2	~~undef~~ 1
3	~~undef~~ 2
4	~~undef~~ 3
5	~~undef~~ 3
6	~~undef~~ 1
7	~~undef~~ 2

$L = [\not{1}, \not{2}, \not{6}, \not{3}, \not{7}, \not{4}, \not{5}]$

Für die Laufzeit des BFS-Algorithmus erhalten wir

$$T_{\text{BFS}} = \text{O}(|V| + |E|) = \text{O}(n + m),$$

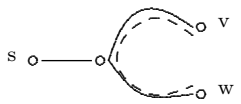
da zu Beginn jeder Knoten einmal initialisiert wird und innerhalb der Schleife jede Kante genau einmal betrachtet wird. Die Laufzeit ist also *linear in der Größe der Graphenbeschreibung*$(n + m)$.
Man beachte dabei, dass die Laufzeit in Abhängigkeit von zwei Größen (n und m) dargestellt wird. Würden wir m mit Hilfe von n nach oben abschätzen, ergäbe sich als Laufzeit $T_{\text{BFS}} = \text{O}(n + n^2) = \text{O}(n^2)$. Diese Abschätzung ist zwar auch richtig, ist aber für Graphen mit relativ wenig Kanten unnötig schlecht.

Beobachtung: Der BFS-Algorithmus ist auch für ungerichtete Graphen anwendbar.

Definition: Ein ungerichteter Graph $G = (V, E)$ heisst *einfach zusammenhängend*, wenn es von jedem Knoten $v \in V$ einen Weg zu jedem anderen Knoten $w \in V$ gibt.

Beobachtung: Mit dem BFS-Algorithmus kann man in Zeit $O(|V| + |E|)$ feststellen, ob ein ungerichteter Graph einfach zusammenhängend ist.

Methode : Man wählt einen beliebigen Knoten $s \in V$ und prüft, ob alle Knoten von s aus erreichbar sind. Falls beispielsweise v und w von s aus erreicht werden, dann gibt es offensichtlich auch einen Weg zwischen v und w.



Wollen wir die Weglänge zwischen je zwei Knoten des Graphen ermitteln, (die volle Distanz-Matrix), dann können wir BFS für jeden Knoten $v \in V$ einmal aufrufen und dabei jeweils eine Zeile der Distanzmatrix ermitteln.

	1	.	.	w	.	.	n
1							
.							
.							
v							
.							
.							
n							

Wir erhalten auf diese Weise folgende Laufzeit des Verfahrens:

$$T = \text{O}(|V| \cdot (|V| + |E|)) = \text{O}(n \cdot (n + m)) = \text{O}(n^2 + n \cdot m)$$

Wenn wir den ungünstigsten Fall betrachten, dass von jedem Knoten aus alle anderen Knoten direkt über eine Kante erreichbar sind ($m = (n-1)\frac{n}{2}$) d.h. $m = \mathrm{O}(n^2)$, dann folgt für die Laufzeit

$$T = \mathrm{O}(n^2 + n \cdot n^2) = \mathrm{O}(n^3)$$

Wir wollen nun noch einen anderen Algorithmus zur Berechnung der vollen Distanzmatrix betrachten, für den sich die gleiche Laufzeitabschätzung ergibt.

8.2 Algorithmus von Floyd-Warshall

Der *Algorithmus von Floyd-Warshall* ermittelt ebenfalls die volle Distanzmatrix D. Zunächst wird D folgendermaßen mit Hilfe der Adjazenzmatrix A initialisiert.

$$D[i][j] = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \text{ und } i \neq j \\ 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

Danach enthält D bereits alle Wege zwischen zwei Knoten, die nur über eine Kante führen, d.h. alle Direktverbindungen. Nach dieser Initialisierung wird das folgende Programmstück ausgeführt, das systematisch alle Wege zwischen Knotenpaaren v_i, v_j betrachtet, die über die Hilfsknoten $v_1, v_2, \ldots, v_k$ laufen.

```
for (k=1; k<=n; k++)
  for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
      if ( (D[i][k] + D[k][j]) < D[i][j])
        D[i][j] = D[i][k] + D[k][j];
```

Offensichtlich hat dieses Verfahren die Laufzeit $T = O(n^3)$, da drei Schleifen mit jeweils n Durchläufen ineinander verschachtelt sind. Die äußere Schleife mit $k = 1, 2, \ldots, n$ bewirkt, dass nacheinander Wege ausprobiert werden, die außer v_i und v_j nur die ersten k Knoten benutzen. Im folgenden soll das Verfahren am obigen Beispielgraphen vorgeführt werden.

Beispiel : Initialisierung der Matrix D:

i/j	1	2	3	4	5	6	7
1	0	1	∞	∞	∞	1	∞
2	∞	0	1	∞	∞	∞	∞
3	∞	∞	0	1	1	∞	∞
4	∞	∞	∞	0	∞	∞	∞
5	∞	∞	∞	∞	0	1	∞
6	∞	1	∞	∞	∞	0	1
7	∞	∞	1	1	∞	∞	0

Diese Matrix soll auch als Matrix D_0 bezeichnet werden. Bei der Abarbeitung des Algorithmus entstehen dann für $k = 1, 2, 3, \ldots$ folgende Matrizen $D_1, D_2, \ldots$. Die Änderungen aus dem jeweiligen Schleifendurchlauf Nummer k sind hervorgehoben.

$$D_1 = \begin{pmatrix} 0 & 1 & \infty & \infty & \infty & 1 & \infty \\ \infty & 0 & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 & 1 & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & 1 & \infty \\ \infty & 1 & \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & 1 & 1 & \infty & \infty & 0 \end{pmatrix} \quad D_2 = \begin{pmatrix} 0 & 1 & \mathbf{2} & \infty & \infty & 1 & \infty \\ \infty & 0 & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 & 1 & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & 1 & \infty \\ \infty & 1 & \mathbf{2} & \infty & \infty & 0 & 1 \\ \infty & \infty & 1 & 1 & \infty & \infty & 0 \end{pmatrix}$$

Beispielsweise kommt der Eintrag $D_2[1][3] = 2$ dadurch zustande, dass es einen Weg vom Knoten 1 zum Knoten 3 über den "Umwegknoten" 2 gibt. Hier ist die Summe $D_1[1][2] + D_1[2][3] = 1 + 1 = 2$ kleiner als der bisherige Wert $D_1[1][3] = \infty$. Nach dem Schleifendurchlauf für $k = 2$ enthält D_2 die Entfernungen, zwischen allen Knoten v und w, wenn außer v und w nur noch die Knoten 1 und 2 als Zwischenknoten zulässt.

Bei der Abarbeitung des Algorithmus fällt auf, dass die `if`-Bedingung nur recht selten erfüllt ist und daher nur sehr wenige Änderungen in der Matrix durchgeführt werden. Beispielsweise ist D_1 völlig identisch mit D_0, weil der Knoten 1 im Beispiel überhaupt nicht als "Zwischenknoten" verwendet werden kann, da keine Kante zum Knoten 1 führt. Entsprechendes gilt auch für den Knoten 4, weshalb $D_4 = D_3$ gilt.

Weitere Abarbeitungsfolge:

$$D_3 = \begin{pmatrix} 0 & 1 & 2 & \mathbf{3} & \mathbf{3} & 1 & \infty \\ \infty & 0 & 1 & \mathbf{2} & \mathbf{2} & \infty & \infty \\ \infty & \infty & 0 & 1 & 1 & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & 1 & \infty \\ \infty & 1 & 2 & \mathbf{3} & \mathbf{3} & 0 & 1 \\ \infty & \infty & 1 & 1 & \mathbf{2} & \infty & 0 \end{pmatrix} \quad D_4 = \begin{pmatrix} 0 & 1 & 2 & 3 & 3 & 1 & \infty \\ \infty & 0 & 1 & 2 & 2 & \infty & \infty \\ \infty & \infty & 0 & 1 & 1 & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & 1 & \infty \\ \infty & 1 & 2 & 3 & 3 & 0 & 1 \\ \infty & \infty & 1 & 1 & 2 & \infty & 0 \end{pmatrix}$$

$$D_5 = \begin{pmatrix} 0 & 1 & 2 & 3 & 3 & 1 & \infty \\ \infty & 0 & 1 & 2 & 2 & \mathbf{3} & \infty \\ \infty & \infty & 0 & 1 & 1 & \mathbf{2} & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & 1 & \infty \\ \infty & 1 & 2 & 3 & 3 & 0 & 1 \\ \infty & \infty & 1 & 1 & 2 & \mathbf{3} & 0 \end{pmatrix} \quad D_6 = \begin{pmatrix} 0 & 1 & 2 & 3 & 3 & 1 & \mathbf{2} \\ \infty & 0 & 1 & 2 & 2 & 3 & \mathbf{4} \\ \infty & \mathbf{3} & 0 & 1 & 1 & 2 & \mathbf{3} \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \mathbf{2} & \mathbf{3} & \mathbf{4} & 0 & 1 & \mathbf{2} \\ \infty & 1 & 2 & 3 & 3 & 0 & 1 \\ \infty & \mathbf{4} & 1 & 1 & 2 & 3 & 0 \end{pmatrix}$$

$$D_7 = \begin{pmatrix} 0 & 1 & 2 & 3 & 3 & 1 & 2 \\ \infty & 0 & 1 & 2 & 2 & 3 & 4 \\ \infty & 3 & 0 & 1 & 1 & 2 & 3 \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & 2 & 3 & \mathbf{3} & 0 & 1 & 2 \\ \infty & 1 & 2 & \mathbf{2} & 3 & 0 & 1 \\ \infty & 4 & 1 & 1 & 2 & 3 & 0 \end{pmatrix}$$

Obwohl der *Algorithmus von Floyd-Warshall* wie die modifizierte BFS-Methode im ungünstigsten Fall die Laufzeit $T = \mathrm{O}(n^3)$ besitzt, ist die modifizierte BFS-Methode besser, weil sie nur die tatsächlich existierenden Kanten benutzt und nicht alle theoretisch möglichen n^2 Verbindungen.

8.3 Dijkstra-Algorithmus

Bisher haben wir bei den Entfernungsberechnungen angenommen, dass alle Kanten die gleiche Länge (z.B. 1) besitzen. Interessanter ist aber der Fall, bei dem die Kanten gewichtet sind. Als Anwendungsbeispiele denke man etwa an Entfernungsangaben (in km) zwischen verschiedenen Orten oder an die Fahrpreise für unterschiedliche Strecken von öffentlichen Verkehrsmitteln.
Die Suche nach einem "gewichteten" Weg zwischen zwei Knoten entspricht dann der Suche nach der kürzesten oder billigsten Verbindung zwischen zwei Orten. Das ist die typische Fragestellung bei Navigationsgeräten.
Ein ähnliches Problem hat man beim Routing im Internet, wenn man den idealen Weg einer Nachricht vom Sender zum Empfänger ermitteln muss. Das ist insbesondere dann von Interesse, wenn zeitweise ein Knoten (Vermittlungsrechner) oder eine Kante (Verbindungsleitung) ausfällt und deshalb ein alternativer Weg gefunden werden muss.
Ein allgemein verwendbares Verfahren zur Berechnung kürzester gewichteter Wege ist der *Bellman-Ford*-Algorithmus mit Laufzeit $T = O(n \cdot m)$.
Für den beim Routing typischerweise vorliegenden Fall, dass alle Kanten ein Gewicht ≥ 0 haben, formulieren wir im folgenden den *Algorithmus von Dijkstra*:

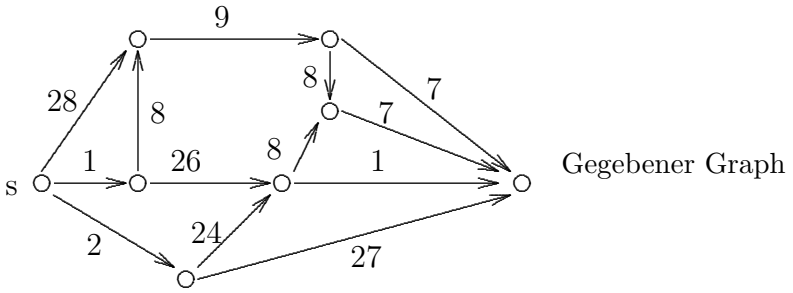
Gegeben ist ein Graph $G = (V, E)$ und eine Gewichtsfunktion $w : E \to R_0^+$, die jeder Kante ein nicht negatives Gewicht zuordnet, sowie ein Startknoten $s \in V$.

Gesucht: Für alle Knoten $v \in V$ die gewichtete Länge $d[v]$ eines kürzesten Weges von s nach v.

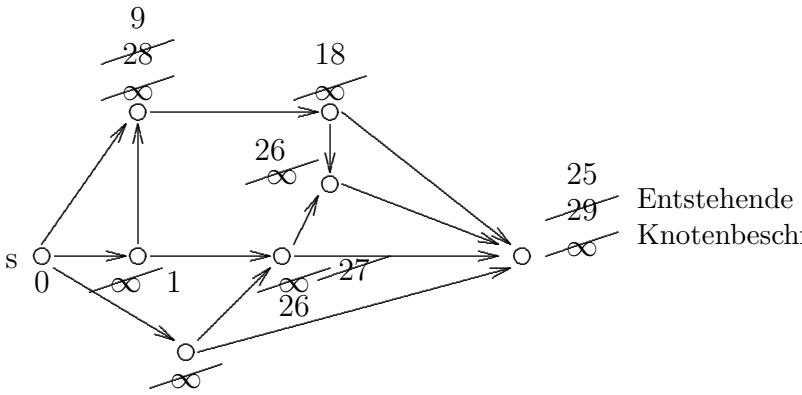
Methode:

$d[v] \leftarrow \infty$ für alle $v \in V \setminus \{s\}$;
$d[s] \leftarrow 0$;
$M \leftarrow V$;
while $M \neq \varnothing$
  {
   finde ein $u \in M$, so dass $d[u]$ minimal ist;
   $M \leftarrow M \setminus \{u\}$;
   Für alle von u ausgehenden Kanten $e = (u, v)$
     $d[v] \leftarrow \min(d[v], d[u] + w(e))$;
  }

Beispiel : Anwendung des Dijkstra-Algorithmus



Gegebener Graph



Entstehende Knotenbeschriftungen

Die Laufzeit des Verfahrens setzt sich aus der Zeit für die Initialisierung und der Zeit für die Schleifendurchläufe zusammen. Für die Initialisierung gilt $T = \mathrm{O}(|V|)$, da für jeden Knoten ein Startwert $d[v]$ festgelegt wird. Die Anzahl der anschließend erforderlichen Schleifendurchläufe beträgt genau $|V|$, da in jedem Schleifendurchlauf ein Element aus der Menge M entfernt wird. Innerhalb der Schleife werden dann für den jeweils gewählten Knoten u alle von u ausgehenden Kanten abgearbeitet, so dass insgesamt jede Kante

genau einmal benutzt wird. Diese Überlegung führt zur Gesamtlaufzeit $T = \mathrm{O}(|V| + |E|)$. Dabei ist allerdings noch nicht berücksichtigt, dass auch Zeit zum Auffinden geeigneter Knoten u investiert werden muss.
Wir müssen deshalb noch überlegen, welche Implementierungsmöglichkeiten wir für den Zugriff auf Knoten mit minimalen "d-Wert" haben.

1. Beispielsweise kann man alle Knoten aus der Menge M nacheinander betrachten, um einen Knoten u mit minimalem Wert $d[u]$ zu ermitteln. In diesem Fall bräuchte jede Suche den Zeitaufwand $\mathrm{O}(|M|) = \mathrm{O}(|V|)$. Da diese Suche in der Schleife $|V|$-mal durchgeführt werden muss, ergäbe sich folgende relativ schlechte Gesamtlaufzeit für den Dijkstra-Algorithmus

$$T = \mathrm{O}\left(|V| + |V|^2 + |E|\right) = \mathrm{O}\left(|V|^2\right)$$

2. Eine bessere Laufzeit erhält man, wenn man dafür sorgt, dass die Knoten der Menge M nach aufsteigenden d-Werten sortiert sind. In diesem Fall lässt sich die Suche in konstanter Zeit durchführen, weil einfach das erste Element benutzt werden kann. Als zusätzliches Problem ist aber zu beachten, dass man die Sortierung immer aktualisieren muss, wenn sich der d-Wert eines Knotens ändert. Immer wenn sich bei der Abarbeitung einer Kante durch die Anweisung "$d[v] \leftarrow \min(d[v], d[u] + w(e))$" eine Knotenbeschriftung ändert, muss der Knoten aus der Sortierung entfernt werden und anschließend mit der neuen Beschriftung wieder in die Sortierung eingefügt werden.

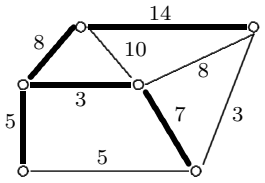
   Benutzen wir für die Sortierung beispielsweise einen balancierten Baum (wie beim Wörterbuch), so kostet das Entfernen und Einfügen eines Knotens jeweils die Zeit $\mathrm{O}(\log(|M|)) = \mathrm{O}(\log(|V|))$. Auf diese Weise ergibt sich sich für den Dijkstra-Algorithmus der Gesamtaufwand

$$\underline{\underline{T = \mathrm{O}\left(|V| + |E| \cdot \log(|V|)\right)}}$$

8.4 Minimale aufspannenden Bäume in Graphen

Bisher haben wir immer Bäume mit einer vorgegebenen Wurzel betrachtet, weil uns die kürzesten Wegen von einem bestimmten Punkt aus interessierten. Nun werden wir uns mit *minimalen aufspannenden Bäumen* beschäftigen. Bevor wir diesen Begriff genauer definieren, zunächst ein Beispiel.

Beispiel : Aufspannender Baum



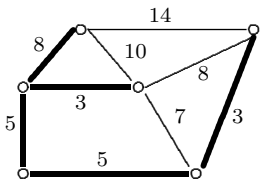
In dem gegebenen ungerichteten Graphen $G = (V, E)$ sind 5 Kanten gekennzeichnet, die einen Baum erzeugen, über den alle Knoten untereinander verbunden sind. Dieser aufspannende Baum hat das Gewicht 37.

Das Gewicht eines Baumes oder eines allgemeinen Teilgraphen ist dabei als Summe der Kantengewichte definiert. Ein Anwendungsbeispiel für minimale aufspannende Bäume ist beispielsweise die Auswahl von Standleitungen für ein möglichst kostengünstiges Rechnernetz.

Definition: Gegeben sei ein ungerichteter zusammenhängender Graph $G = (V, E)$ mit einer Gewichtsfunktion $w : E \to R$. Ein *aufspannender Baum* von G ist ein zusammenhängender Teilgraph (V, T) mit $T \subseteq E$, in dem es keine Kreise gibt. Ein aufspannender Baum (V, T) heißt *minimal*, wenn die Summe der Kantengewichte aus T minimal ist.

Es stellt sich natürlich das Problem, wie man einen minimalen aufspannenden Baum eines Graphen findet. Für das obige Beispiel stellt man leicht fest, dass der angegebene Baum nicht minimal ist. Im folgenden ist für den gleichen Graphen ein minimaler Baum angegeben. Wie erkennt man, dass dieser Baum minimal ist?

Beispiel : Der folgende aufspannende Baum hat das Gewicht 24. Er ist minimal.



Eine systematische Methode zur Konstruktion eines minimalen aufspannenden Baumes besteht darin, die Kanten mit möglichst geringem Gewicht zu verwenden. Dabei darf man allerdings keine Kante benutzen, wenn sie zu einem Kreis führt.

8.4.1 Algorithmus von Kruskal (Version 1)

Diese Idee entspricht dem Algorithmus von *Kruskal*.

Gegeben: Zusammenhängender Graph $G = (V, E)$ und eine Gewichtsfunktion $w : E \to R$.

Gesucht: Ein minimaler aufspannender Baum.

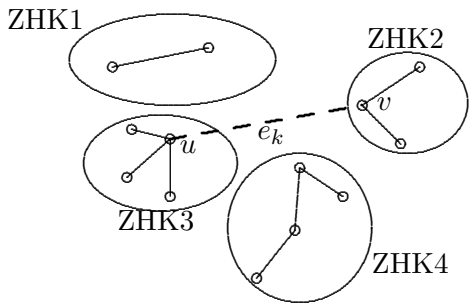
Methode:

Sortiere die Kanten aus E nach aufsteigendem Gewicht.
Die sortierte Folge sei $(e_1, e_2, e_3, \ldots, e_m)$
$T \leftarrow \emptyset$;
for $k \leftarrow 1$ to m do
  if e_k bildet mit den Kanten aus T keinen Kreis
    $T \leftarrow T \cup \{e_k\}$;

Anmerkung: Da man Kanten mit kleinem Gewicht bevorzugt und Kreise vermeidet, entsteht ein minimaler aufspannender Baum.

Die Laufzeit des Verfahrens ergibt sich aus der Initialisierung, die die Zeit $O(m \cdot \log m)$ benötigt, und der Schleife zur Abarbeitung der m Kanten. Unklar ist zunächst, wie aufwendig es ist, festzustellen, ob eine gewählte Kante e_k mit den Kanten aus T einen Kreis bildet. Unser Ziel ist es, die Schleife in Zeit $O(m)$ zu realisieren, so dass die Kreisprüfung praktisch in konstanter Zeit erfolgt. Dieses Ziel ist recht anspruchsvoll und kann nur mit Hilfe besonders geschickter Datenstrukturen erzielt werden.

8.4.2 Zusammenhangskomponenten



Während der Abarbeitung des Algorithmus erzeugen die Kanten aus T Zusammenhangskomponenten (ZHKs) auf der Knotenmenge V. Die Kante $e_k = (u, v)$ erzeugt genau dann einen Kreis, wenn ihre Endknoten u und v bereits in der gleichen Zusammenhangskomponente liegen, also schon in T miteinander verbunden sind.
Zu Beginn des Algorithmus gibt es in T noch keine Kanten und wir haben $|V| = n$ viele verschiedene ZHKs (mit jeweils einem Knoten). Mit jeder Kante, die wir in T aufnehmen, vereinigen wir zwei ZHKs zu einer neuen. Am Ende gibt es nur noch eine einzige Zusammenhangskomponente mit $|V|$ Knoten.
Für eine geschickte Implementierung des Algorithmus benutzen wir eine Funktion `FIND(u)`, die uns sagt, zu welcher Zusammenhangskomponente ein gegebener Knoten `u` gehört. Das Ergebnis könnte etwa in Form einer Nummer oder eines Verweises vorliegen. Die genaue Darstellung ist nicht wichtig, weil wir im Algorithmus lediglich die Gleichheit von Zusammenhangskomponenten erfragen müssen. Dies geht mit `FIND(u) == FIND(v)` bzw. `FIND(u) != FIND(v)`.
Außerdem benutzen wir eine Prozedur `UNION(u,v)`, mit der wir die Zusammenhangskomponenten von `u` und `v` miteinander verschmelzen können.

8.4.3 Algorithmus von Kruskal (Version 2)

Damit sieht der Algorithmus zur Konstruktion eines minimal aufspannenden Baumes dann folgendermaßen aus:

Sortiere die Kanten aus E nach aufsteigendem Gewicht.
Die sortierte Folge sei $(e_1, e_2, e_3, \ldots, e_m)$

$T \leftarrow \emptyset$;

// Initialisierung der ZHKs
for $i \leftarrow 1$ to n do ZHK$[i] \leftarrow i$;

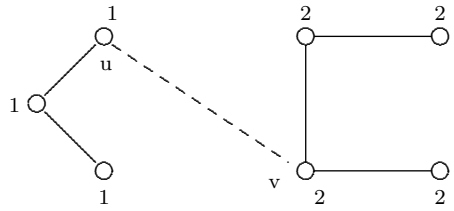
for $k \leftarrow 1$ to m **do**
  Bestimme die Endknoten u und v der Kante e_k
  if (`FIND`(u) != `FIND`(v))
    {
     $T \leftarrow T \cup \{e_k\}$;
     // Aktualisieren der ZHKs
     `UNION`(u, v);
    }

Anmerkung: Falls wir es schaffen, `UNION` und `FIND` mit konstanter Laufzeit O(1) zu realisieren, ergibt sich die Gesamtlaufzeit $T = \mathrm{O}(m \log m + n)$

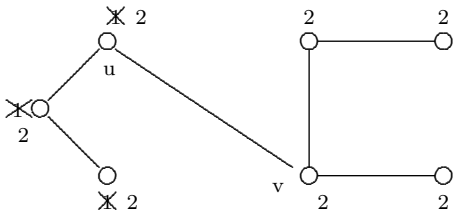
8.4.4 UNION-FIND Datenstruktur

Zunächst diskutieren wir verschiedene Realisierungsideen, um die Funktionen `FIND` und `UNION` mit geringer Laufzeit zu realisieren.

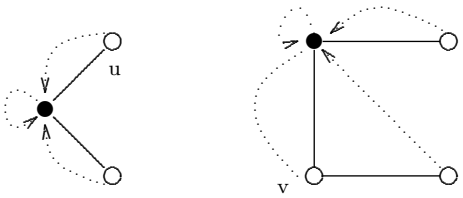
1. Idee: Jeder Knoten wird mit der Nummer seiner ZHK gekennzeichnet



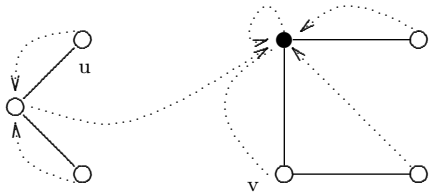
`FIND`(u) hat dann konstante Laufzeit O(1) aber `UNION`(u, v) muss alle Nummern in der einen ZHK ändern. Geschickterweise sollte man die Änderungen in der kleineren der beiden gegebenen ZHKs vornehmen. Trotzdem beträgt dann die Laufzeit für `UNION` $\mathrm{O}(n)$, da im ungünstigsten Fall die Hälfte aller Knoten geändert werden müssen.



2. Idee: Jeder Knoten enthält einen Verweis auf den Repräsentanten seiner ZHK.

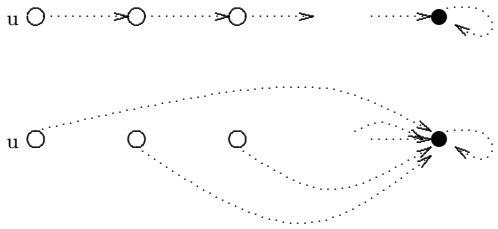


`FIND`(u) folgt dann dem Verweis bis zum Repräsentanten, der auf sich selber zeigt und liefert dann den Repräsentanten als Ergebnis zurück. `UNION`(u, v) lässt dann den Repräsentanten der kleineren ZHK auf den Repräsentanten der größeren ZHK zeigen.



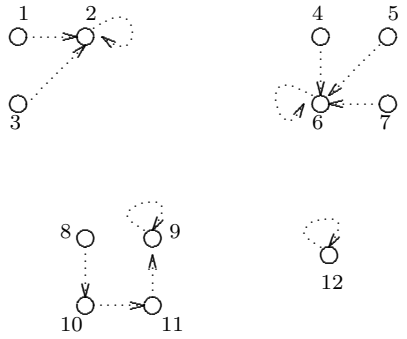
Auf diese Weise entsteht eine Baumstruktur, die von `FIND` durchlaufen werden muss. Leider können dadurch natürlich auch längere Pfade entstehen. Mit der Regellung, dass die Verweise immer von den kleineren zu den größeren ZHKs gerichtet sind, soll erreicht werden, dass möglichst wenige lange Pfade entstehen.

3. Idee: Jedesmal wenn man bei einem `FIND`-Aufruf einen Pfad durchläuft, wird der Pfad komprimiert.



Auf diese Weise wird erreicht, dass eventuell entstehende lange Pfade maximal einmal durchlaufen werden.

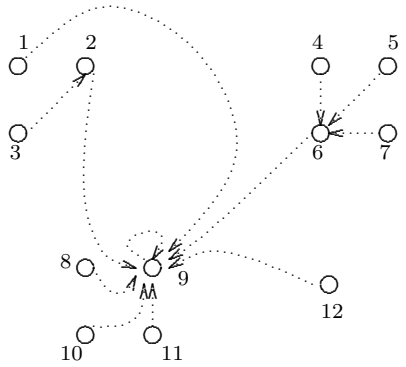
Beispiel einer *UNION-FIND*-Datenstruktur



Führt man auf der angegebenen *UNION-FIND*-Struktur nacheinander die folgenden UNION-Operationen aus,

```
union(3,8), union(4,11), union(12,1)
```

so entsteht die folgende Struktur:



Die resultierende maximale Pfadlänge beträgt also 2.

Beobachtung: Lange Pfade entstehen nur selten und werden bei `FIND`-Operationen nur einmal durchlaufen und danach sofort komprimiert.

Folgerung: Im Durchschnitt ergibt sich pro Aufruf von `UNION` oder `FIND` nur ein fast konstanter Zeitaufwand (wird noch genauer erklärt).

8.4.5 Implementierung

Bei der folgenden C++-Implementierung sollen die Knoten eines Graphen zur Vereinfachung von 0 bis $n-1$ nummeriert sein. Markierungen der Knoten sollen einfach in Feldern der Größe n abgelegt werden. Wir benutzten die Felder `NEXT` und `SIZE`, um bei einem Knoten den Verweis auf den Repräsentanten, bzw. die Größe einer Zusammenhangskomponente zu speichern.

```
int NEXT[n];
int SIZE[n];
```

Die Initialisierung im *Kruskal*-Algorithmus, bei der `n` einelementige ZHKs gebildet werden müssen, kann dann folgendermaßen realisiert werden.

```
for (i=0; i<n; i++)
  {
   NEXT[i]=i; // Knoten i zeigt auf sich selber
   SIZE[i]=1; // Knoten i repräsentiert eine ZHK der Größe 1
  }
```

Als nächstes geben wir eine Implementierung von UNION an:

```
void UNION(int a, int b)
{
 // vereinigt die ZHKs der Knoten a und b
 int ZHKa= FIND(a); // Ermittle Repräsentanten von a
 int ZHKb= FIND(b); // Ermittle Repräsentanten von b
 if (ZHKa != ZHKb)
   {// Die kleinere ZHK soll auf die größere verweisen
    if (SIZE[ZHKa] > SIZE[ZHKb])
      {
       SIZE[ZHKa] += SIZE[ZHKb];
       NEXT[ZHKb] = ZHKa;
      }
    else
      {
       SIZE[ZHKb] += SIZE[ZHKa];
       NEXT[ZHKa] = ZHKb;
      }
   }
}
```

Die Laufzeit dieser Prozedur entspricht offensichtlich der doppelten Laufzeit der Funktion FIND. Betrachten wir also zunächst die Implementierung von FIND.

```
int FIND(int a)
{
 // liefert den Repräsentanten der ZHK des Knoten a
 int result = a;
 while (result != NEXT[result])
   result = NEXT[result];
 // Das Funktionsergebnis steht jetzt in result

 // Es folgt noch die Pfadkomprimierung
 while ( a != result)
  {
   int help = NEXT[a];
   NEXT[a] = result; // a zeigt jetzt auf den Repräsentanten
   a = help;
  }

 return (result);
}
```

Die Laufzeit von `FIND` entspricht offensichtlich der Länge des Pfades vom Knoten `a` zum Repräsentanten. Diese Länge schwankt natürlich. Von Interesse ist daher die mittlere Laufzeit der Aufrufe. Oben wurde angekündigt, diese mittlere Laufzeit sei "fast konstant". Tatsächlich kann diese durchschnittliche Laufzeit durch einen Ausdruck $\alpha(m,n)$ abgeschätzt werden, der extrem langsam wächst. In der Praxis, d.h. für die üblicherweise auftretenden Größen von Graphen wird $\alpha(m,n) \leq 3$ gelten. Erst für astronomisch große Graphen, die wir gar nicht in herkömmlichen Rechnern verarbeiten können, kommen auch größere α-Werte vor.
Die extrem langsam wachsende Funktion α ist eine Umkehrfunktion, der extrem schnell wachsenden *Ackermann*-Funktion $A : N \times N \to N$, die folgendermaßen rekursiv definiert ist.

$$\begin{aligned} A(i,1) &= 2 \text{ für alle } i \geq 1 \\ A(1,x) &= 2^x \text{ für alle } x \geq 1 \\ A(i+1,x+1) &= A(i, A(i+1,x)) \text{ sonst} \end{aligned}$$

Das Verhalten dieser Funktion soll anhand einiger Zahlenbeispiele verdeutlicht werden.

$$\begin{aligned} A(1,4) &= 2^4 = 16 \\ A(2,4) &= A(1,A(2,3)) = A(1,16) = 2^{16} = 65536 \\ A(2,3) &= A(1,A(2,2)) = A(1,4) = 16 \\ A(2,2) &= A(1,A(2,1)) = A(1,2) = 2^2 = 4 \end{aligned}$$

Allgemein erhält man beispielsweise für $A(2,x)$:

$$A(2,x) = \underbrace{2^{2^{2^{\cdots}}}}_{x-\text{mal}}$$

Die Definition der Umkehrfunktion α lautet

$$\alpha(m,n) = \min\left\{ z \geq 1 | A(z, 4 \cdot \lceil \frac{m}{n} \rceil) > \log_2 n \right\}$$

und die für uns relevante Laufzeitabschätzung ist:

Satz: $n-1$ `UNION`-Aufrufe und $m \geq n$ `FIND`-Aufrufe brauchen zusammen die Laufzeit $O(m \cdot \alpha(m,n))$.

Da im *Kruskal*-Algorithmus auf jeden `UNION`-Aufruf mindestens 2 `FIND`-Aufrufe kommen, ist die Voraussetzung des Satzes erfüllt, so dass die Laufzeit für alle Aufrufe zusammen bis auf den sehr kleinen Faktor $\alpha(\#\text{FINDs}, \#\texttt{UNIONs})$ linear in der Anzahl der Aufrufe ist. Gegenüber dem Sortieraufwand ist

diese Laufzeit vernachlässigbar, so dass wir insgesamt für den Kruskal-Algorithmus die folgende Laufzeit erhalten:

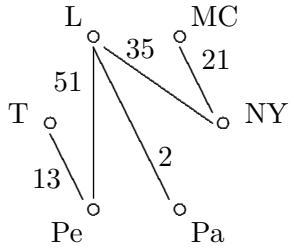
$$T_{Kruskal}(n, m) = \mathrm{O}(n + m \cdot \log m)$$

Beispielanwendung: Ein Anwendungsbeispiel aus dem Bereich der Bildverarbeitung ist die Erkennung von Verkehrszeichen mit Geschwindigkeitsbeschränkungen. Dazu kann man in einer Vorverarbeitung gleichfarbige Regionen des Bildes als interessante Objekte extrahieren. Interpretiert man die Pixel des Bildes als Knoten eines Graphen und verbindet man benachbarte Knoten genau dann mit einer Kante, wenn sie die gleiche Farbe haben, dann sind diese Objekte genau die Zusammenhangskomponenten des Graphen.

Beispielaufgabe: Die folgende Tabelle gibt die Entfernungen (in Einheiten von 100 Meilen) zwischen den Flughäfen der Städte *London*, *Mexiko City*, *New York*, *Paris*, *Peking* und *Tokio* an. Man gebe ein Flugnetz minimaler Länge an, das alle 6 Städte miteinander verbindet.

	L	MC	NY	Pa	Pe	T
L	-	56	35	2	51	60
MC		-	21	57	58	70
NY			-	36	68	68
Pa				-	51	61
Pe					-	13
T						-

Lösung: Mit Hilfe des Kruskal-Algorithmus ergibt sich folgende Lösung:



8.5 Algorithmus von Prim

Ein anderes Verfahren zur Bestimmung eines minimalen aufspannenden Baumes T ist der Algorithmus von Prim. Er sollte natürlich für das oben angegebene Beispiel die gleiche Lösung finden. Wie üblich ist ein Graph $G = (V, E)$ und eine Gewichtsfunktion $w : E \to R_0^+$ gegeben. Die Knoten seien einfach durchnumeriert, d.h. $V = \{1, 2, 3, \ldots, n\}$.

Methode: Algorithmus von Prim

$w[1] \leftarrow 0$;
for $i \leftarrow 2$ to n do $w[i] \leftarrow \infty$;
$S \leftarrow \oslash$; $T \leftarrow \oslash$;
// o.B.d.A. beginnt der Baum am Knoten 1
// S stellt die schon erreichten Knoten dar
// und T die benutzten Kanten

for $i \leftarrow 1$ to n do $e[i] \leftarrow$ `undef`;
// $e[i]$ stellt die zu i führende Baumkante dar

while $S \neq V$ do
  begin
  wähle ein $i \in V \setminus S$ mit minimalem $w[i]$;
  füge i in S ein;
  if $e[i]$ ist definiert
    dann füge $e[i]$ in T ein;

  for all $e[i] \in S \times (V \setminus S)$ do
    begin
    v sei der Endpunkt von e in $V \setminus S$;
    if $w[v] > w(e)$
      then begin
        $w[v] \leftarrow w(e)$;
        $e[v] \leftarrow e$;
        end;
    entferne e im Graphen, um die Kante nicht nochmal zu verwenden;
    end;
  end;

Die Grundidee dieses Algorithmus besteht darin, immer den Knoten an S anzuhängen, der mit den geringst möglichen Kosten angehängt werden kann. Auf diese Weise lässt man die Zusammenhangskomponente S wachsen, bis sie ganz V umfasst.
Bei einer geschickten Implementierung wird man jede Kante nur zweimal benutzen. Durch Knotenmarkierungen erreicht man, dass man jeweils pro Kante in konstanter Zeit erkennen kann, ob die Kante in $V \times (V \setminus S)$ liegt. Die Gesamtlaufzeit beträgt dann $T = O(|V| + |E| \cdot \log |V|)$ wie beim Kruskal-Algorithmus

8.6 Berechnung von Flüssen

Wir wollen uns jetzt mit Transportproblemen befassen, bei denen es darum geht, wie man über ein Netzwerk Objekte von einer Quelle s (*source*) zu einer Senke t (*target*) transportieren kann. Dabei kann es sich beispielsweise um ein System von Pipelines handeln oder um Rechnernetze oder um den Transport von Gütern im Straßennetz. Diese Art von Problemen wird mit Hilfe von Graphen modelliert, bei denen man jeder Kante e eine maximale Kapazität $c(e)$ zuordnet. Beispielsweise bedeutet diese Kapazität eine maximale Anzahl von Personen pro Fahrzeug oder die maximale Anzahl von Fahrzeugen, die eine Straße täglich passieren können, oder die Übertragungsrate einer Standleitung in Bits pro Sekunde.

Gegeben: Ein gerichteter Graph $G = (V, E)$, eine *Kapazitätsfunktion* $c : E \to R_0^+$ und zwei Knoten $s, t \in V$.

Definition: Ein *Fluss* ist ein Abbildung $f : E \to R_0^+$, die folgende Bedingungen erfüllt.

1. $0 \leq f(e) \leq c(e)$ für alle $e \in E$
2. Für jeden Knoten $v \neq s, t$ gilt:

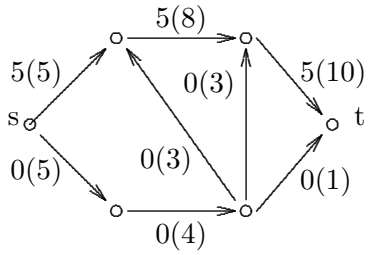
$$\sum_{e=(u,v)\in E} f(e) = \sum_{e=(v,u)\in E} f(e)$$

D.h. Die Stärke des Flusses über eine Kante e ist durch die Kapazität der Kante beschränkt und von der Quelle und Senke abgesehen, fließt aus jedem Knoten genau soviel raus wie rein.

Gesucht: Ein maximaler Fluss von s nach t.

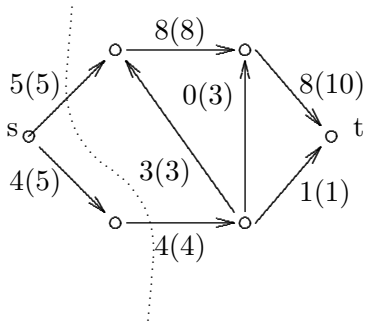
Im folgenden Beispielgraph sind die Kantenkapazitäten in Klammern angegeben. Zusätzlich ist ein Fluss von s nach t durch Kantenbeschriftungen angegeben.

Beispiel: Fluss von s nach t der Stärke 5



Man kann sich leicht davon überzeugen, dass die Kantenbeschriftungen die Bedingungen eines Flusses erfüllen, also tatsächlich ein Fluss vorliegt. Die Summe der aus s fließenden Ströme, bzw. der in t ankommenden Ströme beträgt 5. Wir sagen daher auch, dass der Fluss von s nach t die Stärke 5 besitzt. Durch Ausprobieren findet man leicht auch stärkere Flüsse. Als Beispiel ist im folgenden für den gleichen Graphen ein Fluss der Stärke 9 angegeben.

Beispiel: Maximaler Fluss von s nach t der Stärke 9



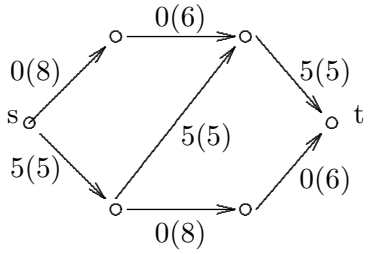
minimaler Schnitt

Dieser Fluss ist maximal, denn es gibt einen Schnitt zwischen s und t, dessen Kapazität genauso groß ist, wie die Stärke des Flusses. Offensichtlich ist dieser Schnitt ein Engpass, der stärkere Flüsse verhindert. Umgekehrt kann man auch kleinste Schnitte dadurch finden, dass man maximale Flüsse konstruiert, denn es gilt folgende Äquivalenz.

Satz: Die Stärke eines maximalen Flusses ist gleich der Kapazität eines minimalen Schnitts

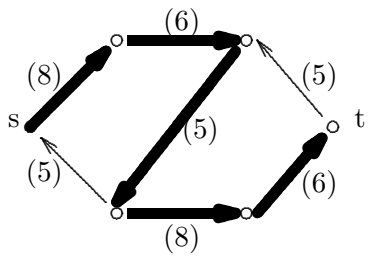
Damit haben wir ein Entscheidungskriterium, das uns zeigt, wann wir mit der Suche nach einem stärkeren Fluss aufhören können. Im folgenden wollen wir überlegen, wie man die Suche nach einem maximalen Fluss automatisieren kann. Dazu zunächst ein etwas komplizierteres Beispiel.

Beispiel: Gegebenen Graph mit einem Fluss der Stärke 5



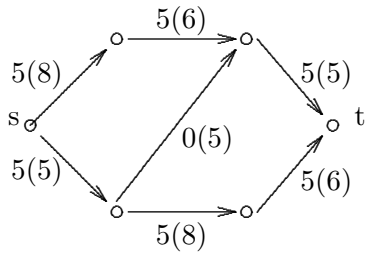
In diesem Beispiel ist bereits ein Fluss der Stärke 5 eingetragen und es soll versucht werden, diesen Fluss zu verbessern. Dazu konstruieren wir einen Hilfsgraphen, der alle möglichen Flussänderungen darstellt. Ist beispielsweise der Fluss $f(e)$ über eine Kante e noch kleiner als die Kantenkapazität $c(e)$, so tragen wir diese Kante mit ihrer verbleibenden freien Kapazität $c(e) - f(e)$ ein. Außerdem fügen wir für jede Kante e mit $f(e) > 0$ eine Kante mit Gegenrichtung und Kapazität $f(e)$ in den Hilfsgraphen ein, um zu kennzeichnen, dass der bisherige Fluss auch reduziert werden kann.
Für das angegebene Beispiel sieht der Hilfsgraph dann folgendermaßen aus.

Erster Hilfsgraph zum obigen Beispiel



Um nun den Fluss von s nach t zu verbessern, sucht man in diesem Hilfsgraphen einen Weg von s nach t. Beispielsweise kann dies mit dem BFS-Algorithmus in linearer Zeit $O(|V|+|E)$ geschehen. In der Abbildung ist ein solcher Weg bereits hervorgehoben. Er beschreibt den Weg für einen zusätzlichen Fluss, mit dessen Hilfe der bisherige Fluss der Stärke 5 verbessert werden kann. Da die minimale Kantenkapazität dieses Weges im Hilfsgraphen den Wert 5 hat, haben wir einen zusätzlichen Fluss der Stärke 5 gefunden. Die folgende Skizze zeigt den daraus resultierenden Gesamtfluss der Stärke 5+5=10 im Ausgangsgraphen.

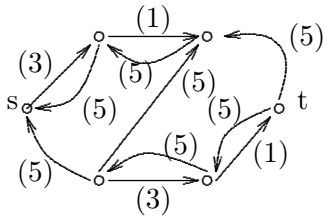
Verstärkter Fluss im Beispiel



Man beachte dabei, dass beispielsweise auf der diagonal verlaufenden Kante der Fluss der Stärke 5 mit dem zusätzlichen Fluss der Stärke 5 in der Gegenrichtung zum neuen Fluss der Stärke 0 verrechnet wurde.
Im allgemeinen ist nicht garantiert, dass immer der bestmögliche zusätzliche Fluss gefunden wird. Deshalb führt man das gesamte Verfahren mit

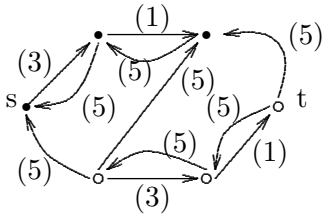
dem bereits verbesserten Fluss immer wieder durch, bis sich keine Verbesserung mehr finden lässt. Im Beispiel erhalten wir dabei den folgenden neuen Hilfsgraphen.

Zweiter Hilfsgraph im Beispiel

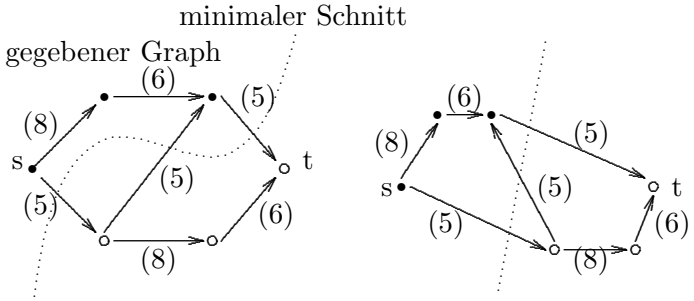


Startet man in diesem Hilfsgraphen den BFS-Algorithmus im Knoten s, um den Weg eines zusätzlichen Flusses nach t zu konstruieren, so stellt man fest, dass der Knoten t gar nicht von s aus erreichbar ist. Dies bedeutet, dass keine Flussverbesserung mehr möglich ist, der bereits gefundene Fluss der Stärke 10 also bereits maximal ist.
In der folgenden Skizze sind die durch den BFS-Algorithmus von s aus erreichten Knoten besonders hervorgehoben. Zwischen diesen Knoten und den anderen nicht erreichten Knoten verläuft der minimale Schnitt zwischen s und t. Er schneidet drei Kanten, von denen allerdings nur zwei Kanten von der "s-Seite" zur "t-Seite" verlaufen. Die Gesamtkapazität dieser beiden Kanten ist tatsächlich 10, also gleich der Stärke des gefundenen Flusses.

Konstruktion des minimalen Schnitts



BFS-Markierungen
im zweiten Hilfsgraphen



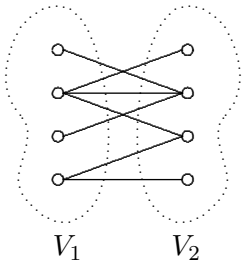
Die Laufzeit zur Berechnung des minimalen Schnitts hängt stark von Details der Implementierung, wie beispielsweise von der geschickten Wahl eines zunehmenden Weges, ab, weil dadurch die Anzahl der BFS-Berechnungen und der Wegberechnungen bestimmt wird. Bei einer guten Implementierung kann die Laufzeit $T = \mathrm{O}(|V|^3)$ erreicht werden.

8.7 Das Heiratsproblem

Im folgenden soll demonstriert werden, wie man Anwendungsprobleme geschickt umformulieren kann, um Lösungen mit Hilfe bekannter Graphenalgorithmen zu konstruieren. Zunächst benötigen wir dazu den Begriff des *bipartiten* Graphen.

Definition: Ein Graph $G = (V, E)$ heißt *bipartit*, wenn die Knotenmenge in zwei disjunkte Teilmengen zerlegt werden kann, d.h. $V = V_1 \dot\cup V_2$ und die Kanten nur zwischen Knoten aus V_1 und V_2 verlaufen, d.h. $E \subseteq V_1 \times V_2$.

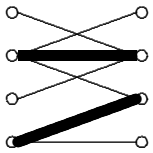
Beispiel: bipartiter Graph



Beispielsweise könnte die Menge V_1 die Männer und V_2 die Frauen in der Kartei eines Ehevermittlungsinstituts darstellen. Die Kanten zwischen Knoten aus V_1 und V_2 geben dann alle möglichen Partnerschaften an und die Aufgabe des Instituts bestünde darin, mit Hilfe dieser Kanten möglichst viele Paare zu bilden.
Genauer: Es ist eine möglichst große Teilmenge M der Kantenmenge E gesucht, so dass keine zwei Kanten aus M einen gemeinsamen Knoten besitzen.
Die obige Aufgabenbeschreibung ist recht einprägsam. Als realistischere Anwendungen betrachte man etwa die Aufgabenverteilung in einer Fabrik. V_1 könnte dann die Menge aller vorliegenden Aufträge darstellen und V_2 die Menge der zur Verfügung stehenden Produktionsmaschinen. Die Kanten geben dann an, welcher Auftrag auf welcher Maschine ausgeführt werden kann. Beispielsweise könnte ein Auftrag mit relativ geringen Qualitätsanforderungen sowohl auf alten, als auch auf modernen Maschinen der Fabrik ausgeführt werden, während besonders anspruchsvolle Aufträge vielleicht nur auf einer ganz bestimmten Maschine ausführbar sind. Die Zielsetzung der

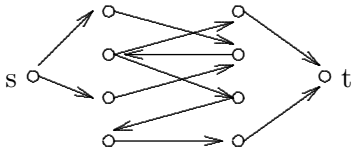
Fabrik ist natürlich, möglichst viele Aufträge gleichzeitig zu erledigen, bzw. eine möglichst hohe Gesamtausnutzung des Maschinenparks zu erreichen.
Ein weiteres Anwendungsbeispiel ist die Ausführung von Computerprogrammen (Jobs) in einem Rechnernetz. Hier ist eine Aufteilung der Jobs auf die Rechner gesucht, bei der möglichst viele Jobs parallel bearbeitet werden können. Die Kanten geben dabei an, welche Jobs auf welchen Rechnern ausgeführt werden können. Mögliche Nebenbedingungen sind dabei etwa die Verfügbarkeit peripherer Geräte, Koprozessoren, interne Speicherkapazität, freie Plattenkapazität, möglicher Direktzugriff auf Datenbestände, ...
Eine beliebige Korrespondenz (ein *Matching*) kann man leicht ermitteln, indem man die Kanten in beliebiger Reihenfolge durchläuft und sie auswählt, wenn die Endknoten noch nicht zu bereits ausgewählten Kanten gehören. Im folgenden ist ein Matching der Größe 2 zum oben gezeigten Beispielgraphen angegeben.

Beispiel: Matching der Größe 2



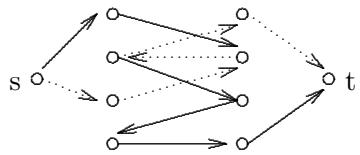
Ein maximales Matching kann man schrittweise konstruieren, indem man ein vorhandenes Matching verbessert. Dazu kann man das Problem in ein Flussproblem umwandeln:
Wir führen zwei zusätzliche Knoten s und t ein und tragen gerichtete Kanten von s zu allen unbenutzten Knoten aus V_1 ein und von allen unbenutzten Knoten aus V_2 nach t.

Beispiel: Konstruktion eines Flussproblems



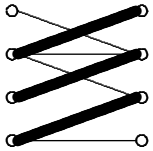
Für jede noch unbenutzte Kante $e \in E \setminus M$ fügen wir eine entsprechende von V_1 nach V_2 gerichtete Kante ein. Für jede Matching-Kante $e \in M$ fügen wir eine von V_2 nach V_1 gerichtete Kante ein.
Anschließend erhält jede Kante die Kapazität 1 und wir suchen einen Fluss von s nach t (z.B. mit dem BFS-Algorithmus). Wenn wir einen solchen Fluss finden, können wir ihn benutzen, um das Matching M zu vergrößern.

Beispiel: Lösung des Flussproblems



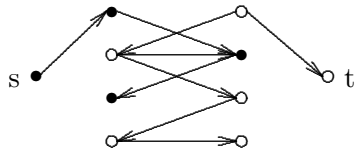
Dazu machen wir alle vom Fluss benutzten $V_1 \times V_2$-Kanten zu Matching-Kanten und entfernen alle vom Fluss benutzten $V_2 \times V_1$-Kanten aus dem Matching. Tatsächlich führt diese Modifikation immer zu einer Vergrößerung des Matchings M, weil jeder Weg von s nach t immer eine Kante mehr aus $V_1 \times V_2$ als aus $V_2 \times V_1$ enthält.

Beispiel: Verbessertes Matching



Dies Verfahren wiederholt man, bis sich kein Fluss mehr von s nach t konstruieren lässt. Für unser Beispiel ist das gefundene Matching der Größe 3 bereits maximal, so dass sich bereits im nächsten konstruierten Flussproblem kein Weg mehr von s nach t finden lässt,

Beispiel: Zweites Flussproblem



Basierend auf dem vorgestellten Verfahren lässt sich ein maximales Matching für bipartite Graphen konstruieren. Dabei kann man ausnutzen, dass es sich bei den Flussproblemen um besonderes einfache Varianten handelt. Außerdem kann man bei etwas geschickter Vorgehensweise in jedem Schritt maximale Verbesserungen des alten Matchings erreichen, so dass sich insgesamt eine Laufzeit von $T = \mathrm{O}(\sqrt{|V|} \cdot |E|)$ erreichen lasst.

9 NP-schwere Probleme

Alle bisher in der Vorlesung betrachteten Probleme waren insofern gutartig, als dass recht effiziente Algorithmen zur Lösung der Probleme angegeben werden konnten. Zum Abschluss der Vorlesung sollen jetzt noch einige besonders schwierige Probleme betrachtet werden, für die bislang nur Algorithmen mit exponentieller Laufzeit existieren. Überraschenderweise sehen einige dieser schwierigen Probleme auf den ersten Blick recht einfach aus, so dass es in der Praxis immer wieder passieren kann, dass man unwissentlich versucht, ein Problem mal eben schnell zu lösen, an dem sich bereits mehrere Generationen von Informatikern vergeblich versucht haben. Es ist deshalb wichtig, zumindest einige typische Vertreter dieser schwierigen Probleme zu kennen, um gegebenenfalls bei Programmentwicklungen vorgewarnt zu sein. Um formal zwischen verschiedenen Schwierigkeitsklassen unterscheiden zu können, führt man die folgenden Definitionen der Problemklassen P und NP ein.

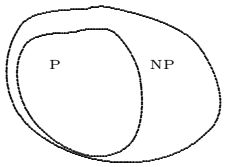
Definition: Mit P bezeichnet man die Menge aller Probleme, die sich auf einem herkömmlichen deterministischen Rechner in polynomieller Zeit bearbeiten lassen.

Die polynomielle Laufzeit bedeutet dabei, dass sich die Laufzeit durch ein Polynom in der Problemgröße n abschätzen lässt. Beispiele sind etwa die Laufzeiten $T = \mathrm{O}(n^2)$, $T = \mathrm{O}(n \cdot \log n)$, oder auch die extrem hohe Laufzeit $T = \mathrm{O}(n^{523})$. Zur Vereinfachung sollen alle polynomiellen Laufzeiten im Gegensatz zu exponentiellen Laufzeiten als "praktikabel" angesehen werden. Unter einem deterministischen Rechner versteht man einen Computer, bei dem alle Arbeitsabläufe eindeutig festgelegt sind, so dass mehrmalige Programmausführungen mit gleichen Parametern immer genauso ablaufen. Im Gegensatz dazu könnte eine nicht-deterministische Maschine Zufallsentscheidungen verwenden, oder gar Zwischenergebnisse auf unerklärliche Art raten. Solche (nicht existierenden) Maschinen werden zur Definition der Klasse NP benutzt.

Definition: Mit NP bezeichnet man die Menge aller Probleme, die sich auf einem nicht-deterministischen Rechner in polynomieller Zeit bearbeiten lassen.

Beispielsweise könnte man das *Faktorisierungsproblem* für n-stellige Zahlen leicht mit einer nicht-deterministischen Maschine lösen und so viele Verschlüsselungsverfahren knacken. Oft basieren Codierungsverfahren nämlich darauf, dass man eine große n-stellige Zahl x in zwei Primfaktoren p und q zerlegen muss. Die nicht-deterministische Maschine rät dazu zur Zahl x einfach die Faktoren p und q (in konstanter Zeit) und führt eine einfache

Multiplikation zur Probe aus, um die Korrektheit des Ergebnisses zu überprüfen. Da die Laufzeit für die Multiplikation sicher polynomiell in n ist, ist das Problem offensichtlich in der Klasse *NP*.
Will man das gleiche Problem deterministisch lösen, so kann man etwa alle Zahlen zwischen 2 und $\sqrt{x}$ nacheinander als Teiler ausprobieren. Dies sind allerdings $2^{n/2}$ Versuche, so dass man also eine exponentielle Laufzeit braucht. Hätte man beliebig viele Computer zur Verfügung, so könnte man natürlich alle diese Versuche auf $2^{n/2}$ verschiedenen Computern parallel probieren lassen, so dass das Ergebnis genauso schnell ermittelt würde, wie bei der nicht-deterministischen Maschine.
Wegen dieser Simulationsmöglichkeit sind die nicht-deterministischen Maschinen keineswegs so unanschaulich, wie es auf den ersten Blick erscheinen mag. Tatsächlich kann man jedes Problem der Klasse *NP* mit exponentiellem Aufwand auf einem herkömmlichen deterministischen Computer lösen.
Die beiden Definitionen legen die Vermutung nahe, dass die Klasse *NP* wesentlich größer ist als die Klasse *P* und dass *NP* die Klasse *P* umfasst. Diese Situation ist in der folgenden Skizze dargestellt.



Die Beziehung $P \neq NP$ würde bedeuten, dass man die bekannten Lösungsalgorithmen mit exponentieller Laufzeit für *NP*-Probleme nicht durch schnellere polynomielle Algorithmen ersetzen kann. Tatsächlich sind bislang alle solchen Optimierungsversuche gescheitert.
Umgekehrt konnte bislang aber auch für keines dieser Probleme bewiesen werden, dass die exponentielle Laufzeit für Lösungsalgorithmen tatsächlich notwendig ist. D.h. es ist unbekannt, ob die perfekte Ratefähigkeit nichtdeterministischer Maschinen tatsächlich nicht in polynomieller Zeit deterministisch simuliert werden kann.
Es ist also ein **offenes Problem**, ob *NP* eine echte Obermenge von *P* ist.
Innerhalb der Klasse *NP* hat man eine Teilmenge von Problemen definiert, die alle gleich schwierig sind und für die man nicht weiß, ob sie in *P* liegen. Diese besonders interessante Teilklasse wird die **Klasse der NP-vollständigen Probleme** genannt. Wenn es gelingt, irgendein Problem dieser Klasse in polynomieller Zeit auf einem deterministischen Rechner zu lösen, dann kann man alle *NP-vollständigen* Probleme in polynomieller Zeit auf deterministischen Rechnern lösen.
Alle im folgenden vorgestellten Probleme sind als NP-vollständig bekannt. Für sie kennt man also bislang keine polynomiellen Lösungsalgorithmen (auf herkömmlichen Maschinen).

9.1 Beispielprobleme

9.1.1 Das Travelling-Salesman-Problem (TSP)

Beim *Travelling-Salesman-Problem* (*TSP*) oder dem *Problem des Handlungsreisenden* geht es darum, eine optimale Rundreise durch mehrere Städte zu planen.

Gegeben ist ein vollständiger ungerichteter Graph mit n Knoten und eine Gewichtsfunktion $c : E = (V \times V) \to R_0^+$.

Gesucht ist eine Rundreise mit minimalem Gewicht, die alle Knoten miteinander verbindet und keine Kante mehrmals benutzt. D.h. gesucht ist der billigste Kreis durch alle Knoten.

Anwendungsbeispiel: Man betrachte die folgende Distanzmatrix zwischen den Städten Aachen, Basel, Berlin, Düsseldorf, Frankfurt, Hamburg, München, Nürnberg und Stuttgart

	Aa	Ba	Be	Dü	Fr	Ha	Mü	Nü	St
Aa	0	57	64	8	26	49	64	47	46
Ba	57	0	88	54	34	83	37	43	27
Be	64	88	0	57	56	29	60	44	63
Dü	8	54	57	0	23	43	63	44	41
Fr	26	34	56	23	0	50	40	22	20
Ha	49	83	29	43	50	0	80	63	70
Mü	64	37	60	63	40	80	0	17	22
Nü	47	43	44	44	22	63	17	0	19
St	46	27	63	41	20	70	22	19	0

Wie sieht die kürzeste Rundreise durch diese Städte aus?

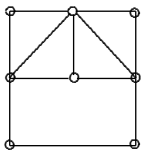
Wir werden später auf diese Beispielaufgabe zurückkommen und verschiedene Lösungsansätze diskutieren.

9.1.2 Das Hamilton-Kreis-Problem

Mit dem TSP verwandt ist das *Hamilton-Kreis-Problem*

Gegeben ist ein ungerichteter Graph.

Gesucht ist ein Kreis, der alle Knoten miteinander verbindet und keinen Knoten mehrfach durchläuft.



Das angegebene Beispiel ist so konstruiert, dass kein solcher Kreis existiert. Tatsächlich liegt bei vielen NP-vollständigen Problemen die Schwierigkeit darin, zu entscheiden, ob überhaupt eine Lösung existiert.

9.1.3 Das Erfüllbarkeitsproblem 3-SAT

Beim Erfüllbarkeitsproblem soll man für eine boolesche Funktion Argumente finden, so dass die Funktion den Wert 1 annimmt.

Gegeben ist eine boolesche Funktion mit mindestens drei Variablen.

Gesucht ist eine Belegung der Variablen, so dass die Funktion den Wert 1 liefert.

Beispiel: $f(x_1, x_2, x_3, x_4) = \overline{x_1 + x_2 + x_3} \cdot (x_3 x_4)$

In dem Beispiel soll man also boolesche Werte x_1, x_2, x_3, und x_4 finden, so dass $f(x_1, x_2, x_3, x_4) = 1$ gilt. Natürlich kann man prüfen, ob eine solche Lösung existiert und falls ja, wie sie aussieht. Dazu braucht man nur die 2^4 möglichen Belegungen der Argumente ausprobieren. Allerdings bedeutet dies eine exponentielle Anzahl von Versuchen in der Anzahl der Variablen.

9.1.4 Das Rucksackproblem

Beim Rucksackproblem geht es anschlaulich darum, welche Gegenstände ein Bergsteiger oder ein Motorradfahrer auf eine Tour mitnehmen sollte. Jeder Gegenstand hat einen bestimmten Wert und ein bestimmtes Gewicht oder Größe. Man muss nun versuchen, den Gesamtwert der gewählten Gegenstände zu maximieren, ohne das zulässige Maximalgewicht zu überschreiten.

Gegeben: Ein Vektor $(w_1, w_2, \ldots, w_n) \in R^+$, ein Vektor $(g_1, g_2, \ldots, g_n) \in R^+$ und ein Wert $G \in R^+$.

Gesucht: Eine Menge $I \subseteq \{1, 2, \ldots, n\}$ von Indizes, so dass

1. $\sum_{i \in I} g_i \leq G$
2. $y = \sum_{i \in I} w_i$ ist maximal

9.2 Lösungsansätze

Um NP-schwere Probleme in der Praxis zu bearbeiten gibt es verschiedene Möglichkeiten:

1. Man versucht Näherungslösungen mit möglichst schnellen polynomiellen Verfahren zu konstruieren.

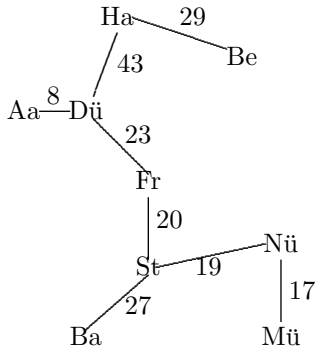
2. Man sucht geschickte exponentielle Verfahren, die das Optimum suchen.

Beides soll im folgenden anhand des Travellings-Salesman-Problems vorgestellt werden.

9.2.1 Näherungen für das Travelling-Salesman-Problem

Wenn die optimale Tour bereits bekannt wäre und wir aus ihr eine Kante entfernen würden, erhielten wir einen aufspannenden Baum. Folglich muss die Länge der optimalen Tour länger sein als die Länge eines minimalen aufspannenden Baumes.
In Zeit $O(|E| \cdot \log |V|)$ können wir einen minimalen aufspannenden Baum ermitteln und seine Länge als untere Schranke für die Länge der optimalen Tour benutzen.

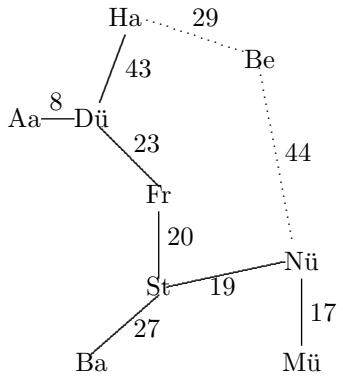
Beispiel: Für das oben vorgestellte Problem der Rundreise durch die Städte Aachen, Basel, Berlin, Düsseldorf, Frankfurt, Hamburg, München, Nürnberg und Stuttgart ergibt sich beisielsweise der folgende minimale aufspannende Baum.



Dieser Baum hat eine Länge von 186. Dies ist also eine untere Schranke für die Länge einer optimalen Rundreise.

Bei dieser Abschätzung machen wir allerdings einen prinzipiellen Fehler, weil wir statt der n Kanten einer Rundreise im Baum nur $n-1$ Kanten benutzen. Eine bessere Abschätzung sieht folgendermaßen aus. Man konstruiert einen minimalen Baum, der $n-1$ Städte untereinander verbindet und addiert zu dessen Länge die 2 kürzesten Kanten des unbenutzten Knotens dazu. Dies liefert ebenfalls eine untere Schranke für die optimale Tour, weil jeder Knoten bei der Tour mit einer Kante erreicht und mit einer anderen wieder verlassen werden muss.

Beispiel: Für die Städterundreise ignorieren wir zunächst den Knoten "Berlin", weil die Summe seiner beiden kürzesten Kanten maximal ist. (29+44)



Auf diese Weise erhalten wir als verbesserte untere Schranke den Wert 230.

Obere Schranken für die Länge der optimalen Tour erhält man dadurch, dass man mit irgendeiner "geschickten" Heuristik eine "gute" Tour konstruiert. Beispielsweise kann man mit irgendeiner Tour durch ein Paar Städte beginnen und nacheinander immer weitere Städte in die Tour einschieben. Eine recht gute Methode geht dabei so vor, dass immer die am weitesten entfernte Stadt dazugenommen wird:

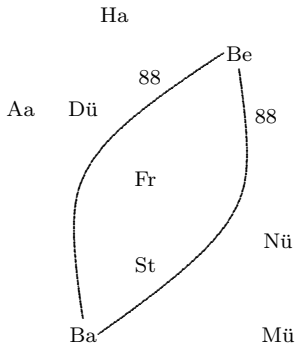
// Die Tour hat die Länge 0 und startet und endet in s
$T \leftarrow (s, s)$; // Tourenbeschreibung
$K \leftarrow \{(s, s)\}$; // Kantenmenge
$w \leftarrow 0$; // Tourenlänge
// Abstände der anderen Knoten von s aus berechnen
for $u \leftarrow 1$ to n do
  $d[u] \leftarrow w_{(s,u)}$;

for $i \leftarrow 1$ to $n - 1$ do
  begin
  Wähle j mit $d[j]$ maximal;
  Berechne für jede Kante $e = (x, y) \in K$:
    $c[e] \leftarrow w_{(x,j)} + w_{(j,y)} - w_{(x,y)}$;
  Wähle eine Kante $f = (u, v) \in K$ mit minimalem c-Wert;
  // f ist die am günstigsten zu ersetzende Kante der Tour
  Füge j zwischen u und v in die Tour T ein
  $K \leftarrow (K \setminus \{f\}) \cup \{(u, j), (j, v)\}$;
  $w \leftarrow w + c[f]$;
  // Aktualisiere die Entfernungen der Städte von der Tour
  $d[j] \leftarrow 0$;
  for $x \in V \setminus T$ do
    $d[x] \leftarrow \min\big(d[x], w_{(j,x)}\big)$;
  end;

Beispiel: Für das Beispiel der Städterundreise beginnen wir willkürlich mit $s =$ Berlin. Der d-Vektor wird dann mit den Entfernungen der anderen Städte nach Berlin initialisiert ("Be"-Zeile der Distanzmatrix).

	Aa	Ba	Be	Dü	Fr	Ha	Mü	Nü	St
$d =$	64	88	0	57	56	29	60	44	63

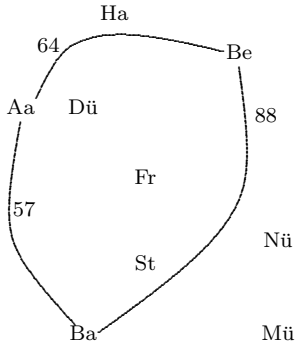
Der Algorithmus wählt dann als nächste aufzunehmende Stadt $j =$ Basel, weil Basel den größten d-Wert hat. Das Einfügen von Basel erzeugt dann die folgende Rundreise



der d-Vektor wird dann mit Hilfe der Städte-Entfernungen nach Basel aktualisiert:

	Aa	Ba	Be	Dü	Fr	Ha	Mü	Nü	St
$d =$	57	0	0	54	34	29	37	43	27

Beim nächsten Schleifendurchlauf muss also Aachen als die mit dem Wert 57 am weitesten entfernte Stadt eingefügt werden. Dazu wird die Kante (Berlin,Basel) durch die beiden Kanten (Berlin,Aachen) und (Aachen,Basel) ersetzt. Die neue Rundreise sieht folgendermaßen aus:

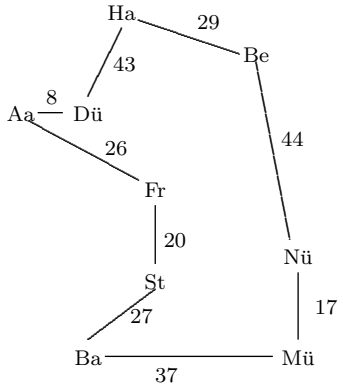


Der neue d-Vektor lautet:

	Aa	Ba	Be	Dü	Fr	Ha	Mü	Nü	St
$d =$	0	0	0	8	26	29	37	43	27

Deshalb muss jetzt $j =$ Nürnberg gewählt werden. Die zu ersetzende Kante wird die Kante (Basel,Berlin) sein.

Führt man das Verfahren entsprechend fort, ergibt sich schließlich die folgende Rundreise.



Die erreichte Tourenlänge beträgt 251.

9.2.2 Dynamische Programmierung für das TSP

Abschließend soll noch auf eine programmtechnische Möglichkeit eingegangen werden, die optimale Lösung algorithmisch zu ermitteln. Dazu verwenden wir die Methode der *dynamischen Programmierung*, mit der wir die $(n-1)!$ verschiedenen Touren auf $n \cdot 2^n$ Fälle reduzieren können. Obwohl dies eine drastische Reduzierung der zu untersuchenden Fälle ist, bleibt es natürlich bei einer exponentiellen Laufzeit. Wir erhalten $T = \mathrm{O}\left(n^2 \cdot 2^n\right)$.
Prinzipiell geht die *dynamische Programmierung* so vor, dass zunächst kleine Probleme gelöst werden und diese Teillösungen dann schrittweise zu Lösungen größerer Probleme fortgesetzt werden.
Für das TSP-Problem geht dies folgendermaßen: Sei $C(S,i)$ die Länge einer optimalen Tour, die im Knoten 1 startet, durch alle Knoten der Menge $S \subset V \setminus \{1\}$ führt und schließlich im Knoten i endet. Dann gilt

$$\begin{aligned} C(\{i\},i) &= w[1,i] \quad \text{für } 2 \le i \le n \\ \text{und} & \\ C(S,i) &= \min_{k \in S \setminus \{i\}} \left(C(S \setminus \{i\},k) + w[k,i]\right) \quad \text{für } |S| \ge 2 \end{aligned}$$

Die erste Gleichung ist trivial und besagt nur, dass die optimale Weglänge vom Knoten 1 zum Knoten i gleich der Länge der Kante $(1,i)$ ist (wenn kein

weiterer Knoten benutzt werden darf). Die zweite Gleichung berechnet die Länge der optimalen Tour von 1 über die Knoten aus S zum Knoten i. Auf dieser Tour muß es natürlich einen (noch unbekannten) Vorgängerknoten k von i geben. Die Tourenlänge ergibt sich dann aus der optimalen Tour von 1 aus über die Knoten aus $S \setminus \{i\}$ nach k zuzüglich der Kante von k nach i. Mit Hilfe dieser beiden Gleichungen lassen sich rekursiv alle relevanten Tourenlängen berechnen. Die Länge der gesuchten optimalen Rundreise ergibt sich dann, indem man vom Zielknoten i zum Ausgangsknoten 1 zurückgeht. Dabei muss man natürlich alle möglichen i ausprobieren:

$$\text{Länge der optimalen Tour} \quad = \min_{2 \leq i \leq n} \left(C(\{2, 3, \ldots, n\}, i) + w[i, 1] \right)$$

Für die Implementierung der Methode bietet sich die rekursive Programmierung an, weil man im wesentlichen nur die oben angegebenen Formeln umsetzen muss. Dabei werden aber viele C-Werte mehrfach berechnet, was zu einer unnötig hohen Laufzeit führt. Geschicker ist es, alle berechneten C-Werte in einer Tabelle abzulegen und bei jedem Funktionsaufruf zunächst zu überprüfen, ob der betreffende Wert bereits in der Tabelle abgelegt ist. Ist der Wert bereits bekannt, so benutzt man ihn, ohne die Berechnung erneut durchzuführen.
Will man nicht nur die Länge der optimalen Tour berechnen, sondern auch deren Verlauf, so muss man zusätzlich bei jedem $C(S, i)$ registrieren über welchen Wert k das Minimum zustandekommt. Dann kann man nachträglich die Knotenfolge der Tour aus den Tabellenwerten ablesen.
Zur Veranschaulichung ist im folgenden die einfache rekursive Implementierung zur Berechnung der Tourenlänge in Java skizziert.

```
public int C(Set<Integer> s, Integer i)
 {int result;
  if (s.size()==1)
     result = w[1][i];
  else
    {result=Integer.MAX_VALUE;
     // smallset = s - i
     Set<Integer> smallSet= new HashSet<Integer>();
     smallSet.addAll(s);
     smallSet.remove(i);
     for (Integer k:smallSet)
        {int help= C(smallSet,k) + w[k][i];
         if (help<result)
           result=help;
        }
    }
  return result;
 }
```

10 Literatur

Sedgewick, Robert; Wayne, Kevin: *Algorithmen und Datenstrukturen*, Pearson Studium, 2014, ISBN: 978-3-86894-184-5

Saake, Gunter; Sattler, Kai-Uwe: *Algorithmen und Datenstrukturen*, dpunkt.verlag, 2013, ISBN: 978-3-86490-136-2