Kurs:Algorithmen und Datenstrukturen

de.wikiversity.org

27. Februar 2016

Inhaltsverzeichnis

1 Einleitung

1.1 Algorithmen im Alltag

- Bedienungsanleitungen
- Gebrauchsanleitungen
- Bauanleitungen
- Kochrezepte
- Berechnungsvorschriften (z.B. Berechnung der Fakultät)

1.2 Intuitive Begriffserklärung Algorithmus

„Ein Algorithmus[1] ist eine präzise (d.h. In einer festgelegten Sprache formulierten), endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer Verarbeitungsschritte."

1.3 Definitionen

Algorithmus

- „systematische Verarbeitung"
- Eine eindeutige Beschreibung eines in mehreren Schritten durchgeführten Bearbeitungsvorgangs
- Ein Algorithmus ist ein allgemeines Verfahren zur Lösung eines Problems ohne Bezug auf einen konkreten Prozessor.

Programm

- Ein Programm[2] ist eine konkrete Formulierung eines Algorithmus für eine konkrete Klasse von Prozessoren.

Prozessor

- Ein Prozessor[3] ist etwas, das die Fähigkeit hat, Programme auszuführen.

Datenstrukturen

- „Ordnungsschema"

1 `https://de.wikipedia.org/wiki/Algorithmus`
2 `https://de.wikipedia.org/wiki/Computerprogramm`
3 `https://de.wikipedia.org/wiki/Prozessor`

- Eine Struktur zur Verwaltung von Daten
- Darstellung von Informationen in maschinenverarbeitbarer Form
- Charakterisieren Daten und mögliche Operationen auf Daten

1.4 Transformationelle Probleme

Ein Algorithmus definiert eine Transformation auf dem gesamten, durch die Eingaben definierten Zustand, aus dem als Bedeutung dann die Werte der Ausgabevariablen ausgelesen werden. Das heißt, ein Algorithmus benutzt kein weiteres Wissen neben der Eingabe und hat keine Seiteneffekte!

1.5 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[4] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 2.1 zu finden.

1.6 Eigenschaften von Algorithmen

Ein Algorithmus heißt ...

- terminierend, wenn er für alle zulässigen Schrittfolgen stets nach endlich vielen Schritten endet
- deterministisch, wenn in der Auswahl der Verarbeitungsschritte keine Freiheit besteht
- determiniert, wenn das Resultat eindeutig bestimmt ist
- sequenziell, wenn die Schritte stets hintereinander ausgeführt werden
- parallel oder neben läufig, wenn gewisse Verarbeitungsschritte nebeneinander (im Prinzip gleichzeitig) ausgeführt werden
- korrekt, wenn das Resultat stets korrekt ist
- effizient, wenn das Resultat in „annehmbarer“ Zeit geliefert wird

1.7 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[5] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 2.1.1 zu finden.

4 `http://www.dpunkt.de/buecher/3358.html`
5 `http://www.dpunkt.de/buecher/3358.html`

2 Algorithmenentwurf

Dieses Kapitel behandelt die Vorgehensweise zum Algorithmenentwurf.

2.1 Vom Algorithmus zur Programmausführung

1. Der Algorithmus wird unabhängig von Programmiersprache und Rechnerhardware entworfen
2. Der Algorithmus wird in einer höheren Programmiersprache, z.B. Java, programmiert
3. Das Programm wird in Maschinensprache übersetzt
4. Die CPU interpretiert den Maschinencode und das Programm wird ausgeführt

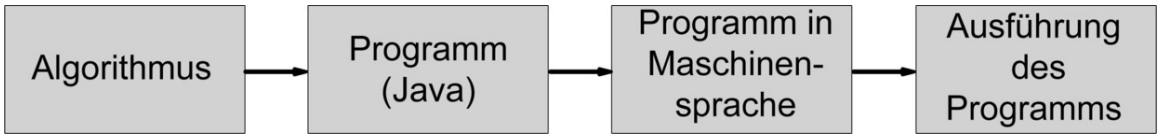

Abb. 1 Vom Algorithmus zum Programm

2.2 Vorgehensweise Algorithmus-Entwurf

- **Hintergrundwissen erwerben:** Derjenige der ein Problem beschreibt ist oft nicht derjenige, der den Algorithmus entwirft, dadurch kommt es zu unklaren Aufgabenstellungen, unterschiedlichem Vorwissen und verschiedenen Annahmen.
- **Problem definieren:** Erfordert Hintergrundwissen und Übung in der Definition von Problemen
- **Algorithmus entwerfen:** Erfordert Wissen zu Algorithmen und Datenstrukturen
- **Programm erstellen:** Erfordert Wissen über Programmiersprache (Java) und Programmierung
- **Lösung überprüfen:** Erfordert methodisches Wissen zu Termination, Korrektheit und Effizienz

2.3 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um

1 `http://www.dpunkt.de/buecher/3358.html`

das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 2 und 8.1 zu finden.

3 Größter gemeinsamer Teiler

In diesem Kapitel wird die im vorherigen Kapitel vorgestellte Vorgehensweise zum Algorithmenentwurf am Beispiel des größten gemeinsamer Teilers[1] gezeigt.

3.1 Hintergrundwissen

Gegeben zwei positive natürliche Zahlen a und b, welche ist die größte positive natürliche Zahl x, so dass x s
natürliche Zahl x' gibt, so dass x' > x und x' teilt sowohl a als auch b.

- Alle Variablen bezeichnen natürliche Zahlen größer 0
- $x|a \Longleftrightarrow \exists\alpha : a = \alpha \wedge x(x\ teilt\ a)$
- $x = ggT(a,b) \Longleftrightarrow x|a \wedge x|b \wedge \forall y : (y|a \wedge y|b \Rightarrow y|x)$
- Anwendungsbeispiel Kürzen: 52/32 hat 4 als ggT, mit 4 gekürzt ergibt sich 13/8

3.2 Problem definieren

Wir betrachten (i. Allg.) hier transformationelle Probleme

```
Problem: ggT-Berechnung
Eingabe: zwei Zahlen a,b ∈ N
Ausgabe: der größte gemeinsame Teiler von a und b
```

Algorithmus definiert also eine Transformation auf dem gesamten, durch die Eingaben definierten Zustand, aus dem als Bedeutung dann die Werte der Ausgabevariablen ausgelesen werden.

3.3 Algorithmus entwerfen

Verfahren von Euklid (300 v. Chr.) für natürliche Zahlen:

1. $b|a \Rightarrow b = ggT(a,b)$
2. $\neg(b|a) \Rightarrow ggT(a,b) = ggT(b, a\%b)$

"%" ist die Modulu Funktion: $r = a\%b \Longleftrightarrow 0 \leq r \leq b \wedge \exists\alpha : a = \alpha * b + r$

1 https://de.wikipedia.org/wiki/Gr%C3%B6%C3%9Fter_gemeinsamer_Teiler

```
ggT(46,18) = ggT(18,10)    (α=2, b=18, r=10)
           = ggT(10,8)     (α=1, b=10, r=8)
           = ggT(8,2) = 2  (α=1, b=8, r=2)
```

In Worten erklärt:

- Wie oft passt 18 in 46? → 2 mal (α)
- 2*18 ist 36, zur 46 fehlen somit noch 10 (r)
- Wie oft passt 10 in 18? → 1 mal (α)
- 1*10 ist 10, zur 18 fehlen somit noch 8 (r)
- Wie oft passt 8 in 10? → 1 mal (α)
- 1*8 ist 8, zur 10 fehlen somit noch 2 (r)
- 8 passt 0 mal in die 2, somit ist der ggT die 2

Idee: Führe die Berechnung von ggT(a,b) auf die Berechnung von ggT(b, a % b) zurück (falls b|a, ansonsten ist das Ergebnis b).

Vorbedingung: Eine Bedingung zur Ausführung des ggT(a,b) ist, dass a,b>0

Wie kann man dies sicherstellen?

- Optimistische Strategie
  - Man geht vom Erfüllt sein der Bedingung aus
    - z.B. Clients bekannt und zuverlässig, z.B. bei Rekursion
- Pessimistische Strategie
  - Man überprüft die Bedingung bei jedem Aufruf
    - z.B. Öffentliche APIs
- Möglichkeiten bei nicht erfüllten Vorbedingungen
  - Ausnahmen werfen
  - Parameter auf Defaultwerte setzen (mit Meldung)
  - Programm nicht ausführen und Defaultwert zurückgeben

3.4 Programm erstellen

Pseudocode

```
Algorithmus euklid
Eingabe: Ganze Zahlen a,b
Ausgabe: Ganze Zahl c=ggT(a,b)
  Setze r = a % b;
  Falls r = 0 gib b zurück;
    Ansonsten gib euklid(b,r) zurück;
```

Rekursiv, optimistisch

```
public int ggT(int a, int b){
   int r = a % b;
   if (r == 0)!
           return b;
   else
          return ggT(b,r);
}
```

Iterativ, pessimistisch – Version 1

```
public int ggT(int a, int b){
    if (a<=0 || b<=0)
          throw new ArithmeticError(“negative Daten bei ggt(“+a+“,“+b+“)“);
    else {
          int r = a % b;
          while (r!=0) {
                  a = b;
                  b = r;
                  r = a % b;
          }
          return b;
     }
}
```

Iterativ, pessimistisch – Version 2

```
public int ggT(int a, int b){
    if (a<=0 || b<=0)
          then throw new ArithmeticError(“negative Daten bei ggt(“+a+“,“+b+“)“);
    else {
          do{
             int r = a % b;
             a=b;
             b=r;
         } while (r!=0);
return a;
 }
}
```

3.5 Algorithmenanalyse

Ist unser ein Algorithmus ein guter Algorithmus?

- Wichtige Fragen:

– Terminiert der Algorithmus?

– Ist der Algorithmus korrekt?

– Welche Laufzeit hat der Algorithmus?

1. Theorem:

Für positive natürliche Zahlen a und b mit a > b, terminiert der Algorithmus Euklid nach endlich vielen Sch

Beweis:

(a) Falls b|a terminiert der Algorithmus in einem Schri5. (b) Andernfalls wird ein Parameter der Algorithmu mindestens den Wert 1 verringert und der Algorithmus rekursiv aufgerufen. Spätestens wenn ein Parameter den Wert 1 erreicht tri5 Fall (a) ein und der Algorithmus terminiert. Für endliche Eingaben bedeutet dies eine endliche Laufzeit. Was ist mit anderen Eingaben?

2. Theorem:

Der Algorithmus Euklid löst das Problem ggT.

Beweis:

Wir haben bereits festgestellt, dass für zwei positive natürliche Zahlen a, b gilt, dass ggT(a,b)=b (falls b|a) und ggT(a,b)=ggT(a%b) (falls b|a nicht gilt). Der Algorithmus Euklid vollzieht genau diese Fallunterscheidung nach.

3. Theorem: Für positive natürliche Zahlen a und b mit a>b, benötigt der Algorithmus Euklid maximal max{a,b} viele rekursive Aufrufe.

Beweis:

Wir haben bereits festgestellt, dass Euklid stets terminiert, dass bei jedem Aufruf ein Parameter um mind verringert wird und dass wenn der zweite (stets kleinere) Parameter den Wert 1 hat die Rekursion spätestens endet. Damit kann es maximal max{a,b} viele rekursive Aufrufe geben.

Anmerkung:

Die obige Laufzeit ist nur eine grobe obere Abschätzung. Die tatsächliche Worst-case-Laufzeit ist O(log(ab
(mehr zur O-Notation später)

3.6 Fazit

Welche Strategie (optimistisch, pessimistisch) und welches Verhalten man bei nicht-erfüllten Vorbedingungen zeigt, hängt von vielen Faktoren ab:

– Bei unkritischen oft aufzurufenden Algorithmen könnte die Überprüfung der Zulässigkeit zu viel Aufwand sein

– Bei zeitintensiven Algorithmen kann durch eine Überprüfung Zeit gespart werden

Man sollte das Verhalten seines Algorithmus im Fehlerfall aber stets gut dokumentieren!

3.7 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 8 zu finden.

2 `http://www.dpunkt.de/buecher/3358.html`

4 Berechenbarkeitsbegriff

Ein Problem (z.B. eine mathematische Funktion) heißt berechenbar[1], falls dafür ein Algorithmus existiert.

- Berechenbar: Algorithmus stoppt nach endlich vielen Schritten
- Funktion f: W $\rightarrow$ V ist
  - partiell: Def(f) $\subseteq$W (Beispiel: f: $\mathbb{R}\rightarrow \mathbb{R}$, f(x)= 1/x)
  - total: Def(f) = W

Ausgangssituation: Wir entwerfen und programmieren einen Algorithmus und wir haben eine Vorstellung was berechenbar ist (intuitiver Berechenbarkeitsbegriff). Das Problem ist nun, wie man diese Berechenbarkeit nachweisen kann. Dazu bringen wir die intuitive Form in eine mathematische Form und können diese mit mathematischen Beweisen belegen.

Formale Definitionen des Berechenbarkeitsbegriff: Turing berechenbare Funktionen, while-Programme, µ-rekursive Funktionen

4.1 Church-Turing-These

Die Klasse der Turing-berechenbaren Funktionen[2] ist genau die Klasse der intuitiv berechenbaren Funktionen. Wobei "intuitiv" nicht exakt formalisierbar ist.

Die durchführbaren Algorithmen sind eine Teilmenge der berechenbaren Funktionen, welche wiederum eine Teilmenge aller existierenden Funktionen sind.

1 https://de.wikipedia.org/wiki/Berechen

2 https://de.wikipedia.org/wiki/Church-Turing-These

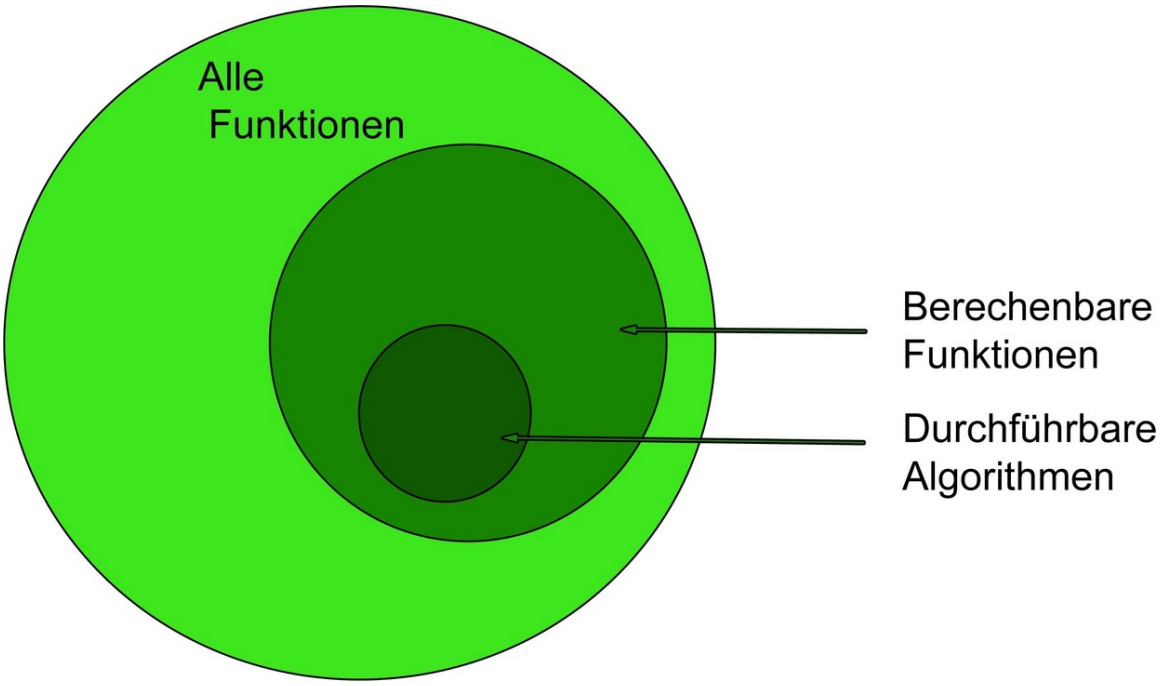

Abb. 2 Funktionen

4.2 Beispiele

Durchführbare Algorithmen

- ggT, Matrizenmultiplikation, Zinsberechnung,...

Berechenbare Funktionen, die nicht durchführbar sind

- Harte Optimierungsprobleme mit Millionen von Variablen
- Vollständige Eigenwertberechnung auf dem Facebook-Graphen

Nicht berechenbare Funktionen

- Haltefunktion (gibt zu einem beliebigen Programm an, ob es hält)
- Äquivalenzfunktion (gibt zu zwei beliebigen Programmen an, ob sie das gleiche Verhalten haben)
- ...

4.3 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[3] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um

3 `http://www.dpunkt.de/buecher/3358.html`

das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 6.4 und 7.1 zu finden.

5 Überblick Theoretische Grundlagen

In diesem Kapitel geben wir einen Überblick über die Theoretischen Grundlagen.

Ein Algorithmenentwurf ist eine kreative Disziplin. Es gibt keine allgemeingültige Anleitung zum Entwerfen und Analysieren von Algorithmen. Wir werden uns in dieser Vorlesung mit vielen Beispielen beschäftigen, die als Inspiration und Werkzeug dienen, weitere Algorithmen zu entwerfen. Einige theoretische Grundlagen sind allerdings notwendig. In diesem Kapitel beschäftigen wir uns näher mit

1. Programmierparadigmen

Was für Möglichkeiten gibt es Algorithmen zu entwickeln und zu implementieren?

2. Laufzeitanalysen

Wie kann man die Laufzeit eines Algorithmus analytisch ableiten und einordnen?

3.Entwurfsmuster

Was sind generelle Prinzipien für das Design eines Algorithmus?

5.1 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3 zu finden.

1 `http://www.dpunkt.de/buecher/3358.html`

6 Paradigmenbegriff

In diesem Kapitel erläutern wir den Paradigmenbegriff[1].

6.1 Definition

„Unter einem Paradigma versteht man unter anderem in der Wissenschaftstheorie ein **‘Denkmuster, welches das wissenschaftliche Weltbild eine Zeit prägt´** - ein Algorithmenparadigma sollte daher ein Denkmuster darstellen, das die Formulierung und den Entwurf von Algorithmen und damit letztendlich von Programmiersprachen grundlegend prägt.“

Oder etwas kürzer: Ein Muster für den Entwurf und die Formulierung von Algorithmen.

6.2 Paradigmen zur Algorithmenkonstruktion

Funktional: Verallgemeinerung der Funktionsauswertung. Rekursion spielt eine wesentliche Rolle.

- f(x) := 2 g(x) + h(x)
- h(x) := 1 + h(x-1)

Logisch: basierend auf logischen Aussagen und Schlussfolgerungen

- „wenn a verwandt mit b und b verwandt mit c, dann ist a verwandt mit c“

Imperativ: basierend auf einem einfachen Maschinenmodell mit gespeicherten und änderbaren Werten. Primär werden Schleifen und Alternativen als Kontrollbausteine eingesetzt.

- „erst: erhöhe a, dann multipliziere b mit c, dann subtrahiere a mit c,....“

Objektorientiert: basierend auf Nachrichtenaustausch zwischen Objekten und Vererbung von Klassen

Beispiel Java: objektorientiert, imperativ, Elemente von funktional

6.3 Paradigmen und Programmiersprachen

Funktional

1 `https://de.wikipedia.org/wiki/Paradigma`

- Haskell, ML, Lisp (Datenauswertung,Datenbank)

Logisch

- Prolog (Datenbank)

Imperativ

- C, Pascal (maschinenorientiert)

Objektorientiert

- Smalltalk, Eiffel („Simulation“ verteilter Systeme)

Mischungen

- C++, C#, Java

6.4 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3 zu finden.

2 `http://www.dpunkt.de/buecher/3358.html`

7 Funktionale Algorithmen

In diesem Kapitel wird die funktionale Programmierung[1] behandelt.

7.1 Grundidee

Definition zusammengesetzter Funktionen durch Terme mit Unbestimmten.

Ein Beispiel einer einfachen Funktionsdefinition ist f(x) = 5x + 1

```
Erinnerung Definition Term :
*Variable ist ein Term
*Konstanten-Symbol ist ein Term
*Sind t_1,...,t_n Terme und f ein n-stelliges Funktionssymbol, so ist f(t_1,...,t_n) ein Term
```

7.1.1 Beispiele für Terme

Unbestimmte (Symbole)

- x,y,z ... vom Typ int
- q,p,r ... vom Typ bool

Terme mit Bestimmten

- 1+1, 3*2, ...

Terme mit Unbestimmten

- Terme vom Typ int
  - x, x-2, 2x+1, (x+1)(y-1)

Terme vom Typ bool

- $p, p \wedge true, (p \vee true) \Rightarrow (q \vee false)$

7.2 Definition

Ein funktionaler Algorithmus ist eine Menge von Funktionsdefinitionen f_1 *bis* f_m mit:

1 https://de.wikipedia.org/wiki/Funktionale_Programmierung

$$
\begin{array}{l}
f_1(v_{1,1},...,v_{1,n_1}) := t_1(v_{1,1},...,v_{1,n_1}), \\
\ldots \\
f_m(v_{m,1},...,v_{m,n_m}) := t_m(v_{m,1},...,v_{m,n_m}).
\end{array}
$$

Die erste Funktion f_1 wird wie beschrieben ausgewertet und ist die Bedeutung (=Semantik) des Algorithmus.

f_1 ist die Zustands-bestimmende Eingabe aus der die Werte der Ausgabe abgelesen werden.

Funktionale Algorithmen sind die Grundlage einer Reihe von universellen Programmiersprachen, z.B. APL und Lisp. Diese Programmiersprachen werden als funktionale Programmiersprachen bezeichnet.

7.3 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.2 zu finden.

2 `http://www.dpunkt.de/buecher/3358.html`

8 Funktionsdefinition und Signatur

In diesem Kapitel wird die Funktionsdefinition und Signatur[1] von funktionalen Algorithmen behandelt.

8.1 Funktionsdefinition

Eine Funktion f ist eine Relation zwischen einer Eingabemenge X und einer Ausgabemenge $Y (f \subseteq X * Y)$ mit der Eigenschaft:

Für alle x ∈X, y,y‘ ∈Y mit (x,y),(x,y‘) ∈ f gilt y=y‘

Wir schreiben dann üblicherweise f(x)=y anstatt(x,y)∈f und deklarieren eine Funktion durch $f : X \rightarrow Y$. Ist $f : X \rightarrow Y$ eine Funktion so heißt X Eingabemenge und Y Ausgabemenge. In der funktionalen Programmierung sind Ein- und Ausgabemengen üblicherweise Terme eines bestimmten Typs.

8.2 Termdefinition

Sei T ein Typ, V_T eine Menge von Variablen vom Typ T und C_T eine Menge von Konstanten vom Typ T. Dann ist jedes $X \in V_T$ ist ein Term vom Typ T, jedes $a \in C_T$ ein Term vom Typ T und ist $f : T^k \rightarrow T$ eine Funktion und $t_1, ..., t_k$ sind Terme vom Typ T, so ist $f(t_1, ..., t_k)$ ein Term vom Typ T.

8.2.1 Beispiel Terme natürlicher Zahlen

Sei int der Typ der natürlichen Zahlen, V_{int} eine Menge von Variablen vom Typ T_{int} und $C_{int} = \mathbb{N} = 1, 2, 3....$ Mögliche Funktionen auf natürliche Zahlen sind

- $+ : int$ x $int \rightarrow int$
- $* : int$ x $int \rightarrow int$

3+4, (8+9)*10, X*4+1 sind dann Terme natürlicher Zahlen.

8.2.2 Beispiel Bool´sche Terme

Sei bool der Typ der Bool´sche Terme, V_{bool} eine Menge von Variablen vom Typ T_{bool} und $C_{bool} = true, false$. Mögliche Funktionen auf Bool´sche Termes sind

1 https://de.wikipedia.org/wiki/Signatur_%28Programmierung%29

- $\bigwedge : bool$ x $bool \rightarrow bool$
- $\neg : bool \rightarrow bool$

true $\bigwedge$ und $\neg Y \bigwedge X$ sind dann Bool´sche Terme.

Sind $v_1, ..., v_n$ Unbestimmte vom Typ $T_1, ..., T_n$ (bool oder int) und ist $t(v_1, ..., v_n)$ ein Term, so heißt $f(v_1, ..., v_n) := t(v_1, ..., v_n)$ eine Funktionsdefinition vom Typ T.

- T ist dabei der Typ des Terms (
  UNKNOWN TEMPLATE mathl
  v_1
  UNKNOWN TEMPLATE kommadots
  v_n
).
- f: ist der Funktionsname
- $v_1, ..., v_n$ ist ein formaler Parameter
- $t(v_1, ..., v_n)$: ist ein Funktionsausdruck

8.2.3 Beispiel

- $f(p,q,x,y) := if\ (p \vee q)\ then\ 2x+1\ else\ 3y-1$
- $g(x) :=\ if\ even(x)\ then\ x/2\ else\ 3x-1$
- $h(p,q) :=\ if\ p\ then\ q\ else\ false$

Jede Funktionsdefinition hat das Schema Funktionsname(formale Parameter):= Funktionsausdruck

8.3 Signatur einer Funktion

Eine Funktion f hat die folgende Funktionsdefinition:

$f(v_1, ..., v_n) := t(v_1, ..., v_n)$

mit $v_1, ..., v_n$ sind vom Typ $T_1, ..., T_n$

$t(v_1, ..., v_n)$ ist vom Typ T

Die Signatur von f ist: $f : T_1 * ... * T_n \rightarrow T$ mit der Struktur

Name mit Stelligkeit: Parameter mit Typ * ... * Parameter mit Typ $\rightarrow$ Typ des Rückgabewertes

8.3.1 Beispiel einer Funktionsdefinition

- f(p,q,x,y) := if (p ∨q) then 2x + 1 else 3y -1
- g(x) := if even(x) then x / 2 else 3x -1
- h(p,q) := if p then q else false, mit h als Funktionsname, (p,q) als formalen Parameter und dem daruaffolgenden Funktionsausdruck.

8.4 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.2.2 zu finden.

2 http://www.dpunkt.de/buecher/3358.html

9 Auswertung von Funktionen

In diesem Kapitel wird die Auswertung[1] funktionaler Algorithmen behandelt.

- Definierte Funktionen können mit konkreten Werten aufgerufen werden.
- Wir wissen, dass eine definierte Funktion folgende Struktur hat $f : T_1 * ... * T_n \to T$
- Sind nun $a_1, ..., a_n$ konkrete Werte vom Typ $T_1, ..., T_n$, so ersetzt man in $f(a_1, ..., a_n)$ jedes Vorkommen der Unbestimmten v_1 mit $a_1 (i = 1, ..., n)$. Somit kann der entstehende Term ausgewertet werden.
- Dabei heißen die konkreten Werte $a_1, ..., a_n$ aktuelle Parameter.
- Ausdruck $f(a_1, ..., a_n)$ heißt Funktionsaufruf.

9.1 Beispiel

- $f(p,q,x,y) := \ if \ (p \vee q) \ then \ 2x+1 \ else \ 3y-1$
  - $Signatur: \ f: \ bool \ x \ bool \ x \ int \ x \ int \ \to \ int$
  - $Aufruf: \ f(true, true, 3, 4) \ wird \ zu \ 7$
- $g(x) := \ if \ even(x) \ then \ x/2 \ else \ 3x-1$
  - $Signatur: \ g: \ int \ \to \ int$
  - $Aufruf: \ g(2) \ wird \ zu \ 1, \ g(9) \ wird \ zu \ 26 \ ausgewertet$
- $h(p,q) := \ if \ p \ then \ q \ else \ false$
  - $Signatur: \ h: bool \ x \ bool \ \to \ bool$
  - $Aufruf: \ h(false, false) \ wird \ ausgewertet \ zu \ false$

9.2 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.2.3 zu finden.

1 https://de.wikipedia.org/wiki/Auswertung_%28Informatik%29
2 http://www.dpunkt.de/buecher/3358.html

10 Auswertung von Funktionen

In diesem Kapitel wird die Auswertung[1] funktionaler Algorithmen behandelt.

- Definierte Funktionen können mit konkreten Werten aufgerufen werden.
- Wir wissen, dass eine definierte Funktion folgende Struktur hat $f : T_1 * ... * T_n \to T$
- Sind nun $a_1, ..., a_n$ konkrete Werte vom Typ $T_1, ..., T_n$, so ersetzt man in $f(a_1, ..., a_n)$ jedes Vorkommen der Unbestimmten v_1 mit $a_1 (i = 1, ..., n)$. Somit kann der entstehende Term ausgewertet werden.
- Dabei heißen die konkreten Werte $a_1, ..., a_n$ aktuelle Parameter.
- Ausdruck $f(a_1, ..., a_n)$ heißt Funktionsaufruf.

10.1 Beispiel

- $f(p,q,x,y) := \ if \ (p \vee q) \ then \ 2x+1 \ else \ 3y-1$
  - $Signatur: \ f: \ bool \ x \ bool \ x \ int \ x \ int \ \to \ int$
  - $Aufruf: \ f(true, true, 3, 4) \ wird \ zu \ 7$
- $g(x) := \ if \ even(x) \ then \ x/2 \ else \ 3x-1$
  - $Signatur: \ g: \ int \ \to \ int$
  - $Aufruf: \ g(2) \ wird \ zu \ 1, \ g(9) \ wird \ zu \ 26 \ ausgewertet$
- $h(p,q) := \ if \ p \ then \ q \ else \ false$
  - $Signatur: \ h: bool \ x \ bool \ \to \ bool$
  - $Aufruf: \ h(false, false) \ wird \ ausgewertet \ zu \ false$

10.2 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.2.3 zu finden.

1 https://de.wikipedia.org/wiki/Auswertung_%28Informatik%29
2 http://www.dpunkt.de/buecher/3358.html

11 Auswertung rekursiver Funktionen

In diesem Kapitel wird die Auswertung[1] rekursiver[2] Funktionen behandelt.

11.1 Erweiterung der Funktionsdefinition

- Erweiterung der Definition von Termen
- Neu: Aufrufe definierter Funktionen sind Terme
- Eine Funktionsdefinition f heißt rekursiv, wenn direkt oder indirekt (über andere Funktionen) ein Funktionsaufruf f(...)in ihrer Definition auftritt.

Gegeben ist die folgende Funktion:

$f(x,y) := \ if\ g(x,y)\ then\ h(x+y)\ else\ h(x-y)$

$g(x,y) \ := \ (x == y) \ \vee \ odd(y)$

$h(x) \ := \ j(x+1) * j(x-1)$

$j(x) \ := \ 2x-3$

Die Auswertung dieser Funktion lautet:

$f(1,2) \ \rightarrow \ if\ g(1,2)\ then\ h(1+2)\ else\ h(1-2)$

$$\rightarrow \ if\ 1 == 2 \ \vee \ odd(2)\ then\ h(1+2)\ else\ h(1-2)$$

$$\rightarrow \ if\ 1 == 2 \ \vee \ false\ then\ h(1+2)\ else\ h(1-2)$$

$$\rightarrow \ if\ false \ \vee \ false\ then\ h(1+2)\ else\ h(1-2)$$

$$\rightarrow \ if\ false\ then\ h(1+2)\ else\ h(1-2)$$

$$\rightarrow \ h(1-2)$$

$$\rightarrow \ h(-1)$$

1 https://de.wikipedia.org/wiki/Auswertung_%28Informatik%29
2 https://de.wikipedia.org/wiki/Rekursion

$$\rightarrow\ j(-1+1)*j(-1-1)$$

$$\rightarrow\ j(0)*j(-1-1)$$

$$\rightarrow\ j(0)*j(-2)$$

$$\rightarrow\ (2*0-3)*j(-2)$$

$$\rightarrow\ (-3)*(-7)$$

$$\rightarrow\ 21$$

11.2 Auswertung rekursive Funktionsdefinition

Gegeben ist folgende rekursive Funktion:

$f(x,y) := if\ x=0\ then\ y\ else($

$$if\ x>0\ then\ f(x-1,y)+1$$

$$else\ -f(-x,-y))$$

Die Auswertung dieser Funktion lautet:

$f(0,y) \rightarrow y\ fuer\ alle\ y$ Hier greift die erste Zeile der Funktionsdefinition. Da x=0 ist nehmen wir y

$f(1,y) \rightarrow f(0,y)+1 \rightarrow y+1$ Hier greift die zweite Zeile der Funktionsdefinition. Da x>0 ist haben wir f(1-1,y)+1. Da x nach diesem Schritt null ist, greift nun die erste Zeile und wir erhalten y+1.

$f(2,y) \rightarrow f(1,y)+1 \rightarrow (y+1)+1 \rightarrow y+2$ Hier greift ebenfalls die zweite Zeile der Funktionsdefinition. Da x>0 ist haben wir f(2-1,y)+1. Anschließend wenden wir noch einmal die zweite Zeile an, da x immer noch größer ist als null und wir erhalten f(1-1,y+1)+1. Da x nun null ist greift die erste Zeile der Funktionsdefinition und wir erhalten y+2.

...

Hier lässt sich bereits abschätzen, dass das Ergebnis der Funktion immer weiter hochgezählt wir und es lässt sich allgemein sagen:

$f(n,y) \rightarrow y+n\ fuer\ alle\ n \in int,\ n>0$

Ist unser x negativ, entwickelt sich die Auswertung wie folgt:

$f(-1,y) \to -f(1,-y) \to -(-y+1) \to y-1$ Hier greift die dritte Zeile der Funktionsdefinition. Da x<0 ist werden die Vorzeichen umgekehrt. Nun, da x=1 ist, greift die zweite Zeile und wir erhalten -f(1-1,-y)+1. Da x nun null ist greift wieder die erste Zeile und wir erhalten y-1.

$f(-2,y) \to -f(2,-y) \to -f(1,-y)+1 \to -(-y+2) \to y-2$

...

Auch hier lässt sich bereits abschätzen, wie sich die Funktion einwickelt und es lässt sich allgemein sagen:

$f(x,y) \to x+y \; fuer \; alle \; x,y \in int$

11.3 Definiertheit

Gegeben ist folgender Algorithmus:

$$f(x) := if \; x == 0 \; then \; 0 \; else \; f(x-1)$$

Auf welchen Eingaben ist der Algorithmus definiert?

Auswertung:

$$f(0) \to 0$$

$$f(1) \to f(0) \to 0$$

$$f(2) \to f(1) \to f(0) \to 0$$

$$f(x) \to 0 \forall x \in int, x > 0$$

$f(-1) \to f(-2) \to ...$ Diese Auswertung terminiert nicht!

Somit gilt:

$$f(x) := \begin{cases} 0 & falls \; x \geq 0 \\ \bot & sonst \end{cases}$$

11.4 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[3] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.2.4 zu finden.

3 `http://www.dpunkt.de/buecher/3358.html`

12 Definiertheit der Fakultätsfunktion

Im folgenden Beispiel wird die Definiertheit anhand des Beispiels der Fakultät[1] gezeigt.

$x! := x * (x-1) * (x-2) ... 2 * 1 \ fuer \ x > 0$

Es ist bekannt, dass $0! := 1$ und

$$x! := x * (x-1)!$$

Für negative Werte sind Fakultäten nicht definiert.

1.Lösung

$$fac(x) := if \ (x == 0) \ then \ 1 \ else \ x * fac(x-1)$$

Das bedeutet: $fac(x) := \begin{cases} x! & falls \ x \geq 0 \\ \bot & sonst \end{cases}$

2.Lösung

$$fac(x) := if \ (x \leq 0) \ then \ 1 \ else \ x * fac(x-1)$$

Das bedeutet:$fac(x) := \begin{cases} x! & falls \ x \geq 0 \\ 1 & sonst \end{cases}$

12.1 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.2.6 zu finden.

1 https://de.wikipedia.org/wiki/Fakult%C3%A4t_%28Mathematik%29
2 http://www.dpunkt.de/buecher/3358.html

13 Größter gemeinsamer Teiler - funktional

In folgendem Beispiel werden wir den größten gemeinsamen Teiler[1] mit Hilfe eines funktionalen Algorithmus berechnen.

13.1 Hintergrundwissen

$Fuer\ x,y > 0\ gilt:$
$ggT(x,y) := x$
$ggT(x,y) := ggT(y,x)$
$ggT(x,y) := ggT(x,y-x)\ \ fuer\ x \leq y$

Wir haben folgende funktionale Spezifikationen:

$$ggt(x,y) := \begin{cases} if(x \leq 0) \vee (y \leq 0) & then\ ggT(x,y)\ else \\ if\ x == y & then\ x\ else \\ if\ x > y & then\ ggT(y,x)\ else \\ & ggT(x,y-x) \end{cases}$$

13.2 Auswertung

Eine beispielhafte Auswertung sieht wie folgt aus:

$ggT(39,15) \rightarrow ggT(15,39) \rightarrow (15,24)$

$$\rightarrow ggT(15,9) \quad \rightarrow (9,15)$$

$$\rightarrow ggT(9,6) \quad \rightarrow (6,9)$$

$$\rightarrow ggT(6,3) \quad \rightarrow (3,3)$$

$$\rightarrow 3$$

1 https://de.wikipedia.org/wiki/Gr%C3%B6%C3%9Fter_gemeinsamer_Teiler

13.3 Abbruchbedingungen und Rekursion

Der ggT lässt sich nur korrekt berechnen, wenn positive Eingaben gemacht werden. Bei negativen Eingaben ist der ggT undefiniert und der Algorithmus terminiert nicht.

- Abbruchbedingungen:

$$x \leq 0$$

$$y \leq 0$$

$$x == y$$

Im Fall des Abbruchs wird eine Evaluierung oder Ausnahme angegeben.

- Bedingungen für rekursive Verwendung der Funktion, "einfachste" Rekursion zuerst

  1. $x, y > 0,\ x \geq y$
  2. $x, y > 0,\ y < x$

Im Fall der Rekursion wird eine Evaluierung angegeben.

13.4 Programm

```
public static int ggT(int x, int y)
{
     if  ((x <= 0) || (y <= 0))
         throw new ArithmeticError(‘‘negative Daten bei ggt‘‘));
     else   if  (x==y)  then return x;
               else
                         if x > y  then return ggT(y,x);
                         else return ggT(x,y-x);
}
```

13.5 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.2.6 zu finden.

2 http://www.dpunkt.de/buecher/3358.html

14 Fibonacci Zahlen - funktional

In folgendem Beispiel werden wir die Fibonacci-Zahlen[1] mit Hilfe eines funktionalen Algorithmus berechnen.

14.1 Hintergrundwissen

Bei den Fibonacci Zahlen handelt es sich um eine unendliche Zahlenreihe. Ursprünglich wurde die Fibonacci-Folge zur Beschreibung des Wachstums einer Kaninchenpopulation verwendet. Diese erfolgt progressiv. Am Anfang gibt es ein Kaninchenpaar, dieses wird im zweiten Monat zeugungsfähig und zeugt jeden Monat ein weiteres Paar Kaninchen. Keins der Kaninchen stirbt. Das heißt die die Summe der benachbarten Zahlen ergibt die nächste Zahl (0,1,1,2,3,5,8,...).

$f_0 = 0$

$f_1 = 1$

$f_2 = 1 = f_0 + f_1$

$f_3 = 2 = f_1 + f_2$

$f_4 = 3 = f_2 + f_3$

...

$$fib(x) := \begin{cases} x-te\ Fibonacci-Zahl & fals\ x \geq 0 \\ \bot & sonst \end{cases}$$

14.2 Programm

```
fib(x)  :=  if (x==0) then 0
        else if (x==1)  then  1
                else  fib(x-1) + fib(x-2)
```

14.3 Literatur

Da die Vorlesungsinhalte auf dem Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das

1 https://de.wikipedia.org/wiki/Fibonacci-Folge
2 http://www.dpunkt.de/buecher/3358.html

hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.2.6 zu finden.

15 Prädikatenlogik und Hornlogik

In diesem Kapitel werden die Grundlagen der Prädikatenlogik[1] und Hornlogik erläutert.

15.1 Grundlagen

Sei U eine Menge von Konstanten, V eine Menge von Variablen, und P eine Menge von Prädikatsymbolen.

- Ein Term ist entweder eine Konstante oder eine Variable (prinzipiell sind auch Funktionsterme möglich, werden hier aber ignoriert)
- X,Y,... sind Variablen (und Terme)
- anna, bob, dave,... sind Konstanten (und Terme)
- Ein Atom ist ein n-stelliges Prädikat, gefolgt von n Termen
- parent(bob,anna) ist ein Atom
- sibling(anna, X) ist ein Atom
- Eine atomare Konjunktion ist eine Menge von Atomen
- parent(X,anna) ∧sibling(anna,Y) ∧parent (anna,tina)
- Bei logischer Programmierung wird oft das Komma für die Konjunktion verwendet: parent (X,anna), sibling(anna,Y), parent(anna,tina)
- Eine Hornklausel ist eine Implikation einer atomaren Konjunktion zu einem Atom
- grandparent(X,Y) ⇐parent(X,Z) ∧parent(Z,Y)
- In Prolog: grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
- Anmerkung: Ein Hornklausel ist eigentlich definiert als eine Disjunktion mit maximal einem positiven Atom

15.2 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.4.1 zu finden.

1 https://de.wikipedia.org/wiki/Pr%C3%A4dikatenlogik

2 http://www.dpunkt.de/buecher/3358.html

16 Prolog

Ein Prolog-Programm[1] ist eine Menge von Hornklauseln und Fakten (=Atome ohne Variablen)

16.1 Beispiel 1

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).

brother(X,Y) :- male(X), male(Y), parent(Z,X), parent(Z,Y).

hasUncle(X) :- parent(Y,X), brother(Y,_).

parent(bob, anna).

parent(carl, bob).

male(bob).
```

Anmerkung: „_“ ist eine beliebige (unbenannte) Variable

16.2 Beispiel 2

```
s(X,Y) :- r(X,Y), t(Y).

r(a,b). r(a,e). r(c,d).

t(b). t(d).
```

16.3 Anfragen

Prolog ist eine Anfrage-basierte Programmiersprache. Das bedeutet jede Ausführung eines Prolog-Programms muss mit einer Anfrage parametrisiert werden.

Die Anfragen zu oben gezeigten Prolog Programm aus **Beispiel 1** lauten:

```
?grandparent(carl,anna) → Antwort YES
?male(anna) → Antwort NO (Closed World Assumption)
```

1 https://de.wikipedia.org/wiki/Prolog_%28Programmiersprache%29

Anfragen können aber auch Variablen enthalten, so wie in **Beispiel 2** .

```
?s(a,X) → Antwort X=b
?r(a,X) → Antwort X=b, X=e
```

Die Semantik logischer Programme leitet sich direkt von der klassisch logischen Semantik der Prädikatenlogik ab (siehe Logik-Vorlesung).

Techniken:

- Grundierung des Programms (ersetze Variablen durch alle Kombinationen von Konstanten) und aussagenlogische Verarbeitung
- Unifikation des Anfrageterms und Backtracking

16.3.1 Beispiel

Problem: Wegfindung in gerichteten Graphen

- Gegeben ein Graph mit Knoten $a_1, ..., a_n$
- Gibt es einen Weg zwischen Knoten a_i *und* a_j (für beliebige i,j)?

Lösung als Prolog-Programm:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(Z,Y), edge (X,Z).
edge(a1,a2). edge(a2,a3). edge(a2,a4). edge(a5,a1).
```

Anfragen:

```
?path(a1,a3) → Antwort YES
?path(a5,X) → Antwort X=a1, X=a2, X=a3, X=a4 (alle von a5 erreichbare Knoten)
```

16.4 Logische vs. Funktionale Programmierung

Hornklauseln sind Funktionen im Sinne von atomaren Operationen. Sie haben gemeinsam, dass sie die Rekursion als zentrales Paradigma haben und eine mathematische Basis. Zu den Unterschieden zählt, dass sie Atome entweder wahr oder falsch sind und dass die Funktionswerte beliebige Typen haben können.

16.5 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um

2 http://www.dpunkt.de/buecher/3358.html

das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.4.1 zu finden.

17 Liste

Eine Liste[1] ist entweder die leere Liste oder ein Term gefolgt von einer Liste.

17.1 Definition in Prolog

```
list([]).
list([X|Y]) :- list(Y).
```

Der |-Operator trennt den Kopf (Head=erstes Element)einer Liste vom Rumpf (Tail=Restliste) ab

17.1.1 Beispiele

- Liste von Zahlen:

```
[1|[2|[3]]] = [1,2,3]
```

- Liste von beliebigen Termen:

```
[male(bob), female(anna),male(carl)]
```

17.2 Listenmanipulation

- Aneinanderreihung:

append(X,Y,Z): X ist die Liste, die entsteht, wenn Z an Y angehängt wird

```
append(X,X,[]).
append([Y|X], [Y|Z],L) :- append(X,Z,L).
```

- Invertierung:

invert(X,Y): X ist die Invertierung von Y

```
invert([],[]).
invert([X|Y],L) :- invert(Y,Z), append(L,Z,[X]).
```

1 https://de.wikipedia.org/wiki/Liste

17.3 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.4.1 zu finden.

2 `http://www.dpunkt.de/buecher/3358.html`

18 Imperative Algorithmen

Die imperative Vorgehensweise[1] ist eine verbreitete Art, um Algorithmen für Computer zu formulieren. Sie basiert auf den Konzepten Anweisung und Variablen und wird durch Programmiersprachen wie Java, C, PASCAL, FORTRAN, COBOL, Maschinencode, ... realisiert. Das Prinzip ist ein abstraktes Rechnermodell. Werte werden gespeichert und anschließend schrittweise bearbeitet. Imperative Algorithmen sind nicht so elegant, verständlich und wartbar wie funktionale, objektorientierte oder logische Algorithmen.

18.1 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.3 zu finden.

1 https://de.wikipedia.org/wiki/Imperativer_Algorithmus
2 http://www.dpunkt.de/buecher/3358.html

19 Variablen

Eine Variable[1] besteht aus einem Namen (z.B. X), einem veränderlichen Wert und einem Typ. Bei Variablen handelt es sich um Speicherplätze für Werte. Ist t ein Term ohne Variablen und w(t) sein Wert, dann heißt das Paar X:=t eine Wertzuweisung. Ihre Bedeutung ist festgelegt durch

```
Nach Ausführung von X:=t gilt  X=w(t)
Vor Ausführung der ersten Wertzuweisung gilt X=?(undefiniert)
```

19.1 Beispiel

$X := 7$

$X := (3 - 7) * 9)$

$F := true$

$Q := \neg(true \ \vee \ false) \vee \ \neg\neg \ true$

19.2 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.3.1 zu finden.

1 https://de.wikipedia.org/wiki/Variable_%28Programmierung%29
2 http://www.dpunkt.de/buecher/3358.html

20 Zustände

Ist $\underline{X} = X_1, X_2, ...$ eine Menge von Variablen (-namen) von denen alle nur Werte aus der Wertemenge W haben können (alle Variablen vom gleichen Typ), dann ist der Zustand Z eine partielle Abbildung.

$Z : \underline{X} \to W$ `(Zuordnung des momentanen Wertes)`

- Beispiel in einem gewissen Zustand

$$Z(X_1) = 42$$

$$Z(X_2) = 17$$

$$Z(X_3) = 23$$

- Nach $X_1 := 29$ folgt:

$$Z(X_1) = 29$$

$$Z(X_2) = 17$$

$$Z(X_3) = 23$$

Ist $Z : \underline{X} \to W$ ein Zustand und wählt man eine Variable X und einen Wert w aus dem Wertebereich W, so ist der transformierte Zustand wie folgt definiert:

$Z(X \leftarrow w) : \underline{X} \to W$ *mit*

$$Z_{(X \leftarrow W)}(Y) \to \begin{cases} w & falls\ X = Y \\ Z(Y) & sonst \end{cases}$$

20.1 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.3.1 zu finden.

1 `http://www.dpunkt.de/buecher/3358.html`

21 Anweisungen

In diesem Kapitel behandelt wir das Thema Anweisungen.

21.1 Arten von Anweisungen

Dabei unterscheiden wir in zwei verschiedene Anweisungsarten. Zum einen die elementaren Anweisungen wie Wertezuweisungen[1] und zum anderen die komplexen Anweisungen[2].

21.2 Semantik einer Anweisung

Funktion, die einen Zustand in einen neuen Zustand überführt. $[[\alpha]](z)$

Allgemein gesagt ist es die Wirkungsweise von α auf den Zustand Z

21.3 Beispiele Zuweisung als Anweisung

21.3.1 Beispiel 1

Ein Beispiel ist die Wertezuweisung:

$$\alpha_1 = (X := 2 \cdot Y + 1)$$

α_1 ist eine elementare Anweisung

Diese Wertezuweisung transformiert in eine Funktion auf Zustände sieht wie folgt aus:

$$[[\alpha_1]](Z) = Z\langle X \leftarrow 2 \cdot Z(Y) + 1\rangle$$

Die Zuweisung berechnet den neuen Zustand.

Der alte Zustand ist Z und der neue Zustand ist $[[\alpha_1]](Z)$

1 https://de.wikipedia.org/wiki/Wertzuweisung
2 https://de.wikipedia.org/wiki/Anweisung_%28Programmierung%29

21.3.2 Beispiel 2

Ein weiteres Beispiel ist die Zuweisung mit gleichen Variablen auf beiden Seiten.

$$\alpha_1 = \langle X := 2 \cdot X + 1 \rangle$$

Die Transformation in eine Funktion auf Zustände lautet:

$$[[\alpha_1]](Z) = Z\langle X \leftarrow 2 \cdot Z(X) + 1 \rangle$$

Bei der letzten Anweisung handelt es sich nicht um eine rekursive Gleichung! An dieser Stelle sei vermerkt, dass Wertezuweisungen die einzigen elementaren Anweisungen sind.

21.4 Komplexe Anweisungen

Bisher haben wir elementare Anweisungen (Wertzuweisungen) als Funktionen auf Zustände verstanden. Komplexe Anweisungen nehmen Konstrukte bzw. Bausteine von imperativen Algorithmen. Diese Bausteine sind

1. Sequenz
2. Auswahl/Selektion
3. Iteration

Die Semantik wird wiederum durch Konstruktion von Funktionen definiert. Iteration ist das Gegenstück zu rekursiven Funktionsaufrufen bei funktionalen Algorithmen

21.5 Sequenz

Sequenzen, oder auch Folgen, sind α_1 und α_2 Anweisungen, so ist $\alpha_1;\alpha_2;$ auch eine Anweisung. Die Zustandstransformation beschreibt die Semantik der Sequenz.

$$[[\alpha_1;\alpha_2]](Z) = [[\alpha_2]]([[\alpha_1]](Z))$$

Die Semantik ist das Schachteln der Funktionsaufrufe und das daraus folgende hintereinander ausführen der beiden Funktionen.

21.6 Selektion

Eine Selektion, bzw. eine Auswahl, liegt beispielsweise vor, wenn α_1 und α_2 Anweisungen sind und B ein boolescher Ausdruck ist, dann ist auch

$if\ B\ then\ \alpha_1\ else\ \alpha_2$

eine Anweisung.

Die zugehörige Zustandstransformation ist: $[[if\ B\ then\ \alpha_1\ else\ \alpha_2]](Z) = \begin{cases} [[\alpha_1]](Z) & falls\ Z(B) = true \\ [[\alpha_2]](Z) & falls\ Z(B) = false \end{cases}$

Voraussetzung ist, dass Z(B) definiert ist, sonst ist die Bedeutung der Auswahlanweisung undefiniert.

21.7 Iteration

Wiederholung (Iteration, Schleife):

```
Ist α eine Anweisung und B ein boolescher Ausdruck, so ist:
while B do α
auch eine Anweisung
```

Zustandstransformation: $[[while\ B\ do\ \alpha]](Z) = \begin{cases} Z & falls\ Z(B) = false \\ [[while\ B\ do\ \alpha]]([[\alpha]](Z)) & sonst \end{cases}$

Ist Z(B) undefiniert, so ist die Bedeutung dieser Anweisung ebenfalls undefiniert.

21.8 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[3] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.3.1 und 3.3.2 zu finden.

3 http://www.dpunkt.de/buecher/3358.html

22 Syntax und Semantik

In diesem Kapitel wird die Syntax[1] und Semantik[2] von imperativen Algorithmen behandelt.

22.1 Umsetzung in Programmiersprachen

In realen imperativen Programmiersprachen gibt es fast immer diese Anweisungen, da imperative Algorithmen die Grundbausteine imperativer Programmiersprachen sind. While-Schleifen sind rekursiv definiert, ihre rekursive Auswertung braucht nicht zu terminieren. Bereits Programmiersprachen mit diesen Sprachelementen sind universell. Wir werden uns hier zunächst auf die Datentypen bool und int beschränken und können nun die Syntax imperativer Algorithmen festlegen.

22.2 Syntax

<Programmname>:
var X,Y,...:int; P,Q,...:bool; (Variablen Deklaration)
input $X_1,...,X_n$; (Eingabe Variablen)
α (Anweisungen)
output $Y_1,...,Y_m$ (Ausgabe-Variablen)

22.3 Semantik

Die Festlegung der formalen Bedeutung ist hier etwas komplexer als bei den funktionalen Algorithmen. Das Ziel ist aber das gleiche: Die Funktion zur Semantikfestlegung.

Die Bedeutung (Semantik) eines imperativen Algorithmus ist eine partielle Funktion:

$[[PROG]]W_1 \cdot .... \cdot W_n \to V_1 \cdot V_m$

$[[PROG]](w_1,...,w_n) = (Z(Y_1),...,Z(Y_m))$

$$wobei\ Z = [[\alpha]](Z_0),$$

$$Z_o(X_i) = w_i,\ i = 1,..,n$$

$$und\ Z_0(Y) = \perp,\ fuer\ Variablen\ Y \neq X_i (i = 1,...,n)$$

1 https://de.wikipedia.org/wiki/Syntax
2 https://de.wikipedia.org/wiki/Semantik

Es gilt:

$PROG$ Programme
$W_1,...,W_n$ Wertebereich der Typen von $X_1,...,X_n$
$V_1,...,V_m$ Wertebereich der Typen von $Y_1,...,Y_m$

Das bedeutet, dass der Algorithmus eine Transformation auf den gesamten initialen Zustand (geg. durch die Eingabe)definiert. Die Bedeutung gibt die Werte der Ausgabevariablen an.

$[[PROG]](w_1,...,w_n) = Z(Y_1,...,Z(Y_m))$

$$wobei\ Z = [[\alpha]](Z_0),$$

$$Z_o(X_i) = w_i,\ i = 1,..,n$$

$$und\ Z_0(Y) = \perp,\ fuer\ Variablen\ Y \neq X_i (i = 1,...,n)$$

Die Funktion Z ist nicht definiert, falls die Auswertung von α nicht terminiert.

22.4 Charakterisierung

Die Algorithmenausführung imperativer Algorithmen besteht aus einer Folge von Basisschritten, oder genauer gesagt Wertzuweisungen. Diese Folge wird mittels Selektion und Iteration basierend auf booleschen Tests über dem Zustand konstruiert. Jeder Basisschritt definiert eine Transformation des Zustands. Die Semantik des Algorithmus ist durch die Kombination all dieser Zustandstransformationen zu einer Gesamttransformation festgelegt.

22.5 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[3] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.3.2 zu finden.

3 http://www.dpunkt.de/buecher/3358.html

23 Fakultätsfunktion als imperativer Algorithmus

Im Folgenden werden wir die Fakultätsfunktion[1] als imperativen Algorithmus[2] entwerfen.

23.1 Hintergrundwissen

Fakultätsfunktion: $0! = 1, x! = x \cdot (x-1)!$ für $x > 0$

$FAC\ var\ X, Y : int;$

$$input X;$$

$$Y := 1$$

$$while\ X > 1\ do\ Y := Y \cdot X;\ X := X - 1$$

$$output\ Y$$

Es ist:

$$[[FAC]](x) = \begin{cases} x! & \texttt{für } x \geq 0 \\ 1 & sonst \end{cases}$$

Falls die Bedingung der while-Schleife $x \neq 0$ lautet, dann ist:

$$[[FAC]](x) = \begin{cases} x! & \texttt{für } x \geq 0 \\ \bot & sonst \end{cases}$$

Gesucht ist das Ergebnis des Aufrufs FAC(3).

Die Abkürzung der while β für die Zeile ist

1 https://de.wikipedia.org/wiki/Fakult%C3%A4t_%28Mathematik%29
2 https://de.wikipedia.org/wiki/Imperative_Programmierung

$$while\ \ X > 1\ \ do\ \ Y := Y \cdot\ \ X; X := X - 1$$

Die Signatur der Semantikfunktion ist

$$[[FAC]] : int \to int$$

Die Funktion ist durch Lesen von Y im Endzustand Z definiert

$$[[FAC]](w) = Z(Y)$$

Der Endzustand ist definiert durch

$$Z = [[\alpha]](Z_0), \text{ wobei } \alpha \text{ die Folge aller Anweisungen des Algorithmus ist.}$$

Der initiale Zustand Z_0 ist definiert als

$$Z_0 = (X = w, Y = \perp)$$

Die Zustände abkürzend ohne Variablennamen sind

$$Z_0 = (w, \perp)$$

23.2 Die Auswertung

$Z = [[\alpha]](Z_0)$

$$= [[\alpha]](3, \perp)$$

$$= [[Y := 1; while\ \beta]](3, \perp)$$

$$= [[while\ \beta]]([[Y := 1]](3, \perp))$$

$$= [[while\ \beta]](3, \perp)Y \leftarrow 1$$

$$= [[while\ \beta]](3, 1)$$

$$= \begin{cases} Z & falls\ Z(B) = false \\ [[while\ B\ do\ \alpha']]([[\alpha']](Z)) & sonst \end{cases}$$

$$= \begin{cases} (3,1) & falls\ Z(X > 1) = (3 > 1) = false \\ [[while\ \beta]]\ ([[Y := Y \cdot X;\ x := X - 1]](Z)) & sonst \end{cases}$$

$$= [[while\ \beta]]([[Y := Y \cdot X;\ X := X - 1]](3,1))$$

$$= [[while\ \beta]]([[X := X - 1]]([[Y := Y \cdot X]](3,1)))$$

$$= [[while\ \beta]]([[X := X - 1]](3,3))$$

$$= [[while\ \beta]](2,3)$$

$$= \begin{cases} (2,3) & falls\ Z(X > 1) = (2 > 1) = false \\ [[(while\ \beta)]]\ ([[Y := Y \cdot X;\ X := X - 1]](Z)) & sonst \end{cases}$$

$$= [[while\ \beta]]([[Y := Y \cdot X;\ X := X - 1]](2,3))$$

$$= [[while\ \beta]]([[X := X - 1]]([[Y := Y \cdot X]](2,3)))$$

$$= [[while\ \beta]]([[X := X - 1]](2,6))$$

$$= [[while\ \beta]](1,6)$$

$$= \begin{cases} (1,6) & falls\ Z(X > 1) = (1 > 1) = false \\ [[(while\ \beta)]]\ ([[Y := Y \cdot X;\ X := X - 1]](Z)) & sonst \end{cases}$$

$$= (1,6)$$

23.3 Schlussfolgerung

Das bedeutet $Z = [[\alpha]](Z_0)$

$$= [[\alpha]](3, \perp)$$

$$= (1, 6)$$

Damit gilt

$[[FAC]](3) = Z(Y) = 6$

23.4 Beobachtungen

Der Übergang von der 3. auf die 4. Zeile folgt der Definition der Sequenz, indem der Sequenzoperator in einen geschachtelten Funktionsaufruf umgesetzt wird. Nur in der 5. Zeile wurde eine Wertzuweisung formal umgesetzt,später sind sie einfach verkürzt direkt ausgerechnet. In der 7. Zeile haben wir die Originaldefinition der Iteration eingesetzt (nur mit Kürzel α' statt α, da α bereits verwendet wurde). Dies entspricht im Beispiel α' = {Y:= Y · X; X:= X - 1}. Das Z in der 7. und 8. Zeile steht für den Zustand (3,1). (In späteren Zeilen analog für den jeweils aktuellen Zustand.)Bei diesem Beispiel sieht man folgendes sehr deutlich: Die Ausführung einer while-Schleife erfolgt analog zur rekursiven Funktionsdefinition!

23.5 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[3] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.3.3 zu finden.

3 `http://www.dpunkt.de/buecher/3358.html`

24 Fibonacci Zahlen: Funktional vs. Imperativ

In diesem Kapitel werden wir den funktionalen Algorithmus[1] der Fibonacci-Zahlen[2] mit dem imperativen Algorithmus[3] vergleichen.

Funktionale Umsetzung

```
fib(x)  :=  if (x==0) then 0
       else if (x==1)  then  1
                else  fib(x-1) + fib(x-2)
```

Imperative Umsetzung

```
FIB var X,A,B,C: int;
                  input X;
                  A := 0; B:=1; C:=1;
                  while  X > 0 {
                           C :=  A+B;
                           A := B;
                           B := C;
                           X := X-1;
                  }
                  output A;
```

Für beliebige X gibt die Auswertung das Ergebnis von FIB(X). Wir erkennen, der imperative Algorithmus FIB berechnet folgende Funktion:

$$[[FIB]](x) = \begin{cases} x-te\ Fib.\ Zahl & falls\ x \geq 0 \\ 0 & sonst \end{cases}$$

24.1 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[4] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.3.3 zu finden.

1 https://de.wikipedia.org/wiki/Funktionale_Programmierung
2 https://de.wikipedia.org/wiki/Fibonacci-Folge
3 https://de.wikipedia.org/wiki/Imperative_Programmierung
4 http://www.dpunkt.de/buecher/3358.html

25 ggT: Funktional vs. Imperativ

In diesem Kapitel werden wir den funktionalen Algorithmus[1] des größten gemeinsamen Teilers[2] mit dem imperativen Algorithmus[3] vergleichen.

25.1 Version 1

```
GGT1 var X,Y: int;
                input X,Y;
                while  X =/Y {
                       while  X > Y   { X :=  X-Y; }
                       while  X < Y   { Y :=  Y-X; }
                 }
                output X;
```

Die Auswertung für X=19 und Y=5 lautet:

X	**Y**
19	5
14	5
9	5
4	5
4	1
3	1
2	1
1	1

Die Berechnung erfolgt durch die Subtraktion der jeweils kleineren Zahl. Es ist zu Beobachten, dass der ggT mittels Subtraktion nicht effizient berechnet werden kann.

25.2 Version 2

```
GGT2 var X,Y,R: int;
                input X,Y;
                R := 1
                while  R =/0 {
                    R := X % Y;  X := Y;  Y := R;
                }
                output X;
```

1 https://de.wikipedia.org/wiki/Funktionale_Programmierung
2 https://de.wikipedia.org/wiki/Gr%C3%B6%C3%9Fter_gemeinsamer_Teiler
3 https://de.wikipedia.org/wiki/Imperative_Programmierung

Die Auswertung für X=19 und Y=5 lautet:

X	**Y**	**R**
19	5	1
5	4	4
4	1	1
1	0	0

Die Auswertung für X=2 und Y=1000 lautet:

X	**Y**	**R**
1000	2	1
2	0	0

Die Berechnung erfolgt hier durch die Modulo Funktion. Falls X<Y sein sollte, werden X und Y erst vertauscht, wie in der zweiten Auswertung.

Dieser Algorithmus ist folgendermaßen definiert:

$$[[GGT2]](x,y) = \begin{cases} ggT(x,y) & falls\ x,y > 0 \\ y & falls\ x = y\ \neq 0\ oder\ x = 0, y \neq 0 \\ \perp & falls\ y = 0 \\ ggT(|x|,|y|) & falls\ x < 0\ und\ y > 0 \\ -ggT(|x|,|y|) & falls\ y < 0 \end{cases}$$

25.3 Vergleich

Intuitiv ist GGT2 schneller als GGT1, was man durch die Komplexität von Algorithmen zeigen kann.

25.4 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[4] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 3.3.3 zu finden.

4 `http://www.dpunkt.de/buecher/3358.html`

26 Komplexität

Auf dieser Seite wird das Thema Komplexität[1] behandelt. Gegeben ist ein zu lösendes Problem. Es ist wünschenswert, dass der Algorithmus zur Berechnung der Lösung einen möglichst geringen Aufwand hat. Daher wird der Aufwand des Algorithmus (Komplexität) abgeschätzt . Zur Lösung von Problemen einer bestimmten Klasse gibt es einen Mindestaufwand.

26.1 Motivierendes Beispiel

Als Beispiel nutzen wir die sequentielle Suche in Folgen. Gegeben ist die Zahl b und n Zahlen, z.B. mit A[0...n-1] mit n>0, wobei die Zahlen verschieden sind. Gesucht ist ein Index $i \in \{0,...,n-1\}$ mit $b = A[i]$, falls der Index existiert, sonst ist i = n. Die Lösung für das Problem ist:

```
i = 0;
  while (i < n  &&  b != A[i])
    i++;
```

Der Aufwand der Suche hängt nun von der Eingabe ab, d.h vom gewählten Wert n, den Zahlen A[0],...,A[n] und von b. Es gibt zwei Möglichkeiten, eine erfolgreiche oder eine erfolglose Suche. Eine erfolgreiche Suche haben wir, wenn b=A[i] dann ist S=i+1 Schritte. Ist die Suche jedoch erfolglos, dann ist S=n+1 Schritte. Das Problem ist, dass die Aussage von zu vielen Parametern abhängt und unser Ziel ist eine globale Aussage zu finden, die nur von einer einfachen Größe abhängt, z.B. der Länge n der Folge.

26.2 Analyse erfolgreiche Suche

Im schlechtesten Fall wird b erst im letzten Schritt gefunden, d.h. b=A[n-1]. Dann wäre S=n. Im Mittel wird die Anwendung mit verschiedenen Eingaben wiederholt. Wenn man beobachtet wie oft b an erster, zweiter,..., letzter Stelle gefunden wird, hat man eine Annahme über die Häufigkeit. Läuft der Algorithmus k mal (k>1), so wird b gleich oft an erster, zweiter,....,letzter Stelle gefunden und somit k/n mal an jeder Stelle. Die Anzahl der Schritte insgesamt für k Suchvorgänge lässt sich folgendermaßen berechnen:

$M = \frac{k}{n} \cdot 1 + \frac{k}{n} \cdot 2 + ... + \frac{k}{n} \cdot n$

$$= \frac{k}{n} \cdot (1 + 2 + ... + n)$$

1 https://de.wikipedia.org/wiki/Komplexit%25C3%25A4t_%2528Informatik%2529%20

$$= \frac{k}{n} \cdot \frac{n \cdot (n+1)}{2}$$

$$= k \cdot \frac{n+1}{2}$$

Für eine Suche benötigt man $S = \frac{M}{k}$ Schritte Daraus folgt, dass im Mittel (bei einer Gleichverteilung) $S = \frac{n+1}{2}$

26.3 Asymptotische Analyse

Zur Analyse der Komplexität geben wir eine Funktion als Maß für den Aufwand an. $f : \mathbb{N} \to \mathbb{N}$. Das bedeutet f(n)=a bei Problemen der Größe n beträgt der Aufwand a. Die Problemgröße ist der Umfang der Eingabe, wie z.B. die Anzahl der zu sortierenden oder zu durchsuchenden Elemente. Der Aufwand ist die Rechenzeit(Abschätzung der Anzahl der Operationen, wie z.B. Vergleiche) und der Speicherplatz.

26.4 Aufwand für Schleifen

Wie oft wird die Wertezuweisung x=x+1 in folgenden Anweisungen ausgeführt?

```
x = x +1
```

1-mal

```
for (i = 1; i <= n; i++)
  x = x + 1;
```

n-mal

```
for (i = 1; i <= n; i++)
 for (j = 1; j <= n; j++)
       x = x + 1;
```

n^2-mal

26.5 Aufwandsfunktion

Die Aufwandsfunktion $f : \mathbb{N} \to \mathbb{N}$ ist meist nicht exakt bestimmbar. Daher wird der Aufwand im schlechtesten Fall und im mittleren Fall abgeschätzt und die Größenordnung ungefähr errechnet.

26.5.1 Vergleich Größenordnung

Funktion	**n=100**	**n=10.000**|| n=100.000	
log n	4,6	9,2	11,5
n^2	10.000	100.000.000	10.000.000.000
n^3	1.000.000	10^{12}	10^{15}

26.6 Problemstellung

Wie können wir das Wachstum von Funktionen abschätzen und wie verhalten sich die Funktionen zueinander? Das Ziel ist, die Funktion $t_i(n)$ zu wählen, die $f(n)$ nach oben beschränkt.

$f(n) = \frac{1}{3}\ n^2$

$t_1(n) = \frac{1}{4}\ n^2$

$t_2(n) = n$

$t_3(n) = \frac{1}{3}\ n^2 + 2$

$t_4(n) = 2^n$

26.7 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 7.3 zu finden.

2 `http://www.dpunkt.de/buecher/3358.html`

27 O-Notation

Auf dieser Seite wird die O-Notation[1] behandelt. Bei der O-Notation werden die asymptotischen oberen Schranke für Aufwandsfunktion angegeben. Das heißt deren Wachstumsgeschwindigkeit bzw. Größenordnung. Eine Asymptote ist eine Gerade, der sich eine Kurve bei immer größer werdender Entfernung vom Koordinatenursprung unbegrenzt nähert. Eine einfache Vergleichsfunktion ist $f(n) \in O(g(n))$ für Aufwandsfunktionen mit $g : \mathbb{N} \to \mathbb{N}$

27.1 Definition

Für eine Funktion $f : \mathbb{N} \to \mathbb{N}$ ist die Menge $O(f(n))$ wie folgt definiert:

$O(f(n)) = \{g : \mathbb{N} \to \mathbb{N} | \exists c \in \mathbb{R}^{>0}, \exists n_o \in \mathbb{N}\ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$

Anschaulich formuliert bedeutet das, dass O(f(n)) die Menge aller durch f nach oben beschränkter Funktionen ist und somit die asymptotische obere Schranke ist.

Die Definition veranschaulichst sieht folgendermaßen aus:

$g(n) \in O(f(n)) \Leftrightarrow \exists c > 0, \exists n_0 \forall n \geq n_0 : g(n) \leq c \cdot f(n)$

Das heißt g wächst nicht schneller als f. Das bedeutet wiederrum $\frac{g(n)}{f(n)}$ ist für genügend große n durch eine Konstante c nach oben beschränkt.

27.2 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 7.3.2 zu finden.

1 https://de.wikipedia.org/wiki/O-Notation
2 http://www.dpunkt.de/buecher/3358.html

28 Ω -Notation

Für eine Funktion $f : \mathbb{N} \to \mathbb{N}$ ist die Menge $\Omega(f(n))$ wie folgt definiert:

$\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{N} | \exists c \in \mathbb{R}^{>0}, \exists n_o \in \mathbb{N}\ \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$

Anschaulich formuliert bedeutet das, dass $\Omega(f(n))$ die Menge aller durch f nach unten beschränkter Funktionen ist und somit die asymptotische untere Schranke ist.

29 Θ -Notation

Die exakte Ordnung Θ von f(n) ist definiert als:

$\Theta(f(n)) = \{g : \mathbb{N} \to \mathbb{N} | \exists c_1 \in \mathbb{R}^{>0}, \exists c_2 \in \mathbb{R}^{>0}, \exists n_o \in \mathbb{N}\ \forall n \geq n_0 : c_1 \cdot f(n) \geq g(n) \geq c_2 \cdot f(n)\}$

Oder etwas kompakter:

$\Theta(f(n)) = O(f(n)) \bigcap \Omega(f(n))$

Anschaulich formuliert bedeutet das, dass Θ die Menge aller durch f nach unten und oben beschränkter Funktionen und somit die asymptotische untere und obere Schranke ist.

29.1 Beweis

Zu zeigen: $\Theta(f(n)) \subseteq O(f(n)) \cap \Omega(f(n))\ und\ \Theta(f(n)) \supseteq O(f(n)) \cap \Omega(f(n))$

$\Theta(f(n)) \subseteq O(f(n)) \cap \Omega(f(n)) :$

Zeige $g(n) \in \Theta(f(n)) \Rightarrow g(n) \in O(f(n)) \cap \Omega(f(n))$.

- $g(n) \in \Theta(f(n)) \Rightarrow \exists c_1, c_2, n_0 : \forall n \geq n_0 : c_1 f(n) \geq g(n) \geq c_2 f(n)$

$\Rightarrow \exists c_1, n_0 : \forall n \geq n_0 : c_1 f(n) \geq g(n)\ und\ \exists c_2, n_0 : \forall n \geq n_0 : c_1 f(n) \geq g(n) \geq c_2 f(n)$

$\Rightarrow g(n) \in O(f(n))\ und\ g(n) \in \Omega(f(n))$

$\Rightarrow g(n) \in O(f(n)) \cap \Omega(f(n))$

29.2 Beispiel 1

Wir stellen uns die Frage, ob $n^2 \in O(n^3)$ bzw. ob n^3 eine obere Schranke für n^2 ist. Die Antwort ist ja. Die Begründung dazu lautet folgendermaßen:

$n_0 = 1, c = 1$

$\Rightarrow n^2 \leq n^3$

$\Rightarrow 1 \leq n$ für $n \geq 1$

29.3 Beispiel 2

Wir stellen uns die Frage, ob $n^3 \in O(n^2)$ bzw. ob n^2 eine obere Schranke für n^3 ist. Die Antwort ist nein. Beweisen kann man das durch Widerspruch. Unsere Annahme ist: $\exists c, n_0 \in \mathbb{N} : n^3 \leq c \cdot n^2,$ für alle $n > n_0$

$n^3 \leq c \cdot n^2$, für alle $n > n_0$

$\Rightarrow n \leq c$, für alle $n > n_0$

Wähle $n = c + n_0 \Rightarrow c + n_0 \leq c$ Widerspruch!!

30 Lemma

Für beliebige Funktionen f,g gilt:
$O(f(n)+g(n)) = O(max(f(n),g(n))$

30.1 Beweis in beide Richtungen

$t(n) \in O(f(n)+g(n)) \Rightarrow t(n) \in O(max(f(n),g(n)))$

$t(n) \in O(f(n)+g(n)) \Leftarrow t(n) \in O(max(f(n),g(n)))$

Als erstes machen wir den Beweis nach rechts ($\Rightarrow$)

$\exists c, n_0 \in \mathbb{N} : t(n) \leq c \cdot (f(n)+g(n)) \ \forall n > n_o$

$\Rightarrow t(n) \leq 2 \cdot c \cdot max(f(n),g(n)) \ \forall n > n_0$

$\Rightarrow t(n) \in O(max(f(n)),g(n)))$

nun der Beweis nach links ($\Leftarrow$)

$\exists c, n_0 \in \mathbb{N} : t(n) \leq c \cdot (max(f(n),g(n))) \ \forall n > n_0$

$\Rightarrow t(n) \leq c \cdot (f(n)+g(n)) \ \forall n > n_0$

$\Rightarrow t(n) \in O(f(n),g(n))$

30.2 Beispiel

$O(n^4+n^2) = O(n^4)$

$O(n^4+4 \cdot n^3) = O(n^4)$

$O(n^4+2^n) = O(2^n)$

31 Lemma

1. $O(f(n)) \subseteq O(g(n))$ genau dann wenn $f(n) \in O(g(n))$
2. $O(f(n)) = O(g(n))$ genau dann wenn $f(n) \in O(g(n)) g(n) \in O(f(n))$
3. $O(f(n)) \subset O(g(n))$ genau dann wenn $f(n) \in O(g(n)) g(n) \notin O(f(n))$

31.1 Beweis in beide Richtungen

Beweis zu 1. nach rechts ($\Rightarrow$)

$O(f(n)) \subseteq O(g(n)) \Rightarrow f(n) \in O(g(n))$

$f(n) \in O(f(n)) \subseteq O(g(n)) \Rightarrow f(n) \in O(g(n))$

Beweis zu 1. nach links ($\Leftarrow$)

$O(f(n)) \subseteq O(g(n)) \Leftarrow f(n) \in O(g(n))$

$f(n) \in O(g(n)) \Rightarrow \exists c_0, n_0 \in \mathbb{N} : f(n) \leq c_0 \cdot g(n)\ \forall n > n_0$ (siehe Definition)

und sei t(n) ein beliebiges Element der Menge O(f(n))

$t(n) \in O(f(n)) \Rightarrow \exists c_1, n_1 \in \mathbb{N} : t(n) \leq c_1 \cdot f(n)\ \forall n > n_1$ (siehe Definition)

$\Rightarrow t(n) \leq c_1 \cdot f(n) \leq c_1 \cdot c_0 \cdot g(n)\ \forall n > max(n_0, n_1)$

$t(n) \in O(f(n)) \Rightarrow t(n) \in O(g(n))$

$O(f(n)) \subseteq O(g(n))$ (Definition der Teilmenge, da t(n) ein beliebiges Element ist)

31.2 Beispiele

$O(n^2) = \{n^2, 2n^2 - 6, 3n^2 + 5, \frac{1}{2}n^2 + 8, ...\}$

Damit ist

$(3n^2 + 5) \in O(n^2)$

$O(3n^2 + 5) \subseteq O(n^2)$

$O(3n^2 + 5) = \{n^2, 2n^2 - 6, 3n^2 + 5, \frac{1}{2}n^2 + 8, ...\}$

Damit ist

$n^2 \in O(3n^2 + 5)$

$O(n^2) \subseteq O(3n^2 + 5)$

Damit ist

$O(n^2) = O(3n^2 + 5)$

32 Lemma

Falls $f(n) \in O(g(n))$ *und* $g(n) \in O(h(n))$, dann ist auch $f(n) \in O(h(n))$.

32.1 Beweis

$f(n) \leq c_0 \cdot g(n)\ \forall n > n_0$ *und*

$g(n) \leq c_1 \cdot h(n)\ \forall n > n_1$ *und*

$\Rightarrow f(n) \leq c_0 \cdot g(n) \leq c_0 \cdot c_1 \cdot h(n)\ \forall n \geq max(n_0, n_1)$

Dabei ist $c_0 \cdot c_1$ eine Konstante.

32.2 Beispiel

$O(n^2) = O(3n^2) = O(\frac{1}{2}n^2)$

$O(n^2) \subseteq O(3n^2) \subseteq O(\frac{1}{2}n^2)$

$O(n^2) \subseteq O(n^{2,5}) \subseteq O(n^3)$

$O(n^2) \subset O(n^{2,5}) \subset O(n^3)$

33 Lemma

1. $lim_{n\to\infty}(f(n)/g(n)) = c, c > 0 \Rightarrow O(f(n)) = O(g(n))$
2. $lim_{n\to\infty}(f(n)/g(n)) = 0 \Rightarrow O(f(n)) \subset O(g(n))$

Ein häufiges Problem sind Grenzwerte der Art $\frac{\infty}{\infty}$ oder $\frac{0}{0}$ Bei diesem Problem kann man als Ansatz die Regel von de l'Hospital verwenden.

Satz(Regel von de L'Hospital) $x \to \infty$
Seien f und g auf dem Intervall $[\alpha, \infty)$ differenzierbar.
Es gelte $lim_{x\to\infty} f(x) = lim_{x\to\infty} g(x) = 0 (bzw. = \infty)$
und es existiere $lim_{x\to\infty} \frac{f'(x)}{g'(x)}$.
Dann existiert auch $lim_{x\to\infty} \frac{f(x)}{g(x)}$ und es gilt:
$lim_{x\to\infty} \frac{f'(x)}{g'(x)} = lim_{x\to\infty} \frac{f(x)}{g(x)}$

33.1 Beispiel

1. $f(n) = 3n + 5, g(n) = n$

$$lim_{n\to\infty} \frac{3n+5}{n} \Rightarrow lim_{n\to\infty} \frac{3}{1} = 3 > 0 \Rightarrow O(3n+5) = O(n)$$

2. $f(n) = n^2 + 5, g(n) = n^3$

$$lim_{n\to\infty} \frac{n^2+5}{n^3} \Rightarrow lim_{n\to\infty} \frac{2n}{3n^2} \Rightarrow lim_{n\to\infty} \frac{2}{6n} = 0 \Rightarrow O(n^2+5) \subset O(n^3)$$

Beim zweiten Beispiel musste die Regel von de l'Hospital wiederholt angewandt werden.

34 Lemma

Gibt es immer eine Ordnung zwischen den Funktionen? Es gibt Funktionen f und g mit $f(n) \notin O(g(n))$ *und* $g(n) \notin O(f(n))$. Ein Beispiel sind die Funktionen sin(n) und cos(n).

Für alle $m \in \mathbb{N}$ *gilt* : $O(n^m) \subseteq O(n^{m+1})$

34.1 Beweis durch Widerspruch

Wir nehmen an, dass $s(n) \in O(n^k)$,

das heißt $\exists c, n_0, \forall n > n_0 : s(n) \leq c \cdot n^k$.

Aber es muss auch $s(n) \notin O(n^{k+1})$ gelten,

das heißt $\exists n > n_0 : s(n) > c \cdot n^{k+1}$

$\Rightarrow \exists n > n_0 : und\ n < 1$

35 Komplexitätsklassen

Auf dieser Seite werden die Komplexitätsklassen[1] behandelt.

Wir sagen sei $f(n) = a_m \cdot n^m + a_{m-1} \cdot n^{m-1} + ... + a_1 \cdot n + a_0$, *wobei* $a_i \in \mathbb{R}^+$für $0 \leq i \leq m$. *Dann gilt* $f(n) \in O(n^m)$. Und wir sagen, ein Algorithmus mit Komplexität f(n) benötigt höchstens polynomielle Rechenzeit, falls es ein Polynom p(n) gibt, mit $f(n) \in O(p(n))$. Des weiteren sagen wir, dass ein Algorithmus höchstens exponentielle Rechenzeit benötigt, falls es eine Konstante $a \in \mathbb{R}^+$ gibt, mit $f(n) \in O(a^n)$.

Die Komplexitätsklassen sind:

$O(1)$	der konstante Aufwand, das bedeutet der Aufwand ist nicht abhängig von der Eingabe
$O(log\ n)$	der logarithmische Aufwand
$O(n)$	der lineare Aufwand
$O(n \cdot log\ n)$	
$O(n^2)$	der quadratische Aufwand
$O(n^k)$ für $k \geq 0$	der polynomiale Aufwand
$O(2^n)$	der exponentielle Aufwand

35.1 Wachstum

f(n)	n=2	$2^4 = 16$	$2^8 = 256$	$2^{10} = 1024$	$2^{20} = 1048576$
ldn	1	4	8	10	20
n	2	16	256	1024	1048576
$n \cdot ldn$	2	64	2048	10240	20971520
n^2	4	256	65536	1048576	$\approx 10^{12}$
n^3	8	4096	16777200	$\approx 10^9$	$\approx 10^{18}$
2^n	4	65536	$\approx 10^{77}$	$\approx 10^{308}$	$\approx 10^{315653}$

35.2 Zeitaufwand

Nun stellen wir uns die Frage, wie groß bezüglich der Rechenschritte darf, oder kann ein Problem sein, je nach Komplexitätsklasse, wenn die Zeit T begrenzt ist? Wir nehmen an,

1 `https://de.wikipedia.org/wiki/Komplexit%C3%A4tsklasse`

dass wir pro Schritt eine Rechenzeit von $1\mu s = (10^{-6}s)$ brauchen. In der folgenden Tabelle steht T für die Zeitbegrenzung und G für die maximale Problemgröße.

G	T=1Min.	1 Std.	1 Tag	1 Woche	1 Jahr
n	$6 \cdot 10^7$	$3{,}6 \cdot 10^9$	$8{,}6 \cdot 10^{10}$	$6 \cdot 10^{11}$	$3 \cdot 10^{13}$
n^2	7750	$6 \cdot 10^4$	$2{,}9 \cdot 10^5$	$7{,}8 \cdot 10^5$	$5{,}6 \cdot 10^6$
n^3	391	1530	4420	8450	31600
2^n	25	31	36	39	44

Ein Beispiel ist für T=1 Min. : $1000 \cdot 1000 \cdot 60 = 6 \cdot 10^7 \mu s$ (10^7 *Schritte*)

35.3 Typische Problemklassen

Aufwand	**Problemklasse**
$O(1)$	für einige Suchverfahren für Tabellen (Hashing)
$O(log\ n)$	für allgemeine Suchverfahren für Tabellen (Baum-Suchverfahren)
$O(n)$	für sequenzielle Suche, Suche in Texten, syntaktische Analyse von Programmen (bei "guter" Grammatik)
$O(n \cdot log\ n)$	für Sortieren
$O(n^2)$	für einige dynamische Optimierungsverfahren (z.B. optimale Suchbäume), einfache Multiplikation von Matrix-Vektor
$O(n^3)$	für einfache Matrizen Multiplikationen
$O(2^n)$	für viele Optimierungsprobleme (z.B. optimale Schaltwerke), automatisches Beweisen (im Prädikatenkalkül 1.Stufe)

35.4 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 7.3.3 zu finden.

2 `http://www.dpunkt.de/buecher/3358.html`

36 Aufwandsanalyse von iterativen Algorithmen

Auf dieser Seite wird der Aufwand[1] von iterativen[2] Algorithmen analysiert. Als Aufwand wird die Anzahl der durchlaufenen Operationen zur Lösung des Problems bezeichnet (Zuweisungen, Vergleiche...). Häufig ist der Aufwand abhängig vom Eingabeparameter (Problemgröße). Die Aufwandsklasse sagt, wie der Aufwand in Abhängigkeit von der Problemgröße wächst. Doch wie kann man nun bei beliebigem Java Code die Aufwandsklasse bestimmen?

36.1 Aufwand von Programmen ablesen

```
void alg1(int n){
     int m = 2;
     int i;
     int k = n;
     while (n > 0){
         i = k;
         while (i > 0) {
               m = m + i;
               i = i / 2;
         }
         n = n - 1;
    }
}
```

Die Aufwandsklasse ist $O(n \cdot log\ n)$. Die äußere Schleife wird n-mal durchlaufen und die Innere Schleife log n-mal.

```
void alg1(int n) {
      int m = 1;
      int i = 0;
      while (m < n) {
         while (i < m)
              i = i + 1;
      m = m + i;
      }
}
```

Hier ist die Aufwandsklasse O(n+log n). In jedem Durchlauf der äußeren Schleife wird m verdoppelt, d.h. sie läuft log n Mal. Die innere Schleife läuft bis n/2, aber nicht jedes Mal, weil i nur ein Mal auf 0 gesetzt wird. Man könnte als Aufwandsklasse auch O(n) sagen, da der Summand log n nicht ins Gewicht fällt.

---

1 https://de.wikipedia.org/wiki/Aufwandsklassen

2 https://de.wikipedia.org/wiki/Aufwandsklassen

36.2 Bestandteile iterativer Algorithmen

Zum einen haben wir elementare Anweisungen wie Zuweisungen und Vergleiche. Diese haben einen Aufwand von 1.

Des Weiteren haben wir Sequenzen α_1 *und* α_2 oder auch $\alpha_1;\alpha_2$ geschrieben. Die obere Grenze ist $O(f_{\alpha_1}(n)) + O(f_{\alpha_2}(n))$ und die untere Grenze ist $\Omega(f_{\alpha_1}(n)) + \Omega(f_{\alpha_2}(n))$. Dabei ist $f_{\alpha_1}(n)$ der Aufwand, der bei der Ausführung von α_1 entsteht.

Ein weiterer Bestandteil ist die Selektion. $if(B)\{\alpha_1\}else\{\alpha_2\}$. Hier ist die obere Grenze $O(f_B(n)) + O(max(f_{\alpha_1}(n), f_{\alpha_2}(n)))$ und die untere Grenze $\Omega(f_B(n)) + \Omega(min(f_{\alpha_1}(n), f_{\alpha_2}(n)))$.

Außerdem haben wir Iterationen $while(B)\{\alpha\}$. Hierbei ist die obere und die untere Grenze die Anzahl der Schleifendurchläufe, $(O(f_B(n)) + O(f_\alpha(n)))$ und die untere Grenze $(\Omega(f_B(n)) + \Omega(f_\alpha(n)))$. Doch wie ist der Aufwand für eine for-Schleife? Ein Beispiel ist $for(\alpha_1; B; \alpha_2)\{\alpha_3\}$. Die Antwort ist die Abbildung auf eine while-Schleife.

α_1;

while(B) {

α_3;

α_2;

}

36.2.1 Beispiel Sequenz

```
public int berechne(int n) {
  int x = 0;
  x = x + 1;
  return x;
}
```

Jede Zeile hat den Aufwand $\Theta(1)$. Wie viele Operationen werden nun durchlaufen? Und ist die Anzahl abhängig vom Eingabeparameter? Der Aufwand ist $f(n) = \Theta(1) + \Theta(1) + \Theta(1) = 3 \cdot \Theta(1)$

Die Aufwandsklasse ist somit $\Theta(f(n)) = \Theta(1)$

36.2.2 Beispiel Schleifen

```
public int berechne(int n) {
  int x = 0;
  for (int i=0; i < n; i++) {
    x = x + 1;
  }
  return x;
}
```

Die for Schleife hat den Aufwand $n \cdot \Theta(1)$. Die Initialisierung und das return haben jeweils den Aufwand $\Theta(1)$.

Der Gesamtaufwand ist somit $f(n) = \Theta(1) + n \cdot \Theta(1) + \Theta(1) = 2 \cdot \Theta(1) + \Theta(n)$. Somit ist die Aufwandsklasse $\Theta(f(n)) = \Theta(n)$.

```
public int berechne(int n) {
  int x = 0;
  for (int i=0; i < n; i++) {
    for (int j=0; j < n; j++) {
      x = x + 1;
    }
  }
  return x;
}
```

Hier hat die for-Schleife den Aufwand $n \cdot (n \cdot \Theta(1))$ und die Initialisierung und das return wieder $\Theta(1)$. Damit ergibt der sich Gesamtaufwand $f(n) = \Theta(1) + n^2 \cdot \Theta(1) + \Theta(1) = 2 \cdot \Theta(1) + \Theta(n^2)$. Daraus folgt die Aufwandsklasse $\Theta(f(n)) = \Theta(n^2)$.

36.2.3 Beispiel Selektion

```
public int berechne(int n) {
   if (n % 2 == 0) {
      int x = 0;
      for (int i=0; i < n; i++) {
         x = x + 1;
      }
      return x;
   }else{
      return n;
   }
}
```

Hier hat die for-Schleife einen Aufwand von $\Theta(n)$. Die Initialisierung und das return wieder $\Theta(1)$.

Die obere Grenze ist somit $O(f(n)) = \Theta(1) + O(max(\Theta(n), \Theta(1)) = O(n)$ und die untere Grenze $\Omega(f(n)) = \Theta(1) + \Omega(min(\Theta(n), \Theta(1)) = \Omega(1)$

36.2.4 Faustregeln

Zu den häufig verwendeten Faustregeln gehört, dass wenn wir keine Schleife haben, der Aufwand konstant ist. Eine weitere ist, dass bei einer Schleife immer ein linearer Aufwand vorliegt. Bei zwei geschachtelten Schleifen haben wir immer einen quadratischen Aufwand. Doch die Faustegeln gelten nicht ohne Ausnahmen. Besonders Acht geben muss man bei Aufwandsbestimmungen für Schleifen, bei mehreren Eingabevariablen, bei Funktionsaufrufen und bei Rekursionen.

36.2.5 Aufwandsbestimmung für Schleifen

```
public int berechne(int n) {
  int x = 0;
```

```
  for (int i=0; i < 5; i++) {
    x = x + 1;
  }
  return x;
}
```

Der Schleifenabbruch hängt nicht vom Eingabeparameter ab. Der Aufwand beträgt $f(n) = \Theta(1) + 5 \cdot \Theta(1) + \Theta(1) = 7 \cdot \Theta(1)$ somit haben wir die Aufwandsklasse $\Theta(fn)) = \Theta(1)$

```
public int berechne(int n) {
  int x = 0;
  for (int i=1; i < n; i = 2*i) {
    x = x + 1;
  }
  return x;
}
```

Hier wächst die Laufvariable nicht linear an.Daher ist der Aufwand $f(n) = \Theta(1) + log_2 n \cdot \Theta(1) + \Theta(1)$ und wir haben die Aufwandsklasse $\Theta(f(n)) = \Theta(log\ n)$.

Doch gibt es eine allgemeine Methodik zum Bestimmen des Schleifenaufwands?

```
for (int i=1; i < n; i=2*i) {
  x = x + 1;
}
```

Schritt 1 : Wie entwickelt sich hier die Laufvariable? Der Startwert i ist 1 und die Veränderung in jedem Schritt ist $i = 2 \cdot i$. Die Laufvariable entwickelt sich somit wie folgt:

Nach dem 1. Durchlauf $i = 1 \cdot 2 = 2^1$

Nach dem 2. Durchlauf $i = (1 \cdot 2) \cdot 2 = 2^2$

Nach dem 3. Durchlauf $i = ((1 \cdot 2) \cdot 2) \cdot 2 = 2^3$

Nach dem k. Durchlauf $i = 2^k$

Schritt 2: Nach wie vielen Durchläufen wird die Schleife abgebrochen?

Der Abbruch erfolgt, wenn $i \geq n$

$:i \geq n \qquad |i = 2^k$

$\Leftrightarrow 2^k \geq n \qquad |log_2$

$\Leftrightarrow k \geq log_2 n$

Somit erfolgt ein Abbruch nach $k = \lceil\ log_2 n\ \rceil$ Durchläufen.

```
public int berechne(int[] f1, int[] f2) {
   int result = 0;
   for (int i=0; i < f1.length; i++) {
      for (int j=0; j < f2.length; j++) {
         if (f1[i] == f2[j]) result++;
      }
   }
   return result;
}
```

Hier haben wir nun eine for Schleife mit mehreren Eingabevariablen. Die Problemgrößen sind $n = f1.length\ und\ m = f2.length$.

```
public int berechne2(int[] f1, int[] f2){
   f2 = mergeSort(f2);
   int result = 0;
   for (int i=0; i < f1.length; i++) {
      if (binarySearch(f2, f1[i])) result++;
   }
   return result;
}
```

Der Aufwand ist hier $f(n,m) = \Theta(m \cdot log\ m) + \Theta(1) + n \cdot (\Theta(log\ m) + O(1)) + \Theta(1)$. Somit ist die Aufwandsklasse $\Theta(f(n,m)) = \Theta(m \cdot log\ m + n \cdot log\ m)$.

In diesem Beispiel haben wir wieder mehreren Eingabevariablen. Diese sind die gleichen Problemgrößen $n = f1.length\ und\ m = f2.length$.

```
public int berechne2(int[] f1, int[] f2){
   int result = 0;
   for (int i=0; i < f1.length; i++) {
      for (int j=0; j < f2.length; j++) {
         if (f1[i] == f2[j]) result++;
      }
   }
   return result;
}
```

Der Aufwand ist hier wie folgt: $f(n,m) = \Theta(1) + n \cdot (m \cdot (\Theta(1) + O(1))) + \Theta(1)$. Somit ist die Aufwandsklasse $\Theta(f(n,m)) = \Theta(n \cdot m)$.

36.3 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[3] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 7.3.4 zu finden.

3 http://www.dpunkt.de/buecher/3358.html

37 Aufwandsanalyse von rekursiven Algorithmen

Auf dieser Seite wird der Aufwand von rekursiven Algorithmen untersucht.

```
public int fib(int n) {
   if (n == 0 || n == 1) {
      return 1;
   } else {
      return fib(n-1) + fib(n-2);
   }
}
```

Wie ist nun der Aufwand für Fibonacci? Bei Rekursionsabbruch $f(n) = \Theta(1) + \Theta(1)$ und im Rekursionsfall $f(n) = \Theta(1)+???$. Zur Bestimmung benutzen wir Rekursionsgleichungen.

37.1 Rekursionsgleichungen

Eine Rekursionsgleichung[1] ist eine Gleichung oder Ungleichung, die eine Funktion anhand ihrer Anwendung auf kleinere Werte beschreibt.

Rekursionsgleichung für Fibonacci:

$$T(n) := \begin{cases} \Theta(1) & \text{für } (n = 0 \vee n = 1) \\ \Theta(1) + T(n-1) + T(n-2) & \textit{sonst} \end{cases}$$

37.2 Lösung von Rekursionsgleichungen

Die Frage ist nun welche Aufwandklasse T(n) beschreibt. Dies könnten alle möglichen Aufwandsklassen sein. Methoden um dieses Problem zu lösen sind die vollständige Induktion und das Master-Theorem.

37.3 Spezialfall Divide and Conquer Algorithmus

Ein Divide-and-Conquer Algorithmus[2] stellt im Allgemeinen eine einfache, rekursive Version eines Algorithmus dar und hat drei Schritte:

1 https://de.wikipedia.org/wiki/Rekursionsgleichung
2 https://de.wikipedia.org/wiki/Divide_and_conquer

1. Divide: Unterteile das Problem in eine Zahl von Teilproblemen
2. Conquer: Löse das Teilproblem rekursiv. Wenn das

Teilproblem klein genug ist, dann löse das Teilproblem direkt (z.B. bei leeren oder einelementigen Listen)

1. Combine: Die Lösungen der Teilprobleme werden zu einer Gesamtlösung kombiniert.

Merge Sort ist beispielsweise ein Divide and Conquer Algorithmus.

1. Divide: Zerteile eine Folge mit n Elementen in zwei Folgen mit je n/2 Elementen.
2. Conquer: Wenn die resultierende Folge 1 oder 0 Elemente enthält, dann ist sie sortiert.Ansonsten wende Merge Sort rekursiv an.
3. Combine: Mische die zwei sortierten Teilfolgen.

```
public List mergeSort(List f) {
  if (f.size() <= 1) {
    return f;
  } else {
    int m = f.size() / 2;
    List left = mergeSort(f.subList(0,m));
    List right = mergeSort(f.subList(m,f.size());
    return merge(left, right);
  }
}
```

Die dazugehörige Rekursionsgleichung lautet:

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

Im Allgemeinen ist die Rekursionsgleichung für Divide and Conquer Algorithmen:

$$T(n) := \begin{cases} \Theta(1) & \text{für } (n \leq 1) \\ D(n) + a \cdot T(n/b) + C(n) & \textit{sonst} \end{cases}$$

mit D(n) als Aufwand für Divide, T(n/b) als Aufwand für Conquer und C(n) als Aufwand für Combine.

37.3.1 Ab- und Aufrunden

Die Rekursionsgleichung von MergeSort beschreibt den Aufwand für den schlechtesten Fall.

$$T(n) := \begin{cases} \Theta(1) & \text{für } (n \leq 1) \\ \Theta(1) + T(n/2) + T(n/2) + \Theta(n) & \textit{sonst} \end{cases}$$

Aber die Annahme, dass n eine geeignete ganze Zahl ist ergibt normalerweise das gleiche Ergebnis wie eine beliebige Zahl mit Auf- bzw. Abrunden. Dies führt zur einfacheren Rekursionsgleichung:

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

37.3.2 Beispiel Binäre Suche

```
public List binarySearch(ArrayList<Integer> f, int e) {
  if (f.size() == 0) {
    return -1;
  } else {
    int m = f.size() / 2;
    if (f.get(m) == e) {
      return m;
    } else if (f.get(m) < e) {
      return binarySearch(f.subList(0, m), e);
    } else {
      return binarySearch(f.subList(m+1, f.size()), e);
    }
  }
}
```

Die Rekursionsgleichung lautet $T(n) := \begin{cases} \Theta(1) & \text{für } (n = 0) \\ \Theta(1) + T(n/2) & sonst \end{cases}$

37.4 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[3] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 7.3.4 zu finden.

3 http://www.dpunkt.de/buecher/3358.html

38 Vollständige Induktion

Auf dieser Seite wird die vollständige Induktion[1] behandelt. Es handelt sich hierbei um eine rekursive Beweistechnik aus der Mathematik. Sie ist gut geeignet, um Eigenschaften von rekursiv definierten Funktionen zu beweisen.

38.1 Vorgehen

Zunächst vermutet man eine Eigenschaft (z.B. Aufwandsklasse einer Rekursionsgleichung). Nun folgt der Induktionsanfang: Eigenschaft hält für ein kleines n. Als nächstes folgt der Induktionsschritt: Die Annahme ist, dass wir es bereits für ein kleineres n gezeigt haben und wenn die Eigenschaft für kleinere n hält, dann hält sie auch für das nächstgrößere n!

38.2 Beispiel 1

$$T(n) := \begin{cases} \Theta(1) & \text{für } n \leq 1 \\ 4 \cdot T(\frac{n}{2}) + \Theta(n^3) & \textit{sonst} \end{cases}$$

Nun wollen wir die obere Grenze für den Aufwand bestimmen. Unsere Vermutung ist, dass $T(n) \in O(n^3)$. Nun müssen wir zeigen, dass $\exists n_0, c : \forall n \geq n_o : T(n) \leq c \cdot n^3$ (siehe Definition der O-Notation). Die vereinfachte Annahme lautet $n = 2^k$. Hierbei werden keine Spezialfälle behandelt und im Induktionsschritt wird von $\frac{n}{2}$ nach n gegangen.

Induktionsvermutung: $T(\frac{n}{2}) \leq c \cdot (\frac{n}{2})^3$

Induktionsschritt: Wir beweisen von $\frac{n}{2}$ *nach* n

zu zeigende obere Grenze:

$$T(n) \leq c \cdot n^3 \quad |T(n) = 4 \cdot T(\frac{n}{2}) + n^3$$

Rekursionsgleichung einsetzen:

$\Leftrightarrow 4 \cdot T(\frac{n}{2}) + n^3 \leq c \cdot n^3 | T(\frac{n}{2}) \leq c \cdot (\frac{n}{2})^3$

Induktionsvermutung einsetzen:

$\Leftarrow 4 \cdot c \cdot (\frac{n}{2})^3 + n^3 \leq c \cdot n^3$

$\Leftrightarrow 4 \cdot c \cdot (\frac{n^3}{8}) + n^3 \leq c \cdot n^3 | - c \cdot n^3$

1 https://de.wikipedia.org/wiki/Vollst%C3%A4ndige_Induktion

$\Leftrightarrow -\frac{1}{2} \cdot c \cdot n^3 + n^3 \leq 0 | : n^3$

$\Leftrightarrow -\frac{1}{2} \cdot c + 1 \leq 0 | + \frac{1}{2} \cdot c$

$\Leftrightarrow 1 \leq \frac{1}{2} \cdot c | \cdot 2$

$\Leftrightarrow 2 \leq c$

Somit ist der Induktionsschritt erfolgreich, wenn $c \geq 2$.

Induktionsanfang

Wir zeigen die Induktionsvermutung für einen Anfangswert, am einfachsten ist es, dies für den Rekursionsabbruch zu zeigen.

Zu zeigende obere Grenze:

$$T(1) \leq c \cdot 1^3 | T(1) = 1$$

Rekursionsgleichung einsetzen:

$\Leftrightarrow 1 \leq c$

Der Induktionsanfang ist erfolgreich, wenn $c \geq 1$ ist. Doch wann können wir zeigen, dass $T(n) \leq c \cdot n^3$ ist? Für den Wert, den wir im Induktionsanfang gezeigt haben, also für $n_0 = 1$ und wenn $(c \geq 2c \geq 1) \Rightarrow c \geq 2$.

38.3 Beispiel 2

$$T(n) := \begin{cases} \Theta(1) & \text{für } n \leq 1 \\ 4 \cdot T(\frac{n}{2}) + \Theta(n) & \textit{sonst} \end{cases}$$

Nun wollen wir die obere Grenze für den Aufwand bestimmen. Unsere Vermutung ist, dass $T(n) \in O(n^2)$. Nun müssen wir zeigen, dass $\exists n_0, c : \forall n \geq n_0 : T(n) \leq c \cdot n^2$. Die vereinfachte Annahme lautet $n = 2^k$.

Induktionsvermutung: $T(\frac{n}{2}) \leq c \cdot (\frac{n}{2})^2$

Induktionsschritt: Wir beweisen von $\frac{n}{2}$ *nach* n

$$T(n) \leq c \cdot n^2 | T(n) = 4 \cdot T(\frac{n}{2}) + n$$

$\Leftrightarrow 4 \cdot T(\frac{n}{2}) + n \leq c \cdot n^2 | T(\frac{n}{2}) \leq c \cdot (\frac{n}{2})^2$

$\Leftarrow 4 \cdot c \cdot (\frac{n}{2})^2 + n \leq c \cdot n^2$

$\Leftrightarrow 4 \cdot c \cdot (\frac{n^2}{4}) + n \leq c \cdot n^2 | - c \cdot n^2$

$\Leftrightarrow n \leq 0$

Das Problem ist nun, dass wir den Induktionsschritt für positive n zeigen wollen und nicht für negative, daher müssen wir neu ansetzen.

Induktionsvermutung:

Dabei gibt es folgenden Trick: Modifiziere die Induktionsvermutung, in dem ein kleineres Polynom addiert wird.

$T(\frac{n}{2}) \leq c_1 \cdot (\frac{n}{2})^2 + c_2 \cdot \frac{n}{2}$

Induktionsschritt: Wir beweisen von $\frac{n}{2}$ *nach* n

$$T(n) \leq c_1 \cdot n^2 + c_2 \cdot n$$

$\Leftrightarrow 4 \cdot T(\frac{n}{2}) + n \leq c_1 \cdot n^2 + c_2 \cdot n$

$\Leftarrow 4 \cdot (c_1 \cdot (\frac{n}{2})^2 + c_2 \cdot \frac{n}{2}) + n \leq c_1 \cdot n^2 + c_2 \cdot n$

$\Leftrightarrow c_1 \cdot n^2 + 2 \cdot c_2 \cdot n + n \leq c_1 \cdot n^2 + c_2 \cdot n | - c_1 \cdot n^2; - c_2 \cdot n$

$\Leftrightarrow c_2 \cdot n + n \leq 0$

$\Leftrightarrow c_2 + 1 \leq 0$

$\Leftrightarrow c_2 \leq -1$

Induktionsanfang für n=1

$$T(1) \leq c_1 \cdot 1^2 + c_2 \cdot 1 | T(1) = 1$$

$\Leftrightarrow 1 \leq c_1 + c_2 | - c_2$

$\Leftrightarrow 1 - c_2 \leq c_1$

Wann können wir nun zeigen, dass $T(n) \leq c_1 \cdot n^2 + c_2 \cdot n$?

Für $n_0 = 1$ *und wenn* $(c_2 \leq -1 c_1 \geq 1 - c_2)$. Somit haben wir gezeigt, dass $T(n) \in O(n^2 + n) \Rightarrow T(n) \in O(max(n^2, n)) \Rightarrow T(n) \in O(n^2)$

39 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 7.2.5 zu finden.

1 http://www.dpunkt.de/buecher/3358.html

40 Mastertheorem

Auf dieser Seite wird das Master Theorem[1] behandelt. Die Mastermethode ist ein „Kochrezept“ zur Lösung von Rekursionsgleichungen der Form:

$T(n) = aT(n/b) + f(n)$ mit den Konstanten $a \geq 1$ *und* $b > 1$, f(n) ist eine asymptotische, positive Funktion, d.h. $f(n) > 0 \forall n > n_0$

- a steht dabei für die Anzahl der Unterprobleme.
- n/b ist die Größe eines Unterproblems
- T(n/b) ist der Aufwand zum Lösen eines Unterproblems (der Größe n/b)
- f(n) ist der Aufwand für das Zerlegen und Kombinieren in bzw. von Unterproblemen

Bei der Mastermethode handelt es sich um ein schnelles Lösungsverfahren zur Bestimmung der Laufzeitklasse einer gegebenen rekursiv definierten Funktion. Dabei gibt es 3 gängige Fälle:

- Fall 1: Obere Abschätzung
- Fall 2: Exakte Abschätzung
- Fall 3: Untere Abschätzung

Lässt sich keiner dieser 3 Fälle anwenden, so muss die Komplexität anderweitig bestimmt werden und wir müssen Voraussetzungen für die Anwendung des Mastertheorems überprüfen.

Dafür vergleicht man $f(n)$ mit $n^{lob_b a}$. Wir verstehen n/b als $\llcorner n/b \lrcorner$ *oder* $\ulcorner n/b \urcorner$. Im Folgenden verwenden wir die verkürzte Notation $log_2 n$ *als* $ld\ n$.

40.1 Fall 1

Wenn $f(n) \in O(n^{log_b a - \epsilon})$für ein $\epsilon > 0$. Daraus folgt, dass f(n) polynomiell langsamer wächst als $n^{log_b a}$ um einen Faktor n^{ϵ}. Damit haben wir die Lösung $T(n) = \Theta(n^{log_b a})$.

40.2 Fall 2

Wenn $f(n) \in \Theta(n^{log_b a} \cdot ld^k n)$für ein $k \geq 0$. Daraus folgt, dass f(n) und $n^{log_b a} \cdot ld^k n$ vergleichbar schnell wachsen. Damit haben wir die Lösung $T(n) = \Theta(n^{log_b a} \cdot ld^{k+1} n)$.

1 https://de.wikipedia.org/wiki/Master-Theorem

40.3 Fall 3

Wenn $f(n) \in \Omega(n^{log_b a+\epsilon})$für ein $\epsilon > 0$ und die Regularitätsbedingung $a \cdot f(n/b) \leq c \cdot f(n)$ für eine Konstante $c \in (0,1)$ und genügend große n erfüllt. Daraus folgt, dass f(n) polynomiell schneller wächst als $n^{log_b a}$ um einen Faktor n^{ϵ} und f(n) erfüllt die sogenannte Regularitätsbedingung. Damit haben wir die Lösung $T(n) \in \Theta(f(n))$.

40.3.1 Bedeutung

In jedem Fall vergleichen wir f(n) mit $n^{lob_b a}$. Intuitiv kann man sagen, dass die Lösung durch die größere Funktion bestimmt wird. Im zweiten Fall wachsen sie ungefähr gleich schnell. Im ersten und dritten Fall muss f(n) nicht nur kleiner oder größer als $n^{lob_b a}$ sein, sondern auch polynomiell kleiner oder größer um einen Faktor n^{ϵ}. Der dritte Fall kann nur angewandt werden, wenn die Regularitätsbedingung erfüllt ist.

Regularitätsbedingung

Doch wozu wird die Regularitätsbedingung benötigt? Zur Erinnerung, im dritten Fall dominiert f(n) das Wachstum von T(n). Wir müssen an dieser Stelle sicherstellen, dass auch bei rekursivem Anwenden, also wenn die Argumente kleiner werden, T(n) von f(n) dominiert wird. Veranschaulicht heißt das:

$T(n) = aT(n/b) + f(n)$

$$= a(aT(n/b^2) + f(n/b)) + f(n)$$

$$= a^2T(n/b^2) + af(n/b) + f(n)$$

für $af(n/b) \leq cf(n)(c \in (0,1))$ Das Wachstum muss durch f(n) dominiert werden und darf f(n) nicht dominieren.

Die Regularitätsbedingung gilt wenn sie für f(n) und g(n) gilt auch für $f(n) \cdot g(n)$ und auch für $f(n) + g(n)$

Nachweis für $f(n) \cdot g(n)$

Voraussetzung ist, dass die Regularitätsbedingung für f(n) und g(n) gilt, d.h.:

$\exists c_1 \in (0,1), \exists n_1 \in \mathbb{N}\ \forall n \geq n_1 : af(n/b) \leq c_1 f(n)$

$\exists c_2 \in (0,1), \exists n_2 \in \mathbb{N}\ \forall n \geq n_2 : ag(n/b) \leq c_2 g(n)$

Für $(f \cdot g)(n)$ gilt:

$a(f \cdot g)(n/b) = af(n/b) \cdot ag(n/b)$

man wählt $c = c_1 \cdot c_2 \in (0,1)$

und $n_0 = max\ \{n_1, n_2\}$

$\forall n \geq n_0 : af(n/b) \cdot ag(n/b) \leq c_1 f(n) \cdot c_2 g(n) = c(f \cdot g)(n)$

Nachweis für $f(n) + g(n)$

Voraussetzung ist, dass die Regularitätsbedingung für f(n) und g(n) gilt, d.h.:

$\exists c_1 \in (0,1), \exists n_1 \in \mathbb{N}\ \forall n \geq n_1 : af(n/b) \leq c_1 f(n)$

$\exists c_2 \in (0,1), \exists n_2 \in \mathbb{N}\ \forall n \geq n_2 : ag(n/b) \leq c_2 g(n)$

Für $(f+g)(n)$ gilt:

$a(f+g)(n/b) = af(n/b) + ag(n/b)$

man wählt $c = max\ \{c_1, c_2\}$

und $n_0 = max\ \{n_1, n_2\}$

$\forall n \geq n_0 : af(n/b) + ag(n/b) \leq c_1 f(n) + c_2 g(n) \leq c(f+g)(n)$

40.4 Überblick

Ist T(n) eine rekursiv definierte Funktion der Form

$$T(n) = aT(n/b) + f(n)\ \ mit\ \ a \geq 1, b > 1, \forall n > n_0 : f(n) > 0$$

Dann gilt:

- 1. Fall: Wenn $f(n) \in O(n^{log_b a - \epsilon})$ für ein $\epsilon > 0$ *dann* $T(n) = \Theta(n^{log_b a})$
- 2. Fall: Wenn $f(n) \in \Theta(n^{log_b a} \cdot ld^k n)$ für ein $k \geq 0$ *dann* $T(n) = \Theta(n^{log_b a} \cdot ld^{k+1} n)$
- 3. Fall: Wenn $f(n) \in \Omega(n^{log_b a + \epsilon})$ für ein $\epsilon > 0$ und $a \cdot f(n/b) \leq c \cdot f(n)$ für eine Konstante $c \in (0,1)$ und genügend große n dann $T(n) = \Theta(f(n))$

40.5 Idee

Wir haben folgenden Rekursionsbaum:

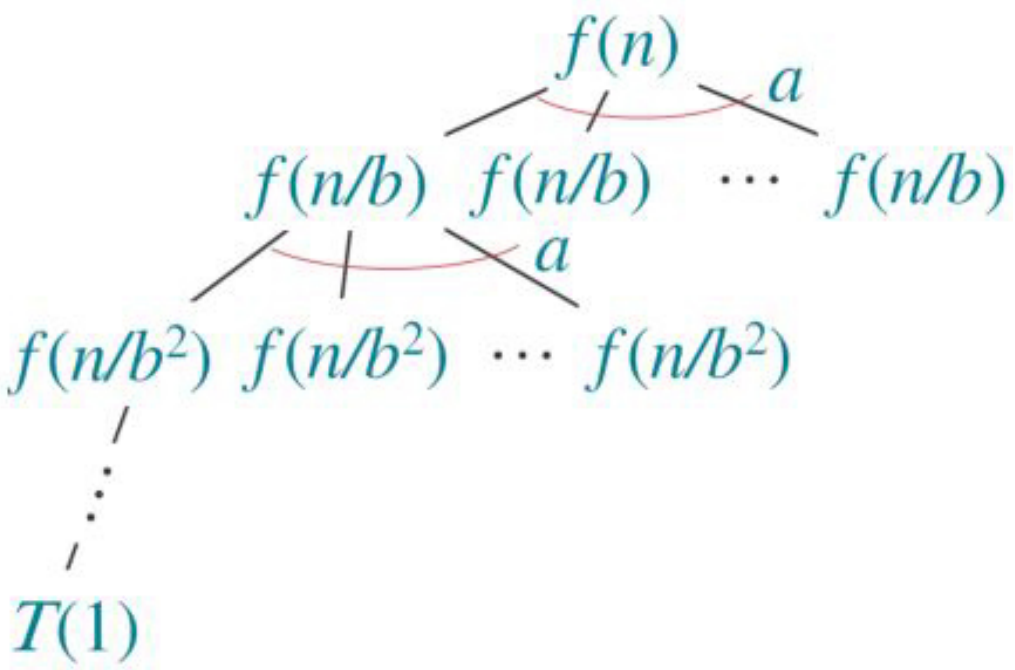

Abb. 3 Rekursionsbaum_Mastertheorem

Auf der ersten Ebene ist der Aufwand f(n), auf der zweiten Ebene $af(n/b)$ und auf der dritten Ebene $a^2f(n/b^2)$. Die Höhe des Baumes beträgt $h = log_b n$. Die Anzahl der Blätter berechnet sich durch a^h und beträgt somit $a^{log_b n} = n^{log_b a}$.

Fall 1: Das Gewicht wächst geometrisch von der Wurzel zu den Blättern. Die Blätter erhalten einen konstanten Anteil des Gesamtgewichts.

$\Theta(n^{log_b a})$

Fall 2: k ist 0 und das Gewicht ist ungefähr das Gleiche auf jedem der $log_b a$ Ebenen.

$\Theta(n^{log_b a} \cdot ld\ n)$

Fall 3: Das Gewicht reduziert sich geometrisch von der Wurzel zu den Blättern. Die Wurzel erhält einen konstanten Anteil am Gesamtgewicht.

$\Theta(f(n))$

40.6 Beispiel 1

$T(n) = 4T(n/2) + n$

$a = 4, b = 2 \Rightarrow n^{log_b a} = n^{log_2 4} = n^2$

$f(n) = n$

Fall 1: $f(n) \in O(n^{2-\epsilon})$für $\epsilon > 0$

$\Rightarrow T(n) = \Theta(n^2)$

40.7 Beispiel 2

$T(n) = 4T(n/2) + n^2$

$a = 4, b = 2 \Rightarrow n^{log_b a} = n^{log_2 4} = n^2$

$f(n) = n^2$

Fall 2: $f(n) \in \Theta(n^2\ ld^k\ n)$ für $k = 0$

$\Rightarrow T(n) = \Theta(n^2\ ld\ n)$

40.8 Beispiel 3

$T(n) = 4T(n/2) + n^3$

$a = 4, b = 2 \Rightarrow n^{log_b a} = n^{log_2 4} = n^2$

$f(n) = n^3$

Fall 3: $f(n) \in \Omega(n^{2+\epsilon})$für $\epsilon > 0$

und $4(\frac{n}{2})^3 \leq cn^3$ (Regularitätsbedingung)

für $c = \frac{1}{2}$

$\Rightarrow T(n) = \Theta(n^3)$

40.9 Beispiel 4

$T(n) = 4T(n/2) + \frac{n^2}{log\ n}$

$a = 4, b = 2 \Rightarrow n^{log_b a} = n^{log_2 4} = n^2$

$f(n) = \frac{n^2}{log\ n}$

Welcher Fall liegt nun vor? Das Mastertheorem kann an dieser Stelle nicht benutzt werden, da

- 1. Fall $f(n) \notin O(n^{2-\epsilon})$
- 2. Fall $f(n) \notin \Theta(n^2 \cdot ld^k\ n)$ für $k \geq 0$
- 3. Fall $f(n) \notin \Omega(n^{2+\epsilon})$

40.10 Nützliche Hinweise

- Basisumrechnung

$log_b x = \frac{log_a x}{log_a b} \Rightarrow O(log_b x) = O(log_a x)$

- de L'Hospital

$lim_{x\to\infty}\frac{f(x)}{g(x)} = lim_{x\to\infty}\frac{f'(x)}{g'(x)}$

- Vergleiche Logarithmus vs. Polynom

$lim_{x\to\infty} log_b x = \infty$

$lim_{x\to\infty} x^\epsilon = \infty$ für $\epsilon > 0$

$lim_{x\to\infty}\frac{log_b x}{x^\epsilon} = lim_{x\to\infty}\frac{(log_b x)'}{(x^\epsilon)'} = lim_{x\to\infty}\frac{\frac{1}{x}}{\epsilon x^{\epsilon-1}} = lim_{x\to\infty}\frac{1}{x\epsilon x^{\epsilon-1}} = lim_{x\to\infty}\frac{1}{\epsilon x^\epsilon} = 0$ für $\epsilon > 0$

41 Rekursionsbäume

Auf dieser Seite wird das Thema Rekursionsbäume behandelt. Das allgemeine Problem ist, dass man zum Abschätzen von der Aufwandsklasse einer Rekursionsgleichung gute Vermutungen braucht. Doch wie kommt man darauf? Ein Ansatz ist die Veranschaulichung durch einen Rekursionsbaum. Die Aufwandsklasse wird dann durch die Rekursionsbaummethode bestimmt. Das ist sehr nützlich, um eine Lösung zu raten, die danach durch eine andere Methode (z.B. Induktion) gezeigt wird. Rekursionsbäume sind besonders anschaulich bei Divide-and-Conquer-Algorithmen.

41.1 Spezialfall Divide and Conquer

Bei MergeSort sehen die Divide and Conquer Schritte wie folgt aus:

1. Divide: Zerteile eine Folge mit n Elementen in zwei Folgen mit je n/2 Elementen.
2. Conquer: Wenn die resultierende Folge 1 oder 0 Elemente enthält,dann ist sie sortiert. Ansonsten wende MergeSort rekursiv an.
3. Combine: Mische die zwei sortierten Teilfolgen.

$$T(n) := \begin{cases} \Theta(1) & \text{für } (n \leq 1) \\ D(n) + \alpha \cdot T(n/b) + C(n) & \text{sonst} \end{cases}$$

```
public List mergeSort(List f) {
  if (f.size() <= 1) {
    return f;
  } else {
    int m = f.size() / 2;
    List left = mergeSort(f.subList(0,m));
    List right = mergeSort(f.subList(m,f.size());
    return merge(left, right);
  }
}
```

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

41.1.1 Rekursionsbaum

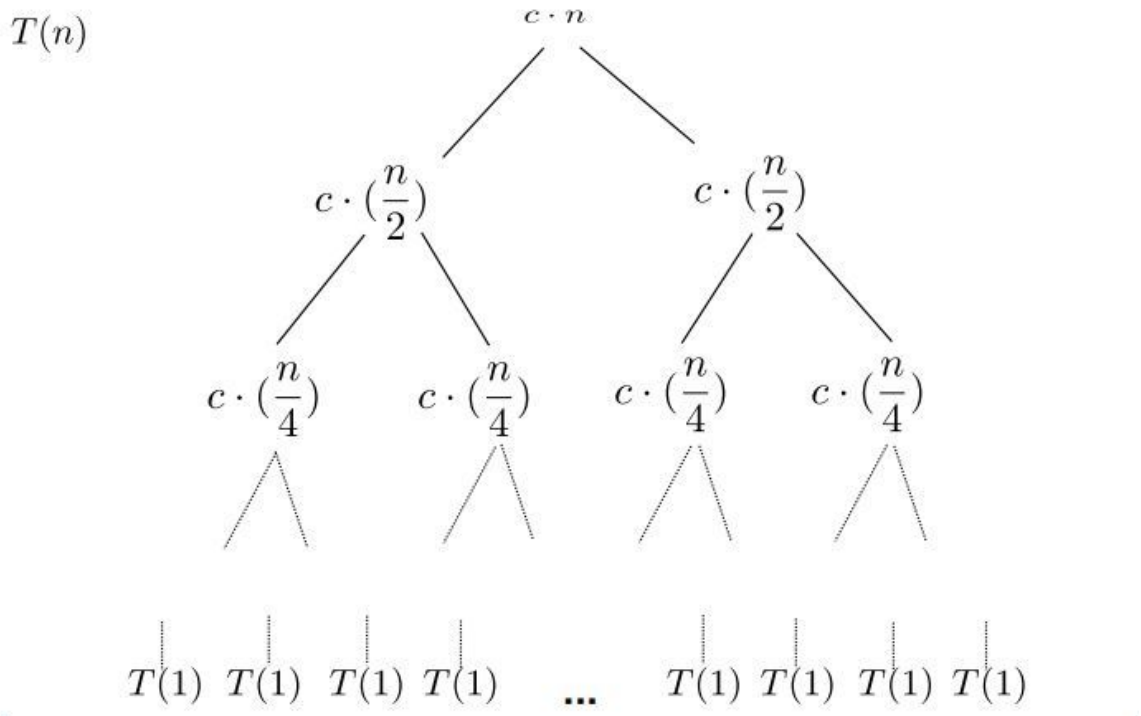

Abb. 4 Rekursionsbaum

41.2 Herleitung des Aufwandes

Die Grundidee ist das wiederholte Einsetzen der Rekursionsgleichung in sich selbst als Baum dargestellt. Das Ziel ist ein Muster zu erkennen. Bei einem Rekursionsbaum beschreibt ein Knoten die Kosten eines Teilproblems. Die Blätter sind die Kosten der Basis fällt T(0) und T(1). Der Aufwand bestimmt sich aus der Summe über alle Ebenen.

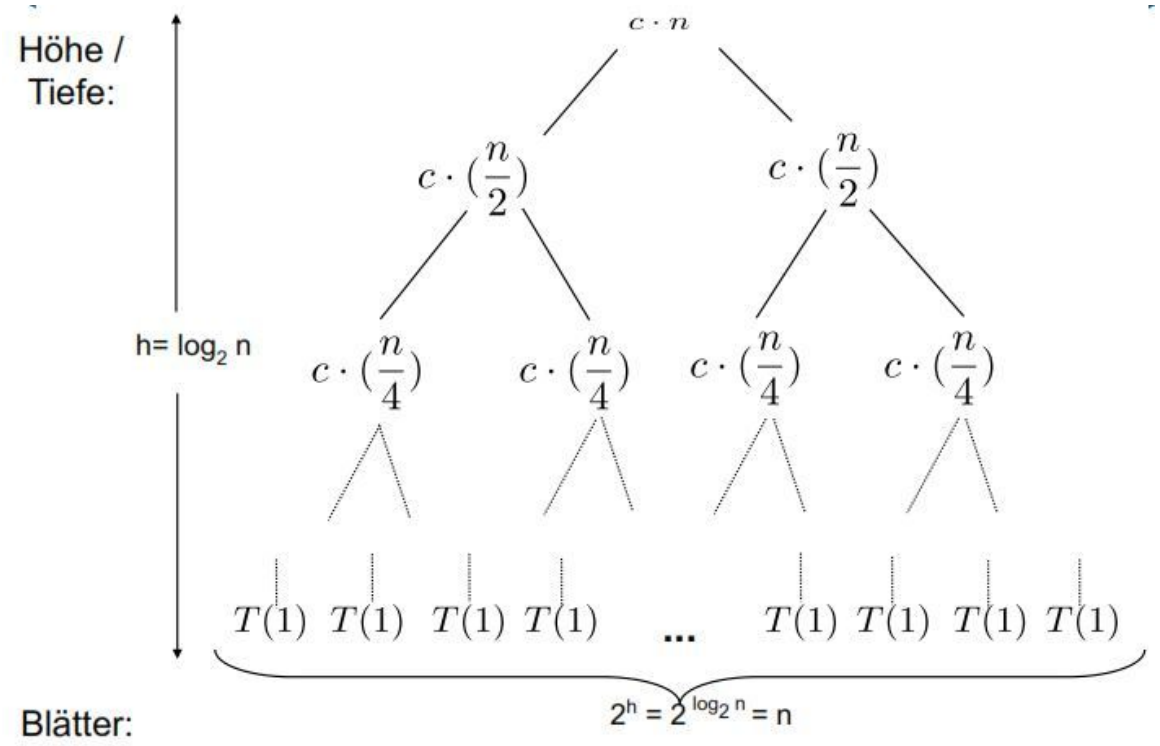

Abb. 5 Rekursionsbaum

1. Ebene $c \cdot n$

2. Ebene $2 \cdot \frac{1}{2} \cdot c \cdot n$

3. Ebene $4 \cdot \frac{1}{4} \cdot c \cdot n$

....

n. Ebene $\Theta(1) \cdot n = \Theta(n)$

Der Aufwand berechnet sich nun wie folgt:

$T(n) = c \cdot n + 2 \cdot \frac{1}{2} \cdot c \cdot n + 4 \cdot \frac{1}{4} \cdot c \cdot n + 2^{log_2 n - 1} \cdot \frac{1}{2^{log_2 n - 1}} \cdot c \cdot n + \Theta(n)$

$$= \sum_{i=0}^{log_2 n - 1} c \cdot n + \Theta(n)$$

$$= c \cdot n \cdot \sum_{i=0}^{log_2 n - 1} 1 + \Theta(n)$$

$$= c \cdot n \cdot log_2 n + \Theta(n)$$

$$= \Theta(n \cdot log_2 n) + \Theta(n) = \Theta(n \cdot log_2 n)$$

Allgemein bestimmt sich der Aufwand T(n) durch die Summe des Aufwands je Ebene und des Aufwands der Blattebene.

Bezogen auf den gegebenen Rekursionsbaum wäre das $T(n) = 3 \cdot T(n/4) + c \cdot n^2$

41.3 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 8.3 zu finden.

1 `http://www.dpunkt.de/buecher/3358.html`

42 Entwurfsprinzipien

Auf dieser Seite werden wir und mit Entwurfsprinzipien und einer Einführung in die Entwurfsmuster[1] beschäftigen. Die Ableitung eines optimalen Algorithmus aus Anforderungsbeschreibungen ist nicht automatisierbar. Der Algorithmenentwurf ist eine kreative Tätigkeit, die durch Muster (best practices) unterstützt wird. Vergleichbar ist das mit Mustern von Gebäuden in der Architektur oder mit Mustern aus der Softwarearchitektur.

42.1 Schrittweise Verfeinerung

Der Entwurf von Algorithmen erfolgt nach dem Prinzip der schrittweisen Verfeinerung von Pseudo Code Algorithmen. Pseudo Code Teile werden im ersten Schritt durch verfeinerten Pseudo Code ersetzt und im nächsten Schritt durch Programmiersprachen Code.

42.1.1 Beispiel 1

1. Pellkartoffeln kochen

verfeinert zu :

1.1 Fülle Topf mit Kartoffeln

1.2 Füge Wasser dazu

1.3 Stelle topf auf Herdplatte

1.4 Stelle Drehknopf auf 7

1.5 Koche das Wasser

42.1.2 Beispiel 2

Wir benutzen die Fakultät als Prozeduraufruf

```
Factorial(n)
```

Nun schreiben wir die Fakultät als Algorithmus

1 https://de.wikipedia.org/wiki/Entwurfsmuster

```
Fac: var X;Y:int;
input X;
Y:=1;
while X>1 do Y:=Y*X; X:=  X-1 od
output Y
```

Nun schreiben wir die Fakultät als Implementierungscode

```
public static int factorial (int x) {
...
}
```

42.2 Einsatz von Algorithmenmustern

Die Idee ist, dass generische Algorithmenmuster für bestimmte Problemklassen an eine konkrete Aufgabe angepasst werden. Das Lösungsverfahrens wird am Beispiel eines einfachen Vertreters der Problemklasse dokumentiert. Es wird eine Bibliothek von Mustern (Design Pattern) zur Ableitung eines abstrakten Programmrahmens benutzt. Durch parametrisierte Algorithmen und Vererbung wird die Programmiersprache unterstützt.

43 Greedyalgorithmus

Auf dieser Seite wird der Greedyalgorithmus[1] behandelt.

Greedy bedeutet ”gierig”. Der Algorithmus erfolgt nach dem Prinzip, dass versucht wird mit jedem Teilschritt so viel wie möglich zu erreichen. Greedy-Algorithmen (gierige Algorithmen) zeichnen sich dadurch aus, dass sie immer denjenigen Folgezustand auswählen, der zum Zeitpunkt der Wahl den größten Gewinn bzw. das beste Ergebnis verspricht.

43.1 Lokales Optimum

Der Greedy Algorithmus berechnet im jedem Schritt das lokale Optimum, dabei kann jedoch das globale Optimum verfehlt werden.

Jedoch entspricht in vielen Fällen das lokale Optimum auch dem globalem Optimum, bzw. es reicht ein lokales Optimum aus.

43.2 Problemklasse

1. Gegebene Menge von Eingabewerten
2. Menge von Lösungen, die aus Eingabewerten aufgebaut sind
3. Lösungen lassen sich schrittweise aus partiellen Lösungen, beginnend bei der leeren Lösung, durch Hinzunahme von Eingabewerten aufbauen. Alternativ: bei einer ganzen Menge beginnend schrittweise jeweils ein Element wegnehmen
4. Bewertungsfunktion für partielle und vollständige Lösungen
5. Gesucht wird die/eine optimale Lösung

1 `https://de.wikipedia.org/wiki/Greedy-Algorithmus`

44 Das Münzwechselproblem

Auf dieser Seite wird das Münzwechselproblem behandelt.

44.1 Beispiel

Als Beispiel nehmen wir die Herausgabe von Wechselgeld auf Beträge unter 1€. Verfügbar sind die Münzen mit den Werten 50ct, 10ct, 5ct, 2ct, 1ct. Unser Ziel ist, so wenig Münzen wie möglich in das Portemonnaie zu bekommen.

Ein Beispiel: $78ct = 50 + 2 \cdot 10 + 5 + 2 + 1$

Es wird jeweils immer die größte Münze unter dem Zielwert genommen und von diesem abgezogen. Das wird so lange durchgeführt, bis der Zielwert Null ist.

44.2 Formalisierung

Gesucht ist ein Algorithmus der folgende Eigenschaften beschreibt.

Bei der *Eingabe* muss gelten

1. dass die eingegebene Zahl eine natürliche Zahl ist, also $amount > 0$
2. dass eine Menge von Münzwerten zur Verfügung steht $currency = \{c_1, ..., c_n\}$ z.B.$\{1, 2, 5, 10, 20, 50\}$

Die *Ausgabe* besteht dann aus ganzen Zahlen $change[1], ..., change[n]$. Dabei ist $change[i]$ die Anzahl der Münzen des Münzwertes für c_i für $i = 1, ..., n$ und haben die Eigenschaften

1. $change[1] \cdot c_1 + ... + change[n] \cdot c_n = amount$
2. $change[1] + ... + change[n]$ ist minimal unter allen Lösungen für 1.

44.3 Algorithmus

1. Nehme jeweils immer die größte Münze unter dem Zielwert und ziehe sie von diesem ab.
2. Verfahre derart, bis der Zielwert gleich Null ist.

Der dazugehörige Code in Java:

```
public int[] moneyChange(int[] currency, int amount){
   int[] change = new int[currency.length];
   int currentCoin = currency.length-1;
   while(amount > 0){
      while(amount < currency[currentCoin] && currentCoin > 0)
         currentCoin--;
      if(amount >= currency[currentCoin] && currentCoin >= 0){
         amount -= currency[currentCoin];
         change[currentCoin]++;
      } else return null;
   }
   return change;
}
```

Die Methode moneyChange wird dabei aufgerufen durch:

```
int[] currency = {1,2,5,10,20,50};
int amount = 78;
int[] change = moneyChange(currency, amount);
```

44.4 Lokales Optimum

Der Greedy Algorithmus berechnet im jedem Schritt das lokale Optimum, dabei kann jedoch das globale Optimum verfehlt werden.

Beispiel: Münzen 11ct, 5ct und 1ct. Unser Zielwert ist 15ct. Nach Greedy benutzen wir 11+1+1+1+1, das Optimum wäre aber 5+5+5.

44.5 Analyse

44.5.1 Theorem

Für *currency* endlicher Länge und mit endlichen positiven Werten und endlichem positivem *amount*, terminiert der Algorithmus *moneyChange* nach endlich viele Schritten.

44.5.2 Beweis

- In Zeile 03 wird *currentCoin* mit einem endlichen positiven Wert initialisiert
- In Zeile 05 und 06 wird *currentCoin* nur dekrementiert, spätestens beim Wert 0 wird die Schleife beendet (also eine endliche Wiederholung)
- Falls die Zeilen 08 und 09 nicht ausgeführt werden, endet die Berechnung direkt in 10; andernfalls wird *amount* in Zeile 08 echt kleiner
- Irgendwann ist also der Bedingung in Zeile 04 nicht mehr gegeben und die Berechnung terminiert

44.5.3 Theorem

Für Eingabe *currency* mit $|currency| = m$ und $amount = n$ hat der Algorithmus *moneyChange* eine Laufzeit von O(m+n).

44.5.4 Beweis

- Die Zeile 6 wird maximal m-mal ausgeführt
- Die Zeile 8 wird maximal n-mal ausgeführt, falls es nur eine Münze mit dem Wert "1" gibt

44.5.5 Theorem

Der Algorithmus *moneyChange* löst für $currency = \{1, 2, 5, 10, 20, 50\}$ das Münzwechselproblem.

44.5.6 Beweis

Bei der Lösung musste zum Einen gelten, dass $change[1] \cdot c_1 + ... + change[n] \cdot c_n = amount$ ist. Da der Wert *amount* stets um den Wert einer Münze c_i verringert wird, während $change[i]$ um eins inkrementiert wird, ist dies erfüllt.

Die zweite Aussage zur Lösung war, dass $change[1] + ... + change[n]$ minimal unter allen Lösungen sein soll. Dies wird hier nur für Münzen vom Wert 1,2 und 5 betrachtet, wobei es für die Münzen 10, 20 und 50 analog zu beweisen ist.

Zunächst gilt, dass 2er-Münzen stets 1er-Münzen zu bevorzugen sind, da es keinen Sinn macht im Algorithmus auf eine 2er-Münze zu verzichten, um dann im nächsten Schritt mehr 1er-Münzen zu bekommen. Das bedeutet, dass eine optimale Lösung maximal eine 1er-Münze beinhaltet.

Weiterhin gilt, eine optimale Lösung hat nicht mehr als zwei 2er-Münzen. Sollten drei 2er-Münzen in der Lösung sein, ist es besser diese durch eine 1er und eine 5er-Münze zu ersetzen.

Des Weiteren gilt, eine optimale Lösung kann nicht gleichzeitig eine 1er-Münze und zwei 2er-Münzen enthalten, weil dies durch eine 5er-Münze dargestellt werden kann.

Es folgt, dass der durch 1er- und 2er-Münzen dargestellte Betrag nicht mehr als 4 sein kann. Also ist eine maximale Wahl von 5er-Münzen im Greedy-Verfahren optimal.

45 Divide and Conquer

Auf dieser Seite wird Divide and Conquer[1] behandelt. Divide and Conquer bedeutet "Teile und Herrsche". Quick Sort und Merge Sort sind typische Vertreter. Es verfolgt das Prinzip, dass auf identische Probleme mit einer kleinen Eingabemenge eine rekursive Rückführung geschieht.

45.1 Grundidee

Teile das gegebene Problem in mehrere getrennte Teilprobleme auf, löse diese einzeln und setze die Lösungen des ursprünglichen Problems aus den Teillösungen zusammen. Wende dieselbe Technik auf jedes der Teilprobleme an, dann auf deren Teilprobleme, usw, bis die Teilprobleme klein genug sind, dass man eine Lösung explizit angeben kann. Strebe an, dass jedes Teilproblem von derselben Art ist wie das ursprüngliche Problem, so dass es mit demselben Algorithmus gelöst werden kann.

45.2 Muster

```
procedure DIVANDCONQ (P: problem)
begin
        …
        if [P klein ]
        then [explizite Lösung ]
        else [ Teile P auf in P1, …, Pk ];
                DIVANDCONQ (P1 ) ;
                … ;
                DIVANDCONQ (Pk) ;
                [ Setze Lösung für P aus Lösungen für P1, …, Pk zusammen ]
        fi
end
```

45.3 Beispiel

Wir nehmen als Beispiel die Spielpläne für Turniere. Gegeben sind $n = 2^k$ Spieler, wobei k ganzzahlig und größer 0 ist. Des weiteren sind mindestens n-1 Turniertage gegeben und jeder Spieler spielt gegen jeden anderen. Der Turnierplan T_k ist bekannt und die Aufgabe ist T_{k+1} für $m = 2n = 2^{k+1}$ zu konstruieren (Rekursionsprinzip).

Spielplan für T_2

1 https://de.wikipedia.org/wiki/Teile_und_herrsche_%28Informatik%29

	Tag 1	Tag 2	Tag 3
Spieler 1	2	3	4
Spieler 2	1	4	3
Spieler 3	4	1	2
Spieler 4	3	2	1

Spielplan für T_1 Für kleine Problemgröße kann Lösung direkt angegeben werden:

	Tag 1
Spieler 1	2
Spieler 2	1

Nun wird aus $T_k T_{k+1}$ konstruiert.

	Tag 1...n-1	Tag n...m-1
Spieler 1... $n = 2^k$	T_k	S_k
n+1...$m = 2k+1$	$T_k^{[+n]}$	Z_k

1. $T_k^{[+n]} : T_k$ mit jeweils um n erhöhten Elementen
2. $Z_k : (n\ x\ n)$ Matrix, konstruiert durch zyklisches Verschieben der Zeile (1,2,..., n)
3. $S_k : (n\ x\ n)$ Matrix, konstruiert durch zyklisches Verschieben der Spalte (n+1,...,m) für $n = 2^k$ *und* $m = 2^{k+1}$

$$Z_2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{pmatrix}$$

$$S_2 = \begin{pmatrix} 5 & 8 & 7 & 6 \\ 6 & 5 & 8 & 7 \\ 7 & 6 & 5 & 8 \\ 8 & 7 & 6 & 5 \end{pmatrix}$$

Spielplan für T_3

	Tag 1	Tag 2	Tag 3	Tag 4	Tag 5	Tag 6	Tag 7
Spieler 1	2	3	4	5	8	7	6
Spieler 2	1	4	3	6	5	8	7
Spieler 3	4	1	2	7	6	5	8
Spieler 4	3	2	1	8	7	6	5
Spieler 5	6	7	8	1	2	3	4
Spieler 6	5	8	7	2	3	4	1
Spieler 7	8	5	6	3	4	1	2
Spieler 8	7	6	5	4	1	2	3

T_k = Spieler 1-4 Tag 1-3

S_k = Spieler 1-4 Tag 4-7

$T_k^{[+n]}$ = Spieler 5-8 Tag 1-3

Z_k = Spieler 5-8 Tag 4-7

45.4 Türme von Hanoi

Bei den Türmen von Hanoi sind 3 Stapel mit Scheiben unterschiedlicher Größe gegeben. In jedem Schritt darf eine Scheibe, die ganz oben auf einem Stapel liegt, auf einen anderen Stapel gelegt werden. Allerdings unter der Bedingung, dass sie nur auf eine kleinere Scheibe gelegt werden darf.

Das Ziel ist es alle Scheiben vom ganz linken Stapel auf den ganz rechten Stapel zu verschieben.

Illegale Spielzüge sind dabei, wenn eine größere Scheibe auf eine kleinere Scheibe gelegt wird.

45.4.1 Algorithmenentwurf

Reduziere das Problem n Scheiben zu verschieben darauf nur noch n-1 Scheiben zu verschieben, bis schlussendlich nur noch eine Scheibe übrig bleibt.

Dies ist ein ähnliches Prinzip wie bei Induktionsbeweisen. Dabei kann das Nutzen des Algorithmus für n-1 Scheiben als der Induktionsschritt gesehen werden. Der Basisfall, also der Induktionsanfang, ist dabei, wenn es nur eine Scheibe gibt.

Um die Aufgabe zu lösen, muss beim Verschieben von mehr als einer Scheibe der dritte Stapel immer als ”Zwischenlager” genutzt werden. Welcher der drei Stapel das ”Zwischenlager” ist, kann ja nach Schritt wechseln. In dem Beispiel bei 5 Scheiben auf dem linken Stapel(A), dient dieser als Startstapel und soll auf den linken Stapel(C), also auf den Zielstapel, verschoben werden. Im ersten Schritt dient der mittlere Stapel(B) somit als Zwischenlager.

Das erste Unterziel ist es die obersten vier Scheiben von A zu verschieben. Dafür dient A als Startstapel, B als Zielstapel und C als Zwischenlager.

Weiterhin muss gewährleistet sein, dass das Zwischenlager nach Abschluss wieder genauso aussieht wie zuvor.

Der Pseudocode sieht wie folgt aus:

```
Algorithmus hanoi
Eingabe:
   Startstapel S,
   Zielstapel Z,
   Zwischenlager L,
   Anzahl der Scheiben n
Ausgabe:
   Aktionenfolgen um alle Scheiben von S nach L zu verschieben

Falls n=1
   Entferne die oberste Scheibe k von S und füge sie Z hinzu
   Gib aus "Verschiebe k von S nach Z"
Ansonsten
```

```
   hanoi(S,L,Z,n-1);
   Entferne die oberste Scheibe k von S und füge sie Z hinzu
   Gib aus "Verschiebe k von S nach Z"
   hanoi(L,Z,S,n-1);
```

Für die Implementierung in Java wird als Repräsentation der Stapel je ein *Stack* und für eine Scheibe je ein *int* verwendet. Bei den Scheiben gibt der Wert jeweils die Größe der Scheibe an.

```
public void hanoi(Stack<Integer> start, Stack<Integer> goal, Stack<Integer> tmp,
 int numDiscs){
   if(numDiscs == 1){
      int disc = start.pop();
      goal.push(disc);
      System.out.println("Moving disc " + disc);
   }else{
      hanoi(start, tmp, goal, numDiscs-1);
      int disc = start.pop();
      goal.push(disc);
      System.out.println("Moving disc " + disc);
      hanoi(tmp, goal, start, numDiscs-1);
   }
}
```

Der Aufruf erfolgt durch:

```
Stack<Integer> start = new Stack<Integer>();
for(int i = 5; i > 0; i--) start.push(i);
hanoi(start, new Stack<Integer>(), new Stack<Integer>(), 5);
```

45.4.2 Analyse

Theorem

Der Algorithmus *hanoi* terminiert nach endlich vielen Schritten, wenn die Anzahl der Scheiben positiv ist.

Beweis

- Die Zeile 03 stellt das Rekursionsende da und der Algorithmus terminiert bei numDiscs $= 1$
- Die Else-Bedingung führt dazu, dass durch die rekursiven Aufrufe der Wert von numDiscs sich immer um 1 verringert.

Theorem

Für n Schreiben hat der Algortihmus *hanoi* eine Laufzeit von $O(2^n)$.

Beweis

Die Rekursionsgleichung für *hanoi* ist:

$$T(n) := \begin{cases} 1 & \text{falls } n = 1 \\ 2 \cdot T(n-1) + O(1) & sonst \end{cases}$$

Der Beweis erfolgt damit durch Induktion.

Theorem

Der Algorithmus *hanoi* löst das Problem der Türme von Hanoi.

Beweis

Zu zeigen gelten:

1. Der Algorithmus hält sich an die Spielregeln, dass in jedem Zug nur eine Scheibe von oben von einem Stapel entfernt werden darf und auf einen leeren Stapel oder auf eine größere Scheibe gelegt wird.
2. Bei der Terminierung sind alle Scheiben auf dem Zielstapel.

Beide Aussagen kann man durch Induktion nach der Anzahl der Scheiben n beweisen.

Für $n = 1$: hier wird eine Scheibe direkt vom Startstapel zum leeren Zielstapel verschoben. In diesem Fall sind beide Bedingungen erfüllt.

Für $n - 1 \rightarrow n$: Es sind n Scheiben von Stapel A zu C zu verschieben. Dazu sei B der dritte Stapel. Zunächst wird der Algorithmus rekursiv für die obersten n-1 Scheiben mit Zielstapel B aufgerufen. Da alle diese n-1 Scheiben kleiner als die unterste Scheibe ist und diese nicht bewegt wird, ist dies das gleiche Problem, als wenn die unterste Scheibe gar nicht da wäre. Nach rekursiven Aufruf werden also die n-1 Scheiben legal nach B verschoben. C ist dabei anschließend wieder leer. Die Verschiebung der untersten Scheibe nach C ist legal. Die rekursive Verschiebung der auf B liegenden n-1 Scheiben nach C ist nun wieder legal, aufgrund der Tatsache, dass auf C nur eine größere Scheibe liegt.

Der Algorithmus ist ebenso optimal, das heißt, er findet eine minimale Anzahl von Zügen zur Lösung des Problems.

Weiterhin ist eine Animation des Algorithmus[2] verfügbar.

2 `http://britton.disted.camosun.bc.ca/hanoi.swf`

46 Backtracking

Auf dieser Seite wird das Backtracking[1] behandelt.

Die Idee des Backtracking ist das Versuchs-und-Irrtum-Prinzip (trial and error). Versuche, die erreichte Teillösung schrittweise zu einer Gesamtlösung auszubauen. Falls die Teillösung nicht zu einer Lösung führen kann, dann nimm den letzten Schritt bzw. die letzten Schritte zurück und probiere stattdessen alternative Wege.Alle in Frage kommenden Lösungswege werden ausprobiert. Vorhandene Lösung wird entweder gefunden (unter Umständen nach sehr langer Laufzeit) oder es existiert definitiv keine Lösung. Backtracking ("Zurückverfolgen") ist eine allgemeine systematische Suchtechnik. KF ist die Menge von Konfigurationen. K_0 ist die Anfangskonfiguration. Für jede Konfiguration K_i gibt es eine direkte Erweiterung $K_{i,1}, ..., K_{i,n_i}$. Außerdem ist für jede Konfiguration entscheidbar, ob sie eine Lösung ist. Aufgerufen wird Backtracking mit $BACKTRACK(K_0)$.

46.1 Labyrinth Suche

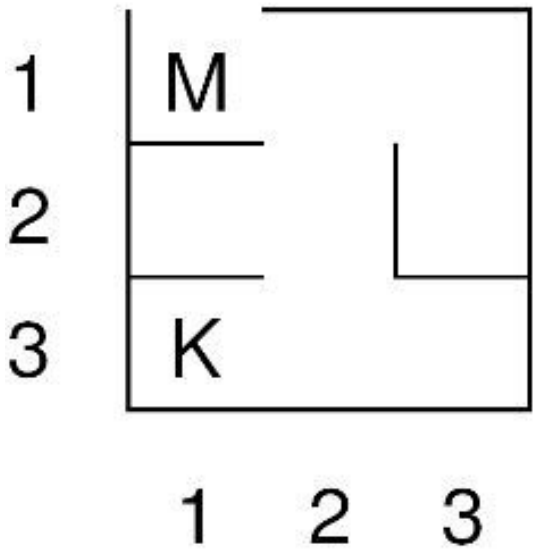

Abb. 6 Backtracking1

1 https://de.wikipedia.org/wiki/Backtracking

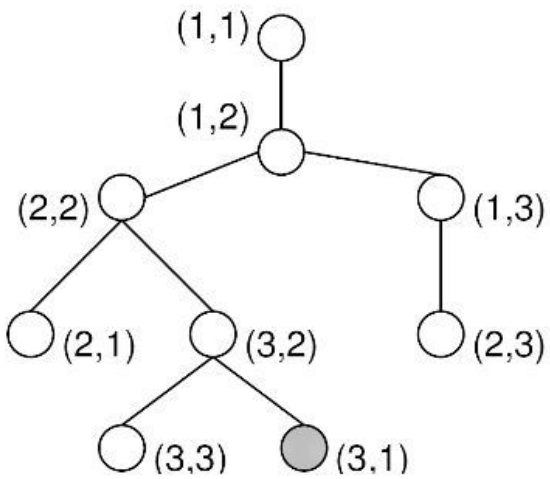

Abb. 7 Backtracking2

46.2 Backtracking Muster

```
procedure BACKTRACK (K: Konfiguration)
begin
        …
        if [ K ist Lösung ]
        then [ gib K aus ]
        else
                for each [ jede direkte Erweiterung KO von K ]
                        do
                                BACKTRACK (KO)
                        od
        fi
end
```

46.3 Einsatzfelder

Zu den typischen Einsatzfeldern von Backtracking gehören zum Beispiel einige Spielprogramme (Schach, Dame, Labyrinthsuche,...). Aber auch die Erfüllbarkeit von logischen Aussagen wie logische Programmiersprachen, Optimierung von Gattern oder Model checking (Theorembeweiser). Ein weiteres Einsatzfeld sind Planungsprobleme und Konfigurationen wie logistische Fragestellungen (Traveling Salesman, der kürzeste Wege, die optimale Verteilung, das Färben von Landkarten oder auch nichtdeterministisch-lösbare Probleme.

46.3.1 Beispiel Acht Damen Problem

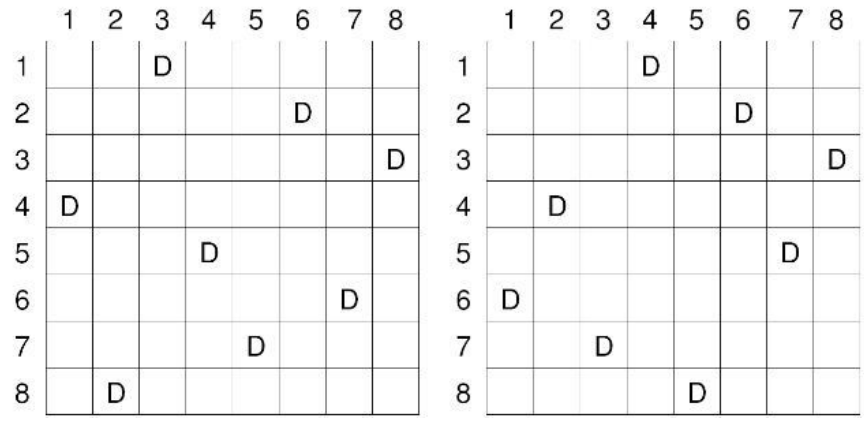

Abb. 8 Acht Damen Problem

Gesucht sind alle Konfigurationen von 8 Damen auf einem 8 x 8-Schachbrett, so dass keine Dame eine andere bedroht. Gesucht ist nun ein geeignetes KF. Für jede Lösungskonfigurationen L mit gelten $L \subseteq KF$. Für jedes $k \in KF$ ist leicht entscheidbar, ob $k \in L$. Die Konfigurationen lassen sich schrittweise erweitern und wir erhalten eine hierarchische Struktur. Es sollte auch beachtet werden, dass KF nicht zu groß sein sollte.

L_1: Es sind 8 Damen auf dem Brett

L_2: Keine zwei Damen bedrohen sich.

KF wird so gewählt, dass die Konfiguration mit je einer Dame in den ersten n Zeilen, $0 \leq n \leq 8$, so dass diese sich nicht bedrohen.

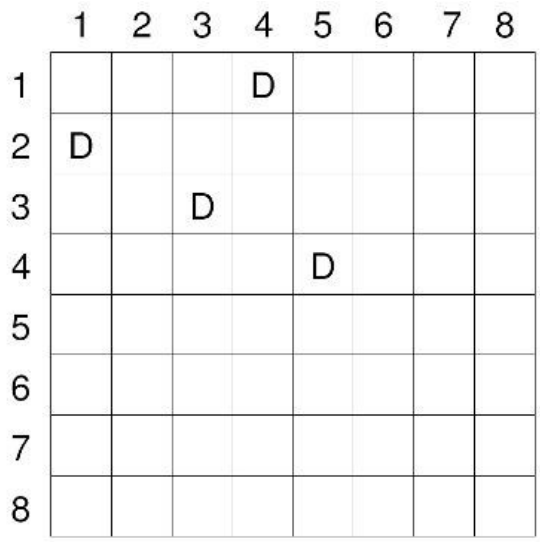

Abb. 9 Acht Damen Problem2

Diese Konfiguration ist nun nicht mehr erweiterbar. Jedes Feld in Zeile 7 ist bereits bedroht.

```
procedure PLATZIERE (i:[1..8]);
begin
   var h: [1..8];
   for h:=1 to 8 do
      if [Feld in Zeile i, Spalte h nicht bedroht]
      then
         [Setze Dame auf dieses Feld (i,h)];
         if [Brett voll] /* i=8*/
         then [Gib Konfiguration aus ]
         else PLATZIERE (i+1)
         fi
      fi
   od
end
```

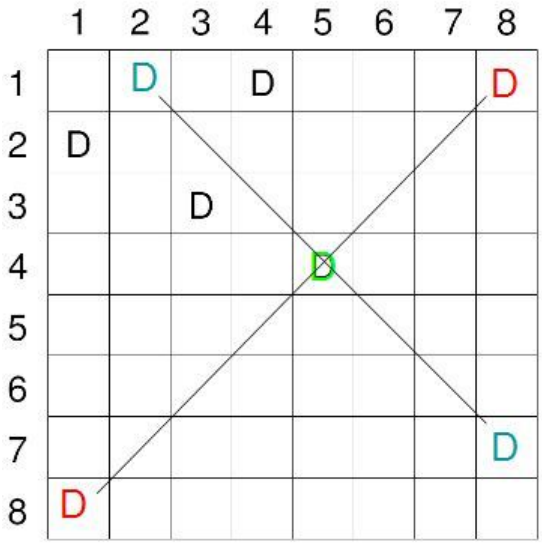

Abb. 10 Acht Damen Problem4

Die Array Repräsentation ist [4,1,3,5,0,0,0,0]. Die Diagonalen sind belegt wenn:

i+h = i+h

i-h = i-h

(i = Spalte, h = Zeile)

Die Zeilen snd belegt, wenn die Position im Array besetzt ist und die Spalten sind belegt, wenn die Nummer im Array existiert.

Der initiale Aufruf geschieht mir Platziere(1). Es gibt insgesamt 92 Lösungen. Die Konfigurationen ist etwa als zweidimensionales boolesches Array oder als eindimensionales Array mit einer Damenposition pro Zeile realisierbar. Redundante Informationen über bedrohte Spalten und Diagonalen bieten Optimierungspotential.

46.3.2 Algorithmus in Java

Der Code zu dem Problem im allgemeinen Fall sieht in Java wie folgt aus.

```
public boolean isValid(int[] board, int current, int place){
   for(int i = 0; i < current-1; i++){
      if(board[i] == place) return false;
      if(place+current == board[i] + (i+1)) return false;
      if(place-current == board[i] - (i+1)) return false;
   }
   return true;
}

public int[] placeQueen(int[] board, int current){
   int[] tmp;
   for(int i=0; i< board.length; i++){
      if(isValid(board, current, i)){
         board[current-1] = i;
         if(current == board.length) return board;
         else{
            tmp = placeQueen(board, current+1);
            if(tmp != null) return tmp;
         }
      }
   }
   return null;
}
```

Aufgerufen wird der Code durch:

```
int[] result = placeQueen(new int[8], 1);
```

46.3.3 Analyse

Theorem

Der Algorithmus *placeQueen* terminiert nach endlich vielen Schritten, wenn die Anzahl der Felder positiv ist.

Beweis

Die Methode $isValid$ terminiert offensichtlich immer.

In *placeQueen* wird rekursiv *placeQueen* stets um einen erhöhten Parameter *current* aufgerufen. Die for-Schleife hat auch stets eine konstante Zahl an Durchgängen.

Theorem

Für ein Feld der Größe n x n hat der Algorithmus *placeQueen* eine Laufzeit von $O(n^n)$.

Beweis

Im schlimmsten Fall werden alle Konfigurationen betrachtet:

- n Positionen für eine einzelne Dame
- n Damen sind zu plazieren

Die tatsächliche Laufzeit ist weitaus geringer, da viele Konfigurationen schon früh als nicht-erweiterbar erkannt werden. Dennoch ist die Laufzeit im schlimmsten Fall exponentiell $O(2^n)$.

Theorem

Der Algorithmus *placeQueen* löst das n-Damenproblem.

47 Dynamische Programmierung

Auf dieser Seite wird die dynamische Programmierung[1] behandelt.

Die dynamische Programmierung vereint die Ideen verschiedener Muster. Zum einen die Wahl der optimalen Teillösung des Greedy Musters und zum anderen die Rekursion und den Konfigurationsbaum aus Divide and Conquer und Backtracking. Die Unterschiede sind, dass Divide and Conquer unabhängige Teilprobleme löst und in der dynamischen Programmierung eine Optimierung von abhängigen Teilproblemen durchgeführt wird. Die dynamische Programmierung ist eine „bottom-up"-Realisierung der Backtracking-Strategie. Die Anwendungsbereiche sind die selben wie bei Greedy, jedoch wird dynamische Programmierung insbesondere dort angewandt, wo Greedy nur suboptimale Lösungen liefert.

47.1 Idee

Bei der dynamischen Programmierung werden kleinere Teilprobleme zuerst gelöst, um aus diesen größere Teillösungen zusammenzusetzen. Das Problemlösen geschieht quasi auf Vorrat. Es werden möglichst nur die Teilprobleme gelöst, die bei der Lösung der großen Probleme auch tatsächlich benötigt werden. Wir erzielen einen Gewinn, wenn identische Teilprobleme in mehreren Lösungszweigen betrachtet werden. Rekursives Problemlösen wird ersetzt durch Iteration und abgespeicherte Teilergebnisse.

Nicht immer ist es überhaupt möglich, die Lösungen kleinerer Probleme so zu kombinieren, dass sich die Lösung eines größeren Problems ergibt. Die Anzahl der zu lösenden Probleme kann unvertretbar groß werden. Es können zu viele Teillösungen entstehen, die dann doch nicht benötigt werden oder der Gewinn der Wiederverwendung ist zu gering, da die Lösungszweige disjunkt sind.

1 `https://de.wikipedia.org/wiki/Dynamische_Programmierung`

48 Beispiel Editierdistanz

Gegeben sind zwei Zeichenketten s und t, was ist die minimale Anzahl an Einfüge-, Lösch- und Ersetzoperationen um s in t zu transformieren?

Als Beispiel entspricht s "Haus" und t "Maus". Die Lösung ist hier, dass "H" durch "M" ersetzt wird. Bei s= "Katze" und t="Glatze" wird "K" durch "G" ersetzt und "l" hinzugefügt. Die Editierdistanz kommt in der Rechtschreibprüfung und Plagiatserkennung zur Anwendung.

48.1 Formalisierung

Definition (Ein-Schritt Modifikation)

Beachte $s = s_1...s_m$

1. Jedes $s' = s_1...s_{i-1}s_{i+1}....s_m$ (für $i = 1,...,m$)
2. Jedes $s' = s_1...s_{i-1}xs_{i+1}....s_m$ (für $i = 1,...,m$ und $x! = s_i$)
3. Jedes $s' = s_1...s_ixs_{i+1}....s_m$ (für $i = 0,1,...,m$ und $bel.$ x)

heißt Ein-Schritt Modifikation von s.

Definition (k-Schritt Modifikation) Eine Zeichenkette t heißt k-Schritt Modifikation ($k > 1$) von s, wenn es Zeichenketten u gibt mit:

1. u ist eine Ein-Schritt Modifikation von s
2. t ist eine k-1-Schritt Modifikation von u

Definition (Editierdistanz, auch Levenshtein-Distanz) $D(s,t) = min\{d|s$ ist $eine$ $d-Schritt$ $Modifikation$ von $t\}$

Ist s eine d-Schritt Modifikation von t, so ist auch s eine d+2j Modifikation von t für jedes j>0.Eine minimale Modifikation muss nicht eindeutig sein. Wir sind aber hier nur an dem Wert einer minimalen Modifikation interessiert.

48.2 Charakterisierung und Algorithmus

Die Idee ist, dass die Berechnung von D(s,t) auf die Berechnung von D auf die Präfixe von s und t zurückgeführt wird.

Definition $D_{ij}(s,t)$

Sei $s = s_1...s_m$ und $t = t_1...t_n$

Definiere $D_{ij}(s,t) = D(s_1...s_i, t_1...t_j)(fueri = 0,...,m, j = 0,...,n)$

Beachte für z.B i=0 haben wir $s_1...s_i = \epsilon$ (leerer String).

Wir beobachten, dass gilt $D_{mn}(s,t) = D(s,t)$. Dies ist nun zu berechnen. Zudem ist $D_{00}(s,t) = D(\epsilon,\epsilon) = 0$, also sind zwei leere Strings identisch.

$D_{0j}(s,t) = D(\epsilon,t) = j$ für j=1,..,n. Also alle Zeichen $t_1...t_j$ müssen eingefügt werden.

$D_{i0}(s,t) = D(s,\epsilon) = i$ für i=1,...,m. Also alle Zeichen $s_1...s_i$ müssen eingefügt werden.

48.2.1 Theorem der zentralen Charakterisierung der Editierdistanz

Falls $s_i = t_j : D_{ij}(s,t) = D_{i-1,j-1}(s,t)$.

Ansonsten: $D_{ij}(s,t) = min := \begin{cases} D_{i-1,j-1}(s,t)+1 & Ersetzung \\ D_{i,j-1}(s,t)+1 & Einfuegung \\ D_{i-1,j}(s,t)+1 & Loeschung \end{cases}$

48.2.2 Algorithmus

```
For j=0,...,n set D_0j(s,t)=j
For i=0,...,m set D_i0(s,t)=i
For i=1,..,m
   For j=1,...,n
      If s_i = t_j set D_ij(s,t) = D_{i-1,j-1}(s,t)
      else D_ij(s,t) =
         min {D_{i-1,j-1}(s,t)+1, D_{i,j-1}(s,t), D_{i-1,j}(s,t)+1 }
Return D_mn(s,t)
```

48.2.3 Analyse

Theorem

Für endliche Zeichenketten s und t terminiert der Algorithmus editdistance nach endlich vielen Schritten.

Beweis

Der Beweis folgt auf dem nächsten Theorem.

Theorem

Für die Eingaben $s = s_1...s_m$ und $t = t_1...t_n$ hat der Algorithmus eine Laufzeit von $\Theta(mn)$.

Beweis

Der Beweis besteht aus einer einfachen Schleifenanalyse.

Theorem

Der Algorithmus editdistance berechnet die Editierdistanz zweier Zeichenketten s und t.

Beweis

Der Beweis folgt direkt aus der zentralen Charakterisierung der Editierdistanz.

49 Einleitung Suchen

In diesem Kapitel geben wir einen Überblick über das Thema Suchen[1]. Suchprobleme sind eine der häufigsten Probleme in der Informatik. Man kann in sortierten Folgen suchen, Zeichenketten im Text suchen, Dokumente in Textkorpora suchen, oder allgemeine Lösungen von Problemräumen, wie der Spielbaumsuche oder der Plansuche, suchen. Hier behandeln wir zunächst die Suche in sortierten Folgen.

49.1 Motivation

Beim Suchen wiederholt man häufig sehr nützliche Beispielalgorithmen, oder lernt diese sogar neu kennen. Außerdem dient es der Vorbereitung der theoretischen Betrachtungen zur Komplexität von Algorithmen. Des weiteren dient es der informellen Diskussion von Entwurfsentscheidungen.

49.2 Suchen in sortierten Folgen

- Annahme:

  Die Folge F ist ein Feld mit numerischen Werten. Dazu ist die Folge sortiert, das heißt, wenn i<j, dann ist F[i]<F[j]. Auf das i-te Element hat man Zugriff über F[i]. Es wird nur der Suchschlüssel berücksichtigt.

Ein Beispiel ist ein Telefonbuch, in dem wir nach Namen suchen möchten. Doch wie repräsentiert man diese Daten?

49.2.1 Einschub lineare Datenstrukturen

Definition

Eine lineare Datenstruktur L ist eine Sequenz $L = (a_1 ..., a_n)$. Die lineare Datenstruktur ordnet Elemente (entweder primitive Datentypen oder komplexere Datenstrukturen) in einer linearen Anordnung an.

1 `https://de.wikipedia.org/wiki/Suchverfahren`

Beispiel

Zahlenfolgen

5	4	6	1	3	2

Strings

L	I	N	E	A	R

Atomare Operationen

Zu den Operationen gehören Lesen mit

- get(i): Element an Position i lesen
- first(): erstes Element lesen
- last(): letztes Element lesen
- next(e): Element nach Element e lesen

und Schreiben mit

- set(i,e): Element an Position i auf e setzen
- add(i,e): Element e an Position i einfügen
- del(i): Element an Position i löschen

Arrays und Listen

Es gibt zwei Möglichkeiten lineare Datenstrukturen zu realisieren. Entweder Arrays oder (verlinkte) Listen. Arrays belegen einen zusammenhängenden Bereich im Speicher. Elemente einer verlinkten Liste können beliebig verteilt sein. Ob zur Realisierung einer linearen Datenstruktur ein Array oder eine Liste verwendet wird, hängt von der Anwendung ab. Arrays werden meist für statische Datenstrukturen verwendet, d.h. wenn die Länge des Arrays nicht verändert wird. Listen werden meist für dynamische Datenstrukturen verwendet, d.h. wenn die Länge variabel ist. Zu den positiven Eigenschaften von Arrays zählen der schneller Zugriff auf Einzelelemente durch den Index. Zu den negativen Eigenschaften von Arrays zählen das sehr aufwändige Einfügen der Elemente. Zu den positiven Eigenschaften von Listen zählen die relativ effiziente Manipulation, zu den negativen Eigenschaften der ineffiziente Direktzugriff.

Einfache verlinkte Liste von Zahlen in Java

```
public class IntegerList {
  private class IntegerListElement{
   int value;
   IntegerListElement next;
  }

  IntegerListElement first;
  int size = 0;
     private IntegerListElement getElement(int i){
   if(i+1 > size)
    throw new IllegalArgumentExcepHon();
   int idx = 0;
```

```
  IntegerListElement current = first;
  while(idx != i){
   current = current.next;
   idx ++;
  }
  return current;
 }
 public int get(int i){
  return this.getElement(i).value;
 }
public int add(int pos, int val){
  IntegerListElement newElem = new IntegerListElement();
  newElem.value = val;
  if(pos > 0){
   IntegerListElement e = this.getElement(pos-1);
   newElem.next = e.next;
   e.next = newElem;
  }else{
   newElem.next = this.first;
   this.first = newElem;
  }
```

Suchen und Sortieren

Suchen und Sortieren sind voneinander abhängige Operationen. Dabei gibt es zwei Ansätze: Wenn Elemente nie sortiert sind, dann ist die Suche sehr aufwändig. Wenn die Elemente sortiert sind, wird die Suche erleichtert, jedoch kann das Sortieren an sich sehr aufwändig sein. Wenn Elemente hinzugefügt oder gelöscht werden ist diese Problematik noch sichtbarer. Nur ein unsortiertes Element macht die Suche aufwändig, doch bei jeder Einfügung oder Löschung zu sortieren ist ebenfalls sehr aufwändig. Spezielle dynamische Datenstrukturen erlauben eine automatische und effiziente Sortierung bei Einfügung oder Löschung.

Lineare Datenstruktur in Java

- Arrays:

  int[] arr = new int[10];

  arr[1] = 4;

- Listen:

  List<Integer> myList = new LinkedList<Integer>();

  myList.add(5);

- Neben LinkedList unterstützt Java eine Reihe weiterer Listenimplementierungen mit unterschiedlichen Vor- und Nachteilen
- Schnittstelle List<Type> beinhaltet die gemeinsamen Methoden

49.2.2 Suche

- Problembeschreibung

Die Eingabe ist eine Folge F mit n Elementen von Zahlen und Suchelementen k. Die Ausgabe ist eine erfolgreiche oder nicht erfolgreiche Suche. Erfolgreich ist sie, wenn der Index p ($0 \leq p < n$) ist. Eventuell muss man festlegen, was bei Mehrfachvorkommen passiert. Normalerweise gilt dann das erste Vorkommen. Ist die Suche nicht erfolgreich, dann ist die Ausgabe -1.

- Merkmale der Suche

  Es gibt immer einen Suchschlüssel für Suchelemente, z.B. Zahlen. Außerdem ist eine Suche immer erfolgreich oder erfolglos. Die Suche basiert auf Vergleichsoperationen und die Daten sind zunächst als Feld, bzw. Array, oder Liste dargestellt.

49.2.3 Suchverfahren

In den nachfolgenden Kapiteln lernen Sie die sequentielle, binäre und Fibonacci Suche kennen.

49.3 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5 zu finden.

2 `http://www.dpunkt.de/buecher/3358.html`

50 Sequentielle Suche

Dieses Kapitel handelt von der sequentiellen Suche[1]. Die Idee dieses Suchalgorithmus ist, dass zuerst das erste Elemente der Liste mit dem gesuchten Elemente verglichen wird, wenn sie übereinstimmen wird der aktuelle Index zurückgegeben. Wenn nicht wird der Schritt mit dem nächsten Element wiederholt. Sollte das gesuchte Element bis zum Ende der Folge nicht gefunden werden, war die Suche erfolglos und -1 wird zurückgegeben.

50.1 Algorithmus

```
int SeqSearch(int[] F, int k) {
   / * output: Position p  (0 ⊠ p ⊠ n-1) */
}
 int n = F.length;
 for (int i = 0; i < n; i++) {
    if (F[i] == k) {
          return i;
 }
 return -1;  }
```

Dabei ist Int[] die sortierte Folge von int, int k der Suchschlüssel und die Folge F hat die Länge n.

50.2 Aufwands Analyse

Das Terminierungs-Theorem besagt, dass der Algorithmus SeqSearch für eine endliche Eingabe nach endlicher Zeit terminiert. Das Korrektheits-Theorem besagt, falls das Array F ein Element k enthält, gibt SeqSearch(F,k) den Indes des ersten Vorkommens von k zurück. Ansonsten gibt SeqSearch(F,k) den Wert -1 zurück Im besten Fall beträgt die Anzahl der Vergleiche 1, das heißt direkt bei dem ersten Suchdurchlauf wird der Suchschlüssel gefunden. Im schlechtesten Fall beträgt die Anzahl der Vergleiche n, das heißt im letzten Suchdurchlauf wird der Suchschlüssel gefunden. Der Durchschnitt bei einer erfolgreichen Suche beträgt (n+1)/2 und der Durchschnitt einer erfolglosen Suche n. Die Folgen müssen nicht sortiert sein. Der Algorithmus SeqSearch hat also eine Worst-Case Laufzeit von $\Theta(n)$.

1 https://de.wikipedia.org/wiki/Sequentielle_Suche

50.3 Sequentielle Suche in Java

```
public class SequentialSearch{
     public final static int NO_KEY = -1;

static int SeqSearch(int[] F, int k) {
   for (int i = 0; i < F.length; i++)
   if (F[i] == k)
         return i;
   return NO_KEY;
}
 public static void main(String[ ] args){
    if (args.length != 1) {
           System.out.println(''usage: SequentialSearch
             <key>'');
    return;
    }

    int[ ] f = {2, 4, 5, 6, 7, 8, 9, 11};
    int k = Integer.parseInt(args[0]);
    System.out.println(''Sequentiell:''+seqSearch(f,k));
 }
}
```

In der Klasse SeqSearch ist eine Konstante NO_KEY definiert, die als Ergebnis zurückgegeben wird, wenn der gesuchte Wert nicht im Feld gefunden wurde. Die Methode search wird schließlich in der Klassenmethode main aufgerufen, um das Feld f nach dem Schlüsselwert k zu durchsuchen. Dieser Wert ist als Parameter beim Programmaufruf anzugeben. Da die Programmparameter als Feld args von Zeichenketten übergeben werden, ist zuvor noch eine Konvertierung in einen int-Wert mit Hilfe der Methode parseInt der Klasse java.lang.Integer vorzunehmen. Somit bedeutet der Programmaufruf "java SeqSearch 4" die Suche nach dem Wert 4 in der gegebenen Folge. Der Aufruf erfolgt mit java SequentialSearch 4

50.4 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5.1.1 zu finden.

2 http://www.dpunkt.de/buecher/3358.html

51 Binäre Suche

Dieses Kapitel behandelt die binäre Suche[1]. Wir stellen uns die Frage, wie die Suche effizienter werden könnte. Das Prinzip der binären Suche ist zuerst den mittleren Eintrag zu wählen und zu prüfen ob sich der gesuchte Wert in der linken oder rechten Hälfte der Liste befindet. Anschließend fährt man rekursiv mit der Hälfte fort, in der sich der Eintrag befindet. Voraussetzung für das binäre Suchverfahren ist, dass die Folge sortiert ist. Das Suchverfahren entspricht dem Entwurfsmuster von Divide-and-Conquer.

51.1 Beispiel

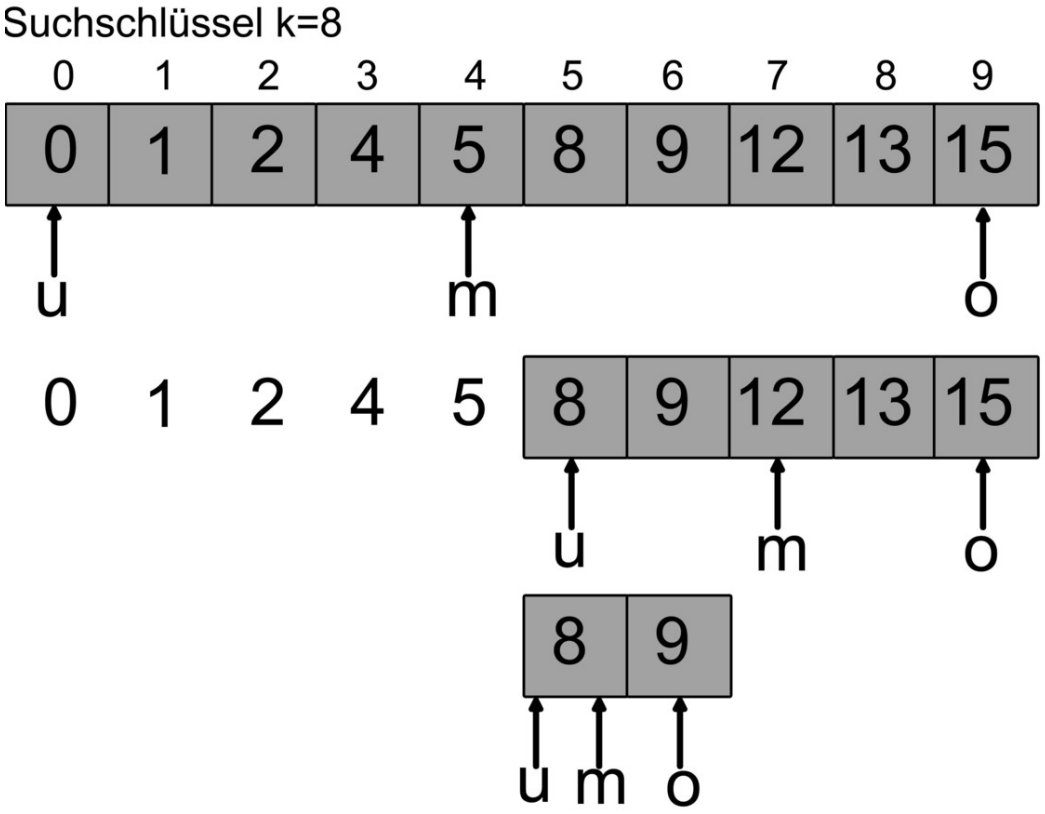

Abb. 11 Binäre Suche

1 https://de.wikipedia.org/wiki/Bin%C3%A4re_Suche

51.2 Rekursiver Algorithmus

```
int BinarySearch(int[] F, int k){
  /*input: Folge F der Länge n, Schlüssel k */
  /*output: Position p */
amk
   return  BinarySearchRec(F, k, 0, F.length-1); //initialer Aufruf
}
int BinarySearchRec (int[] F, int k, int u, int o) {
  /* input: Folge F der Länge n, Schlüssel k,
        untere Schranke u, obere Schranke o */
  /* output: Position p */

 m = (u+o)/2;
 if  (F[m] ==  k) return m;
 if  ( u == o) return -1;
 if  (F[m] >  k) return BinarySearchRec(F,k,u,m-1);
 return BinarySearchRec(F,k,m+1,o);
}
```

51.2.1 Aufwands Analyse

Das **Terminierungs-Theorem** besagt, dass der Algorithmus BinarySearch für jede endliche Eingabe F nach endlicher Zeit terminiert. In jedem Rekursionsschritt verkürzt sich die Länge des betrachteten Arrays F um mehr als die Hälfte. Nach endlichen vielen Schritten hat das Array nur noch ein Element und die Suche endet entweder erfolgreich oder erfolglos. Falls das Element vorher gefunden wird terminiert der Algorithmus schon früher.

Das **Korrektheits-Theorem** besagt, dass falls das Array F ein Element k enthält, gibt BinarySearch(F.k) den Index eines Vorkommens von k zurück. Ansonsten gibt BinarySearch (F,k) den Wert -1 zurück. Beweisen kann man das durch die verallgemeinerte Induktion nach der Länge n von F. n=1: Der erste Aufruf von BinarySearchRec ist BinarySearchRec(F,k,0,0) und somit m=0. Ist F[0]=k so wird 0 zurückgegeben, ansonsten -1 da 0=0. n>1: Der erste Aufruf von BinarySearchRec ist BinarySearchRec(F,k,0,n-1) und somit m=(n-1)/2. Ist F[m]=k, so wird m zurückgegeben. Ansonsten wird rekursiv auf F[0...m-1] oder F[m+1...n] fortgefahren. Da die Folge sortiert ist, kann k nur in einem der beiden Teile vorhanden sein.

Da die Liste nach jedem Aufruf halbiert wird, haben wir nach dem ersten Teilen der Folge noch n/2 Elemente, nach dem zweiten Schritt n/4 Elemente, nach dem dritten Schritt n/8 Elemente... daher lässt sich allgemein sagen, dass in jedem i-ten Schritt maximal $n/2^i$ Elemente, das heißt $log_2 n$ Vergleiche bei der Suche. Im besten Fall hat die Suche nur einen Vergleich, weil der Suchschlüssel genau in der Mitte liegt. Im schlechtesten Fall und im Durchschnitt für eine erfolgreiche und eine erfolglose Suche liegt die Anzahl der Vergleiche bei $log_2 n$.

51.2.2 Rekursionsgleichung

Für die erfolglose Suche ergibt sich folgende Rekursionsgleichung.

$$T(n) := \begin{cases} \Theta(1) & falls\ n = 1 \\ T(n/2) & sonst \end{cases}$$

Das Auflösen von T(n) nach Induktion ergibt eine $T(n) = \Theta(log\ n)$ Laufzeit für eine erfolglose, also Worst-Case, Suche.

51.3 Iterativer Algorithmus

```
int  BinarySearch(int[] F, int k) {
   /* input: Folge F der Länge n, Schlüssel k */
   /*  output: Position p  (0 ≤ p ≤ n-1)  */

   int u = 0;
   int o = F.length-1;
   int m;
   while (u <= o) {
       m = (u+o)/2;
       if  (F[m] ==  k)
           return m;
       else
           if (k < F[m])
               o = m-1;
           else
               u = m+1;

  }
  return -1;
}
```

Der erste Teil des Algorithmus ist die Initialisierung. Die while Schleife, besagt, dass so lange wiederholt werden soll, bis die angegebenen Schranken erreicht sind. Die if Anweisung ist die Abbruchbedingung. Der letzte Teil des Algorithmus (else) passt die obere, bzw. untere Schranke an.

51.4 Vergleich der Suchverfahren

Verfahren / #**Elemente**	10	10^2	10^3	10^4
sequenziell (n/2)	≈ 5	≈ 50	≈ 500	≈ 5000
binär $log_2 n$	$\approx 3{,}3$	$\approx 6{,}6$	$\approx 9{,}9$	$\approx 13{,}3$

51.5 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5.1.2 zu finden.

2 http://www.dpunkt.de/buecher/3358.html

52 Fibonacci Suche

Dieses Kapitel behandelt die Fibonacci Suche. Die im vorherigen Kapitel behandelte binäre Suche hat Nachteile. Die binäre Suche ist der am häufigsten verwendete Algorithmus zur Suche in sortierten Arrays. Die Sprünge zu verschiedenen Testpositionen sind allerdings immer recht groß. Dies kann nachteilig sein, wenn das Array nicht vollständig im Speicher vorliegt (oder bei Datenträgertypen wie Kassetten). Außerdem werden neue Positionen durch Division berechnet und je nach Prozessor ist dies eine aufwändigere Operation als Addition und Subtraktion. Daher nehmen wir die Fibonacci Suche als eine weitere Alternative.

52.1 Fibonacci Zahlen

Zur Erinnerung, die Folge der Fibonacci Zahlen[1] F_n für $n \geq 0$ ist definiert durch

$F_0 = 0$
$F_1 = 1$
$F_2 = 1$
$F_i = F_{i-1} + F_{i-2}$ für $i > 1$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F_i	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377

Anstatt wie bei der binären Suche das Array in gleich große Teile zu teilen, wird das Array in Teilen entsprechend der Fibonacci-Zahlen geteilt Es wird zunächst das Element an Indexposition m betrachtet, wobei m die kleinste Fibonaccizahl ist, die größer als die Arraylänge ist. Nun fährt man rekursiv mit dem entsprechenden Teilarray fort.

52.2 Rekursive Fibonacci Suche

```
public int fibonacciSearch(int[] arr, int elem) {
        return fibonacciSearchRec(arr,elem,0,arr.length-1);
}

public int fibonacciSearchRec(int[] arr, int elem, int u, int o) {
        int k = 0;
        while (fib(k) < o-u) k++;
        if (elem == arr[u+fib(--k)])
                return u+fib(--k);
        if (u == o)
```

1 https://de.wikipedia.org/wiki/Fibonacci-Folge

```
                return -1;
        if (elem < arr[u+fib(k)])
                return fibonacciSearchRec(arr, elem, u, u+fib(k)-1);
        return fibonacciSearchRec(arr ,elem, u+fib(k)+1, o);
}
```

52.3 Beispiel

9	19	21	34	87	102	158	159	199	205

Wo befindet sich die 133?

1. fibonacciSearchRec(arr,133,0,9)
   a) fib(6)=8 < 9-0 (und maximal)
   b) arr[fib(6)+0] = arr[8] = 199 > 133
2. fibonacciSearchRec(arr,133,0,7)
   a) fib(5)=5 < 7-0 (und maximal)
   b) arr[fib(5)+0] = arr[5] = 102 < 133
3. fibonacciSearchRec(arr,133,6,7)
   a) fib(0)=0 < 7-6 (und maximal)
   b) arr[fib(0)+6] = arr[6] = 158 > 133
4. fibonacciSearchRec(arr,133,6,6)
   a) Suche erfolglos

Wo befindet sich die 87?

1. fibonacciSearchRec(arr,87,0,9)
   a) fib(6)=8 < 9-0 (und maximal)
   b) arr[fib(6)+0] = arr[8] = 199 > 87
2. fibonacciSearchRec(arr,87,0,7)
   a) fib(5)=5 < 7-0 (und maximal)
   b) arr[fib(5)+0] = arr[5] = 102 > 87
3. fibonacciSearchRec(arr,87,0,4)
   a) fib(4)=3 < 4-0 (und maximal)
   b) arr[fib(4)+0] = arr[3] = 34 < 87
4. fibonacciSearchRec(arr,87,4,4)
   a) Suche erfolgreich

52.4 Aufwands Analyse

Die Fibonacci Suche hat dieselbe Komplexität wie die binäre Suche. Die Anzahl der Vergleiche im besten Fall ist 1 und die Anzahl der Vergleiche im Durchschnitt (erfolgreich/erfolglos) und im schlechtesten Fall ist $log_2 n$. Die nötigen Fibonaccizahlen können vorab berechnet und in einem (statischen) Array gespeichert werden. Für Arrays mit weniger als 100.000.000 Elementen werden "nur" die ersten 50 Fibonaccizahlen benötigt. Also Operationen können nur Subtraktion und Addition genutzt werden und die "Sprünge" zwischen Arrayposition ist im Durchschnitt geringer als bei binärer Suche.

53 Einleitung Suchen in Texten

Nun behandeln wir das Suchen in Texten. Das Problem ist das Suchen eines Teilwortes in einem langen anderen Wort. Dies ist eine typische Funktion der Textverarbeitung. Nun ist eine effiziente Lösung gesucht. Das Maß der Effizienz ist hierbei die Anzahl der Vergleiche zwischen den Buchstaben der Worte. Den Vergleich von Zeichenketten nennt man String-Matching und eine nicht übereinstimmende Position nennt man Mismatch.

53.1 Vorgegebene Daten

• Worte als Array:

- text[] zu durchsuchender Text

- pat[] 'Pattern', gesuchtes Wort

• Wortlängen:

- n Länge des zu durchsuchenden Textes

- m Länge des gesuchten Wortes

• $\sum$ Alphabet, ϵ leerer String

Abstrakte Algorithmenbeschreibung:

Eingabe: text[], pat[]

Ausgabe: Index i mit text[i...i+m]=pat[1...m+1] oder -1 falls das gesuchte Wort nicht im Text vorkommt.

53.2 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5 zu finden.

1 `http://www.dpunkt.de/buecher/3358.html`

54 Einleitung Suchen in Texten

Nun betrachten wir einen naiven Algorithmus zur Textsuche.

55 Problem der Worterkennung

Direkte Lösung - brute force

a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b

a (1)	b (2)	a (3)	c (4)	a (5)	b (6)

a (7)	b	a	c	a	b

a (8)	b (9)	a	c	a	b

....

a (22)	b (23)	a (24)	c (25)	a (26)	b (27)

55.1 Pseudocode Brute Force Algorithmus

for i=1 to n-m+1 do

Falls pat = text[i...i+m-1] gib i zurück;

Gib -1 zurück

In Java:

```
int bruteforce_search(char[] text, char[] pat){
    int i,j;
    for(i = 0; i < text.length - pat.length+1; i++){
        for(j = 0; j < pat.length && pat[j] == text[i+j]; j++)
            ;
        if(j == pat.length)
            return i;
    }
    return -1;
}
```

55.2 Analyse

Das Terminierungstheorem besagt, dass der Algorithmus bruteforce_search bei endlicher Eingabe nach endlich vielen Schritten terminiert.

Das Theorem der Korrektheit besagt, wenn text die Zeichenkette pat enthält, so gibt bruteforce_search(text,pat) den Startindex des ersten Vorkommens von pat zurück, ansonsten -1.

Das Theorem der Laufzeit besagt, dass der Algorithmus bruteforce_search einen Worst-Case Laufzeit von $\Theta(mn)$ hat. Beweisen lässt sich das durch eine einfache Schleifenanalyse. Die äußere for-Schleife wird maximal (n-m)-mal durchlaufen, die innere for-Schleife wird jedes mal maximal m-mal durchlaufen:

$\Theta((n-m)*m) = \Theta(mn)$

Dafür hat bruteforce_search nun einen Platzbedarf von $\Theta(1)$. Kann man durch zusätzlichen Platzbedarf die Laufzeit verbessern?

55.3 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5 zu finden.

1 http://www.dpunkt.de/buecher/3358.html

56 Einleitung Algorithmus von Knuth-Morris-Pratt

Auf dieser Seite behandeln wir den Algorithmus von Knuth-Morris-Pratt[1]. Die Idee ist, dass bereits gelesene Informationen bei einem Mismatch genutzt werden. Kommt es an Stelle j von pat zum Mismatch, so gilt:

```
pat[1...j-1]=text[i...i+j-2]
```

a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b

a (1)	b (2)	a (3)	c (4)	a (5)	b (6)

Das a an Stelle 5 ist das Suffix von pat[1..5]. Nun gilt: schiebe Muster um 4, überprüfe weiter ab Position 6 im Text, ab Position 2 im Muster

a	b (7)	a	c	a	b

Das erste a ist nun das Präfix

a (8)	b (9)	a (10)	c (11)	a (12)	b

a (13)	b	a	c	a	b

a (14)	b (15)	a (16)	c (17)	a (18)	b (19)

1 https://de.wikipedia.org/wiki/Knuth-Morris-Pratt-Algorithmus

56.1 Realisierung mit Fehlerfunktion

Bestimme für jedes j der Länge f[j] des längsten Präfixes von pat der Suffix von pat[1..j] ist. Gibt es einen Fehler an Stelle j, dann verschiebe die Suchposition im Muster auf j:=f[j-1]+1=border[j].

Position j im Pattern	1	2	3	4	5	6
Pattern pat[j]	a	b	a	c	a	b
Längster Präfix f[j]	0	0	1	0	1	2
Verschiebeposition	0	1	1	2	1	2

56.2 border im Detail

Preprocessing bedeutet, dass für jedes $j, 1 \leq j \leq m$ das größte k so bestimmt wird, dass pat [1...k-1] ein echter Suffix von pat[1...j-1] ist. Genauer berechnet und als border bezeichnet wird:

$$border[j] := max_{a \leq k \leq j-1}\{k | pat[1...k-1] = pat[j-k+1...j-1]\}$$

Bei einem Mismatch an Position j verschiebe die Position im Text auf i:=i+ border[j])oder 1 falls nicht definiert, z.B. erste Position) und die Position im Suchmuster auf j:=border[j]

56.3 Die border-Tabelle

Beispiel: Drei Zeilen j, pat[j] und border [j]

1	2	3	4	5	6	7	8	9	10	11	12	13	j
a	b	a	a	b	a	b	a	a	b	a	a	b	pat[j]
0	1	1	2	2	3	4	3	4	5	6	7	5	border[j]

Dieses Beispiel ist ein so genannter Fibonacci String.

F_7 :

1.$F_0 = \epsilon, F_1 = b, F_2 = a$

2.$F_n = F_{n-1}F_{n-2}$

56.4 Algorithmus von border

```
Eingabe: char-Array pattern[]
Ausgabe: int-Array border[]
```

```
int[] border = new int[pattern.length];
for(int k = 0; k < border.length; k++){
    border[k] = 0;
}

int i = 1, j = 0;
while(i < border.length){
     while(i+j < border.length-1 &&
           pattern[j] == pattern[i+j]){
        border[i+j+1] = max(border[i+j+1],j+1);
        j++;
     }
     i++;
}
```

56.5 sborder als Verbesserung von border

Problem:

pat:							a	b	a	a	b	a	-	-
text:	-	-	-	-	-	-	a	b	a	a	b	c	-	-

Hier gibt es ein mismatch an der Stelle j=g, border[6]=3. Daher muss um 3 verschoben werden.

pat:										a	b	a	a	b	a	-	-
text:	-	-	-	-	-	-	a	b	a	a	b	c	-	-			

Nun haben wir als Result sofort wieder ein Mismatch. Wir wissen bereits, dass an der Mismatch Stelle kein a stehen darf.

Verbesserung:

$$sborder[j] = max_{1 \leq k \leq j-1}\{k | pat[1...k-1] = pat[j-k+1...j-1] pat[k] \neq pat[j]\}$$

Falls kein deratiges k existiert, dann 0.

Beispeil vier Zeilen mit j, pat[j], border[j] und sborder[j]:

1	2	3	4	5	6	7	8	9	10	11	12	13	j
a	b	a	a	b	a	b	a	a	b	a	a	b	pat[j]
0	1	1	2	2	3	4	3	4	5	6	7	5	border[j]
0	1	0	2	1	0	4	0	2	1	0	7	1	sborder[j]

56.6 Algorithmus

```
Eingabe: char-Array text[], char-Array pattern[]
Ausgabe: true/false
```

```
int[] sborder = new int[pattern.length];
for(int k = 0; k < sborder.length; k++){
    sborder[k] = 0;
}

int i = 1, j = 0;
while(i < sborder.length){
   while(i+j < sborder.length-1 &&
                pattern[j] == pattern[i+j]){
      if(pattern [j+1] == pattern[i+j+1])
          sborder[i+j+1] = max(sborder[i+j+1],j+1);
      j++;
    }
    i++;
}
i = 0;
j = 0;
while(i < text.length() - pattern.length() + 1){
     while(j < pattern.length() && text[i+j] == pattern[j]){
       j++;
     }
     if(j == pattern.length()) return true;
     i = i + max(border[j], 1);
     j = border[j];
}
```

56.7 Analyse

Das Theorem der Terminierung besagt, dass der Algorithmus von Knuth-Morris-Pratt für endliche text[] und pat[] eine endliche Laufzeit hat.

Das Theorem der Korrektheit besagt, wenn text die Zeichenkette pat enthält, so gibt der Algorithmus von Knu5-Morris-Pra5 TRUE zurück, ansonsten FALSE.

Das Theorem der Laufzeit besagt, dass der Algorithmus von Knutt-Morris-Pratt eine Worst-Case Laufzeit von $\Theta(m+n)$ hat. Beweisen kann man das durch eine einfache Schleifenanalyse: $\Theta(m)$ für die Berechnung von sborder und </math> \Theta (n) </math> für die Hauptschleife. Der zusätzliche Platzbedarf des Algorithmus von Knutt-Morris-Pratt ist $\Theta(m)$.

56.8 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5 zu finden.

2 http://www.dpunkt.de/buecher/3358.html

57 Sortieren

Dieses Kapitel gibt eine grundlegende Einführung in das Thema Sortieren[1]. Sortieren ist ein grundlegendes Problem in der Informatik. Es beinhaltet das Ordnen von Dateien mit Datensätzen, die Schlüssel enthalten und das Umordnen der Datensätze, so, dass eine klar definierte Ordnung der Schlüssel (numerisch/alphabetisch) besteht. Eine Vereinfachung ist die Betrachtung der Schlüssel, z.B. ein Feld von int-Werten.

57.1 Ordnung

- Partielle Ordnung

  Sei M eine Menge und $\leq \subseteq (MxM)$ = binäre Relation.

  Es gilt:

  - Reflexivität $x \leq x \; \forall x \in M$
  - Transitivität $x \leq yy \leq z \rightarrow x \leq z \; \forall x, y, z \in M$
  - Antisymmetrie $x \leq yy \leq x \rightarrow x = y \; \forall x, y \in M$
- Strikter Anteil einer Ordnungsrelation $\leq$

$$x < y := x \leq yx \neq y$$

- Totale Ordnung
  - Partielle Ordnung $(M, \leq)$
  - Trichotomie ("Dreiteilung") $x < y \vee x = y \vee x > y \; \forall x, y \in M$

57.2 Grundbegriffe

Das Verfahren ist intern, wenn auf Hauptspeicherstruktur, wie Felder und Listen sortiert wird. Hingegen ist es extern, wenn die Datensätze auf externen Medien, wie Festplatten und weitere sortiert werden. Die Annahmen sind eine totale Ordnung, aufsteigend vs. absteigend und der Platzbedarf.

1 `https://de.wikipedia.org/wiki/Sortierverfahren`

57.3 Problembeschreibung

Als Eingabe haben wir eine Folge von Zahlen $\langle a_1, ..., a_n \rangle$. Als Ausgabe haben wir die Permutation $\langle a'_1, ..., a'_n \rangle$ der Zahlen mit der Eigenschaft $a'_1 \leq a'_2 \leq, ..., a'_n$. Die Sortierung erfolgt nach einem Schlüssel, z.B. Zahlen. In Programmen ist es übertragbar auf beliebige Datenstrukturen mit Schlüssel.

57.4 Stabilität

Ein Sortierverfahren heißt stabil, wenn es die relative Reihenfolge gleicher Schlüssel in der Datei beibehält. Beispiel: alphabetisch geordnete Liste von Personen soll nach Alter sortiert werden. Personen mit gleichem Alter sollen weiterhin alphabetisch geordnet bleiben:

```
Name          Alter              Name          Alter
Aristoteles   24                 Aristoteles   24
Platon        28     SORTIEREN → Platon        28
Sokrates      30                 Theophrastos  28
Theophrastos  28                 Sokrates      30
```

57.5 Sortieralgorithmen

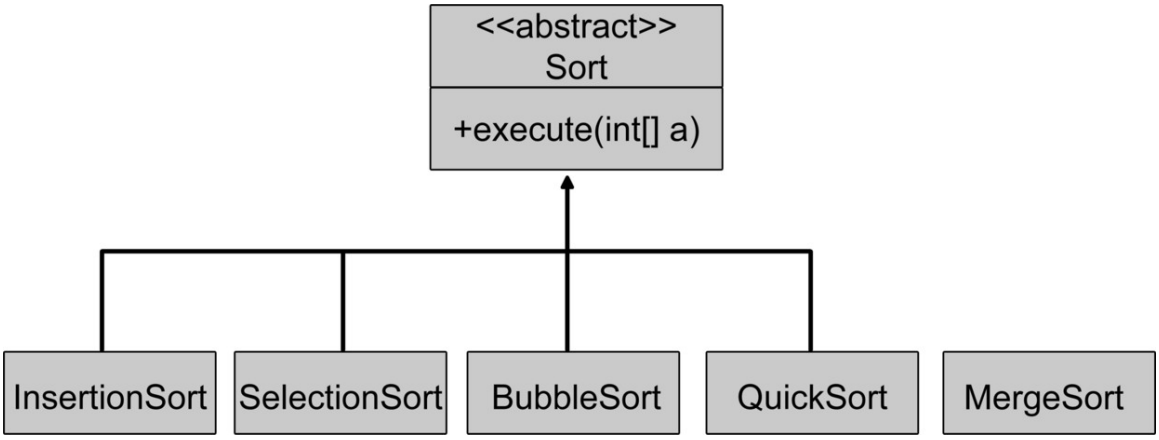

Abb. 12 Sortieralgorithmen

57.6 Java Stub

```
public class InsertionSort extends Sort {
 /*
  * Sortiert die Sequenz a nach dem Verfahren
  * „Sortieren durch Einfügen“
  */
 @Override
 public void execute(int[] a) {
  // Elemente: a[0], … , a[n-1]
  int n=a.length;
```

```
  int x;
  int j;

  // HIER KOMMT DER SORTIERALGORITHMUS
  // assert: a[0] <= … <= a[n-1]
 }
}
```

58 Vergleichsbasiertes Sortieren

Das vergleichbasierte Sortieren ist ein wichtiger Spezialfall des Sortierproblems. Zur Sortierung können nur direkte Vergleiche zweier Werte benutzt werden. Der Wertebereich der Schlüssel kann beliebig sein. Als Eingabe haben wir ein Array ganzer Zahlen und als Ausgabe ein sortiertes Array mit den selben Zahlen mit erhaltenen Mehrfachvorkommen. Einige Sortierverfahren sind effizienter, wenn Listen anstatt Arrays benutzt werden.

58.1 Sortierinterface in Java

```
public interface Sort {

   /**
    * sorts the given array.
    * @param toSort - array to sort.
    */
    public void execute(int[] toSort);
}
```

58.2 Ausblick

Auf den folgenden Seiten werden die Sortieralgorithmen Insertion Sort, Selection Sort, Merge Sort und Quick Sort behandelt.

58.3 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5.2 zu finden.

1 http://www.dpunkt.de/buecher/3358.html

59 InsertionSort

Dieses Kapitel behandelt die Sortiermethode InsertionSort[1] oder auch Sortieren durch Einfügen genannt. Die Idee des Algorithmus ist, die typische menschliche Vorgehensweise, etwa beim Sortieren eines Stapels von Karten umzusetzen. Das heißt es wird mit der ersten Karte ein neuer Stapel gestartet. Anschließend nimmt man jeweils die nächste Karte des Originalstapels und fügt diese an der richtigen Stelle im neuen Stapel ein.

59.1 Beispiel

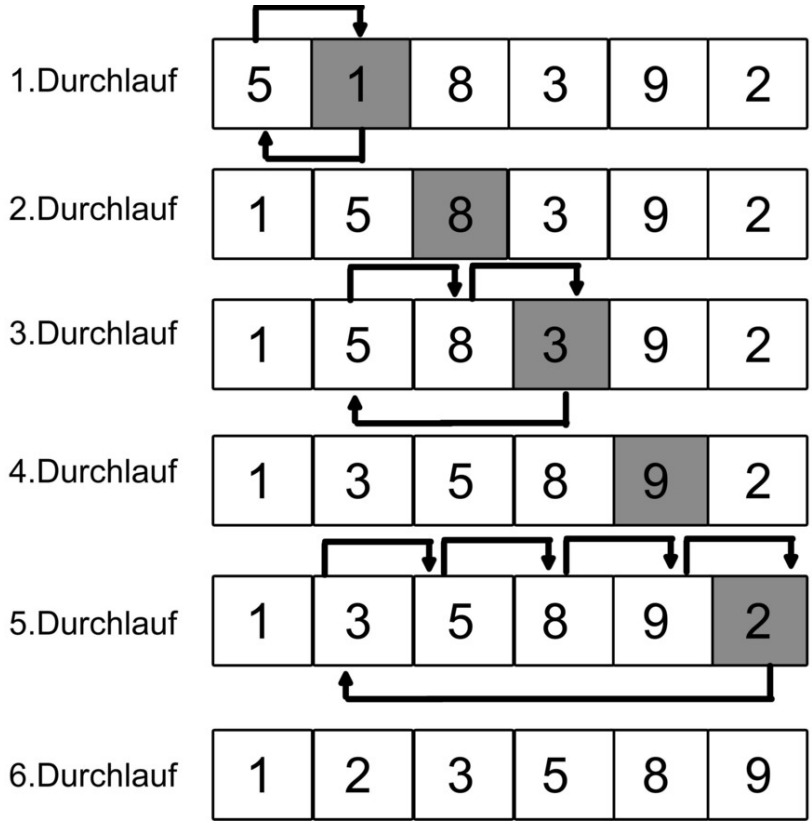

Abb. 13 InsertionSort

1 https://de.wikipedia.org/wiki/Insertion_sort

59.2 Java Code

```
void  InsertionSort(int[] F) {

int m,j;
for (int i = 1; i < F.length; i++){
      j = i;
      m = F[i];
      while (j > 0 && F[j-1] > m) {
             /*verschiebe F[j-1] nach rechts */
              F[j] = F[j-1];
              j--;
       }
       F[j] = m;
     }
}
```

Das Array hat F.length viele Elemente von Position 0 bis F.Length-1. Wenn F[j-1] größer m ist, dann wird F[j-1] nach rechts verschoben. Am Ende des Algorithmus wird F[i] an Position F[j] gesetzt.

59.3 Analyse

59.3.1 Theorem der Terminierung

Das Theorem der Terminierung besagt, dass der Algorithmus InsertionSort für jede Eingabe int[] F nach endlicher Zeit terminiert.

Beweis

Die Laufvariable i in der äußeren for-Schleife wird in jedem Durchgang um eins erhöht und wird damit irgendwann die Abbruchbedingung (eine Konstante)erreichen. Die Laufvariable j der inneren while-Schleife wird in jedem Durchgang um eins verringert und somit die Schleifenbedingung j>0 irgendwann nicht mehr erfüllen.

59.3.2 Theorem der Korrektheit

Das Theorem der Korrektheit besagt, dass der Algorithmus InsertionSort das Problem des vergleichsbasierten Sortierens löst. Beweisen

Beweis

Wir zeigen, dass die folgende Aussage eine Invariante der äußeren for-Schleife ist (d.h. sie ist am Ende eines jeden Schleifendurchgangs gültig): Das Teilarray F[0..i] ist sortiert Damit gilt auch, dass nach Abbruch der for-Schleife das Array F[0..n]=F (mit n=F.length-1) sortiert ist. Zu zeigen ist nun, dass am Ende jeden Durchgangs der äußeren for Schleife F[0...i] sortiert ist. Dies wird durch Induktion nach i gezeigt. Für i=1 gilt im ersten Durchgang wird das erste Element F[0] mit dem zweiten Element F[1] verglichen und ggfs. getauscht

um Sortierung zu erreichen (while-Bedingung). Für $i \to i+1$ gilt angenommen F[0...i] ist am Anfang der äußeren for-Schleife im Durchgang i+1 sortiert. In der while-Schleife werden Elemente solange einen Platz weiter nach hinten verschoben, bis ein Index k erreicht wird, sodass alle Elemente mit Index 0..k-1 kleiner/gleich dem ursprünglichen Element an Index i+1 sind (Induktionsbedingung) und alle Elemente mit Index k+1...i+1 größer sind (while-Bedingung). Das ursprüngliche Element an Index i+1 wird dann an Position k geschrieben. Damit gilt, dass F[0...i+1] sortiert ist.

59.3.3 Theorem der Laufzeit

Das Theorem der Laufzeit besagt, dass die Anzahl der Vergleichsoperationen von Insertion Sort im besten Fall $\Theta(n)$ ist und im durchschnittlichen und schlechtesten $O(n^2)$.

Beweis

Für die Aufwandsanalyse sind die Anzahl der Vertauschungen und der Vergleiche relevant. Allerdings dominieren die Vergleiche die Vertauschungen, das heißt es werden wesentlich mehr Vergleiche als Vertauschungen benötigt. Wir müssen in jedem Fall alle Elemente i:=1 bis n-1 durchgehen, d.h. immer Faktor n-1 für die Anzahl der Vergleiche. Dann müssen wir zur korrekten Einfügeposition zurückgehen

Im **besten Fall** ist die Liste schon sortiert. Die Einfügeposition ist gleich nach einem Schritt an Position i-1, d.h. die Anzahl der Vergleiche ist gleich der Anzahl der Schleifendurchläufe = n-1. Bei jedem Rückweg zur Einfügeposition nimmt man den Faktor 1. Somit beträgt die Gesamtzahl der Vergleiche: $(n-1) \cdot 1 = n-1$. Für große Listen lässt sich $n-1 \approx n$ abschätzen. Damit haben wir einen linearen Aufwand.

Im **mittleren Fall** ist die Liste unsortiert. Die Einfügeposition befindet sich wahrscheinlich auf der Hälfte des Rückwegs. Bei jedem der n-1 Rückwege, muss ein (i-1)/2 Vergleich addiert werden. Die Gesamtzahl der Vergleiche beträgt dann:

$(n-1)/2 + (n-2)/2 + (n-3)/2 + ... + 2/2 + 1/2$

$= \frac{(n-1)+(n-2)+(n-3)+...+2+1}{2}$

$= \frac{1}{2} \cdot \frac{n \cdot (n-1)}{2}$

$= \frac{n \cdot (n-1)}{4}$

$\approx \frac{n^2}{4}$

Daraus ergibt sich ein quadratischer Aufwand, wenn konstante Faktoren nicht berücksichtigt werden.

Im **schlechtesten Fall** ist die Liste absteigend sortiert. Die Einfügeposition befindet sich am Ende des Rückgabewertes bei Position 1. Bei jedem der n-1 Rückwege müssen i-1 Elemente verglichen werden (d.h. alle vorherigen Elemente F[1...i-1]). Analog zu vorhergehenden Überlegungen, gibt es hier aber die doppelte Rückweglänge. Daraus ergibt sich die Gesamtanzahl der Vergleiche:

$(n-1) + (n-2) + (n-3) + ... + 2 + 1 = \frac{n \cdot (n-1)}{2} \approx \frac{n^2}{2}$

Daraus ergibt sich ein quadratischer Aufwand, wenn konstante Faktoren nicht berücksichtigt werden.

59.4 Optimierung

In der vorgestellten Version des Algorithmus wird die Einfügeposition eines Elements durch (umgekehrte) sequenzielle Suche gefunden. Verwendet man hier binäre Suche (das Teilarray vor dem aktuellen Element ist sortiert!) kann die Anzahl der Vergleichsoperationen gesenkt werden zu O(n log n) (genauere Analyse zeigt, dass die Zahl noch kleiner ist)

59.5 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5.2.2 zu finden.

2 `http://www.dpunkt.de/buecher/3358.html`

60 SelectionSort

Dieses Kapitel behandelt die Suchmethode SelectionSort[1]. Die Idee dieses Suchalgorithmus ist, den jeweils größten Wert im Array zu suchen und diesen an die letzte Stelle zu tauschen. Anschließend fährt man mit der um 1 kleineren Liste fort.

60.1 Beispiel

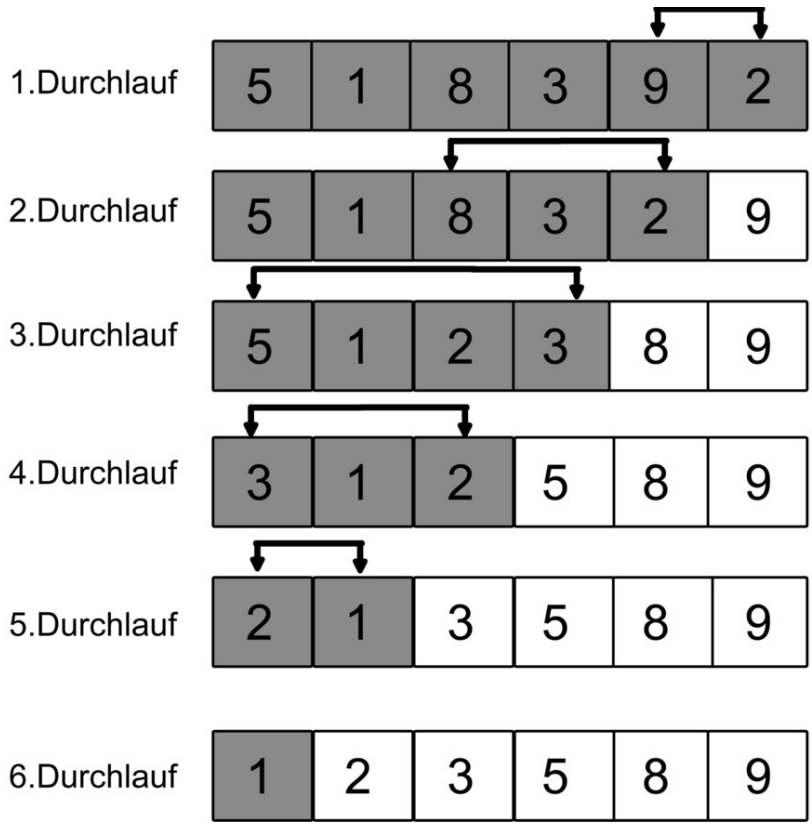

Abb. 14 SelectionSort

60.2 Java Code

```
void  SelectionSort(int[] F) {
```

1 https://de.wikipedia.org/wiki/Selection_Sort

```
   int  marker = F.length -1;
   while (marker >= 0) {
     /*bestimme größtes Element links v. Marker*/
       int max = 0;  /* Indexposition*/
       for (int i = 1; i <= marker; i++){
           if (F[i] > F[max])
               max = i;
       swap(F, marker, max);
       marker--;   /*verkleinere Array */
   }
void  swap(int[] F, int idx1, int idx2) {
    int tmp = F[idx1];
    F[idx1] = F[idx2];
    F[idx2] = tmp;
}
```

In Java benutzt man die Hilfsmethode swap, welche zwei Elemente im Array vertauscht.

60.3 Analyse

60.3.1 Theorem der Terminierung

Das Theorem der Terminierung besagt, dass der Algorithmus SelectionSort für jede Eingabe int[]F nach endlicher Zeit terminiert.

Beweis

Die Variable marker wird zu Anfang des Algorithmus auf einen positiven endlichen Wert gesetzt und in jedem Durchgang der äußeren while-Schleife um 1 verkleinert. Abbruch der while Schleife erfolgt, wenn marker kleiner 0 ist, also wird die while-Schleife endlich of durchlaufen. Die innere for-Schleife hat in jedem Durchgang marker-viele (also endlich viele) Durchläufe.

60.3.2 Theorem der Korrektheit

Das Theorem der Korrektheit besagt, dass der Algorithmus SelectionSort das Problem des vergleichsbasierten Sortierens löst.

Beweis

Es gilt zunächst, dass die innere for-Schleife stets den Index des Maximums des Teilarrays F[0...marker] berechnet und in max speichert. Weiterhin gilt, dass die Methode swap(F, marker, max) die Werte F[marker] und F[max] vertauscht.

Wir zeigen nun, dass die folgenden Aussagen Invariante der äußeren while-Schleife ist (d.h. sie sind am Ende eines jeden Schleifendurchgangs gültig):

1.) Das Teilarray F[marker+1...n] ist sortiert
2.) Alle Zahlen in F[0...marker] sind nicht größer als jede Zahl in F[marker+1...n]

Damit gilt (nach 1.) auch, dass nach Abbruch der while-Schleife das Array F[0..n]=F (mit n=F.length-1) sortiert ist.

Zeige dies durch Induktion nach i=n-marker:

- i=1: Im ersten Durchgang wird das Maximum as F[0..n] bestimmt und an die letzte Stelle F[n] vertauscht. Damit gilt sowohl 1.) als auch 2.)
- i→i+1: Nehme an, dass nach dem i-ten Durchgang der while-Schleife gilt

  1.) Das Teilarray F[(i-n+1)...n] ist sortiert
  2.) Alle Zahlen in F[0...(i-n)] sind nicht größer als jede Zahl in F[(i-n+1)...n]

Im (i+1) Durchgang wird zunächst das Maximum aus F[0...(i-n)] bestimmt und anschließend mit dem Element an Position F[i-n] getauscht. Da nach 2.) dieses Element nicht größer war als jede Zahl in F[(i-n+1)...n], ist nun F[(i-n)...n] sortiert (Invariante 1.). Da wir aus F[0...(i-n)] ein Maximum genommen haben, kann nun auch kein Element in F[0...(i-n-1)] größer sein, als jede Zahl in F[(i-n)...n] (Invariante 2.)

60.3.3 Theorem der Laufzeit

Das Theorem der Laufzeit besagt, dass der Algorithmus SelectionSort eine Laufzeit von $\Theta(n^2)$ hat im besten, mittleren und schlechtesten Fall.

Beweis

Die äußere while-Schleife wird genau n-mal (n=F.length) durchlaufen. Dort werden somit n Vertauschungen vorgenommen (=jeweils konstanter Aufwand). Die innere for-Schleife hat im i-ten Durchlauf der while-Schleife n-i Durchläufe mit jeweils einem Vergleich, deswegen insgesamt

$(n-1)+(n-2)+(n-3)+...+1 = \frac{n\cdot(n-1)}{2} \approx \frac{n^2}{2}$

Die Anzahl der Vergleiche ist im besten, mittleren und schlechteste Fall identisch, da immer das komplette Array durchlaufen wird.

60.4 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5.2.3 zu finden.

2 `http://www.dpunkt.de/buecher/3358.html`

61 BubbleSort

Dieses Kapitel behandelt die Suchmethode BubbleSort[1]. Es handelt sich hierbei um ein sehr bekanntes, aber nicht besonders effizientes Sortierverfahren. Es ist eine einfach zu implementierende zugrunde liegende Vorstellung. Bei einer vertikalen Anordnung von Elementen in Form von Luftblasen (bubbles) werden wie in einer Flüssigkeit von alleine sortiert, da die größeren Blasen die kleiner „überholen“. Das Grundprinzip ist somit die Folge immer wieder zu durchlaufen und dabei benachbarte Elemente, die nicht die gewünschte Sortierreihenfolge haben, zu vertauschen. Das bedeutet Elemente die größer sind als ihre Nachfolger, überholen diese.

1 `https://de.wikipedia.org/wiki/Bubble_Sort`

61.1 Beispiel

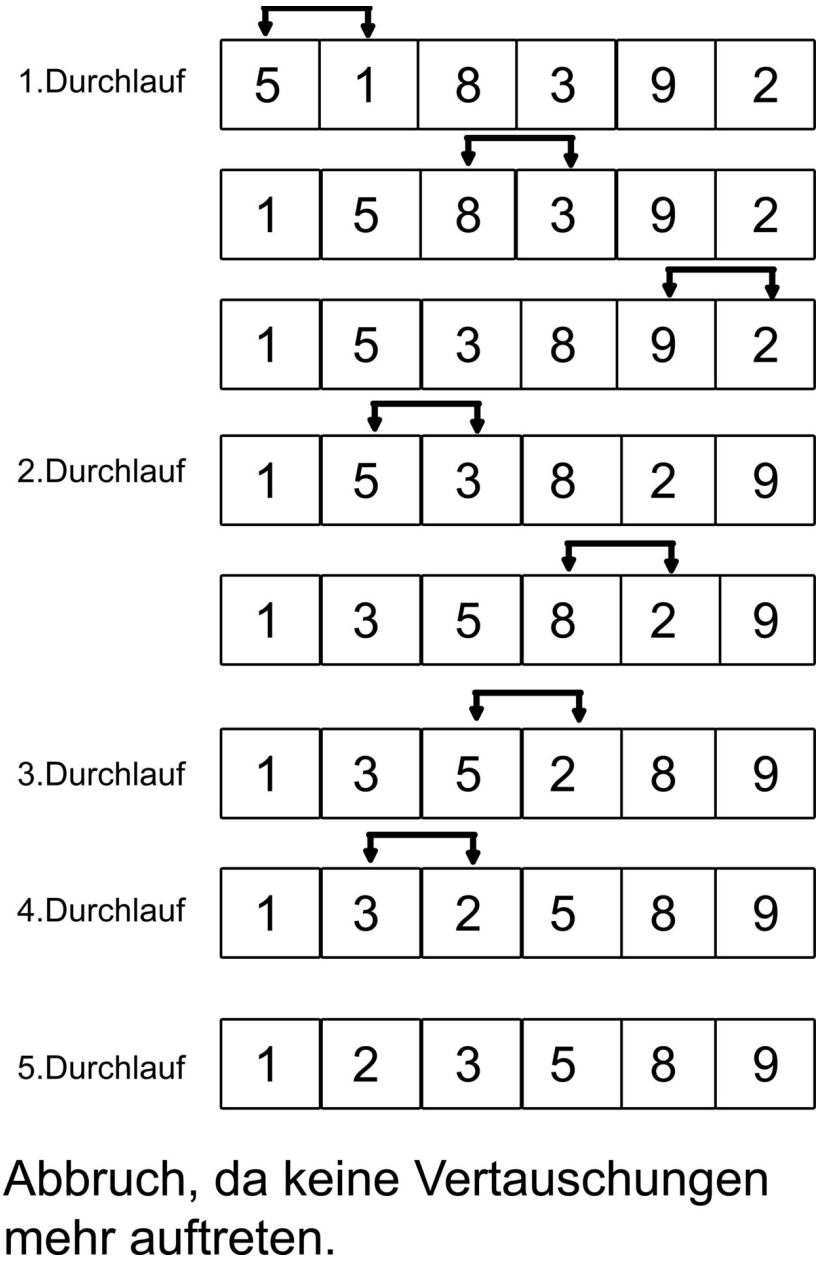

Abb. 15 BubbleSort

61.2 Java Code

```
void  BubbleSort(int[] F) {

    for (int n= F.length; n >1; n=n-1) {
      for (int i =0; i < F.length-1; i++)  {
           if (F[i] > F[i+1]){
```

```
                swap(F, i, i+1);
            }
        }
    }
}
```

Hierbei handelt es sich um die einfachste Form, doch der Algorithmus kann auch optimiert werden. Wir haben beobachtet, dass die größte Zahl in jedem Durchlauf automatisch an das Ende der Liste rutscht. Daraus folgt in jedem Durchlauf j reicht die Untersuchung bis Position n-j, das heißt im j.ten Durchlauf sind die Elemente zwischen den Positionen n-j und n-1 sortiert. Wenn keine Vertauschung mehr stattfindet, soll das Programm abbrechen.

```
void  BubbleSort(int[] F) {
   boolean swapped;
   int n = F.length;
   do {
       swapped = false;
       for (int i =0; i < n-1; i++)  {
           if (F[i] > F[i+1]){
               swap(F, i, i+1);
               swapped = true;
           }
       }
       n--;
    }while (swapped);

}
```

61.3 Aufwand

Im **besten Fall** beträgt der Aufwand n. Im **mittleren Fall** ohne Optimierung n^2 und mit Optimierung $\frac{n^2}{2}$. Im **schlechtesten Fall** ohne Optimierung beträgt der Aufwand n^2 und mit Optimierung $\frac{n^2}{2}$.

61.4 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5.2.4 zu finden.

2 http://www.dpunkt.de/buecher/3358.html

62 MergeSort

In diesem Kapitel wird der Sortieralgorithmus MergeSort[1] behandelt.

62.1 Rückblick

Die bisherige Verfahren erforderten einen direkten Zugriff auf einzelne Elemente (z.B. in einem Array). Sie sind besonders geeignet für internes Sortieren. Allerdings gibt es Probleme, wenn Daten sortiert werden sollen, die nicht in den den Hauptspeicher passen. Daher brauchen wir andere Verfahren, die nicht zwingend Elemente intern verwalten. Das Prinzip dieser Algorithmen ist das Sortieren in mehreren Phasen oder Schritten.

62.2 Idee

MergeSort ist ein Divide-and-Conquer Algorithmus zum vergleichsbasierten Sortieren. Zuerst wird die zu sortierende Folge in zwei Teile geteilt. Anschließend werden beide Teile voneinander getrennt sortiert. Zuletzt werden beide Teilergebnisse in der richtigen Reihenfolge zusammen gemischt.

1 `https://de.wikipedia.org/wiki/Merge_Sort`

62.3 Beispiel

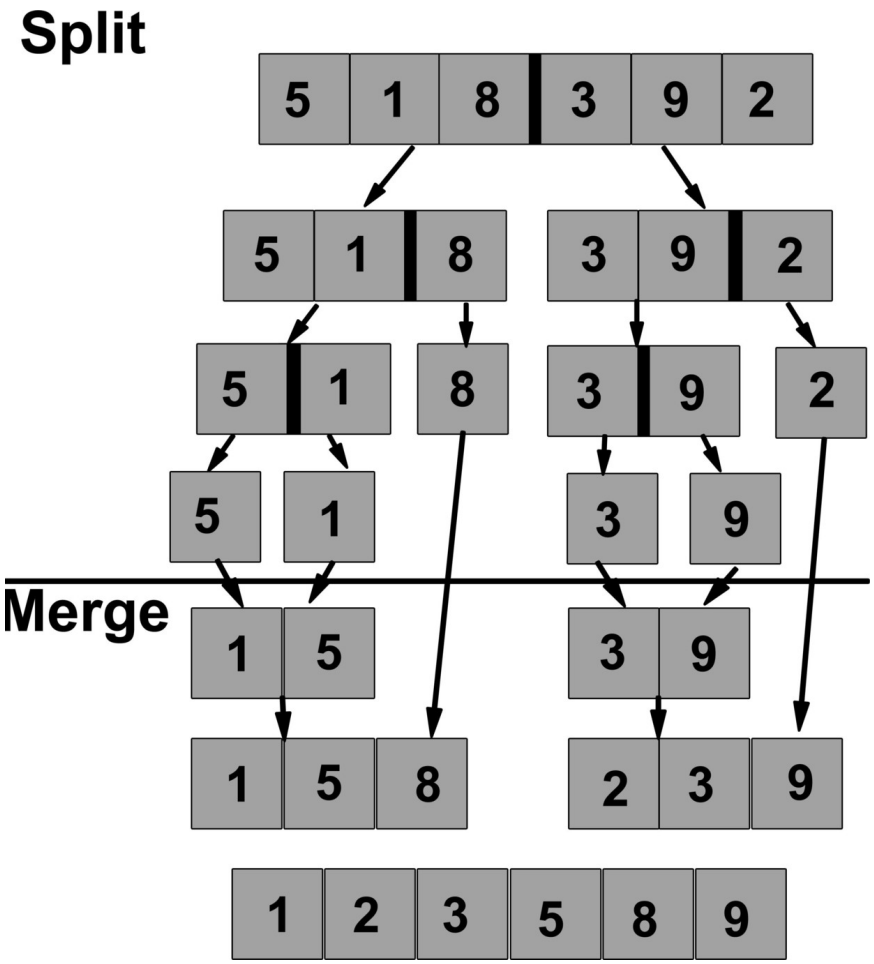

Abb. 16 MergeSort

62.4 Algorithmus

```
void mergeSort(int[] F) {
    int[] tmpF = new int[F.length];
    mergeSort(F, tmpF, 0, F.length -1);
}


void mergeSort(int[] F, int[] tmpF, int left,int right)
{
    if (left < right) {
        int m = (left + right)/2;
        mergeSort(F, tmpF, left, m);
        mergeSort(F, tmpF, m+1, right);
        merge(F, tmpF, left, m+1, right);
    }
}
void merge(int[] F, int[] tmpF, int startLeft, int startRight, int endRight) {
  int endLeft = startRight-1;
```

```
  int tmpPos = startLeft;
  int numElements = endRight - startLeft +1;
  while (startLeft <= endLeft && startRight <= endRight)
     if (F[startLeft] < F[startRight])
         tmpF[tmpPos++] = F[startLeft++];
     else
         tmpF[tmpPos++] = F[startRight++];

  while (startLeft <= endLeft)
      tmpF[tmpPos++] = F[startLeft++];
  while (startRight <= endRight)
      tmpF[tmpPos++] = F[startRight++];

  for (int i = 0; i < numElements; i++, endRight--)
         F[endRight] = tmpF[endRight];
}
```

Das Abbruchkriterium für den rekursiven Aufruf ist eine einelementige Liste.Der Misch-Vorgang erfordert in der Regel doppelten Speicherplatz, da eine neue Folge aus den beiden Sortierten generiert werden muss. Eine Alternative ist das Mischen in einem Feld (in-place), das erfordert aber aufwendiges Verschieben.

62.5 Analyse

62.5.1 Theorem der Terminierung

Das Theorem der Terminierung besagt, dass der Algorithmus MergeSort für jeden Eingabe int[]F nach endlicher Zeit terminiert.

Beweis

Zeige zunächst, dass jeder Aufruf mergeSort(int[] F, int[] tmpF, int left,int right) terminiert:

- Falls lef $<$ right nicht gilt, terminiert der Aufruf sofort
- Andernfalls rufen wir mergeSort rekursiv auf, wobei entweder lef einen echt größeren oder right einen echt kleineren Wert erhält. In jedem Fall wird nach einem gewissen rekursiven

Abstieg irgendwann lef$<$right nicht mehr gelten.

62.5.2 Theorem der Korrektheit

Das Theorem der Korrektheit besagt, dass der Algorithmus MergeSort das Problem des vergleichsbasierten Sortierens löst.

Beweis

Durch Induktion nach n = F.length. Annahme n=2 für eine ganze Zahl k.

- n=1: Für n=1 ist der erste Aufruf der mergeSort Hilfsmethode mergeSort(F, tmpF, 0, 0)

und somit gilt nicht lef $<$ right. Die Methode terminiert ohne Änderung an F. Dies ist korrekt, da jedes einelementige Array sortiert ist.

- n/2 $\rightarrow$ n: Sei F[0...n-1] ein beliebiges Array. Der erste Aufruf mergeSort(F, tmpF, 0, n-1) erfüllt lef<right und es werden folgende Rekursive Aufrufe getätigt: mergeSort(F, tmpF, 0, n/2-1) mergeSort(F, tmpF, n/2, n-1) Beide Aufrufe erhalten ein Array der Länge n/2. Nach Induktionsannahme gilt, dass anschliessend sowohl F[0...n/2-1] als auch F[n/2...n-1] separat sortiert sind. Noch zu zeigen ist, dass merge korrekt zwei sortierte Arrays in ein sortiertes Array mischt.

62.5.3 Theorem der Laufzeit

Das Theorem der Laufzeit besagt, dass der Algorithmus MergeSort eine Laufzeit von $\Theta(nlog_2n)$ hat. Diese Laufzeit ist die selbe für den besten, mittleren und schlechtesten Fall.

Beweis

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

Nun wenden wir das Master Theorem an.

Im 2. Fall, wenn $f(n) \in \Theta(n^{log_b a} * ld^k n)$ für ein $k \geq 0$ dann $T(n) = \Theta(n^{log_b a} * ld^{k+1} n)$

Hier ist a=2 und b=2 und es folgt $n^{log_b a} = n^{log_2 2} = n^1 = n$.

Es ist zudem f(n)=n und es gilt für k=0:

$n \in \Theta(n\ ld^k\ n) = \Theta(n)$

Es folgt $T(n) \in \Theta(n\ ld^{k+1}\ n) = \Theta(n\ ld\ n)$.

62.6 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5.2.5 zu finden.

2 `http://www.dpunkt.de/buecher/3358.html`

63 Zwischenbemerkungen

An dieser Stelle gibt es einige Zwischenbemerkungen zu den vorgestellten Sortieralgorithmen.

63.1 Einordnung der elementaren Sortierverfahren

	Implementierung Array	**Implementierung Liste**
greedy	selection sort, bubble sort	insertion sort
divide-and-conquer	quicksort	merge sort

63.1.1 Eigenschaften

Divide and conquer[1] bedeutet teile und herrsche. Dabei wird das eigentliche Problem so lange in kleiner und einfachere Teilprobleme zerlegt, bis man diese lösen kann.Oft können Teilprobleme parallel gelöst werden. Des weiteren sind Teilprobleme „eigenständige“ Probleme. Anschließend werden die Teillösungen zu einer Gesamtlösung zusammengeführt.

Greedy[2] bedeutet gierig. Hierbei wird schrittweise ein Folgezustand ausgewählt, der aktuell den größten Gewinn und das beste Ergebnis verspricht. Die Auswahl des Folgezustands erfolgt anhand von Bewertungsfunktionen und Gewichtsfunktionen. Ein Problem dabei ist, dass oft nur ein lokales Maximum gewählt wird. Mehr dazu im Thema Entwurfsmuster.

63.2 Generische Implementierung

Algorithmen werden „parametrisiert“ durch Vergleichsoperator. Im Paket java.lang gibt es dafür ein Interface Comparable. Der Aufruf der Vergleichsmethode a.compareTo(b) liefert ein Zahl <0, =0, >0 für a<b, a=b und a größer b. Das Muster für Objekte vom Referenztyp Comparable lautet:

```
public class MyObject  implements Comparable {
      MyType data;
      public int compareTo (MyObject obj) {
            if (,,this.data < obj.data‘‘) return -1;
            if (,,this.data = obj.data‘‘) return 0;
            if (,,this.data > obj.data‘‘) return 1;
     }
}
```

1 https://de.wikipedia.org/wiki/Divide_and_conquer
2 https://de.wikipedia.org/wiki/Greedy-Algorithmus

Das Muster für Aufrufe in Klassenmethoden bei Suchverfahren lautet:

```
public static int binarySearch( Comparable[] f,
       Comparable a,  int l,  int r) {
           int p = (l+r)/2;
           int c = f[p].compareTo(a);
  ... }
```

63.2.1 Listenimplementierung generisch

```
public class MyObject implements Comparable {. . .}

public class Node {
    MyObject data;
    Node next;
}
public class OrderedList {
 private Node head;
 public OrderedList sort ( ) {. . .}
```

63.2.2 Interne Hilfsmethoden

- int findMin(){...}
  - F.findMin() bestimmt den Index des minimalen Elements von OrderedList F
- void insertLast(int a)
  - F.insertLast(a) fügt Element mit Index (Key) a an das Ende von F an
- void deleteElem(int a)
  - F.deleteElem(a) löscht Element mit Index a aus der Liste F
- Aufwand: jeweils = O(n), wenn n = Anzahl der Objekte in Liste

63.2.3 MergeSort generisch

```
public class OrderedList {
   OrderedNode head;
   int length;
   // ...

   /**    * Sorts this list in non-descending order       */
   public void mergeSort() {
     OrderedList aList, bList; // the divided lists
     OrderedNode aChain; // start of first node chain
     OrderedNode bChain; // start of second node chain
     OrderedNode tmp; // working node for split

     // trivial cases
     if ( (head==null) ||  (head.next == null) )
          return;
// divide: split the list in two parts
    aChain = head;
```

```
    tmp = head;      // init working node for split
    // advance half of the list
    for (int i=0; i < (length-1) / 2; i++)
      tmp=tmp.next;

    // cut chain into aChain and bChain
    bChain=tmp.next;
    tmp.next=null;

    // encapsulate the two node chains in two lists
    aList = new OrderedList();
    aList.head=aChain;
    aList.length=length/2;
    bList = new OrderedList();
    bList.head=bChain;
    bList.length=length - aList.length;

    // conquer: recursion
    aList.mergeSort(); bList.mergeSort();
    // join: merge
    merge(aList, bList);
  }
}
```

Aus Gründen der Übersichtlichkeit erzeugt dieses Programm im Divide-Schritt jeweils gekapselte Zwischenlisten vom Typ OrderedList. In der Praxis würde man hierauf verzichten und rein auf Knoten-Ketten arbeiten, da insgesamt O(n) Objekte vom Typ OrderedList erzeugt und wieder vernichtet werden(maximal O(log n) davon sind gleichzeitig aktiv).

64 QuickSort

In diesem Kapitel wird der Sortieralgorithmus QuickSort[1] behandelt.

64.1 Idee

Es gibt eine rekursive Aufteilung (wie bei MergeSort), aber hier werden Mischvorgänge vermieden (speicherintensiv!). Die Teillisten werden in zwei Hälften geteilt bezüglich eines Pivot-Elements, wobei in einer Liste alle Elemente größer als das PivotElement sind und in der anderen Liste alle kleiner. Das Pivot Element ist ein beliebiges Element der Liste/Folge, z.B. das linke, mittlere oder rechte Element.

1 `https://de.wikipedia.org/wiki/Quicksort`

64.2 Beispiel

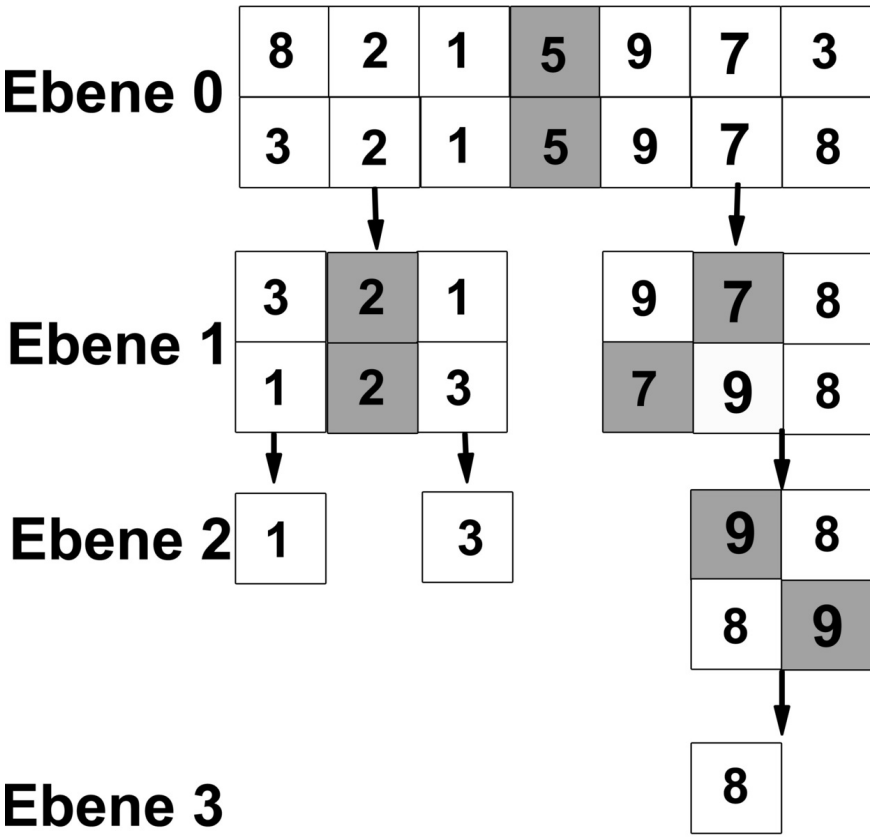

Abb. 17 QuickSort Algorithm

64.3 Vertauschen von Elementen

Für gegebenes Pivot-Element p wird die Folge von links durchsuchen, bis das Element gefunden wurde, das größer oder gleich p ist. Und gleichzeitig wird die Folge von rechts durchsuchen, bis das Element gefunden ist, das kleiner p ist. Dabei werden die Elemente ggf. getauscht.

64.4 Sortierprinzip

Sortieren einer Folge F[u...o] nach dem „divide-and-conquer“Prinzip. Divide heißt die Folge F[u...o] wird in zwei Teilfolgen F[u...p-1] und F[p+1...o] geteilt. Die zwei Teilfolgen haben folgende Eigenschaften:

- F[i] ≤F[p] für alle i = u,...,p-1
- F[i] > F[p] für alle i = p+1, ..., o

Conquer bedeutet, dass die Teilfolgen sortiert werden. Mit combine werden die Teilfolgen zu F[u...o] verbunden. Vergleiche sind an dieser Stelle nicht erforderlich, da die Teilfolgen bereits sortiert sind.

64.5 Pivot Element

Im Prinzip muss man nicht das letzte Element als Pivot-Element wählen. Je nach Verteilung der Daten, kann es sinnvoll sein ein anderes Element zu wählen. Wenn beispielsweise die Liste schon fast sortiert ist, sollte man immer das mittlere Element wählen Eine optimale Rekursion erhält man, wenn man immer den Median als Pivot-Element wählt (dieser ist aber nicht direkt bestimmbar, dafür müsste man die Liste erst sortiert haben. Hat man ein Pivot-Element ausgewählt, tauscht man dies einfach mit dem letzten Element und benutzt den Algorithmus wie zuvor.

64.6 Algorithmus

```
void quickSort(int[] F, int u, int o) {
     if (u < o) {
        int p = (u+o)/2;
        int pn = zerlege(F,u,o,p);
        quickSort(F,u,pn-1);
        quickSort(F,pn+1,o);
     }
int zerlege(int[] F, int u, int o, int p) {
     int pivot = F[p];
     int i = u;
     int j = o;

     while (i < j) {
         while (F[i] < pivot)
             i++;
         while (F[j] > pivot)
             j--;
         if (i < j) {
             swap(F,i , j );
         }
     }
     return i;
}
int zerlege(int[] F, int u, int o, int p) {
     int pivot = F[p];

     //Tausche Pivot-Element mit dem letzten Element
     //kann entfallen, wenn immer p=o gewählt wird
     swap(F,p, o);
     int pn = u;

     //bringe kleinere Elemente nach vorne und größere nach hinten
     for (int j = u; j < o; j++) {
          if (F[j] <= pivot){
               swap(F,pn, j );
               pn++;
          }
```

```
    }

    //bringe das Pivot-Element an die richtige Position und gebe diese zurück
    swap(F,pn, o);
    return pn;
}

void swap(int[] f, int x, int y){   //Hilfsmethode zum Vertaucshen
   int tmp = f[x];
   f[x] = f[y];
   f[y] = tmp;
}

}
```

P gibt an ,an welcher Position das Pivot Element ist. Bei diesem Beispiel ist es in der Mitte. Es kann aber auch an Stelle o oder u sein.

64.6.1 Beispiel 1

Zerlege (F,0,6,3) mit 3=(0+6)/2

8	2	1	5	9	7	3

...

3	2	1	5	9	7	3

64.6.2 Beispiel 2

Sei f[8]=5 das Pivot-Element

8	9	2	6	7	3	4	1	5

Suche von links aus das Element, welches kleiner als das Pivot-Element ist

8	9	2	6	7	3	4	1	5

Vertausche mit dem ersten größeren Element

2	9	8	6	7	3	4	1	5

Suche das nächste kleinere Element als die 5

2	9	8	6	7	3	4	1	5

Vertausche dieses mit dem zweiten größeren Element

2	3	8	6	7	9	4	1	5

Suche wieder das nächste kleinere Element

2	3	8	6	7	9	4	1	5

und vertausche dies mit dem dritt größeren Element

2	3	4	6	7	9	8	1	5

2	3	4	6	7	9	8	1	5

2	3	4	1	7	9	8	6	5

nun ist man rechts angekommen und hier wird nun das Pivot-Element getauscht

2	3	4	1	5	9	8	6	7

Von nun an steht das Pivot-Element an seiner finalen Position. Alle Elemente links vom Pivot-Element sind kleiner und alle auf der rechten Seite sind größer. Das bedeutet, dass nun ein rekursiver Abstieg für die Folgen

2	3	4	1

und

9	8	6	7

beginnen würde. Wenn das letzte Element wieder als Pivot-Element gewählt werden würde, dann hat die erste erste Folge nun das Pivot-Element 1 und in der zweiten Folge währe es das Element 7.

64.7 Alternative: Zerlegung mit while-schleifen

Man wählt zuerst ein Pivotelement, beispielsweise das mittlere Element. Nun beginnt man von unten an und vergleicht die Einträge mit dem Pivot. Danach beginnt man von oben und vergleicht die Elemente mit dem Pivot. Wenn ein Element kleiner bzw. größer ist als das Pivot Element, dann wird dieses Element getauscht.

64.8 Analyse

64.8.1 Theorem der Terminierung

Das Theorem der Terminierung besagt, dass der Algorithmus quickSort für jede Eingabe int[]F nach endlicher Zeit terminiert.

Beweis

In jedem rekursiven Aufruf von quickSort ist die Eingabelänge um mindestens 1 kleiner als vorher und die Rekursionsanfang ist erreicht wenn die Länge gleich 1 ist. In der Methode split gibt es nur eine for-Schleife, dessen Zähler j in jedem Durchgang inkrementiert wird. Da u<o wird die for-Schleife also nur endlich oft durchlaufen.

64.8.2 Theorem der Korrektheit

Das Theorem der Korrektheit besagt, dass der Algorithmus quickSort das Problem des vergleichsbasierten Sortierend löst.

Beweis

Die Korrektheit der Methode swap ist zunächst offensichtlich. Zeige nun, dass nach Aufruf pn=split(f,u,o,p) für u<o und $p \in [u, o]$ gilt:

- f[p] wurde zu f[pn] verschoben

Dies ist klar (vorletzte Zeile der Methode split)

- f[i] ≤f[pn] für i=u,...,pn-1

pn wird zu anfangs mit u initialisiert und immer dann inkrementiert, wenn die Position f[pn] durch ein Element, das kleiner/gleich dem Pivot-Element ist, belegt wird.

- f[i] > f[pn] für i=pn+1,...,o

Folgt aus der Beobachtung, dass in 2.) immer „genau dann“gilt. Beachte zudem, dass Element immer getauscht werden, also die Elemente im Array stets dieselben bleiben.

Die Korrektheit der Methode quickSort folgt nach Induktion nach der Länge von f (n=f.length):

- n=1: Der Algorithmus terminiert sofort und ein einelementiges Array ist stets sortiert
- n→n+1: Nach Korrektheit von split steht das Pivot-Element an der richtigen Stelle und links und rechts stehen jeweils nur kleinere/größere Element. Die beiden rekursiven Aufrufe von quickSort erhalten jeweils ein Array, das echt kleiner als n+1 ist (mindestens das Pivot-Element ist nicht mehr Teil des übergebenen Arrays). Nach Induktionsannahme folgt die Korrektheit von quickSort.

64.8.3 Theorem der Laufzeit

Das Theorem der Laufzeit besagt, dass wenn als Pivot Element stets der Median des aktuell betrachteten Arrays gewählt wird, so hat der Algorithmus quickSort eine Laufzeit von $\Theta(n\ log\ n)$.

Beweis

Es gilt zunächst, dass split $\in \Theta(n)(mit\ n = o - u)$. Ausschlaggebend ist hier die for-Schleife, die genau n-mal durchlaufen wird. Gilt nach dem Aufruf von split stets pn=(u+o)/2 (dies ist gleichbedeutend damit, dass das Pivot-Element stets der Median ist), so erhalten wir folgende Rekursionsgleichung für quickSort:

$$T(n) := \begin{cases} \Theta(1) & \text{für } n \leq 1 \\ 2T((n-1)/2) + \Theta(n) & sonst \end{cases}$$

Die ist fast dieselbe Rekursionsgleichung wie für MergeSort und es folgt $T(n) \in \Theta(n\ log\ n)$.

Doch was ist, wenn die Voraussetzung des Theorems nicht erfüllt ist und wir ungleiche Rekursionsaufrufe haben?

64.8.4 Theorem der Laufzeit 2

Das Theorem der Laufzeit besagt, dass der Algorithmus quickSort im schlechtesten Fall eine Laufzeit von $\Theta(n^2)$ hat.

Beweis

Angenommen, die Aufteilung erzeugt ein Teilarray mit Länge n-1 und ein Teilarray mit Länge 0 (Pivot-Element ist also immer Minimum oder Maximum), dann erhalten wir folgende Rekursionsgleichung für die Laufzeit:

$$T(n) := \begin{cases} \Theta(1) & \text{für } n \leq 1 \\ 2T(n-1) + \Theta(n) & sonst \end{cases}$$

Durch Induktionsbeweis kann leicht gezeigt werden, dass $T(n) \in \Theta(n^2)$. Dies ist auch tatsächlich der schlechteste Fall.

Für den mittleren Fall kann gezeigt werden, dass quickSort einen Aufwand von $\Theta(n\ log\ n)$ hat (wie im besten Fall), die in Θ versteckten Konstanten sind nur etwas größer.

64.9 Bemerkung

Im Gegensatz zu MergeSort ist QuickSort durch die Vorgehensweise bei Vertauschungen instabil, d.h. relative Reihenfolge gleicher Schlüssel werden nicht notwendigerweise beibehalten.

64.10 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5.2.6 zu finden.

2 http://www.dpunkt.de/buecher/3358.html

65 Untere Schranke

Auf dieser Seite wird die untere Schranke für vergleichbare Sortierverfahren behandelt.

65.1 Eigenschaften der betrachteten Algorithmen

Die Komplexität im Durchschnittsfall und im Schlechtesten Fall ist nie besser als n ·log n. Die Sortierung erfolgt ausschließlich durch Vergleich der Eingabe-Elemente (comparison sorts), es handelt sich somit um vergleichsorientierte Sortierverfahren. Nun zeigen wir, dass n ·log n Vergleiche eine untere Schranke für „Comparison Sort"-Algorithmen ist. Dies heißt dann, dass Sortieralgorithmen mit Komplexität (schlechtester Fall) von n ·log n (z.B. MergeSort) asymptotisch optimal sind.

65.2 Problembeschreibung

Zuerst die Problembeschreibung. Als Eingabe haben wir $\langle a_1, a_2, ..., a_n \rangle$. Als Vergleichstests nehmen wir $a_i < a_j, a_i \leq a_j, a_i \equiv a_j, a_i \geq a_j, a_i > a_j$. Als vereinfachte Annahmen nehmen wir an, dass es nur verschiedene Elemente gibt, somit entfällt $a_i \equiv a_j$. Die restlichen Test liefern alle gleichwertige Informationen. Sie bestimmen die Reihenfolge von a_i *und* a_j. Außerdem können sie und auf $a_i \leq a_j$ beschränken. Somit haben wir eine binäre Entscheidung und es gilt entweder $a_i \leq a_j$ *oder* $a_i > a_j$

65.3 Entscheidungsbaum

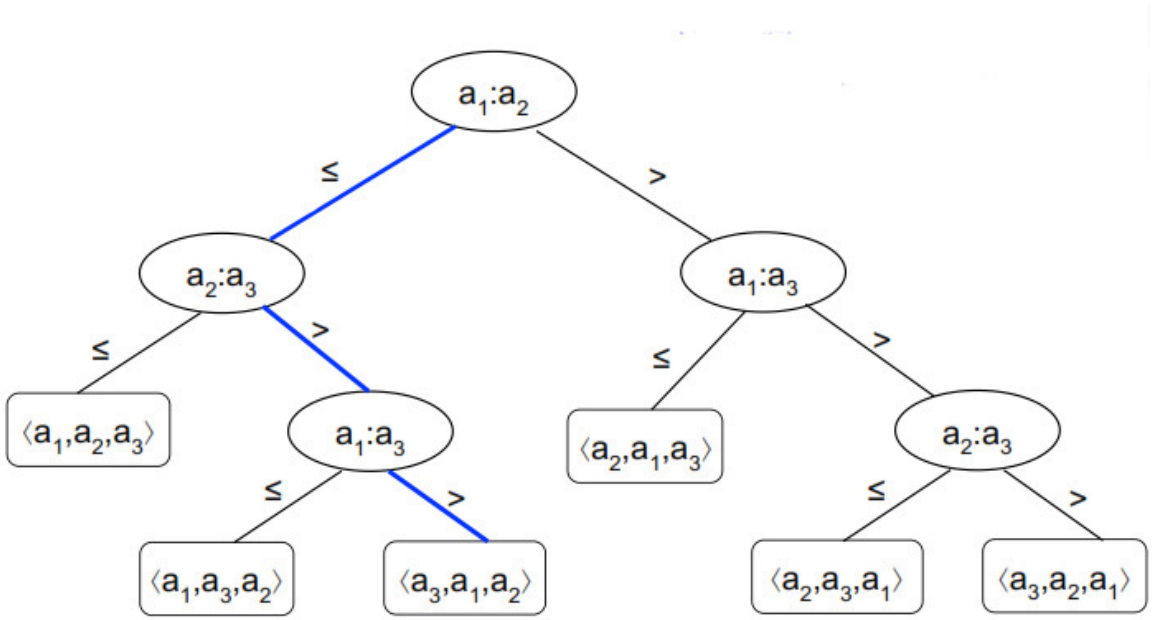

Abb. 18 Entscheidungsbaum

Eine beispielhafte Eingabe ist $a_1 = 6, a_2 = 8, a_3 = 5$ Die inneren Knoten vergleichen die Elemente a_i *und* a_j. Es wird ein Test durchgeführt ob $a_i \leq a_j$ gilt oder nicht. Die Blätter sind Permutationen mit $\langle \pi(a_1), ..., \pi(a_n) \rangle$ Sortieren heißt das Finden eines Pfades von der Wurzel zu einem Blatt. An jedem internen Knoten erfolgt ein Vergleich und entsprechend wird links oder rechts weiter gesucht. Ist ein Blatt erreicht, dann hat der Sortieralgorithmus eine Ordnung und die Permutation der Elemente ist erstellt. Daraus lässt sich schlussfolgern, dass jeder Sortieralgorithmus jede Permutation der n Eingabe-Elemente erreichen muss (n!). Daraus folgt wiederum, dass es n! Blätter geben muss, die alle von der Wurzel erreichbar sind. Andernfalls kann er zwei unterschiedliche Eingaben nicht unterscheiden und liefert für beide dasselbe Ergebnis und eins davon muss falsch klassifiziert sein. Die Anzahl an Vergleichen im schlechtesten Fall ist die Pfadlänge von Wurzel bis Blatt, oder auch Höhe genannt.

```
Somit erhalten wir das Theorem, dass jeder vergleichsorientierte
Sortieralgorithmus im schlechtesten Fall mindestens n*log n Versuche braucht.
```

65.3.1 Beweis

Gegeben ist die Anzahl der Elemente n, h die Pfadlänge bzw. Höhe des Baums und b die Anzahl der Blätter. Jede Permutation muss in einem Blatt sein, das bedeutet $n! \leq b$. Der Binärbaum hat die Höhe h und maximal 2^h Blätter, daraus folgt $n! \leq b \leq 2^h$. Wenn man nun logarithmiert, erhält man

$h \geq log_2(n!)$

$$\sim n \cdot log_2(n)$$

$$(genauer \ = \Omega(n \cdot log_2(n)))$$

65.4 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 5.2.7 zu finden.

1 `http://www.dpunkt.de/buecher/3358.html`

66 Dynamische Datenstrukturen

Auf dieser Seite wird es eine Einführung in die dynamischen Datenstrukturen[1] geben. Unter dynamischen Datenstrukturen verstehen wir Datenstrukturen bei denen man Elemente löschen und hinzufügen kann, eine interne Ordnung (z.B. Sortierung) vorliegt und diese Ordnung unter Änderungen aufrecht erhalten bleibt. Ein Beispiel sind Lineare Datenstrukturen und Sortierung. Bei unsortierte Liste sind Änderung einfach, aber Zugriff aufwändig. Bei einer Neusortierung einer Liste sind Änderung schwierig, aber Zugriff einfach. Bei Trade-of ist eine "intelligente Datenstruktur" gesucht, die Änderungen und Zugriffe einfach, sprich effizient, halten. Viele dynamische Datenstrukturen nutzen Bäume als Repräsentation.

66.1 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 8.5 zu finden.

1 https://de.wikipedia.org/wiki/Dynamische_Datenstruktur
2 http://www.dpunkt.de/buecher/3358.html

67 Bäume

In diesem Kapitel werden Bäume[1] als kurzen Einschub behandelt behandelt. Ein Baumelement e ist ein Tupel $e = (v, \{e_1, ..., e_n\})$ mit v vom Wert e und $\{e_1, ..., e_n\}$ sind die Nachfolger, bzw. Kinder von e. Ein Baum T ist ein Tupel $T = (r, \{e_1, ..., e_n\})$ mit r als Wurzelknoten (ein Baumelement) und $\{e_1, ..., e_n\}$ als Knoten (Baumelemente) des Baumes mit $r \in \{e_1, ..., e_n\}$ und für alle $e_i = (v_i, K_i)$ *und* $e_j = (v_j, K_j) \in \{e_1, ..., e_n\}$ *gilt* $K_i \bigcap K_j =$

Man spricht von einem geordneten Baum, wenn die Reihenfolge der Kinder $\{e_1, .., e_n\}$ eines jeden Elements $e = (v, \{e_1, ..., e_n\})$ festgelegt ist (schreibe dann $(e_1, ..., e_n)$ statt $\{e_1, ..., e_n\}$).

67.1 Beispiel

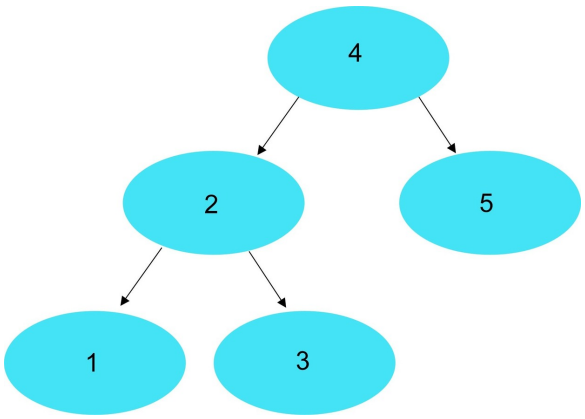

Abb. 19 Binärbaum Beispiel 1

$T = (v_4, \{v_1, v_2, v_3, v_4, v_5\})$

$v_1 = (1, \{\})$

$v_2 = (2, \{v_1, v_3\})$

$v_3 = (3, \{\})$

$v_4 = (4, \{v_2, v_5\})$

$v_5 = (5, \{\})$

1 https://de.wikipedia.org/wiki/Baum_%28Graphentheorie%29

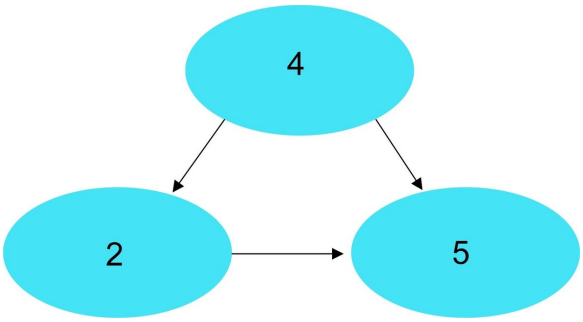

Abb. 20 Binärbaum Beispiel 2

$T' = (v_4, \{v_2, v_5\})$

$v_2 = (2, \{v_5\})$

$v_4 = (4, \{v_2, v_5\})$

$v_5 = (5, \{\})$

T' ist kein Baum, da v_4 *und* v_2 ein gemeinsames Kind haben.

67.2 Begriffe

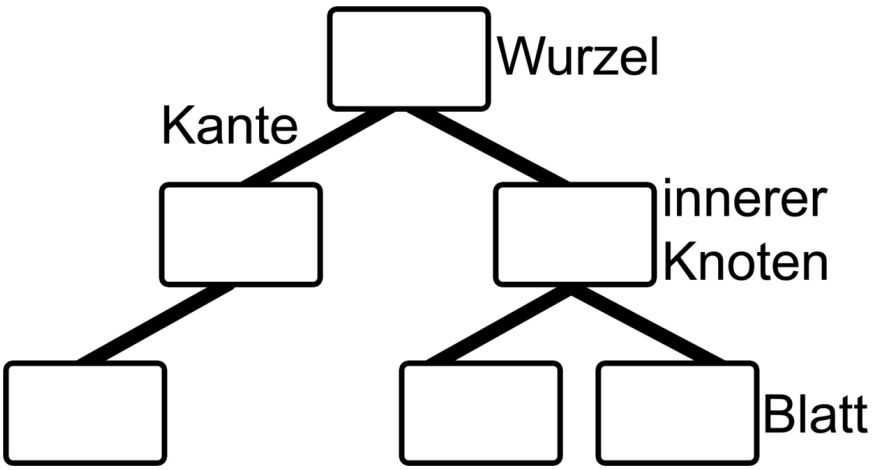

Abb. 21 BinärBaum Beschriftung

Ein Pfad folgt über Kanten zu verbundenen Knoten, dabei existiert zu jedem Knoten genau ein Pfad von der Wurzel. Ein Baum ist immer zusammenhängend und zyklenfrei.

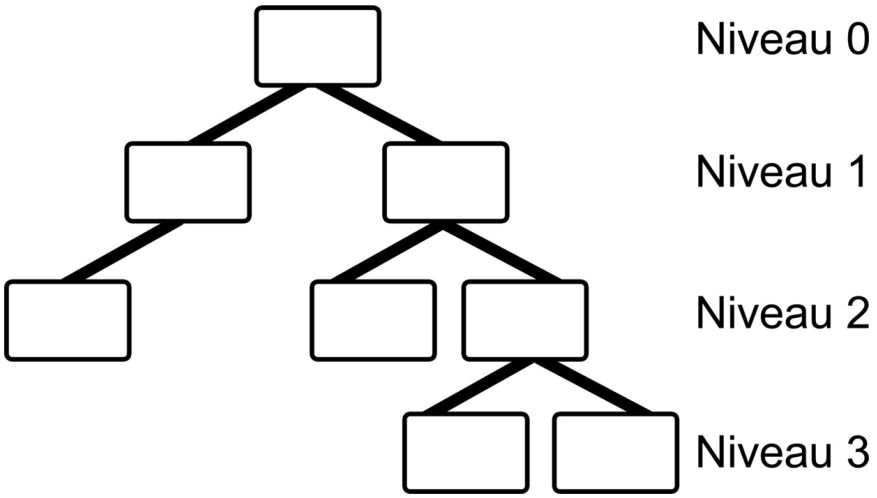

Abb. 22 BinärBaum Niveau

Das Niveau der jeweiligen Ebene entspricht immer der jeweiligen Länge des Pfades. Die Höhe eines Baumes entspricht dem größten Niveau+1.

67.3 Anwendungen

Man benutzt Bäume beispielsweise zur Darstellung von Hierarchien, wie Taxonomien, oder für Entscheidungsbäume. Bäume werden oft genutzt um sortierte, dynamische oder lineare Datenstrukturen zu repräsentieren, da Einfüge-und Löschoperationen leicht so definiert werden können, dass die Sortierung beibehalten wird. Ein Baum kann auch als Datenindex genutzt werden und stellt so eine Alternative zu Listen und Arrays dar.

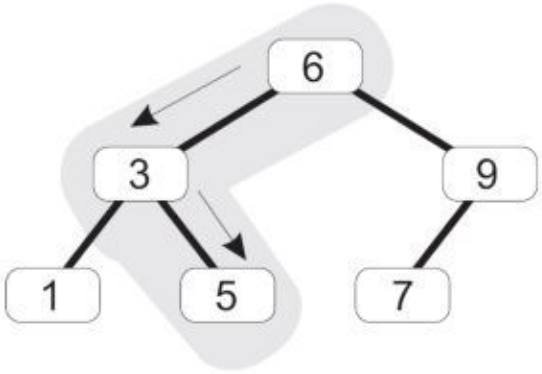

Abb. 23 Suchbaum

Hier wird beispielsweise nach der 5 gesucht und der Baum wird als Suchbaum genutzt.

Man kann auch einen Baum aus Termen bilden. Der Term (3+4) * 5 + 2 * 3 gibt folgenden Baum:

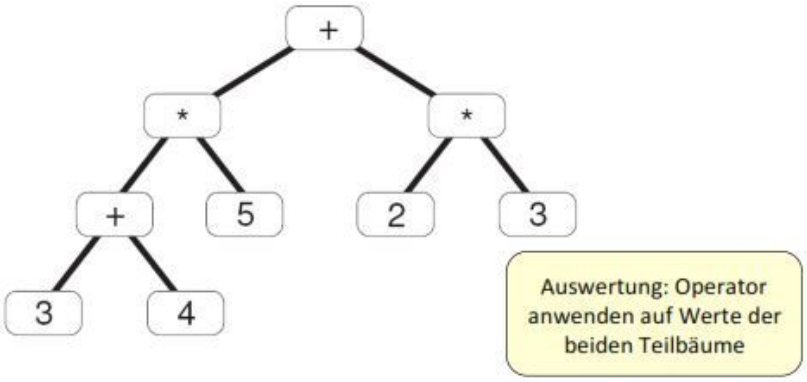

Abb. 24 Termbaum

67.4 Atomare Operationen auf Bäumen

Zu den Operationen zählen **lesen** mit

- root(): Wurzelknoten eines Baums
- get(e): Wert eines Baumelements e
- children(e): Kinderknoten eines Elements e
- parent(e): Elternknoten eines Elements e

und **schreiben** mit

- set(e,v): Wert des Elements e auf v setzen
- addChild(e,e'): Füge Element e' als Kind von e ein (falls geordneter Baum nutze addChild(e,e',i) für Index i)
- del(e): Lösche Element e (nur wenn e keine Kinder hat)

67.5 Spezialfall: Binärer Baum als Datentyp

```
class TreeNode<K extends Comparable<K>>{

       K key;
       TreeNode<K> left = null;
       TreeNode<K> right = null;

       public TreeNode(K e) {key = e; }
       public TreeNode<K> getLeft() {return left; }
       public TreeNode<K> getRight()  {return right; }
       public K getKey() {return key; }

       public void setLeft(TreeNode<K> n) {left = n;}
       public void setRight(TreeNode<K> n) {right = n;}

         ...

 }
```

67.5.1 Beispiel

TreeNode<Character> root = new TreeNode<Character>(‘A‘);

TreeNode<Character> node1 = new TreeNode<Character>(‘B‘);

TreeNode<Character> node2 = new TreeNode<Character>(‘C‘);

TreeNode<Character> node3 = new TreeNode<Character>(‘D‘);

root.setLeft(node1);

root.setRight(node2);

node2.setLeft(node3);

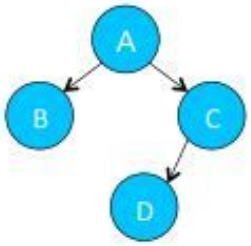

Abb. 25 Beispiel Tree

67.6 Typische Problemstellungen

Als typische Problemstellung haben wir zum einen die Traversierung, zum Anderen das Löschen eines inneren Knotens und die daraus folgende Re-strukturierung des Baumes und das Suchen in Bäumen.

67.6.1 Traversierung

Bäume können visuell gut dargestellt werden. Manchmal ist jedoch eine Serialisierung der Elemente eines Baumes nötig. Man kann die Elemente eines Baumes durch Preorder-Aufzählung, Inorder-Aufzählung, Postorder-Aufzählung oder Levelorder-Aufzählung eindeutig aufzählen.

Bei der Traversierung werden systematisch alle Knoten des Baumes durchlaufen.

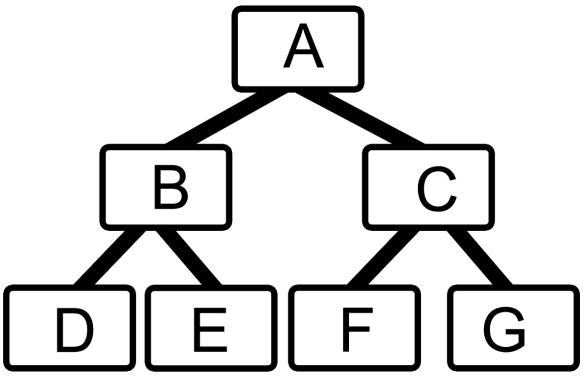

Abb. 26 Traversierung

Preorder (W-L-R): $A \to B \to D \to E \to C \to F \to G$

Inorder (L-W-R): $D \to B \to E \to A \to F \to C \to G$

Postorder (L-R-W): $D \to E \to B \to F \to G \to C \to A$

Levelorder: $A \to B \to C \to D \to E \to F \to G$

Traversierung mit Iteratoren

Bei der Traversierung sind Iteratoren erlaubt. Diese werden schrittweise abgearbeitet und es werden Standardschleifen für die Baumdurchläufe verwendet.

```
for  (Integer i : tree)
            System.out.print(i);
```

Dabei ist es allerdings notwendig, dass der Bearbeitungszustand zwischengespeichert wird.

```
public class BinarySearchTree<K extends Comparable<K>>
      implement Iterable<K> {

   public static final int INORDER = 1;
   public static final int PREORDER = 2;
   public static final int POSTORDER = 3;
   public static final int LEVELORDER = 4;

   private int iteratorOrder;
    ...

  public void setIterationOrder(int io) {
      if (io < i || io > 4)
          return;
      iteratorOrder = io;
  }
 public Iterator<K> iterator() {
     switch (iterationOrder) {
         case INORDER:
            return new InorderIterator<K>(this);
         case PRORDER:
            return new PreorderIterator<K>(this);
         case POSTORDER:
            return new PostorderIterator<K>(this);
         case LEVELORDER:
            return new LevelorderIterator<K>(this);
```

```
        default:
           return new InorderIterator<K>(this);
    }
}
```

Preorder Traversierung

Bei der Preorder Traversierung wird der aktuelle Knoten zuerst behandelt und dann der linke oder rechte Teilbaum.

```
static class TreeNode<K extends Comparable<K>> {
  ...
  public void traverse() {
      if (key==null)
          return;
      System.out.print(" " + key);
      left.traverse();
      right.traverse();
  }
```

Preorder Iteratoren

Der Wurzelknoten wird auf den Stack gelegt, anschließend der rechte Knoten und dann der linke Knoten.

```
class PreorderIterator<K extends Comparable <K>>
        implements Iterator<K> {

    java.util.Stack<TreeNode<K>> st =
        new java.util.Stack<TreeNode<K>>();

    public PreorderIterator(BinarySearchTree<K> tree){
          if (tree.head.getRight() != nullNode)
               st.push(tree.head.getRight());

    }
  public boolean hasNext() {
         return !st.isEmpty();
  }

   public K next(){
       TreeNode<K> node = st.pop();
       K obj = node.getKey();
       node = node.getRight();
       if(node != nullNode) {
          st.push(node);  //rechten Knoten auf den Stack
       }
       node = node.getLeft();
       if(node != nullNode) {
          st.push(node);  //linken Knoten auf den Stack
       }
       return obj;
   }
}
```

Inorder Traversierung

Bei der Inorder Traversierung wird zuerst der linke Teilbaum behandelt, dann der aktuelle Knoten und dann der rechte Teilbaum. Als Ergebnis erhält man den Baum in sortierter Reihenfolge.

```
static class TreeNode<K extends Comparable<K>> {
  ...
  public void traverse() {
      if (key==null)
          return;
      left.traverse();
      System.out.print(" " + key);
      right.traverse();
  }
```

Inorder Iteratoren

Der Knoten head hat immer einen rechten Nachfolger. Es wird vom Wurzelknoten begonnen alle linken Knoten auf den Stack zu legen.

```
 class InorderIterator<K extends Comparable <K>>
         implements Iterator<K> {

     java.util.Stack<TreeNode<K>> st =
         new java.util.Stack<TreeNode<K>>();

     public InorderIterator(BinarySearchTree<K> tree) {
           TreeNode<K> node = tree.head.getRight();
           while (node != nullNode) {
                st.push(node);
                node = node.getLeft();
           }
    }
  public boolean hasNext() {
          return !st.isEmpty();
  }

   public K next(){
       TreeNode<K> node = st.pop();
       K obj = node.getKey();
       node = node.getRight();  //rechten Knoten holen
       while (node != nullNode) {
          st.push(node);
          node = node.getLeft();  //linken Knoten auf den Stack
       }
       return obj;
   }

}
```

Postorder Traversierung

Bei der Postorder Traversierung wird zuerst der linke und der rechte Teilbaum behandelt und dann der aktuelle Knoten. Dies kann beispielsweise genutzt werden, um einen Baum aus Termen, entsprechend der Priorität der Operatoren, auszuwerten.

```
static class TreeNode<K extends Comparable<K>> {
  ...
```

```
  public void traverse() {
      if (key==null)
          return;
      left.traverse();
      right.traverse();
      System.out.print(" " + key);
  }
```

Levelorder Iteratoren

Der Wurzelknoten wird in der Warteschlange eingefügt. Dann wird zuerst der linke und dann der rechte Knoten in die Warteschlange eingefügt. In dieser Implementierung wird die queue als LinkedList repräsentiert. Dies ist jedoch beliebig.

```
 class LevelorderIterator<K extends Comparable <K>>
         implements Iterator<K> {

   //Wurzelknoten in die Warteschlange (queue) einfügen
   java.util.Queue<TreeNode<K>> q =
         new java.util.LinkedList<TreeNode<K>>();

   public LevelorderIterator(BinarySearchTree<K> tree){
         TreeNode<K> node = tree.head.getRight();
         if (node != nullNode)
                q.addLast(node);}

   public K next(){
       TreeNode<K> node = q.getFirst();
       K obj = node.getKey();
       if (node.getLeft() != nullNode)
            q.addLast(node.getLeft());
       if (node.getRight() != nullNode)
            q.addLast(node.getRight());
       return obj;
   }
}
```

67.7 Bäume in Java

In Java gibt es keine hauseigene Implementierung für allgemeine Bäume. Einige Klassen (TreeMap, TreeSet) benutzen Bäume zur Realisierung anderer Datenstrukturen. Andere Klassen (JTree) benutzen Bäume als Datenmodell zur Visualisierung.

67.8 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 14 zu finden.

2 http://www.dpunkt.de/buecher/3358.html

68 Binäre Suchbäume

Auf dieser Seite werden die binären Suchbäume[1] behandelt. Er ermöglicht einen schneller Zugriff auf Daten mit dem Aufwand O(log n) unter geeigneten Voraussetzungen. Des weiteren ermöglicht er effiziente Sortierung von Daten, durch Heapsort und effiziente Warteschlangen. Der binäre Suchbaum dient als Datenstruktur für kontextfreie Sprachen. In der Computergrafik sind Szenengraphen oft (Beinahe-)Bäume. Bei Informationssysteme dienen binäre Suchbäume zur Datenindizierung und Anfrageoptimierung.

68.1 Operationen

Auf Suchbäumen können die Operationen Suchen von Elementen, Einfügen von Elementen und Entfernen von Elementen angewandt werden, wobei letztere zwei voraussetzen, dass die Ordnung der Schlüssel erhalten bleibt.

68.2 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 14 zu finden.

1 https://de.wikipedia.org/wiki/Bin%C3%A4rer_Suchbaum
2 http://www.dpunkt.de/buecher/3358.html

69 Suchen

Ein binärer Suchbaum kann für viele Anwendungen eingesetzt werden.

Hier ist der Baum ein Datenindex und eine Alternative zu Listen und Arrays. Beispielsweise kann dieser Baum als Suchbaum verwendet werden und nach ”5” gesucht werden.

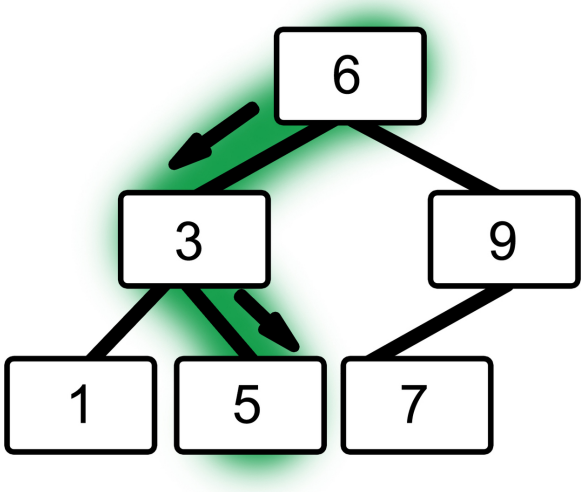

Abb. 27 BinärBaum Suchbaum

Bei der Anwendung von Bäumen zur effizienten Suche gibt es pro Knoten einen Schlüssel und ein Datenelement und die Ordnung der Knoten erfolgt anhand der Schlüssel. Bei einem binärer Suchbaum enthält der Knoten k einen Schlüsselwert k.key. Alle Schlüsselwerte im linken Teilbaum k.left sind kleiner als k.key und alle Schlüsselwerte im rechten Teilbaum k.right sind größer als k.key. Die Auswertung eines Suchbaums sieht wie folgt aus:

1. Vergleich des Suchschlüssels mit Schlüssel der Wurzel
2. Wenn kleiner, dann in linken Teilbaum weiter suchen
3. Wenn größer, dann in rechtem Teilbaum weiter suchen
4. Sonst gefunden/nicht gefunden

```
static class  TreeNode<K extends Comparable<K>>{

      K key;
      TreeNode<K> left = null;
      TreeNode<K> right = null;

      public TreeNode(K e) {key = e; }
      public TreeNode<K> getLeft() {return left; }
      public TreeNode<K> getRight()  {return right; }
      public K getKey() {return key; }
```

```
      public void setLeft(TreeNode<K> n) {left = n;}
      public void setRight(TreeNode<K> n) {right = n;}

        ...

}
```

69.1 Knotenvergleich

```
class  TreeNode<...> {
         ...

     public int compareKeyTo(K k) {
          return (key == null ? -1:
                       key.compareTo(k));
     }
         ...


}
```

69.2 Rekursives Suchen

```
protected  TreeNode<K>
         recursiveFindNode(TreeNode<K>  n, k){
          /* k wird gesucht */

          if (n!= nullNode) {
              int cmp = n.compareKeyTo(k.key);
              if (cmp == 0)
                  return n;
              else if (cmp > 0)
                  return
                     recursiveFindNode(n.getLeft(),k);
              else
                  return
                     recursiveFindNode(n.getRight(),k);
          }
          else
                return null;
}
```

69.3 Iteratives Suchen

```
protected  TreeNode<K> iterativeFindNode(TreeNode<K> k){
          /* k wird gesucht */
          TreeNode<K> n = head.getRight();
          while (n!= nullNode) {
              int cmp = n.compareKeyTo(k.key);
              if (cmp == 0)
                  return n;
              else
                  n = (cmp > 0 ?
                      n.getLeft(): n.getRight());
```

```
          }
          return null;
 }
```

69.4 Suchen des kleinsten Elements

```
protected  K findMinElement(){
          TreeNode<K> n = head.getRight();
          while (n.getLeft() != nullNode)
              n = n.getLeft();
          return n.getKey();
}
```

69.5 Suchen des größten Elements

```
protected  K findMaxElement(){
          TreeNode<K> n = head.getRight();
          while (n.getRight() != nullNode)
              n = n.getRight();
          return n.getKey();
}
```

Eine weitere Anwendungsmöglichkeit ist der Baum aus Termen. Wir haben den Term $(3+4)\cdot 5+2\cdot 3$ als Baumdarstellung sieht es so aus:

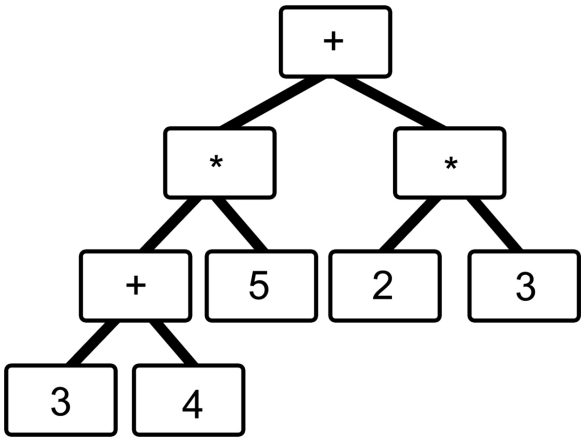

Abb. 28 BinärBaum Term

Bei der Auswertung müssen die Operatoren auf die beiden Werte der Teilbäume angewandt werden.

69.6 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 14.3 zu finden.

1 http://www.dpunkt.de/buecher/3358.html

70 Einfügen

Das Finden der Einfügeposition erfolgt durch Suchen des Knotens, dessen Schlüsselwert größer als der einzufügende Schlüssel ist und der keinen linken Nachfolger hat oder durch Suchen des Knotens, dessen Schlüsselwert kleiner als der einzufügende Schlüssel ist und der keinen rechten Nachfolger hat. Das Einfügen erfolgt prinzipiell in 2 Schritten. Im ersten Schritt wird die Einfügeposition gesucht, sprich der Blattknoten mit dem nächstkleineren oder nächstgrößerem Schlüssel. Im zweiten Schritt wird ein neuer Knoten erzeugt und als Kindknoten des Knotens aus Schritt eins verlinkt. Wenn in Schritt eins der Schlüssel bereits existiert, dann wird nicht erneut eingefügt.

70.1 Programm in Java

```
 /* Einfügeposition suchen */
public boolean  insert(K k){
          TreeNode<K> parent = head;
          TreeNode<K> child = head.getRight();
          while (child != nullNode) {
              parent = child;
              int cmp = child.compareKeyTo(k);
              //Schlüssel bereits vorhanden
              if (cmp == 0)
                  return false;
              else if (cmp > 0)
                  child = child.getLeft();
              else
                  child = child.getRight();
           }
/* Neuen Knoten verlinken */
  TreeNode<K> node = new TreeNode<K>(k);
            node.setLeft(nullNode);
            node.setRight(nullNode);
            if (parent.compareKeyTo(k) > 0)
                  parent.setLeft(node);
            else
                  parent.setRight(node);
            return true;
```

70.2 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 14.3.2 zu finden.

1 http://www.dpunkt.de/buecher/3358.html

71 Löschen

Zuerst wird das zu löschendes Element gesucht, der Knoten k. Nun gibt es drei Fälle

1. k ist Blatt: löschen
2. k hat ein Kind: Kind „hochziehen“
3. k hat zwei Kinder: Tausche mit weitest links stehenden Kind des rechten Teilbaums, da dieser in der Sortierreihenfolge der nächste Knoten ist und entferne diesen nach den Regeln 1. oder 2.

Ein Schlüssel wird in drei Schritten gelöscht. Im ersten Schritt wird der zu löschende Knoten gefunden. Im zweiten Schritt wird der Nachrückknoten gefunden. Dafür gibt es mehrere Fälle. Im Fall 1 handelt es sich um einen externen Knoten, sprich ein Blatt, ohne Kinder. Dabei wird der Knoten durch einen nullNode ersetzt. Im Fall 2a gibt es nur einen rechten Kindknoten, dabei wird der gelöschte Knoten durch den rechten Kindknoten ersetzt. Im Fall 2b gibt es nur einen linken Kindknoten und der gelöschte Knoten wird durch diesen ersetzt. Im Fall 3 gibt es einen internen Knoten mit Kindern rechts und links. Dabei wird der gelöschte Knoten durch den Knoten mit dem kleinstem (alternativ größtem) Schlüssel im rechten (alternativ linken) Teilbaum ersetzt. im dritten und letzten Schritt wird nun der Baum reorganisiert. Während dem Löschen kann sich die Höhe von Teilbäumen ändern.

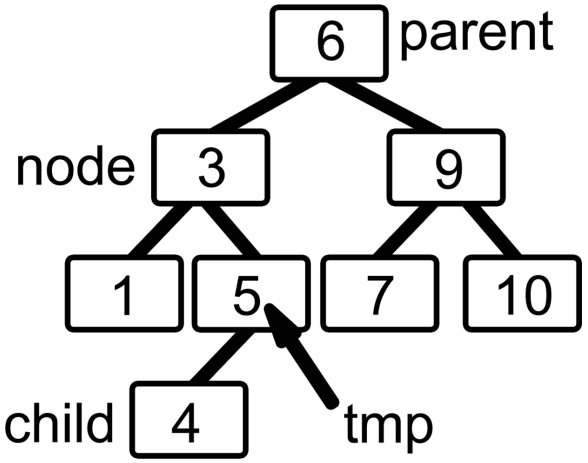

Abb. 29 BinärBaum Einfügen

71.1 Programm in Java

```
 /* Knoten suchen */
public boolean  remove(K k){
          TreeNode<K> parent = head;
```

```
          TreeNode<K> node = head.getRight();
          TreeNode<K> child = null;
          TreeNode<K> tmp = null;
          while (node != nullNode) {
              int cmp = node.compareKeyTo(k);
              //Löschposition gefunden
              if (cmp == 0)
                 break;
              else {
                 parent = node;
                 node = (cmp > 0 ?
                      node.getLeft() : node.getRight());
              }
          }
          //Knoten k nicht im Baum
          if (node == nullNode)
             return false;
/* Nachrücker finden */
    if (node.getLeft() == nullNode  &&
            Node.getRight() == nullNode)  //Fall 1
          child = nullNode;
     else if (node.getLeft() == nullNode)  //Fall 2a
           child = node.getRight();
     else if (node.getRight() == nullNode)  //Fall 2b
           child = node.getLight();
          ...
  //Fall 3
  else {
       child = node.getRight();
       tmp = node;
       while (child.getLeft() != nullNode) {
            tmp = child;
            child = child.getLeft();
       }
       child.setLeft(node.getLeft());
       if (tmp != node) {
            tmp.setLeft(child.getRight());
            child.setRight(node.getRight());
       }
  }
/* Baum reorganisieren */
        if (parent.getLeft() == node)
                 parent.setLeft(child)
        else
                 parent.setRight(child);
        return true;
      ...
```

71.2 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 14.3.2 zu finden.

1 http://www.dpunkt.de/buecher/3358.html

72 Implementierung

Ein binärer Suchbaum ist eine häufig verwendete Hauptspeicherstruktur und ist besonders geeignet für Schlüssel fester Größe, z.B. numerische wie int, float und char[n]. Der Aufwand von O(log n) für Suchen, Einfügen und Löschen ist garantiert, vorausgesetzt der Baum ist balanciert. Später werden wir lernen, dass die Gewährleistung der Balancierung durch spezielle Algorithmen gesichert wird. Des weiteren sind größere, angepasste Knoten für Sekundärspeicher günstiger, diese nennt man B-Bäume. Für Zeichenketten benutzt man als Schlüssel variable Schlüsselgröße, sogenannte Tries.

```
public class
      BinarySearchTree<K extends Comparable<K>>
           implements Iterable<K> {

          ...

    static class TreeNode<K extends Comparable<K>> {
         K key;
         TreeNode<K> left = null;
         TreeNode<K> right = null;

         ...

    }
 }
```

Die Schlüssel müssen Comparable-Interface, d.h. compareTo()-Methode, implementieren, da der Suchbaum auf Vergleichen der Schlüssel basiert. Der Baum selbst implementiert Iterable-Interface, d.h. iterator()-Methode, um Traversierung des Baums über Iterator zu erlauben (später Baumtraversierung).TreeNode und alles weitere werden als innere Klassen implementiert. Dadurch werden Zugriff auf Attribute und Methoden der Baumklasse erlaubt. Eine Besonderheit der Implementierung sind die „leeren“ Pseudoknoten head und nullNode zur Vereinfachung der Algorithmen (manchmal „Wächter“ / „sentinel“ genannt). Grundlegende Algorithmen sind

- Suchen
- Einfügen
- Löschen

72.1 Implementierung mit Pseudoknoten

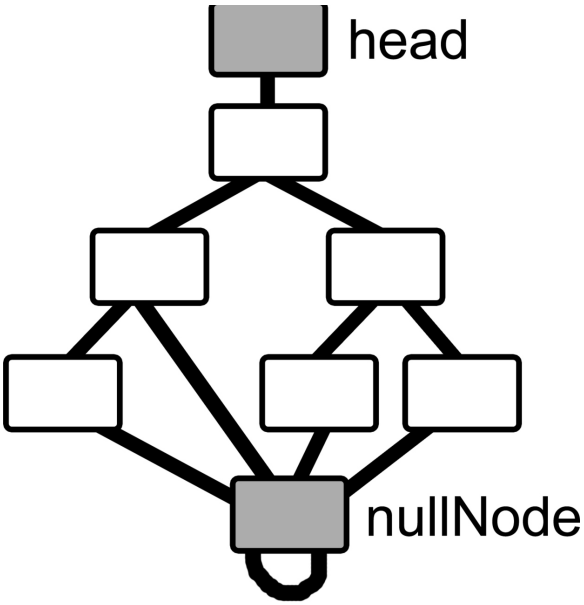

Abb. 30 BinärBaum Pseudoknoten

Wir vereinbaren an dieser Stelle, dass man auf dem Kopf kein getRight() anwenden kann.

```
public class
      BinarySearchTree<K extends Comparable<K>>
           implements Iterable<K> {

          ...

       pulic BinarySearchTree(){
            head = new TreeNode<K>(null);
            nullNode = new TreeNode<K>(null);
            nullNode.setLeft(nullNode);
            nullNode.setRight(nullNode);
            head.setRight(nullNode);
       }
         ...


 }
```

Das Ziel der Implementierung ist, die Reduzierung der Zahl an Sonderfällen. Im head würde das Einfügen oder Löschen des Wurzelknotens spezielle Behandlung in der Baum-Klasse erfordern. Der nullNode erspart den Test, ob zum linken oder zum rechten Teilknoten navigiert werden kann. Des weiteren ist im NullNode ein einfaches Beenden der Navigation (z.B. Beenden der Rekursion) möglich.

72.2 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[1] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 14.2.1 zu finden.

1 `http://www.dpunkt.de/buecher/3358.html`

73 Weitere Aspekte

Die Komplexität der Operation hängt von der Höhe ab. Der Aufwand für die Höhe des Baumes beträgt O(h). Die Höhe eines ausgeglichenen binären Baumes ist h=ld(n) für Knoten. Bei eines ausgeglichenen oder balancierten Baum unterscheiden sich zum einen der rechte und linke Teilbaum eines jeden Knotens in der Höhe um höchstens 1 und zum anderen unterscheiden sich je zwei Wege von der Wurzel zu einem Blattknoten höchstens in um 1 in der Länge. Rot-Schwarz Bäume und AVL Bäume benötigen einen Ausgleich nach dem Einfügen und Löschen.

73.1 Entartung von Bäumen

Ungünstige Einfüge- oder Löschreihenfolge führt zu extremer Unbalanciertheit. Im Extremfall wird der Baum zur Liste, dann haben die Operationen eine Komplexität von O(n). Beispiel:

```
for (int i = 0; i < 10; i++)
tree.insert(i);
```

Vermeiden kann man dies durch spezielle Algorithmen zum Einfügen und Löschen, z.B. mit Rot-Schwarz-Bäumen und AVL-Bäumen.

73.2 Heaps

74 Heap Sort

Auf dieser Seite wird das Thema Heap Sort[1] behandelt. Von "Heap" gibt es zwei völlig verschiedene Definitionen. Zum einen ist es ein größeres Gebiet im Hauptspeicher, aus dem Programmierer Blöcke beanspruchen und wieder freigeben können und zum anderen ist es ein balancierter, linksbündiger Binärbaum in dem kein Knoten einen Wert hat, der größer ist als der Wert seines Elternknoten. Im Falle von Heapsort wird die zweite Definition benutzt.

74.1 Balancierter Binärbaum

Jeder Knoten ist in einer Ebene platziert, der Wurzelknoten in Ebene 0. Die Höhe eines Baumes ist die Distanz von seiner Wurzel zum weitest entfernten Knoten plus 1. Ein Knoten ist tiefer als ein anderer Knoten, wenn seine Ebene eine höhere Zahl hat. Ein Binärbaum der Höhe h ist balanciert, wenn alle Knoten der Ebenen 0 bis h-3 zwei Kinder haben.

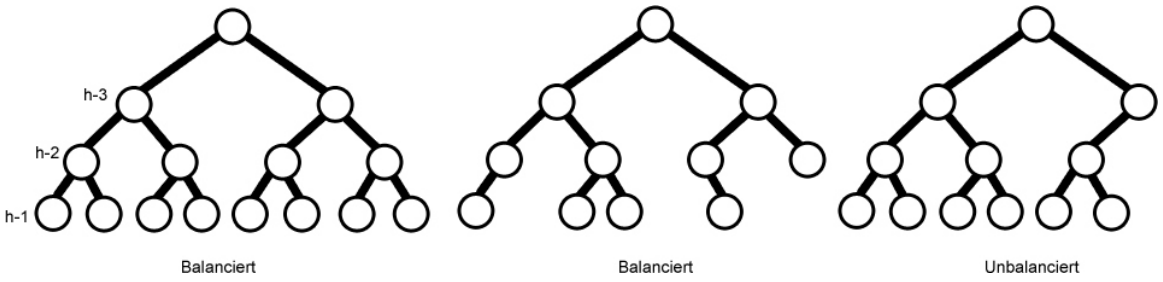

Abb. 31 Balancierter Binärbaum2

Ein balancierter Binärbaum der Höhe h ist linksbündig, wenn er 2^k Knoten in der Ebene k hat für alle $k < h - 1$ und alle Blätter der Ebene h-1 so weit wie möglich links sind.

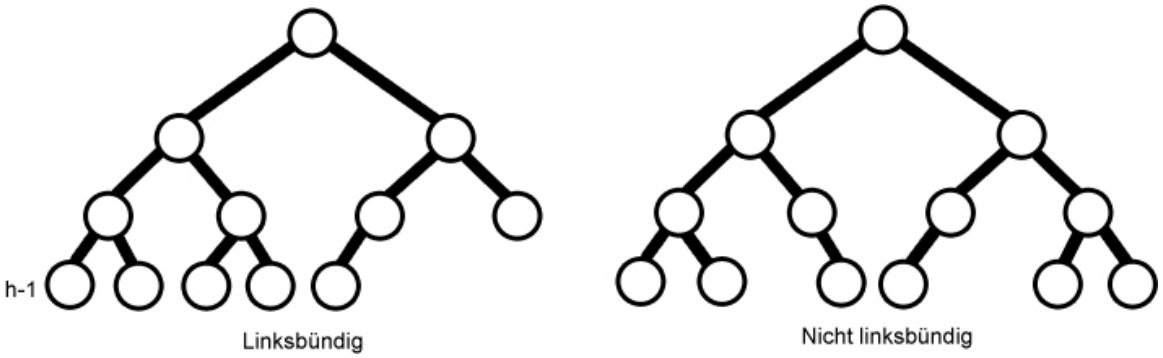

Abb. 32 Linksbündiger Binärbaum

1 https://de.wikipedia.org/wiki/Heap_Sort

74.2 Motivation

Der Vorteil von MergeSort gegenüber QuickSort ist, dass MergeSort einen garantierten Aufwand von O(n log n) hat. Der Vorteil von QuickSort gegenüber MergeSort ist, dass QuickSort n viel Speicher benötigt und MergeSort 2n viel Speicher. Gibt es nun einen Sortieralgorithmus, der n viel Speicher benötigt und garantiert in O(n log n) läuft? Ja HeapSort! Mit HeapSort lassen sich zudem die Warteschlangen mit Prioritäten effizient implementieren. Außerdem ist die Idee des Heaps sehr interessant. Eine komplexe Datenstruktur (Baum) wird in einer einfacheren Struktur (Array) abgebildet.

74.3 Heap Eigenschaft

Ein Knoten hat die Heap-Eigenschaft, wenn der Wert in dem Knoten so groß oder größer ist als die Werte seiner Kinder. Alle Blattknoten haben dann auch automatisch die Heap Eigenschaft. Ein Binärbaum ist nur dann ein Heap, wenn alle Knoten die Heap Eigenschaft besitzen.

Abb. 33 Heap Eigenschaft

74.4 Anmerkung

Ein Heap ist kein binärer Suchbaum. Das Sortierkriterium bei Suchbäumen war, dass der Wert eines Knotens stets größer ist, als die Werte der Knoten, die im linken Teilbaum liegen und, dass der Wert eines Knotens stets kleiner ist, als die Werte der Knoten, die im rechten Teilbaum liegen. Das Sortierkriterium beim Heap ist, dass die Werte eines Knotens stets größer oder gleich der Werte der Knoten sind, die in beiden Teilbäumen liegen.

74.5 Literatur

Da die Vorlesungsinhalte auf dem Buch Algorithmen und Datenstrukturen: Eine Einführung mit Java[2] von Gunter Saake und Kai-Uwe Sattler aufbauen, empfiehlt sich dieses Buch um das hier vorgestellte Wissen zu vertiefen. Die auf dieser Seite behandelten Inhalte sind in Kapitel 14.6.1 zu finden.

74.6 Hashtabellen

2 http://www.dpunkt.de/buecher/3358.html

75 Hashtabellen

Auf dieser Seite wird das Thema Hashtabellen[1] behandelt. Gesucht ist eine dynamische Datenstruktur mit sehr schnellem direktem Zugriff auf Elemente. Die Idee der Hashfunktion ist, dass ein Feld von 0 bis N-1 benutzt wird, beispielsweise ein Array. Die einzelnen Positionen im Feld werden oft als Buckets bezeichnet. Die Hashfunktion h(e) bestimmt für Elemente e die Position im Feld. H(e) ist sehr schnell berechenbar. Es gilt $h(e) \neq h(e')$ $wenn$ $e \neq e'$

75.1 Beispiele

Wir haben ein Array von 0 bis 9 und h(i)=i mod 10. Das Array sieht nach dem Einfügen der Zahlen 42 und 119 wie folgt aus:

Index	Eintrag
0	
1	
2	42
3	
4	
5	
6	
7	
8	
9	119

Der Vorteil von Hashing ist, dass Anfragen der Form " Enthält die Datenstruktur das Element 42?" schnell beantwortbar sind. Dazu verhalten sich Hashtabellen ähnlich zu binären Suchbäumen wie BucketSort zu vergleichsbasierten Sortierverfahren.

75.2 Hashfunktionen

Die Hashfunktionen hängen vom Datentyp der Elemente und der konkreten Anwendungen ab. Für den Datentyp Integer ist die Hashfunktion meist h(i)=i mod N. Das funktioniert im Allgemeinen sehr gut, wenn N eine Primzahl ist und hängt mit Methoden zur Erzeugung von Zufallszahlen zusammen. Für andere Datentypen führt man eine Rückführung auf Integer aus. Bei Fließpunkt-Zahlen werden Mantisse und Exponent einfach addiert.

1 `https://de.wikipedia.org/wiki/Hashtabelle`

Die Hashwerte sollten gut streuen. Das ist eventuell von den Besonderheiten der Eingabewerte abhängig. Beispielsweise tauchen Buchstaben des Alphabets in Namen unterschiedlich oft auf. Des weiteren müssen die Hash-Werte effizient berechenbar sein. Ein konstanter Zeitbedarf ist erfordert, dieser ist nicht von der Anzahl der gespeicherten Werte abhängig.

75.3 Ungünstige Hashfunktionen

Als erstes Beispiel wählen wir $N = 2^i$ und eine generierte Artikelnummer mit den Kontrollziffern 1,3 oder 7 am Ende. Damit wäre die Abbildung nur auf ungeraden Adressen möglich. Als zweites Beispiel wählen wir Matrikelnummern in einer Hashtabelle mit 100 Einträgen. In der ersten Variante nutzen wir die ersten beiden Stellen als Hashwert, damit kann eine Abbildung nur auf wenige Buckets erfolgen. In der zweiten Variante nutzen wir die beiden letzten Stellen und erhalten eine gleichmäßige Verteilung.

76 Typen von Graphen und Anwendungen

In diesem Kapitel werden Graphen[1] behandelt. Ein Graph ist das mathematische Modell eines Netzwerks bestehend aus Knoten und Kanten. Graphen haben einen vielfältigen Einsatz. So kommen sie bei Verbindungsnetzwerken (Bahnnetz, Flugververbindungen, Straßenkarten, ...), Verweisen (WWW, Literaturverweise, Wikipedia, symbolische Links, ...), Technischen Modellen (Platinen-layout, finite Elemente, Computergrafik) und Software Reengineering und - dokumentation zum Einsatz. Bäume und Listen sind spezielle Graphen.

76.1 Ungerichteter Graph

Es gibt verschiedene Typen von Graphen. Der ungerichtete Graph ist beispielsweise eine Straßenverbindung, eine Telefonnetz oder ein soziales Netzwerk. Ein ungerichteter Graph ist ein Tupel G=(V,E). Wir haben eine endliche Menge V von Knoten (Vertices) und eine Menge E von Kanten (Edges), die aus ungeordneten Paaren aus V besteht. Es gilt, dass $E \subseteq VxV$ und jedes $e \in E$ ist eine zweielementige Teilmenge der Knotenmenge $V (e = a, b$ mit $a, b \in V)$. Im ungerichteten Graphen gibt es keine Schleifen, das heißt es gibt keine Kanten die von einem Knoten zu sich selbst laufen. Außerdem gibt es keine mehrfachen Kanten zwischen zwei Knoten, Parallelkanten genannt.

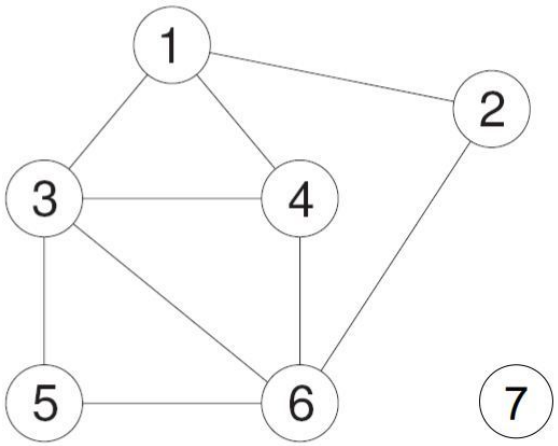

Abb. 34 Ungerichteter Graph

$V = \{1, 2, 3, 4, 5, 6, 7\}$

1 https://de.wikipedia.org/wiki/Graph_%28Graphentheorie%29

$E = \{\{1,2\},\{1,3\},\{1,4\},\{2,6\},\{3,4\},\{3,5\},\{3,6\},\{4,6\},\{5,6\}\}$

Hier können zum Beispiel die kürzesten Wege bei sozialen Netzwerken wie Facebook berechnet werden.

76.1.1 Spezielle Graphen

Sei $G = (V, E)$ ein Graph.

G heißt **planar** , falls er ohne Überschneidungen der Kanten in der Ebene gezeichnet werden kann.

G heißt **vollständig** , falls $E = V \times V$

G heißt **regulär** , falls alle Knoten denselben Grad haben

G heißt **bipartit** , falls $V = V_1 \cup V_2$ und

- • keine zwei Knoten in V_1 sind adjazent
  • keine zwei Knoten in V_2 sind adjazent

Beispiele

Dieser Graph ist sowohl planar, regulär als auch vollständig.

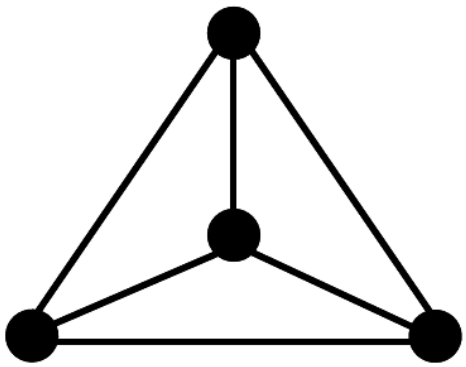

Abb. 35 ohne

Dieser Graph ist jedoch nur regulär und vollständig.

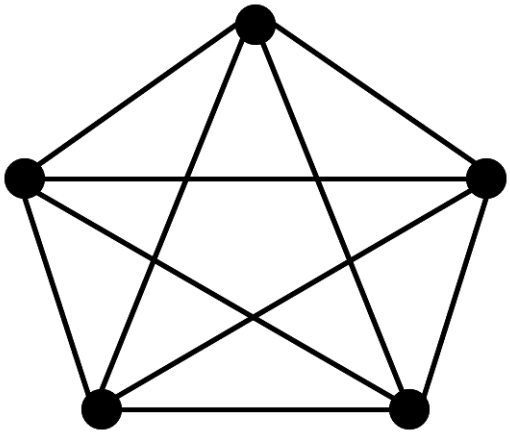

Abb. 36 ohne

Hier handelt es sich nur um einen regulären Graphen.

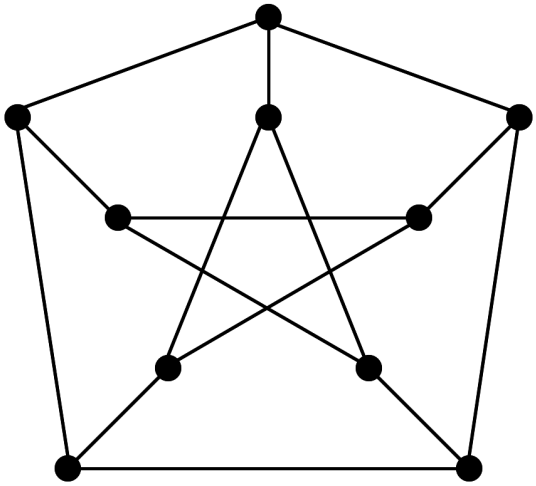

Abb. 37 ohne

Dies ist ein Beispiel für einen bipartiten Graph.

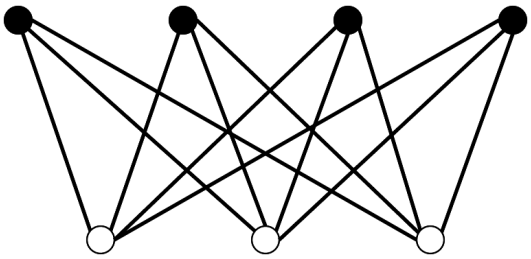

Abb. 38 ohne

76.2 Gerichteter Graph

Der gerichtete Graph ist beispielsweise eine Förderanlage oder ein Kontrollfluss in Programmen. Der gerichtete Graph (auch Digraph) ist ein Tupel G=(V,E) mit V als endliche Menge von Knoten und E einer Menge von Kanten, geordneten Paaren aus V. Jedes $e \in E$ ist nun ein Tupel e=(a,b) mit $a, b \in V$. Schleifen der Form (a,a) sind nun erlaubt. Dazu ist (a,b) eine andere Kante als (b,a). Der Unterschied zwischen (a,b) und {a,b} besteht darin, dass das Tupel (a,b) geordnet ist. Die Reihenfolge kann nicht verändert werden. Hingegen ist {a,b} eine Menge, in der die Reihenfolge der Elemente keine Rolle spielt.

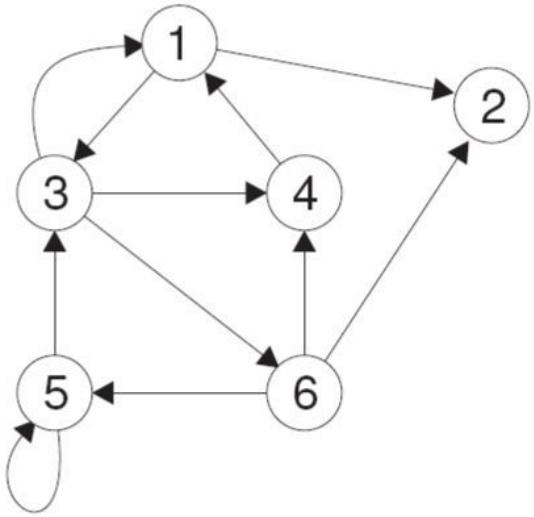

Abb. 39 Gerichteter graph

$G_g = (V_g, E_g)$

$V_g = \{1,2,3,4,5,6\}$

$E_g = \{(1,2),(1,3),(3,1),(3,4),(3,6),(4,1),(5,3),(5,5),(6,2),(6,4),(6,5)\}$

Gerichtete Graphen werden zum Beispiel als Web-Graph (Google‘s PageRank) benutzt. Aber auch in der Scientometrie kommen sie zum Einsatz bei der Impact Faktoren Berechnung. Bei Datenstrukturen im Semantik Web werden gerichtete Graphen zum Speichern von Daten genutzt.

76.3 Gerichtete und ungerichtete Graphen

Ein ungerichteter Graph kann in einen gerichteten Graphen transformiert werden, indem jede ungerichtete Kante {v,w} durch zwei gerichtete Kanten (v,w) und (w,v) ersetzt wird. Dann ist beispielsweise der Zusammenhang identisch mit dem starken Zusaamenhang. Dazu haben gerichtete Graphen haben eine größere Ausdrucksstärke und daher wird ”Graph” oft als Synonym für einen Digraph verwendet.

76.4 Gewichteter Graph

Ein ungerichteter gewichteter Graph ist beispielsweise eine Flugverbindung mit Meilen oder Kosten, ein Straßennetz mit Kilometern oder ein Rohrsystem mit Durchfluss.

Ein gerichteter gewichteter Graph ist beispielsweise eine Straßennetz mit Einbahnstraßen, Rohre mit Ventilen oder ein Förderband.

```
Der Graph ist ein Paar G=(V,E) und wir haben eine Kantengewichtsfunktion g.
Daraus erhalten wir G=(V,E,g) mit g : E → ℕ. Der Graph kann gerichtet oder ungerichtet
sein und die Kantengewichte müssen nicht notwendigerweise natürliche Zahlen
sein.
```

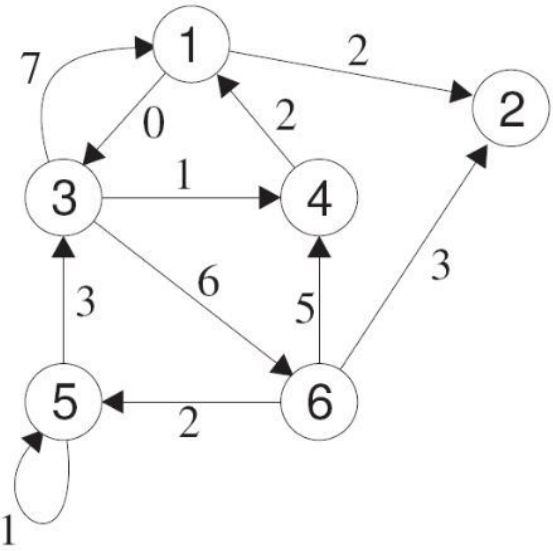

Abb. 40 Gewichteter Graph

Ungerichtete gewichtete Graphen kommen zum Beispiel bei der Navigation beim Berechnen des kürzesten Weges zum Einsatz.

Gerichtete gewichtete Graphen kommen bei der Optimierung in der Telekommunikation zum Einsatz.

76.5 Hypergraph

Es gibt aber noch viele weitere Varianten von Graphen wie Multigraphen oder Hypergraphen.

Ein Hypergraph ist ein Paar G=(V,E) mit einer Menge von Knoten V und einer Menge von Hyperkanten $E \subseteq 2^V$.

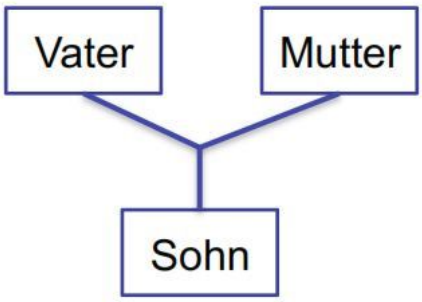

Abb. 41 Hypergraph

77 Definitionen

Hier werden allgemeine Definitionen bezüglich der Graphen behandelt. Dazu werden immer wieder Beispiele gebracht, die sich auf folgende Graphen beziehen. Dabei gilt je nach Beispiel G=(V,E) entweder für den ungerichteten oder den gerichteten Graphen.

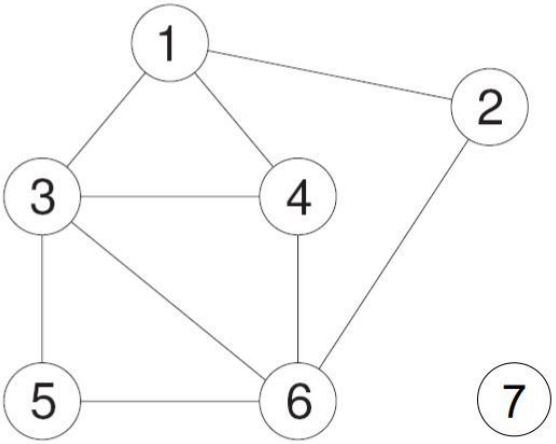

Abb. 42 Ungerichteter Graph

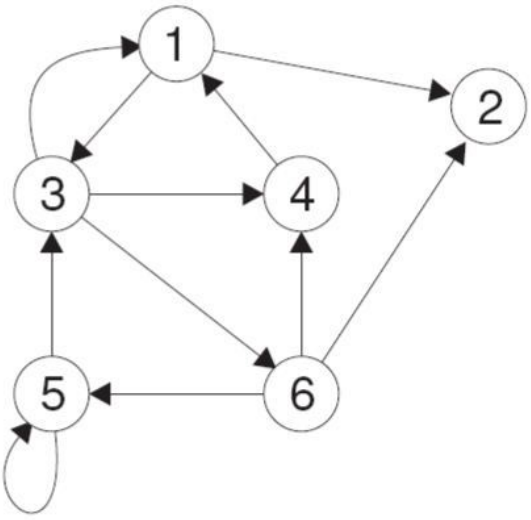

Abb. 43 Gerichteter graph

77.1 Adjazenz

77.1.1 Ungerichteter Graph

Zwei Knoten $v, w \in V$ heißen *adjazent* , falls $\{v, w\} \in E$.

Hier heißt v heißt auch *Nachbar* von w.

Beispiel:

- Knoten 1 und 3 sind adjazent

77.1.2 Gerichteter Graph

Zwei Knoten $v, w \in V$ heißen *adjazent* , falls $(v, w) \in E$ oder $(w, v) \in E$.

Für $(v, w) \in E$ heißt w *Nachfolger* von v und v **Vorgänger** von w.

Beispiele:

- Knoten 1 ist Vorgänger zu Knoten 3
- Knoten 4 ist Nachfolger zu Knoten 6

77.2 Inzidenz

77.2.1 Ungerichteter Graph

Eine Kante $\{v, w\} \in E$ ist *inzident* zu einem Knoten $z \in V$, falls $v = z$ oder $w = z$.

77.2.2 Gerichteter Graph

Eine Kante $(v, w) \in E$ ist *inzident* zu einem Knoten $z \in V$, falls $v = z$ oder $w = z$.

77.3 Grad

77.3.1 Ungerichteter Graph

Der *Grad* (engl. degree) eines Knotens $v \in V$ ist die Anzahl seiner inzidenten Kanten, das heißt: $degree(v) = |\{\{w, x\} \in E | w = v \text{ oder } x = v\}|$.

Beispiel:

- Der Grad von Knoten 4 ist 3

77.3.2 Gerichteter Graph

Der *Eingangsgrad* (engl. in-degree) eines Knotens $v \in V$ ist die Anzahl seiner Vorgänger: $indeg(v) = |\{(w,v) \in E\}|$.

Der *Ausgangsgrad* (engl. out-degree) eines Knotens $v \in V$ ist die Anzahl seiner Nachfolger: $outdeg(v) = |\{(v,w) \in E\}|$.

Beispiele:

- Der Eingangsgrad von Knoten 3 ist 2
- Der Ausgangsgrad von Knoten 3 ist 3

77.4 Weg

77.4.1 Ungerichteter Graph

Ein *Weg* W ist eine Sequenz von Knoten $W = (v_1, ..., v_n)$ mit $v_1, ..., v_n \in V$ für die gilt: $\{v_i, v_{i+1}\} \in E$ für alle $i = 1, ..., n-1$

Beispiel:

- (1,3,5,6,3,4) ist ein Weg

77.4.2 Gerichteter Graph

Ein (gerichteter) *Weg* W ist eine Sequenz von Knoten $W = (v_1, ..., v_n)$ mit $v_1, ..., v_n \in V$, für die gilt: $(v_i, v_{i+1}) \in E$ für alle $i = 1, ..., n-1$.

Beispiel:

- (1,3,6,5,5,3,1) ist ein (gerichteter) Weg

77.5 Pfad

Ein Weg W heißt *Pfad* , falls zusätzlich gilt $v_i \neq v_j$ für alle $i, j = 1, ..., n$ mit $i \neq j$. Das heißt, der Weg enthält keine doppelten Knoten. Diese Definition gilt sowohl für ungerichtete als auch gerichtete Graphen.

Beispiel:

- (1,4,6,5) ist ein Pfad

77.6 Kreis

Ein Weg P heißt *Kreis* , falls $v_1 = v_n$. Dazu ist ein Kreis K *elementar* , falls $v_i \neq v_j$ für alle $i, j = 1, ..., n-1$ mit $i \neq j$. Der Kreis enthält also keine doppelten Knoten bis

auf den Anfangs- und den Endpunkt. Diese Definition gilt sowohl für ungerichtete als auch gerichtete Graphen.

Beispiel:

- (1,3,4,6,3,4,1) ist ein Kreis
- (3,4,6,3) ist ein elementarer Kreis

77.7 Länge

Die *Länge* eines Weges ist die Anzahl der durchlaufenen Kanten. Die Länge eines Pfades ist also n-1. Diese Definition gilt sowohl für ungerichtete als auch gerichtete Graphen.

Beispiel:

- Die Länge von (3,4,6,3,4,1) ist 4
- Die Länge von (1,3,6) ist 2

77.8 Teilgraph

77.8.1 Ungerichteter Graph

Ein Graph $G' = (V', E')$ heißt *Teilgraph* von G, falls $V' \subseteq V$ und $E' \subseteq E \cap (V' \times V')$.

Beispiel:

- G'=({3,4,6},
  UNKNOWN TEMPLATE 3,4,4,6
) ist ein Teilgraph von G

77.8.2 Gerichteter Graph

Ein Graph $G' = (V', E')$ heißt *Teilgraph* von G, falls $V' \subseteq V$ und $E' \subseteq E \cap (V' \times V')$.

Beispiel:

- $G' = (\{1,3,4\}, \{(1,3),(4,1)\})$ ist ein Teilgraph von G_g.

77.9 Erreichbarkeit

77.9.1 Ungerichteter Graph

Ein Knoten $w \in V$ heißt *erreichbar* von einem Knoten $v \in V$, falls ein Weg $W = (v_1, ..., v_n)$ existiert mit $v_1 = v$ und $v_n = w$.

Beispiele:

- Knoten 6 ist erreichbar von Knoten 1

- Knoten 7 ist nicht erreichbar von Knoten 1

77.9.2 Gerichteter Graph

Ein Knoten $w \in V$ heißt *erreichbar* von einem Knoten $v \in V$, falls ein Weg $W = (v_1, ..., v_n)$ existiert mit $v_1 = v$ und $v_n = w$.

Beispiele:

- Knoten 6 ist erreichbar von Knoten 1
- Knoten 5 ist nicht erreichbar von Knoten 2

77.10 Zusammenhang

77.10.1 Ungerichteter Graph

G heißt (einfach) *zusammenhängend* , falls für alle $v, w \in V$ gilt, dass w von v erreichbar ist

Ein Teilgraph $G' = (V', E')$ von G heißt *Zusammenhangskomponente* von G, falls G' zusammenhängend ist und kein Teilgraph $G'' = (V'', E'')$ von G existert mit $V' \subset V''$.

Beispiele:

- G ist nicht zusammenhängend
- Der Teilgraph $G'' = (V'', E'')$ mit $V'' = \{1,2,3,4,5,6\}$ und $E'' = \{\{1,2\},\{1,3\},\{1,4\},\{2,6\},\{3,4\},\{3,5\},\{3,6\},\{4,6\},\{5,6\}\}$ ist eine Zusammenhangskomponente von G
- Der Teilgraph $G''' = (\{7\}, \emptyset)$ ist eine Zusammenhangskomponente von G

77.10.2 Gerichteter Graph

G heißt (stark) *zusammenhängend* , falls für alle $v, w, \in V$ gilt, dass w von v und v von w erreichbar ist.

Ein Teilgraph $G' = (V', E')$ von G heißt *starke Zusammenhangskomponente* von G, falls G' stark zusammenhängend ist und kein Teilgraph $G'' = (V'', E'')$ von G existiert mit $V' \subset V''$.

Beispiel:

- Der Teilgraph $G'' = (V'', E'')$ mit $V'' = \{1,3,4,5,6\}$ und $E'' = \{(1,3),(3,1),(3,4),(3,6),(4,1),(5,3),(5,5),(6,4),(6,5)\}$ ist eine starke Zusammenhangskomponente von G_g.

78 Repräsentation von Graphen

Auf dieser Seite wird die Repräsentation von Graphen behandelt. Wir fragen uns wie effizient die Datenstruktur für Graphen ist.

78.1 Kanten- und Knotenlisten

Bei durchnummerierten Knoten erfolgt eine einfache Realisierung. Historisch gesehen ist es die erste verwendete Datenstruktur. Außerdem ist sie als Austauschformat geeignet und die Auflistung ist nach Knoten oder nach Kanten sortiert.

78.1.1 Beispiel Kantenliste

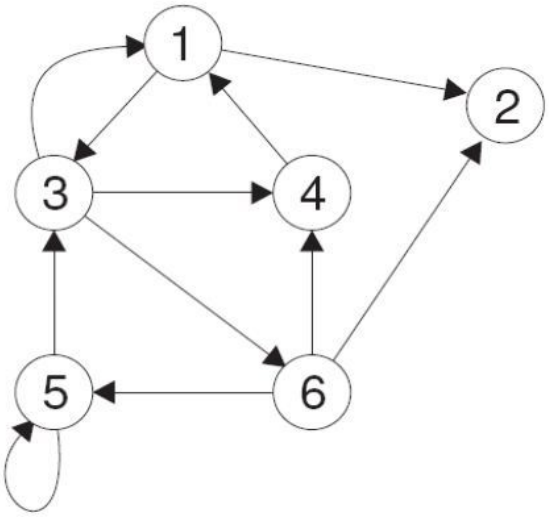

Abb. 44 Graph

Gegeben ist eine Kantenliste für G_g :

Die erste Zahl (6) steht für die Knotenzahl. Die zweite Zahl (11) steht für die Kantenzahl. Die weiteren Paare (1,2 ; 1,3...) stehen für die Kanten.

6,11,1,2,1,3,3,1,4,1,3,4,3,6,5,3,5,5,6,5,6,2,6,4

78.1.2 Beispiel Knotenliste

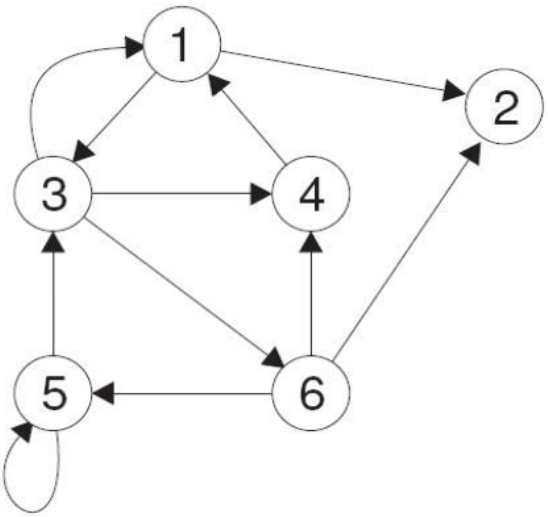

Abb. 45 Graph

Gegeben ist eine Knotenliste für G_g :

6,11,2,2,3,0,3,1,4,6,1,1,2,3,5,3,2,4,5

Die Teilfolge 2,2,3 bedeutet, dass der Knoten 1 den Ausgangsgrad 2 hat und herausgehende Kanten zu den Knoten 2 und 3.

Vergleich Kanten-und Knotenliste

Falls ein Graph mehr Kanten als Knoten hat (=„Normalfall“),benötigen Knotenlisten weniger Speicherbedarf als Kantenlisten. Das bedeutet für die Kantenlisten gilt $2+2|E|$ und für die Knotenliste gilt $2+|V|+|E|$.

78.1.3 Adjazenzmatrix

Adjazenz bedeutet berühren oder aneinander grenzen. Hier werden die Graphen als Boole‘sche Matrix dargestellt. 1-Einträge werden für direkte Nachbarschaften verwendet. A ist eine Adjazenzmatrix für den Graph $G = (V, E) : (A_{ij}) = 1$ *genau dann wenn* $(i, j) \in E$

Beispiel

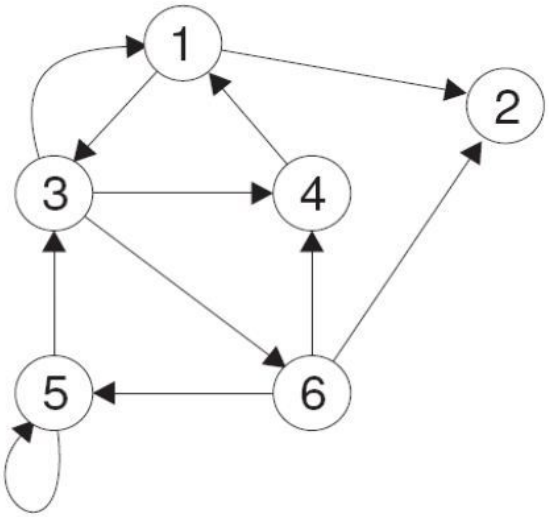

Abb. 46 Graph

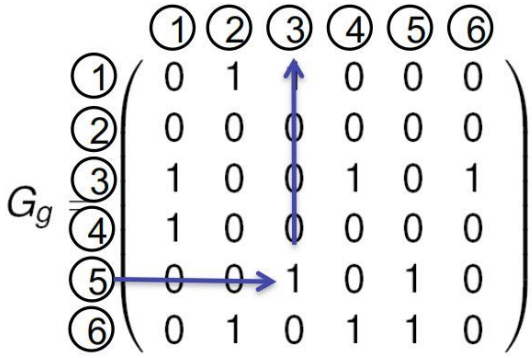

Abb. 47 Adjazensmatrix

Eigenschaften

Bei ungerichteten Graphen reicht eine Halbmatrix (ein Dreieck) aus. Bei gewichteten Graphen werden Gewichte statt Boolsche Werte genutzt. Der Vorteil einer Adjazenzmatrix ist, dass einige Graphenoperationen als Matrixoperation möglich sind. So ist sie beispielsweise durch iterierte Matrixmultiplikation erreichbar und besitzt schöne Eigenschaften für die mathematische Analyse.

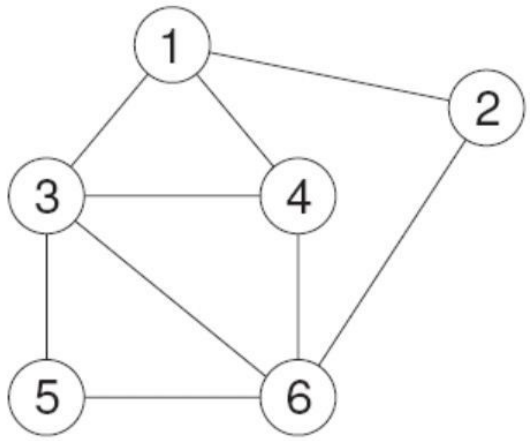

Abb. 48 Ungerichtet2

$$G_u = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Abb. 49 Ungerichtet

$$G_u = \begin{pmatrix} 0 & & & & & \\ 1 & 0 & & & & \\ 1 & 0 & 0 & & & \\ 1 & 0 & 1 & 0 & & \\ 0 & 0 & 1 & 0 & 0 & \\ 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Abb. 50 Ungerichtet3

So sieht die Darstellung als Dreiecksmatrix aus. Die Diagonale kann ebenfalls weggelassen werden, wenn Schleifen verboten sind.

78.1.4 Adjazenzliste

Wir haben eine Liste der Knoten oder alternativ ein Array. Pro Knoten werden die von ihm ausgehenden Kanten als Liste, welche besonders geeignet für dünn besetzte Matrizen sind, oder als Array von Zeigern dargestellt. Der Graph wird durch |V|+1 verkettete Listen realisiert. In Adjazenzlisten sind dynamische Erweiterungen im Sinne verketteter Listen erlaubt. Knotenlisten können natürlich auch als verkettete Listen realisiert werden.

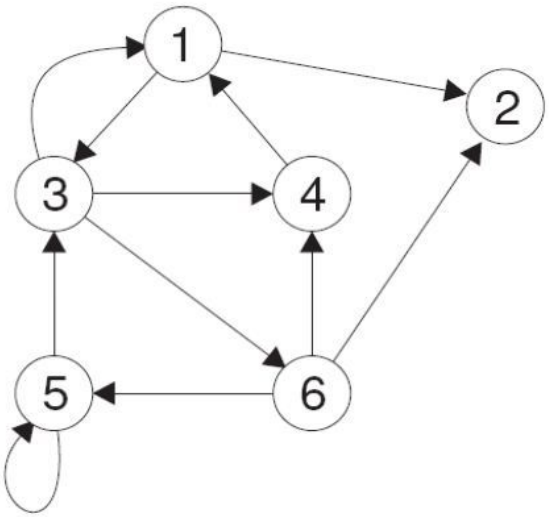

Abb. 51 Graph

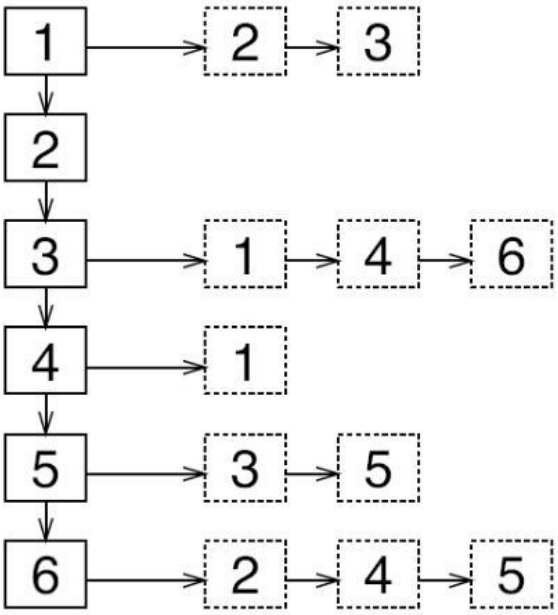

Abb. 52 Adjazenzliste

Speicherbedarf

Seien n=|V| und m=|E|. Benötigt werden insgesamt $n + \sum_{i=1}^{n} ag(i) = n + m$ Listenelemente. ag(i) ist die Anzahl der Nachbarn von i (gerichtet).

78.2 Transformation zwischen den Darstellungen

Die vorgestellten Realisierungsvarianten sind äquivalent. Jede Darstellung kann in jede andere ohne Informationsverlust transformiert werden. Dafür wird die eigene Darstellung ausgelesen und anschließend die andere Darstellung erzeugt. Der Aufwand dieser Transformationen variiert von O(n+m) bis $O(n^2)$ wobei im schlechtesten Fall $m = n^2$ gilt. n^2 tritt immer auf, wenn eine naive Matrixdarstellung beteiligt ist. Nicht naive Darstellungen sind für sehr dünn besetzte Matrizen nötig.

78.3 Komplexitätsbetrachtung

Bei Kantenlisten ist das Einfügen von Kanten (Anhängen von zwei Zahlen) und von Knoten (Erhöhung der ersten Zahl um 1) besonders günstig. Das Löschen von Kanten zieht das Verschieben der nachfolgenden Kanten mit sich und die Knoten müssen neu nummeriert werden.

Bei Knotenlisten ist das Einfügen von Knoten, also die Erhöhung der ersten Zahl und das Anhängen einer 0, günstig.

Bei der Matrixdarstellung ist das Manipulieren von Kanten sehr effizient ausführbar. Der Aufwand beim Knoteneinfügen hängt von der Realisierung ab. Im worst case wird die Matrix in eine größere Matrix kopiert.

Bei Adjazenzlisten gibt es unterschiedlichen Aufwand, je nachdem, ob die Knotenliste ein Feld mit Direktzugriff oder eine verkettete Liste mit sequenziellem Durchlauf realisiert.

Operation	Kantenliste	Knotenliste	Adjazenzmatrix	Adjazenzliste
Einfügen Kanten	O(1)	O(n+m)	O(1)	O(1)/O(n)
Löschen Kanten	O(m)	O(n+m)	O(1)	O(n)
Einfügen Knoten	O(1)	O(1)	$O(n^2)$	O(1)
Löschen Knoten	O(m)	O(n+m)	$O(n^2)$	O(n+m)

Das Löschen eines Knotens impliziert für gewöhnlich auch das Löschen der dazugehörigen Kanten.

79 Datenstrukturen für Graphen

Auf dieser Seite werden die Datenstrukturen für Graphen behandelt. In Java gibt es keine hauseigene Graphimplementierung, aber es gibt diverse Pakete für verschiedene Anwendungen.

- Jung (`http://jung.sourceforge.net`)

```
Graph<Integer, String> g = new SparseMultigraph<Integer, String>();
g.addVertex((Integer)1);
g.addVertex((Integer)2);
g.addEdge("Edge1", 1, 2);
```

- Neo4j (`http://www.neo4j.org`)

```
GraphDatabaseService= new
"GraphDatabaseFactory().newEmbeddedDatabase(“PATH”);
Transaction tx = graphDb.beginTx();
try{
   Node firstNode = graphDb.createNode();
   Node secondNode = graphDb.createNode();
   Relationship relationship = firstNode.createRelationshipTo(secondNode,
    … );
   tx.success();
}finally{
   tx.finish();
}
```

Die allgemeine Schnittstelle für die Vorlesung ist:

```
 public interface Graph {
   public int addNode();
   public boolean addEdge (int orig, int dest);
}
```

79.1 Implementierung Adjazenzliste

```
 public class AdjazenzListGraph implements Graph {
   private int [][] adjacencyList=null;

   //Knoten hinzufügen:
   public int addNode() {
      int nodeNumber = (adjacencyList ==null)?0: adjacencyList.length;
      int [][] newAdjacencyList= new int [nodeNumber+1][];
      //alte adjacencyList kopieren
      for (int i=0; i< nodeNumber; i++)
         newAdjacencyList [i]=adjacencyList[i];
      //neuer Knoten hat noch keine Kanten
      newAdjacencyList[nodeNumber] =null;
      adjacencyList=newAdjacencyList;
      return nodeNumber+1;
```

```
   }

   //Kante hinzufügen:
   public boolean addEdge (int orig, int dest){
      int nodeNumber = (adjacencyList == null)? 0: adjacencyList.length;
      if (orig > nodeNumber || dest > nodeNumber || orig < 1 || dest < 1 )
         return false;
      if (adjacencyList[orig-1] != null)
         for (int n : adjacencyList[orig-1])
            //Kante bereits vorhanden?
            if (n==dest) return false;
      //Erste Kante am Knoten orig?
      if ( adjacencyList[orig-1] == null ) {
         adjacencyList [orig-1] = new int[1];
         adjacencyList[orig-1][0]=dest;
      }
      else {
         int[] newList= new int[adjacencyList[orig-1].length+1];
         S
ystem.arraycopy(adjacencyList[orig-1],0,newList,0,adjacencyList[orig-1].length);
         newList[adjacencyList[orig-1].length]=dest;
         adjacencyList [orig-1]=newList;
      }
      return true;
   }
}
```

80 Breitensuche

Auf dieser Seite behandeln wir die Breitensuche[1]. Wir fragen uns wie man die Knoten eines Graphen effizient aufzählt. Die Lösung ist der Breitendurchlauf (Breadth-First-Search, BFS). Dabei werden die Knoten eines Graphen nach der Entfernung vom Zielknoten aufgezählt. Eine andere Methode ist der Tiefendurchlauf, zu dem kommen wir aber später. Bei dem Breitendurchlauf für ungerichtete Graphen gibt es eine Warteschlange als Zwischenspeicher. Farbmarkierungen beschreiben den Status der Knoten. Weiß bedeutet er ist unbearbeitet, grau bedeutet er ist in Bearbeitung und schwarz bedeutet, dass er abgearbeitet ist. Pro Knoten wird die Entfernung zum Startknoten berechnet. Bei der Initialisierung wird der Startknoten in eine Warteschlange eingefügt, die Farbe auf grau gesetzt und die Entfernung mit 0 berechnet. Die anderen Knoten haben eine unendliche Entfernung und sind weiß markiert.

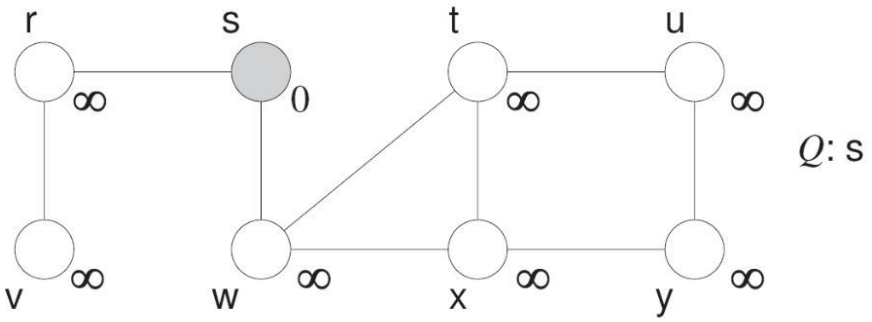

Abb. 53 Breitendurchlauf

Beim Breitendurchlauf wird der aktuelle Knoten k aus der Warteschlange genommen und schwarz gefärbt. Alle von k aus erreichbaren weißen Knoten werden grau gefärbt, die Entfernung ist der Entfernungswert von k+1 und sie werden in der Warteschlange aufgenommen.

1 `https://de.wikipedia.org/wiki/Breitendurchlauf`

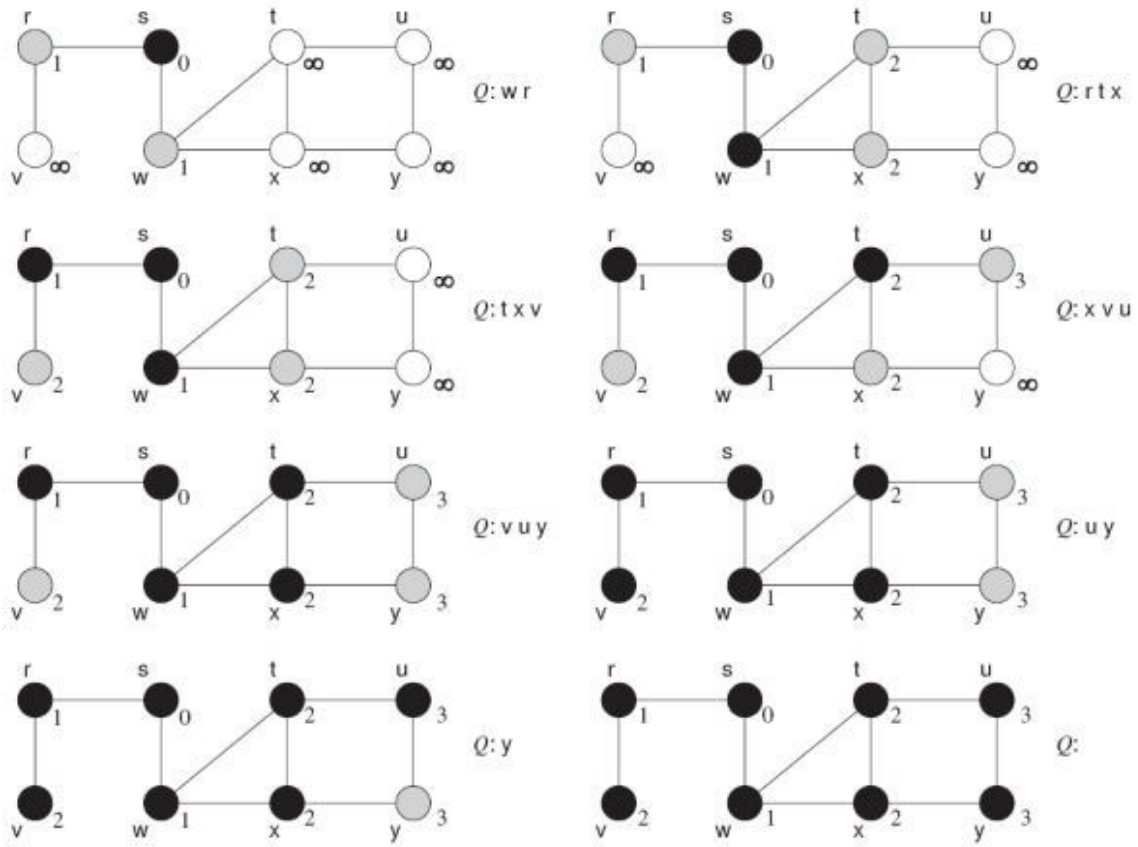

Abb. 54 Breitendurchlauf

80.1 Algorithmus

Ergänzung zum Graph-Interface:

```
public interface Graph{
   public int addNode();
   public boolean addEdge(int orig, int dest);
   public Collection<Integer> getChildren(int node);!
}
```

Breitendurchlauf als Iterator:

```
public class BfsIterator implements Iterator<Integer>{
   private Graph g;
   private Queue<Integer> q;
   private Set<Integer> visited;

   public BfsIterator(Graph g, int s){
      this.g = g;
      this.q = new LinkedList<Integer>();
      q.add(s);
      this.visited = new HashSet<Integer>();
   }

   public boolean hasNext() { return !this.q.isEmpty(); }
```

```
   public Integer next() {
      Integer n = this.q.poll();
      for(Integer m: this.g.getChildren(n))
           if(!this.visited.contains(m) && !this.q.contains(m))
             this.q.add(m);
      this.visited.add(n);
      return n;
   }
}
```

Ausgabe aller Knoten:

```
//Sei g ein Graph
Iterator<Integer> it = new BfsIterator(g,1);
while(it.hasNext())
   System.out.println(it.next());
```

80.2 Analyse

80.2.1 Theorem der Terminierung

Die Breitensuche terminiert nach endlicher Zeit

80.2.2 Theorem der Korrektheit

Ist G zusammenhängend, so werden alle Knoten von G genau einmal besucht.

80.2.3 Theorem der Laufzeit

Ist G=(V,E) zusammenhängend und ist die Laufzeit von getChildren linear in der Anzahl der Kinder, so hat die Breitensuche eine Laufzeit von O(|V| + |E|).

81 Tiefendurchlauf

Auf dieser Seite wird der Tiefendurchlauf behandelt. Der Tiefendurchlauf wird auch Depth-First-Search, oder abgekürzt DFS, genannt. Die Knoten werden aufgezählt indem vom Startknoten aus ein Pfad so weit wie möglich verfolgt wird und bei Bedarf ein Backtracking durchgeführt wird. Bei Tiefendurchlauf werden die Knoten ebenfalls farblich markiert. Weiß bedeutet der Knoten ist noch nicht bearbeitet, grau bedeutet der Knoten ist in Bearbeitung und schwarz bedeutet der Knoten ist bereits fertig abgearbeitet.

Ergänzung zum Graph Interface:

```
public interface Graph{
   public int addNode();
   public boolean addEdge(int orig, int dest);
   public Collection<Integer> getChildren(int node);
   public Collection<Integer> getNodes();
}
```

81.1 Algorithmus

```
enum Color {WHITE, GRAY, BLACK};

Map<Integer,Color> color = new HashMap<Integer,Color>();
Map<Integer,Integer> pi = new HashMap<Integer,Integer>();
Map<Integer,Integer> f = new HashMap<Integer,Integer>();
Map<Integer,Integer> d = new HashMap<Integer,Integer>();

int time = 0;
```

color speichert die Farbe, bzw. den Bearbeitungszustand eines Knotens.

pi speichert den Vorgänger eines Knotens beim Durchlauf.

f speichert den Zeitpunkt des Bearbeitungsbeginns eines Knotens.

d speichert den Zeitpunkt des Bearbeitungsendes eines Knotens.

```
public void dfs(Graph g){
   for(Integer n: g.getNodes())
      color.put(n, Color.WHITE);
   for(Integer n: g.getNodes())
      if(color.get(n).equals(Color.WHITE))
          dfsVisit(g,n);
}

public void dfsVisit(Graph g, Integer n){
   color.put(n, Color.GRAY);
   time++;
   d.put(n, time);
   for(Integer m: g.getChildren(n)){
```

```
      if(color.get(m).equals(Color.WHITE)){
         pi.put(m, n);
         dfsVisit(g,m);
      }
   }
   color.put(n, Color.BLACK);
   time++;
   f.put(n, time);
}
```

81.2 Vorgehen

Der Tiefendurchlauf ist ein rekursiver Abstieg. Pro Knoten haben wir zwei Werte und deren Farbwerte. Beginn der Bearbeitung ist d und Ende der Bearbeitung ist f. Der rekursive Aufruf erfolgt nur bei weißen Knoten, die Terminierung der Rekursion ist hier garantiert. Die Ausführung von DFS resultiert in einer Folge von DFS-Bäumen. Der erste Baum wird aufgebaut bis keine Knoten mehr hinzugefügt werden können. Anschließend wird ein unbesuchter Knoten gewählt und fortgefahren. Bei den Kanten des aufgespannten Baumes ist der Zielknoten beim Test weiß. An den B-Kanten ist der Zielknoten beim Test grau. Hierbei handelt es sich um Back Edges oder Rückkanten im aufgespannten Baum. Eine mit B markierte Kante zeigt einen Zyklus an. Bei F Kanten werden beim Test schwarze Knoten gefunden, dessen Bearbeitungsintervall ins Intervall des aktuellen bearbeiteten Knotens passt. Es handelt sich hierbei um Forward Edges bzw. Vorwärtskanen in dem aufgespannten Baum. Bei C Kanten haben wir schwarze Zielknoten v, dessen Intervalle nicht in das aktuelle Intervall passen (d[u]>f[v]). Hierbei handelt es sich um Cross Edges, eine Kante die zwei aufgespannte Bäume verbindet.

81.3 Beispiel

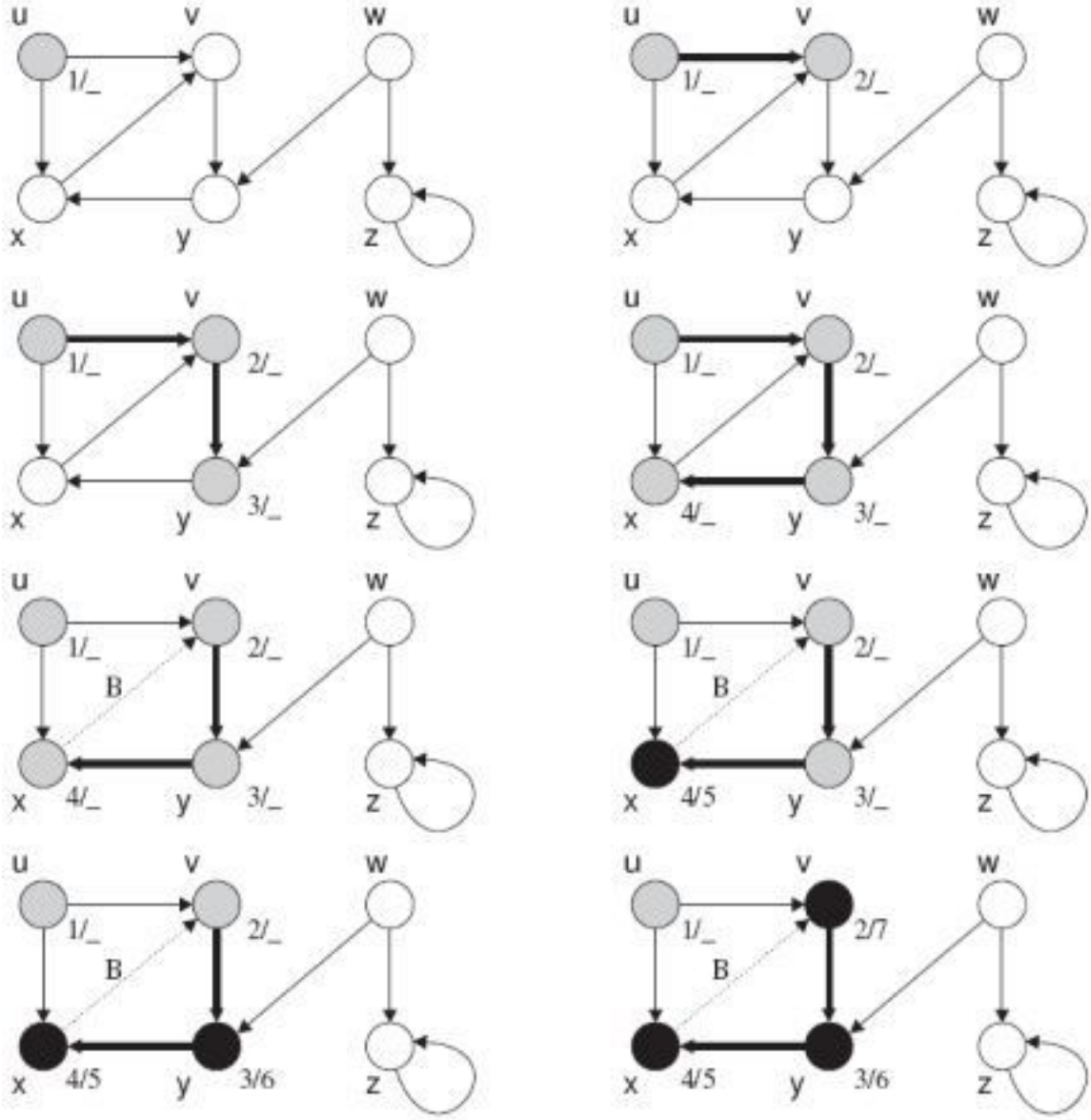

Abb. 55 Tiefendurchlauf

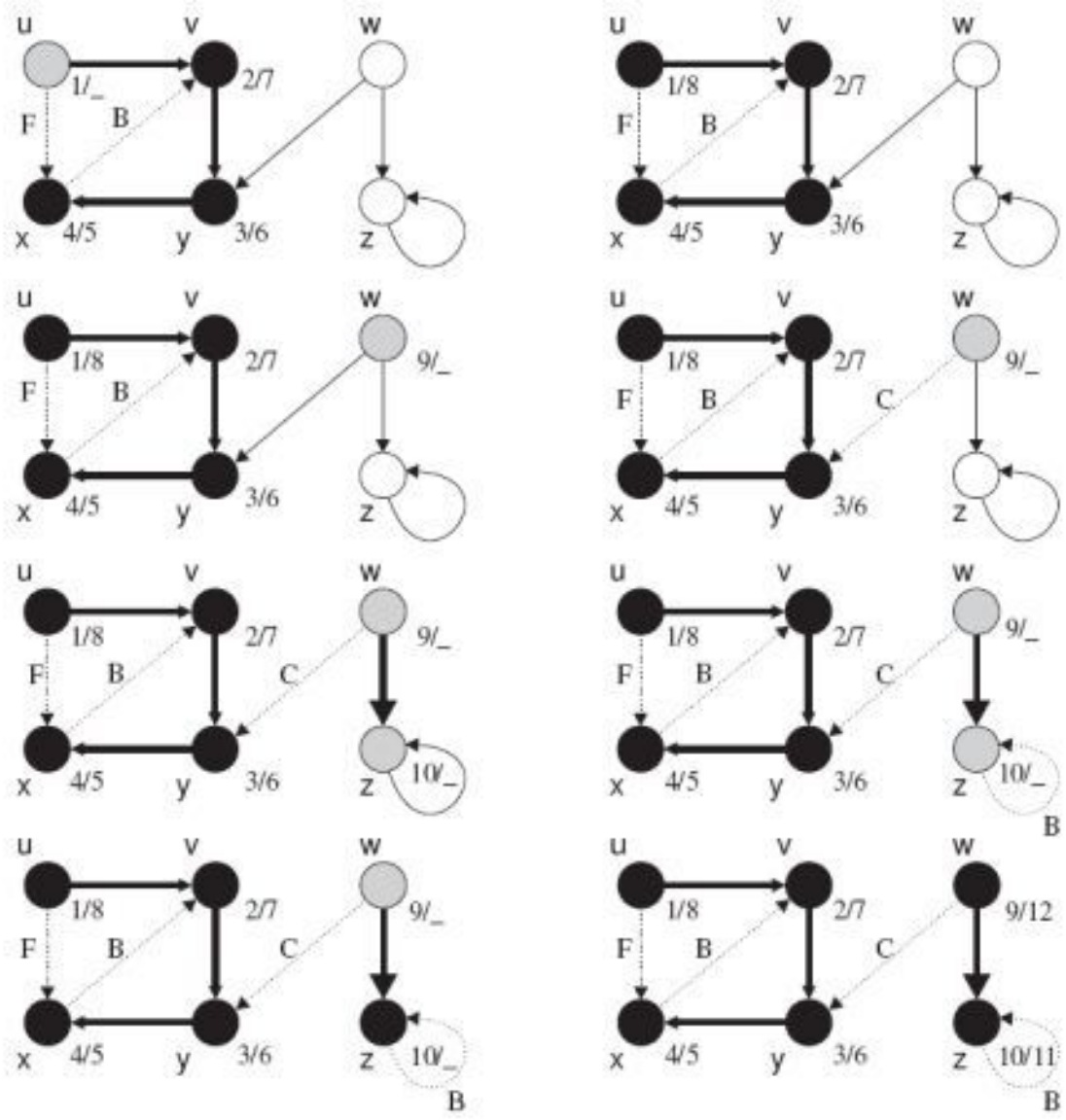

Abb. 56 Tiefendurchlauf2

Die Notation an den Knoten ist dabei durch <Beginn der Bearbeitung d> / <Ende der Bearbeitung f> gegeben.

81.4 Analyse

81.4.1 Theorem der Terminierung

Die Tiefensuche terminiert nach endlicher Zeit.

81.4.2 Theorem der Korrektheit

Es werden alle Knoten von G genau einmal besucht.

81.4.3 Theorem der Laufzeit

Ist sowohl die Laufzeit von getChidlren linear in der Anzahl der Kinder als auch getNodes linear in der Anzahl der Knoten, so hat die Tiefensuche eine Laufzeit von O(|V|+|E|).

81.5 Anwendung

Der Tiefendurchlauf wird beispielsweise bei dem Test auf Zyklenfreiheit verwendet. Damit ein Graph zyklenfrei ist, darf kein Kreis K in dem Graph G vorhanden sein. Deshalb basiert dieser Test auf dem Erkennen von Back Edges. Er ist effizienter als beispielsweise die Konstruktion einer transitiven Hülle. Die Tiefensuche wird aber auch beim topologischen Sortieren verwendet. Topologisch bedeutet sortieren nach Nachbarschaft, nicht nach totaler Ordnung.

82 Topologisches Sortieren

Auf dieser Seite wird das topologische Sortieren[1] behandelt. Wir fragen uns, wie Knoten unter Berücksichtigung von Abhängigkeiten aufgezählt werden können bei gegebenem azyklischem gerichteten Graph. Zur Anwendung kommt diese Sortierung bei Scheduling bei kausalen und zeitlichen Abhängigkeiten, zum Beispiel bei der Netzplantechnik. Mathematisch liegt hier eine Konstruktion einer totalen Ordnung aus einer Halbordnung vor.

82.1 Beispiel

Die sorgfältige Mutter legt ihrem Kind morgens die Kleidungsstücke so auf einen Stapel, dass das Kind nur die Kleidungsstücke vom Stapel nehmen und anziehen muss und dann richtig gekleidet ist. Hierfür legt sie die Reihenfolgebedingungen fest:

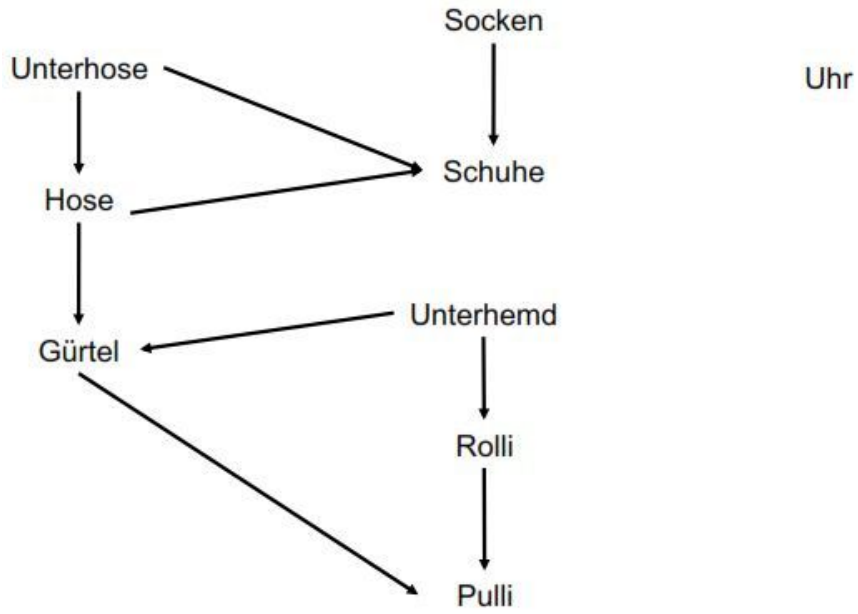

Abb. 57 TopologischesSortieren

Unterhose vor Hose

Hose vor Gürtel

1 https://de.wikipedia.org/wiki/Topologisches_Sortieren

Unterhemd vor Gürtel

Gürtel vor Pulli

Unterhemd vor Rolli

Rolli vor Pulli

Socken vor Schuhen

Hose vor Schuhen

Uhr: egal

DFS erstellt die topologische Ordnung on the fly. Das Sortieren nach f-Wert (invers) ergibt eine korrekte Reihenfolge. Statt der expliziten Sortierung nach f werden beim Setzen des f-Wertes die Knoten vorne in eine verkettete Liste eingehängt.

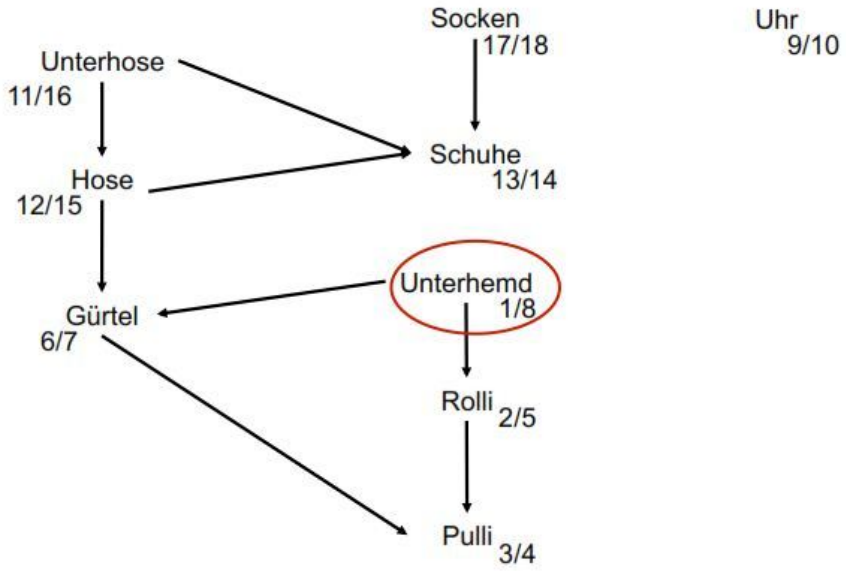

Abb. 58 TopologischeSortieren

18 Socken

16 Unterhose

15 Hose

14 Schuhe

10 Uhr

8 Unterhemd

7 Gürtel

5 Rolli

4 Pulli

Alternativer Durchlauf:

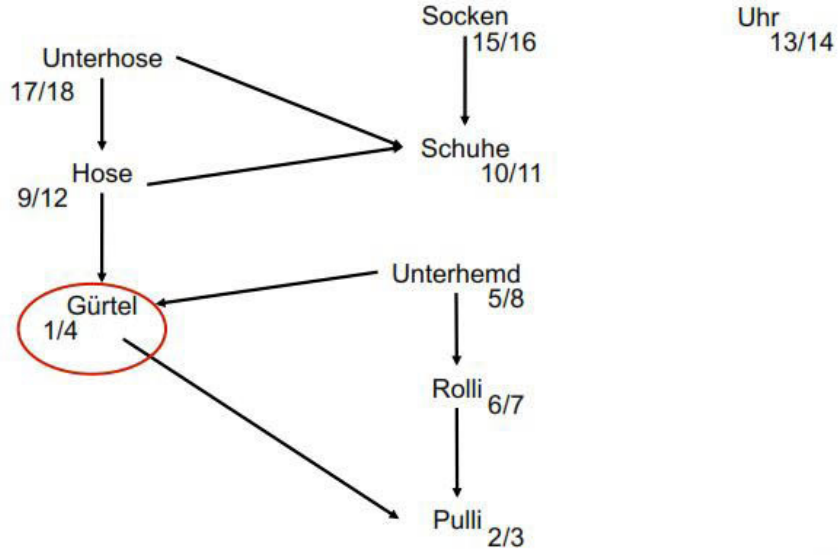

Abb. 59 TopologischeSortieren2

82.2 Berechnung kürzester Wege

Auf dieser Seite wird die Berechnung der kürzesten Wege[2] behandelt.

Gegeben ist ein (Di-)Graph $G = (V, E, \gamma)$ mit einer Gewichtsfunktion: $\gamma : E \to \mathbb{N}$. Der Pfad durch G ist eine Liste von aneinanderstoßenden Kanten $P = \{(v_1, v_2), (v_2, v_3), ...(v_{n-1}, v_n)\} \subseteq E$. Das Gewicht oder die Länge eines Pfades ist die Aufsummierung der einzelnen Kantengewichte. $w(P) = \sum_{i=1}^{n-1} \gamma((v_i, v_{i+1}))$. Die Distanz zweier Punkte d(u,v) ist das Gewicht des kürzesten Pfades von u nach v.

Es existieren verschiedene kürzeste Wege Probleme.

SPSP: Single pari shortest path

Eingabe: Graph G, Startknoten s, Endknoten t

Ausgabe: Distanz d(s,t)

SSSP: Single source shortest paths

Eingabe: Graph G, Startknoten s

Ausgabe: Distanzen d(s,v) für alle Knoten v

APSP: All-pairs shortest paths

Eingabe: Graph G

2 https://de.wikipedia.org/wiki/K%C3%BCrzester_Pfad

Ausgabe: Distanzen d(v,w) für alle Knoten v,w

Auf den nächsten Seiten lernen wir zwei Algorithmen zum Berechnen des kürzesten Weges kennen.

83 Dijkstra Algorithmus

Auf dieser Seite wird der Dijkstra Algorithmus[1] behandelt. Der Dijkstra Algorithmus wird zur Berechnung des kürzesten Weges benutzt (SSSP). Der Algorithmus stammt von 1959. Es erfolgt eine iterative Erweiterung einer Menge von günstig erreichbaren Knoten. Der Greedy Algorithmus hat eine ähnliche Breitensuche ist aber nur für nichtnegative Gewichte. Er berechnet iterativ verfeinert die Distanzwerte d(v,w) und es gibt eine Prioritätswarteschlange zum Herauslesen des jeweils minimalen Elements.

83.1 Priority Queues

Eine Priority-Queue P ist eine dynamische Datenstruktur, die (mindestens) die folgenden Operationen unterstützt:

- P.add(Element): Element hinzufügen
- P.poll(): Minimalste Element zurückgeben
- P.contains(Element): Enthält P das Element?

Die Ordnung zur Sortierung muss dabei vorab definiert sein.

Ein Heap kann beispielsweise zur Implementierung einer Priority-Queue benutzt werden (add-Operation ist dann O(log n), poll-Operation O(log n), und contains-Operation ist O(n)). Benutzt man zusätzlich zum Heap noch einen binären Suchbaum auf denselben Element so ist auch contains in O(log n) realisierbar.

83.1.1 Priority Queue in Java

```
class DijkstraComparator implements Comparator<Integer>{
   Map<Integer,Integer> d = new HashMap<Integer,Integer>();

   public DijComparator(Map<Integer,Integer> d){
      this.d = d;
   }

   public int compare(Integer o1, Integer o2) {
      return d.get(o1).compareTo(d.get(o2));
   }
}
```

Ist d eine Map "Knoten"->"Aktueller Distanzwert von s aus", so ist PriorityQueue<Integer> queue = new PriorityQueue<Integer>(g.getNumberOfNodes(),new DijkstraCompara-

1 https://de.wikipedia.org/wiki/Dijkstra-Algorithmus

tor(d)); eine Priority-Queue, die bei iterativen Aufruf queue.poll() immer das Element mit dem minimalsten d-Wert zurückliefert.

83.2 Idee

1. Initialisiere alle Distanzwerte von s zu v mit ∞ (und von s zu s mit 0)
2. Initialisiere eine Priority-Queue Q mit allen v
3. Extrahiere das minimale Element w_{min} aus Q
4. Aktualisiere alle Distanzwerte der Nachfolger von w_{min} in Q:

- Ist es günstiger über w_{min} zu einem Knoten w zu kommen?
- Falls ja setzte d(s,w)=d(s,w_{min})+y(w_{min},w)

5. Wiederhole bei 3 solange Q noch Elemente hat

83.3 Algorithmus in Java

```
Map<Integer,Integer> dijkstra(Graph g, int s){
   Map<Integer,Integer> d = new HashMap<Integer, Integer>();
   PriorityQueue<Integer> queue = //Initialisiere Priority-Queue entsprechend
   for(Integer n: g){
      if(!n.equals(s)){
         d.put(n, Integer.MAX_VALUE);
         queue.add(n);
      }
   }
   d.put(s, 0);
   queue.add(s);

   while(!queue.isEmpty()){
      Integer u = queue.poll();
      for(Integer v: g.getChildren(u)){
         if(queue.contains(v)){
            if(d.get(u) + g.getWeight(u,v) < d.get(v){
               d.put(v, d.get(u) + g.getWeight(u,v));
            }
         }
      }
   }
   return d;
}
```

83.4 Algorithmus

algorithm Dijkstra (G,s)

Eingabe: Graph G mit Startknoten s

for each Knoten u ∈ V[G] -s do // Initialisierung

D[u] := ∞

od;

D[s]:= O; PriorityQueue Q := V;

while not isEmpty (Q) do

U := extractMinimal (Q);

for each v ∈ ZielknotenAusgehenderKanten (u) ∩ Q do

if D[u] + γ ((u,v)) < D[v] then // Entfernung über u nach v kleiner als aktuelle Entfernung D[v]

D[v] := D[u] + γ ((u,v));

adjustiere Q an neuen Wert D[v]

fi

od

od

83.4.1 Initialisierung

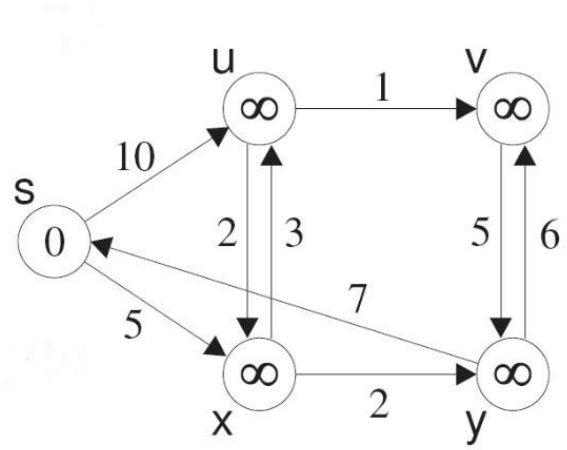

Abb. 60 Dijkstra

$D[s]+\gamma(s,u) < D[u]$?

$0+10 < \infty$

$\Rightarrow D[u] = 10$

$D[s]+\gamma(s,x) < D[x]$?

$0+5 < \infty$

$\Rightarrow D[x] = 5$

$Q = \langle (s:0),(u:\infty),(v:\infty),(y:\infty)\rangle$

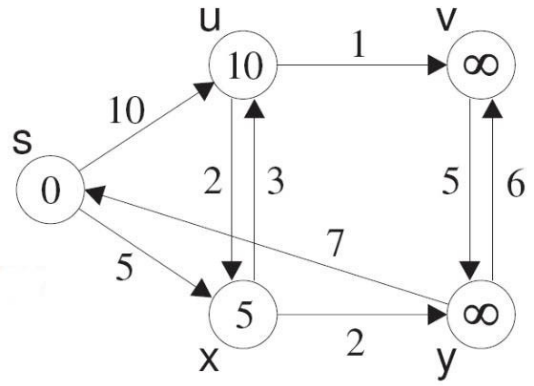

Abb. 61 Dijkstra2

$D[x] + \gamma(x,u) < D[u]?$

$5 + 3 < 10$

$\Rightarrow D[u] = 8$

$D[y] analog$

$Q = \langle (x:5), (u:10), (v:\infty), (y:\infty) \rangle$

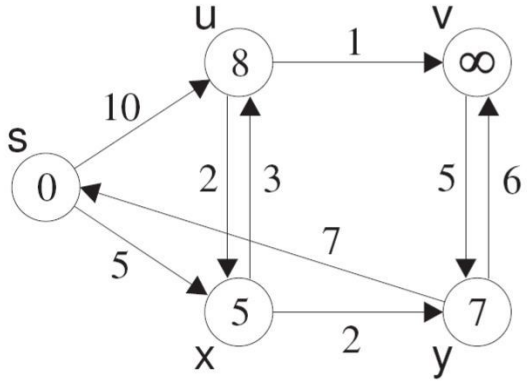

Abb. 62 Dijkstra3

$Q = \langle (y:7), (u:8), (v:\infty) \rangle$

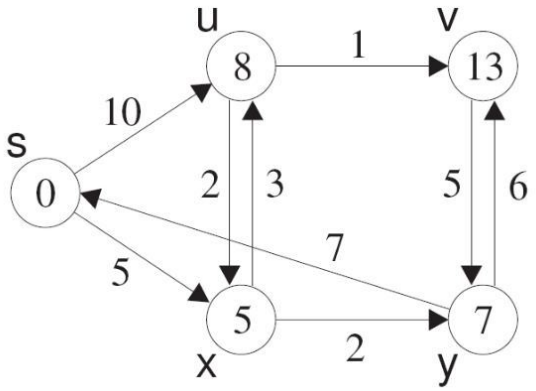

Abb. 63 Dijkstra4

$Q = \langle (u : 8), (v : 13) \rangle$

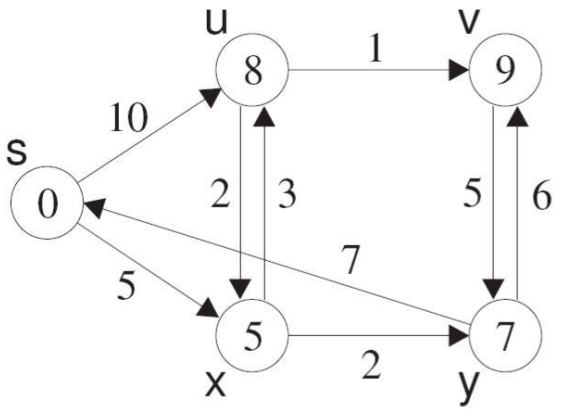

Abb. 64 Dijkstra5

$Q = \langle (v : 9) \rangle$

Der Iterationsstart ist korrekt für die Tiefe 0. Wir nehmen an, dass der vorherige Iterationsschritt korrekt war (Induktionsbeweis). Der Ein Iterationsschritt ist jeweils die günstigste Verbindung zu einem noch nicht bearbeiteten Knoten hinzunehmen. Da die bisher bearbeiteten Knoten den korrekten Distanzwert haben, ist der neue Distanzwert durch den „günstigsten“ aus dem bisher bearbeiteten Teilgraphen um genau eine Kante hinausgehenden Pfad bestimmt. Jeder Pfad zum Zielknoten dieses Pfades, der um mehr als eine Kante aus dem bearbeiteten Bereich hinausgeht, ist teurer als die gewählte, da Kosten mit zusätzlich hinzu genommenen Kanten nicht sinken können.

83.5 Analyse

83.5.1 Terminierungstheorem

Der Algorithmus von Dijkstra terminiert für eine endliche Eingabe nach endlicher Zeit.

Beweis

In jedem Schritt der while-Schleife wird ein Element aus queue entfernt und die Schleife endet sobald queu leer ist. Jeder Knoten hat nur endliche viele Kinder, deswegen ist auch die Laufzeit der inneren for-Schleife endlich.

83.5.2 Korrektheitstheorem

Sind alle Kantengewichte nicht-negativ, so enthält d am Ende die Distanzwerte von s zu allen anderen Kno

Beweis

Beachte, dass sobald ein Knoten v aus queue entfernt wird, der Wert für v in d nicht mehr geändert wird.

Zeige nun, dass gilt: Wird v aus queue entfernt, so enthält d den Distanzwert von s nach v. Zeige dies dur

- i=0: Am Anfang hat queue nur für s einen endlichen Wert gespeichert, alle anderen Werte sind ∞. Der K selbst Distanz 0 hat und alle anderen Knoten keine geringere Distanz von s aus haben können (da alle K
- i → i+1: Sei v der (i+1)te Knoten, der aus queue entfernt wird.
  - Da die bisher bearbeiteten Knoten den korrekten Distanzwert haben, ist der neue Distanzwert durch den „günstigsten“ aus dem bisher bearbeiteten Teilgraphen um genau eine Kante hinausgehenden Pfad bestimmt.
  - Jeder Pfad zum Zielknoten dieses Pfades, der um mehr als eine Kante aus dem bearbeiteten Bereich hinausgeht, ist teurer als die gewählte, da Kosten mit zusätzlich hinzugenommenen Kanten nicht sinken können.

83.5.3 Laufzeittheorem

Sei G=(V,E,g) ein gerichteter Graph. Der Laufzeitaufwand von Dijkstras Algorithmus für einen beliebigen Knoten s in G ist O((|E| + |V|) log |V|).

Beweis

Beachte: Wird für die Priority-Queue beispielsweise ein Heap verwendet, so hat die Operation poll() einen Aufwand von O(log k) (mit k=„Anzahl Elemente in Queue“). Sei |V|=n und |E|=m. Insgesamt: O(n log n) + O(n) + n* O(log n) + m *O(log n) = O((m + n) log n) Durch Benutzung sog. Fibonacci-Heaps (anstatt normaler Heaps) kann die Laufzeit von O((m + n) log n) verbessert werden zu O(m + n log n)

83.6 Nachteile

Der kürzeste Weg wird immer gefunden, aber es werden viele unnötige und sinnlose Wege gegangen. Bei negativen Kanten resultieren auch falsche Ergebnisse.

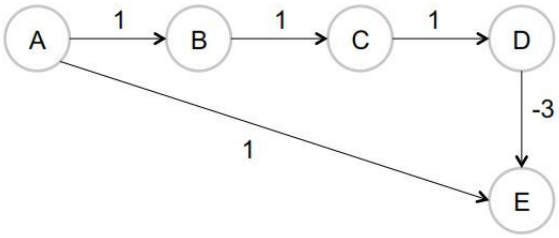

Abb. 65 Nachteil dijkastra

84 Bellmann-Ford

Auf dieser Seite wird der Bellmann-Ford Algorithmus[1] behandelt. Bei Dijkstra dürfen nur nichtnegative Gewichte benutzt werden. Doch gibt es auch eine Variante mit negativen Gewichten? Das würde nur bei gerichteten Graphen Sinn machen. Das Problem sind Zyklen mit negativem Gesamtgewicht. Ein Beispiel für Gewinn statt Kosten ist beispielsweise ein Verbindungsnetz mit Bonus Gewinnen für bestimmte Verbindungen um Auslastungen zu erhöhen. Dies ist bei Flügen mit Zwischenstopps der Fall, die oft billiger sind. Dieser Algorithmus löst ebenfalls das SSSP Problem.

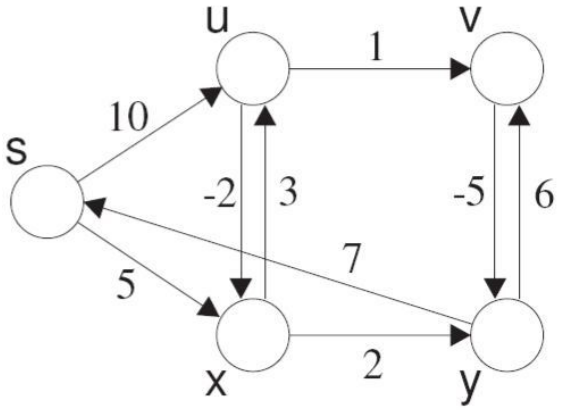

Abb. 66 Beispielgraph

84.1 Prinzip

Der Algorithmus erfolgt in mehreren Durchläufen. Es wird zunächst die bisher beste mögliche Verbindung bestimmtl, die die um eine Kante länger ist. Der i-te Durchlauf berechnet korrekt alle Pfade vom Startknoten der Länge i. Der längste Pfad ohne Zyklus hat eine Länge kleiner als |V|-1, somit hat man spätestens nach |V|-1 Durchläufen ein stabiles Ergebnis. Sollte das Ergebnis nach |V|-1 Durchläufen nicht stabil sein, so ist ein negativ bewerteter Zyklus enthalten. Hierbei wird das Prinzip der dynamischen Programmierung verwendet.

1 `https://de.wikipedia.org/wiki/Bellmann-Ford`

84.2 Algorithmus

```
algorithm BF(G, s)
   Eingabe: ein Graph G mit Startknoten s

   D[s] = 0
   D[t] = ∞ for all other t
   for i := 1 to |V|-1 do
      for each (u,v)∈ E do
         if D[u]+γ((u,v)) < D[v] then
            D[v] := D[u] + γ((u,v))
         fi
      od
   od
```

84.3 Beispiel

Bei der Initialisierung wird der Startknoten auf den Wert 0 gesetzt und alle weiteren Knoten erhalten den Wert ∞.

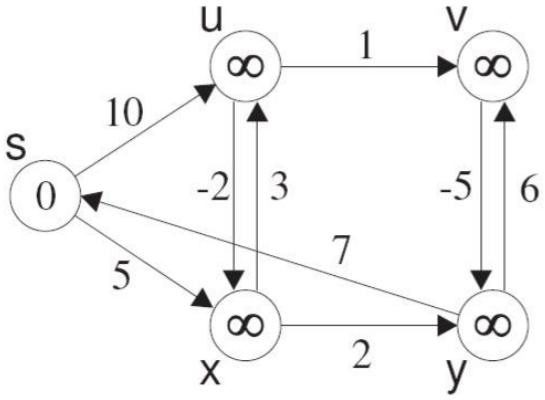

Abb. 67 BellmanFord

Beim ersten Schleifendurchlauf bekommt x den Wert 5 und u den Wert 10 zugewiesen.

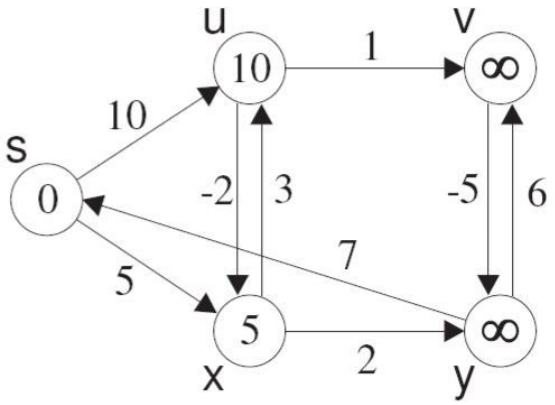

Abb. 68 BellmanFord2

Im zweiten Schleifendurchlauf werden alle weiteren Verbindungen aktualisiert, sowohl von u als auch von x. Dabei ändern sich die Werte von v, y und auch u. Die Änderung an u wird aber erst im nächsten Schritt an v propagiert.

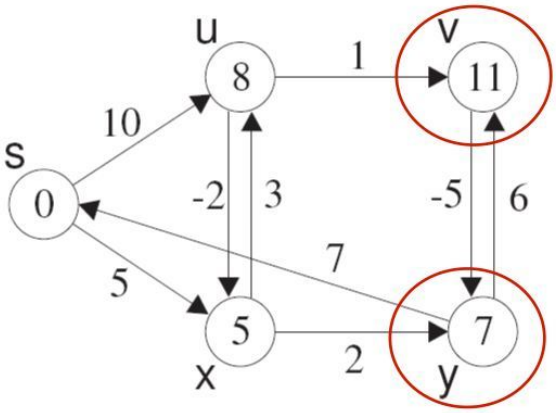

Abb. 69 BellmanFord3

Im dritten, i=3, Schleifendurchlauf verändern sich diesmal nur noch die Werte der Knoten v und y. Der neue Wert aus y berechnet sich durch den vorherigen Wert aus v=11 und der negativ gewichteten Kante -5. Hier wird also die negativ gewichtete Kante (v,y) zur Berechnung von D[y] genutzt.

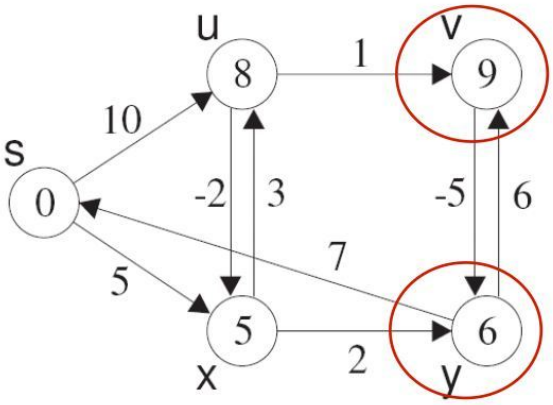

Abb. 70 BellmanFord4

Im vierten, i=4, Schleifendurchlauf wird nochmals die negativ gewichtete Kante (v,y) zur Berechnung von D[y] genutzt. Das Greedy-Verfahren, das jeden Knoten nur einmal besucht, hätte für y den in jedem Schritt lokal optimalen Pfas $< s, x, y >$ gewählt und nicht das beste Ergebnis geliefert.

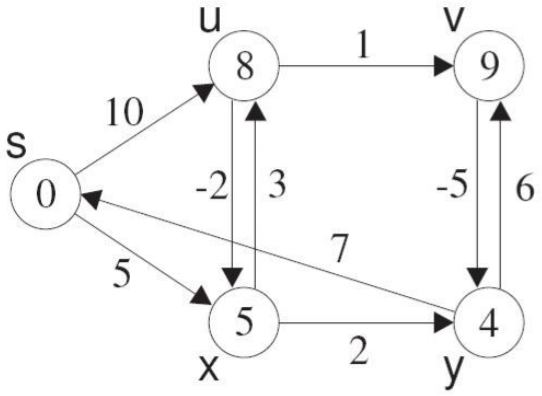

Abb. 71 BellmanFord5

84.4 Analyse

84.4.1 Terminierungstheorem

Der Algorithmus BF(G,s) terminiert für eine endliche Eingabe G in endlicher Zeit.

Beweis

Alle Schleifen sind endlich.

84.4.2 Korrektheitstheorem

Ist G ein Graph, der keinen Zyklus mit negativem Gewicht hat, so enthält D nach Aufruf BF(G,s) die Distanzwerte von s zu allen Knoten.

Beweis

Wir zeigen, dass die folgenden Aussagen Schleifeninvariante der for- Schleife (Schleifenvariable i) sind:

1. Ist D[v] $< \infty$, so ist D[v] der Wert eines Pfades von s nach v
2. Ist D[v] $< \infty$, so ist D[v] der kleinste Wert eines Pfades von s nach v mit maximal i Kanten
3. D[v] $< \infty$gdw. es einen Pfad von s nach v mit gleich oder weniger als i Kanten gibt

Da G keine Zyklen mit negativem Gewicht hat, ist die Länge des längsten kürzesten Pfades maximal |Anzahl Knoten|-1 (jeder Knoten wird auf diesem Pfad einmal besucht). Also gilt nach dem letzten Schleifendurchlauf nach 2 und 3. die Aussage des Theorems. Wir zeigen diese Aussagen durch Induktion nach i(=#Schleifendurchläufe).

- Bei i=0 gilt vor dem ersten Schleifendurchlauf nur D[s]=0 $< \infty$. Daraus folgt direkt 1., 2., 3.
- Bei i -> i+1 beweisen wir zunächst Aussage 3.
  - War D[v] schon vorher endlich, so gilt die Aussage nach IV.
  - Ist D[v] in diesem Schritt auf einen endlichen Wert gesetzt worden, so gab es ein u, so dass D[u] vorher schon endlich war und D[v]=D[u]+γ(u,v). Nach IV gibt es einen Pfad von s nach u der Länge i. Damit gibt es einen Pfad der Länge i+1 von s nach v.
  - Umgekehrt wird bei Existenz eines Pfades der Länge i+1 dieser auch gefunden und D[v] auf einen endlichen Wert gesetzt.
- Die Aussage 1 wird dadurch bewiesen, dass nach IV der Wert eines Pfades von s nach u D[u] ist. Wird D[v]=D[u]+γ(u,v) gesetzt so ist somit D[v] der Wert des Pfades von s nach v über u.
- Die Aussage 2 wird dadurch bewiesen, dass nach IV der kleinste Wert eines Pfades von s nach v mit maximal i Kanten D[v] ist. Mache folgende Fallunterscheidung:
  - 1.Fall: Es existiere ein Pfad P1 von s nach v mit i+1 Kanten, der minimalen Wert unter allen Pfaden von s nach v mit gleich oder weniger als i+1 Kanten hat. Betrachte den vorletzten Knoten u auf diesem Pfad und den Teilpfad P2 von P1 von s nach u. Dieser Teilpfad hat minimalen Wert unter allen Pfaden der maximalen Länge i von s nach u (ansonsten wäre P1 kein Pfad mit minimalem Wert). Nach IV ist D[u] genau dieser Wert und D[u]+γ(u,v) der Wert von P1, der dann im i+1ten Durchgang aktualisiert wird.
  - 2.Fall: Es existiere kein Pfad von s nach v mit i+1 Kanten, der minimalen Wert unter allen Pfaden von s nach v mit gleich oder weniger als i+1 Kanten hat.
    - 1. Unterfall: Es existiert kein Pfad von s nach v mit maximal i+1 Kanten. Dann bleibt nach 3. D[v]=∞.
    - 2. Unterfall: Es existiert ein Pfad von s nach v mit k<i+1 Kanten, der minimalen Wert unter allen Pfaden von s nach v mit gleich oder weniger als i+1 Kanten hat.

Dann ist nach IV D[v] genau dieser Wert und wird im i+1ten Durchgang auch nicht aktualisiert.

84.4.3 Graph mit negativ gewichtetem Zyklus

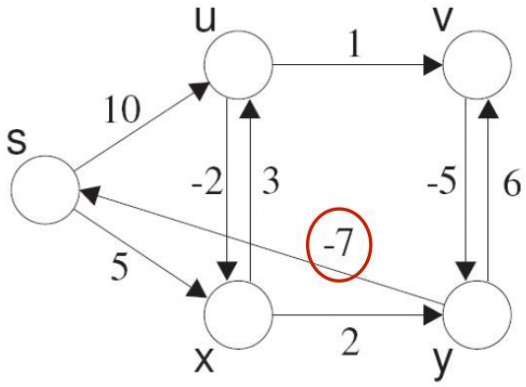

Abb. 72 BellmanFord Negativ

Betrachten wir die Situation nach |V|-1 Iterationen. Eine Kante könnte noch verbessert werden genau dann wenn der Graph einen Zyklus negativer Länge enthält. Der Zyklus s,x,u,v,y,s hat die Kosten 5+3+1-5-7=-3. Jeder Durchlauf durch den Zyklus erzeugt also einen Gewinn. Es gibt hier keinen günstigen Pfad endlicher Länge!

84.4.4 Laufzeittheorem

Sei G=(V,E,g) ein gerichteter Graph. Der Laufzeitaufwand vom Algorithmus von Bellmann-Ford für einen beliebigen Knoten s in G ist O(|V||E|).

Beweis

Einfache Schleifenanalyse.

85 Floyd-Warshall

Auf dieser Seite wird der Floyd-Warshall Algorithmus[1] behandelt. Der Dijkstras Algorithmus und Bellman-Ford berechnen zu einem gegebenen Startknoten die kürzesten Wege zu allen anderen Knoten (Single Source Shortest Paths – SSSP. Aber wie kann man die kürzesten Wege zwischen zwei Knoten v und w berechnen? Man könnte die bereits kennengelernten Algorithmen für jeden einzelnen Startknoten neu aufrufen, doch das geht auch geschickter. Hier kommt der Floyd-Warshall Algorithmus ins Spiel, welcher das All Pairs Shortest Path Problem löst. Zwar nicht unbedingt effizienter, aber eleganter. Dies geschieht nach dem Prinzip der dynamischen Programmierung.

85.1 Problemdefinition

Gegeben ist ein Graph G=(V,E). Wir möchten für jedes Paar $(v, w) \in V \times V$ den Wert D(v,w) eines kürzesten Pfades finden. Wir nehmen an, dass es keine negativen Kreise gibt.

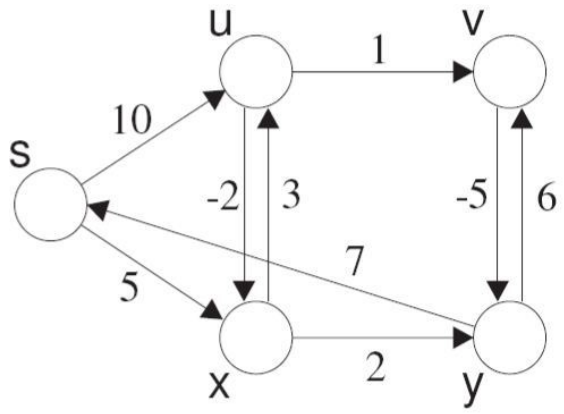

Abb. 73 Floyd Warshall

D	**s**	**u**	**v**	**x**	**y**
s	0	8	9	5	4
u	3	0	1	-2	-4
v	2	10	0	7	-5
x	6	3	4	0	-1
y	7	15	6	12	0

1 https://de.wikipedia.org/wiki/Algorithmus_von_Floyd_und_Warshall

85.2 Idee

Die Grundidee des Floyd-Warshall Algorithmus ist, dass wenn ein kürzester Weg $\{(v,a_1),...,(a_n,k),(k,a_{n+1}),...,(a_m,w)\}$ von v nach w über k geht, dann gilt:

- $\{(v,a_1),...,(a_n,k)\}$ ist ein kürzester Weg von v nach k
- $\{(k,a_{n+1}),...,(a_m,w)\}$ ist ein kürzester Weg von k nach w

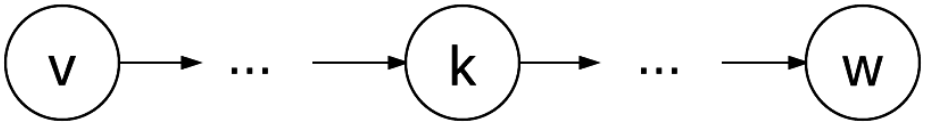

Abb. 74

Im obigen Beispiel gilt folgendes:

- $s \to y : \{(s,x),(x,u),(u,v),(v,y)\}$
- $s \to u : \{(s,x),(x,u)\}$
- $u \to y : \{(u,v),(v,y)\}$

Die Umkehrung gilt jedoch nicht. Ist $\{(v,a_1),...,(a_n,k)\}$ ein kürzester Weg von v nach k und ist $\{(k,a_{n+1}),...,(a_m,w)\}$ ein kürzester Weg von k nach w dann gilt nicht notwendigerweise, dass $\{(v,a_1),...,(a_n,k),(k,a_{n+1}),...,(a_m,w)\}$ ein kürzester Weg von v nach w ist!

Im obigen Beispiel bedeutet dies:

- $x \to y : \{(x,y)\}$
- $y \to v : \{(y,v)\}$
- $x \to v : \{(x,y),(y,v)\}$ ist nicht der kürzeste Weg!

Jedoch gilt, wenn bekannt ist, dass ein kürzester Weg zwischen v und w nur Knoten aus $V' \subseteq V$ enthält, so gilt entweder der kürzeste Weg zwischen v und w benutzt nur Knoten aus $V'\backslash\{k\}$ oder der kürzeste Weg zwischen v und w ist Konkatenation aus dem kürzesten Weg zwischen v und k und dem kürzesten Weg zwischen k und w und beide Wege enthalten nur Knoten aus $V'\backslash\{k\}$.

$$D^{V'}[i,j] = fac(x) := \begin{cases} \gamma(i,j) & falls\ k=0 \\ min\{D^{V'\backslash\{k\}}[i,j], D^{V'\backslash\{k\}}[i,k] + D^{V'\backslash\{k\}}[k,j]\} & falls\ k \geq 1 \end{cases}$$

85.3 Algorithmus

```
algorithm FW(G)
   Eingabe: ein Graph G

   for each v,v‘ V
      D[v,v‘] = γ((v,v‘)) (or )
   for each k  V do
      for each i  V do
```

```
        for each j ∈ V do
           if D[i,k]+D[k,j] < D[i,j] then
              D[i,j] := D[i,k]+D[k,j]
           fi
        od
     od
  od
```

85.4 Beispiel

Initialisiere D mit den Kantengewichten. Nicht vorhandene Kanten haben das Gewicht ∞. Die Kantengewichte zum Knoten selber sind 0. Im folgenden betrachten wir nur Schleifendurchgänge mit $k \neq i, k \neq j, i \neq j$

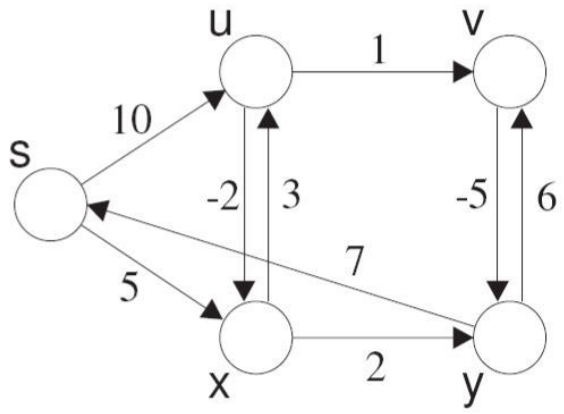

Abb. 75 Floyd Warshall

D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	∞	6	∞	0

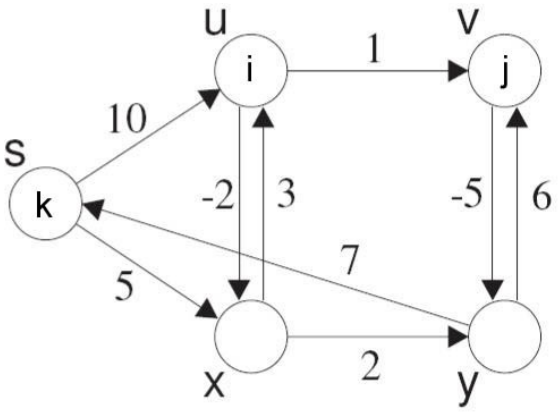

Abb. 76 1: D[u,s]+D[s,v]<D[u,v]?

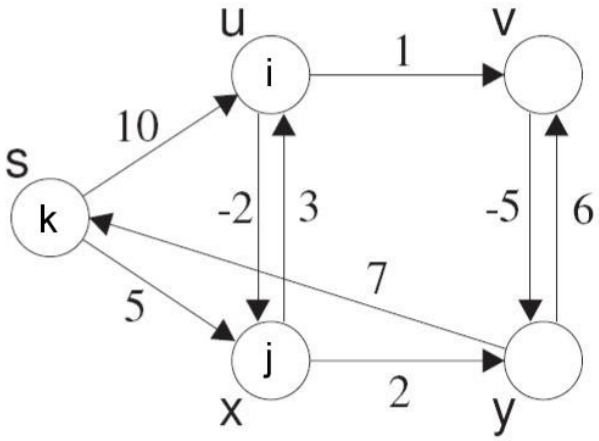

Abb. 77 2: D[u,s]+D[s,x]<D[u,x]?

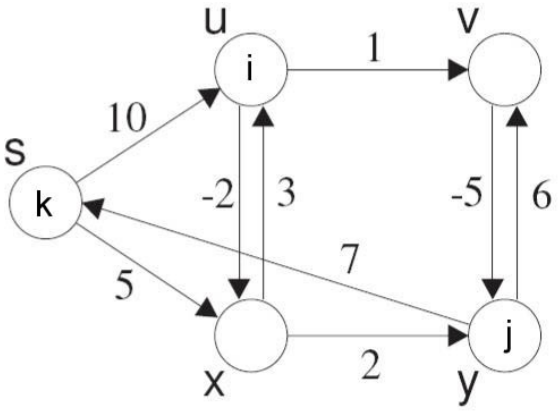

Abb. 78 3: D[u,s]+D[s,y]<D[u,y]?

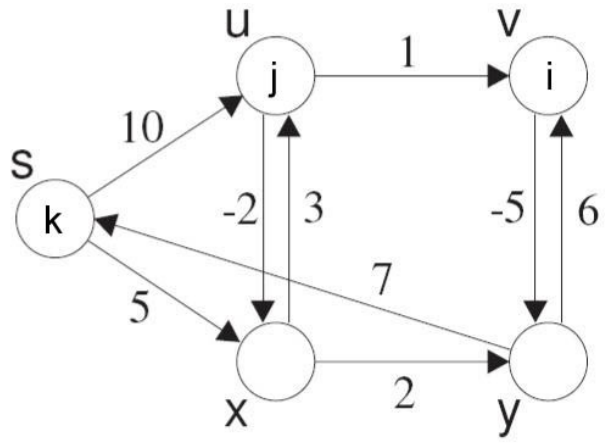

Abb. 79 4: D[v,s]+D[s,u]<D[v,u]?

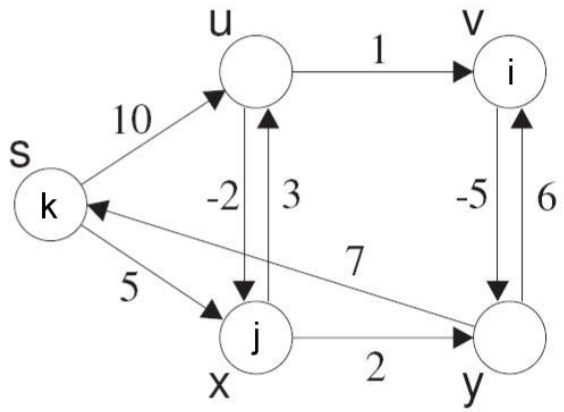

Abb. 80 5: D[v,s]+D[s,x]<D[v,x]?

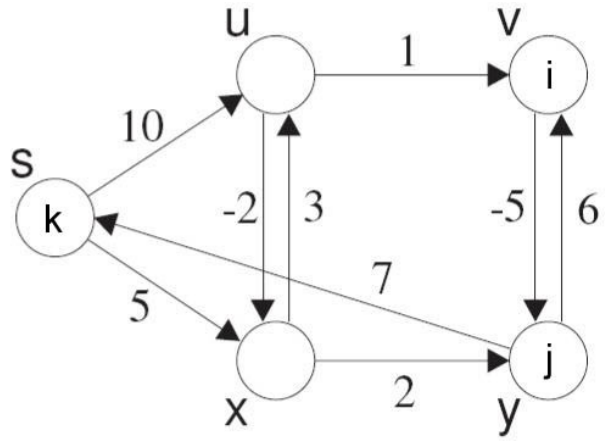

Abb. 81 6: D[v,s]+D[s,y]<D[v,y]?

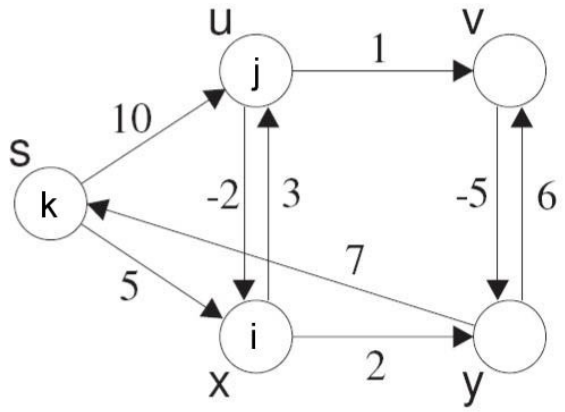

Abb. 82 7: D[x,s]+D[s,u]<D[x,u]?

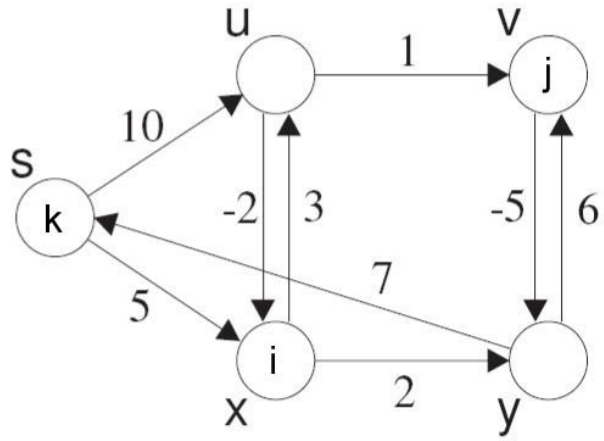

Abb. 83 8: D[x,s]+D[s,v]<D[x,v]?

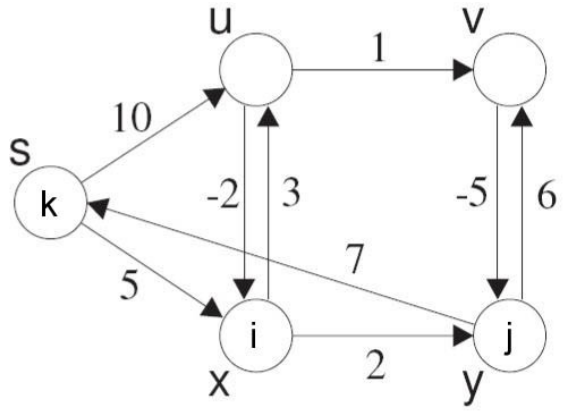

Abb. 84 9: D[x,s]+D[s,y]<D[x,y]?

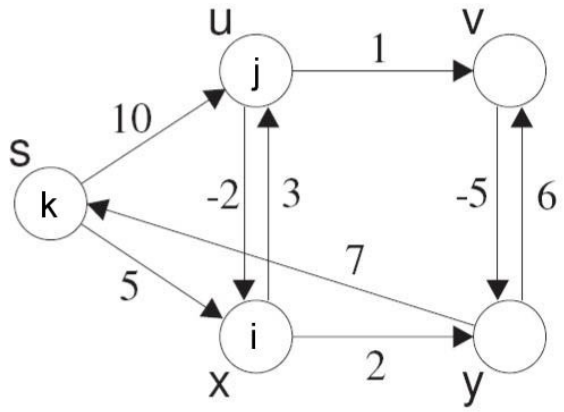

Abb. 85 10: D[y,s]+D[s,u]<D[y,u]? → D[y,u]= D[y,s]+D[s,u]=7+10=17

D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	17	6	∞	0

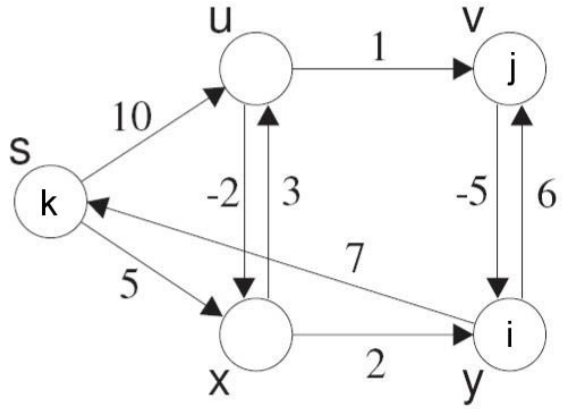

Abb. 86 11: D[y,s]+D[s,v]<D[y,v]?

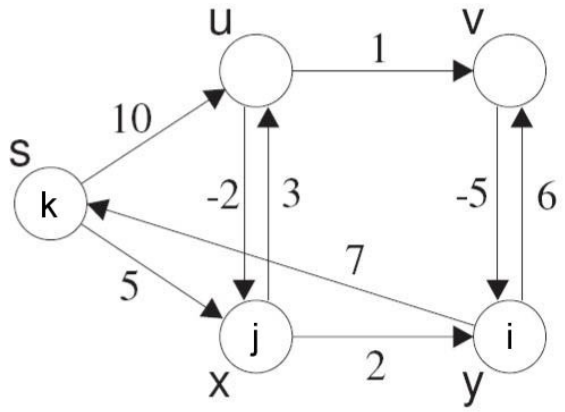

Abb. 87 12: D[y,s]+D[s,x]<D[y,x]? → D[y,x]= D[y,s]+D[s,x]=7+5=12

D	s	u	v	x	y
s	0	10	∞	5	∞
u	∞	0	1	-2	∞
v	∞	∞	0	∞	-5
x	∞	3	∞	0	2
y	7	17	6	12	0

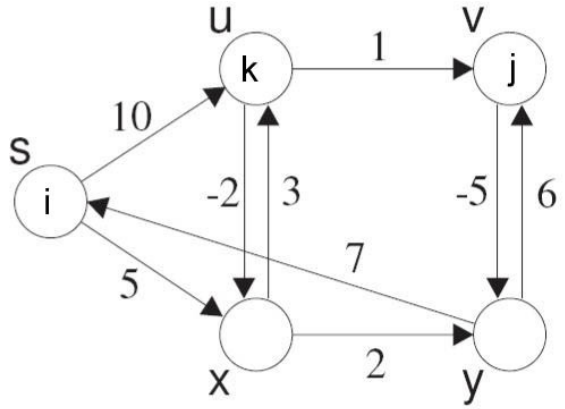

Abb. 88 13: D[s,u]+D[u,v]<D[s,v]? → D[s,v]= D[s,u]+D[u,v]=10+1=11

Führt man den Algorithmus weiter durch, kommt man zu folgendem Endergebnis:

D	s	u	v	x	y
s	0	8	9	5	4
u	3	0	1	-2	-4
v	2	10	0	7	-5
x	6	3	4	0	-1
y	7	15	6	12	0

85.5 Analyse

85.5.1 Terminierungstheorem

Der Algorithmus FW(G) terminiert für eine endliche Eingabe G in endlicher Zeit.

Beweis

Alle Schleifen sind endlich.

85.5.2 Korrektheitstheorem

Ist G ein Graph, der keinen Zyklus mit negativem Gewicht hat, so enthält D nach Aufruf FW(G) die Distanzwerte von allen Knoten zu allen anderen Knoten.

Beweis

Betrachte dazu folgende Schleifeninvariante, die äußerste for-Schleife mit der Laufvariablen k): Nach der k-ten Schleifeniteration gilt, dass D[v,w], für alle v,w, der Wert eines kürzesten Pfades ist, der nur Knoten 1,...,k benutzt. Wenn der Algorithmus endet, gilt damit die Aussage des Theorems. Dies zeigen wir durch Induktion.

- k=0 (bei der Initialisierung): Nach der Initialisierung gilt D[v,w]= ∞gdw. es keine Kante von v nach w gibt. Das bedeutet, dass jeder Pfad zwischen v und w mindestens einen anderen Knoten enthalten haben muss. Ist D[v,w] endlich, so ist dies genau der Wert der Kante. Dann gibt es also einen Pfad, der keine weiteren Knoten beinhaltet.
- k -> k+1: Nach der Induktionsannahme ist D[v,w] der Wert eines kürzestens Pfades, der nur Knoten aus 1,...,k enthält. Im k+1-Schleifendurchgang wird überprüft, ob es einen kürzeren Weg über k+1 gibt und ggfs. aktualisiert. Es wird also genau folgende Gleichung ausgenutzt:

$$D^{V'}[i,j] = \begin{cases} \gamma(i,j) & \text{falls } V' = \emptyset \\ min\{D^{V'\setminus\{k\}}[i,j], D^{V'\setminus\{k\}}[i,k] + D^{V'\setminus\{k\}}[k,j]\} & \text{für } k \in V' \end{cases}$$

Anschließend ist also D[v,w] der Wert eines kürzestens Pfades, der nur Knoten 1,...,k+1 benutzt.

Ein anderer Ansatz ist dies per Induktion nach der kürzesten Länge eines kürzesten Weges für jedes Knotenpaar (v,w) zu zeigen. Anmerkung: zwischen v und w können mehrere Wege mit minimalem Gewicht existieren, diese können auch unterschiedliche Länge haben. Angenommen zwischen v und w existiert ein kürzester Weg der Länge 1, dann ist der Wert dieses Weges gleich dem Wert der Kante (die existieren muss. Dieser wird in der Initialisierungsphase gesetzt und später nicht mehr geändert. Angenommen zwischen v und w gibt es einen kürzesten Pfad (=minimales Gewicht) der Länge l≥ 2 , dann gibt es einen Knoten k auf diesem Pfad, so dass zum einen

der Teilpfad von v nach k ein kürzester Weg von v nach k ist und zum anderen, dass der Teilpfad von k nach w ein kürzester Weg von k nach w ist. Somit haben beide Pfade haben Länge $<$ l, d.h. die Werte D[v,k] und D[k,w] müssen schon korrekt berechnet sein (die Induktionsvoraussetzung greift).
Da alle potentiellen "Mi5elknoten" überprüft werden, wird ein geeignetes k gefunden und der Wert D[v,w] aktualisiert.

85.5.3 Laufzeittheorem

Sei G=(V,E,g) ein gerichteter Graph. Der Laufzeitaufwand vom Algorithmus von Floyd-Warshall auf G ist $O(\mid V \mid^3)$.

Beweis

Einfache Schleifenanalyse.

86 Flussproblem

Auf dieser Seite wird das Flussproblem[1] behandelt. Die Bestimmung des maximalen Flusses muss in vielen logischen Aufgaben angewandt werden. Beispielsweise bei Verteilungsnetzen mit Kapazitäten wie Wasserrohren, Förderbändern oder Packetvermittlungen mit Rechnernetzen. Die Quellen liefert beliebig viele Objekte pro Zeiteinheit und die Senke verbraucht diese. Jede Verbindung hat eine maximale Kapazität c und einen aktuellen Fluss f. Wie hoch ist nun die Übertragungskapazität?

86.1 Definition Fluss

Ein Fluss f von $q \in V$ nach $z \in V$ ist eine Funktion $f_{q,z} : E \to \mathbb{R}$. Für diese Funktion $f_{q,z}$ gelten folgende zwei Bedingungen:

1. Die Kapazitäten werden eingehalten: $\forall e \in E : f_{q,z}(e) \leq c(e)$
2. Was in einen Knoten hereinfließt, muss wieder herausfließen, mit Ausnahme von q und z: $\forall v \in V \backslash \{q,z\} : \sum_{u \in P(v)} f((u,v)) = \sum_{w \in S(v)} f((v,w))$, wobei $P(v) = \{u | (u,v) \in E\}$ der Vorgänger von v ist und $S(v) = \{w | (v,w) \in E\}$ der Nachfolger von v ist.

Einschränkungen der Kapazität der Kanten werden eingehalten, auch bei negativem Fluss:

$$|f_{q,z}(u,v)| \leq c((u,v))$$

Außerdem ist der Fluss konsistent. Bei in beiden Richtungen nutzbaren Verbindungen wird als Nettoeffekt nur in eine Richtung gesendet und der entstehende negative Fluss nimmt den korrekten Wert an:

$$f_{q,z}(u,v) = -f_{q,z}(v,u)$$

Der Fluss wird für jeden Knoten $v \in V \backslash \{q,z\}$ mit Ausnahme der Quelle q und des Ziels z bewahrt:

$$\sum_{u \in V} f_{q,z}(v,u) = 0$$

Der Wert eines Flusses beträgt:

1 https://de.wikipedia.org/wiki/Flussproblem

$val(G, f_{q,z}) = \sum_{u \in S(q)} f_{q,z}(q,u)$

Gesucht wird der maximale Fluss:

$max\{val(G,f)|$ `f ist korrekter Fluss von q nach z`$\}$

86.2 Beispiel

Definiere f_1 durch

$f_1((1,2)) = 2,\ f_1((1,3)) = 4,\ f_1((2,4)) = 1,\ f_1((2,5)) = 1,\ f_1((3,2)) = 0,\ f_1((3,5)) = 4,\ f_1((4,5)) = 0,\ f_1((4,6)) = 1,\ f_1((5,6)) = 5.$

Daraus folgt, dass der Wert des Flusses 6 ist: $val(G, f_1) = 6$.

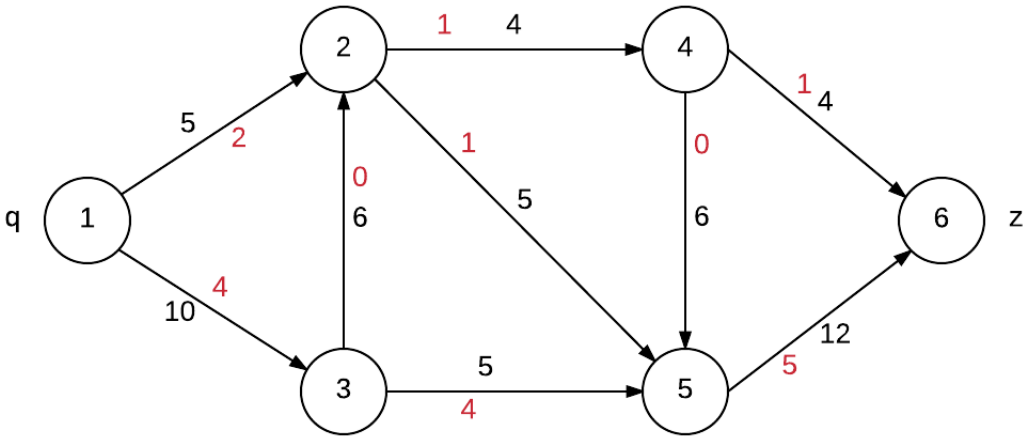

Abb. 89 ohne

Definiere f_2 durch

$f_2((1,2)) = 5,\ f_2((1,3)) = 3,\ f_2((2,4)) = 4,\ f_2((2,5)) = 1,\ f_2((3,2)) = 2,\ f_2((3,5)) = 1,\ f_2((4,5)) = 1,\ f_2((4,6)) = 3,\ f_2((5,6)) = 3.$

Daraus folgt, dass f_2 kein Fluss ist.

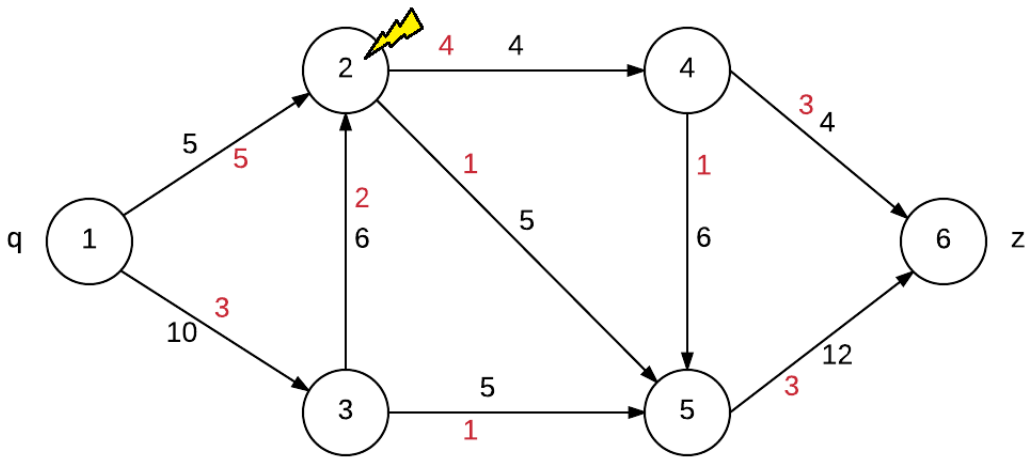

Abb. 90

Definiere f_3 durch

$f_3((1,2)) = 5,\ f_3((1,3)) = 9,\ f_3((2,4)) = 4,\ f_3((2,5)) = 5,\ f_3((3,2)) = 4,\ f_3((3,5)) = 5,\ f_3((4,5)) = 0,\ f_3((4,6)) = 4,\ f_3((5,6)) = 10.$

Daraus folgt, dass der Wert des Flusses 14 ist: $val(G, f_3) = 14$. Damit ist der Fluss f_3 maximal.

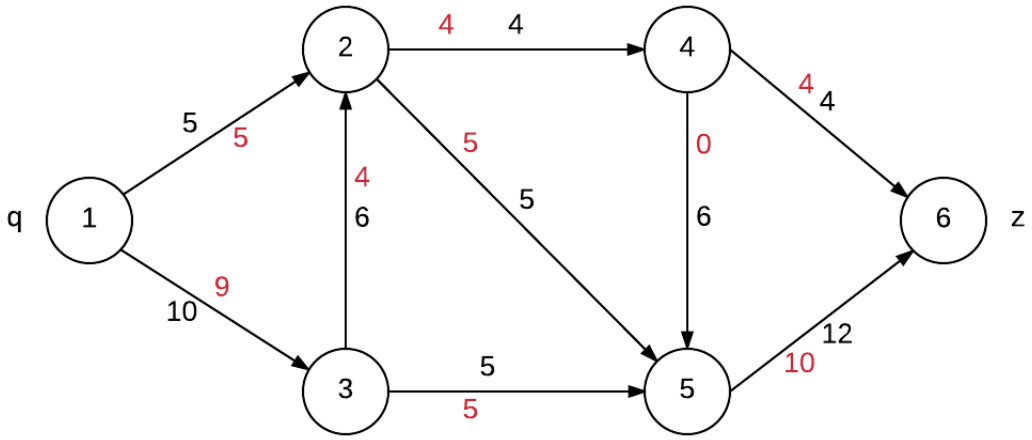

Abb. 91 ohne

87 Ford-Fulkerson

Auf dieser Seite wird der Ford Fulkerson[1] Algorithmus zur Berechnung des maximalen Flusses behandelt.

87.1 Berechnung des maximalen Flusses

Der Ford-Fulkerson Algorithmus ist ein effizienter Algorithmus zur Bestimmung eines maximalen Flusses von q nach z. Dabei wird der Greedy Algorithmus mit Zufallsauswahlen gemischt. Hier wird das Prinzip "Füge so lange verfügbare Pfade zum Gesamtfluss hinzu wie möglich" verfolgt. Zuerst soll ein nutzbarere Pfad durch Tiefensuche gefunden werden. Für die Kanten werden dann drei Werte notiert. Zum einen der aktuellen Fluss entlang der Kante. Im initialisierten Graphen ist dieser Wert überall 0. Zudem wird die vorgegebene Kapazität c notiert und die abgeleitete noch verfügbare Restkapazität von c-f.

87.2 Algorithmus

```
initialisiere Graph mit leerem Fluss;
do
   wähle nutzbaren Pfad aus;
   füge Fluss des Pfades zum Gesamtfluss hinzu;
while noch nutzbarer Pfad verfügbar
```

Ein nutzbarere Pfad ist ein zyklenfreier Pfad von der Quelle q zum Ziel z, der an allen Kanten eine verfügbare Kapazität hat. Ein nutzbarer Fluss ist das Minimum der verfügbaren Kapazitäten der einzelnen Kanten.

Der nachfolgende Pseudocode realisiert das Problem mit zusätzlichen Rückkanten.

```
für jede Kante(u,v) füge Kante (v,u) mit Kapazität 0 ein;
initialisiere Graph mit leerem Fluss;
do
   wähle nutzbaren Pfad aus;
   füge Fluss des Pfades zum Gesamtfluss hinzu;
while noch nutzbarer Pfad verfügbar
```

1 https://de.wikipedia.org/wiki/Algorithmus_von_Ford_und_Fulkerson

87.3 Beispiele

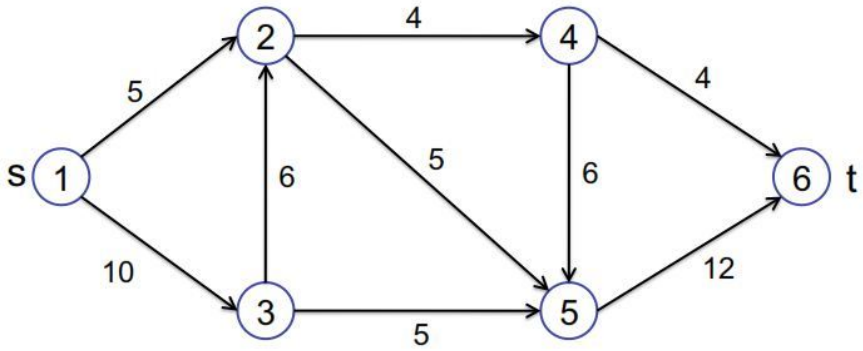

Abb. 92 FordFulkerson1

Wir haben einen Graph mit Kapazitäten gegeben

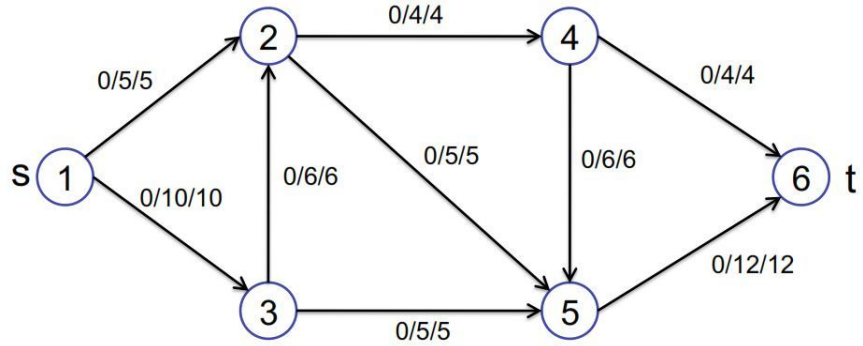

Abb. 93 FordFulkerson2

Es wird mit dem Fluss 0 initialisiert. Notation: <aktueller Fluss f> / <Kapazität c> / <verfügbare Kapazität c-f>

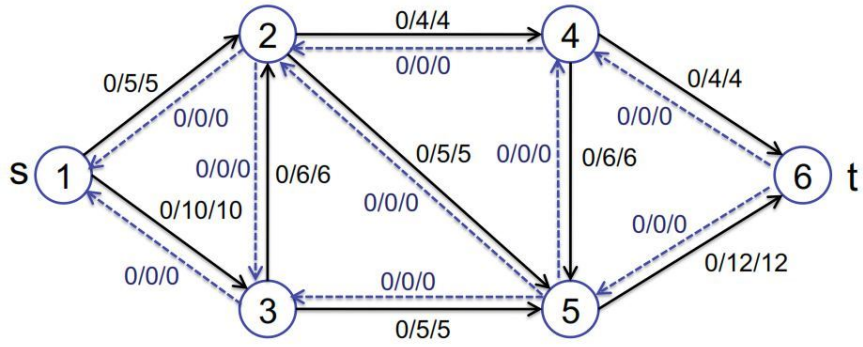

Abb. 94 FordFulkerson3

Die Auswahl der nutzbaren Pfade geschieht zufällig oder durch geeignete Heuristik. Es gibt auch kürzere Pfade mit höheren Kapazitäten. Die Rückkanten werden mit der Kapazität 0

eingefügt. Die Auswahl eines Pfades geschieht durch $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$ Der nutzbare Fluss beträgt 4.

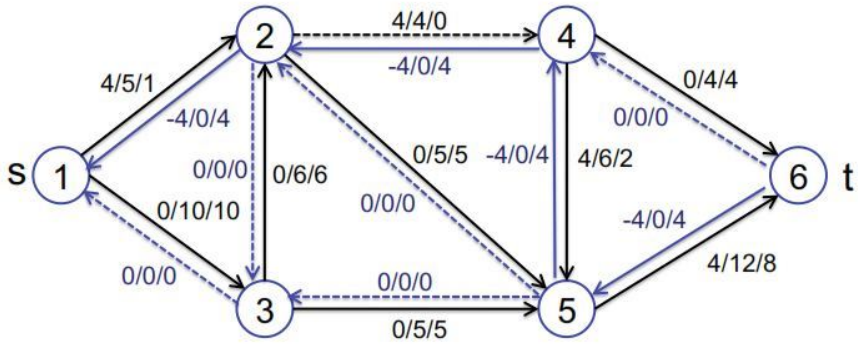

Abb. 95 FordFulkerson4

Der Fluss wird aktualisiert. Die Auswahl des Pfades ist nun : $1 \rightarrow 3 \rightarrow 5 \rightarrow 6$. Der nutzbare Fluss beträgt 5.

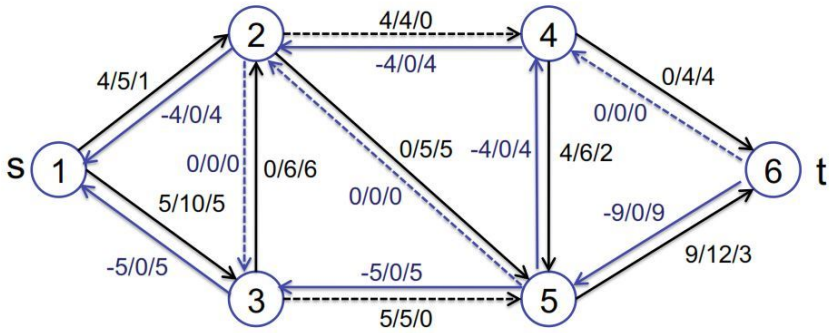

Abb. 96 Ford fulkerson5

Der Fluss wird aktualisiert. Die Auswahl des Pfades ist nun : $1 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 6$. Der nutzbare Fluss beträgt 3.

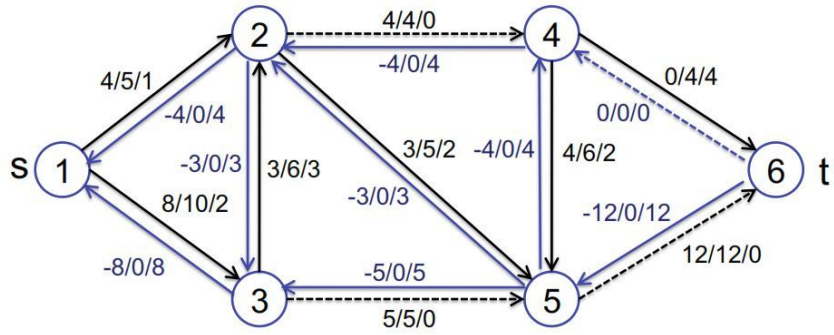

Abb. 97 Ford fulkerso6

Der Fluss wird aktualisiert. Die Auswahl des Pfades ist nun : $1 \to 3 \to 2 \to 5 \to 4 \to 6$. Der nutzbare Fluss beträgt 2.

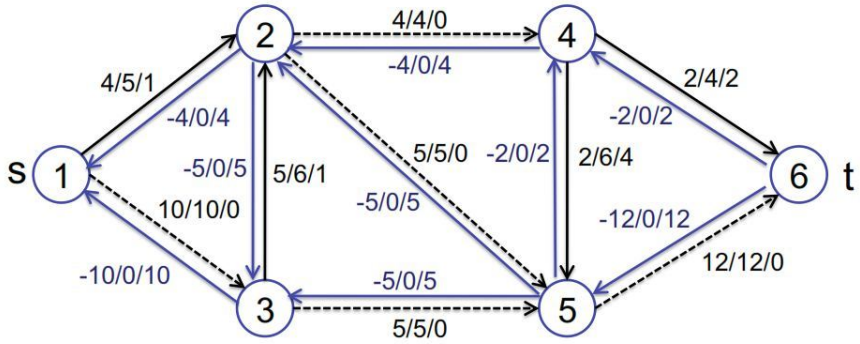

Abb. 98 Ford fulkerson7

An dieser Stelle sind keine Kapazitäten mehr über und die Berechnung wir beendet. Der maximale Fluss beträgt 14.

Der Algorithmus kann dabei auf verschiedene Ergebnisse kommen, jedoch ist der maximale Fluss immer gleich. Eine weitere Lösung ist folgende:

Zunächst wird der Pfad $1 \to 2 \to 5 \to 6$ mit dem nutzbaren Fluss 5 ausgewählt.

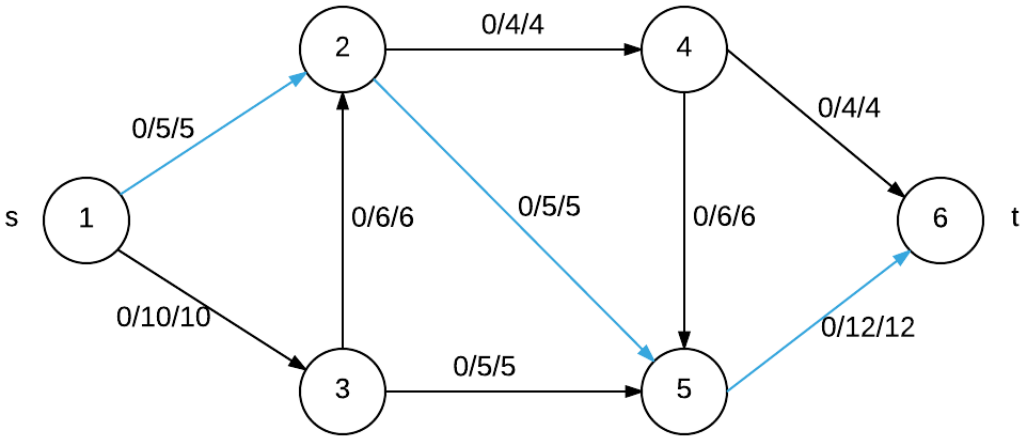

Abb. 99

Anschließend wird der Fluss aktualisiert. Im nächsten Schritt wird dann der Pfad $1 \to 3 \to 5 \to 6$ gewählt. Ebenfalls ist hier wieder ein nutzbarer Fluss von 5.

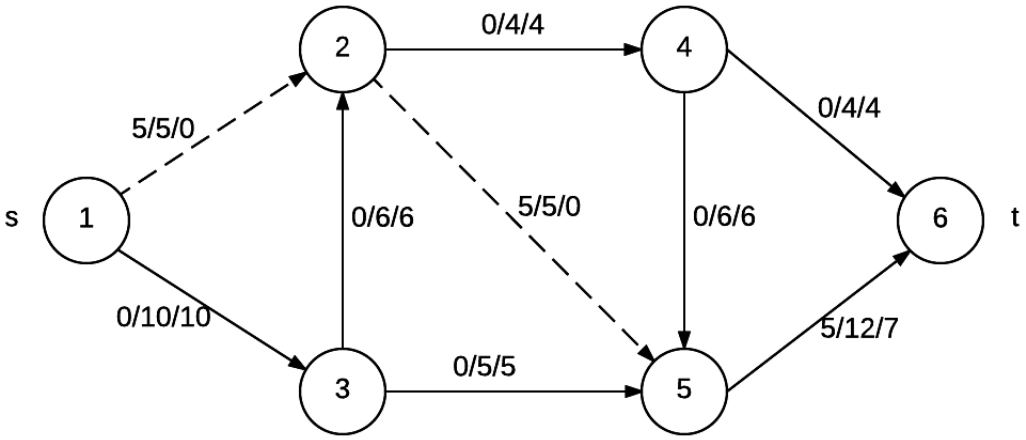

Abb. 100

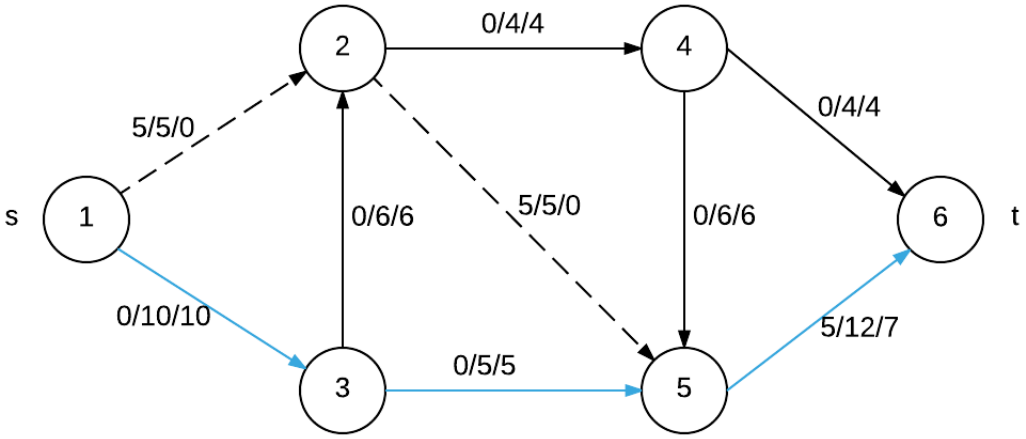

Abb. 101

Nach der zweiten Aktualisierung ist nur noch ein Pfad vom Start zum Ziel möglich. Also wird der Pfad $1 \to 3 \to 2 \to 4 \to 6$ ausgewählt. Dieser Fluss enthält allerdings nur noch einen nutzbaren Fluss von 4.

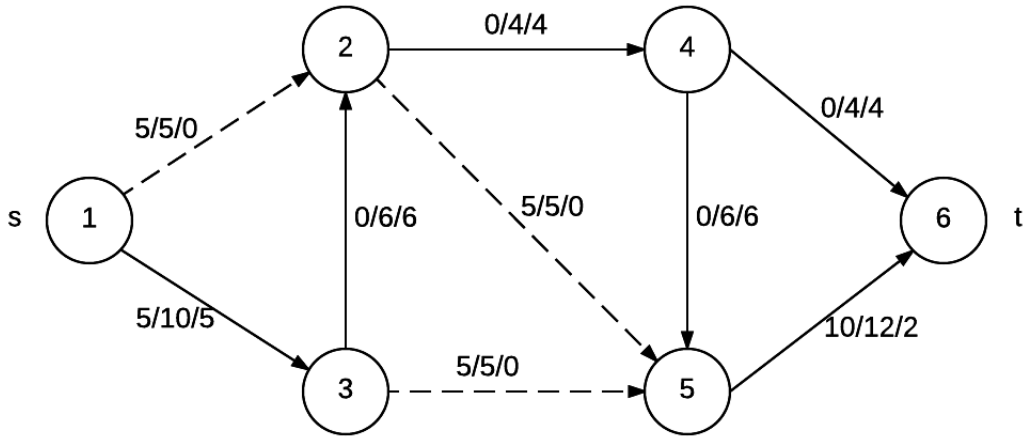

Abb. 102

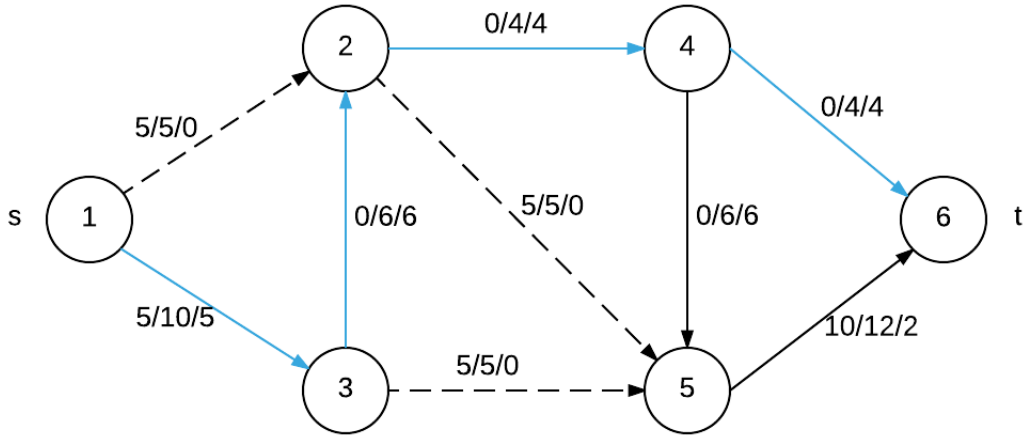

Abb. 103

Nach dem Aktualisieren des Flusses ist es nicht mehr möglich einen Pfad vom Start zum Ziel zu finden. Damit ist die Berechnung beendet. Wie zuvor berechnet ist der maximale Fluss 14.

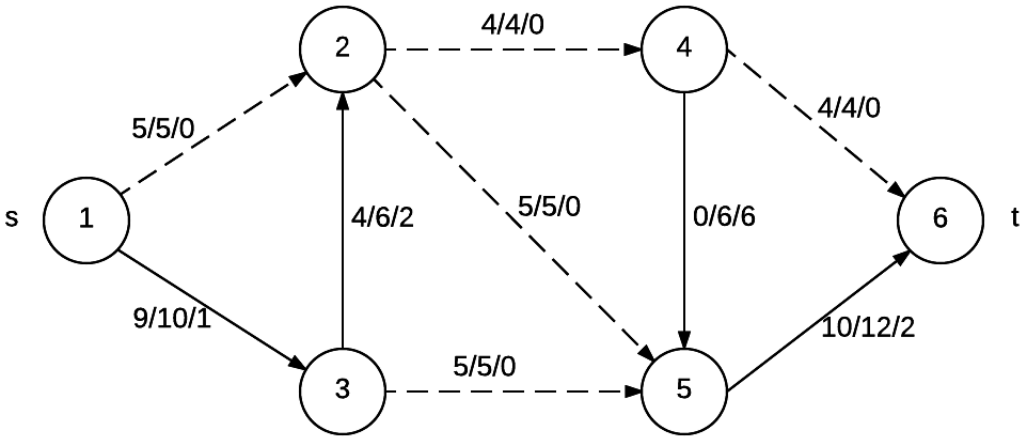

Abb. 104

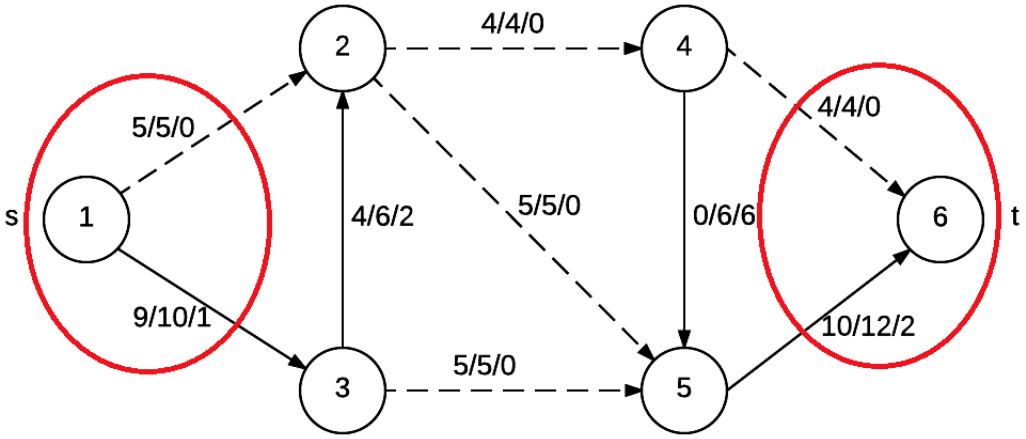

Abb. 105

87.4 Problem: Ungünstige Pfadwahl

Die bisher betrachtete Version des Algorithmus ist nicht immer optimal.

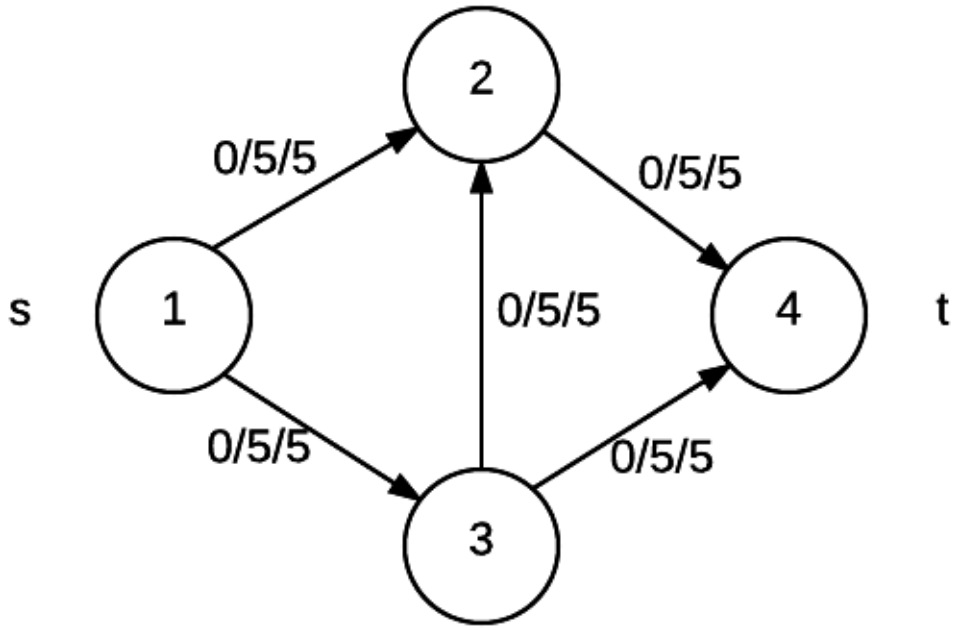

Abb. 106

Wählen der Pfad $1 \to 3 \to 2 \to 4$ ausgewählt, besitzt dieser Pfad einen nutzbaren Fluss von 5.

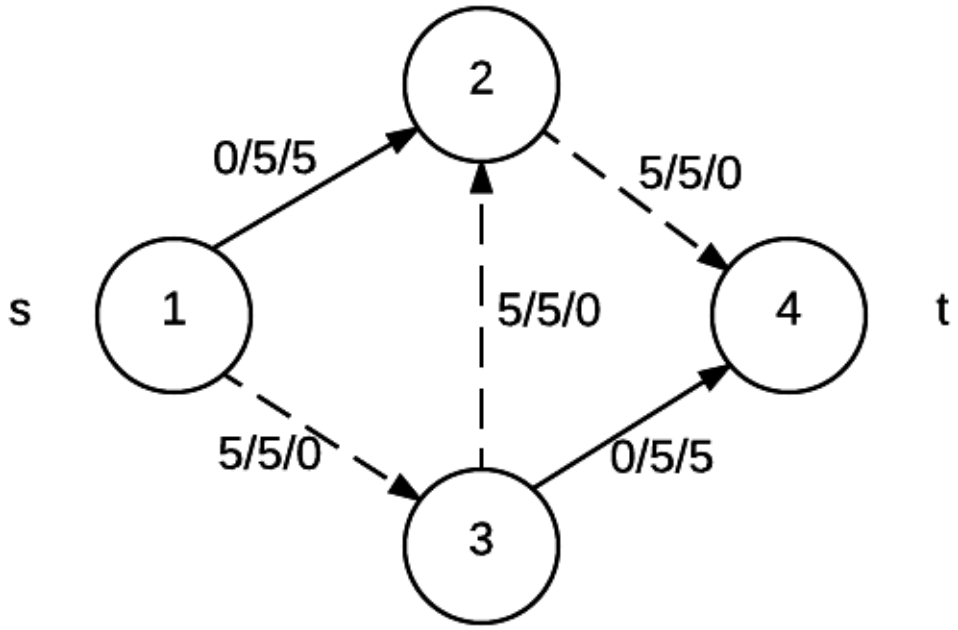

Abb. 107

Nun wird der Fluss aktualisiert. Daraus folgt, dass keine weitere Pfadwahl mehr möglich ist. Dabei wäre die optimale Lösung über die Pfade $1 \to 2 \to 4$ und $1 \to 3 \to 4$.

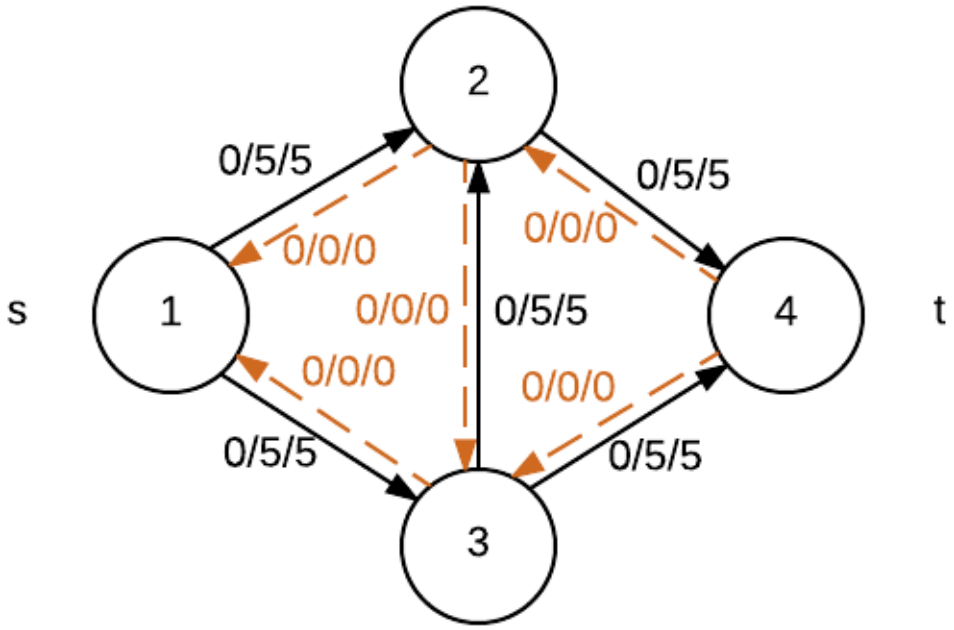

Abb. 108

Das Problem ist, dass der Fluss nicht zurückgenommen werden kann. Die Lösung dazu ist, dass man entgegengesetzte Flussrichtung durch Rückkanten erlaubt. Auch hier wird wieder der ungünstige Pfad $1 \to 3 \to 2 \to 4$ mit einem nutzbaren Fluss von 5 im ersten Schritt ausgewählt.

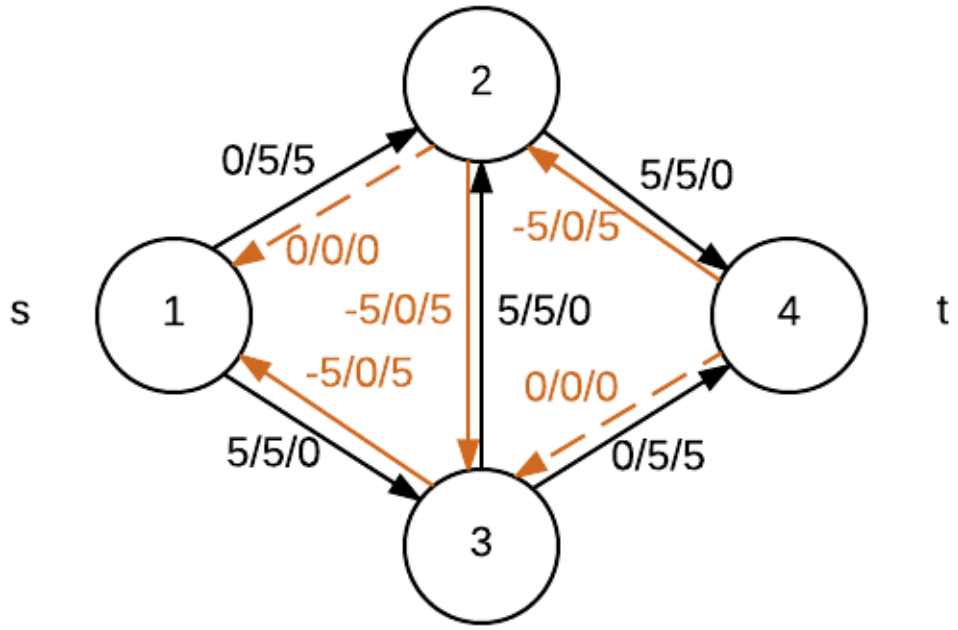

Abb. 109

Anschließend wird der Fluss aktualisiert. Dabei wird der Pfad $1 \to 2 \to 3 \to 4$ mit dem nutzbaren Fluss von 5 ausgewählt.

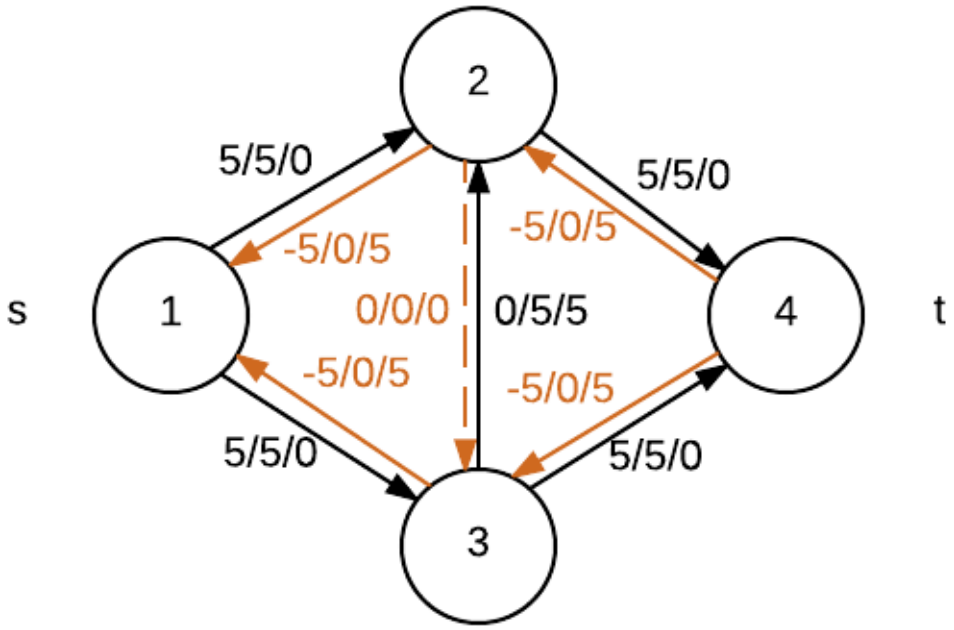

Abb. 110

Beim erneuten aktualisieren des Flusses, stellt sich heraus, dass keine weiteren Pfade möglich sind. Damit ist die Berechnung, bei einem maximalen Fluss von 10, beendet.

87.5 Analyse

87.5.1 Terminierungstheorem

Sind alle Kapazitäten in G nicht-negativ und rational, dann terminiert der Algorithmus von Ford-Fulkerson nach endlicher Zeit.

87.5.2 Laufzeittheorem

Ist X der Wert eines maximales Flusses in G=(V,E) und sind alle Kapazitäten in G nicht-negativ und ganzzahlig, so hat der Algorithmus von Ford-Fulkerson eine Laufzeit von O(|E|X).

87.5.3 Korrektheitstheorem

Sind alle Kapazitäten in G nicht-negativ und rational, dann berechnet der Algorithmus von Ford-Fulkerson den Wert eines maximalen Flusses.

87.5.4 Anmerkung

Die Wahl des Pfades beeinflusst die Anzahl benötigter Iteratoren. Bei dem Verfahren von Edmons und Karp muss die Anzahl der Pfade die in einem Graphen G = (V,E) bis zum Finden des maximalen Flusses verfolgt werden, kleiner sein als |V||E|, wenn jeweils der kürzeste Pfad von Quelle q zu Ziel z gewählt wird. Daher kann die Auswahl des nächsten kürzesten Pfades basierend auf einer Variante der Breitensuche erfolgen. Dadurch wird die Laufzeit auf $O(\mid V \mid\mid E \mid^2)$ verbessert.

88 Spannbäume

Auf dieser Seite werden Spannbäume und in diesem Zusammenhang der Algorithmus von Prim[1] behandelt.

88.1 Beispiel Kommunikationsnetz

Zwischen n Knotenpunkten $v_1 ... v_n$ soll ein möglichst billiges Kommunikationsnetz geschaltet werden, so dass jeder Knotenpunkt mit jedem anderen verbunden ist, ggf. auf einem Umweg über andere Knotenpunkte. Bekannt sind die Kosten c_{ij} für die direkte Verbindung zwischen v_i und $v_j 1 \leq i, j \leq n$. Alle Kosten c_{ij} seien verschieden und größer Null. Die Modellierung geschieht somit als gewichteter, ungerichteter und vollständiger Graph mit einer Gewichtungsfunktion c.

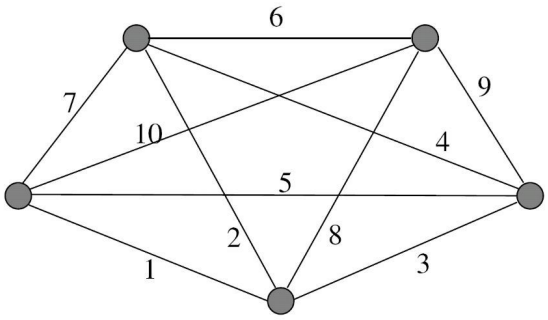

Abb. 111 Kommunikationsnetz

$G = (V, E)$

$V = \{v_1, ..., v_5\}$

$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_1, v_5), (v_2, v_3), (v_2, v_4), (v_2, v_5), (v_3, v_4), (v_3, v_5), (v_4, v_5)\}$

$c((v_1, v_2)) = 6, c((v_1, v_3)) = 7$ etc; abgekürzt $c_{1,2} = 6, c_{1,3} = 7$ etc

88.2 Problemstellung: Finde minimal aufspannenden Baum

Einige Definitionen für ungerichtete Graphen:

Ein Graph G=(V,E) heißt zusammenhängend, wenn für alle v,w∈V ein Pfad von v nach w in G existiert.

1 https://de.wikipedia.org/wiki/Algorithmus_von_Prim

Ein Graph G=(V,E) enthält einen Zyklus, wenn es unterschiedliche Knoten $v_1, ..., v_n \in V$ gibt, so dass $\{v_1, v_2\}, ..., \{v_{n-1}, v_n\}, \{v_n, v_1\} \in E$. Ein Graph G=(V,E) heißt Baum, wenn er zusammenhängend ist und keinen Zyklus enthält.

Ein Graph G'=(V',E') heißt Teilgraph von G=(V,E), wenn $V' \subseteq V$ und $E' \subseteq E \cap (V'xV')$.

Ein Graph G'=(V',E') heißt induzierter Teilgraph von G=(V,E) bzgl. $V' \subseteq V$, wenn $E' = E \cap (V'xV')$

Ein Graph G‘=(V‘,E‘) heißt Spannbaum von G=(V,E), wenn V'=V und G' ein Teilgraph von G und ein Baum ist.

Das Gewicht einen Graphen G=(V,E) ist $C(G) = \sum_{(i,j)\in E} c_{i,j}$.

Ein Graph G'=(V',E') ist ein minimaler Spannbaum von G=(V,E), wenn G' ein Spannbaum von G ist und G' unter allen Spannbäumen von G das minimalste Gewicht hat.

89 Algorithmus von Prim

Der Algorithmus wird schrittweise verfeinert und der Aufbau eines aufgespannten Baumes erfolgt durch das Hinzufügen von Kanten. Das Greedy Muster, also jeweils die Wahl der kostengünstigsten Kante als Erweiterung, wird hier benutzt.

89.1 Aufspannender minimaler Baum

```
//Teilbaum B besteht anfangs aus einem beliebigen Knoten
while [ B noch nicht GV aufspannt ]
do [ suche kostengünstige von B ausgehende Kante ];
     [ füge diese Kante zu B hinzu ];
od
```

Eine Verfeinerung der Suche nach der kostengünstigsten Kante ist notwendig!

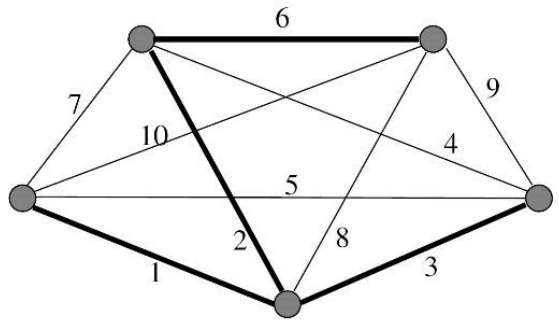

Abb. 112 Prim Algorithm

89.2 Suche nach kostengünstigster Kante

Die intuitive Vorgehensweise erfordert jeweils |W|(|V|-|W|) Vergleiche für ein gegebenes W. Das ganze |V| mal, also eine Gesamtlaufzeit von $O(n^3)$. Man kann die Suche auf die Teilmengen $F \subseteq E$ beschränken, so dass F immer die günstigste aus b ausgehende Kante enthält, wesentlich weniger Kanten hat als |W|(|V|-|W|) und im Verlauf des Algorithmus einfach anpassbar ist.

89.3 Wahl von F

Alternativen:

a) F enthält für jeden Knoten v in B die günstigste von v aus B herausführende Kante

b) F enthält für jeden Knoten v außerhalb B die günstigste von v in B hineinführende Kante

Bewertung:

a) Mehrere Kanten können zum gleichen Knoten herausführen – redundant und änderungsaufwändig (bei Wahl dieses Knotens darf er nicht mehr verwendet werden und alle Verbindungen zu diesem Knoten müssen gelöscht werden)

b) Daher: Wahl von b)

89.4 Erste Verfeinerung

```
// Teilbaum B
               [ B:= ({ beliebiger Knoten v }, {}) ]

               // Menge der Kandidatenkanten F
               [ F:= alle nach v führenden Kanten ]

               // alle Knoten betrachten
               for i := 1 to |V|-1
               do         [ suche günstigste Kante f=(u,w) in F ];
                       [ Füge f zu B hinzu (natürlich auch w) ];
                            [ Aktualisiere F ];
               od
```

F muss nach jedem Durchlauf angepasst werden. Wenn f aus F entfernt wird erkennt man, dass der Teilgraph B tatsächlich ein Baum ist. Nun haben wir den neu verbundenen Knoten w. Jeder noch nicht verbundene Knoten x hat nun eine günstigste Verbindung entweder wie zuvor, oder aber mit dem neu hinzugefügten Knoten w!

89.5 Zweite Verfeinerung

```
// Teilbaum B
               [ B:= ({ beliebiger Knoten v },{}) ]
               // Menge der Kandidatenkanten F
               [ F:= alle nach v führenden Kanten ]

               for i := 1 to |V|-1
               do
                       // Sei v∈B, w∉B
                       [ suche günstigste Kante f=(v,w) in F ];
                       [ Füge f zu B hinzu ];
                       // Aktualisiere F
                            [ Entferne f aus F ];
                       // x in B, w neuerdings in B, y noch nicht in B
                       for [ alle Kanten e=(x,y)∈F]
                       do
                               if [ c((w,y))<c(e)] then [ Ersetze e durch (w,y) ] fi
```

```
                od
        od
```

89.6 Kommunikationsnetz

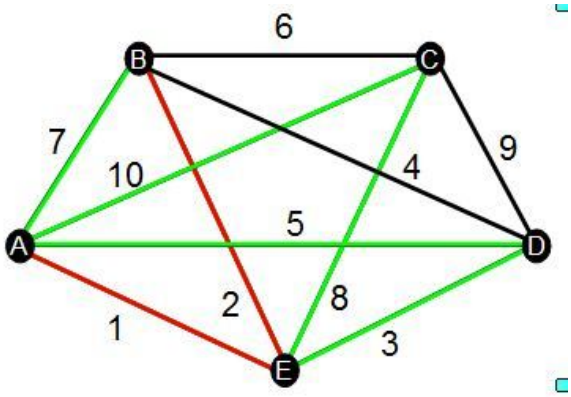

Abb. 113 Kommunikationsnetz2

i:

$B_0 = (\{A\}, \{\})$

$F_0 = \{(A,E),(A,D),(A,C),(A,B)\}$

$f_1 = (A,E)$ ist am günstigsten

$B_1 = (\{A,E\}, \{(A,E)\})$

$F_1 = \{(A,D),(A,C),(A,B)\}$

$(A,D) : F_1^1 = \{(D,E),(A,C),(A,B)\}$

$(A,C) : F_1^2 = \{(D,E),(C,E),(A,B)\}$

$(A,B) : F_1^3 = \{(D,E),(C,E),(B,E)\}$

$f_2 = (E,B)$

....

89.7 Analyse

89.7.1 Terminierungstheorem

Der Algorithmus von Prim terminiert nach endlicher Zeit.

Beweis

Einfache Schleifenanalyse

89.7.2 Laufzeittheorem

Wird für die Implementierung von F ein Fibonacci-Heap benutzt, so hat der Algorithmus von Prim eine Laufzeit von O(|E| + |V| log |V|).

89.7.3 Korrektheitstheorem

Ist G ein verbundener ungerichteter gewichteter Graph, so berechnet der Algorithmus von Prim einen minimalen Spannbaum von G.

Beweis

Wir betrachten eine einfache Version des Algorithmus.

```
while [ B noch nicht GV aufspannt ]
do [ suche kostengünstige von B ausgehende Kante ];
     [ füge diese Kante zu B hinzu ];
od
```

Wir beobachten, dass B am Ende ein Spannbaum ist. Jetzt ist noch zu zeigen, dass B am Ende ein minimaler Spannbaum ist.

Sei B‘ ein minimaler Spannbaum von G und B≠B‘. Betrachte den Zeitpunkt in der Hauptschleife, an dem sich die Konstruktion von B von B‘ unterscheidet. Sei e die Kante, die dann zu B hinzugefügt wird. Sei V_1 die Menge der Knoten, die schon in B sind und $V_2 = V \setminus V_1$ Da B‘ ein minimaler Spannbaum ist, gibt es eine Kante e’, die V_1 mit V_2 verbindet. Da im Algorithmus stets eine günstigste Kante gewählt wird, muss gelten g(e)≤g(e‘). Tauschen wir in B‘ die Kante e‘ durch e erhalten wir also einen minimalen Spannbaum, der nicht mehr kostet als B‘, es folgt g(e)=g(e‘). Induktiv folgt damit die Korrektheit.

89.8 Grundlagen

90 Grundlagen der Optimierung

Auf dieser Seite gibt es eine Einführung in das Thema Optimierung[1].

Die (Mathematische) Optimierung beschreibt eine Familie von Lösungsstrategien zur Maximierung/Minimierung einer Zielfunktion unter Nebenbedingungen. Viele der bisher untersuchten Probleme können als Optimierungsproblem modelliert werden. Zum Beispiel das Kürzeste Wege Problem: Minimiere die Länge eines Pfades unter der Nebenbedingung, dass der Pfad zwei gegebene Knoten verbindet. Oder das Rucksackproblem: Maximiere den Gesamtnutzen der Gegenstände unter der Nebenbedingung, dass die Kapazität des Rucksacks eingehalten wird. Oder das Flussprobleme: Maximiere den Fluss unter der Nebenbedingung, dass Kantenkapazitäten eingehalten werden.

Algorithmen wie Djikstra und Ford-Fulkerson sind domänenspezifische Algorithmen zur Lösung ihrer jeweiligen Optimierungsprobleme. Mathematische Optimierungsverfahren sind allgemeine Verfahren, die auf eine Vielzahl von Problemen anwendbar sind, dabei aber eventuell nicht immer so effizient wie speziellere Algorithmen sind.

Optimierung ist eine weites Feld, wir werden uns in dieser Vorlesung auf einen kleinen Ausschnitt konzentrieren:

- Grundlagen der Optimierung
- Kombinatorische Optimierung
- Lineare Optimierung
- Das Simplex-Verfahren

90.1 Begriffe

Ein allgemeines (reelles) Optimierungsproblem ist gegeben durch P: Minimiere f(x)unter der Nebenbedingung $x \in X \; mit X \subseteq \mathbb{R}^n \; und \; f : X \to \mathbb{R}$. f ist dabei die Zielfunktion. $x \in \mathbb{R}^n$ heißt zulässig für P, falls $x \in X$. X ist die zulässige Menge und $x' \in X$ heißt globales Minimum von P, falls $\forall x \in X : f(x') \leq f(x)$. Äquivalent gilt P: Minimiere f(x) unter der Nebenbedingung $x \in X$ und P': Maximiere -f(x) unter der Nebenbedingung $x \in X$.

90.2 Beispiel Gewinnmaximierung

Eine Firma produziert zwei verschiedene Waren. Ware x1 erbringt einen Gewinn von einem Euro. Ware x2 erbringt einen Gewinn von 6 Euro.

1 `https://de.wikipedia.org/wiki/Optimierung`

Frage: Welches Verhältnis von x1 und x2 führt zum größten Gewinn?

Nebenbedingungen:

Die Firma kann täglich maximal 200 Einheiten der Ware x1 produzieren und maximal 300 Einheiten der Ware x2 .

Insgesamt kann die Firma maximal 400 Einheiten pro Tag produzieren.

Zuerst wird nun die Zielfunktion formuliert: Maximiere Gewinn (1 Euro pro x_1, 6 Euro pro x_2) : $max\ x_1 + 6 \cdot x_2$.

Anschließend werden die Nebenbedingungen formuliert.

Maximal 200 Exemplare von x1 $x_1 \leq 200$

Maximal 300 Exemplare von x2 $x_2 \leq 300$

Insgesamt maximal 400 Exemplare $x_1 + x_2 \leq 400$

Es müssen Waren produziert werden $x_1, x_2 \geq 0$

Der Punkt $(0,0) \in \mathbb{R}^2 (x_1 = 0, x_2 = 0)$ ist zulässig mit Funktionswert 0 $max x_1 + 6 \cdot x_2$

Der Punkt $(100,200) \in \mathbb{R}^2$ ist zulässig mit Funktionswert 1400 $x_2 \leq 200$

Der Punkt $(100,300) \in \mathbb{R}^2$ ist zulässig mit Funktionswert 1900 und globales Maximum $x_1 + x_2 \leq 400$

Der Punkt $(200,300) \in \mathbb{R}^2$ ist unzulässig $x_1, x_2 \geq 0$

Dieses Beispiel ist ein lineares Optimierungsproblem.

90.3 Beispiel Kürzester Weg

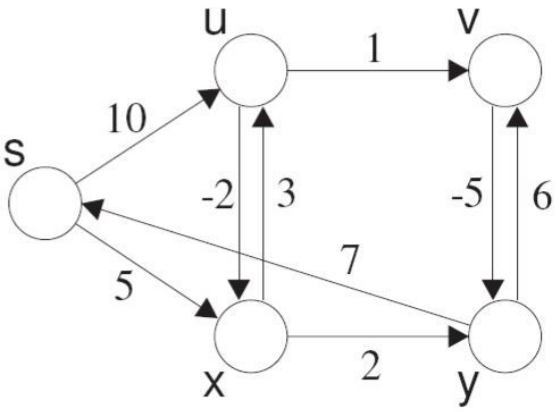

Abb. 114 Beispielgraph

Es soll die Distanz von s nach y bestimmt werden. Dafür sollen folgende Variablen und Bezeichner betrachtet werden.

- $e_{a,b} \in \{0,1\}$: die Kante von a nach b ist Teil des kürzesten Pfades von s nach y für alle Kanten (a,b)
- $w_{a,b}$: Das Gewicht der Kante von a nach b, zum Beispiel $w_{s,u} = 10$
- Die Zielfunktion ist $min\ e_{s,u}w_{s,u} + e_{s,x}w_{s,x} + ... + e_{v,y}w_{v,y}$

Es gelten folgende Nebenbedingungen:

1. Die Gewichte müssen wie im Graph sein $w_{s,u} = 10, w_{s,u} = 5, ...$
2. Alle Kanten (a,b) mit $e_{a,b} = 1$ müssen einen Pfad von s nach y bilden:
   a) Es gibt genau eine Kante mit Startpunkt s: $e_{s,u} + e_{s,x} = 1$
   b) Es gibt genau eine Kante mit Zielpunkt y: $e_{v,y} + e_{x,y} = 1$
   c) Für jeden anderen Knoten gilt, falls eine Kante in diesen Knoten reinführt, muss er auch wieder eine rausführen, zum Beispiel für $x : e_{s,x} + e_{u,x} = e_{x,u} + e_{x,y}$

Beachte durch die Minimierung werden Kreise auf dem Pfad automatisch verhindert.

Vollständiges Optimierungsproblem für ein kleines Beispiel von u nach x:

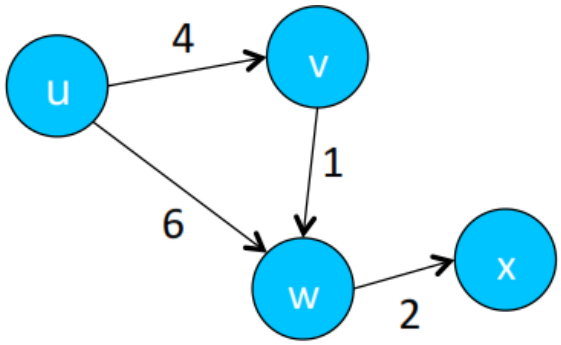

Abb. 115 Kürzester Weg

$min\ e_{u,v}w_{u,v} + e_{u,w}w_{u,w} + e_{v,w}w_{v,w} + e_{w,x}w_{w,x}$

$w_{u,v} = 4$

$w_{u,w} = 6$

$w_{v,w} = 1$

$w_{w,x} = 2$

$e_{u,v} + e_{u,w} = 1$

$e_{w,x} = 1$

$e_{u,v} = e_{v,w}$

$e_{u,w} + e_{v,w} = e_{w,x}$

$e_{u,v}, e_{u,w}, e_{v,w}, e_{w,x} \in \{0,1\}$

Das Problem der Kürzesten-Wegfindung ist ein ganzzahliges Optimierungsproblem, allgemeiner ein kombinatorisches Optimierungsproblem.

90.4 Problemklassen

Optimierungsprobleme sind unterschiedlich schwer lösbar $max, min\ f(x_1, ...x_n)$

- lineare Probleme: f,h sind linear, z.B. f/x,y)=3x+4y. Diese sind einfach zu lösen $h_1(x_1, ..., x_n) \leq b_1$
- Quadratische Probleme: z.B. $f(x, y) = x^2 + xy$ sind auch noch einfach zu lösen. $h_{m_1}(x_1, ..., x_n) \leq b_{m_1}$
- Konvexe Probleme: z.B. min $f(x, y) = log(x) + log(y)$ sind schon schwerer zu lösen. $i_1(x_1, ..., x_n) < c_1$
- Nicht-konvexe Probleme: z.B. $f(x, y) = x\ \ sin(x)$ sind ziemlich schwer zu lösen. $i_{m-2}(x_1, ..., x_n) < c_{m-2}$
- Ganzzahlige Probleme: $x_1, ..., x_n \in \mathbb{Z}$ sind überraschenderweise schwerer zu lösen als reelle Probleme. Etwa allgemeiner handelt es sich hier um kombinatorische Probleme (diskrete Elemente, nicht notwendigerweise Zahlen)
- Weitere Parameter
  - Restringierte Probleme: zulässige Menge ist beschränkt
  - Unrestringierte Probleme: zulässige Menge ist unbeschränkt

Hier werden wir uns aber nur mit linearer Optimierung befassen.

91 Kombinatorische Optimierung

Auf dieser Seite wird die kombinatorische Optimierung[1] behandelt. Kombinatorische Optimierungsprobleme sind im allgemeinen sehr schwer. Beispielsweise das Travelling Salesman Problem[2], oder die Knotenüberdeckung(Vertex Cover). Allgemeine Algorithmen sind meist sehr ineffizient. Deswegen benutzt man meistens domänenspezifische Algorithmen, so wie bei unseren bisherigen Beispielen. Wir schauen und jetzt noch ein weiteres Beispiel an.

1 https://de.wikipedia.org/wiki/Kombinatorische_Optimierung
2 https://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden

92 Das Rucksackproblem

Hierbei handelt es sich um ein einfaches kombinatorisches Optimierungsproblem. Gegeben ist ein Rucksack mit der maximalen Kapazität C und n Gegenstände mit jeweils dem Gewicht g_i und dem Wert w_i. Gesucht wird die Auswahl der Gegenstände, so dass das Gesamtgewicht die Kapazität nicht überschreitet $\sum_{i \in I} g_i \leq C$ und die Summe der Werte maximal ist $\sum_{i \in I} w_i$ ist maximal. Es gibt dafür 2^n Möglichkeiten.

92.1 Generieren

zunächst werden die Objekte generiert:

```
public class Ding {
   public int gewicht, nutzen;
   static private java.util.Random ra = new java.util.Random();

   // generieren
   Ding() {
      gewicht = ra.nextInt(MAX_GEWICHT) + 1;
      nutzen = ra.nextInt(MAX_NUTZEN) + 1;
   }
}
```

Es werden statische Variablen für die Problembeschreibung genutzt. Das Gewicht und der Nutzen der Objekte werden in einem eindimensionalem Array der Größe anzahlObjekte erstellt.

```
public class Rucksack {
   static int anzahlObjekte=10;
   static ding [ ] auswahlObjekte = null;
   …
```

Die gewichte und Nutzwerte werden zufällig zwischen 1 und dem jeweiligem Maximalwert generiert.

```
static final int MAX_GEWICHT = 10;
static final int MAX_NUTZEN = 10;
```

Generierung der Auswahlobjekte:

```
static Ding [ ] erzeugeObjekte() {
   Ding[] r = new Ding[anzahlObjekte];
   for (int i = 0; i < anzahlObjekte; i++ ) {
      r[i] = new ding();
   }
```

```
   return r;
}
 …
```

Die Kapazität der Rucksäcke ist eine weitere statische Variable.

```
 static int kapazitaet;
```

Eine willkürlich gewählte Initialisierung der Main Methode ist:

```
kapazitaet = (int) (anzahlObjekte * MAX_GEWICHT / 4);
```

Dadurch passen im Schnitt nur die Hälfte der Gegenstände in den Rucksack.

Nun wird ein Rucksack als Auswahl der vorgegebene Dinge Implementiert, hierbei handelt es sich um die einzige nicht statische Variable.

```
 boolean[] auswahl = null;
```

Der Konstruktor zum Erzeugen einen leeren Rucksacks lautet:

```
Rucksack () {
   auswahl = new boolean [anzahlObjekte];
   for (int i = 0; i < anzahlObjekte; i++ ){
      auswahl[i] = false;
   }
}
```

Es gibt einen Copy-Konstruktor zum Erzeugen einer Kopie eines existierenden Rucksacks. Die toString() Methode wird zur Ausgabe eines Rucksacks benutzt. Eine Methode zum Berechnen des Gesamtgewichts und des Gesamtnutzens lautet zum Beispiel:

```
int gewicht () {
   int g = 0;
   for (int i=0; i < auswahl.length; i++ )
      if (auswahl[i] == true)
         g = g + auswahlObjekte[i].gewicht;
   return g;
}
```

Index i	0	1	2	3	4	5
Gewicht	7	5	8	3	3	2
Nutzen	1	5	2	2	1	9

Rucksack 1 beinhaltet die Gegenstände 0,2 und 3, hatte ein Gesamtgewicht von 18 und einen Gesamtnutzen von 5.

Rucksack 1	T	F	T	T	F	F

Rucksack 2 beinhaltet die Gegenstände 1,2 und 5, hatte ein Gesamtgewicht von 15 und einen Gesamtnutzen von 16.

Rucksack 2	F	T	T	F	F	T

93 Das Rucksackproblem als Greedy Algorithmus

Nun wird das Rucksackproblem mit dem Greedy Algorithmus gelöst. Wir erinnern uns, das Greedy-Grundprinzip ist es in jedem Berechnungsschritt die jeweils aktuell geeignetste Zwischenlösung zu verwenden. Angewandt auf unser Rucksackproblem bedeutet das, lege von den noch nicht im Rucksack befindlichen Gegenständen jeweils den „besten“ hinzu. Doch was ist der beste Gegenstand? Der nützlichste? Der leichteste? Der mit dem besten Verhältnis aus Nutzen und Gewicht?

93.1 Algorithmus nach Nutzen

```
static Rucksack packeGierigNachNutzen() {
        Rucksack r = new Rucksack();
        while (true) {
                int pos=-1; int besterNutzen = 0;
                for (int i=0; i<auswahlObjekte.length; i++)
                        if (r.auswahl[i] == false &&
                                auswahlObjekte[i].nutzen > besterNutzen &&
                                r.gewicht() + auswahlObjekte[i].gewicht <=
                                        kapazitaet) {
                           besterNutzen = auswahlObjekte[i].nutzen;
                           pos = i;
                        }
                if (pos == -1) break;
                else r.auswahl[pos] = true;
        }
        return r;
}
```

93.2 Algorithmus nach Gewicht

```
static Rucksack packeGierigNachGewicht() {
        Rucksack r = new Rucksack();
        while (true) {
                int pos=-1; int bestesGewicht = MAX_GEWICHT+1;
                for (int i=0; i<auswahlObjekte.length; i++)
                        if (r.auswahl[i] == false &&
                                auswahlObjekte[i].gewicht < bestesGewicht &&
                                r.gewicht() + auswahlObjekte[i].gewicht <=
                                        kapazitaet) {
                           bestesGewicht = auswahlObjekte[i].gewicht;
                           pos = i;
                        }
                if (pos == -1) break;
                else r.auswahl[pos] = true;
```

```
        }
        return r;
}
```

93.3 Aufruf in main()

```
public static void main (String args[]) {
        if (args.length == 1)
                anzahlObjekte = Integer.parseInt(args[0]);
        kapazitaet = (int) (anzahlObjekte * MAX_GEWICHT / 4);
        auswahlObjekte = erzeugeObjekte();

        Rucksack r1 = packeGierigNachGewicht();
        System.out.println(,,Greedy Gewicht: ,, + r1);

        Rucksack r2 = packeGierigNachNutzen();
        System.out.println(,,Greedy Nutzen: ,, + r2);
        …
```

93.4 Analyse

Der Vorteil ist der relativ geringe Berechnungsaufwand durch die quadratische Komplexität $O(n^2)$. Das Problem ist aber, dass nicht die optimale Lösung gefunden wird.

94 Rucksackproblem als Backtracking

Nun wird das Rucksackproblem mit Backtracking gelöst. Das Grundprinzip ist es, die optimale Lösung durch systematisches Absuchen des gesamten Lösungsraums zu finden. Angewandt auf unser Rucksackproblem bedeutet das, es gibt 2^n verschiedene Möglichkeiten, wir generieren und testen alle möglichen Rucksäcke und wir wenden Rekursion an.

94.1 Rekursionseinstieg

```
static Rucksack packeOptimalmitBacktracking() {
   return rucksackRekursiv(0, new Rucksack());
}
```

- Erster Parameter: Level i – Entscheidung, ob Objekt i in den Rucksack kommt
- Durchlaufen des Auswahl-Arrays von links nach rechts
- Aufrufgraph: Aufspannen eines binären Baumes durch ja/nein-Entscheidungen

94.2 Rekursion

```
static rucksackRekursiv(int i, Rucksack r) {
                if (i==auswahlObjekte.length) return r;
                // Objekt i nicht nehmen und rekurrieren
                Rucksack r1 = new Rucksack(r);
                r1 = rucksackRekursiv(i+1, r1);
                // Objekt i - falls moeglich - nehmen und rekurrieren
                if (r.gewicht()+auswahlObjekte[i].gewicht<=kapazitaet){
                        Rucksack r2 = new Rucksack(r);
                        r2.auswahl[i] = true;
                        r2 = rucksackRekursiv(i+1,r2);
                        // Den besseren Rucksack immer zurueckgeben
                        if (r2.nutzen() > r1.nutzen())
                                return r2;
                        }
                return r1;
        }
```

94.3 Analyse

Das Problem ist hier, dass es einen extrem hohen Berechnungsaufwand für die große Auswahl an Objekten gibt. Die Komplexität liegt bei $O(2^n)$. Der Vorteil ist, dass man garantiert die optimale Lösung finden, da im schlimmsten Fall jede Möglichkeit ausprobiert wird. Also wird in jedem Fall ein Optimum gefunden.

95 Rucksackproblem als dynamische Programmierung

Nun wird das Rucksackproblem mit dynamischer Programmierung gelöst. Wir erinnern uns, dass das Grundprinzip der dynamischen Programmierung die Wiederverwendung von bereits berechneten Teillösungen ist. Aber an dieser Stelle ist Vorsicht geboten mit den anderen Lösungen aus den vorherigen Seiten, wo Teillösungen Bottom up zusammengesetzt wurden. Hier basieren die Lösungen auf der Backtracking Variante. Teillösungen werden zwischengespeichert. Die Existenz von Teillösungen wird als Abbruchkriterium für die Rekursion verwendet.

95.1 Rekursionseinstieg

```
static Rucksack packeMitDynamischerProgrammierung()
   Rucksack [][] zwischenErgebnisse=
      new Rucksack[kapazitaet+1][anzahlObjekte];
   retunr rucksackRekursivDP (0, new Rucksack(), zwischenErgebnisse);
}
```

Ein Eintrag zwischenErgebnisse[g][i] bedeutet, dass wir schon ein mal dabei waren, das Objekt i in einen Rucksack mit dem Gewicht g zu legen. In diesem Fall können wir alle vor berechneten Entscheidungen für die Objekte i bis anzahlObjekte -1 wiederverwenden, da diese bereits optimal sind (Backtracking: äquivalenter Teilbaum).

95.2 Rekursion

```
static Rucksack rucksackRekursivDP (int i, Rucksack r, Rucksack [][]
 zwischenErgebnisse)
   if (I== auswahlObjekte.length) return r;
   int gewicht= r.gewicht();
   //Wiederverwendung von Teillösungen:
   if (zwischenErgebnisse [gewicht] [i] != null){
      for (int j=i; j< anzahlObjekte; j++)
         r.auswahl [j]=zwischenErgebnisse[gewicht][i].auswahl[j];
      return r;
   }
   Rucksack r1=new Rucksack (r);
   r1= rucksackRekursivDP (i+1, r1, zwischenErgbenisse);
   if (gewicht+auswahlObjekte [i][0] <= kapazitaet){
      Rucksack r2 = new Rucksack (r);
      r2.auswahl [i] = true;
      r2= rucksackRekursivDP (i+1,r2, zwischenErgebnisse);
      if (r2.nutzen() > r1.nutzen()) r1=r2;
   }
   //Merken von Teillösungen:
```

```
   zwischenErgebnisse[gewicht][i]=r1;
   return r1;
}
```

95.3 Analyse

Die Vorteile der dynamischen Programmierung sind, dass auf jeden Fall die optimale Lösung gefunden wird. In vielen Fällen hat sie auch einen geringeren Aufwand als Backtracking. Das Problem ist allerdings, dass die Anwendbarkeit und der Aufwand abhängig von der Größe und der Struktur des Suchraums sind. Die Komplexität beträgt O(z), wobei z die Anzahl der möglichen Zwischenergebnisse ist. Zum Beispiel: bei vielen unterschiedlichen Gewichtskombinationen kaum Ersparnis (Erhöhen von MAX_GEWICHT). Außerdem existieren polynomielle Approximationen!

95.4 Lineare Optimierung

96 Lineare Optimierung

Auf dieser Seite wird die lineare Optimierung[1] behandelt. Eine lineare Optimierungsaufgabe ist: Maximiere eine lineare Funktion in mehreren Variablen

$max\ c_1x_1 + ... + c_nx_n\ c_i, x_i \in \mathbb{R} \Leftrightarrow maxc^Tx.$

Die lineare Nebenbedingung sind gegeben als lineare Gleichungen:

$a_{11}x_1 + ...a_{1n}x_n = b_1$

...

$a_{m1}x_1 + ...a_{mn}x_n = b_m$

$\forall i = 1, ..., n : x_i \geq 0$

Dies entspricht dem Gleichungssystem:

$Ax = b, x \geq 0, a \in \mathbb{R}^{mxn}, b \in \mathbb{R}^m, x \in \mathbb{R}^n$

Doch ist das ausdrucksstark genug für unsere Probleme?

96.1 Umformung von Gleichungssystemen

Beliebige Systeme lassen sich in die Standardform übertragen.

- Minimieren ist wie maximieren: $min\ c^Tx = max - c^Tx$
- $\geq$ Bedingungen statt $\leq$ Bedingungen: $a_i^Tx \leq b_i \Leftrightarrow -a_i^Tx \geq -b$
- Gleichung zu Ungleichung: $a_i^Tx = b_i \Leftrightarrow a_i^Tx \geq ba_i^Tx \leq b_i$
- Ungleichung zu Gleichung (Schlupfvariablen einführen): $a_i^Tx \leq b_i \Leftrightarrow a_i^Tx + s = b, s \geq 0$
- x_i kann negativ sein: $x_i = s - t, s \geq 0, t \geq 0$

96.2 Beispiel Gewinnmaximierung

Eine Firma produziert zwei verschiedene Waren. Ware x_1 erbringt einen Gewinn von einem Euro. Ware x_2 erbringt einen Gewinn von 6 Euro. Die Frage hierzu lautet, welches Verhältnis von x_1 und x_2 führt zum größten Gewinn? Dazu gibt es zwei Nebenbedingungen:

- Die Firma kann täglich maximal 200 Einheiten der Ware x_1 produzieren und maximal 300 Einheiten der Ware x_2
- Insgesamt kann die Firma maximal 400 Einheiten pro Tag produzieren

1 `https://de.wikipedia.org/wiki/Lineare_Optimierung`

Die Firma beschließt eine weitere Ware zu produzieren.

- Die Ware x_3 bringt einen Gewinn von 13 Euro.
- Die maximale Tagesproduktion liegt weiterhin bei 400 Einheiten.
- Für die Produktion von Ware x_2 und Ware x_3 wird dieselbe Maschine verwendet, allerdings ist der Produktionsaufwand für x_3 dreimal höher. Insgesamt kann die Maschine 600 Arbeitsschritte leisten.

Formuliere, die Zielfunktion: $max\ x_1 + 6 \cdot x_2 + 13 \cdot x_3$.

Anschließend formuliere die Nebenbedingungen:

$x_1 \leq 200$

$x_2 \leq 300$

$x_1 + x_2 + x_3 \leq 400$

$x_2 + 3 \cdot x_3 \leq 600$

$x_1, x_2, x_3 \geq 0$

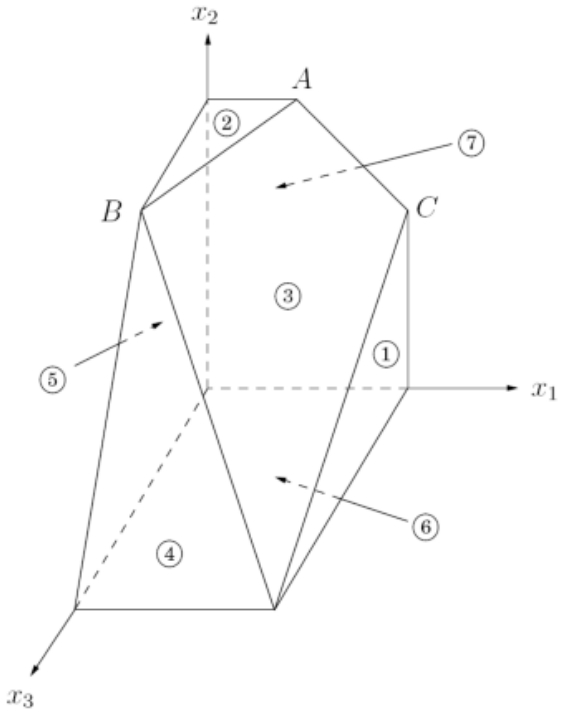

Abb. 116 Polyeder

Die Nebenbedingungen definieren ein drei-dimensionales Polyeder, in dem die optimale Lösung liegt.

Nun formen wir in die Normalform um:

$max\ x_1 + 6 \cdot x_2 + 13 \cdot x_3 \rightarrow max\ x_1 + 6 \cdot x_2 + 13 \cdot x_3$

$x_1 \leq 200 \rightarrow x_1 + s_1 = 200$

$x_2 \leq 300 \rightarrow x_2 + s_2 = 300$

$x_1 + x_2 + x_3 \leq 400 \rightarrow x_1 + x_2 + x_3 + s_3 = 400$

$x_2 + 3 \cdot x_3 \leq 600 \rightarrow x_2 + 3 \cdot x_3 + s_4 = 600$

$x_1, x_2, x_3 \geq 0 \rightarrow x_1, x_2, x_3, s_1.s_2.s_3, s_4 \geq 0$

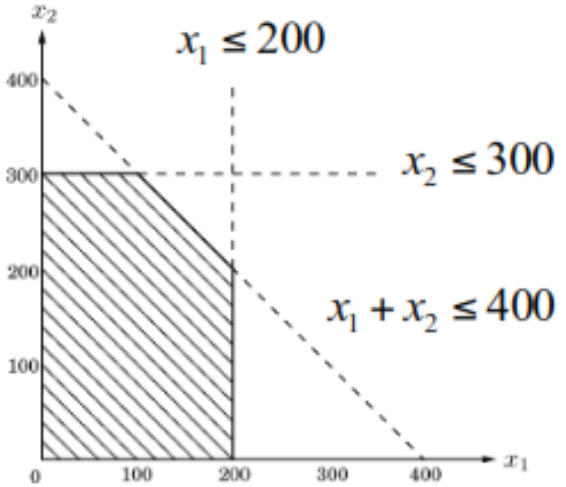

Abb. 117 Polyeder

Die Nebenbedingungen definieren nun ein Polyeder, in dem die optimale Lösung liegt.

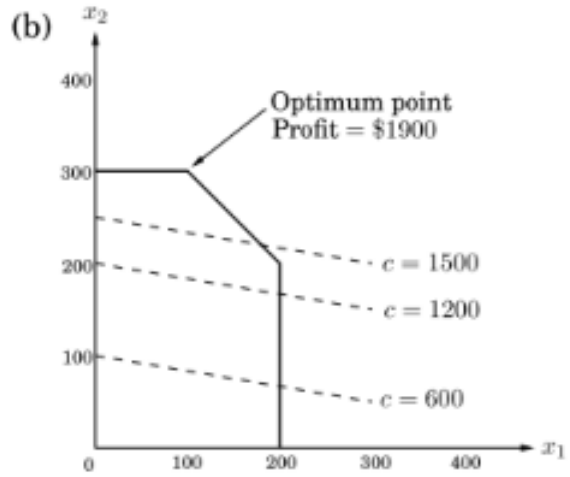

Abb. 118 Polyeder

Die Zielfunktion $c = x_1 + 6 \cdot x_2$ ist eine Gerade. Nun wird die Gerade verschoben, bis das Maximum erreicht ist.

Die Linearen Optimierungsprobleme der Form

$max\ c^T x$

$(1) Ax = b;\ x \geq 0$

(2)$Ax \leq b;\ x \geq 0$

besitzen genau dann eine endliche Optimallösung, wenn sie eine optimale Ecklösung (= „Ecke“ des zugehörigen Polyeders) besitzen. D.h. man muss zur Lösungsfindung nur die Ecken des Polyeders betrachten.

96.3 Simplex Verfahren

97 Simplex Verfahren

Auf dieser Seite wird das Simplex Verfahren[1] behandelt.

97.1 Idee

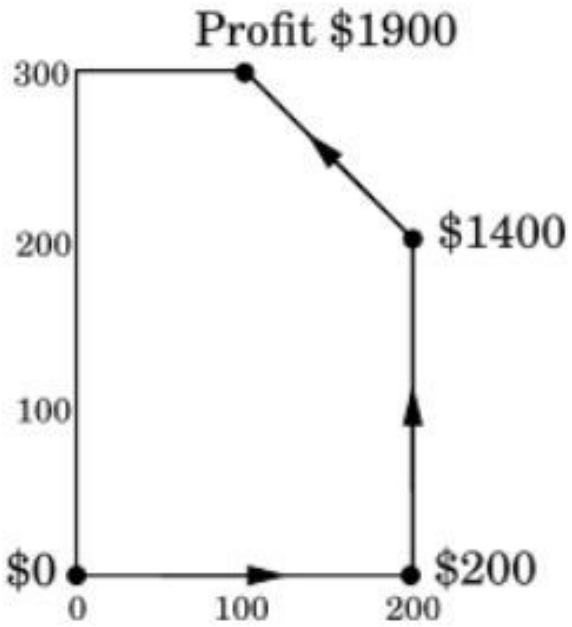

Abb. 119 Simplex Verfahren

Es wird in einer beliebigen Ecke des Polyeders begonnen. Dann wird verglichen, ob einer der Nachbarn eine bessere Lösung für die Optimierung bietet und anschließend wird dieser Knoten betrachtet. Am Ende erreichen wir eine Ecke, die keinen Nachbarn mit einer besseren Lösung hat. Die Lösung ist nun ein lokales Optimum. Bei der linearen Optimierung gilt, dass ein lokales Optimum automatisch ein globales Optimum ist, da der Polyeder eine konvexe Menge ist. Graphisch kann mit dieser Idee jedes lineare Optimierungsproblem gelöst werden. Dies wird aber sehr schnell unübersichtlich (und kann schlecht implementiert werden). Wir benötigen eine einfache Charakterisierung der "Ecken" des Polyeders. Diese erhalten wir durch Betrachtung der Basen der Matrix A.

1 `http://de.wikipedia.org/wiki/Simplex-Verfahren`

Das Simplex-Verfahren löst ein lineares Programm in endlich vielen Schritten oder stellt seine Unlösbarkeit oder Unbeschränktheit fest. Im Worstcase hat es exponentielle Laufzeit unabhängig von den gewählten Pivotregeln, in der Praxis ist es sehr effizient. Das Simplex-Verfahren berechnet auch die Lösung für das duale Problem zu einem linearen Programm.

97.2 Wiederholung algebraischer Grundlagen

Seien $v_1, ..., v_n \in \mathbb{R}^m$.

- Die Linearkombination von $v_1, ..., v_n$ mit den Koeffizienten $\alpha_1, ..., \alpha_n \in \mathbb{R}^m$ ist der Vektor $\alpha_1 v_1 + ... + \alpha_n v_n$.
- Die Vektoren $v_1, ..., v_n$ sind linear abhängig, wenn es ein $i \in \{1, ..., n\}$ gibt, so dass sich v_i als Linearkombination von $v_1, ..., v_{i-1}, v_{i+1}, ..., v_n$ darstellen lässt.
- Eine maximale Menge linear unabhängiger Vektoren heißt Basis des zugehörigen Raumes. Eine Basis des $\mathbb{R}^m$ besteht beispielsweise aus m linear unabhängigen Vektoren.
- Der Rang einer Matrix A ist die maximale Anzahl linear unabhängiger Spaltenvektoren.

97.3 Matrix lineares Optimierungsproblem

$max\ x_1 + 6 \cdot x_2$

$x_1 + s_1 = 200$

$x_2 + s_2 = 300$

$x_1 + x_2 + s_3 = 400$

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix},\ b = \begin{pmatrix} 200 \\ 300 \\ 400 \end{pmatrix}$$

Da wir für jede unserer ursprünglichen Ungleichungen eine Schlupfvariable eingeführt haben, gilt stets Rang(A)=m (=Anzahl der Gleichungen=Länge des Vektors b).

97.4 Basis und Basislösung

Auf dieser Seite werden die Basen und Basislösungen beim Simplex Verfahren behandelt. Gegeben ist ein lineares Gleichungssystem $ax = b, A \in \mathbb{R}^{mxn}, b \in \mathbb{R}^m, Rang(A) = m$.

Dann bilden m lineare unabhängige Spaltenvektoren aus A eine Basis von A. Diese wird mit A_B bezeichnet. B enthält die Indices der Basisvektoren. N enthält die Indices der Nichtbasisvektoren. Die Basislösung x_B *von* A_B ist gegeben durch: $A_B x_B = b$ dies gilt genau dann wenn: $x_B = A_B^{-1} b$. A_B ist eine zulässige Basis von A, wenn gilt $A_B^{-1} b \geq 0$. Wenn $(X_B X_N)$ *mit* $X_N = 0$ ist, dann ist es eine zulässige Basislösung von A.

97.4.1 Beispiel 1

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} 200 \\ 300 \\ 400 \end{pmatrix}$$

$B1 = \{3,4,5\}$ $N1 = \{1,2\}$

$$A_{B1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \; X_{B1} = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

$$A_{B1}X_{B1} = b \Rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} 200 \\ 300 \\ 400 \end{pmatrix} \Rightarrow s_1 = 200, s_2 = 300; s_3 = 400$$

Nicht-Basisvariablen werden stets auf 0 gesetzt. Die zulässige Basislösung von A mit Zielfunktionswert 0, die man durch einsetzen erhält ist dann (0,0,200,300,400).

97.4.2 Beispiel 2

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} 200 \\ 300 \\ 400 \end{pmatrix}$$

$B2 = \{1,4,5\}$ $N2 = \{2,3\}$

$$A_{B2} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \; X_{B2} = \begin{pmatrix} x_1 \\ s_2 \\ s_3 \end{pmatrix}$$

$$A_{B2}X_{B2} = b \Rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} 200 \\ 300 \\ 400 \end{pmatrix} \Rightarrow x_1 = 200, s_2 = 300; s_3 = 200$$

Nicht-Basisvariablen werden stets auf 0 gesetzt.

Die zulässige Basislösung von A, die man durch einsetzen erhält ist dann (200,0,0,300,200) mit dem Zielfunktionswert 200.

97.4.3 Basen von A

Hier gibt es eine Übersicht der Basen von A mit dessen zulässigen Lösungen.

A_B	x_B	x_N	x

$A_{B1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$(s_1, s_2, s_3) = (200, 300, 400)$	(x_1, x_2)	$(0, 0, 200, 300, 400)$
$A_{B2} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	$(x_1, s_2, s_3) = (200, 300, 200)$	(x_2, s_1)	$(200, 0, 0, 300, 200)$
$A_{B3} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$	$(x_1, x_2, s_2) = (200, 200, 100)$	(s_1, s_3)	$(200, 200, 000, 100, 0)$
$A_{B4} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}$	$(x_1, x_2, s_1) = (100, 300, 100)$	(s_2, s_3)	$(100, 300, 100, 0, 0)$
$A_{B5} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	$(x_2, s_1, s_3) = (300, 200, 100)$	(x_1, s_3)	$(0, 300, 200, 0, 100)$

$$b = \begin{pmatrix} 200 \\ 300 \\ 400 \end{pmatrix}$$

97.4.4 Basen von A- mit unzulässigen Lösung

Hier gibt es eine Übersicht der Basen von A mit unzulässigen Lösungen.

A_B	x_B	x_N	x
$A_{B6} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$(x_1, x_2, s_3) = (100, 300, -100)$	(s_1, s_2)	$(200, 300, 0, 0, -100)$
$A_{B7} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$	$(x_2, s_1, s_2) = (400, 200, -100)$	(x_1, s_3)	$(0, 400, 200, -100, 0)$
$A_{B8} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$	$(x_2, s_1, s_2) = (400, -200, 300)$	(x_1, x_3)	$(200, 200, 000, 100, 0)$
$A_{B4} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}$	$(x_1, x_2, s_1) = (100, 300, 100)$	(s_2, s_3)	$(100, 300, 100, 0, 0)$
$A_{B5} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	$(x_2, s_1, s_3) = (300, 200, 100)$	(x_1, x_3)	$(0, 400, -200, 300, 0)$

Diese Basen haben keine zulässige Lösungen, da x_B negative Werte enthält.

Die Teilmengen $\begin{pmatrix}1&1&0\\0&0&0\\1&0&1\end{pmatrix}\begin{pmatrix}0&0&0\\1&1&0\\1&0&1\end{pmatrix}$ von A sind keine Basen von A, da die Vektoren jeweils linear abhängig sind.

97.5 Charakterisierung von Polyederecken

Warum schauen wir uns Basen und Basislösungen an? Wir waren doch an Ecken des Polyeders interessiert...

Sei das System $Ax = b, x \geq 0$ gegeben, $Rang(A) = m < n$. Dann sind äquivalent:

1. x ist eine Ecke des zugehörigen Polyeders
2. x ist eine zulässige Basislösung von Ax=b

Wir wissen, dass die optimale Lösung in einem Eckpunkt liegen muss, falls sie existiert. D.h. wir müssen nur über die Basen von A optimieren (diese bestimmen ja die zulässigen Basislösungen von Ax=b). Dies erfolgt mit sogenannten Tableaus.

Das Simplex-Verfahren besteht aus einer Folge von Basen bzw. Tableus.

1. Zuerst wird die zulässige Basis A_B gefunden und daraus das Starttableau konstruiert.
2. Anschließend wir eine neue zulässige Basis $A_{B'}$ aus A_B konstruiert, so dass die zulässige Basislösung von $A_{B'}$ besser ist, als die von A_B. Das Tableau wird nun aktualisiert.
3. Wenn es keine bessere Basislösung mehr gibt, dann ist die letzte optimal.

Ein Tableau entspricht dem Gleichungssystem $\begin{pmatrix}c^T\\A\end{pmatrix}x = \begin{pmatrix}c^Tx\\b\end{pmatrix}$ mit $max\ c^Tx, Ax = b\ und\ x \geq 0$.

T_B ist ein Simplextableau zur Basis A_B

$$T_B = \begin{pmatrix}c_N^T - c_B^T A_B^{-1} A_N & -c_B^T A_B^{-1} b\\ A_B^{-1} A_N & A_B^{-1} b\end{pmatrix} \text{ mit } A = (A_B A_N), x = (x_B x_N), c^T = (c_N^T c_B^T)$$

97.5.1 Beispiel Gewinnmaximierung

Nun wird der Simplex Algorithmus anhand des Beispiels der Gewinnmaximierung Schritt für Schritt durchgegangen.

Zielfunktion:

$max\ x_1 + 6 \cdot x_2 + 13 \cdot x_3$.

Nebenbedingungen:

$x_1 \leq 200$

$x_2 \leq 300$

$x_1 + x_2 + x_3 \leq 400$

$x_2 + 3 \cdot x_3 \leq 600$

$x_1, x_2, x_3 \geq 0$

Das System lässt sich umschreiben zu:

$x_1 + 6 \cdot x_2 + 13 \cdot x_3 = z$

$x_1 + s_1 = 200$

$x_2 + s_2 = 300$

$x_1 + x_2 + x_3 + s_3 = 400$

$x_2 + 3 \cdot x_3 + s_4 = 600$

$x_1, x_2, x_3, s-1, s_2, s_3, s_4 \geq 0$

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 3 & 0 & 0 & 0 & 1 \end{pmatrix}, b = \begin{pmatrix} 200 \\ 300 \\ 400 \\ 600 \end{pmatrix}$$

Initialisierung

Gestartet wird mit der Basislösung, die durch die Schlupfvariable gegeben ist.

$$A_B = (s_1 \ s_2 \ s_3 \ s_4) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = A_B^{-1}$$

$$A_N = (x_1 \ x_2 \ x_3) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 3 \end{pmatrix} \ b = \begin{pmatrix} 200 \\ 300 \\ 400 \\ 600 \end{pmatrix}$$

$$c^T = \begin{pmatrix} 1 & 6 & 13 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} c_N^T & c_B^T \end{pmatrix}$$

$$A_B^{-1} A_N = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 3 \end{pmatrix} \ A_B^{-1} b = \begin{pmatrix} 200 \\ 300 \\ 400 \\ 600 \end{pmatrix}$$

Starttableau

	x_1	x_2	x_3	b
z	1	6	13	0
s_1	1	0	0	200
s_2	0	1	0	300
s_3	1	1	1	400
s_4	0	1	3	600

x_1, x_2, x_3 sind Nichtbasiselemente, Z ist die Zielfunktion und s_1, s_2, s_3, s_4 sind Basiselemente. Dabei sind die blau hinterlegten Felder das $c_N^T - c_N^T A_B^{-1} A_N$, die gelb hinterlegten Felder stellen den Teil von $A_B^{-1} A_N$ dar und die grünen Felder sind $A_B^{-1} b$. Das nicht markierte Feld ist dabei der negative Zielfunktionswert $-c_B^T A_B^{-1} b$.

Update eines Tableau

Für das Update eines Tableau wird eine neue zulässige Basis bestimmt, indem ein Basisvektor durch einen Nichtbasisvektor ausgetauscht wird. Die Menge der Nichtbasisvektoren, die getauscht werden können, ist über die positiven Koeffizienten c der Zielfunktion definiert als: $E = \{j | c_x x_j > 0\}$. Wenn $E =$ dann breche ab und gebe x zurück. Die Menge der Basisvektoren, die getauscht werden können, ist über ihre j-te Komponente bestimmt: $L_j = \{i | x_j^i > 0\}$. Wenn $L_j =$ für alle $j \in E$ dann ist das LP unbeschränkt, da die Zielfunktion $c^T x$ durch x_j unbeschränkt wächst.

Optimierungsphase

Berechne für eine zulässige Basis, das zugehörige Tableau. Nun wird E bestimmt. Wenn $E =$ dann wird abgebrochen und x zurückgegeben. Ansonsten wird $j \in E$ durch eine geeignete Pivotregel gewählt. Als nächstes wird L_j bestimmt. Wenn $L_j =$ dann wird zurückgegeben, dass LP unbeschränkt ist. Ansonsten wird $i \in L_j$ durch eine geeignete Pivotregel gewählt. Führe nun einen Basiswechsel durch und starte wieder oben.

Beispiel

	x_1	x_2	x_3	b
z	1	6	13	0
s_1	1	0	0	200
s_2	0	1	0	300
s_3	1	1	1	400
s_4	0	1	3	600

$E = \{j | c_x x_j > 0\} = \{1,2,3\}$ $\{x_1, x_2, x_3\}$

$L_1 = \{i | x_1^i > 0\} = \{1,3\}$ $\{s_1, s_3\}$

$L_2 = \{i | x_2^i > 0\} = \{2,3,4\}$ $\{s_2, s_3, s_4\}$

$L_3 = \{i | x_3^i > 0\} = \{3,4\}$ $\{s_3, s_4\}$

Heuristik für die Auswahl der Tauschvektoren

Als erstes werden die größten Koeffizienten in der Zielfunktion gewählt (Dantzig). Eine andere Möglichkeit ist das steepest-edge pricing, welches die Kombination aus Spalten- und

Zeilenvektor wählt, die den größten Zuwachs für die Zielfunktion bringt. Oder der kleinste Index wird gewählt. Die letzte Möglichkeit ist eine zufällige Auswahl.

Erste Iteration

Heuristik: Ersetze einen Basisvektor durch den Nichtbasisvektor, der den größten Zugewinn für die Zielfunktion bringt.

x_1

$0 \leq s_1 = 200 - x_1$

$0 \leq s_3 = 400 - x_1$

$x_1 = min(200, 400) = 200 \Rightarrow z = 200$

Hier wird die Zeile von s_1 *und* s_3 betrachtet und die Spalte von x_1. Der alte Wert ist 0. Der Koeffizient von x_1 in der Zielfunktion ist 1 und der Zugewinn durch x_1 ist 200.

x_2

$0 \leq s_2 = 300 - x_2$

$0 \leq s_3 = 400 - x_2$

$0 \leq s_4 = 600 - x_2$

$x_2 = min(300, 400, 600) = 300 \Rightarrow z = 1800$

Hier wird die Zeile von s_2, s_3 *und* s_4 betrachtet und die Spalte von x_2. Der alte Wert ist 0. Der Koeffizient von x_2 in der Zielfunktion ist 6 und der Zugewinn durch x_2 ist 1800.

x_3

$0 \leq s_3 = 400 - x_3$

$0 \leq s_4 = 600 - 3x_3$

$x_3 = min(400, 200) = 200 \Rightarrow z = 2600$

Hier wird die Zeile von s_3 *und* s_4 betrachtet und die Spalte von x_3. Der alte Wert ist 0. Der Koeffizient von x_3 in der Zielfunktion ist 13 und der Zugewinn durch x_3 ist 2600. Nun wird s_4 durch x_3 ersetzt.

Update des Tableaus

Der neue Wert von x_3 wird nun berechnet.

$s_4 = 600 - x_2 - 3x_3 \Leftrightarrow x_3 = 200 - \frac{x_2}{3} - \frac{s_4}{3}$.

Dieser Wert wird nun eingesetzt.

$z = x_1 + 6_{x_2} + 13 \cdot (200 - \frac{x_2}{3} - \frac{s_4}{3}) = x_1 + \frac{5}{3}x_2 - \frac{13}{3}s_4 + 2600$

$s_3 = 400 - x_1 - x_2 - (200 - \frac{x_2}{3} - \frac{s_4}{3}) = 200 - x_1 - \frac{2}{3}x_2 + \frac{s_4}{3}$

$x_3 = 200 - \frac{x_2}{3} - \frac{s_4}{3}$

Das neue Tableau sieht nun so aus:

	x_1	x_2	s_4	b
z	1	$\frac{5}{3}$	$-\frac{13}{3}$	-2600
s_1	1	0	0	200
s_2	0	1	0	300
s_3	1	$\frac{2}{3}$	$-\frac{1}{3}$	200
x_3	0	$\frac{1}{3}$	$\frac{1}{3}$	200

Was haben wir nun gemacht? Von der Basis $B = (s_1, s_2, s_3, s_4)$ haben wir zu der Basis $B' = (s_1, s_2, s_3, x_3)$ gewechselt und zu der neuen Basis haben wir das entsprechende Tableau bestimmt.

$$T'_B = \begin{pmatrix} c_{N'}^T - c_{B'}^T A_{B'}^{-1} A_{N'} & -c_{B'}^T A_{B'}^{-1} b \\ A_{B'}^{-1} A_{N'} & A_{B'}^{-1} b \end{pmatrix}$$

$$A_{B'} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 3 \end{pmatrix} \quad A_{B'}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{3} \\ 0 & 0 & 0 & \frac{1}{3} \end{pmatrix} \quad A_N = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \quad A_{B'}^{-1} A_N = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & \frac{2}{3} & -\frac{1}{3} \\ 0 & \frac{1}{3} & \frac{1}{3} \end{pmatrix}$$

$$A_{B'}^{-1} b = \begin{pmatrix} 200 \\ 300 \\ 200 \\ 200 \end{pmatrix}$$

$$c^T = \begin{pmatrix} 1 & 6 & 0 & 0 & 0 & 0 & 13 \end{pmatrix} = \begin{pmatrix} c_{N'}^T & c_{B'}^T \end{pmatrix}$$

$$c_{N'}^T - c_{B'}^T A_{B'}^{-1} A_{N'} = \begin{pmatrix} 1 & \frac{5}{3} & -\frac{13}{3} \end{pmatrix}$$

$$-c_B^T A_B^{-1} b = -2600$$

Zweite Iteration

$E = \{j | c_x x_j > 0\} = \{1,2\}$ $\{x_1, x_2\}$

$L_1 = \{i | x_1^i > 0\} = \{1,3\}$ $\{s_1, s_3\}$

$L_2 = \{i | x_2^i > 0\} = \{2,3,4\}$ $\{s_2, s_3, x_3\}$

Heuristik: Ersetze einen Basisvektor durch den Nichtbasisvektor, der den größten Zugewinn für die Zielfunktion bringt.

x_1

$0 \leq s_1 = 200 - x_1$

$0 \leq s_3 = 200 - x_1$

$x_1 = 200 \Rightarrow z = 2800$

Hier wird die Zeile von s_1 *und* s_3 betrachtet und die Spalte von x_1. Der alte Wert ist 2600. Der Koeffizient von x_1 in der Zielfunktion ist 1 und der Zugewinn durch x_1 ist 200.

x_2

$0 \leq s_2 = 300 - x_2$

$0 \leq s_2 = 200 - \frac{2}{3}x_2$

$0 \leq x_2 = 200 - \frac{1}{3}x_2$

$x_2 = min(300, 600) = 300 \Rightarrow z = 4400$

Hier wird die Zeile von s_2, s_3 *und* x_3 betrachtet und die Spalte von x_2. Der alte Wert ist 2600. Der Koeffizient von x_2 in der Zielfunktion ist 6 und der Zugewinn durch x_2 ist 1800. Nun wird s_2 durch x_2 ersetzt.

Update des Tableaus

Der neue Wert von x_2 wird nun berechnet.

$s_2 = 300 - x_2 \Leftrightarrow x_2 = 300 - s_2$. Dieser Wert wird nun eingesetzt.

$z = x_1 + \frac{5}{3} \cdot (300 - s_2) - \frac{13}{3}s_4 + 2600 = x_1 - \frac{5}{3}s_2 - \frac{13}{3}s_4 + 3100$

$s_3 = 200 - x_1 + \frac{2}{3} \cdot (300 - s_2) + \frac{s_4}{3} = -x_1 + \frac{2}{3}s_2 + \frac{s_4}{3}$

$x_3 = 200 - \frac{1}{3} \cdot (300 - s_2) - \frac{s_4}{3} = 100 + \frac{1}{3}s_2 - \frac{s_4}{3}$

Das neue Tableau sieht nun so aus:

	x_1	s_2	s_4	b
z	1	$-\frac{5}{3}$	$-\frac{13}{3}$	-3100
s_1	1	0	0	200
x_2	0	1	0	300
s_3	1	$-\frac{2}{3}$	$-\frac{1}{3}$	0
x_3	0	$-\frac{1}{3}$	$\frac{1}{3}$	100

Dritte Iteration

$E = \{j | c_x x_j > 0\} = \{1\}\ \{x_1\}$

$L_1 = \{i | x_1^i > 0\} = \{1, 3\}\ \{s_1, s_3\}$

Ersetze einen Basisvektor durch den Nichtbasisvektor, der den größten Zugewinn für die Zielfunktion bringt. Es müssen nur Terme aus z mit positivem Vorzeichen betrachtet werden, d.h. es bleibt nur noch x_1 übrig.

x_1

$0 \leq s_1 = 200 - x_1$

$0 \leq s_3 = 0 - x_1$

$x_1 = min(200, 0) \Rightarrow z = 3100$

Update des Tableaus

Nun wird s_3 durch x_1 ersetzt.

$s_3 = -x_1 + \frac{2}{3}s_2 + \frac{s_4}{3} \Leftrightarrow x_1 = -s_3 + \frac{2}{3}s_2 + \frac{s_4}{3}$. Dieser Wert wird nun eingesetzt.

$z = -s_3 + \frac{2}{3}s_2 + \frac{s_4}{3} - \frac{5}{3}s_2 - \frac{13}{3}s_4 + 3100 = -s_3 - s_2 - 4s_4 + 3100$

$s_1 = 200 - (-s_3 - \frac{2}{3}s_2 - \frac{s_4}{3}) = 200 + s_3 + \frac{2}{3}s_2 + \frac{s_4}{3}$

Das neue Tableau sieht nun so aus:

	s_3	s_2	s_4	b
z	-1	-1	-4	-3100
s_1	-1	$\frac{2}{3}$	$\frac{1}{3}$	200
x_2	0	1	0	300
x_1	1	$-\frac{2}{3}$	$\frac{1}{3}$	0
x_3	0	$-\frac{1}{3}$	$\frac{1}{3}$	100

Die Zielfunktion kann nun nicht weiter verbessert werden. Unser x ist nun (0,300,100) und unser z ist 3100.

97.6 Analyse

Das Simplex-Verfahren löst ein lineares Programm in endlich vielen Schritten oder stellt seine Unlösbarkeit oder Unbeschränktheit fest. Im Worstcare hat es eine exponentielle Laufzeit, unabhängig von den gewählten Pivotregeln. In der Praxis ist es sehr effizient.

98 Autoren

Edits	User
1	Bocardodarapti[1]
22	Dirk Hünniger[2]
155	Dirk Hünniger (hsrw)[3]
288	Ekreckel[4]
11	Lkastler[5]
1	Matthias.Thimm[6]
905	Mhombach[7]

1 https://de.wikiversity.org/wiki/Benutzer:Bocardodarapti
2 https://de.wikiversity.org/wiki/Benutzer:Dirk_H%25C3%25BCnniger
3 https://de.wikiversity.org/wiki/Benutzer:Dirk_H%25C3%25BCnniger_(hsrw)
4 https://de.wikiversity.org/w/index.php%3ftitle=Benutzer:Ekreckel&action=edit&redlink=1
5 https://de.wikiversity.org/wiki/Benutzer:Lkastler
6 https://de.wikiversity.org/w/index.php%3ftitle=Benutzer:Matthias.Thimm&action=edit&redlink=1
7 https://de.wikiversity.org/w/index.php%3ftitle=Benutzer:Mhombach&action=edit&redlink=1

Abbildungsverzeichnis

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- LGPL: GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. http://artlibre.org/licence/lal/de
- CFR: Copyright free use.

- EPL: Eclipse Public License. `http://www.eclipse.org/org/documents/epl-v10.php`

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[8]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

8 Kapitel 99 auf Seite 361

1	Mhombach[9], Mhombach[10]	CC-BY-SA-3.0
2	Mhombach[11], Mhombach[12]	CC-BY-SA-3.0
3	Mhombach[13], Mhombach[14]	CC-BY-SA-3.0
4	Mhombach[15], Mhombach[16]	CC-BY-SA-3.0
5	Mhombach[17], Mhombach[18]	CC-BY-SA-3.0
6	Mhombach[19], Mhombach[20]	CC-BY-SA-3.0
7	Mhombach[21], Mhombach[22]	CC-BY-SA-3.0
8	Mhombach[23], Mhombach[24]	CC-BY-SA-3.0
9	Mhombach[25], Mhombach[26]	CC-BY-SA-3.0
10	Mhombach[27], Mhombach[28]	CC-BY-SA-3.0
11	Mhombach[29], Mhombach[30]	CC-BY-SA-3.0
12	Mhombach[31], Mhombach[32]	CC-BY-SA-3.0
13	Mhombach[33], Mhombach[34]	CC-BY-SA-3.0
14	Mhombach[35], Mhombach[36]	CC-BY-SA-3.0
15	Mhombach[37], Mhombach[38]	CC-BY-SA-3.0
16	Mhombach[39], Mhombach[40]	CC-BY-SA-3.0
17	Mhombach[41], Mhombach[42]	CC-BY-SA-3.0

9 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
10 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
11 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
12 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
13 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
14 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
15 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
16 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
17 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
18 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
19 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
20 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
21 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
22 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
23 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
24 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
25 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
26 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
27 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
28 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
29 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
30 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
31 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
32 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
33 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
34 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
35 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
36 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
37 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
38 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
39 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
40 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
41 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
42 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1

18	Mhombach[43], Mhombach[44]	CC-BY-SA-3.0
19	Mhombach[45], Mhombach[46]	CC-BY-SA-3.0
20	Mhombach[47], Mhombach[48]	CC-BY-SA-3.0
21	Mhombach[49], Mhombach[50]	CC-BY-SA-3.0
22	Mhombach[51], Mhombach[52]	CC-BY-SA-3.0
23	Mhombach[53], Mhombach[54]	
24	Mhombach[55], Mhombach[56]	
25	Mhombach[57], Mhombach[58]	
26	Mhombach[59], Mhombach[60]	CC-BY-SA-3.0
27	Mhombach[61], Mhombach[62]	CC-BY-SA-3.0
28	Mhombach[63], Mhombach[64]	CC-BY-SA-3.0
29	Mhombach[65], Mhombach[66]	CC-BY-SA-3.0
30	Mhombach[67], Mhombach[68]	CC-BY-SA-3.0
31	Mhombach[69], Mhombach[70]	CC-BY-SA-3.0
32	Mhombach[71], Mhombach[72]	CC-BY-SA-3.0
33	Mhombach[73], Mhombach[74]	CC-BY-SA-3.0
34	Mhombach[75], Mhombach[76]	CC-BY-SA-3.0

43 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
44 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
45 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
46 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
47 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
48 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
49 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
50 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
51 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
52 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
53 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
54 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
55 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
56 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
57 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
58 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
59 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
60 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
61 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
62 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
63 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
64 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
65 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
66 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
67 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
68 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
69 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
70 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
71 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
72 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
73 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
74 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
75 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
76 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1

35	Ekreckel[77], Ekreckel[78]	
36	Ekreckel[79], Ekreckel[80]	
37	Ekreckel[81], Ekreckel[82]	
38	Ekreckel[83], Ekreckel[84]	
39	Mhombach[85], Mhombach[86]	CC-BY-SA-3.0
40	Mhombach[87], Mhombach[88]	CC-BY-SA-3.0
41	Mhombach[89], Mhombach[90]	CC-BY-SA-3.0
42	Mhombach[91], Mhombach[92]	CC-BY-SA-3.0
43	Mhombach[93], Mhombach[94]	CC-BY-SA-3.0
44	Mhombach[95], Mhombach[96]	CC-BY-SA-3.0
45	Mhombach[97], Mhombach[98]	CC-BY-SA-3.0
46	Mhombach[99], Mhombach[100]	CC-BY-SA-3.0
47	Mhombach[101], Mhombach[102]	CC-BY-SA-3.0
48	Mhombach[103], Mhombach[104]	CC-BY-SA-3.0
49	Mhombach[105], Mhombach[106]	CC-BY-SA-3.0
50	Mhombach[107], Mhombach[108]	CC-BY-SA-3.0
51	Mhombach[109], Mhombach[110]	CC-BY-SA-3.0

77 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
78 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
79 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
80 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
81 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
82 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
83 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
84 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
85 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
86 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
87 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
88 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
89 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
90 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
91 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
92 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
93 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
94 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
95 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
96 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
97 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
98 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
99 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
100 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
101 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
102 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
103 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
104 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
105 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
106 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
107 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
108 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
109 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
110 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1

52	Mhombach[111], Mhombach[112]	CC-BY-SA-3.0
53	Mhombach[113], Mhombach[114]	CC-BY-SA-3.0
54	Mhombach[115], Mhombach[116]	CC-BY-SA-3.0
55	Mhombach[117], Mhombach[118]	CC-BY-SA-3.0
56	Mhombach[119], Mhombach[120]	CC-BY-SA-3.0
57	Mhombach[121], Mhombach[122]	CC-BY-SA-3.0
58	Mhombach[123], Mhombach[124]	CC-BY-SA-3.0
59	Mhombach[125], Mhombach[126]	CC-BY-SA-3.0
60	Mhombach[127], Mhombach[128]	CC-BY-SA-3.0
61	Mhombach[129], Mhombach[130]	CC-BY-SA-3.0
62	Mhombach[131], Mhombach[132]	CC-BY-SA-3.0
63	Mhombach[133], Mhombach[134]	CC-BY-SA-3.0
64	Mhombach[135], Mhombach[136]	CC-BY-SA-3.0
65	Mhombach[137], Mhombach[138]	CC-BY-SA-3.0
66	Mhombach[139], Mhombach[140]	CC-BY-SA-3.0
67	Mhombach[141], Mhombach[142]	CC-BY-SA-3.0
68	Mhombach[143], Mhombach[144]	CC-BY-SA-3.0

111 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
112 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
113 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
114 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
115 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
116 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
117 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
118 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
119 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
120 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
121 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
122 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
123 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
124 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
125 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
126 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
127 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
128 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
129 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
130 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
131 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
132 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
133 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
134 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
135 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
136 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
137 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
138 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
139 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
140 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
141 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
142 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
143 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
144 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1

69	Mhombach[145], Mhombach[146]	CC-BY-SA-3.0
70	Mhombach[147], Mhombach[148]	CC-BY-SA-3.0
71	Mhombach[149], Mhombach[150]	CC-BY-SA-3.0
72	Mhombach[151], Mhombach[152]	CC-BY-SA-3.0
73	Mhombach[153], Mhombach[154]	CC-BY-SA-3.0
74	Ekreckel[155], Ekreckel[156]	
75	Mhombach[157], Mhombach[158]	CC-BY-SA-3.0
76	Mhombach[159], Mhombach[160]	CC-BY-SA-3.0
77	Mhombach[161], Mhombach[162]	CC-BY-SA-3.0
78	Mhombach[163], Mhombach[164]	CC-BY-SA-3.0
79	Mhombach[165], Mhombach[166]	CC-BY-SA-3.0
80	Mhombach[167], Mhombach[168]	CC-BY-SA-3.0
81	Mhombach[169], Mhombach[170]	CC-BY-SA-3.0
82	Mhombach[171], Mhombach[172]	CC-BY-SA-3.0
83	Mhombach[173], Mhombach[174]	CC-BY-SA-3.0
84	Mhombach[175], Mhombach[176]	CC-BY-SA-3.0
85	Mhombach[177], Mhombach[178]	CC-BY-SA-3.0

145 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
146 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
147 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
148 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
149 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
150 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
151 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
152 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
153 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
154 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
155 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
156 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
157 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
158 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
159 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
160 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
161 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
162 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
163 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
164 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
165 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
166 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
167 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
168 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
169 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
170 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
171 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
172 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
173 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
174 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
175 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
176 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
177 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
178 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1

86	Mhombach[179], Mhombach[180]	CC-BY-SA-3.0
87	Mhombach[181], Mhombach[182]	CC-BY-SA-3.0
88	Mhombach[183], Mhombach[184]	CC-BY-SA-3.0
89	Ekreckel[185], Ekreckel[186]	
90	Ekreckel[187], Ekreckel[188]	
91	Ekreckel[189], Ekreckel[190]	
92	Mhombach[191], Mhombach[192]	CC-BY-SA-3.0
93	Mhombach[193], Mhombach[194]	CC-BY-SA-3.0
94	Mhombach[195], Mhombach[196]	CC-BY-SA-3.0
95	Mhombach[197], Mhombach[198]	CC-BY-SA-3.0
96	Mhombach[199], Mhombach[200]	CC-BY-SA-3.0
97	Mhombach[201], Mhombach[202]	CC-BY-SA-3.0
98	Mhombach[203], Mhombach[204]	CC-BY-SA-3.0
99	Ekreckel[205], Ekreckel[206]	
100	Ekreckel[207], Ekreckel[208]	
101	Ekreckel[209], Ekreckel[210]	
102	Ekreckel[211], Ekreckel[212]	

179 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
180 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
181 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
182 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
183 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
184 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
185 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
186 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
187 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
188 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
189 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
190 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
191 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
192 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
193 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
194 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
195 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
196 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
197 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
198 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
199 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
200 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
201 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
202 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
203 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
204 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
205 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
206 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
207 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
208 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
209 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
210 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
211 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
212 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1

103	Ekreckel[213], Ekreckel[214]	
104	Ekreckel[215], Ekreckel[216]	
105	Ekreckel[217], Ekreckel[218]	
106	Ekreckel[219], Ekreckel[220]	
107	Ekreckel[221], Ekreckel[222]	
108	Ekreckel[223], Ekreckel[224]	
109	Ekreckel[225], Ekreckel[226]	
110	Ekreckel[227], Ekreckel[228]	
111	Mhombach[229], Mhombach[230]	CC-BY-SA-3.0
112	Mhombach[231], Mhombach[232]	CC-BY-SA-3.0
113	Mhombach[233], Mhombach[234]	CC-BY-SA-3.0
114	Mhombach[235], Mhombach[236]	CC-BY-SA-3.0
115	Mhombach[237], Mhombach[238]	
116	Mhombach[239], Mhombach[240]	
117	Mhombach[241], Mhombach[242]	
118	Mhombach[243], Mhombach[244]	
119	Mhombach[245], Mhombach[246]	CC-BY-SA-3.0

213 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
214 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
215 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
216 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
217 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
218 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
219 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
220 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
221 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
222 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
223 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
224 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
225 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
226 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
227 http://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
228 https://commons.wikimedia.org/w/index.php?title=User:Ekreckel&action=edit&redlink=1
229 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
230 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
231 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
232 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
233 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
234 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
235 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
236 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
237 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
238 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
239 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
240 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
241 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
242 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
243 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
244 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
245 http://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1
246 https://commons.wikimedia.org/w/index.php?title=User:Mhombach&action=edit&redlink=1

99 Licenses

99.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from

conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

99.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free"in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Versionöf the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A SSecondary Sectionïs a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The Ïnvariant Sectionsäre certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Textsäre certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent"copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparentïs called Öpaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page"means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page"means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher"means any person or entity that distributes copies of the Document to the public.

A section Ëntitled XYZ"means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as Äcknowledgements", "Dedications", Ëndorsements", or "History".) To "Preserve the Titleöf such a section when you modify the Document means that it remains a section Ëntitled XYZäccording to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "Historyïn the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History§ection. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled Äcknowledgementsör "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled Ëndorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled Ëndorsementsör to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled Ëndorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "Historyïn the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled Äcknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled Ëndorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an ägg regateïf the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled Äcknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ör any later versionäpplies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site"(or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration"(or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA"means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

Ïncorporate"means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is ëligible for relicensingïf it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts."line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

99.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user’s computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy’s public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.